C++入门经典

(第3版)

(美) Ivor Horton 著

李予敏 译

清华大学出版社

北 京

内 容 简 介

C++在几乎所有的计算环境中都非常普及，而且可以用于几乎所有的应用程序。C++从C中继承了过程化编程的高效性，并集成了面向对象编程的功能。C++在其标准库中提供了大量的功能。有许多商业C++库支持数量众多的操作系统环境和专业应用程序。但因为它的内容太多了，所以掌握C++并不十分容易。本书详述了C++语言的各个方面，包括数据类型、程序控制、函数、指针、调试、类、重载、继承、多态性、模板、异常和输入输出等内容。每一章都以前述内容为基础，每个关键点都用具体的示例进行详细的讲解。

本书基本不需要读者具备任何C++知识，书中包含了理解C++的所有必要知识，读者可以从头开始编写自己的C++程序。本书也适合于具备另一种语言编程经验但希望全面掌握C++语言的读者。

EISBN：1-59059-227-1

Ivor Horton's Beginning ANSI C++: The Complete Language, Third Edition

Ivor Horton

Original English language edition published by Apress L. P., 2560 Ninth Street, Suite 219, Berkeley, CA 94710 USA.

北京市版权局著作权合同登记号　图字：01-2005-4579

图书在版编目(CIP)数据

C++入门经典(第3版)/(美)霍顿(Horton, I.)著；李予敏　译. —北京：清华大学出版社，2006.1

书名原文：Ivor Horton's Beginning ANSI C++: The Complete Language, Third Edition

ISBN 7-302-12062-5

Ⅰ. C…　Ⅱ. ①霍…②李…　Ⅲ. C语言—程序设计　Ⅳ. TP312

中国版本图书馆CIP数据核字(2005)第127516号

出 版 者：清华大学出版社　　**地　　址**：北京清华大学学研大厦

http://www.tup.com.cn　　**邮　　编**：100084

社 总 机：010-62770175　　**客户服务**：010-62776969

组稿编辑：曹　康

文稿编辑：徐燕华

封面设计：康　博

版式设计：康　博

印 刷 者：北京国马印刷厂

装 订 者：三河市新茂装订有限公司

发 行 者：新华书店总店北京发行所

开　　本：185×260　**印张**：50　**字数**：1280千字

版　　次：2006年1月第1版　2006年1月第1次印刷

书　　号：ISBN 7-302-12062-5/TP·7808

印　　数：1～4000

定　　价：98.00元

作者简介

Ivor Horton是世界著名的计算机图书作家，主要从事与编程相关的顾问及撰写工作，曾帮助无数程序员步入编程的殿堂。他曾在IBM工作多年，能使用多种语言进行编程(在多种机器上使用汇编语言和高级语言)，设计和实现了实时闭环工业控制系统。Horton拥有丰富的教学经验(教学内容包括C、C++、Fortran、PL/1、APL等)，同时还是机械、加工和电子CAD系统、机械CAM系统和DNC/CNC系统方面的专家。Ivor Horton还著有*Beginning Visual C++ 6*、*Beginning C Programming*和*Beginning Java 2*等多部入门级好书。

译者简介

李予敏，男，计算机科学及应用专业博士，某研究院高级研究员，拥有丰富的C、C++编程经验，在核心期刊、国际国内会议上发表多篇文章，拥有著作3本、译著2本。

译 者 序

C++自诞生以来，已成为使用最广泛的一种编程语言。C++从 C 中继承了过程化编程的高效性，集成了面向对象编程技术。C++还在其标准库中提供了大量的功能。它有着极大的灵活性、强大的功能和非常高的效率，常常用于专业应用程序的开发，由于其内容较多，掌握起来也不易。

本书是《C++入门经典》的第 3 版，采用的编写方法与前两版相同，第 3 版在第 1、2 两版基础上又做了修订和更新，包括示例，对于自学的学生来说也是理想的选择。

本书详细介绍了 C++语言的各个方面，包括数据类型、程序控制、函数、指针、调试、类、重载、继承、多态性、异常和输入输出等内容。本书还深入讨论了类模板，包括标准模板库(STL，Standard Template Library)。每章都以前述内容为基础，每个关键点都通过具体的示例进行讲解。每章的最后都提供了练习题。

本书主要介绍标准的 C++编程语言，涉及 C++的语法、面向对象的功能和标准库等所有基本内容。通过本书可以获得编写 C++应用程序的所有必要知识。

对于初学者来说，C++语言似乎比其他语言更难，但其功能和适用范围要远远超过其他编程语言。读者只要抱以正确的态度、具备编程的基本知识以及掌握 C++的热情，在 C++的学习和应用上就不会有太大的问题。学习本书，读者基本不需要具备任何编程语言的知识，只需了解基本的编程概念，就可以读懂本书，编写自己的 C++程序，这也是学好 C++的惟一方式。如果您了解像分支和循环这样的概念，那本书就非常适合您。

本书也适合于已有其他语言编程经验但希望全面掌握 C++语言的读者。

作者 Ivor Horton 以善于教学而著称，他的 C、C++等书都拥有大量忠实读者。

由于受时间和译者自身水平的限制，翻译过程中难免出现错误和疏漏，敬请读者多多批评指正，反馈信息请发至 fwkbook@tup.tsinghua.edu.cn 信箱。

译者

2005.7

前　言

本书主要介绍标准的 C++编程语言，涉及 C++的语法、面向对象的功能和标准库等所有基本内容。阅读本书将获得编写 C++应用程序的所有必要知识。

为什么要学习 C++

C++自问世以来，已成为应用最广泛的一种编程语言。C++由于其极高的灵活性、强大的功能和非常高的效率，常常用于专业应用程序的开发，C++非常适合于编写各种编程环境下的高性能代码。

它要比许多人想像的更容易理解。只要有正确的引导，掌握 C++编程语言是比较容易的。开发 C++技巧，学习许多人已在使用的语言，在自己的编程工具箱中就会多一种功能强大的新工具。

C++的标准

1998 年，C++的国际化标准 ISO/IEC 14882 最终定稿，并被美国国家标准协会 ANSI 和信息技术标准国际协会 INCITS 采纳。这是 ANSI/ISO 小组 9 年工作的成果，其目的是为 C++编程语言开发一种世界标准。尽管编写本书时 1998 年的标准仍在使用，但改进该语言的工作一直在进行，因此将来 C++一定会添加新特性。

C++的 1998 标准为编译器的编写人员提供了一幅蓝图，所以，目前许多(但不是全部)编译器都遵循该标准。如果使用遵循该标准的编译器，代码的可移植性将非常高，将来，还可以避免非标准语言元素带来的麻烦。

当然，C++的标准定义为开发在任何硬件或操作系统环境下运行的编译器的参考框架。另外，它还将试图在任何开发环境下尽可能地提高性能。也就是说，编译器编写人员在许多领域都有非常大的灵活性，以包容机器体系之间的差异。例如，该标准定义了数字数据和算术操作，这样编译器编写人员就可以充分利用各种机器的不同特性，优化执行性能。编译器编写人员还可以选择用于定义 C++程序的字符编码。这样，就可以包容默认字符编码在不同操作系统上的变化。没有这种灵活性，在某些机器上该标准就会有一定的局限性，导致性能较差，这非常不利于一般目的的编程语言。

本书将指出机器之间重要的、潜在的不同。但是，这需要一个实际有效的环境来显示本书中各个例子的输出。因此，所有的例子都在一台安装了 Intel 处理器体系结构的 PC 上运行。

错误和更正

作者和 Apress 的编辑们已经尽最大努力确保本书中的文本和代码没有错误，但是错误仍

然在所难免。如果您发现本书存在错误，请进入 Apress 网站的下述 Web 页面：

http://www.apress.com/book/download.html

如果在这个页面的列表中选择本书的书名，就可以下载勘误表和本书所有例子的代码，还可以记录下您找到的其他错误。下载的代码也包含所有练习的答案，但读者最好在完成了练习后再看答案。

使用本书

要通过本书学习 C++，需要一个与 ANSI/ISO 兼容的编译器和一个适合于编写程序代码的文本编辑器。目前，大多数专业 C++开发环境所附带的编译器都遵循这个标准，但在购买之前最好检查一下。另外，Internet 上的一些免费软件和开放源代码的 C++编译器也遵循 C++标准。可以使用其中一个编译器和免费的程序文本编辑器，建立起一个经济、可行的学习环境。

本书的内容循序渐进，所以读者应从头开始一直阅读到最后。但是，没有人能仅从一本书中获得所有的编程技巧。本书仅介绍了如何使用 C++编程，读者应自己输入所有的例子，而不是从下载文件中复制它们。再编译和执行输入的代码，这似乎很麻烦，但输入 C++语句可以帮助理解 C++，特别是觉得某些地方很难掌握时，自己输入代码就显得非常有帮助。如果例子不工作，不要直接从书中查找原因，而应在自己输入的例子代码中找原因，这是编写 C++代码时必须做的一个工作。

犯错误也是学习过程中不可避免的，练习应提供大量犯错误的机会，犯的错误越多，对 C++的功能和错误的原因认识得就越深刻。读者应完成所有的练习，记住不要看答案，直到肯定不能自己解决问题为止。许多练习都涉及某章内容的一个直接应用，换言之，它们仅是一种实践，但也有一些练习需要多动脑子，甚至需要一点灵感。

希望每个人都能成功驾驭 C++。

目　录

第 1 章　基本概念 ····· 1
1.1　编程语言 ····· 1
1.1.1　编程语言简史 ····· 1
1.1.2　解释性程序和编译性程序的执行过程 ····· 2
1.1.3　库 ····· 3
1.2　C++是一种强大的语言 ····· 3
1.3　一个简单的 C++程序 ····· 4
1.3.1　名称 ····· 6
1.3.2　命名空间 ····· 7
1.4　关键字 ····· 9
1.5　C++语句和语句块 ····· 9
1.6　程序结构 ····· 10
1.7　从源文件中创建可执行文件 ····· 12
1.7.1　编译 ····· 12
1.7.2　链接 ····· 13
1.8　C++源字符 ····· 14
1.8.1　通用字符集 ····· 15
1.8.2　三字符序列 ····· 15
1.8.3　转义序列 ····· 16
1.8.4　语句中的空白 ····· 18
1.9　程序的注释 ····· 19
1.10　标准库 ····· 20
1.11　用 C++编程 ····· 21
1.12　本章小结 ····· 22
1.13　练习 ····· 23

第 2 章　基本数据类型和计算 ····· 24
2.1　数据和数据类型 ····· 24
2.2　进行简单的计算 ····· 24
2.2.1　字面量 ····· 25
2.2.2　整型字面量 ····· 25
2.2.3　整数的算术运算 ····· 27
2.2.4　运算符的优先级和相关性 ····· 30
2.3　使用变量 ····· 32
2.4　整型变量 ····· 33
2.4.1　整型变量类型 ····· 35
2.4.2　整数的取值范围 ····· 37
2.4.3　整型字面量的类型 ····· 38
2.5　赋值运算符 ····· 39
2.5.1　多次赋值 ····· 40
2.5.2　修改变量的值 ····· 40
2.6　整数的递增和递减 ····· 42
2.7　const 关键字 ····· 44
2.8　整数的数字函数 ····· 45
2.9　浮点数 ····· 49
2.9.1　浮点数的数据类型 ····· 49
2.9.2　浮点数的操作 ····· 51
2.9.3　使用浮点数值 ····· 53
2.9.4　数值函数 ····· 55
2.10　使用字符 ····· 57
2.10.1　字符字面量 ····· 57
2.10.2　初始化 char 变量 ····· 58
2.10.3　使用扩展字符集 ····· 60
2.11　初始值的函数表示法 ····· 62
2.12　本章小结 ····· 62
2.13　练习 ····· 63

第 3 章　处理基本数据类型 ····· 64
3.1　混合的表达式 ····· 64
3.1.1　赋值和不同的类型 ····· 65
3.1.2　显式强制转换 ····· 66
3.1.3　老式的强制转换 ····· 68
3.2　确定类型 ····· 70
3.3　按位运算符 ····· 73
3.3.1　移位运算符 ····· 74
3.3.2　位模式下的逻辑运算 ····· 76
3.4　枚举数据类型 ····· 85
3.4.1　匿名枚举 ····· 86

3.4.2 在整型和枚举类型之间强制转换 …… 87
3.5 数据类型的同义词 …… 89
3.6 变量的生存期 …… 90
3.6.1 自动变量 …… 90
3.6.2 定位变量的声明 …… 92
3.6.3 全局变量 …… 92
3.6.4 静态变量 …… 95
3.7 特殊的类型修饰符 …… 96
3.8 声明外部变量 …… 96
3.9 优先级和相关性 …… 96
3.10 本章小结 …… 97
3.11 练习 …… 98

第 4 章 选择和决策 …… 99
4.1 比较数据值 …… 99
4.1.1 应用比较运算符 …… 100
4.1.2 比较浮点数值 …… 102
4.2 if 语句 …… 102
4.3 if-else 语句 …… 110
4.4 逻辑运算符 …… 114
4.4.1 逻辑与运算符 …… 115
4.4.2 逻辑或运算符 …… 115
4.4.3 逻辑非运算符 …… 115
4.5 条件运算符 …… 118
4.6 switch 语句 …… 120
4.7 无条件分支 …… 124
4.8 决策语句块和变量作用域 …… 125
4.9 本章小结 …… 126
4.10 练习 …… 126

第 5 章 循环 …… 127
5.1 理解循环 …… 127
5.2 while 循环 …… 128
5.3 do-while 循环 …… 130
5.4 for 循环 …… 133
5.4.1 循环和变量作用域 …… 135
5.4.2 用浮点数值控制 for 循环 …… 137
5.4.3 使用更复杂的循环控制表达式 …… 140
5.5 嵌套的循环 …… 143
5.6 跳过循环迭代 …… 147
5.7 循环的中断 …… 150
5.8 本章小结 …… 155
5.9 练习 …… 155

第 6 章 数组和字符串 …… 156
6.1 数据数组 …… 156
6.1.1 使用数组 …… 156
6.1.2 初始化数组 …… 161
6.1.3 字符数组 …… 164
6.2 多维数组 …… 168
6.2.1 初始化多维数组 …… 170
6.2.2 多维字符数组 …… 172
6.3 string 类型 …… 174
6.3.1 声明 string 对象 …… 175
6.3.2 使用 string 对象 …… 177
6.3.3 访问字符串中的字符 …… 179
6.3.4 访问子字符串 …… 182
6.3.5 比较字符串 …… 182
6.3.6 搜索字符串 …… 188
6.3.7 修改字符串 …… 196
6.4 string 类型的数组 …… 201
6.5 宽字符的字符串 …… 202
6.6 本章小结 …… 202
6.7 练习 …… 203

第 7 章 指针 …… 204
7.1 什么是指针 …… 204
7.2 指针的声明 …… 205
7.3 指针的初始化 …… 210
7.4 常量指针和指向常量的指针 …… 220
7.5 指针和数组 …… 221
7.5.1 指针的算术运算 …… 221
7.5.2 使用数组名的指针表示法 …… 224
7.5.3 对多维数组使用指针 …… 227
7.5.4 C 样式字符串的操作 …… 229
7.6 动态内存分配 …… 231
7.6.1 自由存储区 …… 232
7.6.2 运算符 new 和 delete …… 232
7.6.3 数组的动态内存分配 …… 233
7.6.4 动态内存分配的危险 …… 235
7.6.5 转换指针 …… 241

7.7 本章小结 ……241
7.8 练习 ……242

第 8 章 使用函数编程 ……243
8.1 程序的分解 ……243
8.2 理解函数 ……245
8.2.1 定义函数 ……245
8.2.2 函数的声明 ……249
8.3 给函数传送参数 ……251
8.3.1 按值传送机制 ……251
8.3.2 按引用传送机制 ……260
8.3.3 main()的参数 ……264
8.4 默认的参数值 ……265
8.5 从函数中返回值 ……268
8.5.1 返回一个指针 ……268
8.5.2 返回一个引用 ……272
8.5.3 从函数中返回新变量 ……273
8.6 内联函数 ……273
8.7 静态变量 ……273
8.8 本章小结 ……276
8.9 练习 ……276

第 9 章 函数 ……278
9.1 函数的重载 ……278
9.1.1 函数的签名 ……278
9.1.2 重载和指针参数 ……281
9.1.3 重载和引用参数 ……281
9.1.4 重载和 const 参数 ……283
9.1.5 重载和默认参数值 ……284
9.2 函数模板 ……285
9.2.1 创建函数模板的实例 ……286
9.2.2 显式指定模板参数 ……288
9.2.3 模板的说明 ……289
9.2.4 函数模板和重载 ……291
9.2.5 带有多个参数的模板 ……292
9.3 函数指针 ……293
9.3.1 声明函数指针 ……294
9.3.2 把函数作为参数传送 ……297
9.3.3 函数指针的数组 ……299
9.4 递归 ……299
9.5 本章小结 ……307
9.6 练习 ……307

第 10 章 程序文件和预处理器指令 ……309
10.1 使用程序文件 ……309
10.1.1 名称的作用域 ……310
10.1.2 “一个定义”规则 ……312
10.1.3 程序文件和链接 ……313
10.1.4 外部名称 ……314
10.2 命名空间 ……318
10.2.1 全局命名空间 ……319
10.2.2 定义命名空间 ……319
10.2.3 使用 using 声明 ……322
10.2.4 函数和命名空间 ……322
10.2.5 函数模板和命名空间 ……326
10.2.6 扩展命名空间 ……327
10.2.7 未指定名称的命名空间 ……330
10.2.8 命名空间的别名 ……331
10.2.9 嵌套的命名空间 ……331
10.3 预处理器 ……332
10.3.1 在程序中包含头文件 ……333
10.3.2 程序中的置换 ……334
10.3.3 宏置换 ……336
10.3.4 放在多行代码中的预处理器指令 ……338
10.3.5 把字符串作为宏参数 ……339
10.3.6 在宏表达式中连接参数 ……340
10.4 逻辑预处理器指令 ……340
10.4.1 逻辑#if 指令 ……341
10.4.2 测试特定值的指令 ……343
10.4.3 多个代码选择块 ……343
10.4.4 标准的预处理器宏 ……344
10.4.5 #error 和#pragma 指令 ……345
10.5 调试方法 ……346
10.5.1 集成调试器 ……346
10.5.2 调试中的预处理器指令 ……347
10.5.3 使用 assert 宏 ……353
10.6 本章小结 ……354
10.7 练习 ……355

第 11 章 创建自己的数据类型 ……356
11.1 对象的概念 ……356

11.2 C++中的结构……357
11.2.1 理解结构……357
11.2.2 定义结构类型……358
11.2.3 创建结构类型的对象……360
11.2.4 访问结构对象的成员……360
11.2.5 对结构使用指针……366
11.3 联合……370
11.3.1 声明联合……371
11.3.2 匿名联合……372
11.4 更复杂的结构……373
11.5 本章小结……379
11.6 练习……380

第 12 章 类……381
12.1 类和面向对象编程……381
12.1.1 封装……382
12.1.2 继承……383
12.1.3 多态性……384
12.1.4 术语……385
12.2 定义类……385
12.3 构造函数……388
12.3.1 把构造函数的定义放在类的外部……390
12.3.2 默认的构造函数……392
12.3.3 默认的初始化值……395
12.3.4 在构造函数中使用初始化列表……396
12.3.5 使用 explicit 关键字……397
12.4 类的私有成员……398
12.4.1 访问私有类成员……402
12.4.2 默认的副本构造函数……404
12.5 友元……405
12.5.1 类的友元函数……405
12.5.2 友元类……408
12.6 this 指针……409
12.7 const 对象和 const 成员函数……413
12.7.1 类中的 mutable 数据成员……415
12.7.2 常量的强制转换……416
12.8 类的对象数组……416
12.9 类对象的大小……419
12.10 类的静态成员……421
12.10.1 类的静态数据成员……421
12.10.2 类的静态成员函数……426
12.11 本章小结……427
12.12 练习……428

第 13 章 类的操作……429
13.1 类对象的指针和引用……429
13.2 指针作为数据成员……430
13.2.1 定义 Package 类……431
13.2.2 定义 TruckLoad 类……434
13.2.3 实现 TruckLoad 类……435
13.3 控制对类的访问……443
13.4 副本构造函数的重要性……445
13.5 对象内部的动态内存分配……453
13.5.1 析构函数……453
13.5.2 定义析构函数……453
13.5.3 默认的析构函数……454
13.5.4 实现析构函数……456
13.6 类的引用……457
13.7 本章小结……459
13.8 练习……460

第 14 章 运算符重载……461
14.1 为自己的类实现运算符……461
14.1.1 运算符重载……461
14.1.2 可以重载的运算符……462
14.1.3 实现重载运算符……462
14.1.4 全局运算符函数……466
14.1.5 提供对运算符的全部支持…466
14.1.6 运算符函数术语……470
14.1.7 重载赋值运算符……470
14.1.8 重载算术运算符……477
14.1.9 重载下标运算符……482
14.1.10 重载类型转换……489
14.1.11 重载递增和递减运算符……490
14.1.12 智能指针……491
14.1.13 重载运算符 new 和 delete…497
14.2 本章小结……497
14.3 练习……498

第 15 章　继承 ……499
15.1　类和面向对象编程 ……499
15.2　类的继承 ……500
15.2.1　继承和聚合 ……501
15.2.2　从基类中派生新类 ……502
15.3　继承下的访问控制 ……505
15.4　把类的成员声明为 protected ……508
15.5　派生类成员的访问级别 ……510
15.5.1　在类层次结构中使用访问指定符 ……511
15.5.2　改变继承成员的访问指定符 ……512
15.6　派生类中的构造函数操作 ……514
15.7　继承中的析构函数 ……520
15.8　重复的成员名 ……522
15.9　多重继承 ……524
15.9.1　多个基类 ……524
15.9.2　继承成员的模糊性 ……526
15.9.3　重复的继承 ……531
15.9.4　虚基类 ……532
15.10　在相关的类类型之间转换 ……533
15.11　本章小结 ……534
15.12　练习 ……534

第 16 章　虚函数和多态性 ……536
16.1　理解多态性 ……536
16.1.1　使用基类指针 ……536
16.1.2　调用继承的函数 ……538
16.1.3　虚函数 ……542
16.1.4　虚函数中的默认参数值 ……549
16.1.5　通过引用来调用虚函数 ……553
16.1.6　调用虚函数的基类版本 ……554
16.1.7　在指针和类对象之间转换 ……555
16.1.8　动态强制转换 ……557
16.2　多态性的成本 ……559
16.3　纯虚函数 ……560
16.3.1　抽象类 ……560
16.3.2　间接的抽象基类 ……563
16.4　通过指针释放对象 ……566
16.5　在运行期间标识类型 ……569
16.6　类成员的指针 ……570
16.6.1　数据成员指针 ……570
16.6.2　成员函数指针 ……574
16.7　本章小结 ……578
16.8　练习 ……578

第 17 章　程序错误和异常处理 ……580
17.1　处理错误 ……580
17.2　理解异常 ……581
17.2.1　抛出异常 ……581
17.2.2　导致抛出异常的代码 ……586
17.2.3　嵌套的 try 块 ……588
17.3　用类对象作为异常 ……591
17.3.1　匹配 Catch 处理程序和异常 ……592
17.3.2　用基类处理程序捕获派生类异常 ……596
17.3.3　重新抛出异常 ……598
17.3.4　捕获所有的异常 ……601
17.4　抛出异常的函数 ……603
17.4.1　函数 try 块 ……603
17.4.2　在构造函数中抛出异常 ……605
17.4.3　异常和析构函数 ……606
17.5　标准库异常 ……606
17.5.1　标准库异常类 ……607
17.5.2　使用标准异常 ……608
17.6　本章小结 ……609
17.7　练习 ……610

第 18 章　类模板 ……611
18.1　理解类模板 ……611
18.2　定义类模板 ……612
18.2.1　模板参数 ……613
18.2.2　简单的类模板 ……613
18.2.3　创建类模板的实例 ……617
18.2.4　类模板的静态成员 ……625
18.2.5　非类型的类模板参数 ……625
18.2.6　非类型参数示例 ……626
18.2.7　默认的模板参数值 ……636
18.3　模板的显式实例化 ……636
18.4　类模板的友元 ……637
18.5　特殊情形 ……638

18.6 带有嵌套类的类模板……640
18.7 更高级的类模板……648
18.8 本章小结……649
18.9 练习……649

第 19 章 输入输出操作……651
19.1 C++中的输入输出……651
19.1.1 理解流……651
19.1.2 使用流的优点……652
19.2 流类……653
19.2.1 标准流……654
19.2.2 流的插入和提取操作……655
19.2.3 流操纵程序……657
19.3 文件流……659
19.3.1 写入文件……659
19.3.2 读取文件……662
19.3.3 设置文件打开模式……664
19.4 未格式化的流操作……672
19.4.1 未格式化的流输入函数……673
19.4.2 未格式化的流输出函数……674
19.5 流输入输出中的错误……675
19.6 使用二进制模式流操作……677
19.7 对流的读写操作……685
19.8 字符串流……692
19.9 对象和流……693
19.9.1 重载类对象的插入运算符…693
19.9.2 重载类对象的提取运算符…696
19.9.3 流中更复杂的对象……698
19.10 本章小结……710
19.11 练习……710

第 20 章 标准模板库……711
20.1 STL 架构简介……711
20.1.1 STL 组件……711
20.1.2 STL 头文件……716
20.2 使用 vector 容器……717
20.2.1 创建 vector 容器……717
20.2.2 访问 vector 容器中的元素…720
20.2.3 vector 容器的基本操作……722
20.2.4 使用 vector 容器进行数组操作……726
20.2.5 使用输入流迭代器……730
20.3 创建自己的迭代器……734
20.3.1 给算法传送迭代器……736
20.3.2 STL 迭代器类型的要求……738
20.3.3 STL 迭代器成员函数的要求……740
20.3.4 插入迭代器……744
20.4 list 容器……745
20.4.1 创建 list 容器……746
20.4.2 访问 list 容器中的元素……747
20.4.3 list 容器上的操作……747
20.5 关联 map 容器……753
20.6 性能和规范……761
20.7 本章小结……763
20.8 练习……763

附录 A ASCII 码……764
附录 B C++关键字……768
附录 C 标准库头文件……769
附录 D 运算符的优先级和相关性……774
附录 E 理解二进制和十六进制数……777
附录 F 项目示例……783

第1章 基本概念

初看起来，学习C++编程与家禽不可能有关系，但它们还是有关系的——这是一个鸡生蛋、蛋生鸡的问题。特别是在学习C++的初期，我们常常要利用还不能理解的例子来学习C++。本章将概述C++语言，介绍C++的各种特性，解决这个鸡生蛋、蛋生鸡的问题，再讨论几个要使用的概念，以后会详细论述它们。

这里介绍的所有概念都将在后面的章节详细讨论。大多数内容仅是在编写C++程序之前介绍一下基本概念。我们将列举一个简单的C++程序，并说明它的组成部分。本章还将介绍C++中编程的一些主要概念，以及如何根据所编写的源代码文件创建可执行的程序。

不必努力记住本章介绍的概念，而应把注意力集中在涉及到的理念上。本章提到的所有概念都将在后面的章节中详细论述。

本章主要内容

- C++的哪些特性使之这么普及
- C++程序的基本元素
- 如何注释程序的源代码
- 源代码如何变成可执行程序
- 面向对象的编程方式与过程编程方式的区别

1.1 编程语言

读者可能很熟悉编程和编程语言的基本概念，下面从普遍的意义上简要描述一下本书将用到的一些术语，并阐述C++与其他编程语言的关系。

目前有许多编程语言，每一种语言都有其优缺点，都有其吹捧者和批评者。除了C++之外，读者一定还听说过Java、BASIC(Beginner's All-purpose Symbolic Instruction Code的首字母缩写)、COBOL(Common Business-Oriented Language的首字母缩写)、FORTRAN(formula translator的前几个字母缩写)、PASCAL(以一位法国数学家Blaise Pascal命名)和C(只是因为它是B语言的后续语言)等编程语言。所有这些统称为高级语言，因为它们可以比较容易地表达出要计算机完成的工作，而且不针对某台计算机。高级语言中的每个源语句一般映射为几个内部机器指令，低级语言比较接近内部机器指令，通常称为汇编语言，一种汇编语言专门用于一种硬件设计，一般一个汇编指令映射为一个内部机器指令。

1.1.1 编程语言简史

FORTRAN是第一种开发出来的高级语言，第一个FORTRAN编译器是在上个世纪50年代后期开发出来的。FORTRAN已有40多年的历史了，目前仍广泛应用于科学和工程计算中，但C++和其他语言也逐渐进入这些领域。

COBOL语言专门用于商务数据处理应用程序，它的历史几乎与FORTRAN语言一样长。

目前几乎不用 COBOL 编写新代码，而是多年前编写的大量代码仍在使用，所以必须维护它们。C++也逐渐成为许多商务数据处理程序的可选语言。

BASIC 在上个世纪 70 年代诞生，那时已经有了个人计算机的概念。有趣的是，Microsoft 销售的第一个产品是一个 BASIC 解释程序。这种语言所固有的易用性使之很快普及，直到今天仍非常流行。

Java 是在上个世纪 90 年代开发的，它最初开发为 Oak 语言，用于给小型电子设备编程。1995 年，Oak 演变为 Java 语言，可以在 Web 页面中内嵌代码，从那时起直到现在，这已经成为 Java 的主要用途。Java 成功的主要原因是它的可移植性。Java 程序可以在任何支持它的硬件平台上运行，而且不需要任何修改。Java 语言的语法有许多特性，使它看起来很象 C++，但有很大的区别。Java 在可移植性方面比 C++好，但执行性能比不上 C++。

C 在上个世纪 70 年代被开发为一种高级语言，用于低级编程，例如实现操作系统。大多数 Unix 操作系统就是用 C 编写的。

C++是 Bjarne Stroustrup 在上个世纪 80 年代早期开发的，是一种基于 C 的面向对象语言。顾名思义，C++表示 C 的累加。由于 C++基于 C，所以这两种语言有许多共同的语法和功能，C 中所有低级编程的功能都在 C++中保留下来。但是，C++比其前身丰富得多，用途也广泛得多。C++对内存管理功能进行了非常大的改进，C++还具有面向对象的功能，所以 C 在功能上只是 C++的一个很小的子集。C++在适用范围、性能和功能上也是无可匹敌的。因此，目前大多数高性能的应用程序和系统仍使用 C++编写。

1.1.2 解释性程序和编译性程序的执行过程

无论使用哪种编程语言，编写出来的程序都是由各个指令或源语句构成的，它们描述了希望计算机执行的动作。这些指令或源语句统称为源代码，存储在磁盘的源文件中。任何规模的 C++程序都是由若干个源文件组成的。

编程语言的目的是，与计算机可以执行的程序相比，能够更简单地描述希望计算机执行的动作。计算机只能执行包含机器指令(也称为机器代码)的程序，不能直接执行我们编写的程序。用前面提到的语言编写的程序基本上有两种执行方式，在大多数情况下，一种语言会选择其中一种执行方式。例如，用 BASIC 语言编写的程序通常是解释性的，也就是说，另一个称为解释器的程序会检查 BASIC 源代码，确定该程序要做什么，再让计算机完成这些动作。如图 1-1 所示。

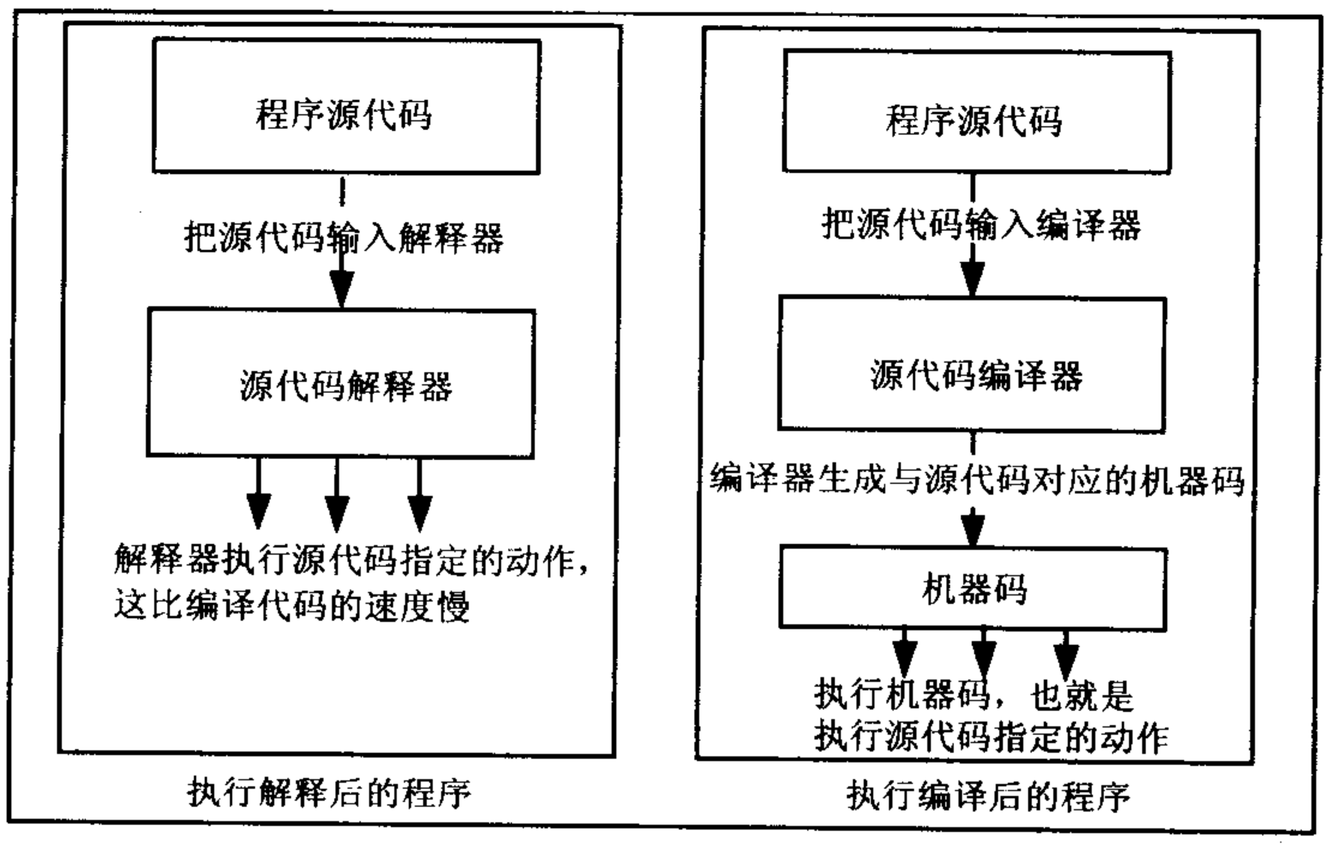


图 1-1 解释性程序和编译性程序的执行过程

而 C++是一种编译语言。在执行 C++程序之前，必须用另一个程序(即编译器)把它转换为机器语言。编译器会检查并分析 C++程序，并生成机器指令，以执行源代码指定的动作。当然，解释和编译都不像这里描述的那样简单，但其工作原理就是这样。

使用解释性语言，执行过程是间接的，也就是说，每次执行程序时，都需要确定源代码的意图。因此，这种语言比编译语言的对应程序的执行速度慢得多，有时要慢 100 倍。其优点是在运行之前，不必等待程序的编译。使用解释性语言，一旦输入代码，就可以立刻执行程序。任何一种语言要么是解释性的，要么是编译性的，这通常由该语言的设计和用途来决定。前面说过 BASIC 是一种解释性语言，但这不是绝对的，目前有许多 BASIC 语言的编译器。

于是，就有了“哪种语言比较好”这个问题。简单地说，没有所谓“最好”的语言，因为这取决于环境。例如，用 BASIC 编写程序通常比使用其他语言快得多，所以，如果开发速度比较重要，但执行性能不是很重要，BASIC 就是一种非常好的选择。另一方面，如果程序要求具有 C++提供的执行性能，或者应用程序需要 C++中的功能而不是 BAISC，显然就应使用 C++。如果应用程序必须在许多不同的计算机上执行，而且执行性能不是很重要，可能 Java 就是最佳选择。

当然，不同的语言学习起来，所需的时间和难度也不一致。根据学习一种语言所需的时间，C++可能是比较难的，但是不应放弃，这并不是说 C++非常难，而是说 C++比其他语言难一些，需要较长的学习时间。

学习哪种语言的最后一个要点是，任何专业程序员都需要掌握几种编程语言。如果读者是一位初学者，这可能会使人觉得沮丧，但一旦学习并掌握了一两种编程语言，再掌握其他语言就非常容易了。第一种编程语言总是最难的。

1.1.3 库

每次编写程序时，如果总是要从头开始编写，就相当繁琐。在许多程序中，常常需要某种相同的功能，例如从键盘上输入数据，或在屏幕上显示信息，或按照指定的顺序对数据记录排序。为了解决这个问题，编程语言通常提供了大量预先编写好的代码，以执行标准的操作，这样就不必重新编写这些代码了。

可用于任意程序的标准代码都保存在一个库中。编程语言附带的库跟语言本身一样重要，因为库的质量和使用范围对完成某一编程任务所需的时间有非常大的影响。

1.2 C++是一种强大的语言

C++在几乎所有的计算环境中都非常普及：个人电脑、Unix 工作站和大型计算机。如果考察一下新编程语言的发展史，就可以看出 C++的这种普及率是非常高的。用以前的语言编写的程序量非常大，这无疑会降低对新语言的接受程度。除此以外，大多数专业程序员总是愿意使用他们已熟知的、使用起来得心应手的语言，而不是转而使用新的、不熟悉的语言，花大量的时间来研究其特性。当然，C++是建立在 C 的基础之上(在 C++出现之前，许多环境都使用 C 语言)，这对于 C++的普及有很大的帮助，但是 C++的流行远不只这一个原因。C++有许多优点：

- C++适用的应用程序范围极广。C++可以用于几乎所有的应用程序，从字处理应用程序到科学应用程序，从操作系统组件到计算机游戏等。
- C++可以用于硬件级别的编程，例如实现设备驱动程序。
- C++从 C 中继承了过程化编程的高效性，并集成了面向对象编程方式的功能。
- C++在其标准库中提供了大量的功能。
- 有许多商业 C++库支持数量众多的操作系统环境和专门的应用程序。

因为几乎所有的计算机都可以使用 C++编程，所以 C++语言普及到几乎所有的计算机平台上。也就是说，把用 C++编写的程序从一台机器迁移到另一台机器上不需要费什么力气。当然，如果这个过程真的非常简单，那么编写在另一台机器上运行的程序时就需要考虑使用 C++语言了。

C++的 ANSI/ISO 标准

C++的国际标准由 ISO/IEC 14882 文档定义，该文档由美国国家标准协会 ANSI 发表。读者可以获得该标准的副本，但要记住，该标准主要由编译器编写人员使用，而不是学习该语言的人使用。如果读者不在意这一点，就可以从 http://webstore.ansi.org/ansidocstore/default.asp 上用合理的费用下载该标准的副本。

标准化是把所编写的程序从一种类型的计算机迁移到另一种类型的计算机上的基础。标准的建立使语言在各种机器上的实现保持一致。在所有相容编程系统上都可用的一组标准功能意味着，用户总是能确定下一步会获得什么结果。C++的 ANSI 标准不仅定义了语言，还定义了标准库。使用 ANSI 标准后，C++使应用程序可以轻松地在不同的机器之间迁移，缓解了在多个环境上运行的应用程序的维护问题。

C++的 ANSI 标准还有另一个优点：它对用 C++编程所需要学习的部分进行了标准化。这个标准将使后续的程序具有一致性，因为它只为 C++编译器和库提供了一个定义参考。在编写编译器时，该标准的存在也使编写人员不再需要许可。读者在购买遵循 ANSI 标准的 C++编译器时，就知道会得到什么语言和标准库功能。

1.3 一个简单的 C++程序

下面介绍一个非常简单的 C++程序，了解 C++程序的组成。现在读者不需要输入代码，只是了解一下建立程序的过程。这里也不详细介绍所有的细节，因为这些内容将在后面的章节中探讨。如图 1-2 所示。

图 1-2 中所示的程序会显示如下消息：

```
The best place to start is at the beginning
```

这个程序不是很有用，但说明了几点。该程序由一个函数 main()组成。函数是代码的一个自包含块，用一个名称表示，在本例中是 main。程序中还可以有许多其他代码，但每个 C++程序至少要包含函数 main()，且只能有一个 main()函数。C++程序的执行总是从 main()中的第一条语句开始。

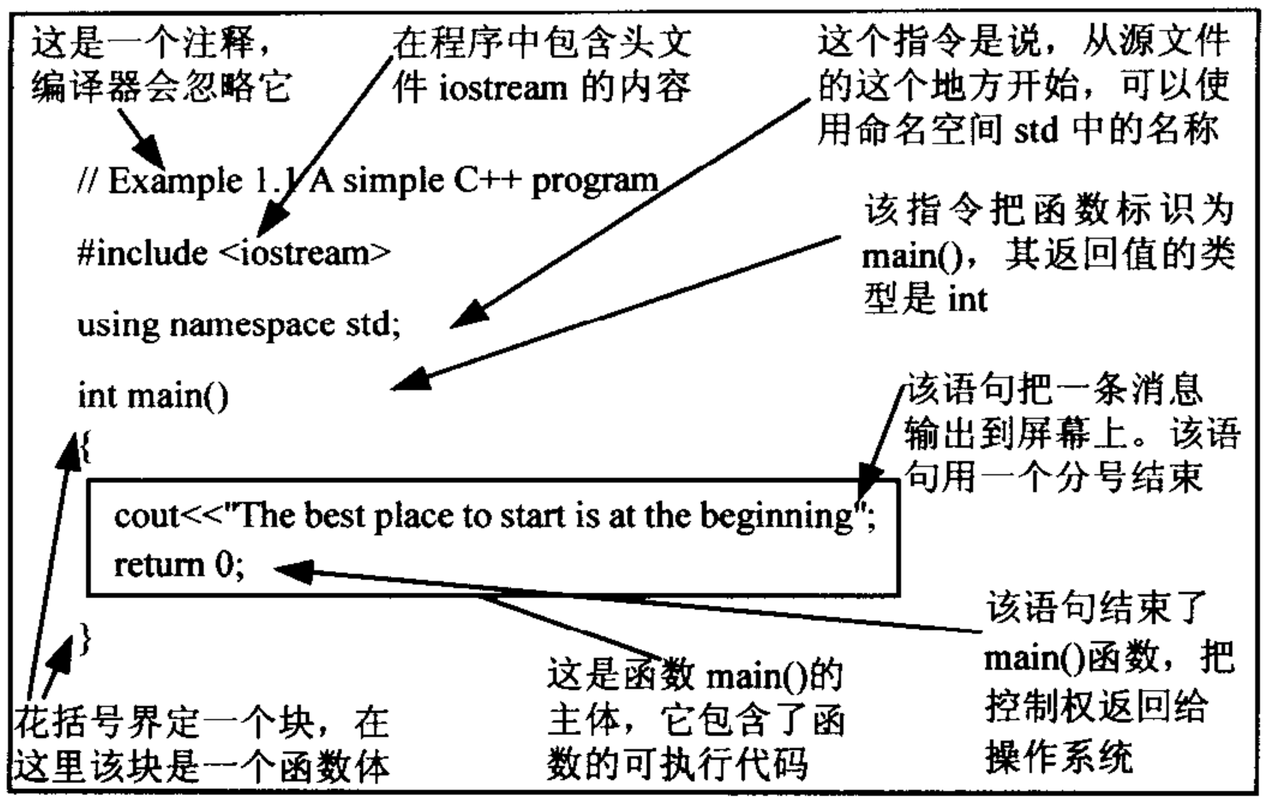


图 1-2　一个简单的 C++程序

该函数的第一行语句是：

```
int main()
```

这行语句指出，这是函数 main 的开始。开头的 int 表示这个函数在执行完后返回一个整数值。因为这是函数 main()，所以最初调用它的操作系统会接收这个值。

函数 main()包含两个可执行语句，每个语句放在一行上：

```
cout << "The best place to start is at the beginning";
return 0;
```

这两个语句会按顺序执行。通常情况下，函数中的语句总是按顺序执行，除非有一个语句改变了执行顺序。第 4 章将介绍什么类型的语句可以改变执行顺序。

在 C++中，输入和输出是使用流来执行的。如果要从程序中输出消息，可以把该消息放在输出流中，如果要输入消息，则把它放在输入流中。因此，流是数据源或数据池的一种抽象表示。在程序执行时，每个流都关联着某个设备，关联着数据源的流就是输入流，关联着数据目的地的就是输出流。对数据源或数据池使用抽象表示的优点是，无论流代表什么，编程都是相同的。例如，从磁盘文件中读取数据的方式与从键盘上读取完全相同。在 C++中，标准的输出流和输入流称为 cout 和 cin，在默认情况下，它们分别对应计算机屏幕和键盘。

main()中的第一行代码利用插入运算符<<把字符串 The best place to start is at the beginning 放在输出流中，[illegible]它输出到屏幕上。在编写涉及到输入的程序时，应使用提取运算符>>。

头文件包含[illegible]代码定义了一组可以在需要时包含在程序源文件中的标准功能。C++标准库中提供的功能存储在头文件中，但头文件不仅仅用于这个目的。我们可以创建自己的头文件，包含自己的代码。在这个程序中，名称 cout 在头文件 iostream 中定义。这是一个标准的头文件，它提供了在 C++中使用标准输入和输出功能所需要的定义。如果程序不包含下面的代码行：

```
#include <iostream>
```

就不会进行编译，因为<iostream>头文件包含了 cout 的定义，没有它，编译器就不知道 cout 是什么。这是一个预处理指令，详见本书后面的内容。#include 的作用是把<iostream>头文件的内容插入程序源文件中该指令所在的位置。这是在程序编译之前完成的。

提示：

在尖括号和标准头文件名之间没有空格。在许多编译器中，两个尖括号<和>之间的空格是很重要的，如果在这里插入了空格，程序就可能不编译。

函数体中的第二个语句，也是最后一个语句：

```
return 0;
```

结束了该程序，把控制权返回给操作系统。它还把值 0 返回给操作系统。也可以返回其他值，来表示程序的不同结束条件，操作系统还可以利用该值来判断程序是否执行成功。一般情况下，0 表示程序正常结束，非 0 值表示程序不正常结束。但是，非 0 返回值是否起作用取决于操作系统。

1.3.1 名称

C++程序中的许多元素都有用来表示它们的名称，也称为标识符。在 C++程序中，可以命名的 5 种元素是：

(1) **函数**。函数是自包含的、可执行代码的命名块。第 8 章将详细讨论如何定义函数。

(2) **变量**。变量是内存中的指定区域，用于存储数据项。第 2 章将论述变量。

(3) **类型**。类型是可以存储的数据种类。例如类型 int 用于存储整数。第 2 章和后续的章节将介绍类型，尤其是第 11 章。

(4) **标签**。标签提供了表示特定语句的方式。它们很少使用，第 4 章将详细介绍。

(5) **命名空间**。命名空间是用一个集合名称标识程序中一组命名项的方式。这听起来可能让人混淆，但不必担心，稍后就介绍命名空间，第 10 章将详细论述。

在 C++中，名称可以包含大小写拉丁字母 a～z 和 A～Z、下划线(_)和数字 0～9。C++的 ANSI 标准还允许在名称中包含通用字符集(Universal Character Set)(稍后介绍)中的字符。

ANSI 标准还允许名称有任意长度，但有的编译器对此有某种长度限制，这个限制常常比较宽(几千个字符)，不是一个严格的限制。

空白在 C++中用于表示空格、垂直或水平制表符、换行符和换页符。不能在名称的中间加上空白字符，否则，编译器就不会把该名称看作是一个名称，而是看作两个或多个名称，从而导致处理不正确。另一个限制是名称不能以数字开头。

下面是一些合法的名称例子：

```
value2    Mephistophele    BettyMay   Earth_weight    PI
```

下面这些名称就不合法：

```
8Ball     Mary-Ann    Bet+y+May   Earth-weight    2PI
```

提示：

包含两个下划线的名称，或者以下划线开头，后跟一个大写字母的名称，是 C++标准库的保留名称，在程序中不应使用这类名称。编译器不会检查这类名称，用户只能在程序出错时发现有一个冲突的名称。

使用扩展字符集的名称

如上一节所述，C++标准允许在名称中包含 UCS 字符。可以把这些字符写作\Udddddddd 格式或\udddd 格式，其中 d 是 UCS 码中某字符的的 16 进制数字。

但没有人希望在名称中包含这样的字符，在名称中嵌入\U 后跟一组 16 进制数字不会提高代码的可读性。允许在名称中使用 UCS 字符的目的是为了让编译器编写人员可以包容用非英语的其他语言编写的字符，例如希腊语、韩语或俄语。

C++标准允许实现编译器，以使用任何字符来指定名称。利用这一点的所有编译器在开始编译代码之前，都必须把非基本集合的字符转换为如前面所示的 UCS 字符的标准表示。例如，在源代码中，有人编写了名称 KHHRa，它对俄国程序员是有意义的。编译器在编译代码之前，先在内部把这个名称转换为 UCS 字符的一个标准化表示，例如/u041A/u043D/u0438/u0433/u0430。实际上，无论在源代码中使用什么字符集指定名称，最终都会得到基本集合中的字符，再加上 UCS 字符\Udddddddd 或\udddd。

在名称中，必须总是显式使用基本字符集中的字符 a～z、A～Z、0～9 和下划线。在名称中为这些字符使用 UCS 码是非法的。其原因是标准没有指定用于基本字符集的编码，所以这个任务就留给了编译器。因此，如果要根据 UCS 码指定基本字符，该字符可能不同于显式指定字符时编译器为该字符使用的编码，从而产生混乱的结果。

注意，并没有要求编译器支持在指定名称时显式使用字符，如果编译器不支持这些字符，在处理之前，这些字符就必须映射为 UCS 格式。遵循该标准的编译器必须在任何情况下都支持名称使用基本字符集中的字符，并允许以本节开头介绍的不太友好的方式使用 UCS 字符。

1.3.2 命名空间

在上面的简单 C++程序中，有一行代码没有解释。为了理解这行代码，需要知道什么是命名空间。为了说明命名空间的含义，下面先讨论名称。上面 C++程序中还没有解释的代码行如下：

```
using namespace std;
```

在前一节讨论的标识符规则中，可以为程序中的元素选择使用任何名称。显然，可以为标准库中已经用于其他目的的元素选择一个名称。同样，如果两个或多个程序员为同一个大型工程的不同部分工作，就会有潜在的名称冲突。显然，为两个或多个不同的元素使用相同的名称会导致冲突，命名空间就解决了这个问题。

命名空间的名称有点像姓氏。家庭中的每个成员都有自己的姓名，在大多数家庭中，每个家庭成员都有一个惟一的名字。在 Smith 家中，有 Jack、Jill、Jean 和 Jonah。在家庭成员之间，用名字来指代每个人。但是，其他家庭的成员可能与 Smith 家的成员有相同的名字。例如，在 Jones 家中，其成员的名字是 John、Jean、Jeremiah 和 Jonah。Jeremiah Jones 在称呼 Jean 时，显然是指 Jean Jones。如果他想指代 Smith 家中的 Jean，就要使用全名 Jean Smith。如果不是这两个家庭的成员，就只能使用每个人的全名来指代他本人，例如 Jack Smith 或 Jonah Jones。

这就是命名空间的作用。命名空间的名称类似于姓氏。在命名空间内部，可以使用其成员的名字。在命名空间的外部，就只能把某个实体的名字和命名空间的名称组合起来，表示该命

名空间中的实体。命名空间的目的是提供一种机制，使大程序的各个部分中因出现重名而导致冲突的可能性降到最低。一般情况下，一个程序中包含几个不同的命名空间。

C++标准库中的实体都是在命名空间 std 中定义的，所以标准库中的所有实体名都用 std 来限定。cout 的全名就是 std::cout，其中的两个冒号有一个非常好听的名称：范围解析运算符，稍后详述。在这个例子中，该运算符把命名空间的名称 std 和流的名称 cout 分隔开来。

在这个简单的 C++程序中，开头的 using 指令表示我们希望在每次引用命名空间 std 中的元素时，不指定命名空间的名称。继续前面的类推，使程序文件成为 std 家族的一组荣誉成员，就可以只用名字来引用每个成员了。其优点之一是不需要把 cout 表示为 std::cout，这样程序代码就更简单。如果省略 using 指令，就必须把输出语句写为：

```
std::cout <<"The best place to start is at the beginning";
```

当然，尽管这使代码略为复杂一些，但这样编写出的代码比较安全，也比较好。Using 指令的作用是允许使用命名空间中的某个名称，而不必用命名空间的名称来限定它。有时也可以这么做：用命名空间的名称明确限定 cout，就不必在程序中使命名空间中的所有名称都可用了。这样，在程序中定义的任何名称和在命名空间中定义的名称就不可能出现冲突了。

如果把这个例子的程序代码改成下面的形式，就比较好：

```
// Program 1.1 A simple C++ program
#include <iostream>
int main() {
  std::cout <<"The best place to start is at the beginning";
  return 0;
}
```

但是，这些代码虽然比较安全，如果代码中有许多 std::cout，代码看起来就很混乱了。我们还必须在程序的许多地方重复输入 std::，在这种情况下，可以使用 using 指令在程序源文件中引入命名空间中的一个名称。例如，使用下面的指令可以在程序文件中引入 std 命名空间中的名称 cout：

```
using std::cout;
```

使用这个指令，可以两全其美：以未限定的方式使用 std 命名空间中的名称 cout，代码中的名称也不会与 std 命名空间中的其他名称冲突，因为它们不使用 std 限定符是不能使用的。程序现在变成：

```
// Program 1.1A A simple C++ program
#include <iostream>
using std::cout;
int main() {
cout <<"The best place to start is at the beginning";
   return 0;
}
```

当然，也可以为每个名称使用 using 指令，把 std 命名空间中的一些名称引入程序文件。应为代码中常用的名称使用 using 指令，通过完全限定的名称来访问 std 命名空间中不太常用的其他名称。

命名空间和 using 指令的内容远比这里介绍的多，详见第 10 章。

1.4 关键字

C++中有一些保留字，称为关键字，它们在 C++语言中有特殊的含义。前面讨论的 return 和 namespace 就是关键字。

本书将介绍更多的关键字。在程序中，实体的名称绝不能与 C++中的关键字相同。附录 B 列出了 C++中使用的关键字的完整列表。

注意：

关键字与 C++语言的其他内容一样，也是区分大小写的。

1.5 C++语句和语句块

语句是指定程序做什么和程序所处理的数据元素的基本单元。大多数 C++语句都以分号结尾。语句有许多不同的种类，最基本的语句是把一个名称引入到程序源文件中的语句。

把名称引入源文件的语句称为声明。声明只是引入名称，指定该名称表示什么，它与定义不同，定义是分配一些内存，来包含名称所指代的内容。大多数声明也是定义。

变量是内存中一个可以存储数据项的空间。下面的语句示例声明了一个变量名，定义并初始化了一个变量：

```
double result=0.0;
```

这个语句把名称 result 声明为一个 double 类型的变量(声明)，把内存分配给该变量(定义)，并设置其初始值为 0.0(初始化)。

下面的例子是另一种类型的语句，称为选择语句：

```
if (length>25)
  boxLength=size+2;
```

这个语句测试一个条件“length 的值大于 25 吗？”，如果条件为真，就执行第二行语句，即给存储在 size 变量中的值加 2，并把计算所得的结果存储在变量 boxLength 中。如果测试的条件不为真，就不执行第二行语句，程序会继续执行后面的语句。

可以把几个语句放在一对花括号{}中，此时这些语句就称为语句块。函数体就是一个语句块，如前面第一个例子所示，main()函数体中的语句就放在花括号中。语句块也称为复合语句，因为在许多情况下，语句块可以看做是一个语句，详见第 4 章中 C++的决策功能。实际上，在 C++中，在可以放置一个语句的任何地方，都可以放置一个包含在花括号对中的语句块。因此，语句块可以放在其他语句块内部，这个概念称为嵌套。事实上，语句块可以嵌套任意级。

语句块对用于存储数据项的变量有重要的作用。第 3 章在讨论变量作用域时将详细介绍。

代码的显示样式

代码排列的方式对代码的可读性有非常重要的影响。这有两种基本的方式。首先，可以使用制表符和空格缩进程序语句，显示出这些语句的逻辑；再以一致的方式使用定义程序块的匹配花括号，使块之间的关系更清晰。其次，可以把一个语句放在两行或多行上，提高程序的可读性。安排匹配花括号和缩进语句的约定称为显示样式。

有许多不同的显示样式可以使用。下面的代码显示了三种常用的代码显示样式：

```
namespace mine {
int test()
{
  if(isGood) {
    good();
    return 0;
  } else
    return 1;
}
--
```

```
namespace mine
{
int test()
{
 if(isGood) {
   good();
   return 0;
 } else
   return 1;
}
}
```

```
namespace mine {
  int test()
  {
    if(isGood) {
       good();
       return 0;
    } else
      return 1;
  }
}
--
```

本书使用上述右边的显示样式，选择它是因为它比较清晰，而且没有过多的空格。使用哪个样式并不重要，只要坚持使用一种样式即可。

1.6 程序结构

每个 C++程序都由一个或多个文件组成。根据约定，用于存储源代码的文件有两类：头文件和源文件。头文件可以包含描述程序所需的数据类型的代码，以及其他类型的声明。这些文件之所以称为头文件，是因为通常在其他源文件的开头包含它们。头文件通常用文件扩展名.h 来区分，但这不是强制的，在一些系统中，也使用其他扩展名来标识头文件，例如.hxx。

源文件的扩展名是.cpp，它包含了函数声明，即程序的可执行代码。这些代码通常引用在自己的头文件中定义的数据类型的声明或定义。编译器在编译代码时，需要知道这些声明或定义，因此应在文件的开头通过#include 指令指定.cpp 文件中需要的.h 文件。#include 指令是编译器的一个指令，它可以把指定头文件的内容插入代码。还需要为代码需要的标准库头文件添加#include 指令。

图 1-3 说明，程序中的源代码包含在两个.cpp 文件和三个头文件中。第一个.cpp 文件使用前两个头文件中的信息，第二个.cpp 文件需要后两个头文件中的内容。第 10 章将介绍#include 指令的更多内容。

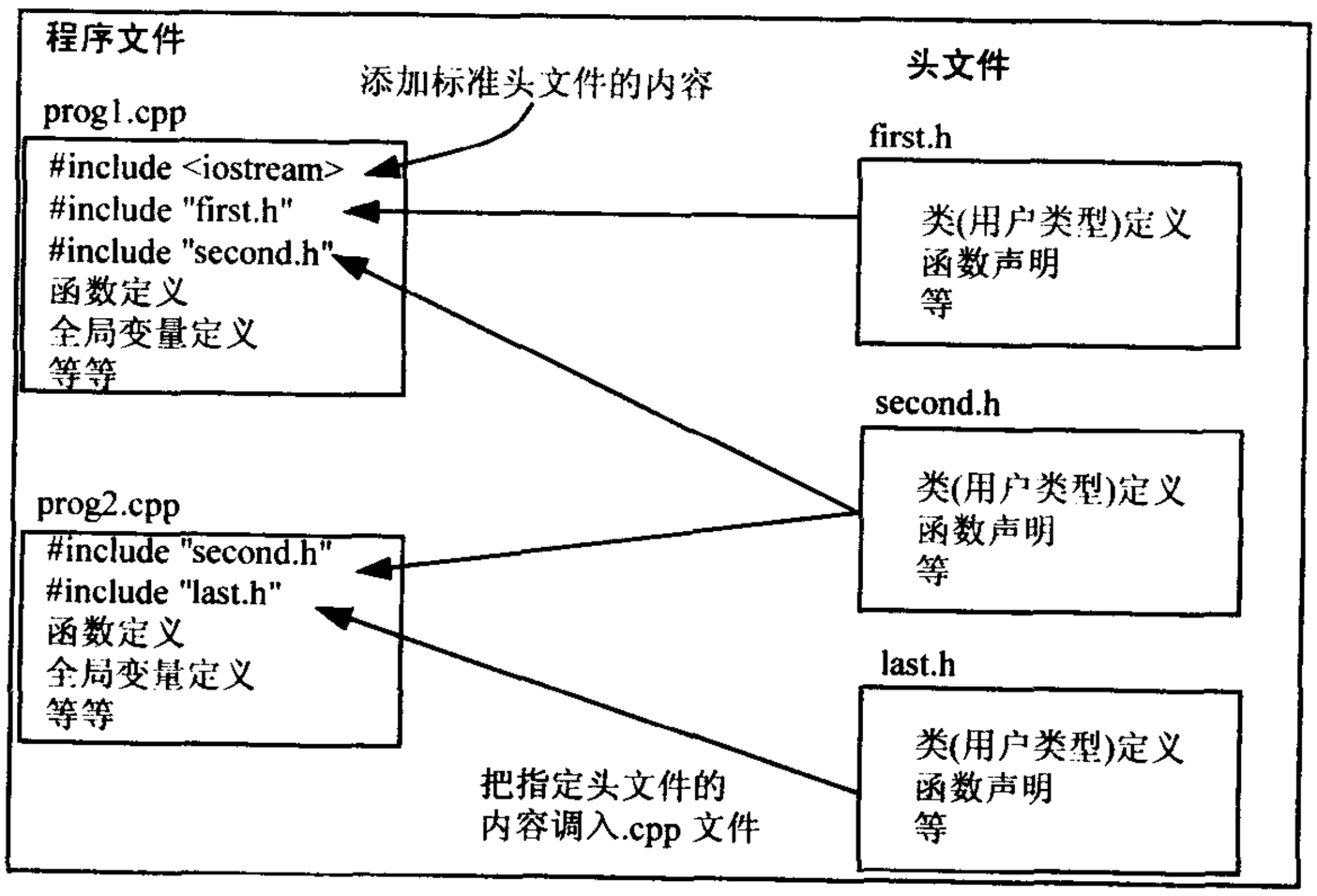


图 1-3 C++程序的源文件

编译器提供了大量的标准头文件，其中包含使用标准库功能所需要的声明。例如，这些头文件包含可用标准库函数的声明。图 1-3 中的第一个.cpp 文件包含<iostream>头文件，在前面的 C++示例程序中就使用过这个头文件。在本例中，C++的头文件名都没有扩展名。实际上，C++的标准头文件名都没有扩展名，这就把它们与其他头文件区分开来。

提示：

附录 C 中有 ANSI/ISO 标准库头文件的详细信息。

编译器系统还有其他许多头文件，为使用操作系统函数提供了所需要的定义，并减少了编程量。这里的例子只使用了几个头文件，但在大多数实际的 C++应用程序中，要使用非常多的头文件。

程序的函数和执行

如前所述，C++程序至少包含一个函数 main()，但程序一般还包含许多其他函数，一些是我们自己编写的，另外一些是标准库函数。程序的函数存储在许多源文件中，其文件的扩展名通常是.cpp，其他扩展名.cxx 和.cc 也较常见。

图 1-4 显示了程序的执行顺序，该程序包含几个函数。main()函数在被操作系统调用时开始执行，程序中的其他函数由 main()或其他函数调用。执行一个函数就称为调用函数。在调用函数时，可以给它传送数据项。要传送给函数的数据项放在调用操作中函数名后面的括号中。在函数执行完后，执行控制就返回到调用函数的地方。

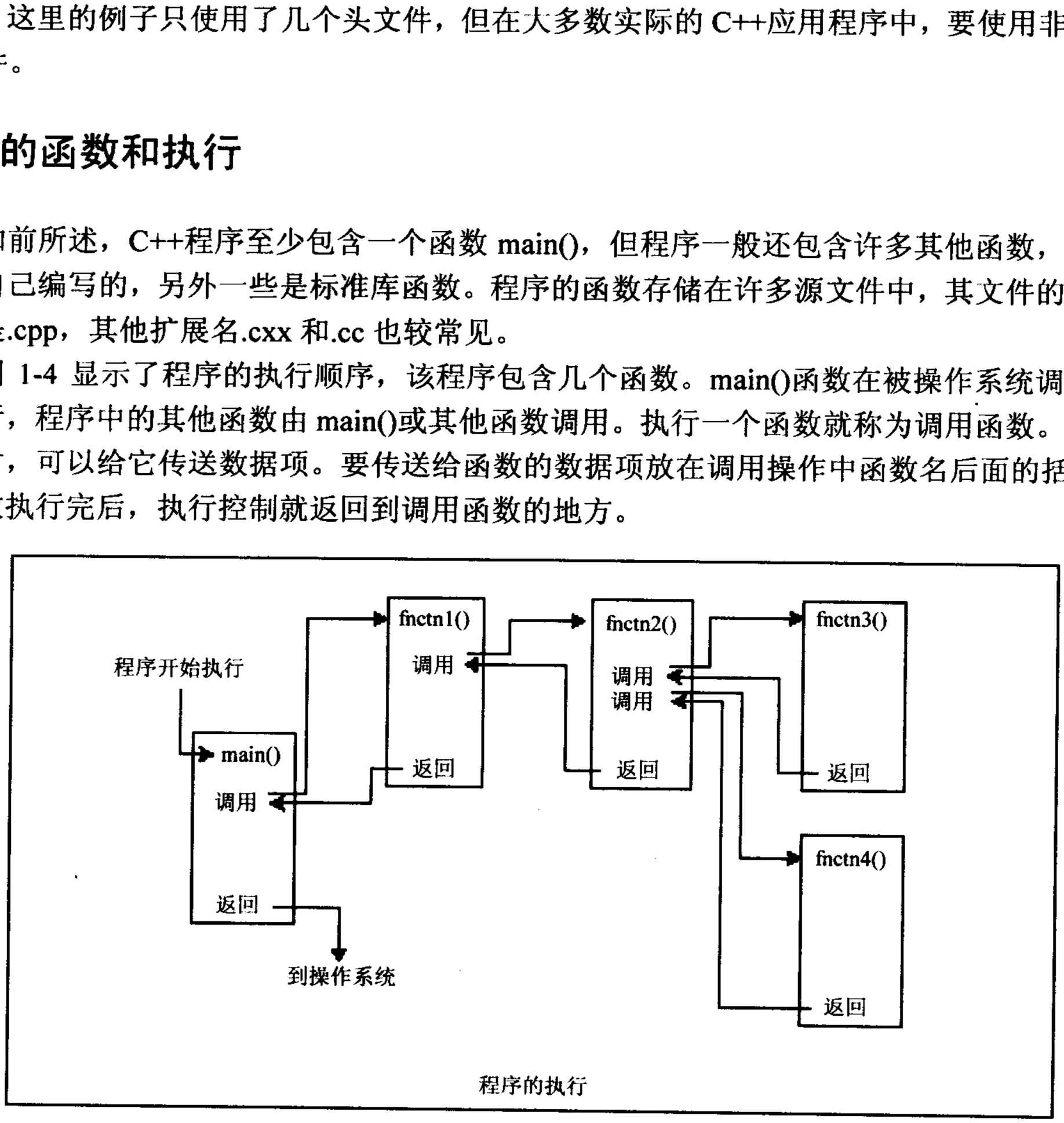


图 1-4　程序函数的执行方式

函数在执行完毕时，还可以把一个值返回到调用的位置上。返回的值可以存储起来，以备以后使用，也可以参与某种类型的计算，例如算术表达式的计算。第 8 章将学习如何创建自己的函数，下一章将使用标准库中的函数。

1.7 从源文件中创建可执行文件

从C++源代码中创建可以执行的程序模块需要两步。第一步是编译器把每个.cpp文件转换为对象文件，其中包含了与源文件内容对应的机器码。第二步是链接程序把编译器生成的对象文件合并到包含完整可执行程序的文件中。

图1-5表明，3个源文件经过编译后，生成3个对应的对象文件。用于标识对象文件的文件扩展名在不同的机器环境上是不同的，这里没有显示。组成程序的源文件可以在不同的编译器运行期间单独编译，但大多数编译器都允许在一次运行期间编译它们。无论采用哪种方式，编译器都把每个源文件看作一个独立的实体，为每个.cpp文件生成一个对象文件。然后在链接步骤中，把程序的对象文件和必要的库函数组合到一个可执行文件中。

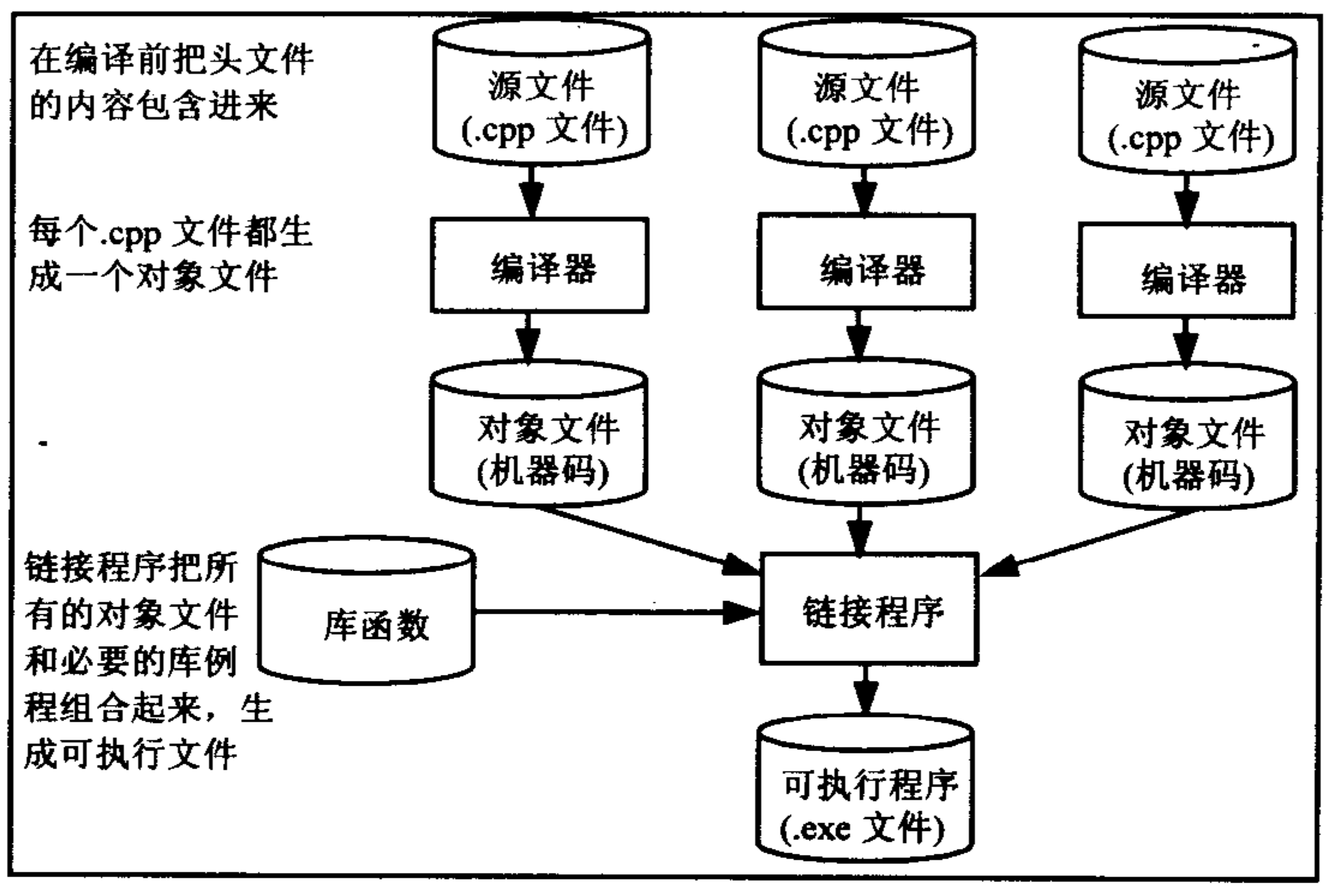


图1-5 编译和链接过程

实际上，编译是一个迭代的过程，因为在源代码中总是会有输入错误或其他错误。更正了每个源文件中的这些错误后，就可以进入链接步骤，但在这一步可能会发现有更多的错误！即使链接步骤生成了可执行模块，程序仍有可能包含逻辑错误，即程序没有生成希望的结果。为了更正这些错误，必须回过头来修改源代码，再编译。这个过程会继续下去，直到程序按照希望的那样执行为止。如果程序的执行结果不象我们宣称的那样，其他人就有可能找到我们本应发现的许多错误，这是毋庸置疑的。一般说来，如果程序非常大，就总是包含错误。

下面详细讨论一下这两个基本步骤(即编译和链接)，因为在这两个步骤中有一些有趣的东西。

1.7.1 编译

源文件的编译过程包含两个主要阶段，如图1-6所示，而它们之间的转换是自动的。第一个阶段是预处理阶段，在正式的编译阶段之前进行。预处理阶段将根据已放置在文件中的预处理指令来修改源文件的内容。#include指令就是一个预处理指令，它把头文件的内容添加到.cpp文件中。还有其他许多预处理指令，详见第10章。

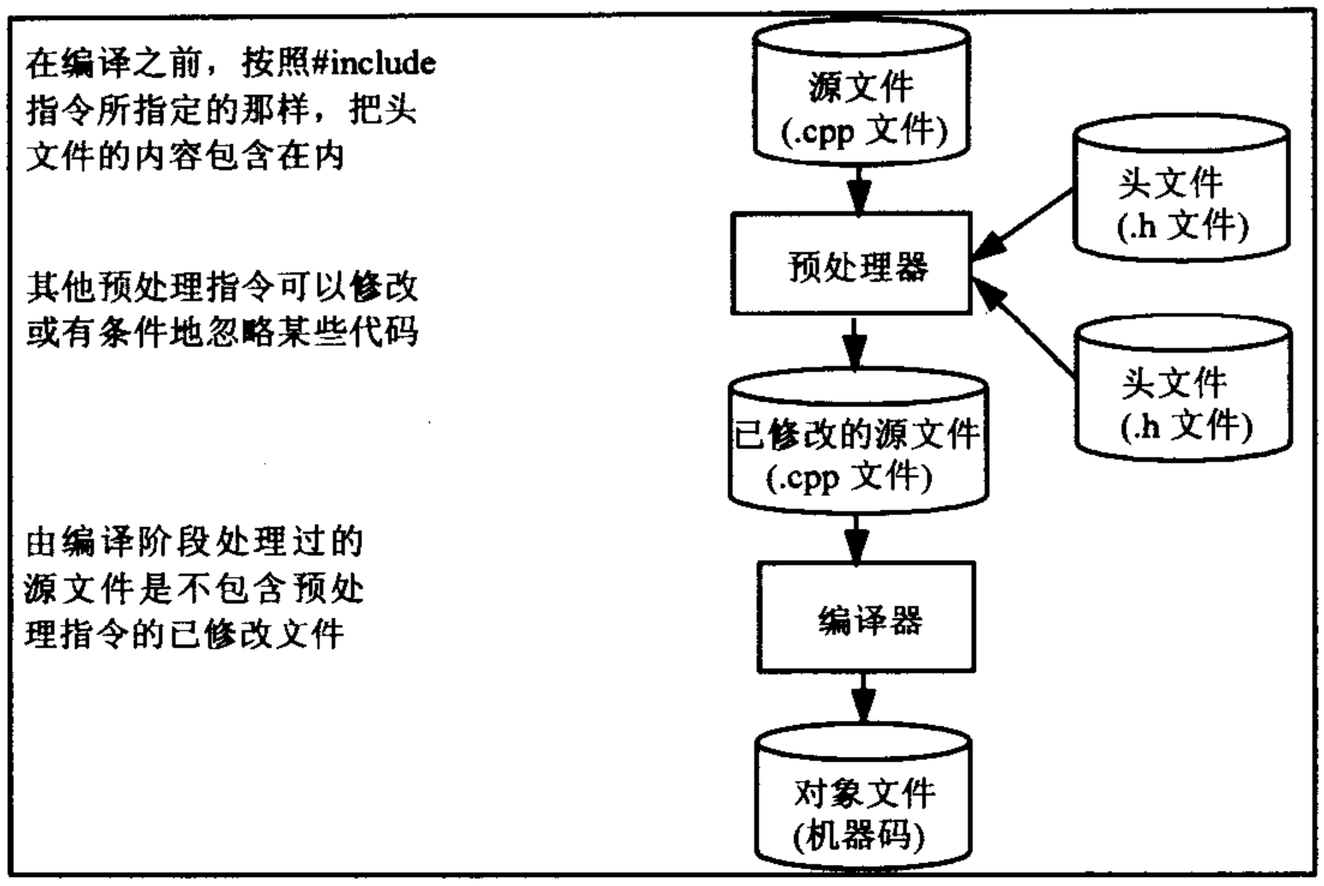


图 1-6　编译过程

这个在编译之前修改源文件的方式提供了很大的灵活性，以适应不同的计算机和操作系统环境。一个环境需要的代码跟另一个环境所需的代码可能有所不同，因为可用的硬件或操作系统是不同的。在许多情况下，可以把用于不同环境的代码放在同一个文件中，再在预处理阶段修改代码，使之适应当前的环境。

在图 1-6 中，预处理器显示为一个独立的操作，但一般不能独立于编译器来执行这个操作。调用编译器会自动执行预处理过程，之后才编译代码。

1.7.2　链接

编译器为给定源文件输出的是机器码，执行这个过程需要较长时间。在对象文件之间并没有建立任何连接。对应于某个源文件的对象文件包含在其他源文件中定义的函数引用或其他指定项的引用，而这些函数或项仍没有被解析。同样，也没有建立同库函数的链接。实际上，这些函数的代码并不是文件的一部分。这些工作是由链接程序(有时称为链接编辑器)完成的。

如图 1-7 所示，链接程序把所有对象文件中的机器码组合在一起，并解析它们之间的交叉引用。它还集成了对象模块所使用的库函数的代码。这是链接程序的一种简化表示，因为这里假定在可执行模块中，模块之间的所有链接都是静态建立的。实际上有些链接是动态的，即这些链接是在程序执行时建立的。

链接程序静态地建立函数之间的链接，即在程序执行之前建立组成程序的源文件中所包含的函数链接。动态建立的函数之间的链接(在程序执行过程中建立的链接)将函数编译并链接起来，创建另一种可执行模块——动态链接库或共享库。动态链接库中的函数链接是在程序调用函数时才建立的，在程序调用之前，该链接是不存在的。

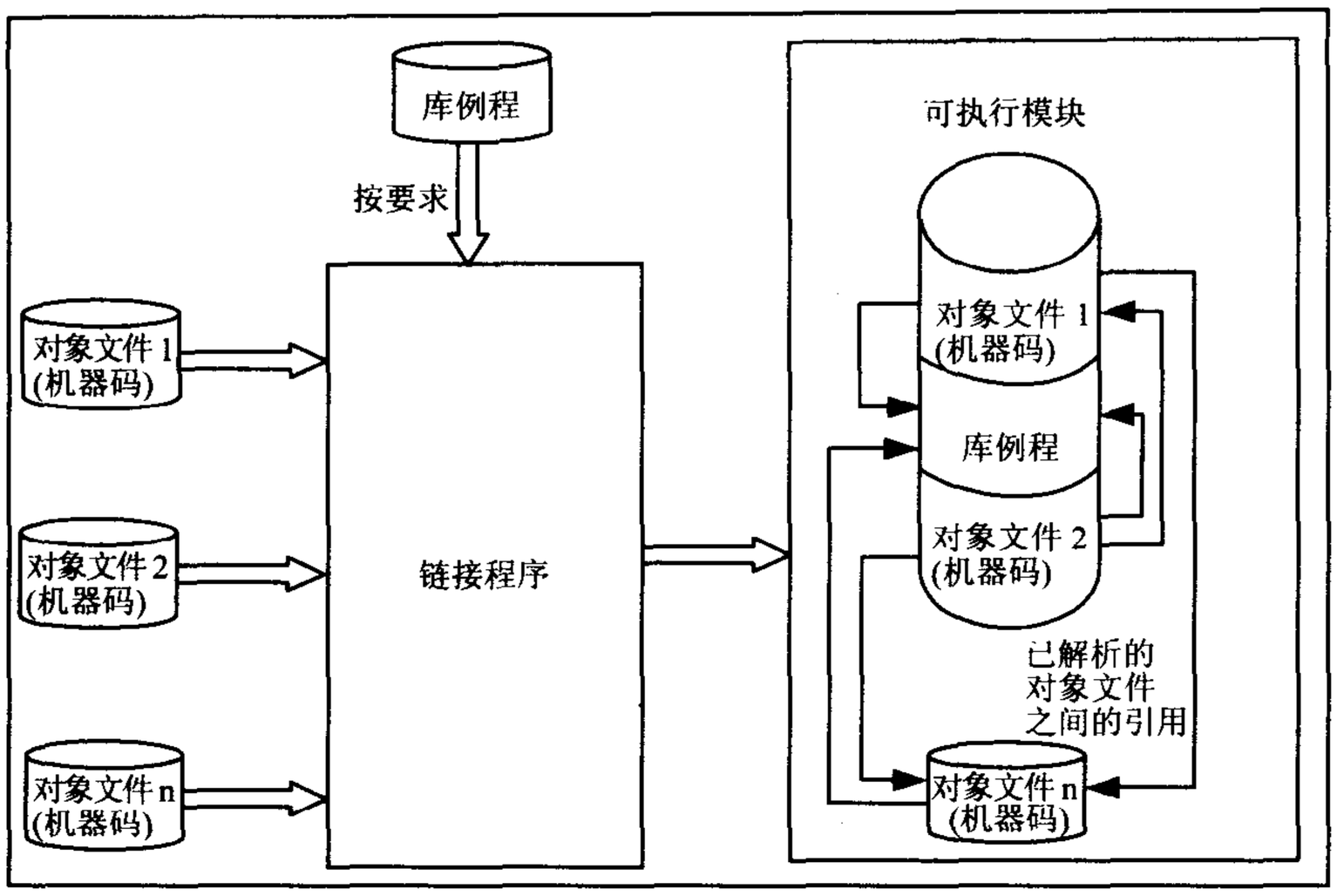


图 1-7　链接过程

动态链接库有几个重要的优点。一个主要的优点是动态链接库中的函数可以在几个并行执行的程序之间共享，在执行的多个函数需要动态链接库中的函数所提供的服务时，这将节省同一个函数占用的内存空间。另一个优点是动态链接库在调用其中的函数之前是不会加载到内存中的。也就是说，如果不使用给定动态链接库中的函数，该动态链接库就不会占用内存空间。动态链接库是与操作系统紧密相关的一个系统功能，本书不再详述。

1.8　C++源字符

编写 C++语句要使用基本源字符集，这些是在 C++源文件中可以显式使用的字符集。显然，用于定义名称的字符集是上述字符集合的一个子集。当然，基本源字符集并没有限制代码中使用的字符数据。程序可以用各种方式创建没有包含在该字符集中的字符串，基本源字符集包括下述字符：

- 大小写字母 A～Z 和 a～z
- 数字 0～9
- 控制字符，如换行符、水平和垂直制表符、换页符
- 字符_{}[]#()<>%:;.?*+-/^&～!=,\" '

这很直观。我们一共可以使用 96 个字符，这些字符可以满足大多数要求。

在 C++中使用的字符定义并没有说明字符的编码方式。编译器将决定用于编写 C++源代码的字符在计算机上如何表示。在 PC 上，这些字符一般在机器上显示为 ASCII 码(例如 ISO Latin-1)，也可以使用其他的字符编码方式。

大多数情况下，基本源字符集足够用了，但偶尔需要使用不包含在基本集合中的字符。前面就提到，可以在名称中包含 UCS 字符。还可以在程序的其他部分包含 UCS 字符，例如指定字符数据。下一节将详细介绍 UCS。

1.8.1 通用字符集

通用字符集 (Universal Character Set，UCS)由 ISO/IEC 10646 标准指定，定义了目前在所有语言中使用的字符编码和其他内容。ISO/IEC 10646 标准定义了几种字符编码形式，最简单的形式是 UCS-2，它把字符表示为 16 位码，所以可以包含 65536 个不同的字符编码，这些字符编码可以写为 4 个 16 进制数 dddd。这种编码描述为基本多语言平台，因为它包含目前使用的所有语言的字符。UCS-4 是 ISO/IEC 10646 标准中的另一个编码，它把字符表示为 32 位码，32 位码可以写为 8 个 16 进制数 dddddddd。UCS-4 包含 40 多亿个不同的编码，能容纳所有需要的字符集。

但这并不是 UCS 的所有内容。例如，另一个 16 位编码叫做 UTF-16(UTF，Unicode Transformation Format)，它与 UCS-2 不同，可以包含的字符超过了 65535 个，通过所谓 16 位码值的代理对给 65535 个以后的字符编码。UCS 还有其他字符编码。一般说来，给定的字符在任何 UCS 编码中的码值都相同，US_ASCII 中的码值与 UCS 字符编码中的码值相同。

无论编译器是否支持扩展字符集来编写源语句，都可以在源代码中包含 UCS 中的字符，把它们指定为编码的 16 进制表示\udddd 或\Udddddddd(其中 d 是一个 16 进制数)。注意第一种形式使用小写 u，第二种形式使用大写 U。但是，不能以这种方式指定基本源字符集中的字符。这是因为这些字符的编码由编译器决定，它们可能与 UCS 编码并不一致。

如果编译器支持扩展字符集和超出基本源字符集中的字符，就可以在源代码中使用这些字符，编译器在开始编译之前，会把它们转换为内部的表示方式。

注意：

由 UCS 标准定义的字符编码与 Unicode 定义的编码相同，所以 Unicode 实际上是 UCS 的另一个名称。如果读者希望了解 UCS 和 Unicode 详细信息，可以参考 http://www.unicode.org。

1.8.2 三字符序列

三字符序列不太常见，但 C++标准允许把某些字符指定为三字符序列。三字符序列就是用于表示另一个字符的三个字符序列。以前为了表示键盘上没有的字符，这是必不可少的一种方法。表 1-1 列出了以这种方式在 C++中指定的字符。

表 1-1　三字符序列的字符

字　　符	三字符序列
#	??=
[	??(
]	??)
\	??/
{	??<
}	??>
^	??'
|	??!
~	??-

编译器会用对应的字符替代它们，再对源代码进行其他处理。

1.8.3 转义序列

在程序中使用字符常量时，某些字符是会出问题的。字符常量是程序以某种方式使用的数据项，它可以是一个字符，也可以是一个字符串，例如前面例子中使用的字符串。显然，不能直接把 newline 或 tab 这样的字符输入为字符常量，因为它们只完成自己该做的工作：在源代码文件中换行或进入下一个制表位置。字符常量中应包含该字符的相应编码。

通过转义序列可以把控制字符输入为字符常量。转义序列是指定字符的一种间接方式，通常以一个反斜杠\开头。表示控制字符的转义序列如表 1-2 所示。

表 1-2 表示控制字符的转义序列

转义序列	控制字符
\n	换行符
\t	水平制表符
\v	垂直制表符
\b	退格字符
\r	回车字符
\f	换页字符
\a	警告字符

还有其他一些字符在直接表示时会出问题。显然，表示反斜杠字符本身是很困难的，因为它表示转义序列的开头。其他控制字符也有其特殊的含义。可以用转义序列指定的“问题”字符如表 1-3 所示。

表 1-3 用转义序列指定的“问题”字符

转义序列	字符
\\	反斜杠
\'	单引号
\"	双引号
\?	问号

由于反斜杠表示转义序列的开始，因此把反斜杠字符输入为一个字符常量的惟一方式是使用两个连续的反斜杠。

转义序列还提供了用键盘不支持的语言来表示字符的一种通用方式，因为可以使用十六进制(基数是 16)或八进制(基数是 8)数字前置一个反斜杠来指定字符的编码。由于使用了数字编码，可以用这种方式指定任何字符。在 C++中，十六进制数据以 x 或 X 开头，\x99A 和\XE3 都是以这种方式进行转义的序列。

还可以使用至多 3 个八进制数字前置一个反斜杠来表示字符，例如\165。没有 x 或 X，就

表示该编码应解释为一个八进制数字。

程序示例 1.1——使用转义序列

下面创建一个程序示例，使用转义序列来指定要显示在屏幕上的消息。要查看该程序的结果，需要输入、编译、链接和执行下面的程序。

如前言所述，如何执行这些步骤取决于编译器，读者可以查看编译器的说明文档，以了解更多的信息。如果查找“编辑”、“编译”和“链接”(在一些编译器中是“建立”)，就可以找到需要的信息。

```
//Program 1.2 Using escape sequences
#include <iostream>
using std::cout;

int main() {
  cout<<"\n\"Least said\n\t\tsoonest mended.\"\n\a";
  return 0;
}
```

在编译、链接和运行这个程序时，会显示如下结果：

```
"Least said
                soonest mended."
```

还会听到一声鸣响或计算机配备的声音输出设备发出的声音。

例子的说明

所得的输出由语句中双引号之间的内容确定：

```
cout<<"\n\" Least said\n\t\tsoonest mended.\ "\n\a";
```

原则上，上述语句中外部双引号之间的所有内容都会发送给 cout。双引号之间的字符串称为字符串字面量。双引号字符表示该字符串字面量的开始和结束；它们不是字符串的一部分。这里说“原则上”是因为字符串字面量中的转义序列会被编译器转换为它表示的字符，所以该字符会发送给 cout，而不是发送转义序列。字符串字面量中的反斜杠总是表示转义序列的开始，所以发送给 cout 的第一个字符是\n，即换行符，它将屏幕光标定位到下一行的开头。

字符串中的下一个字符由另一个转义序列\"指定，所以给 cout 发送一个双引号，并显示在屏幕上，其后是字符串 Least said。接着是另一个换行符\n，因此光标移动到下一行的开头。然后给 cout 发送两个制表符\t\t，让光标向右移动两个制表位置。之后显示字符串 soonest mended，其后是转义序列\"，即显示另一个双引号。最后是另一个换行符\n，它把光标移动到下一行的开头，其后的转义序列\a 表示发出一声鸣响。

字符串内部的双引号不会解释为字符串字面量的结束，这是因为每个双引号的前面都有一个反斜杠，表示这是一个转义序列。如果没有转义序列\"，就无法输出双引号，因为它会被解释为字符串的结尾。

名称 endl 在<iostream>头文件中定义，在输出语句中使用它会输出一个换行符，因此可以使用 endl 代替\n。但\n 和 endl 不完全相同，因为使用 endl 会使输出缓存溢出，并把仍在内存中的字符全部写到输出设备上。而\n 不是这样。显然，不能在字符串字面量中包含 endl，因为

它会解释为四个字母 e、n、d 和 l。

注意：
endl 的最后一个字符是字母 l，而不是数字 1。这有时很难区分。

使用 endl，上述代码中输出字符串的语句就应改写为：

```
cout  << endl
      << "\" Least said"
      << endl
      << "\t\tsoonest mended.\ "\a"
      << endl;
```

这条语句给 cout 按顺序发送了 5 个字符串字面量：end、"\" Least said"、endl、"\t\tsoonest mended.\ "\a"和 endl。其结果与前面的语句相同。当然，为了使该语句可以编译，需要在程序的开头添加另一个 using 指令：

```
using std::endl;
```

不必为换行符选择是使用 endl 还是转义序列。它们不是互斥的，可以根据自己的情况混合使用。例如，用下面的语句也可以生成相同的结果：

```
cout << endl
     << "\" Least said"\n\t\tsoonest mended.\ "\a"
     << endl;
```

这里只对第一个和最后一个换行符使用了 endl。中间的那个换行符仍使用转义序列来生成。当然，输出中 endl 的每个实例都会在流中写入一个换行符，再使输出缓存溢出。

1.8.4 语句中的空白

如前所述，空白是在 C++中描述空格、水平和垂直制表符、换行符及换页符的术语。在许多情况下，空白把语句的一部分与另一部分分隔开，允许编译器标识语句中的一个元素在哪里结束，下一个元素从哪里开始。例如下面的代码行：

```
int fruit;
```

该语句涉及到 int 和 fruit，int 是一个类型名称，而 fruit 是一个变量名称。在 int 和 fruit 之间至少要有一个空白字符(通常是一个空格)，这样编译器才能区分它们。这是因为 intfruit 是一个可接受的变量名称或其他成员的名称，如果中间没有空格，编译器就会把它解释为变量或其他成员的名称。

再看下面的语句：

```
fruit = apples + oranges;
```

在 fruit 和=之间以及=和 apples 之间不需要空白字符，但如果愿意，也可以加入空白。这是因为=不是字母也不是数字，所以编译器会把它跟其周围的字符区分开来。同样，+号的两边也不需要插入空白。实际上，可以插入任意多个空白，例如，把上面的语句改写为：

```
fruit
=
apples
+
oranges;
```

这并不是什么好的编程风格，但编译器不会在意。

除了在语句的元素之间或引号中的字符串内用作分隔符之外，编译器会忽略空白。因此可以在代码中包含任意多个空白，使程序的可读性更高。在一些编程语言中，语句的结尾是代码行的末尾，但在C++中，语句的结尾用分号来表示。因此，可以把一个语句分散放在好几行代码行上，例如一个语句可以编写为：

```
std::cout << std::endl <<"\" Least ::said"<<std::endl
                                <<"\t\tsoonest mended.\ "\a"<<std::endl;
```

也可以编写为：

```
std::cout << std::endl
          << "\" Least said"
          << endl
          << "\t\tsoonest mended.\ "\a"
          << std::endl;
```

1.9 程序的注释

程序代码的注释是非常重要的。在编写代码时，这些代码的含义是很清晰的，但再过一个月后再看这些代码，它们的含义就很模糊了。可以用注释解释代码，在C++中注释有两种形式：单行注释和多行注释(注释可以放在几行上)。

单行注释以双斜杠开头(//)。例如：

```
//Program to forecast stock market prices
```

编译器会忽略双斜杠后面的所有内容，但这并不表示注释要占满一整行。可以使用这种类型的注释来解释一个语句：

```
length = shrink(length, temperature);     //Compensate for wash shrinkage
```

也可以在代码行的开头加上双斜杠，临时删除该行代码：

```
//length = shrink(length, temperature);     //Compensate for wash shrinkage
```

这会把该语句转换为一个注释，例如在测试程序时就可以这么做。代码行中从第一个双斜杠//开始到该行末尾的所有内容都会被忽略，包括其中的所有//。

多行注释有时用于编写较烦琐的、一般描述性材料，例如解释函数中使用的算法。这种注释以/*开头，以*/结尾。在/*和*/之间的所有内容都被忽略。这可以修饰多行注释，以突出显示它们，例如：

```
/*******************************************
* This function predicts future stock prices    *
```

```
* using advanced tea leaf simulation techniques. *
*************************************************/
```

还可以使用这种注释临时禁用一个代码块。方法是在代码块的开头加上/*，在末尾加上*/。但是，必须注意/*...*/注释不能嵌套，否则会使编译器发出错误消息。因为内层嵌套注释的结束*/会与外层注释的开始/*匹配：

```
// You must not nest multiline comments
/* This starts an outer comment
/* This is an inner comment, but the start will not be recognized
   because of the outer comment.
   Instead, the end of the inner comment will be interpreted as the end
   of the outer comment. */
   This will cause the complier to try to compile this part of the
   outer comment as C++ code. */
```

外层注释的最后一部分并没有放在注释块中，编译器会试图编译它，显然这会导致错误。所以，//形式的注释在C++程序中应用最广泛。

注释：

多行注释有时也称为C样式的注释，这是因为/*...*/语法只能用在C语言中，以创建注释。

1.10 标准库

标准库包含了大量的函数和其他支持实体，增加和扩展了C++的基本语言功能。标准库的内容是C++的一部分，在语言的语法和语义方面跟C++相同。C++的标准定义了这两者，所以每个符合该标准的编译器都提供了完整的标准库。

标准库的范围是很特殊的。使用该标准库将获得非常多的功能，包括基本元素如基本语言支持、输入输出函数和异常处理(异常是在程序执行过程中发生的偶然事件，常常是某种错误)，实用函数，数学例程和各种预先编写好并测试通过的功能。在程序执行过程中可借助这些功能来存储和管理数据。

要高效地使用C++，应非常熟悉标准库的内容。本书在介绍C++语言时，将讨论标准库的许多功能，但本书介绍的内容肯定是不完整的。要完整地介绍标准库的功能和用法，需要用与本书同样篇幅的另一本书来讨论。

使用标准库所需要的定义和声明位于前面介绍的标准头文件中。在有些情况下，标准头文件默认包含在程序文件中，但在大多数情况下，必须添加一个#include 指令，把要使用的库功能所在的头文件包含进来。附录C中列出了一个完整的头文件列表，并简要说明了每个头文件支持的功能。

C++标准库中的几乎所有内容都是在命名空间 std 中定义的。也就是说，库中使用的所有名称都应加上前缀 std。如本章前面所述，在引用标准库中的内容时，需要在名称前面加上前缀 std，如下面的语句所示：

```
std::cout << "The best place to start is at the beginning";
```

另外，也可以在源文件的开头加上一个 using 指令：

```
using std::cout;
```

这样就可以使用名称 cout，而不必给它添加 std 前缀了，因此可以使用下面的语句：

```
cout<<"The best place to start is at the beginning";
```

还可以把 std 命名空间的名称引入程序文件：

```
using namespace std;
```

这样，在程序中包含的头文件所定义的所有标准库名称就可以省略前缀 std 了，但是，这也有一个缺点：自己定义的名称与标准库头文件定义的名称可能发生冲突。

本书总是在需要的代码段中包含 std 命名空间前缀。在完整的程序中，一般要为在代码中重复使用的标准库名称添加 using 语句。只使用一两次的名称可以用命名空间的名称来限定。

1.11 用 C++编程

因为 C++继承并改进了原来 C 语言的功能，提高了其灵活性，所以，现在处理时间紧迫的低级编程任务就有了一个丰富的功能库，还可以处理传统过程编程方法擅长处理的问题。但 C++的主要优点是其强大、可扩展的面向对象功能。与以前的过程式程序相比，用 C++编写的程序不易出错，维护所需的时间较短，扩展和理解起来也容易。

这两种编程方法之间有着本质的区别。下面将着重描述它们的区别，并说明面向对象方法吸引人的一些原因。

过程化编程方法和面向对象编程方法

在历史上，过程化编程方法曾是编写几乎所有程序的方式。要创建过程化编程解决方案，必须考虑程序实现的过程，由此才能解决问题。一旦需求明确地确定下来，就可以写出完成任务的大致提纲，如下所示：

- 为程序要实现的整个过程创建一个清晰的高级定义。
- 将整个过程分为可工作的计算单元，这些计算单元应尽可能是自包含的。常常称它们为函数。
- 把每个计算单元的逻辑和工作分解为更详细的动作序列。这就下降到对应于编程语言语句的一级。
- 根据正处理的数据的基本类型：数值数据、单个字符和字符串，来编写函数。

在解决相同问题时，除了开始时对问题进行清晰的说明这一点相同之外，面向对象的编程方法在其他地方都完全不同：

- 根据问题的详细说明确定该问题所涉及的对象类型。例如，如果程序处理的是篮球运动员，就应把 BaseballPlayer 标识为程序要处理的一个数据类型。如果程序是一个账户打包程序，就应定义 Account 类型和 Transaction 类型的对象。还要确定程序需要对每种对象执行的操作集。这将产生一组与应用程序相关的数据类型，用于编写程序。
- 为问题需要的每种新数据类型生成一个详细的设计方案，包括可以对每种对象类型执行的操作。

- 根据已定义的新数据类型及其允许的操作，编写程序的逻辑。

面向对象解决方案的程序代码完全不同于过程化解决方案，理解起来也比较容易，维护也方便得多。面向对象解决方案所需的设计时间要比过程化解决方案长一些。但是，面向对象程序的编码和测试阶段比较短，问题也比较少，所以这两种方式的整个开发时间大致相同。

下面简要论述面向对象方式。假定要实现一个处理各种盒子的程序。这个程序的一个合理要求是把几个小盒子装到另一个大一些的盒子中。在过程化程序中，需要在一组变量中存储每个盒子的长度、宽度和高度。包含几个盒子的新盒子尺寸必须根据每个被包含盒子的尺寸，按照为打包一组盒子而定义的规则进行计算。

面向对象的解决方案首先需要定义一个 Box 数据类型，这样就可以创建变量，引用 Box 类型的对象，并创建 Box 对象。然后定义一个操作，把两个 Box 对象加在一起，生成包含前两个 Box 对象的第三个 Box 对象。使用这个操作，就可以编写如下语句：

```
bigBox = box1 + box2 +  box3;
```

在这个语句中，+操作的含义远远超出了简单的相加。+运算符应用于数值，会象以前那样工作，但应用于 Box 对象时，就有一个特殊的含义。这个语句中每个变量的类型都是 Box，上述代码会创建一个 Box 对象，其尺寸足够包含 box1、box2 和 box3。

编写像这样的语句要比分别处理所有的尺寸容易得多，计算过程越复杂，面向对象编程方式的优点就越明显。但这只是一个很粗略的说明，对象的功能要比这里所描述的强大得多。介绍这个例子的目的是让读者对使用面向对象方法来解决问题有一个大致的了解。面向对象的编程方法基本上是根据问题涉及到的实体来解决问题，而不是根据计算机喜欢使用的实体，即数字和字符，来解决问题。第 11 章将详细讨论 C++中的面向对象编程方法。

1.12 本章小结

本章简要介绍了 C++的一些基本概念。后面将详细讨论本章提及的内容。本章介绍的基本概念如下：

- C++程序至少包含一个 main()函数。
- 函数的可执行部分由包含在一对花括号中的语句组成。
- 一对花括号定义了一个语句块。
- 在 C++中，语句用分号结束。
- 关键字是 C++中有特殊含义的一组保留字。程序中的实体不能与 C++语言中的任何关键字同名。
- C++程序包含在一个或多个文件中。
- 定义函数的代码通常存储在扩展名为.cpp 的文件中。
- 定义数据类型的代码通常存储在扩展名为.h 的头文件中。
- C++标准库提供了支持和扩展 C++语言的大量功能。
- C++中的输入和输出是利用流来执行的，并且需要使用插入和提取运算符，即<<和>>。
- 面向对象的编程方式需要定义专用于某程序的新数据类型。一旦定义好需要的数据类型，就可以根据这些新数据类型来编写程序。

1.13 练习

1. 编写一个程序，在屏幕上输出文本"Hello World"。

2. 修改第 1 题中的程序，使程序使用字符的十六进制值拼写出该短语。如果计算机使用 ASCII 来编码其字符，则可以参考附录 A 中的 ASCII 码值表(提示：使用十六进制 ASCII 值，"He"可以用 std::cout<<"\x48\x65";来显示)。

3. 下面的程序有几处编译错误。请指出这些错误并更正，使程序能正确编译并运行。

```
#include <iostream>
using namespace std;
int main() {
  cout << endl
       << "Hello World"
       << endl

  return0;
}
```

4. 如果第 3 题的程序不包含 using 指令，会出现什么错误？除了使用 using 指令外，还可以用什么方法修正程序？为什么您的解决方案比保留原来的 using 指令更好？

注释：

本书所有练习的答案都在 Apress 网站的 Downloads 部分：http://www.apress.com/book/download.html。

第2章　基本数据类型和计算

本章将介绍在程序中经常使用的、C++内置的一些基本数据类型，并探讨如何执行一些简单的数值计算。C++的面向对象功能全部建立在内置于该语言的基本数据类型的基础之上，因为用户创建的所有数据类型最终都是根据基本类型定义的。因此，很好地掌握它们的用法非常重要。学习完本章后，读者将能编写传统格式(输入-处理-输出)的简单 C++程序。

本章主要内容

- C++中的数据类型
- 什么是字面量，如何在程序中定义它们
- 整型数的二进制和十六进制表示
- 如何在程序中声明和初始化变量
- 使用整型数进行计算的过程
- 用非整型数值——浮点数计算的值编程
- 如何防止修改存储在变量中的值
- 如何创建可以存储字符的变量

2.1　数据和数据类型

C++是一种强类型语言，换言之，程序中的每个数据项都有一个与之相关的类型，该类型定义了数据项的含义。只要可能，C++编译器就会检查数据类型在给定的环境下是否正确，合并的不同类型是否兼容。因为进行这种类型检查，编译器就能检测并报告把一种类型的数据解释为另一种类型时可能发生的大多数错误，或尝试把不兼容的数据类型组合在一起而产生的错误。

在 C++中，可以使用的数值分为两大类：整型数和浮点数(可以是分数)。但是，不能由此判断出只有两种数值数据类型。实际上，在每个大类中，还有几种数据类型，每种类型都有其可以存储的数值范围。在下面的内容中，我们将学习如何在 C++中进行算术运算，首先介绍如何使用整型数进行计算。

2.2　进行简单的计算

首先，介绍一些术语。运算(例如算术运算)是由运算符定义的，例如，+用于相加，*用于相乘。运算符操作的数值称为操作数，在表达式 2*3 中，操作数是 2 和 3。

乘号运算符需要两个操作数，所以称为二元运算符。其他一些运算符只需要一个操作数，称为一元运算符。一元运算符的一个例子是 -2 中的减号。减号可作用于一个操作数，值 2，改变它的符号。表达式 4-2 中的二元减号运算符与此相反，它作用于两个操作数 4 和 2。

2.2.1 字面量

在 C++中，任一类型的固定值，例如 42，2.71828 或"Mark Twain"都称为字面量。第 1 章在屏幕上输出文本字符串时，使用了“字符串字面量”这一术语，它表示用一对双引号括起来的一系列字符所定义的常量。"Mark Twain"就是一个字符串字面量。下面要研究数值常量的字面量类型。这些普通数字我们每天都会遇到：鞋子的尺寸、铅的沸点，站在针尖上的天使个数等所有已定义好的数字。

在 C++中，可以使用两大类数值常量：

- 整型数字面量：即不包含小数点的整数。
- 浮点字面量(通常称为浮点数)：包含小数点或指数或两者都包含的小数(稍后介绍指数)。

当所处理的是一个整型数时，就应使用整型数字面量，例如球队中的队员数或一本书的页数。在数值不是整型数时，应使用浮点字面量，例如圆的周长除以其直径，或英镑与美元的汇率。

在处理非常小或非常大的数值时，也应使用浮点数，例如电子的重量、银河的直径或蝙蝠飞行的速度。使用术语“浮点数”的原因是这些数值是用固定位数的数字和小数点表示的，其中小数点可以根据数字的固定设置左右移动。

小数点浮动

看看下面两个数字：

0.00000000000000000000000001234567	1.234567×10^{-21}
12345670000000000000000000000000.0	$1.234567\times10^{+26}$

这两个数值都有 7 位数，但它们是完全不同的数值。第一个数字非常小，第二个数字非常大。左边每个数字的浮点数表示方式显示在右边。数字乘以 10 的幂会在基本数 1.234567 中移动小数点。使小数点的灵活定位允许用最有限的内存表示和存储范围非常大的数值，包括非常小的数值和非常大的数值。

下面首先介绍如何使用整型数，因为整型数是两者中比较简单的类型。然后讨论浮点数。

2.2.2 整型字面量

可以用非常直接的方式表示整型字面量。下面是一些例子：

```
-123    +123    123    22333
```

其中，前两个例子中的“+”和“-”号是前面提及的一元运算符。在第二个例子中，可以省略“+”，因为这是默认的，但为了使该数值的含义更清晰，加上“+”也不会出问题。字面量+123 与 123 是相同的。第 4 个例子在一般情况下写为 22,333，但在整型字面量中不能使用逗号。如果使用了逗号，编译器就会把该数值当做用逗号分隔开的两个数值。

也不能使用老式的整数值。举一个极端的例子，系统不接受有 100 位的整数。整型字面量是有上限和下限的，这是由存储每种类型的整数值的内存量决定的。本章在后面介绍整型变量时将讨论这个问题，并论述指定整型字面量的更多选项。

当然，可以把整型字面量写成小数值，在计算机中这些值将存储为二进制数值。在编程时理解二进制是非常重要的，为了防止读者对二进制数字的工作方式有误解，附录 E 作了简要的介绍。如果对二进制数和 16 进制数不理解，最好先浏览一下附录 E，再继续下一节的内容。

1. 十六进制的整型字面量

在前面的例子中，整型字面量都是十进制整数，也可以把整数表示为十六进制数。为了说明某个数是十六进制，应在该数值的前面加上 0x 或 0X 前缀。如果某个数是 0x999，就表示这是一个十六进制数，而 999 就是一个十进制数，它们是完全不同的。下面的例子把整型字面量写为十六进制数值。如表 2-1 所示。

表 2-1　十六进制值及对应的十进制值

十六进制值	对应的十进制表示	十 进 制 值
0x1AF	$1\times16^2+10\times16^1+15\times16^0$	431
0x123	$1\times16^2+2\times16^1+3\times16^0$	291
0xA	10×16^0	10
0xCAD	$12\times16^2+10\times16^1+13\times16^0$	3245
0xFF	$15\times16^1+15\times16^0$	255

第 1 章介绍了在定义字符的转义序列中使用的十六进制表示法。现在讨论的是如何定义整数。本章后面还将讨论定义字符字面量。

十六进制的整型字面量主要在定义位的特定模式时使用。因为每个十六进制位都对应二进制值的 4 位，所以很容易把位的特定模式表达为十六进制的字面量。下一章将深入探讨这个问题。

2. 八进制的整型字面量

还可以把整数表示为八进制值，即以 8 为基数。把数值表示为八进制时，要给它加上一个前导 0。表 2-2 列出了八进制值的一些例子。

表 2-2　八进制值及对应的十进制值

八 进 制 值	对应的十进制整数
0123	83
077	63
010101	4161

当然，八进制值的数字只能是 0～7。目前八进制值使用得不多，它在 C++中依旧存在主要是其历史原因：以前计算机处理的字的长度是 3 的倍数。但是，知道存在八进制是很重要的，因为如果不小心使用了一个带有前导 0 的十进制数，计算机就会把它解释为八进制值。

注意：

不要给十进制的整数值加上前导 0。编译器会把这种数值解释为八进制(基数为 8)，因此表示为 065 的值就等价于十进制表示法中的 53。

从编译器的角度来看，它并不在意用户表示整数值时使用什么进制，该数值最终都会在计算机中存储为一个二进制数。在表示整数时使用什么方式只取决于是否方便。可以把整数 15 表示为 15，0xF 或 017，这些数值都将存储为同一个二进制值，所以应选择使用适合于当前环境的某种进制。

2.2.3 整数的算术运算

对整数可以进行的基本算术运算如表 2-3 所示。

表 2-3 基本的算术运算

运 算 符	运 算
+	加
-	减
*	乘
/	除
%	取模(除法运算后的余数)

表 2-2 中的运算符的工作方式与我们期望的大致相同，注意它们都是二元运算符。但是，除法运算略微特殊一点，下面详细说明。整数运算总是得到整数结果，例如，表达式 11/4 的结果不是 2.75，而是 2。“整数除法”返回被除数和除数相除后的整数部分，舍弃了余数部分。在 C++标准看来，除以 0 的结果是不确定的，但特定的实现方式通常定义了其结果，在某些情况下，会提供一种响应的编程方式，所以读者应查阅产品的文档说明。

图 2-1 说明了除法和取模的不同结果。

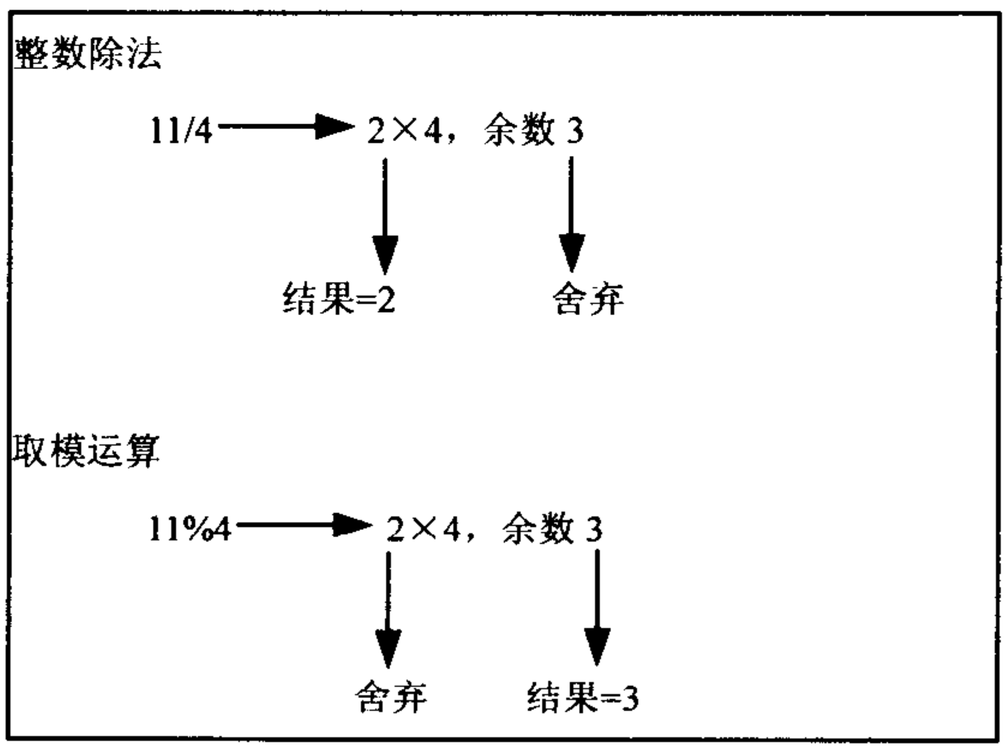


图 2-1 比较除法和取模运算符

取模运算符%有时称为取余运算符，它是对除法运算符的补充，它提供了一种在整数除法运算后获取其余数的方式。表达式 11%4 的结果是 3，这是 11 除以 4 后的余数。取模运算符的一个或两个操作数是负数时，余数的符号由所使用的 C++实现方式来确定，因此不同的系统会有不同的结果。使用取模运算符肯定要涉及相除，所以在右操作数为 0 时，其结果是不确定的。

下面用一个例子来说明算术运算符的使用。

程序示例 2.1——整数的算术运算

下面编写一个程序，输出一个涉及整数的杂项表达式集合的结果，以此来说明算术运算符的运算方式。

```
// Program 2.1 - Calculating with integer constants
#include <iostream>                    //For output to the screen
using std::cout;
using std::endl;

int main() {
cout << 10 + 20                <<endl;     //Output is   30
cout << 10 - 5                 <<endl;     //Output is    5
cout << 10 - 20                <<endl;     //Output is  -10

cout << 10 * 20                <<endl;     //Output is  200
cout << 10 / 3                 <<endl;     //Output is    3
cout << 10 % 3                 <<endl;     //Output is    1
cout << 10 % -3                <<endl;     //Output is    1
cout << -10 % 3                <<endl;     //Output is   -1
cout << -10 % -3               <<endl;     //Output is   -1

cout << 10 + 20 / 10 - 5       <<endl;     //Output is    7
cout << (10 + 20)/(10 - 5)     <<endl;     //Output is    6
cout << 10 + 20/(10 - 5)       <<endl;     //Output is   14
 cout << (10 + 20)/10 - 5      <<endl;     //Output is  - 2

 cout <<4*5/3%4 + 7/3          <<endl;   //Output is      4
 return 0;                                //End the program
}
```

该例子的输出如下：

```
30
5
-10
200
3
1
1
-1
-1
7
6
14
-2
4
```

这种锯齿状的输出格式并不是很好。而这就是默认情况下整数的输出结果。稍后就解释如何使其格式更悦目。下面先讨论这个例子中有趣的地方。

例子的说明

每个语句都计算了一个算术表达式，并在屏幕上输出结果，之后用一个换行符把光标移到下一行的开头。这里的所有算术表达式都是常量表达式，因为它们的值在程序执行前就已由编译器确定了。

前 5 个语句都很简单，它们之所以会输出那些结果的原因也很明显：

```
cout << 10+20              <<endl;     //Output is    30
cout << 10 - 5             <<endl;     //Output is     5
cout << 10 - 20            <<endl;     //Output is   -10

cout << 10*20              <<endl;     //Output is   200
cout << 10/3               <<endl;     //Output is     3
cout << 10%3               <<endl;     //Output is     1
```

因为整数运算将生成整数结果，所以表达式 10/3 的结果就是 3，因为 10 除以 3 的整数部分是 3，余数部分 1 则被舍弃。

下面四行代码说明取模运算符的操作：

```
cout << 10 % 3             <<endl;     //Output is    1
cout << 10 % -3            <<endl;     //Output is    1
cout << -10 % 3            <<endl;     //Output is   -1
cout << -10 % -3           <<endl;     //Output is   -1
```

这是取模运算符的操作数符号的所有组合。第一行代码中的两个操作数都是正的，其输出是在任何系统上运行该代码时惟一一个相同的。其他三行代码的结果可能有不同的符号。

以下 4 个语句说明了括号的作用：

```
cout << 10+20/10 - 5       <<endl;     //Output is    7
cout << (10+20)/(10 - 5)   <<endl;     //Output is    6
cout << 10+20/(10 - 5)     <<endl;     //Output is   14
cout << (10+20)/10 - 5     <<endl;     //Output is   -2
```

括号重写了表达式中运算符的执行顺序。括号中的表达式总是先执行，如果括号是嵌套的，就先执行最内层括号中的表达式，再执行外层括号中的表达式。

在涉及几个不同运算符的表达式中，运算符的执行顺序由运算符的优先权来决定。赋予运算符的优先权称为其优先级。对于前面用于整数算术运算的运算符来说，运算符“*”、“/”和“%”构成一个优先级组，其优先级要高于运算符“+”和“-”，而运算符“+”和“-”又构成了另一个优先级组。运算符“*”、“/”和“%”的优先级总是高于“+”和“-”。给定组中的运算符，例如“+”和“-”，有相同的优先级。本例中最后一个输出语句说明了优先级如何决定运算符的执行顺序：

```
cout << 4*5/3%4 + 7/3 <<endl;  //Output is 4
```

“+”运算符的优先级要低于其他运算符，所以相加运算最后执行。也就是说，两个子表达式 4 * 5 / 3 % 4 和 7 / 3 的值应先计算。子表达式 4 * 5 / 3 % 4 中的运算符具有相同的优先级，所以其执行顺序由运算符的相关性来确定。运算符组的相关性可以是“左”或“右”。具有“左”

相关性的运算符会首先绑定到运算符左边的操作数上，所以这种运算符的执行顺序就是从左到右。下面用例子来说明。

在表达式 4 * 5 / 3 % 4 中，每个运算符都是左相关，也就是说，每个运算符左边的操作数就是该运算符左边的所有内容。例如，乘号*的左操作数是 4，除号/的左操作数是 4*5，取模%运算符的左操作数是 4*5/3，因此该表达式应计算为((4 * 5) / 3) % 4，即从左向右计算。

在表达式中，运算符的相关性决定了同一组中运算符的执行顺序，但这与操作数没有任何关系。例如，在表达式 4 * 5 / 3 % 4 + 7 / 3 中，并没有指定子表达式 4 * 5 / 3 % 4 要在表达式 7 / 3 之前计算，或者表达式 7 / 3 要在表达式 4 * 5 / 3 % 4 之前计算。先计算哪一个表达式都是可以的，这取决于编译器，因为计算的结果是没有区别的。在本例中也是如此：先计算相加的哪一个操作数并不重要，但其所在的环境可能会有差别，稍后在编程过程中将讨论这一点。

2.2.4 运算符的优先级和相关性

在 C++中，几乎所有的运算符组都具有左相关性，所以大多数涉及优先级相同的运算符的表达式都是从左到右计算。惟一一个右相关性的运算符是前面已提及的一元运算符和赋值运算符(稍后介绍)。

下面把整数算术运算符的优先级和相关性列在表 2-4 中，以表示算术表达式中的执行顺序。

表 2-4 整数算术运算符的优先级和相关性

运 算 符	相 关 性
一元 + -	右
* / %	左
+ -	左

表 2-4 中的每一行都是一个优先级相同的运算符组。组的排列也是按顺序的，优先级最高的运算符组放在第一行，优先级最低的运算符组放在最后一行。由于这个表只包含三行，所以非常简单，但后面还将介绍更多的运算符，在学习 C++的过程中会在这个表中添加更多的行。

注意：

C++标准没有直接定义运算符的优先级，但这可以根据标准中定义的语法规则来确定。在大多数情况下，很容易从运算符的优先级上看出给定表达式的执行顺序，而不是从语法规则中看出。所以本书在介绍运算符时都会提及其优先级。

如果要查看 C++中所有运算符的优先级表，可参阅附录 D。

程序示例 2.1A——美化输出结果的外观

前面例子的输出结果肯定是正确的，尽管其看起来并不像。出现“锯齿状”效果的原因是每个整数的输出都放在一个字段宽度中，也就是说，用对应个数的字符来容纳该数值。下面把每个数据项的字段宽度设置为我们选择的值，如下所示：

```
//Program 2.1A - Producing neat output
#include <iostream>                    //For output to the screen
```

```
#include <iomanip>                  //For manipulators
using std::cout;
using std::endl;
using std::setw;

int main() {
  cout<<setw(10)<<10+20                    <<endl;     //Output is 30
  cout<<setw(10)<<10-5                     <<endl;     //Output is  5
  cout<<setw(10)<<10-20                    <<endl;     //Output is -10

  cout<<setw(10)<<10*20                    <<endl;     //Output is 200
  cout<<setw(10)<<10/3                     <<endl;     //Output is  3
  cout<<setw(10)<<10%-3                    <<endl;     //Output is  1
  cout<<setw(10)<<-10%3                    <<endl;     //Output is  1
  cout<<setw(10)<<-10%-3                   <<endl;     //Output is  1
  cout<<setw(10)<<10%3                     <<endl;     //Output is  1
  cout<<setw(10)<<10+20/10-5               <<endl;     //Output is  7
  cout<<setw(10)<<(10+20)/(10-5)           <<endl;     //Output is  6

  cout<<setw(10)<<10+20/(10-5)             <<endl;     //Output is  14
  cout<<setw(10)<<(10+20)/10-5             <<endl;     //Output is -2
  cout<<setw(10)<<4*5/3%4+7/3              <<endl;     //Output is  4
  return 0;                                            //End the program
}
```

这样，输出结果应如下所示：

```
 30
  5
-10
200
  3
  1
  1
 -1
 -1
  7
  6
 14
 -2
  4
```

例子的说明

这看起来就整齐多了。通过改变输出语句，就可以得到这种整洁的格式。输出结果中所显示的每个值都在前面加上 setw(10)，如第一个语句所示：

```
cout<<setw(10)<<10+20      <<endl;    //Output is 30
```

我们将 setw()称为操纵程序(manipulator)，因为它允许操纵或控制输出的外观。操纵程序不输出任何内容，只是修改输出过程。它的作用是把下一个要输出的值的字段宽度设置为括号中指定的字符数，在本例中就是 10。使用 setw()设置的字段宽度只应用于下一个写入 cout 的值。

后续的值会以默认格式显示。

还需要加上另一个#include 语句来包含标准头文件<iomanip>，以使 setw()操纵程序可以在程序中使用。本例还添加了一个 using 指令，以使用未限定的 setw 名称。后面还将在其他例子中使用其他操纵程序。另外，用户也可以用其他字段宽度进行试验，看看其效果如何。

2.3 使用变量

整型常量的运算十分简单，但我们肯定还要在 C++程序中完成更复杂的任务。为此，需要在程序中存储数据项，这种功能可由变量提供。变量是内存中的一个区域，由用户指定的名称来标识，可以存储某种类型的数据项。因此，指定变量需要两方面的内容：必须给变量指定名称，还必须标识要存储的数据类型。首先介绍定义变量名时可以使用的选项。

变量名

如第 1 章所述，给变量指定的名称可以包含任意大小写字母、下划线和数字 0～9 的组合，但必须以字母或下划线开头。ANSI 标准规定，变量名还可以包含 UCS 字符。这将允许编译器使用非基本大小写字母集合的字符。

不能把基本源字符集的字符表示为 UCS 字符。基本源字符集的字符必须表示为它们自己的字符形式。

第 1 章介绍了一些有效变量名的例子，下面的这些变量名也是有效的：

```
value    monthlySalary   eight_ball   FIXED_VALUE  JimBob
```

变量名不能以数字开头，所以 8ball 和 7UP 这样的变量名是无效的。另外，C++语言是区分大小写的，也就是说，republican 和 Republican 是不同的名称。不应使用以下划线开头、后跟一个大写字母的变量名，或者包含连续两个下划线的变量名，因为这种形式的变量名是标准库中使用的保留名称。

一般情况下，变量名应表示出其保存的数据类型。例如，shoe_size 这样的名称与名称 ss 一样，不包含什么特殊的意义，当然应总是假定该名称用于处理鞋子的号码。变量名常常由两个或多个单词组合在一起，这样可以使程序更容易理解。一个常见的方法是在单词之间使用下划线，例如：

```
line_count    pay_rise     current_debt
```

在 C++中经常采用的一种约定是用以大写字母开头的名称表示类，类就是用户定义的类型。第 11 章将介绍如何定义自己的数据类型。采用这种方法，Point、Person 和 Program 就可以标识为用户定义的类型，而不是变量。当然，可以给变量指定任意名称(只要它们不是关键字即可)，但如果选择有意义的名称，以一致的方式命名变量，就会使程序的可读性更高，更不容易出错。C++关键字的完整列表请参阅附录 B。

2.4 整型变量

假定要使用一个变量来记录我们拥有的苹果数。可以用变量的声明语句创建一个变量apples。如图 2-2 所示。

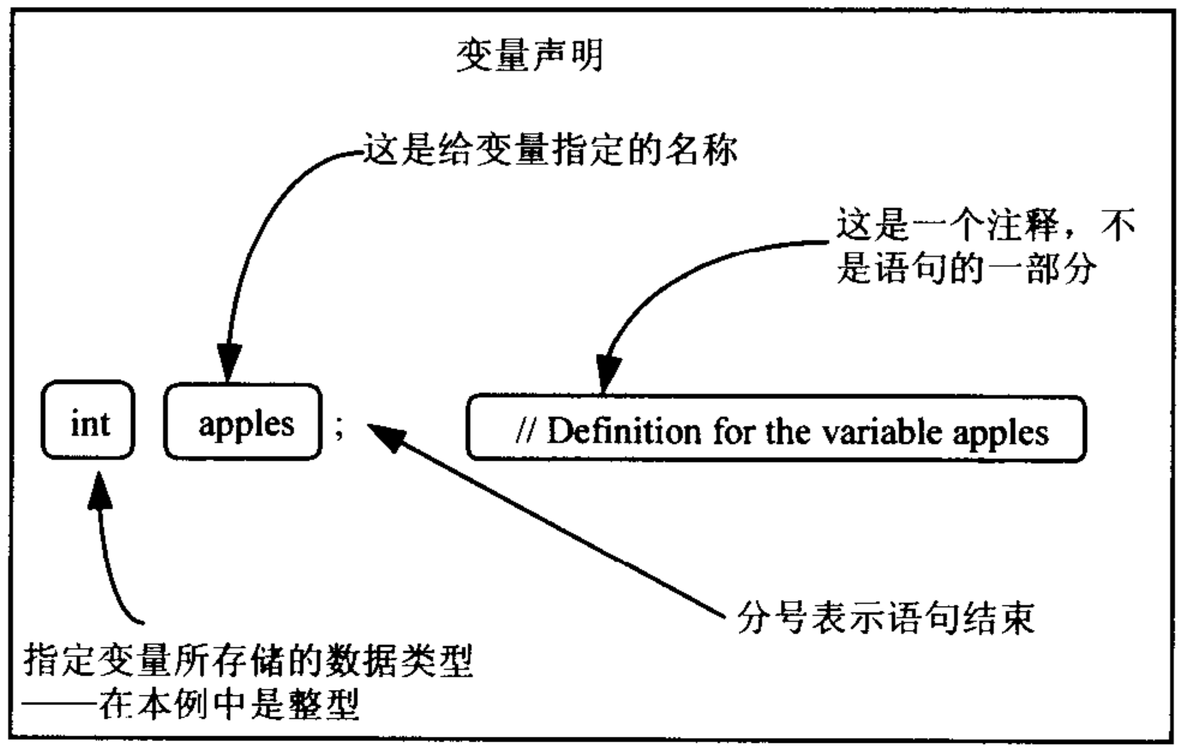


图 2-2 变量的声明

图 2-2 中的这个语句称为声明，因为它声明了名称 apples。把名称引入程序的语句称为该名称的声明。例子中的语句也称为定义，因为它为变量 apples 分配了内存空间。有的声明语句并不是定义。变量根据其定义来创建，所以只有在定义语句之后才能引用这个变量。如果试图在定义之前就引用变量，编译器就会发送一个错误消息。

在定义变量时，还可以指定它的初始值。例如：

```
int apples=10;        //Definition for the variable apples
```

这个语句定义了变量 apples，并把它的初始值设置为 10。图 2-2 中的定义没有指定初始值，所以分配给该变量的内存包含上次使用该内存空间时遗留下来的垃圾值。让垃圾值留在程序中并不好，现在定义第一个黄金规则：

在定义变量时一定要指定其初始值。如果不知道该变量应包含什么值，可以把它初始化为 0。

可以把变量用作前面介绍的算术运算符的操作数，其方式与使用字面量相同。变量的值就是操作数的值。如果对一个变量应用一元减号运算符，就会得到该变量值的负值，但数值是相同的。这不会改变变量中存储的值。稍后讨论它。

下面在一个小程序中使用整型变量。

程序示例 2.2——使用整型变量

下面的程序说明了如何把苹果平均地分给一组小朋友：

```
//Program 2.2 - Working with integer variables
#include <iostream>          //For output to the screen
using std:cout;
using std:endl;

int main() {
```

```
int apples=10;          //Definition for the variable apples
int children=3;         //Definition for the variable children

// Calculate fruit per child
cout << endl            //Start on a new line

     << "Each child gets    //Output some text
     << apples / children   //Output number of apples per child
     << " fruit.";          //Output some more text

// Calculate number left over
cout << endl                    //Start on a new line
     << " We have "             //Output some text
     << apples % children       //output apples left over
     << "left over.";           //Output some more text

cout << endl;
return 0;                       //End the program
}
```

该程序带有非常详细的注释，说明了每个语句的执行情况。一般不需要在注释中放置如此一目了然的信息。这个程序的输出结果如下所示：

```
Each child gets 3 fruits.
We have 1 left over.
```

例子的说明

这个例子不会使读者的脑细胞负担过重。main()中的前两个语句定义了变量 apples 和 children：

```
int apples=10;          //Definition for the variable apples
int children=3;         //Definition for the variable children
```

变量 apples 初始化为 10，变量 children 初始化为 3。还可以在一个语句中定义这两个变量。例如：

```
int apples=10, children=3;
```

此语句把 apples 和 children 都声明为 int 类型，并把它们分别初始化为 10 和 3。使用逗号把要声明的变量分隔开，整个语句用一个分号结束。当然，在这里添加额外的注释并不是很容易，因为空间不够，但可以把该语句分开放在两行上：

```
int apples=10,          //Definition for the variable apples
    children=3;         //Definition for the variable children
```

仍然用逗号把两个变量分隔开，现在就有足够的空间在每一行的末尾添加注释了。可以在一条语句中声明任意多个变量，也可以把语句分散放在任意多行代码上。但是，最好是每条语句只有一个声明。

下一条语句计算每个孩子可以得到几个苹果，并输出结果：

```
cout << endl                                    // Start on a new line
```

```
    << "Each child gets "          // Output some text
    << apples / children           // Output number of apples per child
    <<" fruit.";                   // Output some more text
```

注意这4行代码组成了一条语句，我们把注释放在语句中的每一行上。算术表达式使用除法运算符获得每个小朋友得到的苹果数。这个表达式只涉及刚才定义的两个变量，但一般情况下，可以在表达式中以任意方式混合变量和字面量。

下一条语句计算并输出剩余的苹果数：

```
cout << endl                       // Start on a new line
     << " We have "                // Output some text
     << apples % children          // Output apples left over
     << "left over.";              // Output some more text
```

这里使用取模运算符计算余数，结果是输出语句中的文本字符串。还可以把所有的输出生成为一行语句。另外，也可以在单独语句中输出每个字符串和数据值。

在这个例子中，变量使用了int类型，还有其他整型变量。

2.4.1 整型变量类型

整型变量的类型将决定给它分配多少内存空间，最终决定该变量可以存储的最大和最小值。整型变量有4种基本类型，如表2-5所示。

表2-5 整型变量的基本类型

类 型 名	每个变量的一般内存空间
char	1个字节
short int	2个字节
Int	4个字节
long int	8个字节

char类型总是1个字节，除了char类型之外，在表2-5其他三种类型的整型变量中存储的内存量并没有标准的值。C++标准惟一要求的是该序列中的每种类型都至少要拥有与前一种类型相同的内存空间。这里显示的是在某一系统中这些类型占用的内存大小，而且是比较常见的。short int类型通常写为其缩写形式short，long int类型也通常写为其缩写形式long。这些缩写形式对应于最初的C类型名称，所以C++编译器一般都接受。初看起来，char这个名称对于整数类型来说似乎有点古怪，但如后面所述，它主要用于存储表示字符的整数代码。。

前面介绍了如何声明int类型的变量，可以以完全相同的方式声明short int和long int类型的变量。例如，用下面的语句可以定义并初始化一个short int类型的变量bean_count。

```
short int bean_count = 5;
```

也可以把这个语句改写为：

```
int bean_count = 5;
```

同样，可以用下面的语句声明一个 long int 类型的变量：

```
long int earth_diameter = 12756000L;  //Diameter in meters
```

注意，在初始化值的后面加了一个 L，表示该变量的类型是 long int。如果不加上 L，也不会出问题，编译器会自动把这个值从 int 类型转换为 long int 类型。但最好使初始化值的类型与变量的类型保持一致。

1. 带符号和不带符号的整数类型

short int、int 和 long int 类型的变量可以存储正数值和负数值，所以它们是隐含的带符号的整数类型，如果要明确这一点，也可以把它们改写为 signed short int、signed int 和 signed long int。但它们通常不使用 signed 关键字。

可以只使用 signed 关键字来表示类型，它表示 signed int。但是，这并不常见，因为 int 只需输入三个字符。有时 unsigned int 会简写为 unsigned，这些简写形式都来自于 C。笔者个人的观点是应总是使用关键字 signed 或 unsigned 来指定底层的类型，因为这不会对其含义有影响。

不带符号的整型变量只能存储正数值，显然，这 3 种不带符号的类型分别是 unsigned short int、unsigned int 和 unsigned long int。事先知道只处理正数时，就可以使用这些类型，但它们常常用于存储位模式的值，而不是数值。第 3 章在学习按位运算符时将详细讨论这些不带符号的类型，按位运算符用于操纵变量中的各个位。

我们需要一种区分带符号的整型字面量和不带符号的整型字面量的方式。65535 可以在 16 位中存储为一个不带符号的值，也可以在 32 位中存储为一个带符号的值。不带符号的整型字面量用数字后跟字母 U 或 u 来表示，这可以应用于十进制、16 进制和 8 进制整型字面量。如果要把一个字面量指定为 unsigned long int 类型，就可以使用 U 或 u 和 L。

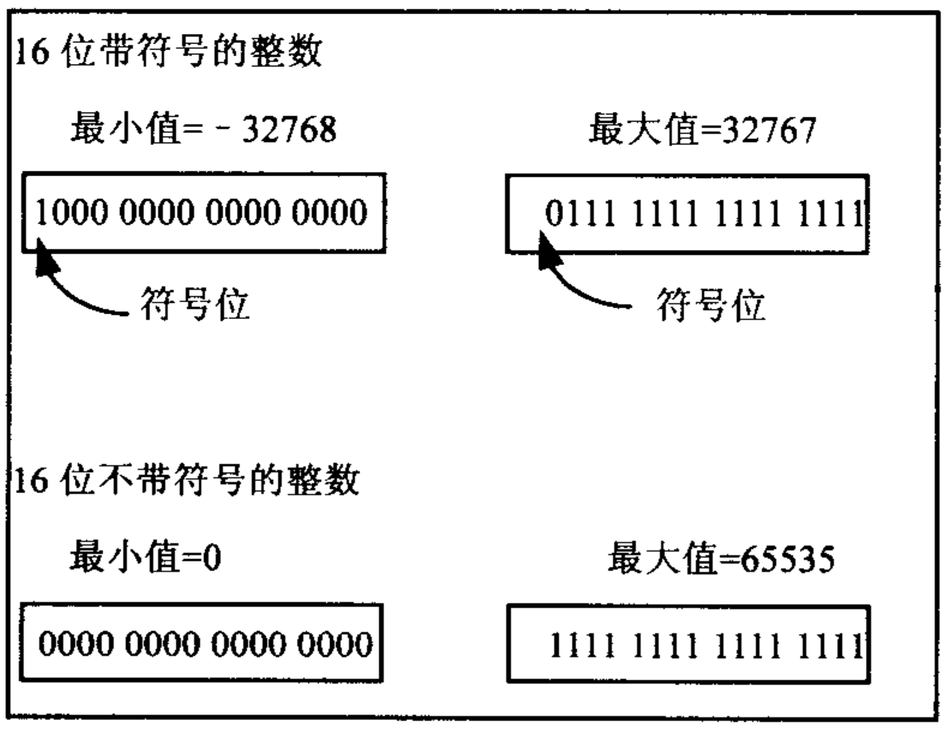


图 2-3　带符号的整数和不带符号的整数

图 2-3 演示了 16 位带符号的整数和 16 位不带符号的整数之间的区别。在带有符号的整数中，最左边的一位表示该数值的符号。它可以是 0，表示正数，也可以是 1，表示负数。对于不带符号的整数，所有的位都可以看作是数据位。由于不带符号的数值总是正的，所以它没有符号位，最左边的一位是数值的一部分。

如果觉得 - 32768 的二进制形式比较奇怪，应注意负数一般表示为 2 的补码形式。如附录 E 所述，要利用 2 的补码形式把正的二进制数值转换为负的二进制数值(或把负的二进制数值转

换为正的二进制数值)，只需反转所有的位，再加 1。当然，不能把+32768 表示为 16 位带符号的整数，因为 16 位带符号的整数的取值范围是 - 32768～ +32767。

2. 带符号和不带符号的 char 类型

存储为类型 char 的值可以是带符号的，也可以是不带符号的，这取决于编译器如何实现它。不同计算机甚至同一台计算机上的不同编译器对 char 类型的实现方式有所不同。如果要用一个字节存储整数值，而不是存储字符编码，就应把变量的类型显式声明为 signed char 或 unsigned char。

注意尽管在任意给定的编译器环境下，类型 char 等价于 signed char 或 unsigned char，但这 3 种类型实际上是不同的。当然，char、short、int、long、signed 和 unsigned 都是关键字。

2.4.2 整数的取值范围

在 C++中，内存的基本单位是字节。就 C++而言，一个字节足以包含 C++编译器使用的基本字符集中的任意字符，但这是没有定义的。只要一个字节可以容纳至少 96 个字符，就满足了 C++标准。也就是说，C++标准中的一个字节至少有 7 位，还可以有更多位，8 位系统目前比较常见。其目的是去除标准对硬件结构的依赖。如果将来生产出了 16 位字节的机器，C++标准将包容它，仍可以使用。不过，目前假定一个字符占用 8 位总是安全的。

在 ANSI C++标准中，并没有规定为每种整型变量分配的内存空间。本节将讨论的内容如下：

- char 类型的变量占用一个字节，可以存储基本字符集中的任一字符。
- int 类型的数值所占用的字节数由被编译程序的硬件环境来定。
- 类型的 signed 和 unsigned 版本占用相同的内存空间。
- short int 类型的数值至少占用与 char 类型相同的字节，int 类型的数值至少占用与 short int 类型相同的字节数，long int 类型的数值至少占用与 int 类型相同的字节数。

总之，类型 char 占用的字节数最少，只有一个字节，类型 long 占用的字节数最多。类型 int 介于两者之间，但占用的字节数最好符合计算机的整数运算功能。其原因是在给定的计算机上，类型 int 占用的字节数应使整数运算达到最高的效率。这取决于机器的结构。在大多数机器上，int 类型是 4 个字节，但随着计算机硬件结构和性能的提高，int 类型很有可能变成 8 个字节。

编译器给每种整数类型分配的字节数决定了该类型可以存储的值的范围。表 2-6 是整型变量常见的取值范围。

表 2-6 整型变量的取值范围

类　　型	字 节 数	取 值 范 围
char	1	- 128～127
unsigned char	1	0U～255U
short	2	- 32768～32767
unsigned short	2	0U～65535U

(续表)

类　型	字 节 数	取 值 范 围
int	4	-2147483648～2147483647
unsigned int	4	0U～4294967295U
long	8	-9223372036854775808L～9223372036854775807L
unsigned long	8	0～18446744073709551615UL

2.4.3 整型字面量的类型

前面介绍了在整数值前面加上前缀会影响该值的基数，还提到后者 U 和 L 用于把整数表示为不带符号的类型或 long 类型。下面就详细讨论这些问题，理解编译器确定给定整型字面量的类型的方式。

首先，表 2-7 总结了整数值的前缀和后缀。

表 2-7　整数值的前缀和后缀

后缀/前缀	说　明
没有前缀	该值是一个十进制值
前缀为 0x 或 0X	该值是一个十六进制值
前缀为 0	该值是一个八进制值
后缀 u 或 U	该值是不带符号的类型
后缀 L 或 l(小写字母)	该值是 long 类型

上表中的后两项可以以任何大小写方式组合，UL、LU、uL、Lu 等都是可接受的。可以使用后缀 l，它是 L 的小写字母，但应避免使用它，因为它很容易与数字 1 混淆。

下面看看编译器是如何解释可以和整型字面量一起使用的前后缀的各种组合的：

- 十进制整型字面量没有前缀，如果其值位于 int 类型的取值范围内，它就解释为 int 类型。否则，就解释为 long 类型。
- 八进制和十六进制字面量如果没有后缀，就根据其取值范围，解释为 int、unsigned int、long 或 unsigned long 类型。
- 字面量带有后缀 u 或 U，如果其值位于 unsigned int 类型的取值范围内，就解释为 unsigned int 类型。否则，就解释为 unsigned long 类型。
- 字面量带有后缀 l 或 L，如果其值位于 long 类型的取值范围内，就解释为 long 类型，否则，就解释为 unsigned long 类型。
- 字面量带有 U 和 L 的大小写组合后缀，就解释为 unsigned long 类型。

如果字面量的值超出了可能类型的取值范围，该行为就是不确定的，编译器通常会发出一个错误消息。

注意，不能把整型字面量指定为 short int 或 unsigned short int 类型。在变量的声明中提供这些类型的初始值时，编译器会自动把字面量的值转换为需要的类型，例如：

```
unsigned short n = 1000;
```

这里根据前面的规则，字面量应解释为 int 类型。编译器会把这个值转换为 unsigned short 类型，并使用它作为变量的初始值。如果使用 - 1000 作为初始值，就不能把它转换为 unsigned short 类型，因为按照定义，负数超出了该类型的取值范围。编译器肯定会发出一个错误消息。

各种整数类型的取值范围都取决于编译器。表 2-6 列出了常见的取值范围，但不同的编译器可能会给整数类型分配不同数量的内存空间，因此提供不同范围的值。在应用程序从一个系统移植到另一个系统上时，还需要注意类型可能的变化。

前面基本上忽略了字符字面量和 char 类型的变量。它们拥有一些独特的特性，所以本章将在后面介绍字符字面量和存储字符编码的变量，而先讨论整数的计算。特别是需要知道如何存储结果。

2.5 赋值运算符

使用赋值运算符=可以把计算的结果存储到变量中。下面看一个例子。假定用下面的语句声明 3 个变量：

```
int total_fruit = 0;
int apples = 10;
int oranges = 6;
```

用下面的语句可以计算出水果的总数：

```
total_fruit = apples + oranges;
```

这个语句首先计算“=”号右边的值，即苹果和桔子的总和，再把结果存储在“=”号左边的 total_fruit 变量中。

等号右边的表达式可以非常复杂。如果定义了变量 boys 和 girls，他们分别包含分享水果的男孩和女孩，用下面的语句可以计算出每个孩子得到的水果数：

```
int fruit_per_child = 0;
fruit_per_child = (apples + oranges) / (boys + girls);
```

注意可以声明变量 fruit_per_child，再直接用表达式的结果对它进行初始化：

```
int fruit_per_child = (apples + oranges) / (boys + girls);
```

可以用任何表达式来初始化变量，只要所涉及的变量都已定义即可。

程序示例 2.3——使用赋值运算符

可以把上面的代码段打包到一个可执行程序中，看看它的执行结果：

```
//Program 2.3 - Using the assignment operator
#include <iostream>
using std::cout;
using std::endl;

int main() {
```

```
int apples = 10;
int oranges = 6;
int boys=3;
int girls=4;

int fruit_per_child = (apples + oranges) / (boys + girls);

cout << endl
     << " Each child gets "
     << fruit_per_child<<" fruit.";
cout << endl;
return 0;
}
```

程序的执行结果如下所示：

```
Each child gets 2 fruits.
```

这就是上面讨论的结果。

2.5.1 多次赋值

还可以在一个语句中执行多次赋值。例如，下面的代码给 apples 和 oranges 赋予相同的值：

```
apples = oranges =10;
```

因为赋值运算符是右相关的，所以这个语句首先在 oranges 中存储值 10，再把 oranges 的值存储在 apples 中，它相当于：

```
apples = (oranges =10);
```

这说明，表达式(oranges =10)有一个值，也就是存储在 oranges 中的值 10。这没有什么可惊讶的。有时需要在表达式中把一个值赋予一个变量，再把该值用于其他目的。可以编写下面的语句：

```
fruit = (oranges =10) + (apples = 11);
```

这个语句在 oranges 中存储值 10，在 apples 中存储值 11，然后把两个变量加在一起，把结果存储在 fruit 中。这说明赋值表达式有一个值。但是，尽管可以编写这样的语句，但最好不要这么做。作为一个规则，应限制每个语句中的操作次数。应提高代码的清晰度，避免其含义模糊，以便于其他程序员以后理解和修改这些代码。

2.5.2 修改变量的值

因为赋值操作先计算右边的内容，再把结果存储在左边的变量中，所以可以编写下面的语句：

```
apples = apples *2;
```

这个语句使用 apples 的当前值，先计算右边的值 apples *2，然后再把结果存储在 apples

变量中。因此，该语句的结果就是使存储在 apples 中的值翻倍。

对变量的已有值进行操作，这种情况已越来越频繁了。因此，C++提供了一种特殊形式的赋值运算符，以缩写方式表达这种表达式。

op= 赋值运算符

之所以称为 op= 赋值运算符，是因为它们由一个运算符和一个等于号“=”组成。使用这样的运算符，上面使 apples 值翻倍的语句就可以改写为：

```
apples *=2;
```

这个语句执行的操作与上一个语句完全相同。apples 变量先乘以等号右边的表达式的值，再把结果存储回 apples 中。等号右边的表达式可以是任意表达式。例如，下面的语句：

```
apples *= oranges + 2;
```

等价于：

```
apples = apples * (oranges + 2);
```

这里先给 oranges 加 2，把相加的结果与存储在 apples 中的值相乘，再把相乘的结果存储在 apples 中(在此不要考虑为什么要把苹果和桔子的数量相乘)。

op= 形式的赋值也可以和加法运算符一起使用，所以，要给 oranges 加 2，可以编写下面的语句：

```
oranges += 2;
```

这等价于：

```
oranges = oranges + 2;
```

现在就产生了一种模式。使用 op= 运算符编写赋值语句的一般形式如下：

```
lhs op = rhs;
```

其中 lhs 是一个变量，rhs 是一个表达式。这等价于语句：

```
lhs = lhs op (rhs);
```

rhs 外的括号表示，表达式 rhs 先计算，其结果是 op 操作的右操作数。

注意：

lhs 是 lvalue，是可以赋值的一个实体。之所以称为 lvalue，是因为它们出现在等号的左边。C++中每个表达式的结果都是 lvalue 或 rvalue。rvalue 是一个结果，而不是 lvalue，即它不能出现在赋值操作的左边。

可以对所有的运算符使用 op= 形式。表 2-8 是一个完整的列表，包括第 3 章将介绍的一些运算符。

表 2-8　op=赋值运算符

操　作	运 算 符	操　作	运 算 符
加	+	按位与	&
减	-	按位或	|
乘	*	按位异或	^
除	/	向左移位	<<
取模	%	向右移位	>>

注意在运算符和“=”之间没有空格。如果包含空格，就会出现错误。

2.6　整数的递增和递减

前面介绍了如何使用赋值运算符修改变量，以及如何用+=运算符递增变量的值。还可以用-=运算符递减变量的值。C++也提供了另外两个不寻常的算术运算符来执行递增和递减任务，它们分别称为递增和递减运算符，即++和 -- 。

这两个运算符并不只是递增和递减的另一个选项，在进一步应用 C++的过程中，就可以看出它们的价值了。递增和递减运算符是一元运算符，可以应用于整型变量。例如，假定变量的类型是 int，下面三个语句完成的任务是一样的：

```
count=count+1;
count +=1;
++count;
```

这 3 个语句都给变量 count 递增 1。最后一种形式使用了递增运算符，显然是最简洁的一种。这个运算符的操作不同于前面介绍的其他运算符，因为它直接修改其操作数的值。表达式的结果是递增变量的值，再在表达式中使用已递增的值。例如，如果 count 的值是 5，则执行下面的语句：

```
total= ++count+6;
```

递增和递减运算符的优先级高于其他二元算术运算符，因此，count 的值先递增为 6，再在赋值运算符的右边表达式中使用这个值 6，所以变量 total 的值就是 12。

可以用相同的方式使用递减运算符：

```
total= --count+6;
```

在执行这个语句之前，假定 count 的值为 6，递减运算符把 count 的值减为 5，这个值再用来计算存储在 total 中的值，结果是 11。

递增和递减运算符的后缀形式

前面都是把运算符放在变量的前面，这称为前缀形式。运算符也可以放在变量的后面，这称为后缀形式，其结果与前缀形式略有不同。在使用++的后缀形式时，先在表达式中使用变量

的值进行计算，再递增该变量的值。例如，把前面的例子改写为：

```
total =  count++  + 6;
```

count 的初始值还是 5，但 total 的值应是 11，因为该语句将使用 count 的初始值计算表达式，再递增 count 的值，使之递增为 6。上面的语句等价于：

```
total= count +6;
++count;
```

在像 a++ + b，甚至 a+++b 这样的表达式中，其含义并不是很明显，或者不清楚编译器会执行什么操作。这两个表达式的含义是相同的，但第二个表达式也可能意味着 a + ++b，它的含义就不同了，等价于另外两个表达式，如下表达更清晰：

```
total= 6 + count++;
```

另外，还可以使用括号：

```
total = (count++) +6;
```

前面应用于递增运算符的规则也适用于递减运算符。例如，如果 count 的初始值是 5，则语句：

```
total =- -count + 6;
```

total 的值是 10。如果将语句改写为：

```
total = 6 + count- -;
```

total 的值就是 11。

必须避免在一个表达式中多次使用这些运算符的前缀形式。假定变量 count 的值是 5，则语句：

```
total = ++count * 3 + ++count * 5;
```

首先，这个语句理解起来很费劲，其次也是最重要的是，该语句多次修改了变量的值，在C++中，结果就是不确定的，编译器会给这个语句生成一个错误消息，但在某些情况下不会出错。在程序中这可不是一个理想的功能，所以不要在一个语句中多次修改变量的值。

还要注意，下面语句的结果也是不确定的：

```
k = ++k + 1;
```

这个语句递增赋值运算符右边的变量的值，因此在一个表达式中对变量k的值修改了两次。计算一个表达式只能对每个变量修改一次，变量以前的值只能用于确定要存储的值。根据 C++标准，这种表达式是不确定的，但这并不表示编译器不会编译它。这只表示不能保证结果的一致性。

递增和递减运算符通常应用于整数，尤其常用于循环，详见第 5 章。在后面的章节中，它们可应用于浮点数。后面的章节及探讨它们如何应用于 C++中的某些其他数据类型，并能够得到特别且非常有用的结果。

2.7 const 关键字

我们常常需要在程序中使用某种类型的常量，例如一月的天数或圆周率π(圆的周长与其直径之比)，甚至烘炉中的面包数。但是，在计算过程中应避免显式使用数值字面量，而使用初始化为指定值的变量会更好。例如，把码转换为英尺时，并不一定要把一个值与 3 相乘，而可以把该值与已初始化为 3 的 feet_per_yard 变量相乘，这样会使代码的含义更明确。在程序中显式使用的数值字面量有时称为幻数(magic number)，尤其在这些字面量的目的和来源不很明显的情况下，就更是如此。

用变量代替幻数的另一个原因是可以减少代码中要维护的点数。假定幻数表示时常变化的某个内容，例如利率，而且该幻数还出现在代码中的好几个地方。当利率变化时，就需要修改程序中的所有这些幻数。如果定义了一个变量，就只需在初始化时修改这个值。

当然，如果用一个变量存储这样的常量，就希望固定该值，防止不小心修改它。为此，可以使用关键字 const。例如：

```
const int feet_per_yard = 3;        //Conversion factor yards to feet
```

可以把任何类型的“变量”定义为 const，编译器会检查用户是否试图修改这种变量的值。例如，如果在赋值运算符的左边加上了 const，就会出现错误。因此，在把变量声明为 const 时，必须为它提供一个初始值。

注意把变量声明为 const 会改变它的类型。类型为 const int 的变量与类型为 int 的变量完全不同。

程序示例 2.4——使用 const

下面编写一个小程序，把用码、英尺和英寸表示的长度转换为用英寸表示的长度。

```
//Program 2.4 -Using const
#include <iostream>
using std::cin;
using std::cout;
using std::endl;

int main() {
  const int inches_per_foot=12;
  const int feet_per_yard=3;
  int yards=0;
  int feet = 0;
  int inches=0;

  //Read the length from the keyboard
  cout<< "Enter a length as yards, feet, and inches: ";
  cin >>yards>>feet>>inches;

  //Output the length in inches
  cout << endl
       << "Length is inches is "
       << inches + inches_per_foot *(feet+feet_per_yards*yards)
```

```
        << endl;
  return 0;
}
```

结果如下：

```
Enter a length as yards, feet and inches: 2  2  11

Length is inches is 107
```

例子的说明

与前面的例子相比，本例多了一个 using 语句：

```
using std::cin;
```

这个语句把 std 命名空间中的名称 cin 引入程序文件，该名称表示标准输入流，即键盘。

下面的语句定义了两个转换常量：

```
const int inches_per_foot=12;
const int feet_per_yard=3;
```

用关键字 const 声明它们，将避免直接修改这些变量。添加下面的语句可以测试一下：

```
inches_per_foot=15;
```

在常量的声明之后使用这样的语句，程序就不再编译。

用下面的语句提示输入，并读取 yards、feet 和 inches 的值：

```
cout << "Enter a length as yards, feet and inches: ";
cin >> yards>>feet>>inches;
```

注意第二行代码使用流 cin 指定几个连续的输入操作，为此使用了第 1 章简要介绍的提取运算符<<，它的用法类似于流输出操作 cout，可以输入多个值。插入和提取运算符明确说明了数据的流向。

从键盘输入的第一个值存储在 yards 中，第二个值存储在 feet 中，第三个值存储在 inches 中。这里的输入处理非常灵活，可以在一行上输入 3 个值，其中用空格分隔开(实际上是用空白字符分隔开)，也可以分几行输入它们。

在输出语句中执行英寸的转换：

```
cout << endl
     << "Length is inches is "
     << inches + inches_per_foot *(feet+feet_per_yards*yards)
     << endl;
```

转换因子声明为 const，只要不修改它们，就不会影响它们在表达式中的使用。

2.8 整数的数字函数

第 8 章将详细介绍函数，但在此之前仍可以使用标准库中的一些函数。下面将简要讨论使

用函数的场合和函数的一些术语。

函数是一个自包含的被命名的代码块，执行某个特定的任务，这常常要对所提供的数据执行一些操作，再把操作的结果返回给程序。如果函数返回了一个数字值，该函数就可以像一般的变量那样参与算术表达式。一般情况下，函数的调用如下：

```
FunctionName(argument1, argument2, ...)
```

根据所使用的函数，可以为它提供 0 个、1 个或多个值，在程序中调用该函数时，应把这些值放在函数名后面的括号中。以这种方式传送给函数的值称为变元。与 C++中的所有值一样，传送给函数的变元和函数返回给程序的值都是有类型的，必须遵循其类型的规则，才能正确使用函数。

在源文件的开头为头文件<cstdlib>添加#include 指令，就可以访问几个应用于整数的数字函数了。这个头文件的内容与原来的 C 库头文件 stdlib.h 相同。下面将通过几个例子语句来学习这些函数。

abs()函数返回变元的绝对值，其变元可以是 int 类型或 long 类型。数字的绝对值就是不包括符号的数值部分，所以负数的绝对值就是其数字，但符号是正的，而正数的绝对值是其本身。abs()函数返回的值与变元有相同的类型，例如：

```
int value = -20;
int result = std::abs(value);    // Result is 20
```

头文件<cstdlib>还定义了 labs()函数，它也生成 long 类型的变元的绝对值。包含这个函数的原因是旧式的 C 程序可能使用它，但最好只使用 abs()函数。

div()函数有两个变元，它们的类型都是 int，该函数返回第一个变元除以第二个变元的结果，相除的余数采用 div_t 类型的结构形式。后面将详细介绍结构，现在通过一个例子学习如何访问 div()函数返回的商和余数：

```
int value = 93;
int divisor = 17;
div_t results = std::div(value, divisor);          //Call the function
std::cout << "\nQuotient is"  << results.quot;     //Quotient is 5
std::cout << "\nRemainder is" << results.rem;      //Remainder is 8
```

前两个语句定义了变量 value 和 divisor，并给它们指定初始值 93 和 17。下一个语句调用 div()函数，对 value 除以 divisor。函数返回的结果是 div_t 类型的结构，它存储在变量 results 中，该变量也是 div_t 类型。在第一个输出语句中，在 results 名称的后面加上 quot，并用一个句点隔开，以访问 results 中的商，这个句点叫作成员访问运算符，这里使用它访问 results 结构的 quot 成员。同样，在最后一个语句中，使用成员访问运算符输出余数，它可以通过 results 结构的 rem 成员来获得。div_t 类型的所有结构都有 quot 成员和 rem 成员，使用成员访问运算符总是可以访问它们。

注意可以直接把字面量用作 div()函数的变元。此时，调用函数的语句如下：

```
div_t results = std::div(93, 17);
```

ldiv()函数执行的操作与 div()函数相同，但其变元是 long 类型。返回的结果是 ldiv_t 类型的结构，其成员 quot 和 rem 也是 long 类型。

注意：

<cstdlib>头文件是从 C 继承来的，所以许多实现方式包含了原来 C 函数的定义，它们没有在 std 命名空间中定义，所以函数名不使用 std 限定符。这将允许 C 程序在相同的环境下编译和执行，但在 C++中编码时，应给这些函数加上 std 限定符。

生成随机数

在程序中生成随机数是非常有用的。例如游戏程序中就需要有随机性，否则它们很快让人厌烦。<cstdlib>头文件定义了一个生成随机整数的函数rand()。更明确地说，该函数会生成伪随机整数。根据定义，随机数是不能预测的，所以数字算法生成的任意序列数字都不是真正随机的，只是表面上看起来像是随机的而已。但是，理解了这一点后，就可以把 rand()函数生成的数字看作随机数。注意 rand()不是一个杰出的随机数生成器，许多应用程序都使用更专业的随机数生成器。

rand()函数把一个随机整数返回为 int 类型。该函数不需要任何变元，所以可以按照如下方式使用它：

```
int random_value = std::rand();  //A random integer
```

这里把 rand()函数返回的整数存储在变量 random_value 中，也可以在算术表达式中使用它，如下所示：

```
int even = 2 * std::rand();
```

rand()函数返回的值是从 0 到 RAND_MAX 之间的一个值。RAND_MAX 是一个在<cstdlib>中定义的符号。在代码中使用 RAND_MAX 时，编译器会用一个整数值代替它。在一些系统中，它表示值 0x7fff，在另外一些系统中，它有不同的值，它至多可以是 0x3fffffff，这是可以存储为 int 类型的最大整数。如果 RAND_MAX 的值是 0x3fffffff，就不能像以前那样，把 rand()生成的值乘以 2，这可能会生成不正确的结果。下一章将解决这个问题。

RAND_MAX 由一个预处理的宏定义(第 10 章介绍预处理的宏)，它不在 std 命名空间中，所以使用时不需要限定名称。由宏定义的任何符号都不在 std 命名空间中，因为它不是一个表示什么的名称。在编译器开始编译代码时，这样的符号就不再出现，因为它已经在预处理阶段被其他内容替代了。

数字序列的随机化

使用前面介绍的 rand()，数字序列总是相同的，这是因为该函数在生成随机数的算法中使用默认的种子值。这非常适合于测试，但一旦有了可以工作的游戏出现，就希望每次运行程序时都有不同的序列。为此，可以修改用于生成随机数的种子值，方法是把一个新的种子值作为整数变元传送给在<cstdlib>中定义的 srand()函数，例如：

```
std::srand(13);  //Set send for rand to 13
```

srand()函数的变元必须是 unsigned int 类型的值。前面的语句将使 rand()产生一个与默认不同的序列，但我们真正需要的是在每次执行程序时，都要从 rand()函数中用一个随机种子生成不同的序列。幸好，计算机的时钟是随机种子值的一个很好的来源。

<ctime>标准库头文件定义了几个与日期和时间相关的函数。这里要介绍 time()函数，因为这就是获得随机种子值的方式。time()函数返回自从 1970 年 1 月 1 日以来过去的秒数。如果使用这个秒数作为种子，每次执行程序时都会使用不同的种子值，这个值返回为 time_t 类型，这个类型是在标准库中定义的，等价于一个整数类型，通常是 long 类型。返回类型指定为 time_t 类型，就为不同 C++实现方式处理返回值类型提供了一定的灵活性。可以使用 time()函数为随机数序列创建种子，如下所示：

```
std::srand((unsigned int)std::time(0));
```

这里要注意几个问题。time()函数的变元是 0，该变元还有另外一种可能，但这里不需要，所以忽略它。子表达式(unsigned int)把 time()函数的返回值转换为 unsigned int 类型，这是 srand()方法需要的变元类型。没有它，这个语句就不会编译。类型转换在以后介绍。

下面用一个例子来说明随机数的生成。

程序示例 2.5——生成随机整数

下面是代码：

```
//Program 2.5 Using Random Integers
#include <iostream>
#include <cstdlib>
#include <ctime>
using std::cout;
using std::endl;
using std::rand;
using std::srand;
using std::time;

int main() {
  const int limit1 = 500;  //Upper limit for on set of random values
  const int limit2 = 31;   //Upper limit for another set of values

  cout << "First we will use the default sequence from rand(). \n ";
  cout << "Three random integer from 0 to " << RAND_MAX << ": "
       << rand() << "  " << rand() << "  " <<rand() << endl;

  cout << endl << "Now we will use a new seed for rand().\n ";
  srand((unsigned int)time(0));          //Set a new seed

  cout << "Three random integer from 0 to " << RAND_MAX <<": "
       << rand() << "  " <<rand()<<"  " << rand()<<endl;
  return 0;
}
```

结果如下：

```
First we will use the default sequence from rand().
Three random integer from 0 to 32767: 6334  18467  41

Now we will use a new seed for rand().
Three random integer from 0 to 32767: 4610  32532  28452
```

例子的说明

这是 rand()函数的一个简单用法，首先使用默认种子开始序列：

```
cout << "A random integer from 0 to " << RAND_MAX << ": "
     << rand() << endl;
```

对 rand()的每次调用都返回一个 0 到 RAND_MAX 之间的数值，这个函数调用三次，就会获得三个随机整数序列。

接着，使用下面的语句把种子值设置为系统时钟的当前值：

```
srand((unsigned int)time(0));          //Set a new seed
```

在每次执行程序时，这个语句都会设置一个不同的种子，接着重复前面用默认种子集执行的语句，因此，每次运行这个程序时，第一组总是生成相同的结果，而第二组的输出应是不同的。

2.9 浮点数

不是整数的数值存储为浮点数，浮点数在内部分为 3 个部分：符号(正号或负号)、尾数(大于或等于 1、且小于 2 的数值，其位数是固定的)和指数。当然在计算机中，尾数和指数都是二进制值，但为了解释浮点数的工作原理，下面把它们当作十进制数来讨论。

浮点数的值是带符号的尾数值乘以以 10 为底的指数幂。如表 2-9 所示。

表 2-9 浮点数值

符号(+/-)	尾　数	指　数	值
-	1.2345	3	1.2345×10^3(即 -1234.5)

浮点字面量有 3 种基本形式：

- 小数形式，包括一个小数点，例如 110.0。
- 指数形式，例如 11E1，其中十进制数部分要乘以以 10 为底，以 E 之后的数字为指数的幂。指数前面的 E 可以是大写，也可以是小写。
- 使用小数点和指数，例如 1.1E2。

上面的例子都对应于同一个值 110.0。注意浮点字面量内部不允许有空格，例如，不能写成 1.1 E2。1.1 E2 会被编译器解释为两个值 1.1 和 E2(即 100.0)。

注意：

浮点字面量必须包含一个小数点或指数，或者两者都包含。如果数值字面量不包含这两者，那么该数值就是一个整型数。

2.9.1 浮点数的数据类型

浮点数的数据类型有 3 种，如表 2-10 所示。

表 2-10　浮点数的数据类型

数 据 类 型	说　　明
float	单精度浮点数
double	双精度浮点数
long double	扩展的双精度浮点数

这里的术语“精度”是指尾数中的位数。上述数据类型的精度按从上到下的顺序逐步增加，float 在尾数中的位数最少，long double 的位数最多。注意精度只确定尾数中的位数。某一类型表示的数值的取值范围主要由指数的可能范围确定。

C++的 ANSI 标准并没有描述精度和数值范围，所以这些类型的精度和数值范围就由编译器决定，编译器通常会最大限度地利用计算机提供的浮点数功能。一般情况下，long double 类型提供的精度大于等于 double 类型，double 类型提供的精度大于等于 float 类型。

通常，float 类型提供 7 位精度，double 类型提供 15 位精度，long double 类型提供 19 位精度，但 double 类型和 long double 类型在一些编译器上的精度是相同的。除了精度有所增加之外，double 类型和 long double 类型的取值范围也在扩大。

在 Intel 处理器上，浮点数类型表示的取值范围如表 2-11 所示。

表 2-11　浮点数类型的取值范围

类　　型	精度(位数)	取值范围(+或 -)
float	7	$1.2\times10^{-38}\sim3.4\times10^{38}$
double	15	$2.2\times10^{-308}\sim1.8\times10^{308}$
long double	19	$3.3\times10^{-4932}\sim1.2\times10^{4932}$

表 2-11 中数字的精度都是大约数。显然，这些类型都可以表示 0，但不能表示 0 和正负范围中下限之间的值，所以这些下限是非 0 值中最小的值。

简单的浮点字面量只带有一个小数点，它是 double 类型，下面就看看如何定义这种类型的变量。可以使用关键字 double 指定浮点数变量，如下面的语句所示：

```
double inches_to_mm=25.4;
```

这个语句把变量 inches_to_mm 声明为 double 类型，并把它的值初始化为 25.4。在声明浮点数变量时，也可以使用 const，在需要浮点数常量时，就可以这么做。如果希望修改变量的值，声明语句应如下所示：

```
const double inches_to_mm=25.4;       //A constant conversion factor
```

如果不需要 double 变量提供的精度和取值范围，可以选择使用关键字 float 来声明浮点数变量。例如：

```
float pi = 3.14159f;
```

这个语句定义了一个变量 pi，并将其初始值设置为 3.14159。字面量尾部的 f 表示这是一个 float 类型。如果没有 f，该字面量就是 double 类型，这不会出什么问题，但编译器会发出一

个警告消息。还可以使用大写字母 F 来表示浮点数字面量是 float 类型。

要指定类型为 long double 的字面量，应在数值的最后加上大写或小写字母 L。用下面的语句就可以声明并初始化这种类型的变量：

```
long double root2 = 1.41421356237309504880L;     //Square root of 2
```

2.9.2 浮点数的操作

取模运算符%不能用于浮点操作数，但前面介绍的其他二元算术运算符如+、-、*和/，都可以用于浮点操作数。还可以对浮点数变量应用前缀和后缀形式的递增及递减运算符，其作用与处理整数相同，变量会递增或递减 1。

与整型操作数一样，根据标准，除 0 的结果也是不确定的。但一些 C++实现方式有自己的处理方式，所以读者应参阅产品的文档说明。

在目前的大多数计算机上，硬件的浮点操作都是根据 IEEE 754 标准(也称为 IEC 559)实现的。C++标准并不需要 IEEE 754，但 IEEE 754 提供了在应用 IEEE 754 标准的机器上表示浮点操作的一些方面。浮点标准定义了几个特殊的值，它们的二进制尾数都是 0，指数都是 1，根据其符号表示+infinity 和 - infinity。在一个非 0 的正数除以 0 时，结果就是+infinity，一个非 0 的负数除以 0 时，结果就是 - infinity。IEEE 754 定义的另一个特殊的浮点值称为 Not a Number，通常缩写为 NaN，用于表示数学上没有定义的结果，例如 0 除 0 或无穷大除以无穷大。

一个或两个操作数是 NaN 时，所有后续的操作结果都是 NaN。程序中的一个操作得到值±infinity，就会影响该值参与的所有后续操作。把一个正常的值与±infinity 加起来会得到±infinity，±infinity 除以±infinity 或±infinity 乘以 0 会得到 NaN。表 2-12 总结了这些操作。

表 2-12　NaN 操作数的浮点操作

操　作	结　果	操　作	结　果
±N/0	±infinity	0/0	NaN
±infinity±N	±infinity	±infinity/±infinity	NaN
±infinity*N	±infinity	Infinity- Infinity	NaN
±infinity/N	±infinity	Infinity *0	NaN

使用浮点数变量是很简单的，但没有使用这种变量的经验终究不太好，所以下面举一个这方面的例子。

程序示例 2.6——浮点数的算术运算

假定要构建一个圆形的池塘，其中喂养一些鱼。通过研究发现，必须保证该池塘的表面积为 2 平方英尺，才能确保每条鱼有 6 英寸长。本例需要确定池塘的直径，以确保鱼有足够的空间。下面就是实现过程：

```
//Program 2.6 -Sizing a pond for happy fish
#include <iostream>
#include <cmath>                          //For square root calculation
using std::cout;
```

```
using std::cin;
using std::sqrt;

int main() {
  const double fish_factor=2.0/0.5;        //Area per unit length of fish
  const double inches_per_foot = 12.0;
  const double pi=3.14159265;

  double fish_count = 0.0;                 //Number of fish
  double fish_length = 0.0;                //Average length of fish

cout << "Enter the number of fish you want to keep: ";
cin >> fish_count;
cout << "Enter the average fish length in inches: ";
cin >> fish_ length;
fish_ length = fish_ length/ inches_per_foot;   //Convert to feet

//Calculate the required surface area
double pond_area = fish_count *fish_length * fish_factor;

//Calculate the pond diameter from the area
double pond_diameter = 2.0 * sqrt ( pond_area / pi );

cout << " \n Pond diameter required for " << fish_count <<" fish is "
     << pond_diameter << " feet. \n";
  return 0;
}
```

输入 20 条鱼，每条鱼的平均长度为 9 英寸，这个例子的输出如下所示：

```
Enter the number of fish you want to keep: 20
Enter the average fish length in inches: 9
Pond diameter required for 20 fish is 8.74039 feet.
```

例子的说明

首先声明要在计算中使用的 3 个 const 变量：

```
const double fish_factor=2.0/0.5;        //Area per unit length of fish
const double inches_per_foot = 12.0;
const double pi=3.14159265;
```

注意使用一个常量表达式来指定 fish_factor 的值。可以使用任意表达式来生成合适类型的结果，以定义变量的初始化值。这里把 fish_factor、inches_per_foot 和 pi 声明为 const，因为不希望修改它们的值。

接着声明存储以后输入的变量：

```
double fish_count = 0.0;                 //Number of fish
double fish_length = 0.0;                //Average length of fish
```

不必初始化它们，但最好进行初始化。

输入的鱼长度使用英寸作为单位，所以需要把它转换为英尺，再在鱼池的计算中使用它：

```
fish_ length = fish_ length/ inches_per_foot;   //Convert to feet
```

这会把转换后的值存储回原来的变量。

使用下面的语句获得池塘所需要的面积：

```
double pond_area = fish_count * fish_length * fish_factor;
```

fish_count 和 fish_length 的乘积给出了所有鱼的总长，把这个值与 fish_factor 相乘，就得到了需要的面积。

圆的面积可以由公式πr^2得到，其中 r 是半径。所以，可以计算出池塘的半径，即面积除以 π，再开方。直径是半径的两倍，整个计算过程由下面的语句完成：

```
pond_diameter = 2.0 * sqrt ( pond_area / pi );
```

使用一个在标准头文件<cmath>中声明的函数来获得平方根。sqrt()函数返回函数名后面的括号中表达式的值的平方根。在本例中，因为表达式的值是 double 类型，所以返回值的类型是 double，而返回的 float 值的平方根就是 float 类型。<cmath>头文件包含了许多其他标准库数值函数的声明，稍后介绍这些函数。第 8 章将详细解释函数，包括如何定义自己的函数。当然，与标准库中的其他名称一样，sqrt 也是在 std 命名空间中定义的，所以可以在程序文件的开头为该名称添加一个 using 声明。

退出 main()之前的最后一步是输出结果：

```
cout << " \n Pond diameter required for " << fish_count <<" fish is "
     << pond_diameter << " feet. \n";
```

这输出了指定鱼数所需要的池塘直径。

2.9.3 使用浮点数值

对于大多数使用浮点数值的计算来说，类型 double 就够用了。但是，应了解使用浮点数变量的局限和缺点。如果不小心，结果可能不准确，甚至不正确。下面是使用浮点数值时常见的错误原因：

- 一些小数值没有准确转换为二进制浮点数值。在计算过程中，很容易把一些小错误放大为大错误。
- 考虑两个非常接近的数值之间的区别会丧失精度。如果考虑两个 float 数值的区别，而这两个数值仅在第 6 位的数字有区别，那么其结果是只有一或两位是精确的，其他位则可能出错。
- 处理范围较宽的数值会导致错误。可以用一个简单的例子来验证一下：把两个值存储为精度为 7 位的 float 类型的浮点数，可是，其中一个值比另一个值大 10^8 倍，对它们执行相加操作。把较小的值加到较大值上任意多次，较大的值是不会有明显变化的。<cfloat>头文件为浮点类型定义了常量，在最小值上加 1.0 会得到不同于的结果。这些常量是 FLT_EPSILON、DBL_ EPSILON 和 LDBL_ EPSILON。

下面看看这些错误在实际中是如何放大的(虽然这是一种有点虚拟的情况)。

程序示例 2.7——浮点数计算中的错误

下面的例子说明了前两个浮点数是如何组合在一起而出错的：

```
//Program 2.7 Floating point errors
#include <iostream>
using std::cout;
using std::endl;

int main() {
  float value1 = 0.1f;
  float value2 = 2.1f;
  value1 -= 0.09f;                    //Should be 0.01
  value2 -= 2.09f;                    //Should be 0.01
  cout<<value1 - value2<<endl;        //Should output zero
  return 0;
}
```

显示出来的值应是 0，但这个程序的结果如下：

```
7.45058e-009
```

例子的说明

产生错误的原因是存储的数值不准确。如果添加代码，输出 value1 和 value2 修改后的值，就会看到它们之间的差异。

当然，value1 和 value2 的值之间的最终差异是非常小的，但可以在其他可能放大错误的计算过程中使用这个完全伪造的值。如果把这个结果乘以 10^{10}，就会得到 7.45，而该结果本应是 0。同样，如果比较这两个值，希望得到相等的结果，但实际上得不到这样的结果。

注意：

在程序代码中，不要依赖十进制值的精确浮点数表示。

改变输出

前面的程序以非常明智的方式输出浮点数值，浮点数值的小数位数为 5 位，并使用科学计数法(即尾数加指数的形式)。还可以通过某些输出操纵程序，使用“正常”的小数表示法来显示结果。

程序示例 2.8——更多的输出操纵程序

下面的例子与前面的例子相同，但使用其他操纵程序改善了输出结果：

```
//Program 2.8 Experimenting with floating point output
#include <iostream>
#include <iomanip>
using std::setprecision;
using std::fixed;
using std::scientific;
using std::cout;
using std::endl;

int main() {
  float value1=0.1f;
```

```
float value2=2.1f;
value1 -= 0.09f;                    //Should bd 0.01
value2 -= 2.09f;                    //Should bd 0.01

cout<<setprecision(14)<<fixed;      //Change to fixed notation
cout<<value1 - value2<<endl;        //Should output zero

cout<<setprecision(5)<<scientific;  //Set scientific notation
cout<<value1 - value2<<endl;        //Should output zero

return 0;
}
```

运行修改后的程序，得到的结果如下：

```
0.00000000745058
7.45058e-009
```

例子的说明

这段代码使用了 3 个新的操纵程序：setprecision()、fixed 和 scientific。setprecision()指定了用多少位数来表示浮点数，fixed 和 scientific 互相补充，选择显示浮点数的格式。

在默认情况下，C++编译器会根据输出的值来选择 scientific 或 fixed，在这个程序的第一个版本中，编译器很好地执行了这个任务。位数的默认值没有在标准中定义，但通常使用 5。

下面看看对该程序的修改。正如本章前面使用 setw()那样，需要添加对 iomanip 的#include 语句和其他 using 语句。除此之外，还有下面 4 行代码：

```
cout<<setprecision(14)<<fixed;      //Change to fixed notation
cout<<value1 - value2<<endl;        //Should output zero

cout<<setprecision(5)<<scientific;  //Set scientific notation
cout<<value1 - value2<<endl;        //Should output zero
```

第一行代码很简单：使用操纵程序，就像使用 setw()那样，用插入运算符把它们发送到输出流中。其结果会清晰地显示在输出的第一行上：浮点数值占据了 14 位，且没有指数。

但是，这些操纵程序不同于 setw()，因为它们是模式化的。换言之，它们的效果要在程序结束后才显示出来。这就是上面使用第三行代码的原因——必须显式设置 scientific 模式，精度设置为 5，才能返回“默认的”结果。但我们成功了，因为输出的第二行与原程序的结果相同。

注意：

实际上，只有 setprecision()操纵程序才需要<iomanip>头文件，fixed 和 scientific 都来自于<iostream>。要讨论的操纵程序还有很多，但规则是需要值的操纵程序(如 setw()和 setprecision())都在<iomanip>中定义，其他则在<iostream>中定义。

2.9.4 数值函数

<cmath>标准库头文件定义了许多可以在程序中使用的三角函数和数值函数，前面已经学习了 sqrt()函数。表 2-13 列出了这个头文件的其他数值函数。

表 2-13　<cmath>的数值函数

函　　数	说　　明
abs(arg)	返回 arg 的绝对值，其类型与 arg 相同，arg 可以是任意浮点类型。在<cstdlib>头文件中还声明了变元是 int 和 long 类型的 abs()函数版本
fabs(arg)	返回 arg 的绝对值，其类型与 arg 相同，变元可以是 int、long、float、double 或 long double
ceil(arg)	返回与 arg 类型相同的一个浮点值，该值是大于或等于 arg 的最小整数，所以 ceil(2.5)返回值 3.0。arg 可以是任意浮点类型
floor(arg)	返回与 arg 类型相同的一个浮点值，该值是小于或等于 arg 的最大整数，所以 floor(2.5)返回值 2.0。arg 可以是任意浮点类型
exp(arg)	返回 e^{arg} 的值，其类型与 arg 相同，arg 可以是任意浮点类型
log(arg)	log 函数返回 arg 的自然对数，其类型与 arg 相同，arg 可以是任意浮点类型
log10(arg)	log10 函数返回 arg 以 10 为底的对数，其类型与 arg 相同，arg 可以是任意浮点类型
Pow(arg1,arg2)	pow 函数返回 arg1 的 arg2 次方，即 $arg1^{arg2}$。因此，pow(2,3)的结果是 8，pow(1.5,3)的结果是 3.375。两个变元的类型都是 int 或任意浮点类型。当第一个变元 arg1 是 int、long 类型或任意浮点类型时，第二个变元 arg2 还可以是 int 类型，返回的值与 arg1 的类型相同

表 2-14 列出了<cmath>头文件中的三角函数。

表 2-14　<cmath>的三角函数

函　　数	说　　明
cos(angle)	返回 angle 的余弦，angle 变元以弧度为单位
sin(angle)	返回 angle 的正弦，angle 变元以弧度为单位
tan(angle)	返回 angle 的正切，angle 变元以弧度为单位
cosh(angle)	返回 angle 的双曲线余弦，angle 变元以弧度为单位。变量 x 的双曲线余弦由公式$(e^x-e^{-x})/2$确定
sinh(angle)	返回 angle 的双曲线正弦，angle 变元以弧度为单位。变量 x 的双曲线正弦由公式$(e^x+e^{-x})/2$确定
tanh(angle)	返回 angle 的双曲线正切，angle 变元以弧度为单位。变量 x 的双曲线余弦是 x 的双曲线正弦除以 x 的双曲线余弦
acos(arg)	返回 arg 的反余弦。变元必须在－1 到＋1 之间。结果以弧度为单位，其范围是 0 到 π
asin(arg)	返回 arg 的反正弦。变元必须在－1 到＋1 之间。结果以弧度为单位，其范围是$-\pi/2$ 到$+\pi/2$
atan(arg)	返回 arg 的反正切。结果以弧度为单位，其范围是$-\pi/2$ 到$+\pi/2$
atan2(arg1,arg2)	这个函数需要两个浮点类型的变元，返回 arg1/arg2 的反正切。结果以弧度为单位，其范围是 0 到 π，其类型与变元相同

这些函数的变元可以是任意浮点类型，返回的结果与变元类型相同。

下面是使用这些函数的例子。下面的语句可以计算出一个角度的正弦：

```
double angle = 1.5;                    //In radians
double sine_value = std::sin(angle);
```

如果角度以度为单位，就可以使用 π 的值把它转换为弧度，再计算正切：

```
float angle_deg = 60.0f;          //Angle in degrees
const float pi = 3.14159f;
const float pi_degrees = 180.0f;
float tangent = std::tan(pi*angle_deg/pi_degrees);
```

如果知道教堂尖塔的高度是 100 英尺，并可以站在距离尖塔底部 50 英尺的地方，就可以计算出尖塔的顶角，单位是弧度，如下所示：

```
double height = 100.0;                        // Steeple height- feet
double distance = 50.0;                       // Distance from base
angle = std::atan2(height, distance); // Result in radians
```

可以使用 angle 中的值和 distance 中的值计算当前位置到尖塔顶部的距离：

```
double toe_to_tip = distance*std::cos(angle);
```

当然，Pythagoras of Samos 的拥护者还可以用更简单的方法获得这个结果，如下所示：

```
double toe_to_tip = std::sqrt(std::pow(distance,2)+std::pow(height,2);
```

2.10 使用字符

char 类型的变量主要用于存储单个字符的编码，占用 1 个字节的内存。C++标准没有指定用于表示基本字符集的字符编码，所以这由编译器指定。但一般使用 ASCII 编码。

char 类型的变量声明与其他类型的变量声明相同，如下所示：

```
char letter;
char yes, no;
```

第一个语句声明了一个 char 类型的变量 letter，第二个语句声明了两个 char 类型的变量 yes 和 no。这些变量都可以存储一个字符的编码。我们没有为这些变量提供初始值，所以它们包含垃圾值。

2.10.1 字符字面量

在声明 char 类型的变量时，可以用字符字面量初始化它。具体方法是把字符字面量编写为需要的字符，放在单引号中。例如，'z'、'3'和'?'都是字符字面量。

一些字符在输入为字面量时会出问题。显然，单引号表示起来就有点困难，因为它是字符字面量的界定符。实际上，在 C++中，在单引号之间放置一个单引号或反斜杠字符是不合法的。控制字符如换行符和制表符也不好表示，因为它们会在按下相应字符健时换行或移动光标，而不会把该字符作为数据输入。使用以反斜杠开头的转义序列，可以指定所有这些问题字符，如

表 2-15 所示。

表 2-15　问题字符的转义序列

字　　符		转 义 序 列
换行符	NL(LF)	\n
水平制表符	HT	\t
垂直制表符	VT	\v
退格	BS	\b
回车	CR	\r
换页	FF	\f
警铃	BEL	A
反斜杠	\	\\
单引号	'	\'
双引号	"	\"
问号	?	\?

要指定对应于这些字符的字符字面量，只需在单引号之间输入相应的转义序列，例如，换行符是'\n'，　反斜杠是'\\'。

还有一些转义序列可以用于指定用八进制或十六进制值表示的字符编码。八进制字符编码的转义序列是一个反斜杠后跟三个八进制数。十六进制字符编码的转义序列是\x 后跟一个或多个十六进制数。在定义字符字面量时，这两种形式都放在单引号中。例如，在 US-ASCII 编码中，字母'A'可以写为十六进制的'\x41'，或八进制的'\81'。显然，可以编写不能放在一个字节中的编码，此时结果是实现方式已定义好的内容。

如果在单引号中编写了包含多个字符的字符字面量，这些字符不表示一个转义序列，例如'abc'，该字面量就描述为多字符字面量，其类型为 int。这种字面量的数字值是实现方式已定义好的内容，但通常是在 int 值的后续字节中放置字符的 1 字节编码。如果指定的多字符字面量多于四个字符，编译器通常会发出一个错误消息。

现在就可以使用字符字面量正确初始化 char 类型的变量了。

2.10.2　初始化 char 变量

下面的语句可以定义并初始化一个 char 类型的变量：

```
char letter='A';        //Stores a single letter 'A'
```

这个语句定义了 char 类型的变量 letter，其初始值是'A'。如果编译器用 US-ASCII 码表示字符，该字符的十进制值就是 65。

可以在一个语句中声明和初始化多个变量：

```
char yes='y', no = 'n', tab = '\t';
```

因为可以把 char 类型的变量看作整型数，所以可以用下面的语句来声明和初始化变量 letter：

```
char letter=65;       //Stores the ASCII code for 'A'    .
```

char 类型在默认情况下可以带符号，也可以不带符号，这取决于编译器，这会影响变量包含的数字值。如果 char 是不带符号的，其数字值将在 0 到 255 之间。如果它是带符号的，其值在 - 128 到 127 之间。当然，可以存储的位模式的范围也是这样。它们的解释是不同的。

当然，可以把 letter 当作整型数来操作，所以可以编写下面的语句：

```
letter += 2;
```

这会使存储在 letter 中的值递增为 67，也就是 US-ASCII 码中的'C'。本书最后的附录 A 列出了所有的 ASCII 码。

注释：

本例中假定使用 US-ASCII 编码，但不一定非如此不可。特别是在许多大型计算机上，字符用 EBCDIC (Extended Binary Coded Decimal Interchange Code，扩充的二进制编码的十进制交换码)表示，这些码与 US-ASCII 编码是不同的。

可以显式地把变量声明为 signed char 或 unsigned char 类型，这两个修饰符会影响变量能表示的整数范围。例如，可以把变量声明为：

```
unsigned char ch =0U;
```

这里，数值的取值范围是 0 到 255。

在从流中读取 char 类型的变量时，将存储第一个非空白的字符。也就是说，不能用这种方式读取空白字符，它们会被忽略。而且，不能把数值读入 char 类型的变量，如果这么做，会在变量中存储第一个数字的字符码。在屏幕上输出 char 类型的变量时，它会显示为一个字符，而不是数值。下面用一个例子来说明。

程序示例 2.9——处理字符值

本例从键盘读取一个字符，输出该字符和对应的数值码，递增该字符的值，把结果输出为一个字符和对应的整数：

```
//Program 2.9 - Handling character values
#include <iostream>
using std::cin;
using std::cout;
using std::endl;

int main() {
  char ch=0;
  int ch_value=0;

  //Read a character from the keyboard
  cout << " Enter a character: ";
```

```
cin >> ch;
ch_value=ch;                    //Get integer value of character

cout << endl
     << ch << " is " << ch_value;

ch_value=++ch;                  //Increment ch and store as integer
cout << endl
     << ch<< " is " << ch_value
     << endl;

  return 0;
}
```

这个例子的结果如下所示：

```
Enter a character: w

w is 119
x is 120
```

例子的说明

在提示输入后，程序利用下面的语句从键盘上读取一个字符：

```
cin >> ch;
```

因为 cin 只接受非空白的字符，所以可以按下回车键，或输入空格和制表符，它们会被忽略。

输出流总是把变量 ch 输出为一个字符。为了获得该字符对应的数值，需要把它转换为整数类型，如下面的语句所示：

```
ch_value = ch;      //Get integer value of character
```

编译器会把存储在 ch 中的值从 char 类型转换为 int 类型，使之能存储在变量 ch_value 中。第 3 章在讨论涉及不同类型的值的表达式时，将详细描述自动转换。

现在用下面的语句输出字符及其对应的整型数值：

```
cout << endl
     << ch << " is " << ch_value;
```

接着，把 char 类型的变量用作整型数：

```
ch_value = ++ch;      //Increment ch and store as integer
```

这个语句递增 ch 的值，并把结果存储在变量 ch_value 中。这样就得到了下一个字符及其对应的数值表示。使用与前面相同的语句把这些结果显示出来。这里只递增了 ch，而 char 类型的变量可以像整数类型一样，和所有的算术运算符一起使用。

2.10.3 使用扩展字符集

通常，单字节字符如 ASCII 或 EBCDIC 对使用拉丁字符的国家语言字符集来说足够了。还有 8 位字符编码包容其他语言，如希腊语和俄语。但是，如果要同时使用这些语言和拉丁字

符，或者要处理亚洲语言的字符集，它们需要的字符编码比 ASCII 字符集多得多，256 个字符编码就远远不够了。

类型 wchar_t 是一种字符类型，它可以存储实现方式支持的最大扩展字符集中的所有成员。这个类型名来自于宽字符(wide characters)，因为字符的范围比通常的单字节字符宽。Char 类型则“比较窄”，因为可用的字符编码比较有限。C++标准没有设定 wchar_t 类型的变量的长度，只是它具有其他整数类型的特性。在 PC 上，它通常是 2 字节，其底层类型是 unsigned short，在一些编译器中也可以是 4 字节，特别是在 Unix 工作站的实现方式中常常是 4 字节。

1. 宽字符字面量

定义宽字符字面量的方式与前面处理 char 类型的窄字符字面量相同，但要在字面量的前面加上前缀字母 L，例如：

```
wchar_t wide_letter=L'Z';
```

此语句把变量 wide_letter 定义为 wchar_t 类型，并将其初始化为 Z 的宽字符表示。

键盘上可能没有表示其他国家语言字符的键，但仍可以使用十六进制表示法来创建它们。例如：

```
wchar_t wide_letter=L'\x0438';      //Cyrillic H
```

单引号中的值是一个转义序列，它可以利用字符代码的十六进制表示指定一个字符。反斜杠表示转义序列的开始，反斜杠之后的 x 表示该代码是十六进制的。没有 x 或 X，就表示其后的字符应解释为八进制数字。

当然，也可以使用 UCS 字符字面量的记号法：

```
wchar_t wide_letter=L'\u0438';      //Cyrillic H
```

如果编译器支持 4 字节的 UCS 字符，也可以用指定为\Udddddddd(其中 d 是一个十六进制数字)的 UCS 字符来初始化 wchar_t 类型的变量。

2. 宽字符流

前面使用的流 cin 和 cout 都是窄字符流，它们只能处理由一个字节组成的字符，所以不能从 cin 中读取 wchar_t 类型的变量。<iostream>头文件定义了特殊的宽字符流 wcin 和 wcout，用于宽字符的输入和输出。使用宽字符流的方式与窄字符流相同。例如，可以从 wcin 中读取一个宽字符，如下所示：

```
wchar_t wide_letter=0;
std::wcin>> wide_letter;  //Read a wide character
```

可以把宽字符输出到 wcout 中，但这并不表示这种字符会正确显示。这取决于操作系统是否能识别该字符编码。

2.11 初始值的函数表示法

在声明变量时，为它指定初始值还有另一个方法，称为函数表示法。使用这个名称，是因为初始值放在变量名后面的括号中，看起来像是函数调用。

下面看一个例子，本例不是把声明写成如下形式：

```
int unlucky=13;
```

而是写成：

```
int unlucky(13);
```

这两个语句的效果是一样的，都是把变量 unlucky 声明为 int 类型，并指定初始值为 13。

可以用函数表示法初始化其他类型的变量。例如，用下面的语句可以声明并初始化一个存储字符的变量：

```
char letter('A');
```

但是，初始化变量的函数表示法主要用于用户定义的数据类型的变量初始化。在这种情况下，要涉及到调用函数。在 C++中，基本类型的变量初始化通常使用前面所述的方法。第 11 章将介绍如何创建自己的类型，以及如何初始化这些类型的变量。

2.12 本章小结

本章介绍了 C++中计算的基础知识，学习了该语言提供的大多数基本数据类型，本章的主要内容如下：

- 数值和字符常量称为字面量。
- 可以把整数字面量定义为十进制、十六进制或八进制。
- 浮点字面量必须包含小数点或指数，或两者都包含。
- C++中的已命名对象，例如变量，其名称可以包含一组字母和数字，但第一个字符必须是字母，下划线也看作是字母。大小写字母是不同的。
- 由于以下划线开头后跟一个大写字母的名称，以及包含两个连续下划线的名称，是标准库中使用的保留名称，因此它们不应用作变量名。
- C++中的所有字面量和变量都有给定的类型。
- 可以存储整数的基本类型有 short、int 和 long。它们在默认情况下存储带符号的整数，也可以在这些类型名称的前面使用类型修饰符 unsigned，使该类型占用相同的字节数，但只存储不带符号的整数。
- char 类型的变量可以存储单个字符，占用 1 个字节。char 类型在默认情况下可以是带符号的，也可以是不带符号的，这取决于编译器。也可以使用 signed char 和 unsigned char 类型的变量存储整数。
- 类型 wchar_t 可以存储宽字符，占用 2 或 4 个字节，这取决于编译器。
- 浮点数的数据类型有 float、double 和 long double。

- 变量的名称和类型出现在声明语句中，以一个分号结束。声明一个变量，如果给该变量分配了内存空间，那么也就定义了该变量。
- 变量在声明时可以指定初始值，这是一种很好的编程习惯。
- 可以用修饰符 const 保护基本类型的“变量”值。编译器会在程序源代码文件中检查是否试图修改声明为 const 的变量。
- lvalue 是出现在等号左边的一个对象或表达式，非 const 的变量就是 lvalue。

本章讨论了许多基本类型，但没有囊括所有的类型。还有其他一些基本类型，以及建立在这些基本类型之上的复杂类型，甚至还可以创建自己的类型。

2.13 练习

1. 编写一个程序，计算圆的面积。该程序应提示输入圆的半径，使用公式 area=pi*radius*radius 计算面积，再显示结果。

2. 使用第 1 题的解决方案，改进代码，使用户可以输入所需的位数，控制输出结果的精度(提示：使用 setprecision()操纵程序)。

3. 创建一个程序，把英寸转换为英尺和英寸。例如，输入 77 英寸，程序就应生成 6 英尺 5 英寸的结果。提示用户输入一个单位是英寸的整数值，再进行转换，输出结果(提示：使用 const 存储 inches-to-feet 转换率，并使用取模运算符)。

4. 在生日那天，您得到了一个卷尺和一个可以确定角度的仪器，例如测量水平线和树高之间的夹角。如果知道自己与树之间的距离 d 和眼睛平视量角器的高度 h，就可以用下面的公式计算出树的高度：

```
h+d*tan(angle)
```

创建一个程序，从键盘上输入 h(单位是英寸)、d(单位是英尺和英寸)和 angle(单位是度)，输出树的高度(单位是英寸)。

注意：

要验证程序的正确性不需要砍断树。只需查看 Apress 网站上的解决方法即可(http://www.apress.com/book/download.html)。

5. 这个题较难。编写一个程序，提示用户输入两个不同的正整数，在输出中指出哪个较大，哪个较小(这可以用本章学习的内容来完成)。

第 3 章　处理基本数据类型

本章将扩展第 2 章中讨论的类型，并学习基本类型的变量如何在复杂的环境下交互。此外，还要论述 C++的一些新功能，讨论使用这些功能的一些方式。

本章主要内容

- 如何计算涉及混合数据类型的表达式
- 如何把数值从一种基本类型转换为另一种基本类型
- 按位运算符的概念以及用法
- 如何定义新类型，把变量的值限定为一组固定数量的可能值
- 如何为已有的数据类型定义替代名称
- 变量的存储期限及其决定因素
- 变量作用域的概念及作用

3.1　混合的表达式

计算机只能对相同类型的值进行算术操作。它可以把两个整型数相加，把两个浮点数相加，但不能把一个整型数和一个浮点数直接相加。例如，表达式 2 + 7.5 就不能直接计算，因为 2 是一个整数，而 7.5 是一个浮点数。

要执行这种计算，惟一的方法是把一个值转换为与另一个值相同的类型。一般情况下，是把整数值转换为其相应的浮点数类型，这样，上面的表达式就等价于 2.0 + 7.5。这个规则适用于 C++中的混合表达式。每个二元算术运算都要求两个操作数的类型相同。如果它们的类型不同，就必须把其中一个操作数转换为另一个操作数的类型。考虑下面的语句块：

```
int value1=10;
long value2=25L;
float value3=30.0f;
double result= value1+ value2+ value3;      // Mixed calculation
```

result 的值应计算为不同类型的变量总和。对于每个相加操作，在进行相加之前，必须把一个操作数的类型转换为另一个操作数的类型，对操作数所进行的转换取决于一组规则，该组规则将按顺序检查表达式，直到操作可以进行为止。上面语句的执行步骤如下：

- 计算 value1+value2 时，先把 value1 转换为 long 类型，再计算其和。结果也是 long 类型，这样计算过程是 10L+25L=35L。
- 下一个操作就计算 35L+ value3。把前面的结果 35L 转换为 float 类型，再加到 value3 上。结果是 float 类型，所以操作过程是 35.0f+30.0f=65.0f。
- 最后，把前面的结果转换为 double 类型，存储在 result 中。

只有当二元运算符的操作数的类型不同时，才能应用处理混合表达式的规则。这些规则按照应用的顺序，如下所述：

(1) 如果一个操作数的类型是 long double，则另一个操作数就转换为 long double 类型。

(2) 如果一个操作数的类型是 double，则另一个操作数就转换为 double 类型。

(3) 如果一个操作数的类型是 float，则另一个操作数就转换为 float 类型。

(4) 只要 int 类型可以表示原操作数类型的所有值，类型为 char、signed char、unsigned char、short 或 unsigned short 的操作数就转换为 int 类型。否则，操作数就转换为 unsigned int 类型。

(5) 枚举类型转换为可包容该枚举范围的 int、unsigned int、long 或 unsigned long 中的第一个类型。

(6) 如果一个操作数的类型是 unsigned long，则另一个操作数就转换为 unsigned long 类型。

(7) 如果一个操作数的类型是 long，另一个操作数的类型是 unsigned int，且类型 long 可表示 unsigned int 类型中的所有值，就把 unsigned int 类型的操作数转换为 long 类型。否则，两个操作数都转换为 unsigned long 类型。

(8) 如果一个操作数的类型是 long，另一个操作数就转换为 long 类型。

前面没有介绍枚举类型，本章稍后将讨论它。这里提到这种类型，是为了列出完整的规则集。这些规则看起来相当复杂，实际上并非如此。其中一些规则表面看起来比较复杂，是因为整数类型的取值范围是彼此相关的，因而规则需要容纳它们。编译器将对代码按顺序检查这些规则，直到找出可应用的规则为止。如果在应用该规则后，操作数的类型相同，就执行操作。否则就应用另一个规则。

基本理念是非常简单的。如果两个操作数的类型不同，就把取值范围较窄的那个值的类型转换为另一个值的类型。正式的规则如下：

(1) 如果操作涉及到两个不同的浮点数类型，就把精度较低的浮点数提升为另一个浮点数的类型。

(2) 如果操作涉及到一个整数和一个浮点数，就把整数提升为浮点数类型。

(3) 如果操作涉及到混合的整数类型，就把取值范围较小的整数提升为另一个整数类型。

(4) 如果操作涉及到枚举类型，就把它们转换为合适的整数类型。

术语“转换”(coversion)表示从一种类型到另一种类型的自动转换。术语“提升”(promotion)一般表示把一个数据值从一种取值范围较窄的类型转换为另一种取值范围较宽的类型。稍后将介绍如何把一种数据类型显式转换为另一种数据类型。这种转换称为“强制转换”(cast)，把一个值显式转换为另一种类型的过程称为“强制转换过程”(casting)。

C++支持涉及混合类型的表达式，这并不是说使用混合类型的表达式很好，因为结果常常不是我们希望的，特别是在混合带符号和不带符号的类型时，结果更是出人意料。所以应尽可能避免编写混合表达式。

3.1.1 赋值和不同的类型

如果赋值运算符右边的表达式的类型不同于赋值运算符左边的变量类型，计算右边表达式所得的结果就会自动转换为左边变量的类型，之后才存储在该变量中。在许多情况下，这样会丢失信息。例如，假定定义了一个浮点数值：

```
double root=1.732;
```

若要编写下面的语句：

```
int value=root;
```

则把 root 的值转换为 int，就会在 value 中存储 1。因为 int 类型的变量只能存储整数，所以存储在 root 中的小数部分就会在转换为 int 类型的过程中舍弃。甚至在不同类型的整数之间使用等号，也会丢失信息：

```
long count=60000;
short value=count;
```

如果 short 是两个字节，long 是 4 个字节，前者就没有足够的空间存储 count 的值，因而就会产生不正确的值。

许多编译器都会检测出这些转换，并发出警告消息，但我们不应依赖这些警告消息。为了尽可能避免这类问题，应避免把一种类型的数值赋予另一种取值范围较窄的类型。如果无法避免这类赋值，则可以指定进行显式转换，为的是说明这不是偶然事件，而是确实要这么做。下面看看其工作过程。

3.1.2 显式强制转换

混合的算术表达式涉及到基本类型，编译器会在需要时自动转换操作数，也可以使用显式转换方式，强制从一种类型转换为另一种类型。要把表达式的值强制转换为给定的类型，应编写如下格式的转换语句：

```
static_cast<转换后的类型>(表达式)
```

关键字 static_cast 表示这个强制转换要进行静态检查，也就是说，在程序编译时进行检查。后面在介绍类的处理时，会遇到动态的强制转换，这种转换要进行动态检查，即在程序执行时进行检查。强制转换的结果是把从表达式中计算的值转换为尖括号中指定的类型。表达式可以是任何内容，包括从单个变量到包含许多嵌套括号的复杂表达式等所有内容。

下面是使用 static_cast<>()的一个例子：

```
double value1=10.5;
double value2=15.5;
int whole_number= static_cast<int>( value1)+ static_cast<int>( value2);
```

因为变量 whole_number 的初始值是 value1 和 value2 的整数部分之和，所以它们必须分别显式强制转换为 int 类型。变量 whole_number 的初始值应为 25。强制转换不会影响存储在 value1 和 value2 中的值，它们仍然是 10.5 和 15.5。由强制转换得到的值 10 和 15 只是临时存储，在计算中使用后就删除。这两个强制转换会在计算过程中丢失信息，但编译器总是假定在显式指定强制转换时，用户知道会发生什么。

在前面的例子中，根据赋予不同的类型，可以把强制转换设置为显式执行，清晰地说明该强制转换是必须的：

```
int value= static_cast<int>( root);
```

一般情况下，很少需要显式强制转换，特别是在数据为基本类型时。如果必须在代码中包含大量的显式强制转换，则通常表明应为变量选择更合适的类型。但仍有一些情况需要进行强制转换。下面就介绍一个这类情况的例子。

程序示例 3.1——显式强制转换

假定需要把单位为码的长度(小数值)转换为码、英尺和英寸(整数值)，可以编写如下的程序：

```
//Program 3.1 Using Explicit Casts
#include <iostream>
using std::cin;
using std::cout;
using std::endl;

int main() {
  const long feet_per_yard=3;
  const long inches_per_foot=12;

  double yards=0.0;     // Length as decimal yards
  long yds=0;           // Whole yards
  long ft=0;            // Whole feet
  long ins=0;           // Whole inches

  cout << "Enter a length in yards as a decimal: ";
  cin >> yards;

  //Get the length as yards, feet and inches
  yds = static_cast<long>( yards);
  ft = static_cast<long>((yards-yds)*feet_per_yard);
  ins = static_cast<long>
  (yards*feet_per_yard*inches_per_foot)% inches_per_ foot;

  cout << endl
       << yards << "yards converts to "
       <<yds    << "yards "
       <<ft     << "feet "
       <<ins      << "inches.";

  cout << endl;
  retuen 0;
}
```

这个程序的输出如下：

```
Enter a length in yards as a decimal: 2.75

2.75 yards coverts to 2 yards 2 feet 3 inches.
```

例子的说明

main()中的前两个语句声明了后面要使用的两个转换常量：

```
const long feet_per_yard=3;
const long inches_per_foot=12;
```

把这些变量声明为 const，以防止程序不经意地修改它们，使用类型 long 是为了同其他值保持一致。虽然这里使用类型 short 就足以存储这些值了，但从长远来看，使用 short 会增加程序文件的长度(而不是减小)。这是因为在表达式中使用 short 和其他整型值时，需要进行隐式的转换。

接着的 4 个声明定义了要在计算过程中使用的变量：

```
double yards=0.0;      // Length as decimal yards
long yds=0;            // Whole yards
long ft=0;             // Whole feet
long ins=0;            // Whole inches
```

利用下面的语句，提示用户输入数据，再从键盘中读取一个值：

```
cout << "Enter a length in yards as a decimal: ";
cin >> yards;
```

下一个语句先对输入的值进行显式的强制转换，再计算 yards 的整型值：

```
yds= static_cast<long>(yards);
```

强制转换为 long 类型时，会舍弃 yards 值的小数部分，把整数部分存储在 yds 中。如果这里省略了显式强制转换过程，一些编译器就会编译该程序，而不是发出警告，说明已插入了需要的强制转换。但是，显然在这个转换过程中存在潜在的信息丢失，在这种情况下，应总是编写一个显式强制转换语句，以表明需要这么做。如果忽略了这一步，就不清楚这个转换的必要性如何，也没有注意到潜在的数据丢失。

用下面的语句获得长度的英尺值：

```
ft= static_cast<long>((yards-yds)*feet_per_yard);
```

因为我们希望英尺数不包含在码数中，所以从 yards 中减去 yds 的值。编译器会把 yds 中的值自动转换为 double 类型，进行减法运算，其结果也是 double 类型。再把 feet_per_yard 的值自动转换为 double 类型，进行乘法运算，最后对乘积进行显式的强制转换，把它从 double 类型转换为 long 类型。

最后一部分计算是获得剩余长度的英寸值：

```
ins= static_cast<long>
(yards*feet_per_yard*inches_per_foot)% inches _ per _ foot;
```

计算原长度的总英寸值，再利用显式的强制转换，将它转换为 long 类型，接着除以每英尺的英寸数，得到了剩余的英寸值。最后，用下面的语句输出结果：

```
cout << std::endl
     << yards << "yards converts to "
     << yds   << "yards "
     << ft    << "feet "
     << ins   << "inches.";
```

3.1.3 老式的强制转换

前面介绍了 C++中的 static_cast<>()(以及本书后面要介绍的 const_cast<>()、dynamic _ cast<>()和 reinterpret_cast<>())，把表达式的结果显式强制转换为另一种类型的过程可以表示为：

(转换后的类型) 表达式

表达式的结果强制转换为括号中的类型。例如，前面例子中计算 ins 的语句可以改写为：

```
ins= (long)(yards*feet_per_yard*inches_per_foot)% inches_per_foot;
```

基本上，有 4 种不同类型的强制转换，老式的强制转换语法包含了这 4 种转换。所以，使用老式强制转换的代码更容易出错——它并不是很清晰，可能得不到希望的结果。尽管老式强制转换语法目前仍使用得很广泛(它仍是语言的一部分)，但最好在代码中使用新型的强制转换语法。

伪随机数的生成

了解了强制转换后，就可以确保在算术表达式中使用 rand()返回的值时不出问题。上一章提到，rand()返回 0 到 RAND_MAX 之间的值，RAND_MAX 可以定义为 int 取值范围内任何正的 int 值。假定类型 long 的取值范围比 int 大，在用随机整数执行算术操作时，把 rand()返回的值强制转换为 long 类型，就可以避免可能的问题，例如：

```
long even = 2*static_cast<long>(std::rand());
```

rand()返回的值是 long 类型，乘法运算就会在值 2 之转换相同类型之后进行。因此，乘法运算的结果就总是在 long 类型的取值范围之内。把字面量定义为 long 类型，可以得到相同的效果：

```
long even = 2L* std::rand();
```

2L 是 long 类型，所以编译器会把 rand()返回的值强制转换为 long 类型，再执行乘法运算。

可以使用 rand()函数获得取值范围比 0 到 RAND_MAX 更小的随机整数，例如，假定希望随机整数在 0 到 10 之间，就可以从 rand()返回的值中得到如下结果：

```
const int limit = 11;
int random_value = static_cast<int>(
                        (limit*static_cast<long>(std::rand()))
                        /(RAND_MAX+1L));
```

这里把 0 到 RAND_MAX 的取值范围分成 limit 部分，在给定的部分中，rand()返回的所有值都在 0 到 limit－1 的范围内。具体方法是，把 limit 乘以 rand()/(RAND_MAX+1L)，除以(RAND_MAX+1L)而不是 RAND_MAX，是为了处理 rand()正好返回 RAND_MAX 的情况。如果除以 RAND_MAX，结果就是 limit，而不是 limit－1。加到 RAND_MAX 上的常量 1L 是 long 类型，所以 RAND_MAX 也会在执行加法操作之前转换为 long 类型。前面说过，在 rand()的某些实现方式中，RAND_MAX 定义为 int 类型的最大整数。在这种情况下，不先把 RAND_MAX 转换为 long 类型，就不能给它加 1，也得不到正确的结果。

如果希望随机值在 1 和某个上限之间，而不是下限为 0，就应使用下面的代码：

```
const int limit = 100;
int random_value = static_cast<int>
                        (1L +(limit*static_cast<long>(std::rand()))/
                        (RAND_MAX+1L));
```

这里使用与前面相同的表达式生成 0 到 limit－1 之间的值，然后给它加 1，得到 1 到 limit 之间的值。

开头说过，这些都建立在类型 long 的取值范围比 int 大的假定基础之上。如果不是这样，就需要生成 int 类型取值范围之外的随机值，此时只能把 rand()返回的值强制转换为浮点数，并把计算的结果存储为浮点数。例如，要产生 0 到 limit 之间的随机值，可以使用下面的语句：

```
const double limit = 11;
double random_value = limit*std::rand()/(RAND_MAX+1.0);
```

这里把 limit 声明为 double 类型，编译器会把 rand()返回的整数提升为 double 类型，因此，不需要插入显式的强制转换。

3.2 确定类型

前面多次提及，由于一些类型使用的字节数在 C++标准中没有指定，因此由编译器指定。那么就有可能需要知道某个变量类型在编译器中占据多少字节。这些信息可以从编译器的说明文档中找到，也可以使用 sizeof 运算符通过编程来获得。

sizeof 是一个一元运算符，带一个操作数。它返回一个整数值，表示某个变量或类型占用的内存量。sizeof 返回的值实际上计量了多个 char 类型的字节数，但因为 char 类型的变量占用一个字节，所以返回的值是操作数占用的字节数。

要获得 type 类型的变量占用的字节数，可以使用表达式 sizeof(type)。所以，下面的语句可以输出 int 类型的变量占用的字节数：

```
std::cout << std::endl
          << "Size of type int is "
          << sizeof(int);          //Output the size of type int
```

表达式 sizeof(int)返回声明为 int 类型的实体占用的字节数。可以用这种方式确定任何数据类型占用的字节数。要获得 long double 值占用的字节数，可以编写下面的代码：

```
std::cout << std::endl
          << "Size of type long double is "
          <<sizeof(long double);          //Output the size of type long double
```

还可以把 sizeof 运算符应用于某个变量，甚至应用于表达式。在这种情况下，表达式不必放在括号中，当然也可以加上括号。下面的例子会输出变量 number 占用的字节数：

```
long number=999999999;
std::cout << std::endl
          << "Size of the variable number is "
          << sizeof number;          //Output the size of a variable
```

sizeof 运算符返回的值是一个整数，但实际上它是 size_t 类型。这不是什么新类型，其名称 size_t 在标准头文件中定义为一种基本整数类型的同义词，通常是 unsigned int 类型。因为这是一个同义词，而不是标准库中一个实体的名称，所以也可以使用它。那么，为什么要使用另一个不同的类型名，为什么不叫作 unsigned int 类型？

指定由 sizeof 运算符返回的类型的原因是该类型的建立非常灵活。sizeof 运算符总是返回一个 size_t 类型的值，编译器会把它定义为 unsigned int 类型，但这是不必要的。如果这可能出

于为某个硬件平台开发 C++编译器的人员的方便，以便把 size_t 类型定义为与其他整数类型等价。这不会影响代码，因为假定其类型是 size_t。本章的后面将介绍如何为已有的类型定义另一个同义词。size_t 中的 t 表示 type，所以选择这个名称表示类型占用的字节数。

程序示例 3.2——确定数据类型占用的字节数

很容易编写一个程序，列出前面介绍的所有数据类型占用的字节数：

```
//Program 3.2 Finding the sizes of data types
#include <iostream>
using std::cout;
using std::endl;

int main() {
  //Output the sizes for integer types
  cout << endl
       << "Size of type char is"
       << sizeof(char);
  cout << endl
       << "Size of type short is "
       << sizeof(short);
  cout << endl
       << "Size of type int is "
       << sizeof(int);
  cout << endl
       << "Size of type long is "
       << sizeof(long);

  //Output the sizes for floating-point types
  cout << endl
       << "Size of type float is "
       << sizeof(float);
  cout << endl
       << "Size of type double is "
       << sizeof(double);
  cout << endl
       << "Size of type long double is "
       << sizeof(long double);
  cout << endl;
  return 0;
}
```

该程序的输出如下：

```
Size of type char is 1
Size of type short is 2
Size of type int is 4
Size of type long is 4
Size of type float is 4
Size of type double is 8
Size of type long double is 8
```

可以修改这个程序，使用 sizeof 运算符获得示例变量和表达式占用的字节数。

确定数值的上下限

有时希望了解某一类型除了其占用的字节数之外的其他信息。例如希望了解该类型能包含的数值的上下限。标准头文件<limits>包含所有标准数据类型的上下限信息。该信息是通过每种类型的类来提供的，因为现在还没有讨论到类，所以其工作方式目前就不是很明显了。但是，这里将介绍如何使用头文件<limits>提供的功能获得该信息，至于具体的说明将在第 13 章讨论类时描述。

下面举一个例子。要显示可以存储在 double 类型的变量中的最大值，可以使用下面的语句：

```
std::cout << std::endl
          << "Maximum value of type double is "
          << std::numeric_limits<double>::max();
```

表达式 std::numeric_limits<double>::max()输出了我们希望的值。把不同的类型名称放在尖括号中，就可以得到其他数据类型的最大值，还可以用 min()代替 max()来获得最小值，但整数和浮点数类型的最小值的含义是不同的。对于整数类型，min()会得到真正的最小值，即带符号的整数类型的负数。对于浮点数类型，min()会返回可以存储的最小正数。

可以检索出各种类型的许多其他信息。例如，下面的表达式就返回二进制数字的位数：

```
std::numeric_limits<类型名>::digits
```

只需把自己感兴趣的类型名插入到尖括号中即可。对于浮点数类型，就会获得尾数中二进制数字的位数；对于带符号的整数类型，就可以获得除符号位之外的二进制数字的位数。

下面编写一个小程序来显示数值数据类型的最大值和最小值。

程序示例 3.3——确定最大值和最小值

下面是程序代码：

```
//Program 3.3 Finding maximum and minimum values for data types
#include <limits>
#include <iostream>
using std::cout;
using std::endl;
using std::numeric_limits;

int main() {
  cout << endl
       << "The range for type short is from "
       << numeric_limits<short>::min()
       << " to "
       << numeric_limits<short>::max();
  cout << endl
       << "The range for type int is from "
       << numeric_limits<int>::min()
       << " to "
       << numeric_limits<int>::max();
```

```
    cout << endl
         << "The range for type long is from "
         << numeric_limits<long>::min()
         << " to "
         << numeric_limits<long>::max();
    cout << endl
         << "The range for type float is from "
         << numeric_limits<float>::min()
         << " to "
         << numeric_limits<float>::max();
    cout << endl
         << "The range for type double is from "
         << numeric_limits<double>::min()
         << " to "
         << numeric_limits<double>::max();
    cout << endl
         << "The range for type long double is from "
         << numeric_limits<long double>::min()
         << " to "
         << numeric_limits <long double>::max();
    cout << endl;
    return 0;
}
```

该程序的执行结果如下：

```
The range for type short is from -32768 to 32767
The range for type int is from -2147483648 to 2147483647
The range for type long is from -2147483648 to 2147483647
The range for type float is from 1.17549e-038 to 3.40282e+038
The range for type double is from 2.22507e-308 to 1.79769e+308
The range for type long double is from 2.22507e-308 to 1.79769e+308
```

例子的说明

这是上一节内容的直接应用。使用本例的方法得到的值有一个类型，所以在使用它们时编译器会对它们进行类型检查。

刚才说过，数值数据类型的取值范围也可以用继承 C 语言的头文件中的宏来定义。<cfloat>头文件定义了浮点数上下限的符号，<climits>头文件定义了整数类型上下限的符号。例如，在<climits>头文件中，符号 INT_MAX 表示 int 类型的数值的最大值，而 SCHAR_MIN 表示 signed char 值的最小值。其他类型还有 type_MAX 和 type_MIN 符号。但是，这些都只是编译器在代码中用对应的数字替换字面量的符号。因此，在运行期间访问类型的上下限时，最好使用 runtime_limits<>机制。

3.3 按位运算符

顾名思义，按位运算符允许按照位来操作整型变量。可以把按位运算符应用于任意 signed 和 unsigned 整型，包括 char 类型。但是，它们通常应用于不带符号的整型。

这些运算符的一个常见应用是在整型变量中使用单个的位存储信息。例如标记，它用于描述二进制状态指示符。可以使用一个位来描述有两个状态的值：开或关、男或女，真或假。

也可以使用按位运算符处理存储在一个变量中的几个信息项。例如，颜色值常常记录为三个八位值，分别存储颜色中红、绿和蓝的强度。这些常常保存到四字节变量中的三个字节。第四个字节也不会浪费，包含表示颜色透明度的值。显然，要处理各个颜色成分，需要从变量中分离出各个字节，按位运算符就可以做到这一点。

再看另外一个例子，假定需要记录字体的信息，那么，只要存储每种字体的样式和字号，以及字体是黑体还是斜体，就可以把这些信息都存储在一个二字节的整型变量中，如图 3-1 所示。

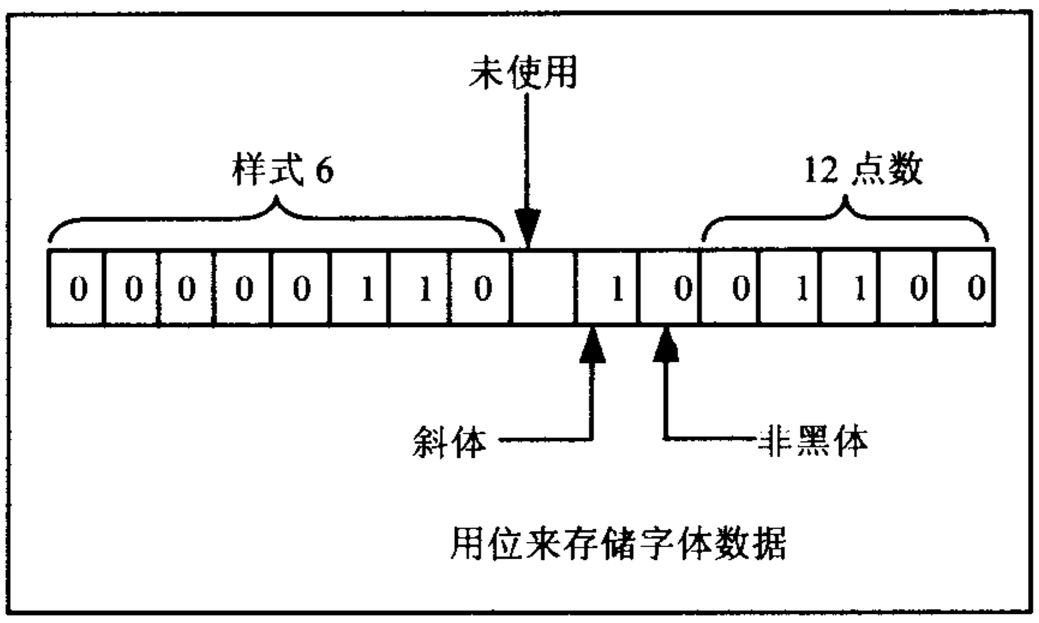


图 3-1　把字体数据存储在 2 个字节中

可以使用一位来记录字体是否为斜体——1 表示斜体，0 表示一般。同样，用另一位来指定字体是否为黑体。使用一个字节可以从多达 256 种不同的样式中选择一个，再用另外 5 位记录最多 32 磅的字号。因此，在一个 16 位的字中，可以记录四个不同的数据项。按位运算符提供了访问和修改整数中单个位和一组位的便利方式，能方便地组合和分解一个 16 位的字。

3.3.1　移位运算符

移位运算符可以把整型变量中的内容向左或向右移动指定的位数。移位运算符和其他按位运算符一起使用，可以获得前面描述的结果。>>运算符把位向右移动，<<运算符把位向左移动，移出变量两端的位被舍弃。

所有的按位操作都可以处理任何类型的整数，但本章的例子使用 16 位的变量，使例子较为简单。用下面的语句声明并初始化一个变量 number：

```
unsigned short number=16387U;
```

如第 2 章所述，不带符号的字面量应在数字的最后添加字母 U 或 u。

在下面的语句中，对这个变量的内容进行移位，并存储结果：

```
unsigned short result = number <<2;      //Shift left two bit positions
```

移位运算符的左操作数是要移位的值，右操作数指定要移动的位数。图 3-2 列出了该操作的过程。

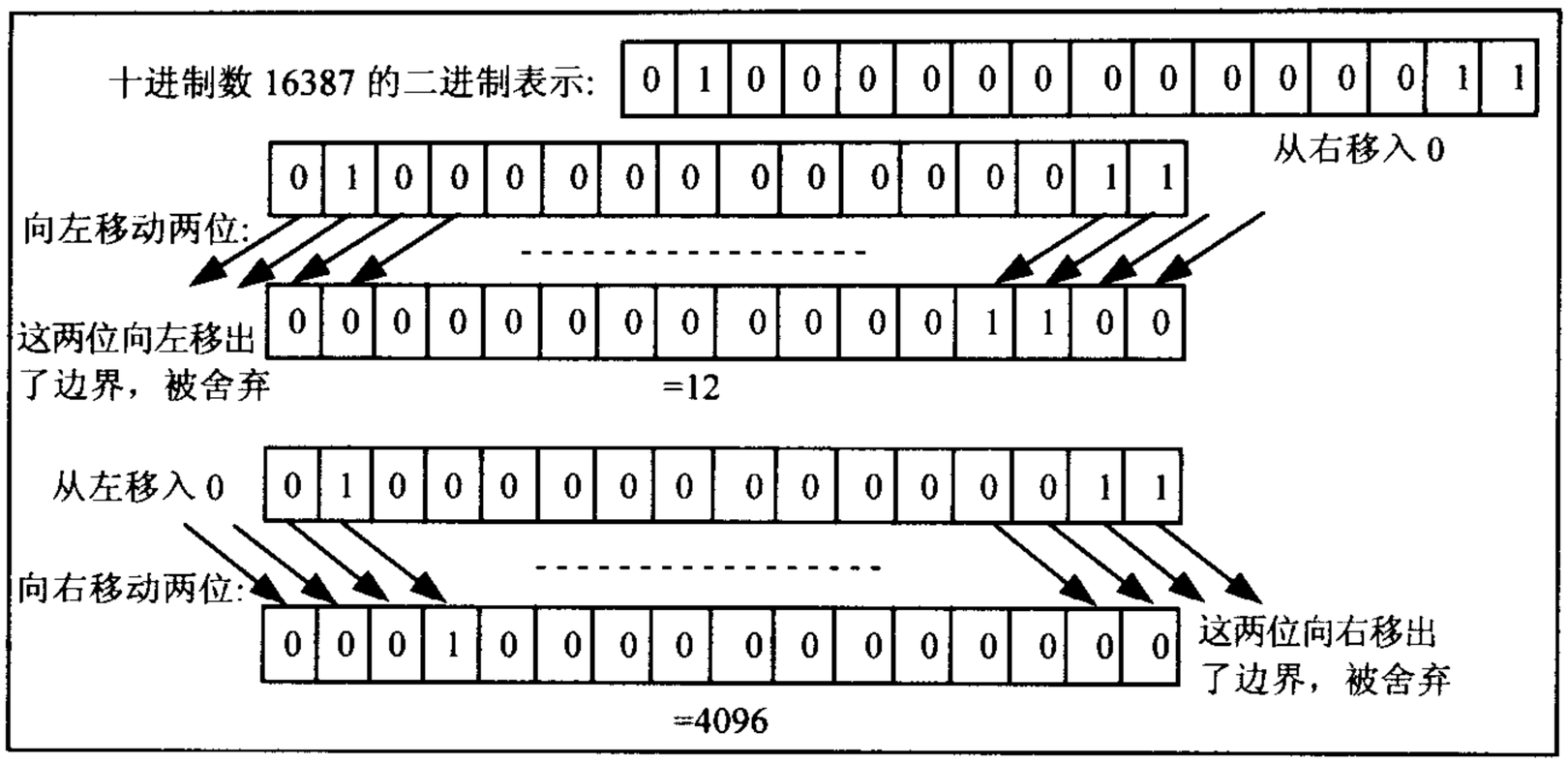


图 3-2　移位运算

从图 3-2 可以看出，把数值 16387 向左移动两位，得到数值 12。数值的这种剧烈变化是舍弃高位数字的结果。

把数值向右移动，可以使用下面的语句：

```
result = number >>2;      //Shift right two bit positions
```

把数值 16387 向右移动两位，得到数值 4096。向右移动两位相当于使该数值除以 4。

只要没有舍弃位，向左移动 n 位就相当于把该数值乘以 2 的 n 次方。换言之，就等于把该数值乘以 2^n。同样，向右移动 n 位就相当于把该数值除以 2 的 n 次方。但要注意，变量 number 向左移位时，如果舍弃了重要的位，结果就不是我们希望的那样了。可是，这与乘法运算并没有不同。如果把 2 字节的数值乘以 4，就会得到相同的结果，所以向左移位和相乘仍是等价的。出现问题的原因是相乘的结果超出了 2 字节整数的取值范围。

如果需要修改原来的值，可以使用 op= 赋值运算符。在这种情况下，可以使用>>=或<<=运算符。例如：

```
number >>= 2;       // Shift contents of number two positions to the right
```

这等价于：

```
number =number >> 2;       // Shift contents of number two positions to the right
```

这些移位运算符跟前面用于输入输出的插入和提取运算符有可能搞混。从编译器的角度来看，其含义一般可以从上下文中判断出来。否则，编译器就会生成一个消息，但用户需要非常小心。例如，如果输出变量 number 向左移动两位的结果，就应编写下面的代码：

```
cout << (number << 2);
```

其中，括号是必不可少的。没有括号，编译器就会把移位运算符解释为流插入运算符，从而得不到想要的结果。

按位移动带符号的整数

可以把移位运算符应用于带符号和不带符号的整型数。但是，向右移位运算符在带符号整

数类型的操作随系统的不同而不同，这取决于编译器的实现。在一些情况下，向右移位运算符会在左边空出来的位上填充 0。在其他情况下，符号位向右移动，在左边空出来的位上填充 1。

移动符号位的原因是为了保持向右移位和除法运算的一致性。可以用 char 类型的变量来说明这一点，解释其工作原理。假定把 value 定义为 char 类型，其初始值为 - 104(十进制)：

```
signed char value=-104;
```

其二进制表示为 10011000。使用下面的操作把它向右移动两位：

```
value >>= 2;        //Result 11100110
```

注释中显示了其二进制结果。右边溢出了两个 0，因为符号位是 1，就在左边空出来的位上填充 1。该结果的十进制表示是 - 26，这正好是 value 的值除以 4 的结果。当然，对于不带符号的整数类型的操作，符号位不移动，在左边空出来的位上填充 0。

前面说过，在向右移位负整数时，其操作是已定义好的，所以实现该操作时不一定要依赖它。因为在大多数情况下，使用这些运算符是为了进行按位操作，此时维护位模式的完整性是非常重要的。所以，应总是使用不带符号的整数，以确保避免高阶位的移位。

3.3.2 位模式下的逻辑运算

修改整数值中的位时，可以使用 4 个按位运算符，如表 3-1 所示。

表 3-1 按位运算符

运 算 符	说 明
~	这是按位求反运算符。它是一个一元运算符，可以反转操作数中的位，即 1 变成 0，0 变成 1
&	这是按位与运算符，它对操作数中相应的位进行与运算。如果相应的位都是 1，结果位就是 1，否则就是 0
^	这是按位异或运算符，它对操作数中相应的位进行异或运算。如果相应的位各不相同，例如一个位是 1，另一个位是 0，结果位就是 1。如果相应的位相同，结果位就是 0
|	这是按位或运算符，它对操作数中相应的位进行或运算。如果两个对应的位中有一个是 1，结果位就是 1。如果两个位都是 0，结果就是 0

表 3-1 中的运算符按照其优先级排列，在这个集合中，按位求反运算符的优先级最高，按位或运算符的优先级最低。在附录 D 的运算符优先级表中，按位移动运算符<<和>>具有相同的优先级，它们位于～运算符的下面，&运算符的上面。

如果以前没有见过这些运算符，就会问“这非常有趣，但这是为什么？”。下面就将它们用于实践。

1. 使用按位与运算符

按位与运算符一般用于选择整数值中特定的一个位或一组位。为了说明这句话的含义，下面再次使用本节开头的例子，利用一个 16 位整数存储字体的特性。

假定声明并初始化一个变量，指定一种 12 磅字号、斜体、样式为 6 的字体。实际上，就

是图 3-1 中的字体。样式的二进制值是 00000110，斜体位是 1，黑体位是 0，字号是 01100。还有一个没有使用的位，需要把 font 变量的值初始化为二进制数 0000 0110 0100 1100。

由于 4 位二进制数对应于一个 16 进制数，因此最简单的方法是以十六进制方式指定初始值：

```
unsigned short font=0x064C;          // Style 6, italic, 12 point
```

注释：

在建立像这样的位模式时，十六进制表示法要比十进制表示法更合适。

要使用字号，需要从 font 变量中提取它，这可以使用按位与运算符来实现。只有当两个位都是 1 时，按位与运算符才会产生 1，所以可以定义一个值，在将定义字号的位和 font 执行按位与操作时选择该位。为此，只需定义一个值，该值在我们感兴趣的位上包含 1，在其他位上包含 0。这种值称为掩码，用下面的语句定义它：

```
unsigned short size_mask=0x1F;       //Mask is 0000 0000 0001 1111
                                     //to select size
```

font 变量的 5 个低位表示其字号，把这些位设置为 1，剩余的位设置为 0，这样它们就会被舍弃(二进制数 0000 0000 0001 1111 可转换为十六进制数 1F)。

现在可以用下面的语句提取 font 中的字号了：

```
unsigned short size=font & size_mask;
```

在&操作中，当两个对应的位是 1 时，结果位就是 1。任何其他组合起来的结果就是 0。因此组合起来的值如下：

```
font                0000 0110 0100 1100
size_mask           0000 0000 0001 1111
font & size_mask    0000 0000 0000 1100
```

把二进制值分解为 4 位一组的形式并不是很重要，这只是易于表示对应的十六进制数，看出其中有多少位。掩码的作用是把最右边的 5 位分隔出来，这 5 位表示点数(即字号)。

可以使用同样的方法选择字体的样式，只是还需要使用按位移动运算符把样式值向右移动。可以用下面的语句定义一个掩码，选择左边的 8 位，如下所示：

```
unsigned short style_mask=0xFF00;    //Mask is 1111 1111 0000 0000
                                     //for style
```

用下面的语句获取样式值：

```
unsigned short style=(font & style_mask) >> 8;      //Extract the style
```

该语句的结果如下：

```
font                          0000 0110 0100 1100
style_mask                    1111 1111 0000 0000
font & style_mask             0000 0110 0000 0000
(font & style_mask) >> 8      0000 0000 0000 0110
```

为表示斜体和黑体的位定义掩码，并把相应的位设置为 1，就很容易把它们分隔出来。当然，还需要一种方式来测试得到的位，这部分内容详见第 4 章。

按位与运算符的另一个用途是关闭位。前面介绍的是掩码中为 0 的位在结果中也将输出 0。例如，为了关闭表示斜体的位，其他的位不变，只需定义一个掩码，使该掩码中的斜体位为 0，其他位为 1，再对 font 变量和该掩码进行按位与操作即可。实现此操作的代码将在按位或运算符一节中介绍。

2. 使用按位或运算符

可以使用按位或运算符设置一个或多个位。继续操作前面的 font 变量，现在需要设置斜体和黑体位。用下面的语句可以定义掩码，选择这些位：

```
unsigned short italic=0x40U;      //Seventh bit from the right
unsigned short bold=0x20U;        //Sixth bit from the right
```

用下面的语句设置黑体位：

```
font |= bold;                    // Set bold
```

位的组合如下：

```
font                  0000 0110 0100 1100
bold                  0000 0000 0010 0000
font | bold           0000 0110 0110 1100
```

现在，font 变量指定它表示的字体是黑体和斜体。注意这个操作会设置位，而不考虑以前的状态。如果以前位的状态是开，则现在仍保持开的状态。

也可以对掩码执行按位或操作，设置多个位。下面的语句就同时设置了黑体和斜体：

```
font |= bold | italic;          //Set bold and italic
```

该语言很容易让人选择错误的运算符。“设置斜体和黑体”很容易让人觉得应使用&运算符，而这是错误的。对两个掩码执行按位与操作会得到一个所有位都是 0 的值，这不会改变字体的任何属性。

如上一节最后所述，可以使用&运算符关闭位。也就是定义一个掩码，把其中要关闭的位设置为 0，其他位设置为 1。但如何指定这样的掩码？如果要显式指定它，就需要知道变量中有多少个字节，如果希望程序可以任何方式移植，这就不很方便。可是，在通常用于打开位的掩码上使用按位求反运算符，就可以得到这样的掩码。在 bold 掩码上关闭黑体位，就可以得到该掩码：

```
bold                  0000 0000 0010 0000
~bold                 1111 1111 1101 1111
```

按位求反运算符的作用是反转原数值中的每一位，使 0 变成 1，1 变成 0。无论 bold 变量占用 2 个字节、4 个字节还是 8 个字节，这都会生成我们期望的结果。

提示：

按位求反运算符有时称为 NOT 运算符，因为对于它操作的每个位，都会得到跟开始不同的值。

因此，在关闭黑体位时，只需对掩码 bold 的反码和 font 变量执行按位与操作，可用的语句如下所示：

```
font &= ~bold;          //Turn bold off
```

还可以使用&运算符把几个掩码组合起来，再对结果跟要修改的变量执行按位与操作，将多个位设置为 0。例如：

```
font &= ~bold & ~italic;          //Turn bold and italic off
```

这个语句把 font 变量中的斜体和黑体位设置为 0。注意这里不需要括号，因为~运算符的优先级高于&运算符。但是，如果不清楚运算符的优先级，就应加上括号，表示希望执行的操作。这肯定是无害的，在需要括号时还可以正常发挥作用。

3. 使用按位异或运算符

按位异或运算符的使用频率远远低于&和 | 运算符，有关它的使用例子也比较少。但它的一个重要应用是图形编程。在屏幕中创建动画的一种方式是绘制一个对象，删除它，再在一个新位置重新绘制。如果要求动画很平滑，这个过程就需要重复得很快，其中删除是一个重要的部分。我们并不想删除和重新绘制整个屏幕，因为这非常费时，屏幕也会出现闪烁。最理想的是，只删除屏幕上要移动的对象。使用所谓的异或模式就可以做到这一点，得到非常平滑的动画。

异或模式的理念是，在屏幕上用给定的颜色绘制对象，如果接着用背景色重新绘制它，它就会消失。如图 3-3 所示。

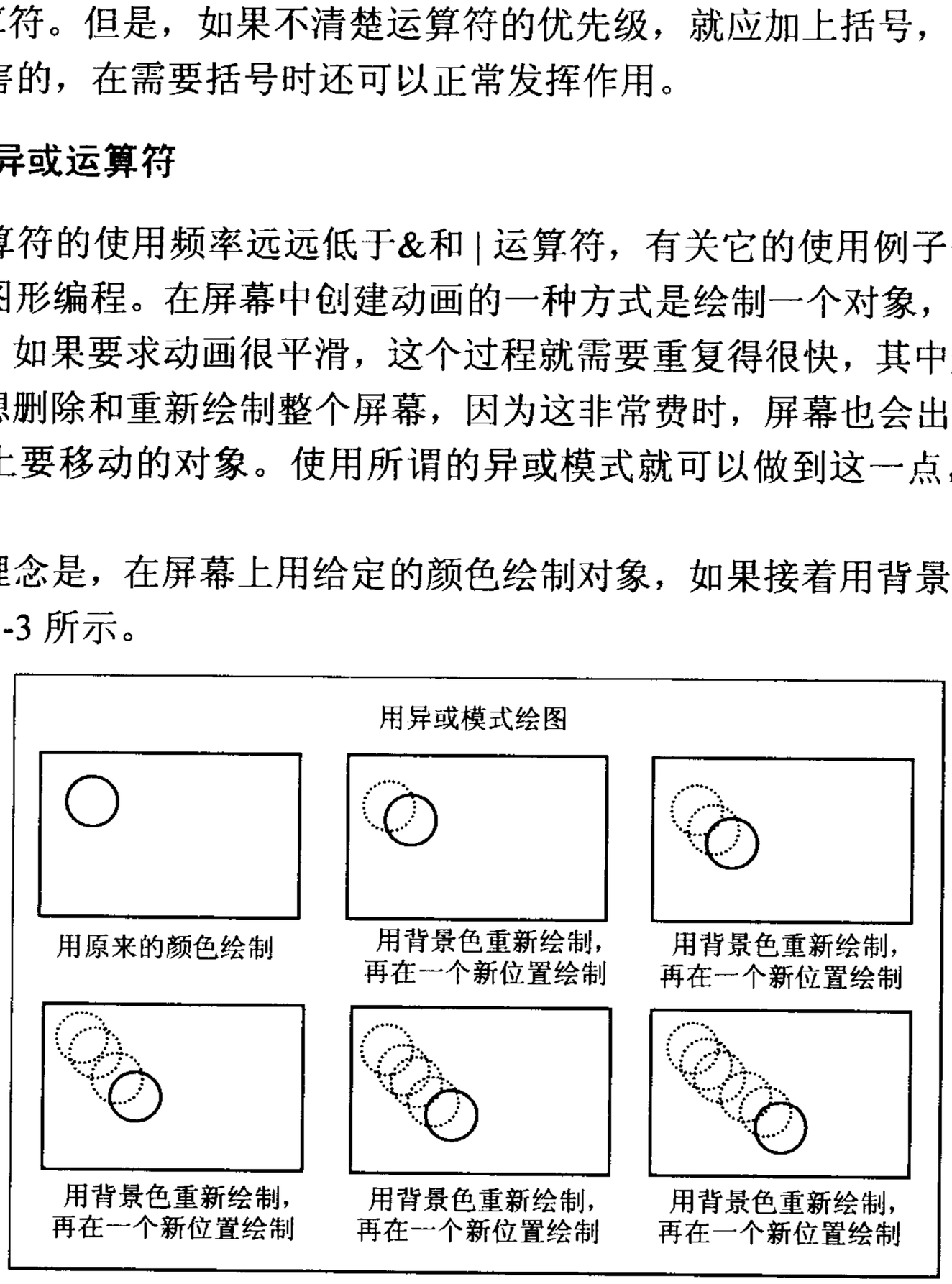


图 3-3　用异或模式绘图

以异或模式在屏幕上绘制对象时，每次绘制对象的颜色会自动在为对象所选的颜色和背景色之间来回变化。得到这一效果的关键是使用按位异或运算符快速而自动地改变颜色。它使用异或运算符的一个特性，即如果对两个值进行异或操作，再对所得的结果和一个原始值执行异或操作，就会得到另一个值。这听起来很复杂，下面就用一个例子来说明。

假定要在前景色(这里使用红色)和背景色(白色)之间来回切换。颜色通常用 3 个 8 位值来表示，分别对应于红、蓝、绿的亮度，存储在一个 4 字节的整数中。通过改变颜色中的红、蓝

和绿的比例，就可以获得大约 1600 万种不同的颜色，包括从白色到黑色之间的所有颜色。纯红色是 0xFF0000，这时红色成分设置为其最大值，其他两种颜色即蓝色和绿色的成分设置为 0。在相同颜色模式下，绿色就是 0xFF00，蓝色是 0xFF。在白色中，红、蓝、绿的成分具有相同的最大值，即 0xFFFFFF。

可以用下面的语句定义表示红色和白色的变量：

```
unsigned long red=0XFF0000UL;          //Color red
unsigned long white=0XFFFFFFUL;        //Color white-RGB all maximum
```

接着创建一个掩码，用于在红色和白色之间来回切换，并把包含绘图颜色的变量初始化为红色：

```
unsigned long mask=red ^ white;        //Mask for switching colors
unsigned long draw_color=red;          //Drawing color
```

变量 mask 初始化为要切换的两种颜色的按位异或操作结果，因此：

```
red                    1111 1111 0000 0000 0000 0000
white                  1111 1111 1111 1111 1111 1111
mask(即 red ^ white)   0000 0000 1111 1111 1111 1111
```

如果对 mask 和 red 执行异或操作，就会得到 white，如果对 mask 和 white 执行异或操作，就会得到 red。因此，使用 draw_color 中的颜色绘制对象，就可以通过下面的语句切换颜色：

```
draw_color ^= mask;      //Switch the drawing color
```

当 draw_color 包含 red 时，其执行过程如下：

```
draw_color             1111 1111 0000 0000 0000 0000
mask                   0000 0000 1111 1111 1111 1111
draw_color ^ mask      1111 1111 1111 1111 1111 1111
```

显然，draw_color 的值从红色变为白色。再次执行这个语句，就会把颜色改回为红色：

```
draw_color^=mask;        //Switch the drawing color
```

其执行过程如下：

```
draw_color             1111 1111 1111 1111 1111 1111
mask                   0000 0000 1111 1111 1111 1111
draw_color ^ mask      1111 1111 0000 0000 0000 0000
```

draw_color 又变成了红色。这个技术适用于任意两种颜色，当然它实际上与特定颜色没有一点关系，可以把它用于切换任意一对整型数值。

程序示例 3.4——使用按位运算符

下面用一个例子来试验按位运算符，看看它们如何一起工作。本例还演示了如何使用异或运算符在两个值之间切换，以及如何使用掩码来选择和设置各个位。代码如下：

```
//Program 3.4 Using the bitwise operators
#include <iostream>
#include <iomanip>
```

```
using std::cout;
using std::endl;
using std::setfill;
using std::setw;

int main() {
  unsigned long red=0xFF0000UL;          //Color red
  unsigned long white=0xFFFFFFUL;        //Color white - RGB all maximum

  cout << std::hex;                 //Set hexadecimal output format
  cout << setfill('0');             //Set fill character for output

  cout << "\nTry out bitwise AND and OR operators.";
  cout << "\nInitial value     red      = "<< setw(8) << red;
  cout << "\nComplement       ~red      = "<< setw(8) <<~ red;

  cout << "\nInitial value   white      = "<< setw(8) << white;
  cout << "\nComplement    ~ white      = "<< setw(8) <<~ white;

  cout << "\n Bitwise AND   red & white = " << setw(8)  << (red & white);
  cout << "\n Bitwise OR     red | white = " << setw(8)  << (red | white);

  cout << "\n\nNow we can try out successive exclusive OR operations.";

  unsigned long mask=red ^ white;

  cout << "\n      mask= red ^ white = " << setw(8) << mask;
  cout << "\n      mask ^ red = " << setw(8)  << (mask ^ red);
  cout << "\n      mask ^ white = " << setw(8)  << (mask ^ white);

  unsigned long flags=0xFF;             //Flags variable
  unsigned long bit1mask=0x1;           //Selects bit 1
  unsigned long bit6mask=0x20;          //Selects bit 6
  unsigned long bit20mask=0x80000;      //Selects bit 20

  cout << "\n\nNow use masks to select or set a particular flag bit.";
  cout << "\nSelect bit 1 from flags  : " << setw(8)  << (flags & bit1mask);
  cout << "\nSelect bit 6 from flags  : " << setw(8)  << (flags & bit6mask);
  cout << "\nSwitch off bit 6 in flags    : " << setw(8)  << (flags &= ~bit6mask);
  cout << "\nSwitch on bit 20 in flags    : " << setw(8)  << (flags |= bit20mask);
  cout <<endl;
  cout <<endl;
  return 0;
}
```

该例子的输出如下：

```
Try out bitwise AND and OR operators.
Initial value    red          = 00ff0000
Complement      ~red          = ff00ffff
Initial value  white          = 00ffffff
Complement   ~ white          = ff000000
  Bitwise AND   red & white   = 00ff0000
```

```
  Bitwise OR     red | white = 00ffffff

Now we can try out successive exclusive OR operations.
          mask= red ^ white =0000ffff
                mask ^ red = 00ffffff
                mask ^ white =00ff0000

Now use masks to select or set a particular flag bit.
Select bit 1 from flags    : 00000001
Select bit 6 from flags    : 00000020
Switch off bit 6 in flags  : 000000df
Switch on bit 20 in flags  : 000800df
```

例子的说明

本例中添加了对标准头文件<iomanip>的#include 指令，这个头文件在第 2 章介绍过，因为代码将使用操纵程序控制输出的格式。首先，定义两个整数变量，它们包含的值表示要用于后续按位运算的颜色：

```
unsigned long red=0xFF0000UL;          //Color red
unsigned long white=0xFFFFFFUL;        //Color white - RGB all maximum
```

为了把数据显示为十六进制值，可用下面的语句指定：

```
cout << std::hex;          //Set hexadecimal output format
```

其中 hex 是一个操纵程序，它把整数值的输出表示为十六进制。注意，这是模式化的，该程序以后在标准输出流中的所有整数输出都采用十六进制格式。不需要把 hex 发送给输出流 cout。如果需要，可以用下面的语句把输出格式改回为十进制：

```
cout << std::dec;          //Set decimal output format
```

这个语句使用 dec 操纵程序，把整数输出重新设置为默认的十进制表示。注意，把输出格式设置为十六进制，仅影响整数值。浮点数值会继续显示为一般的十进制。

如果在输出整数时加上前导 0，就会使结果更清晰易懂。用下面的语句设置这种模式：

```
cout << setfill('0');          //Set fill character for output
```

其中 setfill()是一个操纵程序，它把填充字符设置为括号中的字符。这也是模式化的，这样，以后的所有整数输出都会在需要时使用这个填充字符。它对十进制和十六进制输出都起作用。如果要用星号代替该填充字符，则可以使用下面的语句：

```
cout << setfill('*');          //Set fill character for output
```

要把填充字符设置回原来的默认值，只需在括号中使用空格：

```
cout << setfill(' ');          //Set fill character for output
```

下面的语句显示 red 的值及其反码：

```
cout << "\nInitial value    red       = "<< setw(8) << red;
cout << "\nComplement      ~red       = "<< setw(8) <<~ red;
```

这里使用第 2 章介绍的 setw()操纵程序，把输出字段宽度设置为 8。如果所有的输出值都采用相同的字段宽度，就很容易比较它们。设置宽度不是模式化的，它只应用于跟在字段宽度设置点后面的下一条语句的输出。在 red 和 white 的输出中，～运算符反转了其操作数的位。

下面的语句使用按位与以及按位或运算符来合并 red 和 white：

```
cout << "\n Bitwise AND   red & white = " << setw(8)  << (red & white);
cout << "\n Bitwise OR    red | white = " << setw(8)  << (red | white);
```

注意输出中表达式的括号。它们是必需的，因为<<的优先级高于&和|。没有括号，该语句就不会编译。如果查看一下输出，就会看出它跟这里讨论的相同。若对两个值都为 1 的位执行按位与操作，就会得到 1，否则结果就是 0。在对两个位执行按位或操作时，除非两个位都是 0，否则结果就是 1。

然后，创建一个掩码，在通过按位异或运算符组合两个值时，该掩码用于反转 red 和 white 的值。

```
unsigned long mask=red ^ white;
```

如果查看一下 mask 值的输出，就会发现在两个位的值不同时，对两个位执行异或操作的结果是 1，在两个位的值相同时，该操作的结果是 0。利用异或运算符把 mask 和两个颜色值中的一个组合起来，就会得到另一个颜色值。这可以用下面的语句来说明：

```
cout << "\n      mask ^ red = " << setw(8)  << (mask ^ red);
cout << "\n      mask ^ white = " << setw(8)  << (mask ^ white);
```

最后一组语句演示了如何使用掩码从一组标记位中选择一个位。选择某个位的掩码必须使该位的值为 1，其他位的值为 0。因此，从一个 32 位 long 变量中选择第 1、6 和 20 位的掩码定义如下：

```
unsigned long bit1mask=0x1;         //Selects bit 1
unsigned long bit6mask=0x20;        //Selects bit 6
unsigned long bit20mask=0x80000;    //Selects bit 20
```

要从 flags 中选择一个位，只需对相应的掩码和 flages 的值执行按位与操作。例如：

```
cout << "\nSelect bit 6 from flags : " << setw(8)  << (flags & bit6mask);
```

从输出中可以看到，表达式(flags & bit6mask)的结果是只设置了第 6 位的整数。当然，如果 flages 中的第 6 位为 0，该表达式的结果就是 0。

要关闭一个位，需要对 flages 变量和一个掩码执行按位与操作。在该掩码中，要关闭的那个位是 0，其他位是 1。对掩码和对应的位执行按位求反操作，也可以关闭该位。bit6mask 就是这样的一个掩码。下面的语句把 flags 中的第 6 位关闭，并显示结果：

```
cout << "\nSwitch off bit 6 in flags : " << setw(8)  << (flags &=  ~bit6mask);
```

当然，如果第 6 位已经是 0，该位就保持不变。要打开一个位，只需对 flages 和一个掩码执行按位或操作，在该掩码中，要打开的那个位是 1：

```
cout << "\nSwitch on bit 20 in flags : " << setw(8)  << (flags |=  bit20mask);
```

这个语句把 flags 中的第 20 位设置为 1，并显示结果。如果这个位已经是 1，该位将保持不变。

4. 输出操纵程序

把第 2 章介绍的也算在内，到目前为止我们已介绍了 5 个模式化输出操纵程序，它们都是在<iostream>头文件中定义的：scientific、fixed、dec、hex 和 oct。表 3-2 列出所有的其他相似的操纵程序。目前不介绍后两项中的 bool 值，它将在第 4 章讨论。

表 3-2　输出操纵程序

操 纵 程 序	执行的动作
dec	把整数值格式化为十进制。这是默认的表示法
hex	把整数值格式化为十六进制
oct	把整数值格式化为八进制
left	使输出字段中的值左对齐，其右端用填充字符填充。默认的填充字符是空格
right	使输出字段中的值右对齐，其左端用填充字符填充。这是默认的对齐方式
fixed	以固定点表示法输出浮点数值，即不带指数
scientific	以科学表示法输出浮点数值，即尾数加指数的方式。默认的模式根据要显示的数值，选择 fixed 或 scientific 表示法
showpoint	给浮点数值显示小数点和尾部的 0
noshowpoint	与上一个操纵程序相反。这是默认的
showbase	在八进制的输出前面加上前导 0；在十六进制的输出前面加上前导 0x 或 0X
noshowbase	八进制和十六进制的输出中不显示前缀。这是默认的
showpos	正数前面加上+号
noshowpos	正数前面不显示+号，这是默认的
uppercase	在以十六进制格式输出整数时，给十六进制数字显示大写字母 A 到 F。如果设置了 showbase，还要显示 0X。在以科学计数法输出数值时，给指数显示 E，而不是使用小写字母 e
nouppercase	对上述项使用小写字母，这是默认的
boolalpha	把 bool 值显示为 true 和 false
noboolalpha	把 bool 值显示为 1 和 0

可以一次设置多种模式，方法是在流中插入多个操纵程序。例如，如果要把整型数据输出为十六进制值，且在输出字段中左对齐，就可以使用下面的语句：

```
std::cout << std::hex << std::left << value;
```

这个语句会把 value(以及程序中后续的所有整数，除非改变了设置)输出为左对齐的十六进制数值。

表 3-3 列出了需要提供参数值的输出操纵程序。

表 3-3　需要参数的输出操纵程序

操 纵 程 序	执行的动作
setfill()	把填充字符设置为参数指定的字符。默认的填充字符是空格
setw()	把字段宽度设置为参数指定的值
setprecision()	把浮点值的精度设置为参数指定的值。精度是输出中十进制数字的个数

3.4　枚举数据类型

有时需要使变量具有限定的一组值，并可以通过名称来引用这些值。例如一星期的每一天或一年中的各个月份。在 C++中有一种特殊的功能来处理这种情况，称为枚举。因为在定义枚举时，实际上是在创建一个新的类型，所以它也称为已枚举的数据类型。下面创建一个利用该理念的例子——一个变量可以假定某值对应于一星期的某一天。定义该枚举的语句如下：

```
enum Weekday {Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday};
```

这个语句声明一个已枚举的数据类型 Weekday，这个类型的变量值只能是花括号中的值，即从 Monday 到 Sunday。如果要把 Weekday 类型的变量值设置为不在花括号中的值，就会出现错误。列在花括号中的符号名称叫作枚举成员。

实际上，每一天的名称都自动定义为表示一个固定的整数值。列表中的第一个名称 Monday，其值为 0，Tuesday 的值为 1，依此类推，Sunday 的值为 6。可以用下面的语句把 today 声明为枚举类型 Weekday 的一个实例：

```
Weekday today=Tuesday;
```

类型 Weekday 的用法跟前面介绍的基本类型相似。这个 today 的声明还把该变量的值初始化为 Tuesday。如果输出 today 的值，就会显示 1。

在默认情况下，枚举声明中每个枚举成员的值都比前面一个枚举成员的值大 1，第一个枚举成员的值则是 0。如果喜欢从另一个数值开始计数，可以使用下面的语句，使枚举成员的值为 1 到 7：

```
enum Weekday {Monday=1, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday};
```

枚举成员不一定有惟一值。可以把 Monday 和 Mon 都定义为 1，例如下面的语句：

```
enum Weekday {Monday=1, Mon=1, Tuesday, Wednesday, Thursday, Friday, Saturday,
              Sunday};
```

该语句允许用 Mon 或 Monday 作为一星期中的第一天。然后，用下面的语句设置已声明为类型 Weekday 的变量 yesterday：

```
yesterday=Mon;
```

还可以根据列表中以前的枚举成员定义新的枚举成员。把前面的所有代码都放在一个例子中，将类型 Weekday 声明为：

```
enum Weekday { Monday,                      Mon= Monday,
               Tuesday = Monday +2,         Tues= Tuesday,
               Wednesday= Tuesday+2,        Wed= Wednesday,
               Thursday= Wednesday+2,       Thurs= Thursday,
               Friday= Thursday+2,          Fri= Friday,
               Saturday= Friday+2,          Sat= Saturday,
               Sunday= Saturday +2,         Sun= Sunday };
```

现在，类型为 Weekday 的变量值可以是从 Monday 到 Sunday，从 Mon 到 Sun，它们对应于整数值 0、2、4、6、8、10 和 12。

如果愿意，可以给所有的枚举成员显式赋值。例如，可以定义下面的枚举：

```
enum Punctuation {Comma=',', Exclamation='!', Question='?'};
```

这里把类型为 Punctuation 的变量的可能值定义为对应符号的数字值。如果查看一下附录 A 中的 ASCII 表，就可以看到，这些符号在十进制中分别是 44、33 和 63。这说明不一定必须以升序方式赋值。如果没有显式指定，枚举成员的值就是前一个枚举成员的指定值加 1，如第二个 Weekday 例子所示。

为枚举成员指定的值必须是编译期间的常量，即编译器可以计算出来的常量表达式。这种表达式只包括字面量、以前定义的枚举成员和声明为 const 的变量。不能使用非 const 的变量，即使已初始化，也不行。

3.4.1 匿名枚举

在定义枚举的同时声明变量，如果不需要在以后声明这种类型的其他变量，就可以省略枚举类型。例如：

```
enum {Monday, Tuesday, Wednesday, Thursday,
                          Friday, Saturday, Sunday} yesterday, today, tomorrow;
```

这个语句声明了 3 个变量，它们的值可以假设为从 Monday 到 Sunday。由于没有指定枚举类型，所以不能引用它。以后也不能再声明这种枚举的其他变量，因为这么做需要重复定义，这是不允许的。

匿名枚举类型的一个常见应用是作为定义整数常量的另一种方式。例如：

```
enum {feetPerYard = 3, inchesPerFoot = 12, yardsPerMiles= 1760 };
```

这个枚举包含了3个已显式赋值的枚举成员。尽管没有声明这种已枚举的数据类型的变量，但仍可以在算术表达式中使用其中的枚举成员。可以编写下面的语句：

```
std::cout << std::endl << "Feet in 5 miles = " << 5 * feetPerYard * yardsPerMile;
```

枚举成员会自动转换为 int 类型。这看起来似乎是使用枚举来定义整型常量，但在讨论类时就会看到，它提供了在类中包含常量的一种有效方式。现在详细论述一下已枚举的数据类型的转换。

3.4.2 在整型和枚举类型之间强制转换

在混合的算术表达式中，除了枚举成员本身之外，还可以使用枚举类型的变量。已枚举的数据类型会自动转换为相应的类型，但整型类型不能自动转换为枚举类型。如果把变量 today 声明为前面定义的 Weekday 类型，则可以使用下面的语句：

```
today = Tuesday;                    //Assign an enumerator value
int day_value= today +1;            //Calculate with an enumerator type
```

因为today的值是Tuesday，它对应于1，所以day_value就设置为2。尽管枚举成员 Wednesday 对应于值 2，但下面的语句不会编译：

```
today = day_value;                  //Error - no conversion
```

但是，把这个语句放在显式的强制转换语句中，就可以得到正确的结果：

```
today = static_cast<Weekday>(day_value);                    //OK
```

在进行显式的强制转换时，所强制转换的整数值必须位于枚举成员的取值范围之内，否则结果就是未定义的。这并不意味着它必须对应于某个枚举成员的值，而是说它必须等于或大于最小的枚举成员，且小于或等于最大的枚举成员。例如，定义一个枚举 Height，再声明这种类型的一个变量，其代码如下：

```
enum Height {Bottom, Top=20 } position;
```

枚举成员 Bottom 对应于值 0，而 Top 枚举成员对应于值 20。因此其取值范围是 0 到 20。可以用下面的语句给变量 position 赋值：

```
position= static_cast<Height>(10);
```

赋予 position 的值不对应于任何一个枚举成员，但这是一个合法的值，因为它处于枚举成员的最大值和最小值之间。对于前面 Punctuation 类型的变量，可以把 33 到 63 之间的任何整数合法地强制转换为这种类型，并存储它，但在这个例子中，很难看出这么做的目的。

程序示例 3.5——已枚举的数据类型

在通过比较已枚举的数据类型的变量值和可能的枚举成员以作决策时，枚举是非常有用的。第 4 章将对此详细介绍。下面仅举一个简单的例子，来演示对已枚举的数据类型的一些操作：

```
//Program 3.5 - Exercising an enumeration
#include <iostream>
using std::cout;

int main() {
  enum Language { English, French, German, Italian, Spanish } ;

  //Display range of enumerators
  cout << "\nPossible languages are:\n"
       << English << ". English\n"
```

```
        << French  << ". French\n"
        << German  << ". German\n"
        << Italian << ". Italian\n"
        << Spanish << ". Spanish\n";

Language tongue = German;
Cout << "\n Current language is " << tongue;

tongue = static_cast< Language >( tongue +1);
cout << "\n Current language is now " << tongue
     << std::endl;
  return 0;
}
```

这个例子的输出如下：

```
  Possible languages are:
  0. English
  1. French
  2. German
  3. Italian
  4. Spanish

    Current language is 2
    Current language is now 3
```

例子的说明

首先定义了一个枚举 Language，代码如下：

```
enum Language { English, French, German, Italian, Spanish } ;
```

Language 类型的变量值可以是花括号中的任何枚举成员值。用下面的语句列出所有可能的值：

```
Cout << "\nPossible languages are:\n"
     << English  << ". English\n"
     << French   << ". French\n"
     << German   << ". German\n"
     << Italian  << ". Italian\n"
     << Spanish  << ". Spanish\n";
```

枚举成员显示为其数字值，这样在输出枚举成员时，同时输出一个文本字符串，说明它对应的语言。

用下面的语句声明并初始化 Language 类型的一个变量：

```
Language tongue = German;
```

这个变量的值显示为 2，接着在下一个语句中给它指定新值：

```
tongue = static_cast< Language >( tongue +1);
```

在表达式 tongue +1 中，tongue 的值转换为 int 类型，再给它加上 1，得到 3，其类型为 int。之后通过显式强制转换，把它的类型转换回类型 Language，然后存储在 tongue 中。若没有显式强制转换，该语句就不会编译，因为整数类型不会自动转换为枚举类型。当然，tongue 的值显示为 3。

3.5 数据类型的同义词

枚举提供了定义自己的数据类型的一种方式。typedef 关键字允许把自己的数据类型名称指定为另一个类型的替代名称。在声明语句中使用 typedef 关键字，可以把类型名称 BigOnes 声明为标准类型 long 的等价名称：

```
typedef long BigOnes;          //Defining BigOnes as a type name
```

当然，这并没有定义一个新类型，只是把 BigOnes 定义为 long 的替代类型指定符，因此，下面的语句可以把 mynum 变量声明为 long 类型：

```
BigOnes mynum = 0;             //Declare & initialize a long int variable
```

这个声明跟使用标准的内置类型名称进行的声明没有区别，还可以使用下面的语句：

```
long int mynum = 0;            //Declare & initialize a long int variable
```

其效果是相同的。实际上，如果声明自己的类型名称(例如 BigOnes)，就可以在同一个程序中使用这两个类型指定符来声明类型相同的不同变量。但是，很难判断这么做的原因。

因为 typedef 只是创建了已有类型的同义词，所以它显得有点多余。实际上并不完全如此。typedef 的一个重要用途是为可能需要在多台计算机上运行的程序所使用的数据类型提供灵活性。本章在前面介绍过，标准库使用 typedef 定义了 size_t 类型，下面用一个实例来说明 typedef 的用法。

假定编写一个程序，它使用几个变量记录某类事件的个数，如记录在高速生产机器上每小时生产出来的巧克力棒数。这些数值通常需要使用 4 个字节的整数。

在一些计算机中，类型 int 占 2 个字节，不足以存储该程序中的整数。而在其他计算机上，类型 int 占 4 个字节，正好可以存储该程序中的整数。可以使用 long 类型来解决这个问题，long 类型通常至少是 4 个字节，但在一些计算机上，它是 8 个字节，这样就太浪费了，特别是在程序存储大量的整数时，就更是如此。在程序中，可以声明自己的类型，从而提供处理这种情况的灵活性。例如：

```
typedef int Counter;           //Define the integer type for the program
```

现在可以根据类型 Counter 来编写程序了，而不是使用标准类型 int。如果要在 int 类型的取值范围不够的机器上编译这个程序，可以把 Counter 重新定义为：

```
typedef long Counter;          //Define the integer type for the program
```

现在，程序中所有声明为 Counter 的整数都是 long 类型。

在学习了更多的预处理指令后，还可以做得更好。使用<climits>头文件中的信息，可以编写一个程序，根据每个整数类型的取值范围自动定义 Counter 类型。

typedef 还可以简化更复杂的类型声明。稍后将介绍类提供的定义全新数据类型的方式，那时就可以完全控制应用于新类型的属性和操作。

3.6 变量的生存期

在程序执行时，所有的变量都有有限的生存期。它们从被声明的那一刻起存在，并在某一刻消失，最迟也要在程序终止时消失。变量生存多长时间取决于属性“存储持续时间”。变量拥有的存储持续时间有 3 种：

- 自动的存储持续时间
- 静态的存储持续时间
- 动态的存储持续时间

变量具有哪种存储持续时间取决于创建它的方式。第 7 章将讨论具有动态存储持续时间的变量，本章只讨论其他两种的特性。

变量拥有的另一个属性是作用域。变量的作用域就是变量名有效的那部分程序。在变量的作用域中，可以合法地引用它，设置它的值，或在表达式中使用它。在变量的作用域之外，就不能引用它的名称，这么做会导致一个编译错误。注意，变量在其作用域之外仍然存在，只是不能根据名称来引用它。稍后将举几个例子来说明。

前面声明的所有变量都具有自动的存储持续时间，因此称为自动变量。下面做详细讨论。

3.6.1 自动变量

前面声明的变量都是在一个块中声明的，即在一对花括号中声明。它们称为自动变量，具有局部作用域或块作用域。自动变量的作用域是从声明它的那行代码起，直到包含其声明的块的结尾。

自动变量在被声明的那一刻“出生”，在包含其声明的块的结尾自动消失。这个结尾就是跟变量声明前面的第一个左花括号相匹配的右花括号。每次执行到包含自动变量声明的语句块时，就创建一个新的变量。如果为自动变量指定了初始值，则每次创建该变量时都会重新初始化它。

关键字 auto 可以用于显式指定自动变量，但它很少使用，因为在默认情况下它是隐含的。下面把前面讨论的内容放在一个例子中。

程序示例 3.6——自动变量

下面的例子演示了自动变量的生存期：

```
//Program 3.6 Demonstrating variable scope
#include <iostream>
using std::cout;
using std::endl;

int main() {                    //Function scope starts here
  int count1=10;
  int count3=50;
  cout << endl << "Value of outer count1 = " << count1;

  {                             // New block scope starts here……
    int count1=20;              // This hides the outer count1
```

```
  int count2=30;
  cout << endl << "Value of inner count1 = " << count1;
  count1 += 3;              // This changes the inner count1
  count3 += count2;
}                           //…and ends here.

cout << endl
     << "Value of outer count1 = " << count1
     << endl
     << "Value of outer count3 = " << count3;

  //cout << endl << count2;         //Uncomment to get an error
  cout << endl;
  return 0;
}                           //Function scope ends here
```

这个例子的输出如下：

```
Value of outer count1 = 10
Value of inner count1 = 20
Value of outer count1 = 10
Value of outer count3 = 80
```

例子的说明

前两个语句声明和定义了两个整数变量 count1 和 count3，其初始值分别是 10 和 50：

```
int count1=10;
int count3=50;
```

这两个变量在代码的这个位置开始存在，到程序结束的右花括号处消失。这些变量的作用域也是到 main()结束的右花括号为止。

提示：

变量的生存期和作用域是两个不同的概念，生存期是变量存在的执行时间段，作用域是可以使用变量名的程序代码区域。不要把它们混淆。

在定义变量后，输出 count1 的值，产生上面的第一行结果：

```
cout << endl << "Value of outer count1 = " << count1;
```

接着是第二个左花括号，开始一个新块。在这个块中定义了两个变量 count1 和 count2，其初始值分别是 20 和 30。这里声明的 count1 与第一个 count1 不同。尽管第一个 count1 仍然存在，但其名称被第二个 count1 掩盖了。在内层块中，在该声明之后使用的所有 count1 名称都是指在该块中声明的 count1。

以这种方式重复使用名称，只是为了说明发生的情况。一般情况下，这并不是一种好的编程习惯。在实际使用的程序中重复使用名称会产生混淆，也是不必要的，而且代码也非常容易出错。

下面的输出语句在第二行上显示变量的值，说明这里使用了内层作用域中的 count1，即在最内层的括号对中的 count1：

```
{                              // New block scope starts here…
  int count1=20;               // This hides the outer count1
  int count2=30;
  cout << endl << "Value of inner count1 = " << count1;
```

如果仍在使用外层的 count1，这个语句应输出值 10。下面的语句递增变量 count1：

```
count1 += 3;                   // This changes the inner count1
```

递增操作应用于内层作用域中的变量，因为外层的变量仍被掩盖。但是，在外层作用域中定义的 count3 可以通过下面的语句递增其值：

```
count3 += count2;
```

这说明，在外层作用域的开始处定义的变量仍可以在内层作用域中访问。它们可以在内层右花括号之后定义，仍位于外层作用域中，但在这种情况下，它们在内层块中不存在。

在结束内层作用域的括号之后，count2 和内层作用域的 count1 消失了，而外层作用域的 count1 和 count3 仍存在。它们的值用下面的语句显示，说明 count3 确实已在内层作用域中递增了：

```
cout << endl
     << "Value of outer count1 = " << count1
     << endl
     << "Value of outer count3 = " << count3;
```

如果去掉下面这一行的注释，

```
//cout << endl << count2;          //Uncomment to get an error
```

程序就不能再正确编译了，因为它试图输出不存在的变量。

3.6.2 定位变量的声明

把变量的声明放在什么地方有非常大的灵活性。最重要的问题是要考虑变量需要什么样的作用域。除此之外，一般应把声明放在靠近变量在程序中第一次使用的地方。尽量使程序编写得易于为其他程序员理解，把变量的声明放在靠近第一次使用该变量的地方，将有助于程序的理解。

也可以把变量的声明放在组成程序的所有函数之外。下面看看这对变量有什么影响。

3.6.3 全局变量

在所有块和类外部的声明的变量称为全局变量，具有全局作用域(也称为全局命名空间作用域)。也就是说，在声明后，它们在源文件的所有函数中是可以访问的。如果在一开始就声明变量，就可以在文件的任何位置访问它们。

全局变量在默认情况下也具有静态的存储持续时间。具有静态存储持续时间的全局变量从程序开始执行起存在，直到程序结束时消失。如果没有为全局变量指定初始值，在默认情况下它就初始化为 0。因为全局变量的初始化在 main()开始执行之前进行，所以它们总是可以在该

变量作用域内的任何代码中使用。

图 3-4 显示了源文件 Example.cpp 的内容，箭头表示每个变量的作用域。

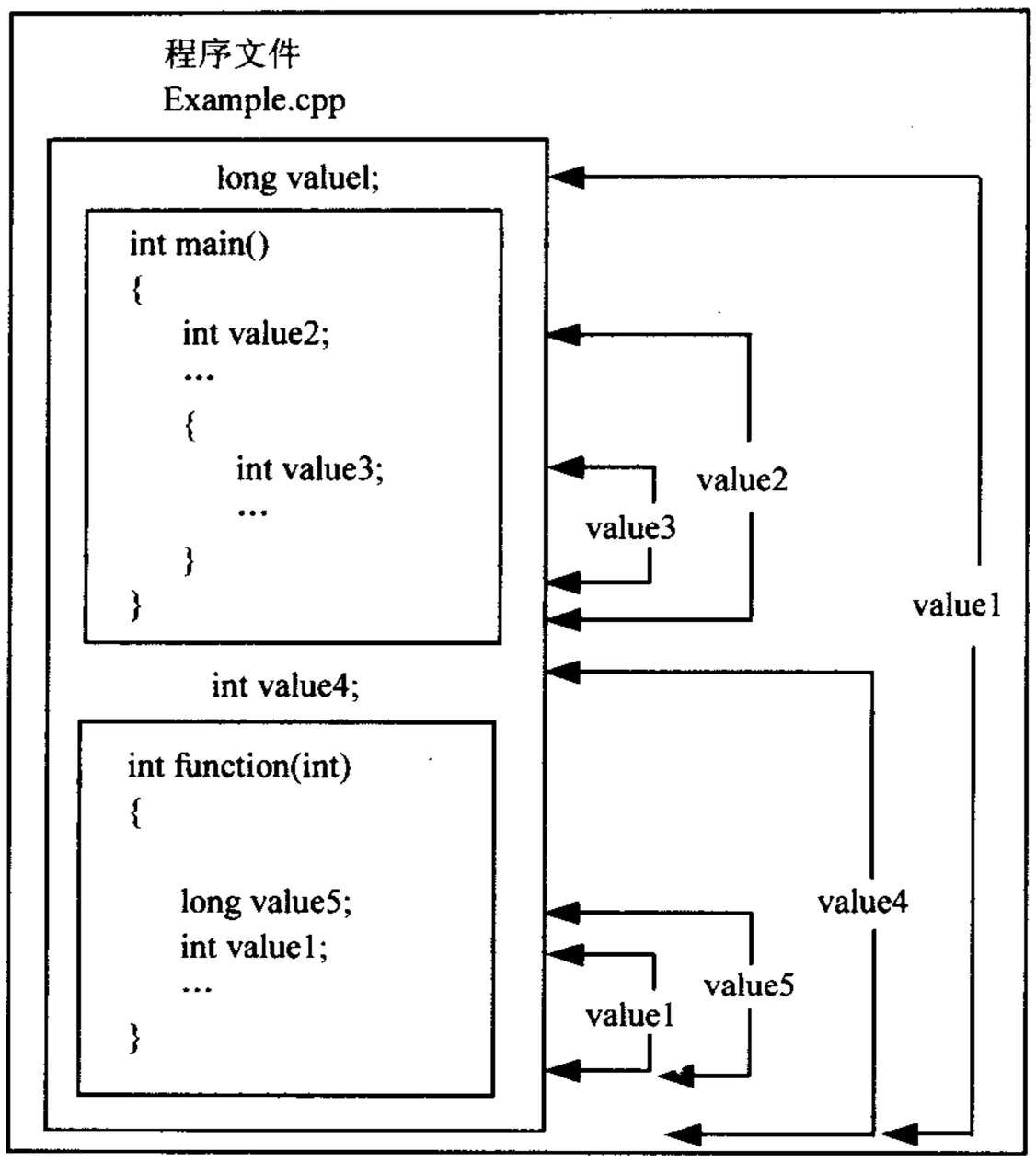


图 3-4　变量的作用域

图 3-4 列出了文件中每个变量的作用域。出现在文件开头的变量 value1 声明为全局作用域，出现在函数 main()之后的 value4 也是这样。全局变量的作用域从声明它们的那一行代码，到文件的结尾。即使 value4 在执行开始时就存在，也不能在 main()中引用它，因为 main()不在该变量的作用域中。为了在 main()中使用 value4，需要把它的声明移动到文件的开头。value1 和 value4 在默认情况下都初始化为 0，而自动变量就不是这样了。function()中的局部变量 value1 掩盖了同名的全局变量。

只要程序在运行，全局变量就一直存在，那么，为什么不把所有的变量都声明为全局变量，避免因局部变量消失而产生错误？这初听起来很吸引人，但实际上存在非常严重的缺陷，完全抵消了使用全局变量的优势。

实际应用的程序一般由大量的语句、函数和变量组成。把所有的变量都声明为全局作用域，会大大增加修改变量时出错的可能性，也使命名变量的工作变得非常难处理。这些变量还会在程序的整个执行期间占用内存。把变量声明为函数或块的局部变量，就可以确保它们几乎不受外界的影响，得到完全的保护。局部变量只在定义它们的那一行代码到块结束的这一区域存在和占用内存，整个开发过程会更容易维护。

程序示例 3.7——作用域解析运算符

如前所述，全局变量可以被同名的局部变量掩盖。但是，使用作用域解析运算符(::)(第 1 章在讨论命名空间时介绍过)可以访问该全局变量。下面用上一个例子的修订版本来演示该运算符的用法：

```
//Program 3.7 Using the scope resolution operator
#include <iostream>
using std::cout;
using std::endl;

int count1=100;                  //Global version of count1

int main() {                     //Function scope starts here
  int count1=10;
  int count3=50;
  cout << endl << "Value of outer count1 = " << count1;
  cout << endl << "Value of global count1 = " << ::count1;

  {                              // New block scope starts here…
    int count1=20;               // This hides the outer count1
    int count2=30;
    cout << endl << "Value of inner count1 = " << count1;
    cout << endl << "Value of global count1 = " << ::count1;
    count1 += 3;                 // This changes the inner count1
    count3 += count2;
  }                              //…and ends here.

  cout << endl
       << "Value of outer count1 = " << count1
       << endl
       << "Value of outer count3 = " << count3;

  //cout << endl << count2;          //Uncomment to get an error
  cout << endl;
  return 0;
}                                //Function scope ends here
```

编译并运行这个例子，会得到如下输出：

```
Value of outer count1 = 10
Value of global count1 = 100
Value of inner count1 = 20
Value of global count1 = 100
Value of outer count1 = 10
Value of outer count3 = 80
```

例子的说明

黑体字的代码行指出了对上一个例子的修改，这里仅需要讨论它们的作用。在函数 main() 的定义之前声明的 count1 是全局变量。按照规则，它可以在 main()函数中的任何地方使用。这个全局变量在声明时初始化为 100：

```
int count1=100;          //Global version of count1
```

还有另外两个变量也叫 count1，它们是在 main()中定义的，所以在整个程序中，全局变量 count1 被局部变量 count1 掩盖。实际上，在内层块中，全局变量 count1 被两个局部变量 count1 掩盖：一个是内层的 count1，另一个是外层的 count1。

第一个新的输出语句如下：

```
int count1=10;
int count3=50;
cout << endl << "Value of outer count1 = " << count1;
cout << endl << "Value of global count1 = " << ::count1;
```

这里使用了作用域解析运算符，告诉编译器引用全局变量 count1，而不是局部变量。因此输出中会显示全局变量的值。全局作用域解析运算符还可以应用于内层块，这可以从下面语句生成的输出中看到：

```
int count1=20;          // This hides the outer count1
int count2=30;
cout << endl << "Value of inner count1 = " << count1;
cout << endl << "Value of global count1 = " << ::count1;
```

上述语句输出了值 100，与前面一样——以这种方式使用作用域解析运算符总是可以引用全局变量。

在讨论面向对象的编程方法时，将详细讨论这个运算符。在面向对象的环境中，该运算符的使用更为广泛。第 10 章还将进一步论述命名空间，包括如何创建自己的命名空间。

3.6.4 静态变量

有时希望变量在本地的一个块中定义和访问，但在退出声明它的块后还继续存在。换言之，需要在块作用域中声明一个变量，但给它指定静态的存储持续时间。static 关键字就提供了这种功能，在第 8 章处理函数时，这种需求会更明显。

声明为 static 的变量会在程序的整个生存周期中存在，即使它是在一个块中声明的，只能在这个块(或其子块)中使用，也是如此。它仍具有块作用域，但具有静态的存储持续时间。为了声明静态的变量 count，可以使用下面的语句：

```
static int count;
```

如果没有提供初始值，则具有静态存储持续时间的变量也总是会自动初始化。这里声明的变量 count 将初始化为 0。如果在声明静态变量时没有提供初始值，则它总是会自动初始化为 0，并转换为该变量适用的类型。自动变量就不是这样。如果没有初始化自动变量，它们就会包含程序在上次使用该变量占用的内存时遗留下来的垃圾值。

寄存器存储类指定符

register 指定符用于表示变量对于执行的速度非常重要，因此应放在机器的寄存器中(寄存器是一种特殊的高速存储工具，它独立于主内存，通常在处理器芯片上)。使用这个指定的例子如下：

```
register int index=0;
```

这里要求变量 index 使用寄存器。编译器没有义务完成这个请求，在许多编译器中，不会因此而分配一个寄存器。一般情况下，不应使用寄存器，除非绝对肯定要做的工作。大多数编译器会在没有任何提示的情况下很好地决定应该如何使用寄存器。

3.7 特殊的类型修饰符

volatile 修饰符用于指定变量的值可以由外部过程异步修改，例如中断例程：

```
volatile long data = 0L;   //Value may be changed by another process
```

使用 volatile 修饰符的结果是抑制编译器在一般情况下进行的优化。例如，在程序引用非 volatile 的变量时，编译器会重复使用该变量在以前加载到寄存器中的已有值，避免从内存中检索同一值的系统开销。如果变量声明为 volatile，则在每次使用其值时，都会进行检索。

3.8 声明外部变量

第 1 章介绍过，程序可以由几个源文件组成，大多数程序一般都会有几个源文件。如果程序由多个源文件组成，就需要在一个源文件中访问在另一个源文件中声明的全局变量，这一点使用 extern 关键字就可以实现。假定一个程序文件包含如下内容：

```
//File1.cpp
int shared_value=100;

//Other program code…
```

如果另一个文件中的代码需要访问 shared_value 变量，就可以使用下面的语句：

```
//File2.cpp
extern int shared_value;            //Declare variable to be external

int main() {
  int local_value= shared_value+10;
..//Plus other code…
}
```

由于 File2.cpp 中的第一个语句把 shared_value 变量声明为外部，因此这只是一个声明，不是定义。接着，在 main()中对 shared_value 变量的引用就是引用在第一个文件 File1.cpp 中定义的变量。

在声明外部变量时，不能使用初始化值。假定在第二个文件中有如下代码：

```
extern int shared_value=0;     //Wrong! Not an external declaration.
```

如果变量定义为局部变量，extern 声明就会被忽略。

3.9 优先级和相关性

本章介绍了许多新的运算符，表 3-4 总结一下这些运算符的优先级和相关性。

表 3-4　运算符的优先级和相关性

运　算　符	相　关　性	运　算　符	相　关　性
static_cast<>()	右	* / %	左
后缀++		二元+	左
后缀 - -		二元 -	
一元+	右	<< >>	左
一元-		&	左
前缀++		^	左
前缀--		|	左
~		= op=	右

表 3-4 中的这些运算符按优先级的高低顺序排列，表中每一行包含的运算符都具有相同的优先级。在表达式中，具有相同优先级的运算符的执行顺序取决于其相关性。如前所述，附录 D 中列出了所有 C++运算符的优先级。

3.10　本章小结

本章介绍了 C++中一些较复杂的计算，还讨论了如何定义自己的数据类型，但这里介绍的内容跟定义通用类型没有任何关系。通用类型的定义详见第 11 章。本章的主要内容如下：

- 可以在表达式中混合使用不同类型的变量和常量。编译器会在需要时，自动把变量转换为相应的类型。
- 当等号右边的类型与等号左边的类型不同时，也可以把等号右边的类型自动转换为等号左边的类型。当左边的类型不能完全包含与右边类型的信息相同的信息时，就可能丢失信息。例如把 double 转换为 int 或把 long 转换为 short。
- 使用 static_cast<>()，可以把一种基本类型的值显式转换为另一种基本类型。
- 在默认情况下，在一个块中声明的变量是自动变量，也就是说，它在声明它的那行代码处开始存在，到包含其声明的块的结尾处消失。块尾用右花括号表示。
- 变量可以声明为静态，此时该变量存在于程序的整个生存周期。但是，它只能在定义它的作用域中访问。如果没有显式初始化静态变量，它就会默认初始化为 0。
- 在程序中，变量可以在所有块的外部声明，此时该变量具有全局命名空间作用域，在默认情况下具有静态的存储持续时间。在包含它们的程序文件中，具有全局作用域的变量可以在声明它之后的任何位置访问，除非存在一个与该全局变量同名的局部变量。即使如此，全局变量仍可以使用作用域解析运算符(::)访问。
- 关键字 typedef 允许定义其他类型的同义词。
- extern 关键字允许引用在另一个文件中定义的全局变量。

3.11 练习

1. 编写一个程序，计算用户输入的非 0 整数的倒数(整数 n 的倒数就是 1/n)。该程序应把计算的结果存储在 double 类型的变量中，再输出它。

2. 创建一个程序，提示用户以十进制形式输入一个整数，再对其二进制表示的最后一位求反。也就是说，如果最后一位是 1，就把它改为 0，反之亦然。结果应显示为一个十进制数。这种调整如何影响整数值(提示：使用按位运算符)?

3. 编写一个程序，计算矩形搁板的一层可以容纳多少个正方形盒子，且不会出现盒子悬垂的情况。使用 double 类型的变量表示搁板的长度和深度(单位是英尺)以及盒子一边的长度(单位是英寸)，从键盘上读取这些值。需要声明并初始化一个常量，用于把英尺转换为英寸。在一个语句中，计算搁板的一层可以容纳多少个盒子，并把结果赋予类型为 long 的变量。

4. 如果不运行下面的代码，能不能看出这些代码的输出结果？

```
unsigned int k = 430U;
unsigned int j = ( k >> 4 ) & ~ ( ~0 << 3 );
std::cout <<j;
```

5. 编写一个程序，从键盘上读取四个字符，把它们放在一个四字节的整型变量中，把这个变量的值显示为十六进制。分解变量的四个字节，以相反的顺序输出它们，先输出低位字节。

第 4 章　选择和决策

作决策是任何一种计算机编程的基础。如果不能根据比较数据值的结果来改变程序中指令的执行顺序，就不能用计算机程序解决大多数问题。

本章将探讨如何在 C++程序中作出选择和决策。这将允许检查程序输入的有效性，编写出根据输入数据来执行相应动作的程序。程序应可以解决基于逻辑的问题。

本章主要内容

- 如何比较数据值
- 如何根据结果修改程序的执行顺序
- 逻辑运算符和逻辑表达式的概念以及用法
- 如何处理多个选择

4.1　比较数据值

要作出决策，需要一种比较机制，而比较有几种类型。像“如果交通信号灯是红色，就停车”这样的决策就涉及到相等比较。在这种情况下，需要比较信号灯的颜色和参考颜色(即红色)，如果它们相同，就停车。另一方面，像“如果车速超过极限值，就减速”这样的决策涉及到另一种关系：检查车速是否大于当前的速度极限值。这两种比较是类似的，因为它们都会得到下面两个值中的一个：真或假。这就是 C++中的比较过程。

使用一些运算符就可以比较数据值，这些运算符称为关系运算符。用于比较两个值的有 6 个基本运算符，如表 4-1 所示。

表 4-1　关系运算符

关系运算符	说　明	关系运算符	说　明
<	小于	<=	小于等于
>	大于	>=	大于等于
==	等于	!=	不等于

注释：

其中的“等于”比较运算符(==)是两个连续的等号，它不同于只包含一个等号的赋值运算符(=)。在进行相等比较时，常常容易犯只使用一个等号的错误。对此，编译器一般不会发出错误消息，因为表达式可能是有效的，只是不能得出我们希望的结果。所以用户需要非常小心，避免出现这种错误。

这些二元运算符都是比较两个值，如果比较是真，就返回 true，否则就返回 false。值 true 和 false 是 C++中的关键字，也是一种新的字面量，称为布尔字面量(以布尔代数之父 George Boole 的名字命名)，其类型是 bool。

如果把值 true 强制转换为整数类型，结果就是 1，如果把值 false 强制转换为整数类型，结果就是 0。还可以把数值强制转换为 bool 类型。0 会强制转换为 false，其他非 0 值则强制转换为 true。

与其他标准类型一样，也可以创建 bool 类型的变量，来存储布尔值。声明该变量的方法与声明其他类型的变量一样，例如：

```
bool decision = true;    //Declare, define and initialize a logical variable
```

这个语句把变量 decision 声明并定义为布尔类型，还给它赋予初始值 true。

4.1.1 应用比较运算符

下面介绍几个使用比较运算符的例子，说明其工作原理。假定有两个整数变量 i 和 j，其值分别是 10 和 - 5。在下面的逻辑表达式中使用它们：

```
i >j    i!=j        j >-8       i <= j + 15
```

这些表达式都等于 true。注意在最后一个表达式 i <= j + 15 中，相加运算 j + 15 要先执行，因为+的优先级高于<=的优先级。可以在 bool 类型的变量中存储这些表达式的结果。例如：

```
decision = i >j;    //true if i is greater than j, false otherwise
```

如果 i 大于 j，就在 bool 变量 decision 中存储 true，否则就存储 false。

也可以比较 char 类型的变量。假定定义了下面的变量：

```
char first = 'A';
char last = 'z';
```

现在编写使用这些变量的比较示例，如下所示：

```
first < last          'E' <= first          first != last
```

第一个表达式检查 first 的值'A'是否小于 last 的值'z'，这总是 true。参考附录 A 中这些字符的 ASCII 编码，就可以验证这一点。大写字母用一组递增的数值 65～90 来表示，65 表示'A'，90 表示'Z'。第二个表达式的结果为 false，因为'E'大于 first 的值。最后一个表达式是 true，因为'A'肯定不等于'z'。

输出 bool 值与输出其他类型的值一样简单，下面举一个例子。

程序示例 4.1——比较数据值

这个例子从键盘上读取两个 char 值，输出比较它们的结果：

```
//Program 4.1 Comparing data values
#include <iostream>
using std::cin;
using std::cout;
using std::endl;

int main() {
  char first = 0;                    //Stores the first character
```

```
char second = 0;                    //Stores the second character

//Prompt for and read in the first character
cout<<"Enter a character:";
cin>> first;

//Prompt for and read in the second character
cout << "Enter a second character:";
cin >> second;

  cout << "The value of the expression first < second is: "
       << ( first < second)
       << endl
       << "The value of the expression first == second is: "
       << ( first == second)
       << endl;

  retuen 0;
}
```

这个程序输出的结果如下所示：

```
Enter a character: p
Enter a second character: t
The value of the expression first < second is: 1
The value of the expression first == second is: 0
```

例子的说明

提示用户输入，并从键盘上读取字符的过程是前面介绍过的标准过程。在下面的语句中输出应用<和==运算符的结果：

```
cout << "The value of the expression first < second is: "
     << (first < second)
     << endl
     << "The value of the expression first == second is: "
     << (first == second)
     << endl;
```

注意把比较表达式括起来的括号是必不可少的，否则编译器就不能正确解释该语句，而输出错误消息。该表达式比较用户输入的第一个和第二个字符。从上面的输出结果可以看出，值 true 显示为 1，值 false 显示为 0。

true 和 false 的输出值 1 和 0 是 true 和 false 的默认表示。如果希望布尔值在屏幕上显示为 true 和 false，可以使用输出操纵程序 boolalpha 来实现。在 main()的开头添加下面的语句：

```
cout  << std::boolalpha;
```

如果再次编译并运行这个例子，bool 值就会显示为 true 或 false。要在程序中把布尔值的结果返回为默认设置，可以使用操纵程序 noboolalpha。

4.1.2 比较浮点数值

当然，也可以比较浮点数值。考虑一个略复杂的数值比较。首先，用下面的语句定义一些变量：

```
int i = -10;
int j = 20;
double x = 1.5;
double y = -0.25E-10;
```

现在看看下面的逻辑表达式：

```
-1 < y      j < (10-i)      2.0*x >= (3+y)
```

在表达式中，可以把数据值用作比较中的操作数。因为比较运算符的优先级总是低于算术运算符(参见附录 D)，所以不需要使用括号，但使用括号有助于使表达式更容易理解。

第一个比较的结果是 true，因为变量 y 是一个非常小的负值(- 0.000000000025)，它大于 - 1。第二个比较的结果是 false，因为表达式 10 - i 的值是 20，与 j 相同。第三个表达式为 true，因为 3+y 略小于 3。

可以使用关系运算符比较任何基本类型的值，现在需要一种实用的方式来使用比较的结果，以改变程序的行为。下面就介绍这些内容。

4.2 if 语句

如果给定的条件是 true，基本的 if 语句允许程序执行一个语句或一个包含在花括号中的语句块。如图 4-1 所示。

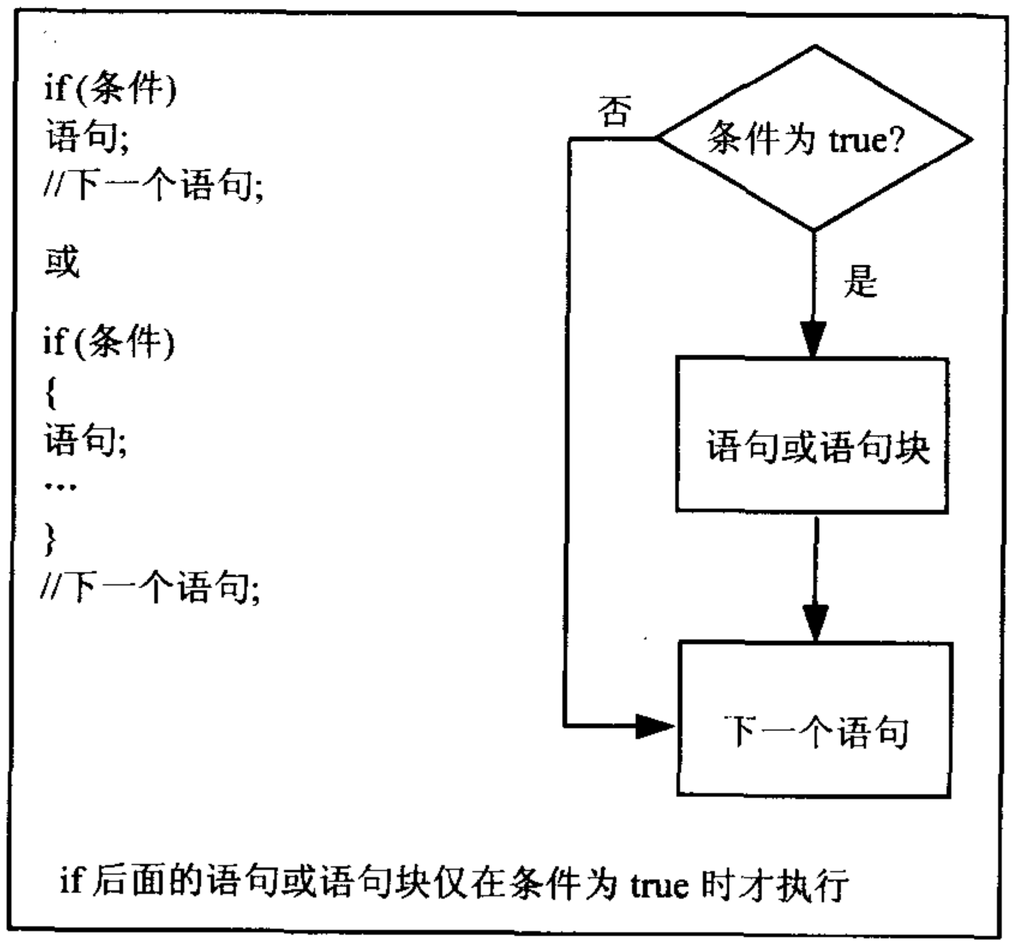


图 4-1　简单 if 语句的逻辑

下面是 if 语句的一个简单例子，它测试类型为 char 的变量 letter 的值：

```
if(letter == 'A')
  std::cout << "The first capital, alphabetically speaking.\n";

std::cout << "This statement always executes.\n";
```

如果 letter 的值是'A'，条件就为 true，这些语句就输出下面的结果：

```
The first capital, alphabetically speaking.
This statement always executes.
```

如果 letter 的值不是'A'，就只输出第二行语句。要测试的条件放在关键字 if 后面的括号中。注意分号的位置，它位于 if 和括号中的条件后面的语句之后。在括号中的条件的后面不能有分号，因为 if 和条件是和后面的语句或语句块绑定在一起的。它们本身是不能单独存在的。

if 之后的语句只有在条件为 true 时才执行。对于程序的编译来说，语句的缩进是不必要的，但这种缩进有助于理解 if 条件和依赖它的语句之间的关系。有时，简单的 if 语句还可以写在一行上：

```
if (letter == 'A') std::cout << "The first capital, alphabetically speaking.\n";
```

一般情况下，最好把语句(或语句块)和 if 条件放在不同的代码行上，这样会更清楚。

扩展这个例子，如果 letter 的值是'A'，就改变它的值；

```
if(letter == 'A') {
  std::cout << 'The first capital, alphabetically speaking.\n";
  letter = 'a';
}
std::cout << "This statement always executes.\n";
```

在 if 条件为 true 时，就执行块中的所有语句。如果没有加上花括号，则只有第一个语句是 if 块的内容，给 letter 赋予'a'的语句将总是执行。注意在块中每个语句的最后都有一个分号，而在块结束的右花括号后面没有分号。在块中可以放置任意多个语句，甚至还可以嵌套块，因为 letter 的值是'A'，所以块中的两个语句都会执行，在输出与前面相同的消息后，就把它的值改为'a'。如果条件为 false，就不执行这两个语句。当然，if 块后面的语句总是执行。

程序示例 4.2——作出决策

下面试用 if 语句。创建一个程序，检查从键盘上输入的一个整数值：

```
//Program 4.2  Using an if statement
#include <iostream>
using std::cin;
using std::cout;
using std::endl;

int main() {
  cout << " Enter an integer between 50 and 100: ";

  int value=0;
  cin>> value;

  if(value<50)
```

```
  cout << " The value is invalid - it is less than 50."<<endl;

  if(value>100)
  cout << " The value is invalid - it is greater than 100."<<endl;

  cout << " You entered"<<value<<endl;
  return 0;
}
```

输出取决于输入的值。对于值为 50～100 之间的数据，输出应如下所示：

```
Enter an integer between 50 and 100: 77
You entered 77
```

如果输入的值在 50～100 的范围之外，就给出一个消息，说明值是无效的，并显示该值。如果该值小于 50，输出应如下所示：

```
Enter an integer between 50 and 100: 27
The value is invalid - it is less than 50.
You entered 27
```

如果该值大于 100，输出应如下所示：

```
Enter an integer between 50 and 100: 270
The value is invalid - it is greater than 100.
You entered 270
```

例子的说明

在给出提示，读取一个值后，第一个 if 语句就会检查输入的值是否小于下限 50：

```
if(value<50)
   cout << " The value is invalid - it is less than 50."<<endl;
```

只有 if 条件为 true 时，也就是当 value 小于 50 时，才执行输出语句。下一个 if 语句检查上限：

```
if(value>100)
   cout <<" The value is invalid - it is greater than 100."<<endl;
```

如果 value 大于 100，就执行该输出语句。最后一个输出语句是：

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

这个语句总是执行。

嵌套的 if 语句

在 if 语句中的条件为 true 时才执行的语句本身也可以是一个 if 语句，这种情况称为嵌套的 if 语句。只有外层 if 的条件为 true 时，才测试内层 if 的条件。嵌套在一个 if 语句中的 if 语句也可以包含另一个嵌套的 if 语句。一般情况下，可以像这样继续嵌套 if 语句，嵌套的次数也可以是任意多次。

程序示例 4.3——使用嵌套的 if 语句

下面用一个工作示例来演示嵌套的 if 语句。该示例测试从键盘上输入的字符是否为字母。这个示例很好地使用了嵌套的 if 语句，但其中一些固有的假设最好避免，读者可以试着找出这些假设。下面是代码：

```
//Program 4.3  Using a nested if
#include <iostream>
using std::cin;
using std::cout;
using std::endl;

int main() {
  char letter=0;                          // Store input in here

  cout << "Enter a letter: ";             // Prompt for the input
  cin >> letter;                          // then read a character

  if(letter >= 'A') {                 // Test for 'A' or larger
    if(letter <= 'Z') {               // Test for 'Z' or smaller
     cout << "You entered an uppercase letter."
          << endl;
     return 0;
    }
  }

   if(letter >= 'a')                   //Test for 'a' or larger
    if(letter <= 'z') {               //Test for 'z' or smaller
      cout << "You entered an lowercase letter."
           << endl;
      return 0;
    }

    cout<<" You do not enter a letter."<<endl;
    return 0;
}
```

本例的输出如下所示：

```
Enter a letter: H
You entered an uppercase letter.
```

例子的说明

这个程序首先是通常的注释行和支持输入输出的头文件<iostream>的#include 语句，以及程序中 std 名称的 using 声明。在为 char 变量 letter 分配内存空间，并初始化为 0 后，函数 main() 提示输入一个字母。

之后的 if 语句检查输入的字符是否为'A'或更大：

```
if (letter >= 'A') {                  //Test for 'A' or larger
```

```
    if (letter <= 'Z') {          //Test for 'Z' or smaller
      cout << "You entered an uppercase letter."
           << endl;
      return 0;
    }
}
```

如果 letter 大于等于'A'，嵌套的 if 就检查输入的字符是否为'Z'或更小的字母。如果该字母是'Z'或更小的字母，就说明该字符是一个大写字母，于是显示一个消息，执行 return 语句，结束程序。因为这两个语句都放在花括号中，所以在嵌套的 if 条件为 true 时，这两个语句都会执行。

这对嵌套的 if 语句建立在用编码表示字母字符的两个假设的基础上。第一个假设是字母 A 到 Z 用一组编码表示，其中'A'的编码最小，'Z'的编码最大。第二个假设是大写字母的编码是连续的，在'A'编码和'Z'编码之间不存在非字母字符。在代码中建立这样的假设并不好，因为这限制了程序的可移植性。例如，在 EBCDIC 编码中，字母的字符编码是不连续的。稍后介绍如何避免这种限制。

下一个 if 语句使用了与第一个 if 语句相同的机制，检查输入的字符是否为小写，然后显示一个消息并返回。

```
if (letter >= 'a')                //Test for 'a' or larger
  if (letter <= 'z') {            //Test for 'z' or smaller
    cout << "You entered an lowercase letter."
         << endl;
    return 0;
}
```

但如果仔细检查，就会注意到小写字符的测试只包含一对花括号，而大写字母的测试包含两对花括号。这里花括号中的代码块属于内层的 if 语句。实际上，这两种方式都是对的：在 C++中，if(condition){...}是一个语句，不需要放在花括号中。同样，如果觉得使用较多的花括号能使代码更清晰，就可以使用它们。最后，与大写字母测试一样，这些代码隐含了小写字母编码的假设。

只有在输入的字符不是字母时，才执行最后一个 if 语句块后面的输出语句，它会显示一个消息，然后执行 return 语句。嵌套的 if 语句和输出语句之间的关系更容易理解，因为每个 if 语句块都进行了缩进。在 C++中，缩进格式通常用于提供程序逻辑的可视化线索。

如本例开头所述，该程序演示了嵌套的 if 语句块如何工作，但这并不是测试字符的好方法。使用标准库可以编写出独立于字符编码的程序。下面就介绍这部分内容。

不依赖编码的字符处理

标准库提供了许多函数，它们可以在程序中执行许多任务。表 4-2 列出了这些函数。在程序中包含<cctype>头文件，就可以访问一组非常有用的函数集，来测试字符。在测试字符时，需要给函数传送一个 int 类型的变量或字面量。如果传送了 char 类型的值，编译器会把它自动转换为 int。

表 4-2　测试字符的函数

函　　数	所执行的动作
isupper()	测试大写字母 A 到 Z
islower()	测试小写字母 a 到 z
isalpha()	测试大写字母或小写字母
isdigit()	测试数字 0 到 9
isxdigit()	测试十六进制数字 0 到 9、a 到 f 或 A 到 F
isalnum()	测试字母或数字(例如字母表上的字符和数字)
isspace()	测试空白，空白可以是一个空格、换行符、回车符、换页符、水平制表符或垂直制表符
iscntrl()	测试控制符
isprint()	测试可打印的字符，即大写或小写字母、数字、标点符号或空格
isgraph()	测试图形字符，即除了空格之外的所有可打印字符
ispunct()	测试标点符号，即非字母或数字的所有可打印字符，标点符号可以是空格或下面的字符：_ { } [ ] # ( ) < > % : ; . ? * + - / ^ & | ~ ! = , \ " '

这些函数都返回一个 int 类型的值。如果字符的类型与要测试的类型相同，该值就为正(true)，否则就为 0(false)。为什么这些函数不返回 bool 类型的值？这看上去似乎更有意义。原因是包含这些函数的 C 标准库是在 bool 类型引入 C++之前建立的。

<cctype>头文件还提供了两个函数，如表 4-3 所示，在大写字符和小写字符之间转换，传送给这两个函数的字符应是 int 类型，返回的结果为 int 类型：

表 4-3　转换字符的函数

函　　数	说　　明
tolower()	如果给函数传送了一个大写字母，就返回该字母的小写形式，否则就返回未修改的传送进来的字母
toupper()	如果给函数传送了一个小写字母，就返回该字母的大写形式，否则就返回未修改的传送进来的字母

可以使用这些函数实现前面的例子，而无需字符编码的任何假设。不同环境中的不同字符编码总是由标准库函数来考虑，用户不需要考虑。因为使用标准库函数后，也不需要使用嵌套的 if 语句，所以代码要比前面简单得多。

注意所有这些字符测试函数，除 isdigit()和 isxdigit()之外，都在当前环境下测试变元。本地环境确定了本地数据是如何处理的。不同的国家使用不同的字符集表示字母，所以某个字符编码是否解释为字母取决于本地环境。货币单位和小数的显示方式也随着本地环境的不同而不同。调用在<clocale>头文件中声明的 setlocale()函数可以设置本地环境。这个函数接受两个变元：第一个变元指定应用本地环境的函数的类别，第二个变元指定本地环境。用于第一个变元的值必须是在<clocale>头文件中声明的值，表 4-4 列出了这些值。

表 4-4　受本地环境影响的类别值

值	说　明
LC_ALL	指定所有的类别
LC_CTYPE	指定字符处理
LC_COLLATE	指定字符串比较中的比较顺序
LC_MONETARY	指定货币信息的格式
LC_NUMERIC	指定小数点字符
LC_TIME	指定时间值的格式

setlocale()的第二个变元是一个指定本地环境的字符串。字符串"C"是默认值，对应于拉丁字母'A'到'Z'。可以用于指定其他本地环境的字符串集是由实现方式定义的，但通常包括简单明了的国家规范，如 Germany。

C++头文件<locale>提供了许多扩展功能的声明，它们可以处理依赖本地环境的数据，在程序中需要支持多个本地环境时，就应使用它们。

程序示例 4.4——使用标准库字符转换函数

在修改上一个例子，使用标准库函数时，还可以扩展程序的功能，试用转换函数：

```
//Program 4.4  Using standard library character testing and conversion
#include <iostream>
#include <cctype>                       // Character testing and conversion
using std::cin;
using std::cout;
using std::endl;

int main() {
  char letter=0;                        // Store input in here

  cout<<endl
      << "Enter a letter: ";            // Prompt for the input
  cin>> letter;                         // then read a character
  cout<<endl;

  if (std::isupper(letter)) {           //Test for upper case letter
     cout << "You entered a capital letter."
          <<endl;
     cout << " Converting to lowercase we get "
          <<static_cast<char>( std::tolower(letter))<<endl;
     return 0;
}

if (std::islower(letter)) {                 //Test for lower case letter
  cout << "You entered a small letter."
       <<endl;
  cout << "Converting to uppercase we get "
       << static_cast<char>( std::toupper(letter)) <<endl;
```

```
    return 0;
  }

  cout<<" You id not enter a letter."<<endl;
  return 0;
}
```

该例子的输出如下所示：

```
Enter a letter: t
You entered a small letter.
Converting to uppercase we get T
```

例子的说明

if 表达式已改为使用标准库函数，不再需要嵌套的 if 语句，因为前面要测试的两个条件现在都包含在 isupper()或 islower()函数中。

我们并不关心这些函数的工作原理。要使用它们，只需知道它们完成什么任务，需要给它们传送多少参数，传送什么类型的参数，以及它们返回什么类型的值。有了这些信息，就可以使用标准库函数，使代码更简单、更一般化。程序的这个版本可以处理使用任何字符编码的 char 类型。

注意，在输出语句中直接使用从转换函数中返回的结果，如下所示：

```
cout << "Converting to upper case we get "
     << static_cast<char>(std::toupper(letter)) <<endl;
```

由于 toupper()函数返回的值是 int 类型，因此把它强制转换为 char 类型，并发送给输出流 cout。如果要存储返回的字符，而不是进行显式强制转换，就可以将它存储在原来的变量 letter 中，如下面的语句所示：

```
letter =std::toupper(letter);
```

接着在输出语句中使用变量 letter 输出转换后的字符：

```
cout << "Converting to uppercase we get "<<letter <<endl;
```

如果需要使用多字节字符(其类型是 wchar_t)，就可以包含头文件<cwctype>。该文件包含了在<cctype>中声明的所有函数的对应多字节字符。每个测试函数名都在 is 的后面加上了 w，所以它们的名称就变成：

```
iswupper()      iswdigit()      iswspace()      iswgraph()
iswlower()      iswxdigit()     iswcntrl()      iswpunct()
iswalpha()      iswalnum()      iswprint()
```

它们都传递多字节字符参数，并且返回一个 int 值，就像处理 char 类型的字符函数一样。同样，多字节字符转换函数也称为 towupper()和 towlower()。

注意：

C++中的<cctype>和<cwtype>头文件继承于 C。在许多实现方式中，函数在 std 命名空间内部和外部都定义了，以允许旧式 C 程序编译和链接。此时，函数名无论是否带 std 限定符都能工作，但因为我们编写的是 C++程序，所以应限定名称。

4.3 if-else 语句

前面使用的 if 语句在指定的条件为 true 时执行一个语句。接着，程序按顺序执行下一个语句。当然，有时我们希望只有在条件为 false 时才执行某个语句或语句块。为此，就要扩展 if 语句，允许在条件为 true 时执行一组动作，在条件为 false 时执行另一组动作。之后，程序按顺序执行下一个语句。这描述为 if-else 语句。

if-else 组合提供了两个选项供选择，其一般逻辑如图 4-2 所示。

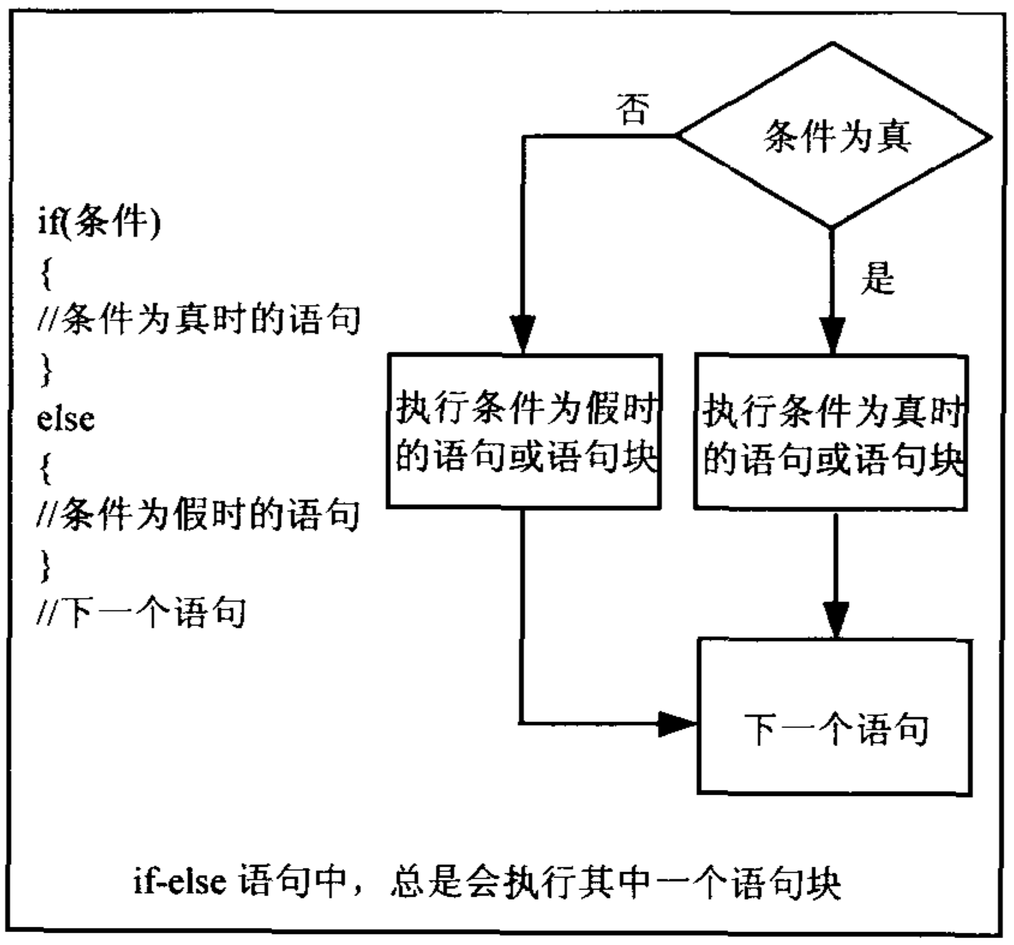


图 4-2　if-else 语句的逻辑

图 4-2 中的流程图指出了语句的执行顺序取决于 if 条件为 true 还是 false。如图所示，在可以使用语句的地方，总是可以使用一个语句块来代替。这就允许为 if-else 语句的每个选项执行任意多条语句。

仍然使用 char 类型的变量，编写一个 if-else 语句，报告存储在变量 letter 中的字符是否为字母或数字。

```
if(std::isalnum(letter))
  std::cout << "It is a letter or a digit." << std::endl;
else
  std::cout << "It is neither a letter nor a digit." << std::endl;
```

这个程序使用了<cctype>头文件中的函数 isalnum()。如果变量 letter 包含字母或数字，函数 isalnum()就返回一个正整数。因为 if 语句把这个看作是 true，所以显示第一个消息。如果变量 letter 包含的不是字母或数字，函数 isalnum()就返回 0。对于 if 来说，这会自动转换为 false，执行 else 之后的输出语句。

程序示例 4.5——扩展 if 语句

下面用一个例子来演示 if-else 语句，这次测试的是数值：

```
//Program 4.5 Using the if-else
```

```
#include <iostream>
using std::cin;
using std::cout;
using std::endl;

int main() {
  long number=0;                        //Store input here
  cout << "Enter an integer less than 2 billion: ";
  cin >> number ;
  cout << endl;

  if (number % 2L ==0)                  //Test remainder after division by 2
    cout << " \nYour number is even."     //Here if remainder is 0
         << endl;
  else
    cout << " \nYour number is odd."    //Here if remainder is 1
         << endl;
  return 0;
}
```

这个程序的输出如下所示：

```
Enter an integer less than 2 billion: 123456

Your number is even.
```

例子的说明

在把输入的值读入 number 之后，就在 if 条件中测试该数除以 2 的余数(使用第 2 章介绍的取余运算符%)，检查它是否为 0。整数除以 2 后的余数只能是 1 或 0，程序中的代码进行了注释。如果余数等于 0，if 条件就是 true，执行 if 之后的语句。如果余数等于 1，则 if 条件为 false，就执行 else 关键字后面的语句。在输出结果后，执行 return 语句，结束程序。

注释：

else 关键字的后面没有分号，这与语句中的 if 一样。这里也采用了缩进格式，作为各个语句之间关系的可视化指示符。可以清楚地看出哪个语句在得到 true 时执行，哪个语句在得到 false 时执行。在程序中，应总是缩进语句，显示它们的逻辑结构。

在这个例子中，演示了编写 if 条件的另一种方式。在转换为 bool 类型时，任何非 0 值都是 true，而 0 值转换为 false。所以，可以把模式化操作的结果用作条件，而不需要比较它和 0 值。这样，if-else 语句就变成：

```
if(number % 2L)                         //Test remainder after division by 2
   cout << " Your number is odd."     //Here if remainder is 1
        << endl;
else
   cout << " Your number is even."    //Here if remainder is 0
        << endl;
```

if 和 else 子句需要调换，因为如果 number 的值是偶数，则(number % 2L == 0)就返回 true，而(number % 2L)会转换为 false。初看起来这似乎有点让人迷惑，但这个条件的第一个版本如下：

“余数为 0 是 true 吗？”

由于 1 会转换为 true，因此，该条件的第二个版本应如下：

“余数是 1 吗？”

嵌套的 if-else 语句

前面介绍了如何在 if 语句中嵌套 if 语句。显然，也可以在 if 语句中嵌套 if-else 语句，在 if-else 语句中嵌套 if 语句，在 if-else 语句中嵌套其他 if-else 语句，这样就提供了极大的灵活性(同时也很容易出现混淆)，下面就举几个例子。先看第一种情况，在 if 语句中嵌套 if-else 语句：

```
if(coffee=='y')
  if(donuts=='y')
  std::cout << " We have coffee and donuts ."
            << std::endl;
else
    std::cout  << " We have coffee, but not donuts."
               << std::endl;
```

其中，coffee 和 donuts 是 char 类型的变量，其值分别是'y'和'n'。由于对 donuts 的测试仅在 coffee 测试的结果为 true 时才进行，因此在每种情况下，消息都会反映正确的情形。else 属于 donuts 测试中的 if 语句。这很容易引起混淆。

如果编写这些代码时，缩进格式有错误，就会推导出错误的结论：

```
if(coffee=='y')
  if(donuts=='y')
    std::cout << std::endl
              << "We have coffee and donuts .";
else                                          //This else is indented incorrectly
    std::cout << " We have no coffee…"        //Wrong!
              << std::endl;
```

代码的缩进让人错误地认为 if 语句嵌套在 if-else 语句中，实际上并非如此。第一个消息是正确的，但执行 else 后的结果就是错误的。这个语句仅在对 coffee 的测试为 true 时才执行，因为 else 属于 donuts 的测试，不属于 coffee 的测试。这个错误很容易看出，但 if 结构越大就越复杂，就越需要弄清楚哪个 if 拥有哪个 else。

注意：

else 总是属于前面最接近的那个 if(只要另一个 else 还不属于这个 if)。这种混淆称为 else 悬挂问题。

在程序中，只要一组 if-else 语句看起来有些复杂，就可以应用这个规则，对该组语句排序。在编写程序时，应总是使用花括号，使代码更清晰。在这样简单的情形中不需要使用花括号，但可以把上一个例子改写为：

```
if(coffee=='y') {
  if(donuts=='y')
```

```
    std::cout << "We have coffee and donuts ."
              << std::endl;
else
    std::cout << " We have coffee, but not donuts."
              << std::endl;
}
```

现在代码非常清晰了，else 肯定属于测试 donuts 的 if 语句。

理解嵌套的 if 语句

知道了规则后，理解 if 语句嵌套在 if-else 语句的情形就比较容易了：

```
if(coffee=='y') {
  if(donuts=='y')
    std::cout << " We have coffee and donuts ."
              << std::endl;
}
else if(tea=='y')
    std::cout << " We have no coffee, but we have tea"
              << std::endl;
```

提示：

注意这里的代码格式。一个 if 语句嵌套在 else 的下面，可以把 else 和 if 写在一行上，本书后面的例子就是这样编写的。

这次，花括号是必不可少的。如果省略了花括号，else 就属于测试 donuts 的 if 语句了。在这种情况下，很容易忘记加上花括号，生成一个很难找出的错误。有这种错误的程序也会编译，因为代码是完全正确的。有时甚至结果也是正确的，但它没有表达出我们真正的意图。

如果在这个例子中删除花括号，则只要 coffee 和 donuts 都等于'y'，就不会执行 if(tea == 'y') 检查，从而得到正确的结果。

最后，看看一个 if-else 语句嵌套在另一个 if-else 语句中的情况。即使只有一层嵌套，也可能非常混乱。最好对 coffee 和 donuts 进行彻底的分析，再开始使用这种嵌套：

```
if(coffee=='y')
  if(donuts=='y')
    std::cout<< " We have coffee and donuts ."
             << std::endl;
else
    std::cout<< " We have coffee, but not donuts."
             << std::endl;
else if(tea=='y')
    std::cout<< " We have no coffee, but we have tea, and maybe donuts…."
             << std::endl;
else
    std::cout<< "No tea or coffee, but maybe donuts…"
             << std::endl;
```

即使采用了正确的缩进格式，这里的逻辑看起来也不是很明显。不需要使用花括号，因为前面的规则已校验，但如果加上花括号，看起来会更清楚一些：

```
if(coffee=='y') {
  if(donuts=='y')
    std::cout<< " We have coffee and donuts ."
             << std::endl;
else
    std::cout << " We have coffee, but not donuts"
              << std::endl;
}
else
{
if(tea=='y')
    std::cout<< " We have no coffee, but we have tea, and maybe donuts…."
             << std::endl;
else
    std::cout<< " No tea or coffee, but maybe donuts…"
             << std::endl;
}
```

在程序中处理这种逻辑还有更好的方式。如果把足够多的嵌套 if 语句放在一起，肯定会出错。下一节将简化这个过程。

4.4 逻辑运算符

如前所述，在有两个或更多相关条件的地方使用 if 语句有点烦琐。上一个例子把精力放在寻找 coffee 和 donuts 上，但在实际中可能要检查更复杂的条件，例如搜索年龄在 21～35 岁之间、拥有大学学历、未婚、说印地语或乌尔都语的女性的个人文件。定义这样一个测试将涉及到许多 if 语句。

C++的逻辑运算符提供了一种简洁而简单的解决方案。使用逻辑运算符，可以把一系列比较组合到一个表达式中，这样，无论条件多么复杂，都只需要一个 if。而且，逻辑运算符只有三个，如表 4-5 所示：

表 4-5　逻辑运算符

运　算　符	作　　用
&&	逻辑与
||	逻辑或
!	逻辑非

前两个&&和||是二元运算符，它组合了类型为 bool 的两个操作数，生成 bool 类型的结果。第三个运算符！是一元运算符，它应用于一个 bool 类型的操作数，并生成 bool 结果。下面首先从一般意义上介绍这些运算符的应用，接着举一个例子。把逻辑运算符和前面介绍的按位运算符区分开来是非常重要的，按位运算符处理的是整数操作数中的位，而逻辑运算符处理的是 bool 类型的操作数。

4.4.1 逻辑与运算符

如果当两个条件必须都是 true 时结果才是 true，就可以使用逻辑与运算符&&。前面在使用嵌套的 if 语句判断字符是否为大写字母，要测试的值就必须大于等于'A'，且小于等于'Z'。这两个条件都必须是 true，该字符才是大写字母。&&运算符只有在两个操作数都是 true 的情况下才会生成 true，在其他情况下，结果都是 false。

例如，有一个值存储在 char 类型的变量 letter 中，下面用一个 if 语句和&&运算符来替代前面的两个 if 语句：

```
if(letter >= 'A' && letter <= 'Z')
   std::cout << "This is an uppercase letter."
             << std::endl;
```

只有用运算符&&组合在一起的两个条件都为 true 时，才执行输出语句。在表达式中，不需要使用括号，因为比较运算符的优先级高于&&。与往常一样，也可以加上括号，把语句改写为：

```
if((letter >= 'A') && (letter <= 'Z'))
  std::cout << "This is an uppercase letter."
            << std::endl;
```

现在，括号中的比较操作肯定先执行。

4.4.2 逻辑或运算符

如果两个条件中的一个为 true 或两个都是 true 时结果为 true，就可以使用逻辑或运算符||。只有||运算符的两个操作数都是 false 时，其结果才是 false。在其他情况下，结果都是 true。

例如，如果收入至少是每年 10 万美元，或有 10 万美元的现金，肯定可以从银行贷到款。这可以使用下面的 if 来测试：

```
if(income >= 100000.00  ||  capital >= 1000000.00)
  std::cout << "Of course, how much do you want to borrow?"
            << std::endl;
```

在一个条件或两个条件是 true 时就会出现响应(更好的响应是“为什么贷款？”。奇怪，为什么银行总是在客户不需要钱时给客户发放贷款)。

4.4.3 逻辑非运算符

逻辑非运算符！接受一个其逻辑值为 true 或 false 的操作数，并反转该值。如果布尔变量 test 的值是 true，则！test 就是 false；如果 test 的值是 false，则！test 就是 true。例如，如果 x 的值是 10，则下面的表达式：

```
!(x > 5)
```

就是 false，因为 x > 5 是 true。还可以在 Charles Dickens 表达式中应用！运算符：

```
!(income > expenditure)
```

如果这个表达式是 true，结果就是不幸的——至少银行开始退您的支票了。

所有的逻辑运算符都可以应用于等于 true 或 false 的表达式。操作数可以是各种内容，如 bool 类型的简单变量，或比较和逻辑变量的复杂组合。

程序示例 4.6——组合使用逻辑运算符

可以把条件表达式和逻辑运算符组合在一起，例如，可以构建一个调查问卷，确定一个人是否有贷款风险。下面为此编写一个程序：

```
//Program 4.6 Combining logical operators
#include <iostream>
using std::cin;
using std::cout;
using std::endl;

int main() {
  int age=0;                    //Age of the prospective borrower
  int income=0;                 //Income of the prospective borrower
  int balance=0;                //Current bank balance
//Get the basic data
  cout<<endl<<"Please enter your age in years: ";
  cin>>age;

  cout<<"Please enter your annual income in dollars: ";
  cin>>income;

  cout<<"What is your current account balance in dollars: ";
  cin>>balance;
  cout<<endl;

  //We only lend to people over 21, who make
  //over $25,000 per year, or have over
  // $100,000 in their account, or both.

  if(age >= 21 && (income > 25000 || balance > 100000)) {
    //OK, you are good for the loan - but how much?
    //This will be the lesser of twice income and half balance

    int loan=0;              //Stores maximum loan amout
    if(2*income<balance/2)
      loan=2*income;
    else
      loan=balance/2;

    cout << "You can borrow up to $"
         << loan
         << endl;
  }
  else
    cout << "Sorry, we are out of cash today."
```

```
        << endl;
  return 0;
}
```

这个程序的输出如下所示：

```
Please enter your age in years: 25
Please enter your annual income in dollars: 28000
What is your current account balance in dollars:185000

You can borrow up to $56000
```

例子的说明

首先声明 3 个整型变量，用于存储通过键盘输入的值：

```
int age=0;                  //Age of the prospective borrower
int income=0;               //Income of the prospective borrower
int balance=0;              //Current bank balance
```

接着读取这 3 个值，确定贷款是否符合条件：

```
cout<<endl<<"Please enter your age in years: ";
cin>>age;

cout<<"Please enter your annual income in dollars: ";
cin>>income;

cout<<"What is your current account balance in dollars: ";
cin>>balance;
```

其后的 if 语句确定是否发放贷款：

```
if(age >= 21 && (income > 25000 || balance > 100000))
```

该条件要求申请人的年龄至少为 21 岁，收入高于 2.5 万美元或存款高于 10 万美元。表达式(income > 25000 | | balance > 100000)外部的括号是必须的，这样对收入和存款条件执行逻辑或操作的结果才会和年龄测试进行逻辑与操作。没有括号，年龄测试就会和收入测试进行逻辑与操作，其结果再和存款测试进行逻辑或操作。这是因为&&运算符的优先级高于||运算符，如附录 D 中的表所示。没有括号，即使申请人只有 8 岁，只要他的存款超过了 10 万美元，就可以获得贷款。这不是我们希望的。银行是不会给未成年人发放贷款的。

如果 if 条件是 true，就执行下面的语句块：

```
//OK, you are good for the loan - but how much?
//This will be the lesser of twice income and half balance

int loan=0;             //Stores maximum loan amount
if(2*income<balance/2)
  loan=2*income;
else
  loan=balance/2;

cout << "You can borrow up to $"
```

```
        << loan
        << endl;
}
```

在这个语句块中，用另一个 if 语句来确定贷款的最高额度。贷款的最高额度要少于收入的两倍，或者是当前银行存储额的一半。

当然，如果第一个 if 条件为 false，就执行 else：

```
else
   cout << "Sorry, we are out of cash today."
        << endl;
```

这个语句输出了一个消息，指出现金不足。

4.5 条件运算符

条件运算符有时也称为三元运算符，因为它涉及到三个操作数，这是惟一的一个三元运算符。它类似于 if-else 语句，根据条件的值选择两个选项中的一个。但是，if-else 语句可以选择执行两个语句中的一个语句，而条件运算符是选择两个值中的一个。这最好用一个例子来说明。

假定有两个变量 a 和 b，要把这两个变量值中较大的那个值赋予第三个变量 c。可以使用下面的语句：

```
c = a > b ? a : b;          //Set c to the higher of a and b
```

条件运算符把一个逻辑表达式作为其第一个操作数，在本例中就是 a > b。如果这个表达式是 true，就把第二个操作数(在本例中是 a)选择为该操作的返回值。如果第一个操作数是 false，就把第三个操作数(在本例中是 b)选择为该操作的返回值。因此，如果 a 大于 b，条件表达式的结果就是 a，否则就是 b。在上面的语句中，这个值将存储在 c 中。在赋值语句中使用条件运算符就相当于下面的 if 语句：

```
if(a>b)
  c=a;
else
  c=b;
```

条件运算符也可以用于选择两个值中较小的那个。在上面的程序中，使用 if-else 语句确定贷款额，也可以使用下面的语句：

```
loan = 2*income < balance/2 ? 2*income : balance/2;
```

这会得到相同的结果，且不需要使用括号，因为条件运算符的优先级低于该语句中的其他运算符。条件是 2*income < balance/2，如果它等于 true，就计算表达式 2*income，输出该操作的结果。如果条件是 false，表达式 balance/2 的值就是该操作的结果。

当然，如果觉得使用括号会使代码更清晰，就可以加上括号：

```
loan = (2*income < balance/2) ? (2*income) : (balance/2);
```

条件运算符通常用?:表示，可以写为：

```
条件 ? 表达式 1 : 表达式 2
```

如果条件等于 true，结果就是表达式 1 的值；如果条件等于 false，结果就是表达式 2 的值。如果条件等于一个数值，就会自动转换为 bool 类型，非 0 值转换为 true，0 转换为 false。

注意只执行表达式 1 和表达式 2 中的一个，例如，下面的表达式：

```
a < b ? ++i +i :i+1;
```

如果 a 小于 b，就递增 i，条件表达式的结果是 i 递增后加 1。如果 a<b 为 false，就不递增 i，条件运算符的结果是 i 的当前值加 1。

程序示例 4.7——在输出中使用条件运算符

条件运算符的一个常见用法是根据表达式的结果或变量的值控制输出。可以根据指定的条件选择要输出的文本字符串。

```
//Program 4.7 Using the conditional operator to select output.
#include <iostream>
using std::cin;
using std::cout;
using std::endl;

int main() {
  int mice=0;                  //Count of all mice
  int brown=0;                 //Count of brown mice
  int white=0;                 //Count of white mice

  cout <<" How many brown mice do you have?";
  cin >> brown;

  cout << " How many white mice do you have?";
  cin >> white;

  mice = brown+ white;

  cout << " You have"
       << mice
       << (mice ==1 ? ""mouse" : "mice")
       << " in total."
       << endl;
  return 0;
}
```

这个程序的输出如下所示：

```
How many brown mice do you have? 2
How many white mice do you have? 3
You have 5 mice in total.
```

例子的说明

这个例子中的大多数操作以前都介绍过。惟一有趣的地方是在输入老鼠数后执行的输出语句：

```
cout << " You have"
     << mice
     << (mice ==1 ? ""mouse" : "mice")
     << " in total."
     << endl;
```

如果变量 mice 的值是 1，使用条件运算符的表达式就等于"mouse"，否则就等于"mice"。这样就可以使用同一个输出语句输出任意数量的老鼠，按需要选择单数或复数形式。

这种机制还可以应用于许多其他情形，例如选择 is 和 are 或选择 he 和 she 等有两个选项的情形。甚至可以把两个条件运算符组合在一起，选择三个选项。例如：

```
cout << (a<b ? "a is less than b.":
(a==b ? "a is equal to b.":" a is greater than b."));
```

这个语句将根据变量 a 和 b 的值输出三个消息中的一个。第一个条件运算符的第二个选项是另一个条件运算符的结果。

4.6 switch 语句

我们常常面临多项选择的情形，在这种情况下，需要根据整数变量或表达式的值，从许多选项(多于两个)中确定执行哪个语句集。例如抽奖，顾客购买了一张有号码的彩票，如果运气好，就会赢得大奖。例如，如果彩票的号码是 147，就会赢得头等奖。如果彩票的号码是 387，就会赢得二等奖。如果彩票的号码是 29，就会赢得三等奖。其他号码则不能获奖。处理这类情形的语句称为 switch 语句。

switch 语句允许根据给定表达式的一组固定值，从多个选项中选择。这些选项称为 case。在彩票例子中，有 4 个 case，每个 case 对应于一个获奖号码，再加上一个默认的 case，用于所有未获奖的号码。下面就编写一个 switch 语句，为给定的彩票号码选择消息：

```
switch(ticket_number) {
  case 147:
    std::cout<<"You win first prize!";
    break;
  case 387:
    std::cout<<"You win second prize!";
    break;
  case 29:
    std::cout<<"You win third prize!";
    break;
  default:
    std::cout<<"Sorry, you lose.";
}
```

switch 语句描述起来比其使用难一些。在许多 case 中选择取决于关键字 switch 后面括号中整数表达式的值。选择表达式的结果也可以是已枚举的数据类型，因为这种类型的值可以自动转换为整数。在本例中，它就是变量 ticket_number，它必须是整数类型。

可以根据需要，使用多个 case 值定义 switch 语句中的可能选项。case 值显示在 case 标签中，其形式如下所示：

```
case case_value:
```

称之为 case 标签，是因为它标注了后面的语句。如果选择表达式的值等于 case 值，就执行该 case 标签后面的语句。每个 case 值都必须是惟一的，但不必按一定的顺序，如本例所示。

case 值必须是整数常量表达式，即编译器可以计算的表达式，所以它只能使用字面量、const 变量或枚举成员。而且，所包含的所有字面量都必须是整数类型或可以强制转换为整数类型。

例子中的 default 标签标识默认的 case，它是一个否则模式。如果选择表达式不对应于任何一个 case 值，就执行该默认 case 后面的语句。但是，不一定要指定默认 case，如果没有指定它，且没有选中任何 case 值，switch 语句就什么也不做。

从逻辑上看，每一个 case 语句后面的 break 语句是绝对必须的，它在 case 语句执行后跳出 switch 语句，使程序继续执行 switch 右花括号后面的语句。如果省略了 case 后面的 break 语句，就将执行该 case 后面的所有语句。注意在最后一个 case 后面(通常是默认 case)不需要 break 语句，因为此时程序将退出 switch 语句，但加上 break 是一个很好的编程习惯，因为这可以避免以后添加另一个 case 而导致的问题。

提示：

switch、case、default 和 break 都是关键字。

程序示例 4.8——switch 语句

下面的例子演示了 switch 语句的用法：

```
//Program 4.8 Using the switch statement
#include <iostream>
using std::cin;
using std::cout;
using std::endl;

int main() {
  int choice=0;           //Store selection value here

  cout << endl
       << "Your electronic recipe book is at your service."<<endl
       << "You can choose from the following delicious dishes: "
       << endl
       << "1 Boiled eggs"<<endl
       << "2 Fired eggs"<<endl
       << "3 Scrambled eggs"<<endl
       << "4 Coddled eggs"<<endl
       << endl<<"Enter your selection number: ";
cin >> choice;

switch(choice) {
  case 1:
    cout<<endl<<"Boil some eggs."<<endl;
    break;
  case 2:
    cout<<endl<<"Fry some eggs."<<endl;
    break;
```

```
    case 3:
      cout<<endl<<" Scramble some eggs."<<endl;
      break;
  case 4:
    cout<<endl<<" Coddle some eggs."<<endl;
    break;
  default:
    cout<<endl<<"You entered a wrong number, try raw eggs."<<endl;
  }
  return 0;
}
```

例子的说明

在输出语句中定义选项，将选中的数字读入变量 choice 后，就执行 switch 语句，该语句在关键字 switch 的后面，把选择表达式指定为括号中的 choice。switch 语句中的可能选项放在花括号中，每个选项都用一个 case 标签来标识。如果 choice 的值对应于某个 case 值，就执行该 case 标签后面的语句。在本例中，每个 case 只有一个语句和 break 语句，但一般情况下，case 标签后面可以有许多语句，且不需要把它们括在花括号中。

每组 case 语句后面的 break 语句把执行权传送给 switch 后面的语句。break 语句不是强制的，但如果不加上它，就会执行所选 case 之后的所有语句，这通常不是我们希望的操作。把本例中的 break 语句删除，看看会发生什么。

如果 choice 的值不对应于所指定的所有 case 值，就执行 default 标签后面的语句。如果没有包括 default case，且 choice 的值不等于所有的 case 值，则 switch 语句就什么也不做，程序继续执行 switch 后面的下一条语句，即 return 语句。

程序示例 4.9——共享 case

前面说过，每个 case 值都必须是编译时常量，且必须是惟一的。任何两个 case 值都不能相同的原因是，如果输入了某个指定值，编译器就无法确定应执行哪些语句。但是，case 值不同，并不表示必须执行不同的操作。几个 case 值可以共享相同的操作，如下面的例子所示：

```
//Program 4.9 Multiple case actions
#include <iostream>
#include <cctype>
using std::cin;
using std::cout;
using std::endl;

int main() {
  char letter=0;
  cout << endl
       << " Enter a letter: ";
  cin >> letter;

  if ( std::isalpha(letter))
   switch(std::tolower(letter)) {
     case 'a':  case 'e':  case 'i':  case 'o':  case 'u':
       cout<<endl<<" You entered a vowel."<<endl;
       break;
```

```
    default:
      cout<<endl<<" You entered a consonant."<<endl;
    }
  else
    cout<<endl<<"You did not enter a letter."<<endl;

  return 0;
}
```

这个程序的输出如下所示：

```
Enter a letter: E
You entered a vowel.
```

例子的说明

在这个例子中，使用了标准库的一个字符转换例程和 switch 语句，来确定输入的字符是元音还是辅音。if 条件首先检查是否输入了一个字母，而不是其他字符：

```
if(std::isalpha(letter))
```

如果 isalpha()返回的值是非 0 值，就执行 switch 语句：

```
  switch(tolower(letter)) {
     case 'a':  case 'e':  case 'i':  case 'o':  case 'u':
       cout<<endl<<" You entered a vowel."<<endl;
       break;
     default:
       cout<<endl<<" You entered a consonant."<<endl;
  }
```

switch 由 tolower()函数的返回值控制。可以只使用变量 letter，接着需要把所有的大写元音字母指定为 case 值，也可以把所有的小写元音字母指定为 case 值。如果 tolower()函数返回的值对应于一个元音，就显示确认消息。否则就执行默认的 case，显示消息“输入的字符是一个辅音”。

如果 isalpha()返回 0，就不执行 switch 语句，而执行 else 语句，输出消息“输入的字符不是字母”。

可以利用 if 语句把字母的测试和转换为小写形式这两个操作组合起来，但这需要一些技巧，且会使代码变得比较复杂。例如，可以把 switch 语句改写为：

```
switch(std::tolower(letter)*(std::isalpha(letter) !=0)) {
  case 'a':  case 'e':  case 'i':  case 'o':  case 'u':
    cout<<endl<<" You entered a vowel."<<endl;
    break;
  case 0:
    cout<<endl<<"You did not enter a letter."<<endl;
    break;
  default:
    cout<<endl<<" You entered a consonant."<<endl;
}
```

但如前所述，如果给 isalpha()函数传送非字母字符，它就会返回整数 0；如果给它传送字母，它就会返回一个正整数，但这个正整数不一定是 1。选择表达式变复杂的原因是，isalpha()函数不会生成 bool 值。如果可以生成，就可以使用 tolower(letter)*isalpha(letter)，当 isalpha()返回 false 时，这个表达式等于 0，否则就等于 tolower()返回的小写字母，因为 true 会转换为 1。

另一个方法是把 isalpha()返回的值强制转换为 bool 类型。接着，就可以把 switch 语句改写为：

```
switch(tolower(letter)*static_cast<bool>(isalpha(letter))) {
case 'a':  case 'e':  case 'i':  case 'o':  case 'u':
  cout<<endl<<" You entered a vowel."<<endl;
  break;
case 0:
  cout<<endl<<"You did not enter a letter."<<endl;
  break;
default:
  cout<<endl<<" You entered a consonant."<<endl;
}
```

这段代码可以正常执行，因为 isalpha()返回的整数被强制转换为 bool，编译器会把这个值转换为 int，进行乘法运算，所以它最终是 0 或 1。但是，switch 语句会变得混乱。使用 if 的原始版本肯定是代码最简洁的一个版本，因此是首选的版本，尽管其逻辑不是很好。

4.7 无条件分支

if 语句允许根据指定的条件，选择执行某个语句块或另一个语句块，这样语句的执行顺序就会根据程序中的数据值而变化。刚才介绍的 switch 语句允许根据整数表达式的值从一组范围固定的选项中选择。而 goto 语句是一个很生硬的指令，它允许无条件地分支指定的程序语句。要进行分支的语句必须用语句标签来标识，语句标签是一个根据与变量名相同规则定义的标识符。其后跟一个冒号，放在要使用该标签引用的语句之前。下面是一个带标签的语句示例：

```
MyLabel: x=1;
```

这个语句的标签是 MyLabel，无条件地分支该语句的语句如下所示：

```
goto MyLabel;
```

在程序中，应尽可能避免使用 goto 语句。goto 语句会使代码极难理解。注意如果 goto 语句位于变量的作用域中，但绕过了变量的声明，编译器就会发出一个错误消息。

注释：

goto 语句在理论上是不必要的——总是有另外一种方式可以替代 goto 语句。大多数程序员都从来不使用它。作者不赞成这种极端的态度，毕竟它是一个合法的语句，有时使用它是很方便的。但是，应仅在与其他可用选项相比，使用它的优势非常明显时才使用它。

4.8 决策语句块和变量作用域

switch 语句一般在花括号中包含了自己的语句块，其中包括 case 语句。if 语句也常常在花括号中包含条件为 true 时执行的语句，其 else 部分也可以包含花括号。注意这些语句块与定义变量作用域时涉及到的其他语句块没有任何区别。因为在块中声明的任何变量都会在该块结束时自动消失，所以不能在块外引用它们。

例如，考虑下面这个很随意的计算：

```
if (value>0) {
  int savit=value - 1;     //This only exists in this block
  value += 10;
}
else {
  int savit=value+1;       //This only exists in this block
  value -= 10;
}

std::cout<<savit;          //This will not compile! savit does not exist
```

最后的输出语句会得到一个错误消息，因为此时变量 savit 未定义。在块中定义的任何变量都只能在该块中使用，如果要在块的外部访问块内的数据，就必须把存储该信息的变量声明放在块的外面。

注意 switch 语句块中的声明必须能在执行该语句块时访问，该声明不能被绕过，否则代码就不编译。下面的代码演示了 switch 块中的非法声明：

```
int test=3;
switch(test){
  int i=1;              //ILLEGAL-cannot be reached

  case 1:
  {
    int j=2;            //OK-can be reached and is not bypassed
    cout<<endl<<text+j;
    break;
  }

  int k=3;              //ILLEGAL-cannot be reached

  case 3:
    cout<<endl<<test;
    int m=4;            //ILLEGAL-can be reached but can be bypassed
    break;

  default:
    cout<<endl<<"Default reached.";
    break;

  int n=5;              //ILLEGAL-cannot be reached
}
```

在这个 switch 语句中，只有一个声明(即 j 的声明)是合法的。首先，合法的声明必须可以在该语句块的正常执行过程中访问，而变量 i、k 和 n 的声明就不是这样。其次，不能绕过变量的声明来进入该变量的作用域，变量 m 就是这样。而变量 j 是惟一一个位于其声明到块结尾之间的变量，所以这个声明不能被绕过。

4.9 本章小结

本章给程序添加了决策功能，介绍了 C++中的所有决策语句的工作原理。决策语句的主要元素包括：

- 可以使用比较运算符比较两个值，得到一个 bool 类型的值，它可以是 true 或 false。
- 可以把 bool 值强制转换为整数类型——true 强制转换为 1，false 强制转换为 0。
- 可以把数值强制转换为 bool 类型——0 强制转换为 false，非 0 值强制转换为 true。
- if 语句可以根据条件表达式的值执行一个语句或语句块。如果条件是 true 或非 0 值，就执行语句或语句块。如果条件是 false，或 0，就不执行。
- if-else 语句给简单的 if 语句提供了另一个选项。如果条件为 false 或 0，就执行 else 语句。
- if 和 if-else 语句可以嵌套。
- switch 语句可以根据整数表达式的值，从一组固定的选项中选择。
- 条件运算符根据一个表达式的值，选择两个值中的一个。
- 使用 goto 语句，可以无条件地分支带有指定标签的语句。

4.10 练习

1. 编写一个程序，提示用户输入两个正整数，然后输出一个消息，说明这两个整数是否相等。

2. 创建一个程序，提示用户输入一个 1 到 100 之间的整数。使用嵌套的 if 语句判断该整数是否在设定的范围之内。如果是，再判断该整数是否大于、小于或等于 50。

3. 编写一个程序，接受用户输入的一个字符。使用标准库函数判断它是否为一个元音字母，是否为小写字母。最后，输出小写字母，再把其字符编码输出为一个二进制值。

4. 编写一个程序，仅使用条件运算符确定输入的整数是否是 20 或小于 20；大于 20 且不大小 30；大于 30 但不超过 100；或者大于 100。

5. 编写一个程序，提示用户输入 0 美元和 10 美元之间的一个钱款(允许使用小数)。判断该钱款包含多少个 25 美分、10 美分、5 美分和 1 美分，并把该信息输出到屏幕上。另外，输出结果在语法上应是有意义的(例如，如果只需要一个 1 角，输出就应写为 1dime，而不是 1 dimes)。

第5章 循 环

循环是编程中的另一个基本要素。它允许对一个或多个语句重复执行应用程序所需要的次数。利用循环可以处理重复的工作，对于大多数重要的程序来说，循环都是必不可少的。例如，用计算机计算公司员工的薪水，没有循环是不可能完成的。

C++提供了许多方式来实现循环。这些方式在应用程序中都有特定的用途。本章首先介绍循环的理论，再详细讨论它们的实际应用，当然包括如何编写它们。

本章主要内容

- 不同类型循环的规则
- while 循环的工作原理
- do-while 循环的工作原理
- 如何使用 for 循环
- 循环中的 break 语句
- 循环中的 continue 语句
- 如何构建嵌套的循环

5.1 理解循环

循环是一种机制，它允许重复执行同一个系列的语句，直到满足指定的条件为止。循环中的语句有时称为迭代语句。对循环中的语句块或语句执行一次称为迭代。

循环有两个基本元素：组成循环体的、要重复执行的语句或语句块以及决定何时停止重复循环的循环条件。

循环条件有许多不同的形式，提供了控制循环的不同方式。例如：

- 执行循环指定的次数
- 循环一直执行到给定的值超过另一个值为止
- 循环一直执行到从键盘上输入某个字符为止

可以设置循环条件，以适应使用循环的环境。但循环最终可以分为两种基本形式，如图 5-1 所示。

这两种结构之间的区别是很明显的。在左边的结构中，循环条件在执行循环语句之前测试，因此，如果循环条件测试失败，则循环语句根本就不执行。

在右边的结构中，循环条件是在执行循环语句之后测试。其结果是在第一次测试循环条件之前，就执行了循环语句，所以这种循环至少要执行一次。

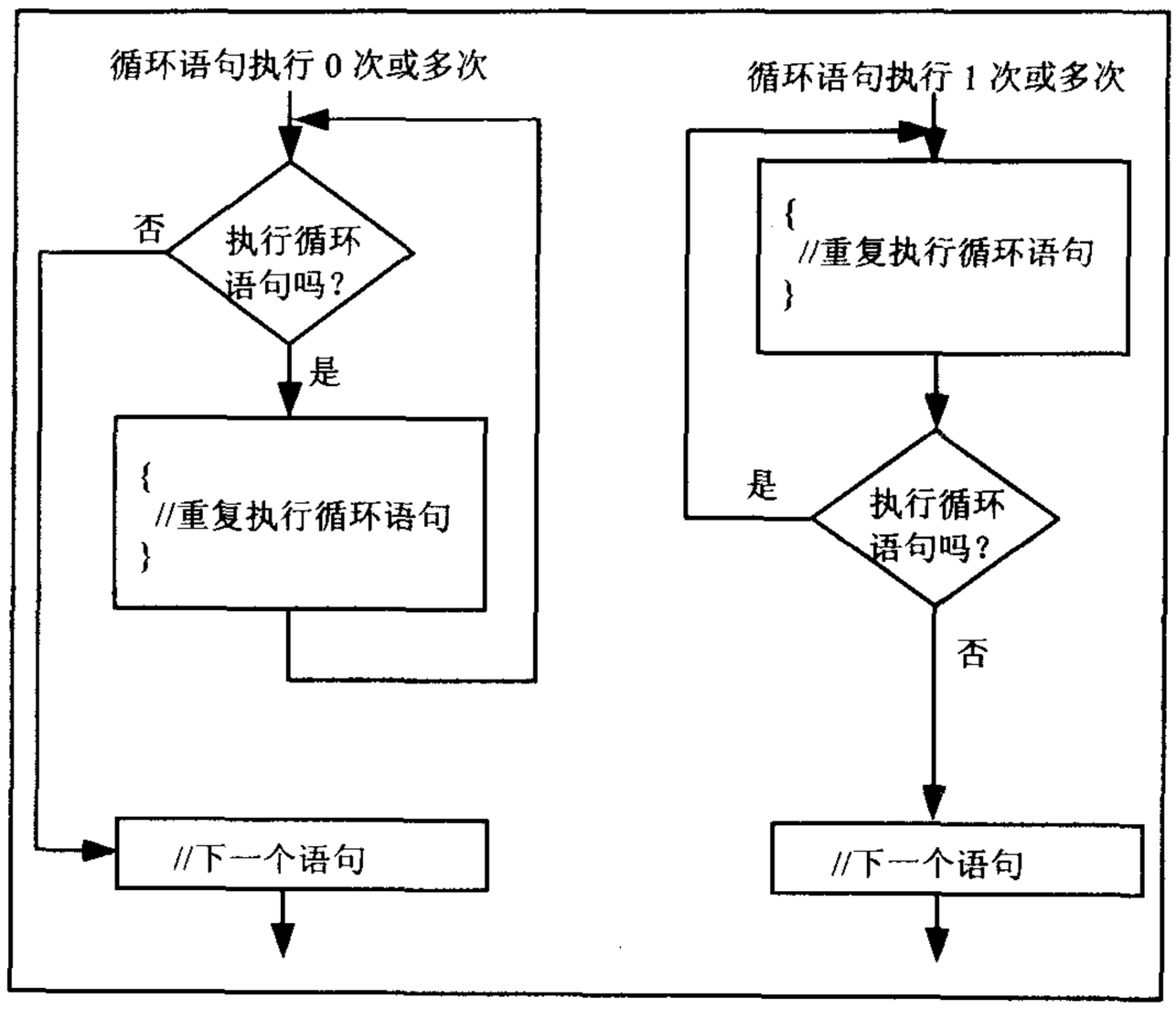


图 5-1　两种基本循环形式

在 C++中，有 3 种循环：

- while 循环
- do-while 循环
- for 循环

while 循环和 for 循环具有与图 5-1 中左边循环相同的结构，这些循环中的语句可能根本不会执行。而 do-while 循环具有图 5-1 中右边的结构，该循环中的语句至少要执行一次。下面先介绍 while 循环的工作原理，因为它是这 3 个循环中最简单的。

5.2　while 循环

while 循环使用逻辑表达式来控制循环体的执行。该循环的一般形式如图 5-2 所示。

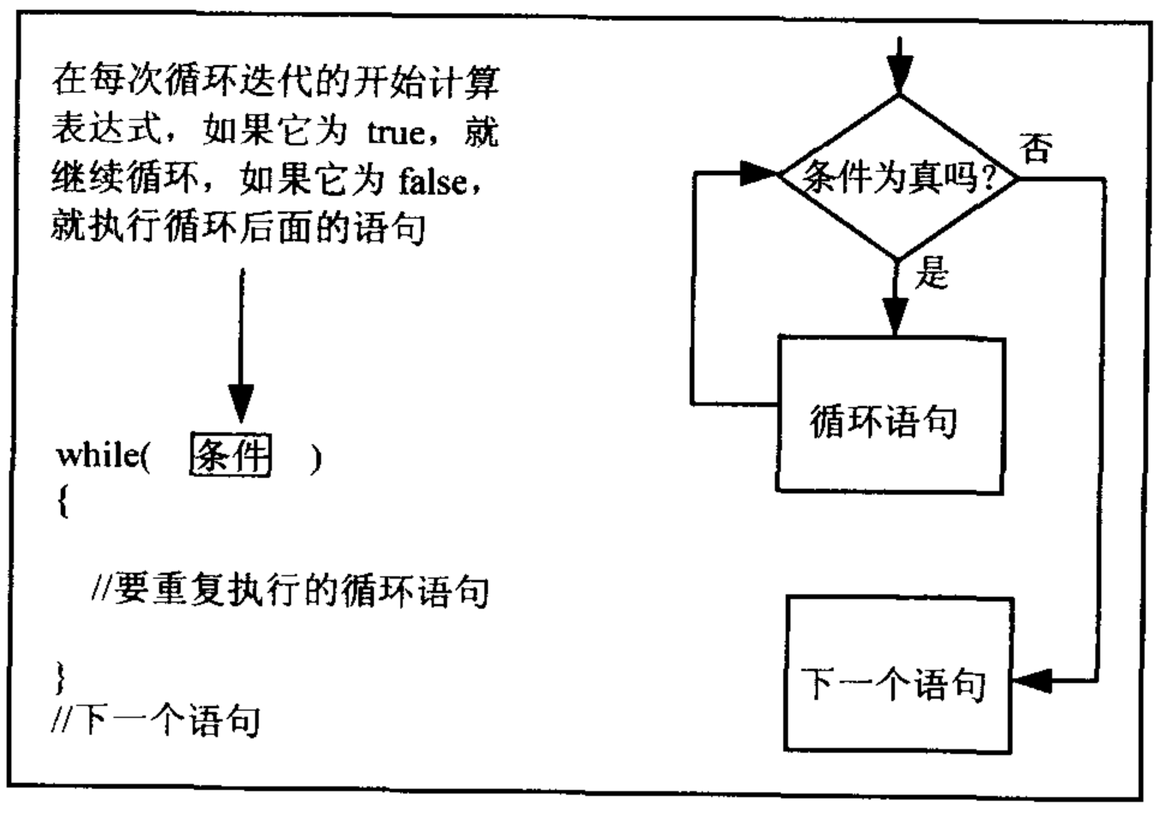


图 5-2　while 循环的执行过程

这个流程图显示了该循环的逻辑。只要条件的值为 true，就执行循环语句或循环语句块。当条件为 false 时，就执行循环语句后面的语句。可以使用任意表达式控制循环，只要该表达式的值为 bool 类型，或整数类型即可。

提示：

如果控制循环的条件表达式结果为整数，只要该数值不是 0，循环就继续。如前所述，任何非 0 整数都会转换为 bool 类型的 true，只有 0 才转换为 bool 类型的 false。

当然，while 是一个关键字，不能用它来命名程序中的任何元素。

程序示例 5.1——使用 while 循环

在本章的第一个例子中，使用 while 循环计算从 1 到 n 的整数和：

```
//Program 5.1 Using a while loop to sum integers
#include <iostream>
#include <iomanip>
using std::cin;
using std::cout;
using std::endl;

int main() {
  int n=0;
  cout << "How many integers do you want to sum: ";
  cin >> n;

  int sum=0;          //Stores the sum of integers
  int I =1;           //Stores the integer to add to the total
  cout << "Values are: "<< endl;
  while(i<=n) {
    cout << std::setw(5)<<i;      //Output current value of i
    if(i%10) ==0)
      cout<<endl;                 //Newline after ever 10 values
    sum += i++;
  }

  cout<<endl<<"Sum is "<<sum<<endl;     //Output final sum
  return 0;
}
```

执行这个程序，输出结果如下所示：

```
How many integers do you want to sum: 25
Values are:
    1    2    3    4    5    6    7    8    9   10
   11   12   13   14   15   16   17   18   19   20
   21   22   23   24   25
Sum is 325
```

例子的说明

main()中的前两个语句读取要求和的整数个数。变量 n 的值用于确定 while 循环何时结束

求和。

在开始循环之前，定义并初始化一个变量 i，它存储了当前加到总和中的整数，然后定义变量 sum，它存储总和。

```
int sum=0;          //Stores the sum of integers
int i=1;            //Stores the integers to add to the total
```

开始循环时，i 等于 1，sum 是 0。

循环条件是表达式 i<=n，只要 i 不超过 n，它就是 true，此时，执行循环语句：

```
cout <<std::setw(5)<<i;      //Output current value of i
if(i%10) ==0)
  cout <<endl;              //Newline after ever 10 values
sum += i++;
```

首先，输出 i 的当前值，其字段宽度为 5 个字符，为了使数字的输出比较整齐，在输出 10 个数值后输出一个换行符。这样一行有 10 个数字，只要最大的数值不超过 5 个数字，所有的数字就都很好地排列在各个列中。这里最后的一个语句把总和放在 sum 中。由于使用了递增运算符的后缀形式，因此 i 的当前值加到 sum 上，之后递增 i。这就执行完了循环块，执行传回到 while 处，再次用 i 的新值测试循环条件。

这种模式继续重复下去，i 递增到 2、3、4 等，直到 n 为止。但是，在加上 n 后，i 就递增到 n+1，此时循环条件为 false。循环停止，程序继续执行循环后面的下一条语句，即输出总和：

```
cout<<"Sum is "<<sum<<endl;     //Output final sum
```

其结果是循环执行 n 次，即把从 1 到 n 的整数加在一起。

注释：

这说明了循环的工作原理，但如果用户喜欢数学，就知道可以用公式 n*(n+1)/2 来计算整数 1 到 n 的总和，实际上并不需要这样一个循环。

5.3 do-while 循环

do-while 循环类似于 while 循环，只要指定的循环条件为 true，循环就继续执行下去。其区别是在 do-while 循环中，循环条件是在循环的最后检查，而不是在开始检查，所以循环语句至少要执行一次。

do-while 循环的逻辑和一般形式如图 5-3 所示。特别要注意 while 语句后面的分号，这是必须的。如果遗漏了它，程序就不会编译。

如果代码块总是要执行一次，也可以执行多次，使用这个逻辑再合适不过了。下面用一个例子来说明。

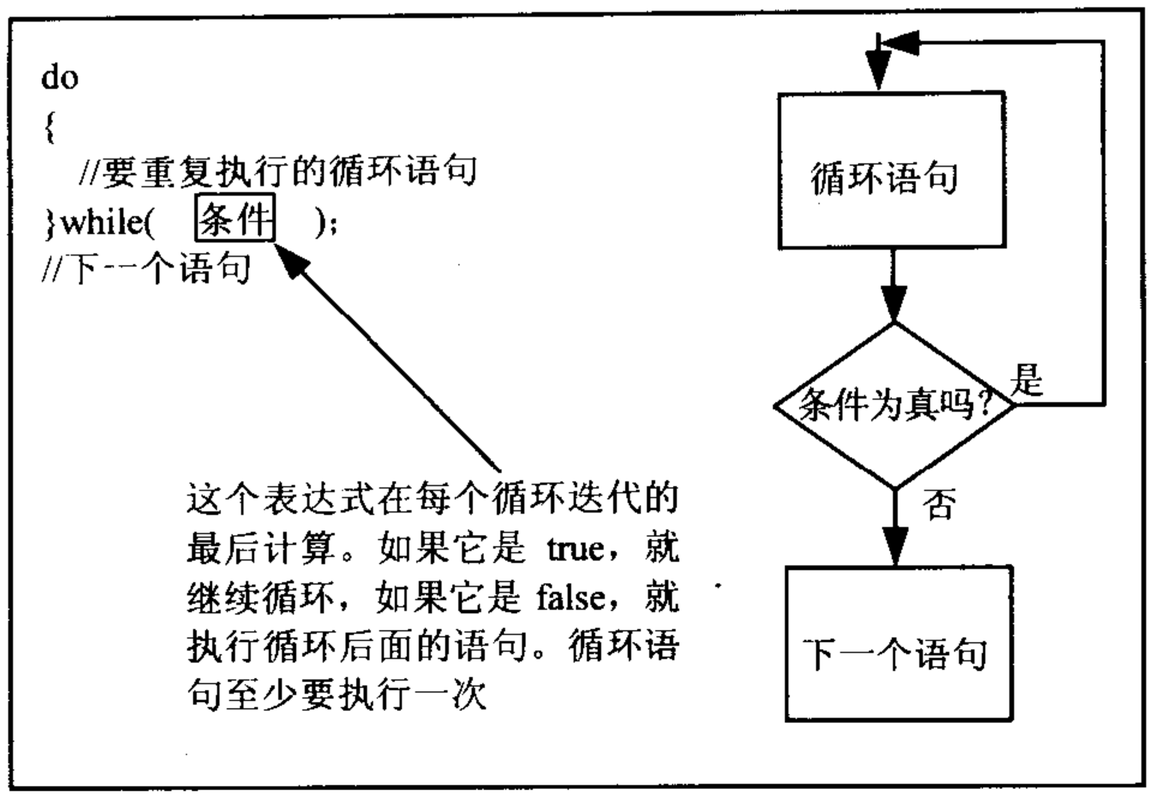


图 5-3　do-while 循环的执行过程

程序示例 5.2——使用 do-while 循环控制输入

假定要计算任意个输入值的平均值，例如，这些输入值可以是在某个时间段搜集来的温度。事先无法知道输入多少个值，但可以假定至少会有一个输入值，否则，程序就根本不会执行。此时最好使用 do-while 循环。下面是程序：

```
//Program 5.2 Using a do-while loop to control input
#include <iostream>
using std::cin;
using std::cout;
using std::endl;

int main() {
  char ch=0;                  //Stores response to prompt for input
  int count=0;                //Counts the number of input values
  double temperature=0.0;     //Stores an input value
  double average=0.0;         //Stores the total and average

  cout<<endl;

  do {
    cout<<"Enter a temperature reading: ";  //Prompt for input
    cin>> temperature;                      //Read input value

    average += temperature;                 //Accumulate total of values
    count++;                                //Increment value count

    cout<<"Do you want to enter another?(y/n):";
    cin>>ch;                                //Get response
    cout<<endl;
  }while(ch=='y');

  average /= count;                         //Calculate the average
  cout<<"Average temperature is "<<average
      <<endl;
return 0;
```

```
}
```

该程序的示例会话如下所示：

```
Enter a temperature reading: 53
Do you want to enter another?(y/n): y

Enter a temperature reading: 65.5
Do you want to enter another?(y/n): y

Enter a temperature reading: 74
Do you want to enter another?(y/n): y

Enter a temperature reading: 69.5
Do you want to enter another?(y/n): n

Average temperature is 65.5
```

例子的说明

首先，程序声明并初始化了循环和计算需要的变量：

```
char ch=0;                  //Stores response to prompt for input
int count=0;                //Counts the number of input values
double temperature=0.0;     //Stores an input value
double average=0.0;         //Stores the total and average
```

变量 ch 用于存储对以后输入提示的响应，它在循环的最后测试。只要输入了 y，程序就继续读取输入值(理想情况下，程序还应接受 Y，稍后修正这个错误)。其他 3 个变量的目的在注释中说得很清楚。

读取输入值的循环如下所示：

```
do {
  cout<<"Enter a temperature reading: "; //Prompt for input
  cin>> temperature;                     //Read input value

  average += temperature;                //Accumulate total of values
  count++;                               //Increment value count

  cout<<"Do you want to enter another?(y/n):";
  cin>>ch;                               //Get response
  cout<<endl;
} while(ch=='y');
```

因为这里使用的是 do-while 循环，所以至少要读取一个值。在提示输入后，循环语句块就会从键盘中读取一个值，并把它存储在 temperature 变量中。接着把这个值加到 average 上。在循环结束时，average 就包含了所有输入值的总和。程序还递增了 count，因为需要知道输入了多少个值，才能计算平均值。最后在循环中，提示输入 y 或 n，指出是否还要输入更多的值。在输入 n(实际上可以输入除 y 之外的所有其他字符)后，循环条件 ch=='y'就是 false，循环终止。程序继续执行下面的语句：

```
average /= count;          //Calculate the average
```

这个语句用在 average 中累加的总和除以输入值的个数 count，得到平均值。存储在 count 中的值自动转换为 double，与 average 的类型相同，之后执行除法操作。最后，输出结果，程序结束。

更复杂的 while 循环条件

当然，控制 while(或 do-while)循环时不仅可以使用简单的比较，还可以使用结果为 true 或 false 的任何表达式，或者生成一个整数值的任意表达式。前面例子的一个问题是，如果从键盘上输入了 Y，而不是 y，程序就会终止。这不是非常好的编程方式。最好允许输入 y 和 Y，继续循环。为此，只需要把循环条件修改为：

```
} while(ch=='y' || ch=='Y');
```

则输入大写的 Y 或小写的 y 都可以使循环继续。另外，还可以使用标准库的字符转换函数(详见第 4 章)。首先，在代码的开始包含头文件，如下所示：

```
//Program 5.2 Using a do-while loop to control input
#include <iostream>
#include <cctype>
```

现在就可以把下面的循环条件放在上面的程序中，确保用 ch 的小写版本与 y 比较：

```
} while(std::tolower(ch)=='y');
```

如前所述，还可以把等于数值的表达式用作循环条件。在这种情况下，编译器会把表达式的结果转换为 bool 类型。记住，0 转换为 false，任何非 0 值，无论正负，都转换为 true。因此，只有在条件为 0 时，用数值控制的 while 循环才会终止。

5.4 for 循环

for 循环主要用于对语句或语句块执行预定的次数，但也可以用于其他方式。

可以使用以分号分隔开的 3 个表达式来控制 for 循环，这 3 个表达式放在关键字 for 后面的括号中。如图 5-4 所示。

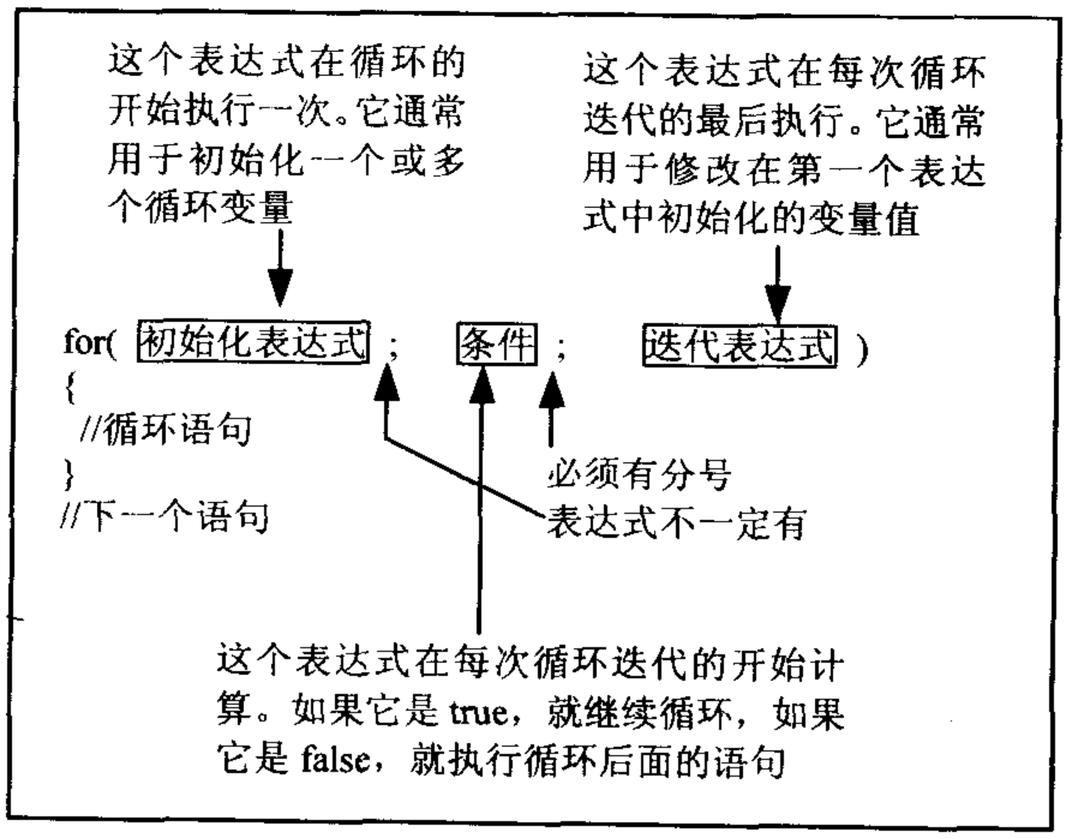


图 5-4　for 循环的控制方式

控制 for 循环的任一表达式或所有表达式都可以省略，但分号必须有。这么做的原因是不明显的，但非常有效，本章后面将探讨省略表达式的一些情况。

图 5-5 显示了 for 循环的流程逻辑。

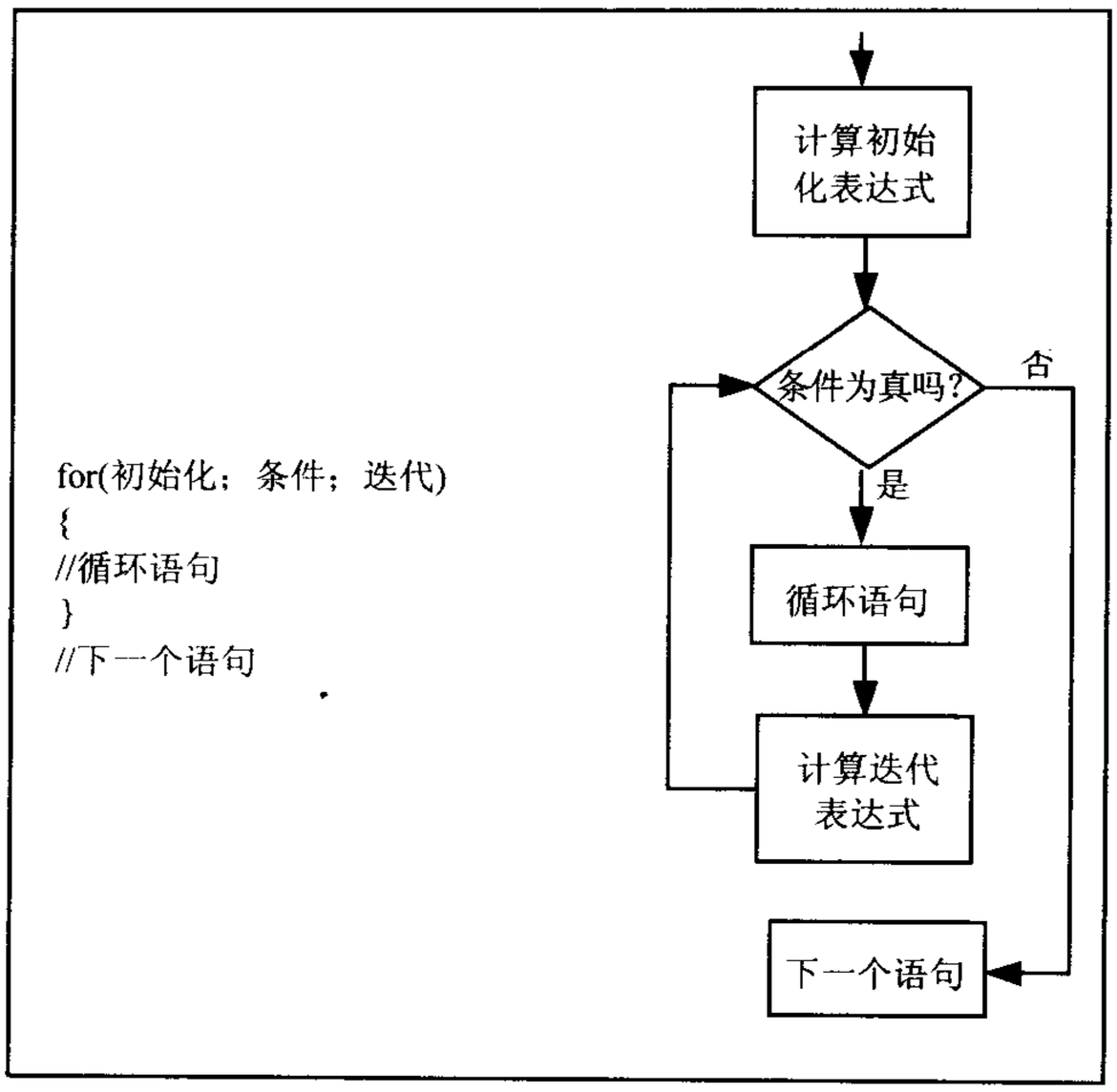


图 5-5　for 循环的逻辑

初始化表达式只在循环的开始处计算一次。接着检查循环条件，如果它是 true，就执行循环语句或语句块。如果条件是 false，就跳过循环语句，执行循环后面的下一条语句。在这方面，for 循环与 while 循环很相近，与 do-while 循环不太相似。

假定条件是 true，就执行循环语句，接着计算迭代表达式，之后再次检查条件，看看是否继续循环。

程序示例 5.3——使用 for 循环

在 for 循环的一般用法中，第一个表达式用于初始化一个计数器，第二个表达式用于检查计数器是否达到了给定的极限，第三个表达式用于递增计数器。下面是一个例子，它使用 for 循环来计算整数的总和：

```
//Program 5.3 Using a for loop
#include <iostream>
using std::cin;
using std::cout;
using std::endl;

int main() {
  int sum=0;                //Accumulates the sum of integers
  int count=0;              //The number to sum

  cout << "How many integers do you want to sum?";
  cin >> count;
```

```
 for (int i=1;i <=count; i++)
   sum +=i;

 cout<< endl
     << "The sum of the integers from 1 to "<<count
     << "is "<<sum <<endl;

 return 0;
}
```

这个程序的输出结果如下所示：

```
How many integers do you want to sum? 25

The sum of the integers from 1 to 25 is 325
```

例子的说明

用下面的语句从键盘上读取要求和的整数的上限：

```
cout<<" How many integers do you want to sum? ";
cin >> count;
```

累加总和的循环语句进行了缩进，来显示这是 for 循环的一部分：

```
for (int i=1;i <=count; i++)
  sum +=i;
```

因为循环语句只有一条，所以没有使用花括号。这个循环的结果是在变量 sum 中累加从 1 到 count 的整数。循环中的执行顺序如下所示：

(1) 执行第一个表达式。此表达式声明整型变量 i，并把它初始化为 1。

(2) 执行第二个表达式，检查 i 是否小于或等于 count。如果 i <=count 为 true，就进入第(3)步。如果它等于 false，就进入第(6)步。

(3) 执行循环语句，把 i 的当前值加到 sum 上。

(4) 执行第三个表达式，递增 i 的值。

(5) 返回到第(2)步。

(6) 退出循环。

将 i 的值连续加到 sum 中，从 1 开始，直到 count 的值为止。最后，当 i 递增到 count+1 时，for 循环结束。程序接着执行循环后面的语句，即输出在 sum 中累加的总和。

在 for 循环的初始化表达式中声明变量是合法的，这种用法很普遍。其中有一些重要的暗示，需要进一步探讨。

5.4.1 循环和变量作用域

for 循环与 while 循环、do-while 循环类似，定义了一个作用域。循环语句或语句块，以及控制循环的任何表达式都在循环的作用域中。这还包括用于控制 for 循环的 3 个表达式。在 ANSI C++中，在循环作用域中声明的自动变量在该循环外都不存在。因此，上一个例子在循环的初始化表达式中声明的变量 i，会在循环结束时释放。如果使用下面的语句在循环结束后引用

变量 i：

```
std::cout << std::endl
          << "The loop counter has the value "<<i<< std::endl;
```

程序就不再编译，ANSI 兼容编译器会生成一个错误消息，说明 i 不存在。

但是，如果确实需要在循环结束后使用 i 的值，该怎么办？只需在执行循环之前声明变量 i 即可。例如：

```
int i=1;                                  //Declare and initialize loop counter
for ( ;i <=count; i++)                    //First expression is omitted
  sum +=i;

std::cout << std::endl
          << "The sum of the integers from 1 to "<<count
          << " is "<<sum
          << std::endl;

std::cout << std::endl
          << "The loop counter has the value "<<i<< std::endl;
```

现在就可以在循环之后访问 i 的值了，因为它是在循环的作用域之外声明的。i 也可以在执行循环之前初始化，这样在本程序中就不需要包含循环的初始化表达式了。但一般情况下，最好在 for 语句中初始化计数器，这样，无论在程序的其他地方如何修改计数器的值，都可以确信该计数器的值。

注意：

尽管省略了初始化表达式，但把初始化表达式和循环条件分隔开的分号是不能省略的。

ANSI 标准之前的 for 循环

在 C++标准推出之前，许多 C++程序都假定 for 循环变量的作用域延伸到 for 循环体的块尾。因此，与 ANSI 不兼容的许多编译器允许在循环结束后引用 for 循环变量。目前大多数 ANSI 兼容编译器不支持这种操作，或者通过一个需要明确打开的编译器选项支持它。但是，有时会遇到旧式编译器，下面看看这种问题是如何产生的。

在 ANSI C++中，可以编写下面的代码：

```
int count=10;

//Calculate the sum of integers 1 to count
int count=10;
long sum=0;
 for(int i=1;i <=count; i++)
   sum +=i;

//Calculate the product of integers 1 to count
long product=1;
for(int i=2;i <=count; i++)
  product *=i;
```

使用不支持 ANSI 标准的编译器时，这段代码不会编译，而会生成一个错误消息，说明变量 i 是重复声明。原因是该编译器允许把 i 的作用域延伸到包含 for 循环的块尾(当然这没有显示在上面的代码中)。因此，尝试在第二个循环中声明变量 i 时，在第一个循环中声明的 i 仍旧存在。

5.4.2 用浮点数值控制 for 循环

前面使用 for 循环的例子都是用整型变量来控制循环，但通常还可以使用变量来控制循环。下面的例子就使用了浮点数：

```
const double pi=3.14159265;
for(double radius=2.5; radius <=20.0; radius +=2.5)
  std::cout<<" radius= "<< std::setw(12)<< radius
           <<"  area= "<< std::setw(12)
           <<pi* radius* radius << std::endl;
```

这个循环用 radius 变量控制，其类型是 double。它的初始值是 2.5，每次循环迭代时都会递增，直到其值超过 20.2 为止，此时循环结束。循环语句利用标准公式πr^2，根据 radius 变量的当前值计算圆的面积，其中 r 是圆的半径。在循环语句中使用操纵程序 setw()给每个输出值指定相同的字段宽度，确保输出值整齐地排列。当然，要在程序中使用操纵程序，需要包含头文件<iomanip>。

在使用浮点数变量控制 for 循环时应小心。如第 2 章所述，小数部分的值可能不能用二进制浮点数准确地表示，这会导致一些意想不到的负面效应。

程序示例 5.4——在 for 循环控制中使用浮点数变量

这个程序对前面的循环进行了一些修改，演示了使用浮点数值可能出现的问题：

```
//Program 5.4 Floating-point control in a for loop
#include <iostream>
#include <iomanip>
using std::cout;
using std::endl;

int main(){
  const double pi=3.14159265;
  cout<<endl;

  for(double radius=.2; radius <=3.0; radius +=.2)
    cout << " radius= "<< std::setw(12)<< radius
         << "  area= "<< std::setw(12)<<pi* radius* radius
         << endl;
  return 0;
}
```

该程序的输出如下所示：

```
radius=          0.2     area=      0.125664
radius=          0.4     area=      0.502655
```

```
radius=        0.6    area=      1.13097
radius=        0.8    area=      2.01062
radius=          1    area=      3.14159
radius=        1.2    area=      4.52389
radius=        1.4    area=      6.15752
radius=        1.6    area=      8.04248
radius=        1.8    area=      10.1788
radius=          2    area=      12.5664
radius=        2.2    area=      15.2053
radius=        2.4    area=      18.0956
radius=        2.6    area=      21.2372
radius=        2.8    area=      24.6301
```

例子的说明

在这个例子中，出现了本不应出现的情况。观察一下代码，应在列表的最后看到半径为 3.0 的圆面积，毕竟循环指定为只要 radius 小于或等于 3.0 就继续执行，但在最后，radius 的值显示为 2.8，什么地方出问题了？

循环结束得比预期早，因为把 0.2 加到 2.8 上时，结果大于 3.0。原因是把 0.2 表示为二进制浮点数时有一个非常小的错误，不能把 0.2 准确地表示为二进制浮点数。错误出在精度的最后一位上，如果编译器对 double 类型支持 15 位精度，错误就是 10^{-15} 那一位。通常这是不会出错的，但这里要给 radius 连续加上 0.2，以得到准确的 3.0，而执行结果不是这样。

在循环中添加一个语句，显示 3.0 和 radius 的下一个值之间的差异：

```
for(double radius=.2; radius <=3.0; radius +=.2)
  cout << " radius= "<< std::setw(12)<< radius
       << "  area= "<< std::setw(12)<<pi* radius* radius
       << "  delta to 3 = "<<(( radius+.2)-3.0)
       << endl;
```

最后一行输出如下所示：

```
radius=     2.8  area=    24.6301      delta to 3 = 4.44089e-016
```

因为 radius + .2 大于 3.0 约 4×10^{-16}，所以循环在下一次迭代之前终止。

那么，为什么无论刚才讨论的误差有多少，radius 值的输出总是正确的？这是因为该过程生成了所显示的值。如果希望得到 radius 的准确值，可以以科学表示法输出它们，即以尾数加指数的形式输出。

注意：

任何数字，只要其分数部分的分母是奇数，就不能准确地表示为二进制的浮点数。

程序示例 5.5——以科学表示法显示数字

为了弄明白循环的执行情况，可以修改程序，使用第 2 章介绍的某个浮点数输出操纵程序来显示值：

```
//Program 5.5 Displaying numbes in scientific notation
#include <iostream>
#include <iomanip>
#include <limits>
```

```
using std::cout;
using std::endl;
using std::setprecesion;
using std::numeric_limits;

int main() {
  const double pi=3.14159265;
  cout<<endl;

  for (double radius=.2; radius <=3.0; radius +=.2)
    cout  << " radius= "
          << setprecision(numeric_limits<double>::digits10+1)
          << std::scientific<< radius
          << "  area= "
          << std::setw(10)<< setprecision(6)
          << std::fixed<<pi* radius* radius
          << endl;
  return 0;
}
```

注意：

不要对哪些 std 名称在 using 声明中标识，哪些名称要明确限定感到迷惑。这常常取决于代码是否能放在有限页宽的页面上。

进行了上述修改后，输出结果如下所示：

```
radius = 2.000000000000001e-001  area =  0.125664
radius = 4.000000000000002e-001  area =  0.502655
radius = 6.000000000000009e-001  area =  1.130973
radius = 8.000000000000004e-001  area =  2.010619
radius = 1.000000000000000e+000  area =  3.141593
radius = 1.200000000000000e+000  area =  4.523893
radius = 1.399999999999999e+000  area =  6.157522
radius = 1.599999999999999e+000  area =  8.042477
radius = 1.799999999999998e+000  area = 10.178760
radius = 1.999999999999998e+000  area = 12.566371
radius = 2.199999999999997e+000  area = 15.205308
radius = 2.399999999999999e+000  area = 18.095574
radius = 2.600000000000001e+000  area = 21.237166
radius = 2.800000000000003e+000  area = 24.630086
```

这里尾数显示的位数是 Intel PC 所通用的，在不同的机器上，其位数可能不同。可以看出，radius 值并不像前面显示的那样精确。

例子的说明

程序中包含<limits>头文件，是因为该程序需要访问 double 类型的浮点数在其尾数中的位数信息(十进制)。循环中的输出语句修改了浮点数显示的方式：

```
cout << " radius= "
     << setprecision(numeric_limits<double>::digits10+1)
     << std::scientific<< radius
```

```
<< "  area= "
<< std::setw(10)<< setprecision(6)
<< std::fixed<<pi* radius* radius
<< endl;
```

正确地使用 scientific 和 setprecision()操纵程序，就可以以科学表示法显示半径值。为了确保显示出所有的位数，必须使用在<limits>头文件中定义的一些信息，该头文件在第 3 章介绍过。由 numeric_limits<double>::digits10 指定的整数值是类型为 double 的值在其尾数中的小数位数。这个值通常反映了尾数中的所有小数位数。因为尾数是二进制形式的，不一定对应于小数位数，所以仍会有一些额外的二进制数字使结果被圆整。因此给位数加 1，以确保不出现圆整情况。

由于不需要用科学表示法表示面积的值，因此使用 fixed 操纵程序把浮点数的输出模式重新设置为显示后续的值，而不带指数。再用操纵程序 setw(10)把面积的字段宽度设置为 10 个字符。当然，由于精确的位数仍是其最大值加 1，但实际上不需要这么多位数，因此用操纵程序 setprecision(6)把位数设置为更有意义的值。

5.4.3 使用更复杂的循环控制表达式

在 for 循环中并不只是仅能使用简单的控制表达式。下面介绍一个略复杂的例子。返回本章前面的第一段代码——计算前 10 个整数的总和。可以在 for 循环的迭代表达式中进行计算，从而省略循环语句：

```
int count=10;
int sum=0;
for(int i=1;i<=count; sum += i++)
  ;
```

注意循环控制表达式后面的分号，这实际上是一个空的循环语句。计算是在迭代表达式 sum += i++中进行的。把 i 的当前值加到 sum 上，再使用后缀递增运算符给 i 加 1，准备进行下一次迭代。以这种方式使用第三个控制表达式，对简单的循环操作非常方便，但不应过多地使用这个技术，因为它会降低代码的可读性。

也可以在 for 循环的第一个控制表达式中初始化多个变量。方法是用逗号分隔开初始化表达式。下面看一个例子。

程序示例 5.6——初始化多个变量

在这个程序中，将在循环控制表达式中初始化 3 个变量：

```
//Program 5.6 Multiple initializations in a loop expression
#include <iostream>
#include <iomanip>
using std::cout;
using std::cin;
using std::endl;
using std::setw;

int main() {
```

```
int count=0;
cout << endl << "What upper limit would you like? ";
cin >> count;

cout << endl
     << "integer"              //Output column headings
     << "       sum"
     << "         factorial"
     << endl;

  for(long n=1,sum=0,factorial=1;n<=count;n++) {
    sum += n;                 //Accumulate sum to current n
    factorial *= n;                 //Calculate n!
    cout  << setw(4)<<n   <<"     "
          << setw(7)<<sum <<"  "
          << setw(15)<< factorial
          << endl;
  }
  return 0;
}
```

这个程序计算从 1 到 count 的所有整数的总和，其中 count 是用户输入的一个值。本程序还计算了每个整数的阶乘(整数 n 的阶乘写作 n!，就是把从 1 到 n 的所有整数乘在一起，例如 5!=1×2×3×4×5=120)。不能给 count 输入太大的值，因为阶乘会增长得非常快，很容易超出 long 整数变量的取值范围。

这个程序的一般输出结果如下所示：

```
What upper limit would you like? 10

integer   sum     factorial
   1        1             1
   2        3             2
   3        6             6
   4       10            24
   5       15           120
   6       21           720
   7       28          5040
   8       36         40320
   9       45        362880
  10       55       3628800
```

例子的说明

首先，需要显示一个提示，通过键盘读取 count 的值：

```
int count=0;
cout<<endl<<"What upper limit would you like? ";
cin >> count;
```

对于 4 个字节的 long 值，超过 13 都会使其阶乘超出 long 变量能存储的最大值，从而得到不正确的结果。

变量 n、sum 和 factorial 都在循环表达式中声明和初始化：

```
for(long n=1,sum=0,factorial=1;n<=10;n++){
```

逗号把每个变量分隔开来，就像在声明语句中那样。在循环中，进行算术运算后，会在循环语句中，为要输出的每个变量给 setw()操纵程序传送不同的值：

```
cout << setw(4)<<n   <<"   "
     << setw(7)<<sum <<" "
     << setw(15)<< factorial
     << endl;
```

setw()操纵程序选择的字段宽度会把数值排列在循环执行前显示的标题下面的列上。

逗号运算符

尽管逗号看起来像是一个分隔符，但实际上它是一个二元运算符。可以把两个表达式组合到一个表达式中，组合后的表达式的结果就是其右操作数的结果。也就是说，只要能编写表达式，就可以编写用逗号隔开的一组表达式。例如，下面的语句：

```
int i=1;
int value1=1;
int value2=1;
int value3=1;
value1+= ++i, value2+= ++i,  value3 += ++i;
```

前 4 个语句把每个变量都初始化为 1。最后一个语句由 3 个赋值表达式组成，用逗号运算符隔开。因为逗号运算符是左相关的，且在所有的运算符中优先级最低(参见附录 D)，所以该语句的执行顺序为：

```
(((value1+= ++i), (value2+= ++i)),  (value3 += ++i));
```

结果是，value1 等于 3，value2 等于 4，value3 等于 5。复合表达式的值是该系列中最右边的表达式的值，所以其值(这里省略了，因为没有把该结果赋予任何变量)应是 5。下面修改前面例子中的循环，把计算组合到第二个循环控制表达式中，以演示逗号运算符的效果。

程序示例 5.7——逗号运算符

下面是上一个例子的修改版本：

```
//Program 5.7 Demonstrating the comma operator
#include <iostream>
#include <iomanip>
using std::cout;
using std::cin;
using std::endl;
using std::setw;

int main() {
  int count=0;
  cout << endl << "What upper limit would you like? ";
  cin >> count;
```

```
cout << endl
     << "integer"              //output column headings
     << "       sum"
     << "         factorial"
     << endl;

for(long n=1, sum=0, factorial=1; sum +=n, factorial *= n, n<=count; n++)

  cout << setw(4)<<n   <<"   "
       << setw(7)<<sum <<" "
       << setw(15)<< factorial
       << endl;
  return 0;
}
```

输入与上一例相同的数值，该程序会输出与上一例完全相同的结果。

例子的说明

为了说明逗号运算符，这里把总和及阶乘的计算放在第二个循环表达式中：

```
for(long n=1, sum=0, factorial=1; sum +=n, factorial *= n, n<=count; n++)

  cout << setw(4)<<n   <<"   "
       << setw(7)<<sum <<" "
       << setw(15)<< factorial
       << endl;
```

尽管输出是相同的，但代码和计算的执行与前面的例子不同。第二个循环控制表达式是在每次迭代的开始计算。所以每次都会递增 sum，计算 factorial。复合表达式的值就是表达式 n<=count 的值，这仍决定了循环何时停止。当这个条件测试失败时，程序已经用 n 的当前值计算了 sum 和 factorical，但这些都不会显示出来，因为循环语句不再执行。在本例中这并不重要，毕竟这里的计算是非常简单的，但在许多情况下这就会出现问题。

可以把计算放在第三个控制表达式中，但这样不会得到正确的结果。因为循环语句在循环的迭代表达式之前执行，所以输出时 sum 和 factorical 的值与 n 的当前值不对应。

坦白地说，以这种方式编程并没有好处，这只是演示语法的一个例子，但不是一种好的编程风格。不过，当需要在一个语句中计算好几个表达式时，逗号运算符偶尔也是很有用的。

5.5 嵌套的循环

可以把一个循环放在另一个循环内部。实际上，可以在循环中嵌套多次，直到解决问题为止。而且，嵌套的循环可以是任何类型：如果需要，可以在 while 循环中嵌套 for 循环，再把该 while 循环嵌套在 do-while 循环中。它们可以以任何方式混合在一起。

嵌套循环最常见的应用是用于数组(详见第 6 章)，但它们也有其他用途。下面用一个例子来演示嵌套循环，该例子提供了许多使用嵌套循环的机会。

程序示例 5.8——使用嵌套循环

乘法表是许多孩子在学校必学的功课，下面使用嵌套循环来生成一个乘法表，代码如下：

```
//Program 5.8 Generating multiplication tables
#include <iostream>
#include <iomanip>
#include <cctype>
using std::cout;
using std::cin;
using std::endl;
using std::setw;

int main() {
  int table=0;                //Table size
  const int table_min=2;      //Minimum table size
  const int table_max=12;     //Maximum table size
  char ch=0;                  //Response to prompt

  do {
    cout  << endl
          << "What size table would you like("
          << table_min<<" to "<< table_max<<")? ";
    cin  >> table;            //Get the table size
    cout  << endl;

 //Make sure table size is within the limits
 if(table< table_min || table > table_max) {
   cout << "Invalid table size entered. Program terminated. "
        <<endl;
   exit(1);
 }

//Create the top line of the table
cout<<"         |";
for(int i=1; i <= table; i++)
  cout<<" "<<setw(3)<<i<<" |";
cout<<endl;

//Create the separator row
for(int i=0; i <= table; i++)
 cout<<"------ ";
cout<<endl;

for(int i=1; i <= table; i++)          //Iterate over rows
  cout<<" "<<setw(3)<<i<<" |";         //Start the row

 //Output the values in a row
for(int j=1; j <= table; j++)
  cout<<" "<<setw(3)<<i*j<<" |";  //For each col.
cout<<endl;                       //End the row
```

```
    }

    //Check if another table is required
    cout<<endl<<"Do you want another table(y or n)? ";
    cin>>ch;
    cout<<endl;
    }while(std::tolower(ch)=='y');

    return 0;
}
```

这个程序的输出结果如下所示。

```
What size table would you like(2 to 12)?10
    |   1 |   2 |   3 |   4 |   5 |   6 |   7 |   8 |   9 |  10 |
----------------------------------------------------------------
  1 |   1 |   2 |   3 |   4 |   5 |   6 |   7 |   8 |   9 |  10 |
  2 |   2 |   4 |   6 |   8 |  10 |  12 |  14 |  16 |  18 |  20 |
  3 |   3 |   6 |   9 |  12 |  15 |  18 |  21 |  24 |  27 |  30 |
  4 |   4 |   8 |  12 |  16 |  20 |  24 |  28 |  32 |  36 |  40 |
  5 |   5 |  10 |  15 |  20 |  25 |  30 |  35 |  40 |  45 |  50 |
  6 |   6 |  12 |  18 |  24 |  30 |  36 |  42 |  48 |  54 |  60 |
  7 |   7 |  14 |  21 |  28 |  35 |  42 |  49 |  56 |  63 |  70 |
  8 |   8 |  16 |  24 |  32 |  40 |  48 |  56 |  64 |  72 |  80 |
  9 |   9 |  18 |  27 |  36 |  45 |  54 |  63 |  72 |  81 |  90 |
 10 |  10 |  20 |  30 |  40 |  50 |  60 |  70 |  80 |  90 | 100 |
Do you want another table (y or n )?n
```

例子的说明

本例包含了 3 个标准头文件：

```
#include <iostream>
#include <iomanip>
#include <cctype>
```

第一个头文件用于流的输入输出，第二个头文件用于访问流操纵程序，包含第三个头文件是因为本例要使用 tolower()字符转换函数。

程序首先声明了一些必要的变量：

```
int table=0;                //Table size
const int table_min=2;      //Minimum table size
const int table_max=12;     //Maximum table size
char ch=0;                  //Response to prompt
```

第二和第三个声明定义了乘法表尺寸的上下限。因为这些内容在任何地方都不应改变，所以把它们声明为 const。这样就有了第 2 章讨论的优势：删除了幻数，减少了维护点。

程序在 do-while 循环中生成一个表，如果需要，还可以生成更多的表。循环执行了 4 个任务，它们都在注释中描述过了：

```
do{
  //Read the table size from keyboard...
```

```
    //Make sure the table size is within the limits…

    //Create and output the table...

    //Get an indication whether another table is required…
} while(std::tolower(ch)== 'y');
```

显然，当提示显示另一个表时，变量 ch 存储了输入的响应。读取表尺寸的代码如下所示：

```
  cout << endl
       << "What size table would you like("
       << table_min << " to "<< table_max<<")? ";
  cin >> table;              //Get the table size
  cout << endl;
```

这段代码使用 const 值 table_min 和 table_max 输出表允许尺寸的上下限。为了处理输入的表尺寸无效的情况，可以将输入的值与允许的上下限进行比较：

```
  if(table< table_min | | table > table_max) {
    cout<<"Invalid table size entered. Program terminated. "
    <<endl;
exit(1);
}
```

如果 table 的值小于最小值或大于最大值，就显示一条消息并使用标准库函数 exit()退出程序。本章后面将介绍如何以更友好的方式处理这种情况。

接着生成表。嵌套在 do-while 循环中的第一个 for 循环生成表的第一行，其中包含乘数：

```
cout<<"          |";
for(int i=1; i <= table; i++)
  cout<<" "<<setw(3)<<i<<" |";
cout<<endl;
```

循环语句输出用空格和竖杠隔开的乘数，从 1 到表的尺寸。每个输出的列都是 6 个字符宽，所有的表项都需要有相同的宽度，以确保数值整齐排列。

下一个嵌套的 for 循环生成一行虚线，把第一行与表的其他行隔开：

```
for(int i=0; i <= table; i++)
  cout<<"------ ";
cout<<endl;
```

这个循环的每次迭代都会给该行添加 6 条虚线。开始时 count 等于 0，而不是 1，总共输出 table+1 个组合，一个组合用于左边的一列乘数，其他用于每一列表项。

最后一个循环本身包含一个嵌套的循环，它输出左边一列乘数和为表项的乘积：

```
for(int i=1; i <= table; i++){          //Iterate over rows
  cout<<" "<<setw(3)<<i<<" |";          //Start the row

  //Output the values in a row
  for(int j=1; j <= table; j++)
```

```
        cout<<" "<<setw(3)<<i*j<<" |";      //For each col.
    cout<<endl;                             //End the row
}
```

加上外层的 do-while 循环，一共有 3 层嵌套。中间层的 for 循环每迭代一次，都会创建表的一行。在每次迭代中，输出语句都会开始一个新行，并显示左边的乘数。接着执行内层的循环，生成当前行的表项。因为内层 for 循环的每次迭代都会生成一个表项，所以变量 j 实际上是表中的一个列号。因为 j 变量在 1 到 table 之间变化，所以每一行都有 table 个表项。在每个位置上显示的值是行号 i 和列号 i 的乘积。

在显示完整个表后，程序还将提示是否需要生成另一个表：

```
cout<<endl<<"Do you want another table(y or n)? ";
cin>>ch;
cout<<endl;
```

输入的字符存储在 ch 中，用在 do-while 循环条件中。如果 ch 是'y'或'Y'，就执行 do-while 循环的另一次迭代，生成另一个表。否则就结束循环，退出程序。

5.6 跳过循环迭代

有时需要跳过一个循环迭代，直接开始下一次循环。continue 语句就可以完成这一操作，其形式如下所示：

```
continue;
```

在循环中执行到这个语句时，程序会立即跳到当前迭代的末尾，如果循环控制表达式允许，程序会继续执行下一次迭代。这最好用一个例子来说明。

程序示例 5.9——使用 continue 语句

假定要输出一个字符表以及对应的十六进制和十进制字符代码。当然，不希望输出没有符号表示的字符，一些字符没有符号表示，例如制表符和换行符，这些字符会使结果变得混乱。下面编写一个程序，只输出“可打印”的字符。程序代码如下所示：

```
// Program 5.9 Using the continue statement
#include <iostream>
#include <iomanip>
#include <cctype>
#include <limits>
using std::cout;
using std::endl;
using std::setw;

int main() {
   // Output the column headings
   cout << endl
        << setw(13)  <<  "Character   "
        << setw(13)  <<  "Hexadecimal  "
        << setw(13)  <<  "Decimal    "
```

```
        << endl;

cout  << std::uppercase;                          // Uppercase hex digits

unsigned char ch = 0;                             // Character code

// Output characters and corresponding codes
do {
  if(!std::isprint(ch))                          // If it does not print
     continue;                                   // skip this iteration

     cout  << setw(7)   << ch
           << std::hex                                  // Hexadecimal mode
           << setw(13)  << static_cast<int>(ch)
           << std::dec                                  // Decimal mode
           << setw(13)  << static_cast<int>(ch)
           << endl;
   } while(ch++ < std::numeric_limits<unsigned char>::max());
   return 0;
}
```

这个程序会输出编码值为 0 到 unsigned char 的最大值的所有可打印字符。在计算机上一般会显示可打印的 ASCII 字符。

例子的说明

用下面的语句输出列的标题：

```
cout << endl
     << setw(13) << "Character  "
     << setw(13) << "Hexadecimal "
     << setw(13) << "Decimal    "
     << endl;
```

因为每个标题的字段宽度都设置为 13，所以需要小心地在每一列的中间放置输出的值。为了获得以大写数字表示的十六进制值，使用下面的语句：

```
cout<< std::uppercase;     //Uppercase hex digits
```

这个语句使用 uppercase 输出修饰符，使后面的所有十六进制输出都使用 A 到 F，而不是 a 到 f。

一旦建立了存储字符编码的变量后，真正有趣的就是 do-while 循环了：

```
do {
  if(!std::isprint(ch))                          // If it does not print
     continue;                                   // skip this iteration

  cout << setw(7)   << ch
       << std::hex                                  // Hexadecimal mode
       << setw(13)  << static_cast<int>(ch)
       << std::dec                                  // Decimal mode
       << setw(13)  << static_cast<int>(ch)
```

```
        << endl;
} while(ch++ < std::numeric_limits<unsigned char>::max());
```

这个循环迭代从 0 到 numeric_limits<unsigned char>::max()输出的值之间的字符编码，其中 numeric_limits<unsigned char>::max()是 unsigned char 类型的最大值。

在上面的循环中，首先使用第 4 章介绍的函数 isprint()，来检查 ch 的当前值是否表示一个可打印字符。如果 isprint(ch)返回 0，表达式!isprint(ch)就是 true，于是执行 continue 语句。这将跳过剩余的循环语句，直接进行下一个迭代。因此，本循环只为可打印字符执行输出语句。

在输出语句中，使用操纵程序 hex 和 dec，把整数的输出模式设置为需要的模式。为了把 ch 的值显示为数值，必须在输出语句中把它强制转换为 int。否则它将总是显示为一个字符。

注意为了使代码更简单，ch 必须使用 unsigned char 类型，而不是 char 类型。根据实现方式的不同，char 类型可以等价于 signed char 类型或 unsigned char 类型(但与此不相同)。我们需要编程，允许把 char 类型转换为等价于 signed char 的类型。带符号的值的一个复杂性是若要包含所有的值，就不能从 0 开始计数，因为给 signed char 的最大值加 1，会产生最小值。如图 5-6 所示。

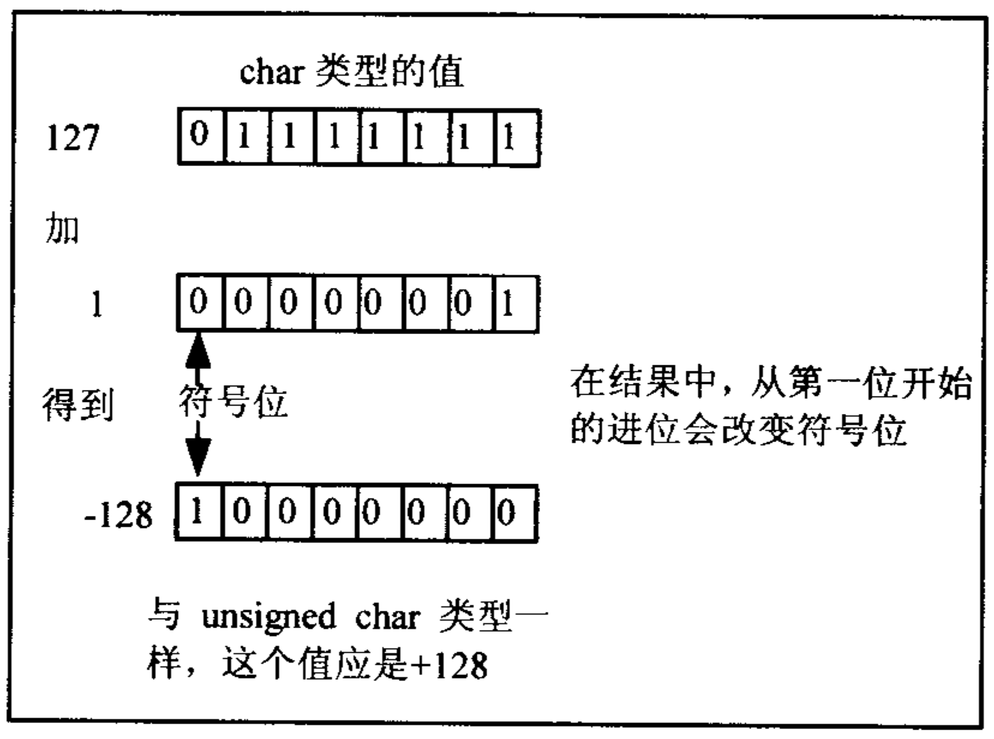


图 5-6　带符号的变量的溢出结果

使用 numeric_limits<char>::min()，把 ch 的初始值设置为该类型的最小值，就可以解决这个问题，但在把负的编码值强制转换为 int 时，会得到一个负的结果，这样十六进制的编码就会把最前面的数字显示为 F。

还要注意这里并不适合使用 for 循环。因为是在循环块执行之前测试条件，所以应把循环写成如下形式：

```
for(unsigned char ch=0;
            ch<=std::numeric_limits<unsigned char>::max(); ch++){
 //Output character and code
}
```

这个循环不会终止，因为在 ch 等于最大值时执行循环块，之后 ch 的下一个递增值是 0，第二个循环控制表达式永远不等于 false。

5.7 循环的中断

有时需要永远地终止循环。当循环语句中没有表示继续执行的代码时，就可以使用 break 语句终止循环。其效果与它在第 4 章介绍的 switch 语句中相同。如果在循环中执行 break 语句，循环就会立即终止，程序将继续执行循环后面的语句。

break 语句在无限循环中用得最多，下面就来看看无限循环。

无限循环

无限循环可以永远运行下去。例如，如果省略 for 循环中的测试条件，循环就没有停止机制了。除非在循环块中采用某种方式退出循环，否则循环会无休止地运行下去。

无限循环有几个实际应用：例如监视某种警告指示器的程序，或在工业园中搜集传感器的数据，有时就是用无限循环编写的。在事先不知道需要迭代多少次时，也可以使用无限循环，例如读取的输入数据量可变时。在这类情况下，退出循环的机制应在循环块中编写，而不应在循环控制表达式中设置。

在 for 无限循环的最常见形式中，所有的控制表达式都被省略了：

```
for( ; ; ) {
  //Statements that do something
  //There must be some way of ending the loop in here
}
```

注意即使没有循环控制表达式，分号也要写上。终止该循环的惟一方式是在循环体中编写终止循环的代码。

也可以使用下面的 while 无限循环：

```
while(true) {
  //Statements that do something
  //There must be some way of ending the loop in here
}
```

由于继续循环的条件总是 true，所以这是一个无限循环。当然，也可以有 do-while 无限循环，但它没有另外两种循环好，所以不常用。

终止无限循环的一种方式是使用 break 语句，如本节开始所述。在循环中执行 break 会立即终止循环，程序将继续执行循环后面的语句。这常常用于处理无效的输入，以输入正确的值，或者重复某个操作，例如玩一个游戏，直到用户选择不继续玩为止。下面用程序 5.8 的新版本来说明这一点，该程序生成了一个乘法表。

程序示例 5.10——使用 break 语句

在程序 5.8 中，如果输入了无效的表尺寸，就终止程序。现在修改该程序，允许用户输入正确值 3 次。下面是程序的新版本：

```
// Program 5.10 Controlling input with an infinite loop
#include <iostream>
```

```
#include <iomanip>
#include <cctype>
using std::cout;
using std::cin;
using std::endl;
using std::setw;

int main(){
  int table = 0;                                  // Table size
  const int table_min = 2;                        // Minimum table size
  const int table_max = 12;                       // Maximum table size
  const int input_tries = 3;
  char ch = 0;                                    // Response to prompt

  do {
    for(int count = 1 ;; count++) {               // Indefinite loop
      cout  << endl
            << "What size table would you like  ("
            << table_min <<" to" << table_max << ")? ";
      cin >> table;                               // Get the table size
      cout << endl;

      // Make sure table size is within the limits
      if(table >= table_min && table <= table_max)
         break;                                   // Exit the input loop
      else if(count  < input_tries)
         cout  <<  "Invalid input  - Try again.";
      else {
         cout  << "Invalid table size entered - for the third time."
               << "\nSorry,  only three goes - program terminated."
               << endl;
         exit(1);
      }
  }
  // Create the top line of the table
  cout <<  "         | ";
  for(int i = 1 ; i <= table ; i++)
     cout <<  " "  << setw(3)  << i <<"  |";
  cout << endl;

  // Create the separator row
  for(int i = 0 ; i <= table ; i++)
     cout  <<  " ------ ";
  cout << endl;

   for(int  i = 1  ;  i  <= table  ;  i++){        // Iterate over rows
     cout <<  " "  << setw(3)  << i <<"  |";      // Start the row

     // Output the values in a row
     for(int j = 1 ; j <= table ; j++)
        cout << " " << setw(3)   << i*j <<"  |";  //For each col.
     cout << endl;                                // End the row
```

```
    }

    // Check if another table is required
   cout << endl << "Do you want another table  (y or n)? ";
   cin >> ch;
   cout << endl;
} while(std::tolower(ch) == 'y');

  return 0;
}
```

这个程序的正常输出与前一版本的输出相同，但如果连续 3 次输入了不正确的表尺寸，就会得到如下结果：

```
What size table would you like (2 to 12)? 1

Invalid input - Try again.
What size table would you like (2 to 12)? 14

Invalid input - Try again.
What size table would you like (2 to 12)? 46

Invalid table size entered - for the third time.
Sorry, only three goes - program terminated.
```

例子的说明

正确输入的最多尝试次数由变量 input_tries 定义，用下面的语句声明它：

```
const int input_tries=3;
```

在程序中有惟一一个新功能，即管理输入的 for 循环：

```
for(int count = 1 ;  ; count++) {                    // Indefinite loop
   cout << endl
        << "What size table would you like  ("
        << table_min <<" to" << table max << ")? ";
   cin >> table;                                     // Get the table size
   cout << endl;

   // Make sure table size is within the limits
   if(table >= table_min && table <= table_max)
       break;                                        // Exit the input loop
   else if(count < input_tries)
       cout << "Invalid input - Try again.";
    else {
       cout << "Invalid table size entered - for the third time."
            << "\nSorry,  only three goes - program terminated."
            << endl;
       exit(1);
    }
}
```

for 循环没有停止循环的控制表达式，所以它会运行无限多次，直到循环体中的代码退出循环为止。变量 count 记录了尝试输入的次数，从而确定了何时放弃输入。如果输入了有效的值，循环中的第一个 if 语句就会执行，从而执行 break 语句，这将终止 for 循环，执行循环后面的语句。

如果输入了无效的值，就执行第二个 if 语句，该语句检查 count 的值。只要 count 小于 3，就显示一个消息，循环进入下一次迭代，进行下一次尝试。如果连续 3 次失败，程序就终止。

如果在输入循环的外部声明并初始化了 count 变量，就可以使用一个 while 无限循环：

```
int count=1;
while(true) {
   // Read input as before...

   // Make sure table size is within the limits
   if(table >= table_min && table <= table_max)
     break;                                    // Exit the input loop
   else if (count++ < input_tries)
     cout<<"Invalid input - Try again.";
     else {
        // Prompt and end the program as before...
     }
  }
```

下面是另一个例子，它使用了不包含控制表达式的 for 循环。

程序示例 5.11——使用不包含控制表达式的 for 循环

编写该程序的另一个版本，计算几个温度的平均值。这次使用 for 无限循环和 break 语句来管理输入：

```
//  Program 5.11  Calculating an average in an indefinite loop
#include <iostream>
#include <cctype>
using std::cout;
using std::cin;
using std::endl;

int main() {
    char ch = 0;                          // Stores response to prompt for input
    int count = 0;                        // Counts the number of input values
    double temperature = 0.0;             // Stores an input value
    double average = 0.0;                 // Stores the total and average

    for( ; ; ) {                          // Indefinite loop
       cout << "Enter a value: ";         // Prompt for input
       cin >> temperature;                // Read input value
       average += temperature;            // Accumulate total of values
       count++;                           // Increment value count

       cout << "Do you want to enter another? (y/n): ";
       cin >> ch;                         // Get response
       cout << endl;
```

```
        if(std::tolower(ch)  ==  'n')              // Check for no
          break;                              //  if so end the loop
    }
    cout << endl
         << "The average temperature is " << average / count
         << endl;
    return 0;
}
```

这个例子的输出结果如下所示：

```
Enter a value: 65.5
Do you want to enter another? (y/n): y

Enter a value: 67.9
Do you want to enter another? (y/n): y

Enter a value: 72.3
Do you want to enter another? (y/n): n

The average temperature is 68.5667
```

例子的说明

这里惟一要说明的是退出循环的方式：

```
for( ; ; ) {                              // Infinite loop
  cout << "Enter a value:  ";             // Prompt for input
  cin >> temperature;                     // Read input value
  average += temperature;                 // Accumulate total of values
  count++;                                // Increment value count

  cout << "Do you want to enter another? (y/n):  ";
  cin >> ch;                              // Get response
  cout << endl
  if(std::tolower(ch) ==  'n')            // Check for no
    break;                                // if so end the loop
}
```

由于 for 循环不包含控制表达式，因此会迭代无限多次。当输入了 n 或 N 时，循环中的 if 语句就会执行 break 语句，终止循环，计算并显示温度的平均值。这里也可以使用 while 无限循环：

```
while(true)  {                            // Infinite loop
    cout << "Enter a value:  ";           // Prompt for input
    cin >> temperature;                   // Read input value
    average += temperature;               // Accumulate total of values
    count++;                              // Increment value count

    cout << "Do you want to enter another?  (y/n):  ";
    cin >> ch;                            // Get response
    cout << endl;
    if(std::tolower(ch)  ==  'n')         // Check for no
```

```
        break;                                  // if so end the loop
    }
```

在这个例子中，while 无限循环与 for 无限循环之间没有区别。选择哪个循环取决于自己的喜好或可读性。

5.8 本章小结

第 6 章将进一步介绍循环的应用，几乎所有的程序都涉及到某种类型的循环。循环对编程来说是非常基本的，必须很好地掌握本章的内容。循环的要点如下：

- 循环是重复执行一组语句的机制。
- 有 3 种循环：while 循环、do-while 循环和 for 循环。
- 只要指定的条件为 true，就重复执行 while 循环。
- do-while 循环至少要执行一次，只要指定的条件为 true，就继续执行该循环。
- for 循环通常用于重复指定的次数，它有 3 个控制表达式。第一个是初始化表达式，仅在循环的开始执行一次。第二个是循环条件，在每次迭代之前执行，它必须为 true，循环才会继续。第三个控制表达式在每次迭代结束时执行，通常用于递增循环计数器。
- 任何类型的循环都可以嵌套在其他类型的循环中，嵌套次数不限。
- 在循环中执行 continue 语句会跳过当前迭代的剩余语句，如果循环控制条件允许，就直接开始下一次迭代。
- 在循环中执行 break 语句会立即退出循环。
- 循环定义了一个作用域，在循环中声明的变量不能在该循环外部访问。特别是，在 for 循环的初始化表达式中声明的变量不能在循环外部访问。

5.9 练习

1. 编写一个程序，输出从 1 开始到用户输入的数字之间所有奇数的平方。

2. 创建一个程序，它使用 while 循环累加用户输入的随机个数的整数和，最后输出所有数字的总和和浮点数类型的平均值。

3. 创建一个程序，它使用 do-while 循环计算用户在一行上输入的非空白字符的个数。在第一次遇到输入中的#字符时，停止计数。

4. 创建一个程序，输出由 8 个随机大小写字母或数字组成的密码。允许输入重复的字符。

5. 创建一个程序，循环 25 次，打印出 1 到 10 的数字，和 20 到 25 的数字。

6. 抽奖时要求在 1 到 49 之间选择 6 个不同的整数。编写一个程序，每次运行时生成 5 个抽奖选项。

第 6 章　数组和字符串

前面介绍了所有的基本数据类型，论述了在程序中进行计算和作出决策的基本知识。本章将拓宽前面介绍的用一个数据元素处理整个数据项集合的基本编程技术的应用范围，并讨论字符串的处理。

本章主要内容

- 数组的概念和用法
- 如何声明和初始化不同类型的数组
- 非空字符串的概念
- 如何使用 char 类型的数组存储字符串
- 如何声明和使用多维数组
- 如何创建和使用 char 类型的数组，并把它用作非空字符串
- 如何创建 string 类型的变量
- string 类型的对象可用于什么操作，以及如何使用它们
- 如何使用包含宽字符的字符串

6.1　数据数组

前面介绍了如何声明和初始化基本类型的变量。每个变量都可以存储指定类型的一个数据项——可以在变量中存储整数、字符等。数组可以存储相同类型的多个数据项，利用数组可以存储几个整数、字符，实际上，数组可以存储任何类型的数据。

下面看一个例子。第 5 章编写了一个程序，来计算温度的平均值。假定还要计算比该平均值高的温度有多少个，比该平均值低的温度有多少个。此时就需要保存初始的示例数据，但在一个变量中存储每个数据项，程序就会很烦琐，而且不像只存储很少几个数据项那么切实可行。而使用数组就可以很轻松地完成这个任务，许多其他操作也会变得更简单。

6.1.1　使用数组

数组只是许多内存空间，每个内存空间都可以存储相同数组类型的一个数据项，所有的数据项都通过相同的变量名来引用。例如，可以在下面的数组中存储 366 个温度值：

```
double temperatures[366];     //An array of temperatures
```

这个语句声明了 double 类型的数组，其名称是 temperatures，有 366 个元素。也就是说，这个数组有 366 个内存空间，每个内存空间都可以用于保存一个 double 类型的值。在方括号中指定的元素个数称为数组的大小。

使用一个整数可以引用数组中的各个数据项，该整数通常称为数组的索引。元素的索引是

指该元素与数组中第一个元素的偏移值。第一个元素的偏移值是 0，因此其索引是 0，索引值为 3 表示数组中的第 4 个元素——与第一个元素偏移 3 个元素。要引用元素，可以在数组名后面的方括号中放置其索引，要把 temperatures 数组的第 4 个元素设置为 99.0，可以使用下面的语句：

```
temperatures[3]=99.0;    //Set the fourth array element to 99
```

下面看看另一个数组。height 数组的基本结构如图 6-1 所示。

height[0]	height[1]	height[2]	height[3]	height[4]	height[5]
26	37	47	55	62	75

图 6-1　有 6 个元素的数组

该数组的类型是 int，有 6 个元素。每个元素都表示一个内存空间，其中存储了一个数组元素的值。每个数组元素都可以用上面的表达式引用。这个数组用下面的语句声明：

```
int height[6];      //Declare an array of six heights
```

在执行这个声明语句时，编译器为这 6 个 int 类型的值分配 6 个连续的存储空间(因此这也是一个定义)。如果 int 类型的值在计算机上需要 4 个字节，则这个数组就要占用 24 个字节。

注意：

数组的类型决定了存储每个数组元素所需要的内存量。数组的所有元素都存储在一个连续的内存块中。

当然，这个声明没有为数组指定初始值，所以它们包含的是垃圾值。

如前所述，height 数组中的每个元素都包含不同的值。例如，这些值可能是某个家庭中所有成员的身高，其单位是英寸。该数组中有 6 个元素，其索引是 0 到 5。还可以用下面的语句计算 height 数组中前 3 个元素的总和：

```
int sum3=height[0]+ height[1]+ height[2];      //The sum of three elements
```

在这里，数组的每个元素可以像普通的整数变量那样操作。但是，不能利用赋值语句把整个数组赋予另一个数组——只能操作数组中的各个元素。因此，要把一个数组的值复制到另一个数组中，就必须一次复制一个值。显然，在处理数组中的所有元素时，循环是非常有效的。

程序示例 6.1——使用数组

下面看看数组的操作。在下面的程序中，使用一个整数数组计算一组人的平均身高，再计算出有多少人的身高超过平均身高。

```
// Program 6.1 Using an array
#include <iostream>
#include <cctype>
using std::cout;
using std::cin;
using std::endl;
```

```
int main ( ) {
   int height[10];                                  // Array of heights
   int count = 0;                                   // Number of heights
   char reply = 0;                                  // Reply to prompt

   // Input loop for heights. Read heights till we are done, or the array is full
   do {
      cout << endl
           << "Enter a height as an integral number of inches: ";
      cin >> height[count++];

      // Check if another input is required
      cout << "Do you want to enter another (y or n)? ";
      cin >> reply;
} while(count < 10 && std::tolower(reply) == 'y');

// Indicate when array is full
if(count == 10)
   cout << endl  << "Maximum height count reached." << endl;

// Calculate the average and display it
double average = 0.0;                    // Stores average height
for(int i = 0; i < count ; i++)
   average += height[i];                 // Add a height
average /= count;                        // Divide by the number of heights
  cout  << endl
        << "Average height is" << average << ' inches."
        << endl;

   // Calculate how many are above average height
   int above_average = 0;                // Count of above average heights
   for(int i = 0 ; i < count ; i++)
      if(height[i] > average)            // Greater than average?
        above_average++;                 // then increment the count

cout   <<  "There"
       <<  (above_average == 1 ?  "is  "  :  "are  ")
       <<  above_average << "height"
       <<  (above_average  ==  1  ?  "  "  :  "s  ")
       <<  "above  average."
           << endl;
  return 0;
}
```

这个程序的输出如下所示：

```
Enter a height as an integral number of inches: 75
Do you want to enter another (y or n)? y

Enter a height as an integral number of inches: 56
Do you want to enter another (y or n)? y
```

```
Enter a height as an integral number of inches: 63
Do you want to enter another (y or n)? y

Enter a height as an integral number of inches: 42
Do you want to enter another (y or n)? y

Enter a height as an integral number of inches: 70
Do you want to enter another (y or n)? n

Average height is 61.2 inches.
There are 3 heights above average.
```

例子的说明

首先声明数组和其他两个计算时需要的变量：

```
int height[10];                                    // Array of heights
int count = 0;                                     // Number of heights
char reply = 0;                                    // Reply to prompt
```

height 数组的大小是 10，最多可以存储 10 个整数。变量 count 用于表示数组中下一个未使用的元素，因为第一个数组索引是 0，这也会反映到数组中存储的数据项个数上。最初，因为第一个元素是空的，没有存储数值，所以 count 是 0。

在一个 do-while 循环中读取身高值：

```
do {
   cout << endl
        << "Enter a height as an integral number of inches:  ";
   cin >> height[count++];

   // Check if another input is required
   cout << "Do you want to enter another  (y or n)? ";
   cin >> reply;
} while(count < 10 && std::tolower(reply)  ==  'y');
```

在显示提示信息后，从键盘上读取一个身高值，把它存储在 count 的当前值所引用的元素中。接着，递增变量 count，使之引用下一个未使用的元素。然后显示一个提示，确定是否输入更多的身高值。其响应将记录在 reply 中。do-while 循环条件比较已存储的数值个数与数组的大小，并检查 reply，以确定是否需要输入更多的元素。如果 count 的值为 10，即数组的元素个数为 10，或 reply 中的字符不是'y'或'Y'，就终止循环。

注意：

C++不检查索引值是否有效。程序员应确保没有引用数组上下限之外的元素。如果存储数据所使用的索引值在数组的有效范围之外，就会覆盖内存中的某个位置，或违反了存储保护规则。无论出现哪种情况，程序都肯定会输出不正确的结果。

在循环结束后，检查 count 的值，确定是否已填满了数组，如果已填满了，就显示一个消息，这会防止有人试图输入比数组能容纳的数据更多的数据项。然后，计算平均身高：

```
double average = 0.0;                              // Stores average height
```

```
for(int i = 0;  i < count ;  i++)
  average += height[i];                          // Add a height
average /= count;                               // Divide by the number of heights
cout << endl
     <<  "Average height is  "  << average <<  "  inches."
     << endl;
```

这段代码使用一个 for 循环在变量 average 中累加身高的值。该循环对 i 从 0 计算到 count-1，这就是元素中包含数据项的索引值。当 i 递增到 count 时，循环结束。要计算平均身高，可以对 average 中的累加值除以身高值的个数(在 count 中)。

最后一个计算是统计超过平均值的身高值个数：

```
int above_average = 0;                           // Count of above average heights
for(int i = 0 ;  i < count ;  i++)
   if(height[i]  > average)                     // Greater than average?
     above_ average++;                          // then increment the count
```

这段代码使用另一个 for 循环，比较每个身高与 average 中的值。如果该值大于 average，就递增 above_average 中的计数。最后，用下面的语句输出超过平均身高的元素个数：

```
cout <<  "There  "
     <<  (above_average == 1 ?  "is  "  :  "are  ")
     <<  above_average <<  "height"
     <<  (above_average == 1 ?  ""  :  "s ")
     <<  "above  average."
     <<  endl;
```

条件运算符会调整输出，以处理单数和复数之间的区别，即选择 is 或 are，以及是否在单词 height 的后面加上 s。

避免幻数

在第 2 章的程序中曾讨论了使用幻数的坏处，但在上一个例子中遗漏了一个重要的内容：height 数组的大小。为了避免出现这个问题，可以把数组的大小声明和初始化为一个常量：

```
const int max_heights=10;       //Array size
```

现在，可以把数组定义为 max_heights 指定的大小：

```
int height[max_heights];        //Array of heights
```

代码中有另外两个地方需要更新，以使用这个新的常量：do-while 循环中的循环条件，以及其后的 if 语句：

```
do {
   cout << endl
        << "Enter a height as an integral number of inches:  ";
   cin >> height[count++];

   // Check if another input is required
   cout << "Do you want to enter another  (y or n)? ";
   cin >> reply;
```

```
} while(count < max_heights && std::tolower(reply)=='y');

// Indicate when array is full
if(count == max_heights)
   cout << endl  << "Maximum height count reached." << endl;
```

现在程序中没有幻数了，如果要调整数组的大小，只需修改 max_heights 的初始值。注意这里必须把 max_heights 声明为 const，否则，编译器就不认为它是数组的大小。

数组的大小可以是一个常量整数表达式，但编译器必须把它计算为一个整数常量，才能给数组的元素分配合适的内存量。也就是说，表达式只能包含字面量、const 和枚举成员。

6.1.2 初始化数组

要初始化数组，元素的初始值应括在花括号中，且放在声明数组名后的等号之后。声明并初始化数组的例子如下所示：

```
int samples[5]={2,3,5,7,11};
```

列表中的值对应于数组的连续索引值，于是在本例中，samples[0]的值为 2，samples[1]的值为 3，samples[2]的值为 5，依此类推。花括号中的初始值列表称为初始化集合列表，或简称为初始化列表。数组在 C++中是集合的一个实例，还有其他集合实例。编译器有时在错误消息中使用这个行话。

不能指定比数组的元素个数还多的初始化值，但可以指定比数组的元素个数少的初始化值。如果指定的初始化值少于数组的元素个数，这些值就赋予从第一个元素(其索引为 0)开始的连续元素。没有提供初始值的数组元素则初始化为 0。这与没有提供初始化列表的情况有所不同。没有初始化列表，数组元素将包含垃圾值。

注意：

C++的语法允许使用空的初始化列表，在这种情况下，所有的元素都初始化为 0。但是，最好在该列表中放至少一个初始化值。只放置一个 0 也会使所有的元素设置为 0。

程序示例 6.2——初始化数组

下面用一个例子来演示上面的讨论，并输出两个数组中的值。

```
// Program 6.2 Initializing an array
#include <iostream>
#include <iomanip>
using std::cout;
using std::endl;
using std::setw;

int main() {
   const int size = 5;
   int values[size] = {1, 2, 3};
   double junk[size];

   cout << endl;
```

```
for(int i = 0 ; i < size ; i++)
    cout <<" "<< setw(12) << values[i];
cout << endl;

for(int i = 0 ; i < size ; i++)
    cout <<" "<< setw(12) << junk[i];
cout << endl;

return 0;
}
```

在这个例子中，声明了两个数组，第一个是 values 数组，它进行了部分初始化，第二个数组是 junk，它没有初始化。程序将输出两行代码，如下所示：

```
           1             2             3             0             0
4.24399e-314  2.2069e-312  1.11216e-306  1.81969e-306  1.99808e-307
```

输出的第二行(对应于 junk[0]到 junk[4]的值)在另一台计算机上可能是不同的。

例子的说明

从输出中可以看出，values 数组的前 3 个元素包含了初始化值，后两个元素则是默认值 0。在 junk 数组中，所有的值都是假的，因为没有为该数组提供任何初始值。数组元素包含了程序上一次使用这 3 个内存空间时遗留下来的值。

1. 把数组元素设置为 0

一般不是把数组设置为包含垃圾值，而是使用前面讨论的技术，把整个数组初始化为 0。在上面的例子中，可以用下面的语句把 junk 数组的所有元素都初始化为 0：

```
double junk[size]={0};        //Initialize all elements to zero
```

或语句：

```
double junk[size]={};         //Initialize all elements to zero
```

初始化值 0 总是会转换为数组的相应类型。在第一个语句中指定的显式值 0，可用于初始化第一个元素，其余元素也会设置为 0，因为它们没有初始化值。最好采用第一种形式，因为这会使编程的意图更清晰。

2. 用初始化列表定义数组的大小

如果在数组声明中提供了初始化值，就可以省略数组的大小。数组的元素个数就与初始化值的个数相同。例如，下面的数组声明：

```
int values[]={2,3,4};
```

这个语句定义了一个类型为 int 的数组，该数组有 3 个元素，其初始值分别是 2、3 和 4。这相当于：

```
int values[3]={2,3,4};
```

第一种形式的优点是数组的大小不会出错，因为编译器会决定数组的大小。但要知道，因为在 C++中，数组的大小不能为 0，所以，如果省略数组的大小，初始化列表就必须至少要包含一个初始化值。

3. 确定数组元素的个数

前面介绍了如何把数组的大小定义为一个常量，来避免元素的个数是一个幻数。当然，如果让编译器通过初始化列表来决定元素的个数，就不能这么做。

第 3 章说过，sizeof()运算符可以提供变量占用的字节数，它还可以确定数组中元素的个数。假定声明了一个数组：

```
int values[]={2,3,5,7,11,13,17,19};
```

表达式 sizeof values 就会计算出整个数组占用的字节数。表达式 sizeof values[0]则计算出一个元素占用的字节数，在本例中就是第一个元素，但数组中所有的元素都占用相同的字节数，因此表达式 sizeof values/ sizeof values[0]就等于数组中的元素个数。下面进行一下试验。

程序示例 6.3——获取数组的元素个数

下面是一个非常简单的程序，它使用了上面的技术：

```
//Program 6.3 Obtaining the number of array elements
#include <iostream>
using std::cout;
using std::endl;

int main() {
  int values[]={2,3,5,7,11,13,17,19,23,29};

  cout << endl
       << "There are "
       << sizeof values/ sizeof values[0]
       << " elements in the array. "
       << endl;

  int sum =0;
  for(int i=0; i< sizeof values / sizeof values[0]; sum+= value[i++])
    ;

  cout << "The sum of the array elements is "<<sum
       << endl;

  return 0;
}
```

这个例子的输出如下所示：

```
There are 10 elements in the array.
The sum of the array elements is 129
```

例子的说明

编译器会根据声明中初始化值的个数来确定数组元素的个数：

```
int values[]={2,3,5,7,11,13,17,19,23,29};
```

在数组声明后，就用下面的语句输出数组的元素个数：

```
cout << endl
     << "There are "
     << sizeof values/ sizeof values[0]
     << " elements in the array. "
     << endl;
```

如前所述，使用 sizeof()运算符计算元素的个数。该数组是 int 类型，可以用 sizeof(int)代替 sizeof values[0]，但这个表达式对任何类型的数组都会生成正确的元素个数。

为了证明这一点，下面在 for 循环条件中再次使用该表达式：

```
int sum =0;
for(int i=0; i< sizeof value / sizeof value[0]; sum+= values[i++])
  ;
```

这个循环在第三个控制表达式中累加数组的元素 sum+= value[i++]，在把当前元素加到 sum 中后递增计数器 i。循环语句本身是空的，在下一行上用一个分号来表示。

最后，用下面的语句输出 sum 的值：

```
cout << "The sum of the array elements is "<<sum
     << endl;
```

6.1.3 字符数组

char 类型的数组有两个含义。它可以是一个字符数组，每个元素存储一个字符；它也可以表示一个字符串。在后一种情况中，字符串中的每个字符存储在一个数组元素中，字符串的结尾用一个特定的字符串终止字符'\0'表示，该字符称为空字符。

用'\0'终止的字符串称为 C 样式的字符串，与在 C++标准库中定义的 string 类型相区别。类型 string 的实体不需要字符串终止字符，也灵活得多。目前在数组中，一般只考虑 C 样式的字符串，本章的最后将论及 string 类型。使用 string 类型进行字符串操作，要比使用 char 类型的数组更强大、更方便。

用下面的语句可以声明并初始化字符数组：

```
char vowels[5]={'a','e','i','o','u'};
```

这不是一个字符串，而只是一个包含 5 个字符的数组。数组的每个元素都用初始化列表中的对应字符进行初始化。与数值数组一样，如果提供的初始化值少于数组的元素个数，没有显式初始化值的元素就初始化为 0，即空字符，其所有的位都是 0，而不是字符'0'。

还可以让编译器把数组的大小设置为初始化值的个数：

```
char vowels[]={'a','e','i','o','u'};          //An array with five elements
```

这个语句也定义了一个包含 5 个字符的数组，并用初始化列表中的元音初始化这个数组。

还可以声明一个 char 类型的数组，并初始化为一个字符串字面量。例如：

```
char name[10]= "Mae West";
```

这里创建了一个 C 样式的字符串。由于用一个字符串字面量初始化了数组，应在该字符串的最后添加空字符，这样该数组的内容如图 6-2 所示。

'M'	'a'	'e'	' '	'W'	'e'	's'	't'	'\0'	'\0'

图 6-2　char 类型的数组的初始化元素

最后一个元素也设置为空字符，因为没有为它指定初始化值。也可以在用字符串初始化数组时，让编译器设置该数组的大小：

```
char name[]="Mae West";
```

这次，数组有 9 个元素；前 8 个元素存储字符串中的字符，最后一个元素存储字符串终止字符。当然，在声明 vowels 数组时，也可以使用这个方法：

```
char vowels[]="aeiou";          //An array with six elements
```

这与前面声明的 vowels 数组有明显的区别：这里用一个字符串字面量初始化了数组。因为在字符串的最后添加了一个'\0'，来标记字符串的结束，所以 vowels 数组包含 6 个元素。前面声明所创建的数组只包含 5 个元素，而且不能用作字符串。

使用数组名可以显示存储在该数组中的字符串。例如用下面的语句可以显示 name 数组中的字符串：

```
std::cout <<name<<std::endl;
```

这将显示整个字符串，直到'\0'，最后必须是'\0'。如果不是，就会继续输出后续内存空间中存储的字符，直到遇到字符串终止字符为止，或者出现不合法的内存引用。

注意：

只使用数组名不能输出数值类型的数组的内容。这个方法只适用于 char 数组。

程序示例 6.4——分析字符串

下面用一个例子来说明如何使用 char 类型的数组。这个程序读取一行文本，确定其中使用了多少元音和辅音。

```
// Program 6.4 Analyzing the letters in a string
#include <iostream>
#include <cctype>
using std::cout;
using std::cin;
using std::endl;

int main() {
  const int maxlength = 100;                    // Array dimension
  char text[maxlength] = {0};                   // Array to hold input string

  cout << endl << "Enter a line of text: " << endl;
```

```
// Read a line of characters including spaces
cin.getline(text, maxlength);

cout << "You entered:" << endl << text << endl;

 int vowels = 0;                              // Count of vowels
 int consonants = 0;                          // Count of consonants
 for(int i = 0 ; text[i]  != '\0'  ; i++)
   if(std::isalpha(text[i]))                        // If it is a letter
     switch(std::tolower(text[i])) {                // Test lower case version

       case  'a':  case  'e':  case  'i':
       case  'o':  case  'u':
         vowels++;                              // It is a vowel
         break;
       default:
          consonants++;                         // It is a consonant
     }

cout << "Your input contained"
     <<  vowels      <<  "  vowels  and  "
     <<  consonants  <<  "  consonants."
     << endl;

 return 0;
}
```

下面是这个程序的输出：

```
Enter a line of text:
A rich man is nothing but a poor man with money.
You entered:
A rich man is nothing but a poor man with money.
Your input contained 14 vowels and 23 consonants.
```

例子的说明

首先声明一个 char 类型的数组，它的元素个数由 const 变量来定义：

```
const int maxlength=100;          //Array dimension
char text[maxlength]={0};         //Array to hold input string
```

在 text 数组中存储输入内容。但是，不能使用通常的方法获取输入，即不能使用提取运算符(>>)，因为它在这些环境下并不能完成我们希望完成的操作。考虑下面的语句：

```
cin >> text;
```

这肯定会把字符读入 text 数组，但只能读到第一个空格为止。因为提取运算符把空格看作是输入值之间的分隔符，所以不能读取包含空格的整个字符串。甚至不能使用提取运算符一次读取一个输入字符，因为任何空白字符，包括'\n'都会被看做是分隔符。也就是说，不能存储换行符，因此不能使用它表示字符串的结尾。要读取一整行文本(包括空格)，需要使用另一种可用于标准输入流的方法。

因此，在提示输入后，用下面的语句读取标准输入流：

```
cin.getline( text, maxlength);
```

cin 流的 getline()函数读取并存储一整行字符，包括空格。在读取了换行符'\n' (即按下回车键)后输入结束。

getline()函数需要两个参数。输入存储在第一个参数指定的位置上，在本例中，就是 text 数组。第二个参数是要存储的最大字符数。这个计数包括字符串终止字符'\0'，该字符会自动追加到输入字符串的末尾。

getline()函数还有一个可选参数，它允许指定'\n'的替代字符，来表示输入的结束。例如，如果输入一个感叹号来表示字符串的结束，就可以使用下面的语句：

```
cin getline( text, maxlength,'!');
```

为什么要这么做？主要原因是允许输入多行文本。用'!'代替'\n'，表示输入的结束，就可以输入任意多行文本，包括'\n'字符。在完成时输入'!'即可。当然，输入的总字符数仍由 maxlength 限制。

回到例子中，为了说明所进行的操作，下面用数组名输出刚才输入的字符串：

```
cout <<" You entered: "<< endl<<text<<endl;
```

读取并显示了输入行后，就要以相当直接的方式分析文本字符串了：

```
int vowels = 0;                                  // Count of vowels
int consonants = 0;                              // Count of consonants
for(int i = 0 ;  text[i]  !=  '\0'  ;  i++)
  if(std::isalpha(text[i]))                      // If it is a letter
     switch(std::tolower(text[i])) {             // Test lowercase version

        case 'a':   case  'e':  case  'i':
        case 'o':  case   'u':
           vowels++;                             // It is a vowel
           break;
        default:
           consonants++;                         // It is a consonant
      }
```

这段代码在声明的两个变量中累加元音和辅音的个数。for 循环条件测试是否找到字符串终止字符，而不是测试计数器是否到达某个极限值。在循环中，使用 isalpha()库函数检查字符是否为字母。如果找到一个字母，就把该字母的小写形式用作 switch 语句中的选择表达式。这可以避免同时编写大写字母和小写字母的 case。因为，如果 text[i]是一个字母，就进入 switch 语句，而不是元音的字母就一定是辅音，所以可以在默认操作中累加 consonants。

最后，用下面的语句输出累加的元音和辅音计数：

```
cout << "Your input contained"
     << vowels      <<  "vowels and"
     << consonants  <<  "  consonants."
     << endl;
```

6.2 多维数组

前面声明的数组都只需要一个索引值来选择元素。这种数组称为一维数组，因为改变一个索引就可以引用所有的元素。也可以声明需要两个或更多索引值才能访问元素的数组，这种数组一般称为多维数组。需要两个索引值来引用元素的数组称为二维数组。需要三个索引值的数组称为三维数组，依此类推。

假定有一个园丁，要记录在小菜园中种植的每颗胡萝卜的重量。这些胡萝卜的种植方式为三行四列，为了存储每颗胡萝卜的重量，可以声明一个二维数组：

```
double carrots[3][4];
```

要引用 carrots 数组中的元素，需要两个索引值，第一个索引值指定行，其值为从 0 到 2，第二个索引值指定该行中的列，其值为从 0 到 3。要存储第二行第三列的胡萝卜重量，可以使用下面的语句：

```
carrots[1][2]=1.5;
```

这个数组在内存中的存储空间如图 6-3 所示。

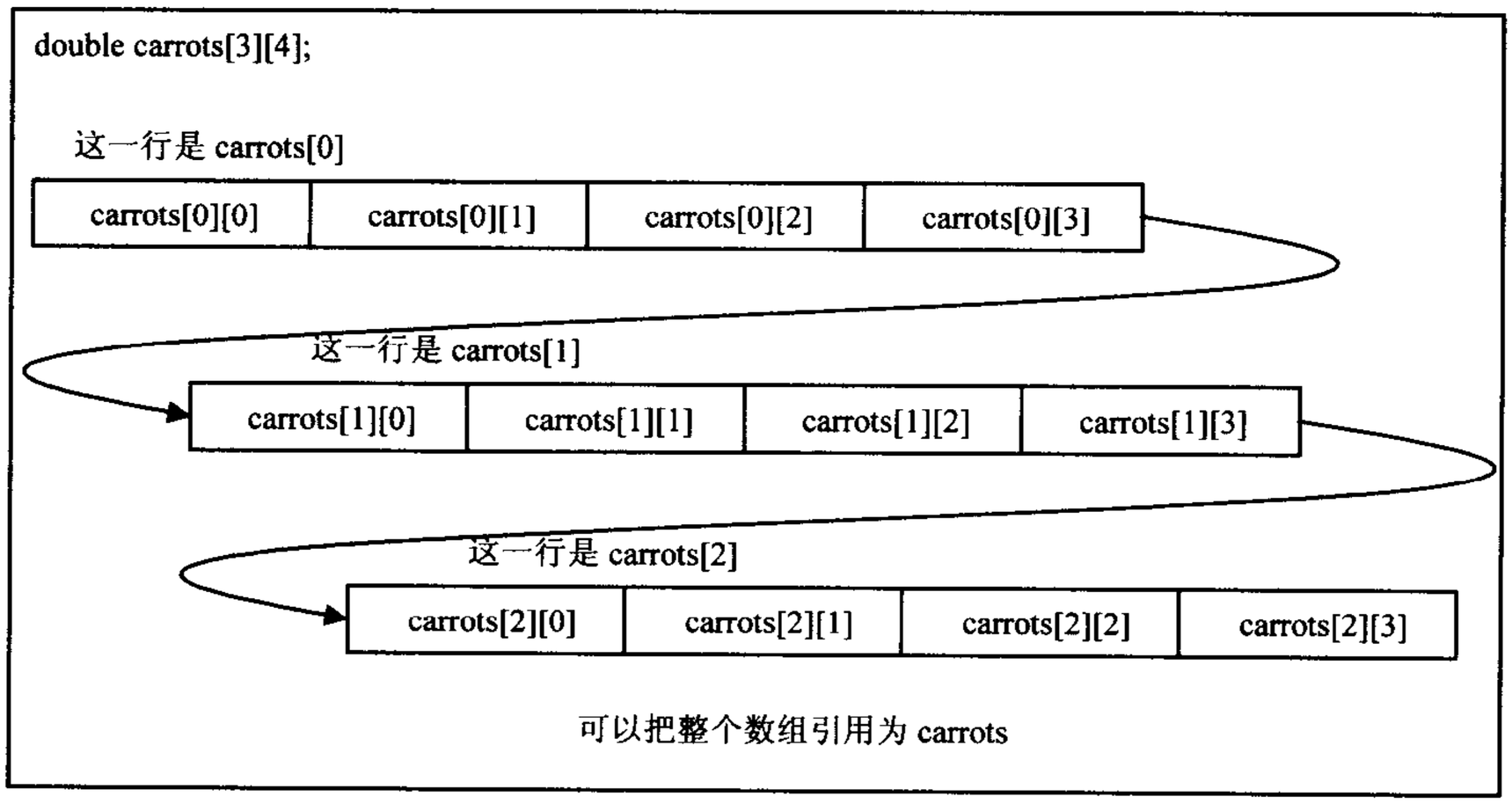


图 6-3　二维数组中的元素

这些行存储在一个连续的内存块中。可以看出，二维数组实际上是一个有 3 个元素的一维数组，每个一维数组都有 4 个元素。于是，该二维数组就变成大小为 4 的 3 个一维数组。

在图 6-3 中，可以使用数组名，后跟一个放在方括号中的索引值，表示该数组中的一整行。第 8 章讨论函数时将经常使用这种方法。

在引用元素时，要使用两个索引值。右边的索引值从一行中选择元素，变化得较快。如果从左至右读取数组，右边的索引就对应列号。左边的索引值选择行，因此表示行号。图 6-4 演示了这个概念。对于多维数组，最右边的索引值总是变化最快的一个，最左边的索引则变化最慢。

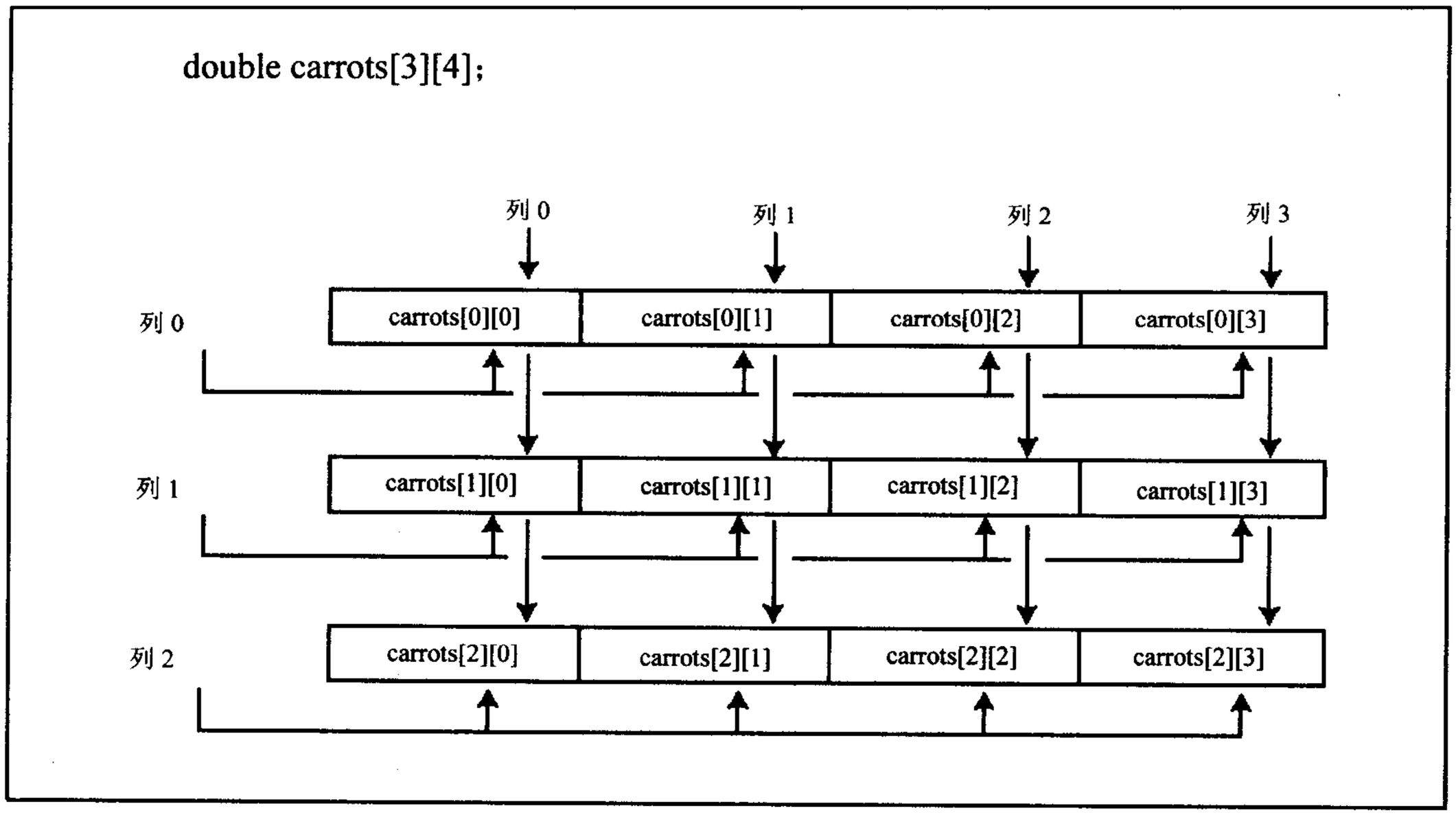


图 6-4　二维数组中的行和列

数组名本身引用整个数组。注意，对于此数组，不能利用这种符号表示法显示一整行或整个数组的内容。例如，下面的代码行：

```
std::cout << carrots;      //Not what you may expect!
```

会输出一个十六进制值，它是数组中第一个元素的内存地址。第 7 章在讨论指针时会说明其原因。char 类型的数组有点不同，如前所述。

要一行一行地显示整个数组的内容，可以使用下面的语句：

```
for(int i = 0;  i < 3 ; i++) {
  for(int j = 0 ; j < 4 ; j++)
    std::cout << std::setw(12) << carrots[i][j];
  std::cout << std::endl;
}
```

这段代码使用了幻数 3 和 4，而使用 sizeof()运算符可以避免使用它们：

```
for(int i=0 ; i < sizeof carrots / sizeof carrots[0] ; i++) {
  for(int j = 0 ; j < sizeof carrots[0] / sizeof(double) ; j++)
    std::cout << std::setw(12)  << carrots[i][j];
  std::cout << std::endl;
}
```

当然，在数组的维数上最好不要使用幻数，下面把这些代码编写得更简洁一些：

```
const int nrows = 3;    //Number of rows in the array
const int ncols = 4;    //Number of columns which is number of elements per row
double carrots[nrows, ncols];
// Code to set values for the elements...
```

```
for(int i = 0 ; i < nrows ; i++){
  for(int j = 0 ; j < ncols ; j++)
    std::cout << std::setw(12)  << carrots[i][j];
  std::out << std::endl;
}
```

要声明一个三维数组，只是再添加一对方括号。例如，如果在一星期的七天中，每天都要记录三个温度，一年就是 52 个星期，则可以声明下面的数组，把该数据存储为 long 类型：

```
long temperatures[52][7][3];
```

该数组在每一行中存储 3 个值，一星期的数据就要存储 7 行，一年就要存储 52 个这样的 7 行数据。这个数组一共有 1092 个 long 类型的元素。要显示第 26 个星期的第 3 天的中间温度，就可以编写下面的代码：

```
std::cout << temperatures[25][2][1];
```

因为所有的索引值都从 0 开始，所以星期数是从 0 到 51，天数是从 0 到 6，一天中的温度数据是从 0 到 2。

6.2.1 初始化多维数组

为多维数组指定初始值的方法派生于二维数组是一维数组的数组这一想法。一维数组的初始值放在花括号中，用逗号分隔开。根据这种方法，用下面的语句可以声明并初始化 carrots 数组：

```
double carrots[3][4]  = {
                          {2.5,  3.2,  3.7,  4.1},     // First row
                          {4.1,  3.9,  1.6,  3.5},     // Second row
                          {2.8,  2.3,  0.9,  1.1}      // Third row
                        };
```

这里使用明确的数组大小使代码比较短小和简单。因为每一行都是一个一维数组，所以每行的初始化值就包含在一对花括号中。这 3 个初始化列表本身包含在一对花括号中，因为二维数组是一维数组的一维数组。可以把这个规则扩展到任意维数——每增加一维，就需要另一对花括号包含初始值。

问题是，如果省略某些初始值，会发生什么？从过去的经验可以推断出结果。每个最内层的花括号对都包含了行中元素的值。第一个列表对应于 carrots[0]，第二个列表对应于 carrots[1]，第三个列表对应于 carrots [2]。每对花括号包含的值都赋予相应行中的元素。如果没有足够的初始化值赋予一行中的所有元素，则没有值的元素就初始化为 0。下面看一个例子：

```
double carrots[3][4]  = {
                          {2.5,  3.2          },     // First row
                          {4.1                },     // Second row
                          {2.8,  2.3,  0.9    }      // Third row
                        };
```

第一行的前两个元素有初始值，第二行只有一个元素有初始值，第三行的 3 个元素有初始值，因此这些元素初始化为图 6-5 所示。

carrots [0][0]	carrots[0][1]	carrots[0][2]	carrots[0][3]
2.5	3.2	0.0	0.0
carrots[1][0]	carrots[1][1]	carrots[1][2]	carrots[1][3]
4.1	0.0	0.0	0.0
carrots[2[0]	carrots[2][1]	carrots[2][2]	carrots[2][3]
2.8	2.3	0.9	0.0

图 6-5　二维数组省略了初始值

可以看出，行中没有指定初始值的元素都设置为 0。如果没有使用足够的花括号对来初始化数组中的所有行，行中没有初始值的元素就设置为 0。在逻辑上，可以用下面的语句把数组中的所有元素都设置为 0：

```
double carrots[nrows][nclos]={0};
```

如果在初始化列表中包含几个初始值，但省略包含行中值的嵌套花括号，值就按顺序赋予元素，存储在内存中，最右边的索引变化较快。例如，假定声明一个数组，如下所示：

```
double carrots[3][4]={1.1,1.2,1.3,1.4,1.5,1.6,1.7};
```

设置了数值的数组如图 6-6 所示。

carrots [0][0]	carrots[0][1]	carrots[0][2]	carrots[0][3]
1.1	1.2	1.3	1.4
carrots[1][0]	carrots[1][1]	carrots[1][2]	carrots[1][3]
1.5	1.6	1.7	0.0
carrots[2][0]	carrots[2][1]	carrots[2][2]	carrots[2][3]
0.0	0.0	0.0	0.0

图 6-6　用一维列表值初始化二维数组

初始值沿着行按顺序赋予元素。如果没有更多的值，剩余的元素就初始化为 0。

在默认情况下设置维数

可以让编译器根据初始化值的设置，决定数组的第一(最左边)维的大小。显然，编译器只能确定多维数组中第一维的大小。例如，如果为二维数组提供了 12 个初始值，编译器就不知道数组是有三行四列，六行二列，还是 12 个元素的其他组合。

用下面的语句可以声明二维 carrots 数组：

```
double carrots[][4] = {
```

```
                                {2.5, 3.2          },        // First row
                                {4.1               },        // Second row
                                {2.8, 2.3, 0.9     }         // Third row
                               };
```

这个数组与以前一样有三行，因为在外层的花括号对中有三组括号对。如果只有两组花括号对，数组就只有两行，因此下面的语句：

```
double carrots[][4] = {
                                {2.5,  3.2         },        // First row
                                {4.1               }         // Second row
                               };
```

创建的数组与下述语句声明的数组一样：

```
double carrots[2][4] = {
                                 {2.5,  3.2        },        // First row
                                 {4.1              }         // Second row
                                };
```

更多维的数组可以采用这种方式声明，让编译器根据初始化值的设置来确定第一维的大小。下面是一个三维数组声明的例子：

```
int numbers[][3][4] = {
                                {
                                  { 2,   4,   6,   8},
                                  { 3,   5,   7,   9},
                                  { 5,   8,  11,  14}
                                },
                                {
                                  {12,  14,  16,  18},
                                  {13,  15,  17,  19},
                                  {15,  18,  21,  24}
                                }
                               };
```

这个数组有三维，其大小分别是 2、3 和 4。外层花括号对包含了两个内层花括号对，而每个内层花括号对又包含了三个花括号对，每个花括号对包含了对应行的 4 个初始值。从这个例子可以看出，处理三维或更多维的数组变得相当复杂，在花括号中放置初始化值时需要特别小心。花括号对的嵌套次数就是数组的维数。

6.2.2 多维字符数组

可以声明二维或更多维的数组来存储任意类型的数据。char 类型的二维数组非常有趣，它可以存储一组 C 样式的字符串。在用包含在双引号中的字符串初始化 char 类型的二维数组时，不需要对每行字符串加上花括号——有双引号就足够了。例如：

```
char stars[][80]  = {
                                "Robert Redford",
                                "Hopalong Cassidy",
```

```
                "Lassie",
                "Slim Pickens",
                "Boris Karloff",
                "Oliver Hardy"
              };
```

这个数组有六行，因为它把六个字符串常量作为初始值。数组中的每一行都存储了一个字符串，其中包含一个电影明星的名字。而且每个字符串都追加了终止空字符'\0'。根据所指定的行大小，每一行都占用了 80 个字符。

程序示例 6.5——使用二维字符数组

下面在一个例子中演示这类数组。这个程序将根据输入的整数选择幸运之星：

```
// Program 6.5 Storing strings in an array
#include <iostream>
using std::cout;
using std::cin;
using std::endl;

int main() {
    char stars[][80]  =  {
                            "Robert Redford",    "Hopalong  Cassidy",
                            "Lassie",            "Slim Pickens",
                            "Boris Karloff",     "Mae  West",
                            "Oliver Hardy",      "Sharon  Stone"
                          };
int choice = 0;

cout << endl
     << "Pick a lucky star!"
     <<  "  Enter  a  number between  1  and  "
     << sizeof stars / sizeof stars[0]  << ":  ";
cin >> choice;


if(choice >= 1 && choice <= sizeof stars / sizeof stars[0])
   cout << endl
        << "Your lucky star is" << stars[choice - 1];
else
   cout << endl                          // Invalid input
        <<  "Sorry,  you haven't got a lucky star.";

    cout << endl;
    return 0;
}
```

这个程序的输出如下所示：

```
Pick a lucky star! Enter a number between 1 and 8: 6

Your lucky star is Mae West
```

例子的说明

除了内在的娱乐价值之外。这个例子的主要看点是声明了数组 stars，这是一个二维 char 数组，可以存储多个字符串，每个字符串至多可以包含 80 个字符，包括编译器自动添加的终止空字符。数组的初始化字符串放在花括号对中，用逗号分隔开：

```
char stars[] [80]  = {
                        "Robert Redford",    "Hopalong Cassidy",
                        "Lassie",            "Slim Pickens",
                        "Boris Karloff",     "Mae West",
                        "Oliver Hardy",      "Sharon Stone"
                      };
```

因为这里省略了第一维数组的大小，所以编译器会用容纳所有初始化字符串所需要的行数创建该数组。如前所述，只能省略第一维的大小。其他维的大小必须指定。

用下面的语句提示输入一个整数：

```
cout << endl
     << "Pick a lucky star!"
     << " Enter a number between 1 and"
     << sizeof stars /  sizeof stars[0]  << ": " ;
```

所输入的整数的上限由表达式 sizeof stars /sizeof stars[0]给定。因为它指定了数组中的行数，所以该语句会自动适应对数组中名字个数的修改。在 if 语句中也使用这个表达式来安排输出的显示：

```
if(choice >= 1 && choice <= sizeof stars / sizeof stars[0])
   cout << endl
        << "Your lucky star is "<< stars[choice - 1];
else
   cout << endl                              // Invalid input
        << "Sorry, you haven't got a lucky star.";
```

if 条件检查输入的整数是否超出范围，之后显示一个名字。在语句中需要引用一个字符串进行输出时，只需指定第一个索引值。一个索引值可以选择某个特定的、有 80 个元素的子数组，因为这包含了一个字符串，所以输出操作会显示该字符串的内容，直到终止空字符为止。索引指定为 choice - 1，因为 choice 值从 1 开始，而索引值用于从数组中选择一个名字，显然需要从 0 开始。在使用数组进行编程时，这是很常见的。

注释：

像本例这样使用数组的一个缺点是，会有大量未使用的内存。所有的字符串都少于 80 个字符，数组中每一行上的过剩元素都被浪费了。第 7 章将介绍一种处理这种情况的更好方法。

6.3 string 类型

前面介绍了 char 类型的数组可以用于存储非空(C 样式)字符串，还有一种更好的方法。<string>头文件定义了 string 类型，该类型比非空字符串更易于使用。string 类型由一个类定义(更

准确地说，是类模板)。目前还没有讨论过类，但这对于理解本节的内容来说并不会有什么困难，因为实际上，类只是把新类型引入语言而已。使用类定义的类型与使用基本数据类型并没有区别，所以这里不提及这些，而是直接使用它们，它们的工作方式将在以后论述。

类类型的实体通常称为对象(而不是变量)，下面讨论 string 类型时就使用这个术语。本节有一两个问题目前还不能完全解释清楚，但可以比较 string 类型和非空字符串，明白为什么使用前者一般比较好。

前面介绍的整型和浮点类型(不包括枚举)称为基本类型，string 类型不是基本数据类型，而称为复合类型。复合类型一般是组合了若干个数据项的类型，这些数据项最终都是根据基本数据类型定义的。string 对象包含的字符构成了它表示的字符串，还可以包含其他数据，例如字符串中的字符个数，枚举和数组都是复合类型。

本节开始时说过，string 类型在<string>头文件中定义，所以在使用 string 对象时总是要包含这个头文件。string 类型名称也在 std 命名空间中定义，所以需要一个 using 声明，才能以未限定的形式使用类型名称。string 类型几乎总是以未限定的形式使用，所以下面假定包含了一个 using 声明，在代码段中使用 string，而不是 std::string。下面先介绍如何创建 string 对象。

6.3.1 声明 string 对象

string 类型的对象包含 char 类型的字符串，该字符串可以为空。用下面的语句可以声明 string 类型的对象，它表示一个空字符串：

```
string myString;        // An empty string
```

这个语句声明了一个 string 对象，可以使用名称 myString 来引用它。在本例中，myString 是一个空 string 对象，即表示一个不包含字符的字符串，其长度为 0。

还可以使用字符串字面量来声明并初始化 string 对象：

```
string proverb="Many a mickle makes a muckle.";
```

其中，proverb 是一个 string 对象，表示字符串字面量所示的字符串。改写前面的语句，还可以使用函数表示法初始化对象：

```
string proverb("Many a mickle makes a muckle.");
```

所存储的字符串不需要字符串终止字符。string 对象会跟踪它表示的字符串长度。使用不带参数的 length()函数，就可以获取 string 对象的字符串长度。例如：

```
std::cout<< proverb. length();
```

这个语句调用 proverb 对象的 length()函数，使用 cout 流的插入运算符输出返回的值。从而显示存储在 proverb 中的字符串长度，在本例中是 29，即字符串中字符的个数。

表达式 proverb.length()中的句点称为成员访问运算符，或简称为句点运算符。它把函数 length()标识为 proverb 对象的一个成员。第 11 章在探讨如何定义自己的数据类型时，将详细讨论其含义。

初始化 string 对象还有其他方法。不能用一个括在单引号中的字符来初始化 string 对象，字符串常量必须总是放在双引号中，即使字符串只有一个字符也是如此。但可以用同一个字符的许多实例(包括一个实例！)来初始化 string 对象。例如，可以用如下语句声明并初始化一个睡眠时间 string 对象：

```
string sleeping(6,'z');
```

string 对象 sleeping 包含字符串"zzzzzz"。如果希望 string 对象适用于浅度睡眠时间，只包含一个'z'，就可以使用下面的代码：

```
string light_sleep(1,'z');
```

这会用字符串字面量"z"来初始化 light_sleep。但不能使用下面的语句：

```
string light_sleep='z';          //Wrong! Won't compile!
```

初始化 string 对象的另一个方法是使用已有的 string 对象。假定 proverb 已声明，就可以使用下面的语句声明另一个对象：

```
string sentence=proverb;
```

因为 sentence 对象用与 proverb 相同的字面量字符串来初始化，所以它也包含"Many a mickle makes a muckle."。还可以在这个语句中使用函数表示法，把前面的声明改写为：

```
string sentence(proverb);
```

string 对象中的字符从 0 开始索引，与数组一样。因此，可以选择已有 string 对象中的一部分，用它初始化另一个 string 对象。例如，下面的代码行：

```
string phrase(proverb,0,13);
```

会把 proverb 对象的一部分放在新的 string 对象 phrase 中，如图 6-7 所示。

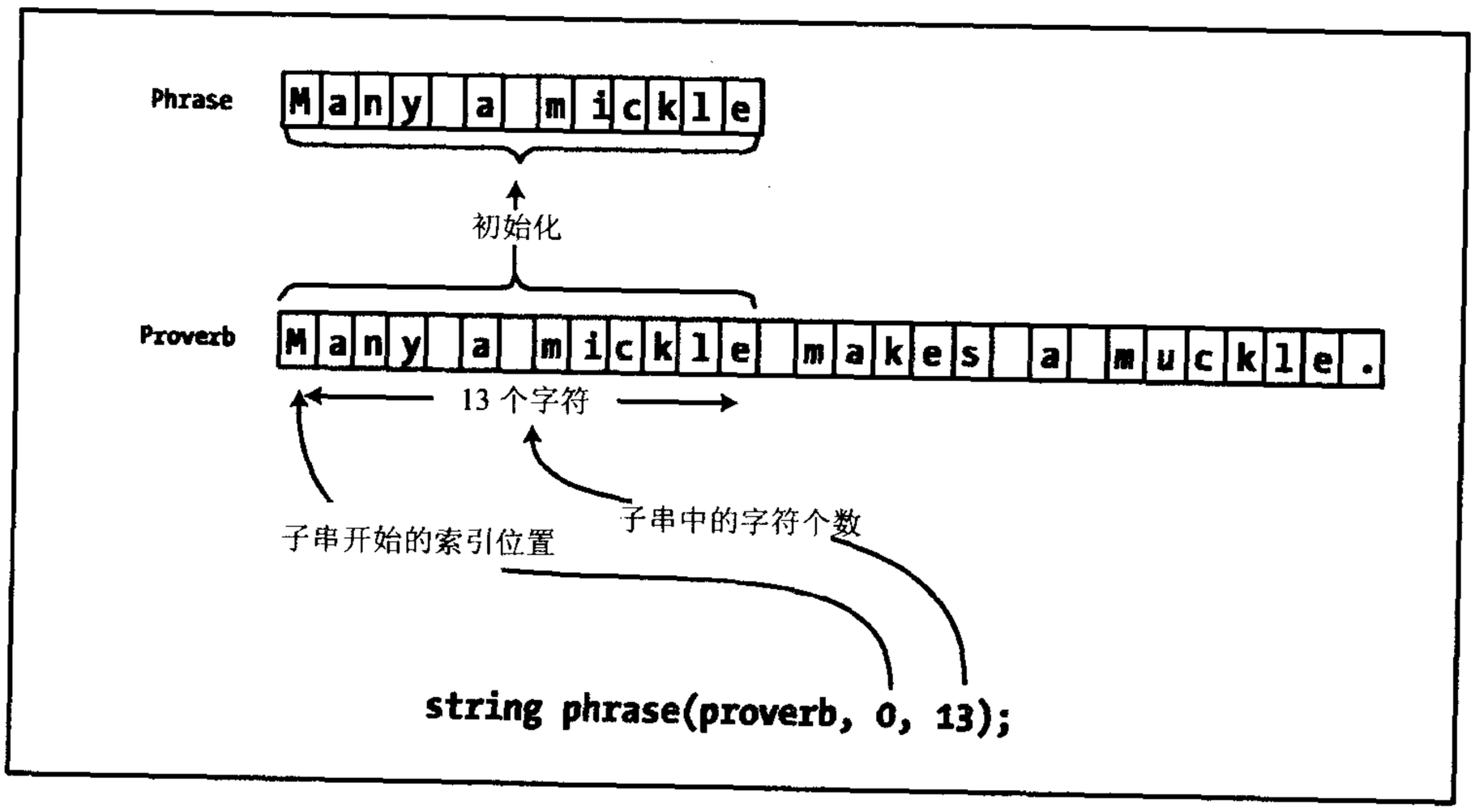


图 6-7　从已有字符串中创建新字符串

括号中的第一个参数是要用作初始化源的 string 对象的名称。第二个参数是 string 对象的起始索引，第三个参数是要用作初始值的字符个数。所以这里从 proverb 的第一个字符(索引位置为 0)开始选择前 13 个字符，作为 phrase 对象的初始值，即 phrase 包含"Many a mickle"。

6.3.2 使用 string 对象

string 对象的最简单操作是赋值。可以把字符串字面量的值或一个 string 对象赋予另一个 string 对象。例如：

```
string adjective="hornswoggling";        //Declares a string
string word="rubbish";                   //Declares another string

word= adjective;                         //Modified word
adjective="twotiming";                   //Modified adjective
```

第三个语句把 adjective 的值 hornswoggling 赋予 word，替换掉了 rubbish。最后一个语句给 adjective 赋予了一个新的字符串字面量 twotiming，替换掉了初始值 hornswoggling。这样，word 就包含 hornswoggling，adjective 包含 twotiming。

连接字符串

使用加号运算符可以连接字符串，其技术术语是“连接”。下面用刚才定义的对象来演示连接：

```
string description=adjective+" "+word+" "+"whippersnapper";
```

执行这个语句后，description 对象就包含字符串"twotiming hornswoggling whippersnapper"。显然，使用+运算符可以方便地把 string 对象的字符串字面量连接在一起。这是因为+运算符重新进行了定义，对 string 对象有了一个特殊的含义。当一个操作数是 string 对象，另一个操作数是另一个 string 对象或一个字符串字面量时，+操作的结果就是一个新的 string 对象，它把两个字符串连接为一个字符串。

注意不能用+运算符来连接字符串字面量。+运算符的一个操作数必须是 string 类型的对象。例如下面的语句就不会编译：

```
string description=" whippersnapper" +" "+word; // Wrong!!
```

其问题是编译器试图把等号右边的表达式计算为(("whippersnapper"+" ")+word)。+运算符不能操作两个字符串字面量。但是，正确使用括号就可以使该语句合法：

```
string description=" whippersnapper" +(" "+word);
```

其中，括号中的+运算合法，生成 string 类型的值，该值可以和字符串字面量"whippersnapper"连接在一起。

程序示例 6.6——连接字符串

讨论完了理论，下面就该进行实践了。该程序将从键盘上读取姓和名：

```
// Program 6.6 Concatenating strings
```

```
#include <iostream>
#include <string>

using std::cout;
using std::cin;
using std::endl;
using std::string;

int main() {
    string first;                                   // Stores the first name
    string second;                                  // Stores the second name

cout << endl <<  "Enter your first name:  ";
cin >> first;                                       // Read first name

cout <<  "Enter your  second name:  ";
cin >> second;                                      // Read second name

string sentence = "Your full name is ";             // Create basic sentence
sentence += first +"  "+ second +" . ";             // Augment with names

cout << endl
     << sentence                                    // Output the sentence
     << endl;
cout << "The string contains"                       // Output its length
     << sentence.length( )
     <<  "  characters."
     << endl;

 return 0;
}
```

该程序的输出如下所示：

```
Enter your first name:  Phil
Enter your second name: McCavity

Your full name is Phil McCavity.
The string contains 32 characters.
```

例子的说明

首先，用下面的语句声明两个 string 对象：

```
string first;          //Stores the first name
string second;         //Stores the second name
```

由于这里没有指定初始值，因此它们就初始化为空字符串。接着提示输入姓名，然后从键盘中读取姓名：

```
cout <<  endl  <<  "Enter  your  first  name:";
cin  >>  first;                                 //  Read  first  name

cout <<  "Enter  your  second  name:";
cin  >>  second;                                //  Read  second  names
```

流提取运算符>>操作 string 对象的方式与它操作 char 类型的数组相同。因为读取字符，直到遇到第一个空白字符为止，所以不能以这种方式读取包含空格的字符串。稍后介绍这个问题的一种解决方法。

在获取姓名后，创建另一个 string 对象，但这次用一个字符串字面量显式初始化它：

```
string sentence="Your full name is ";          //Create basic sentence
```

对象 sentence 用指定的字符串初始化。然后使用这个对象装配一个包含要显示的消息的字符串：

```
sentence += first + " "+ second + ".";         //Augment with names
```

等号右边把 first 和字面量""连接起来，再连接 second，最后是字面量"."。根据前面的输入，右边应等于一个包含"Phil McCavity. "的 string 对象。

如这个语句所示，+=运算符也可以用于 string 类型的对象，其使用方式与前面介绍的基本类型相同，这个语句也等价于：

```
sentence = sentence + (first + " "+ second + ".");         //Augment with names
```

所以+=运算符的结果是把计算右边表达式得到的 string 对象和左边的 sentence 对象连接起来，再把最终的结果存储在 sentence 中。在执行完这个语句后，sentence 对象就包含"Your full name is Phil MoCavity. "。

在使用+=运算符把一个值追加给 string 对象时，右边可以是结果为非空字符串的表达式，或 char 类型的单个字符，或生成 string 类型的对象的表达式。

最后，该程序使用流插入运算符输出 sentence 的内容及字符串的长度：

```
cout << endl
     << sentence                                  // Output the sentence
     << endl;
cout << "The string contains"                     // Output its length
     << sentence.length()
     <<  "  characters."
     << endl;
```

如这个语句所示，输出 string 对象的值与输出其他类型的变量值一样。

6.3.3 访问字符串中的字符

在方括号中使用索引值，就可以引用字符串中的某个字符，就像处理数组一样。string 对象中的第一个字符的索引值是 0。例如，sentence 中的第三个字符可以引用为 sentence[2]。还可以在赋值运算符的左边使用这样的表达式，在访问字符串的同时修改某些字符。下面的语句会把 sentence 中的所有字符改为大写形式：

```
for(int i=0; i < sentence.length(); i++)
   sentence[i] = std::toupper(sentence[i]);
```

这会把 toupper()函数依次应用于字符串中的每个字符，再把得到的结果依次存储回字符串

原来的位置。第一个字符的索引值是 0，最后一个字符的索引值比字符串的长度小 1，所以只要 i < sentence.length()是 true，循环就会继续。

注意采用这种编写循环的方式，编译器会发出一个警告消息。循环变量 i 的类型是 int，而 length()函数返回的整数是 size_t 类型，也就是说，在比较操作中可能出现带符号或不带符号不匹配的情况。这不会造成任何问题，但如果要避免出现这个警告，可以按照如下方式编写循环：

```
for(size_t i=0; i < sentence.length(); i++)
  sentence [i] = std::toupper(sentence [i]);
```

在前面确定字符串中元音和辅音个数的程序中，试用这个数组样式的访问方法。新版本将使用 string 对象。

程序示例 6.7——访问字符

这个程序与程序 6.4 完全相同，只是用 string 对象代替了 char 类型的数组：

```
// Program 6.7 Accessing characters in a string
#include <iostream>
#include <string>
#include <cctype>
using std::cout;
using std::cin;
using std::endl;
using std::string;

int main ( ) {
   string text;                                    // Stores the input

   cout << endl <<  "Enter a line of text:"  << endl;

   // Read a line of characters including spaces
  std::getline(cin, text);

   int vowels = 0;                                 //Count of vowels
   int consonants = 0;                             //Count of  consonants
   for(int i = 0 ; i < text.length()  ; i++)
    if(std::isalpha(text[i]) )                     //Check for a  letter
         switch (std::tolower(text[i])) {          //Test  lowercase
            case  'a': case  'e':  case  'i':
            case  'o': case  'u':
               vowels++;
               break;
           default:
               consonants++;
     }

cout <<  "Your input contained"
     <<  vowels      << " vowels  and  "
     <<  consonants  << " consonants."
     <<  endl;
```

```
    return 0;
}
```

该程序的输出如下所示：

```
Enter a line of text:
A nod is as good as a wink to a blind horse.

Your input contained 14 vowels and 18 consonants.
```

例子的说明

在编译这个例子时，编译器会对下面的语句发出一个警告：

```
for(int i = 0 ; i < text.length()  ; i++)
```

出现这个警告是因为，length()函数返回的整数是 size_t，它定义为 unsigned 整数类型。如果要消除这个警告，可以把循环变量声明为 size_t:

```
for(size_t i = 0 ; i < text.length()  ; i++)
```

在正常情况下，可以忽略这个消息，有这个警告例子也可以运行。类似的警告也会出现在 string 对象的其他返回值为字符串中位置的函数中。

下面仅解释程序的新特性。首先声明 string 类型的对象：

```
string text;          //Stores the input
```

string 类型的 text 对象最初包含空字符串，但我们要从键盘上读取一行，并存储在 text 对象中。在显示提示后，使用 getline()函数获取输入的内容：

```
getline(cin,text);
```

这个版本的 getline()在<string>头文件中声明，它从第一个参数指定的流 cin 中读取字符，直到换行符为止，并把该行输入存储在第二个参数指定的 string 对象 text 中。注意这次不需要考虑输入中有多少个字符。string 对象会自动包容所输入的内容。

如果要把表示输入行结尾的分隔符改为不是'\n'的其他字符，可以使用带有 3 个参数的 getline()，第三个参数指定了表示输入行结尾的分隔符：

```
getline(cin,text.'#');
```

该行代码会查找'#'，作为表示输入行结尾的字符。此时，换行符不表示输入的结束，所以可以输入任意多行内容，它们会合并到一个字符串中。但换行符仍会在字符串中显示出来。

读取文本后，就计算字符串中元音和辅音的字符个数，采用的方法与前面的一样，但在 for 循环中，要对循环条件作一点修改：

```
for(int i=0; i<text.length();i++)
```

使用 text 对象的 length()函数，获取字符串中的字符个数，这个数字控制着 for 循环。也可以通过索引值 i 访问每个字符，就像处理 char 数组一样。

在这个例子中，使用 string 对象的主要优点是不需要考虑该对象包含的字符串的长度。

6.3.4 访问子字符串

使用 substr()函数可以获取 string 对象的一个子字符串。这个函数需要两个参数。第一个参数指定子字符串开始的索引位置，第二个参数指定子字符串中的字符个数，该函数返回一个包含子字符串的 string 对象。例如：

```
string phrase ="The higher the fewer";
string word=phrase.substr(4,6);
```

这两行代码从 phrase 的索引位置为 4 的地方开始，提取 6 个字符的子字符串，于是在执行第二个语句之后，word 就包含 higher。

如果指定的长度超过了 string 对象的结尾，函数就返回从指定位置开始直到该字符串最后的所有字符。如下面的语句所示：

```
string word=phrase.substr(4,100);
```

当然，phrase 没有 100 个字符，子字符串也不会包含 100 个字符。在这种情况下，结果应是 word 包含从索引位置 4 开始直到结束的所有子字符串，即 higher the fewer。

省略长度，只指定表示子字符串开始位置的第一个参数，也会得到相同的结果：

```
string word=phrase.substr(4);
```

这也会返回从索引位置 4 开始直到结束的所有子字符串。如果省略了 substr()函数的两个参数，就把 phrase 的所有内容返回为了字符串。

如果为子字符串指定的起始索引位置超出了要处理的 string 对象的有效边界，就会抛出一个异常，程序将异常地终止，除非实现了一些代码来处理该异常。第 17 章将讨论异常。

6.3.5 比较字符串

在上一个例子中介绍了如何使用索引访问 string 对象中的各个字符，以进行比较。由于在使用索引值访问单个字符时，结果为 char 类型，因此可以使用比较运算符比较各个字符。

在需要比较整个字符串时，也可以使用比较运算符，把 string 对象用作操作数。前面讨论过的比较运算符有：

```
>   >=   <   <=   ==   !=
```

可以以几种方式对字符串使用这些运算符。它们可用于比较 string 类型的两个对象，或比较 string 类型的对象与存储在 char 类型的数组中的字符串字面量或 C 样式字符串。操作数将逐个比较其中的字符，直到找到不同的字符，或到达一个或两个操作数的结尾。在找到不同的字符时，字符代码的数值比较将决定哪个字符串有较小的值。如果没有找到不同的字符，但字符串有不同的长度，则较短的字符串就小于较长的字符串。如果两个字符串包含相同数量的字符，且对应的字符都相同，则这两个字符串就相等。由于比较的是字符编码，因此这种比较就是区分大小写的。

使用 if 语句比较两个 string 对象，如下所示：

```
string word1 = "age";
string word2  =  "beauty";
if(word1 < word2)
   std::cout << word1 << "comes before"  << word2;
else
   std::cout << word1 <<  "does not come before"  << word2;
```

执行上述代码会得到如下结果：

```
age comes before beauty
```

这说明古老的谚语一定正确。

上面的代码使用条件运算符会更好，下面的语句会生成相同的结果：

```
cout << word1
     << (word1 < word2 ?  "comes"  :  "does not come ")
     << "before  "  <<  word2;
```

下面在一个例子中进行一些实际的字符串比较。

程序示例 6.8——比较字符串

下面的程序将读取许多姓名，找出其中的最大值和最小值：

```
// Program 6.8 Comparing strings
#include <iostream>
#include <string>
#include <cctype>
using std::cout;
using std::cin;
using std::endl;
using std::string;

int main() {
   const int max_names = 6;                    // Maximum number of names
   string names[max_names];                    // Array of names
   int count = 0;                              // Number of names
   char answer = 0;                            // Response to a prompt

   do {
      cout << "Enter a name: ";
      cin >> names[count++];                   // Read a name

   cout << "Do you want to enter another name? (y/n): ";
   cin >> answer;                              // Read response
} while(count < max_names && std::tolower(answer) == 'y');

// Indicate when array is full
if(count == max_names)
   cout << endl << "Maximum name count reached." << endl;

// Find the minimum and maximum names
int index_of_max = 0;
```

```
int index_of_min = 0;

for(int i = i ; i < count ;  i++)
   if(names[i]  > names[index_of_max])    // Current name greater?
      index_of_max = i ;                  // then it is new maximum
   else if(names[i]  < names[index_of_min])       // Current name less?
      index_of_min = i;                           // then it is new minimum

// Output the minimum and maximum names
cout << endl
     << "The minimum name is" << names[index_of_min]
     << endl;
   cout << "the maximum name is" << names[index_of_max]
     << endl;
   return 0;
}
```

这个例子的输出如下所示：

```
Enter a name: Meshak
Do you want to enter another name?  (y/n): y
Enter a name: Eshak
Do you want to enter another name?  (y/n): y
Enter a name: Abednego
Do you want to enter another name?  (y/n): n

The minimum name is Abednego
The maximum name is Meshak
```

例子的说明

姓名存储在声明的 string 类型的数组中，该数组使用一个常量定义了其大小：

```
const int max_names=6;          //Maximum number of names
string names[max_names];         //Array of names
```

声明 string 类型的数组与声明其他类型的数组相同。这个声明将创建有 max_names 个元素的数组，每个元素都是一个包含空字符串的 string 对象。

初始化该数组的方式与前面初始化二维 char 类型的数组的方式相同。例如：

```
string names[max_names]={"Zeus","Venus"};          //Array of names
```

这个语句用花括号中的字符串字面量初始化了 names 数组中的前两个元素。其他元素为空字符串。

我们至少要输入一个名称，所以管理输入使用 do-while 循环比较好：

```
do {
   cout <<  "Enter a name:  ";
   cin >> names[count++];                                  // Read a name

   cout << "Do you want to enter another name?  (y/n):  ";
   cin >> answer;                                         // Read response
} while(count < max_names && std::tolower(answer) == 'y');
```

前面也见过这样的代码，这是读取大量数据项的一种简单机制。姓名从键盘读入 names 数组的当前元素中。提取运算符会读取第一个空白字符前面的所有字符。如果输入了'n'或'N'，或者达到了 string 对象数组的最大容量，循环条件就终止循环。

在读取了所有的输入或者达到了数组的最大容量之后，就查找包含最大字符串和最小字符串的数组元素所对应的索引值。把索引值设置为 0，假定第一个字符串包含了最大字符串和最小字符串，接着把它与数组中的其他字符串比较，这是在一个 for 循环中进行的：

```
for(int i = 1 ;  i < count ;  i++)
  if(names[i]  > names[index_of_max])          // Current name greater?
    index_of_max = i ;                        // then it is new maximum
  else if(names[i]  < names[index_of_min])     // Current name less?
    index_of_min = i;                         // then it is new minimum
```

在循环中，嵌套的 if 语句使用<和>比较运算符比较 string 对象，以确定当前数组元素是否大于当前的最大字符串，或小于当前的最小字符串。如果找到一个新的最大字符串，就不必执行 else 子句中测试新的最小字符串的操作。显然，新的最大字符串不会同时是新的最小字符串。注意循环计数器从 1 开始，因为最初假定索引为 0 的值既是最小字符串，也是最大字符串。

最后，用下面的语句输出已找到的最大字符串和最小字符串：

```
cout  <<  endl
      <<  "The minimum name is  "    << names[index_of_min]
      <<  endl;
cout  <<  "The  maximum  name  is  "  <<  names[index_of_max]
      <<  endl;
```

compare()函数

给定了 string 类型的对象后，就可以调用 compare()函数来比较该对象与 string 类型的另一个对象、字符串字面量，或存储在 char 类型数组中的非空字符串。要调用 string 对象的 compare()函数，可以使用下面的语句：

```
object_name. compare(other_object)
```

对象名后面的句点是句点运算符，前面在介绍 length()函数时介绍过。与 string 对象比较的其他对象放在括号中。

下面的表达式调用 word 对象的 compare()函数，将它与一个字符串字面量进行比较：

```
word. compare("and");
```

这里函数比较 word 的内容和字符串 and，返回 int 类型的值。如果 word 大于 and，该函数就返回一个正整数；如果 word 等于 and，该函数就返回 0；如果 word 小于 and，该函数就返回一个负整数。在上一个例子中，可以用 compare()函数代替比较运算符，但代码就不是很清晰了。for 循环变为：

```
for(int  i =  1  ;  i <  count  ;  i++)
  if(names[i].compare(names[index_of_max]) > 0)     // Current name greater?
    index_of_max  =  i;                             // then it is new maximum
  else if(names[i].compare(names[index_of_min])  < 0)// Current name less?
    index_of_min =  i;                              //  then it is new minimum
```

有时，使用 compare()函数会使代码比使用比较运算符更难以理解。但是，在一些情况下使用 compare()函数会比较好。

例如，利用 compare()比较子字符串和参数指定的字符串时，要给函数传送额外的两个参数，即子字符串的起始索引和子字符串中的字符个数。例如下面的语句：

```
string word1 = "A jackhammer";
string word2 = "jack";
if(word1.compare(2, 4, word2) == 0)
   std::cout << "Equal" << std::endl;
```

if 语句比较 word1 的一个子字符串与 word2，该子字符串从原字符串的索引位置 2 开始，包含 4 个字符，如图 6-8 所示。

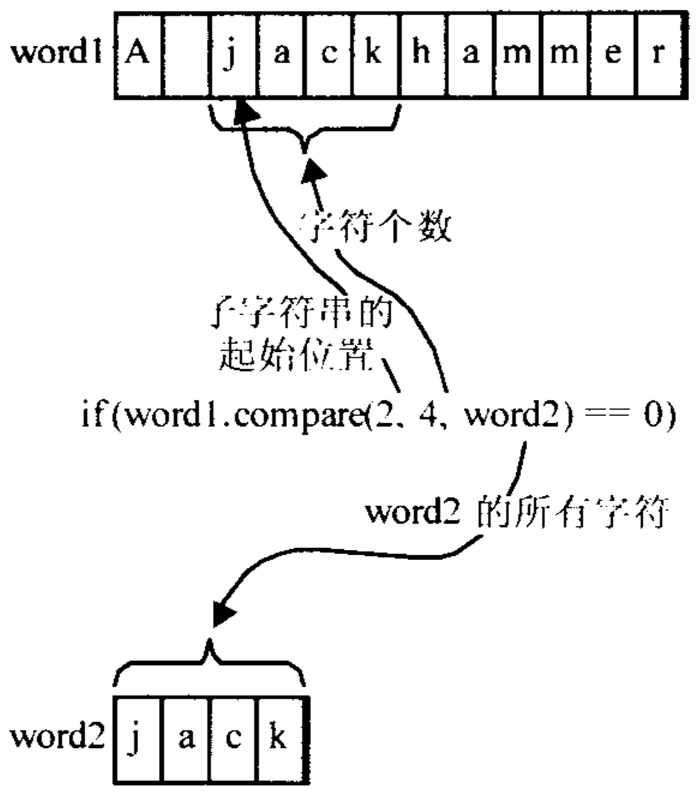


图 6-8　使用 compare()和子字符串

compare()的第一个参数是 word1 中子字符串的起始索引位置，该子字符串将与 word2 进行比较。第二个参数是子字符串的字符个数。在本例中，word2 和 word1 的子字符串相等，会执行输出语句。显然，如果 word2 的长度与指定的子字符串不相等，按照定义 word2 和 word1 的子字符串就是不相等的。

要利用 compare()函数比较一个 string 对象的子字符串和另一个 string 对象的子字符串，需要传送 5 个参数。如下所示：

```
string word1 = "A jackhammer";
string word2 = "It is a jack-in-the-box";
if(word1.compare(2, 4, word2, 8, 4) == 0)
   std::cout << "Equal" << std::endl;
```

compare()的前三个参数与前面相同，后两个参数分别是 word2 的子字符串的索引位置及其长度。如图 6-9 所示。

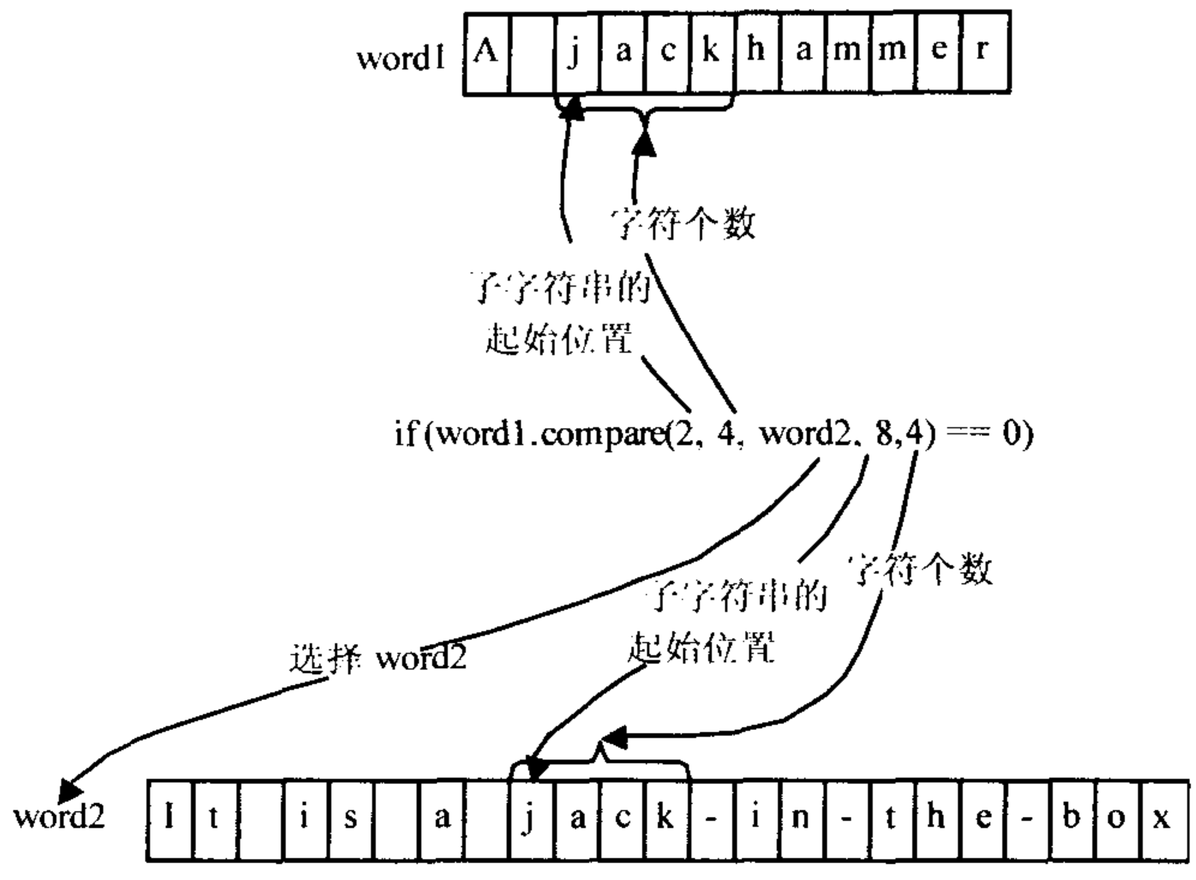


图 6-9　比较两个字符串的子字符串

word1 的子字符串与 word2 的子字符串比较，在本例中，子字符串是相同的，会执行输出语句。

compare()还可以比较 string 对象的子字符串与非空字符串。下一个例子就比较 word1 的子字符串与显示为参数的字符串字面量：

```
if(word1.compare(2,4,"jack")==0)
  std::cout<<"Equal"<< std::endl;
```

word1 的子字符串从索引位置 2 开始，包含四个字符，它与"jack"相同，if 表达式为 true，输出结果。

compare()的另一个用法是指定要使用的字符个数，从非空字符串中选择前 n 个字符。下一个语句比较 word1 的相同子字符串与"jacket"的前 4 个字符：

```
if(word1.compare(2,4,"jacket",4)==0)
  std::cout<<"Equal"<< std::endl;
```

显然，在所有的例子中，if 语句都会检查 compare()的返回值是否非 0。还可以测试 word1 是否小于或大于参数指定的字符串，返回值分别小于和大于 0。如果把 word 声明为：

```
string word="banana";
```

则表达式 word.compare("apple")就会返回一个正整数，因为 banana 大于 apple，word.compare("orange")则返回一个负整数，因为 banana 小于 orange。

注意：

本节介绍了 compare()函数，它可用于操作各种类型的、不同数量的参数。实际上，这些是具有相同名称的不同函数，称为重载函数，第 9 章将进一步讨论它们，并论述如何创建自己的重载函数。

使用 substr()进行比较

当然，如果觉得 compare()函数比较复杂的版本中的参数序列很难记忆，就可以使用 substr()

函数提取感兴趣的子字符串，再使用比较运算符。例如，如上一节的图 6-9 所示，要比较 word1 中的子字符串和 word2，就可以编写下面的测试条件：

```
if(word1.substr(2,4) == word2.substr(8,4))
  std::cout<<"Equal"<< std::endl;
```

这似乎比使用 compare()函数进行的操作更容易理解一些。

当然，把这种比较放在循环中，把循环计数器用作 substr()函数中子字符串开始的索引，就可以搜索特定的子字符串，这里不详细介绍，因为还有更简单的方法。

6.3.6 搜索字符串

搜索 string 对象有许多不同的方法，它们所涉及的函数都会返回所查找的字符串的索引位置。

首先从最简单的搜索开始。string 对象有一个函数 find()，它可以用来确定字符串中一个子字符串或一个字符的索引位置。选择搜索的子字符串可以是另一个 string 对象或非空字符串。例如：

```
string sentence = "Manners maketh man";
string word = "man";
std::cout << sentence.find(word)  << std::endl;  //Outputs 15
std::cout << sentence.find("Man") << std::endl;  //Outputs  0
std::cout << sentence.find('k')   << std::endl;  //Outputs 10
std::cout << sentence.find('x')   << std::endl;  //Outputs string::npos
```

在本例的每个使用 find()的语句中，sentence 对象将从开始处搜索。该函数返回搜索到的第一个子字符串中的第一个字符的索引位置。

在最后一个语句中，因为没有在字符串中找到 x 字符，所以返回内置常量 string::npos，它表示字符串中的非法字符位置，用于说明搜索操作中出现的失败。在一些编译器中，string::npos 的值是 4,294,967,295，但在其他系统中，其值可能有所不同。

当然，可以用下面的语句检查 find()的返回值是否等于 string::npos：

```
if(sentence. find('x')== string::npos)
  std::cout <<"Character not found"<< std::endl;
```

find()函数的另一个变化允许从指定的位置开始搜索字符串的某一部分。例如，定义了 sentence 对象后，就可以编写下面的语句：

```
std::cout << sentence. find("an",1)<< std::endl;      //Outputs 1
std::cout << sentence. find("an",3)<< std::endl;      //Outputs 16
```

这两个语句都从第二个参数指定的索引位置开始搜索 sentence，直到该字符串的结尾。第一个语句搜索第一个"an"，而第二个语句搜索第二个"an"，因为搜索从 sentence 的索引位置 3 开始。可以把 string 对象用作第一个参数，指定要搜索的字符串。例如：

```
string sentence = "Manners maketh man";
string word = "an";
int count = 0;                                  // Count of occurrences
```

```
size_t position = 0;                          // Stores a string index position
for(size_t i = 0; i< sentence.length()-word. length(); ){
  position = sentence find(word,i);
  if(position == string::npos)
    break;
  count++;
  i = position+1;
}
std::cout << "\""<<word<<"\" occurs in \""<<sentence
<<"\" "<<count<<" times.";
```

字符串中的索引位置是 size_t 类型，所以把包含 find()函数返回值的变量 position 声明为 size_t 类型。循环索引 i 用于定义 find()操作的起始位置，其类型也是 size_t。显然，sentence 中的最后一个 word 必须从 sentence 尾部向前 word.length()个位置开始，所以使用它作为循环中 i 的最大值。注意循环变量 i 没有递增表达式，因为变量 i 是在循环体中递增的。

在循环中，如果 find()返回 string::npos 时，就表示没有找到 word，因此执行 break 语句，结束循环。否则，就递增 count，把 i 设置为所找到的 word 后面的一个位置，准备下一次迭代。如果把 i 设置为 i+ word. length()，就不允许重叠找到的 word，例如在 annannanna 字符串中查找 anna。

还可以搜索非空字符串、C 样式字符串中包含的对应于指定字符个数的子字符串。在这种情况下，find()函数的第一个参数是非空字符串，第二个参数是开始搜索的索引位置，第三个参数是非空字符串中要提取作为要查找的字符串的字符个数。例如：

```
std::cout << sentence. find("akat",1,2)<< std::endl; //Outputs 9
```

这个语句在 sentence 中从位置 1 开始，搜索 akat 的前两个字符(即"ak")。下面的搜索都会失败，并返回 string::npos：

```
std::cout << sentence.find("akat",1,3)<< std::endl;  //Outputs string::npos
std::cout << sentence.find("akat",10,2)<< std::endl; //Outputs string::npos
```

第一个搜索失败是因为，字符串 aka 在 sentence 中不存在。第二个搜索查找 ak，ak 在 sentence 中存在，但在 sentence 的第 10 个位置之后就不存在 ak 了，所以该搜索也失败了。

程序示例 6.9——搜索字符串

下面编写一个程序，搜索 string 对象中的指定子字符串，并计算出该子字符串在 string 对象中出现的次数。

```
// Program 6.9 Searching a string
#include <iostream>
#include <string>
using std::cout;
using std::endl;
using std::string;

int main() {
   // The string to be searched
   string text = "Smith, where Jones had had \"had had\", had had \"had\"."
                 "\n \"Had had\" had had the examiners'  approval.";
```

```
string word = "had";                          // Substring to be found

cout << endl   <<  "The string is:  "  << endl << text << endl;

// Count the number of occurrences of word in text
int count = 0;                                // Count of substring occurrences

for(int index = 0  ;  (index = text.find(word,  index))  != string::npos ;
                                              index += word.length(),  count++)
  ;

  cout  << "Your  text  contained  "
        << count  <<  "  occurrences  of  \""
        << word    <<  "\"."
        << endl;

  return 0;
}
```

这个程序的输出如下所示：

```
The string is:
Smith, where Jones had had "had had", had had "had".
"Had had" had had theexaminers' approval.
Your input contained 10 occurrences of "had".
```

这个 string 对象中有 10 个"had"。当然，没有找到“Had”，是因为它的第一个字母是大写。

例子的说明

声明要搜索的 sreing 对象，如下所示：

```
string text= "Smith, where Jones had had \"had had\",had had \"had\"."
             "\n \"Had had\" had had the examiners" approval.";
```

在这样的语句中，字符串字面量会自动连接起来，这样就可以把初始化字符串字面量放在两行代码上。注意必须使用转义序列\"表示字符串中的双引号，因为编译器会把双引号本身解释为分隔符。

定义要搜索的子字符串：

```
string word="had";          // Substring to be found
```

这个语句声明了另一个 string 对象 word，它包含了字符串"had"。接着，在控制 for 循环的表达式中进行搜索和计数：

```
int count = 0;                // Count of substring occurrences
for(int index = 0 ;  (index = text.find(word, index))  != string::npos ;
                                        index += word.length(), count++)
  ;
```

这与前面代码中搜索字符串的方式略有不同，许多操作都是在循环中进行的，为了理解这些动作，在图 6-10 中显示了基本元素。

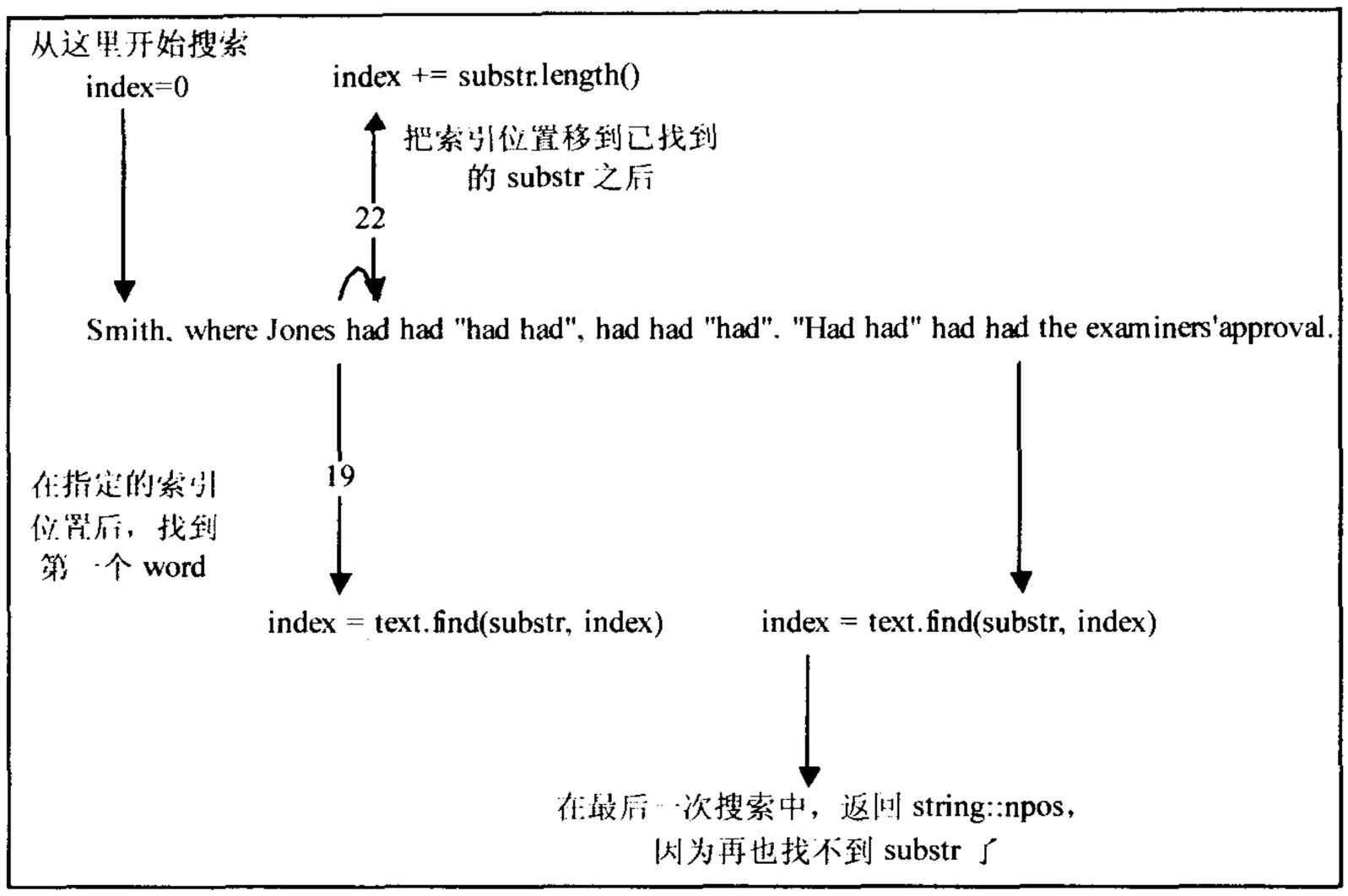


图 6-10　搜索字符串

第一个表达式初始化变量 index，该变量用于指定 text 中每次搜索操作开始的位置。决定循环是否继续的表达式是

```
(index=text.find(word,index))!=string::npos
```

这个语句从 index 位置开始搜索 text，查找第一个 word，并把找到 word 的索引位置存储在 index 中。如果存储在 index 中的值是 string::npos，就表示没有找到 word，循环停止。

如果找到了 word，就执行 for 循环的第三个控制表达式，完成两个操作。第一个操作是用表达式 index+=word.length()递增 index 的值；因为找到了 word 后，就把 word 的出现次数加 1。值 word.length()加到 index 上，把 text 的搜索起始位置移动到刚才找到的 word 后面，准备开始下一次搜索。以这种方式递增，而不是递增 1，是因为我们查找的是所有的 word，并去掉所有重叠的 word。

循环结束后，用下面的语句输出 word 在 text 中的出现次数：

```
cout << "Your text contained"
     << count  <<  "  occurrences  of  \""
     << word   <<  "\"."
     << endl;
```

1. 在字符串中搜索字符集合中的字符

假定有一个字符串，例如一段诗歌，希望将它分解为单个的单词，就需要查找到分隔符，这些分隔符可以是各种不同的字符，例如可以是空格、逗号、句点、冒号等。此时需要一个函数，在字符串中查找给定字符集合中的字符，以确定单词的分隔符。这个函数叫作 find_first_of()：

```
string text = "Smith, where Jones had had \"had had\", had had \"had\"."
              "\"Had had\" had had the examiners'  approval. ";
string  separators = "  ,.\"";
```

```
std::cout << text.find_first_of(separators)          // Outputs 5
          << std::endl;
```

给 find_first_of()函数传送的 string 对象定义了字符集合。在由 separators 定义的字符集合中，text 中的第一个字符是逗号，于是最后一个语句输出 5。如果需要，还可以把参数定义为非空字符串。例如，如果需要查找 text 中的第一个元音，可以编写下面的语句：

```
std::cout << text.find_first_of("AaEeIiOoUu");      // Outputs 2
          << std::endl;
```

结果为 2，这是因为第一个元音是 i，其索引位置是 2。

还可以使用 find_last_of()函数从 string 对象的结尾开始，进行逆向搜索，以查找给定字符集合中的字符最后一次出现的位置。例如，要查找 text 中的最后一个元音，可以使用下面的语句：

```
std::cout << text.find_last_of("AaEeIiOoUu");       // Outputs 92
          << std::endl;
```

text 中的最后一个元音是 approval 中的第二个 a，其索引位置是 92。

在 find_first_of()和 find_last_of()函数中，还可以指定另一个参数，该参数定义在要进行搜索的字符串中，从哪里开始搜索过程。如果使用非空字符串作为第一个参数，还可以用第三个参数指定字符集合中包含多少个字符。

另一个选项是查找不在字符集合中的字符，这可以使用 find_first_not_of()和 find_last_not_of()函数。例如，要查找 text 中第一个不是元音的字符的位置，可以编写下面的语句：

```
std::cout << text.find_first_not_of ("AaEeIiOoUu") // Outputs 0
          << std::endl;
```

因为第一个字符就不是元音，所以结果就是该字符，其索引位置是 0。下面在一个实际的例子中使用这些函数。

程序示例 6.10——查找给定字符集合中的字符

下面编写一个例子，演示 find_first_of()函数的用法，从包含诗歌的字符串中查找单词。这将涉及 find_first_of()和 find_first_not_of()函数的组合使用：

```
// Program 6.10 Searching a string for characters from a set
#include <iostream>
#include <string>
using std::cout;
using std::endl;
using std::string;

int main() {
   // The string to be searched
   string text = "Smith, where Jones had had \"had had\",  had had \"had\"."
                 "  \"Had had\" had had the examiners'  approval.";

   string separators =  "  ,.\  " "  ;        // Word delimiters

   // Find the start of the first word
```

```
size_t start = text.find_first_not_ of(separators);
size_t end = 0;                                  //  Index for the end of a word

// Now find and output the words
int word count = 0;                                          // Number of words
while(start  != string::npos) {
   end = text.find_first_of(separators,  start + 1);    // Find end of word
   if(end == string::npos)                              // Found a separator?
      end = text.length();                              // No,  so set to last + 1

      cout << text.substr(start,  end - start)          // Output the word
           << endl;
      word_count++;                                 // Increase  the  count

      // Find the first character of the next word
      start = text.find_ first_not_of(separators,  end + 1);
      }

   cout  << "Your  string  contained  "
         << word_count  <<  "  words."
         << endl;

    return 0;
}
```

这个程序的输出如下所示：

```
Smith
where
Jones
had
had
had
had
had
had
had
had
Had
had
had
had
the
examiners'
approval
Your string contained 17 words.
```

例子的说明

前面已解释了 string 对象的初始声明，下面直接进行字符串工作原理的分析。因为本例需要查找第一个单词的第一个字符，所以用下面的语句从 text 的开始处，向后搜索分隔符：

```
size_t start=text.find_first_not_of (separators);
```

只要这个语句返回有效值，即不是 string::npos 的值，就表示 start 包含第一个单词的第一个字符的索引位置。稍后将查找该单词在何处结束。

在变量 word_count 中累加单词的数量，变量 word_count 的声明如下：

```
int word_count=0;                    // Number of words
```

while 循环查找当前单词的结尾，显示它，之后查找下一个单词的开头：

```
while(start != string::npos) {
   end = text.find_first_of(separators,  start + 1);
   if(end == string::npos)                          // Found a separator?
      end = text.length();                          // No,  so set to last + 1

   cout << text.substr(start,  end - start)         // Output the word
        << endl;
   word_count++;                                    // Increase the count

   // Find the first character of the next word
   start = text.find_first_not_of(separators, end + 1);
}
```

while 循环条件检查 start 中记录的最初索引位置。如果 text 是空的，或只包含 separators 中定义的字符，循环就会立即终止。否则，text 至少包含一个单词，循环就会继续执行。循环会查找 separators 集合中的字符在 start 之后第一次出现的位置，也就是 word 后面的位置。这个搜索使用 find_first_of()函数从位置 start+1 开始。返回的索引位置存储在 end 中。如图 6-11 所示。

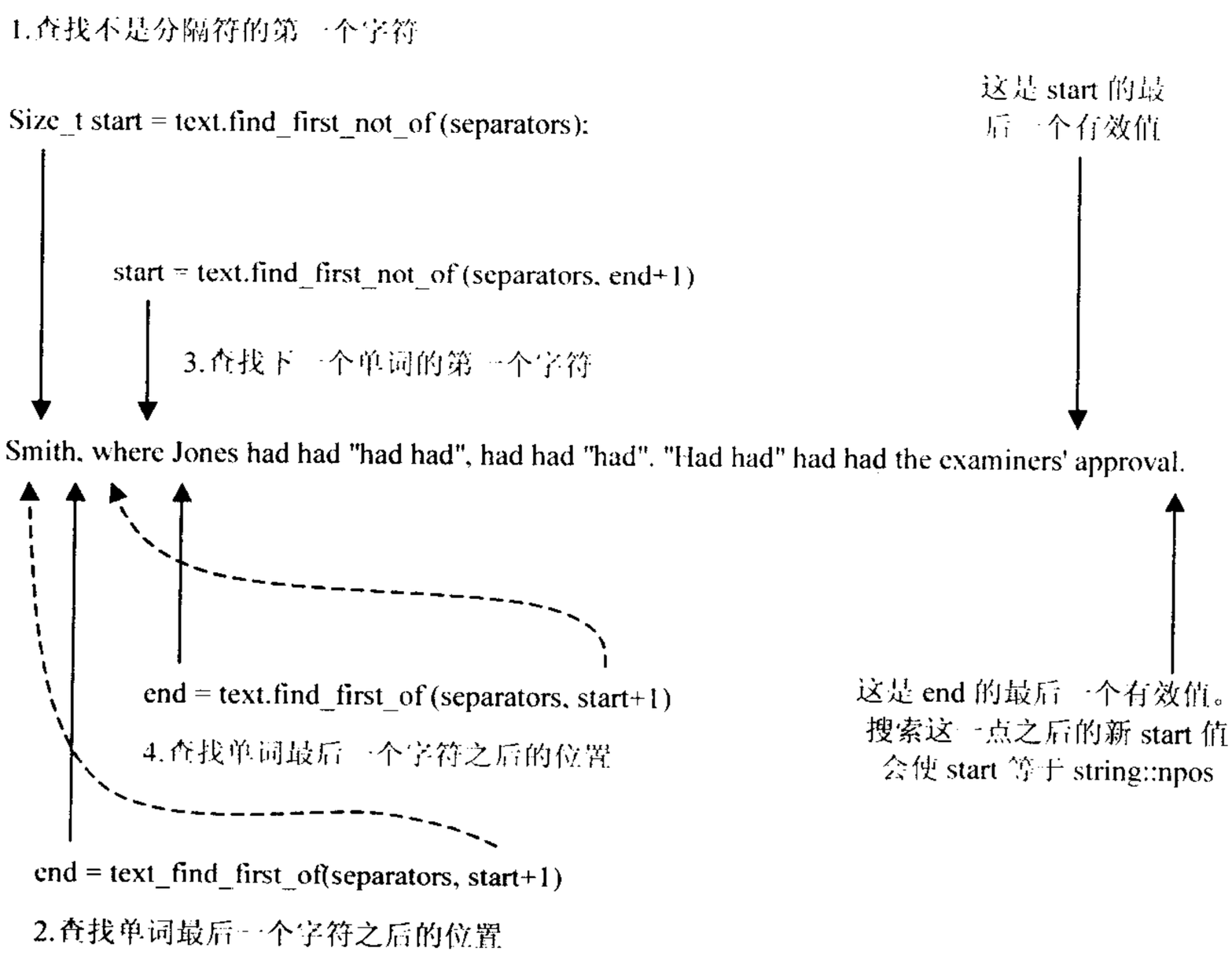


图 6-11　搜索字符串中的所有 word

当然，最后一次搜索可能也失败，end 的值是 string::npos。如果字符串 text 以一个字母结束，或以不在 separators 集合中的字符结尾，这就会发生。为了处理这种情况，可以在 if 语句中检查 end 的值，如果搜索失败，就把 end 设置为 text 的长度。该长度比字符串多一个字符(因为索引从 0 开始，不是从 1 开始)，使 end 对应于单词中最后一个字符后面的位置。

接着使用 substr()函数提取单词。变量 start 包含单词中第一个字符的索引位置，表达式 end-start 就是单词中的字符个数。在显示该单词时，递增 word_count。使用下面的语句查找下一个单词的开头：

```
start=text.find_first_not_of(separators,end+1);
```

这个语句会再次搜索不在 separators 集合中的第一个字符，搜索从 end 中记录的字符位置之后开始。如果 start 包含一个有效的索引，说明字符串中有另一个单词，该单词会在下一次迭代中显示。如果字符串中没有其他单词了，start 就设置为 string::npos，循环停止。

最后，用下面的语句输出找到的单词个数：

```
cout << "Your string contained "
     << word_count<<"words."
     << endl;
```

2. 逆向搜索字符串

find()函数可从前向后搜索字符串，从字符串开头或指定的位置开始搜索。如果要从字符串的末尾开始向前搜索，就可以使用 rfind()函数，其名称来自于逆向 find。

rfind()函数具有和 find()函数相同的变体。可以在整个 string 对象中搜索定义为另一个 string 对象的子字符串，也可以搜索定义为非空字符串的子字符串，还可以搜索一个字符。例如：

```
string sentence = "Manners maketh man";
string word = "an";
std::cout << sentence.rfind(word)   << std::endl;  //Outputs 16
std::cout << sentence.rfind("man")  << std::endl;  //Outputs 15
std::cout << sentence.rfind('e')    << std::endl;  //Outputs 11
```

这些搜索语句都查找 rfind()函数中的参数最后一次出现的位置，返回它找到的第一个字符的位置。如图 6-12 所示。

以 word 作为参数进行搜索，查找字符串中最后一个 an 出现的位置。rfind()函数返回所找到的子字符串中第一个字符的索引位置。如果没有找到子字符串，就返回 string::npos 值。例如，下面的语句就会返回这个值：

```
std::cout<<sentence.rfind("miners")<< std::endl;      //Outputs string::npos
```

由于 sentence 不包含子字符串 miners，因此这个语句会返回 string::npos 值，并显示出来。其他两个搜索与第一个搜索语句类似，也是从字符串的最后开始向前搜索参数第一次出现的位置。

与 find()函数一样，可以给 rfind()函数添加一个参数，指定从后向前搜索的起始位置，当第一个参数是 C 样式的字符串时，还可以添加第三个参数，指定从 C 样式字符串中提取的字符个数，作为要搜索的子字符串。

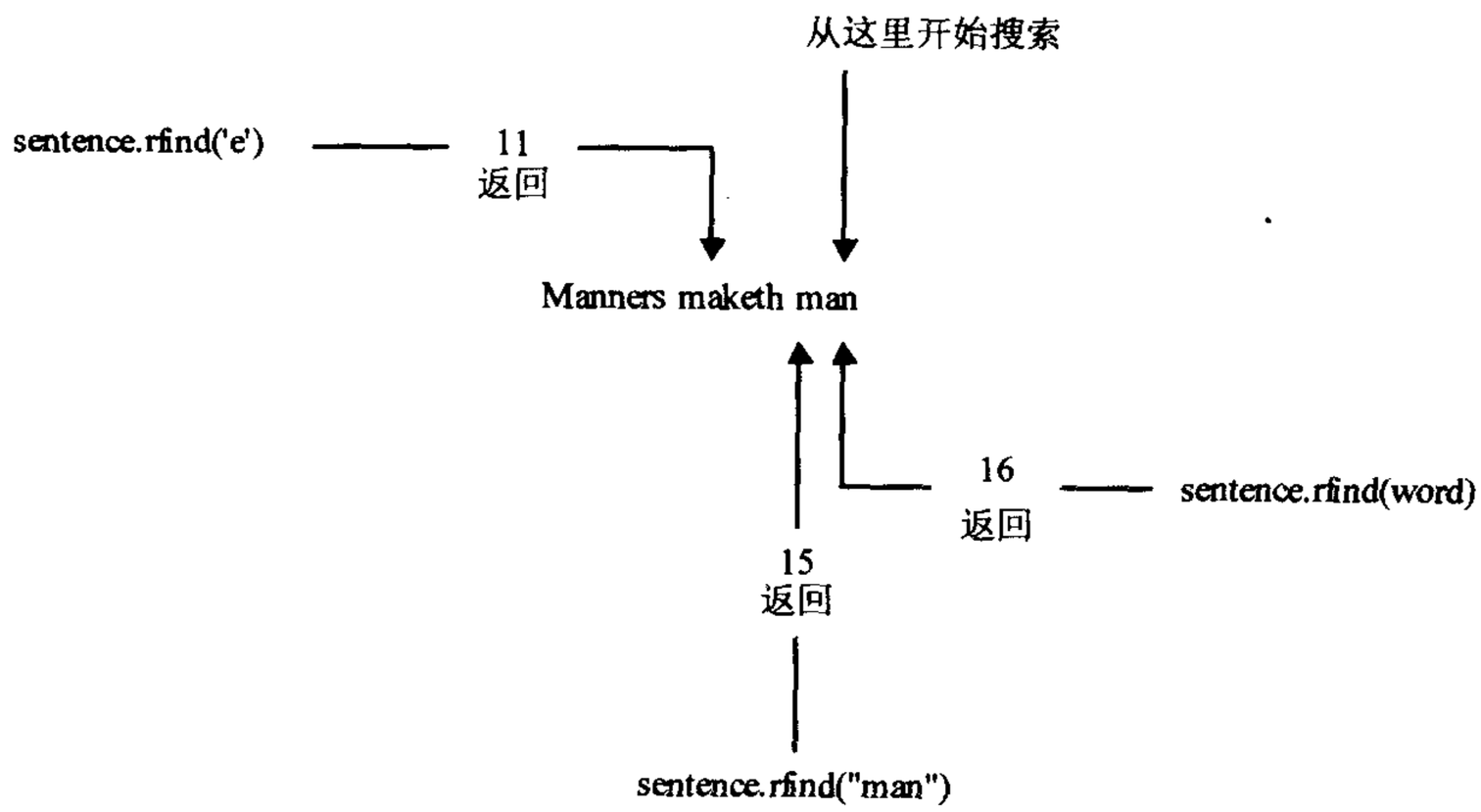


图 6-12　逆向搜索字符串

6.3.7　修改字符串

自然情况下，当搜索字符串，找到需要的内容后，还希望以某种方式修改它。前面介绍了如何使用位于方括号中的索引值，修改 string 对象中的单个字符，还可以在 string 对象中插入子字符串，或替换已有的子字符串。用于插入子字符串的函数叫做 insert()，用于替换子字符串的函数叫做 replace()。下面先插入一个子字符串。

1. 插入字符串

最简单的插入操作是在一个 string 对象的给定位置之前插入另一个 string 对象。下面是一个说明其工作原理的例子：

```
string phrase="We can insert a string.";
string words="a string into ";
phrase.insert(14,words);
```

如图 6-13 所示，words 字符串插入到 phrase 中索引位置为 14 的字符前面。执行这个操作后，phrase 就包含字符串“We can insert a string into a string.”。

也可以把非空字符串插入到 string 对象中。例如，下面的语句可以得到与上面语句相同的结果：

```
phrase.insert(14, "a string into ");
```

当然，'\0'字符在插入前被非空字符串舍弃，因为它是一个分隔符，而不是字符串中的字符。

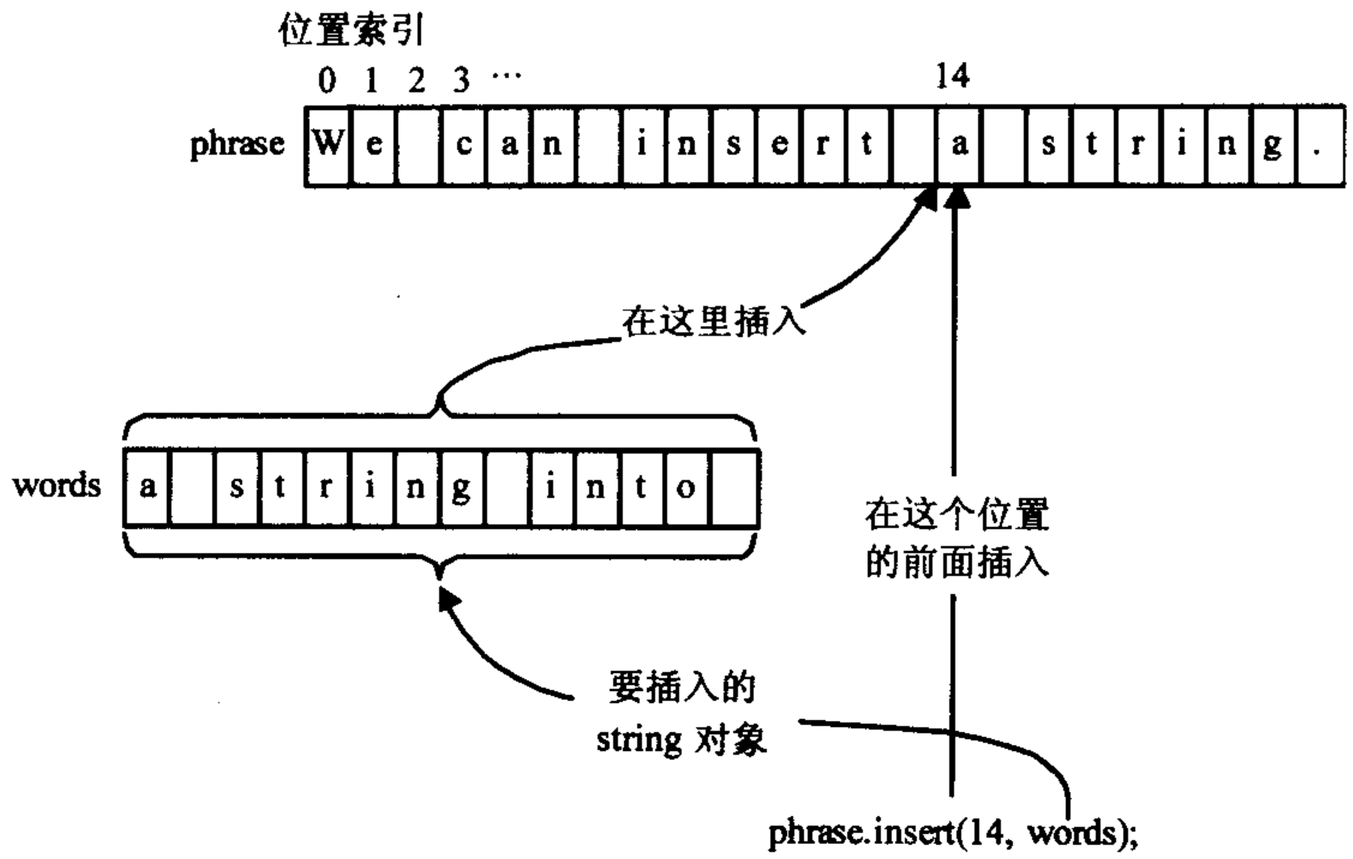


图 6-13　在一个字符串中插入另一个字符串

比这更复杂的是把一个 string 对象的子字符串插入到 string 类型的对象中。只需在 insert() 函数调用中提供另外两个参数，一个参数指定子字符串中第一个字符的索引位置，另一个参数指定子字符串的字符个数。例如：

```
phrase.insert(13,words,8,5);
```

这个语句把 words 中从第 8 个位置开始的 5 个字符组成的子字符串插入到 phrase 中。假定它们表示最初的字符串，则该语句就把"into"插入到"We can insert a string."中，这样，phrase 就变成"We can insert into a string."。

把非空字符串中指定数目的字符插入到 string 对象中也有类似的效果。下面的语句将生成与上面相同的结果。

```
phrase.insert(13, "into something",5);
```

这个语句把“into something”中的前 5 个字符组成的子字符串插入到 phrase 中的第 13 索引位置的字符之前。

如果需要把包含几个相同字符的字符串插入到 string 对象中，可以使用下面的语句：

```
phrase.insert(16,7,'*');
```

这个语句把 7 个星号插入到 phrase 的第 16 索引位置的字符之前。这样，phrase 就包含含义不明的句子"We can insert a *******string."。

2. 替换子字符串

可以用另一个子字符串来替换 string 对象的任意子字符串——即使两个字符串有不同的长度，也可以替换。如果把 text 定义为：

```
string text="Smith, where Jones had had \" had had\", had had \"had\".";
```

下面的语句可以把名字 Jones 替换为一个不常见的名字：

```
text.replace(13,5,"Gruntfuttock");
```

这个语句把 text 中从位置 13 开始的 5 个字符替换为字符串"Gruntfuttock"。如果现在输出 text，结果就是：

```
Smith, where Gruntfuttock had had "had had", had had "had".
```

更实际的做法是先搜索要替换的子字符串。例如：

```
string separators=" ,.\" ";                                    //Word delimiters
size_t start=text.find("Jones");                               //Find the surstring
size_t end=text.find_first_of(separators,start+1);             //Find the end
text.replace(start,end-start," Gruntfuttock");
```

这段代码查找 text 中"Jones"的第一个字母的位置，并把索引值存储在 start 中。用 find_first_of()函数搜索 separators 中的分隔符，查找"Jones"最后一个字符后面的字符。然后在 replace()函数中使用这些索引位置。

替换字符串可以是 string 对象或非空字符串。对于前者，可以指定起始索引和长度，从 string 对象中选择要用作替换字符串的子字符串。例如，上述替换操作可以是：

```
string name="Amos Gruntfuttock";
text.replace(start,end-start,name,5,12);
```

这两个语句与前面使用 replace()函数的效果相同，因为替换字符串都是从 name 的第 5 个位置上的字符开始(即字符 G)，包含 12 个字符。

如果第一个参数是非空字符串，就可以指定要从该字符串中选择的字符个数，作为替换字符串。例如：

```
text.replace(start,end-start, "Gruntfuttock, Amos",12);
```

这次，要替换的字符串是"Gruntfuttock, Amos"中的前 12 个字符，其效果与前面的替换操作一样。

与 insert()函数相同，另一个选项是把替换字符串指定为由重复指定次数的指定字符组成。例如，下面的语句可以用 3 个星号替换"Jones"：

```
text.replace(start,end-start,3,' * ');
```

这个语句假定 start 和 end 按以前那样定义，其结果是 text 将包含：

```
Smith, where *** had had "had had", had had "had".
```

下面在一个例子中使用替换操作。

程序示例 6.11——替换子字符串

这个程序用另一个单词替换字符串中的指定单词：

```
// Program 6.11 Replacing words in a string
#include <iostream>
#include <string>
using std::cout;
using std::cin;
using std::endl;
using std::string;
```

```
int main() {
   // Read the string from the keyboard
   string text;
   cout << "Enter a string terminated by #:"  << endl;
   std::getline(cin,  text,  '#');

   // Get the word to be replaced
   string word;
   cout << endl <<  "Enter the word to be replaced:  ";
   cin >> word;

   // Get the replacement
   string replacement;
   cout << endl <<  "Enter the replacement word:  ";
   cin >> replacement;

   if(word == replacement) {
      cout  << endl
            <<  "The word and its replacement are the same."  << endl
            <<  "Operation aborted."  << endl;
      exit(1);
   }

// Find the start of the first occurrence of word
size_t start = text.find(word);

// Now find and replace all occurrences of word
while(start  != string::npos) {
   text.replace(start, word.length(),  replacement);       // Replace word
   start = text.find(word,  start + replacement.length());
}

cout << endl
     <<  "Your string is now:"  << endl
     << text << endl;

return 0;
}
```

这个程序的输出如下所示：

```
Enter a string terminated by #:
A rose is a rose
is a rose.#

Enter the word to be replaced: rose

Enter the replacement word: dandelion

Your string is now:
A dandelion is a dandelion
is a dandelion.
```

例子的说明

首先声明一个 string 类型的对象 text，用于存储一个字符串，在该字符串中，将替换给定单词的每个实例。在提示输入后，用下面的语句从键盘上读取字符串：

```
std::getline(cin, text,'#');
```

前面介绍过这个函数。它把第一个参数指定的流读入到第二个参数指定的 string 对象中。第三个参数定义了标识字符串末尾的字符。把#指定为字符串结尾，允许输入多行文本。在本例中，终止符#前面的所有换行符都将存储到字符串中。

在提示要替换的单词后，将它读入 string 对象 word 中：

```
cin>>word;
```

这次，由于第一个空白字符会终止输入，所以只存储一个单词。要替换的单词以同样的方式读入 replacement 对象中。

其后的 if 语句仅处理要替换的单词和用于替换的单词相同的情况。

```
if(word == replacement) {
    cout  << endl
          << "The word and its replacement are the same." << cout
          << "Operation aborted." << cout;
    exit(1);
}
```

可以用任何方式处理这种情况，甚至可以选择不检查，仅替换单词，但这里选择退出程序，并提醒用户。还要注意，比较运算符可以用于 string 对象。

现在准备进行替换。首先，用下面的语句找到 word 第一次出现时的索引位置：

```
size_t start=text.find(word);
```

把 start 声明为 size_t 类型，是因为 find()返回该类型的值。这是整数类型的一个同义词，通常是 unsigned int。如果 text 中没有 word，start 的值就是 string::npos。这是在进行替换的 while 循环中处理的：

```
while(start  != string::npos) {
    text.replace(start, word.length(), replacement);          // Replace word
    start = text.find(word, start + replacement.length());
}
```

如果 start 包含 string::npos，循环就会立即终止。假定 word 不存在于 text 中，就会执行 replace()函数。从 start 给定的索引位置开始，用字符串 replacement 替换 word.length()字符。由于 replacement 的长度与 word 不同，在定义从何处开始搜索下一个 word 时就要非常小心。在调用 find()函数时，把 replacement 的长度加到 start 上，从而指定起始的索引位置。循环会继续进行，直到 text 中再也找不到 word 为止。

最后，下面的语句显示更新后的字符串：

```
cout << endl
     << "Your string is now:"<<endl
     << text<<endl;
```

3. 删除字符串中的字符

使用 replace()函数可以删除 string 对象中的子字符串，方法是把替换字符串指定为空字符串。还可以使用另一个专用于此的函数 erase()。可以为要删除的子字符串指定其起始索引位置和长度。例如，要删除 text 中的前 6 个字符，可以使用下面的语句：

```
text.erase(0,6);              //Remove the first 6 characters
```

通常使用这个函数删除以前搜索出来的某个子字符串。erase()的一个更常见的用法如下所示：

```
string word = "rose";
size_t index = text.find(word);
if(index ! = string::npos)
  text.erase(index,word.length());
```

这里试图在 text 中找到 word 的位置。在确认 word 存在后，就使用 erase()函数删除它。要删除的子字符串的字符个数可以调用 word 的 length()函数来得到。

要删除字符串中的所有字符，可以使用 clear()函数，如下所示：

```
text. clear();
```

这个语句删除字符串 text 中的所有字符，因此它变成一个空字符串。

6.4 string 类型的数组

前面的一个例子使用了 string 数组，但应仔细分析一下。使用 string 类型的对象数组与使用其他类型的数组是一样的。例如，下面的语句可以创建 string 类型的数组：

```
string words[]={"this","that","the other"};
```

数组 words 有 3 个元素，这由花括号中的 3 个初始化字符串字面量来决定。当然，也可以显式指定数组的大小，如下面的语句所示：

```
string words[10]={"this","that","the other"};
```

现在，该 string 类型的数组有 10 个元素，前 3 个元素用花括号中的字符串字面量初始化了，其他的 7 个元素是空字符串。

还可以引用 string 数组元素中的各个字符。在 words 数组中，下面的语句可以把第 3 个元素的第 7 个字符改为 t：

```
words[2][6]='t';
```

下面的语句会显示这个字符串：

```
cout<<words[2];
```

在修改后，该字符串显示为：

```
the otter
```

使用 string 类型的数组非常类似于使用其他类型的数组。string 数组元素的操作与单个 string 变量的操作完全相同。

6.5 宽字符的字符串

如果字符串需要包含 wchat_t 类型的字符，而不是 char 类型的字符，可以使用<string>头文件中定义的 wstring 类型。使用 wstring 类型的对象的方式与 string 类型的对象相同。下面的语句就声明了一个宽字符串的对象：

```
wstring quote;
```

这个语句假定文件中包含 std::wstring 的 using 声明。

宽字符串字面量在双引号中包含 wchat_t 类型的字符，并添加一个前缀 L，把它们与包含 char 字符的字符串字面量区分开来。因此，wstring 变量的声明和初始化如下所示：

```
wstring saying = L"The tigers of wrath are wiser than the horses of instruction.";
```

注意必须使用 L 后跟由 wchat_t 字符组成的宽字符串字面量。没有前缀 L，就会得到一个 char 字符串字面量，这个语句也不会编译。

当然，要输出宽字符串，必须使用 wcout 流，例如：

```
std::wcout << saying;
```

前面讨论的 string 对象的所有函数都可以应用于 wstring 对象，所以不再重复。记住，在 wstring 对象的操作中，应使用 wchat_t 类型的字符变量，定义字符和字符串字面量时要加上前缀 L。

6.6 本章小结

本章介绍了如何创建数值数组，探讨了 char 类型数组的特殊属性，但数组的讨论还没有结束。第 7 章将继续讨论数组，探讨指针和数组之间的关系。

本章还论述了如何使用在标准库中定义的 string 类型。一般情况下，string 类型比 char 类型的数组更易于进行字符串操作，所以在需要处理字符串时，string 类型应是首选。

本章主要内容如下：

- 数组是同一类型的数值的命名集合，它们存储在连续的内存块中，每个值都可以通过一个或多个索引值来访问。
- 一维数组需要一个索引值来引用其元素，二维数组需要两个索引值，n 维数组需要 n 个索引值。
- 数组的元素可以用在等号的左边和表达式中，其方法与相同类型的变量一样。
- char 类型的一维数组可以用于存储非空字符串。
- 可以让编译器根据声明语句中初始化值的个数，来决定数组中最左边一维的大小。
- 可以把 char 类型的二维数组用作非空字符串的一维数组。

- string 类型存储了一个字符串，它不需要终止字符。因为 string 对象会跟踪字符串的长度。
- 在 string 变量名后面的方括号中指定索引值，就可以访问 string 对象中的各个字符。索引值从 0 开始。
- 使用+运算符可以把 string 对象与字符串字面量、字符或另一个 string 对象连接起来。
- string 类型的对象可以利用函数来搜索、修改和提取子字符串。
- 字符串中的位置存储为 size_t 类型的整数值。
- 声明 string 类型的数组与声明其他类型的数组所采用的方法是一样的。
- wstring 类型的对象包含 wchar_t 类型的字符串。

6.7 练习

1. 创建一个数组，存储至多 100 个学生的姓。创建另一个数组，存储每个学生的成绩(0~100)。使用一个循环，提示用户给这些数组输入姓名和成绩。计算平均成绩并显示，然后在一个表中显示所有学生的姓名和成绩。

2. 一位气象学者每天(从周一到周五)都要记录大气的湿度三次(早晨、中午和晚上)。编写一个小程序，让用户按年代顺序把这些记录输入到一个 5 行 3 列的 float 数组中。接着计算并显示每天的平均湿度和每天三次的星期平均值。

3. 扩展程序 6.9，搜索子字符串"had"，找出其中的所有"had"单词，且不考虑大小写。(提示：复制原字符串。)

4. 编写一个程序，从键盘上读取一个任意长度的文本字符串，再提示输入要在该字符串中查找的单词。程序应查找出现在字符串中的所有该单词，不考虑大小写，再用与单词中字符个数相同的星号替换该单词，然后输出新字符串。注意必须替换整个单词。例如，如果用户输入了字符串"Our house is at your disposal."，要查找的单词是 our，则得到的字符串应是"*** house is at your disposal."，而不是"*** house is at y*** disposal."。

5. 编写一个程序，提示输入两个字符串，再测试它们，看看其中一个字符串是否为另一个字符串颠倒字母顺序而得到的。

第7章 指 针

下面开始介绍指针，这是C++编程的一个重要元素。指针非常重要，因为它们构成了动态分配和使用内存的基础，在其他许多方面使程序更高效、更富有活力。

本章主要内容

- 指针的定义及声明方式
- 如何获取变量的地址
- 指针与数组的关系
- 带有指针的算术运算如何进行，其适用范围如何
- 哪些标准库函数可用于处理非空字符串
- 在执行程序时，如何为新变量创建内存
- 如何释放动态分配的内存
- 如何把一种类型的指针转换为另一种类型

7.1 什么是指针

程序中的每个变量和字面量都有一个内存地址。内存地址就是存储数据的计算机内存的位置。同样，为了执行，程序使用的函数也必须位于内存的某个地方，所以函数也有地址。这些地址取决于运行程序时，将程序加载到内存的什么地方。这些位置的不同导致程序的执行也有所不同。

指针是一个可用于存储内存地址的变量。存储在指针中的地址通常对应于变量所在内存的位置，当然也可以是函数的地址。函数详见第8章。

图7-1说明了指针这个名称的由来：它指向内存中存储了某些内容(变量或函数)的位置。但是，如果指针仅存储内存地址，是没有用的。为了利用存储在该地址上的实体，需要知道该实体是什么，而不是仅仅知道该实体在什么地方就够了。整型数的表示方式跟浮点数大相径庭，数据项所占用的字节数取决于该数据的内容。因此，要使用存储在指针所包含的地址中的数据项，还需要知道数据的类型。

因此，指针不仅仅是“指针”，还表示了数据项的类型。这样，在进行后面的操作时才会比较清楚。下面看看如何声明指针。

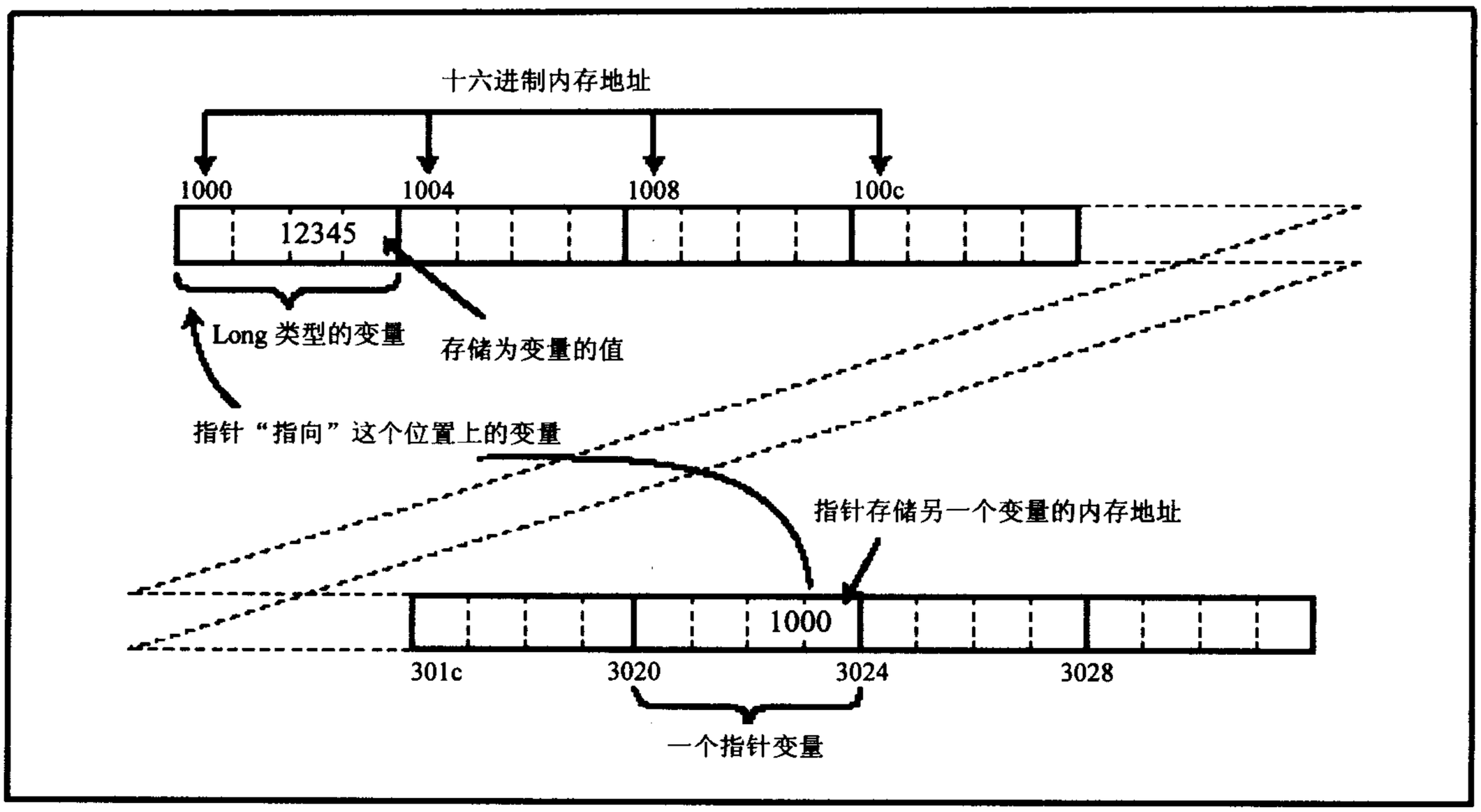


图 7-1　指针的含义

7.2　指针的声明

指针的声明类似于一般的变量，但类型名后跟一个星号，表示声明了一个指针类型的变量。例如，要声明指针 pnumber，它指向 long 类型的值，可以使用下面的语句：

```
long* pnumber;
```

这个语句声明了一个指针变量 pnumber，它可以存储 long 类型的变量的地址。这个变量的类型是“指向 long”，当单独使用类型时(例如在 case 语句中)，通常写为 long*。

上面的声明在类型名的旁边加了一个星号，而这并不是声明指针的惟一方式。还可以把星号加在变量名的旁边，来声明一个指针，如下面的语句：

```
long *pnumber;
```

这个语句声明了与前面相同的变量。编译器接受这两种记号，但前者比较常见，因为它对类型“指向 long”的表达更清晰一些。但是，在同一个语句中混合使用一般的变量和指针的声明时，可能会产生混淆。例如下面的语句：

```
long* pnumber, number;
```

实际上，这个语句声明了一个类型为“指向 long”的变量 pnumber 和类型为 long 的变量 number，但把星号和类型名并列放置，会使其含义不太清晰。如果以另一种形式声明这两个变量，

```
long *pnumber, number;
```

就不容易出现混淆，因为星号现在附加在变量 pnumber 上。但是，这个问题的实际解决方案是不把指针变量和普通变量放在一行中声明。最好在单独的代码行上声明变量，以避免出现这种混淆：

```
long number;              //Declaration of long variable
long* pnumber;            //Declaration of variable of type 'pointer to long'
```

这样就很容易在声明的最后添加注释，解释它们的用途。

在本例中，使用 pnumber 作为指针变量名，这不是强制的，但在 C++中，通常约定用以 p 开头的变量名表示指针。这将很容易看出程序中的哪些变量是指针，从而使源代码更容易理解。

指向非 long 类型的指针的声明方式与此类似。为了说明这一点，下面的语句分别声明了指向 double 和 string 的变量：

```
double* pvalue;           // Pointer to a double value
string* psentence;        // Pointer to a string value
```

使用指针

要使用指针，只需在指针中存储对应类型的另一个变量的地址即可。下面先介绍如何获取变量的地址。

1. 地址运算符

地址运算符&是一个一元运算符，它可以获取存储变量的内存地址。下面的语句声明了一个指针 pnumber 和一个变量 number：

```
long number=12345L;
long* pnumber;
```

因为变量 number 的类型和指针 pnumber 的类型是兼容的，所以可以进行如下赋值：

```
pnumber=&number;          //Store address of number in pnumber
```

也就是说，把 number 的地址赋予 pnumber。这个操作的执行结果如图 7-2 所示。

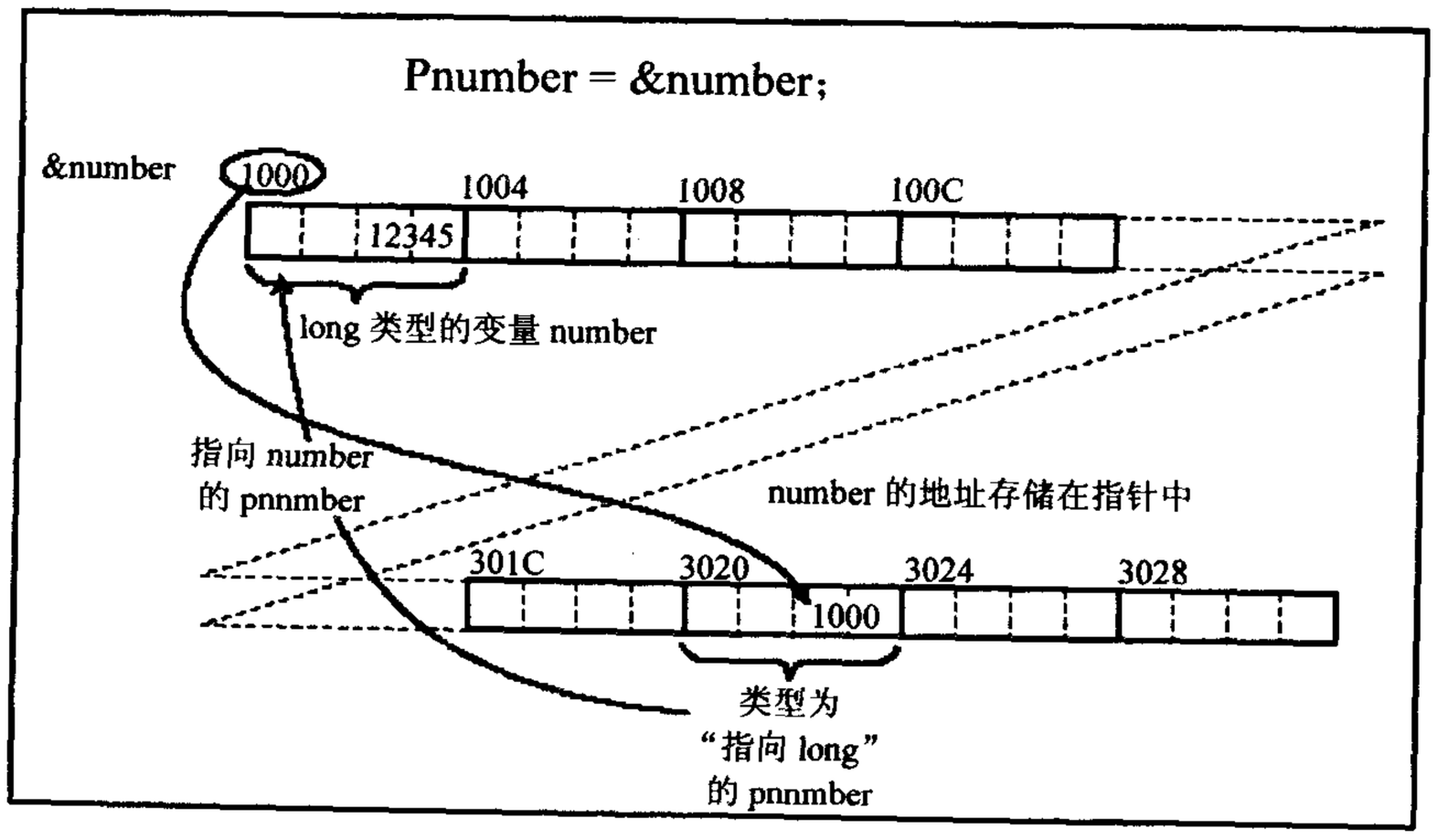


图 7-2　在指针中存储地址

可以使用&运算符获取任何类型的变量地址，但必须在对应类型的指针中存储该地址。例如，如果要存储 double 变量的地址，指针必须声明为类型 double*，即“指向 double”。如果在类型错误的指针中存储了地址，程序就不会编译。

提取变量的地址，并把它存储在一个指针中都很好完成，但是我们真正感兴趣的是如何使用指针。使用指针的基本操作是访问指针所指向的内存位置的数据。这可以使用间接运算符来完成。

2. 间接运算符

间接运算符*和指针变量一起使用，访问指针所指向的内存位置的内容。间接运算符这个名称来自于数据的访问是间接的这一事实。该运算符有时也称为 dereference 运算符，访问指针所指向的内存位置的数据的过程称为“解除指针的引用”。

要访问指针 pnumber 指向的地址中的数据，可以使用表达式*pnumber。下面看看该运算符的用法。

程序示例 7.1——使用间接运算符

这个例子使用间接运算符*显示指针所指向的变量的值：

```
//Program 7.1 The indirection operator in action
#include <iostream>
using std::cout;
using std::endl;

int main(){
  long number=50L;
  long* pnumber;                          //Pointer declaration
  pnumber=&number;                        //Store the address of number
  cout<<endl
      << "The value stored in the variable number is "
      <<*pnumber
      <<endl;
return 0;
}
```

编译并运行这个程序，结果如下：

```
The value stored in the variable number is 50
```

例子的说明

首先声明一个 long 类型的常规变量，并初始化为 50：

```
long number=50L;
```

接着，声明一个指针变量：

```
long *pnumber;          //Pointer declaration
```

由于其类型是 long*，因此这个变量可以存储 long 类型的变量的地址。下面的语句会在 pnumber 中存储 number 的地址：

```
pnumber=&number;        //Store the address of number
```

地址运算符&生成了 number 的内存地址，并存储为 pnumber 的值。现在可以通过解除 pnumber 的引用，显示存储在 number 中的值：

```
cout << endl
     << "The value stored in the variable number is"
     << *pnumber
     << endl;
```

应用于指针的间接运算符引用了存储在指针中的地址内容。因为 pnumber 包含了 number 的地址，所以*pnumber 引用了 number 的值。

间接运算符的一个容易混淆的方面是，同一个符号*有好几种不同的用法。*是一个乘法运算符，间接运算符，还可以用于声明一个指针。每次使用*时，编译器都可以根据上下文区分其含义。在对两个变量进行乘法运算时，如 price*quantity，这个表达式没有什么可解释的，不过是一个乘法运算而已。本章的下一个程序 7.2 将根据上下文来解释*的含义。

3. 为什么使用指针

常常有一个问题会困扰读者：“为什么要使用指针？”毕竟，提取已知变量的地址，再把它存储在指针中，以便以后解除对指针的引用，这看起来似乎是一个负担，可有可无。不要轻易下判断，因为指针非常重要涉及好几个原因。

首先，如后面所述，可以使用指针记号操作存储在数组中的数据，这常常比使用数组表示法快一些。第二，在本书后面定义自己的函数时，指针的使用非常广泛，可以在函数中访问在该函数外部定义的大块数据，例如数组。

第三，也是最重要的原因，可以动态地为新变量分配内存空间。即在程序执行过程中分配。这种功能可以使程序根据输入调整对内存的使用。在程序运行过程中以及需要时创建新变量。由于事先并不知道要动态创建多少个变量，就只能使用指针来创建了，因此一定要掌握这部分内容。

为了更好地理解指针的工作方式，下面用另一个非常简单的例子，练习使用指针。

程序示例 7.2——使用指针

下面的例子将试用前面介绍的所有指针操作：

```
//Program 7.2 Exercising pointers
#include <iostream>
using std::cout;
using std::endl;

int main(){
long *pnumber;              //Pointer declaration
long number1=55L;
long number2=99L;           //A couple of variables

pnumber=&number1;           //Store address in pointer
*pnumber+=11;               //Increment number1 by 11
cout<<endl
```

```
    << "number1="   << number1
    << "  &number1=" <<pnumber
    << endl;

pnumber =&number2;          // Change pointer to address of number2
number1 =*pnumber*10;       // 10 times number2

cout << "number1= "   << number1
     << "  pnumber="  << pnumber
     << "  *pnumber=" << *pnumber
     << endl;
return 0;
}
```

这个程序的输出如下：

```
number1=66    &number=0012FEC8
number1=990    pnumber=0012FEBC  *pnumber=99
```

所显示的地址值在另一台计算机上会有所不同。

例子的说明

这个例子没有输入，所有的操作都是处理程序中设置的值。在指针 pnumber 中存储 number1 的地址后，number1 的值将通过指针进行间接递增，例如下面的语句：

```
*pnumber+=11;      //Increment number1 by 11
```

间接运算符确定把 11 加到指针所指向的变量 number1 的内容上。这说明可以在赋值运算符的左边使用解除引用的指针。如果没有加上*，程序就会改变存储在指针中的地址(稍后详细讨论指针的算术运算)。

下面这个语句将显示 number1 的值，以及存储在 pnumber 中的 number1 的地址：

```
cout << endl
     << "number1="   << number1
     << " &number1=" <<pnumber
     << endl;
```

对于数值类型的指针，把指针名(在本例中是 pnumber)发送给输出流，会输出地址值。因为这是指针类型，所存储的值就显示为十六进制。内存地址一般表示为十六进制形式，这不仅仅在 C++中是这样。而 number 是一般的整数变量，其值显示为十进制形式。

在输出第一行后，下面的语句把 pnumber 的内容设置为 number2 的地址：

```
pnumber =&number2;          // Change pointer to address of number2
```

现在 pnumber 指向变量 number2。以前存储在 pnumber 中的 number1 的地址，现在则被覆盖了。通过指针 pnumber，把变量 number1 改为 number2 的值乘以 10：

```
number1 =*pnumber*10;       //10 times number2
```

等号右边的表达式通过指针间接访问 number2 的值，并把它乘以 10。编译器知道如何解释这行代码中的*，因为它会考虑上下文。下一个输出语句显示这些计算的结果：

```
cout  << "number1= " << number1
      << " pnumber=" <<pnumber
      << " *pnumber=" <<*pnumber
      << endl;
```

这个语句会显示 number1 的值、存储在 pnumber 中的地址，以及存储在 pnumber 所包含的地址中的值。pnumber 的值显示为十六进制，因为它是一个地址。表达式*pnumber 解析为普通的整数值，即存储在 number2 中的值，所以显示为十进制。

7.3 指针的初始化

使用未初始化的指针甚至比使用未初始化的一般变量和数组更危险。如果指针变量包含一个垃圾值，就可能覆盖内存的某个随机区域。因为造成的伤害仅取决于运气，所以一定要使指针初始化。

很容易把指针初始化为已定义的变量地址。使用地址运算符，并把变量名用作指针的初始值，就可以用变量 number 的地址初始化指针 pnumber：

```
int number=0;              // Initialized integer variable
int* pnumber=&number;      // Initialized pointer
```

用另一个变量初始化指针时，必须在声明指针之前声明变量。否则，编译器就会产生错误。

当然，在声明指针时，也可以不用指定变量的地址来初始化指针。此时可以用等于 0 的指针来进行初始化：

```
int* pnumber=0;     // Pointer not pointing to anything
```

这个声明确保 pnumber 不指向任何实体。因此，如果在给它赋值前试图解除对它的引用，程序就会失败，并详细说明这一情形。以这种方式初始化的指针称为空指针。但是，在解除对指针的引用之前，可以测试指针，看看它是否为空：

```
if(pnumber==0)
  std::cout<< std::endl<<"pnumber is null. "<< std::endl;
else
  std::cout<< std::endl<<"Value is " << *pnumber << std::endl;
```

在比较中一定要使用两个等于号==。上面的语句等价于：

```
if(!pnumber)
  std::cout<< std::endl<<"pnumber is null. "<< std::endl;
else
  std::cout<< std::endl<<"Value is " << *pnumber << std::endl;
```

当然，还可以使用下面的形式：

```
if(pnumber!=0)
  std::cout<< std::endl<<"Value is " << *pnumber << std::endl;
```

符号 NULL 在标准库中也定义为 0，它常常用于初始化空指针。但是，NULL 仅同 C 兼容，在 C++中最好使用 0。

把指针初始化为 char 类型

“指向 char”类型的变量有一个有趣的属性，它可以用字符串字面量初始化。例如，下面的语句就声明并初始化了一个这样的指针：

```
char* pproverb="A miss is as good as a mile. ";          // Don't do this!
```

这个语句看起来非常类似于用字符串字面量初始化 char 数组，但不应仅从外观上判断，它们是完全不同的。这个语句利用引号中的字符串创建了一个非空字符串字面量(实际上是 const char 类型的数组)，并把字符串字面量中第一个字符的地址存储在指针 pproverb 中。如图 7-3 所示。

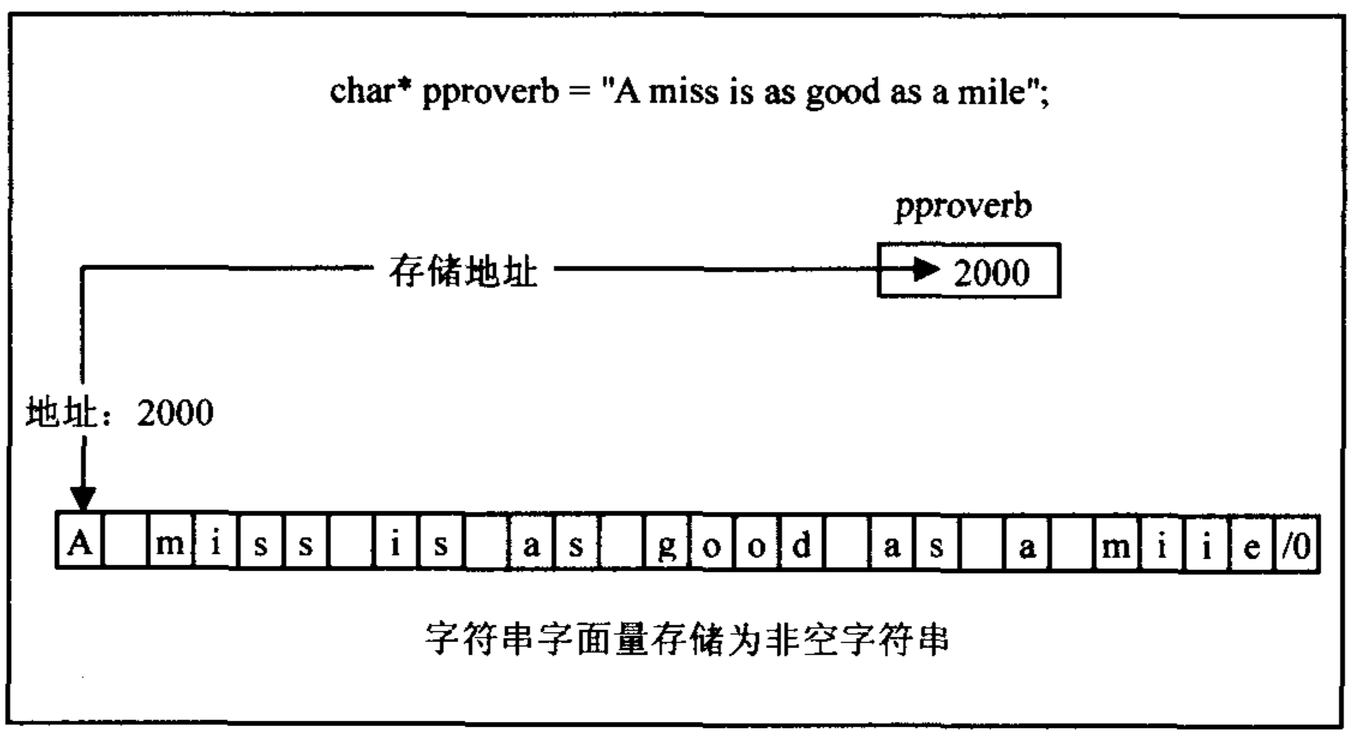


图 7-3　初始化 char*类型的指针

但是，这并不像表面那样。字符串字面量的类型是 const，但指针的类型不是 const。这个语句并没有创建字符串字面量的可修改副本，只是存储了第一个字符的地址。也就是说，如果编写一些代码来修改该字符串，例如下面的语句试图把第一个字符改为 X：

```
*pproverb='X';
```

于是，编译器就会编译该语句，因为它找不出错误。指针 pproverb 没有声明为 const，所以编译器可以编译该语句。但是，在运行程序时，会产生错误：内存中的字符串字面量仍是一个常量，不允许修改它。

为什么编译器允许给非 const 类型赋予 const 值，但运行代码会产生这些错误？原因是字符串字面量仅在 C++标准下是常量，而且有许多以前的代码依赖于“不正确的”赋值。但不提倡这种用法，这个问题的正确解决是按如下方法来声明指针：

```
const char* pproverb="A miss is as good as a mile. ";          // Do this instead!
```

这个语句声明 pproverb 指针指向 const 变量，即 const char*类型，这样其类型就与字符串字面量的类型一致了。对指针使用 const 还有许多内容，本章的后面将详细介绍这个主题。现在，在另一个例子中看看如何使用 char*类型的变量操作。

程序示例 7.3——利用指针的幸运之星

下面编写“幸运之星”例子(程序示例 6.5)的一个新版本，它使用指针来代替数组，看看指

针的工作原理：

```
// Program 7.3 Initializing pointers with strings
#include <iostream>
using std::cout;
using std::cin;
using std::endl;

int main() {
  // The lucky stars referenced through pointers
  const char* pstar1 = "Mae West";
  const char* pstar2 = "Arnold Schwarzenegger";
  const char* pstar3 = "Lassie";
  const char* pstar4 = "Slim Pickens";
  const char* pstar5 = "Greta Garbo";
  const char* pstar6 = "Oliver Hardy";
  const char* pstr   = "Your lucky star is  ";

  int choice = 0;                    // Star selector

  cout << endl
       << "Pick a lucky star!"
       << " Enter a number between 1 and 6: ";
  cin >> choice;

  cout << endl;

  switch(choice) {
     case 1:
        cout << pstr << pstar1;
        break;
     case 2:
        cout << pstr << pstar2;
        break;
     case 3:
        cout << pstr<< pstar3;
        break;
     case 4:
        cout << pstr << pstar4;
        break;
     case 5:
        cout << pstr << pstar5;
        break;
     case 6:
        cout << pstr << pstar6;
        break;
     default:
        cout << "Sorry, you haven't got a lucky star.";
  }

  cout << endl;
  return 0;
```

```
}
```

这个例子的输出如下：

```
Pick a lucky star! Enter a number between 1 and 6: 5

Your lucky star is Greta Garbo
```

例子的说明

原例子的数组被 6 个指针(pstar1 到 pstar6)代替了，每个指针都用一个名称来初始化。接着还声明了一个指针 pstr，它用一个短语来初始化，该短语会在正常输出行的开头使用。因为所有的指针都用于指向字符串字面量，所以把它们声明为 const。

由于这些指针都是离散的，因此使用 switch 语句比使用原版本中的 if 语句更容易选择合适的输出消息。如果输入了不正确的值，就由 switch 语句的 default 选项处理。

输出一个指针指向的字符串是很容易的，只需编写该指针名即可。注意，标准输出流 cout 根据指针指向的类型对指针名进行不同的处理。在程序示例 7.2 中，代码

```
cout<<pnumber;
```

会输出包含在 pnumber 指针中的地址。而在本例中，代码

```
cout<<pstar1;
```

会输出一个字符串字面量，而不是地址。区别在于 pnumber 是一个数值类型的指针，而 pstar1 是一个指向 char 的指针。输出流 cout 把指向 char 的变量看做非空字符串，并显示该字符串。

指针数组

这样会得到什么好处？使用指针就可节省该程序的数组版本所浪费的内存空间，因为每个字符串现在都只占用容纳其所有字符所必需的字节数。但是，程序看起来有点啰嗦。还有一种更好的方法，即使用指针数组。

程序示例 7.4——指针数组

在 char 类型的指针数组中，每个元素都可以指向一个独立的字符串，每个字符串的长度都可以不同。声明指针数组的方式与声明其他数组是一样的。下面是上述例子的另一个版本，它使用指针数组：

```
//Program 7.4 Using an array of pointers to char
#include <iostream>
using std::cout;
using std::cin;
using std::endl;

int main() {
  const char* pstars[] = {                    // Initializing a pointer array
                          "Mae West","Arnold Schwarzenegger","Lassie",
                          "Slim Pickens","Greta Garbo","Oliver Hardy"
                         };
```

```
const char* pstr =  "Your lucky star is  ";
int  choice = 0;

const int starCount = sizeof pstars/sizeof pstars[0];   // Get array size

cout << endl
     << "Pick a lucky star!"
     << " Enter  a number between  1  and  "
     << starCount
     << ": ";
  cin  >>  choice;

  cout <<  endl;
  if(choice >= 1 && choice <= starCount)                 // Check for valid input
   cout << pstr << pstars[choice - 1];                   // OutPut star name
  else
   cout << "Sorry, you haven't got a lucky star.";      // Invalid input

  cout <<  endl;
  return 0;
}
```

例子的说明

在这个版本中，几乎获得了最佳的效果。本例声明了 char 类型的一维指针数组，让编译器根据初始化字符串的个数计算出数组的大小：

```
const char* pstars[] ={                          // Initializing a pointer array
                       "Mae West","Arnold Schwarzenegger","Lassie",
                       "Slim Pickens","Greta Garbo","Oliver Hardy"
                      };
```

这个语句的内存使用情况如图 7-4 所示。

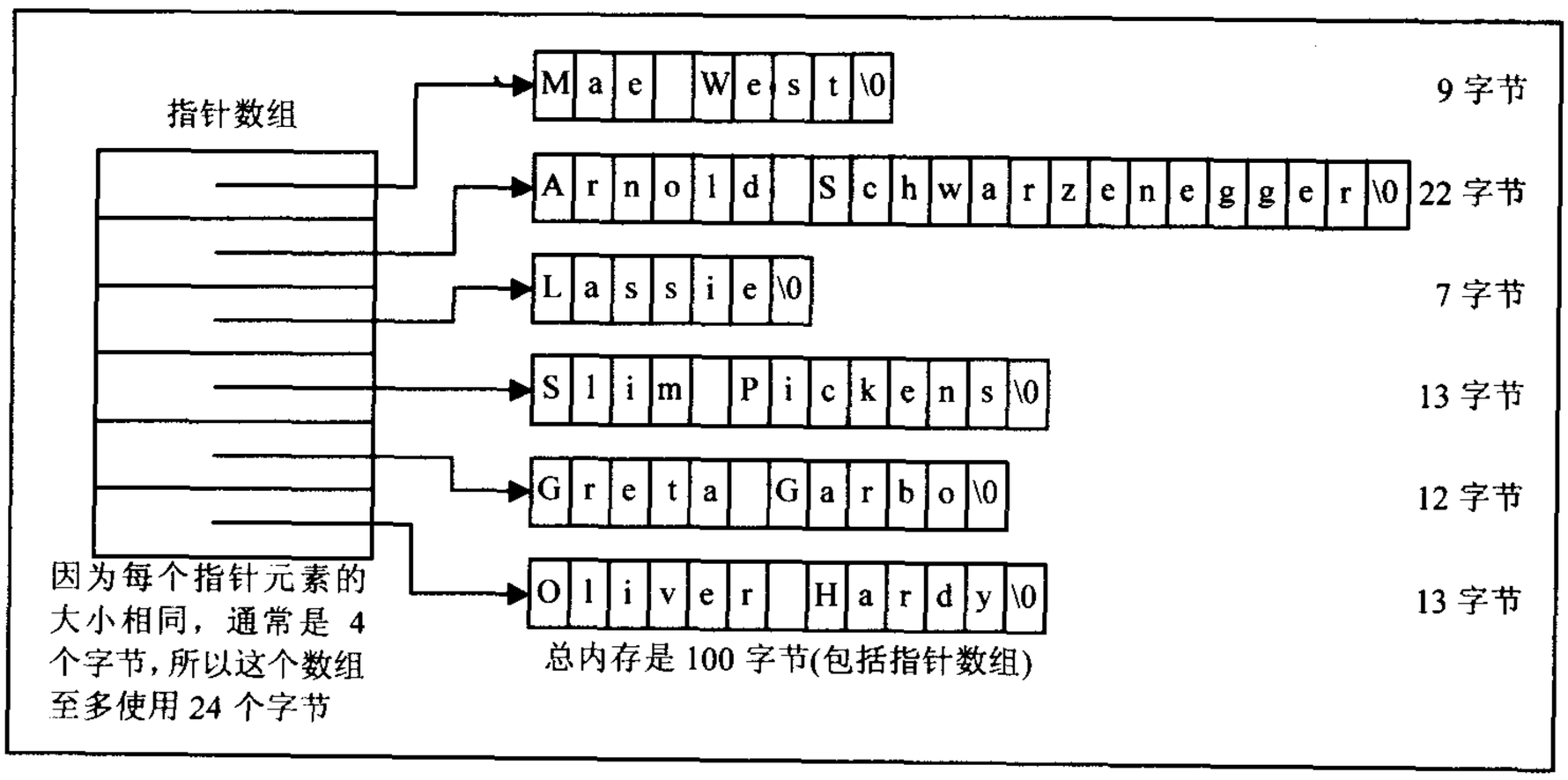


图 7-4　指针数组

从图 7-4 中可以看出，每个非空字符串都需要分配内存，数组中的元素也需要内存，而每

个元素都是一个指针，这样算下来总共 100 个字节。与使用 char 类型的数组相比，指针数组需要的内存较少。而使用旧式的静态数组，每一行至少必须有最长字符串的长度；每行 22 个字节，6 行就是 132 个字节，而使用指针数组，可以节省 32 个字节。当然，具体会节省多少字节，取决于字符串的个数和每个字符串的长度。有时根本节省不了几个字节，但一般情况下指针数组是比较节约的。

使用指针不仅可以节省内存空间，在许多情况下，还可以节省时间。例如，如果要把第 5 个位置的"Greta Garbo"与开始的"Mae West"交换位置，一般需要对字符串按照字母顺序来排序。而使用指针数组，则只需交换彼此的指针，字符串本身可以不动。

如果字符串存储在 char 类型的数组中，就需要进行大量的复制工作。即把整个字符串"Greta Garbo"复制到一个临时位置，再把"Mae West"复制到"Greta Garbo"原来的位置上，之后把"Greta Garbo"复制到"Mae West"原来的位置上，这需要多得多的计算机时间来执行。整个逻辑也可以应用于 string 类型的对象。

回到本例，我们把选择幸运之星的基本消息的地址存储在另一个指针中：

```
const char* pstr="Your lucky star is ";
```

接着，下面的语句计算指针数组 pstars 中的元素个数：

```
const int starCount = sizeof pstars/sizeof pstars[0];      //Get array size
```

以这种方式计算出数组的大小，可以使程序的剩余部分自动调整，以容纳任意多个幸运之星。接着提示输入 choice 的值：

```
cout << endl
     << "Pick a lucky star! "
     << "Enter a number between 1 and "
     << starCount
     << ": ";
```

像程序的前一个版本那样读取 choice 的值后，就通过一个非常简单的 if 语句选择要输出的字符串。可以显示 pstars 数组中的一个选项，如果用户输入了一个无效值，则可以显示适当的消息。if 条件使用 starCount 变量作为 choice 的上限，这样就可以自动容纳任意多个幸运之星选项。如果希望程序中有更多的选项，则只需把它们添加到初始化字符串的列表中。

使用指针对字符串排序

在上一个例子的讨论中提到，如果使用指针引用字符串，就可以在不移动字符串的情况下对字符串排序。

下面创建程序示例 6.10 的一个新版本，程序示例 6.10 从文本字符串中提取单词。这就为应用 string 对象、使用指向 string 对象的指针数组提供了很好的经验。同时，还可以学习如何使用指针进行排序。

程序示例 7.5——使用指针对字符串排序

从键盘上读取一组单词，以预定义的顺序对它们排序。下面是代码：

```
// Program 7.5 Sorting strings using pointers
#include <iostream>
```

```
#include <string>
using std::cout;
using std::cin;
using std::endl;
using std::string;

using std::cout;

int main() {
   string text;                                  // The string to be sorted
   const string separators =" ,.\"\n";           // Word delimiters
   const int max_words = 1000;                   // Maximum number of words
   string words[max_words];                      // Array to store the words
   string* pwords[max_words];                    // Array of pointers to the words

   // Read the string to be searched from the keyboard
   cout << endl << "Enter a string terminated by #:" << endl;
   getline(cin, text,  '#');

   // Extract all the words from the text
   int start = text.find_first_not_of(separators)   // Word start index
   int end = 0;                                     // End delimiter index
   int word_count = 0;                              // Count of words stored
   while(start != string::npos && word_count < max_words) {
      end = text.find_first_of(separators, start + 1);
      if(end == string::npos)                       // Found a separator?
         end = text.length();                       // No,  so set to last + 1

       words[word_count]  = text.substr(start, end - start); // Store the word
       pwords[word_count] = &words[word_count];     // Store the pointer
       word_count++;                                // Increment count

      // Find the first character of the next word
      start = text.find_first_not_of(separators, end + 1);
   }

   // Sort the words in ascending sequence by direct insertion
   int lowest = 0;                                  // Index of lowest word
   for(int j =0; j < word_count - 1; j++) {
      lowest = j;                                   // Set lowest

    // Check current against all the following words
   for(int i = j + i ; i < word_count ; i++)
      if(*pwords[i]  < *pwords[lowest])             // Current is lower?
        lowest = i;

   if(lowest !=j) {

                                                    // Then swap pointers
     string* ptemp = pwords[j];                     // Save current
     pwords[j]  = pwords[lowest];                   // Store lower in current
     pwords[lowest]  = ptemp;                       // Restore current
   }
```

```
  }

// Output the words in ascending sequence
   for(int i = 0 ; i < word_count ;  i++)
     cout << endl << *pwords[i];

    cout << endl;
    return 0;
}
```

这个例子的输出如下：

```
Enter a string terminated by #:
In this world nothing can be said to be certain, except death and taxes.#

In
and
be
be
can
certain
death
except
nothing
said
taxes
this
to
world
```

例子的说明

在声明如下的变量中，将读入包含要排序的单词的文本字符串：

```
string text;                              //The string to be sorted
```

常量 separators 包含用作单词分隔符的所有字符，它们是空格、逗号、句点、双引号和换行符。

```
const string separators = " ,.\ "\n ";       //Word delimiters
```

如果需要，则还可以在该列表中添加制表符。另外，还可以搜索 text 中所有不是字母、数字或单引号的字符，构建包含这些分隔符的字符串。

把从 text 中提取出来的单词存储在一个数组中，该数组至多可容纳 1000 个单词：

```
const int max_words=1000;                 //Maximum number of words
string words[max_words];                  //Array to store the words
string* pwords[max_words];                //Array of pointers to the words
```

words 数组存储单词，指针数组 pwords 存储 words 中每个元素的地址。这有点麻烦，但如果要在给 string 对象排序时避免重复地复制它们，就需要使用指针。还必须为 string 对象 max_words 和指针分配内存空间，但一般不需要使用数组中所有的元素。本章后面将介绍一种更好的方法，即使用动态内存分配。

用前面的方式把多行文本读入 text，而在输入#后就终止输入，这样就可以输入任意多行文本：

```
cout<<endl<< "Enter a string terminated by #: "<<endl;
getline(cin,text,'#');
```

在 while 循环中，从 text 中提取每个单词，把它们存储在 words 数组中，其代码如下：

```
// Extract all the words from the text
int start = text.find_first_not_of(separators);     // Word start index
int end = 0;                                        // End delimiter index
int word_count = 0;                                 // Count of words stored
while(start != string::npos && word_count < max_words) {
  end = text.find_first_of(separators, start + 1);
  if(end == string::npos)                           // Found a separator?
    end = text.length();                            // No,  so set to last + 1

   words[word_count] = text.substr(start, end - start);        // Store the word
   pwords[word_count] = &words[word_count];         // Store the pointer
   word_count++;                                    // Increment count

   // Find the first character of the next word
   start = text.find_first_not_of(separators, end + 1);
}
```

这些工作也可以使用前面的方式来完成。找出单词中第一个字母的索引位置，把它保存在 start 中，再找出单词后面的第一个分隔符，把它的索引位置存储在 end 中。然后使用 substr() 函数把单词提取为一个 string 对象，存储在 words 数组的下一个可用元素中。另外把该 words 数组元素的地址存储在 pwords 数组的对应元素中。如果到达 text 的结尾或填满了 words 数组，循环条件就会停止搜索单词。

为了进行排序操作，声明一个变量 lowest，以记录在排序过程中找到的“排在最后的”单词的索引位置：

```
int lowest=0;              //Index of lowest word
```

单词的排序在嵌套的 for 循环中进行：

```
for(int j = 0;  j < word_count - 1;  j++) {
  lowest = j;                                       // Set lowest

  // Check current against all the following words
  for(int i = j + 1 ; i < word_count ; i++)
    if(*pwords[i]  < *pwords[lowest])               // Current is lower?
      lowest = i;

  if(lowest != j) {                                 // Then swap pointers
    string* ptemp = pwords[j];                      // Save current
    pwords[j]  = pwords[lowest];                    // Store lower in current
    pwords[lowest]  = ptemp;                        // Restore current
  }
}
```

该操作重新安排了 pwords 数组中的指针，使它们按照升序指向 words 数组中的单词。这个过程非常简单。外层的循环从 pwords 数组中的第一个元素遍历到最后一个元素。在内层的循环中，比较一个指针指向的单词和其后的指针指向的所有单词，找出“排在最后的”单词的索引位置。如果排在最后的指针不是当前的指针，就把 ptemp 用作临时存储器，以交换它们。对 pwords 数组中的每个元素重复这个过程。

通过这个方法，该数组中的第一个元素就包含指向“排在最后的”单词的指针，第二个元素指向排在倒数第二个单词，依此类推。注意，在输出中 In 排在 and 的前面，因为运行程序的机器使用了 ASCII，所以 I 的字符编码低于 a 的字符编码。

最后，在 for 循环中按升序输出单词：

```
for(int i=0;i<word_count;i++)
  cout<<endl<<*pwords[i];
```

pwords 数组的元素指向按升序排列的单词。为了按顺序输出它们，只需显示数组中已解除引用的指针。

使输出结果更整齐

一行输出一个单词对于短小的文本来说比较好，但如果文本很大，就非常不方便。最好把以相同字母开头的所有单词组成一个组输出。如果准备使用略微复杂的输出机制，就可以用下面的代码替代前面的 for 循环：

```
// Output up to six words to a line in groups starting with the same letter
char ch =  (*pwords[0])[0];                    // First letter of first word
int words_ in_line = 0;                        // Words in a line count
for(int i = 0; i < word_count ; i++) {
   if(ch != (*pwords[i])[0])                   // New first letter?
   {
      cout << endl;                           // Start a new line
      ch =  (*pwords[i]) [0];                 // Save the new first letter
      words_ in_ line = 0;                    // Reset words in line count
   }
   cout << *pwords[i]  <<"   ";
   if(++words_in_line == 6) {                 // Every sixth word
      cout << endl;                           // Start a new line
      words_ in _line = 0;
  }
}
```

这段代码也以升序输出所有的单词，但这次以相同字母开头的单词被组合在一起。每个组都另起一行，一行至多显示 6 个单词。

一组中的第一个字母存储在变量 ch 中。注意，我们引用了第一个 string 对象的第一个字符。表达式*pwords[0]解除了 pwords 数组中第一个元素的指针的引用，给出了该指针指向的单词。表达式(*pwords[0])[0]给出了该单词的第一个字母，因为圆括号外的方括号包含了单词中的字母的索引。由于方括号的优先级高于间接运算符*，因此需要使用圆括号。变量 words_in_line 跟踪当前行上的单词数，当它等于 6 时，就输出一个换行符。当然，只要要显示的单词的第一个字母不同于 ch 中的字符，就开始一个新组，同时输出一个换行符，把 words_in_line 重新设置为 0。

7.4 常量指针和指向常量的指针

本章前面在处理指向 char 类型的指针时，介绍了使用指向常量的指针来处理字符串字面量，这是 C++标准强制使用的技术。例如在“幸运之星”程序中，编译器就试图修改由 pstars 数组的元素指向的字符串，而 pstars 数组则是用 const 声明的：

```
const char* pstars[ ] = {
                                "Mae West","Arnold Schwarzenegger","Lassie",
                               "Slim Pickens","Greta Garbo","Oliver Hardy"
                            };
```

在这个声明中，把由指针数组的元素指向的对象声明为常量。由于编译器会试图修改该对象，因此会把下面的赋值语句标识为错误，防止在运行期间出现令人头痛的问题：

```
*pstars[0]='X';
```

但是，仍可以合法地编写下面的语句，把存储在等号运算符右边的元素中的地址复制到等号左边的元素中：

```
pstars[0]= pstars[5];
```

现在，这些获得 Ms.West 的幸运儿会得到 Mr.Hardy，因为两个指针都指向同一个名字。注意，指针数组的元素所指向的对象值并没有变化，只是存储在 pstars[0]中的地址发生了变化，因此没有违背 const 规范。

必须习惯这种变化，因为有些人可能料想，已老去的 Olly 没有 Mae West 那么性感。看看下面的语句：

```
const char* const pstars[ ] = {
                                  "Mae West","Arnold Schwarzenegger","Lassie",
                                  "Slim Pickens","Greta Garbo","Oliver Hardy"
                                };
```

const 声明了一个常量指针，现在指针及其指向的字符串都声明为常量。这个数组就不能修改了。

总之，对指针及其指向的内容使用 const 有 3 种不同的情形。

(1) 指向常量的指针。指针指向的内容不能修改，但可以把指针设置为指向其他内容：

```
const char* pstring="Some text that cannot be changed";
```

当然，这也适用于其他类型的指针。例如：

```
const int value=20;
const int* pvalue=&value;
```

value 是一个常量，不能修改。pvalue 是一个指向常量的指针，可以用于存储 value 的地址。不能在不是常量的指针中存储 value 的地址(因为这意味着可以通过指针修改常量)，但可以把不是常量的变量的地址赋予 pvalue。在后一种情况中，通过指针修改变量是非法的。

一般情况下，总是可以用这种方式加强常量的不变性，而不是减弱，这是不允许的。

(2) 常量指针。存储在指针中的地址不能修改。像这样的指针只能指向初始化时指定的地址。但是，地址的内容不是常量，可以修改。

下面用一个数值例子来说明常量指针。假定声明了一个整数变量 value 和一个常量指针 pvalue：

```
int value=20;
int* const pvalue=&value;
```

这个语句声明，指针 pvalue 是 const，只能指向 value。使它指向另一个 int 变量的任何尝试都会让编译器生成一个错误消息。但 value 的内容不是 const，可以随时修改。如果 value 声明为 const，就不能用&value 初始化 pvalue，指针 pvalue 只能指向不是常量的 int 类型的变量。

(3) 指向常量的常量指针。因为存储在指针中的地址和该指针指向的内容都声明为常量，所以两者都不能修改。

下面是一个数值例子，把 value 声明为：

```
const int value=20;
```

value 现在是一个常量，不能修改它。但仍可以用 value 的地址来初始化一个指针：

```
const int* const pvalue=&value;
```

pvalue 现在是一个指向常量的常量指针，不能修改 pvalue 指向的内容，也不能修改它包含的地址上的值。

注意：

一般情况下，这并不仅限于前面介绍的 char 和 int 类型指针，而是可以应用于任意类型。

7.5 指针和数组

指针和数组名之间有很密切的关联。实际上，在许多情况下，可以把数组名用作一个指针。如第 6 章所述，在输出语句中，数组名本身可以像指针那样操作。如果在输出数组时只使用数组名(只要它不是 char 类型的数组)，就会得到该数组在内存中的十六进制地址。因为在这种方式下，数组名的操作就像指针一样，所以可以用于初始化指针。例如：

```
double values[10];
double* pvalue=values;
```

这个语句把数组 values 的地址存储在指针 pvalue 中，即数组名存储了一个地址。

了解了数组名和指针之间的类似性之后，还要注意它们是完全不同的实体，应注意区分它们。指针和数组名之间最大的区别是，存储在指针中的地址可以修改，而数组名所表示的地址是固定的。

7.5.1 指针的算术运算

可以对指针进行操作，修改它包含的地址。根据算术运算符，只能对指针进行加减运算，

但可以比较指针，得到逻辑结果。

对于加法操作，可以给指针加上一个整数值(或等于整数的表达式)，其结果是一个地址。也可以从指针中减去一个整数，其结果也是一个地址。最后，可以比较两个指针的不同，其结果是一个整数，而不是地址。其他算术运算符不能用于指针。

对指针的算术运算采用一种特殊的方式。假定下面的语句给指针加 1：

```
pvalue++;
```

这个语句给指针递增 1。给指针递增 1 采用什么方法并不重要，还可以使用赋值或+=运算符，其结果跟下面的语句是一样的：

```
pvalue +=1;
```

有趣的是，存储在指针中的地址并不是按一般的算术运算那样递增 1，指针的算术运算意味着，假定指针指向一个数组，算术运算是对包含在该指针中的地址进行的，这对应于数组中的一个元素。给指针加 1 就表示给它递增一个元素。编译器知道存储一个数组元素需要的字节数，给指针加 1 就是给指针中的地址递增该字节数。换言之，给指针加 1 就是把指针移向数组中的下一个元素。

例如，如果在前面的声明中，pvalue 是指向 double 的指针，编译器要为 double 类型的变量分配 8 个字节的内存，这样 pvalue 中的地址就递增 8。如图 7-5 所示。

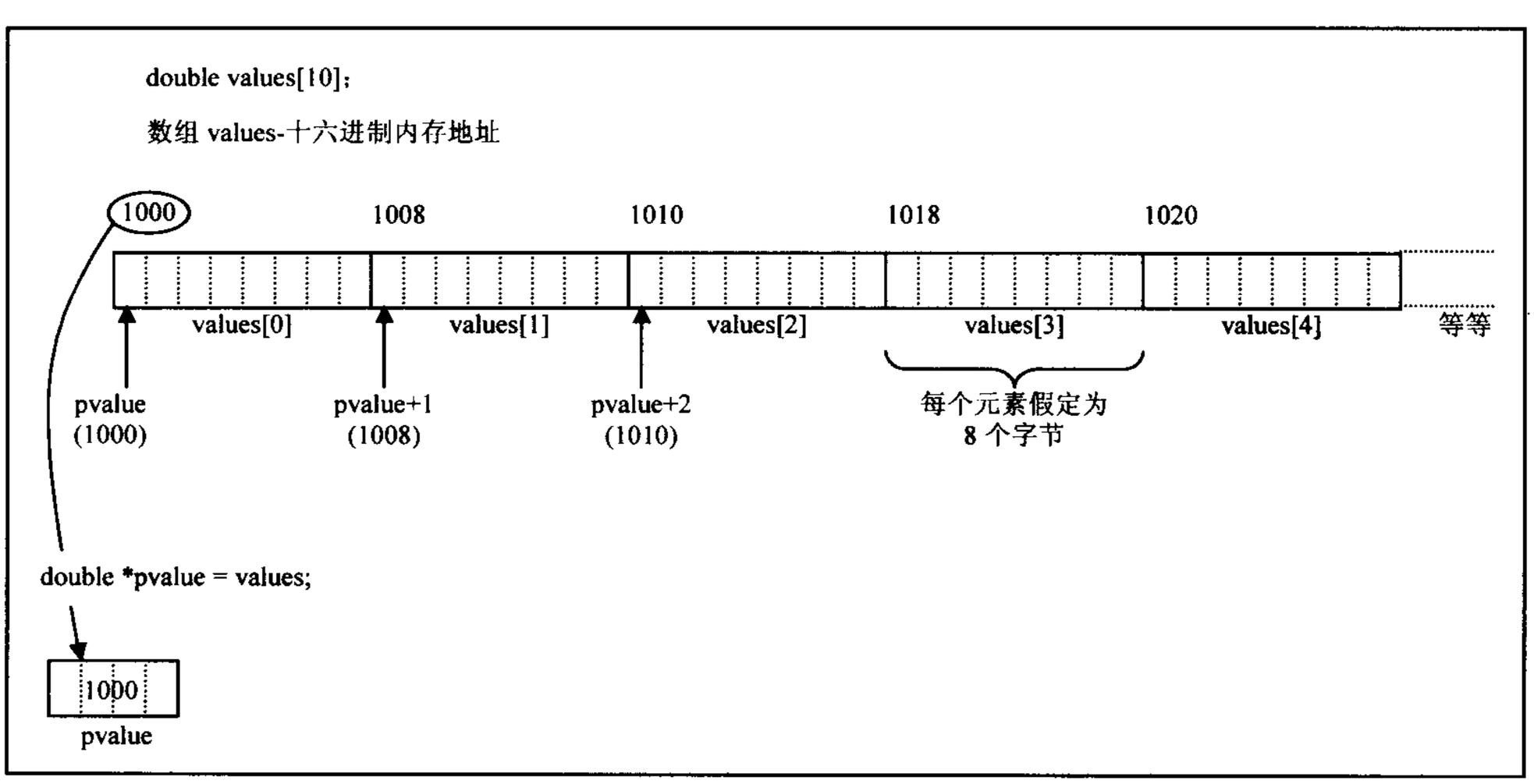


图 7-5　指针的递增

如图 7-5 所示，pvalue 开始存储的地址对应于数组的开始。给 pvalue 加 1 就把它包含的地址递增 8，结果就是下一个数组元素的地址。给指针递增 2，就是把该指针移动两个元素。当然，指针 pvalue 不一定指向 values 数组的开始。下面的语句把数组中第 3 个元素的地址赋予该指针：

```
pvalue=&values[2];
```

表达式 pvalue+1 就等于 values[3]的地址，即 values 数组的第 4 个元素。下面的语句可以使指针直接指向这个元素：

```
pvalue +=1;
```

这个语句给包含在pvalue中的地址加上values数组的一个元素占用的字节数。一般情况下，表达式pvalues+n(其中n是等于整数的任意表达式)的结果是给包含在指针pvalue中的地址加上n*sizeof(double)，因为pvalue声明为指向double的指针。

这个逻辑可应用于从指针中减去一个整数。如果pvalue包含values[2]的地址，则表达式pvalue－2就等于数组中第一个元素values[0]的地址。换言之，递增或递减指针的操作将根据指针所指向的对象的类型来进行。给指向long的指针递增1，会使它的内容变成下一个long地址，即给地址加上sizeof(long)个字节。从指针中减去1，就是给它包含的地址减去sizeof(long)。

注意：

指针算术运算所得的地址范围应该是从该指针所指向的数组中第一个元素的地址到比最后一个元素多1的地址。若超出了这个范围，该操作就是不确定的。

当然，可以对已执行了算术运算的指针解除引用(否则就不是指向它了)。例如，假定pvalue仍指向values[2]，则下面的语句

```
*(pvalue+1)=*(pvalue+2);
```

就等价于：

```
values[3]=values[4];
```

在递增了指针包含的地址后，要解除对指针的引用，就需要使用括号，因为间接运算符的优先级高于算术运算符+和－。如果使用表达式*pvalue+1，而不是*(pvalue+1)，就会给存储在pvalue中所包含的地址值加1，也就是values[2]+1。而且，由于结果是一个数值，不是地址(因此不是lvalue)，故若在上述赋值语句中使用它，编译器就会产生一个错误消息。

像pvalue+1这样的表达式不会改变pvalue中的地址，这个表达式的结果跟pvalue的类型相同。在本例中，其结果是pvalue所指向的元素后面的一个元素地址。

当然，如果指针包含无效的地址(例如在它指向的数组上下限之外的地址)，使用该指针来存储一个值，就会改写该地址所在的内存。这一般会导致灾难，程序将以某种方式失败。问题的原因是误用了指针，而这一原因不是很明显。

计算两个指针之间的差

可以从一个指针中减去另一个指针，但这仅在指针的类型相同，且指向同一个数组中的元素时才有意义。假定有一个一维数组number，其类型是long，声明语句如下：

```
long numbers[]={10L,20,30,40,50,60,70,80};
```

声明并初始化两个指针变量：

```
long *pnum1=&numbers[6];                    //Points to seventh array elememt
long *pnum2=&numbers[1];                    //Points to second array elememt
```

现在计算这两个指针之间的差：

```
int difference=pnum1-pnum2;                    //Result is 5
```

变量difference是5，因为地址之间的差由元素而不是由字节来决定。

7.5.2 使用数组名的指针表示法

可以把数组名用作指针，来确定数组元素的地址。如果把一维数组声明为

```
long data[5];
```

就可以使用指针表示法，例如，可以把元素 data[3]表示为*(data+3)。这种表示法可以应用于一般情形，如元素 data[0]、data[1]、data[2]等，可以分别表示为*data、*(data+1)、*(data+2)等。数组名 data 本身表示数组开始时的地址，而表达式 data+2 就表示跟起始位置偏离两个元素的地址。

数组名的指针表示法与索引值的表示法的使用方式相同，这种表示法可以用于表达式或等号的左边。下面的循环把 data 数组的值设置为整数：

```
for(int i=0;i<5;i++)
  *(data+i)=2*(i+1);
```

(data+i)表示数组中的元素，(data+0)对应于 data[0]，*(data+1)对应于 data[1]，依此类推。循环把数组元素的值设置为 2、4、6、8、10。

如果要累加数组元素的值，就可以使用下面的语句：

```
long sum=0;
for(int i=0;i<5;i++)
  sum+=*(data+i);
```

下面在一个有更多实际内容的例子中使用这个表示法。

程序示例 7.6——把数组名用作指针

前面对字符串使用了指针，下面对数组使用指针，在一个面向数值的程序中计算质数(质数是只能被 1 和它本身整除的整数)。

```
//Program 7.6 Calculating primes
#include <iostream>
#include <iomanip>
using std::cout;
using std::endl;

int main() {
const int max = 100;                    // Number of primes required
long primes[max] = {2, 3, 5};           // First three primes defined
int count = 3;                          // Count of primes found
long trial = 5;                         // Candidate prime
bool isprime = true;                     // Indicates when a prime is found

do {
   trial += 2;                          // Next value for checking
   int i = 0;                           // Index to primes array

   // Try dividing the candidate by all the primtes we have
   do {
```

```
      isprime = trial % *(primes + i) > 0; // False for exact division
    } while(++i < count && isprime);

    if(isprime)                          // We got one...
      *(primes + count++) = trial;       // ...so save it in primes array
} while(count < max);

// Output primes 5 to a line
for(int i = 0 ; i < max ; i++){
    if(i % 5 == 0)                  // Newline on 1st line and after every 5th prime
      cout << endl;
    cout << std::setw(10)  << *(primes + i);
  }
  cout << endl;
  return 0;
}
```

这个程序的输出如下：

```
         2         3         5         7        11
        13        17        19        23        29
        31        37        41        43        47
        53        59        61        67        71
        73        79        83        89        97
       101       103       107       109       113
       127       131       137       139       149
       151       157       163       167       173
       179       181       191       193       197
       199       211       223       227       229
       233       239       241       251       257
       263       269       271       277       281
       283       293       307       311       313
       317       331       337       347       349
       353       359       367       373       379
       383       389       397       401       409
       419       421       431       433       439
       443       449       457       461       463
       467       479       487       491       499
       503       509       521       523       541
```

例子的说明

利用#include 语句包含用于输入输出的<iostream>和<iomanip>，因为本程序要使用流操纵程序设置输出的字段宽度。利用常量 max 定义程序要生成的质数个数：

```
const int max=100;                  //Number of primes required
long primes[max]={2,3,5}            //First three primes defined
int count=3;                        //Count of primes found
```

存储结果的 primes 数组已在过程的开始处定义好了前 3 个质数。因为变量 count 记录质数的个数，所以它初始化为 3。

质数的测试要使用下面语句声明的变量：

```
long trial=5;                    //Candidate prime
bool isprime=true;               //Indicates when a prime is found
```

因为变量 trial 存储了下一个要测试的整数，所以开始时将它设置为 5。布尔变量 isprime 是一个标志，用于表示 trial 的值是否为质数。

所有的工作都是在两个循环中完成的。外层的 do-while 循环提取下一个要测试的整数，如果它是一个质数，就把该整数添加到 primes 数组中。内层的循环测试整数，看看它是否是质数。在填满 primes 数组后，外层的循环就停止。

在执行内层循环之前，下面的语句把变量 trial 设置为下一个要测试的整数：

```
trial +=2;                       //Next value for checking
```

循环中的算法非常简单，它的依据是：任何不是质数的整数都可以被更小的质数整除。因为本程序以升序的方式查找质数，所以 primes 数组总是包含了比当前整数小的所有质数。如果所有的质数都不是当前整数的除数，则该整数一定是一个质数。

注释：

实际上，如果当前整数只能被小于或等于该整数的平方根的质数整除，就需要测试，因此本例并不是很高效。

在内层循环中检查变量 trial 是否为质数：

```
do {
  isprime= trial % *(primes+i)>0;              //False for exact division
} while(++i<count && isprime);
```

在循环中，isprime 设置为表达式 trial % *(primes+i)>0 的值，确定 trial 除以存储在(primes+i)地址中的质数后的余数。如果该余数是正数，isprime 就是 true。

如果 i 达到 count 或 isprime 设置为 false，该循环就结束。如果 trial 能被 primes 数组中的质数整除，trial 就不是质数，就结束循环。如果 trial 不能被 primes 数组中的所有质数整除，isprime 就是 true，循环在 i 达到 count 时结束。

在内层循环因为 isprime 设置为 false 或用尽了 primes 数组中的除数而结束之后，就必须确定 trial 中的值是否为质数。这由 isprime 中的值表示，该 isprime 值在 if 语句中测试：

```
if(isprime)                          //We got one...
  *(primes+count++)=trial;           //...so save it in primes array
```

如果 isprime 包含 false，就说明在除法运算中 trial 可以被某个 primes 数组元素整除，trial 不是质数。如果 isprime 是 true，赋值语句就把 trial 中的值存储在 primes[count]中，再用后缀递增运算符递增 count。

一旦找到 max 个质数，就显示它们，一行显示 5 个质数，字符宽度设置为 10 个字符，如其语句如下：

```
if(i%5==0)                           //Newline on 1st line and after every 5th prime
  cout<<endl;
cout<<std::setw(10)<<*(primes+i);
```

当 i 的值是 0、5、10 等时，就另起一行。

7.5.3 对多维数组使用指针

使用指针存储一维数组的地址是相当简单的，但对于多维数组，就比较复杂了。这个主题比较难，读者第一次阅读时可以跳过去。但是，如果以前使用过 C 语言，本节就值得一看。

对多维数组使用数组表示法通常可以完成需要完成的所有工作，最好坚持使用数组表示法。如果必须对多维数组使用指针，就需要弄明白这么做会发生什么情况。下面通过具体的例子来说明。把一个数组 beans 声明为：

```
double beans[3][4];
```

再声明并初始化指向 double 的指针变量 pbeans：

```
double* pbeans=&beans[0][0];
```

这里把指针设置为 double 类型的数组中第一个元素的地址。下面的语句还可以把指针设置为数组中第一行的地址：

```
double* pbeans=beans[0];
```

这等价于使用一维数组名，被其地址替代了，前面讨论过这一点。但是，因为 beans 是一个二维数组，所以下面将一个地址放在指针中的尝试是非法的：

```
double* pbeans=beans;                    //Will cause an error!!
```

问题是类型不匹配。前面定义的指针类型是 double*，而数组的类型是 double[3][4]。存储这个数组地址的指针必须也是 double*[4]类型。C++把数组的大小与其类型关联起来，上面的语句只有在指针声明为要求的大小时才是合法的。其记号要比前面的复杂一些：

```
double (*pbeans)[4]=beans;
```

这里的圆括号是必需的，否则，就声明了一个指针数组。现在指向 beans 的指针的初始化是合法的，但应注意，这个指针只能用于存储上述大小的数组地址。

对多维数组名使用指针表示法

可以对数组名使用指针表示法来引用数组的元素。可以用 3 种方式引用上面声明的 beans 数组中的每个元素，该数组有 3 行 4 列元素：

- 以通常的方式使用数组名，带有两个索引值

例如 beans[i][j]。它使用约定的数组索引来引用数组中的第 i 行第 j 个元素。

- 在指针表示法中使用数组名

例如*(*(beans+i)+j)。从内层开始确定这个表达式的含义。beans 表示数组中第一行的地址，beans+i 表示数组的第 i 行。表达式*(beans+i)是第 i 行上第一个元素的地址，*(beans+i)+j 是第 i 行上第 j 个元素的地址。整个表达式*(*(beans+i)+j)就是该元素的值。除非有很充分的理由用这种方式引用数组的元素，否则最好避免使用它。这种表示法的含义不是很明确，代码也会变得难以理解。

- 混合使用指针表示法和索引值

这是合法的，但不推荐使用。下面两个语句合法引用了数组中的同一个元素：

```
*(beans[i]+j)
(*(beans+i))[j]
```

其中混合使用了数组和指针表示法。

程序示例 7.7——对多维数组使用指针表示法

在程序示例 5.8 中，生成了一个乘法表。下面就编写这个程序的另一个版本，把表存储为一个数组。为了避免重复不需要再解释的代码，我们把它简化为计算固定大小的表：

```
// Program 7.7 Using pointer notation with a multidimensional array
#include <iostream>
#include <iomanip>
#include <cctype>
using std::cout;
using std::endl;
using std::setw;

int main(){
   const int table = 12;                          // Table size
   long values[table][table] = {0};               // Stores the table values

   // Calnculate the table entries
  for(int i = 0; i < table ; i++)
    for(int j= 0; j < table ; j++)
      *(*(values + i) + j) = (i + 1)  *  (j + 1);   // Full use of pointer notation

  // Create the top line of the table
  cout  <<   "           | ";
  for(int i = 1 ;  i <= table ;  i++)
     cout << " " << setw(3)  << i <<  "  |";
  cout << endl;

  // Create the separator row
  for(int i = 0 ; i <= table ; i++)
     cout  <<  " ------";
  cout << endl;

   for(int i = 0 ; i < table ; i++) {                // Iterate over the rows
      cout<<" "<<setw(3)<<i+1<<"  |";                // Start the row

      //Output  the  values in a row
      for(int j=0;j<table;j++)
         cout<<" "<<setw(3)<<values[i][j]<< "  |";   // Array notation
      cout<<endl;                                    // End the row
   }
   return 0;
}
```

这个程序的输出如下：

```
   |   1 |   2 |   3 |   4 |   5 |   6 |   7 |   8 |   9 |  10 |  11 |  12 |
---------------------------------------------------------------------------
 1 |   1 |   2 |   3 |   4 |   5 |   6 |   7 |   8 |   9 |  10 |  11 |  12 |
 2 |   2 |   4 |   6 |   8 |  10 |  12 |  14 |  16 |  18 |  20 |  22 |  24 |
 3 |   3 |   6 |   9 |  12 |  15 |  18 |  21 |  24 |  27 |  30 |  33 |  36 |
 4 |   4 |   8 |  12 |  16 |  20 |  24 |  28 |  32 |  36 |  40 |  44 |  48 |
 5 |   5 |  10 |  15 |  20 |  25 |  30 |  35 |  40 |  45 |  50 |  55 |  60 |
 6 |   6 |  12 |  18 |  24 |  30 |  36 |  42 |  48 |  54 |  60 |  66 |  72 |
 7 |   7 |  14 |  21 |  28 |  35 |  42 |  49 |  56 |  63 |  70 |  77 |  84 |
 8 |   8 |  16 |  24 |  32 |  40 |  48 |  56 |  64 |  72 |  80 |  88 |  96 |
 9 |   9 |  18 |  27 |  36 |  45 |  54 |  63 |  72 |  81 |  90 |  99 | 108 |
10 |  10 |  20 |  30 |  40 |  50 |  60 |  70 |  80 |  90 | 100 | 110 | 120 |
11 |  11 |  22 |  33 |  44 |  55 |  66 |  77 |  88 |  99 | 110 | 121 | 132 |
12 |  12 |  24 |  36 |  48 |  60 |  72 |  84 |  96 | 108 | 120 | 132 | 144 |
```

例子的说明

首先声明数组的大小及数组本身：

```
const int table=12;                          //Table size
long values[table] [table]={0};                 //stores the table values
```

必须把 table 声明为 const，这是因为它将用于指定表的大小。数组的大小必须是一个常量表达式，不允许使用不是常量的变量。该数组有 12 行，每一行 12 个元素。

在一个嵌套的 for 循环中，使用指针表示法存储表中的数据项：

```
for(int i=0;i<table;i++)
  for(int j=0;j<table; j++)
    *(*(values+i)+j)=(i+1)*(j+1);            //Full use of pointer notation
```

表达式 values+i 表示数组中的第 i 行；*(values+i)表示第 i 行上第一个元素的地址。给表达式*(values+i)加上 j，变成*(values+i)+j，就获得了第 i 行第 j 个元素的地址。用表达式*(*(values+i)+j)解除指针的引用，就得到了数组中第 i 行第 j 个元素的内容。

为了输出表中的数据项，需要编写另一个嵌套循环：

```
for (int i = 0 ; i < table ; i++) {                  // Iterate over the rows
  cout << " " << setw(3) << i + 1 << " |";           // Start the row

  // Output the values in a row
  for (int j = 0 ; j < table ; j++)
    cout << " " << setw(3) << values[i][j] << " |"; // Array notation
  cout << endl;                                      // End the row
}
```

使用一般的数组表示法输出存储在数组中的值。用指针表示法表示的表达式*(*(values+i)+j)等价于用数组表示法表示的 values[i][j]。显然，数组表示法理解起来要容易得多。

7.5.4 C 样式字符串的操作

第 4 章简要介绍了在<cctype>头文件中声明的函数，这些函数可以分析和转换单个字符。标准库还包含一些可用于分析和转换非空字符串的函数。这些函数在头文件<cstring>中声明，

要使用它们，必须包含头文件<cstring>。头文件<cctype>和<cstring>的名称都以 c 开头，表示它们都是继承自 C 的库。

注意：

不要把这里使用的头文件<cstring>和前面的<string>头文件相混淆，后者定义了 string 类型。

头文件<cstring>中的函数在参数和返回值上都使用了指针，这也是在本节讨论它们的原因。这些函数提供的非空字符串的功能类似于第 6 章介绍的 string 对象的功能。在这一方面，这些函数有些落伍，但它们仍存在于语言中，读者以前肯定见过它们。

这里对 C 样式字符串的操作比较感兴趣，因为它们提供了一些指针操作的练习。头文件<cstring>中共有 22 个函数，这里仅介绍连接非空字符串的函数。

连接字符串

第 6 章介绍了如何使用+运算符组合 string 对象。对于非空字符串，就不是这么简单了。头文件<cstring>声明了两个函数 strcat()和 strncat()，用于连接非空字符串。

函数 strcat()带两个参数，分别是 char*类型和 const char*类型的字符串，该函数把后一个参数追加到前一个参数中，在这个过程中修改第一个参数。它会改写第一个字符串尾部的'\0'，并把'\0'加到组合字符串的最后，但用户应确保第一个参数指向的字符串有足够的空间容纳结果。第一个参数也是函数的返回值。

strncat()函数带有三个参数，前两个参数与 strcat()相同。第三个参数是一个整数，指定把第二个字符串的多少个字符加到第一个字符串上。如果指定的字符个数多于字符串中的字符个数，就追加整个字符串。除此之外，该函数的使用条件和返回值与 strcat()相同。

这两个函数很容易使用。假定声明两个 char 类型的数组：

```
char name[50]= "Bing";
char surname[]="Crosby";
```

下面的语句把一个空格追加到 name 字符串上：

```
strcat(name, " ");
```

接着，下面的语句追加 surname：

```
strcat(name, surname);
```

结果是 name 包含字符串"Bing Crosby"。因为函数总是返回第一个参数，故可以用下面的语句完成上述两个操作：

```
strcat(strcat(name, " "), surname);
```

其缺点是不容易理解。在这个语句中，第一个 strcat()函数调用把一个空格追加到 name 上，其结果再用作外层 strcat()函数调用的第一个参数，而外层 strcat()函数调用追加了 surname。

为了验证在第一个参数中有足够的空间容纳组合字符串，可以使用 strlen()函数，它返回其惟一参数的字符串的长度。如下所示：

```
if(sizeof name/sizeof name[0]>(strlen(name)+strlen(surname)+1))
  std::cout<<strcat(strcat(name, " "), surname);
```

if 条件计算 name 包含多少个字符，并确保该字符数大于要构建的字符串的长度(if 条件中的＋1 是用于空格字符)。

在 cstring 中定义的其他函数执行的操作有复制(strcpy())、比较(strcmp())和搜索(strchr())非空字符串，它们在操作上类似于上面介绍的函数。函数把指向 char 的指针作为参数，避免了必须在内存中移动和复制整个字符串的系统开销，第 8 章和本书的其他部分将进一步讨论这些内容。

学习了这些函数后，以后如果读者见到它们，就应认识它们，因为它们很好地演示了指针的用法。在编写代码时，如果需要进行字符串处理，最好使用 string 对象，而不是使用 char* 类型的变量。

7.6 动态内存分配

前面编写的所有代码都是在编译期间给数据分配内存空间。我们在源代码中指定需要的变量和数组大小，在执行程序时，这些内存空间已分配好了，不管是否需要。在程序中使用固定的变量集合是非常受限制的，而且常常比较浪费。

程序示例 7.5 从文本字符串中提取单词时，为 1000 个单词分配了内存空间。在许多情况下，并不需要存储 1000 个单词，但分配了超过需要的内存空间。这将阻止其他程序使用内存空间，但在程序示例 7.5 中这些空间完全没有用。另一方面，如果试图分析一个包含 1001 个单词的文本字符串，该程序就不能处理它——尽管计算机中有许多空闲的内存空间。

在另一个环境下，对一组数据使用一个很大的整数数组是很合适的，但对另一组输入数据需要使用一个很大的浮点数数组。如果需要的数组非常大，机器上就没有足够的内存空间同时存储这些数据。所有这些困难和缺点的解决方案是使用动态内存分配，它意味着在程序运行时(运行期间)为正在处理的数据分配存储它所需要的内存空间，而不是在编译它时(编译期间)分配空间。图 7-6 显示了静态和动态内存分配的区别。

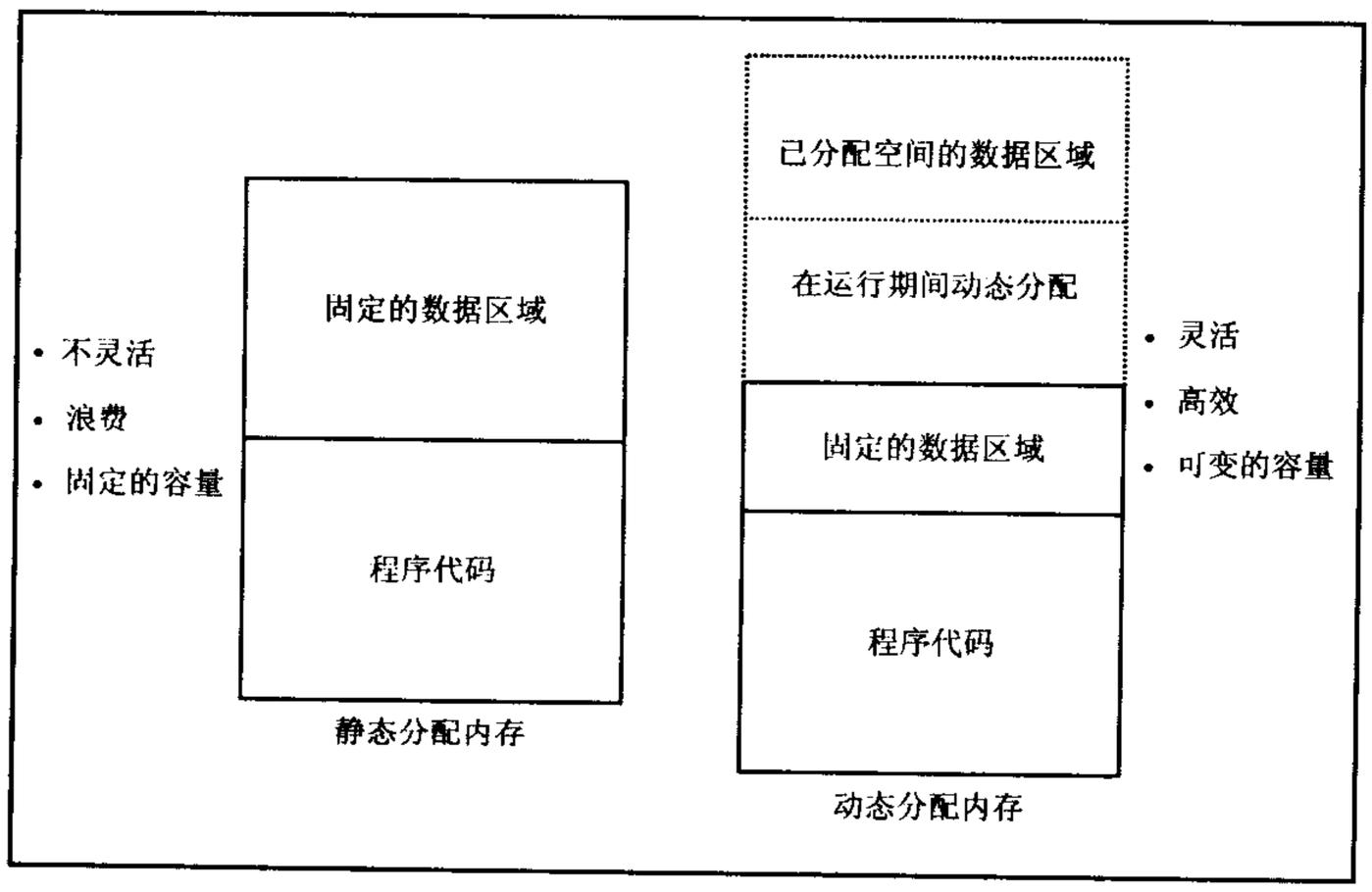


图 7-6 静态和动态内存分配

因为根据定义，动态分配内存的变量不能在编译期间声明，所以它们不能在源程序中指定。在动态分配内存时，请求分配的空间由其地址标识。存储这个地址的惟一地方显然是指针。利

用指针的功能和 C++中的动态内存管理工具，编写具有这种灵活性的程序就非常快捷和容易了。

第 3 章介绍了变量可以拥有的三种存储持续时间：自动、静态和动态，并讨论了前两种变量如何创建。在自由存储区中分配内存的变量总是具有动态的存储持续时间。

7.6.1 自由存储区

在大多数情况下，程序执行时计算机上总是有未使用的内存。在 C++中，这些未使用的内存称为自由存储区，有时称为堆(heap)。使用一个特殊的 C++运算符，就可以把自由存储区中的空间分配给给定类型的新变量，并返回所分配空间的地址。这个运算符就是 new，同它对应的运算符是 delete，它可以释放以前用 new 分配的内存。

可以在程序的一个部分把自由存储区中的空间分配给一些变量，在使用完这些变量后，就释放已分配的空间，把它返回给自由存储区。这样，这些内存就可以在程序的后面由其他动态分配的变量重复使用了。这将允许非常高效地使用内存。在许多情况下，程序将能处理非常大的数据，涉及的数据量要比不使用动态分配的变量多得多。

注意：

在使用 new 为变量分配内存空间时，就是在自由存储区创建该变量。在变量占用的内存没有用运算符 delete 释放前，该变量将一直存在。

7.6.2 运算符 new 和 delete

假定一个 double 类型的变量需要内存空间。可以定义一个指向 double 类型的指针，再在执行程序时，请求为该变量分配内存空间。具体方法是在下面的语句中使用运算符 new：

```
double* pvalue=0;              //Pointer initialized with null
pvalue=new double;             //Request memory for a double variable
```

注意，所有的指针都应初始化。使用动态分配的内存一般涉及到许多浮动的指针，这些指针不应包含垃圾值，这是非常重要的。如果指针没有包含合法的地址，就应总是让它包含 0。

在上述代码中，第二行中的 new 运算符会返回在自由存储区给 double 变量分配的内存地址，这个地址应存储在 pvalue 指针中。接着就可以通过前面介绍的间接运算符，使用这个指针引用该变量了。例如：

```
*pvalue=999.0;
```

当然，在极端情况下，由于自由存储区的内存空间已经用尽，内存分配就不可行。另外，自由存储区还有可能因以前的使用而被分隔成了小碎块，此时自由存储区就不能提供一个足够大的连续空间来容纳要获得内存空间的变量了。存储一个 double 值不可能需要这么大的空间，但在处理非常大的实体时，例如数组或复杂的类对象，就需要很大的空间。显然应考虑这种情况，但现在仅假定我们总是能得到需要的内存空间。第 17 章在讨论异常时，再回过头来讨论这种情况。

可以初始化用 new 创建的变量。再看看前面的例子：double 变量用 new 分配内存空间，

其地址存储在 pvalue 指针中。下面的语句可以把它的值初始化为 999.0：

```
pvalue=new double(999.0);          //Allocate a double and initialize it
```

当不再需要动态分配内存的变量时，就可以使用 delete 运算符，释放它在自由存储区中占用的内存：

```
delete pvalue;          //Release memory pointed to by pvalue
```

这将确保该内存可以在以后由另一个变量使用。如果没有使用 delete，后来又在指针 pvalue 中存储了另一个地址，就不能释放原来的内存空间，也不能使用它包含的变量，因为不能再访问该地址了。

注意，delete 运算符释放了内存，但没有改变指针。运行完上面的语句后，pvalue 仍包含已分配给它的内存地址，但该内存现在已经自由了，可以立即分配给其他实体，例如另一个程序。为了避免使用包含垃圾地址的指针，除非要重新给它赋值或它已超出作用域，否则就应在释放内存时重新设置该指针。为此，可使用下面的语句：

```
delete pvalue;          //Release memory pointed to by pvalue
pvalue=0;               //Reset the pointer to 0
```

7.6.3 数组的动态内存分配

为数组动态分配内存是很简单的。假定已经把 pstring 声明为“指向 char”类型，就可以使用下面的语句为 char 类型的数组分配自由存储区中的内存空间：

```
pstring=new char[20];              //Allocate a string of twenty characters
```

这个语句为包含 20 个字符的 char 数组分配内存空间，并把它的地址存储在 pstring 中。要删除刚才在自由存储区中创建的数组，必须使用 delete 运算符。语句如下：

```
delete [] pstring;                 //Delete array pointed to by pstring
```

注意：

这里的方括号非常重要，它们表示要删除的是一个数组。从自由存储区中删除数组时，应总是包含方括号，否则结果就是不可预料的。还要注意不能在这里指定维数，只写上[]就可以了。

当然，还应重新设置指针，因为它已不再指向原来的内存了：

```
pstring=0;                 //Reset the pointer
```

下面先使用数值数据来说明动态内存分配的工作方式。

程序示例 7.8——使用自由存储区

重新编写前面计算 100 个质数的例子，学习这些操作如何进行。这次计算任意个质数，并使用自由存储区中的内存存储需要的质数。

```
// Program7.8 Calculating primes using dynamic memory allocation
#include <iostream>
```

```
#include <iomanip>
using std::cout;
using std::endl;
using std::cin;

int main() {
   int max= 0;                          // Number of primes required
   int count = 3;                       // Count of primes found
   long trial = 5;                      // Candidate prime
   bool isprime = true;                 // Indicates when a prime is found

cout << endl
     << "Enter the number of primes you would like: ";
cin >> max;                             // Number of primes required

long* primes = new long[max];           // Allocate memory for them
*primes = 2;                            // Insert three seed primes...
*(primes + 1) = 3;
*(primes + 2) = 5;

do {
  trial += 2;                           // Next value for checking
  int i = 0;                            // Index to primes array

  // Try dividing the candidate by all the primes we have
  do {
    isprime = trial % *(primes + i) > 0;   // False for exact division
  } while(++i < count && isprime);

  if(isprime)                           // We got one...
    *(primes + count++)  = trial;       //  ...so save it in primes array
} while(count < max);

// Output primes 5 to a line
for(int i = 0  ;  i < max ;  i++) {
  if(i % 5 == 0)                  // Newline on 1st line and after every 5th prime
    cout << endl;
  cout << std::setw(10)  << *(primes + i);
}
cout << endl;
delete [] primes;                       // Free up memory
return 0;
}
```

输出与该程序的前一个版本大致相同，当然需要先选择要计算多少个质数，这里不再重新生成。

例子的说明

这个程序在整体上非常类似于以前的版本。在从键盘上读取需要的质数个数，并存储在 int 变量 max 之后，就使用运算符 new 在自由存储区中为该大小的数组分配内存空间。把变量 max 放在数组类型声明后面的方括号中，为数组指定需要的大小：

```
long* primes=new long[max];              //Allocate memory for them
```

new 返回的地址存储在指针 primes 中，这是 long 数组的 max 个元素中的第一个元素地址。接着，把前 3 个数组元素设置为前 3 个质数的值：

```
* primes=2;                              //Insert three seed primes…
* (primes+1)=3;
* (primes+2)=5;
```

这里使用了指针表示法，也可以使用数组表示法。如果愿意，可以把这 3 个语句改写为：

```
primes[0]=2;                            //Insert three seed primes…
primes[1]=3;
primes[2]=5;
```

注意：

不能为动态分配内存的数组元素指定初始值。必须使用显式赋值语句，为数组元素赋值。

质数的计算与前面相同。即使质数占用的内存是在运行期间分配的，也不需要修改。同样，输出过程也是一样的。动态获得内存空间根本不是问题。一旦分配了内存空间，就不会影响编写计算过程的方式。

使用完数组后，就要使用 delete 运算符把它从自由存储区中删除，不要忘了包含方括号(表示要删除的是一个数组)：

```
delete [] primes;                       //Free up memory
```

由于这是程序的结尾，不可能误用指针，因此不需要把指针重新设置为 0。如果程序要继续进行其他操作，当然要采用不同的方式处理。

7.6.4 动态内存分配的危险

在动态分配内存空间时，可能出现两种问题。第一种称为内存泄漏，是由代码中的错误导致的。可是内存泄漏是相当常见的。第二种是内存碎片，通常是由于动态分配的内存使用不当而造成的。内存碎片问题比较少见。

1. 内存泄漏

在使用 new 分配内存空间时，由于在使用完该内存后没有释放它，从而会出现内存泄漏。在这种情况下，常常因为改写了用于访问它的指针中的地址，而丢失了内存块的地址。这一般在循环中发生，而且比我们想像的更容易出这类问题。结果是程序在自由存储区中消耗的内存空间越来越多，并可能在分配完自由存储区中的所有空间之后，再次请求分配内存时失败。如图 7-7 所示。

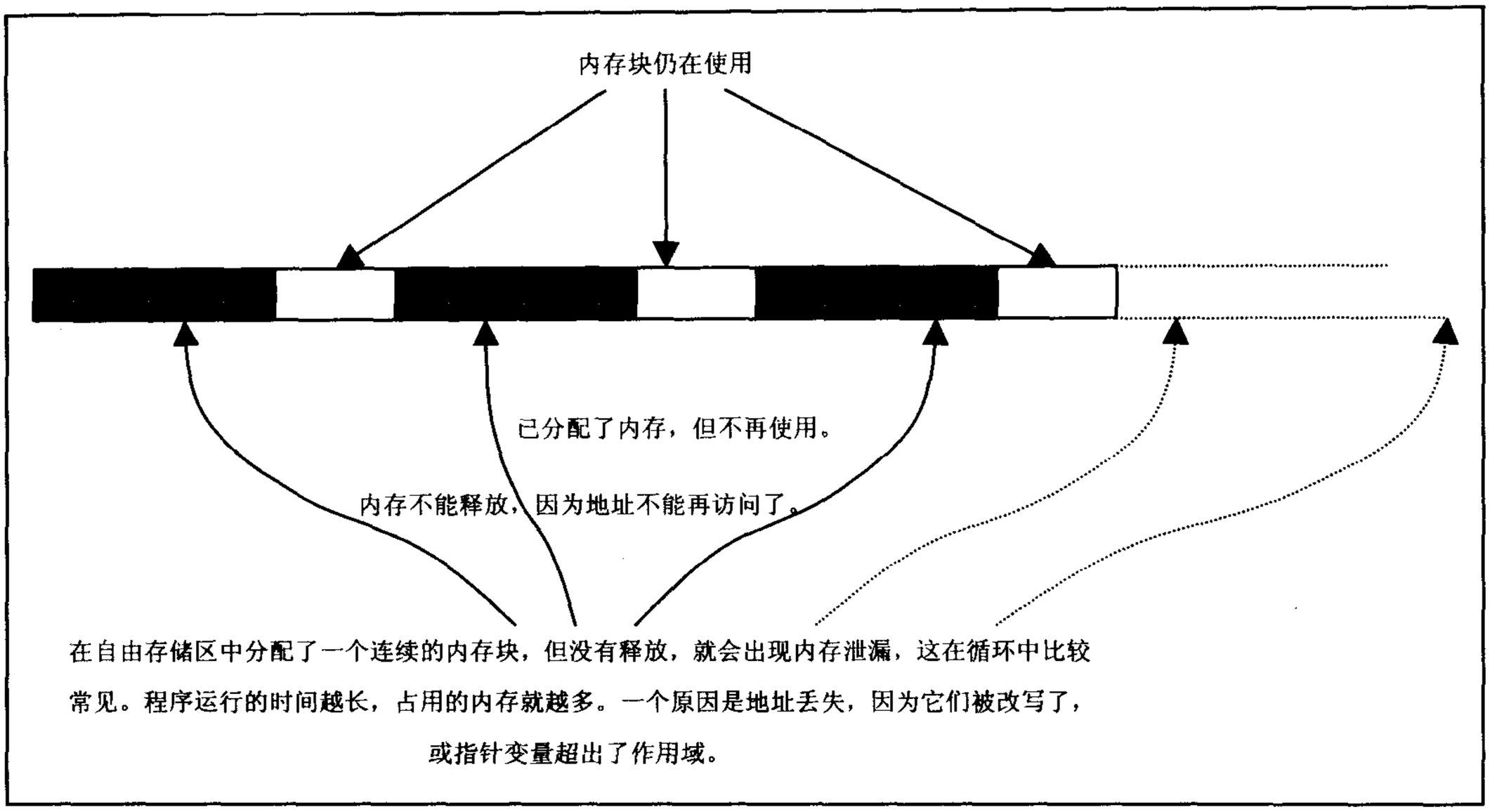


图 7-7　内存泄漏

很容易看出，在所分配的内存附近的代码中，该内存在某个地方不再使用了，但忘记使用 delete 释放内存了。在复杂的程序中很难确定在哪里不再使用该内存了，内存是在程序的一个部分分配的，却应在另一个完全不同的部分释放。最好的解决方法是在使用 new 运算符之后的某个适当的地方，添加 delete 操作。

指针和变量作用域

谈到作用域时，指针同其他变量是一样的。指针变量的作用域是从块中声明它的地方开始，到该块结束为止。在该块之后，该指针就不存在了，因此它包含的地址也不能访问了。

这与内存泄漏的讨论相关。如果指针包含自由存储区中某个内存块的地址，在该指针超出作用域时，就不能释放该内存块了。最重要的是要考虑指针的作用域，特别是在使用动态分配的内存时，就更应考虑。

2. 自由存储区的碎片

内存碎片在频繁地分配和释放内存块的程序中会出现。每次使用 new 运算符时，都会分配自由存储区中的一个连续内存空间。如果创建并释放许多不同大小的变量，所分配的内存就有可能散布在小的自由内存块中，每个小内存块都不足以容纳程序(或同时执行的其他程序)需要的新动态变量。自由内存的总量非常大，但每个内存块都相当小，不能满足当前的需求。如图 7-8 所示。

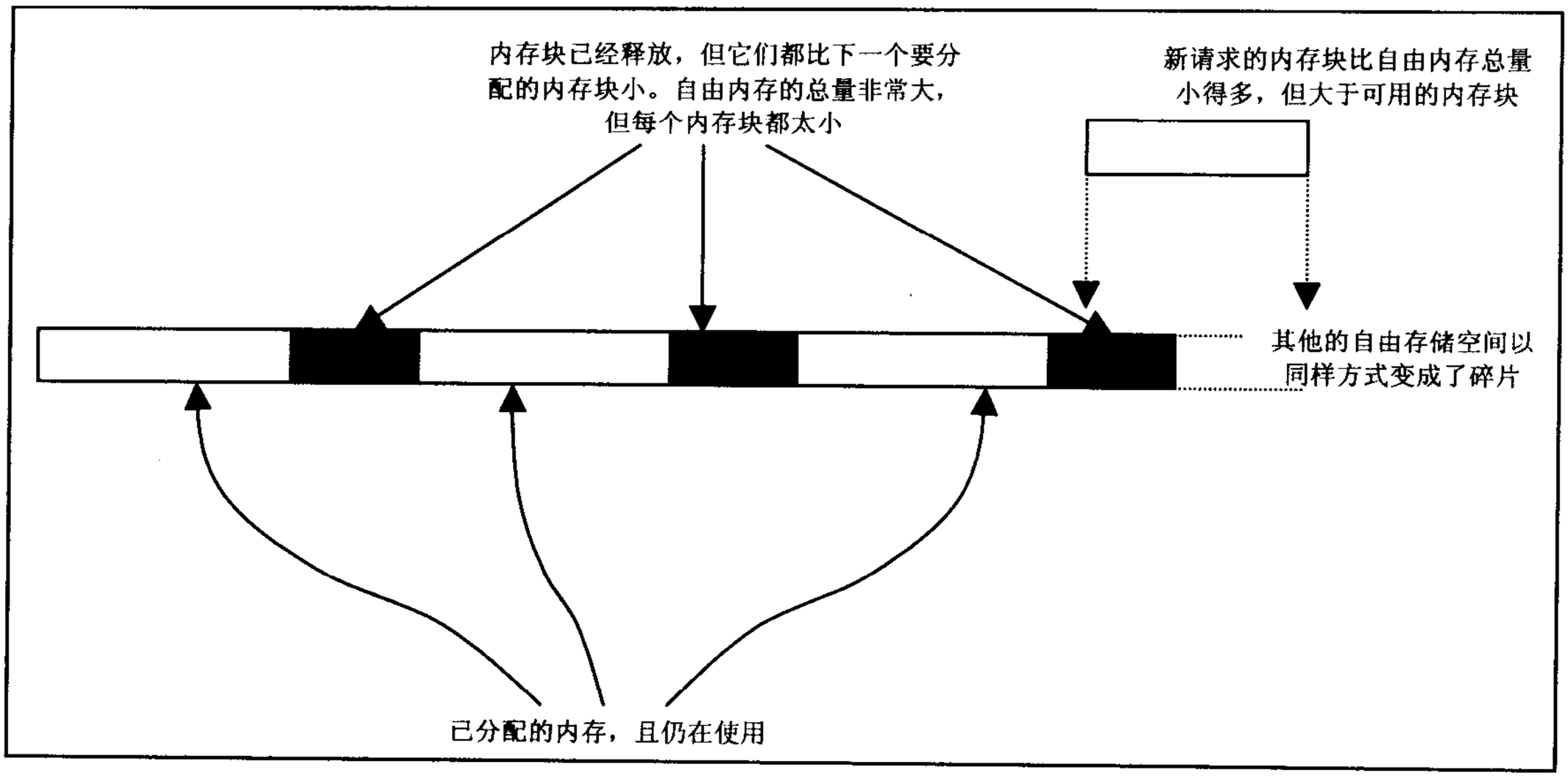


图 7-8　自由存储区的碎片

这个问题目前比较少见，因为在最现代的计算机上，虚拟内存提供了非常大的内存。在发生这个问题时，解决方法是避免分配较小的内存块，或者自己分配较大的内存块，并管理内存的使用。

程序示例 7.9——给字符串排序

本节刚开始讨论动态内存分配时，批评了从文本字符串中提取单词并排序的程序示例，现在来改善该程序示例。这里创建的版本不对文本设置 1000 个单词的限制，而且即使输入少于 1000 个单词，也不会浪费内存。为了恰当地修改原代码，应一步一步地修改旧代码，组装成新版本。

从这个程序中可以看出，使用动态分配的内存不需要对程序的机制做太大的修改。这里对程序示例 7.5 的修改大都与动态内存的管理有关。下面把代码分成多个部分进行说明，并突出显示新增和修改的代码。

读取并计数多行字符串中的单词

实际上，可以用程序示例 7.5 中的方式借助键盘读取字符串。区别是何时和如何把指针赋予字符串中的单词。上一次在输入字符串之前，创建了一个固定的指针数组，并在对单词计数时把它们赋予指针。而这次将在输入文本后创建指针，为此，需要确定文本中有多少个单词：

```
// Program 7.9 Sorting strings using dynamic memory allocation
#include <iostream>
#include <string>
using std::cout;
using std::cin;
using std::endl;
using std::string;

int main() {
   string text;                                    // The string to be sorted
```

```
const string separators ="  ,.\"\n";      // Word delimiters

// Three lines deleted that allocated the array

// Read the string to be searched from the keyboard
cout << endl <<  "Enter a string terminated by #:" << endl;
std::getline(cin,  text,  '#');

// Count the words in the text
size_t start = text.find_first_not_of(separators);// Word start index
size_t end = 0;                                   // End delimiter index
int word_count = 0;                               // Count of words stored

while(start != string::npos) {     // Deleted the check on the size of the
                                      array
   end = text.find_first_of(separators,  start + 1);
   if(end == string::npos)                        // Found a separator?
      end = text.length();                        // No,  so set to last + 1

   // Two lines deleted that stored each and its pointer

   word_count++;                                  // Increment count

   // Find the first character of the next word
   start = text.find_first _not_ of(separators,  end + 1);
   }
```

这里的修改都是删除操作。已删除的 5 行创建了一个固定大小的单词数组和指针，并把指针赋予单词，在这个程序中不需要该行代码。还可以去掉 while 循环中限制程序处理的单词数的条件。在执行完该行代码后，text 中会有一个字符串，word_count 则存储了单词的个数，这样就知道需要分配多少内存了。

创建指向单词的指针数组

现在编写一些新代码。需要一个 word_count 指针数组指向 text 中的单词，创建这个数组只需一行代码：

```
//Allocate an array of pointers to strings in the free store
string** pwords=new string*[word_count];
```

运算符 new 创建了一个 string*类型(即指向 string)的数组，其中包含 word_count 个元素。因为需要在 pwords 中存储 new 返回的地址，所以 pwords 也必须是一个指针。由于 pwords 指向 string*类型的数组，因此其类型也必须是 string**，即指向 string 的指针的指针。

创建单词对象

在指向单词的指针数组可用时，就准备创建表示单词的 string 对象。我们从包含单词所有字符的子字符串中创建 string 对象，这样就需要确定单词开始处的索引位置，计算单词中的字符数。考虑到代码的重复使用，可以用一个循环来执行这个任务，因为它与开始时计算 text 中单词数的代码几乎完全相同：

```
// Create words in the free store and store the addresses in the array
start = text.find_first_not_of(separators);   // Start of first word
end =0;                                       // Index for the end of a word
int index = 0;                                // Pointer array index

while(start != string::npos) {
   end = text.find_first_of(separators, start + 1);
   if(end == string::npos)                    // Found a separator?
     end = text.length();                     // No,  so set to last + 1
   pwords[index++]  = new string(text.substr(start, end - start));
   start = text.find_first_not_of(separators, end+1);   // Find start of next word
}
```

变量 start 存储了每个单词中第一个字符的索引位置，而 end 包含了在 start 处开始的单词后面第一个分隔符的位置。单词的长度由表达式 end-start 给出，可以利用这个长度在自由存储区中为单词创建 string 对象，这是单词计数循环中惟一需要修改的代码行。

用 substr()函数从 text 中提取出单词，用该单词初始化 string 对象。new 运算符返回该对象的内存地址，该地址存储在 pwords[index]中。再使用后缀递增运算符，把 index 递增到下一个自由数组元素的位置上。

排序并输出单词

除了 pwords 指针数组是动态创建的之外，它跟其他数组完全相同。也就是说，排序和输出单词的代码可以跟程序示例 7.5 中的对应代码完全相同。把这些代码复制过来：

```
// Sort the words in ascending sequence by direct insertion
int lowest = 0;                                   // Index of lowest word
for(int j = 0 ; j < word_count - 1 ; j++) {
   lowest = j;                                    // Set lowest

   // Check current against all the following words
   for(int i = j + 1 ;  i < word_count ;  i++)
     if(*pwords[i]  < *pwords[lowest])            // Current is lower?
       lowest =  i;

   if(lowest != j) {          // Then swap pointers…
     string* ptemp = pwords[j];                   // Save current
     pwords[j] = pwords[lowest];                  // Store lower in current
     pwords[lowest] = ptemp;                      // Restore current
   }
}

// Output up to six words to a line in groups starting with the same letter
char ch =  (*pwords[0]) [0];                      // First letter of first word
int words_in_line = 0;                            // Words in a line count
for(int i = 0 ;  i < word_count ;  i++) {
  if(ch != (*pwords[i]) [0]) {                    // New first letter?
    cout << endl;                                 // Start a new line
    ch = (*pwords[i]) [0];                        // Save the new first letter
    words_in_line = 0;                            // Reset words in line count
  }
```

```
    cout << *pwords[i]  <<"   ";
    if(++words_in_line == 6) {                    // Every sixth word
      cout << endl;                               // Start a new line
      words _in_ line = 0;
    }
}
```

释放自由存储区的内存

这次输出并不表示程序的结束。必须使用 delete 运算符释放单词和指针数组所使用的内存：

```
// Delete words from free store
  for(int i = 0 ; i < word_count ; i++)
    delete pwords[i];

  // Now delete the array of pointers
  delete[] pwords;

   return 0;
}
```

在 for 循环中删除包含单词的 string 对象，接着删除指针数组 pwords。注意因为这里删除的是数组，故必须在关键字 delete 之后加上方括号。

如果把上述代码都放在一起，所得的程序就会输出如下结果：

```
Enter a string terminated by #:
Little Willie from his mirror
Licked the mercury right off,
Thinking, in his childish error
It would cure the whooping cough.
At the funeral Willie's mother
Brightly said to Mrs Brown,
"Twas a chilly day for Willie
When the mercury went down."#

At
Brightly  Brown
It
Licked  Little
Mrs
Thinking  Twas
When  Willie  Willie  Willie's
a
childish  chilly  cough  cure
day  down
error
for  from  funeral
his  his
in
mercury  mercury  mirror  mother
off
right
```

```
said
the  the  the  the  to
went  whooping  would
```

7.6.5 转换指针

reinterpret_cast<>()运算符允许强制转换任何指针类型，其限制是如果要强制转换的指针类型声明为 const，就不能把它强制转换为不是 const 的类型。还可以把整数值强制转换为指针类型，把指针类型强制转换为整型值。这不会改变所涉及的值，只会改变解释值的方式。因此，这是一个风险性很高的运算符，应仅在绝对必需时才使用。

reinterpret_cast<>()运算符的一般形式如下：

```
reinterpret_cast<指针类型>(表达式)
```

圆括号中的表达式解释为尖括号中的指针类型。为了演示一般的转换过程，下面假定由于某种奇怪的因素，需要把 float 值解释为 long。这可以使用下面的语句来完成：

```
float value=2.5f;
float* pvalue=&value;
long* pnumber= reinterpret_cast<long*>(pvalue);
```

执行这些语句后，指针 pnumber 就指向 value 中的值(即 2.5)，但其类型为 long。值本身并没有改变，跟以前一样。下面的语句可以把该值输出为 long 类型：

```
std::cout<<std::endl<<*pnumber;
```

这说明浮点数值 2.5 把 1075838976 看作 long 类型。其逆过程——存储 long 值，并把它解释为 float 类型——肯定会生成一个格式不正确的浮点数。

再次强调，reinterpret_cast<>()运算符是很危险的，因为它提供了把一种类型的值随意解释为另一种类型的方式。应避免使用这个运算符，除非这是解决问题所必不可少的，而且很清楚自己在做什么。

7.7 本章小结

本章介绍了一些非常重要的概念，因为在 C++程序中可以广泛使用指针，所以应好好掌握这一内容，本书的其他地方会常常使用指针。

本章的主要内容如下：

- 指针是包含地址的变量。
- 使用地址运算符&可以获取变量的地址。
- 要引用指针指向的值，应使用间接运算符*。它也称为解除引用运算符。
- 可以对存储在指针中的地址加减整数值。其结果就像指针引用一个数组一样，指针会变为整数值所指定的数组元素的个数。
- 运算符 new 会分配自由存储区中的一块内存，返回所分配的内存地址，使它可在程序中使用。

- 运算符 delete 可以释放用运算符 new 分配的内存块。
- reinterpret_cast<>()运算符可以把一种类型的指针转换为另一种类型。

7.8 练习

1. 编写一个程序，声明并初始化一个数组，其中包含前 50 个偶数。使用数组表示法输出该数组中的数字，每一行显示 10 个数字。再使用数组表示法以逆序输出这些数字。

2. 创建一个程序，从键盘上读取数组的大小，对这个数组进行动态分配内存，以存储浮点值。使用指针表示法初始化数组的所有元素，使索引位置为 n 的元素值是 $1.0/(n+1)^2$。使用指针表示法计算元素的总和，对该总和乘以 6，输出该结果的平方根。试着给程序提供更大的数组大小，例如，超过 1000000 元素，结果有什么有趣的地方吗？

3. 修改练习 6.1，使用动态内存来存储学生的信息，使程序可以处理任意数量的学生。

4. 二维数组是数组的数组，而数组可以通过指针动态创建。如果动态创建的数组元素也是指针，数组中的每个元素就都可以存储数组的地址。使用这个概念，创建一个数组，其中包含三个数组指针，每个数组都可以包含六个 int 类型的值。把第一个整数数组的值设置为 1 到 6，下一个数组的元素值设置为第一个数组元素的平方，第三个数组的元素值设置为第一个整数数组元素的立方。输出这三个数组的内容，再释放已分配的内存。

第 8 章　使用函数编程

把程序分解为易于管理的代码块是每种语言编程的基本理念。函数是所有 C++程序中的一个基本组成块。前面利用了标准库中的一些函数，而自己编写的函数只有 main()。本章将介绍如何定义自己的函数。

本章主要内容

- 函数的概念，为什么应把程序分解为函数
- 如何声明和定义函数
- 如何把参数传送给函数，如何返回值
- 按值传送的含义
- 把参数指定为指针如何影响按值传送机制
- 把 const 用作参数类型的限定符，将如何影响函数的操作
- 按引用传送的含义，如何在程序中声明引用
- 如何从函数中返回一个值
- 内联函数的概念
- 在函数中把变量声明为 static 的影响

8.1　程序的分解

前面编写的所有程序都只由一个函数 main()组成。所有的 C++程序都必须有函数 main()，这是程序执行的开始位置。但是，C++允许在程序中包含所需要的许多其他函数，前面的例子就已经使用了标准库中的一些函数。定义和使用自己的函数非常简单。图 8-1 展示了把任意程序分解成几个函数的整体结构。

图 8-1 中函数的执行顺序由箭头上的数字表示。一般情况下，在程序的给定点调用函数时，将执行函数包含的代码。在执行完这些代码之后，就继续执行程序中该函数后面的代码。因为任何函数都可以调用其他函数(这就是 main()的作用)，其他函数还可以再调用别的函数，所以一个函数调用可能会导致执行好几个函数。

一个函数调用另一个函数，另一个函数再调用别的函数，此时，就会有好几个函数同时执行，每个函数都在等待它调用的函数返回。在上面的例子中，由于 main()调用 doThat()，doThat()再调用 calcThat()，因此这三个函数都在同时运行。calcThat()函数在执行的同时，doThat()函数在等待 calcThat()函数返回，而 main()函数在等待 doThat()函数返回。

对于正在执行的函数来说，其代码必须位于内存中的某个地方。前面所说的“一个函数在等待另一个函数返回”，但为了使函数返回，必须跟踪在内存的哪个地方进行了函数调用，函数必须把值返回到什么地方。这些信息都会自动记录并保存在调用堆栈中。调用堆栈包含某个给定时刻所有函数调用的信息，以及传送给每个函数的参数信息。大多数 C++开发系统内置的调试功能通常都提供了在程序运行期间查看调用堆栈的方式。如图 8-1 所示。

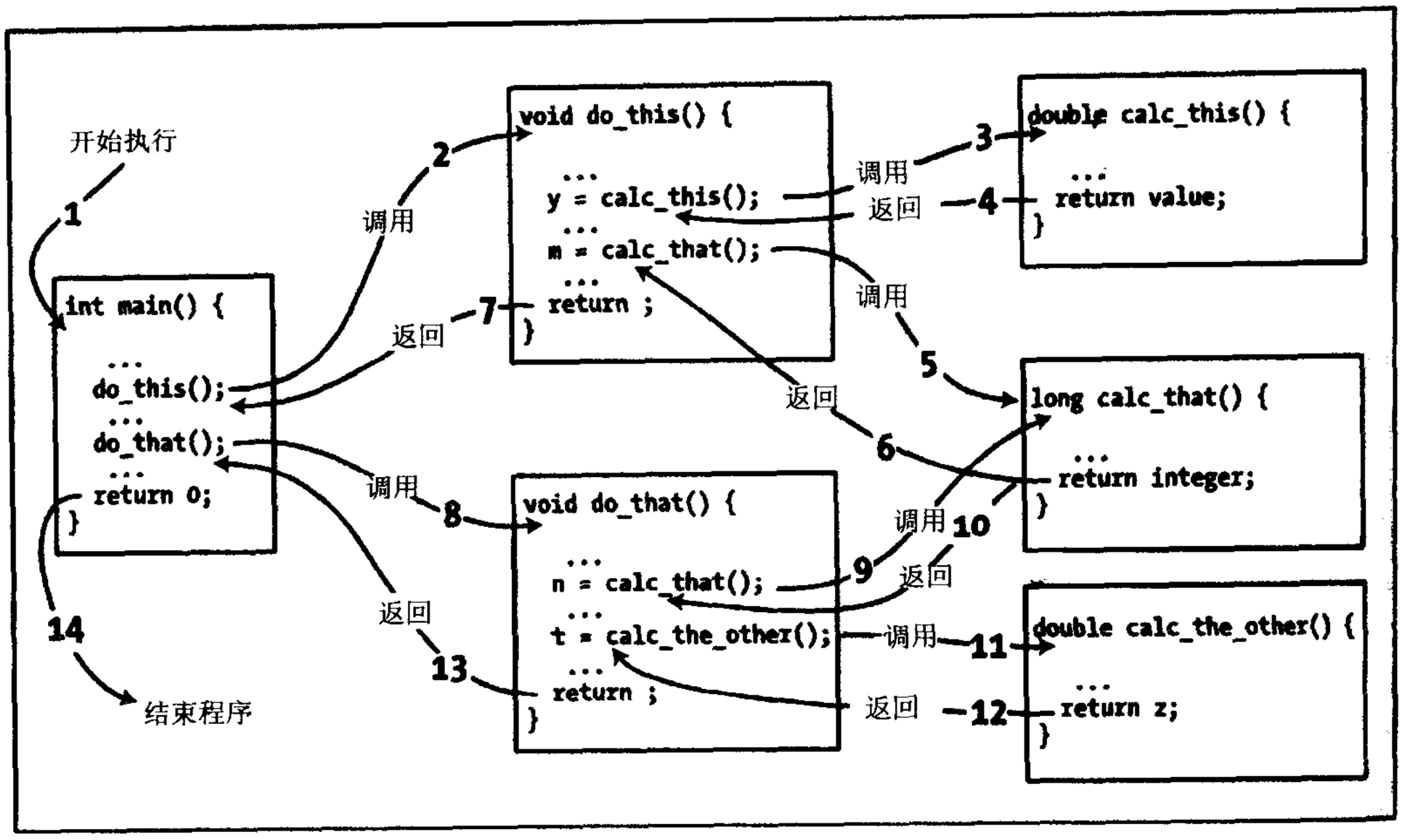


图 8-1　函数调用函数

把程序分解为函数的原因

把程序分解为若干个函数有几个原因。

首先，分解程序会使程序更容易阅读和管理。main()很容易变得非常大，无法管理，下面举一个极端的例子。目前许多应用程序都要运行成千上万行代码，其中一些要运行上百万行代码。想想 main()如果包含 100000 行代码，该如何处理。掌握程序的逻辑根本就不可能，因为在这种程序中，只能跟踪最简单的错误。分解这种规模的程序是绝对有必要的。

其次，函数可以重复使用。例如，假定要编写一个给字符串排序的程序，用于某个特定的程序。这个函数编写好后，还可以用于另一个程序。标准库就是重复使用函数的一个绝佳例子。在开发标准库时，进行了上千小时的编程和测试，这样用户就可以随时访问它提供的功能。在许多不同的程序中使用它们。

第三，把程序分解为几个函数，可以减少运行它所需要的内存量。大多数应用程序都涉及到一些重复使用的计算操作。如果把执行这些计算操作的代码包含在一个可以随时调用的函数中，则计算操作所需要的代码就只编写一次。没有这样的函数，每次需要它时都必须重复编写该代码，编译出来的程序也会比较大。

第四，确定如何实现程序，以解决给定问题的过程肯定需要把问题分解为较小的单元。一般程序的设计过程都会产生自包含的代码单元，把这些设计编写为各个单元，使某个单元仅涉及一个或多个函数是非常有意义的。

8.2 理解函数

前面已编写了相当多的 main()函数，现在用户就应对函数的外观有了很好的认识，这里复习一下基本概念，以确保概念的清晰。下面首先讨论从广义上看函数的工作原理。

函数是有特定目标的自包含代码块。其含义是需要很清楚问题的解决方案是如何分解为函数单元的。在 C++中，没有比函数单元更小的单位了，本书将在第 11 章讨论数据类型的定义时再详细讨论这个问题。这将对设计程序的方法有深远的影响。

下面看看如何构建函数。在讨论过程中，理解下面的例子是很有帮助的，如图 8-2 所示。

8.2.1 定义函数

在函数定义中指定函数的作用，如图 8-2 所示。函数定义中有两个部分：函数头和函数体，函数头是定义中的第一行，在左花括号之前；函数体位于花括号对中。从 main()函数可知，函数体包含调用函数时执行的代码。

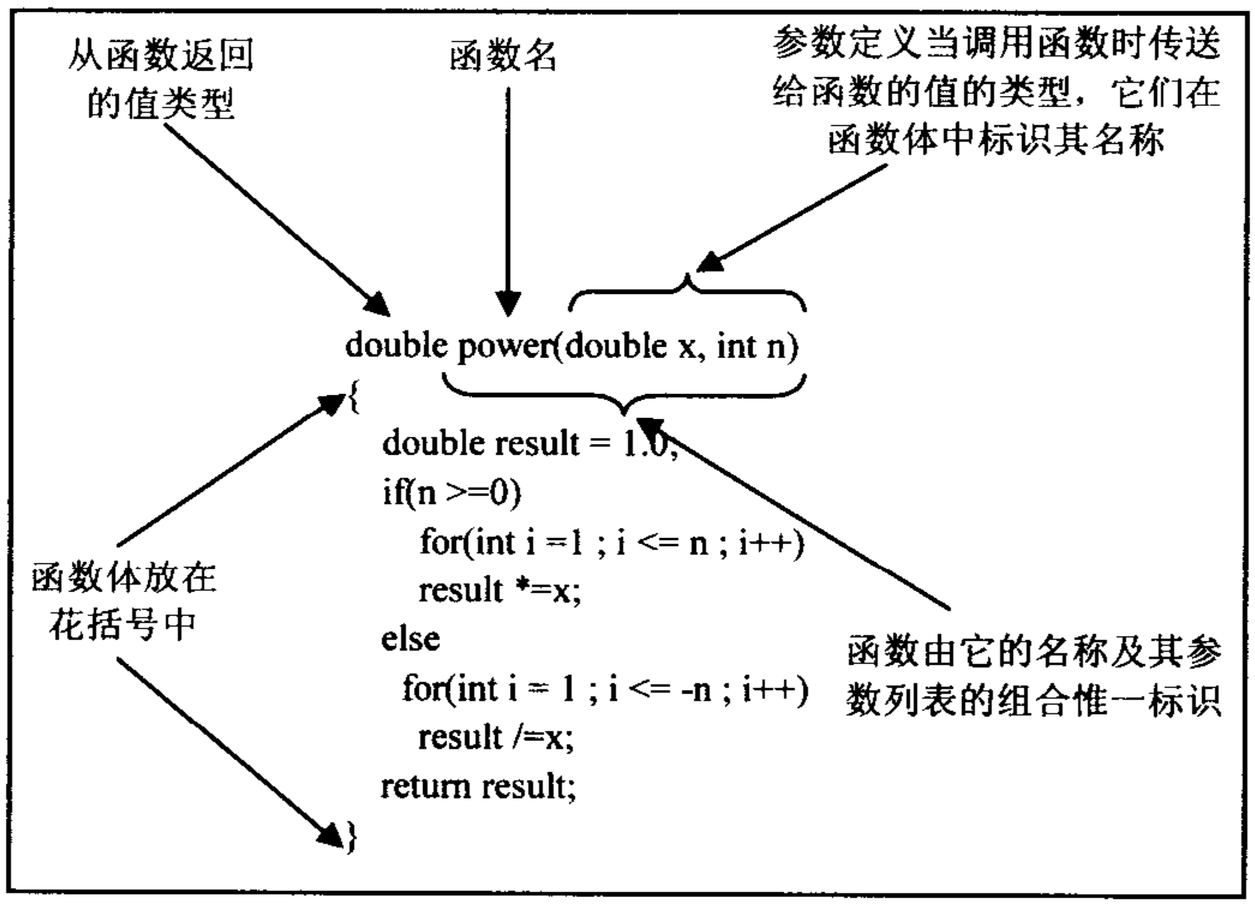


图 8-2 函数定义的例子

1. 函数头

首先看看例子中的函数头。这是函数的第一行：

```
double power(double x, int n)
```

其中包含三部分：返回值的数据类型(在本例中是 double)，函数名 power 和放在圆括号中的函数的参数列表。这里的参数名是 x 和 n，在指定函数功能的语句中将使用这些名称，它们对应于调用函数时所传送的参数。参数 x 是 double 类型，参数 n 是 int 类型，在调用函数时，应传送这些类型的值。

注意：

在函数头的最后不需要加上分号。

函数头的一般形式

函数头的一般形式如下所示：

返回类型　函数名(参数列表)

函数名是在程序中用于调用函数的名称，最好给每个函数指定不同的名称，但在第 2 章的一些标准库函数中，只要每个函数有不同的参数列表，几个不同的函数就可以有相同的名称。这些函数一般实现相同的功能，但参数集合不同。

函数名的命名规则与变量名相同。因此，函数名是一组字母和数字，第一个字符是字母，下划线也算作字母。函数名一般应反映其功能，例如数豆的函数可以称为 CountBeans()，计算给定值的幂的函数称为 power()。

返回类型设置了函数要返回的值的数据类型，它可以是任意合法的数据类型，包括自己创建的数据类型。如果函数没有返回值，返回类型就由 void 关键字指定。

参数列表标识可以在调用函数中给被调用函数传送的内容，并指定了每个参数的类型和名称。参数名在函数体中用于访问调用函数传送进来的数据项。函数也可以没有任何参数，这由一个空参数列表来表示，或用括号中的关键字 void 来表示。没有参数也没有返回值的函数头如下所示：

```
void MyFunction()
```

或者：

```
void MyFunction(void)
```

注意参数的类型不能是 void。

注意：

由于用 void 指定返回类型的函数没有返回值，因此这种函数不能在调用程序的表达式中使用。此函数不等于任何值，测试其值或给它赋值是没有意义的。以这种方式使用这类函数，编译器会产生一个错误消息。

2. 函数体

函数进行调用时，会执行函数体中的语句。在前面的例子中，函数体的第一行声明了一个 double 变量 result，该变量初始化为值 1.0。result 是一个自动变量，它在函数中定义，仅存在于函数体中。也就是说，变量 result 在函数执行完后就不存在了。

计算在两个 for 循环中的一个进行，这取决于参数 n 的值。值 n 是调用函数时给函数传送的第二个参数值。如果 n 大于或等于 0，就执行第一个 for 循环。如果 n 的值等于 0，就不执行循环体，因为循环条件是 false。在这种情况下，result 就等于 1.0。否则，循环控制变量 i 就从 1 递增到 n，变量 result 在每个循环迭代中都与 x 相乘，生成需要的值。如果 n 是负数，就执行第二个 for 循环，在每个循环迭代中将 result 除以 x。

在函数体中声明的变量和所有的函数参数都在函数中定义为局部变量。在一个函数中使用的变量名和参数名可以在另一个函数中用于不同的目的。在函数中声明的变量的作用域由前面介绍过的方式来确定：变量在定义它的块中创建，在包含该变量的块结束时消失。这个规则的惟一例外是声明为 static 的变量，本章后面将详细讨论它。

在深入论述变元、参数和返回值之前，先把函数 power()放在一个完整的程序中。

程序示例 8.1——使用函数

用下面的例子来试验函数 power()：

```
// Program 8.1 Calculating powers
#include <iostream>
#include <iomanip>
using std::cout;
using std::endl;

// Function to calculate x to the power n
double power(double x, int n) {
   double result = 1.0;
   if(n >= 0)
      for(int i = 0 ; i < n ; i++)
         result  *=x;
   else
      for(int i = 0 ; i < -n ; i++)
         result /= x;
   return result;
}

int main() {
   cout << endl;
   // Calculate powers of 8 from -3 to +3
   for(int i = -3 ; i <= 3 ; i++)
      cout << std::setw(10) << power(8.0, i);

   cout << endl;
   reurn 0;
}
```

这个程序的输出如下所示：

```
0.00195313    0.015625    0.125    1    8    64    512
```

例子的说明

main()中的 for 循环完成了所有的操作：

```
for(int i=-3; i<=3; i++)
   cout<<std::setw(10)<<power(8.0,i);
```

power()函数调用了 7 次。每次调用时，第一个参数都是 8.0，而第二个参数是 i 的值，从 -3 到+3。输出为 7 个值，分别对应于 8^{-3}、8^{-2}、8^{-1}、8^{0}、8^{1}、8^{2} 和 8^{3}。

3. 参数和变元

在调用函数值，要通过指定的变元把信息传送给函数。在调用时，变元放在函数名后面的圆括号中，如上面例子中的代码所示：

```
cout<< std::setw(10)<<power(8.0,i);
```

在调用函数时，指定的变元将代替在函数定义中使用的参数。函数中的代码在执行时，就使用变元值来初始化对应的参数。函数调用中的变元和函数定义中的参数之间的关系如图 8-3 所示。

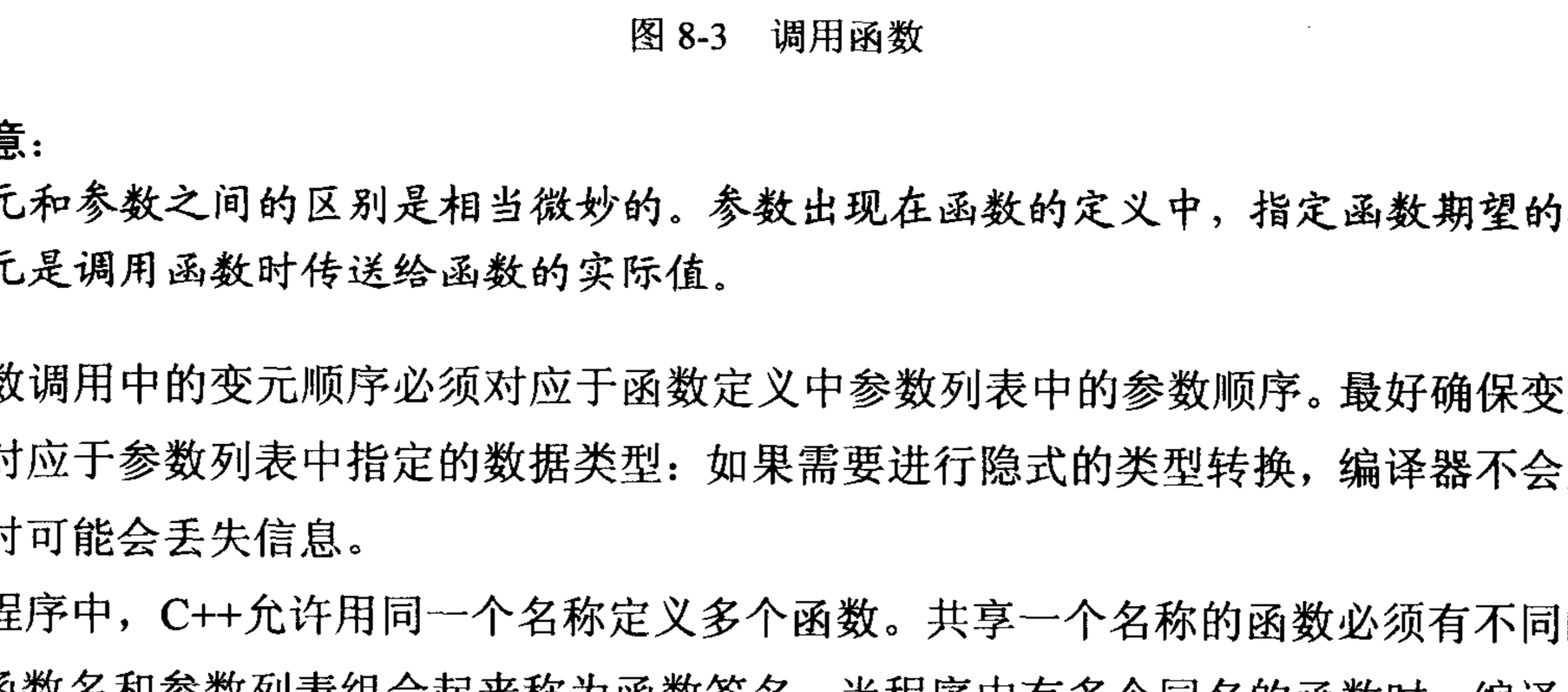


图 8-3　调用函数

注意：

变元和参数之间的区别是相当微妙的。参数出现在函数的定义中，指定函数期望的数据类型。变元是调用函数时传送给函数的实际值。

函数调用中的变元顺序必须对应于函数定义中参数列表中的参数顺序。最好确保变元的数据类型对应于参数列表中指定的数据类型：如果需要进行隐式的类型转换，编译器不会发出警告，此时可能会丢失信息。

在程序中，C++允许用同一个名称定义多个函数。共享一个名称的函数必须有不同的参数列表。函数名和参数列表组合起来称为函数签名。当程序中有多个同名的函数时，编译器就使用函数签名来确定应调用哪个函数。这是确保变元和参数类型对应的另一个原因。

4. 返回值

一般情况下，在调用返回类型不是 void 的函数时，该函数必须返回一个值，其类型已在函数头中指定。返回值在函数体中计算，在执行完函数后返回。

因此，在调用函数 power()的表达式中，power()实际上是一个 double 类型的值。但是，不一定要在表达式中调用返回一个值的函数，只需调用函数本身，如下面的语句所示：

```
power(10.5, 2);
```

这个语句会执行函数，从函数返回的值会被舍弃。以这种方式执行 power()没有意义，但并不是所有的函数都是这样。例如，有一个复制文件的函数，其返回值提供了该操作的其他信息(复制成功或失败)，有时可以选择舍弃该返回值。当然，这也是调用返回类型为 void 的函数的方式。这种函数不返回任何值，所以不能用于表达式。

函数只能返回一个值，这看起来约束性很强，但事实并非如此。返回的一个值可以是指向所需内容的指针，例如数据数组，甚或指针数组。函数总是返回在函数头中定义的给定类型的一个值。

return 语句

例子中的 return 语句把 result 的值返回到调用函数的地方。但是，在函数执行完后，result 不是已经不存在了吗？它是如何返回的？答案是系统会自动复制返回值，该副本在函数的调用位置是可用的。

return 语句的一般形式如下所示：

```
return expression;
```

其中，*expression* 必须等于在函数头中为返回值指定的类型的值。该表达式可以是任何表达式，只要其值是指定的类型即可。它可以包含函数调用，甚至可以包含同一函数的调用，详见第 9 章。

如果返回类型指定为 void，return 语句中就不应有表达式。此时该语句必须写成：

```
return;
```

如果在执行过程中到达函数体的右花括号，就等于执行一个没有表达式的 return 语句。当然，在返回类型不是 void 的函数中，这会产生一个错误，函数也不会编译。

8.2.2 函数的声明

前一个例子，即程序示例 8.1 工作得很好。下面重新安排代码，把函数 main()放在函数 power()定义的前面。程序文件中的代码如下所示：

```
// Program 8.2 Calculating powers - rearranged
#include <iostream>
#include <iomanip>
using std::cout;
using std::endl;

int main() {
   cout << endl;

   // Calculate powers of 8 from -3 to +3
   for(int i = -3 ; i <= 3 ; i++)
      cout << setw(10) << power(8.0, i);

   cout << endl;
   return 0;
}
```

```
// Function to calculate x to the power n
double power(double x, int n) {
   double result = 1.0;
   if(n >=0)
      for(int i=0; i<n; i++)
         result *= x;
   else
      for(int i = 0 ; i < -n ; i++)
         result /= x;
   return result;
}
```

编译程序的这个版本是不会成功的。编译器会生成一个错误，因为在执行函数 main()时，函数 power()还没有定义。当然，可以简单地放弃这个版本，转而使用程序示例 8.1，但这并没有解决问题。这里有两个重要的问题。

首先，如后面所述，程序可以包含几个源文件。在这种情况下，被调用的函数定义可能没有放在进行调用的函数前面。例如，它可能包含在一个完全独立的文件中。

其次，假定函数 A()调用了函数 B()，B()又调用了函数 C()。如果先定义 A()，它就不会编译，因为它调用了 B()，如果先定义 B()，也会出现同样的问题，因为 B()调用了 C()。

当然，这些问题都有解决方法。可以在使用之前声明函数，或通过函数原型来定义函数。

1. 函数原型

函数原型是一个描述函数的语句，它可以让编译器编译对该函数的调用。函数原型声明了函数名、函数的返回类型及其参数的类型。函数原型常常称为函数声明，它类似于变量声明，因为在程序文件中，除非把声明放在调用之前，否则函数是不能调用的。函数的定义也是一个声明，这就是在程序示例 8.1 中，power()不需要函数原型的原因。

可以把 power()函数的函数原型写为：

```
double power(double x, int n);
```

如果把函数原型放在包含该函数调用的程序文件的开头，则无论函数的定义在什么地方，编译器都能编译代码。要编译程序示例 8.2，可以在 main()定义的前面插入函数 power()的原型：

```
// Program 8.2 Calculating powers - rearranged
#include <iostream>
#include <iomanip>
using std::cout;
using std::endl;

double power(double x, int n);      // Prototype for power function

int main() {
  // main () function as before
}

double power (double x, int n) {
  // power () function as before
```

```
}
```

这里的函数原型与函数头相同，只是最后加上了一个分号。函数原型总是由一个分号来结束，但一般情况下它不必与函数头相同。定义中的参数可以使用不同的名称(但不能使用不同的类型)。例如：

```
double power(double value, int exponent);
```

这个函数原型也能使用。这里选择什么名称并不重要，但在函数原型中使用解释性比较强的名称，会对程序的理解有很大的帮助，但这种名称在函数定义中使用会比较啰嗦。

因为编译器只需要知道每个参数的类型，所以可以在原型中省略参数名，如下所示：

```
double power(double, int);
```

像这样编写函数原型没有什么好处，它提供的信息也比带有参数名的函数原型少得多。如果函数的参数都是相同的类型，则像这样的函数原型根本没有提供每个参数的信息。在函数原型中，最好总是包含参数名。

2. 使用函数原型

应习惯于在程序文件的开头为在程序中使用的每个函数编写函数原型。当然 main()例外，它根本不需要原型。这样，就不会因函数的位置不正确而产生编译错误了。另外，这样还允许其他程序员大致了解代码的功能。

前面多次使用了库函数，这些函数的原型在什么地方？实际上，它们在已经包含的标准头文件中。头文件的一个主要用途是把相关函数组的函数原型组合在一起。详见第 10 章。

8.3 给函数传送参数

理解如何给函数传送变元是非常重要的，因为这会影响到如何编写函数，最终将影响函数如何运行。要避免的陷阱很多，下面先解释其机制。

如前所述，调用函数时指定的变元通常应对应于函数定义中参数的类型和顺序。虽然变元的顺序不能改变，但变元的类型有时可以不一致。如果在函数调用中指定的变元类型与函数定义中的参数类型不对应，编译器就会把变元的类型转换为参数类型。自动转换的规则与赋值语句中控制自动转换的规则相同，详见第 3 章。如果不能进行自动转换，编译器就会生成一个错误消息。

在 C++中，通常使用两种机制给函数传送变元。一种称为按值传送方法，另一种称为按引用传送方法。下面先详细讨论按值传送机制，之后论述按引用传送机制。

8.3.1 按值传送机制

利用这种机制给函数传送数据时，指定为变元的变量或常量实际上根本不会传送给函数。与返回值一样，系统会创建变元的副本，把这些副本用作要传送的值。下面用 power()函数来说明，如图 8-4 所示。

每次调用 power()函数时，编译器都会把指定的变元副本存储在内存的一个临时位置，即

本章前面提到的调用堆栈中。在执行过程中，组成函数体的代码中对函数参数的所有引用都映射到变元的临时副本上。执行完函数后，就废弃变元的副本。

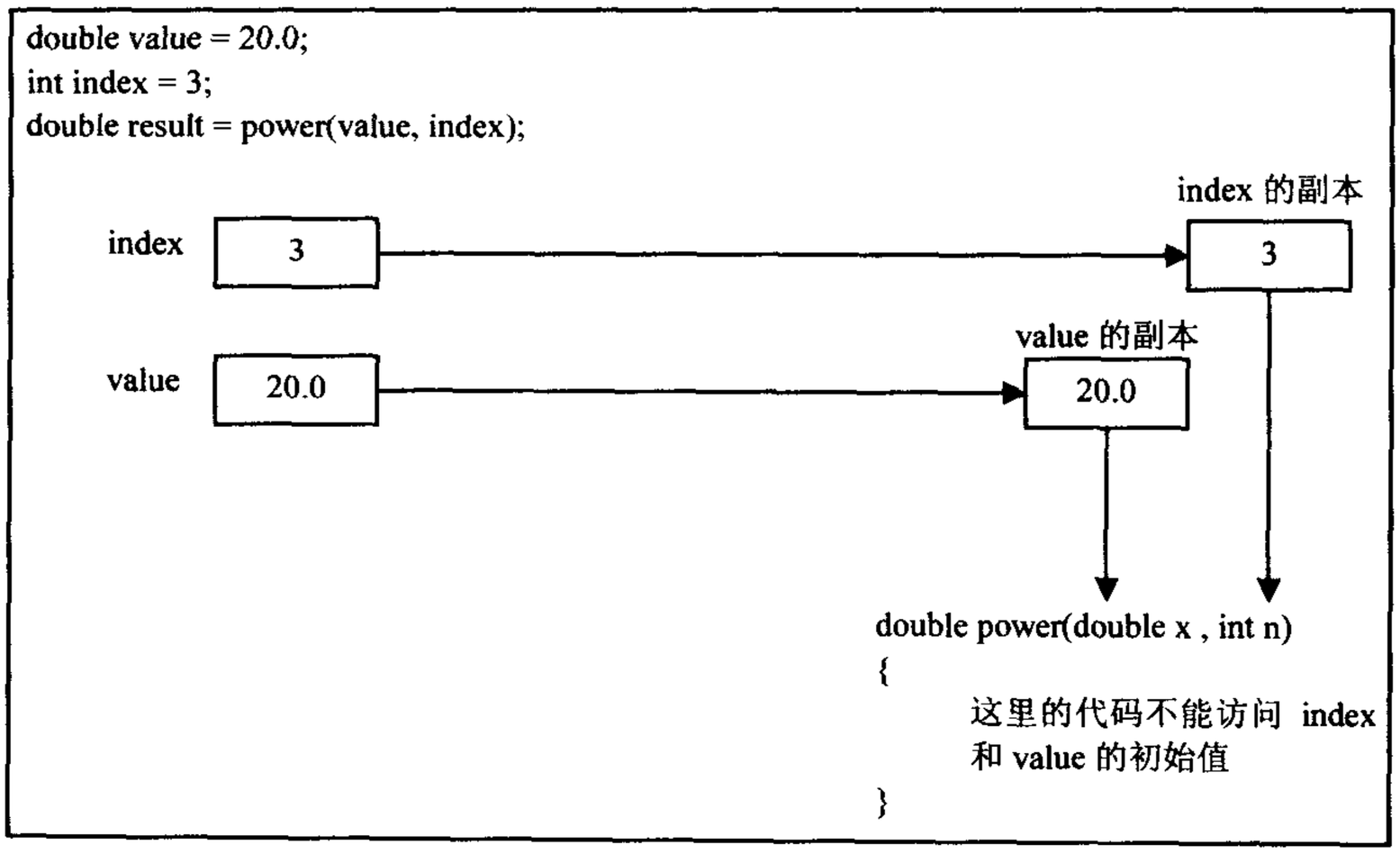


图 8-4　按值传送机制

下面用一个简单的例子来说明。

程序示例 8.3——给函数传送变元

编写一个函数，它试图修改它的一个变元，当然肯定会失败。

```
// Program 8.3 Failing to modify the original value of a function argument
#include <iostream>
#include <iomanip>
using std::cout;
using std::endl;

double change_it(double it);                    // Function prototype

int main () {
  double it = 5.0;
  double result = change_it(it);

  cout << "After function execution, it = " << it       << endl
       << "Result returned is "                 << result   << endl;
  return 0;
}

// Function to attempt to modify an argument and return it
double change_it (double it) {
  it += 10.0;                                   // This modifies the copy of the original
  cout << endl
       << "Within function, it = " << it << endl;
  return it;
}
```

这个例子的结果如下所示：

```
Within function, it = 15
After function execution, it = 5
Result returned is 15
```

例子的说明

从输出中可以看出，在函数 change_it()中给变量 it 加上 10，对 main()中的变量 it 没有任何影响。函数 change_it()中的变量 it 是该函数的局部变量，在调用该函数时，会引用所传送的变元值的副本。如图 8-5 所示。

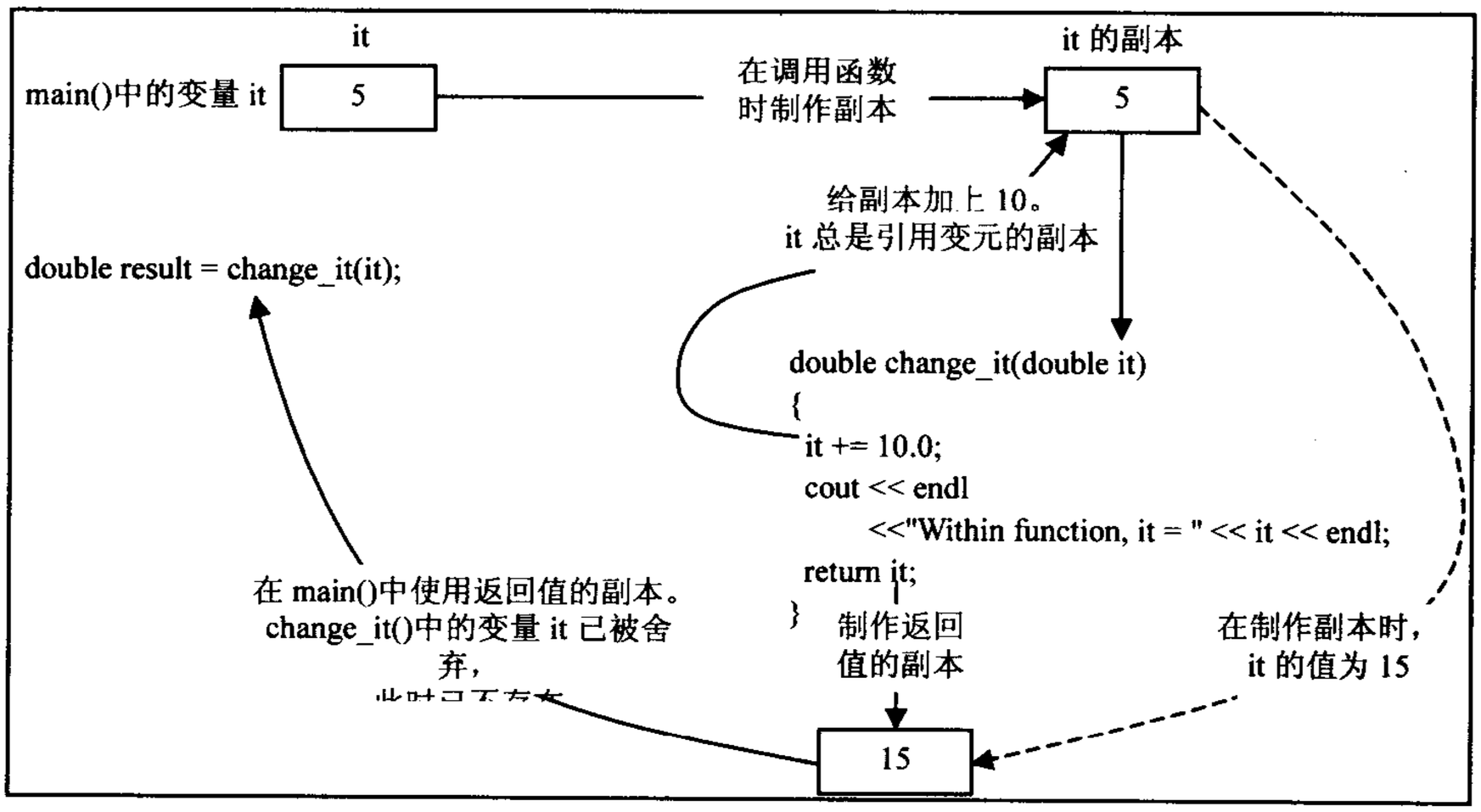


图 8-5　修改按值传送的参数值

当然，在返回 change_it()的局部变量 it 的值时，会制作其当前值的副本，把该副本返回给调用程序。

这种按值传送机制是给函数传送变元的默认机制，它为调用程序提供了许多安全，防止函数修改调用函数拥有的变量。但如果要在调用函数中修改值，有什么方法？使用指针就是一个好方法。

1. 给函数传送指针

在把指针用作变元时，按值传送机制会像以前那样运行。但是，指针包含另一个变量的地址，指针的副本会指向内存中的同一个地址。

把 change_it()函数的定义改为接受类型为 double*的变元。在 main()中，调用函数时可以把变量 it 的地址传送给函数。当然，还必须修改 change_it()函数体中的代码，解除所传送的指针的引用。其工作原理如图 8-6 所示。

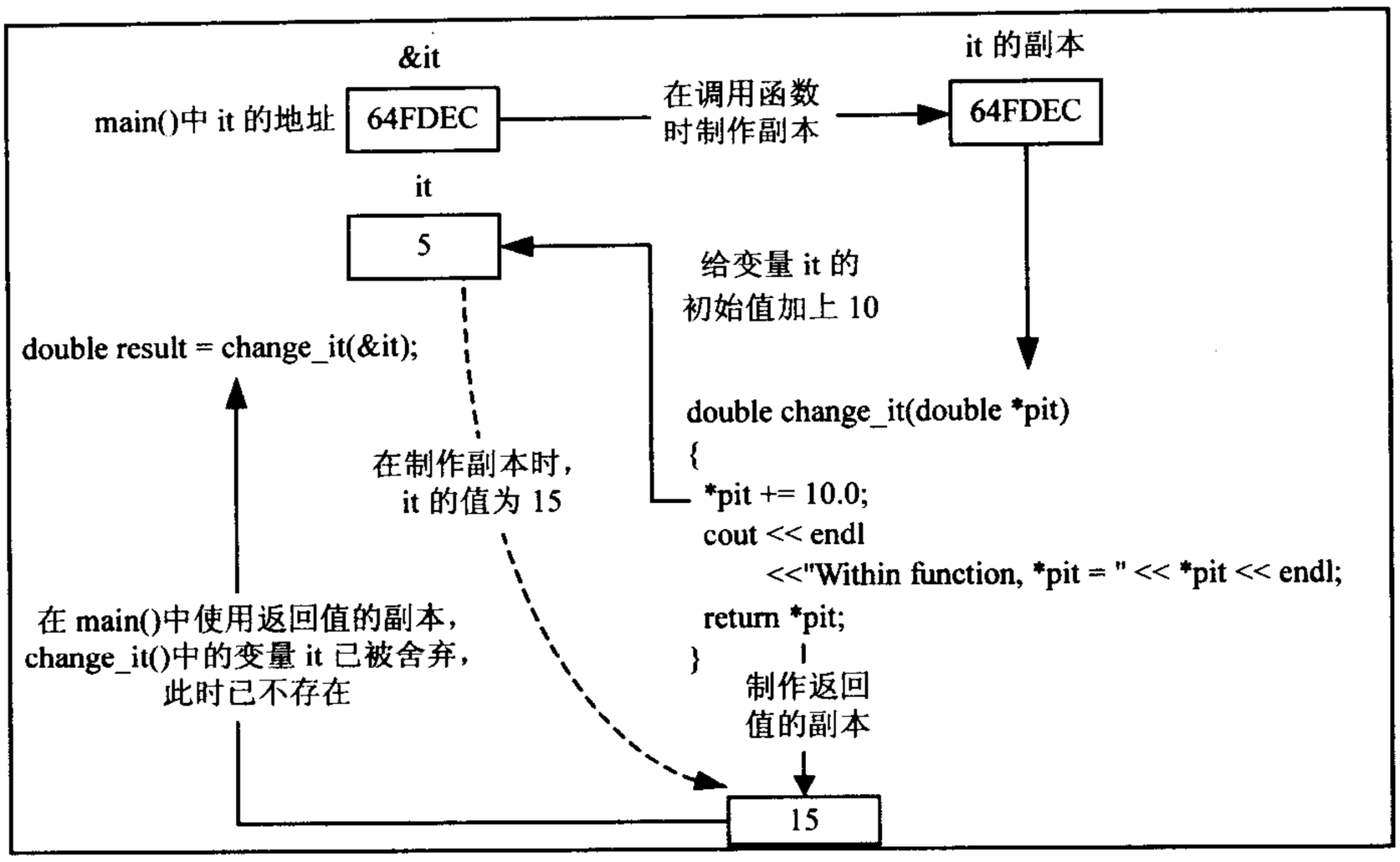


图 8-6　给函数传送指针

由于函数 change_it()现在拥有 it 在 main()中存储的初始地址，因此它可以直接修改变量。下面看看具体的操作。

程序示例 8.4——传送指针

把上一个例子改为使用指针，看看其结果：

```
// Program 8.4 Modifying the original value of a function argument
#include <iostream>
using std::cout;
using std::endl;

double change_it(double* pointer_to_it);   // Function prototype

int main() {
  double it = 5.0;
  double result = change_it(&it);          // Now we pass the address

  cout << "After function execution,  it =" << it     << endl
       << "Result returned is  "          << result << endl;
  return 0;
}

// Function to modify an argument and return it
double change_it(double* pit) {
  *pit += 10.0;                            // This modifies the original it
  cout << endl
       << "Within function,  *pit ="<< *pit << endl;
  return *pit;
}
```

程序的这个版本输出的结果如下：

```
Within function, *pit = 15
After function execution, it = 15
Result returned is 15
```

例子的说明

在 change_it()的定义中已修改了参数类型，下面修改函数原型：

```
double change_it(double* pointer_to_it);      //Function prototype
```

还可以使用与函数定义中使用的参数名不同的参数名。

在 main()中，声明并初始化变量 it 后，就在调用 change_it()函数时，给它传送变量 it 的地址：

```
double result=change_it(&it);                  //Now we pass the address
```

不需要创建一个指针变量来存储 it 的地址。只需要给函数传送的地址，可以在函数调用中使用地址运算符。

change_it()函数的新版本使用解除引用运算符，给存储在 it 变量中的值加上 10，it 变量在 main()中定义：

```
*pit +=10.0;                                   //This modifies the original it
```

函数用下面的语句返回修改后的值：

```
return *pit;
```

仍旧要制作返回值的副本，系统总是会自动制作函数返回值的副本。

change_it()函数的这个版本仅说明了指针参数如何修改调用函数中的变量，但这不是编写函数的模板。由于直接修改了 it 的值，返回它的值就有点多余。

2. 给函数传送数组

因为数组名可以用作地址，所以也可以把数组名作为变元传送给函数。在这种情况下，会复制数组的地址，并把它传送给被调用的函数，这样做有许多优点。

首先，传送数组的地址要比按值传送数组更高效，如果按值传送数组的所有元素，大数组的传送会很费时间，因为需要复制每个元素。实际上，数组不能作为一个变元按值传送每个元素，因为每个参数都表示一个数据项。

其次，也是更重要的，被调用的函数不处理原数组地址，而是处理该地址的副本，这样就可以把参数作为指针来看待，包括修改它包含的地址。也就是说，在函数体中可以给数组参数使用指针表示法。下面介绍最简单的情况：使用数组表示法处理数组参数。

程序示例 8.5——把数组作为函数的参数传送

下面演示数组参数的用法：编写一个函数，计算多个数的平均值，这些数放在传送给函数的数组中。

```
//program 8.5 Passing an array
#include <iostream>
using std::cout;
using std::endl;
```

```
double average(double array[], int count);          //Function prototype

int main() {
  double values[]={1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0, 10.0};

  cout << endl
       << " Average= "
       << average(values, (sizeof values)/(sizeof values[0]))
       << endl;
  return 0;
}

//Function to compute an average
double average(double array[], int count) {
  double sum=0.0;                    //Accumulate total in here
  for(int i=0; i<count; i++)
     sum+= array[i];                    //Sum array elements
  return sum/count;                  //Return average
}
```

这个例子的结果非常简短：

```
Average=5.5
```

例子的说明

函数 average()可用于处理任意长度的数组。从函数原型中可以看出，它接受两个参数：数组和数组中元素的个数：

```
double average(double array[], int count);          //Function prototype
```

第一个参数指定为 double 类型的数组。不能在方括号中指定数组的大小，即使指定也没有效果。这是因为数组第一维的大小不是其类型的一部分(在考虑多维数组的指针时，也会有类似的问题)。实际上，可以把 double 类型的任何一维数组作为参数传送给函数，这取决于调用程序提供的 count 参数值，该值表示数组中的元素个数。

在 average()的函数体中，计算过程如下所示：

```
double average(double array[], int count) {
  double sum=0.0;                    //Accumulate total in here
  for(int i=0; i<count; i++)
     sum+= array[i];                    //Sum array elements
  return sum/count;                  //Return average
}
```

如果直接在 main()中进行计算，所编写出的代码与此相同。函数无法检查数组是否能容纳 count 指定的元素个数，如果 count 的值大于数组的长度，函数可以访问数组外部的内存地址。程序员应确保不会发生这种情况。

main()在输出语句中调用该函数：

```
cout << endl
     << " Average= "
```

```
        << average(values, (sizeof values)/(sizeof values[0]))
        << endl;
```

传送给 average()函数的第一个参数是数组名 values，第二个参数是一个表达式，其值等于数组的元素个数。

在这个例子中，传送给 average()函数的数组元素使用正常的数组表示法来访问。也可以把传送给函数的数组看作指针，使用指针表示法进行计算。下面就试一试这种表示法。

程序示例 8.6——在传送数组时使用指针表示法

修改程序示例 8.5 中的函数，即使在使用数组时也利用指针表示法。这里要修改函数原型和函数头，说明给第一个参数使用了指针表示法(而不是数组表示法)，但这不是绝对必须的。可以在函数体中使用指针表示法，但第一个参数的类型仍指定为数组。修改后的程序如下所示：

```
//Program 8.6 Handling an array parameter as a pointer
#include <iostream>
using std::cout;
using std::endl;

double average(double* array, int count);          //Function prototype

int main() {
  double values[]={1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0, 10.0};

  cout << endl
       << " Average= "
       << average(values, (sizeof values)/(sizeof values[0]))
       << endl;

  return 0;
}

//Function to compute an average
double average(double* array, int count) {
  double sum=0.0;                          //Accumulate total in here
  for(int i=0; i<count; i++)
    sum+= *array++;                        //Sum array elements
  return sum/count;                        //Return average
}
```

这个例子的输出结果与上一个例子完全相同。

例子的说明

可以看出，程序只需作很少的修改，就可以把数组用作指针。函数原型和函数头都已修改，但实际上，这两处修改都不是绝对必须的(用程序示例 8.5 中的对应代码代替这里的函数原型和函数头，但函数体用指针来编写。此时该程序的运行情况也是正常的)。

在这个版本的程序中，最有趣的是 for 循环语句：

```
sum+= *array++;                    //Sum array elements
```

这里违背了一个规则：不能修改指定为数组名的地址。本程序给指针加上了 array 中存储

的地址。当然，这并不是真的违反该规则。

按值传送机制复制了原数组地址，并把副本传送给函数。我们修改的是副本，原数组地址并没有修改。一般情况下，只要给函数传送一维数组，就可以把所传送的值作为指针，以任何方式修改地址。

常量指针参数

只需在参数类型的指定过程中使用 const，就可以确保函数不会在无意中修改传送给它的数组元素。例如，如果要阻止 average()函数修改第一个参数中的元素，就可以把函数改写为：

```
double average(const double* array, int count) {
  double sum=0.0;                    //Accumulate total in here
  for(int i=0; i<count; i++)
    sum+= *array++;                      //Sum array elements
  return sum/count;                  //Return average
}
```

现在编译器会验证数组的元素没有在函数体中修改。当然，还需要修改函数原型，以反映第一个参数的新类型。记住 const 类型与非常量类型大不相同。

在把指针参数声明为 const 时，就是告诉编译器“在调用这个函数时，指针参数指向的内容应看作一个常量”。这会有两个结果。编译器会检查函数体中的代码，确保该声明是正确的，而且函数体的代码不会试图修改指针所指向的值。它还允许用指向一个常量的参数调用函数。

注意：

把一般的参数类型如 int 声明为 const 是没有意义的。因为按值传送机制会在调用函数时复制参数，所以不能在函数体中修改参数原来的值。

把多维数组传送给函数

从前面介绍的内容可以推断出，给函数传送多维数组也是很简单的。例如，把一个二维数组声明为：

```
double beans[2][4];
```

则编写一个假想函数 yield()的原型，如下所示：

```
double yield(double beans[2][4]);
```

这里显式声明了两个维的大小，但实际上无法检查数组第一维的大小是 2。对于数组，由于第一维的大小不是类型定义的一部分，因此数组 beans 的类型实际上是 double[][4]。第二维的大小指定为 4 的任意二维数组都可以传送给函数 yield()。

编译器如何知道函数 yield()的第一个参数是一个数组，而不是一个数组元素？答案很简单：在函数定义或原型中，不能把数组的一个元素指定为参数，但可以在调用函数时把数组的一个元素作为参数传送。对于把数组的一个元素作为参数的函数，参数本身就是该元素的类型，而不考虑整个数组。

在把多维数组定义为参数时，应省略第一维的大小(除非要表示函数仅处理固定大小的数组)。当然，函数需要某种方式来确定第一维的大小，为此，可以采用本章前面对一维数组使用的方法，即添加第二个参数来指定数组第一维的大小：

```
double yield(double beans[][4], int index);
```

如果在函数中使用 sizeof()来确定数组的大小，就需要这个额外的参数。在函数体中，对数组参数名使用 sizeof()，就会返回数组名相同的数组的大小。

下面用一个具体的例子给函数传送一个二维数组。

程序示例 8.7——传送多维数组

下面在函数 yield()中累加传送给它的数组元素。程序和函数的代码如下所示：

```
// Program 8.7 Passing a two-dimensional array to a function
#include <iostream>
using std::cout;
using std::endl;

double yield(double values[] [4], int n);

int main ( ) {
   double beans[3] [4] = {
                           { 1.0,   2.0,   3.0,   4.0},
                           { 5.0,   6.0,   7.0,   8.0},
                           { 9.0,  10.0,  11.0,  12.0}
                         };
   cout << endl
        << "Yield = " << yield(beans, sizeof  beans/sizeof  beans[0] )
        << endl;
   return 0;
}

// Function to compute total yield
double yield(double array[] [4], int count) {
   double sum = 0.0;
   for(int i = 0; i < count ; i++)        // Loop through number of rows
     for(int j=0; j < 4 ; j++)            // Loop through elements in a row
         sum += array[i] [j];
   return sum;
}
```

这个程序的输出结果如下所示：

```
Yield=78
```

例子的说明

函数 yield()的第一个参数定义为一个数组，它有任意多行，但每行有 4 个元素。在调用该函数时，第一个参数是数组 beans，它有 3 行。第二个参数是数组的总长度(字节数)除以第一行的长度所得的结果。它等于数组的行数。

函数中的计算放在一个嵌套的 for 循环中，内层的循环累加了单独一行中的元素，而外层循环则累加了各个行。

在把多维数组作为参数的函数中，并不适合使用指针表示法。在上面的例子中，传送数组时，会传送指向包含 4 个元素的数组(只有一行)的指针。在函数中，这并不是一个简单的指针操作，修改嵌套 for 循环中的语句：

```
sum += *(*(array+i)+j);
```

而该计算使用数组表示法会清楚得多。

8.3.2 按引用传送机制

引用只是另一个变量的别名，引用类型的变量存储对其他变量的引用。在把函数参数指定为引用类型时，函数就要使用按引用传送机制来传送参数。在调用函数时，对应于引用参数的变元不会复制，因为参数名就是调用程序中该变元值的别名。只要在函数体中使用参数名，它就会直接访问调用函数中的变元值。

要指定引用类型，只需在类型名的后面加上&。例如，要把类型指定为引用 int，就可以把该类型写为 int&。图 8-7 演示了用引用参数调用函数的过程。

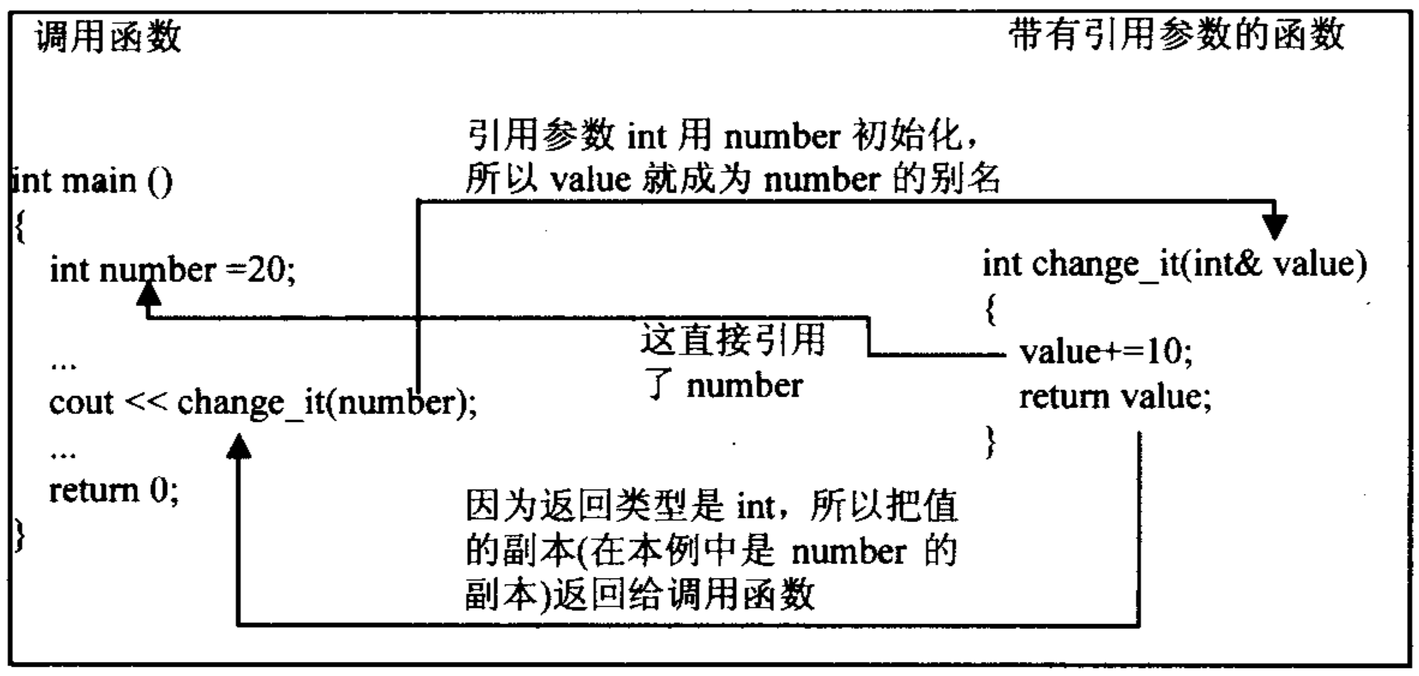


图 8-7 使用引用参数

调用包含引用参数的函数与调用包含按值传送的一般参数的函数没有区别。只要调用了 change_it()函数，引用参数 value 就用指定的变元初始化，在执行 change_it()函数时，value 就成为作为参数传送的变量 number 的另一个名称。如果以后用另一个变元再次调用该函数，value 就成为该变元的别名。

把函数的参数指定为引用有两个主要作用。首先，参数是不复制的，函数将直接访问调用程序中的变元。其次，不制作副本，会使函数调用执行得更快，特别是在变元是大而复杂的对象时，就更是如此，在类对象传送给函数时，这是非常重要的。下面详细介绍如何使用引用参数，特别是要与使用指针进行比较。

1. 引用是有风险的

在为函数指定引用参数时，常常只注意到它的高效，而忘记了这种操作的负面影响。使用引用参数可以在被调用函数中修改调用函数中最初的变元，但与使用指针的语法相比，调用包含引用变元的函数的语法会使其效果不明显。错误地假定变元是按值传送的，因此不会修改参数值，从而引起错误的可能性非常大，而在函数中修改原参数是比较安全的。

图 8-8 演示了调用使用指针参数的函数和使用引用参数的函数的区别。

带有指针参数的函数	带有引用参数的函数

int change_it(int* value)
{
*value+=10;
return *value;
}
调用这个函数时，变元必须是一个地址。在这种情况下，下面的函数可以修改变元的值
int number=20;
cout<< change_it(&number);
	int change_it(int& value)
	{
	value +=10;
	return value;
	}
	值以引用的形式传送，函数调用并没有说明参数的值可以由下面的函数改变
	int number=20;
	cout<< change_it(number);

图 8-8　指针参数和引用参数

在像 change_it()这样的函数中，要修改调用函数中的变元，可以把参数声明为指针或引用。如果把要修改的参数指定为指针，则在调用函数时所传送的变元就必须是一个地址，以清晰地说明在调用程序中，有可能改变参数的值。因为对应于引用参数的参数只是原参数的名称，所以这种修改的可能性并不明显。

如何利用给函数传送引用的无系统开销特性，同时不损失变元的安全性？答案是使用 const 引用，下面的例子探讨一下其效果。

程序示例 8.8——引用参数

下面的简单程序使用了一个函数 larger()，该函数返回两个值中的较大者。我们将以各种方式使用该函数，探讨引用参数如何影响函数的操作方式。下面是其第一个版本：

```
//Program 8.8 Using reference parameters
#include <iostream>
using std::cout;
using std::endl;

int larger(int& m, int& n);

int main() {
  int value1=10;
  int value2=20;
  cout<<endl<<larger(value1,value2)<<endl;

  return 0;
}

//Function to the larger of two integers
int larger(int& m, int& n) {
  return m>n ? m : n;            //Return the larger value
}
```

这个程序的运行结果是 20。

例子的说明

因为函数 larger()带有引用参数，所以在 main()中用如下语句调用它时，操作的是参数 value1

和 value2 的原始值：

```
cout<<endl<<larger(value1,value2)<<endl;
```

如果对此有怀疑，可以在 larger()的函数体中添加一个语句，以修改第二个参数的值：

```
int larger(int& m, int& n) {
  n=30;
  return m>n ? m : n;              //Return the larger value
}
```

如果在 main()中调用 larger()之后输出 value2 的值，就会看到它的值已改变。

假定要比较 value1 和常量值 15，并返回其中的较大者，就可以在 main()中添加如下语句：

```
int main() {
  int value1=10;
  int value2=20;
  cout<<endl<<larger(value1,value2)<<endl;
  cout<<endl<<larger(value1,15)<<endl;        //Won't compile!
return 0;
}
```

添加了这个语句后，程序就不再编译。编译器不允许创建对常量 15 的引用，因为它知道函数会直接访问它，并可能修改它。从这里可以推论出一个重要结论：

当函数的参数指定为非常量引用时，不能把常量作为参数传送给函数。

2. 使用常量引用

如果修改 larger()的函数定义，使其参数为常量引用，就可以解决这个难题。函数原型改为：

```
int larger(const int& m, const int& n);
```

现在函数可以处理变量和常量了。只要不打算修改传送过来的参数，就总是可以把参数定义为 const。可以直接访问从调用函数传送过来的参数(不需要复制)，编译器会检查是否包含修改它们的代码，函数是否可以处理变量和常量值。使用 const 引用参数，就可以获得引用参数的高性能和高效率，以及按值传送方法的安全性。

3. 引用和指针

在大多数情况下，使用引用参数比使用指针更好。只要可能，就应把引用参数声明为 const，因为这会为调用程序提供参数的安全性。当然，如果需要在函数体中修改引用参数的值，就不能把它声明为 const，但应考虑在这种情况下，指针是否是更好的参数类型。对调用程序来说，指针参数总是可以修改的。

指针和引用的一个重要区别是，指针可以为空，而引用总是要引用某个数据项—— 当然，只要它不是空指针的一个别名。如果允许参数为空，惟一的选择就是指针参数。指针参数可以为空，所以必须在解除对指针的引用之前测试它。如果试图解除对空指针的引用，程序就会崩溃。

第 13 章将开始介绍引用参数在类环境中提供的其他重要功能，在一些情况下，指针参数还可以得到使用引用参数不能得到的结果。

4. 声明引用

引用不会显示为函数的参数。引用可以以另一个变量的别名独立存在。假定有一个变量声明为：

```
long number = 0;
```

使用下面的声明语句，可以为该变量声明一个引用：

```
long& rnumber=number;                    //Declare a reference to variable number
```

类型 long 后面的宏&表示声明一个引用，等号后面的初始化值指定为变量名 number。因此，rnumber 是“引用 long”类型。

与声明指针一样，也可以把&放在变量名的旁边，上面的语句可以写为：

```
long &rnumber=number;                    //Declare a reference to variable number
```

提示：

编译器不介意使用哪种形式。本书使用第一种表示法，即类型名后跟&。

在声明引用时使用&似乎容易混淆。在引用环境下&的用法看起来非常类似于前面在获得变量地址时&的用法。但是，经过一定的实践就会明白它们的区别。

在声明引用时，引用必须总是初始化为变量的一个别名。在声明引用时，决不能不对它进行初始化。更进一步的约束是：引用是固定的，在声明之后就不能修改，它总是同一个变量的别名。引用参数在不同的时候可以显示为引用不同的变量，是因为每次调用函数时，引用参数都会重新创建，并重新初始化。所以对于每个函数调用，都会有一个全新的引用。

引用总是可以用来替代原变量名。例如：

```
rnumber += 10;
```

这个语句给变量 number 加 10，因为 rnumber 是 number 的一个别名。要确保清楚它们之间的区别，下面比较在如下语句中声明的引用 rnumber 和指针 pnumber：

```
long* pnumber = &number;          //Initialize a pointer with an address
```

这个语句声明了指针 pnumber，并用变量 number 的地址初始化它。接着用下面的语句递增该变量：

```
*pnumber += 10;                   //Increment number through a pointer
```

使用指针和使用引用有一个显著的区别：指针需要解除引用，才能获得或操作它指向的变量值，而引用不需要这一步。在某些方面，引用类似于已解除引用的指针，但引用不能修改为引用另一个变量。引用与是引用的变量完全等价。

现在已讨论完目前应讨论的内容，但引用的讨论还远未结束。在第 13 章介绍类是操作时，引用将会得到广泛的应用。

8.3.3 main()的参数

可以把函数 main()定义为在运行程序时接受从命令行上输入的参数。可以指定的参数类型是标准化类型：可以把 main()定义为没有参数，前面的例子都是这样，也可以把 main()定义为如下形式：

```
int main(int argc, char* argv[]) {
  //Code for main()
}
```

第一个参数 argc 是从命令行上输入的字符串的个数。第二个参数 argv 是一个数组，它包含指向每个在命令行上输入的字符串的指针，包括程序名本身。数组 argv 中的最后一个元素(即 argv[argc])总是 0，argv 中的元素个数是 argc+1。

下面看几个例子。假定要运行程序，在命令行上输入下面的内容：

```
Myprog
```

在这个例子中，argc 是 1，argv[]包含两个元素。第一个元素包含字符串 Myprog 的地址，第二个元素是 0。

如果输入下面的内容：

```
Myprog 2 3.5 "Rip Van Winkle"
```

argc 是 4，argv[]包含 5 个元素。前 4 个元素是指向字符串"Myprog.exe"、"2"、"3.5"和"Rip Van Winkle"的指针。第 5 个元素 argv[4]是 0。

如何处理命令行参数完全取决于用户。为了说明如何访问命令行参数，考虑下面的 main()的实现代码：

```
//Program that lists its command line arguments
#include <iostream>
using std::cout;
using std::endl;

int main(int argc, char* argv[]) {
  for(int i=0; i<argc; i++)
     cout<<endl<<argv[i];

  cout<<endl;
  return 0;
}
```

这段代码列出了所有的命令行参数，包括程序名。命令行参数可以是任何内容：例如文件复制程序中的文件名，或要在联系人文件中搜索的人名。换言之，命令行参数可以是在程序执行时所输入的任何内容。

8.4 默认的参数值

在许多情况下，给一个或多个函数参数赋予默认值是非常有用的。这意味着，仅需要在希望参数值不同于默认值时，为参数指定值。

例如，有一个函数要用于输出标准的错误消息。在大多数情况下，使用默认的消息就足够了，但有时需要指定另一个消息。为此，可在函数原型中为参数指定默认值。输出消息的函数的定义如下所示：

```
//Output an error message to the command line
void show_error(const char* message) {
  cout<<endl<<message<<endl;
}
```

要指定默认消息，可以在这个函数的原型中编写一个字符串，用作默认的参数值，如下所示：

```
void show_error(const char* message="Program Error");
```

如果要使用该函数输出默认的消息，在调用它时可以不指定参数值：

```
show_error();              //Display default message
```

该函数会显示结果：

```
Program Error
```

如果要提供某个消息，就可以指定函数的参数：

```
show_error("Nothing works!");
```

下面使用 string 类型的参数定义 show_error()函数，而不是 C 样式的字符串。在这种情况下，参数有默认值的函数原型就应写为：

```
void show_error(const string message="Program Error");
```

当然，还需要在程序中包含 string 头文件，使 string 类型可用。

以这种方式指定默认参数值，会使函数更容易使用，而且可以有任意多个默认参数。下面进一步探讨多个默认参数。

多个默认参数值

在编写函数时，指定了默认值的参数都必须放在参数列表的最后。原因很简单。在调用函数时要使用参数的默认值，应省略相应的参数。省略列表中间的参数会让编译器混淆参数！因此编译器假定，省略了的参数都对应于最右边的参数。

下面看一个有几个默认参数值的函数例子。假定编写一个函数，显示一个或多个数据值，且一行显示几个数据值，如下所示：

```
void show_data(const int data[], int count, const string& title,
                                 int width, int perLine) {
```

```
  std::cout<< std::endl<<title;                    //Display the title

  //Output the data values
  for(int i=0; i<count; i++) {
    if(i%perLine==0)                               //Newline before the first
      std::cout<< std::endl;                       //and after perLine
    std::cout<< std::setw(width)<<data[i];         //Display a data item
  }
  std::cout<< std::endl;
}
```

参数 data 指定了要显示的数据值，count 指定了数据值的个数。第三个参数的类型是 const string&，它指定输出的标题。第四个参数确定每个数据项的字段宽度，最后一个参数是每行显示的数据值个数。

在函数调用时，这个长长的参数列表很烦琐——必须指定所有 5 个参数，才能输出一个数据项！下面给一些参数使用默认值，使之变得容易一些。

程序示例 8.9——使用多个默认参数值

下面的例子调用函数时，为除第一个参数之外的所有参数使用默认值：

```
// Program 8.9 Using multiple default parameter values
#include <iostream>
#include <iomanip>
#include <string>
using std::cout;
using std::endl;
using std::string;

// The function prototype including defaults for reference parameters
void show_data(const int data[], int count = 1,
        const string& title = "Data Values", int width = 10, int perLine = 5);

int main() {
  int samples[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12};

  int dataItem = 99;
  show_data(&dataItem);

  dataItem = 13;
  show_data(&dataItem, 1, "Unlucky for some!");

  show_data(samples, sizeof samples/sizeof samples[0]);
  show_data(samples, sizeof samples/sizeof samples[0], "samples");
  show_data(samples, sizeof samples/sizeof samples[0], "samples", 14);
  show_data(samples, sizeof samples/sizeof samples[0], "samples", 14, 4);

  return 0;
}

// Function to output one or more integer values
void show_data(const int data[],int count, const string& title,
```

```
                                        int width, int perLine) {
    cout << endl << title;                              // Display the title

    // Output the data values
    for(int i = 0 ; i < count ; i++) {
       if(i % perLine ==0)                              // Newline before the first
          cout << endl;                                 // and after perLine

      cout << std::setw(width) << data[i];              // Display a data item
   }
   cout << endl;
}
```

这个程序的结果如下所示：

```
Data Values
          99

Unlucky for some!
          13

Data Values
           1         2         3         4         5
           6         7         8         9        10
          11        12
Samples
           1         2         3         4         5
           6         7         8         9        10
          11        12
Samples
           1         2         3         4         5
           6         7         8         9        10
          11        12
Samples
           1         2         3         4
           5         6         7         8
           9        10        11        12
```

例子的说明

show_data()函数的原型为除第一个参数之外的所有参数指定了默认值：

```
void show_data(const int data[], int count=1,
               const string& title="Data Values", int width=10, int perLine=5);
```

在第三个参数中，可以像非引用类型的参数那样，为引用参数提供默认值。后 4 个参数都有默认值，因此调用该函数有 5 种方式：指定所有 5 个参数、或者省略最后一个参数、省略最后两个参数、省略最后三个参数、省略最后四个参数。只能省略列表最后的几个参数。例如，不能省略第 2 个参数和第 5 个参数：

```
show_data(samples, ,"Samples",15);        //Wrong!
```

在函数 main()中，把用于输出的整数数组定义为：

```
int samples[]={1,2,3,4,5,6,7,8,9,10,11,12};
```

这个数组用于测试 show_data()函数。该函数的第一次应用会输出一个数据值：

```
int dataItem=99;
show_data(&dataItem);
```

因为第一个参数是 int 类型的数组，所以必须把 dataItem 的地址传送为参数。这里应用了 count 的默认值 1 和其他 3 个默认参数值。

接着，输出 dataItem 的修改值，以及一个新标题：

```
dataItem=13;
show_data(&dataItem,1,"Unlucky for some!");
```

我们只想显示一个新标题，但如果指定了第三个参数，还必须指定第二个参数。count 的值与默认值相同，因为本例仍只输出一个数据项。

在下面 4 个语句中输出 samples 数组中的内容：

```
show_data(samples, sizeof samples / sizeof samples[0] );
show_data(samples, sizeof samples / sizeof samples[0] , "Samples");
show_data(samples, sizeof samples / sizeof samples[0] , "Samples",14);
show_data(samples, sizeof samples / sizeof samples[0] , "Samples",14,4);
```

对于数组，需要指定 count 参数，否则 show_data()就只输出该数组的第一个元素。后面的语句多指定了一个参数。

在使用有默认值的多个参数时，需要注意参数的顺序应正确。参数的顺序应是，最不需要指定的参数放在最后，前面的参数需要明确指定值的可能性逐个递增。显然，在许多情况下，这是一个很好的规则。

8.5 从函数中返回值

可以从函数中返回任意类型的值。返回一个基本类型的值是很简单的，但在返回一个指针时有一些陷阱。

8.5.1 返回一个指针

在从函数中返回一个指针时，必须确保它指向的地址是 0，或者在调用函数中仍旧有效的内存地址。换言之，在指针返回到调用函数中后，指针所指向的变量必须仍在其作用域中。这隐含了下面的黄金规则：

不要从函数中返回自动局部变量的地址。

下面看一个例子。假定要编写一个函数，它返回两个值中较大者的地址。这个函数可以用在等号的左边，这样才能改变包含较大值的变量：

```
*larger(value1,value2)=100;          //Set the larger variable to 100
```

这种情况很容易使人误入歧途。下面的实现代码就不能运行：

```
int* larger(int a, int b) {
  if(a>b)
    return &a;                    //Wrong!
  else
    return &b;                    // Wrong!
}
```

很容易看出其中的错误：a 和 b 是局部变量。原整数参数值的副本会传送到局部变量 a 和 b 中，但在返回&a 和&b 时，这些地址中的变量在调用程序中超出了其作用域。如果试图编译这个函数的实现代码，编译器就会发出一个警告。

可以把参数指定为指针：

```
int* larger(int* a, int* b) {
  if(*a>*b)
    return a;                     //OK
  else
    return b;                     //OK
}
```

用下面的代码调用该函数：

```
*larger(&value1,&value2)=100;          //Set the larger variable to 100
```

编写一个返回两个值中较大者的地址的函数并不是很有效，但返回数组中最大元素的地址的函数比较有趣。

程序示例 8.10——返回一个指针

下面编写两个函数，扩展前面的内容。一个函数返回数组中包含最小值的元素的地址，另一个函数返回数组中包含最大值的元素地址。这也可以利用前面的 show_data()函数。本程序示例会修改 double 数组，使其值在 0 到 1 之间。如果只对值的相对大小感兴趣，就可以这么做——把值限制在一个固定的范围内，将更容易以图形方式显示它们。下面是代码：

```
// Program 8.10 Returning a pointer
#include <iostream>
#include <iomanip>
#include <string>
using std::cout;
using std::endl;
using std::string;

void show_data(const double data[], int count = 1,
     const string& title = "Data Values",  int width = 10,  int perLine = 5);
double* largest(double data[], int count);
double* smallest(double data[], int count);

int main() {
  double samples[] = {
                        11.0,   23.0,   13.0,   4.0,
                        57.0,   36.0,  317.0,  88.0,
```

```
                                9.0,  100.0,  121.0,  12.0
                              };

const int count = sizeof samples/sizeof samples[0];

show_data(samples, count, "Original Values");

int min = *smallest(samples, count);

// Shift range of values so smallest is zero
for(int i = 0; i < count ; i++)
   samples[i]  -= min;

int max = *largest(samples, count);


// Normalize range to 0 to 1.0
for(int i = 0; i < count ; i++)
   samples[i] /= max;

show_data(samples, count,  "Normalized Values", 12);
return 0;
}

// Function to find the largest of an array of double values
double* largest (double data[], int count) {
   int index_max =0;
   for(int i = 1; i < count; i++)
      if(data[index_max] < data[i])
         index_max = i;
   return &data[index_max];
}

// Function to find the smallest of an array of double values
double* smallest(double data[], int count) {
   int index_min =0;
   for(int i = 1; i < count; i++)
     if(data[index_min] > data[i] )
         index_min = i;

   return &data[index_min];
}

// Function to display an array of double values
void show_data(const double data[],int count,
               const string& title, int width, int perLine) {
   cout << endl << title;
   for(int i = 0 ;i < count ; i++){
      if(i % perLine == 0)
        cout << endl;
      cout << std::setw(width) << data[i];
   }
```

```
   cout << endl;
}
```

这个程序的结果如下所示：

```
Original Values
          11          23          13           4          57
          36         317          88           9         100
         121          12

Normalized Values
   0.0223642   0.0607029    0.028754           0    0.169329
    0.102236           1    0.268371   0.0159744    0.306709
    0.373802   0.0255591
```

例子的说明

这个程序使用了两个类似的函数 largest()和 smallest()，它们分别返回 double 数组中最大值和最小值的地址。函数 largest()的代码如下所示：

```
double* largest(double data[],  int count) {
   int index_max = 0;
   for(int i = 1; i < count; i++)
      if(data[index_max]  < data[i])
         index_max = i;
   return &data[index_max];
}
```

首先假定数组中的第一个元素是最大值，把 index_max 的值设置为 0。在 for 循环中比较索引位置 index_max 上的元素与数组中的其他元素。如果 index_max 元素小于循环中的当前元素，就把 index_max 设置为当前的索引值。函数返回最大元素的地址，该最大元素由表达式&data[index_max]指定，其中使用了地址运算符。smallest()函数与 largest()的区别仅是在 if 条件中使用的比较运算符不同。

在 main()中，定义一个数组 samples，接着，把数组的元素个数赋予 count。这意味着，不必在每次需要时重复计算该值：

```
const int count = sizeof sample/sizeof samples[0];
```

下面的语句显示元素的初始值：

```
show_data(samples, count, "Original Values" );
```

除了第一个参数的类型之外，show_data()函数的定义与前面的一样。该函数的原型为后 4 个参数定义了默认值，在本例中使用后两个参数的默认值。

使用 smallest()函数找出 samples 数组中的最小值，语句如下所示：

```
int min = *smallest(samples, count);
```

为了获得最小元素的值，可对函数返回的地址应用解除引用运算符*。

还可以改变值的范围，从数组的每个元素中减去当前最小值，使最小值为 0。这在一个 for

循环中实现：

```
for(int i=0;i<count; i++)
  samples[i] -= min;
```

现在值的范围在 0 到某个最大值之间。修改这些值，把它们除以当前最大值，使它们位于 0 到 1 之间。当前最大值是用 largest()函数得到的：

```
int max = *largest(samples, count);
```

接着，在另一个 for 循环中修改这些值，把它们除以当前最大值：

```
for(int i=0;i<count; i++)
  samples[i] /= max;
```

最后输出新值：

```
show_data(samples, count, "Normalized Values" ,12);
```

这次，给字段宽度重新赋值(覆盖默认值)，以容纳每个值的小数位数。

在本节的最后，应注意 largest()和 smallest()函数返回一个地址，这是一个 lvalue，通过解除引用运算符，可以在赋值操作左边的表达式中使用它们返回的值：

```
*largest(samples, count) *=2.0;
```

上面的语句使 samples 数组中最大元素的值翻倍。

8.5.2 返回一个引用

从函数中返回一个指针肯定能达到目的，但也可能出问题，特别是在等号运算符左边使用函数时，就更是如此，如上一节最后所述。指针可以为空，解除对空指针的引用会使程序失败。

解决方法就是从函数中返回引用。因为引用是另一个变量的别名，下面是引用的黄金规则，与指针的黄金规则一样，且应用于同一作用域：

不要从函数中返回自动局部变量的引用。

引用是另一个变量的别名，按照定义，它是一个 lvalue。所以从函数中返回引用，允许在等号左边使用函数调用。实际上，从函数中返回引用是在等号左边直接使用函数调用(不需要解除引用)的惟一方式。

假定 larger()函数定义为：

```
int& larger(int& m, int& n) {
   return m>n?m:n;          //Return a reference to the larger value
}
```

返回类型是对 int 的引用，参数是非常量的引用。我们要返回某个引用参数，就不能把参数声明为 const。

现在可以使用该函数修改两个参数中的较大值，如下面的语句所示：

```
larger(value1,value2)=50;   //Change the value of the larger one to 50
```

以这种方式声明函数，就无法把常量用作参数。因为参数是非常量引用，所以编译器就不允许把常量用作参数。引用参数允许修改值，而编译器肯定不认可改变常量。

这里不打算介绍使用引用返回类型的扩展示例，但不久就会遇到它。在使用类创建自己的数据类型时，引用返回类型是必不可少的。

8.5.3 从函数中返回新变量

在函数中，可以在自由存储区中创建新变量，并通过指针返回值将它返回给调用程序。使用 new 运算符就可以为新变量分配内存空间，并返回其地址。

使用这个技术的危险性在于，出现内存泄漏的可能性非常高。每次调用这样的函数时，都要在自由存储区中分配更多的内存。调用函数应负责使用 delete 运算符释放这些内存。

8.6 内联函数

如果函数非常短(刚才讨论的 larger()函数就很短)，编译器为处理传送过来的参数以及返回结果的代码的系统开销，与进行实际计算的代码相比就非常多。这两类代码的执行时间非常相关。

在极端情况下，调用函数的代码占用的内存会比函数体中的代码还多。在这种情况下，编译器就应使用函数体中的实际代码替代函数的调用，并作适当的调整，以处理局部名称。这会使程序更短或更快。

为了让编译器完成此任务，可以在函数的定义中使用 inline 关键字。如下所示：

```
inline int larger(int m, int n) {
   return m>n?m:n;
}
```

通过这个定义，编译器就会用内联代码替代调用。但是，这只是一个建议，它取决于编译器是否采纳这个建议。把函数声明为 inline，函数的定义就必须可以在调用函数的每个源文件中使用。因此，内联函数的定义通常放在头文件中，而不是在源文件中，该头文件包含在使用该函数的每个源文件中。

不同的编译器采用不同的规则来确定定义为 inline 的函数是否用内联代码代替其调用。基本的先决条件是这种函数必须非常短、非常简单。显然，把一个较长的函数声明为 inline 会降低效率，如果该函数还在程序中调用许多次，效率就更低。这种做法会大大增加程序的长度，而执行时间没有或很少有改善。

编译器选择不按照请求把函数看作内联函数，这有一个缺点。在这种情况下，函数调用会按照正常的函数调用来编译，但编译器一般还会把该函数看作源文件的本地函数，所以每个使用它的源文件都要拥有该函数的已编译副本。结果是，如果函数在几个不同的源文件中使用，函数的代码就要进行不必要的重复。

8.7 静态变量

在前面编写的所有函数中，函数体在每次执行之后都不会保留任何信息。假定要计算某个

函数的调用次数，该怎么办？一种方法是在文件作用域内定义一个变量，然后再在函数中递增它。但这个方法的一个潜在问题是，该变量可能被程序文件中的任何函数修改，这样就不能肯定它会按照我们希望的那样变化。

较好的解决方法是在函数体中把该变量声明为 static，静态变量在第 3 章简要介绍过。静态变量在定义它的语句中创建，在此之后，它就一直存在，直到程序结束为止。也就是说，该变量可以在函数的多次调用之间传送值。

为了把变量声明为静态，只需在变量的声明中，在类型名前面加上关键字 static。例如：

```
static int count=1;
```

在执行这个语句时，就会创建静态变量 count，并初始化为 1。以后再执行该语句就没有效果了。变量 count 将一直存在，直到程序结束为止。如果在声明中省略了初始化值，静态变量就初始化为 0。

下面的函数示例声明了一个静态变量：

```
void nextInteger() {
  static int count=1;
  cout<<endl<<count++;
}
```

这个函数在显示静态变量 count 的当前值后，递增它的值。函数第一次调用时，该变量显示值 1。第二次调用时，则显示值 2。每次调用函数时，都会显示一个比上一次调用大 1 的值。静态变量 count 只在第一次调用函数时创建和初始化一次。只要程序在执行，以后对函数的调用都会使用 count 的当前值。

可以把任意类型的变量声明为静态，也可以对任何需要在函数的多次调用之间保留下来的数据项使用静态变量。例如，可以存储读取最后一个文件记录的次数，或所传送的参数中的最大值。

下面用一个例子来说明静态变量。

程序示例 8.11——在函数中使用静态变量

斐波纳契级数是一组整数，其中每个整数都等于前面两个整数之和。下面编写一个程序，使用静态变量回调函数计算的前两个整数，返回序列中的下一个整数。

显然，因为在程序启动前是没有“前面的整数”的，所以先把两个变量初始化为 0 和 1。这不是一种好的编程模式，但说明了静态变量在复杂环境下的属性。

```
//Program 8.11 Using a static variable
#include <iostream>
#include <iomanip>
using std::cout;
using std::endl;

long next_Fibonacci();

int main() {
  cout<<endl<<"The Fibonacci Series"<<endl;
  for(int i=0; i<30; i++) {
    if(i%5==0)                          //Every fifth number…
```

```
        cout<<endl;                          //…start a new line
    cout<<std::setw(12)<<next_Fibonacci();
}
cout<<endl;
return 0;
}

//Function to generate the next number in the Fibonacci series
long next_Fibonacci() {
  static long last=0;                        //Last number in sequence
  static long last_but_one=1;                //Last but one number

long next = last + last_but_one;             //Next is sum of last two
last_but_one = last;                         //Update last but one
last = next;                                 //last is new one
return last;                                 //Return the new one
}
```

这个程序的结果如下所示：

```
The Fibonacci Series
           1            1            2            3            5
           8           13           21           34           55
          89          144          233          377          610
         987         1597         2584         4181         6765
       10946        17711        28657        46368        75025
      121393       196418       317811       514229       832040
```

例子的说明

函数 main()在一个循环中调用了 next_Fibonacci()函数 30 次：

```
for(int i=0; i<30; i++) {
  if(i%5==0)                                 //Every fifth number…
    cout<<endl;                              //…start a new line
  cout<<std::setw(12)<<next_Fibonacci();
}
```

因为没有传送参数，所以返回的值在 next_Fib()onacci 函数内部生成。所声明的两个静态变量存储了序列中最近生成的整数和其前面的整数：

```
static long last=0;                          //Last number in sequence
static long last_but_one=1;                  //Last but one number
```

把 last 初始化为 0，last_but_one 初始化为 1，就得到了这个序列中的第三个数，即 1。每次调用 next_Fibonacci()函数时，都会把序列中的前两个数加在一起，计算出下一个数。结果存储在自动变量 next 中：

```
long next = last + last_but_one;             //Next is sum of last two
```

可以这么做是因为 last 和 last_but_one 是静态变量，它们包含了在 next_Fibonacci()函数的前一次迭代中赋予它们的值。

在返回 next 值之前，必须把 last 中的值赋予 last_but_one，把新值赋予 last：

```
last_but_one = last;                    //Update last but one
last = next;                            //Last is new one
```

只要程序存在，静态变量就存在，但静态变量只能在声明它们的块中访问，所以变量 last 和 last_but_one 只能在 next_Fibonacci()函数体中访问。

注释：

斐波纳契级数看起来像是一组无趣的数字，但它们应用于各个领域。例如，植物长出新叶子时，它们之间的角度是 2π乘以这个序列中间隔数字的比率。所以，植物长叶子的角度是 2π乘以 1/2(第 1 个数和第 3 个数)，2π乘以 1/3(第 2 个数和第 4 个数)，2π乘以 2/5(第 3 个数和第 5 个数)，等。是不是很神奇？

8.8 本章小结

本章介绍了函数的编写和使用。第 9 章将更详细地论述函数，并主要讨论第 11 章才开始介绍的用户定义类型。本章的主要内容如下：

- 函数是一个自包含的代码单元，它有着已定义好的目的。一般的程序总是包含大量的小函数，而不是包含几个大函数。
- 函数定义包含定义了函数名称、参数和返回类型的函数头，以及包含函数的可执行代码的函数体。
- 函数原型允许编译器处理对该函数的调用，但此时函数定义可能还没有处理。
- 由于给函数传送参数的按值传送机制，是传送原参数值的副本，因此原参数值不能在函数中访问。
- 给函数传送指针可以修改该指针所指向的值，但指针本身是按值传送。
- 把指针参数声明为 const 可以防止修改原来的值。
- 可以把数组的地址作为指针传送给函数。
- 使用引用给函数传送值，即按引用传送机制，可以避免按值传送参数中的隐含复制过程。在函数中不应修改的参数都应指定为 const。
- 为函数的参数指定默认值后，只要参数有默认值，就允许有选择地省略参数。
- 从函数中返回引用，允许在等号运算符的左边使用该函数。把返回类型声明为 const 引用，可以阻止在等号运算符的左边使用该函数。

8.9 练习

1. 编写一个函数 value_input()，它接受两个整数参数和一个提示用户输入值的字符串参数，函数会提示所输入的值应在参数指定的范围之内。函数应一直提示用户输入值，直到输入的值有效为止。

在程序中使用该 value_input()函数，获取用户的生日，验证月份、日期和年份是否有意义。最后，以下面的格式在屏幕上输出该生日：

```
Noverber 21, 1977
```

这个程序应使各个函数 month()、year()和 day()管理对应数字的输入，不要忘了闰年。

2. 编写一个函数，它要求输入一个字符串或一个字符数组，并反转它的顺序。使用什么类型的参数最好？用 main()函数测试该函数，提示用户输入一个字符串，反转其顺序，再输出反转后的字符串。

3. 编写一个程序，它接受 2 到 4 个命令行参数。如果用少于 2 个或多于 4 个参数调用该程序，就输出一个消息，告诉用户应怎么做，然后退出。如果参数的个数是正确的，就输出参数，一行输出一个参数。

4. 修改上一题中的程序，让它只接受两个参数，把第二个参数传送给第 2 题中的字符串反转函数。输出反转后的字符串。

5. 编写一个函数，它返回两个 long 变量中较小值的引用。编写另一个函数，它返回较大值的引用。使用这两个函数生成斐波纳契级数的元素，即 1，1，2，3，5，8，13.....序列，其中每个数都等于前两个数之和，元素的个数由用户指定(提示：把序列中的两个数字存储在 n1 和 n2 中，它们都从 1 开始。如果把两个数的和存储在较小的变量中，并输出较大的值，就会得到希望的结果。难点是为什么这是可行的)。

第9章 函 数

本章将讨论设计和使用函数的过程中比较微妙的一些方面。主要论述参数类型和返回类型对函数操作方式的影响，以及函数的使用场合。还要阐述自动创建函数定义的方法。

本章主要内容

- 如何实现若干个同名函数
- 如何把指针和引用用作参数
- 用 const 修饰符声明参数的效果
- 函数模板的概念和用法
- 如何定义和使用函数指针
- 递归函数的概念和工作原理

9.1 函数的重载

我们常常需要用两个或多个函数完成相同的任务，但其参数的数据类型不同。例如，需要使用函数 larger()的不同版本，查找出任何基本数据类型的两个变量中的较大者。当然，在理想情况下，所有这些函数都有相同的名称 larger()，另一种方法是为 int 类型的变量定义函数 larger_int()，为 float 类型的变量定义函数 larger_float()，这很笨拙、费力。这种情况称为函数的重载。

利用函数的重载，可以在一个程序中使用同名的若干个函数。主要的限制是给定名称的每个函数必须有不同的参数列表。也就是说，一个参数列表中的参数类型与另一个不同——不仅仅是参数名不同。实际上，如果满足下列条件之一，两个同名函数就是不同的：

- 每个函数的参数个数不同
- 参数的个数相同，但至少有一对对应参数的类型不同

这里需要讨论一些比较微妙的地方。

9.1.1 函数的签名

函数的名称及其参数类型组合在一起，就定义了一个惟一的特性，称为函数签名。函数签名可以区分不同的函数，所以程序中的每个函数都必须有惟一的函数签名。只要在程序中调用函数，就必须满足这个条件。

从函数原型或函数定义来看，编译器为程序文件中的每个函数都定义了一个函数签名。在编写包含函数调用的语句时，编译器就会使用该调用创建一个函数签名，再把它与函数原型和/或定义中可用的函数签名集比较。如果找到匹配的函数签名，就建立所调用的函数。如果没有找到匹配的函数签名，就检查转换参数类型后是否有匹配的函数签名。

注释：

返回类型不是函数签名的一部分。实际上，这是符合逻辑的。在调用函数时，不需要存储返回的值，因此函数的返回类型就不必由调用语句确定。

程序示例 9.1——重载一个函数

下面是一个非常简单的例子，它使用前面编写的 larger()函数：

```
// Program 9.1 Overloading a function
#include <iostream>
using std::cout;
using std::endl;

//Prototype for two different functions
double larger(double a, double b);
long larger(long a,  long b);

int main() {
   double a_double = 1.5, b_double = 2,5;
   float a_float = 3.5f, b_float = 4.5f;
   long a_long = 15L, b_long = 25L;
   cout << endl;
   cout << "The larger of double values "
        << a_double <<" and " << b_double <<" is "
        << larger(a_double, b_double)   << endl;
   cout << "The larger of float values"
        << a_float <<" and " << b_float <<" is "
        << larger(a_float, b_float)   << endl;
   cout << "The larger of long values"
        << a_long <<" and " << b_long <<" is "
        << larger(a_long, b_long)   << endl;
  return 0;
}

// Function to return the larger of two floating point values
double larger(double a, double b) {
  cout << "double larger() called" << endl;
  return a>b ? a : b;
}

// Function to return the larger of two integer values
long larger(long a, long b) {
  cout << "long larger() called" << endl;
  return a>b ? a : b;
}
```

这个程序的结果如下所示：

```
double larger() called
The larger of double values 1.5 and 2.5 is 2.5
double larger() called
The larger of float values 3.5 and 4.5 is 4.5
```

```
long larger() called
The larger of long values 15 and 25 is 25
```

例子的说明

函数 larger()有两个重载版本，它们都在调用时在命令行上写一个消息。main()中的第一次调用使用了参数类型为 double 的版本：

```
cout << "The larger of double values "
     << a_double <<" and " << b_double <<" is "
     << larger(a_double, b_double)   << endl;
```

因为函数签名集中有一个匹配的函数签名，所以编译器会选择正确的函数来使用。输出显示，选择了参数类型为 double 的函数。

下一个调用有点不同，如下所示：

```
cout << "The larger of float values "
     << a_float <<" and " << b_float <<" is "
     << larger(a_float, b_float)   << endl;
```

这个语句调用了参数类型为 float 的函数 larger()。我们没有定义这样的函数。但是，编译器可以把 float 转换为 double，且没有数据丢失；于是，这是一个可接受的自动转换，编译器可以使用接受 double 参数的 larger()版本。

下一个语句给函数 larger()传送了 long 类型的参数：

```
cout << "The larger of long values "
     << a_long <<" and " << b_long <<" is "
     << larger(a_long, b_long)   << endl;
```

这与接受 long 类型的参数的版本完全匹配，编译器就调用这个版本。

当然，因为类型 int 转换为类型 long 时，没有数据丢失，所以在本例中，可以使用带有 long 参数的函数版本。下面在 main()中添加几个语句：

```
int a_int =35,  b_int = 25;
cout  << "The larger of int values "
      << a_int <<" and " << b_int <<" is "
      << larger(a_int, b_int)   << endl;
```

程序就不运行了。编译器不能确定该使用哪个版本的 larger()。因为从 int 自动转换为 long 或从 int 自动转换为 double 都是可以接受的，所以无法作出决策。换言之，代码是含糊的。编译器不打算做猜钱币反正面的赌博，而希望用户来确定使用哪个版本。只要进行显式的强制转换，就可以解决这个问题：

```
int a_int =35,  b_int = 25;
cout << "The larger of int values "
     << a_int <<" and " << b_int <<" is "
     << larger(static_cast<long>( a_int), static_cast<long>( b_int))
     << endl;
```

目前有一个接受 long 参数的版本。把两个参数中的任何一个强制转换为 long 就足够了——编译器可以选择出接受 long 参数的版本。但是，对两个参数都进行强制转换，会使任何阅读该段代码的人都非常清楚这段代码的意图。

9.1.2 重载和指针参数

显然，由于指向不同类型的指针是不同的，因此下面的原型声明了两个不同的重载函数：

```
int larger(int* pValue1, int* pValue2);
int larger(float* pValue1, float* pValue2);
```

可以使用指向给定类型的指针作为参数。注意，它的解释方式与该类型的数组相同。例如，int*类型的参数处理起来与 int[]的参数类型相同。下面的原型声明了相同的函数，而不是两个不同的函数：

```
int largest(int values[], int count);    // Identical signature to below
int largest(int* values, int count);     // Identical signature to above
```

指定这两种参数类型中的任何一种，所传送的参数都是地址，可以使用数组表示法或指针表示法来实现该函数。

9.1.3 重载和引用参数

如果重载带有引用参数的函数，就需要小心了。原因是，不能把参数是给定类型 data_type 的函数，重载为参数类型是“引用 data_type”的函数。否则，编译器就不能确定要调用哪个函数。为了说明这一点，下面声明两个函数原型：

```
int do_it(int number);            //These are not
int& do_it(int& number);          //distinguishable
```

假定 value 的类型是 int，则下面的语句：

```
do_it(value);
```

就可以调用这两个函数中的任何一个。无法区分应调用哪个函数，因此不能根据一个版本的参数是给定类型，另一个版本的参数是该类型的引用，来区分重载函数。

还要注意，在重载函数时，声明两个版本，一个版本的参数是“引用 type1”，另一个版本的参数是"引用 type2"。调用哪个函数取决于所使用的参数类型，但会得到意想不到的结果。

程序示例 9.2——重载带有引用参数的函数

下面深入探讨引用参数的用法，修改上一个例子的代码：

```
//  Program 9.2  Overloading a  function with  reference parameters
#include <iostream>
using std::cout;
using std::endl;

double larger(double a,  double b);
long& larger(long& a,  long& b);

int main() {
   double a_double = 1.5, b_double = 2,5;
```

```
cout << endl;
cout << "The larger of double values "
     << a_double <<" and " << b_double <<" is "
     << larger(a_double, b_double)   << endl;

int a_int =35,  b_int = 25;
cout << "The larger of int values "
     << a_int <<" and " << b_int <<" is "
     << larger(static_cast<long>( a_int), static_cast<long>( b_int))
     << endl;
 return  0;
}

// Function to return the larger of two floating point values
double larger(double a, double b) {
   cout << "double larger() called. " << endl;
   return a>b  ?  a : b;
}

// Return the larger of two long references
long& larger(long& a, long& b) {
   cout << "long ref larger() called. " << endl;
   return  a>b  ?  a : b;
}
```

这个版本的程序输出如下结果：

```
double larger() called.
The larger of double values 1.5 and 2.5 is 2.5
double larger() called.
The larger of int values 15 and 25 is 25
```

例子的说明

输出结果中的第二行并不是我们期望的结果，我们希望第二个输出语句调用带有 long&参数的 larger()版本：

```
cout << "The larger of int values "
     << a_int <<" and " << b_int << " is "
     << larger(static_cast<long>( a_int), static_cast<long>( b_int))
     << endl;
```

而本例中的语句调用带有 double 参数的函数，为什么？我们已经把两个参数都强制转换为 long 了！

实际上，这就是问题所在。参数不是 a_int 和 b_int，而是包含相同值的临时位置，这两个值转换为 long 类型。在幕后，编译器没有准备使用临时地址来初始化引用，这太冒险了。larger()中的代码可以自由控制它对引用参数进行的操作，在理论上，两个引用参数都可以修改和/或返回。因为以这种方式使用临时位置不是很明智，所以编译器不使用。

该如何处理这个问题？有两个选择。可以把 a_int 和 b_int 声明为 long 类型。编译器就会调用参数类型为“引用 long”的 larger()版本。

如果环境不允许这么做，还可以把引用参数声明为 const：

```
long larger(const long& a, const long& b);
```

一定要在函数原型和函数定义中同时进行修改，通知编译器函数不能修改参数，于是编译器就允许调用这个版本，而不是参数为 double 的版本。

注意只返回了 long 类型。如果一定要返回引用，就必须把返回类型声明为 const，因为编译器不能把 const 引用转换为非 const 引用。const 引用肯定不是 lvalue，所以不能在等号左边使用它。因此，在这个例子中返回 long 类型的值不会有任何数据损失。

9.1.4 重载和 const 参数

const 仅能用于在定义函数签名时，区分是为引用定义参数，还是为指针定义参数。对于基本类型(如 int)，从重载的观点来看，const int 与 int 是相同的。因此，下面的原型具有相同的函数签名，并声明同一个函数：

```
long& larger(long a, long b);
long& larger(const long a, const long b);
```

编译器会忽略第二个声明中参数的 const，这是因为参数是按值传送的。也就是说，会把每个参数的副本传送给函数，函数不会修改参数的初始值。

注意：

对于任何基本类型 T，参数类型是 const T 的函数都会解释为参数类型为 T 的重载函数，const 会被忽略。

1. 重载和 const 指针参数

如果在两个重载函数中，一个函数的参数类型是“指向 type 的指针”，而另一个函数的参数类型是“指向 const type 的指针”，这两个函数就是不同的。对应参数是指向不同实体的指针，实际上，它们有不同的类型。例如，下面的原型有不同的函数签名：

```
long* larger(long* a, long* b);                          // pointer parameters
const long* larger(const long* a, const long* b);// pointer to const parameter
```

应用于指针所指向的值的修饰符 const 禁止修改该值。而没有 const 修饰符，初始值可以通过指针来修改，按值传送机制不能阻止这种修改。在这个例子中，第一个函数用下面的语句调用：

```
long num1=1;
long num2=2;
long num3=*larger(num1,num2);
```

下面的代码调用带有 const 参数的第二个版本 larger()：

```
const long num10=1;
const long num20=2;
const long num30=*larger(num10,num20);
```

编译器不会把常量值传送给参数是指针的函数。通过指针传送常量值违反了变量的 const 声明。因此编译器必须选择 larger()的后一个版本。

通过比较，如果在两个重载函数中，一个函数的参数类型是“指向 type”，另一个函数的参数类型是 const“指向 type”，这两个函数就是相同的。例如：

```
long* larger(long* a, long* b);                          // These are indentical
long* const larger(long* const a, long* const b);
```

这两个函数没有区别，也不会编译。原因很清楚，第一个原型的参数类型是“指向 long”，第二个原型的参数类型是 const“指向 long”。如果把“指向 long”看做一种类型 T，则这两个函数的参数类型是 T 和 const T，这是无法区分的(本节的开始解释过这种情形)。

2. 重载和 const 引用参数

引用参数在声明为 const 时比较简单。对类型 T 的引用和对类型 T 的 const 引用总是不同的，例如，类型 const int&总是与类型 int&不同。也就是说，可以用下述原型中的方式来重载函数：

```
long& larger(long& a, long& b);
long  larger(const long& a, const long& b);
```

每个函数都有相同的函数体，返回两个参数中的较大者，但函数的运行是不同的。第一个原型声明的函数不接受常量作为参数，但该函数可以放在等号的左边，修改引用参数。第二个原型声明的函数接受常量作为参数(当然也接受非常量参数)，但由于返回类型不是 lvalue，不能在等号左边使用它，这一点将在程序示例 9.2 中讨论：把返回类型声明为 const 引用没有意义。它仍不能表示 lvalue。

提示：

返回值对重载没有影响。在生成函数的签名时可以不考虑它。但是，如前所述，返回值对函数的使用场合有影响。

9.1.5 重载和默认参数值

第 8 章说过，可以为函数指定默认的参数值。但对于重载函数，指定默认参数值有时会影响编译器区分函数调用的能力，产生不确定的结果。例如，假定 show_error()函数有两个版本，第 8 章曾使用该函数显示错误消息。下面是用 C 样式字符串参数定义的 show_error()函数：

```
void show_error(const char* message) {
  std::cout << std::endl << message << std::endl;
}
```

另一个版本用 string 参数来定义：

```
void show_error(const string& message) {
  std::cout << std::endl << message << std::endl;
}
```

无法为这两个函数定义默认参数，因为这会出现不确定性。用这两个版本生成默认消息的语句如下所示：

```
show_error();
```

对于这个函数调用，编译器不知道应调用哪个函数。当然，这是一个很无聊的例子：根本就不必为这两个函数指定默认值。其默认值可以是任何内容。但是，我们会遇到不那么无聊的情况，此时就必须确保所有的函数调用都可以惟一地确定应调用的函数。

9.2 函数模板

在上面描述的一些情形中，我们编写了包含相同代码的重载函数。似乎没有必要重复编写相同的代码，最好代码只编写一次，即采用函数模板的方式。

函数模板是函数的蓝图或处方，编译器使用它生成函数系列的新成员。新函数在第一次使用时创建。从函数模板中生成的函数称为该模板的一个实例或模板的实例化。描述函数模板的一种技巧性较高的方式是，与参数化的函数定义一样，一个函数由一个或多个参数来选择。参数是通常的数据类型(但并不总是这样，例如，参数还可以用于提供传送过来的整数值所占用的字节数)。下面用一个例子来详细说明函数模板。

前面介绍的 larger()函数就很适合于做模板。这个函数的模板定义如图 9-1 所示。

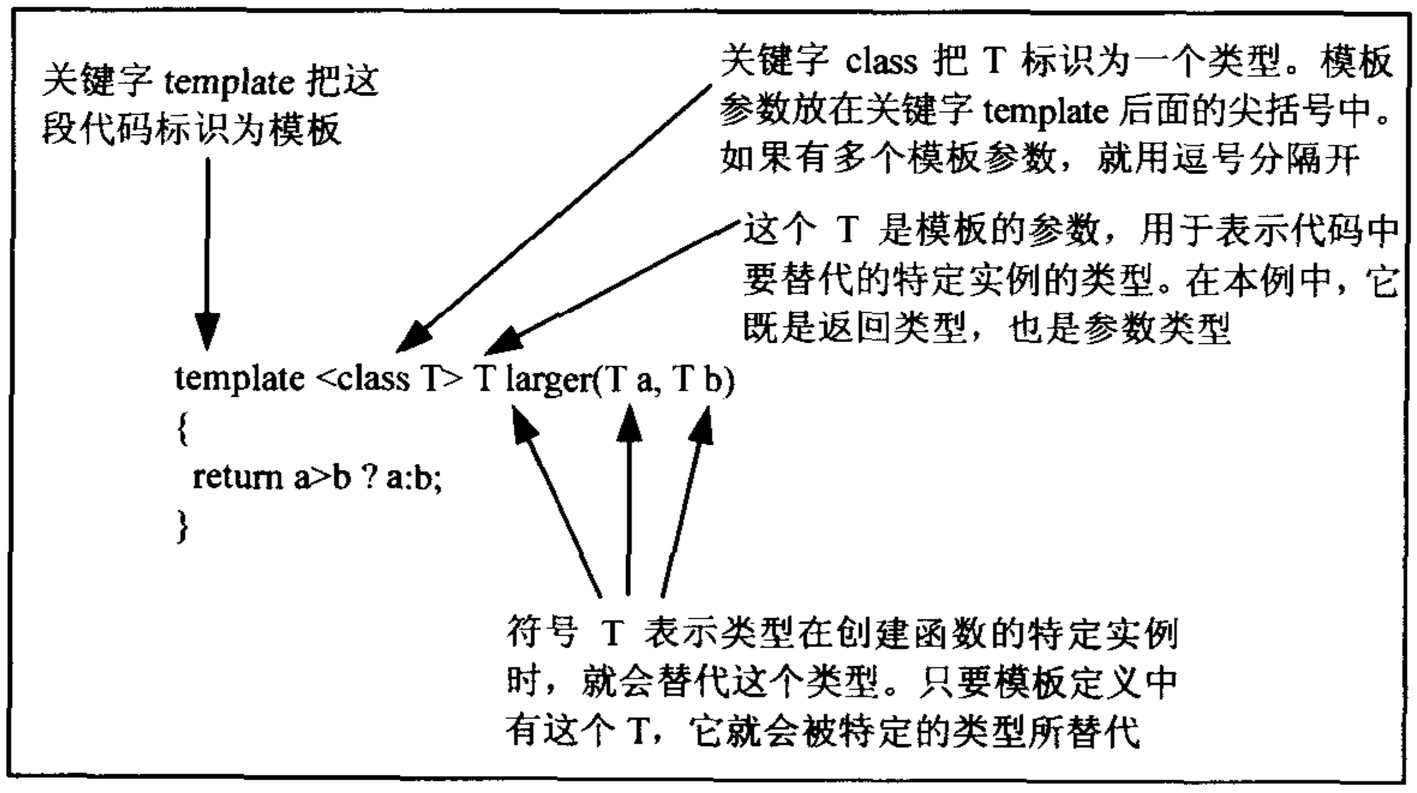


图 9-1 一个简单的函数模板

函数模板的开头是关键字 template，表示这是一个模板。其后是一对尖括号，它包含了参数列表。在本例中只有一个参数 T。T 通常用作参数名，因为大多数参数都是类型，而 T 是单词 type 的第一个字母。实际上，参数可以使用任何名称，replace_it 和 my_type 这样的名称都是有效的。

单词 class 是一个关键字，它表示 T 是一个类型。但这不表示 T 必须是类类型。T 可以是基本类型，例如 int 或 long，也可以是用户定义的类型(换言之即类)。在后面讨论类时将论述如何创建自己的类型。

注释：

在模板定义中，尖括号中可能会包含关键字 typename，这是关键字 class 的另一个常见名称，与 class 有相同的作用。本书主要使用关键字 class(优先于 typename)。

定义中的其他内容就像函数的定义一样，但其中有参数 T。编译器会创建函数的一个新版本，用某个用于创建模板实例的特定类型替换模板定义中的 T。

在代码中编写模板就像编写正常的函数定义一样，甚至可以编写模板函数原型。如下所示：

```
template<calss T> T larger(T a, T b);
```

在使用从模板中生成的函数之前，必须确保把声明(即原型)或模板的定义放在源文件中。

9.2.1 创建函数模板的实例

编译器将从使用函数 larger()的语句中，按照要求创建出函数的实例。例如：

```
cout<<"Larger of 1.5 and 2.5 is "<<larger(1.5,2.5)<<endl;
```

可以看出，函数是按一般方式使用的。尤其是没有为模板参数 T 指定值——编译器会从函数调用的参数中推断出 T 的值。这里，因为 larger()的参数是 double 类型，所以这个调用会让编译器搜索带有 double 参数的 larger()版本。如果没有找到，编译器就会从模板中创建这个 larger()版本，并用类型 double 替换模板定义中的 T。

得到的模板函数接受 double 类型的参数，返回一个 double 值。用 double 替换了 T 后，模板实例就变成：

```
double larger(double a, double b) {
  return a>b ? a:b;
}
```

注释：

这个模板的实例化只生成一次。如果后续的函数调用需要同一个实例，就会调用已经创建好的实例。即使同一个实例在不同的源文件中生成，程序也仅包含该实例定义的一个副本。

在后面对函数模板的详细讨论中，要注意两个问题。第一，函数模板本身不做任何工作，它是编译器用于从函数调用中创建函数定义的处方或蓝图。第二，所有工作都在编译和链接过程中完成。编译器使用模板生成函数定义的源代码，再编译这些代码。链接程序的作用是仅把函数的一个实例链接到可执行模块上，即使几个不同的源文件调用同一个实例，也只链接一个实例。在执行程序时，源代码中是否存在模板根本不重要。

熟悉了这些概念后，下面在程序中测试函数模板。

程序示例 9.3——使用函数模板

使用模板是很简单的。下面的代码以各种方式使用函数模板：

```
// Program 9.3 Using a function template
#include <iostream>
using std::cout;
using std::endl;

template<class T> T larger(T a, T b);       // Function template prototype

int main() {
```

```
cout << endl;
cout << "Larger of 1.5 and 2.5 is" << larger(1.5, 2.5) << endl;
cout << "Larger of 3.5 and 4.5 is" << larger(3.5, 4.5) << endl;

int a_int = 35;
int b_int = 45;
cout << "Larger of" << a_int << "and" << b_int << " is "
     << larger(a_int, b_int)
     << endl;

long a_long = 9;
long b_long = 8;
cout << "Larger of" << a_long << "and" << b_long <<" is"
     << larger(a_long, b_long)
     << endl;

  return 0;
}

// Template for functions to return the larger of two values
template <class T> T larger(T a, T b) {
  return a>b ? a : b;
}
```

这个程序的结果如下所示：

```
Larger of 1.5 and 2.5 is 2.5
Larger of 3.5 and 4.5 is 4.5
Larger of 35 and 45 is 45
Larger of 9 and 8 is 9
```

例子的说明

前面说过，模板的定义或原型必须放在调用任何函数实例之前，这个规则与一般的函数相同，因此把模板原型定义为：

```
template<calss T> T larger(T a, T b);              //Function template prototype
```

这实际上与模板定义的第一行相同，只是在语句的最后有一个分号。

在 main()中，第一次调用函数 larger()，如下面的语句所示：

```
cout << "Larger of 1.5 and 2.5 is " << larger(1.5, 2.5) << endl;
```

执行这个语句后，编译器会自动创建接受 double 类型的参数的 larger()版本，接着调用该版本。下一个语句也需要调用接受 double 参数的 larger()版本：

```
cout << "Larger of 3.5 and 4.5 is" << larger(3.5, 4.5) << endl;
```

编译器将使用为前一个语句生成的版本。

下一个使用 larger()的语句如下：

```
cout << "Larger of" << a_int << " and " << b_int << " is "
     << larger(a_int, b_int)
```

```
        << endl;
```

由于这次要调用参数类型为 int 的函数，因此创建一个 larger()新版本，接受 int 参数。larger()函数的最后一次调用有两个 long 类型的参数：

```
cout  << "larger of " << a_long << " and" << b_long << "is "
      << larger(a_long, b_long)
      << endl;
```

这次将获得带有两个 long 类型参数的新版本。这个程序从一段源代码中获得了总共三个不同版本的 larger()(其对象代码有所不同)。

9.2.2 显式指定模板参数

在调用函数时，可以显式指定模板的参数，以控制使用哪个版本的函数。编译器不再推断用于替换 T 的类型，只是接受指定的版本。在下列情况下，指定模板参数是很有用的：

- 函数调用不是很确切，编译失败。此时可以使用该技巧帮助编译器去除不确定性。
- 在一些情况下，编译器不能推断出模板参数，因此无法选择要使用哪个版本的函数(在考虑有多个参数的模板时，将介绍一个这方面的例子)。在这种情况下，就必须显式指定模板参数。
- 为了避免有太多的函数版本(从而避免过多占用内存)，可以强迫函数调用使用某个版本的函数。

下面是一个例子。在程序示例 9.3 中，int 类型的参数可以由接受 long 类型参数的 larger()版本处理(因为总是需要后一个版本)。在调用函数时，指定要使用的模板参数类型，就可以强迫使用该版本的函数：

```
cout  << "Larger of" << a_int << " and " << b_int << "is"
      << larger<long>(a_int, b_int)
      << endl;
```

在这个语句的函数调用中，模板参数值 long 在函数名后面的尖括号中定义。因此，会从模板中生成并使用对应于 long 类型的函数。编译器可以把参数的类型自动转换为函数参数所需要的类型。

另外，如果觉得参数类型为 double 的函数版本足够用了，只需在尖括号中指定 double；编译器就会生成并使用该版本。甚至可以迫使编译器使用可能导致数据丢失的版本，例如 short 类型。一般情况下，编译器会警告数据有可能丢失，但仍实现和使用指定的版本。

下面的情形也适合于使用明确的模板参数：

```
cout  << "larger of" << a_long << "and" << a_int << "is "
      << larger(a_long, a_int)
      << endl;
```

其中，变量 a_long 和 a_int 分别是 long 类型和 int 类型，如程序示例 9.3 所示。因为这两个参数有不同的类型，这种情况与函数模板不匹配，所以编译器不能生成合适的函数。为此，可以显式指定模板参数：

```
cout << "larger of " << a_long << " and" << a_int << "is "
     << larger<long>(a_long, a_int)
     << endl;
```

现在，编译器会把 T 替换为 long，生成函数，再强制转换 a_int，以完成调用。

另一个解决方案是把函数调用中的一个参数强制转换为另一个参数的类型。接着，编译器就会把该函数解释为接受两个相同类型的参数，并调用合适的函数版本。

9.2.3 模板的说明

假定扩展程序示例 9.3，用地址参数调用函数 larger()：

```
cout << "Larger of" << a_long << " and " <<b_long << " is "
     << *larger (&a_long, &b_long)
     << endl;
```

这个语句执行后，编译器会创建一个模板参数类型是 long*的函数版本，这个函数有如下原型：

```
long* larger (long*, long*);
```

返回值是一个地址，必须解除对它的引用，才能输出其值。但是，这么做结果常常不正确！这是因为函数体中的比较不正确。生成的函数如下所示：

```
long* larger(long* a, long*b) {
  return a>b ? a:b;
}
```

这是在比较地址，而不是比较值！这个函数返回值比较大的地址，而我们希望该地址包含值较大的 long 整数值。这说明使用模板很容易出错。在模板中把指针类型用作参数值时要特别小心。

对于这种情况，该如何处理？可以定义模板的说明。对于某个参数值(在有多个参数的模板中，就是一组参数值)，模板的说明定义了它不同于标准模板的动作。模板说明的定义必须放在原语句的声明或定义之后。如果把说明放在前面，程序就不会编译。

定义模板说明

说明的定义以关键字 template 开头，但要省略参数，所以原声明中模板参数外部的尖括号就是空的。必须定义说明的参数值，而且必须放在模板函数名后面的尖括号中。例如，long*的 larger()函数，其说明如下所示：

```
template <> long* larger<long*>(long* a, long*b) {
  return *a>*b ? a:b;
}
```

函数体的惟一改变是解除参数 a 和 b 的引用，以便比较数值，而不是地址。下面是一个实际的例子。

程序示例 9.4——使用明确的说明

修改程序示例 9.3，以使用明确的说明。添加上述把地址传送给 larger()函数的问题语句，再添加一个明确的说明，来处理 long*类型的参数：

```
// Program 9.4 Using function template specialization
#include <iostream>
using std::cout;
using std::endl;

template<class T> T larger(T a, T b);    // Function template prototype
template<> long* larger<long*>(long* a,  long* b);  // Specialization

int main() {
  cout << endl;
  cout << "Larger of 1.5 and 2.5 is "<< larger(1.5, 2.5) << endl;
  cout << "Larger of 3.5 and 4.5 is "<< larger(3.5, 4.5) << endl;

  int a_int = 35;
  int b_int = 45;

  cout << "Larger of "<< a_int << "and" << b_int <<" is"
       << larger(a_int, b_int)
       << endl;

  long a_long = 9;
  long b_long = 8;
  cout << "Larger of" << a_long << "and" << b_long <<" is"
       << larger(a_long, b_long)
       << endl;

  cout << "Larger of" << a_long << "and" << b_long << "is"
       << *larger(&a_long, &b_long)
       << endl;

return 0;
}

// Template for functions to return the larger of two values
template <class T> T larger(T a, T b) {
  cout << "standard version" << endl;
  return a>b ? a:b;
}

// Template specialization definitions
template <> long* larger<long*>(long* a,  long* b) {
  cout << "specialized version" << endl;
  return *a>*b ? a  :  b;
}
```

模板和说明都有一个语句，在调用它们时都会输出一个跟踪结果，说明在这两种情况下调用了哪个版本。本程序的结果如下所示：

```
standard version
Larger of 1.5 and 2.5 is 2.5
standard version
Larger of 3.5 and 4.5 is 4.5
standard version
Larger of 35 and 45 is 45
standard version
Larger of 9 and 8 is 9
specialized version
Larger of 9 and 8 is 9
```

例子的说明

在程序文件的开头，声明了模板说明，同时声明了模板：

```
template<> long* larger< long*>(long* a, long*b);        //Specialization
```

这是必须的，因为使用该说明的 main()在定义的前面。在程序中，说明的定义与前面讨论的一样——添加一个输出语句，以便于跟踪。

结果与预料的一样。前一个版本的程序中所有对 larger()的调用这里都有。最后一个调用如下所示：

```
cout << "Larger of" <<a_long <<"and "<<b_long <<" is "
     << *larger (&a_long, &b_long)
     << endl;
```

这个语句使用了模板的说明，得到了预期的结果。

9.2.4 函数模板和重载

重载从函数模板中生成的函数有不同的方式。一种方式前面提到过，就是用从模板生成的一个函数重载另一个函数。另外，还可以直接定义同名的其他函数，来重载函数。使用重载方式，可以为特定的情况定义重写版本，这些重写函数在使用时优先于模板。在这种情况下，每个重载的函数都必须有惟一的签名。

再看看前面的情形，即重载 larger()函数，以使用指针参数。这次不使用模板说明，而是显式声明一个重载函数。如果采用这种方法，就要用下面的重载函数原型代替程序示例 9.4 中的说明原型：

```
long* larger(long* a, long*b);        //overloaded function
```

这里没有使用程序示例 9.4 中的说明定义，而是添加了如下函数定义：

```
long* larger(long* a, long*b) {
  cout<<"overloaded version for long*"<<endl;
  return *a>*b ? a:b;
}
```

可以对程序示例 9.4 进行这样的修改，再次运行它。在用 long*参数调用函数 larger()时，编译器会确定已有一个合适的 larger()版本，而不使用模板。实际上，这个函数定义重写了模板。

还可以用一个模板重载另一个已有的模板。例如，扩展程序示例 9.4，添加一个重载的模

板，查找包含在一个数组中的最大值。该模板的定义如下所示：

```
template <class T> T larger(const T array[], int count) {
  cout<<"template overload version for arrays"<<endl;
  T result = array[0];
  for (int i=1; i<count; i++)
    if(array[i]>result)
       result = array[i];
  return result;
}
```

这个函数会指出任意类型的数组中的最大元素。在程序示例 9.4 中，给 main()添加如下语句：

```
double x[]={10.5, 12.5, 2.5, 13.5, 5.5};
cout << "Largest element has the value "
     << larger(x, sizeof x/ sizeof x[0])
     << endl;
```

当然，所添加的模板还需要一个原型。

9.2.5 带有多个参数的模板

前面使用了带有一个参数的函数模板，也可以在模板中使用多个参数。第二个类型参数的典型应用是提供控制函数模板中返回类型的方式。可以为函数 larger()定义另一个模板，允许独立于函数参数类型来指定返回类型：

```
// Template for functions to return the larger of two values
template <class TReturn, class TArg> TReturn larger(TArg a, TArgb) {
  return a>b ? a:b;
}
```

注意，因为编译器不能推断返回值的类型 TReturn，所以必须指定该类型。但是，因为编译器可以推断出参数的类型，所以只需指定返回类型。例如：

```
cout << "Larger of 1.5 and 2.5 is "
     << larger<int>(1.5, 2.5)
     << endl;
```

在尖括号中，返回类型指定为 int，参数类型则根据参数推断为 double。该函数调用的结果是 2。还可以指定 TReturn 和 TArg：

```
cout << "larger of 1.5 and 2.5 is "
     << larger<double, double>(1.5, 2.5)
     << endl;
```

编译器会创建一个函数，其参数类型是 double，返回类型也是 double。

显然，模板定义中的模板参数的顺序是非常重要的。如果在定义模板时，把返回类型定义为第二个参数，在函数调用中，就必须总是指定两个参数。如果只指定一个参数，该参数就解释为参数的类型，而返回类型则未定义。

非类型的模板参数

前面处理的所有模板参数都是数据类型。实际上，模板也可以有非类型的参数。在这种情况下，函数调用就使用非类型的参数。对应于非类型参数的变元必须是整型的，且在编译期间是常量，或者是带有外部链接的对象引用或指针。

在声明模板时，非类型的模板参数(和其他类型的参数一起)放在参数列表中。稍后介绍一个例子。非类型模板参数的类型可以是：

- 整型类型，如 int、long 等
- 枚举类型
- 对象类型的引用或指针
- 函数的引用或指针
- 类成员的指针

后两种还未介绍。本章的后面将介绍函数的指针，函数的引用和类成员的指针则在讨论类时论述。非类型模板参数应用于这些类型超出了本书的范围。这里仅举一个基本的例子，参数的类型是 int，介绍其工作原理。

假定需要一个函数检查值的作用域，就可以定义一个模板来处理各种类型：

```
template <class T, int upper, int lower> bool isin_range(T value) {
return (value<=upper) && (value>=lower);
}
```

对于这个模板，编译器不能从所使用的函数中推断出所有的模板参数。下面的函数调用不会编译：

```
double value=100.0;
std::cout<< is_in_range(value);                    //Won't compile - incorrect usage
```

这是因为参数 upper 和 lower 没有指定。要使用这个模板，必须指定模板参数值。正确的使用方式如下：

```
cout<< is_in_range <double,0,500>(value);  //OK - check 0 and 500
```

在本例中，上下限最好使用函数参数，而不应使用模板参数。毕竟，函数参数可以灵活地在程序运行时传送需要计算的值，而本例必须在编译期间提供上下限。

9.3 函数指针

指针存储了一个地址值。前面都是使用指针存储另一个与指针具有相同数据类型的变量的地址。这就允许通过指针在不同的情况下使用不同的变量。

指针也可以指向函数的地址。在程序执行过程中，这种指针可以在不同的时候指向不同的函数。在程序中，总是可以使用指针来调用函数，被调用的函数地址是最近赋予指针的。

函数指针必须包含要调用的函数的内存地址。为了工作正确，指针还必须包含其他信息，即指针所指向的函数的参数列表中的参数类型，以及返回类型。因此，在声明函数指针时，必须指定该指针可以指向的函数的参数类型和返回类型，以及指针名。

函数指针声明中需要的信息限制了指针可以指向的函数个数。这类似于存储数据项地址的指针。例如，指向类型 int 的指针只能指向包含 int 类型的数值的位置——如果要存储 long 类型的数值的地址，就需要另一种指针。

同样，假定要声明一个函数指针，该函数接受一个 int 类型的参数，返回 double 类型的值。这个指针只能用于存储这种形式的函数地址。如果要存储另一个函数的地址，该函数接受两个 int 类型的参数，返回类型是 char，就必须定义另一个有这些特性的指针。基本上，函数指针的唯一可变的特性是函数名称。对同一个函数指针来说，参数的个数和类型以及返回类型总是相同的。

提示：

函数指针在 C++中没有 C 中那样普遍。这是因为 C++提供了其他能完成类似任务的功能(类、函数重载等)。但是，函数指针在 C++语言中仍有重要的地位。

9.3.1 声明函数指针

下面声明一个指针 pfun，用于指向一个函数，该函数带有两个参数，其类型分别是 long* 和 int，其返回值的类型是 long。该声明如下所示：

```
long (*pfun)(long*, int);      //Pointer to function declaration
```

这初看起来有点怪异，因为所有的实体都加上了括号。

提示：

指针名 pfun 的括号和星号是必须有的，没有它们，这个语句声明的就是函数，而不是指针，因为*会先加在 long 上，而不是 pfun 上。

声明函数指针的一般形式如下所示：

返回类型(*指针名)(参数类型列表);

指针名上的括号是必不可少的，否则这就变成一个函数原型了。指针只能指向具有声明中指定的返回类型和参数类型列表的函数。

把该声明按顺序分解为三个部分：

- 所指向的函数的返回类型
- 放在括号中的指针名：前面有一个星号，表示这是一个指针
- 放在括号中的参数类型列表

注意：

如果试图把函数赋予不遵循指针声明中类型的指针，编译器就会生成一个错误消息。

当然，在声明指针时，应总是初始化它。可以在指针的声明中，把函数指针初始化为函数名。假定函数的原型如下所示：

```
long max_element(const long* array, int count);    //Function prototype
```

接着，声明并初始化函数的指针，如下面的语句所示：

```
long (*pfun)(long*, int)= max_element;
```

指针初始化为指向函数 max_element()。该指针以后还可以设置为指向具有相同参数和返回类型的其他函数。

当然，也可以使用赋值语句来初始化函数指针。假定指针 pfun 象前面那样声明，就可以用下面的语句把该指针的值设置为指向另一个函数，如下所示：

```
long min_element(const long* array, int count);    //Function prototype
pfun = min_element;                                //Set pointer to function min_element()
```

指针和变量一样，在使用指针调用函数之前，必须先初始化函数指针。如果没有进行初始化，程序就会失败。

提示：

可以把函数指针初始化为 0。但是，如果指针指向 0，则在使用它调用函数时，其行为就是不确定的，也可能导致程序失败。

如果要使用指针 pfun 调用函数 min_element()，只需使用指针名，就好像它是函数名一样。例如，可以使用下面的语句：

```
long data[]={23,34,22,56,87,12,57,76};
std::cout << "value of mimimum is "
          << pfun(data, sizeof data/sizeof data[0]);
```

这个语句会输出数组 data 中的最小元素值。

注释：

函数模板和函数指针的概念可以用一种古怪的方式连接起来。在某种意义上，函数指针是函数模板的反面。函数指针可以指向名称不同，但参数和返回类型相同的一组函数中的任何一个函数。反过来，函数模板定义了一组名称相同、但参数类型和返回类型不同的函数。

程序示例 9.5——函数指针

为了正确理解这些新奇的指针，以及它们的操作过程，下面看一个程序：

```
// Program 9.5 Exercising pointers to functions
#include <iostream>
using std::cout;
using std::endl;

long sum(long a,  long b);                          // Function prototype
long product(long a,  long b);                      // Function prototype

int main ( ) {
long  (*pDoIt)(long,  long)  = 0;          // Pointer to function declaration

   pDo_it = product;
   cout << endl
        << "3*5 =" << pDo_it(3,  5);    // Call product thru a pointer

   pDo_it = sum;                          // Reassign pointer to sum()
```

```
    cout << endl
         << "3 *  (4+5)  + 6 ="
         << pDo_it(product(3, pDo_it(4,  5)),  6);      // Call thru a pointer twice

   cout << endl;
   return 0;
}

// Function to multiply two values
long product(long a,  long b) {
   return a*b;
}

// Function to add two values
long sum(long a, long b) {
    return a+b;
}
```

这个例子的结果如下所示：

```
3*5=15
3*(4+5)+6=33
```

例子的说明

这并不是一个有用的程序，但它展示了函数指针的声明、赋值和调用函数的过程。

在通常的前置语句之后，声明并初始化了 pDo_it，这是一个函数指针，它可以指向前面定义的两个函数 sum()或 product()。

```
long (*pDo_it)(long, long)=0;          //Pointer to function declaration
```

把 pDo_it 初始化为 0，但如果指针指向函数，就只能调用函数。因此把函数 product()的地址赋予 pDo_it：

```
pDo_it=product;
```

在初始化指针时，应提供指针所指向的函数名，不需要括号或其他修饰符。函数名自动转换为地址，存储在指针中。

接着，在输出语句中，通过指针 pDo_it 间接调用函数 product()：

```
cout << endl
     << "3*5="<<pDo_it(3,5);          //Call product thru a pointer
```

指针名使用起来就像是函数名一样，其后是放在括号中的参数，传送参数的方式与直接使用了原来的函数名一样。

说明了可以做的工作之后，修改指针，使之指向函数 sum()。然后在一个非常复杂的表达式中使用它进行一些简单的算术计算，从而说明函数指针与它指向的函数可以以相同的方式使用。表达式中的操作序列如图 9-2 所示。

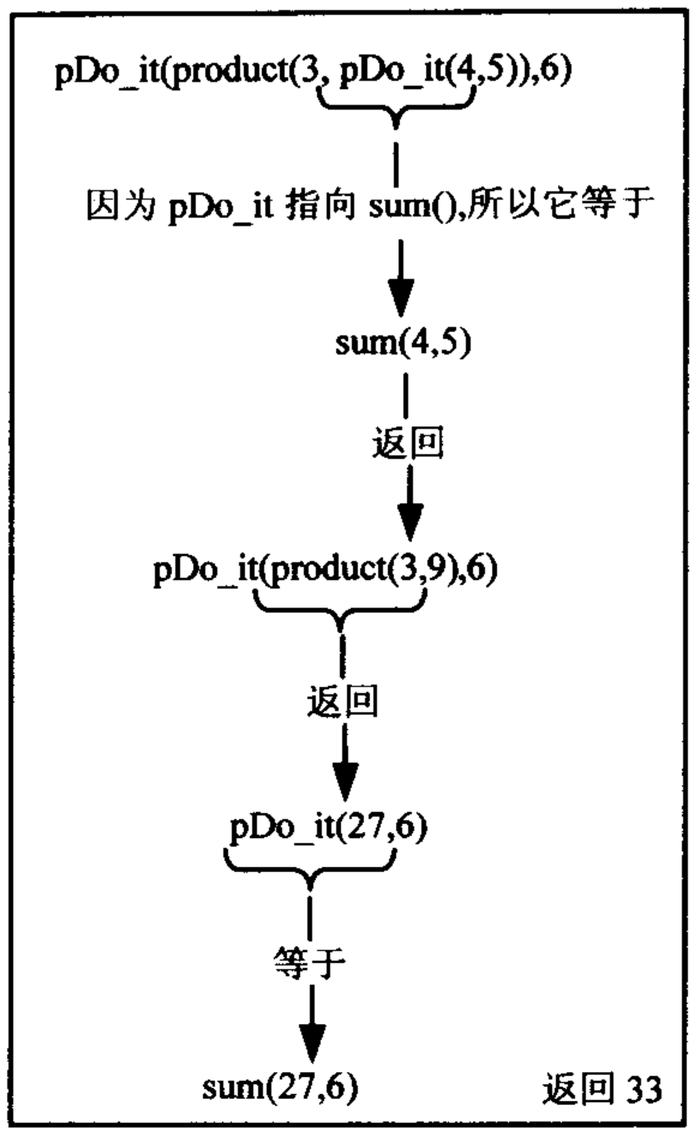


图 9-2　使用函数指针计算表达式

9.3.2　把函数作为参数传送

实际上，函数指针是一种非常好的类型。因此，可以编写一个函数，它的一个参数是函数指针。然后，在(外部)函数使用其函数指针参数时，就间接地调用在调用函数时对应参数指向的函数。

由于指针在不同的情况下可以指向不同的函数，因此允许调用程序确定要从外部函数中调用哪个函数。

在用函数指针类型的参数调用函数时，参数可以是包含函数地址的相应类型的指针。还可以把函数名作为参数，显式传送函数。作为参数传送给另一个函数的函数有时称为回调函数。

程序示例 9.6——传送函数指针

下面用一个例子来说明传送函数指针。假定需要一个函数，它处理一组数字，在一些情况下把每个数字的平方加在一起，在其他情况下则把每个数字的立方加在一起。为此，可以把函数指针用作一个参数。

```
// Program 9.6 A pointer to a function as an argument
#include <iostream>
using std::cout;
using std::endl;

// Function prototype
double squared(double);
double cubed(double);
double sum_array(double array[], int len, double (*pfun) (double));

int main () {
```

```
double array[]  = { 1.5, 2.5,  3.5, 4.5, 5.5,  6.5, 7.5 };
int len = sizeof array/sizeof array[0];

cout  << endl
      << "Sum  of  squares  =  "
      << sum_array(array, len, squared)
      << endl;

cout  <<  "Sum of cubes ="
      << sum_array(array,  len, cubed)
      << endl;
return 0;
}

// Function for a square of a value
double squared(double x) {
    return x*x;
}

// Function for a cube of a value
double cubed(double x) {
   return x*x*x;
}

// Function to sum functions of array elements
double sum_array(double array[] ,int len, double (*pfun) (double)) {
  double total = 0.0;       // Accumulate total in here

  for(int i = 0 ; i < len ; i++)
     total += pfun(array[i]);
  return total;
}
```

这个例子的结果如下所示：

```
Sum of squares = 169.75
Sum of cubes = 1015.88
```

例子的说明

我们感兴趣的第一个语句是函数 sum_array()的原型，它的第三个参数是一个函数指针，该函数的参数类型是 double，返回值的类型也是 double：

```
double sum_array(double array[], int len, double (*pfun)(double));
```

函数 sum_array()使用其第三个参数指向的函数，处理作为第一个参数传送的数组中的每个元素。返回所处理的数组元素的和。

函数 sum_array()在 main()中调用了两次。第一次调用时，最后一个参数是 squared。第二次调用时，最后一个参数是 cubed。在这两次调用中，都把对应于函数名的地址用作一个参数，并替换 sum_array()函数体中的函数指针，从而在 for 循环中调用相应的函数。

很容易获得这个例子的结果，但使用函数指针具有更大的一般性。可以把任何已定义的函数传送给 sum_array()，只要该函数带有一个 double 参数，返回 double 类型的值即可。

9.3.3 函数指针的数组

可以声明函数指针的数组，这类似于声明一般的指针数组。还可以在声明数组时，初始化函数指针的数组。声明指针数组的例子如下所示：

```
double sum(double, double);                    //Function prototype
double product(double, double);                //Function prototype
double difference(double, double);             //Function prototype
double (*pfun[3])( double, double) =
          { sum, product, difference };   //Array of function pointers
```

数组中的每个元素都用相应的函数地址(显示在花括号中的初始化列表)初始化了。为了使用数组指针的第二个元素调用函数 product()，可以使用下面的语句：

```
pfun[1](2.5,3.5);
```

方括号选择函数指针数组中的元素，放在数组名的后面，要调用的函数参数之前。当然，可以在允许使用原函数的表达式中，通过函数指针数组中的元素来调用函数。选择指针的索引值可以是结果为有效索引值的任何表达式。

9.4 递归

在本章的最后，深入讨论一下递归函数。本节的最后将使用递归函数，重新设计第 7 章介绍的排序问题的解决方法。

C++中的函数允许在合适的情况下调用它本身。在函数包含自己的调用时，该函数就称为递归函数。递归函数的调用也可以是间接的，例如，函数 fun1()调用另一个函数 fun2()，fun2()又调用了函数 fun1()。

递归看起来似乎是一个无限循环，如果不小心，就容易变成无限循环。避免无限循环的一个先决条件是函数包含停止该过程的机制。

什么事件可以用于递归？最好看看以前介绍过的技术，否则答案不是很明显。以树型结构组织的数据就是一个递归的例子，如图 9-3 所示，其中显示了一个树，包含可以看做是子树的结构。描述机械装配件如汽车的数据就常常组织为一个树。汽车由许多子装配件组成，例如车体、引擎、传输系统和悬挂系统。这些子装配件又由其他子装配件和部件组成，最后，树的叶子是没有进一步内部结构的组件。

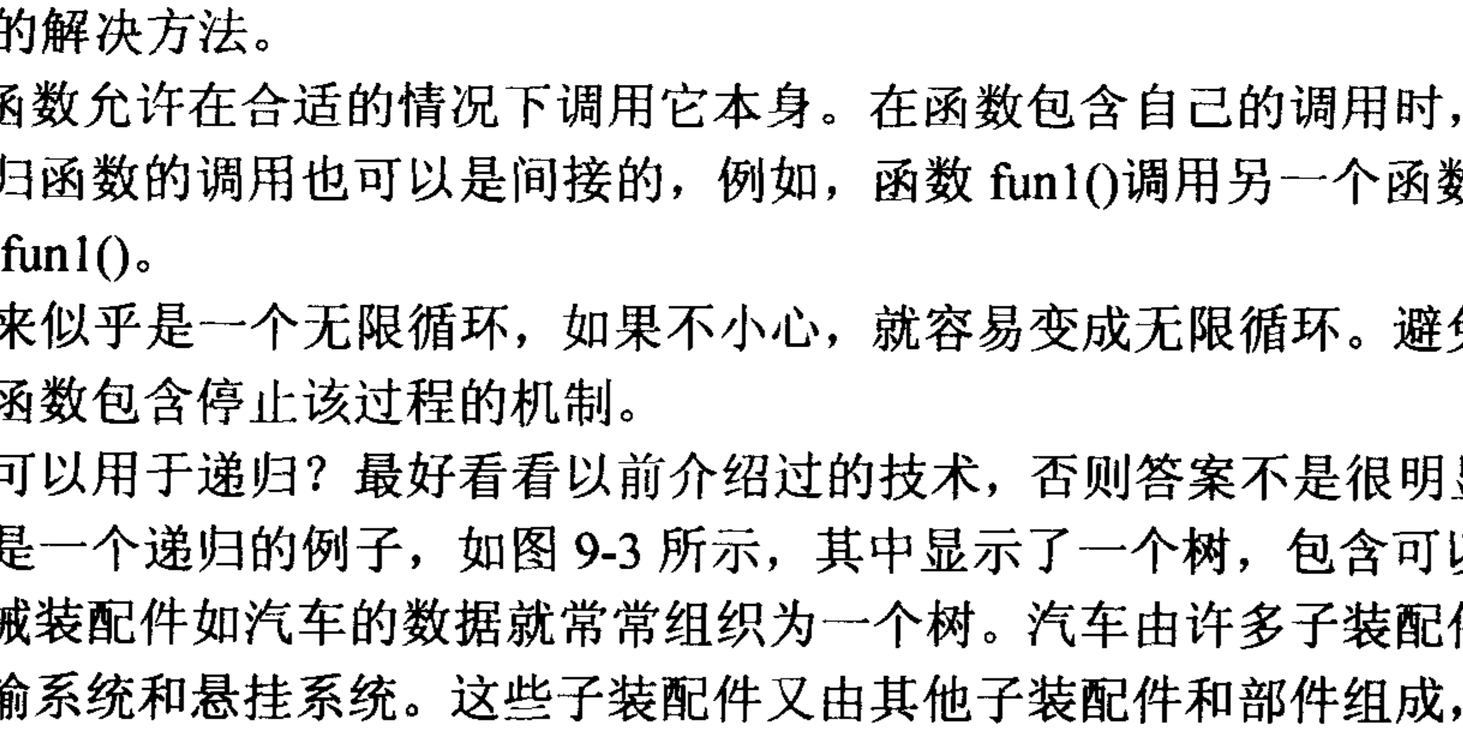


图 9-3　树结构的例子

在需要处理组织为树的数据时，树常常可以使用递归有效地遍历。树的每个分支都可以看作子树，所以访问树中每个数据项的函数在遇到分支节点时，就可以调用它本身。在遇到一个数据项时，函数就对它进行必要的处理，再返回调用点。因此，在函数找到树的叶子节点即数据项时，就提供函数停止递归调用的方式。

在物理和数学中，有许多事件都涉及到递归。一个简单的例子就是正数的阶乘，因为正数的阶乘(写作 n!)就是 n 个事务安排的一种方式。对于给定的整数 N，其阶乘是乘积 1×2×3.....×N。为了计算这个乘积，可以定义如下的递归函数：

```
long factorial(long n) {
  if(n==1L)
    return 1L;
  return n*factorial(n-1);
}
```

如果用变元 4 来调用这个函数，就执行 else 子句，用值 3 调用该函数。如果使用这个过程，最终会用变元 1 调用 factorial()函数，它会返回 1，再对 1 乘以 2，依次类推，直到第一个调用返回值 1×2×3×4 为止。这个例子常常用于说明递归的操作过程。但是，下面介绍一些更简单的递归。

程序示例 9.7——递归函数

在第 8 章的开始(程序示例 8.1)，编写了一个函数，计算一个值的整数幂，即计算 x^n。对于正数 n，这个计算过程就是把 1 乘 x 重复 n 次，对于负数 n，就是把 1 除 x 重复 n 次。如果 n 是 0，结果就是 1。下面用一个递归函数来实现这一计算，以演示递归的操作过程。

```
// Program 9.7 recursive version of x to the power n
#include <iostream>
#include <iomanip>
using std::cout;
using std::endl;

double power(double x,  int n);

int main() {
   cout << endl;

   // Calculate powers of 8 from -3 to +3
   for( int i =-3 ; i <= 3 ; i++)
      cout << std::setw(10) << power(8.0,  i);

   cout << endl;
   return 0;
}

// Recursive function to calculate x to the power n
double power(double x,  int n) {
   if(0 == n)
      return 1.0;
   if(0 < n)
      return x*power(x, n-1);
```

```
    return 1.0/power(x, -n);
}
```

函数 main()与上一个版本一样，结果也是一样的：

```
0.00195313    0.015625    0.125    1    8    64    512
```

例子的说明

如果n是0，第一个if语句就返回值1.0。对于正数n，下一个if语句返回表达式x*power(x,n-1)的结果。这个表达式将进一步调用函数 power()，但索引值要减 1。如果在这个调用中，n 仍是正数，就继续调用 power()，但索引值要再减 1。函数的每次调用和调用的参数都在调用堆栈中记录下来。这个过程一直重复下去，直到 n 是 0 为止，此时返回 1，并把前面调用的结果依次释放，在每次调用中都是乘以 x。实际上，对于大于 0 的给定值 n，函数会调用它本身 n 次。这个机制如图 9-4 所示，该图假定索引参数 n 的值为 3。

可以看出，为生成 x^3，power()函数一共调用了 4 次。

对于负数 n，要计算 x^n，也会使用相同的过程。

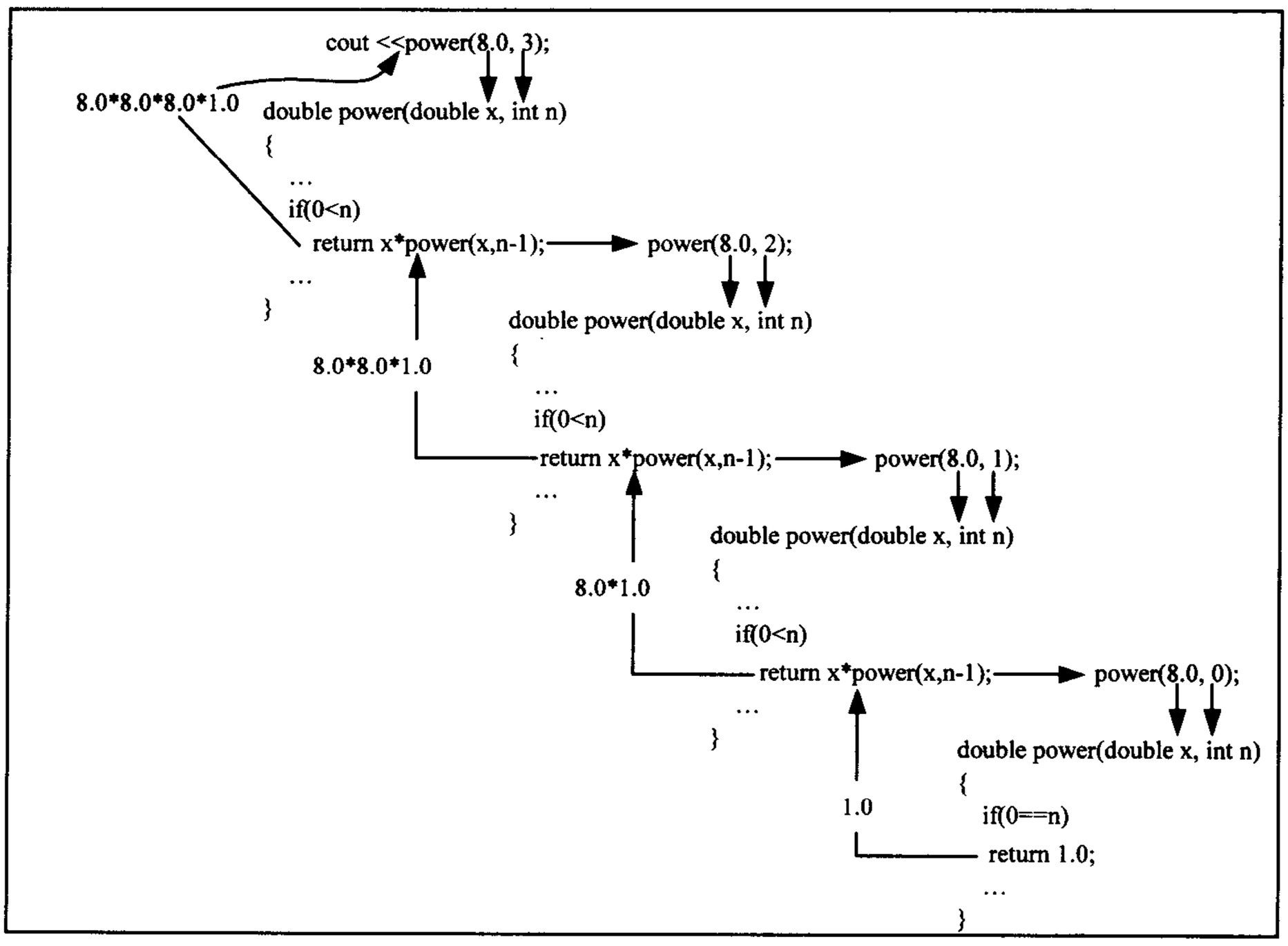


图 9-4　执行 power()函数的递归调用

有时，使用条件运算符可以缩短 power()函数的代码。该函数体可缩减为一行代码：

```
double power(double x, int n) {
  return 0==n?1.0 : (0>n?1.0/power(x,-n):x*power(x,n-1));
}
```

这不会改善操作，而且代码也不是很容易理解。实际上，递归调用过程与循环相比的效率非常低。每次函数调用都涉及到许多操作，包括获取和存储返回的地址，复制每个参数，处理传送给被调用函数的参数。使用循环来实现函数 power()，而不是用递归调用，就会快得多：

```
double power(double x, int n) {
  if(0==n)
    return 1.0
  if(0>n) {
     x = 1.0/x;
     n = -n;
  }

  double result=x;
  for(int i=1;i<n;i++)
    result *= x;
  return result;
}
```

使用递归

除非必须使用递归函数才能解决问题，或者没有更好的方法，否则一般最好采用其他方式，例如循环。这要比使用递归函数调用高效得多。还需要注意解决问题所必须的递归深度本身并没有问题。例如，如果函数要调用它本身一百万次，存储变元值副本和返回地址所需的内存量就是非常可观的。

但有时，使用递归可以大大简化编码，这种简化有时能抵消效率方面的损失。

实现排序和合并操作常常是演示递归的一个好例子。排序和合并一组数据是一个递归过程，在这个过程中，要对原数据中的小子集重复应用同一个算法。下面看一个例子。

实现递归排序

在程序示例 7.9 中，我们从一些输入文本中提取单词，并使用一个指针数组，按首字母的升序方式对它们排序。这里要实现一个递归函数，用一个著名的排序算法，即 Quicksort 来对单词排序。这是一个古老的方法，已有 40 多年的历史了，但仍是最快的排序方法。

要采用 Quicksort 算法来对单词排序，首先从单词集合中任意选择一个单词，例如中间的一个单词。然后安排剩余单词的位置，把所有小于所选单词的单词放在所选单词的左边(不必按顺序)，把所有大于所选单词的单词放在所选单词的右边(也不必按顺序)。图 9-5 演示了这个过程。

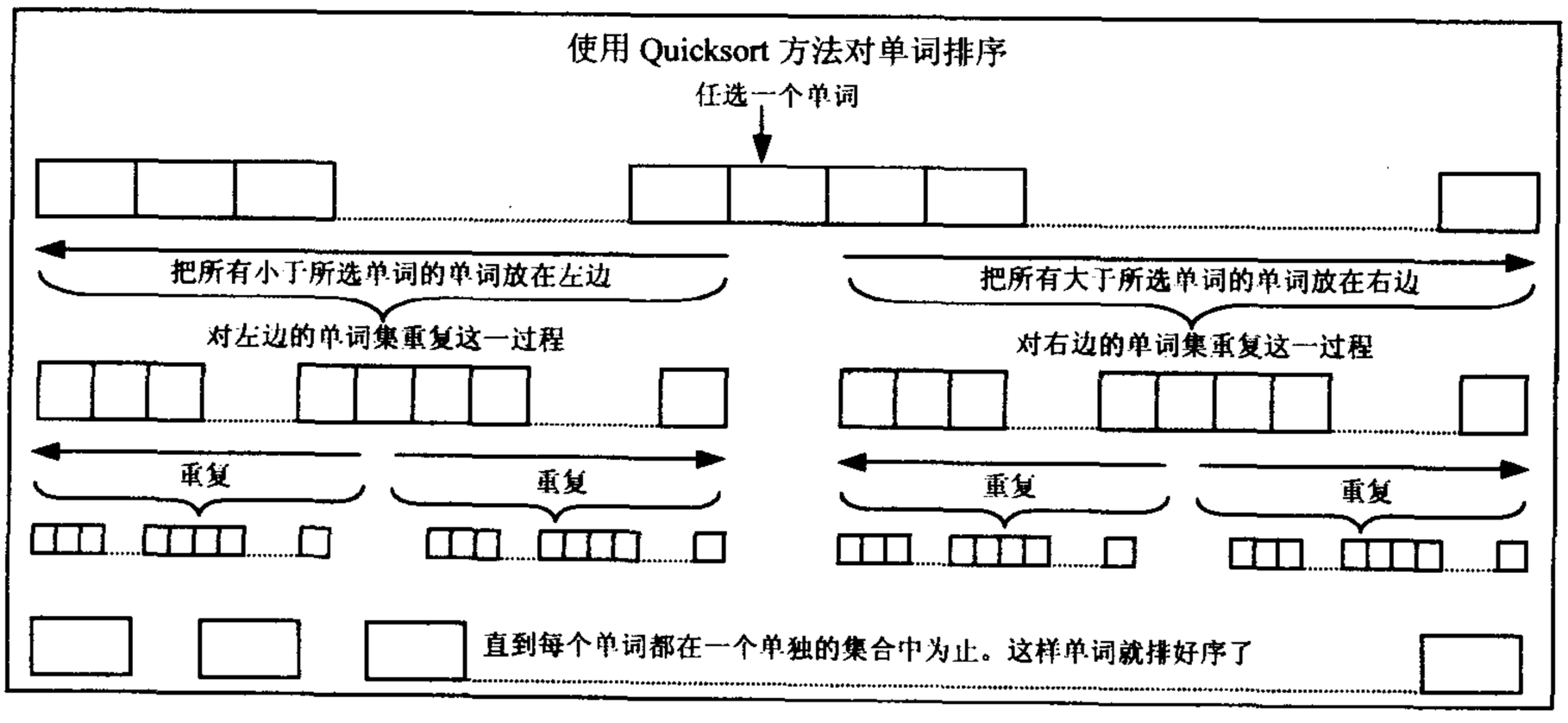


图 9-5　Quicksort 算法

执行了一次这个过程之后，对两个得到的单词子集重复这个过程，生成 4 个单词子集。再重复这个过程，直到每个单词都位于一个独立的集合中为止。这样单词就按升序排列好了。我们对集合的子集重复同一个过程，达到了最终的目标。这说明可以使用递归实现这个过程。

当然，为此应重新安排数组中的地址，而不是移动单词。图 9-6 中的方法是把单词的地址分为两组，并在原数组中存储重新安排的地址。

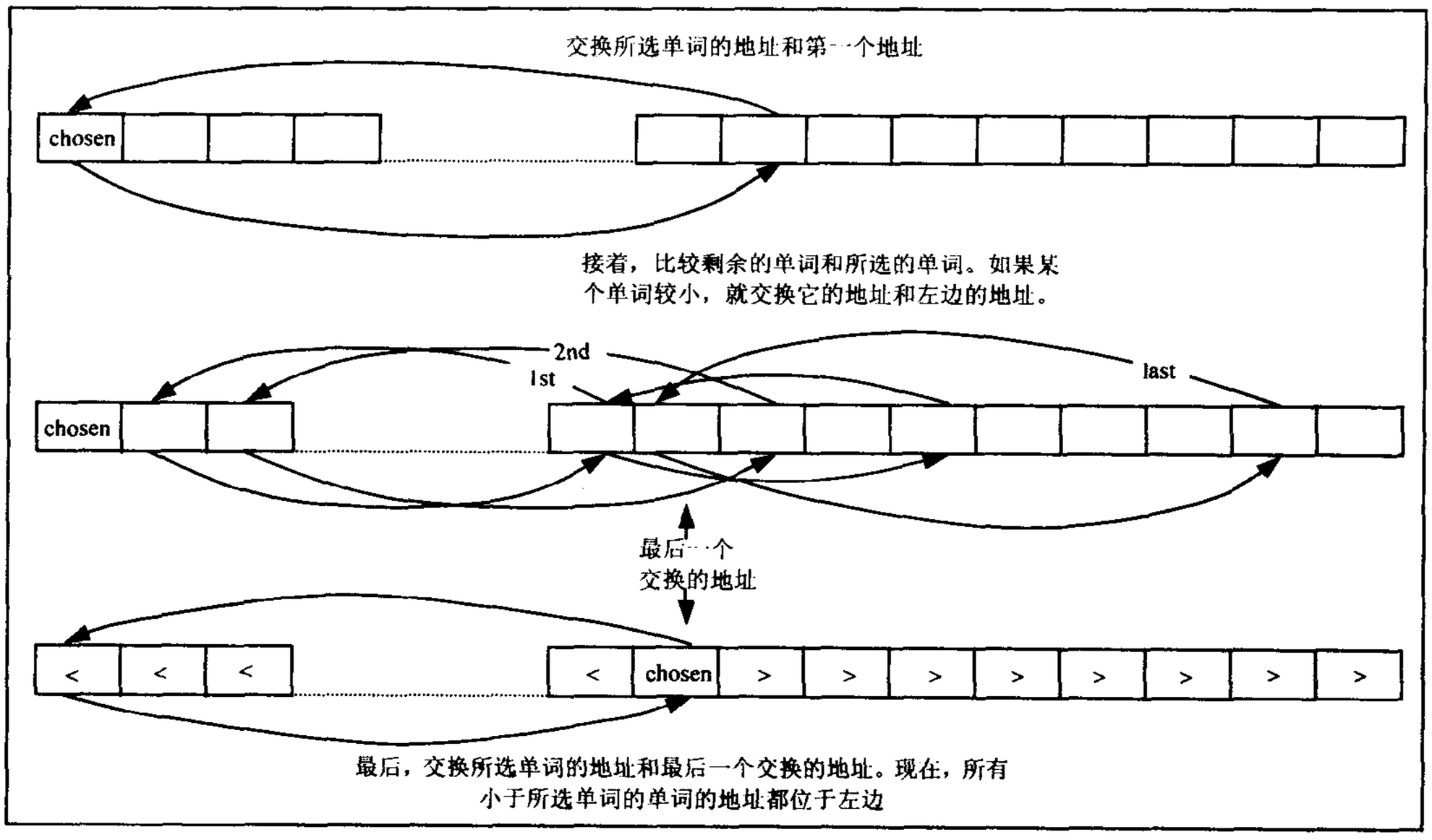


图 9-6　Quicksort 算法中的交换地址

需要交换数组中几个地方的地址，为此，最好编写一个函数：

```
// Swap address at position first with address at position second
void swap(string* pStr[],  int first,  int second) {
   string* temp = pStr[first];
   pStr[first]  = pStr[second];
   pStr[second]  = temp;
}
```

使用这个函数，以图 9-5 中的方式重新安排地址数组中的元素，就可以实现 Quicksort 方法。排序算法如下所示：

```
// Sort strings in ascending sequence
// Addresses of words to be sorted are from pStr[start] to pStr[end]
void sort(string* pStr[],  int start,  int end) {
// start index must be less than end index for 2 or more elements
if(!(start<end))
   return;                          // Less than 2 elements - nothing to do

// Choose middle address to partition set
swap(pStr,  start,  (start+end)/2);     // Swap middle address with start
```

```
// Check words against chosen word
int current = start;
for(int i = start+1;  i<=end ;  i++)
  if(*(pStr[i])  < *(pStr[start]))     // Is word less than chosen word?
     swap(pStr,  ++current,  i);       // Yes,  so swap to the left


  swap(pStr,  start,  current);        // Swap the chosen word with last in

  sort(pStr, start, current-1);       // Partition the left set
  sort(pStr, current+1,end);          // Partition the right set
}
```

函数 sort()使用了三个参数：要排序的地址数组 pStr 以及集合中第一个和最后一个地址的索引位置。函数第一次调用时，start 是 0，end 是最后一个数组元素的索引。在后续的递归调用中，仅使用了一部分数组元素，所以在许多情况下，start 和 end 包含的是内部索引位置。

函数 sort()的执行步骤如下所示：

1. 递归函数可能陷入无限循环，因此函数首先要检查是否停止递归函数调用。如果集合中少于两个元素，就不能再分解了，因此应返回。对于每个递归调用，都把集合分解为两个更小的集合，这样，最终得到的集合就必须只包含一个元素，或不包含元素。

2. 接着，交换集合中间的地址与索引位置为 start 的第一个地址。for 循环比较所选单词和 start 后面的地址所指向的单词，对于小于所选单词的单词，其地址要与 start 之后的地址交换：第一个地址变成 start+1，第二个地址变成 start+2，依此类推。循环结束时，current 包含这些已交换地址的单词中最后一个地址的索引位置。所选单词的地址仍在 start 位置上，此时交换所选单词的地址和 current 的地址。结果，小于所选单词的所有单词地址就都在 current 的左边，大于所选单词的所有单词的地址就都在 current 的右边。

3. 前面分解了单词集合，现在对这个集合排序就很容易了。只需对已生成的两个子集排序即可，为此，对每个子集调用 sort()函数。小于所选单词的所有单词地址从 start 排到 current - 1，大于所选单词的所有单词地址从 current+1 排到 end。

使用递归方法会使代码很容易理解，而且代码也比较短，没有使用循环来排序那么复杂。但循环比较快。

程序示例 9.8——单词的递归排序

下面创建程序示例 7.9 的另一个版本，该版本使用刚才编写的 sort()函数。同时，把该程序分解为函数，会使程序更容易维护。在 main()的原版本中有三个代码块，可以用不同的函数来实现：计算单词的总数，创建对应于单词的 string 对象和按顺序显示单词。

第一个函数计算输入文本中的单词总数，它需要访问原文本和包含分隔符字符的 string 对象。它把单词总数返回为 int。这个函数的原型如下所示：

```
int count_words(const string& text, const string& separators);
```

两个参数都应是 const 引用，因为函数不需要修改它们。这个函数生成的单词总数用于在自由存储区中创建对应大小的指针数组。

下一个函数从输入中提取文本，它需要把地址数组作为一个参数，也需要输入文本和分隔符，其原型如下所示：

```
void extract_words(string** pStr[], const string& text,
                                        const string& separators);
```

显然，数组参数不能声明为 const，因为函数要在该数组中存储从 new 中获取的地址。该函数不需要访问单词总数，因为提取过程不需要它。

最后一个函数输出单词，它需要把指针数组和单词总数作为参数访问，其原型如下所示：

```
void show_words(string** pStr[], int count);
```

现在把这三个函数和刚才开发的 sort()、swap()函数都放在 main()的定义中：

```
// Program 9.8 Sorting strings recursively
#include <iostream>
#include <string>
using std::cout;
using std::cin;
using std::endl;
using std::string;

// Function prototype
void swap(string* pStr[],  int first,  int second);
void sort(string* pStr[],  int start,  int end);
int count_words(const string& text, const string& separators);
void extract_words(string* pStr[], const string& text, const string& separators);
void show_words(string* pStr[],  int count);

int main() {
   string text;                                    // The string to be sorted
   const string separators ="  ,.\"\n";            // Word delimiters

   // Read the string to be searched from the keyboard
  cout << endl <<  "Enter a string terminated by #:"  << endl;
  getline(cin,  text,  '#');

  int word_count = count_words(text,  separators); // Get count of words

  if(0 == word_ count) {
    cout << endl <<  "No words in text."  << endl;
    return 0;
  }

  string** pWords = new string*[word_count];      // Array of pntrs to the words

  extract_words(pWords,  text,  separators);
  sort(pWords,  0, word_count-1);                 // Sort the words
  show_words(pWords, word_count);                 // Output the words

  // Delete words from free store
  for(int i = 0 ; i<word_count ;  i++)
    delete pWords[i];

  // Now delete the array of pointers
```

```
    delete[] pWords;

    return 0;
}
```

要完成该程序，除了 sort()和 swap()之外，只需定义三个新函数。下面是 count_words()函数的代码：

```
//Function to count the words in the text
int count_words(const string& text,  const string& separators) {
    size_t start = text.find_first_not_of(separators);    // Word start index
    size_t end = 0;                                       // End delimiter index
    int word_count = 0;                                   // Count of words stored
    while(start  != string::npos) {
       end = text.find_first_of(separators,  start+1);
       if(end == string::npos)                            // Found one?
          end = text.length();                            // No,  so set to last+1
      word_count++;                                       // Increment count

      // Find the first character of the next word
      start = text.find_first_not_of(separators, end+1);
   }
    return word_count;
}
```

这与原来的代码非常类似，只是将它打包在一个函数中。extract_words()函数的代码如下所示：

```
//Function to extract words from the text
void extract_words(string* pStr[],
                           const string& text, const string& separators) {
   size_t start = text.find_first_not_of(separators); // Start 1st word
   size_t end = 0;                                    // Index for the end of a word
   int index = 0;                                     // Pointer array index

      while(start  != string::npos) {
      end = text.find_first_of(separators,  start+1);      // Find end separator
       if(end == string::npos)                             // Found one?
          end = text.length();                             // No,  so set to last+1
      pStr[index++]  = new string(text.substr(start, end-start));
         start = text.find_first_not_of(separators, end+1); // Find next word
   }
}
```

输出单词时，把以同一个字母开头的单词放在一组中，一行显示五个单词，于是，实现 show_words()函数的代码如下所示：

```
// Function to Output the words
void show_words(string* pStr[], int count) {
   const int words_per_line = 5;                    // Word_per_line
   cout << endl <<"   "<< *pStr[0];                 // Output the first word
int words_in_line = 0;                             // Words in the current line
```

```
for(int i = 1 ;  i<count ;  i++){                    // Output remaining words
   // Newline when initial letter changes or after 5 wrds per line
   if((*pStr[i])[0]  != (*pStr[i-1]) [0]  ||
                              words_in_line++ == words_ per_line) {
      words _in_line = 0;
      cout << endl;
   }
   cout <<  "   " << *pStr[i];                  // Output a word

}
cout << endl;
   cout << endl;
}
```

如果把这些函数组装成一个完整的程序，就成功地把程序分解为几个函数。运行这个程序，它的工作方式与第 7 章的版本完全相同，但代码用函数提供的结构就更容易理解。如果需要，程序的修改也比较容易。这里使用一个数组在文本中查找两遍单词，因为需要确定指针数组中需要多少个元素。还可以采用其他方法来达到这一目的，而无需知道单词的总数。一种方法就是在链接列表中存储地址，然后遍历一次文本，创建单词对象，把它们添加到列表中。

9.5 本章小结

读者现在已全面了解了编写和使用函数的知识。在第 11 章论述用户定义的类型时，还将讨论函数。

本章的主要内容如下：

- 重载函数是名称相同、但参数列表不同的函数。重载函数不能仅通过返回类型来区分。
- 函数签名由函数名和参数和个数及类型来确定。在调用重载函数时，编译器会检查函数签名，把它与可用的函数作比较，然后选择合适的函数。
- 函数模板是自动生成重载函数的一种方法。
- 函数模板有一个或多个参数，这些参数通常是类型变量，也可以是非类型的变量。函数模板的实例，即函数定义，由编译器为每个对应于一组惟一模板参数的函数调用创建的。
- 函数模板可以用其他函数或函数模板来重载。
- 函数指针存储了函数的地址，以及参数的类型和个数、返回类型等信息。
- 可以使用函数指针来存储有对应返回类型、参数类型和个数的任一函数地址。
- 可以使用函数指针来调用它包含的地址上的函数，还可以把函数指针作为函数参数来传送。
- 递归函数是调用它自身的函数。采用递归方式实现算法有时可以得到非常简明的代码，但与实现同一算法的其他方法相比，采用递归方式常常需要更多的执行时间。

9.6 练习

1. 创建一个函数 plus()，它把两个数值加在一起，返回它们的和。提供处理 int、double 和

string 类型的重载版本，测试它们是否能处理下面的调用：

```
int n=plus(3,4);
double d=plus(3.2,4.2);
string s= plus("he","llo");
string s1="aaa"; string s2="bbb";
string s3= plus(s1,s2);
```

给 string 版本的函数传送参数的最有效方式是什么？

为什么下面的调用不工作？

```
d=plus(3,4.2);
```

2. 把函数 plus()变成一个模板，测试它是否能用于处理数值类型。该模板是否能处理第 1 题中的语句(plus("he","llo"))？为什么？提出问题的一种解决方法。

3. 标准库提供了三角函数 sin()、cos()和 tan()，这些函数都带有一个 double 参数，返回一个 double 值。要使用它们，需要包含标准库头文件<cmath>。编写一个函数 calc()，它带有两个参数：一个 double 值和一个指向三角函数的指针，返回把函数应用于 double 值的结果。编写一个程序，测试所编写的函数。如果该函数通过了测试，就建立一个函数指针数组，存储这三个三角函数，并测试它们。

4. 有一个递归函数，称为 Ackerman 函数，它广泛应用于计算机科学和数学，其定义如下：

如果 m 和 n 是整数，若 n>=0，且 m>=0，则：
如果 m==0，则 ack(m,n)=n+1
如果 n==0 且 m>0，则 ack(m,n)=ack(m-1,1)
如果 m>0，且 n>0，ack(m,n)=ack(m-1,ack(m,n-1))

编写一个程序，递归计算 Ackerman 函数，假定 n 为 0 到 5 之间的值，m 为 0 到 3 之间的值，进行测试。这个函数的一个特性是，如果 m 和 n 有小幅度的增加，递归的次数(或深度)就会有非常大的增加，所以不要递归计算 n>8 或 m>3 的情形，实际上一台计算机没有足够的能力计算这种情形。

第 10 章　程序文件和预处理器指令

本章将介绍多个程序文件和头文件如何交互，以及如何管理和控制程序文件的内容。

目前还没有讨论类这个重要的主题。第 11 章将开始定义自己的数据类型，这是介绍类的开始。本章的内容对如何定义自己的数据类型有一些提示，并对这些提示进行详细讨论。

本章主要内容

- 头文件和程序文件如何相互关联
- 转换单元的概念
- 链接的概念及其重要性
- 命名空间的概念，创建和使用它们的方式
- 预处理器的概念，如何使用可用的预处理器指令
- 调试中的基本概念，调试可以从预处理器和标准库中获得的内容

10.1　使用程序文件

第 1 章讨论了 C++程序一般由多个文件组成。下面复习一下。C++程序包含两大类文件：

- 头文件：这种文件一般用文件扩展名.h 来标识(注意一些老系统使用.hpp)。这些文件包含类型定义和其他用于程序中一个或多个源文件的代码。
- 源文件：其扩展名通常是.cpp，也可以使用.c、.cxx 或其他扩展名。这种文件包含要编译成机器指令的代码——主要是函数定义。需要的头文件通过#include 指令添加到源文件中。

前面说过，在 ANSI C++中，库函数的标准头文件(如<iostream>)没有扩展名，所以#include 指令是否用于标准头文件是很明显的。在#include 指令中使用特殊的表示法，即把标准库头文件包含在尖括号中：

```
#include <iostream>
```

注意：

在尖括号中不应添加空格，否则头文件名是不能识别的。

当然，还有其他类型的文件支持编程环境(定义某类资源)，但.h 和.cpp 文件包含了所有的 C++代码。如图 10-1 所示。

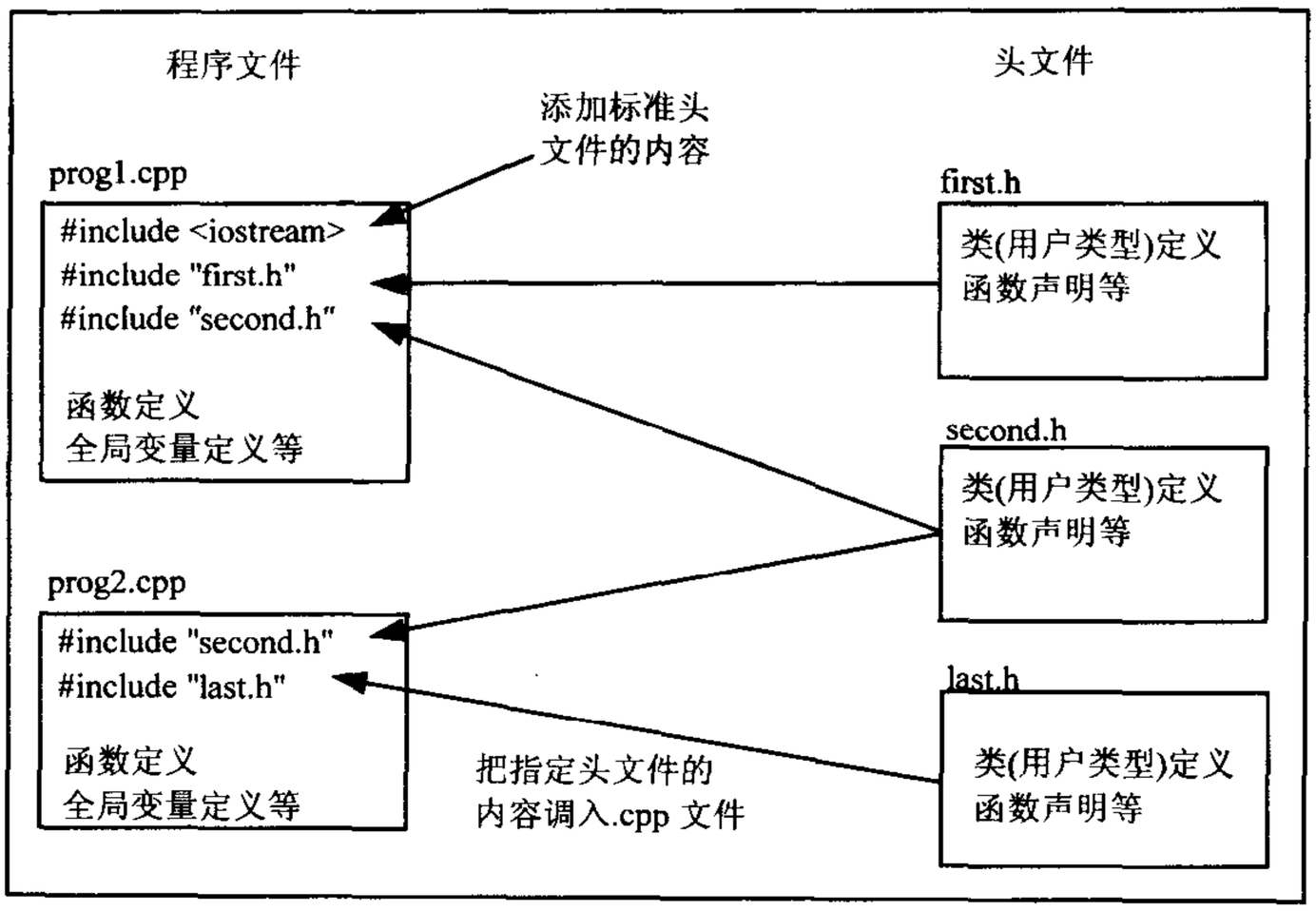


图 10-1　由.cpp 和.h 文件构成的 C++程序

头文件和.cpp 文件之间的主要区别是它们的使用方式不同。根据约定，在头文件和源文件添加什么内容和处理它们的方式如下所示：

- .cpp 文件主要包含确定程序做什么的代码，它们由函数定义组成。
- .h 文件包含函数声明(即函数原型，但不是定义)、内联函数定义、枚举和自己的类型定义，以及预处理器指令。应避免把函数定义放在.h 文件中(inline 函数是一个例外)，否则该函数在程序中就有重复的定义，导致链接错误。
- .h 文件还可以包含在两个或多个.cpp 文件中共享的常量定义。
- 每个.cpp 文件必须包含它需要的头文件。
- 在编译程序时，只编译.cpp 文件，根据#include 指令，该文件包含了程序需要的头文件内容。
- 编译的每个.cpp 文件都会生成一个对象文件。术语“对象文件”与类对象没有任何关系。对象文件只是包含编译器的二进制输出的文件，它常常用扩展名.obj 来标识。对象文件由链接程序组合到一个可执行模块中，该模块的扩展名通常是.exe。

前面使用的头文件仅提供了使用标准库所必须的声明。程序示例短小而简单，因此不必使用独立的头文件，来包含函数声明或常量定义。第 11 章将开始定义自己的数据类型，那时就需要头文件了。实际的 C++程序包含大量的头文件和.cpp 文件。

程序中的每个.cpp 文件和它包含的所有头文件称为一个转换单元。使用这个术语是因为它不必是一个文件，但在大多数 C++实现方式中，它就是一个文件。编译器处理每个转换单元的过程和生成对象文件的过程是无关的。然后由链接程序处理这些对象文件，在对象文件之间建立必要的连接，以生成可执行程序模块。

10.1.1　名称的作用域

对于程序中的一个名称来说，作用域用于指定该名称有效的语句的范围。变量名或其他在语句块中声明的名称(放在花括号中)有块作用域，这也称为局部作用域，因为这种名称就包含

在声明它的语句块中。有块作用域的名称只能在从声明它的位置开始，到该块结束为止之间的范围内使用。下面是一个例子：

```
int main() {
  const int limit = 10;
  for(int i= 1;  i <= limit; i++)
    std::cout << std::endl << i << "squared is " << i*i;
}
```

其中名称 limit 的作用域是函数 main()的整个函数体。循环变量 i 的作用域是 for 循环，它由循环控制表达式和循环体组成。

1. 名称的隐藏

可以在外部作用域中定义一个与内部作用域相同的名称，此时，在内部作用域中定义的名称会遮挡了外部作用域中的名称。如下所示：

```
int main() {
  const int limit = 10;                                          //Outer limit
  std::cout << "Outer limit is " << limit << std::endl;          //Outer limit
  {
    const int limit = 5;                                  //Hides limit with value 10
    std::cout << "Inner limit is " << limit << std::endl;   //Inner limit
    for(int i= 1;  i <= limit; i++)                         //Inner limit
      std::cout << std::endl << i << "squared is "  < i*i;
  }
}
```

在包含 for 循环的内部块中，limit 变量的值是 5，它隐藏了外部作用域中同名的变量的值 10。因此，内部循环中的输出语句会生成 limit 的值 5，for 循环执行 5 次。

访问隐藏的名称

假定前面的代码希望 for 循环迭代的次数由隐藏的 limit 变量值来确定。此时，可以使用作用域解析运算符::选择在外部作用域中定义的变量，而不是在当前作用域中的同名变量。下面是一个例子：

```
int main() {
  const int limit = 10;                                          //Outer limit
  std::cout << "Outer limit is " << limit << std::endl;          //Outer limit
  {
    const int limit = 5;                                  //Hides limit with value 10
    std::cout << "Inner limit is " << limit << std::endl;   //Inner limit
    for(int i= 1;  i <= ::limit; i++)                       //outer limit
      std::cout << std::endl << i << "squared is "  < i*i;
  }
}
```

上面的代码使用作用域解析运算符的一元形式操作名称 limit，还使用该运算符的二元形式通过其命名空间名 std 来限定名称 cout。在 for 循环控制表达式中，表达式::limit 指定 limit 变量在外部作用域中，所以循环会执行 10 次。For 循环中的输出语句会引用内部变量 limit，所以值是 5。

2. 全局作用域

在所有块的外部和所有指定或未指定的命名空间(例如函数或变量)中声明的名称有全局命名空间作用域，也称为全局作用域。术语“文件作用域”有时也表示全局作用域，因为该名称可以在从声明它的位置开始，到包含它的文件(转换单元)结束之间的范围内使用。但“全局作用域”这个名称更好。下面是一个例子：

```
const int limit = 10;          // Global scope, also called file scope

int main() {
  for(int i= 1;  i <= limit; i++)
    std::cout << std::endl << i << "squared is " << i*i;
}
```

在这个例子中，变量 limit 有全局作用域，因为它的声明在文件中的所有函数之外。limit 的作用域从声明它的位置开始，到转换单元结束为止。可以在同一个转换单元中定义的任何函数中访问它。在这个例子中，它在 for 循环条件中用作循环变量 i 的上限。

当然，定义为全局作用域的名称可以用同名的局部变量隐藏。如下面的代码：

```
const int limit = 10;          // Global scope, also called file scope
int main() {
  const int limit = 5;         // Hides limit at global scope
  std::cout << "Inner limit is " << limit << std::endl;     //Inner limit
  for(int i= 1;  i <= limit; i++)                           //Inner limit
     std::cout << std::endl << i << "squared is "  < i*i;
}
```

在 main()函数体中定义的 limit 变量隐藏了全局作用域的变量。如果要访问全局变量 limit，也可以使用作用域解析运算符，如下所示：

```
const int limit = 10;          // Global scope, also called file scope

int main() {
  const int limit = 5;         // Hides limit at global scope
  std::cout << "Inner limit is " << limit << std::endl;     //Inner limit
  for(int i= 1;  i <= ::limit; i++)                         //Outer limit
    std::cout << std::endl << i << "squared is "  < i*i;
}
```

现在循环由拥有全局作用域的 limit 的值控制，而循环前面的语句输出局部变量 limit 的值。这里循环执行 10 次，如果不对 limit 使用作用域解析运算符，循环就只执行五次。

全局变量应总是初始化好的，在默认情况下，如果需要，就初始化它。如果没有为全局变量提供初始值，它就会初始化为 0。

10.1.2 “一个定义”规则

在转换单元(它是添加了所有包含头文件内容的.cpp 文件)中，每个不是局部块作用域的变量、函数、类类型、枚举类型或模板都只能定义一次。例如，变量可以有多个声明，但它表示

的内容只能有一个惟一的定义。

内联函数是这个规则的例外，内联函数的定义必须出现在调用该函数的每个转换单元中，但在所有的转换单元中，给定内联函数的所有定义都必须相同。因此，应在头文件中定义内联函数，并在需要该内联函数的源文件包含这个头文件。

当然，自己的数据类型肯定要在多个转换单元中使用，所以允许不同的转换单元包含给定类型的定义，但这些定义必须相同。为此，可以把类型的定义放在一个头文件中，再使用#include指令把它添加到需要它的源文件中。当然，在一个转换单元中，给定类型的重复定义是非法的。也就是说，应注意定义头文件内容的方式，避免在一个转换单元中出现重复的定义，详见本章后面的内容。

10.1.3 程序文件和链接

转换单元中的实体常常需要在另一个转换单元中访问。显然函数就是这方面的一个例子。还有其他例子，例如在全局作用域中定义的、在若干个转换单元中共享的变量。编译器一次处理一个转换单元，所以这种引用不能由编译器解析。只有在程序的转换单元中所有的对象文件都可用时，链接程序才能解析它。

转换单元中的名称在编译/链接过程中处理的方式由属性 linkage 来确定。linkage 表示由一个名称表示的实体可以在程序代码的什么地方使用。程序中使用的每个名称要么有链接属性，要么没有链接属性。当某个名称用于在声明该名称的作用域外部的代码块中访问程序的变量或函数时，就有链接属性。否则就没有链接属性。如果某个名称有链接属性，就可以有内部链接或外部链接属性。因此，转换单元中的每个名称都有内部链接属性、外部链接属性，或者没有链接属性。

确定名称的链接属性

注意无论名称是在头文件中声明，还是在源文件中声明，应用于名称的链接属性都不受影响。在转换单元中，每个名称的链接属性都是在.cpp 文件中插入了头文件的内容后确定。名称的三个链接属性有如下含义：

- 内部链接属性：该名称表示的实体可以在同一个转换单元的任何地方访问。例如，在全局作用域中定义的、声明为 const 的变量名在默认情况下具有内部链接属性。
- 外部链接属性：具有这种链接属性的名称除了可以在定义它的转换单元中访问之外，还可以在另一个转换单元中访问。换而言之，该名称表示的实体可以在整个程序中共享和访问。前面编写的所有函数以及在全局作用域中定义的非 const 变量都有外部链接属性。
- 没有链接属性：名称如果没有链接属性，它表示的实体只能在应用于该名称的作用域中访问。在块中定义的所有名称，即局部名称都没有链接属性。

现在的问题是：在函数内部，如何访问在另一个转换单元中定义的变量？这涉及到变量如何声明为外部的问题。

10.1.4 外部名称

在由几个文件组成的程序中，一个源文件中的函数调用与另一个文件中的函数定义之间的连接由链接程序建立(或解析)。在链接程序开始动作之前，编译器会编译函数的调用——为此，编译器要从函数原型中提取需要的信息。编译器并不介意函数的定义是在同一个文件中，还是在另一个.cpp 文件中，这是因为函数名在默认情况下具有外部链接属性。如果函数没有在调用它的转换单元中定义，编译器就会把这个调用标记为外部，让链接程序处理它。

变量名是不同的。编译器需要某种提示：某个名称的定义对于当前转换单元来说是外部的。如果希望用一个名称访问当前转换单元外部的变量，就必须用 extern 关键字来声明该变量，如下所示：

```
extern double pi;
```

这个语句声明名称 pi 在当前块的外部定义。类型必须对应于定义中的类型。在 extern 声明中不能有初始值。

把变量声明为 extern，表示它是在另一个转换单元中定义的。因此编译器就把变量标记为具有外部链接属性。链接程序就会在名称和它引用的变量之间建立连接。图 10-2 演示了把变量名声明为 extern 的过程。

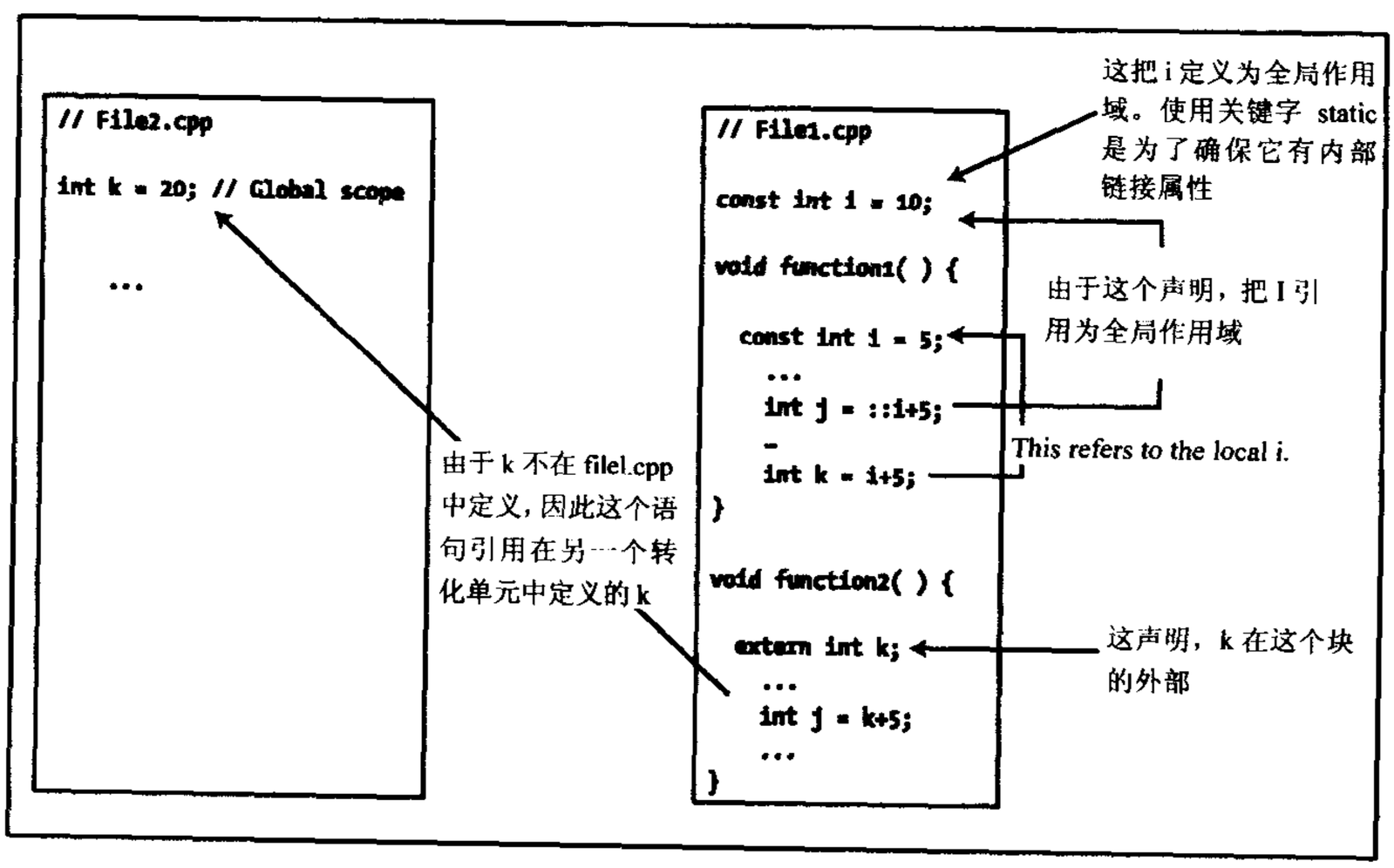


图 10-2 把变量声明为 extern

在 function2()中，名称 k 的 extern 声明表示，该函数中对 k 的后续使用都引用的是在当前块(这里就是函数块)外部定义的 k。注意，extern 声明并没有定义 k，只是表示 k 是在其他地方定义的。这里，名称 k 具有外部链接属性，因为它引用了另一个转换单元中的变量。编译器不能把该名称和变量定义连接起来，因为该定义在另一个转换单元中。编译器会把该变量标记为外部，在链接对象文件以创建一个可执行模块时，由链接程序建立该连接。

图 10-2 还演示了使用作用域解析运算符访问具有内部链接属性的全局变量。还可以使用不带作用域解析运算符的名称访问局部变量。

注意，如果给定块中的一个名称有 extern 声明，就不能在同一个块中定义该名称。所以，

下面的代码不会编译：

```
int main() {
  int limit = 10;           // Illegal - redefinition!!
  std::cout << "Local limit is " << limit << std::endl;
  extern int limit;         // External declaration of limit
    std::cout << "External limit is " << limit << std::endl;
    for(int i= 1;  i <= limit; i++)                    // Inner limit
      std::cout << std::endl << i << "squared is "  < i*i;
  return 0;
}
```

因为 limit 的 extern 声明和定义出现在同一个块作用域中，所以编译器会把它标记为一个错误。这是因为 limit 有两个定义：外部定义和本地定义。

但是，下面的代码就会编译：

```
int main() {
  int limit = 10;           // OK - not in same block as external declaration
  std::cout << "Local limit is " << limit << std::endl;
  {
    extern int limit;         // External declaration of limit
      std::cout << "External limit is " << limit << std::endl;
      for(int i= 1;  i <= limit; i++)
         std::cout << std::endl << i << "squared is "  < i*i;
  {
  return 0;
}
```

假定 limit 在另一个转换单元中声明为全局变量，如下面的语句所示：

```
int limit = 15;           // External linkage
```

这会使 for 循环执行 15 次。

迫使 const 变量具有外部链接属性

下面再看一个例子。假定使用下面的语句在一个文件 file1.cpp 中把变量定义为全局变量：

```
double pi=3.14159265;
string days[]={
                  "Sunday","Monday", "Tuesday", "Wednesday",
                  "Thursday", "Friday", "Saturday"
                };
```

这些语句定义了变量 pi 和数组 days[]，因此就不能在其他地方重复定义这两个变量了。但是，把变量声明为外部，就可以在另一个源文件(或在同一个源文件中)的另一个函数中访问这些变量。例如，文件 file2.cpp 中的一个函数包含如下语句：

```
extern double pi;                    //Variable is defined in another file
extern string days[];                //Array is defined in another file
```

这些语句没有创建变量，只是表示变量在其他地方定义，因此编译器不应在当前的作用域

中查找它们。变量在文件 file2.cpp 中使用，而它们的实际位置由 file1.cpp 中的定义来确定。在链接程序把对象文件链接在一起时，就会建立这个连接。

提示：

当然，如果把一个名称声明为外部，但没有找到对应的定义，链接程序就会生成一个错误消息，也不会创建可执行模块。

在这个例子中，名称 pi 和 days 显然都表示常量，所以应避免修改这些变量。因此在 file1.cpp 中定义它们时，可以把它们声明为 const：

```
const double pi=3.14159265;
const string days[]={
                        "Sunday", "Monday", "Tuesday", "Wednesday",
                        "Thursday","Friday","Saturday"
                     };
```

但是，把这些变量声明为 const，会在默认情况下使它们具有内部链接属性，这样就不能在其他转换单元中使用它们了。为此，在定义这些变量时，可以使用 extern 关键字重写这个属性：

```
extern const double pi=3.14159265;
  extern const string days[]={
                              "Sunday", "Monday", "Tuesday","Wednesday",
                              "Thursday","Friday", "Saturday"
                                 };
```

这些语句仍定义了 const double 变量和 const string 数组。这里使用关键字 extern 是为了告诉编译器，这些变量名应具有外部链接属性，但它们声明为常量。

现在，要在 file2.cpp 中访问这些变量，就必须把它们声明为 const 和 extern：

```
extern const double pi;                    //Variable is defined in another file
extern const string days[];                //Array is defined in another file
```

在有这些声明的块中，使用名称 pi 和 days 引用的是在另一个文件中定义的常量。这些声明可以出现在需要访问这些名称的任意转换单元中，同名可以出现在转换单元的全局作用域中，在源文件的全部代码中使用，也可以出现在块中，此时，它们只能在这个局部作用域中使用。

全局变量对要共享的常量值非常有益，因为它们可以在任何转换单元中访问。在所有需要访问常量值的程序文件中共享它们，就可以确保该常量在整个程序中使用相同的值。但是，前面都是在源文件中定义常量，最好是在头文件中定义它们。

下面看一个实际的例子。

程序示例 10.1——使用外部变量

创建一个源文件，其中包含一些数据定义：

```
// File data10_01.cpp
#include <string>
using std::string;
```

```
// The next two variables have external linkage
int count;                          // Will be initialized to 0 by default
float phi = 1.618f;                 // The divine proportion or golden ratio

// Without the extern, the following would not be accessible
// from another file because they would have internal linkage
extern const double pi = 3.14159265;
   extern const string days[] = {
                    "Sunday",    "Monday",  "Tuesday",  "Wednesday",
                    "Thursday",  "Friday",  "Saturday"
                    };
```

在另一个源文件中访问这些数据：

```
// Program 10.1 Accessing External Variables prog10_01.cpp
#include <iostream>
#include <string>
#include <iomanip>
using std::cout;
using std::endl;
using std::string;

// Declare external variables
extern float phi;
extern const double pi;
extern const string days[];
extern int count;

int main() {
   cout << std::setprecision(3) << std::fixed;
   cout << endl
        << "To 3 decimal places..." << endl;

   cout << "...a circle with a diameter of phi has an area of"
        << pi*phi*phi/4
        << endl;

   cout << "...phi squared is "    << phi*phi << endl;
   cout << "...in fact, phi+1 is also" << phi+1  << endl;

   cout << "Value of count is" << count << endl;

   count +=  3;
   cout << "Today is" << days[count] << endl;
   return 0;
}
```

这个例子的结果如下所示：

```
To 3 decimal places...
...a circle with a diameter of phi has an area of 2.056
...phi squared is 2.618
...in fact, phi+1 is also 2.618
```

```
Value of count is 0
Today is Wednesday
```

例子的说明

该例子的运行情况一如所料。phi 确然是外部变量，因为它在一个文件 main()中使用，却在另一个文件中定义。常量 pi 和常量数组 days 也具有外部链接属性，因为它们声明为 extern。如果删除 extern 关键字，代码就不会编译。

这个程序显示了 count 的默认值，显然，它初始化为 0。最后修改 count 的值，用它来索引 days 数组。

注意 prog10_01.cpp 文件中的 extern 声明使名称可用于该转换单元中的任意函数。如果要把这些变量的访问限制在特定的函数中，就可以把 extern 声明放在该函数体中。

10.2 命名空间

对于比较大的程序，为所有具有外部链接属性的实体指定惟一的名称是比较困难的。特别是当应用程序由多个程序员同时开发时，或销售商打算为第三方建立一个库进行销售时，就更是如此。如果没有某种机制，发生命名冲突的可能性就非常高。在以后几章介绍用户定义的类型或类时，最有可能出现命名冲突。命名空间就是为克服这个困难而设计的。

命名空间是程序中的一个区域，该区域把一个额外的名称(即命名空间的名称)附加到该区域中的所有实体名上。两个不同的命名空间可以包含同名的实体，但这些实体是不同的，因为它们都附加了不同的命名空间名称。图 10-3 演示了在一个程序中定义的两个命名空间，这两个命名空间在同一个源文件中。

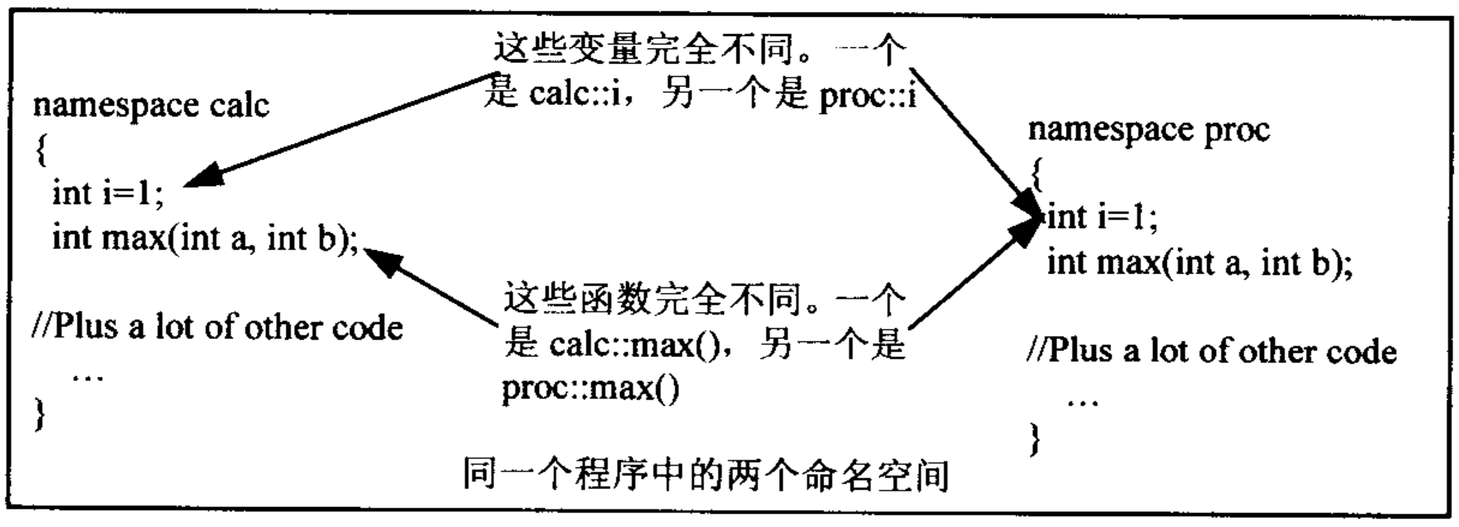


图 10-3 使用命名空间

在每个命名空间中，都定义了一个变量 i 和一个函数 max()；但是，它们引用不同的实体，且不会产生冲突。因为它们的名称用定义它们的命名空间名称进行了限定，于是每个名称都是惟一的。

在一个程序中，常常使用一个命名空间名称表示所有有同一目标的代码集合。每个命名空间都表示函数、相关的全局变量和声明的某个逻辑组合。命名空间还可以用于完全包含一个发布单元，例如一个库。

标准库是在命名空间 std 中定义的。这就是说，标准库中的每个外部名称都加上了前缀 std。例如，输出流的名称是 std::cout，因此要显示一个变量 value 的内容，就可以使用下面的代码：

```
std::cout<<std::endl<<value;
```

运算符::是作用域解析运算符的二元形式，上面的语句给名称 cout 和 endl 加上了修饰符，这就告诉编译器，这些名称的定义在 std 命名空间中。当然，本书前面的例子都使用了 std 命名空间中的名称，或者为名称添加如下的 using 声明：

```
using std::cout;
```

如第 1 章所述，通过如下的 using 指令，就可以使用命名空间中的任意名称：

```
using namespace std;
```

但是，这完全偏离了使用命名空间的目的，增加了不小心使用 std 命名空间中的名称而出错的可能性。因此，最好使用限定的名称，或者为 std 命名空间中要使用的名称加上 using 声明。

10.2.1 全局命名空间

前面编写的所有程序都使用了在全局命名空间中声明的名称。在没有定义命名空间时，就默认使用全局命名空间。全局命名空间中的所有名称就是声明时的名称，没有附加命名空间名称。在有多个源文件的程序中，具有链接属性的所有名称都在全局命名空间中。

对于小程序，可以在全局命名空间中定义名称，这不会遇到任何问题。对于较大的应用程序，出现命名冲突的机会大大增加，此时就应使用命名空间，把代码分成逻辑组。这样，从名称的角度来看，每个代码段都是自包含的，从而防止了命名冲突。

当然，还需要知道如何声明命名空间，下面就介绍这个内容。

10.2.2 定义命名空间

下面的语句就可以声明一个命名空间：

```
namespace myRegion {
  // Code you want to have in the namespace, including
  // function definitions and declarations, global variables,
  // templates, etc.
}
```

注意：

在命名空间定义的最后，右花括号后没有分号。

这里指定的命名空间名称是 myRegion。它惟一标识了命名空间，这个名称会附加于在该命名空间中声明的所有实体上。花括号限定了命名空间 myRegion 的作用域，命名空间作用域中的每个名称都会附加 myRegion。

注意：

当然，在命名空间中不应包含函数 main()。运行时环境要求 main()在全局命名空间中定义，因此必须把函数 main()放在所有命名空间的外部。

在文件中添加第二个命名空间定义，就可以扩展命名空间的作用域。例如，程序文件包含如下内容：

```
namespace calc {
  //This defines namespace calc
  //The initial code in the namespace goes here
}

namespace sort {
  //Code in a new namespace, sort
}

namespace calc {
  //This extends the namespace calc
  //Code in here can refer to names in the previous
  //calc namespace block without qualification
}
```

这里有两个块声明为命名空间 calc，用命名空间 sort 分隔两个块。第二个 calc 块是第一个 calc 块的继续。因此，在这两个 calc 块中定义的函数都属于同一个命名空间。第二个块称为扩展命名空间定义，因为它扩展原来的命名空间定义。在文件中还可以有更多的扩展命名空间定义，在同一个命名空间中添加更多的代码。

当然，通常不能直接用这种方式组织源文件。但是，如果在程序文件中包含了几个头文件，并且每个头文件的一部分代码都属于同一个命名空间，那么就变成上述的情况。一个常见的例子是程序文件包含了许多标准库头文件(每个头文件都属于命名空间 std)，同时自己的头文件散布其中(在另一个命名空间中定义)。例如：

```
#include <iostream>      //In namespace std
#include "mystuff.h"     //In my namespace calc
#include <string>        //In namespace std - extension namespace definition
#include "morestuff.h"   //In my namespace calc - extension namespace definition
```

最后，注意在命名空间内部引用该命名空间中的名称时，不需要添加修饰符。例如，属于命名空间 calc 的名称可以在 calc 中引用，不需要用命名空间名称修饰它们。

下面是一个简单的例子，说明声明和使用命名空间的机制。

程序示例 10.2——使用命名空间

创建一个程序，它包含两个.cpp 文件。第一个文件只包含一些常量的定义，但这次是在命名空间中定义：

```
//Program 10.2 Using a namespace  File:Data10_02.cpp
#include <string>

namespace data {
  extern const double pi=3.14159265;
  extern const std::string days[]={
                          "Sunday", "Monday", "Tuesday", "Wednesday",
                          "Thursday", "Friday", "Saturday"
                                  };
}
```

这个文件在命名空间 data 中定义了 pi 和 days[]。数组 days[]的类型是 string，在标准库中定义，所以需要用标准库的命名空间名称 std 来修饰类型名。

在另一个包含 main()的转换单元中使用这些变量：

```
//Program 10.2 Using a namespace  File:Prog10_02.cpp
#include <iostream>
#include <string>

namespace data {
  extern const double pi;              //Variable is defined in another file
  extern const std::string days[];     //Array is defined in another file
}

int main() {
  std::cout << std::endl
            << "pi has the value "
            << data::pi << std::endl;

  std::cout << "The second day of the week is"
            << data:: days [1]<<std::endl;

  return 0;
}
```

编译并运行，结果如下所示：

```
pi has the value 3.14159
The second day of the week is Monday
```

例子的说明

在包含 main()的文件中，必须把 pi 和 days[]声明为外部，因为它们在一个独立的转换单元中定义，如下面的语句所示：

```
namespace data {
  extern const double pi;              //Variable is defined in another file
  extern const std::string days[];     //Array is defined in another file
}
```

必须把外部变量的声明放在命名空间 data 中，因为该变量在第一个.cpp 文件的这个命名空间中定义。这说明了前面讨论的一个问题——命名空间可以分片定义。甚至在一个文件中，也可以有若干个命名空间块对应于同一个命名空间名称，每个块的内容都在同一个命名空间中。因为类型 string 在标准库的命名空间中定义，所以必须在声明中提供修饰过的名称 std::string。

这不是组织程序代码的最佳方式。一般说来，应把 pi 和 days 的声明放在一个头文件 data.h 中。这个头文件的内容如下所示：

```
//Declarations for globals in namespace data    File: data.h
#include <string>
namespace data {
  extern const double pi=3.14159265;
  extern const std::string days[]={
                          "Sunday", "Monday", "Tuesday", "Wednesday",
                          "Thursday", "Friday", "Saturday"
                                   };
}
```

接着，为了使该定义可以在包含 main()的文件(或其他需要访问这些常量的文件)中使用，就只需在开始处添加#include 指令：

```
#include "data.h"
```

扩展名.h 表示这是一个头文件。程序示例 10.3 将介绍这个头文件。

注释：

这里的#include 语法略有不同。只有在包含标准库的头文件时，才能省略.h 扩展名。本章后面在讨论预处理器时，将介绍使用双引号而不是尖括号的重要性。还要确保头文件的内容在一个转换单元中只包含一次，后面还将介绍其处理过程。

10.2.3 使用 using 声明

using 声明在给定的命名空间中声明一个特定的名称，前面的例子都使用了这个声明。为命名空间中的一个名称进行 using 声明的形式如下所示：

```
using namespace_name::identifier;
```

其中 using 是关键字，namespace_name 是命名空间的名称，identifier 是使用时不加限定的名称。这个声明从命名空间中引入了一个名称，它可以引用不同的实体。例如，在一个命名空间中定义的一组重载函数，就可以用一个 using 声明来引入。

注意，在所有例子的全局作用域中都添加了 using 声明和指令，也可以把它们放在一个命名空间中，或放在一个函数中，甚至语句块中。在这些情况下，声明和指令会应用到包含它的块结束为止。

注意：

在使用未限定的变量名时，编译器在使用前先在当前作用域中查找该变量的定义。如果没有找到，就查找包含当前块的外层块。这样一直继续下去，直到全局作用域为止。如果变量的声明没有在全局作用域中找到(它可以是 extern 声明)，编译器就断定该变量没有定义。

10.2.4 函数和命名空间

如果要把函数放在命名空间中，只需把函数的原型放在该命名空间中即可。函数可以在其他地方定义，但要使用限定过的名称，换言之，函数定义不一定要放在命名空间块中。下面看一个例子。

假定编写两个函数 max()和 min()，分别返回一组数值的最大值和最小值。把函数的声明放在命名空间中，如下所示：

```
//compare.h
namespace compare {
  double max(const double* data, int size);
  double min(const double* data, int size);
}
```

这些语句可以放在头文件 compare.h 中，再在使用该函数的文件中包含该头文件。

函数的定义放在一个.cpp 文件中。现在编写定义，但不把定义语句放在命名空间块中，只要每个函数名用命名空间名称限定过即可。文件的内容如下所示：

```
//compare.cpp
#include "compare.h"

//Function to find the maximum
double compare::max(const double* data, int size) {
  double result = data[0];
  for(int i=1; i<size; i++)
    if(result<data[i])
      result =data[i];
  return result;
}

//Function to find the minimum
double compare::min(const double* data, int size) {
  double result = data[0];
  for(int i=1; i<size; i++)
    if(result>data[i])
      result =data[i];
  return result;
}
```

需要把头文件 compare.h 包含进来，以标识命名空间。这样，编译器就能推断出函数在命名空间中。

当然，也可以把函数定义的代码直接放在 compare 命名空间中。这样，compare.cpp 的内容就如下所示：

```
//compare.cpp
namespace compare {
  //Function to find the maximum
  double max(const double* data, int size) {
    double result = data[0];
    for(int i=1; i<size; i++)
      if(result<data[i])
        result =data[i];
    return result;
  }

  //Function to find the minimum
  double min(const double* data, int size) {
    double result = data[0];
    for(int i=1; i<size; i++)
      if(result>data[i])
        result =data[i];
    return result;
  }
}
```

如果以这种方式编写函数定义，就不需要在文件中包含头文件 compare.h。这是因为定义已经在命名空间中。

无论怎样定义函数，函数的使用方法都是一样的。为了验证这一点，下面就使用刚才定义的函数。

程序示例 10.3——using 声明

如前所述，把函数声明放在头文件 compare.h 中：

```
//compare.h
namespace compare {
  double max(const double* data, int size);
  double min(const double* data, int size);
}
```

把函数的定义放在另一个.cpp 文件中，该文件包含如下代码：

```
//compare.cpp
#include "compare.h"

//Function to find the maximum
double compare::max(const double* data, int size) {
  double result = data[0];
  for(int i=1; i<size; i++)
    if(result<data[i])
      result =data[i];
  return result;
}

//Function to find the minimum
double compare::min(const double* data, int size) {
  double result = data[0];
  for(int i=1; i<size; i++)
    if(result>data[i])
      result =data[i];
  return result;
}
```

下面编写一个包含 main()定义的.cpp 文件，如下所示：

```
//Program 10.3 Using functions in a namespace
#include <iostream>
#include "compare.h"

using compare::max;            //Using declaration for max
using compare::min;            //Using declaration for min
using std::cout;
using std::endl;

int main() {
  double data[]={1.5, 4.6, 3.1, 1.1, 3.8, 2.1};
  const int dataSize = sizeof data/sizeof data[0];
```

```
    cout << endl;
    cout << "Minimum double is "<<min(data, dataSize)<<endl;
    cout << "Maximum double is "<<max(data, dataSize)<<endl;

    return 0;
}
```

编译这两个.cpp 文件并链接它们，运行程序，会得到如下结果：

```
Minimum double is 1.1
Maximum double is 4.6
```

例子的说明

下面的指令在包含 main()的文件中引入了函数的声明：

```
#include "compare.h"
```

注意：

如果文件 compare.h 在与源文件不同的目录上，#include 指令还必须包含从源文件到头文件的目录路径。在本例中，头文件 compare.h 在与源文件相同的目录上。

接着为每个函数编写一个 using 声明，这样在使用函数名时就不必加上命名空间名称了：

```
using compare::max;          //Using declaration for max
using compare::min;          //Using declaration for min
```

对 compare 命名空间也可以使用 using 指令：

```
using namespace compare;
```

因为命名空间只包含函数 max()和 min()，所以这两种方式都可以，只是 using 指令比 using 声明少一行代码。但是，在一般情况下，using 指令的效果与 using 声明是完全不同的。using 指令允许在要使用的文件中声明命名空间中的所有名称；using 声明则只引入一个名称。

接着使用 using 声明引入 std 命名空间中的 cout 和 endl。

在 main()中，首先定义一个数组 data，再把它作为参数传送给两个函数：

```
double data[]={1.5, 4.6, 3.1, 1.1, 3.8, 2.1};
```

需要给每个函数传送数组 data 的长度，下面就计算这个长度，并把它存储在常量 dataSize 中：

```
const int dataSize = sizeof data/sizeof data[0];
```

最后，在输出语句中调用函数，显示数组 data 中的最大值和最小值：

```
cout << "Minimum double is" << min(data, dataSize) << endl;
cout << "Maximum double is" << max(data, dataSize) << endl;
```

如果没有为函数名使用 using 声明(或给 compare 命名空间使用 using 指令)，就必须在输出语句中限定函数：

```
cout << "Minimum double is" << compare::min(data, dataSize) << endl:
cout << "Maximum double is" << compare::max(data, dataSize) << endl:
```

10.2.5 函数模板和命名空间

可以在命名空间中定义函数模板。上面的函数是从模板中生成的，下面就把该函数作为一个例子。在这个例子中，把模板定义放在头文件的命名空间中：

```
//tempcomp.h
namespace compare {
//Function template to find the maximum element in an array
template<class T> T max(const T* data, int size) {
    T result = data[0];
    for(int i=1; i<size; i++)
      if(result<data[i])
         result =data[i];
    return result;
  }

//Function template to find the minimum element in an array
template <class T> T min(const T* data, int size) {
    T result = data[0];
    for(int i=1; i<size; i++)
      if(result>data[i])
         result =data[i];
    return result;
  }
}
```

注释：

可以按照推荐的规则，把模板定义放在一个单独的文件 tempcomp.cpp 中，而仅把模板的原型放在文件 tempcomp.h 中。这里使用了类模板，需要给每个模板定义加上关键字 export 作为前缀。这样代码才能与 ANSI/ISO 标准完全兼容。但是，目前许多编译器都不支持使用 export，所以上述代码对 C++编译器来说更适合。

当然，如果要显式定义特殊情况，例如类型 char*，就可以把特殊情况的原型放在上述命名空间中，把定义放在.cpp 文件中，这些原型和定义都要使用限定过的函数名，或放在命名空间块中。

程序示例 10.4——命名空间中的函数模板

在下面的代码中检验命名空间 compare 中的模板：

```
//Program 10.4 Using function templates in a namespace
#include <iostream>
#include "tempcomp.h"

using compare::max;            //Using declaration for max
using compare::min;            //Using declaration for min
using std::cout;
using std::endl;
```

```
int main() {
  double data[]={1.5, 4.6, 3.1, 1.1, 3.8, 2.1};
  int numbers[]={23, 2, 14, 56, 42, 12, 1, 45};

  cout << endl;

  const int dataSize = sizeof data/sizeof data[0];
  cout << "Minimum double is "<<min(data, dataSize)<<endl;
  cout << "Maximum double is "<<max(data, dataSize)<<endl;

  const int numbersSize = sizeof numbers/sizeof numbers[0];
  cout << "Minimum integer is"<<min(numbers, numbersSize)<<endl;
  cout << "Maximum integer is"<<max(numbers, numbersSize)<<endl;

  return 0;
}
```

这个程序的结果如下所示：

```
Minimum double is 1.1
Maximum double is 4.6
Minimum integer is 1
Maximum integer is 56
```

例子的说明

这个例子与前一个例子非常类似，只是前一个例子显式定义了每个函数。在本例中，没有在.cpp 文件中定义 min()和 max()函数，因为这些定义是由编译器从合适的模板中生成的。因为模板定义放在 compare 命名空间中，所以编译器就利用该命名空间生成函数定义。

在 main()中，函数在使用时没有加上限定符，因为在命名空间 compare 中，对它们的名称使用了 using 声明。没有 using 声明，就需要在输出语句中限定函数名，就像前面的例子那样。例如，前两个输出语句应如下所示：

```
cout << "Minimum double is "<<compare::min(data, dataSize)<<endl;
cout << "Maximum double is"<<compare::max(data, dataSize)<<endl;
```

这两个函数调用的第一个参数都是 double 类型的数组，编译器应为接受这个参数类型的 min()和 max()函数生成定义。

10.2.6 扩展命名空间

前面讨论了命名空间如何在几个块中定义。下面扩展程序示例 10.4，说明如何扩展已有的命名空间。

程序示例 10.5——使用扩展命名空间

下面添加另一个头文件，该头文件包含一个函数的模板，该函数可以对一组数据进行规范化，即该函数会调整数值，使它们位于 0 到 1 之间。下面是头文件的内容：

```
// normal.h
// Normalize an array of values to the range 0 to 1
```

```
#include "tempcomp.h"

namespace compare {
  template<class Toriginal, class Tnormalized>
      void normalize(Toriginal* data, Tnormalized* newData, int size) {
    Toriginal minValue = min( data, size);      //Get minimum element

    //Shift all elements so minimum is zero
    for( int i = 0 ; i<size; i++)
      newData[i] = static_cast<Tnormalized>(data[i]-minValue);

    Tnormalized maxValue = max( newData, size);    //Get max of new set

    //Scale elements so maximum is 1
    for( int i = 0 ; i<size; i++)
      newData[i] /= maxValue;
  }
}
```

要使用这个模板，可以在前一个例子的 main()函数的最后添加几个语句：

```
//Program 10.5 Using a function template in a namespace extension
#include <iostream>
#include <iomanip>
#include "normal.h"

using compare::max;            //Using declaration for max
using compare::min;            //Using declaration for min
using std::cout;
using std::endl;

int main() {
  double data[]={1.5, 4.6, 3.1, 1.1, 3.8, 2.1};
  int numbers[]={23, 2, 14, 56, 42, 12, 1, 45};

  cout << endl;

  const int dataSize = sizeof data/sizeof data[0];
  cout << "Minimum double is"<<min(data, dataSize)<<endl;
  cout << "Maximum double is "<<max(data, dataSize)<<endl;

const int numbersSize = sizeof numbers/sizeof numbers[0];
cout << "Minimum integer is"<<min(numbers, numbersSize)<<endl;
cout << "Maximum integer is "<<max(numbers, numbersSize)<<endl;

double newData[numbersSize];                              //Array for result
compare::normalize(numbers, newData, numbersSize);   // Normalize

//Output the normalized array values
for( int i = 0 ; i< numbersSize; i++) {
  if(i%5 == 0)
    cout<<endl;
```

```
        cout<<std::setw(12)<<newData[i];
    }
    cout<<endl;
    return 0;
}
```

这个程序的结果如下所示：

```
Minimum double is 1.1
Maximum double is 4.6
Minimum integer is 1
Maximum integer is 56

            0.4       0.0181818       0.236364          1       0.745455
            0.2                 0            0.8
```

例子的说明

下面先检验新的头文件。对包含 max()和 min()的函数模板的头文件使用#include 指令，在包含头文件的内容后，新的头文件如下所示：

```
namespace compare {
  //Templates for min() and max()
}
namespace compare {
  //Template for normalize()
}
```

第二个模板定义是对第一个模板的扩展。

normalize()函数把结果存储为 0 到 1 之间的值的数组，则无论原始数值的类型 Toriginal 是什么，类型 Tnormalized 都必须能容纳浮点数值。模板头文件可以灵活地处理原始数组的类型和结果的类型：

```
template<class Toriginal, class Tnormalized>
      void normalize(Toriginal* data, Tnormalized* newData, int size)
```

第一个参数 data 是类型为 Toriginal 的数组，第二个参数 newData 是与第一个参数相同大小的数组，它存储 Tnormalized 类型的数值。对这两个参数还可以使用数组表示法，这样头文件就变成：

```
template<class Toriginal, class Tnormalized>
   void normalize(Toriginal data[], Tnormalized newData[], int size)
```

normalize()、max()和 min()的函数模板都在同一个命名空间中，因此函数 max()和 min()可以在 normalize()中使用，且不加限定符。我们使用 min()函数模板，获取作为第一个参数传送过来的数组中的最小元素值：

```
Toriginal minValue = min(data, size);          //Get minimum element
```

可以把 minValue 声明为 Toriginal 类型，这与数组 data 的类型一致。接着，从 data 数组的每个元素中减去 minValue，把计算结果存储在类型为 Tnormalized 的新数组 newData 中：

```
for(int i = 0 ; i<size; i++)
     newData[i] = static_cast<Tnormalized>(data[i]-minValue);
```

从 data 数组的元素中减去 minValue 的结果显式强制转换为 Tnormalized 类型。执行这个操作后，新数组中的最小元素是 0。其他的元素都是正数。

现在需要在新数组中找出最大元素的值：

```
Tnorm maxValue = max(newData, size);          //Get max of new set
```

这个语句使用 max()函数模板生成一个函数，该函数把 Toriginal 类型的数组作为第一个参数。在下一个语句中，使用循环对新数组的每个元素除以 maxValue：

```
for(int i = 0 ; i<size; i++)
  newData[i] /= maxValue;
```

执行这个操作后，最小的元素仍是 0，但最大的元素是 1。其他的元素在 0 和 1 之间。

现在看看程序的主要部分。main()中的新语句使用了一个操纵程序，所以必须添加对<iomanip>的#include 指令。用下面的指令替换 tempcomp.h 的#include 指令：

```
#include "normal.h"
```

注意从主源文件中删除了 tempcopm.h 的#include 指令。如果保留该指令，max()和 min()的定义就会包含两次，一次是直接包含，一次是通过 normal.h 的#include 指令间接包含。多次包含会产生编译错误。本章后面将介绍如何防止这种情况的发生——甚至在包含了两个指令的情况下。

调用 normalize()函数规范化数组中的整数值：

```
compare::normalize(numbers, newData, numbersSize);      // Normalize
```

normalize()的模板放在命名空间 compare 中，且没有添加 using 声明或指令，因此函数名必须加上限定符，这样编译器才能辨识出该名称。编译器从函数模板中创建一个定义，该函数把 int 类型的数组作为第一个参数，把 double 类型的数组作为第二个参数。

现在数组的值已规范化，用熟悉的方式输出它们：

```
for(int i = 0 ; i< numbersSize; i++) {
  if(i%5 == 0)
     cout<<endl;
  cout<<std::setw(12)<<newData[i];
}
cout<<endl;
```

10.2.7 未指定名称的命名空间

不必给命名空间指定名称，但这并不意味着命名空间就没有名称。下面的代码会声明一个没有指定名称的命名空间：

```
namespace  {
  //Code in the namespace, functions, etc.
}
```

这个语句创建一个命名空间，它有一个由编译器生成的内部名称。在一个文件中只能有一个未指定名称的命名空间，如果其他命名空间声明时没有名称，就是第一个未指定名称的命名空间的扩展。

但是，每个未指定名称的命名空间在转换单元中都是惟一的。在不同转换单元中的未指定名称的命名空间都是不同的命名空间。

未指定名称的命名空间不在全局命名空间中。未指定名称的命名空间在转换单元中是惟一的，这两点有非常重要的意义。这意味着，在未指定名称的命名空间中声明的函数、变量和其他实体都是定义它们的转换单元中的本地成员。它们不能在其他转换单元中访问。

把函数定义放在未指定名称的命名空间中，与在全局命名空间中把函数声明为 static 有相同的效果。把函数和变量声明为全局作用域的 static，是确保它们不能在转换单元外部访问的一种常用方式。未指定名称的命名空间可以更好地限制可访问性，现在已不采用以这种方式使用 static。

10.2.8 命名空间的别名

在涉及多个开发组的大型程序中，需要使用很长的命名空间名称，以确保不发生命名冲突。但这么长的名称使用起来很繁琐。把 SystemGroup5_Process3_Subsection2 这样的名称附加在每个函数调用上，是非常麻烦的。

为了解决这个问题，可以在本地为很长的命名空间名称定义一个别名。为命名空间名称定义别名的一般语法如下所示：

```
namespace  别名 = 原来的命名空间名称;
```

然后就可以使用别名代替原来的命名空间名称，来访问命名空间中的名称。
例如，为上一段中的命名空间名称定义一个别名，如下所示：

```
namespace SG5P3S2 = SystemGroup5_Process3_Subsection2;
```

现在，用下面的语句调用原命名空间中的函数：

```
int maxValue = SG5P3S2::max(data, size);
```

10.2.9 嵌套的命名空间

可以在一个命名空间中定义另一个命名空间。如果在特定的环境中，处理这种嵌套命名空间的机制就很容易理解。假定有如下嵌套的命名空间：

```
// outin.h
namespace outer {
  double max(double* data, const int& size) {
    //body code…
  }

  double min(double* data, const int& size) {
    //body code…
```

```
  }

  namespace inner {
    double* normalize(double* data, const int& size) {
      //…
      double minValue = min(data, size);        //Calls max() in compare
      //…
    }
  }
}
```

在命名空间 inner 中，函数 normalize()可以直接调用函数 min()(它在命名空间 outer 中)。这是因为 normalize()的声明包含在 inner 命名空间中，而 inner 命名空间包含在命名空间 outer 中，也就是说，normalize()的声明包含在命名空间 outer 中。

要调用全局命名空间中的 min()，应按通常的方式限定函数名：

```
int result = outer::min(data, size);
```

当然，还可以对函数名使用 using 声明，或为命名空间指定 using 指令。为了调用全局命名空间中的 normalize()，需要用两个命名空间名称限定函数名：

```
double* newData = outer::inner::normalize(data, size);
```

如果把函数原型包含在命名空间中，并另外提供了函数定义，这个规则也适用。在命名空间 inner 中，可以仅编写 normalize()的原型，再把函数 normalize()的定义放在文件 outin.cpp 中：

```
// outin.cpp
#include "outin.h"
double* outer::inner::normalize(double* data, const int& size) {
  //…
  double minValue=min(data, size);     //Calls max() in compare
  //…
}
```

当然，为了编译成功，编译器需要知道命名空间。所以，在函数定义之前要包含头文件 outin.h，该头文件需要包含命名空间声明。

本章多次使用了#include 指令的新形式。下面详细论述它是如何工作的，并介绍其他可用的预处理器指令。

10.3 预处理器

预处理器通常是编译器的一个组成部分。它是编译器把 C++程序代码编译为机器指令之前执行的一个过程。预处理器的任务是根据包含在源文件中的指令，使源代码正确进入编译阶段——这些指令称为预处理器指令。所有的预处理器指令都以符号#开头，以便与 C++语言语句区分开来。表 10-1 是完整的预处理器指令集合。

表 10-1　预处理器指令

指　令	功　能
#include	支持包含头文件
#if	允许条件编译
#else	#if 的 else
#elif	#else #if
#endif	标记#if 指令的结束
#if defined(或#ifdef)	如果定义了一个符号，就执行操作
#if ! defined(或#ifndef)	如果没有定义一个符号，就执行操作
#define	定义一个符号
#undef	删除一个符号
#line	重新定义当前行号和文件名
#error	输出编译错误消息，停止编译
#pragma	提供机器专用的特性，同时保证与 C++的完全兼容

提示：

尽管所有的预处理器指令都以#开头，但反过来就不对了，例如，#import 就不是一个预处理器指令。

在源文件(.cpp 文件)中，预处理器阶段会分析、执行所有的预处理器指令，然后删除它们，得到一个仅包含 C++语句的转换单元。然后，编译器就开始用这个得到的文件进行编译(假定没有错误)，生成包含机器码的对象文件。然后，这个对象文件必须和程序中的其他对象文件一起，由链接程序处理，生成最终的可执行模块。

前面的例子都使用了预处理器指令#include，现在我们已经很熟悉它了。如表 10-1 所示，还有其他的预处理器指令，它们大大加强了指定程序的编译方式的灵活性。这些预处理器指令会在程序编译之前执行。它们能修改组成程序的语句组，但不涉及程序的执行。

注释：

下面的讨论介绍了使用预处理器指令的基本语法，用法的例子和使用某些预处理器指令的建议。如果需要，可以先阅读“调试方法”部分，再回过头来学习这一节。

10.3.1　在程序中包含头文件

头文件是外部文件，通常存储在磁盘上，其内容可使用#include 预处理器指令包含到程序中。我们很熟悉如下语句：

```
#include <iostream>
```

这个语句把 iostream(标准库头文件，它支持流输入输出操作)的内容提取到程序中。iostream 的内容替代了#include 指令。这是把标准库头文件包含在程序中的一般语句的一种特殊形式，#include 指令的一般形式如下所示：

```
#include <标准库头文件名>
```

其中，标准库头文件名可以放在尖括号中。如果包含了未使用的头文件，主要的影响是占用了较多的内存，延长了编译时间，还可能使阅读程序的人感到有点混乱。

可以把自己的头文件包含在程序中，但#include 语句有点不同，文件名要放在双引号中，例如：

```
#include "myheader.h"
```

在这个语句中，把在双引号中命名的文件的内容引入到程序中，替代#include 指令。通过这种方式，可以把任何文件的内容包含到程序中。只需在双引号中指定文件名即可，如上面的例子所示。对于大多数编译器，可以使用大小写字符来指定文件名。

理论上，可以给自己的头文件指定任何名称，不一定使用扩展名.h。但是，这是大多数 C++程序员都遵守的一个约定，读者最好也遵守。

在双引号中指定文件名和把文件名放在尖括号中的区别是查找该程序的过程。这个操作随编译器的不同而不同，且在编译器说明文档中有详细的说明。通常情况下，如果使用尖括号，编译器就只搜索包含标准库头文件的默认目录。如果头文件名放在双引号中，编译器就会搜索当前目录(一般是正在编译的源文件所在的目录)，之后搜索包含标准库头文件的目录。如果把头文件放在其他目录下，为了查找到它，必须在双引号中指定从源文件到头文件的完整路径。在一些 C++实现方式中，可以指定包含源文件的目录的相对路径。

可以使用#include 机制把程序分为几个文件，并管理库函数的声明。这个特性的一个常见用法是创建一个包含所有函数原型和全局变量的头文件，然后把这些实体作为一个独立的单元来管理，并在程序的开始处包含该头文件。

如果在程序中包含了多个文件，就需要避免信息的重复。代码的重复常常导致编译错误，本章将在后面介绍预处理器指令提供的一些功能，确保这些任意给定的代码块仅在程序中出现一次，即使不小心多次包含了这些代码块，也是如此。

注意：

通过#include 语句引入到源文件中的文件还可以包含其他#include 语句。此时，预处理器会以同样的方式处理这些额外的#include 语句，在代码中没有更多的#include 语句后，就继续执行。程序示例 10.5 就是这样一个例子。

10.3.2 程序中的置换

预处理器指令可以在源代码中使用符号，在预处理过程中以预定义的方式进行置换。符号置换的过程可以是简单的一对一符号置换，也可以是复杂的宏扩展置换。

最简单的符号置换是指定一系列字符，来代替程序文件中的给定符号。为此，可使用#define 指令。例如，可以用表示一个数值的一组字符置换符号 PI，如下所示：

```
#define PI 3.14159265
```

尽管 PI 看起来像是一个变量，但它与变量没有任何关系，这是一个很大的缺点。PI 只是一个符号或标志，在程序代码编译前，该符号会用一组指定的字符来代替。还要注意，3.14159265

不是一个数值，只是一个字符串，不会进行检查。在编译前，预处理器会遍历代码，在它认为置换有意义的地方，用字符串 PI 的定义(字符序列 3.14159265)来代替它。

提示：

这里有必要解释一下。例如，预处理器认为 PI 出现在注释中时，或者是字符串(在双引号之间)的一部分时，不应进行置换。稍后讨论这个问题。

在 C 中，#define 指令常常以这种方式来定义符号常量，但在 C++中，最好用 const 来定义合适的常量。例如：

```
const long double pi=3.14159265L;
```

现在 pi 是 long double 类型的一个常量值。编译器会确保为 pi 指定的值始终是 long double 类型。可以把这个定义放在一个头文件中，再把该头文件包含在需要该值的源文件中。

或者，把该处理定义为具有外部链接属性：

```
extern const long double pi=3.14159265L;
```

这样，只要在需要的地方添加 pi 的 extern 声明，就可以在任意转换单元中使用它。

#define 预处理器指令的一般形式如下所示：

```
#define 标识符  字符序列
```

其中标识符遵循 C++中标识符的常见定义，它可以是任意字母和数字序列，序列中的第一个字符必须是字母，下划线算作一个字母。

注意字符序列可以是任意字符序列，而不仅仅是数字。下面再看一个例子：

```
#define BLACK WHITE
```

预处理器会把程序中的 5 字符序列 BLACK 用另一个 5 字符序列 WHITE 代替。用于代替标志标识符的字符序列没有限制。

注意：

使用#define 指令有三个主要的缺点：#define 指令不支持类型检查，#define 指令不考虑作用域，符号名不能限制在一个命名空间中。

1. 从程序中删除标志

在#define 指令中，如果没有为标识符指定置换字符串，标识符就会用一个空的标志字符串来代替，换言之，标识符被删除了。例如，下面的指令定义了一个标识符：

```
#define VALUE
```

这个指令的结果是在程序文件中，从这个指令后面的语句中删除所有的标识符 VALUE。

2. 取消宏名的定义

还可以使#define 指令的置换结果只应用于程序文件的一部分。使用#undef 指令可以取消对标识符的定义。假定已定义了标识符 VALUE，现在要在文件的某个位置删除该标识符，就

可以使用下面的指令：

```
#undef VALUE
```

在这个语句的后面，标识符 VALUE 已取消了定义，也就不再置换 VALUE 了。下面看一个例子。考虑下面的代码：

```
#define PI 3.142
// All occurrences of PI in code from this point will be replaced
// …
#undef PI
// The identifier PI is no longer defined.
// Any references to PI will be left in the code from this point
```

在#define 和#undef 指令之间，预处理器会把代码中的 PI 用 3.142 代替，在其他地方，则保留 PI。

#define 和#undef 指令的组合使用还有另一个用途，本章后面在介绍作出决策的预处理器指令时探讨。

10.3.3 宏置换

预处理器宏建立在刚才的#define 指令例子所隐含的概念上，它允许调用多个参数化的置换，提供了更大范围的结果。这不仅涉及到用一个固定的字符序列来替换标志标识符，还允许在宏定义中指定参数，在使用宏时，用所提供的参数值替换参数。只要置换序列中有给定的参数，该参数就会用对应的参数来替换。下面看一个例子：

```
#define Print(Var) cout << (Var) << endl
```

这个指令提供了两个级别的置换。在#define 语句中，Print(var)用其后的字符串来置换，var 也可以进行置换。例如，下面的语句：

```
Print(ival);
```

预处理器会把它转换为下面的语句：

```
cout << (ival) << endl;
```

置换指令的一般形式如下所示：

```
#define 标识符(标识符列表) 置换字符串
```

在一般情况下，允许使用任意个参数，因此可以定义比较复杂的置换。

注意：

在第一个标识符和左括号之间不能有空格，否则括号就被解释为置换字符串的一部分。

扩展上面的宏指令，添加第二个参数：

```
#define Print(var, digits) cout << setw(digits) << (var) << endl
```

这个指令可以置换 var 和 digits。例如，下面的语句：

```
Print(ival,15);
```

预处理器会把它转换为下面的语句：

```
cout << setw(15) << (ival) << endl;
```

当然，要编译这个语句，需要包含头文件<iomanip>和<iostream>，以及一些 using 声明。
预处理器宏的这种应用非常普通，但在几乎所有的情况下，最好使用内联函数或函数模板。这样，就可以确保对参数的类型进行相应的检查，减少出错的可能性。除了前面的宏之外，还可以定义一个函数模板：

```
template<class T> inline void Print (const T& var, const int& digits) {
  cout<<setw(digits)<<var<<endl;
}
```

现在对于下面的语句：

```
Print(ival,15);
```

编译器就会从模板中生成一个内联函数，该函数接受合适类型的参数。
这类宏的一个常见用法是把复杂的函数调用表示得很简单，以提高程序的可读性。

注释：

C++开发系统提供了其他宏功能，例如可以通过命令行来定义宏，或在编译环境下定义宏。这些功能都是系统特有的，超出了本书的范围。

宏可能导致错误

为了展示使用宏时内在的错误类型，定义一个宏，用下面的指令生成两个值中的较大者：

```
#define max(x,y) x>y ? x : y
```

接着在程序中添加如下语句，进行一次置换：

```
result = max(myval, 99);
```

预处理器会把这个语句扩展为：

```
result = myval> 99 ? myval : 99;
```

这个置换是成功的，创建了一个函数的幻影。但是，这不是一个函数，在进行置换时要特别小心。对一个不同的语句，就有可能得到奇怪的结果——特别是置换标识符包含显式或隐式赋值时，就更是如此。例如，扩展上一个例子，就会得到意想不到的结果：

```
result = max(myval++, 99);
```

置换过程会生成如下语句：

```
result = myval++> 99 ? myval++ : 99;
```

如果 myval 的值大于 99，myval 就会递增两次。注意在这种情况下，使用括号是没有什么

帮助的。如果把语句写为：

```
result = max((myval++), 99);
```

预处理器会把这个语句转换为：

```
result = (myval++)> 99 ? (myval++) : 99;
```

应避免编写生成任何表达式的宏。一般情况下，应使用内联函数的模板，而不是宏。在这种情况下，这个宏要完成的任务可以使用标准库函数 max()来完成，函数 max()返回两个参数中的较大者。函数 max()定义为一个模板，所以它可以处理各种类型的参数。

除了刚才介绍的置换陷阱之外，在使用宏时优先级规则也会作怪。下面用一个简单的例子来说明。假定编写一个宏，计算两个参数的乘积：

```
#define product(m, n) m*n
```

下面的语句使用这个宏：

```
result = product(x, y+1);
```

这个语句会编译，但不能得到希望的结果，因为宏扩展为：

```
result = x*y+1;
```

当然，这等于(x*y)+1，而不是 x*(y+1)。找出这个错误需要很长时间，因为外部没有指示进行了什么操作。而且，这个错误的值肯定会在程序中传播开来。

此时，解决方法非常简单。如果必须使用宏来生成表达式，就对所有的实体加上括号。上面的例子应改写为：

```
#define product(m, n) ((m)*(n))
```

现在一切正常。外层括号似乎是多余的，但由于不知道会在什么地方使用宏，最好加上括号。更好的解决方法是遵循下面的建议：

除非有非常重要的原因，否则就使用内联函数来代替预处理器宏。

10.3.4 放在多行代码中的预处理器指令

预处理器指令必须放在一个逻辑代码行上，但这并不是说指令不能放在多个物理代码行上。方法是在每行的末尾使用续行符\，但最后一行不用。下面的语句：

```
#define min(x, y) \
                  ((x)<(y) ? (x):(y))
```

指令定义在第二行上继续，从第一个非空白字符开始。因此可以把文本放在第二行上，这增强代码的可读性。注意\必须是代码行上的最后一个字符，之后按下回车符。

10.3.5 把字符串作为宏参数

字符串常量在和宏一起使用时，可能会出现混淆。最简单的字符串置换是一级定义，如下所示：

```
#define MYSTR "This string"
```

定义了这个宏，下面的语句：

```
cout << MYSTR;
```

就会置换为：

```
cout << "This string";
```

这正是我们希望的结果。但是，如果在 C++语句中(而不是在#define 指令中)加上双引号，置换就会失败。下面举例说明。把开头的那个语句改写为：

```
#define MYSTR This string
```

再编写一个语句：

```
cout << "MYSTR";          //Does not invoke the macro
```

这次，不会置换 MYSTR。在程序中，引号内的内容都会被看作字面量字符串，所以预处理器不分析它。

把参数指定为预处理器宏，还有一个特殊的方法，即把宏实现为字符串。例如，可以指定宏显示字符串，如下所示：

```
#define PrintString(arg) cout <<#arg
```

在宏表达式中，字符#放在参数 arg 的前面，表示在进行置换时，参数放在双引号中。因此，如果在程序中编写如下语句：

```
PrintString(Hello);
```

预处理器就会把它转换为：

```
cout << "Hello";
```

提示：

把这个显得比较离奇的技术引入预处理器，原因是没有这个技术，就不可能把可变的字符串放在宏定义中。如果给宏参数加上双引号，预处理器不会把它解释为变量，而只会解释为加上引号的字符串。另一方面，如果在宏表达式中加上引号，引号中的字符串就不会解释为参数变量标识符，而仅解释为字符串常量。

这个机制的一种用法是把变量名转换为字符串，如下面的指令所示：

```
#define show(var) cout<<#var<<" = " (var)<<endl
```

这个宏创建了输出变量名及其值的一种缩写方式。如果编写下面的语句：

```
show(number);
```

就会生成下面的语句：

```
cout << "number" << " = " (number) << endl;
```

还可以进行置换，以显示双引号中的字符串。假定定义了如上所示的宏 PrintString，就可以编写下面的语句：

```
PrintString("Output");
```

该语句将置换为：

```
cout<< "\"Output\"";
```

这是可行的，因为预处理器知道需要在两端加上\"，以正确显示双引号中的字符串。

10.3.6 在宏表达式中连接参数

有时还希望宏可以接受两个或更多的参数，并把它们连接在一起，且其中没有空格。这样就可以根据宏的参数，综合不同的变量名。假定定义如下的宏：

```
#define join(a, b) ab
```

这不会按照希望的方式工作。表达式的定义会解释为两个字符序列 ab，而不会解释为参数 a 后跟参数 b。如果用一个空格把它们分隔开，结果也会用空格分隔开，这也不是我们希望的结果。

实际上，预处理器提供了另一个运算符来解决这个问题。具体方法是把宏指定为：

```
#define join(a, b) a##b
```

这个运算符由两个字符##组成，用于把宏参数分隔开，告诉预处理器，把两个置换的结果连接在一起，且其中没有空格。例如，编写下面的语句：

```
strlen(join(var,123));
```

会得到如下语句：

```
strlen(var123);
```

这初看起来似乎是一个局限性很大的功能，因为只要编写 var123，就不必使用宏了。实际上，让预处理器做决策时，常常要使用这类指令。

10.4 逻辑预处理器指令

在预处理器执行一个指令块，而不执行另一个指令块时，常常要用到宏置换，方法是根据在程序文件中测试的条件，从几个宏所定义的置换操作中选择其中一个宏。

预处理器通过逻辑#if 指令提供了这个功能。#if 指令和 C++中的 if 语句基本相同，它极大地扩展了预处理器的使用范围。

10.4.1 逻辑#if 指令

逻辑#if 指令的使用方式有两种。第一，可以测试某个符号以前是否用#define 指令定义过。第二，可以测试某个常量表达式是否为真。下面先测试符号定义，因为这是最常用的工具。

要测试某个标识符是否存在(在以前的#define 指令中定义)，可以使用如下所示的指令：

```
#if defined 标识符
```

如果指定的标识符已定义，#if 后面的语句组就包含在要编译的源文件中。这个语句组用如下指令结束：

```
#endif
```

如果标识符还未定义，就跳过#if 和#endif 之间的语句，它们也不是程序的一部分。这与C++编程中使用的逻辑处理基本一致——这里把它应用于决定是否把某个语句块包含为程序的一部分。

#if 和#endif 之间的语句块可以包含其他预处理器指令。如果#if 测试的标识符已定义，就执行这些指令。

下面看一个具体的例子。假定在程序文件中有如下代码：

```
//code that sets up the array data[]
  double average = 0.0;

  #if defined CALCAVERAGE
  int count=sizeof data/sizeof data[0];
  for(int i=0; i<count; i++)
    average += data[i];
  average /= count;
  #endif

  //rest of the program…
```

#if 指令测试符号 CALCAVERAGE 是否已由以前的预处理器指令定义。如果已定义，就把#if 和#endif 指令之间的代码编译为程序的一部分。如果符号 CALCAVERAGE 没有定义，就不包含这段代码。

测试标识符是否已定义还有一种缩写方式。下面的测试：

```
#if defined CALCAVERAGE
```

可以替换为：

```
#ifdef CALCAVERAGE
```

这样就更简明，也很清晰。以#ifdef 开头的语句块应以#endif 结束。

防止代码重复

也可以测试标识符是否不存在。这个指令的一般形式如下所示：

```
#if !defined 标识符
```

如果标识符以前没有定义，就把#if 和#endif 之间的语句包含在要编译的源文件中。其缩写方式如下所示：

```
#ifndef 标识符
```

注意：

实际上，#ifndef 比#ifdef 使用得更频繁，因为它禁止把头文件(或其他文件)的内容包含多次。

如前所述，用#include 指令包含到.cpp 文件中的头文件，本身也可以包含#include 指令，组合其他头文件，这个功能在大程序中使用得很广泛。对于涉及到许多头文件的复杂程序，头文件常常在源文件中包含多次——在许多情况下，这是不可避免的。当然，这种代码的重复会产生编译错误，特别是在重复的代码包含定义时，就更容易出错。因此，应尽可能杜绝这种情况。

那么，该如何使用#if 指令来防止程序中出现重复的代码？事实上，这是很简单的。在绝对不能重复的代码块的开头和结尾处，输入下面的代码：

```
#if !defined MYHEADER_H
#define MYHEADER_H

// Block of code that must not be duplicated

#endif
```

在预处理器第一次遇到这段代码时，标识符 MYHEADER_H 还没有定义过，因此，#if 条件为真。于是#if 后面的代码块就包含在程序中。这有两个结果：第一，MYHEADER_H 通过#define 指令来定义。第二，把代码块添加到代码中。以后，这组代码就不会包含在程序中了。这是因为标识符 MYHEADER_H 已经存在，#if 条件为假。因此，语句块只包含一次，且不会重复。

注意，要定义标识符，标识符只能在#define 指令中出现一次，且不需要值。在指令中也可以把标识符定义为空：

```
#define MYHEADER_H
```

这个指令定义了标识符 MYHEADER_H，但它不包含任何值。

把#if/#endif 组合(如上所示)放在每个头文件内容的外部，是一种标准方式。这样可以确保每个头文件的内容在源文件中不会出现多次。通常把头文件名用作标识符，以确保标识符对每个头文件来说是惟一的。下面看一个简单的例子。头文件 compare.h(详见程序示例 10.3)的内容如下所示：

```
// compare.h
#ifndef COMPARE_H
#define COMPARE_H

namespace compare {
  double max(const double* data, int size);
  double min(const double* data, int size);
}
```

```
#endif
```

注意：

以这种方式保护头文件的代码，是一种非常好的编程习惯。一旦把自己的函数收集到几个库中，就非常容易杜绝代码块重复的问题。

预处理器指令#if 不仅能测试一个标识符是否存在，还可以和逻辑运算符组合使用，测试是否定义了多个标识符。例如，下面的语句：

```
#if defined block1 && defined block2
```

如果 block1 和 block2 以前都定义过，这个语句就等于 true，这个指令后面的代码就会包含在程序中。同样，可以使用||运算符，甚至是用 defined 和!defined 把几个运算符组合起来的比较复杂的表达式。

注释：

#ifdef 和#ifndef 常常用于包含某个操作系统环境中专用的代码。例如，有各种 Unix 系统，每个系统都有自己的特性必须遵循。使用这里讨论的逻辑指令，就可以用一组源文件支持多种环境，方法是把每个环境特有的代码放在一个独立的块中，这个块由测试惟一符号的#ifndef 指令来控制。然后定义标识各种操作系统的特定符号，从而选择特定的环境。

10.4.2 测试特定值的指令

也可以使用#if 指令的一种形式来测试常量表达式的值。如果常量表达式的值是 true 或非 0 值，就包含#if 和#endif 指令之间的语句。如果常量表达式等于 false 或 0，就跳过#if 和#endif 指令之间的语句。#if 指令的这种形式如下：

```
#if  常量表达式
```

常量表达式必须是不包含强制转换的整数常量表达式。它可以包含预处理器宏，但在进行完所有的置换后，常量表达式必须是一个整数表达式。所有的算术操作都把值看作 long 或 unsigned long 类型。如果常量表达式的值不是 0，就在要编译的源代码中包含#if 和#endif 指令之间的语句。

这常常应用于测试前面预处理器指令赋予一个标识符的特定值。例如，下面的语句：

```
#if CPU==PENTIUM4
//Code taking advantage of Pentium 4 capability
#endif
```

只有标识符 CPU 在以前的#define 指令中定义为 PENTIUM4，#if 和#endif 指令之间的语句才会包含到程序中。

10.4.3 多个代码选择块

为了补足#if 指令，C++添加了#else 指令。它与 C++的 else 语句非常类似，如果#if 指令条

件失败，#else 指令就标识一组要执行的指令或要包含的语句。这样就可以选择两个代码块中的一个，把它包含到最终的源文件中。例如：

```
#if CPU== PENTIUM4
cout << " PENTIUM4 code version. " <<endl;
// PENTIUM4 oriented code
#else
cout << "Older Pentium code version. " <<endl;
// code for Older Pentium processors
#endif
```

在这个例子中，将根据 CPU 是否定义为 PENTIUM4 ，包含其中一组语句。

预处理器还允许利用#if 的一种特殊形式进行多项选择，可以从几个代码块中选择一个代码块，包含到程序中，或要执行的指令中。这就是#elif 指令，它的一般形式如下所示：

```
#elif 常量表达式
```

使用它的一个例子如下所示：

```
#if LANGUAGE == ENGLISH
#define Greeting"Good Morning. "
#elif LANGUAGE == GERMAN
#define Greeting "Guten Tag. "
#elif LANGUAGE == FRENCH
#define Greeting "Bonjour. "
#else
#define Greeting "Hi. "
#endif
std::cout << Greeting << std::endl;
```

有了这个指令序列，输出语句就会根据在前面#define 指令中赋予标识符 LANGUAGE 的值，显示各种不同的问候语中的一个。

另一个用法是根据表示版本号的一组标识符，在程序中包含不同的代码：

```
#if VERSION==3
// Code for version 3 here…
#elif VERSION==2
// Code for version 2 here…
#else
// Code for original version 1 here…
#endif
```

这样，就可以根据在#define 指令中设置的 VERSION，选择一个源文件，该源文件会编译生成不同版本的程序。

10.4.4 标准的预处理器宏

预处理器定义了几个标准的宏，可以在需要时调用它们。如表 10-2 所示。

表 10-2　标准的预处理器宏

宏　　名	说　　明
__LINE__	当前源文件中的代码行号，十进制整数
__FILE__	源文件的名称，字符串字面量
__DATE__	源文件的处理日期，字符串字面量，其格式是 mmm dd yyyy，其中 mmm 是月份(如 Jan、Feb 等)；dd 是日期，其格式为 01 到 31 的数字，一位数字的日期前面加上一个空格，yyyy 是 4 位数字的年份(如 1994)
__TIME__	源文件的编译时间，也是字符串字面量，其格式是 hh:mm:ss，这是一个字符串，包含小时、分钟和秒数。它们用冒号分隔开
__STDC__	这取决于实现方式。如果编译器选项设置为编译标准的 C 代码，通常就定义它；否则就不定义它
__cplusplus	在编译 C++程序时，它就定义为值 199711L

注意，每个宏名称都以两个下划线开头，除了宏__cplusplus 之外，每个宏名都以两个下划线结束。

__LINE__和__FILE__宏可以显示与源文件相关的信息。使用#line 指令可以修改行号。例如，如果要从某个位置开始，对行从 1000 开始计数，就可以添加下面的指令：

```
#line 1000
```

使用#line 指令可以修改__FILE__宏返回的字符串。__FILE__宏通常会生成完全限定的文件名，也可以把它改为其他内容。例如：

```
#line 1000 "The program file"
```

这个指令把下一行的行号改为 1000，把__FILE__宏返回的字符串改为"The program file"。这不会修改文件名，只修改宏返回的字符串。当然，如果只想修改文件名，而不修改行号，就可以在#line 指令中使用__LINE__宏：

```
#line __LINE__  "The program file"
```

使用日期和时间宏可以记录程序最后一次编译的时间，如下面的语句所示：

```
std::cout << std::endl
          << "Program last complied at "<<__TIME__
          << " on "                      <<__DATE__
          << std::endl;
```

编译完包含这个语句的程序后，该语句显示的值就是固定的，除非再次编译程序。在以后的执行过程中，程序还会输出程序编译的时间和日期。

10.4.5　#error 和#pragma 指令

在预处理阶段，如果出现了错误，#error 指令可以生成一个诊断消息。因此，这个指令一

般在测试某个条件(例如#if 指令)后执行。执行#error 指令的结果是把指令行中包含的内容显示为一个编译错误消息，并立即停止编译。例如，下面的语句：

```
#ifndef _cplusplus
#error "Error - Should be C++"
#endif
```

还可以在#error 指令行中包含其他预处理器宏。例如，如果只包含一个字符串，最好把该字符串放在双引号中。这将防止预处理器把它解析为一个宏。输出的形式是预先定义好的。

#pragma 指令专门用于实现预先定义好的选项，其结果在编译器说明文档中进行了详细解释。编译器未识别出来的#pragma 指令都会被忽略。

10.5 调试方法

在程序第一次完成时，大多数程序都包含错误。调试程序表示要花很多时间重新编写程序。

程序越大越复杂，包含的错误就越多，使之正常运行所需的时间和精力就越多。非常大的程序，例如操作系统，或复杂的应用程序，例如字处理系统，甚至是目前正在使用的 C++程序开发系统，都非常复杂，系统不可能完全不包含错误。读者对此已经有一些体会了，因为计算机上安装了一些系统。通常这类错误相当少，系统在编程时就清理了大多数错误。

有时，清理错误的同时，还会在程序中引入了新的错误。当然，这一定不能阻挡用户的编程脚步。调试是编程过程中的一个重要组成部分。

编写程序的方法主要受测试该程序的难易程度的影响。结构合理的程序由紧凑的函数组成，每个函数都有明确的目的，这种程序测试起来要比没有这些特性的程序容易得多。如果程序中的变量名和函数名做了精心的选择，并有非常多的注释——说明操作及其组件函数的作用。在这种程序中查找错误也会比较容易。很好地使用缩进格式和语句布局也会使测试和查找错误更方便。

全面讲述调试超出了本书的范围，因为本书主要讨论 C++语言。在任何情况下，都要使用 C++开发系统中特有的工具来调试程序。这里将介绍大多数调试系统通用的基本方法，并讨论 C++库中标准的基本调试技巧。

10.5.1 集成调试器

大多数 C++编译器都提供了内置于程序开发环境的大量调试工具。这些工具有非常强大的功能，可以大大减少使程序正常工作的时间。它们一般提供了范围广泛的技术来测试程序。常见的功能包括：

- 跟踪程序流：这个功能可以在执行程序时一次执行一个语句。在执行完每个语句后暂停执行，在按下一个特定的键后，程序继续执行下一个语句。调试环境的其他工具通常允许方便地显示信息，暂停执行，显示程序对数据执行了什么操作，这也称为单步调试程序。
- 设置断点：如果程序比较大、比较复杂，一次执行一个语句是非常烦琐的。甚至不可能在合理的时间内单步执行程序。如果程序中有一个循环要执行 10000 次，则单步执

行就是不现实的。比较好的方法是在代码中设置断点，使用断点来定义程序中选定的语句，在每次到达这些断点时，程序都会暂停执行，查看完成了什么操作。按下一个特定的键后，程序会继续执行到下一个断点。

- 设置观察窗口：这个功能允许指定要在程序执行过程中跟踪的变量值。所选变量的值会显示在程序的每个暂停处。如果单步执行程序，就会看到在某些暂停处，该变量值已改变了，有时在不希望变量值改变的地方，这些变量值也发生了变化。
- 检查程序元素：还可以检查各种程序组件。例如，在断点处，可以查看函数的信息，例如函数的返回类型和参数。还可以查看指针的信息，例如指针的地址，指针存储的地址和在该地址上存储的数据。我们还可以访问表达式的值，修改变量。修改变量有助于绕过有问题的区域，用正确的数据执行后面的区域。

10.5.2 调试中的预处理器指令

许多 C++开发系统都提供了强大的调试工具，但增加自己的跟踪代码仍是很有帮助的。使用条件预处理器语句，可以把某些代码块包含在程序中，以帮助测试。可以完全控制要显示的数据格式，方便调试，甚至可以根据程序中的条件或关系，设置不同的输出。

程序示例 10.6——用预处理器指令进行调试

下面用一个有点做作的程序来演示用预处理器指令进行调试的过程，该程序通过函数指针数组随机调用函数。这个例子还利用了前面介绍的几个技术。为了这个练习，声明 3 个函数，它们都包含在命名空间 fun 中。把该命名空间的声明放在头文件中：

```
// functions.h
#if !defined FUNCTIONS_H
#define FUNCTIONS_H
namespace fun {
  // Function prototypes
  int sum(int, int);                    // Sum arguments
   int product(int, int);                // Product of arguments
   int difference(int, int);             // Difference between arguments
}
#endif
```

把文件的内容放在#if/#endif 指令组合的中间，防止文件的内容在一个转换单元中包含多次。把函数的定义放在文件 functions.cpp 中：

```
// functions.cpp

//#define TESTFUNCTION                    // Uncomment to get trace output

#ifdef TESTFUNCTION
#include <iostream>                       // Only required for trace output
#endif

#include "functions.h"
```

```
// Definition of the function sum
int fun::sum(int x, int y) {
   #ifdef TESTFUNCTION
   std::cout << "Function sum called." << std::endl;
   #endif

   return x+y;
}

// Definition of the function product
int  fun::product(int x, in y) {
   #ifdef TESTFUNCTION
   std::cout << "Function product called." << std::endl;
   #endif

   return x*y;
}

// Definition of the function difference
int fun::difference(int x, int y) {
   #ifdef TESTFUNCTION
   std::cout << "Function difference called."<< std::endl;
   #endif

   return x-y;
}
```

这里需要标准头文件<iostream>，因为要使用流输出语句在每个函数中提供跟踪信息。

只有在文件中定义了符号 TESTFUNCTION，才能包含标准头文件<iostream>，编译输出语句。注意 TESTFUNCTION 目前还没有定义(该指令被注释掉了)。cout 和 endl 有显式的限定，因为在文件中没有使用 using 声明或指令。

在 main()中调用函数，main()在另一个.cpp 文件中，代码如下所示：

```
// Program 10.6 Debugging using the preprocessor
#include <iostream>
#include <cstdlib>                          // For random number generator
#include <ctime>                            // For time function

#include "functions.h"
using satd::cout;
using satd::endl;

#define TESTINDEX

// Function to generate a random integer 0 to count-1
int random(int count) {
returnstatic_cast<int>(
            (count*static_cast<long>(std::rand()))/(RAND_MAX+1));
}

int main ( ) {
```

```
int a = 10, b = 5;                          // Starting values
int result = 0;                            // Storage for relults

// Declaration for an array of function pointers
int (*pfun[])(int, int)= {fun::sum, fun::product, fun::difference};

int fcount = sizeof pfun/sizeof pfun[0];
int select = 0;                                    // Index for function selection
srand(static_cast<unsigned>(time(0)));     // Seed random generator

 // Select function from the pointer array at random
 for(int i = 0 ; i < 10 ; i++) {
    select = random(fcount);          // Generate random index 0 to fcount-1

    #ifdef TESTINDEX
    cout  <<  "Random number=" <<  select  <<  endl;
    if((select>=fcount) || (select<0)) {
      cout << "Invalid array index =" << select << endl;
      return 1;
    }
    #endif

    result = pfun [select](a,b);  //Call random function

cout << "result =" << result << endl;
  }
  result = pfun[1](pfun[0](a, b), pfun[2](a,b));
  cout << endl
       << "The product of the sum and the difference = " << result
       << endl;
  return 0;
}
```

运行该程序，结果如下所示：

```
Random number=2
result =  5
Random number=2
result =  5
Random number=1
result = 50
Random number=0
result = 15
Random number=1
result = 50
Random number=1
result = 50
Random number=0
result = 15
Random number=1
result = 50
Random number=2
```

```
result = 5
Random number=1
result = 50

The product of the sum and the difference = 75
```

一般情况下，读者得到的结果应与此有所不同。如果要跟踪命名空间 fun 中函数的输出，就必须在文件 functions.cpp 的开头取消对#define 指令的注释。

例子的说明

在包含 main()的文件的开头，有 3 个包含标准头文件的#include 指令：

```
#include <iostream>
#include <cstdlib>                  //For random number generator
#include <ctime>                    //For time function
```

头文件<cstdlib>是必须的，因为要使用标准库函数 rand()来生成随机数。头文件<ctime>提供了函数 time()的声明，该函数用于产生随机数过程的种子。稍后介绍它们的操作方式。

给自己的头文件加上一个#include 指令：

```
#include "functions.h"
```

这会添加函数 sum()、product()和 difference()的声明，在源文件中将随机调用这三个函数。这些函数是在命名空间 fun 中声明的。自己的头文件和源文件在同一个目录下，所以不需要指定文件的路径。

符号 TESTINDEX 的定义在 main()中打开可诊断输出：

```
#define TESTINDEX
```

定义了这个符号后，在 main()中输出诊断信息的代码就包含在要编译的源文件中。如果删除了这个指令，源文件中就不包含跟踪代码。在该代码后面的部分，for 循环的中间，将使用这个指令的结果：

```
#ifdef TESTINDEX
cout << "Random number = "<<select << endl;
if((select>=fcount) || (select<0)) {
  count << "Invalid array index = "<<select << endl;
  return 1;
}
#endif
```

这段代码检查数组 pfun 是否使用了有效的索引。我们不希望生成无效的索引值，所以在任何情况下都不应得到这个结果。

提示：

很容易生成无效的索引值，从而执行这个诊断代码。为此，只需让 random()函数生成不是 0、1 和 2 的数字。只需在 random()函数定义的返回语句之前加上++count;语句即可。在程序运行时，常常会得到错误的索引值，其几率大约为 25%。

random()函数的正确定义，会生成某个范围内的随机整数，如下所示：

```
int random(int count) {
returnstatic_cast<int>(
            (count*static_cast<long>(std::rand()))/(RAND_MAX+1));
}
```

标准库函数 rand()生成 0 到 RAND_MAX 之间的随机数(其中 RAND_MAX 是一个在<cstdlib>中定义的常量)。

提示:

似乎可以使用取模运算符来直接生成随机值，例如，使用表达式 rand()%count 就会得到 0 到 count 之间的值。但是，这会把从 rand()返回的值截尾成较低的位，考虑到伪随机数的生成方式，这些数不是随机的。而使用代码中的表达式，就可以使值散布在指定的范围内，而该范围内的数字应和原范围内的数字一样具有随机性。

RAND_MAX 不是一个可以递增的值。如果使用表达式 count*rand()/(RAND_MAX+1)增大 rand()返回的值，而 RAND_MAX 正巧是 int 类型的最大值，在一些实现方式中该表达式就会失败。如果 long 类型与 int 类型有相同的取值范围，前面使用的语句就不编译，而必须使用浮点算术来增大 rand()函数生成的值。rand()函数的下述版本总是有效的:

```
int random(int count) {
return static_cast<int>(
            (count*static_cast<double>(std::rand()))/(RAND_MAX+1.0));
}
```

在 main()中，声明并初始化一个函数指针的数组，如下面的语句所示:

```
int (*pfun[])(int, int)={fun::sum, fun::product, fun::difference};
```

这个数组使用在 functions.h 中声明的 3 个函数名来初始化。初始化列表中的每个函数名都用命名空间名称 fun 进行了限定。

要得到数组的大小，使用下面的语句:

```
int fcount = sizeof pfun/sizeof pfun[0];
```

获取函数指针的数组中元素个数与获取其他数组的元素个数没有区别。数组中有三个指针，因此 fcount 的值是 3。

在声明和初始化用于索引数组 pfun 的变量 select 后，就调用标准库函数 srand()来初始化伪随机数生成过程:

```
srand(static_cast<unsigned>(time(0)));  //Seed the random generator
```

传送给 srand()的无符号整数用于开始这个过程。标准库函数 time()从系统时钟返回当前时间(单位是秒)，每次运行程序时，这个值都是不同的。这样就可以确保每次伪随机数生成过程都是以不同的种子开始的。通常，返回的值是从 1970 年 1 月 1 日开始到现在所过去的秒数，但 C++标准没有指定它，这取决于用户自己的库。

在 main()的 for 循环中，随机调用命名空间 fun 中的函数:

```
for(int i = 0 ;  i < 10  ;  i++) {
   select = random(fcount);          // Generate random index 0 to fcount-1
```

```
#ifdef TESTINDEX
cout <<  "Random number =  " << select << endl;
if((select>=fcount)  ||  (select<0)) {
   cout << "Invalid array index =" << select << endl;
   return 1;
}
#endif

result = pfun[select] (a, b);      // Call random function

   cout <<  "result =  "  << result << endl;
}
```

函数 random()用于产生 0 到 fcount - 1 之间的随机索引，该索引存储在变量 select 中。如果 TESTINDEX 在这个文件的#define 指令中定义，就执行有效性检查代码。在有效性检查代码中，会测试无效的索引值，无效的索引会显示一个消息，再由 return 语句以可控制的方式终止程序。

程序通过函数指针进行调用，显示下述结果，之后结束：

```
result = pfun[1](pfun[0](a, b),pfun[2](a, b));
cout << endl
     << "The product of the sum and the difference = " << result
     << endl;
```

如果在文件 functions.cpp 中定义了符号 TESTFUNCTION，就可以从每个函数中获得跟踪结果。这是控制是否把跟踪语句编译到程序中的一种方便方式。下面以 product()函数为例，说明其工作原理：

```
int fun::product(int x, int y) {
  #ifdef TESTFUNCTION
  std::cout<<"Function product called."<<std::endl;
  #endif

  return x*y;
}
```

每次调用函数时，输出语句都仅显示一个消息，但只有定义了 TESTFUNCTION，输出语句才会编译。

注释：

预处理器符号 TESTFUNCTION 的#define 指令是包含它的文件的本地指令，所以每个需要 TESTFUNCTION 的文件都需要有自己的#define 指令。缺点之一是在所有的程序文件中会有大量的#define 指令。解决方法是把所有控制跟踪和其他调试输出的符号都收集到一个独立的头文件中，再用#include 指令把该头文件包含到所有的.cpp 文件中。这样，就可以通过调整这个头文件，来修改调试结果。

当然，可以定义任意多个不同的符号常量，如本章前面所述。如果需要，可以使用更复杂的控制机制来控制应包含哪些调试代码块，把它们组合到逻辑表达式中，用#ifdef 或#ifndef 条

件指令来测试。

注意：

所有这些诊断代码仅在测试程序时才包含在内。一旦程序可以正常工作，就要删除它们。因此要明白，这类代码不能替代错误检测和修复代码。如果在经过彻底测试的程序中出现了错误(这些错误是很有可能出现的)，就用错误检测和修复代码来处理。

10.5.3 使用 assert 宏

另一个诊断工具 assert()宏是在标准库中提供的。assert()宏在库头文件<cassert>中声明，它可以在程序中测试逻辑表达式，如果指定的逻辑表达式是 false，assert()就会终止程序，并显示诊断消息。

程序示例 10.7——演示 assert 宏

用一个简单的例子来演示：

```
//Program 10.7 Demonstrating assertions
#include <iostream>
#include <cassert>
using std::cout;
using std::endl;

int main() {
  int x=0;
  int y=5;

  cout<< endl;

  for(x=0; x<20; x++) {
    cout << "x= " << x << "   y= " << y << endl;
  assert(x<y);
  }
  return 0;
}
```

在 x 的值达到 5 时，输出中会测试一个断言消息。

例子的说明

除了 assert()语句之外，这个程序不需要什么解释，它只是在 for 循环中输出 x 和 y 的值。

程序由 assert()宏终止，只要条件 x<y 是 false，assert()语句就会调用 abort()。函数 abort()位于标准库中，其作用是立即终止程序。从结果中可以看出，当 x 的值达到 5 时，程序就会终止。assert 宏会在标准错误流 cerr(即显示屏幕)上显示结果。其中消息包含失败的条件，文件名和出现失败的行号。这对于包含多个文件的程序是非常有效的，因为可以准确地指出错误的来源。

assert 宏常常用于程序中的重要条件，在这种程序中，如果某些条件不满足，就肯定会出现灾难性后果。如果发生这种错误，就应确保程序不再继续执行。可以把任意逻辑表达式作为参数传送给 assert()宏，所以这个宏有非常高的灵活性。

程序示例 10.6 用随机数生成器生成了索引值，这个程序就包含这种情况。利用这个技术，总是可以找到错误，得到无效的索引值。如果索引超出了数组 pfun 的界限，结果肯定是灾难性的。

在程序示例 10.6 中，可以使用 assert()语句验证索引值的有效性。如果不使用#ifdef 语句块，可以使用下面的语句：

```
assert((select>=0) &&(select<fcount));
```

这个语句非常简单，也很有效，当出现错误时，该语句会提供足够的信息，指出程序在哪里终止。

关闭断言机制

如果在重新编译程序时，在程序文件的开头定义符号 NDEBUG，就可以关闭断言机制：

```
#define NDEBUG
```

这个语句会忽略转换单元中的所有断言语句。如果把这个#define 指令添加到程序示例 10.7 的开头，就会得到从 0 到 19 的所有 x 值，且不显示诊断消息。注意这个指令仅放在<cassert>的#include 语句之前才有效。

注释：

assert 不是错误处理机制，逻辑表达式的结果不应产生负面效果，也不应超出程序员的控制(例如打开一个文件是否成功)。程序应提供适当的代码来处理这种情况。

10.6 本章小结

本章讨论了在程序文件中、程序文件之间和跨程序文件的操作功能。C++程序一般由许多文件组成，程序越大，要处理的文件就越多。因此，如果要开发实际的 C++程序，很好地掌握命名空间和预处理器就是非常重要的。

本章的要点如下所示：

- 程序中的每个实体都只能有一个定义。
- 名称可以有内部链接属性，即该名称可以在一个转换单元中访问，名称也可以有外部链接属性，即名称可以在任何转换单元中访问。名称还可以没有链接属性，即名称只能在定义它的块中访问。
- 头文件可以包含源文件需要的定义和声明。头文件还可以包含模板和类型定义、枚举、常量、函数声明、内联函数定义，以及指定的命名空间。按照约定，头文件使用扩展名.h。
- 把函数定义和全局变量放在源文件中，C++源文件的扩展名是.cpp。
- 通过#include 指令可以把头文件的内容插入到.cpp 文件中。
- cpp 文件是转换单元的基础，编译器会处理转换单元，以生成对象文件。
- 命名空间定义了一个作用域——在这个作用域内声明的所有名称都附加了命名空间的名称。不在显式命名空间作用域内声明的名称就在全局命名空间中。

- 一个命名空间可以由几个独立的同名命名空间声明组成。
- 在不同的命名空间中声明的相同名称是不同的。
- 为了在命名空间的外部引用在命名空间中声明的标识符，需要指定命名空间的名称和标识符，两者之间用作用域解析运算符::分隔开。
- 在某个命名空间中声明的名称，在这个命名空间中使用时，可以不加限定符。
- 预处理器执行预处理器指令，在编译代码之前传送转换单元中的源代码。处理完所有的指令后，转换单元就只包含 C++代码，没有预处理器指令了。
- 可以使用条件预处理器指令，确保头文件的内容在一个转换单元中没有重复。
- 可以使用条件预处理器指令，控制是否在程序中包含跟踪或其他诊断调试代码。
- assert()宏允许在执行过程中测试逻辑条件，如果逻辑条件为假，就输出一个消息，并终止程序。

10.7 练习

1. 有一个程序调用了两个函数 print_this(const string& s)和 print_that(const string& s)，这两个函数又调用第三个函数 print (const string& s)，输出传送给它的字符串。

在 4 个源文件中实现这 3 个函数和 main()函数，并提供 3 个头文件，分别包含 print_this()、print_that()和 print()的原型。确保头文件只包含一次，main.cpp 包含的#include 语句数量最少。

2. 修改上题的程序，使 print()函数使用一个全局整数变量来计算它被调用的次数。在 main()调用 print_this()和 print_that()后，输出这个变量的值。

3. 在 print.h 头文件中，删除 print()的已有原型，再创建两个命名空间 print1 和 print2，每个命名空间都包含一个函数 print (const string& s)。这些函数都有相同的函数签名，区分它们的惟一方式是命名空间的名称。在 print.cpp 文件中实现这两个函数，输出命名空间的名称和字符串。

现在让 print_this()调用在命名空间 print1 中声明的函数，让 print_that()调用在命名空间 print2 中声明的函数。运行程序，验证是否调用了正确的函数。

提示：从上面可以看出，在 print_this()和 print_that()中调用 print()函数有三种不同的方式(即三种不同的语法形式)。

4. 修改 main()例程，只有定义了预处理器符号 DO_THIS，才能调用 print_this()。否则，就调用 print_that()。

修改代码，定义一个宏 PRINT()，在定义了 DO_THIS 后，就让 PRINT(abc)——注意没有引号——调用 print_this("abc")，否则就调用 print_that("abc")。

第 11 章　创建自己的数据类型

C++的长处是其面向对象的特性。这包含许多内容，下面就用 5 章的篇幅来介绍这个主题。本章和第 12 章讨论面向对象编程的基础——定义自己的数据类型。但这绝不仅仅是添加新类型。面向对象编程提供了强大的编程方式，与以前的编程方式完全不同。第 13 章到第 15 章探讨实现自己的数据类型时需要用到的所有 C++技术。

本章介绍对象的一些基本概念，以及使用它们的一些简单方式。这会为学习第 12 章的内容打下基础，第 12 章将更深入地论述面向对象方法的原则，并学习类和定义类的方式。

本章主要内容

- 对象的概念
- 结构的概念和声明结构的方式
- 结构对象的定义和使用方式
- 结构对象成员的处理
- 联合的概念

11.1　对象的概念

C++语言主要利用了面向对象编程的优点，本章就开始理解所涉及到的概念。

对象到底是什么？在某种意义上，这是一个很容易回答的问题，因为对象可以是任何东西。但我们需要花较多的时间才能解释清楚定义某类对象的方式。

在编写程序解决一个问题时，该问题常常描述为各种类型的对象。开发票程序需要使用 customer 对象、invoice 对象、product 对象和 account 对象。跟踪 CD 集合的程序需要涉及 CD 对象和 artist 对象。每种对象都有一组标识它的特性。customer 对象可以由一个名称、一个地址和一个客户号来定义，这些特性也都是对象。CD 对象可以由其标题、艺术家(这是另一个类型的对象)、录制日期和类别(蓝调音乐、摇滚音乐、经典音乐或其他类型)来定义。面向对象的编程方式就是根据问题所特定的对象类型来解决该问题，而不是采用计算机处理数字和字符的低级方式来解决问题。

在 C++编程术语中，对象是数据类型的一个实例。例如，在定义一个变量时，常常使用下面的语句：

```
string saying = "A good horse cannot be of a bad color.";
```

这里定义了 string 类型的一个实例。我们给这个实例指定了名称，称之为 saying。于是可以说，变量 saying 是类型 string 的一个对象。

在 string 类型的定义中，对对象进行的操作已进行了精确的定义。例如，string 对象 saying 可以和算术运算符+、比较运算符<、==和下标运算符[]一起使用。

对于给定的对象，还有一些不能进行的操作，因为这个对象类型没有定义这些操作。string

对象不能进行减法运算，因为没有为它们定义减法运算。

C++为存储数值数据、字符和其他类型(如 string 类型)提供了许多基本的数据类型。我们还可以添加自己的新数据类型，指定可以应用于这些类型的操作。自己定义的类型通常称为用户定义的数据类型，但这并不表示在这些类型上有任何限制。自己的数据类型可以像应用程序所需要的那样复杂，也可以使用基本数据类型所使用的运算符。

类就是用户定义的数据类型，在大多数情况下，定义类时使用关键字 class。

注意：

不要把类和 typedef 语句或枚举的用法混淆了。typedef 语句不创建新类型，只是为已有的类型定义一个别名。每个枚举都是一独特的类型，但它不是一个类。类是全新的、原始的类型，它不仅有一组惟一的属性，还有可应用于类对象的操作，这些完全是由用户定义的。

在介绍类时，会介绍创建定制数据类型的另外两种方法，即联合和结构，分别用 union 和 struct 关键字来创建新类型。从技术上看，它们都是类。我们用关键字 class 把所定义的类型称为类，另外两种用户定义的类型则分别称为联合和结构。

下面先讨论结构的基本概念，之后简要介绍联合，结构非常类似于类，理解结构的基本内容有助于学习类，第 12 章和第 13 章将详细论述类。

11.2 C++中的结构

从历史上看，结构首先应用于 C 编程语言中，它实际上是不同数据类型的数据项的一个指定聚合。例如，把 char*类型的变量 Name、int 类型的变量 Age 和 char 类型的变量 Gender 收集在一起，把这个数据簇存储在结构 Person 中。

在 C++中，结构通常用于相同的目的。但是，C++中的结构可以完成的工作要比 C 中的结构多得多。C++中的结构也添加了为类开发的一些新功能。实际上，C++结构的功能可以用类来替代。

那么，为什么还要花很多的时间和精力来研究结构这种可以被类替代的功能？部分原因是结构与类不完全相同，而且，就像珠穆朗玛峰对于登山者来说永远有吸引力一样，结构是 C++语言的一部分。更重要的是，在一些环境下，结构仍是非常普遍的。而且，学习结构的努力不会白费，因为它们的功能同样可以应用到类上。

11.2.1 理解结构

本书前面介绍的变量和数据类型都是由单个实体组成的——例如某种类型的数值或字符。但是，生命、宇宙，所有的一切都比基本数据类型复杂得多，除非有人相信答案是 42，在这种情况下，就只需要 int。

为了虚拟地描述真实世界中的一切，需要定义几个值，而这些值通常有许多不同的类型。这个理念催生了结构。

例如，考虑一下描述一本书所需要的信息。这需要书名、作者、出版商、出版日期、页数、价格、主题(或分类)和 ISBN 号，还可以列出更多的属性。要在程序中标识一本书，可以指定

不同的变量，每个变量包含这些属性中的一个。但在理想情况下，最好有一个数据类型，称为Book，它包含了所有这些属性。

这就是结构的功能。结构是一种数据类型，它定义了一种特殊类型的对象。

11.2.2 定义结构类型

下面再利用上面列举的书的例子。假定要在书的定义中包含书名、作者、出版商和出版日期，就可以声明一个结构类型，来容纳这些信息，如下所示：

```
struct Book {
   char title[80];
   char author[80];
   char publisher[80];
   int year;
};
```

注意：

这个声明没有定义任何变量，它指定了一个新类型 Book。编译器会把这个定义用作蓝图，来创建类型 Book 的实体。

关键字 struct 声明，Book 是一个结构。组成 Book 类型对象的元素是在花括号中声明的内容。注意在 struct 中，每个元素的定义都用一个分号结束，在右花括号后面也有一个分号，它看起来就像一个普通的声明。定义中包含在花括号中的元素 title、author、publisher 和 year，通常称为结构 Book 的数据成员(更一般的叫法是成员)，Book 类型的每个对象都包含成员 title、author、publisher 和 year。如图 11-1 所示。

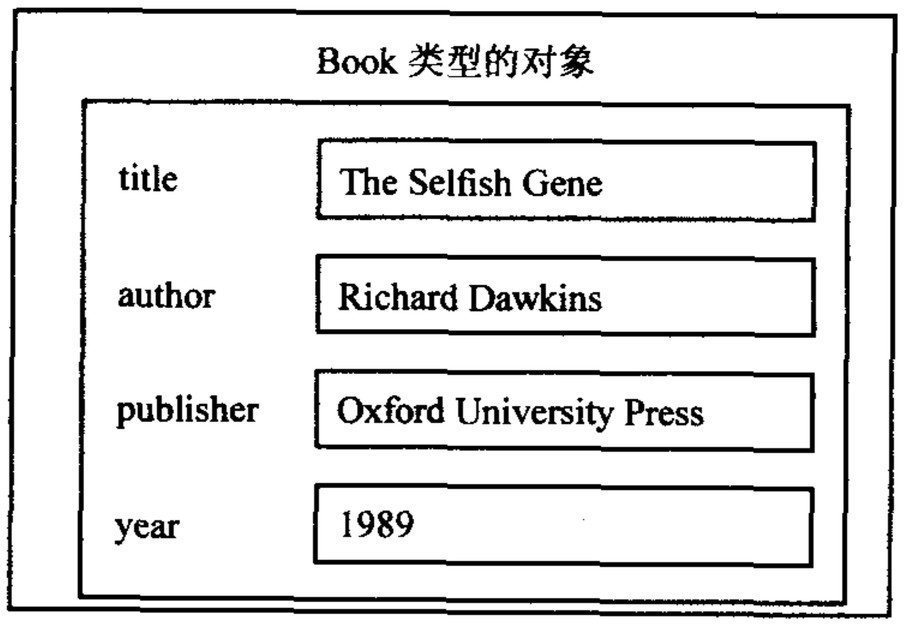


图 11-1 Book 对象的实例

Book 结构的字符串成员定义为类型 char*，因此它们是 C 样式的字符串。一般情况下，最好使用 string 类型，这里选择使用 char*类型是因为 string 类型略为复杂，稍后会讨论 string 类型。

提示：

存储一个结构对象所需要的内存量是存储每个数据成员所需的内存量总和。例如，Book 类型的对象需要足够的内存来包含 3 个字符数组和一个整数。可以使用第 3 章介绍的 sizeof 运算符来计算给定数据类型的对象占用的内存量(第 12 章将讨论这个内容，并涉及到边界对齐的概念)。

structure 的数据成员可以是任何类型，但不能是所定义的结构类型。例如，Book 的结构定义不能包含 Book 类型的数据成员。否则，在初始化 Book 类型的变量时，就需要初始化它包含的 Book 类型的数据成员，该数据成员又包含一个 Book 类型的数据成员，它也需要初始化……实际上，这将陷入一个无限循环。

但是，Book 类型的声明可以包含指向 Book 的指针，这提供了一个非常有用的功能，如后面所述。

声明结构类型的变量

定义了结构 Book 后，下面看看如何定义这种类型的变量。定义 Book 类型的变量与定义其他类型的变量一样：

```
Book paperback;                    //Define variable paperback of type Book
```

这个语句定义了变量 paperback，现在可以用它来存储 book 对象。也可以在定义结构类型的变量时使用关键字 struct。下面的语句也定义了变量 paperback：

```
struct Book paperback;          //Define variable paperback of type Book
```

但是，这是在 C 中的用法，一般不使用。

下面 3 个语句定义了 3 个变量：

```
Book novel;
Book* ptravel_guide;
Book language_guide[10];
```

变量 novel 属于类型 Book 类型，ptravel_guide 是属于类型 pointer to Book，language_guide 是类型 Book 的数组，有 10 个元素。还可以(但不推荐)在一行语句中定义类型 Book 的多个变量。例如，下面的语句也是合法的：

```
Book novel, *ptravel_guide, language_guide[10];
```

与基本类型的变量一样，这个语句也容易出错，且代码不是很清晰。最好使用 3 个定义语句。

最后，可以在定义结构类型时定义变量：

```
struct Book {
   char title[80];
   char author[80];
   char publisher[80];
   int year;
} dictionary, thesaurus;
```

这个语句定义了类型 Book，接着定义了这种类型的两个变量 dictionary 和 thesaurus。这看起来很方便，但是，最好把类型定义放在单独的一个语句中，对象声明放在另一个单独的语句中。一般情况下，在需要使用任何数据类型时，类型定义放在头文件中，并把相应的头文件包含到.cpp 文件中。

11.2.3 创建结构类型的对象

创建 Book 类型的对象的第一种方法是，在初始化列表中为数据成员定义初始值。假定要用一本书的数据初始化变量 novel，可以创建对象，存储在 novel 变量中，初始化它的数据成员，如下面的语句所示：

```
Book novel =
{
  "Feet of Clay ",                    //Initial value for title
  "Terry Pratchett ",                 //Initial value for author
  "Victor Gollanz ",                  //Initial value for publisher
  1996                                //Initial value for year
};
```

初始值放在一对花括号中，用逗号隔开，这种方式与给数组的成员定义初始值一样。显然，初始值的顺序需要匹配结构定义中的成员顺序。这个语句用对应的值初始化对象 novel 的每个成员，如注释所述。名称 novel 现在表示一个特定对象，即数据成员包含指定值的 Book 对象。

结构类型的对象可以用放在花括号中的一组值来初始化，这种类型称为聚合，以这种方式初始化的实体称为聚合。基本类型的数字元素可以用初始化列表来初始化，所以它们也是聚合。

提示：

如果 Book 的成员是类类型，如 string，就不能以这种方式初始化 Book 对象。原因是类类型的对象只能通过调用一个特殊的函数即构造函数来创建。也就是说，结构不是一个聚合。第 12 章将详细论述这个问题。

如果在声明中提供的初始值少于结构变量的数据成员，没有初始值的数据成员就初始化为 0。

也可以初始化结构变量的数组，方法与初始化多维数组一样。把每个数组元素的初始值集放在花括号中，用逗号分隔开即可。下面的语句声明并初始化了一个数组：

```
Book novels[] = {
  { "Our Game ",                  "John Le Carre ",    "Hodder & Stoughton ", 1995},
  { "Trying to Save Piggy Sneed ","John Irving ", "Bloomsbury ",          1993},
  { "Illywhacker ",                "Peter Carey ",   "Faber & Faber " ,   1985}
                  };
```

这个声明创建并初始化了 Book 类型的数组 novels，该数组有 3 个元素。

11.2.4 访问结构对象的成员

要访问结构对象中的各个数据成员，可以使用成员访问运算符，它是一个句点(因此有时称为点表示法)。要引用某个数据成员，可以先写出结构变量的名称，其后跟一个句点，之后是要访问的成员名称。把值 1994 赋给结构 novel 的成员 year：

```
novel.year=1994;
```

可以在算术表达式中使用结构变量的成员，其方法与使用同一类型的其他变量一样。给成员 year 加上 2，使用下面的语句：

```
novel.year += 2;
```

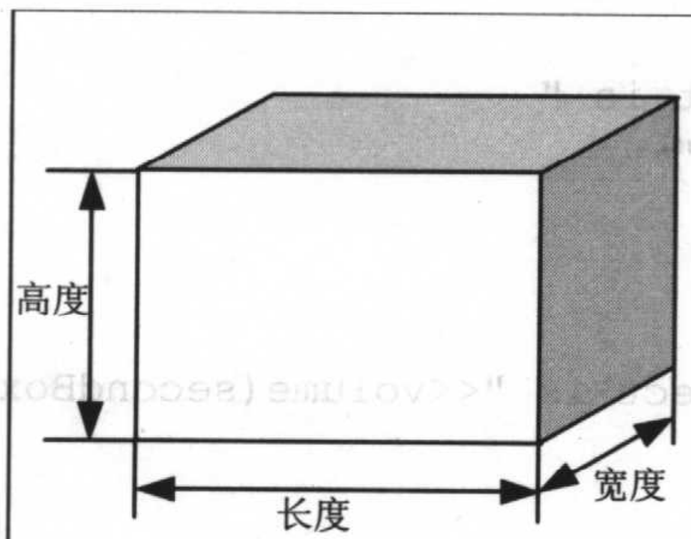
访问结构数组元素的成员也很简单。下面计算数组 novels 中第一本书和最后一本书的出版时间间隔，如下所示：

```
int interval=novels[0].year - novels[2].year;
```

这个语句计算数组中第一个元素和最后一个元素的 year 成员之差。

程序示例 11.1——使用结构

本例需要定义一个非常简单的实际数据项，该数据项需要用多个值来表示，所以把它设计为一个结构。假定要编写一个程序，处理各种尺寸的盒子。它们可以是糖果盒、鞋盒或其他矩形盒子。我们使用 3 个值来定义一个盒子——长度、宽度和高度。结构成员用盒子的物理尺寸来反映这个盒子的外观。下面声明一个结构类型来表示一个盒子，如图 11-2 所示。




```
// Structure to represent a box
struct Box
{
   double length;
   double width;
   double height;
};
```

图 11-2　表示盒子的结构

在程序中使用 Box 结构来创建一些盒子。还要定义并使用一个全局函数来计算 Box 对象的体积：

```
// Program 11.1 Using a Box structure
#include <iostream>
using std::cout;
using std::endl;

// Structure to represent a box
struct Box {
  double length;
  double width;
  double height;
};

// Prototype of function to calculate the volume of a box
double volume(const Box& aBox);

int main() {
```

```
Box firstBox={80.0, 50.0, 40.0};

// Calculate the volume of the box
double firstBoxVolume=volume(firstBox);
cout << endl;
cout << "Size of first Box object is "
     << firstBox.length << " by "
     << firstBox.width << " by "
     << firstBox.height
     << endl;
cout << "Volume of first Box object is " << firstBoxVolume
     << endl;

Box secondBox=firstBox;  // Create a second Box object the same as firstBox

//Increase the dimensions of second Box object by 10%
secondBox. length *=1.1;
secondBox. width  *=1.1;
secondBox. Height *=1.1;

cout << "Size of second Box object is "
     << secondBox.length << " by "
     << secondBox.width << "by "
     << secondBox.height
     << endl;
cout << "Volume of second box object is "<<volume(secondBox)
     << endl;


  cout << "Increasing the box dimensions by 10% has increased the volume by "
       << static_cast<long>
                ((volume(secondBox)- firstBoxVolume)*100.0/ firstBoxVolume)
       << "%"
       << endl;
  return 0;
}

//Function to calculate the volume of a box
double volume(const Box& aBox) {
  return aBox.length* aBox.width* aBox.height;
}
```

编译并运行这个程序，结果如下所示：

```
Size of first Box object is 80 by 50 by 40
Volume of first Box object is 160000
Size of second Box object is 88 by 55 by 44
Volume of second Box object is 212960
Increasing the box dimensions by 10% has increased the volume by 33%
```

例子的说明

在这个程序中，结构定义放在全局作用域中。一般情况下，结构定义在定义它的作用域中

使用。如果 Box 的结构定义放在 main()函数体中，函数 volume()(也在全局作用域中定义)就不能识别参数类型 Box&。

把 Box 结构定义放在全局作用域中，可以在.cpp 文件的任何地方声明 Box 类型的变量。在包含若干个.cpp 文件的程序中，类型定义一般存储在头文件中，再通过#include 指令在每个使用该数据类型的.cpp 文件中包含该头文件。

下面声明函数 volume()的原型，该函数在源文件中定义：

```
double volume(const Box& aBox);
```

参数是一个对 Box 对象的 const 引用。把参数定义为引用，表示在调用函数时不复制原始的 Box 对象。对于复杂的结构类型来说，这是非常重要的(类也是这样，详见第 12 章)，因为复制复杂对象的参数所需要的时间会降低程序的效率。把参数类型定义为 const，是因为函数不应修改参数，而只是使用数据成员。

在 main()中，用下面的语句声明并初始化 Box 对象：

```
Box firstBox={80.0, 50.0, 40.0};
```

花括号中的初始值用于初始化结构中的对应成员，所以 length 是 80，width 是 50，height 是 40。显然，按照正确的顺序获取初始值是很重要的。

下面的语句计算 box 对象 firstBox 的体积：

```
double firstBoxVolume=volume(firstBox);
```

计算出来的体积存储在变量 firstBoxVolume 中，因为程序在后面还需要它。稍后将介绍函数 volume()的工作原理。

在 main()中，下面的语句显示 firstBox 的尺寸：

```
cout << "Size of first Box object is  "
     << firstBox.length << " by "
     << firstBox.width << " by "
     << firstBox.height
     << endl;
```

这个语句访问了 firstBox 的每个成员，其方法是在每个成员名的前面加上了对象名和成员访问运算符，并以通常的方式在输出语句中使用。下一个语句显示前面计算的盒子体积：

```
cout << "Volume of first Box object is " << firstBoxVolume
     << endl;
```

接着，创建一个新的 Box 对象。它与 firstBox 相同：

```
Box secondBox=firstBox;          //Create a second Box object the same as firstBox
```

这个语句创建了一个新的 Box 对象 secondBox，并把它的每个成员值设置为与 firstBox 的对应成员值相等。

下面的语句改变 secondBox 的每个成员值：

```
secondBox.length *=1.1;
secondBox.width *=1.1;
```

```
secondBox.height *=1.1;
```

使用上面 3 个语句给 3 个数据成员分别乘以 1.1。这看起来有点啰嗦，为什么不改写为：

```
secondBox *=1.1;          //Wrong!!!  Won't work!!!
```

编译器会生成一个错误，因为 secondBox 是 Box 类型的对象，编译器不知道应如何对 Box 类型的对象进行乘法运算。另一方面，3 个独立的语句可以编译，因为数据成员都是 double 类型，编译器知道如何对 double 类型的数据成员乘以 1.1。

注释：

通常，如果要对用户定义的类型对象应用运算符，就必须定义运算符的操作方式。只有赋值运算符有默认的定义，而且只用于把用户定义类型的对象赋予该类型的另一个对象。这个赋值运算会逐个成员地把右操作数赋予左操作数。第 14 章会学习如何为自己的类型编写操作。

给 secondBox 的成员递增 10%后，就以与 firstBox 相同的方式输出 secondBox 的尺寸和体积。最后，使用下面的语句计算因这次递增而产生的体积变化：

```
cout << "Increasing the box dimensions by 10% has increased the volume by "
     << static_cast<long>
                ((volume(secondBox)- firstBoxVolume)*100.0/firstBoxVolume)
     << "%"
     << endl;
```

把递增的百分比的值的类型强制转换为 long 类型，使输出变成一个整数百分比。

在继续之前，看看函数 volume()的工作方式。函数 volume()的定义如下所示：

```
double volume(const Box& aBox) {
  return aBox.length* aBox.width* aBox.height;
}
```

把 Box 对象作为一个引用实参传送给函数 volume()，就像其他类型的实参一样。在该函数中，实参用参数名 aBox 来表示。为了计算体积，就把 Box 对象的各个成员相乘。为了访问这些成员，也必须使用成员名和成员选择运算符。这个组合表示在调用该函数时给它传送过去的对象成员。该函数返回得到的乘积。

1. 结构的成员函数

在程序示例 11.1 中，把结构 Box 设计为一个混合的数据项，其中只包含 3 个数据成员。这是结构的一种常见用法——许多程序员都喜欢把结构仅用作数据项的一个集合，在需要更多功能时使用类。但是，C++结构是一个类，所以，C++结构对象也能支持功能。

在程序示例 11.1 中，volume()函数仅与 Box 对象相关。它与其他内容无关。因此，最好把该函数集成到类型中，而不是让它作为一个独立的实体放在全局命名空间中。

可以在 Box 类型定义中定义 volume()函数，使它成为 Box 对象的一个成员，是每个 Box 对象的一个组成部分。下面把 volume()函数定义为结构的一个函数成员：

```
struct Box {
  double length;
  double width;
```

```
  double height;

  //Function to calculate the volume of a box
  double volume() {
    return length*width*height;
  }
};
```

该函数现在成为类型的一个组成部分，只能在 Box 类型的对象中使用该函数。与前面的版本相比，函数有一些重要的变化。没有给函数指定参数，因为不需要参数。在调用这个函数时，它是特定 Box 对象的一个成员，函数体中的操作会直接应用于当前的 Box 对象。函数体也有变化，因为它现在直接引用结构的成员名，在调用函数时，这些成员名表示对象的对应成员。

要给对象调用这个函数，只需以与数据成员相同的方式使用成员选择运算符即可。例如，对 Box 类型的这个定义，可以声明一个 Box 类型的对象 firstBox，接着计算它的体积，如下所示：

```
Box firstBox={80.0, 50.0, 40.0};
double firstBoxVolume= firstBox.volume();
```

这个代码段调用对象 firstBox 的函数 volume()，计算 firstBox 的体积，结果存储在 firstBoxVolume 中。计算中使用的 length、width 和 height 数据成员就是 firstBox 对象的数据成员。

如果为另一个 Box 对象调用这个函数，就使用那个对象的数据成员。例如，要显示 Box 变量 secondBox 的体积，可以使用下面的代码：

```
cout << "Volume of second Box object is "<< newBox.volume()
     << endl;
```

还可以修改程序示例 11.1，使用 volume()成员函数，这样程序的输出不会有变化。

把函数集成到类型中，就改变了函数的性质。每个 Box 对象现在都可以计算它自己的体积。当然，volume()函数只能应用于 Box 对象，如果没有 Box 对象，就不能使用 volume()函数(可以在全局作用域中编写另一个 volume()函数。这两个函数是不会混淆的，因为一个只能在使用 Box 对象时调用，而另一个只能使用函数名来调用)。

2. 放置成员函数的定义

在上面的例子中，成员函数定义包含在结构定义中，也可以把成员函数的声明放在结构定义中，而单独定义成员函数。在这种情况下，结构定义就变成：

```
struct Box {
  double length;
  double width;
  double height;

  double volume();          //Function to calculate the volume of a box
};
```

其中，函数定义与结构定义是分开放置的。在函数定义中，必须告诉编译器要定义的函数

是结构 Box 的一个成员。为此，应使用结构类型名和作用域解析运算符::，其后是函数名。函数定义如下所示：

```
double Box::volume() {
  return length*width*height;
}
```

用类型名称 Box 限定函数名，就表示这个函数属于 Box 结构。其规则与限定命名空间中的名称相同。在函数体中，Box 结构的成员名仍可以不加限定符使用，因为整个函数都用应用于函数名的 Box 限定符定义了。

Box 类型的每个对象都有自己的 volume()函数。但是，无论存在多少 Box 类型的对象，这个函数的代码都仅在内存中保存一次。它们共享该函数的一个副本，所以不会浪费内存。这就出现了一个问题：在执行该函数时，对象的数据成员如何与函数体中使用的数据成员名建立关联？第 12 章深入讨论类时，会给出这个问题的答案。

当然，把结构成员之一的函数定义与结构定义本身分开放置是组织代码的首选方式。在定义新类型时，应把类型定义放在头文件中，在类型定义中也只留下成员函数的声明，把成员函数的定义放在独立的源文件中，例如，对于 Box 结构，把类型定义放在文件 box.h 中，成员函数定义放在文件 box.cpp 中。在源文件中，应总是使用第 10 章介绍的条件预编译器指令，确保不重复头文件的内容。还必须把有类型定义的头文件包括到源文件中，该源文件包含该类型的成员函数定义。

11.2.5 对结构使用指针

如前所述，可以创建一个指针指向结构类型的变量，或指向用户定义的其他类型。例如，要定义指向 Box 对象的指针，声明应如下所示：

```
Box* pBox=0;            //Define a null pointer to an object of type Box
```

这个指针可以保存一个 Box 类型的对象的地址，这里把它初始化为 0。

假定已经定义了一个 Box 对象 aBox，就可以按通常的方法把指针设置为这个变量的地址，即使用地址运算符：

```
pBox=&aBox;             //Set pointer to the address of aBox
```

指针 pBox 现在包含对象 aBox 的地址。

1. 在自由存储区中创建对象

使用 new 运算符，也可以在自由存储区中创建 Book 类型的对象。这个过程与在自由存储区中创建 string 对象相同。下面是一个例子：

```
Book pDictionary = new Book;
```

这个语句创建了指针 pDictionary，它指向在自由存储区中创建的 Book 对象。当然，在使用完这个对象后，必须以通常的方式从自由存储区中删除它：

```
delete pDictionary;
```

对象的成员没有初始化，所以它们包含垃圾值。为了使对象有用，需要显式设置成员的值。为此，需要知道如何通过指针访问对象的成员。

2. 通过指针访问结构成员

假定下面的语句定义了一个 Box 对象：

```
Box theBox = {80.0, 50.0, 40.0};
```

可以声明一个指向 Box 对象的指针，用 theBox 的地址初始化它，如下所示：

```
Box* pBox = &theBox;
```

可以使用指针 pBox 访问它指向的对象的数据成员。这需要两步：首先必须解除指针的引用，获得对象，再使用该对象、成员选择运算符和成员名称。例如，下面的语句递增了 theBox 的数据成员 height：

```
(*pBox).height += 10.0;          //Increment the height member by 10.0
```

在执行这个语句后，height 成员的值就是 60.0，当然，其他的成员没有变化。注意这个语句中的圆括号是必不可少的，因为成员选择运算符的优先级高于解除引用运算符。没有括号，编译器就会把这个语句解释为：

```
*(pBox.height) += 10.0;
```

此语句企图把指针 pBox 当作了结构，访问它的 height 成员，以及要解除对表达式 pBox.height 的引用，pBox 是一个结构，而不是对象，所以这个语句在第一步会失败，不会编译。

3. 指针成员访问运算符

上面介绍了两步操作，即解除对对象指针的引用，再访问该对象的成员。这个两步操作在C++中使用得极为频繁。但是，把*和.运算符组合起来会比较笨拙，而且还需要使用括号，不能立即反映我们的意图。

因此，C++语言提供了一个特殊的运算符，称为指针成员访问运算符，它允许用可读性高且直观的方式来表达这个操作。该运算符看起来像一个箭头，由一个减号和一个大于号组成，专门用于通过指针来访问用户定义类型的对象成员。

下面这个比较笨拙的语句：

```
(*pBox).height += 10.0;          //Increment the height member by 10.0
```

可以使用指针成员访问运算符来表示，如下所示：

```
pBox->height += 10.0;          //Increment the height member by 10.0
```

它很好地表达了访问对象成员的意图。这个运算符还可以用在类类型的对象中，本书的后面将大量使用这个运算符。

如果通过指针 pDictionary 返回自由存储区中的 Book 对象，就可以为其成员设置值，如下所示：

```
pDictionary->title = "The Chambers Dictionary";
```

```
pDictionary->author = "various";
pDictionary->publisher = "Chambers harrap";
pDictionary->year = 1994;
```

之后，就可以使用该对象，但必须在使用完后从自由存储区中删除它。

当然，也可以使用指针成员访问运算符，通过指针来调用结构的成员函数。下面就测试一下对象的指针。

程序示例 11.2——使用对象指针

这个例子涉及三个文件：一个包含 Box 结构定义的头文件，一个包含 Box 结构成员函数定义的源文件，和一个包含 main()的源文件。头文件的代码如下所示：

```
// Program 11.2 Using pointers to Box objects   File:  box.h
// Defines the Box structure type

#ifndef BOX_H
#define BOX_H
struct Box {
  double length;
  double width;
  double height;

  double volume();          //Function to calculate the volume of a box
};
#endif
```

条件预处理器指令可以防止源文件中头文件内容被意外复制。

接着定义 box.cpp，它包含 Box 结构的成员函数定义：

```
// Program 11.2 Using pointers to Box objects   File:  box.cpp
// Defines the Box member function
#include "box.h"

// Box function to calculate volume
double Box::volume() {
  return length*width*height;
}
```

最后，定义 ex11_2.cpp 的内容，它包含函数 main()，对 Box 对象进行一些处理：

```
// Program 11.2 Using pointers to Box objects File: ex11_2.cpp
#include <iostream>
#include "box.h"
using std::cout;
using std::endl;

int main() {
  Box aBox={10, 20, 30};
  Box* pBox=&aBox;          //Store address of aBox
  cout << endl
       << "Volume of aBox is " << pBox-> volume() << endl;
```

```
    Box* pdynBox = new Box;          //Create Box in the free store
    pdynBox->height = pBox-> height +5.0;
    pdynBox->length = pBox-> length -3.0;
    pdynBox->width = pBox-> width -2.0;
    cout << "Volume of Box in the free store is " << pdynBox-> volume() << endl;

    delete pdynBox;
    return 0;
}
```

这个程序的输出结果如下所示：

```
Volume of aBox is 6000
Volume of Box in the free store is 4410
```

例子的说明

在介绍指针的工作方式之前，先注意一下头文件box.h中的Box结构定义只包含了volume()函数的声明。该函数的定义放在box.cpp中。这里头文件box.h的#include指令确保编译器正确连接Box结构，作用域解析运算符::表示函数定义属于Box结构。

在ex11_2.cpp文件中，用#include指令包含标准库头文件iostream和拥有Box定义的头文件。

在main()中创建并初始化对象aBox后，把它的地址存储在指针pBox中：

```
Box* pBox=&aBox;          //Store address of aBox
```

现在可以通过指针pBox和间接成员选择运算符，间接访问aBox的成员了。通过下面的语句，访问aBox的volume()函数：

```
cout <<endl
     << "Volume of aBox is " <<pBox-> volume() << endl;
```

接着，动态创建一个Box对象，把它的地址存储在另一个指针中：

```
Box* pdynBox = new Box;          //Create Box in the free store
```

new返回的地址存储在指针pdynBox中。现在还没有初始化这个新Box对象的数据成员，所以它们包含了垃圾值。使用涉及到aBox成员值的表达式，给这个新对象的数据成员赋值：

```
pdynBox->height = pBox-> height +5.0;
pdynBox->length = pBox-> length -3.0;
pdynBox->width = pBox-> width -2.0;
```

在这3个语句中，使用指针、间接成员选择运算符和成员名引用了每个成员。在Box对象的尺寸有了合适的值后，就计算它的体积：

```
cout << "Volume of Box in the free store is " << pdynBox-> volume()<<endl;
```

最后，从自由存储区中删除Box对象：

```
delete pdynBox;
```

可以看出，使用对象的指针和间接成员选择运算符比较直观，代码也容易阅读和理解。

4. 对象指针的应用

类类型的对象指针非常重要，本节就简要介绍它们的 3 种应用。

第一种应用在程序示例 11.2 中已涉及到。在自由存储区中创建或访问对象时，必须使用指针。程序常常需要存储数量不定的对象，此时最好的处理方法是动态创建对象。

第二种应用是链表。本章前面说过，结构的数据成员不能包含该结构类型的成员，不能把 Book 结构定义为包含 Book 数据类型。但是，结构可以包含指向该结构类型的指针。更进一步则是，结构定义可以包含指向同一类型的结构指针。这有几个用途，一个重要的用途是可以为存储对象集合(即链表)创建一种强大的机制。链表的例子如图 11-3 所示。

图 11-3 演示了给定类型的一组对象如何链接在一起：每个对象都包含一个指针成员，该指针包含了下一个对象的地址。只要知道链中第一个对象的地址，就可以沿着指针成员链逐个对象地查找，获得其他对象。第 13 章在讲述类时将详细论述这一内容。

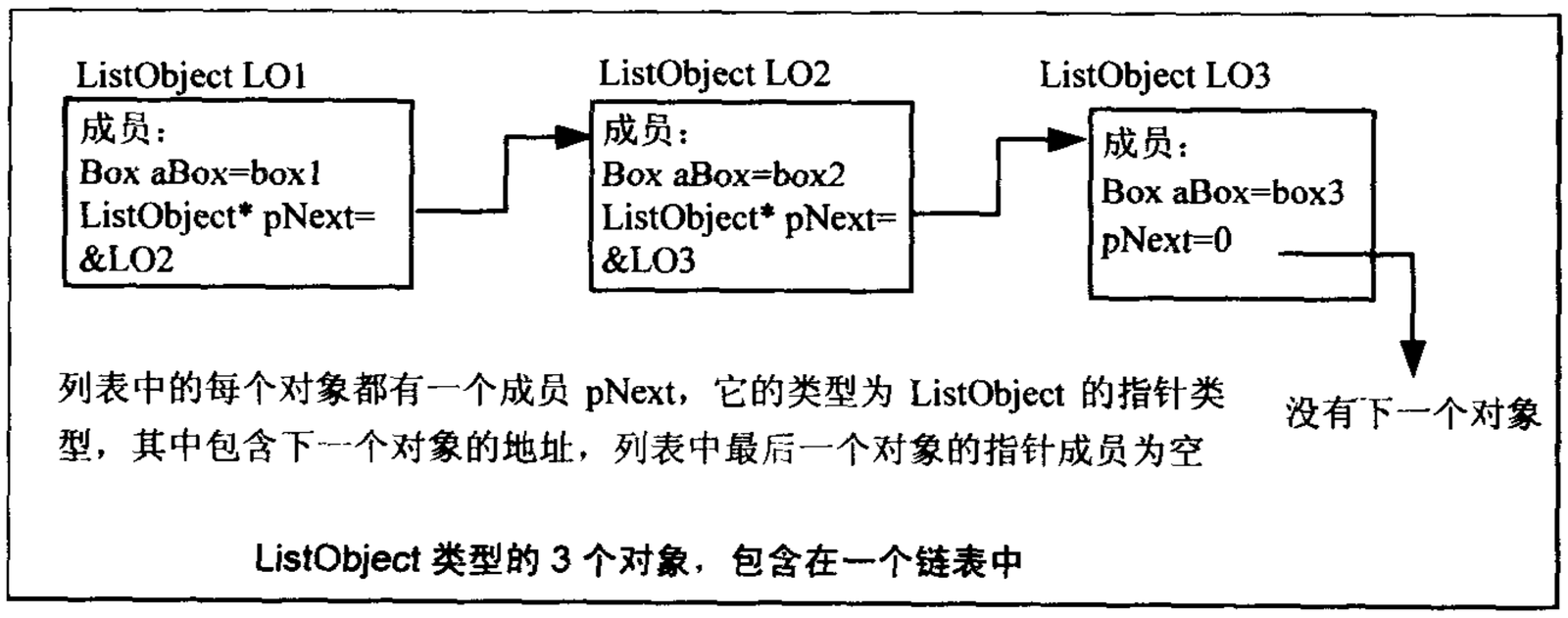


图 11-3 链表的例子

对象指针最重要的应用是实现多态性。多态性是面向对象编程的一个基本机制，详见第 15 章。

11.3 联合

联合是一种数据类型，它允许使用同一个内存块在不同的时间存储不同类型的值。联合中的变量称为联合的成员。使用联合基本上有 3 种方式。

首先，可以使用联合在同一个内存块中存储程序中不同时期的不同变量(其类型也可能不同)。最初的想法是在内存的容量有限且非常昂贵时，使内存的使用更经济。但是，目前已经不存在这种情况了，不值得为了节省内存而冒出现错误的风险。因此不推荐以这种方式使用联合——动态分配内存也可以获得相同的效果。

第二个应用涉及到数组，主要是为了节省大量的内存。假定数据需要存储在一个非常大的数组中，但在执行之前不知道数据的类型是什么——这由输入的数据确定。联合可以把几个不同基本类型的数组作为成员，每个数组都占用同一个内存块(而不是每个数组占用不同的内存区域)。在编译期间，无论输入什么类型的数据，总是可以存储在对应类型的数组中。在执行期间，程序会使用合适的数组来存储所输入的数据。未使用的数组不会占用任何空间。但在这种情况下最好不要使用联合，因为使用不同类型的指针可以获得相同的效果，且内存是动态分配的。

第三，可以使用联合以两种或更多种不同的方式来解释相同的数据。例如，假定有一个 int 类型的变量，可以把它看作是 char 类型的 4 个字符，颜色值就是这样一个例子。联合可以把表示颜色的 int 值看作 4 字节的整数，在复制或处理它时，把它当作一个单元，在需要时则把可以单独访问这四个字节。还可以使用联合来做相反的工作，即把字符串看作一系列整数，以生成某种键。联合会使这两种情况非常容易处理。

不要被联合的这个第三种应用误导。可以通过不同类型的变量访问同一个数据，这并不表示不进行类型转换。实际上，相同的位模式可以用不同的方式来解释，且不检查数据的有效性。显然，在许多情况下，这可能引起灾难性的后果。

一般情况下，联合的成员不是类类型，除非它是一个聚合，即可以用初始化列表进行初始化的对象。尤其是，string 对象不能是联合的一个成员。在大多数情况下，联合的数据成员是基本类型的变量。

下面看看如何声明一个联合。

11.3.1 声明联合

如前所述，联合是通过使用关键字 union 来声明的。下面通过一个具体的例子来说明：

```
union ShareLD {          //Sharing memory between long and double
  double dVal;
  long lVal;
};
```

这个语句声明了一个数据类型 ShareLD，它为 long 类型的变量和 double 类型的变量提供了同一个内存空间。联合类型名有时称为标记名。这个语句很像结构的声明，实际上，它就是一种结构，只是所有的成员都占用相同的内存空间。我们还未定义联合实例，因此此时没有任何变量。现在声明 ShareLD 联合的一个实例 myUnion，如下所示：

```
ShareLD myUnion;
```

也可以在联合定义语句中定义 myUnion，如下所示：

```
union ShareLD {          //Sharing memory between long and double
  double dVal;
  long lVal;
} myUnion;
```

如果要引用联合的成员，应使用直接成员选择运算符(句点)和联合实例名，就像访问类的成员一样。下面的语句把联合实例 myUnion 中的 long 变量 lVal 设置为 100：

```
myUnion.lVal=100;          //Using a member of a union
```

注释：

使用联合在同一个内存区域中存储不同类型的值时，会遇到一个基本问题。由于联合的工作方式，还需要确定哪个成员值是当前值。通常，这一问题是通过另一个变量来解决的，这个变量的作用类似于所存储值的类型的指示器。

联合可以根据需要包含两个以上的数据成员。下面定义另一个联合，让几个变量共享一个内存位置。联合会占用足够的内存空间来存储其最大的成员。例如，假定声明下面的联合：

```
union ShareDLF {
  double dVal;
  long lVal;
  float fVal;
} ;
```

用下面的语句创建这个联合的一个实例：

```
ShareDLF uinst={1.5};
```

注意初始化联合值的方式。在声明语句中初始化联合时，总是初始化联合的第一个数据成员。在本例中，数据成员 dVal 就包含值 1.5。如果要初始化成员 fVal，就必须把联合的声明语句和赋值语句分开编写：

```
ShareDLF uinst;
uinst.fVal=1.5;
```

在计算机上，uinst 会占用 8 个字节，如图 11-4 所示。

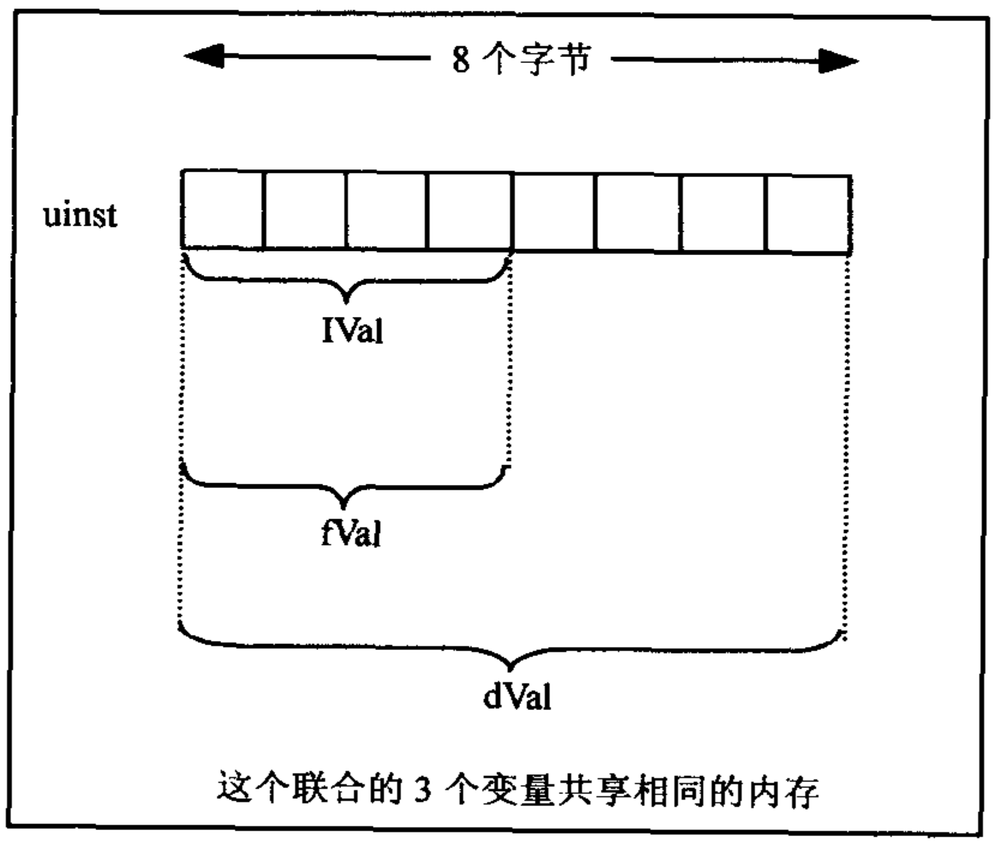


图 11-4　共享联合中的内存

显然，联合不是一种能在不同机器上移植的机制，因为各种数值类型占用的字节数不是固定的。

11.3.2　匿名联合

在声明联合时可以不给出联合类型名，在这种情况下，会自动定义联合的一个实例。例如，定义如下的联合：

```
union  {          // An anonymous union
  char* pVal;
  double dVal;
  long lVal;
} ;
```

这个语句声明了一个没有类型名称的联合，还定义了该联合的一个实例，这个实例也没有名称。因此，联合的成员只用联合定义中的成员名来引用。这比有类型名的联合更方便，但需要特别小心，不要把联合的成员和一般的变量相混淆。联合的成员仍共享同一个内存。

下面举例说明匿名联合是如何工作的。为了使用 double 成员，可以使用下面的代码：

```
dVal=99.5;              //Using a member of an anonymous union
```

这里并没有说明，所使用的变量 dVal 是一个联合成员。如果需要使用匿名的联合，就可以使用某个命名约定来明确联合的成员。

注意，如果声明了没有名称的联合，但在同一个语句中又声明了该类型的对象，联合就不是匿名的。例如：

```
union {
  char* pVal;
  double dVal;
  long lVal;
}uvalue;
```

在本例中，不能用联合成员名来引用它们自身，因为不会存在无名称的联合实例，即有一个 uvalue 实例。可以使用下面的语句：

```
uvalue.dval=10.0;
```

但不能仅使用成员名：

```
dval=10.0;          //Error!! Unnamed union instance does not exist!!
```

11.4 更复杂的结构

前面介绍的结构都比较简单，尤其是在结构的成员中仅使用基本数据类型时。事实上，结构的数据成员可以是任何类型，包括联合和其他结构。下面看一个例子。

假定为了节省内存，必须让变量共享内存。下面定义一个联合，让几个不同类型的变量共享内存中的同一空间：

```
union item {
  double dData;
  float fData;
  long lData;
  int iData;
};
```

类型 item 的变量可以存储 double、float、long 或 int 类型的值，但在任意时刻只能存储一种类型的值。使用该联合是很容易的，如下所示：

```
Item value;             // Create Item instance
value.dData=25.0;
```

这个语句在 dData 中存储了浮点数值。稍后使用下面的语句：

```
value.lData=5;
value.lData++;
```

这个语句重写了浮点数值，在 lData 中存储了数值 5。有时使用这个方法有点恐惧。一个微小的错误——例如假定 item 类型的变量包含 long 值，而实际上它存储了一个 double 值——就会导致灾难性的后果。如果必须采用这种方式，就至少要提供某种检查类型的方式。

为了标识所存储值的类型，可以使用一个枚举：

```
enum Type{Double, Float, Long, Int};
```

现在，可以声明一个结构，它有两个成员。第一个成员是一个联合，它可以存储 double、float、long 或 int 类型的变量。第二个成员是 Type 类型的枚举变量，这个结构的定义如下所示：

```
struct SharedData {
  union  {                    //An anonymous union
    double dData;
    float fData;
    long lData;
    int iData;
  };
  Type type;                  //Variable of the enumeration type Type
};
```

注意，SharedData 的联合成员是匿名的。这表示在引用 SharedData 类型的对象的联合成员时，不必指定联合名。下面声明一个 SharedData 类型的变量：

```
SharedData value={25.0, Double};     //Initializes dData to 25.0
                                     //and type to Double
```

这个语句把 dData 的值初始化为 25.0，因为 dData 是联合的第一个成员(因为是在声明语句中进行的初始化，所以编译器只允许初始化联合的第一个成员 dData)。初始化列表中的第二个值是 type 的值。只能指定在枚举 Type 中声明的一个值。

注意可以省略 type 的初始值，结果也是正确的，因为默认值 0 对应于 Double：

```
SharedData value={25.0 };     //Initializes dData to 25.0 and type to Double
```

这个语句也是对的，因为在 Type 的声明中，Double 是枚举列表中的第一个成员。

接着用下面的语句设置 value：

```
value.lData=10;
value.type=Long;
```

只要每次在修改值时都设置类型，就可以测试所存储值的类型，以确定如何使用它。例如：

```
if(value.type==Long)
  value.lData++;
```

提示：

这个例子主要是说明如何通过使用结构，把多个相关的数据绑定在一起。当然，在这个假想的情况中，整个过程的效率都不高，因为数据共用了一个相当小的内存区域——类型 double 一般占用 8 个字节。每次测试类型时，所使用的内存要比不同类型之间共享内存所节省下来的还要多！

把结构作为成员

刚才是把联合作为结构的一个成员。如前所述，结构也可以是结构类型中的数据成员(假定这两个结构类型是不相同的)。假定要用一个类型表示一个人，记录这个人的一些个人信息，例如姓名、住址、电话号码、生日等。下面看看如何存储这些数据项。

电话号码可以存储为一个整数，但电话号码应与区号分隔开来。如果要记录海外人士的信息，甚至还需要国家代码。所以存储电话号码就需要一个结构。显然，Person 结构的大多数成员本身就是结构。下面列举一个简化版本。

Person 结构的定义如下所示：

```
struct Person {
  Name name;
  Date birthdate;
  Phone number;
};
```

定义一个 Person 类型只需要 3 个数据成员，但每个数据成员本身都是一个结构。存储姓名的类型如下所示：

```
struct Name {
  char firstname[80];
  char surname[80];
};
```

提示：

当然，这里最好把 string 对象用作 Name 结构的成员，但如果这样，初始化 Name 对象就需要一个构造函数。第 12 章将介绍构造函数，这里仍使用老式、笨拙的以空值结束的字符串。

对于 Date 结构，可以把日期记录为 3 个整数，分别对应于某个月中的日期、月份和年份：

```
struct Date {
  int day;
  int month;
  int year;
};
```

对于 Phone 结构，可以把电话号码存储为一个字符串，因为这比较便于拨号。但是在此，则把区号和电话号码存储为两个整数：

```
struct Phone {
  int areacode;
  int number;
};
```

声明 Person 类型变量的方式与本章前面声明结构对象的方式相同：

```
Person him;
```

在声明 Person 类型的变量时，可以使用一个初始化列表：

```
Person her={
            { "Letitia ", "Gruntfuttock "},     //Initializes Name member
            {1, 4, 1965          },             //Initializes Date member
            {212, 5551234        }              //Initializes Phone member
          };
```

初始值列表的排列方式与多维数组类似。对应于每个成员的列表包含在花括号中，每个列表用逗号隔开。每个结构成员的列表不一定要放在花括号中，如果省略花括号，值就会按顺序赋予相应的成员。当然，如果省略花括号，出错的可能性就比较大，而且不能省略内层结构中的任何成员值。

Person 类型的变量处理起来有一定的限制。可以使用赋值语句使一个 Person 对象的成员与另一个 Person 对象的成员相同：

```
Person actress;
actress =her;          //Copy members of her
```

在本例中，her 的成员(以及内层结构的成员)会复制到 actress 的对应成员中。

当然，可以使用成员选择运算符来引用成员的值。例如，下面的语句可以输出 her 的姓名：

```
std::cout << her.name.firstname << "  " << her.name.surname << std::endl;
```

以这种方式显示姓名有点麻烦，此时可以定义一个成员函数。下面用一个简单的例子来实现。

程序示例 11.3——使用 Person 结构

定义 Person 成员的结构可以把函数包含为其成员。下面就给这些内层结构添加一个成员，以输出它们的成员。还可以给 Person 结构添加一个函数，把四个定义放在一个头文件 person.h 中：

```
// person.h Definitions for Person and related structures
#ifndef PERSON_H
#define PERSON_H

// Structure representing a name
struct Name {
  char firstname[80];
  char surname[80];

  void show();              // Display the name
};

// Structure representing a date
struct Date {
  int day;
  int month;
  int year;

  void show();              // Display the date
};

// Structure representing a phone number
```

```
struct Phone {
  int areacode;
  int number;

   void show();        // Display a phone number
};

// Structure representing a person
struct Person {
   Name  name;
   Date birthdate;
   Phone number;

    void show();           // Display a person
   int age(Date& date);      // Calculate the age up to a given date
};
#endif
```

函数 show()输出 Person 对象的信息，除了这个函数之外，还添加了一个成员函数 age()，把当前日期传送给它，计算人的年龄。

在 Person 结构的定义中引用这些结构定义时，必须把它们放在文件的前面。

可以把结构成员函数的实现代码放在一个源文件 person.cpp 中：

```
// Program 11.5  Working with a person.    File:  person.cpp
#include <iostream>
#include "person.h"
  // Display the name
  void Name::show() {
    std::cout << firstname << " " << surname<<std::endl;
  }

  // Display the date
  void Date::show() {
    std::cout << month << "/" << day << "/" << year<<std::endl;
  }

  // Display a person number
  void Phone::show() {
    std::cout << areacode << " " << number<<std::endl;
  }

// Display a person
void Person::show() {
   std::cout << std::endl;
   name.show();
   std::cout << "Born: ";
   birthdate.show();
   std::cout << "Telephone:  ";
   number.show();
}

// Calculate the age up to a given date
```

```
int Person::age(Date& date) {
   if(date.year <= birthdate.year)
      return 0;

   int years = date.year - birthdate.year;
   if((date.month>birthdate.month)  ||
            (date.month == birthdate.month && date.day>= birthdate.day))
   return years;
     else
   return --years;
  }
};
```

现在，在一个小程序中使用该结构：

```
// Program 11.3 working with a Person
#include <iostream>
#include "person.h"
using std::cout;
using std::endl;

int main() {
   Person her = {
                { "Letitia",  "Gruntfuttock" },  // Initializes Name member
                {1, 4, 1965                  },  // Initializes Date member
                {212, 5551234                }   // Initializes Phone member
              };

Person actress;
actress = her;                                   // Copy members of her
her.show();
Date today = { 15, 6, 2003 };

cout  <<  endl  <<  "Today is  ";
today.show();
cout <<  endl;

cout << "Today" << actress.name.firstname <<" is"
     << actress.age(today)  << " years old."
     << endl;
return 0;
}
```

这个程序的输出结果如下所示：

```
Letitia Gruntfuttock
Born: 4/1/1965
Telephone: 212 5551234

Today is 6/15/2003
Today Letitia is 38 years old.
```

例子的说明

首先用前面的语句创建并初始化一个 Person 对象 her，然后用一个赋值语句把 her 的成员复制到 Person 对象 actress 中。为了输出 her 的信息，对 her 对象调用 show()函数：

```
her.show();
```

在执行 show()函数时，会访问 her 对象的 name、birthdate 和 number 成员。这个函数使用这些对象的 show()函数，以显示 Person 对象 her 的全部信息。

下面的声明定义了一个新的 Date 对象：

```
Date today={15, 6, 2003};
```

然后，下面的语句输出该对象表示的日期：

```
cout << endl << "Today is ";
today.show();
cout << endl;
```

或者，也可以用下面的语句显式地显示 today 对象的每个成员：

```
cout << endl << "Today is" << today.month << "/"
                           << today.day << "/"
                           << today.year << endl;
```

最后，使用下面的语句输出对象 actress 的年龄：

```
cout << "Today " <<actress.name.firstname << " is"
     << actress.age(today) << " years old. "
     << endl;
```

这个语句用表达式 actress.name.firstname 直接访问 actress 中 name 成员的一个成员。它还使用 actress 对象的 age()函数计算到 today 表示的日期为止的年龄。

在这个例子中，论述了如何访问结构中本身就是另一个结构的成员，如何给结构添加函数，使对象更容易处理。注意，我们需要一种更好的方式创建对象，给其成员赋值。这些讨论将有助于进一步理解类对象。把功能内置于类型中是面向对象编程的基础；本章所介绍的功能仅是下面两章的冰山一角。

11.5 本章小结

本章通过结构介绍了使用用户定义的数据类型和对象进行编程的一些主要方面。第 12 章将在这个基础上，进一步探讨面向对象的技术和原则。本章的主要内容如下：

- 结构类型是程序中的一个新数据类型。
- 结构对象是带有成员的对象，这些成员在默认情况下可公开访问。结构可以有数据成员和函数成员。
- 可以使用对象名和句点分隔的成员名来引用结构对象的成员。其中句点称为成员选择运算符。

- 联合是一种数据类型，它的对象可以使用同一个内存块在不同的时刻存储几种不同变量的值(也许类型也不同)。
- 在声明联合对象时，只能为联合的第一个成员提供对应类型的初始值。
- 结构的数据成员可以是任意类型，包括其他结构，但数据成员的类型不能与包含它的结构的类型相同。
- 聚合是创建时可以用花括号中的初始值列表来初始化的实体。
- 可以在自由存储区中动态创建对象，但必须在指针中存储这些对象的地址。
- 可以使用间接成员选择运算符访问对象的成员。

11.6 练习

1. 编写一个简单的货币转换程序。为此，需要在货币对象中关联两个实体：货币类型和把货币转换为美元的转换因子。设计一个结构来表示货币对象，编写一个程序，让用户从一个列表中选择转换的货币类型，在任意两种货币之间转换。用户应输入值，并获得转换后的结果，如果输入了一个负值，就退出程序。

2. (较难)提供一种方式，让用户在运行程序时添加新的货币种类。

3. 实现本章中“更复杂的结构”一节中描述的 SharedData 结构。扩展该结构(及其相关的枚举类型)，以存储 4 种类型的指针。测试一下，看看是否能存储变量的指针。

4. 编写一个函数，它接受 SharedData 对象的数组，并以[array_element]type=value 形式输出每个元素的值，例如：

```
[0] double=37.2
[1] float*=2.5
```

在合适的 main()例程中测试这个函数。

第 12 章　类

本章将详细讨论定义为结构的类型，详细论述 C++程序员工具箱中最基础的工具之一：类。并介绍面向对象编程中的一些隐含规则，讨论如何将它们应用于实践。

本章主要内容

- 面向对象编程的基本原则
- 如何把新的数据类型定义为类，如何使用类的对象
- 什么是类的构造函数，如何编写构造函数
- 默认的构造函数是什么，如何提供自己的默认构造函数
- 默认的复制构造函数
- 友元函数
- 友元类的权限
- this 指针的概念、使用它的方式和场合

12.1　类和面向对象编程

在讨论类的语言、语法和编程技巧之前，先考虑一下现有的知识与面向对象编程概念之间的关系。

面向对象编程(通常缩写为 OOP)的本质是，根据要解决的问题范围内所涉及到的对象来编写程序，因此开发过程的一部分是设计一组类型来满足这个要求。如果要编写一个程序，来跟踪银行账户，就需要 Account 和 Transaction 等数据类型。对于分析篮球比分的程序来说，需要 Player 和 Team 等数据类型。基本类型的变量不允许为真实世界中的对象(甚至想像中的对象)建立完整的模型。也不可能仅用 int、double 或其他基本数据类型给篮球队员建模。我们需要使用几个不同类型的值，对篮球队员进行有意义的表述。

结构提供了一种可能的解决方案。前面介绍了如何定义一个结构类型，它组合了几种不同类型的变量。结构类型还可以把函数作为其定义的一个组成部分。第 11 章定义了一个结构 Box 来表示盒子，有了这个新数据类型，就可以定义 Box 类型的变量，就像定义基本类型的变量一样。接着在程序中创建并处理任意多个 Box 对象。这距离按照真实世界中的对象编程还有很长的一段路。显然，可以把结构的理念用来表示篮球队员、银行账户或其他需要表示的东西。使用结构可以为任何类型的对象建模，并围绕这些对象来编程。那么，这就是面向对象编程所赋予我们的全部内容吗？

当然不是。前面定义的结构已向前迈出了一大步，但前面还是有一段距离。除了用户定义的类型这个理念之外，面向对象编程还隐含地组合了许多更重要的理念(最著名的有封装和数据隐藏、继承和多态性)。结构并没有涵盖全部内容。下面用浅显而直观的方式来说明这些 OOP 概念的含义。这将为详细论述 C++编程提供一个参考平台，也是本章和后面的 4 章将详细论述的内容。

12.1.1 封装

一般情况下，给定类型的对象定义需要组合一些不同的内容，使该对象成为我们希望的那个实体。对象包含一组特定的数据值，来详细描述该对象，以符合我们的要求。对于一个盒子，它只能有 3 个尺寸：长度、宽度和高度。对于航空母舰，就需要更多的数据来描述。对象还包含一组函数，这些函数可以使用或改变对象的数据值。它们定义了可以应用于对象的一组操作，即可以对对象做什么，或不能做什么。给定类型的每个对象都组合了下述内容：一组数据值，作为数据成员；一组操作，作为函数成员。

把这些数据值和函数打包到一个对象中，就称为封装。图 12-1 是一个例子，其中的对象表示银行的一个贷款账户。

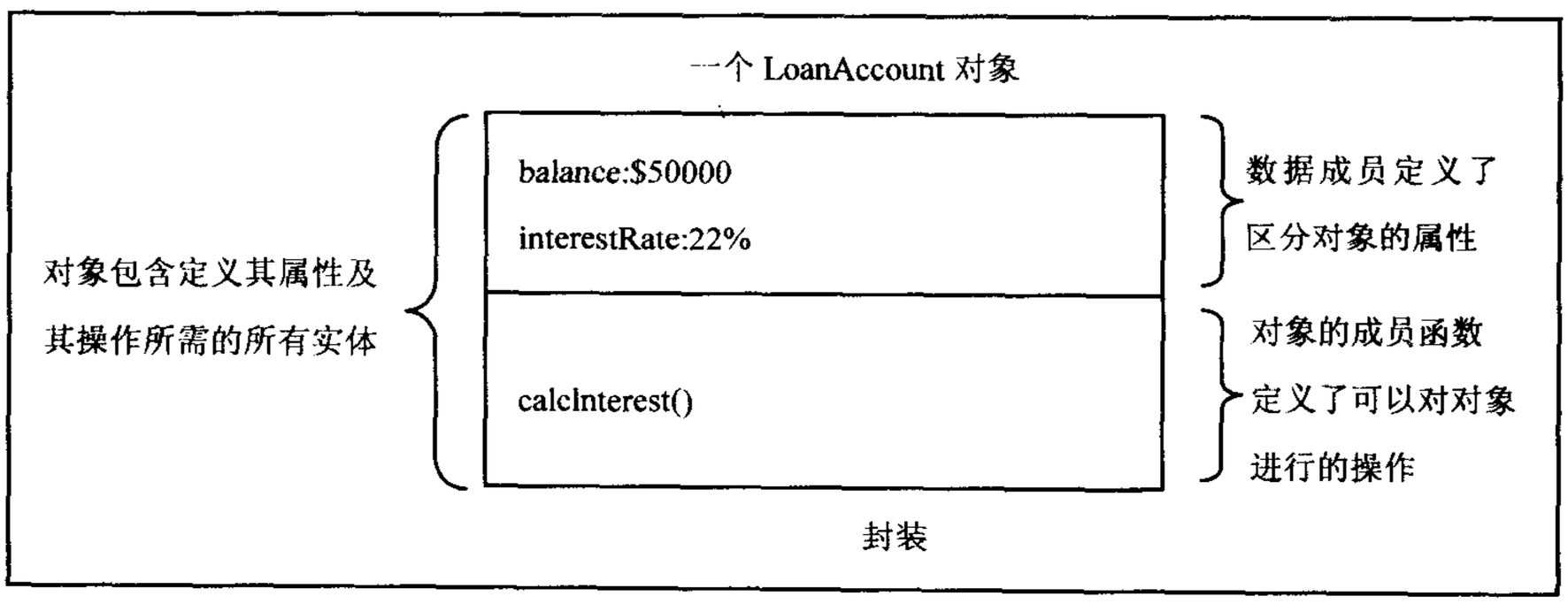


图 12-1　封装的例子

每个 LoanAccount 对象都有用同一组数据成员定义的属性，在本例中，一个数据成员包含债务余额，另一个数据成员包含利率。每个对象还包含一组成员函数，它们定义了对象上的操作：图 12-1 中的函数计算利息，并把它加到余额上。属性和操作都封装在 LoanAccount 类型的每个对象中。当然，选择这样建立 LoanAccount 对象是很随意的。读者也可以根据自己的需要定义完全不同的 LoanAccount 对象，但无论怎样定义 LoanAccount 类型，所指定的所有属性和操作都会封装到该类型的每个对象中。

注意前面说过，定义对象的数据值需要满足我们的要求，而不是满足一般情况下定义对象的要求。如果要编写一个地址簿应用程序，一个人可以定义得非常简单，只有姓名、地址和电话号码。但如果要把一个人定义为公司职员或医院的患者，就需要定义非常多的属性和操作。我们需要决定要在什么环境下使用对象。

数据隐藏

当然，银行不希望贷款账户的余额(或利率)在对象的外部被随意修改，就像在第 11 章随意修改结构对象那样。为了防止进行这种修改，就会为这种情况开出一个处方。在理想情况下，LoanAccount 对象的数据成员应不直接受外界的干扰，而只能以可控制的方式来修改。在一般情况下，不允许访问对象的数据值，这称为数据隐藏。图 12-2 把数据隐藏应用于 LoanAccount 对象。

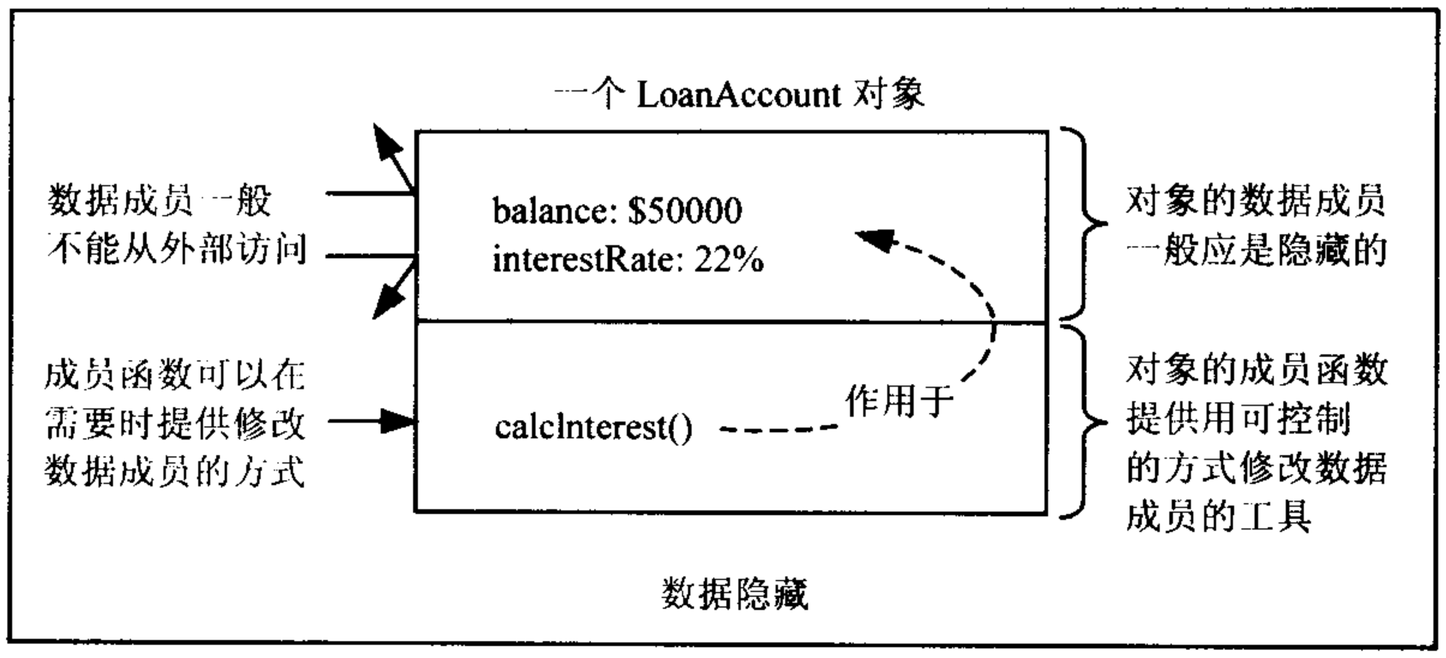


图 12-2　数据隐藏的例子

对于 LoanAccount 对象，对象的成员函数可以提供一种机制，确保对数据成员的所有修改都遵循某个规则，且所设置的值是合适的。例如，利息不应是负的，余额一般应反映属于银行的钱，而不是反过来。

数据隐藏非常重要，因为它对维护对象的完整性是必不可少的。如果对象用于表示一只鸭子，就不应有四条腿，实现这一点的方法就是不允许访问腿的总数，即“隐藏”数据。当然，对象可以拥有多种合法的数据值，但必须控制值的范围；毕竟，一只鸭子通常不会有 300 磅重。隐藏对象中的数据，可以禁止直接访问该数据，但可以通过对象的函数成员来访问。例如以可控制的方式修改数据值，或获取数据的值。这种函数可以在需要时，检查数据所进行的修改是否合法，是否在指定的范围内。

在对象中隐藏数据不是必须的，但一般情况下，这是一个比较好的方法。其原因如下：首先，维护对象的完整性要求控制所进行的修改。第二，直接访问定义对象的值会破坏面向对象编程的整体理念。面向对象编程是根据对象来编程，而不是根据组成对象的位来编程。

数据成员表示对象的状态，操纵它们的成员函数则表示对象与外界的接口。使用类涉及到用声明为接口的函数进行编程。使用类接口的程序仅依赖于函数名、参数类型和为该接口指定的返回类型。这些函数的内在机制不影响程序的创建和使用类的对象。也就是说，在设计阶段，获取类的接口非常重要，但以后修改核心内容的实现方式时，不需要对使用类的程序进行任何修改。

12.1.2　继承

继承是根据一个类型定义另一个类型的能力。例如，假定定义了一个 BankAccount 类型，它包含的成员可以处理银行账户的许多事务。而继承允许把类型 LoanAccount 创建为 BankAccount 的一个特殊类型，即把 LoanAccount 定义为像是一个 BankAccount，但它有一些额外的属性和自己的函数。LoanAccount 类型继承了 BankAccount 的所有成员，BankAccount 就称为它的基类。LoanAccount 派生于 BankAccount。

每个 LoanAccount 对象都包含了 BankAccount 对象的所有成员，它还可以定义自己的新成员，或重新定义继承下来的函数，使它们在自己的环境下更有意义。这个功能非常强大，详见后面的内容。

扩展刚才的例子，再创建一个新类型 CheckingAccount，它给 BankAccount 添加了新特性。

这个过程如图 12-3 所示。

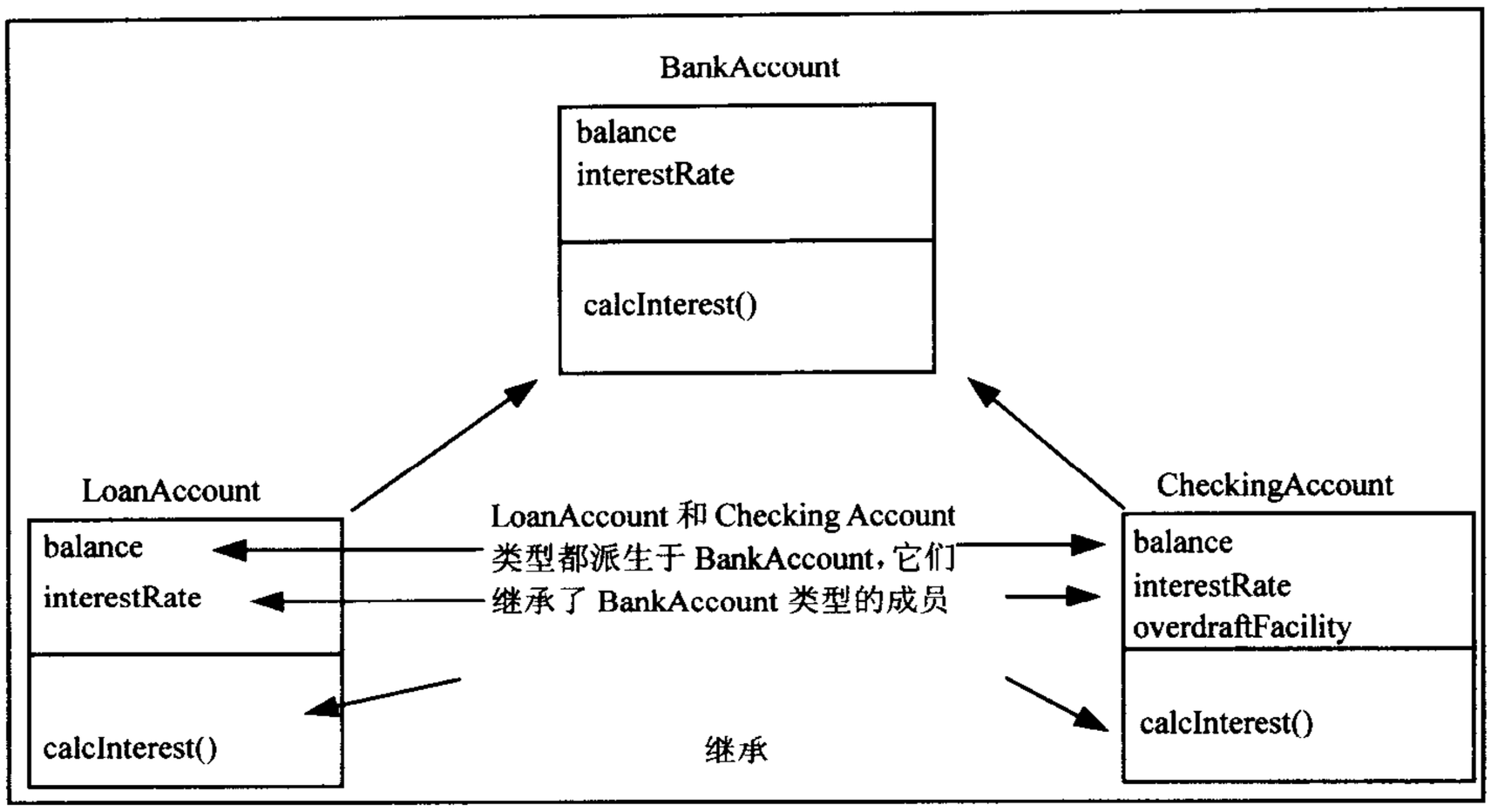


图 12-3　继承的例子

LoanAccount 和 CheckingAccount 类型都声明为它们派生自 BankAccount 类型，它们继承了 BankAccount 的数据成员和成员函数，并可以自由定义适合于自己类型的新特性。

在这个例子中，CheckingAccount 添加了一个数据成员 overdraftFacility，这是该类型惟一的数据成员，另外，两个派生的类都重新定义了从基类继承而来的 calcInterest()成员函数，这是可行的，因为计算和处理支票账户的利息所涉及的事务与贷款账户有所不同。

12.1.3　多态性

多态性表示在不同的时刻有不同的形态。C++中的多态性总是涉及到使用指针或引用来调用对象的成员函数。这种函数调用在不同的时刻有不同的效果——函数调用有多种不同的形式。这种机制仅适合于派生于通用类型的对象，例如 BankAccount 类型。多态性意味着，属于一组继承性相关的类的对象可以通过基类指针和引用来传送和操作。

在上面的例子中，LoanAccount 和 CheckingAccount 对象都可以使用 BankAccount 的指针或引用来传送。而该指针或引用可以用于调用它指向的对象所继承的成员函数。下面用一个例子来说明这个概念。

假定在 BankAccount 类型的基础上定义了 LoanAccount 和 CheckingAccount 类型，并定义了这些类型的对象 debt 和 cash，如图 12-4 所示。由于这两个类型都基于 BankAccount 类型，因此指向 BankAccount 的变量类型(如图 12-4 中的 pAcc)就可以用于存储这两个对象的地址。

提示：

图 12-4 中的代码使用第 11 章介绍的表示法，通过指针调用对象的函数成员。

多态性的优点是通过 pAcc->calcInterest()调用的函数会根据 pAcc 指向的对象发生变化。如果它指向 LoanAccount 对象，就调用该对象的 calcInterest()函数，利息就是从账户中借的。如果它指向 CheckingAccount 对象，结果就完全不同，因为会调用该对象的 calcInterest()函数，利息会加到账户中。通过指针调用哪个函数并不是在编译程序时确定的，而是在程序执行时才确

定。因此，同一个函数调用会根据指针指向的对象完成不同的操作。图 12-4 只显示了两种类型，但一般可以实现应用程序需要的任意多个不同类型的多态操作。

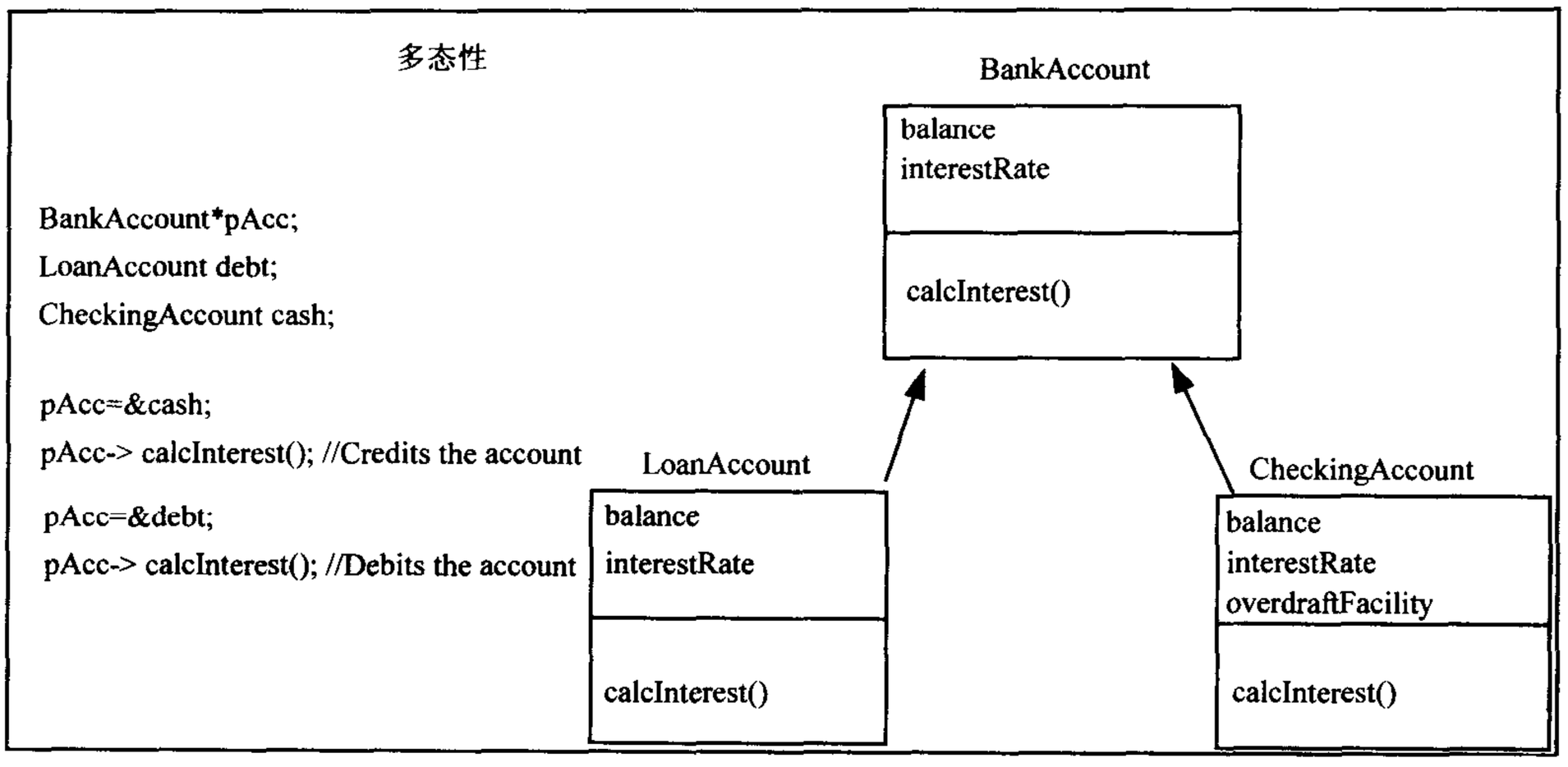


图 12-4　多态性的例子

读者需要具备相当多的 C++语言知识才能理解前面讲述的内容，本章和后面 4 章就探讨这些知识。第 16 章会在程序中使用多态性。首先介绍如何使用关键字 class 来定义新类型。

12.1.4　术语

下面总结一下在讨论 C++中的类时要用到的术语，其中包括前面已介绍过的一些术语：

- 类是用户定义的数据类型。
- 在类中声明的变量和函数称为类的成员。变量称为数据成员，函数称为成员函数。类的成员函数有时也称为方法，但本书不使用这个术语。
- 定义了一个类后，就可以声明类类型的变量(也称为类的实例)。每个实例都是类的一个对象。
- 定义类的实例有时称为实例化。
- 面向对象编程是一种编程模式，它的基础是把自己的数据类型定义为类。其中涉及到刚才讨论的数据封装、类的继承和多态性。

在详细论述面向对象编程时，这种编程模式似乎有些复杂。温习以前的基本知识常常有助于使概念更清晰，常常翻看上述列表总是有助于理解对象的真正含义。面向对象编程就是根据针对问题的对象来编程。在 C++中，类的主要作用就是使这个过程尽可能完备和灵活。下面就开始介绍类，首先从类的定义开始。

12.2　定义类

前面说过，类像结构一样，是用户定义的类型。使用 class 关键字定义类型的过程和使用 struct 关键字定义类型的过程完全相同，但结果是不同的。下面看看如何定义一个表示盒子的类，并使用它来研究所定义的类与前面定义的结构有什么不同。要创建类，只需用关键字 class

替代关键字 struct 即可，如下所示：

```
class Box {
  double length;
  double width;
  double height;

  //Function to calculate the volume of a box
  double volume() {
    return length*width*height;
  }
};
```

这里使用了第 11 章中 Box 结构的定义，只是把关键字 struct 替换为关键字 class。在使用这种类型对象的程序中，结构和类的主要区别会非常明显——不能访问这种对象的任何成员。在类的外部不能引用任何数据成员或调用函数 volume()，否则，代码就不会编译。

在创建类时，已定义的所有成员在默认情况下都是隐藏的，无法从类的外部访问。这称为类的私有成员，私有成员只能在该类的成员函数中访问。

为了在类外部的函数中访问类对象的成员，必须使用关键字 public 把该成员声明为类的公共成员。相比较而言，结构的成员在默认情况下都是公共的，这就是为什么总是可以访问它们的原因。

注意：

关键字 class 和 struct 用于创建遵循面向对象编程原则的数据类型。在默认情况下，类对象的成员是私有的，而结构对象的成员是公共的。

下面看看如何修改 Box 类的定义，把类的成员声明为公共成员：

```
class Box {
  public:
    double length;
    double width;
    double height;

    //Function to calculate the volume of a box
    double volume() {
      return length*width*height;
    }
};
```

关键字 public 是一个访问指定符。访问指定符确定类的成员是否能在程序的各个部分访问。公共的类成员可以直接在类的外部访问，因此这些成员是不隐藏的。为了把类成员指定为公共的，可以使用关键字 public 后跟一个冒号。在类的定义中，这个访问指定符后面的所有类成员都是公共的，直到使用另一个访问指定符为止。当然，类 Box 的对象封装了三个数据成员和一个函数成员。每个 Box 对象都包含这 4 个成员，因为在类中把它们声明为公共成员，所以可以直接从外部访问它们。

还有另外两个关键字也是访问指定符：private 和 protected。私有的类成员和受保护的类成

员都是隐藏的，即它们都不能在类的外部直接访问。如果在上述类的声明中加入下面的语句：

```
private:
```

则在这一行后面的所有成员就都是私有成员，而不是公共成员。本章后面将详细讨论关键字 private 的作用，第 15 章介绍何时使用关键字 protected。

一般情况下，在类的定义中可以多次使用任意访问指定符，这样就在类的定义中把数据成员和成员函数分成不同的组，每个组都有自己的访问指定符。如果根据访问指定符，对数据成员和成员函数分组，就更容易看出类定义的内部结构。

本章的第一个例子是从第 11 章的例子演变而来的，从这两个示例可以看出，结构和类的关系非常密切。这并不会给类建立一个好的设计，但可以逐步改进类，看看增加新特性会产生什么好的效果。

程序示例 12.1——使用类

下面是第 11 章程序示例 11.1 的修订版本，这里使用类来代替结构：

```
//Program 12.1 Using a Box class     File: ex12_01.cpp
#include <iostream>
using std::cout;
using std::endl;

//Class to represent a box
class Box {
  public:
    double length;
    double width;
    double height;

    //Function to calculate the volume of a box
    double volume() {
      return length*width*height;
    }
};

int main() {
  Box firstBox={80.0, 50.0, 40.0};

  // Calculate the volume of the box
  double firstBoxVolume=firstBox.volume();
  cout << endl;
  cout << "Size of first Box object is "
       << firstBox.length << " by "
       << firstBox.width << " by "
       << firstBox.height
       << endl;
  cout << "Volume of first Box object is "<<firstBoxVolume
       << endl;

  Box secondBox=firstBox;       //Create 2nd Box object same as firstBox
```

```
    //Increase the dimensions of second Box object by 10%
    secondBox.length *=1.1;
    secondBox.width *=1.1;
    secondBox.height *=1.1;

    cout << "Size of second Box object is "
         << secondBox.length << " by "
         << secondBox.width << " by "
         << secondBox.height
         << endl;
    cout << "Volume of second Box object is "<< secondBox.volume()
         << endl;

    cout << "Increasing the box dimensions by 10% has increased the volume by "
         << static_cast<long>
              (( secondBox .volume()- firstBoxVolume)*100.0/firstBoxVolume)
         << "%"
         << endl;
    return 0;
}
```

这个例子的结果与前一个版本的结果完全相同。

例子的说明

main()中的代码与前一个版本中的代码完全相同。这是因为把类成员声明为 public，类的工作方式就与结构一样了。前面讨论的与结构相关的内容都可以应用于类。对于结构和类，使用成员访问运算符和指针成员访问运算符的方式也是相同的。

但是，这段代码的一个方面不会反映到类的一般用法上。在例子中，用下面的初始化列表声明并初始化了类对象：

```
Box firstBox={80.0, 50.0, 40.0};
```

这个语句是合法的，至少在本例中是如此。但是，类对象的数据成员一般不以这种方式初始化，因为数据成员通常是隐藏的。不能使用初始化列表来初始化类中隐藏的数据成员。相反，类对象的创建和初始化是通过一种特殊的成员函数完成的，该成员函数可以初始化隐藏的数据成员。这个成员函数称为类的构造函数。

12.3 构造函数

类的构造函数是类中一种特殊的函数，它与普通的成员函数在许多方面都有所不同。构造函数在定义类的新实例时调用，它可以在创建新对象时初始化它，确保数据成员仅包含有效的值。如果在类的定义中包含一个构造函数，类的对象就不能用花括号中的数据值进行初始化。

类的构造函数常常与包含它的类同名。例如函数 Box()就是类 Box 的构造函数。另外，构造函数没有返回值，因此没有返回类型。为构造函数指定返回类型是错误的，甚至不能把它写为 void。类构造函数的主要作用是在创建类对象时，为它的所有数据元素赋予并验证初始值，并不需要或不允许有返回类型。

程序示例 12.2——给 Box 类添加构造函数

下面扩展上一个例子中的 Box 类，添加一个构造函数，再用更简单的 main()进行测试：

```
//Program 12.2 Using a class constructor  File:  ex12_02.cpp
#include <iostream>
using std::cout;
using std::endl;

//Class to represent a box
class Box {
public:
  double length;
  double width;
  double height;

  //Constructor
  Box(double lengthValue, double widthValue, double heightValue) {
    cout<<"Box constructor called"<<endl;
    length= lengthValue;
    width= widthValue;
    height= heightValue;
  }

  //Function to calculate the volume of a box
  double volume() {
      return length*width*height;
    }
};

int main() {
  Box firstBox(80.0, 50.0, 40.0);

  // Calculate the volume of the box
  double firstBoxVolume=firstBox.volume();
  cout << endl;
  cout << " Size of first Box object is "
       << firstBox.length<<" by "
       << firstBox.width<<" by "
       << firstBox.height
       << endl;
  cout << "Volume of first Box object is "<<firstBoxVolume
       << endl;

  return 0;
}
```

这个例子的输出结果如下所示：

```
Box constructor called
Size of first Box object is 80 by 50 by 40
Volume of first Box object is 160000
```

例子的说明

Box 类的构造函数用 3 个 double 类型的参数来定义，这 3 个参数分别对应于对象成员 length、width 和 height 的初始值。

```
Box(double lengthValue, double widthValue, double heightValue) {
  cout << "Box constructor called" << endl;
  length = lengthValue;
  width = widthValue;
  height = heightValue;
}
```

这个构造函数没有指定的返回类型，其名称与类名相同。构造函数中的第一个语句输出一个消息，说明已调用了这个构造函数。但在产品程序中是不能这么做的，因为尽管这有助于显示构造函数已调用，但这个技术常常用于测试程序。这里使用它是为了进行演示，并跟踪例子中发生的情况。构造函数体中的其他代码非常简单，只是把传送来的参数赋予对应的数据成员。如果需要，也可以对盒子的尺寸进行有效性检查，看看参数是否为非负值。在真正的应用程序中，应该进行检查，但这里的主要目的是看看构造函数是如何工作的，所以代码比较简单。

在 main()中，下面的语句声明了对象 firstBox：

```
Box firstBox(80.0, 50.0, 40.0);
```

数据成员 length、width 和 height 的初始值放在对象名后面的花括号中，它们会作为参数传送给构造函数。在调用构造函数时，会显示第一行输出信息，以证明类定义的确调用了刚才添加到类中的构造函数。

在类定义中声明了一个构造函数后，就不能再使用列表来初始化对象的数据成员了。在前面例子中用于定义 Box 对象的语句现在不能编译：

```
Box firstBox={80.0, 50.0, 40.0};          //Not legal! You must call a constructor.
```

注意：

如果类的定义至少包含一个构造函数，就不能使用初始化列表创建类的对象，而必须用构造函数来创建。

main()中的下两个语句输出盒子的尺寸和体积，这与前面的方法一样，从而证明了数据成员已设置为由构造函数的参数指定的值。

12.3.1 把构造函数的定义放在类的外部

在第 11 章处理结构时，成员函数的定义可以放在结构定义的外部。类和构造函数也是这样，在一个头文件中定义类 Box：

```
//Box.h
#ifndef BOX_H
#define BOX_H

class Box {
  public:
```

```
    double length;
    double width;
    double height;

    //Constructor
    Box(double lengthValue, double widthValue, double heightValue);

    //Function to calculate the volume of a box
    double volume();
};
#endif
```

现在把成员函数的定义放在一个.cpp 文件中。每个函数名，包括构造函数名，都必须通过作用域解析运算符用类名 Box 来限定：

```
//Box.cpp
#include <iostream>
#include "Box.h"

using std::cout;
using std::endl;

//Constructor definition
Box::Box(double lengthValue, double widthValue, double heightValue) {
  cout<<"Box constructor called"<<endl;
  length= lengthValue;
  width= widthValue;
  height= heightValue;
}

//Function to calculate the volume of a box
double Box::volume() {
  return length*width*height;
}
```

这里必须把包含 Box 类声明的头文件包括进来，否则编译器就不知道 Box 是一个类。把类的声明与其成员函数的声明分隔开来后，再用.h 文件和.cpp 文件的包含关系将它们组合在一起，这样代码会更容易管理。需要创建 Box 类型对象的源文件只需包含头文件 Box.h 即可。

使用这个类的程序员不需要访问成员函数的定义，只需要头文件中的类定义。只要类定义保持不变，就可以自由修改成员函数的实现代码，而不会影响使用该类的程序操作。

在类定义的外部定义成员函数与把定义放在类的内部并不完全相同。在类定义内部的函数定义隐式地声明为内联函数(这并不是说它们一定都实现为内联函数——这仍由编译器根据函数的特性来决定，如第 8 章所述)。如果定义放在类外部的成员函数显式声明为内联函数，它们就只能是内联函数。

本章后面的例子假定，Box 类被分解到.h 文件、.cpp 文件以及另一个包含 main()函数的文件中，其代码如下所示：

```
//Program 12.2a Using a class constructor    File: ex12_02a.cpp
#include <iostream>
```

```
#include "Box.h"

using std::cout;
using std::endl;

int main() {
  Box firstBox(80.0, 50.0, 40.0);

  // Calculate the volume of the box
  double firstBoxVolume=firstBox.volume();
  cout << endl;
  cout << " Size of first Box object is "
       << firstBox.length<<" by "
       << firstBox.width<<" by "
       << firstBox.height
       << endl;
  cout << "Volume of first Box object is "<<firstBoxVolume
       << endl;

  return 0;
}
```

这个 main()版本与前面的例子完全相同。代码中惟一的区别是 box.h 头文件的#include 指令，box.h 头文件包含了 Box 类的定义。

12.3.2 默认的构造函数

上面示例中类的构造函数看起来非常简单，但在其表面下潜藏了一些奥妙的观念。为类定义了构造函数后，就以一种不太明显的方式修改了类。下面就研究这些内容。

声明的每个类至少有一个构造函数，因为类的对象总是用构造函数创建的。如果没有为类定义构造函数(如程序示例 12.1 所示)，编译器就会提供一个默认的构造函数，用于创建类的对象。

默认的构造函数没有参数。在声明类时，只要没有定义类的构造函数，编译器就会自动提供一个默认的构造函数。一旦添加了自己的构造函数，编译器就假定该构造函数是默认的构造函数，因此不再提供构造函数。下面的例子演示了默认的构造函数如何影响代码。

下面在程序中定义第二个 Box 对象，但其初始值与 firstBox 不同。修改程序示例 12.2，如下所示：

```
int main() {
  Box firstBox(80.0, 50.0, 40.0);

  // Calculate the volume of the box
  double firstBoxVolume=firstBox.volume();
  cout << endl;
  cout << " Size of first Box object is "
       << firstBox.length << " by "
       << firstBox.widdth << " by "
       << firstBox.height
```

```
        << endl;
cout << "Volume of first Box object is "<<firstBoxVolume
        << endl;

Box smallBox;          //Will not compile! Constructor already specified
smallBox.length=10.0;
smallBox.width=5.0;
smallBox.height=4.0;

//Calculate the volume of the small box
cout << "Size of small Box object is"
        << smallBox.length << " by "
        << smallBox.width << " by "
        << smallBox.height
        << endl;
cout << "Volume of small Box object is " << smallBox.volume()
        << endl;

return 0;
}
```

新的代码试图创建一个新对象 smallBox，但其声明并未给构造函数提供初始值，而是用 3 个赋值语句来显式设置数据成员。可是，这段代码不会被编译。编译器会生成如下所示的错误消息：

```
Box smallBox;          //Will not compile! Constructor already specified
```

编译器会一直寻找默认的构造函数(即没有参数的构造函数)。但是，这个程序使用了类 Box，它包含用户定义的构造函数：

```
Box(double lengthValue, double widthValue, double heightValue) {
  Cout << "Box constructor called" << endl;
  length= lengthValue;
  width= widthValue;
  height= heightValue;
}
```

由于我们给类声明了一个构造函数，所以编译器不会生成默认的构造函数，这就是出现错误的原因。

编译器生成的默认构造函数什么也不做——尤其是，默认的构造函数不会初始化所创建的对象中非类类型的数据成员。类类型的数据成员只能调用其默认构造函数才能初始化。这非常不理想：我们的目标是控制包含在变量中的值，如果计划使用默认的构造函数来创建对象，就要以其他方式提供变量值。为了使 main()的新版本能正常工作，可以在类定义中添加自己的默认构造函数。

程序示例 12.3——提供默认的构造函数

下面把我们自己的默认构造函数添加到上一个例子中。之后，再添加调用默认构造函数的代码，并在以后初始化数据成员。下面是 Box.h 中新的类定义：

```
class Box {
```

```
public:
  double length;
  double width;
  double height;

  //Constructors
  Box();            //Default constructor
  Box(double lengthValue, double widthValue, double heightValue);

  //Function to calculate the volume of a box
    double volume();
};
```

还必须在 **Box.cpp** 中添加默认构造函数的定义：

```
// Default constructor definition
Box::Box() {
  cout<<" Default constructor called"<<endl;
  length = width = height =1.0;            //Default dimensions
}
```

在 **main()**的下述版本中使用这个构造函数：

```
//Program 12.3 Defining and using a default class constructor
// File: ex12-03.cpp
#include <iostream>
#include "Box.h"

using std::cout;
using std::endl;

int main() {
Box firstBox(80.0, 50.0, 40.0);

  // Calculate the volume of the box
  double firstBoxVolume=firstBox.volume();
  cout << endl;
  cout << " Size of first Box object is "
       << firstBox.Length << " by "
       << firstBox.width << " by "
       << firstBox.height
       << endl;
cout << "Volume of first Box object is "<<firstBoxVolume
       << endl;

Box smallBox;
smallBox.length=10.0;
smallBox.width=5.0;
smallBox.height=4.0;

//Calculate the volume of the small box
cout <<" Size of small Box object is"
       << smallBox.length << " by "
```

```
        << smallBox.width << " by "
        << smallBox.height
        << endl;
  cout << "Volume of small Box object is " << smallBox.volume()
        << endl;

  return 0;
}
```

运行结果如下：

```
Box constructor called
Size of first Box object is 80 by 50 by 40
Volume of first Box object is 160000
Default constructor called
Size of small Box object is 10 by 5 by 4
Volume of small Box object is 200
```

例子的说明

在程序的这个版本中，提供了自己的构造函数，因此编译器不会提供默认的构造函数。重要的是，我们有了自己的默认构造函数。编译器这次没有生成错误消息，一切正常。程序结果表示，在声明 smallBox 时，调用了默认的构造函数。

默认的构造函数会为所创建的新对象初始化数据成员：

```
length = width = height =1.0;              //Default dimensions
```

现在可以使用默认的构造函数定义并初始化 Box 对象 smallBox：

```
Box smallBox;
```

默认构造函数的一个主要特性是，在调用它时不需要指定参数列表，甚至不需要括号。上面的语句仅指定了类类型 Box 和对象名 smallBox，因此会调用默认构造函数。

定义一个默认的构造函数，来设置数据成员的值，可以确保 Box 对象的数据成员不包含垃圾值。安全地定义和初始化 smallBox 后，就可以按需要调整其尺寸了：

```
smallBox.length = 10.0;
smallBox.width = 5.0;
smallBox.height = 4.0;
```

这个例子被忽略的一点是重载了构造函数，就像在第 9 章重载函数一样。Box 类有两个构造函数，它们仅参数列表不同。一个构造函数有 3 个 double 类型的参数，另一个构造函数没有参数。

12.3.3 默认的初始化值

在讨论 C++中的“原始”函数时，介绍了如何在函数原型中为参数指定默认值。也可以为类的成员函数指定参数的默认值，包括构造函数在内。如果把成员函数的定义放在类定义中，就可以把参数的默认值放在函数头中。如果在类定义中仅包含了函数的声明，则默认的参数值就应放在声明中，而不应放在函数定义中。

在前面例子的默认构造函数中，Box 对象的默认尺寸是一个单位的盒子，即所有的边长都是 1。于是，修改上一个例子中的类定义，如下所示：

```
class Box {
  public:
    double length;
    double width;
    double height;

    //Constructors
    Box();              //Default Constructor
    Box(double lengthValue=1.0, double widthValue=1.0,
                                                  double heightValue=1.0);

    //Function to calculate the volume of a box
    double volume();
};
```

如果对上一个例子进行这些修改，会发生什么情况？编译器会生成另一个错误消息！该错误消息表示定义了多个默认构造函数。main()中的下述代码也会产生一个消息，说明对重载函数的调用不明确：

```
Box smallBox;
```

产生混淆的原因是，这个语句对两个构造函数的调用都是合法的。像这样调用包含默认参数值的构造函数，不能与调用没有参数的默认构造函数区分开。没有指定任何参数，就意味着编译器不能区分这两个调用。换言之，有默认参数值的构造函数也可以用作默认构造函数。

显然，其解决方法是去掉不接受参数的构造函数。这样，代码就能编译，执行也正常了。但是，这并不是实现默认构造函数的最佳方式。在许多情况下，都不希望以这种方式赋予默认值，而应编写一个独立的默认构造函数。甚至有时即使已经定义了另一个构造函数，也不希望有默认的构造函数。确保类的所有对象都在其声明中显式指定了初始值。

12.3.4 在构造函数中使用初始化列表

前面是在类的构造函数体中用显式的赋值语句来初始化对象的成员。还有另一种可用的技术，即使用初始化列表。下面用类 Box 的构造函数来说明。

```
//Constructor definition using an initializer list
Box::Box(double lvalue, double wvalue, double hvalue) :
                              length(lvalue), width(wvalue), height(hvalue) {
  cout<<"Box constructor called"<<endl;
}
```

数据成员的值不是在构造函数体的赋值语句中设置的。因为在声明中，它们用函数表示法指定为初始化值，并显示在初始化列表中，作为函数头的一部分。例如，成员 length 通过 lvalue 的值进行初始化。注意构造函数的初始化列表与参数列表用冒号分隔开，每个初始化值用逗号分隔开。

实际上，这不仅仅是表示法不同，在初始化的方式上也有根本的区别。在使用构造函数体中的赋值语句初始化数据成员时，首先要创建该数据成员(如果这是类的一个实例，就调用构造函数)，再执行赋值语句。而在使用初始化列表时，数据成员在创建时，就用初始值对它进行初始化。这要比在构造函数体中使用赋值语句的效率高得多，特别是在数据成员是一个类实例时，就更是如此。如果用这个版本的构造函数替换上一个例子中的构造函数，它也会正常工作。

在构造函数中初始化参数的技术非常重要还有另一个原因。它是为某些类型的数据成员设置值的惟一方式。

12.3.5 使用 explicit 关键字

类的构造函数只有一个参数是非常危险的，因为编译器可以使用这种构造函数把参数的类型隐式转换为类类型。在某些情况下，这会产生不良的后果。下面介绍一种特殊的情形。

假定定义一个类，该类定义了立方体的盒子，即所有的边长都相等：

```
class Cube {
  public:
    double side;

    Cube(double side);                    //Constructor
    double volume();                      //Calculate volume of a cube
    bool compareVolume(Cube aCube);       //Compare volume of a cube with another
};
```

构造函数定义为：

```
Cube::Cube(double length): side(length){}
```

计算立方体体积的函数定义为：

```
double Cube::volume(){return side * side * side;}
```

最后，compareVolume()成员定义为：

```
bool Cube::compareVolume(Cube aCube){return volume()>aCube.volume();}
```

构造函数只需要一个 double 类型的参数。显然，编译器可以使用构造函数把 double 值转换为 Cube 对象，但在本例中，这会出现什么问题？下面介绍一下 Cube 类。

该类还定义了 volume()函数和另一个函数。该函数比较当前对象和另一个作为参数传送的 Cube，如果当前对象的体积比较大，就返回 true。可以用下面的方式来使用该函数：

```
Cube box1(5.0);
Cube box2(3.0);

if(box1.compareVolume(box2))
  std::cout<< std::endl<<"box1 is larger";
else
  std::cout<< std::endl<<"box1 is not larger";
```

这是非常简单的，但如果使用这个类的人编写了下面的代码：

```
if(box1.compareVolume(50.0))
  std::cout<< std::endl<<" Volume of box1 is greater than 50";
else
  std::cout<< std::endl<<" Volume of box1 is not greater than 50";
```

编写这段代码的人误解了 compareVolume()函数，认为它把当前对象的体积与 50.0 相比较。编译器知道 compareVolume()函数的参数应是一个 Cube 对象，但它也会编译这段代码，因为这个构造函数可以把参数 50.0 转换为一个 Cube 对象。编译器生成的代码如下所示：

```
if(box1.compareVolume(Cube(50.0))
  std::cout<< std::endl<<" Volume of box1 is greater than 50";
else
  std::cout<< std::endl<<" Volume of box1 is not greater than 50";
```

函数没有把 box1 对象的体积与 50.0 相比较，而是与 Cube(50.0)的体积即 125000.0 进行比较。结果与期望的完全不同。但把构造函数声明为 explicit，就可以避免这种情况：

```
class Cube {
  public:
    double side;

    explicit Cube(double side);
    double volume();
    bool compareVolume(Cube aCube);
};
```

编译器不会把声明为 explicit 的构造函数用于隐式类型转换，它只能在程序代码中显式创建对象。因为根据定义，隐式的类型转换是把给定的类型转换为另一种类型，对只有一个参数的构造函数，只需使用 explicit 关键字，就可避免进行隐式类型转换。

12.4 类的私有成员

构造函数的一个主要目的是确保对象的所有数据成员都设置为合适的值。例如，在构造函数中添加几个检查操作，确保 Box 对象的所有尺寸都是正数：

```
Box::Box(double lvalue, double wvalue, double hvalue) :
                                  length(lvalue), width(wvalue), height(hvalue) {
  std::cout<<"Box constructor called"<< std::endl;

  //Ensure positive dimensions
  if(length <= 0.0)
    length = 1.0;
  if(width <= 0.0)
    width = 1.0;
  if(height <= 0.0)
    height = 1.0;
}
```

无论构造函数的参数值是多少，Box 对象都有合法的尺寸。当然，还可以输出一个消息，说明进行了这类调整。

现在的问题是为了保护数据成员的完整性，应阻止外部的语句修改这些尺寸，如下所示：

```
Box theBox(10.0, 10.0, 5.0);
theBox.length=-20.0;              //Set illegal box dimension
```

这是因为数据成员是使用关键字 public 声明的。隐藏数据成员就可以避免修改它们，为此，只需把类的数据成员声明为 private 即可。在大多数情况下，类的私有成员只能由类的成员函数访问。如果某个函数不是给定类的成员，就不能直接访问该类的私有成员。如图 12-5 所示。

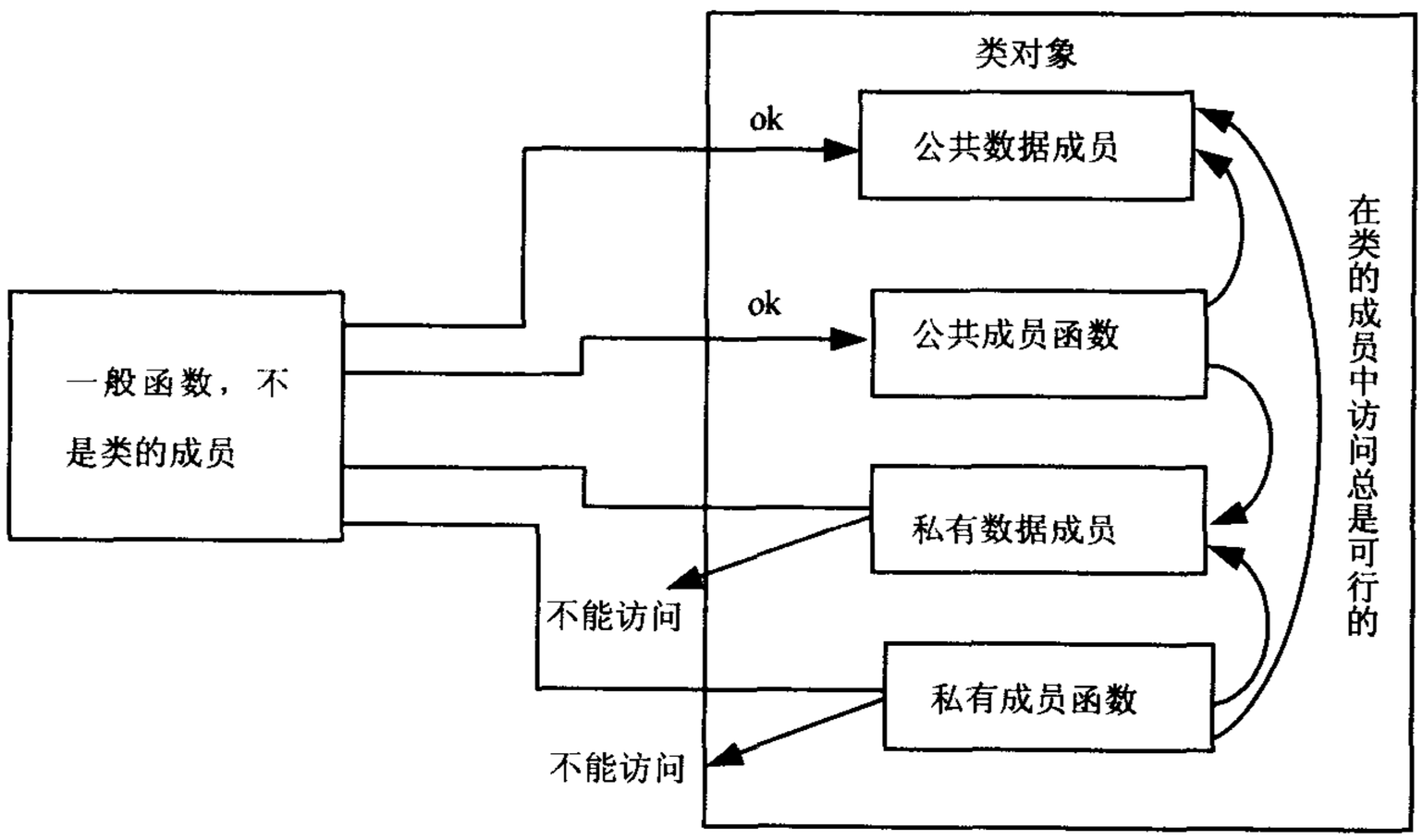


图 12-5　隐藏类的成员

一般情况下，在面向对象编程中，最好尽可能地把类的数据成员设置为私有成员。毕竟，面向对象编程的要点是根据对象进行编程，对于使用这些对象的程序，不应考虑对象的内部机制。

类的公共成员一般是函数，有时称为类的接口。类接口提供了处理和操作类对象的方式，确定可以对对象做什么，以及对象可以完成什么任务。把类的内部机制设置为私有，就可以以自己希望的方式修改它们，而无需修改通过公共接口使用该类的代码。因此，公共接口与类的实现代码要分隔开来。

程序示例 12.4——私有数据成员

下面再次改写 Box 类，使其数据成员变成私有成员，在另一个例子中看看如何使用它。下面修改头文件 Box.h，如下所示：

```
//Box.h Definition of the Box class
#ifndef BOX_H
#define BOX_H

class Box {
  public:
    //Constructor
    Box(double lengthValue=1.0, double widthValue=1.0, double heightValue=1.0);
```

```
    //Function to calculate the volume of a box
    double volume();

  private:
    double length;
    double width;
    double height;
};

#endif
```

在 Box.cpp 中添加检查尺寸的代码：

```
//Box.cpp Box class member function definitions
#include <iostream>
#include "Box.h"

using std::cout;
using std::endl;

//Constructor
Box::Box(double lvalue, double wvalue, double hvalue) :
                              length(lvalue), width(wvalue), height(hvalue) {
  cout<<"Box constructor called"<<endl;

  //Ensure positive dimensions
  if(length <= 0.0)
    length = 1.0;
  if(width <= 0.0)
    width = 1.0;
  if(height <= 0.0)
    height = 1.0;
}

//Function to calculate the volume of a box
double Box::volume() {
  return length*width*height;
}
```

成员函数的定义在类的外部，这不会影响类成员的可访问性。无论把成员函数的定义放在什么地方，都可以在类的函数成员的函数体中访问类的所有成员。

为了试验 Box 类的新版本，编写一个新的 main()函数：

```
//Program 12.4 Using a class with private data members
// File: ex12_04.cpp
#include <iostream>
#include "Box.h"

using std::cout;
using std::endl;
```

```
int main() {
  cout << endl;

  Box firstBox(2.2, 1.1, 0.5);
  Box secondBox;
  Box* pthirdBox = new Box(15.0, 20.0, 8.0);

  cout << "Volume of first box = "
       << firstBox.volume()
       << endl;

// secondBox.length=4.0;          //Uncomment this line to get an error

  cout << "Volume of second box = "
       << secondBox.volume()
       << endl;

  cout << "Volume of third box = "
       << pthirdBox->volume()
       << endl;

  delete pthirdBox;
  return 0;
}
```

这个程序的输出结果如下所示：

```
Box constructor called
Box constructor called
Box constructor called
Volume of first box = 1.21
Volume of second box = 1
Volume of third box = 2400
```

例子的说明

在成员函数(包括构造函数和 volume()函数)的声明中，使用关键字 public 开始声明程序的公共部分。这里故意把类的公共成员声明放在私有成员声明的前面——公共类成员通常对浏览代码的人更有吸引力，因为它们可以从外部访问。

类 Box 的定义现在使用关键字 private 后跟一个冒号，把数据成员声明为私有成员。在这行代码之后到下一个访问指定符之前的所有成员声明都是私有成员，不能从类的外部访问。这就是数据隐藏。如果取消 main()中下述语句的注释：

```
// secondBox.length=4.0;          //Uncomment this line to get an error
```

代码就不会编译，因为在类的外部使用类的私有数据成员是非法的。

注意：

获取 Box 对象中私有数据成员的值的惟一方式是使用构造函数或成员函数。应确保可以通过成员函数设置或修改类的私有数据成员。

也可以把函数放在类的私有部分，这样，这些函数就只能由其他成员函数调用了。如果把函数 volume()放在私有部分，并在 main()函数中使用它，则编译器就会生成错误消息。如果把构造函数放在私有部分，就不能声明该类的对象。

提示：

把某些类成员函数设置为私有成员有许多原因。例如，需要一个帮助函数来实现由其他成员函数在内部使用的功能，但它与类对象的使用没有关系。

回到 main()中，首先用下面的语句声明两个 Box 对象：

```
Box firstBox(2.2, 1.1, 0.5);
Box secondBox;
```

这两个对象调用了相同的构造函数(这个类只有一个构造函数)，但 secondBox 的声明使用了默认的参数值，所以 secondBox 是一个尺寸为 1.0 的立方体。

下一个语句声明了一个 Box 对象的指针 pthirdBox，并使用运算符 new 在自由存储区中创建了一个 Box 对象：

```
Box* pthirdBox = new Box(15.0, 20.0, 8.0);
```

运算符 new 调用了同一个 Box 构造函数，这可以从输出中看出，new 返回的地址存储在 pthirdBox 中。

现在输出每个 Box 对象的体积。对于动态创建的对象，使用指针成员访问运算符来调用函数 volume()：

```
cout << "Volume of third box = "
     << pthirdBox->volume()
     << endl;
```

这说明该类的数据成员用访问指定符 private 定义后，仍能正常工作。主要的区别是它们现在完全不能进行未经授权的访问和修改。由于所有的数据成员都隐藏了，因此设置或改变其值的惟一方法就是利用类的公共成员函数。在 Box 类中，则只有一个函数是公共成员，即构造函数。

12.4.1 访问私有类成员

把类的数据成员声明为私有成员是比较极端的做法。这样可以完全禁止对它们进行未经授权的修改，但也存在一个严重的限制：如果不知道某个 Box 对象的尺寸，就永远都不知道。一定要这样保密吗？

要解决这个问题，并不需要把数据成员用 public 关键字来声明，只需添加一个类的成员函数，返回数据成员的值即可。为了访问 Box 对象的尺寸，只需在类定义中添加 3 个函数：

```
class Box {
  public:
    //Constructor
    Box(double lengthValue=1.0, double widthValue=1.0,
                                          double heightValue=1.0);
```

```
  //Function to calculate the volume of a box
  double volume();

  //Function to provide the values of data members
  double getLength() {return length ;}
  double getwidth () {return width;}
  double getHeight () {return height;}

private:
   double length;
   double width;
   double height;
};
```

在 Box 类中添加了几个函数，返回数据成员的值。也就是说，数据成员的值可以访问，但不能修改它们，以维护类的完整性。这种函数通常把定义放在类定义中，因为它们非常短，在默认情况下是内联函数。因此，获取数据成员值所涉及的系统开销就非常小。提取数据成员值的函数通常称为访问器成员函数。

在前面的例子中就可以使用这些访问器函数，以输出动态创建的 Box 对象的尺寸：

```
cout << "Box size is "
     << pthirdBox-> getLength() << " by "
     << pthirdBox-> getWidth () << " by "
     << pthirdBox-> getHeight ()
     << endl;
```

任何类都可以采用这种方法。只要为每个希望从外界访问的数据成员编写一个这样的函数，就可以访问它们的值，而不会损害类的安全性。当然，如果把这些函数的定义放在类定义的外部，就应把它们声明为 inline。例如，如果在类定义中声明了 getLength()，就需要在 Box.cpp 中把它定义为：

```
inline double Box::getLength(){return length;}
```

在某些情况下，希望在类的外部修改数据成员。如果为此提供了一个成员函数，而不是允许直接访问数据成员，就要对值进行完整性检查。例如，添加一个函数，以修改 Box 对象的高度：

```
class Box {
  public:
   //Constructor
   Box(double lengthValue=1.0, double widthValue=1.0,
                                          double heightValue=1.0);

   //Function to calculate the volume of a box
   double volume();

   //Inline functions to provide the values of data members
   double getLength() {return length ;}
   double getwidth () {return width;}
   double getHeight () {return height;}
```

```
    //Functions to set data member values
    void setHeight(double hvalue){if(hvalue >0) height= hvalue;}

  private:
    double length;
    double width;
    double height;
};
```

if 语句确保 height 仅接受新的正数值。如果为 height 成员提供的新值是 0 或负数，就忽略此操作。允许修改数据成员的成员函数常常称为变异(mutator)成员函数。

12.4.2 默认的副本构造函数

假定下面的语句声明并初始化 Box 对象 firsBox：

```
Box firsBox(15.0, 20.0, 10.0);
```

现在要创建另一个 Box 对象，它与第一个对象完全相同。换言之，就是用 firsBox 初始化第二个 Box 对象。下面看看把 Box 对象用作构造函数的参数时会发生什么情况。

程序示例 12.5——创建对象的一个副本

在此把上一个例子中的 Box 类的定义用于本示例。下面是创建 Box 对象副本的代码：

```
//Program 12.5 Creating a copy of an object     File: ex12_05.cpp
#include <iostream>
#include "Box.h"

using std::cout;
using std::endl;

int main() {
  cout<<endl;

  Box firstBox(2.2, 1.1, 0.5);
  Box secondBox(firstBox);

  cout << "Volume of first box = "
       << firstBox.volume()
       << endl;

  cout << "Volume of second box = "
       << secondBox.volume()
       << endl;

  return 0;
}
```

编译并运行这个例子，输出结果如下所示：

```
Box constructor called
Volume of first box = 1.21
Volume of second box = 1.21
```

例子的说明

显然，程序像我们希望的那样执行，两个盒子有相同的体积。但是，从输出中可以看出，构造函数仅调用了一次(用于创建 firstBox)。那么，secondBox 对象是如何创建的？

其机制与没有定义构造函数时的情形类似，编译器会提供一个默认的构造函数来创建对象，这里则是编译器生成了所谓副本构造函数的默认版本。副本构造函数完成了我们希望完成的工作——创建类的一个对象，用该类已有的一个对象对它进行初始化。副本构造函数的默认版本会复制已有对象中的每个成员，以创建新的对象。副本构造函数可以用这种方式复制任何数据类型。

注意：

创建对象时，不必显式调用副本构造函数。只要把对象作为一个参数按值传送给函数，编译器就会调用副本构造函数，制作该对象的副本。

默认的副本构造函数适用于简单的类，例如这里的 Box 类，但在许多情况下，例如类把指针作为其成员，副本构造函数就会产生不良的后果。对于这种类，副本构造函数会在程序中产生严重的错误，此时，就需要为类定义自己的副本构造函数。

副本构造函数可以从类的已有对象中创建该类的新对象，定义它需要一种特殊的方法，详见第 13 章。

12.5 友元

在正常情况下，要把类的数据成员声明为私有成员，以隐藏它们。类还可以有私有的成员函数。有时还可以把某些选定的函数看作类的“荣誉成员”，允许它们访问类对象中非公共的成员，就好像它们是类的成员一样。这种函数称为类的友元。友元可以访问类对象的任意成员，无论这些成员的访问指定符是什么。

有两种情形需要考虑。可以把一个函数指定为类的友元，或把整个类指定为另一个类的友元。在后者中，友元类的所有成员函数与原类的一般成员有相同的访问权限。下面首先看看作为友元的单个函数。

12.5.1 类的友元函数

如果某个函数不是类的一个成员，但可以访问类的所有成员，这个函数就称为该类的友元函数。为了把函数看作是类的友元函数，必须在类定义中用关键字 friend 来声明它。

注意：

类的友元函数是一个全局函数，也可以是另一个类的成员。但是，函数不能是包含它的类的友元函数，因此，访问指定符不能应用于类的友元。

实际上，对友元函数的需求是比较有限的。当函数需要访问两个不同对象的内部时，才需要把该函数声明为这两个类的友元。这里在比较简单的情形中使用它们，在这个情形中实际上并不需要使用友元，但演示了友元的操作。

假定要在 Box 类中实现一个友元函数，计算 Box 对象的表面积。

程序示例 12.6——用友元计算表面积

为了把函数声明为友元，必须在类定义中把它声明为友元。下面修改程序示例 12.4，首先修改 Box.h 中的定义：

```
class Box {
  public:
    //Constructor
    Box(double lengthValue=1.0, double widthValue=1.0,
                                          double heightValue=1.0);

    //Function to calculate the volume of a box
    double volume();

    // Friend function
    friend double boxSurface(const Box& theBox);

  private:
    double length;
    double width;
    double height;
};
```

这个例子也使用了程序示例 12.4 中的文件 Box.cpp。下面是主程序代码：

```
//Program 12.6 Using a friend function of a class   File: ex12_06.cpp
#include <iostream>
#include "Box.h"

using std::cout;
using std::endl;

int main() {
  cout<<endl;

  Box firstBox(2.2, 1.1, 0.5);
  Box secondBox;
  Box* pthirdBox = new Box(15.0, 20.0, 8.0);

  cout << "Volume of first box = "
       << firstBox.volume()
       << endl;

  cout << "Surface area of first box = "
       << boxSurface(firstBox)
       << endl;
```

```
    cout << "Volume of second box = "
         << secondBox.volume()
         << endl;

    cout << "Surface area of second box = "
         << boxSurface(secondBox)
         << endl;

    cout << "Volume of third box = "
         << pthirdBox->volume()
         << endl;

    cout << "Surface area of third box = "
         << boxSurface(*pthirdBox)
         << endl;

    delete pthirdBox;
    return 0;
}

// friend function to calculate the surface area of a Box object
double boxSurface(const Box& theBox) {
    return 2.0*(theBox.length * theBox.width +
                theBox.length * theBox.height +
                theBox.height * theBox.width);
}
```

程序的运行结果如下所示：

```
Box constructor called
Box constructor called
Box constructor called
Volume of first box = 1.21
Surface area of first box = 8.14
Volume of second box = 1
Surface area of second box = 6
Volume of third box = 2400
Surface area of third box =1160
```

例子的说明

在类定义中，使用关键字 friend 编写函数原型，把函数 boxSurface()声明为 Box 类的一个友元，如下所示：

```
friend double boxSurface(const Box& theBox);
```

我们没有修改把 Box 对象作为参数传送给函数的任何方面，所以可以使用 const 引用参数指定符。把 friend 声明放在类的定义中也比较好。这里选择把该声明放在类中公共成员的后面、私有成员的前面。这是因为函数是类的接口，可以完全访问类的成员。friend 函数不是类的一个成员，所以不能对它使用访问指定符。

函数 boxSurface()本身是一个全局函数，其定义放在 main()的后面。还可以把它放在 Box.cpp 中，因为它与 Box 类相关，但把它放在主文件中，将有助于表示它是一个全局函数。

注意必须在函数 boxSurface()的定义中，把 Box 对象作为一个参数传送给该函数，指定访问对象的数据成员。因为友元函数不是类成员，所以数据成员不能仅通过其名称来引用。它们必须用对象名来限定，其方法与在一般函数中访问类的公共成员一样。友元函数与一般函数一样，但友元函数可以不受限制地访问类中的所有成员。

main()函数改为调用友元函数，输出已创建的 3 个对象的表面积。从输出结果上看，该函数工作正常。

这个例子说明了如何编写友元函数，但该例子并不是很实用。我们可以使用访问器成员函数来返回数据成员的值，因此 boxSurface()根本不需要声明为友元函数。最好的方法是把函数 boxSurface()声明为类的一个公共成员函数，这样计算盒子表面积的功能就成为类接口的一部分。

友元函数是类接口的一部分，但较好的编程方式是只要可能，就应完全根据需要的成员函数来定义类的接口。前面说过，惟一需要友元函数的情形是：需要访问两个不同类的非公共成员，甚至在这种情况下，也有一种避免使用友元函数的方式。

12.5.2 友元类

还可以把整个类声明为另一个类的友元。友元类的所有成员函数都可以不受限制地访问原类的成员。

例如，假定定义一个类 Carton，为了让类 Carton 的成员函数访问 Box 类的成员，只需在 Box 类定义中包含一个把 Carton 声明为 Box 的友元类的语句即可：

```
class Box {
  //Public members of the class…

  friend class Carton;

  //Private members of the class…
};
```

友元关系并不是一个互惠的安排。类 Carton 中的函数现在可以访问 Box 类的所有成员，但 Box 类的函数不能访问类 Carton 中的私有成员。类之间的友元关系是不能传递的，即类 A 是类 B 的友元，类 B 又是类 C 的友元，但类 A 不是类 C 的友元。

友元类的一个常见用法是一个类的功能与另一个类的功能高度缠绕在一起。链表(详见第 11 章)基本上涉及到两个类类型：存储一个对象列表(通常称为节点)的 List 类，和定义节点的 Node 类。List 类需要在每个 Node 对象中设置一个指针，使该指针指向下一个节点，从而把 Node 对象组合在一起。把 List 类声明为定义节点类的友元，可以使 List 类的成员直接访问 Node 类的成员。

12.6 this 指针

在 Box 类中，根据类定义中的类成员名编写了 volume()函数。但已创建的每个 Box 类型的对象都包含这些成员，该函数必须有一种机制来引用调用它的那个对象的成员。换言之，在 volume()函数中的代码引用类的 length 成员时，length 必须有一种方式引用调用函数的对象成员，而不是引用其他对象。

在执行任何类成员函数时，该函数都会自动包含一个隐藏的指针，称为 this，该指针包含了调用该函数的对象的地址。例如，在下面的语句中：

```
cout << firstBox.volume();
```

函数 volume()中的指针 this 就包含 firstBox 的地址。在为另一个 Box 对象调用该函数时，this 指针就设置为包含该对象的地址。

也就是说，在执行 volume()函数的过程中访问数据成员 length，该数据成员就表示为 this->length，这是完全限定的对象成员的引用。编译器会把必要的指针名 this 添加到函数的成员名中。换言之，编译器把函数实现为：

```
double Box::volume() {
  return this->length* this->width* this->height;
}
```

也可以把函数改写为显式使用指针 this，但这是不必要的。然而，在一些情况下，需要使用该指针。例如，在成员函数有多个相同类类型的参数，或者在需要返回当前对象的地址情况下都要使用 this 指针。下面在一个例子中显式使用 this 指针，看看它的工作原理。

程序示例 12.7——显式使用 this

比较 Box 对象的体积是 Box 类的一个很有用的功能。下面在 Box 类中添加一个公共函数，以实现该功能。Box.h 中的类定义如下所示：

```
class Box {
  public:
    //Constructor
    Box(double lengthValue=1.0, double widthValue=1.0,
                                          double heightValue=1.0);

    //Function to calculate the volume of a box
    double volume();

    // Function to compare two Box objects
    int compareVolume( Box& otherBox);

  private:
    double length;
    double width;
    double height;
};
```

需要把 compareVolume()的定义添加到 Box.cpp 文件中，在该文件中，已添加了其他一些函数的定义：

```
// Function to compare two Box objects
//If the current Box is greater than the argument, 1 is returned
//If they are equal, 0 is returned
//If the current Box is less than the argument, -1 is returned
int Box::compareVolume( Box& otherBox) {
   double vol1= this->volume();          //Get current Box volume
   double vol2= otherBox. volume();       //Get argument volume
   return vol1 > vol2 ? 1: (vol1 < vol2 ? -1: 0);
}
```

现在创建两个 Box 对象，并比较它们的体积：

```
//Program 12.7 Using the this pointer    File: ex12_07.cpp
#include <iostream>
#include "Box.h"

using std::cout;
using std::endl;

int main() {
  cout<<endl;

  Box firstBox(17.0, 11.0, 5.0);
  Box secondBox(9.0, 18.0, 4.0);

  cout << "the first box is "
       << (firstBox.compareVolume(secondBox) >= 0 ? "" : "not ")
       << "greater than the second box."
       << endl;

  cout << "Volume of first box = "
       << firstBox.volume()
       << endl;

  cout << "Volume of second box = "
       << secondBox.volume()
       << endl;

  return 0;
}
```

这个程序的运行结果如下所示：

```
Box constructor called
Box constructor called
The first box is greater than the second box.
Volume of first box = 935
Volume of second box = 648
```

例子的说明

成员函数 compareVolume()的实现代码使用了两个 Box 对象：一个是调用该函数的对象，另一个对象作为该函数的参数。为了给调用 compareVolume()函数的对象调用 volume()函数，使用了 this 指针：

```
double vol1=this->volume();        //Get current Box volume
```

提示：

在使用对象选择成员时，可以使用直接成员访问运算符.；在使用对象的指针时，可以使用指针成员访问运算符->。this 是一个指针，所以应使用指针成员访问运算符。

在这个例子中使用 this 指针，只是为了说明它的存在。这里并不一定要使用它，比如可以把该语句改写为：

```
double vol1= volume();        //Get current Box volume
```

在 compareVolume()函数体的内部，或在任何成员函数中，都自动用 this 指针来访问类成员名，所以总是会获得属于当前对象的成员。只有在某些环境下才需要显式使用 this 指针。其中的一个例子是为消除不明确性(例如函数参数与数据成员同名)显式使用 this 指针。如果要从成员函数中返回当前对象的地址，也应使用 this 指针。

为了获取作为参数传送的对象的体积，只需使用参数名来调用函数：

```
double vol2= otherBox.volume();        //Get argument volume
```

compareVolume()函数中的最后一个语句返回对应的整数值：

```
return vol1 > vol2 ? 1: (vol1 < vol2 ? -1: 0);
```

如果 vol1 大于 vol2，条件运算符就会返回 1。否则，就计算括号中的条件运算符。如果 vol1 小于 vol2，就返回 - 1，否则返回 0。

main()中的 compareVolume()函数检查 firstBox 对象和 secondBox 对象的体积之间的关系，如下面的语句所示：

```
cout << "The first box is "
     << (firstBox.compareVolume(secondBox) >= 0 ? "" : "not ")
     << "greater than the second box. "
     << endl;
```

compareVolume()函数返回的值和条件运算符一起使用，来确定在输出中是否包含字符串 "not"。然后输出语句验证 firstBox 对象是否大于 secondBox 对象。

实际上，根本不必把 compareVolume()函数编写为一个类成员。可以把它编写为一个一般函数，并把对象作为其参数。注意，这对于 volume()函数并不适用，因为 volume()函数需要访问类中的私有数据成员。当然，如果函数 compareVolume()作为一般函数实现，就不能使用指针 this，但它仍会非常简单：

```
//Comparing two Box objects - ordinary function version
int compareVolume( Box& box1, Box& box2) {
   double vol1= box1.volume();        //Get first Box volume
```

```
    double vol2= box2.volume();        //Get Second Box volume
    return vol1 > vol2 ? 1: (vol1 < vol2 ? -1: 0);
}
```

下面比较一下这个函数和作为类成员的 compareVolume()函数。在这个一般函数中，两个对象都是参数，但返回相同的值。可以使用这个版本执行相同的操作，如下面的语句所示：

```
cout << "The first box is "
     << (compareVolume(firstBox, secondBox) >= 0 ? "": "not ")
     << "greater than the second box."
     << endl;
```

在这种情况下，选择任何一个版本都是一样的。比较对象还有一种更好的方法，如下所述。

从函数中返回 this

如果把成员函数的返回类型指定为类类型的指针，就可以从函数中返回 this。这是为对象连续调用成员函数提供的一个非常有用的功能。下面举一个例子。

假定给 Box 类添加一个变异函数，设置盒子的长度、宽度和高度，并让这些函数返回 this：

```
class Box {
  public:
    //Constructor
    Box(double lengthValue=1.0, double widthValue=1.0,
                                    double heightValue=1.0);

    //Function to calculate the volume of a box
    double volume();

    // Function to compare two Box objects
    int compareVolume( Box& otherBox);

    //Mutator functions
    Box* setLength(double lvalue);
    Box* setWidth(double wvalue);
    Box* setHeight(double hvalue);

  private:
    double length;
    double width;
    double height;
};
```

在 Box.cpp 中实现这些函数，如下所示：

```
//setXXX() functions
Box* Box::setLength(double lvalue) {
   if (lvalue > 0) length = lvalue;
     return this;
}
```

```
Box* Box::setWidth(double wvalue) {
   if (wvalue > 0) width = wvalue;
     return this;
}

Box* Box::setHeight(double hvalue) {
   If (hvalue > 0) height = hvalue;
     return this;
}
```

下面就可以在一个语句中修改 Box 对象的所有尺寸：

```
Box aBox(10,15,25);    //Create a box
Box * pBox = &aBox;   // and a pointer to aBox
pBox->setLength(20)->setWidth(40)->setHeight(10);  /Set all dimensions of aBox
```

变异函数返回 this 指针，所以可以使用一个函数的返回值调用另一个函数。setLength()返回的指针用于调用 setWidth()，setWidth()又返回一个指针，可以接着用于调用 setHeight()。

12.7 const 对象和 const 成员函数

在停止讨论前面的例子之前，再次考虑 compareVolume()成员函数。由于不修改参数，因此在类定义中应把它声明为 const：

```
class Box {
  //Rest of the class as before…

  int compareVolume(const Box& otherBox);
};
```

当然，需要以相同的方式修改函数定义。之后再次运行这个例子。编译器在把程序的各个部分链接在一起时会出现错误。如果编译器用 C++标准来编译，下面的语句就会出错：

```
double vol2=otherBox.volume();        //Get argument volume
```

如果把一个对象指定为 const，就是告诉编译器不要修改它。在为常量对象 otherBox 调用函数 volume()时，编译器必须把 otherBox 的地址通过 this 指针传送给函数，但不能保证函数不修改该对象。错误消息是不能转换 this 指针，因为编译器在默认情况下不能改变对象的 const 性质。

对于声明为 const 的对象，只能调用也声明为 const 的成员函数。const 成员函数不会修改调用它的对象。要把成员函数声明为 const，需要在类定义中在函数声明的最后加上关键字 const。对于处理 const 参数的 compareVolume()函数，必须把 Volume()函数声明为 const。这样，类定义就变成：

```
class Box {
  public:
    //Constructor
    Box(double lengthValue=1.0, double widthValue=1.0,
                                                  double heightValue=1.0);
```

```
  //Function to calculate the volume of a box
  double volume() const;

  // Function to compare two Box objects
  int compareVolume(const Box& otherBox);

private:
  double length;
  double width;
  double height;
};
```

关键字 const 还必须放在 Box.cpp 中的 volume()函数的定义中：

```
double Box::volume()const {
  return length* width* height;
}
```

程序现在可以编译了，工作也正常了。把成员函数声明为 const，基本上确保了 this 指针指向的对象一定是常量，换言之，*this 是 const。应总是把不修改对象的成员函数声明为 const。这不会阻止其他非常量对象调用它们，而且可以在更大的范围内使用 const 对象，使代码的效率更高，更不易出错。实际上，在这个基础上，应在 Box 类中把 compareVolume()函数也声明为 const：

```
class Box {
  public:
    //Constructor
    Box(double lengthValue=1.0, double widthValue=1.0,
                                                double heightValue=1.0);

    //Function to calculate the volume of a box
    double volume() const;

    // Function to compare two Box objects
    int compareVolume(const Box& otherBox) const;

  private:
    double length;
    double width;
    double height;
};
```

当然，在 Box.cpp 的定义中，compareVolume()函数也必须是 const：

```
int Box::compareVolume(const Box& otherBox) const {
  double vol1= this->volume();            //Get current Box volume
  double vol2= otherBox.volume();        //Get argument volume
  return vol1 > vol2 ? 1: (vol1 < vol2 ? -1: 0);
}
```

其他可以声明为 const 的成员函数是访问器函数。实际上，访问器函数总是声明为 const，

因为它们仅访问数据成员的值，而不修改它。

注意把成员函数声明为 const，会影响函数的签名。也就是说，可以通过添加函数的 const 版本来重载该函数。例如，成员函数的原型如下所示：

```
int Box::compareVolume(const Box& otherBox);

int Box::compareVolume(const Box& otherBox)const;
```

该函数是 compareVolume()函数的重载版本。但是，在 const 的基础上重载成员函数时应小心，因为这有可能使使用类的人感到很迷惑。

12.7.1 类中的 mutable 数据成员

如果把对象声明为 const，就只能调用 const 成员函数。不能修改对象的数据成员值，因为它们也是 const。但是，在一些情况下，即使对象声明为 const，也需要修改类中一些选定的数据成员。

例如，一个对象从远程源(如另一台计算机)上获取数据，并把该数据存储在一个提供内部缓冲区的数据成员中。类的一个对象需要更新其内部缓冲区，即使该对象声明为 const，也是如此。

为了满足这两种情况的要求，需要完成两个任务。首先需要从 const 对象中提取出一个特定的数据成员，其次需要在 const 成员函数中修改该数据成员的值，但成员函数的 const 声明不变。为此，可以把该数据成员声明为 mutable。

为了说明其应用的方式，下面举一个简单的例子。假定为了安全起见，每次调用任何成员函数时，都在对象的一个数据成员中记录一个时间戳。该对象表示对某大厦的可控访问。此时把该对象声明为 const，但仍用时间戳记录对象上一次使用的情况。

为此，把存储时间戳的数据成员声明为 mutable，这样就在该类的声明为 const 的对象中提取了一个数据成员，并允许通过 const 成员函数修改它。具体方法是在成员声明中使用关键字 mutable，如下所示：

```
class SecureAccess {
  public:
    bool isLocked() const;
    //More of the class definition…

  private:
    mutable int time;
    //More of the class definition…
}
```

成员函数 isLocked()的实现代码如下所示：

```
bool SecureAccess::isLocked() const {
  time = getCurrentTime();                  //Store time of function call
  return lockStatus();                      //Return the state of the door
}
```

假定如果门被锁上，lockStatus()函数就返回 true，否则就返回 false。数据成员 time 声明为 mutable，它可以放在赋值语句的左边。在声明为 const 的成员函数中，只有声明为 mutable 的数据成员才能放在赋值语句的左边。

下面创建类的对象，把它声明为 const，调用成员函数 isLocked()：

```
const SecureAccess mainDoor;
bool doorState=mainDoor.isLocked();
```

由于 mainDoor 对象是 const，所以只能调用它的 const 成员函数。类 SecureAccess 中的任何 const 成员函数都可以修改存储在成员 time 中的值，而不必考虑对象是否声明为 const。如果 time 没有声明为 mutable，则任何 const 成员函数试图修改它，都会产生一个编译错误。

12.7.2 常量的强制转换

无论 const 对象是作为参数传送的对象，还是用 this 指针指向的对象，函数处理 const 对象的情形都非常少见，因此有必要把对象变成非常量。其原因可能是想把对象作为参数传送给另一个函数，该函数由其他人编写，它需要非常量的参数。为此，可以使用 const_cast<>()运算符。该运算符的一般形式如下所示：

```
const_cast<类型>(表达式)
```

表达式的类型必须是 const 类型，或与类型相同。不应使用这个运算符来破坏对象的常量性质。使用该运算符的惟一场合是在确保对象的常量性质不被破坏的情况下。

12.8 类的对象数组

如前所述，声明类的对象数组的方式与声明其他类型的数组的方式完全相同。类对象数组的每个元素都是独立创建的，为此，编译器要为每个元素调用默认构造函数。编译器不允许在定义语句中初始化数组。下面用一个例子来说明。

程序示例 12.8——创建 Box 对象的数组

修改前面的 Box 类定义，使之包含一个特定的默认构造函数。下面首先声明两个构造函数，再通过输出行跟踪每个构造函数的调用。类定义如下所示：

```
class Box {
  public:
    //Constructors
    Box();
    Box(double lengthValue, double widthValue, double heightValue);

    //Function to calculate the volume of a box
    double volume() const;

    // Function to compare two Box objects
    int compareVolume(const Box& otherBox) const;
```

```
  private:
    double length;
    double width;
    double height;
};
```

这里从原构造函数中删除了默认的参数值，因此默认构造函数只有一个。Box.cpp 中的成员函数定义放在最后一部分的末尾，但如果在不再使用默认构造函数时删除它，就需要恢复它的定义。

```
// Default constructor
Box::Box() {
  cout << " Default constructor called" << endl;
  length = width = height =1.0;
}
```

创建和使用 Box 数组的代码非常简单：

```
//Program 12.8 Creating an array of Box objects    File:  ex12_08.cpp
#include <iostream>
#include "Box.h"

using std::cout;
using std::endl;

int main() {
  cout << endl;

  Box firstBox(17.0, 11.0, 5.0);
  Box boxes[5];

  cout << "Volume of first box = "
       << firstBox.volume()
       << endl;

  const int count=sizeof boxes/sizeof boxes[0];

  cout << "The boxes array has "<<count<<"elements."
       << endl;


  cout << "Each element occupies "<< sizeof boxes[0]<<" bytes."
       << endl;

  for(int i=0; i<count; i++)
    cout << "Volume of boxes["<<i<<"]="
         << boxes[i].volume()
         << endl;

  return 0;
}
```

这个例子生成如下结果：

```
Box constructor called
Default constructor called
Default constructor called
Default constructor called
Default constructor called
Default constructor called
Volume of first box = 935
The boxes array has 5 elements.
Each element occupies 24 bytes.
Volume of boxes[0] = 1
Volume of boxes[1] = 1
Volume of boxes[2] = 1
Volume of boxes[3] = 1
Volume of boxes[4] = 1
```

例子的说明

输出结果中的第一行由声明中的构造函数调用生成：

```
Box firstBox(17.0, 11.0, 5.0);
```

输出结果中的接下来 5 行由数组的声明创建：

```
Box boxes[5];
```

在输出中可以看出，创建每个数组元素时都会调用一次默认构造函数。这个语句定义了 5 个对象，每个对象的类型都是 Box。由于定义的是一个数组，因此不能为构造函数提供参数。而由于没有提供参数，编译器就使用默认的构造函数创建数组中的 5 个对象。

在显示了 firstBox 的体积后，所显示的内容对于 Box 对象的数组来说就没有什么特别的了，使用 sizeof 运算符计算数组中的元素个数：

```
const int count=sizeof boxes/sizeof boxes[0];
```

存储在 count 中的值以通常的方式显示，如下面的语句所示：

```
cout << "The boxes array has "<<count<<"elements. "
     << endl;
```

每个元素的大小都与 Box 对象的大小相等，如下面的语句所示：

```
cout << "Each element occupies "<< sizeof boxes[0]<< " bytes. "
     << endl;
```

从输出中可以看出，Box 对象的大小是 24 字节，它对应于存储 3 个 double 类型的值所需的内存空间。成员函数的存在与否并不会影响对象的大小。

最后，在这个例子中，用一个循环显示 Boxes 数组中每个元素的体积：

```
for(int i=0; i<count; i++)
  cout << "Volume of boxes[" << i << "]="
       << boxes[i].volume()
       << endl;
```

显然，默认构造函数的初始化工作正常，因为每个数组元素的体积都是 1。

12.9 类对象的大小

从上一个例子可以看出，使用 sizeof 运算符可以获得类对象的大小，而且使用该运算符的方式与前面对基本数据类型使用该运算符的方式完全相同。可以把这个运算符用于一个特定的对象或类类型。类对象的大小一般是类中数据成员大小的总和，但在某些机器上，类对象的大小比类中所有数据成员的字节总和大。不必担心这个问题，但应知道原因。

在一些计算机上，由于性能方面的原因，两个字节的变量必须放在 2 的倍数的地址中，4 个字节的变量必须放在 4 的倍数的地址中，依此类推。这样，在某些情况下，编译器就必须在存储不同值的内存之间留下一定的空隙。如果在这样的机器上，有 3 个占用两个字节的变量，其后是一个需要 4 个字节的变量，就需要留下两个字节的空隙，才能在正确的边界上放置 4 个变量。此时，所有 4 个变量所占用的空间就大于各变量实际需要的字节数总和。下面创建一个例子，说明在 PC 上和在需要边界对齐的另一个系统上对象所占字节数的不同。

程序示例 12.9——受边界对齐影响的对象大小

在 Box 类上定义一个变体，称为 SizeBox，专门用于说明边界对齐对对象大小的影响：

```
//SizeBox.h
#ifndef SIZEBOX_H
#define SIZEBOX_H

class SizeBox {
  public:
    SizeBox();
    int totalSize();                    //Sum of sizes of members

  private:
    char* pMaterial;
    double length;
    double width;
    double height;
};
#endif
```

下面是成员函数的定义：

```
//SizeBox.cpp
#include "SizeBox.h"

SizeBox::SizeBox():
     length(1.0), width(1.0),height(1.0), pMaterial("Cardboard") {}

//Sum of sizes of members
int SizeBox::totalSize() {
  return sizeof(length)+ sizeof(width)+ sizeof(height )+ sizeof(pMaterial);
}
```

类只需创建它的实例，并提供其成员所占用的字节数即可。所以，惟一的成员函数就是构造函数和函数 totalSize()。这个类中有一个额外的数据成员，它是指向非空字符串的指针，该字符串记录了盒子的材料类型。下面的代码创建了一些对象，并报告了它们占用的内存量：

```
//Program 12.9 Trying object sizes    File: ex12_09.cpp
#include <iostream>
#include "SizeBox.h"

using std::cout;
using std::endl;

int main() {
  SizeBox box;
  SizeBox boxes[10];
  cout << endl              << "The data members of a Box object occupy "
       << box.totalSize()   << " bytes.";

  cout << endl              << "A single Box object occupies "
       << sizeof SizeBox    << " bytes.";

  cout << endl              << "An array of 10 Box objects occupies "
       << sizeof(boxes)     << " bytes."
       << endl;
  return 0;
}
```

其结果如下所示：

```
The data members of a Box object occupy 28 bytes.
A single Box object occupies 32 bytes.
An array of 10 Box objects occupies 320 bytes.
```

例子的说明

输出表明，类对象占用的字节数并没有增加。SizeBox 对象占用的内存是 3 个 double 成员所需要的 24 个字节，再加上 4 个字节来存储指针成员 pMaterial，总共 28 个字节，如输出的第一行所示。但在输出 SizeBox 对象的大小时，结果为 32 字节，并且这一点由包含 10 个对象的数组来证实。

额外的 4 个字节就是边界对齐的结果，如图 12-6 所示。如前所述，在许多机器上，编译器必须为 8 个字节的变量分配 8 的倍数的地址，给 4 个字节的变量分配 4 的倍数的地址，以提高性能。

4 字节	4 字节	8 字节	8 字节	8 字节
pMaterial	未使用	length	width	height

图 12-6　因边界对齐而丢失的空间

SizeBox 对象由一个指针和三个 double 类型的成员组成。指针 pMaterial 放在 4 的倍数的地

址上，因为它占用 4 个字节。三个 8 字节成员必须占用 8 的倍数的地址。如果下一个可用的地址不是 8 的倍数，编译器就必须留下一个空隙——在本例中是 4 个字节——使 length 成员放在一个有 8 字节边界的地址上。

也许可以重新安排数据成员，例如把 pMaterial 放在最后，使类对象占用较少的空间，但这通常做不到。编译器必须考虑对象数组的情况，而对象数组要求每个对象都放在有 8 字节倍数的边界的地址上。

下面回过头来，继续介绍类。

12.10 类的静态成员

类的数据成员和成员函数都可以声明为 static。类的静态数据成员可以在类的范围内存储数据，这种数据独立于类类型中的任何对象，但可以由这些对象访问。它们把类作为一个整体来记录类的属性，而不是记录各个对象的属性。使用静态数据成员可以存储类的特定常量，或存储类中对象的一般信息，例如类中有多少个对象等。

静态成员函数有一种独立于单个类对象的计算能力，但如果需要，任何类对象都可以调用该静态成员变量。如果该函数是一个公共成员，还可以从类的外部调用。静态成员函数的一个常见用法是无论是否声明了类的对象，都可以操作静态数据成员。

这个主题是在类定义中讨论的，所以其影响要比类外部的 static 关键字多一些，需要详细论述。下面先介绍静态数据成员。

12.10.1 类的静态数据成员

类的静态数据成员与作为一个整体的类相关，而与类的各个对象无关。在把类的数据成员声明为 static 时，静态数据成员就只定义一次，而且即使类没有创建对象实例，该静态数据成员依然存在。每个静态数据成员都可以在已创建的任何类对象中访问，并在已有的所有对象之间共享。对象包含类的每个原数据成员的独立副本，但无论定义了多少个类对象，静态数据成员总是只有一个。

使用静态数据成员可以记录在类的范围内使用的信息。静态数据成员的一个用途是计算有多少个类对象存在。在类定义中添加如下语句，为 Box 类添加一个静态数据成员：

```
static int objectCount;          //Count of objects in existence
```

图 12-7 演示了在类的所有对象中如何共享静态成员 objectCount。

现在有一个问题。如何初始化静态数据成员？不能把它放在类声明中，类声明只是对象的一个蓝图，不允许初始化值。也不能在构造函数中初始化它，因为每次调用构造函数时，都要递增这个数据成员。而且，即使不存在对象，这个数据成员也存在(此时并没有调用构造函数)。同样，还不能在另一个成员函数中初始化它，因为成员函数与对象相关，而该数据成员应在创建任何对象之前初始化。

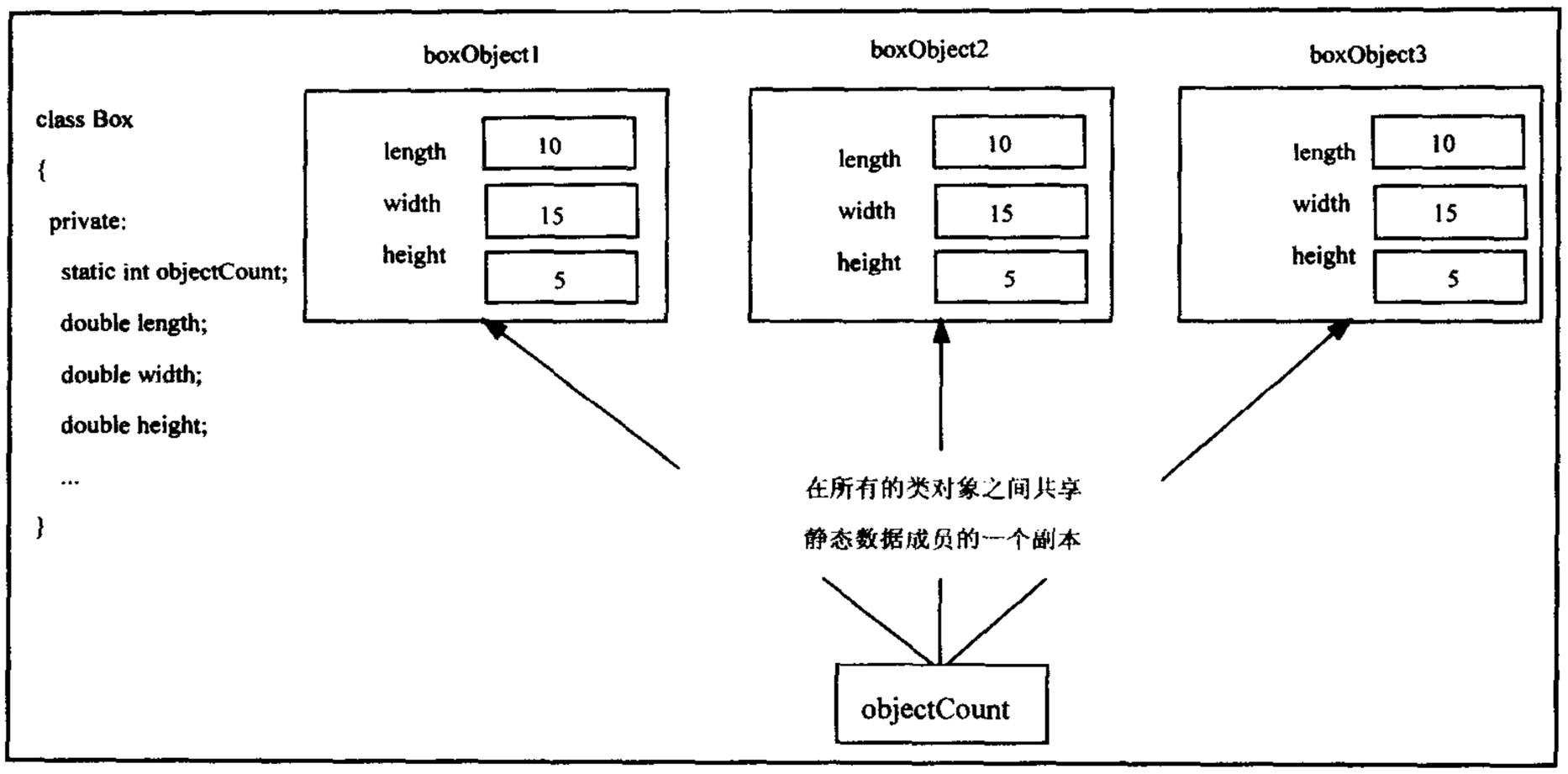


图 12-7　在对象之间共享静态类成员

这个问题的答案是在类声明的外部初始化它，初始化语句如下所示：

```
int Box::objectCount=0;              //Initialize static member of class Box
```

即使静态数据成员被指定为私有成员，仍可以以这种方式初始化它。事实上，这也是初始化它的惟一方式。当然，由于它是私有的，就不能在类的外部直接访问 objectCount。

由于这个语句定义并初始化了类的静态成员，因此该成员在程序中就只能定义一次。因此，定义它的语句就应放在 Box.cpp 文件中。

注意：

关键字 static 并没有包含在定义中，事实上，不能包含这个关键字。但是，必须使用类名和作用域解析运算符来限定成员名，这样编译器才知道这是类的一个静态成员。否则，就会创建一个与类没有任何关系的全局变量。

程序示例 12.10——计算实例的个数

下面给程序示例 12.8 添加静态数据成员和计数的对象。在类定义中只需添加两个语句，一个语句声明新的静态数据成员，另一个语句定义提取其值的函数：

```
class Box {
  public:
    //Constructor
    Box();
    Box(double lengthValue, double widthValue, double heightValue);

    //Function to calculate the volume of a box
    double volume() const;

    // Function to compare two Box objects
    int compareVolume(const Box& otherBox) const;

    int getObjectCount() const {return objectCount;}
```

```
  private:
    static int objectCount;          //Count of objects in existence
    double length;
    double width;
    double height;
};
```

getObjectCount()函数声明为 const，因为它不修改类中的任何数据成员。在 Box.cpp 文件的末尾添加如下语句，初始化静态成员 objectCount：

```
//Initialize static member of class Box
int Box::objectCount=0;
```

还必须修改两个构造函数，在创建对象时更新计数：

```
// Default constructor
Box::Box() {
  cout<<" Default constructor called"<<endl;
  ++objectCount;
  length = width = height =1.0;
}

// Constructor definition using an initializer list
Box::Box(double lvalue, double bvalue, double hvalue) :
                          length(lvalue), width(wvalue), height(hvalue) {
    cout << "Box constructor called" << endl;
    ++ objectCount;

  //Ensure positive dimensions
  if(length <= 0)
    length = 1.0;
  if(width <= 0)
    width = 1.0;
  if(height <= 0)
    height = 1.0;
}
```

修改程序示例 12.8 中的 main()，输出对象的个数：

```
//Program 12.10 Counting Box objects    File:ex12_10.cpp
#include <iostream>
#include "Box.h"

using std::cout;
using std::endl;

int main() {
  cout << endl;

  Box firstBox(17.0, 11.0, 5.0);
  cout << "Object count is " << firstBox.getObjectCount()<<endl;
  Box boxes[5];
  cout << "Object count is " <<firstBox.getObjectCount()<<endl;
```

```
    cout << "Volume of first box = "
         << firstBox.volume()
         << endl;

    const int count=sizeof boxes/ sizeof boxes[0];

    cout << "The boxes array has "<<count<<"elements."
         << endl;

    cout << "Each element occupies "<< sizeof boxes[0]<<" bytes."
         << endl;

      for(int i=0; i<count; i++)
      cout << "Volume of boxes[" << i << "]="
           << boxes[i].volume()
           << endl;

      return 0;
    }
```

这个程序的输出结果如下所示：

```
Box constructor called
Object count is 1
Default constructor called
Default constructor called
Default constructor called
Default constructor called
Default constructor called
Object count is 6
Volume of first box = 935
The boxes array has 5 elements.
Each element occupies 24 bytes.
Volume of boxes[0] = 1
Volume of boxes[1] = 1
Volume of boxes[2] = 1
Volume of boxes[3] = 1
Volume of boxes[4] = 1
```

例子的说明

这段代码说明，的确只有一个静态成员 objectCount，两个构造函数都更新了它。在两种情况下都为 firstBox 对象调用了函数 getObjectCount()，但在第二次调用时，可以使用数组中的任何元素，但得到的结果都相同。

当然，只记录了已创建的对象个数，无法确定对象何时释放，所以该计数不需要反映某一时刻存在的对象个数。下一章将介绍如何计算已释放的对象个数。

注意，即使在类定义中添加了 objectCount，在输出中 Box 对象的大小也与前面例子中的相同。这是因为静态数据成员不是任何对象的一部分，它们属于类。由于静态数据成员不是类对象的一部分，声明为 const 的成员函数就可以修改类的静态数据成员，而不会影响它们的常量性质。

注意：

必须在类定义中定义静态数据成员，否则编译器就会生成错误。类定义中的声明并没有定义静态变量，只有在类外部初始化它，才算定义了它，它才存在。

1. 访问静态数据成员

假定在某个时刻，把 objectCount 数据成员声明为公共成员：

```
class Box {
  public:
    static int objectCount;          //Count of objects in existence

    //Constructors
    Box();
    Box(double lengthValue, double widthValue, double heightValue);

    //Function to calculate the volume of a box
    double volume() const;

    // Function to compare two Box objects
    int compareVolume(const Box& otherBox) const;

    int getObjectCount() const {return objectCount;}

  private:
    double length;
    double width;
    double height;
};
```

现在不需要使用函数 getObjectCount()，而应在 main()中输出对象的个数，语句如下：

```
std::cout << "Object count is " << firstBox.objectCount << std::endl;
```

另外，即使没有创建对象，静态变量也存在。也就是说，在创建 firstBox 对象之前，就可以获得对象个数，但如何引用数据成员？答案是把类名 Box 和作用域解析运算符用作限定符：

```
std::cout << "Object count is " << Box::objectCount << std::endl;
```

实际上，无论是否存在对象，总是可以使用类名访问类的公共静态成员。读者可以修改上面的例子，看看是否可行。

2. 静态数据成员的类型

静态数据成员不是类中各个对象的一部分，所以它可以与类具有相同的类型，例如 Box 类可以包含 Box 类型的静态数据成员。这看起来似乎很奇怪，但非常有效。下面就使用 Box 类来验证一下。

假定需要一个标准的“参考”盒子，并能以各种方式对 Box 对象和标准盒子建立关联。当然，可以在类的外部定义一个标准的 Box 对象，但如果要在类的成员函数中使用它，就需要创建一个外部依赖，而最好不要存在这样的外部依赖。另一个解决方案是把标准 Box 定义为类的一个静态成员：

```
class Box {
  public:
    //Constructors
    Box();
    Box(double lengthValue, double widthValue, double heightValue);

    //Function to calculate the volume of a box
    double volume() const;

    // Function to compare two Box objects
    int compareVolume(const Box& otherBox) const;

  private:
    const static Box refBox;          //Standard reference box
    double length;
    double width;
    double height;
};
```

由于 refBox 是一个不应修改的标准 Box 对象，因此还把它声明为 const。但仍需要在定义它的类外部初始化它。在 Box.cpp 文件中添加如下语句来定义 refBox：

```
const Box Box::refBox(10.0, 10.0, 10.0);
```

这个语句调用 Box 类的构造函数来创建 refBox。在程序中，因为类的静态数据成员是在创建任何对象之前创建的，所以类中至少存在一个 Box 对象，即标准的 refBox 对象。

类对象的任何成员函数都可以访问 refBox，因此 refBox 可以用于所有成员，但不能从类的外部访问它，因为它声明为私有成员。如果类常量在类的外部十分有用，就可以把它声明为公共成员。只要它声明为 const，就不能修改。

12.10.2 类的静态成员函数

把函数成员声明为 static，就可以使它独立于类的对象。与静态数据成员一样，即使没有创建类的对象，类的静态函数成员也存在。声明类中的静态函数非常简单，只需使用关键字 static 即可，就像声明数据成员 objectCount 一样。把前面例子中的 getObjectCount()函数声明为静态函数，如下所示：

```
class Box {
  public:
    //Constructors
    Box();
    Box(double lengthValue, double widthValue, double heightValue);

    //Function to calculate the volume of a box
    double volume() const;

    // Function to compare two Box objects
    int compareVolume(const Box& otherBox) const;
```

```
    static int getObjectCount() {return objectCount;}

  private:
    static int objectCount;          //Count of objects in existence
    double length;
    double width;
    double height;
};
```

注意:

静态成员函数不能声明为 const。因为静态成员函数与类的对象无关，它没有 this 指针，所以不能使用 const 关键字。

静态成员函数的优点在于：即使不存在类的对象，它们也存在，并且可以调用。可以把类名作为限定符来调用静态成员函数。例如：

```
std::cout<<"Object count is "<<Box::getObjectCount()<<std::endl;
```

当然，如果创建了类的对象，就可以通过该类的对象来调用静态成员函数，其方法与调用其他成员函数的方法相同。例如：

```
std::cout<<"Object count is "<<firstBox.getObjectCount()<<std::endl;
```

它与一般成员函数的区别是，静态函数不能访问调用它的对象。为了让静态成员函数访问类的对象，需要把它作为该函数的一个参数传送。之后，就必须使用限定的名称在静态函数中引用类对象的成员(就象一般全局函数在访问公共数据成员一样)。

当然，根据访问权限，静态成员函数是类的一个具有完全访问权限的成员。如果把同一个类的对象作为参数传送给静态成员函数，它就可以访问该对象的私有成员和公共成员。这么做没有什么意义，只是说明在 Box 类中可以包含静态函数的定义，如下所示：

```
static double sum(Box theBox) {
  return theBox.length + theBox.width + theBox.height;
}
```

即使把 Box 对象作为参数传送，也可以访问私有数据成员。当然，用成员函数来访问私有数据成员更有意义，而不是用静态函数来访问。

12.11 本章小结

在本章和第 11 章中，介绍了 C++中类的基本概念，以及定义和使用类的一般规则。但是，这仅是开始。实现可应用于类对象的操作，以及理解类内部的机制还有许多内容要学。

在后续的章节中，将以本章的内容为基础，学习如何扩展类的功能，探讨使用类的更复杂方式。本章的要点如下所示：

- 类提供了定义自己的数据类型的一种方式。类可以反映某个问题所需要的对象类型。
- 类可以包含数据成员和成员函数。类的成员函数总是可以自由访问该类中的数据成员。

- 类的对象用构造函数来创建和初始化。在声明对象时，会自动调用构造函数。构造函数可以重载，以提供初始化对象的不同方式。
- 类的成员可以指定为 public，此时它们可以由程序中的任何函数自由访问。另外，类的成员还可以指定为 private，此时它们只能被类的成员函数或友元函数访问。
- 类的数据成员可以定义为 static。无论类中创建了多少个对象，类中的静态数据成员都只有一个。
- 可以在类对象的成员函数中访问类的静态数据成员，它们不是类对象的一部分，类对象的大小不包括静态数据成员的字节数。
- 即使没有创建类的对象，类的静态函数成员也存在，并可以调用。
- 类的每个非静态函数成员上都包含指针 this，它指向调用该函数的当前对象。
- 类的静态函数成员不包含指针 this。
- 类中声明为 const 的成员函数不能修改类对象的数据成员，除非数据成员声明为 mutable。
- 把类对象的引用用作函数调用的参数，可以避免产生把复杂对象传送给函数的系统开销。
- 副本构造函数可以用类中已有的对象初始化同一个类中的新对象。如果没有定义类的构造函数，编译器就会生成默认的副本构造函数。

12.12 练习

1. 创建一个简单的类 Integer，它只有一个私有数据成员 int。为这个类提供构造函数，并使用它们输出创建对象的消息。提供类的成员函数，获取和设置数据成员，并输出该值。编写一个测试程序，创建和操作至少 3 个 Integer 对象，验证不能直接给数据成员赋值。

在测试程序中获取、设置和输出每个对象的数据成员值，以验证这些函数。

2. 修改上一题类 Integer 的构造函数，把数据成员初始化为初始化列表中的 0，并实现类的副本构造函数。

编写一个成员函数，比较当前对象和作为参数传送的 Integer 对象。如果当前对象小于参数，该函数就返回 - 1，如果它们相等，函数就返回 0，如果当前对象大于参数，函数就返回 +1。测试该函数的两个版本：第一个版本的参数按值传送，第二个版本的参数按引用传送。在调用函数时，构造函数会输出什么结果？解释出现这种结果的原因。

类中的函数不能是重载函数，为什么？

3. 为类 Integer 实现成员函数 add()、subtract()和 multiply()，对当前对象和 Integer 类型的参数值进行加、减和乘法运算。在类中用 main()演示这些函数的操作，main()创建几个 Integer 对象，它们分别包含值 4、5、6、7 和 8，再使用这些对象计算 $4*5^3+6*5^2+7*5+8$ 的值。实现这些函数，使计算和结果的输出在一个语句中完成。

4. 修改题 2 的解决方法，把 compare()函数实现为类 Integer 的一个友元。

第 13 章　类 的 操 作

如果对类对象的操作要求安全而有效，就需要理解类对象的创建和删除。本章就介绍创建和删除类对象的基本知识。

为了说明某些事情需要以某种特定的方式来完成的原因，本章和第 14 章将逐步开发自己的类。中间阶段有时有一些缺点需要克服。一旦理解了这些缺点，就很容易看出在某些环境下使用某种方式的原因。本章将详细论述这些内容，但主要是因为类有许多潜在的功能。

本章主要内容

- 如何对类类型使用指针和引用
- 类的析构函数的定义，何时实现它
- 如何在类中动态分配内存
- 何时必须实现副本构造函数，实现方式是什么
- 如何限制对类的访问
- 嵌套的类

13.1　类对象的指针和引用

第 12 章介绍了如何声明和使用类对象的指针和引用，其方法与基本数据类型完全相同。类对象的指针和引用是面向对象编程的关键特性，具有各自的优点。

在下面 3 种环境下可以对类对象使用指针：

- 作为在对象上执行操作的一种方式，即使用指针成员访问运算符->来调用函数
- 作为函数的参数
- 作为类的数据成员

第一种方式使多态调用函数成为可能，即所调用的函数取决于指针所指向的对象类型。第 16 章将详细介绍这个功能。

把对象的指针作为参数传送给函数，可以避免按值传送机制中隐含的复制操作。这极大地提高了程序的效率，特别是涉及到很大的对象时，就更是如此，因为复制很大的对象是非常费时的。

如果类把对象的指针包含为它的一个数据成员，就可以把一系列对象链接在一起，甚至可以把不同类型的对象链接起来。这对于把数据组织为结构，如图形或树图，或把数据组织为链表(详见第 11 章)来说，是必不可少的。稍后论述链表。

对象引用为函数的参数类型也很重要。一般情况下，按引用传送大对象要比按值传送快得多，因为可以避免按值传送机制中的复制过程。它们也是实现副本构造函数的基础。

注意：

本章的后面将介绍从函数中返回指针的优点。

13.2 指针作为数据成员

把指针声明为类的数据成员是很简单的。下面先看看第 11 章中介绍的一个例子：对象的链表。把这个理念和把指针用作类的数据成员的例子组合起来，再加上第 12 章介绍的一些主题。以后还要介绍，即使很简单的类也会产生意想不到的编译错误，我们必须识别并处理这些错误。

注释：

不需要创建自己的链表类，在标准库中已经定义了非常灵活的版本。但是，用自己的类来完成这一任务，能更好地理解链表的工作原理。

这里定义一个类，来表示任意个 Box 对象的集合(其中 Box 是第 12 章介绍的类)。Box 对象表示要传送的一个产品单元，Box 对象的集合表示一货车盒子，所以这个类称为 TruckLoad。TruckLoad 类可以计划如何装载货车，让司机以正确的顺序卸货。

最终，这种安排可以在分销商办公室使用的程序中使用。例如，办公室给产品排好顺序，并安排产品的分发顺序。分发的方法就是用货车运输。在这种模式下，在库房装载货车的顺序就非常重要，因为货车司机在去往各个目的地的过程中，必须按照正确的顺序卸货。

对于这种情况，使用链表比较合适。也可以使用一个数组，但链表有两个重要的优点：第一，数组在声明时，其大小必须是固定的，而链表的长度可以根据需要来定，这有利于提高内存的管理。第二，分销商办公室雇员计划每次托运时，可能需要在链表的开始、最后或中间添加数据项。在链表中，可以高效地实现这个功能，而数组不能。

我们要从一个 Box 对象中创建新的 TruckLoad 对象，也可以从一个 Box 对象的数组中创建。还要在 TruckLoad 对象中添加 Box 对象，提取 TruckLoad 对象中的所有 Box 对象。下面就开始创建这个类。

首先需要考虑如何把 Box 对象收集在一起，让它们表示一个整体，即 TruckLoad 对象。也就是说，需要一个编程设备把任意个 Box 对象链接在一起。Box 对象没有内置把自己与另一个 Box 对象链接在一起的功能，而修改 Box 类的定义，使之包含这个功能，会与盒子原来的理念不一致—— 盒子本来不需要这个功能。

要把 Box 对象收集到一个组中，一种方式是定义另一个对象 Package。该对象有两个重要的功能：它可以“包含”Box 对象，还可以与另一个 Package 对象链接起来。如图 13-1 所示。

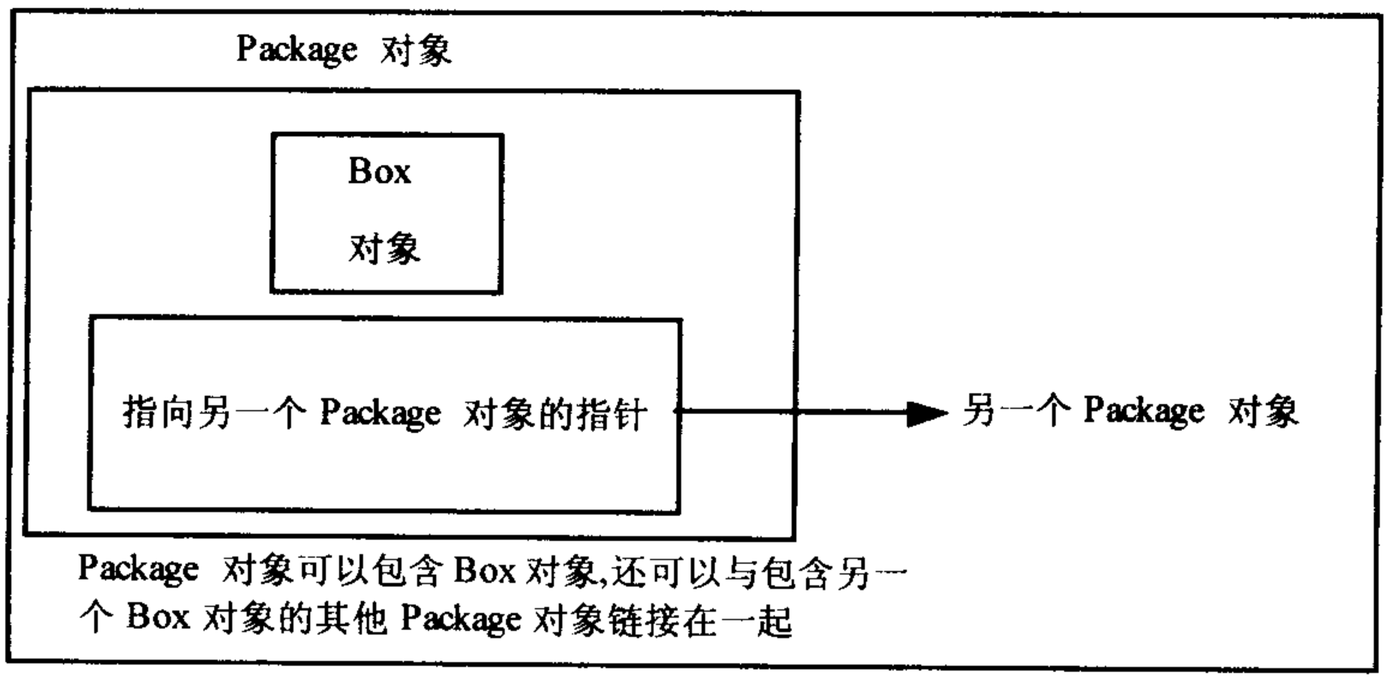


图 13-1 Package 对象的内容

图 13-1 说明，每个 Package 对象都包含一个 Box 对象，并构成了一个 Package 对象链，这是使用指针链接在一起的。因此，Package 对象是组成该列表的一个元素。

注意：

Box/Package 关系说明，Package 是 Box 对象的容器，还提供了链接其他 Package 对象的方式。这个概念是很通用的，可以以相同的方式为任何类型的对象设置容器对象。

在这种情况下，每个 Package 对象都包含一个 Box 对象，而 Package 对象的集合由 TruckLoad 对象创建和管理。TruckLoad 对象表示一货车盒子的实例。在货车中可以有任意多个盒子，每个盒子都位于一个 Package 对象中。Package 对象提供了 TruckLoad 对象跟踪它包含的 Box 对象的机制。这些对象之间的关系如图 13-2 所示。

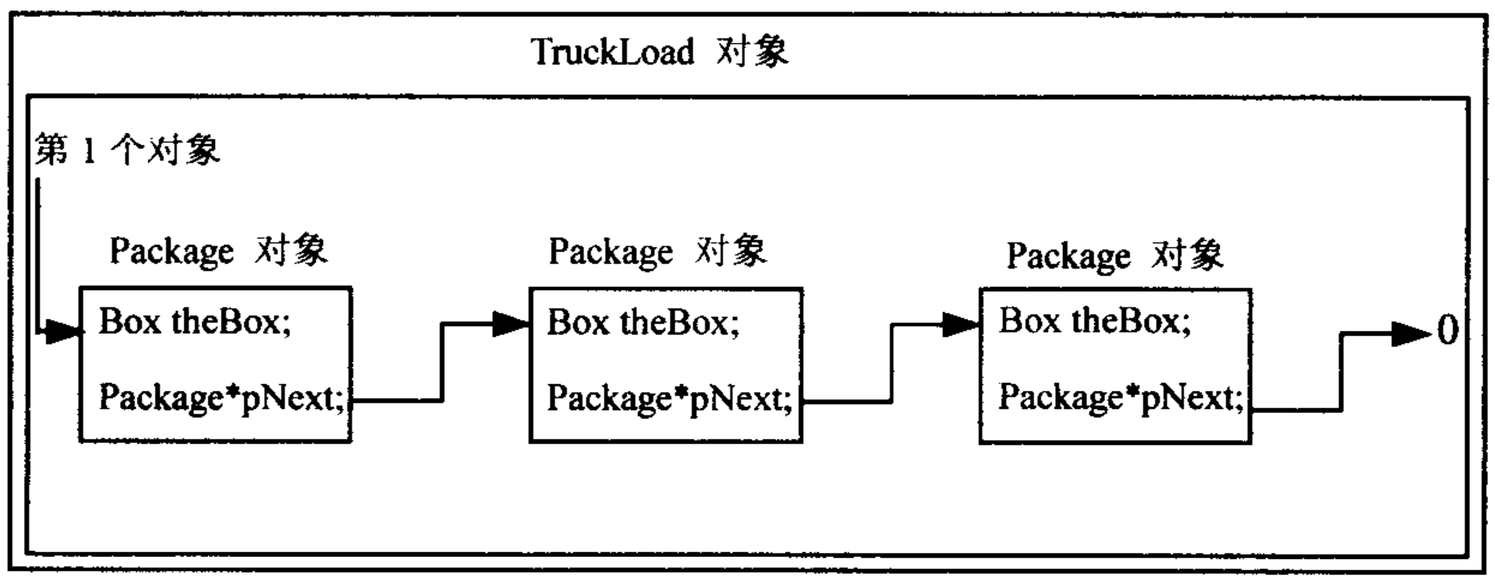


图 13-2　三个对象的链表

图 13-2 显示了一个 TruckLoad 对象，它包含一组 Package 对象，每个 Package 对象又包含一个 Box 对象和指向下一个 Package 对象的指针。这个 TruckLoad 对象包含了 3 个 Package 对象，但它可以包含一个或 1000 个对象。TruckLoad 对象只需跟踪列表中的第一个 Package 对象。按照 Package 对象中的 pNext 指针链接，只要从一个对象移动到下一个对象，就可以找出列表中的任何对象。

在这个基本的实现中，我们仅在列表的最后添加了 Package 对象。此时，Package 类的构造函数还需要创建一个不包含后续 Package 的对象，即 pNext 成员为空的 Package 对象。

要在列表的最后添加一个 Package 对象，可以在 Package 类中定义一个函数，该函数可以用新对象的地址更新列表中最后一个对象的指针成员 pNext。管理这个更新的过程在表示列表的 TruckLoad 类中。

提示：

在比较高级的实现中，实现一个在列表的任意位置添加 Package 对象的函数是相当容易的。标准库链表 list<>就支持这个功能，详见第 20 章。

13.2.1　定义 Package 类

在上述的讨论中，编写 Package 类的第一个想法是设计一个包含两个数据成员的类，一个数据成员的类型是 Box，另一个数据成员的类型是指向 Package 的指针。

```
class Package {
```

```
  public:
    Package(Box* pBox) : theBox(*pBox), pNext(0){} //Constructor

    Box getBox() const {return theBox;}              //Retrieve the Box object
    Package* getNext() const{return pNext;}          //Get next package address
    void setNext(Package* pPackage) {pNext=pPackage;} //Add to end of list

  private:
    Box theBox;                                      //The Box object
    Package* pNext;                                  //Pointer to the next Package
} ;
```

在介绍这个类的其他功能之前，先考虑一下实现这个类定义的结果。在 Package 构造函数从 Box 对象中创建 Package 对象时，会解除传送为参数的指针的引用，以初始化数据成员 theBox。这里有一个非常重要的问题：Package 构造函数创建了 Box 对象的副本。

现在好好想一想这个过程。Package 构造函数先复制已存在的 Box 对象，再添加一个指向 Package 的指针，从而创建了一个 Package 对象。则第一个问题是，显然这浪费了内存，为什么 Package 对象要包含已有 Box 对象的副本？

第二个问题是比较实际的，涉及到这个 Package 构造函数中的复制过程。每次创建 Package 对象时，都会逐个字节地把 Box 对象从原位置复制到新对象中。复制过程一般是比较费时的。如果被复制的对象比较大，或要在一个程序中复制大量的对象，程序的效率就比较低。如果将来要扩展程序的规模，这就是一个成本很高的过程。

第三个问题也比较实际：如果有 Box 对象的多个副本，该如何确保它们都包含相同的值？假定因某种原因，一个 Box 对象的值必须调整，如何确保该对象的其他副本以某种方式修改？这显然不是一个切实可行的方法。图 13-3 显示了实现这个 Package 类的结果。

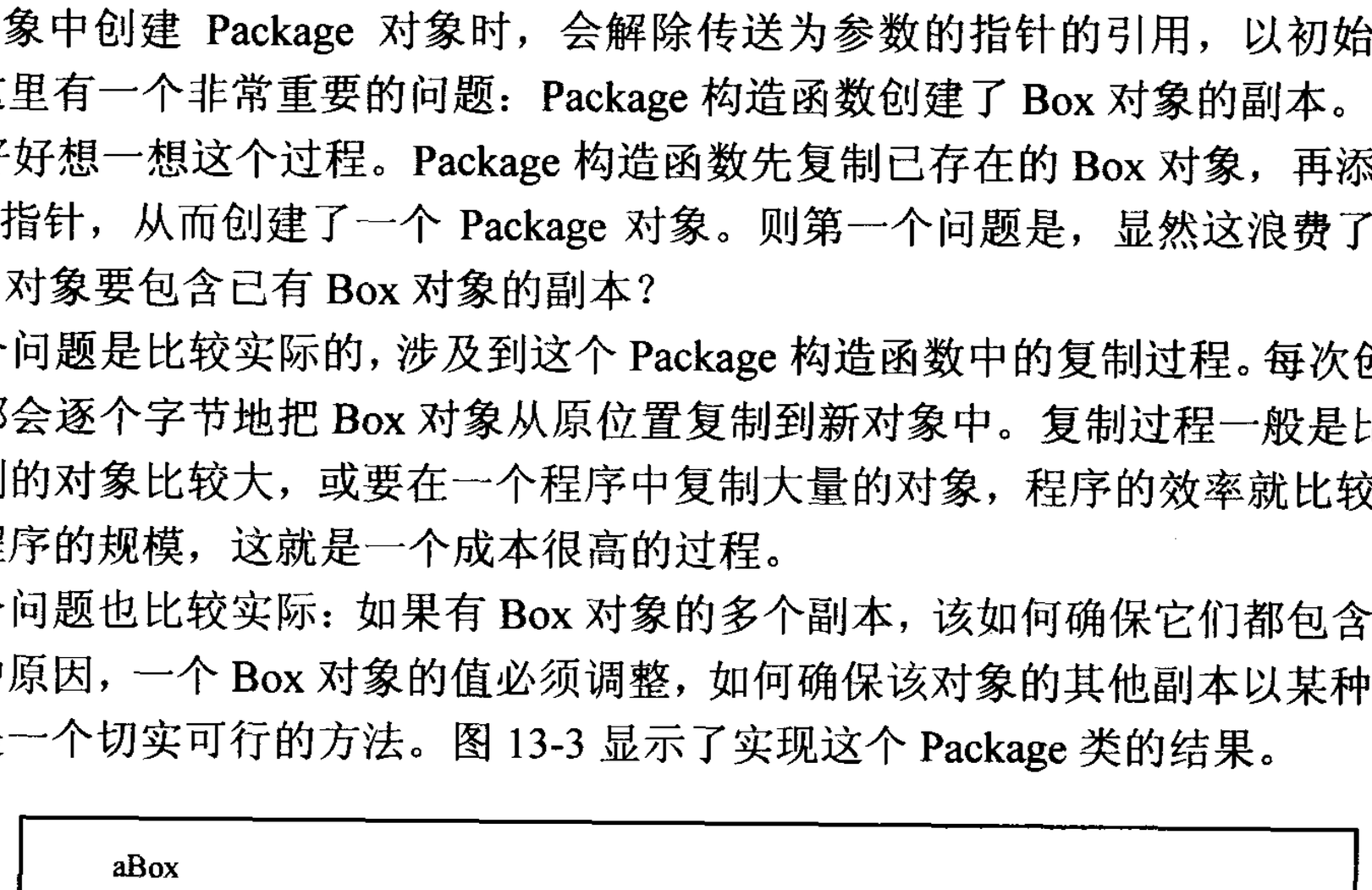


图 13-3　到处都是 Box 对象的副本

从图中可以看出，每个 Box 对象都存在好几个副本。一个副本是由 Package 构造函数生成的。注意 getBox()成员函数(该函数使 Box 的私有数据成员可以在 Package 类的外部使用)在每次调用时，也生成了 Box 数据成员的副本。这不仅会出问题，执行时间和内存也大大增加了。如果说这会产生混乱，只是低估了它。显然，Package 类包含源 Box 对象的一个副本并不合适。需要重新考虑类设计的这个方面。

一个解决方案是重新设计 Package 类，使每个 Package 对象包含指向原 Box 对象的指针(而不是 Box 对象的副本)。于是，类定义变成：

```
class Package {
  public:
    Package(Box* pNewBox) : pBox(pNewBox), pNext(0){}  //Constructor
    Box* getBox() const {return pBox;}          //Retrieve the Box pointer
    Package* getNext() const{return pNext;}     //Get next package address
    void setNext(Package* pPackage) {pNext=pPackage;} //Add to end of list

  private:
    Box* pBox;                                  //Pointer to the Box
    Package* pNext;                             //Pointer to the nextPackage
} ;
```

这看起来就好多了；至少没有前面的问题了。现在仅包含指向 Box 对象的指针，没有重复的 Box 对象。图 13-4 显示了包含 Package 类的这个定义的列表。

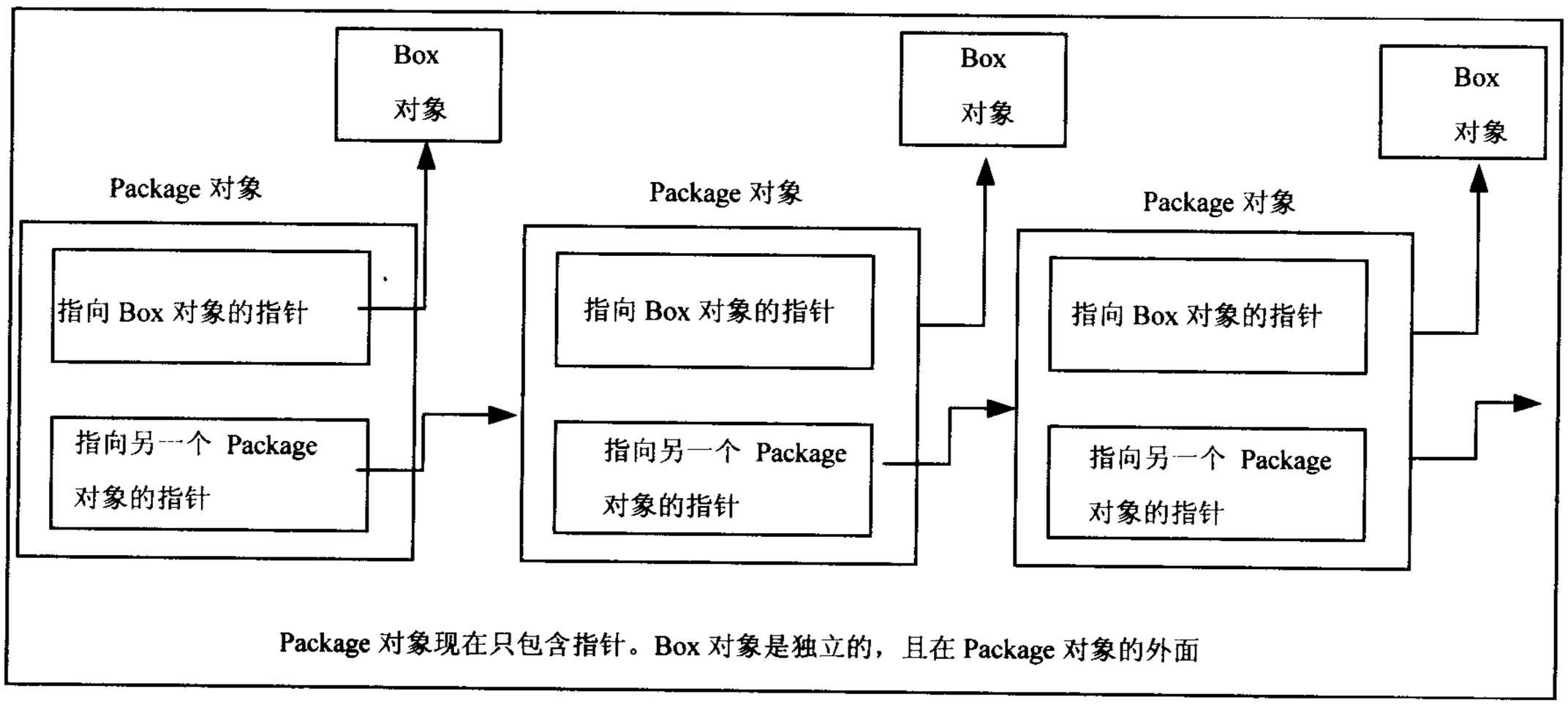


图 13-4　只包含指针的 Package 列表

getBox()函数现在允许在 Package 类的外部访问该类的私有 Box*成员。为 Package 对象调用的函数 getNext()会返回列表中下一个 Package 对象的地址(如果到达列表的末尾，就返回空指针)。

setNext()函数更新指向列表中下一个 Package 对象的指针，它在 pNext 指针中存储了传送为参数的地址。为了在列表的末尾添加一个新的 Package 对象，只需把新对象的地址传送给 setNext()函数即可。

目前，Package 类有足够的能力满足我们的需要。下面使用它来实现 TruckLoad 类中的列表。

注释：

在开发容器类时，例如 TruckLoad 类，需要仔细考虑该类的数据成员是否应是源对象的副本。有时这并不重要，但通常会比较重要。在使用容器类时，常常应避免包含副本。而使用指针并不是惟一的解决方案。后面将介绍如何使用引用来避免复制对象。

13.2.2 定义 TruckLoad 类

TruckLoad 类表示一组 Package 对象。这个类必须提供创建和扩展列表的所有功能，以及提取 Box 对象的方式。如果在列表中，把第一个 Package 对象的地址存储为 TruckLoad 对象的一个数据成员，就可以使用 Package 类中的 getNext()函数，通过 pNext 指针链获取列表中的其他 Package 对象。显然，对列表中的每个 Package 对象都要调用一次 getNext()函数，所以 TruckLoad 对象需要跟踪最近提取出来的对象。存储最后一个 Package 对象的地址也是有用的，因为这可以在列表的末尾添加新对象。如图 13-5 所示。

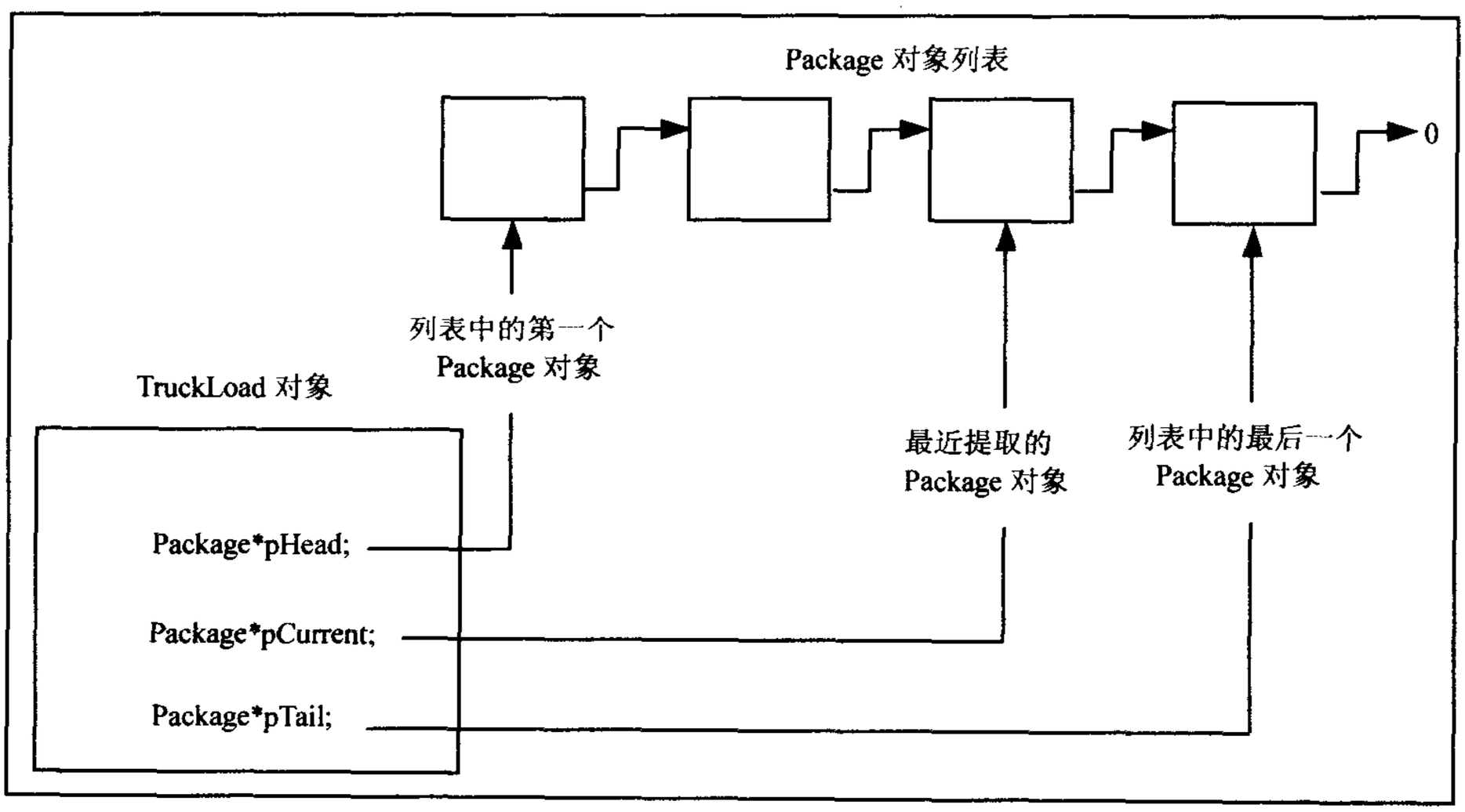


图 13-5　处理列表需要的信息

下面考虑一下从 TruckLoad 对象中提取 Box 对象的过程。起始点是列表中的第一个对象，因此需要在 TruckLoad 类中设计一个提取该对象的函数，称之为 getFirstBox()。如果跟踪最近从 TruckLoad 对象中提取出来的 Package 对象，就可以使用函数 getNextBox()从列表的下一个 Package 对象中提取 Box 对象，并更新上一次提取出来的 Package 对象的记录。另一个有用的功能是在列表中添加 Box 对象，因此还需要一个成员函数，称之为 addBox()。

根据上面的讨论，TruckLoad 类的最初定义如下所示：

```
class TruckLoad {
  public:
    TruckLoad(Box* pBox = 0, int count = 1);       //Constructor

    Box* getFristBox();                            //Retrieve the first Box
    Box* getNextBox();                             //Retrieve the next Box
    void addBox(Box* pBox);                        //Add a new Box to the list

  private:
    Package* pHead;                                //First in the list
   Package* pTail;                                //Last in the list
   Package* pCurrent;                             //Last retrieved from the list
};
```

这里把所有的数据成员都声明为 private，因为这些成员都不需要在类的外部访问。这个类还声明了一个构造函数，但如果加上参数的默认值，它就可以变成 3 个构造函数。如果省略构造函数的两个参数，该构造函数就变成默认的构造函数，创建一个空列表。如果只省略 count 参数，该构造函数就会创建只包含一个 Box 对象的列表，因为 count 默认为 1。如果指定两个参数，该构造函数就会从数组 pBox 中创建一个包含给定数量(count)的 Box 对象的列表。稍后介绍如何在定义中实现这些 TruckLoad 构造函数。

提取 Box 对象的机制是通过成员函数 getFirstBox()和 getNextBox()实现的。这两个函数都需要修改 pCurrent 指针，所以不能把它们声明为 const 成员函数。addBox()函数也会修改列表，也不能声明为 const 成员函数。

类的所有成员函数都需要外部定义，下面看看如何把它们组合在一起。

13.2.3 实现 TruckLoad 类

首先看看构造函数。该构造函数要么从 Box 对象的数组中创建包含一个或多个 Package 对象的列表，要么创建一个空列表：

```
TruckLoad:: TruckLoad(Box* pBox, int count) {
  pHead = pTail = pCurrent = 0;

  if ((count>0)&&(pBox!=0))
     for(int i =0; i<count; i++)
        addBox(pBox+i);
}
```

这段代码把类的数据成员都设置为 0，所以这个列表为空。如果列表中添加了对象，pBox 就非空，count 是正数，这在 if 条件中检查。如果两个条件都是 true，就使用 addBox()函数成员添加 Box 对象。每次调用 addBox()函数，都会创建一个 Package 对象，该对象包含参数指向的 Box 对象。然后把这个 Package 对象添加到列表中。在 for 循环中，把 pBox 指向的 Box 对象添加到列表中，之后添加 pBox+1 指向的 Box 对象，再添加 pBox+2 指向的 Box 对象，依此类推。

提示：

指针递增 1，表示指向下一个对象，而 pBox 包含的地址会递增 sizeof(Box)。

定义成员函数

下面先定义 addBox()函数，因为要在构造函数中调用它：

```
void TruckLoad::addBox(Box* pBox) {
  Package* pPackage = new Package(pBox);    //Create a Package

  if(pHead)                                 //Check list is not empty
    pTail->setNext(pPackage);               //Add the new object to the tail
  else                                      //List is empty
    pHead = pPackage;                       //so new object is the head
  pTail = pPackage;                         //Store its address as tail
}
```

该函数在自由存储区中创建了一个新的 Package 对象，并把它的地址存储在 pPackage 局部变量中。

提示：

这里不能使用运算符 new 分配所需要的内存，但这里先忽略这个问题。第 17 章在讨论异常处理时会探讨这个问题。

在列表中添加对象时，必须考虑列表是否为空。如果 pHead 非空，列表就不为空。此时可以把新对象的地址存储在当前 pTail 的 pNext 成员中，从而把新对象添加到列表的末尾。如果 pHead 是空，就表示列表为空，则只需把新对象的地址存储为列表的第一个成员。在这个例子中，新 Package 对象总是添加到列表的最后，并更新数据成员 pTail，以反映这个添加操作。

getFirstBox()函数的定义如下所示：

```
Box* TruckLoad::getFirstBox() {
  pCurrent = pHead;
  return pCurrent->getBox();
}
```

列表中第一个 Package 对象的地址存储在 pHead 中，所以把这个地址存储在 pCurrent 中。接着调用 Package 对象的 getBox()函数，以获取 Box 对象的地址，并返回。

getNextBox()函数需要通过最近提取的 Package 对象的 getNext ()函数，来访问下一个 Package 对象，而且还需要考虑 pCurrent 是空的情况。定义 getNextBox()函数的代码如下所示：

```
Box* TruckLoad::getNextBox() {
  if(pCurrent)
    pCurrent = pCurrent ->getNext();       // pCurrent is not null so set to next
  else                                     // pCurrent is null
    pCurrent =pHead ;                      // so set to the first list element

  return pCurrent ? pCurrent ->getBox() : 0 ;
}
```

在正常情况下，当 pCurrent 包含 Package 对象的有效地址时，就调用当前 Package 对象的 getNext()函数，获取下一个 Package 对象的地址。如果 pCurrent 为空，就返回列表的开头。这些都是在 if-else 语句中处理的。当然，也可能到达列表的末尾，此时 pCurrent 设置为空；使用一个条件语句返回 0。如果在调用 getBox()函数时，pCurrent 包含空值，程序就会失败(这里还没有考虑从列表中删除元素的问题，但在删除元素时，也可以获得空列表)。返回的空值允许检测何时到达列表的末尾：只需检查一下从 getNextBox()函数返回的指针即可。

前面似乎把所有的内容都组合到一起了，现在可以创建和使用列表了，但不久就会发现，TruckLoad 类还有一个严重的问题。现在先不考虑这个问题，而是在例子中实现该类，以后再设法解决这个问题。

程序示例 13.1—— 使用链表

把 Box 类的完整定义放在头文件 Box.h 中，如下所示：

```
// Box.h - Definition of the Box class
#ifndef BOX_H
```

```
#define BOX_H
class Box {
   public:
      Box(double aLength =  1.0,  double aWidth =  1.0,
                              double aHeight = 1.0);   // Constructor

      double volume()  const;                          // Calculate Box volume

      double getLength()   const;
      double getWidth()  const;
      double getHeight()   const;

         int compareVolume(const Box& otherBox) const;  //Compare volumes of boxes

   private:
      double length;
      double width;
      double height;
 };
#endif
```

再添加访问 Box 对象大小的函数，因为例子需要这些函数。
在另一个.cpp 文件中为这个类定义构造函数和成员函数，这个文件是 Box.cpp：

```
// Box.cpp Implementation of the Box class
#include "Box.h"

// Constructor
Box::Box(double aLength,  double aWidth,  double aHeight) {
   length = aLength > 0.0 ? aLength :  1.0;
   breadth = aWidth > 0.0 ? aWidth  :  1.0;
   height = aHeight > 0.0 ? aHeight  :  1.0;
}

// Calculate Box volume
double Box::volume()  const{ return length*Width*height;  }

// getXXX()  functions
double Box::getLength()   const  { return length;  }
double Box::getWidth()  const  { return Width;  }
double Box::getHeight()   const  ( return height;  }

// Function to compare two Box objects
// If the current Box is greater than the argument,  1 is returned
// If they are equal,  0 is returned
 //  If the current Box is less than the argument -1 is returned
int Box::compareVolume(const Box& otherBox) const {
   double vol1 = volume();                          // Get current Box volume
   double vol2 = otherBox.volume();                 // Get argument volume
   return vol1>vol2 ?  1  :  (vol1<vol2 ?  -1  :  0);
 }
```

把两个实现链表的类的定义放在同一个头文件 List.h 中。为了保持一致，重新安排 Package 类的定义，使构造函数和其他函数在 List.cpp 中定义。该文件的内容如下所示：

```
// List.h classes supporting a linked list
#ifndef LIST_H
#define LIST_H

#include "Box.h"

// Class defining a list element
class Package {
  public:
     Package(Box* pNewBox);                    // Constructor
     Box* getBox()  const;                     // Retrieve the Box pointer
     Package* getNext()  const;                // Get next package address
     void setNext(Package* pPackage);          // Add package to end of list

  private:
     Box* pBox;                               // Pointer to the Box
     Package* pNext;                          // Pointer to the next Package
};

// Class defining a TruckLoad - implements the list
class TruckLoad {
  public:
     TruckLoad(Box* pBox = 0,  int count = 1); // Constructor

     Box* getFirstBox();                       // Retrieve the first Box
     Box* getNextBox();                        // Retrieve the next Box
     void addBox(Box* pBox);                   // Add a new Box to the list

  private:
     Package* pHead;                           // First in the list
     Package* pTail;                           // Last in the list
     Package* pCurrent;                        // Last retrieved from the list
};
#endif
```

注意 Package 类的定义放在 TruckLoad 类定义的前面。如果顺序倒过来了，代码就不会编译，因为 TruckLoad 类要引用 Package 类型。

提示：

有时会有这样的情形：有两个类 A 和 B，每个类定义都包含另一个类的引用。在这种情况下，类 A 必须在定义类 B 之前声明，类 A 的定义可以放在类 B 定义的后面。类 A 的声明可以用下面的语句实现：

```
class A;                    // The name A is a class
```

这就告诉编译器，A 是一个类，于是编译器可以向前移动，去编译类 B，接着编译器知道 B 也是一个类，所以也可以编译类 A 的定义。

成员函数定义可以放在文件 List.cpp 中：

```
// List.cpp
#include "Box.h"
#include "List.h"

// Package class definitions
// Package constructor
Package::Package(Box* pNewBox):pBox(pNewBox), pNext(0){}

// Retrieve the Box pointer
Box* Package::getBox()  const  { return pBox;  }

// Get next package address
Package* Package::getNext()  const  { return pNext;  }

// Add package to end of list
void Package::setNext(Package* pPackage)  { pNext = pPackage;  }

// TruckLoad class member definitions
// TruckLoad constructor
TruckLoad::TruckLoad(Box* pBox,  int count) {
   pHead = pTail = pCurrent = 0;

   if((count > 0) && (pBox != 0))
      for(int i = 0 ; i<count ; i++)
         addBox(pBox+i);
}

// Retrieve the first Box
Box* TruckLoad::getFirstBox(){
  pCurrent = pHead;
  return pCurrent->getBox();
}

// Retrieve the next Box
Box* TruckLoad::getNextBox(){
   if(pCurrent)
      pCurrent = pCurrent->getNext();   // pCurrent is not null so set to next
   else                                 // pCurrent is null
      pCurrent = pHead;                 // so set to the first list element

  return pCurrent ? pCurrent->getBox()  : 0;
}

// Add a new Box to the list
void TruckLoad::addBox(Box* pBox) {
   Package* pPackage = new Package(pBox);  // Create a Package

   if(pHead)                               // Check list is not empty
      pTail->setNext(pPackage);            // Add the new object to the tail
   else                                    // List is empty
```

```
        pHead = DPackage;                    // so new object is the head
    pTail = pPackage;                        // Store its address as tail
}
```

注释：

在类定义中声明为 const 的成员函数，在定义它们时必须也声明为 const。函数签名包含 const，这表示，如果在函数定义中忘记指定该函数是 const，编译器就会认为它们与类定义中的函数不同。

注意文件 List.h 包含一个#include 指令，以包含 Box 类的定义。实际上，有许多头文件的 #include 指令，特别是文件 List.cpp 包含 Box.h 和 List.h 的#include 指令。每个头文件中的 #ifndef/#endif 预处理器指令(详见第 10 章)可以防止源文件中的定义出现多次。没有这些预处理器指令，文件 List.cpp 就不会编译，因为它会包含两次 Box.h 的定义：一次是直接包含，另一次是通过包含 List.h 而间接包含。

在文件 prog13_01.cpp 中定义 main()，创建一些 Box 对象，并把它们组织到一个链表中。然后生成两个列表：一个列表只包含一个 Box 对象，而且可以添加更多的对象，而另一个列表从 Box 对象数组中生成。使用第 10 章编写的 random()函数可以生成 Box 对象的尺寸。为了试用这些列表，下面搜索列表中最大的 Box 对象。代码如下：

```
// Program 13.1 Exercising a linked list of Box objects   File: prog13_01.cpp
#include <iostream>
#include <cstdlib>                          // For random number generator
#include <ctime>                            // For time function

using std::cout;
using std::endl;

#include "Box.h"
#include "List.h"

// Function to generate a random integer 1 to count
inline int random(int count) {
   return 1 + static_cast<int>
            (count*static_cast<double>(std::rand())/ (RAND_MAX+1.0));
}

int main() {
const int dimLimit = 100;                    // Upper limit on Box dimensions
std:: srand((unsigned) std::time(0));        // Initialize the random number
                                             // generator

// Create an empty list
TruckLoad load1;

// Add 10 random sized Box objects to the list
for(int i = 0  ;  i  < 10  ;  i++)
   load1.addBox(new Box(random(dimLimit),  random(dimLimit),  random
                                                        (dimLimit)));
```

```
// Find the largest Box in the list
Box* pBox = loadl.getFirstBox();
Box* pNextBox;
while(pNextBox = loadl.getNextBox())  // Assign and then test pointer to next Box
    if(pBox->compareVolume(*pNextBox)  < 0)
        pBox  = pNextBox;

cout << endl
     << "The largest box in the first list is  "
     << pBox->getLength()  << "by"
     << pBox->getWidth() << "by"
     << pBox->getHeight()  << endl;

 const int boxCount = 20;              // Number of elements in Box array
 Box boxes[boxCount];                  // Array of Box objects

 for(int i = 0  ;  i < boxCount  ;  i++)
   boxes[i]  = Box(random(dimLimit),  random(dimLimit),  random (dimLimit));

TruckLoad load2(boxes, boxCount);

 // Find the largest Box in the list
pBox = load2.getFirstBox();
while(pNextBox = load2.getNextBox())
    if(pBox->compareVolume(*pNextBox)  < 0)
        pBox = pNextBox;

 cout << endl
      << "The largest box in the second list is  "
      << pBox->getLength()   << "  by  "
      << pBox->getWidth()  << "by"
      << pBox->getHeight()   << endl;

// Delete the Box objects in the first list
   pNextBox = loadl.getFirstBox();
while(pNextBox) {
 delete pNextBox;
 pNextBox = loadl.getNextBox();
}
 return 0;
}
```

运行这个程序，结果如下所示：

```
The largest box in the first list is 94 by 68 by 55

The largest box in the second list is 80 by 73 by 78
```

读者得到的结果可能与此不同，因为随机生成的 **Box** 对象的尺寸可能不同。随机数生成器是根据计算机时钟来生成随机数的。

例子的说明

函数 main()创建了两个 Box 对象的列表。在任何一个 Box 对象中，都生成了随机尺寸的盒子，其尺寸是从 1 到 100 的随机整数。随机尺寸由下面的函数生成：

```
inline int random(int count) {
  return 1+static_cast<int>
          (count* static_cast<double>(std::rand())/(RAND_MAX+1.0));
}
```

这段代码调用标准库函数 rand()，该函数(详见第 10 章)生成从 0 到 RAND_MAX 之间的随机整数。这个过程首先用一个种子值调用标准库函数 srand()来初始化，该种子值常常指定为当前的时间，因为每次执行程序时，都会产生一个不同的种子。函数 rand()的返回值再乘以 0 到 count - 1 之间的数，接着加 1，返回 1 到 count 之间的数。把 rand()的返回值强制转换为 double，是为了确保对 double 值进行乘法操作。否则得到的乘积可能超出了 int 的范围。

使用 TruckLoad 的默认构造函数创建一个空列表：

```
TruckLoad load1;
```

接着，使用一个 for 循环在列表 load1 中添加 10 个 Box 对象，每个对象的尺寸都是随机的：

```
for(int i=0; i<10; i++)
  load1.addBox(new Box(random(dimLimit), random(dimLimit), random (dimLimit)));
```

在每个循环迭代中，都会在自由存储区中创建一个 Box 对象，并使用 addBox()函数添加到列表中。不需要跟踪这些对象，因为它们的地址都存储在列表中。

然后，扫描列表，找出体积最大的 Box 对象：

```
Box* pBox=load1.getFirstBox();
Box* pNextBox;
while (pNextBox = load1.getNextBox())  //Assign and then test pointer to next Box
  if(pBox->compareVolume(*pNextBox)<0)
    pBox= pNextBox;
```

在这部分程序中，指针 pBox 用于存储最大对象的地址。首先，pBox 设置为列表中的第一个对象。再用 load1 的 getNextBox()函数返回的地址控制 while 循环。注意 while 循环条件包含赋值运算符(=)，而不是比较运算符(==)，因此，在进行赋值后，在 while 循环中测试的表达式是 pNextBox。代码中的注释说明了这一点。否则，其他程序员很容易误解代码。编译器也可能发出一个警告消息，因为以这种方式使用赋值语句常常会导致一个错误。在 pNextBox 的值为 0 时，就表示已到达列表的末尾，循环结束。

在循环中，使用 Box 类中的 compareVolume()函数比较 pBox 所指向的 Box 对象的体积和列表中当前位置的 Box 对象的体积(pNextBox 指向该 Box 对象)。必须使用->运算符来调用 compareVolume()函数，因为要通过一个指针 pBox 来调用它。该函数需要把一个 Box 类型的对象作为参数，所以必须解除 pNextBox 指针的引用。如果返回值是负的，参数就大于 pBox 所指向的 Box 对象，于是把参数的地址存储在 pBox 中，作为新的最大对象。在循环结束时，pBox 就包含列表中体积最大的 Box 对象的地址。

接着显示这个 Box 对象的尺寸，如下所示：

```
cout << endl
     << "The largest box in the first list is"
     << pBox->getLength() << "by"
     << pBox->getWidth() << "by"
     << pBox->getHeight() << endl;
```

对第二个列表 load2 重复这个过程，load2 是从 Box 对象的数组中创建的。

注意第一个列表中的 Box 对象是在自由存储区中创建的，而数组中的 Box 对象是本地对象。因此，完成后，需要删除第一个列表中的 Box 对象。

所有的程序都运行正常，所得的答案也是正确的，那么 TruckLoad 类还有什么问题呢？实际上，它有 3 个问题要解决：

- 最不重要的问题是 Package 类可以由任何对象访问，即使我们只想在 TruckLoad 类中使用该类。
- 比较严重的问题是(尽管这在前面的例子中不明显)TruckLoad 对象的复制。假定有一个 TruckLoad 对象 load1，再创建它的一个直接副本 load2，其结果是 load2 与 load1 不是彼此独立的，因为两个列表不仅包含指向相同 Box 对象的指针，还包含指向相同 Package 对象的指针。如果删除 load2，属于 load1 列表的对象就会被删除，因为它们是作为 load2 的一部分被删除的。
- 第三个问题是内存管理。看看 TruckLoad 类中的 Package 对象是如何创建的。它们是在自由存储区中创建的，而且没有被删除。每次在列表中添加新 Box 对象时，都会创建一个永远也不会删除的新 Package 对象。即使 TruckLoad 对象超出了作用域，Package 对象也不会删除。这是一个非常严重的内存漏洞，不应忽视。

下面看看如何解决这些问题。首先看看第一个问题。

13.3 控制对类的访问

在实践中，常常要求限制类的可访问性。前面设计的 Package 类是专用于 TruckLoad 类的，因此应确保 Package 对象只能由 TruckLoad 类中的函数创建。这里需要一个机制，把 Package 对象声明为对 TruckLoad 类来说是公共对象，而对其他对象来说是私有对象。为此，最好的方法是使用嵌套类的概念。

嵌套类

嵌套类是把自己的定义放在另一个类的定义中。嵌套类的名称限定为包含类的作用域。下面把 Package 类的定义放在 TruckLoad 类的定义中：

```
class TruckLoad {
  public:
    TruckLoad(Box* pBox = 0, int count = 1);   //Constructor

    Box* getFristBox();                        //Retrieve the first Box
    Box* getNextBox();                         //Retrieve the next Box
    void addBox(Box* pBox);                    //Add a new Box to the list
```

```
private:
  //Class defining a list element
  class Package {
    public:
      Box* pBox;                              //Pointer to the Box
      Package* pNext;                         //Pointer to the next Package

      void setNext(Package* pPackage);        //Add package to end of list
      Package(Box* pNewBox);                  //Constructor
    }

    Package* pHead;                          //First in the list
    Package* pTail;                           //Last in the list
    Package* pCurrent;                        //Last retrieved from the list
};
```

Package类型现在在TruckLoad类定义的范围之内。由于把Package类的定义放在TruckLoad类的私有部分，因此不能在 TruckLoad 类的外部创建 Package 对象。

由于 Package 类是 TruckLoad 类的私有部分，因此可以把 Package 类的成员声明为 public。这样，TruckLoad 对象的函数成员就可以直接访问它们了。于是可以用 Package 的成员函数 getBox()和 getNext()来分发货物。所有的 Package 成员都不能在 TruckLoad 类的外部访问。

还需要修改 TruckLoad 类的成员函数的定义。addBox()函数可以通过直接访问最后一个对象的 pNext 成员，把新对象添加到列表的末尾：

```
void TruckLoad::addBox(Box* pBox) {
  Package* pPackage = new Package(pBox);   //Create a Package

  if(pHead)                                //Check list is not empty
    pTail->pNext=pPackage;                 //Add the new object to the tail
  else                                     //List is empty
    pHead = pPackage;                      //so new object is the head
  pTail = pPackage;                        //Store its address as tail
}
```

提取第一个 Box 对象的函数不再需要调用另一个函数，来获取 Box 对象的指针，于是其定义变成：

```
Box* TruckLoad::getFirstBox() {
  pCurrent = pHead;
  return pCurrent->pBox;
}
```

获取列表中下一个 Box 对象的地址的函数现在可以定义为：

```
Box* TruckLoad::getNextBox() {
  if (pCurrent)
    pCurrent = pCurrent ->pNext;           //pCurrent is not null so set to next
  else                                     //pCurrent is null
    pCurrent =pHead ;                      //so set to the first list element
```

```
    return pCurrent ? pCurrent ->pBox : 0 ;
}
```

注意：

在 TruckLoad 类中嵌套 Package 类只是定义了 Package 类型。TruckLoad 类型的对象并没有受到影响。它们的成员仍与以前一样。

嵌套类的函数成员可以直接引用包含类的静态成员，以及其他在包含类中定义的类型或枚举成员。包含类的其他成员只能在嵌套类中以正常的方式访问：通过类对象、类对象的指针或引用。

1. 用 public 访问指定符修饰的嵌套类

当然，还可以把 Package 类定义放在 TruckLoad 类的公共部分。也就是说，Package 类定义是公共接口的一部分，因此可以在外部创建 Package 对象。由于 Package 类名在 TruckLoad 类的作用域中，所以不能使用它本身。而必须用包含类的名称限定 Package 类名，例如：

```
TruckLoad:: Package aPackage(aBox);          //Define a variable of type Package
```

当然，在这种情况下，把 Package 类型声明为 public，要比把它设计为嵌套类更好。在其他情况下，也需要把嵌套类声明为 public。

2. 友元类

还可以使用友元类来限制类的可访问性。在这种情况下，要把 Package 类的所有数据成员都声明为 private，并把该类声明为 TruckLoad 类的友元，方法在 Package 类的定义中包含下面的语句：

```
friend class TruckLoad;
```

这可以确保 Package 对象的所有成员(即使是私有成员)都可以由 TruckLoad 类使用，Package 类的私有成员不能在类的外部访问。

注释：

在本章后面的例子中，将采用嵌套类的方式。这主要是为了说明 Package 类不是一个独立的实体，仅在 TruckLoad 对象的上下文中承担一定的任务。

13.4 副本构造函数的重要性

副本构造函数用于创建一个与已有对象完全相同的新对象。程序示例 12.5 提到，如果类定义中没有副本构造函数，编译器会在需要时提供默认的副本构造函数。默认的副本构造函数会把原对象的数据成员的值复制到新建对象的对应数据成员中。

在一些情况下，创建副本对象只是复制数据成员的值会出问题，对于 TruckLoad 类来说，则肯定会出问题。假定要复制一个 TruckLoad 对象，例如在两个不同的日期重复相同的传送过程。在复制 TruckLoad 类的对象时，默认的副本构造函数会把存储在数据成员 pHead、pTail 和 pCurrent 中的地址复制到新对象中，这样两个 TruckLoad 对象就共享同一个 Package 对象链。如图 13-6 所示。

这里没有两个列表，只有一个列表。但是可以采用两种不同的方式来访问列表，这就是危险的根源。使用 tLoad1 来修改列表会影响 Package 对象链，但这种修改不会反映到 tLoad2 的数据成员上。尤其是，如果给列表 tLoad1 添加了对象，则 tLoad2 的 pTail 成员就会指向列表中不再是最后一个的 Package 对象。

实际上，还有一个更大的隐患。如果创建一个以 TruckLoad 为参数的函数，就会遇到类似的问题。在函数通过按值传送机制接收参数时，就会调用副本构造函数。因此，函数调用就会使用 TruckLoad 副本构造函数来创建原 TruckLoad 对象的一个副本，而函数仅需要把对象添加到列表中，使原来的 TruckLoad 对象失效。

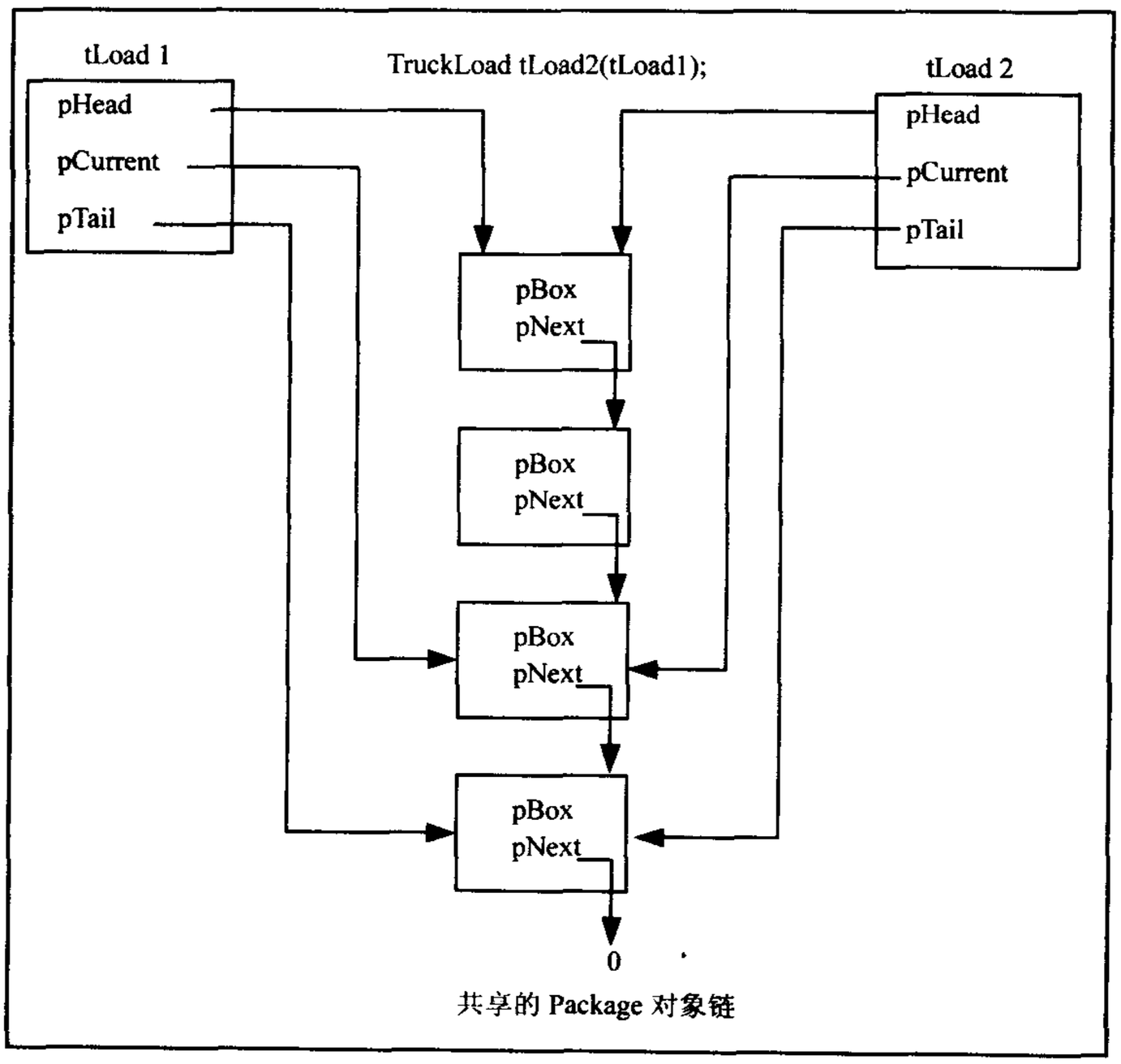


图 13-6　使用默认的副本构造函数复制对象的副本

实现副本构造函数

对象的副本构造函数必须接受同一类类型的参数，以适当的方式创建副本。现在有一个问题必须解决，如果为 TruckLoad 类编写副本构造函数，就可以更清楚地看出这个问题：

```
TruckLoad:: TruckLoad(TruckLoad Load) {
  //Code to create a duplicate of the object Load
}
```

这段代码初看上去很不错，但考虑一下调用构造函数时会发生什么？参数是按值传送的，所以编译器要调用副本构造函数，制作参数的副本。当然，副本构造函数的参数是按值传送的，于是需要再次调用副本构造函数，这样不断重复下去。简言之，这是一个副本构造函数的递归调用。

事实上，编译器不允许编译代码。解决方案是把参数指定为引用。

引用参数

副本构造函数应使用一个 const 引用参数来定义，如下所示：

```
TruckLoad:: TruckLoad(const TruckLoad& Load) {
  //Code to create a duplicate of the object Load
}
```

现在参数不再按值传送，也就避免了副本构造函数的递归调用。编译器会用按引用传送来的对象初始化参数。参数应是 const，因为副本构造函数仅创建副本，它不应修改原对象。因此，副本构造函数的参数类型总是对同一个类对象的 const 引用。换言之，副本构造函数的形式对任何类都是一样的：

```
Type:: Type(const Type& object) {
  //Code to produc a duplicate of object
}
```

在副本构造函数的一般形式中，Type 是类的类型名。

那么，TruckLoad 类的副本构造函数如何实现？必须为要创建的新 TruckLoad 对象创建副本 Package 对象链。对嵌套了 Package 类的 TruckLoad 类来说，代码如下所示：

```
// TruckLoad copy construcor
TruckLoad:: TruckLoad(const TruckLoad& load) {
  pHead = pTail = pCurrent = 0;
  if(load.pHead==0)
    return;

  Package* pTemp=load.pHead;          //Saves addresses for new chain
  do {
    addBox(pTemp->pBox);
  }while(pTemp=pTemp->pNext);         //Assign and then test pointer to next Box

}
```

如果 load 对象的 pHead 是空，列表就是空的，因此返回。如果 pHead 不是空，就执行循环。循环遍历存储在 load 对象中的原列表中的 Package 对象列表。Ptemp 指针最初指向该列表中的第一个对象，循环条件给列表中的下一个对象地址赋值，并测试它是否为空。循环调用 addBox()，给它传送指向原 Package 对象的 Box 对象地址，创建一个对应于原列表中每个 Package 对象的新 Package 对象。使用 pTemp 获取 Box 对象的地址，访问原 Package 对象中的 pBox 指针。addBox()函数会用链中新对象的地址自动更新新列表中前一个 Package 对象的 pNext 指针。

下面看看这个版本是否工作。

程序示例 13.2—— 使用副本构造函数

Box.h 和 Box.cpp 文件与程序示例 13.1 中的相同。这里使用前面讨论的 TruckLoad/Package 嵌套类，添加副本构造函数，此时，List.h 文件应包含下面的类定义：

```
class TruckLoad {
  public:
    TruckLoad(Box* pBox = 0, int count = 1);      //Constructor
```

```
    TruckLoad(const TruckLoad& load);              //Copy constructor

    Box* getFristBox();                            //Retrieve the first Box
    Box* getNextBox();                             //Retrieve the next Box
    void addBox(Box* pBox);                        //Add a new Box to the list

  private:
    //Class defining a list element
    class Package {
  public:
     Box* pBox;                                    //Pointer to the Box
     Package* pNext;                               //Pointer to the next Package

     void setNext(Package* pPackage);              //Add package to end of list
     Package(Box* pNewBox);                        //Constructor
  };

    Package* pHead;                                //First in the list
    Package* pTail;                                //Last in the list
    Package* pCurrent;                             //Last retrieved from the list
  };
```

TruckLoad 副本构造函数的原型添加为类的一个公共成员。文件 List.cpp 必须更新，以包含它的定义。还必须更新 TruckLoad 成员函数 addBox()、getFirstBox()和 getNextBox()的定义，删除 Package 成员函数 getBox()和 getNext()的定义。文件 List.cpp 中最后要修改的是更新其他 Package 成员定义的限定符。所以该文件的内容应如下所示：

```
// List.cpp
#include "Box.h"
#include "List.h"

// Package class definitions
// Package constructor
TruckLoad::Package::Package(Box* pNewBox):pBox(pNewBox), pNext(0){}

// Add package to end of list
void TruckLoad::Package::setNext(Package* pPackage)  { pNext = pPackage;  }

// TruckLoad class member definitions
// TruckLoad constructor
TruckLoad::TruckLoad(Box* pBox,  int count) {
    pHead = pTail = pCurrent = 0;

    if((count > 0) && (pBox != 0))
        for(int i = 0 ; i<count ; i++)
          addBox(pBox+i);
}

// TruckLoad copy construcor
TruckLoad:: TruckLoad(const TruckLoad& load) {
pHead = pTail = pCurrent = 0;
if(load.pHead==0)
```

```
  return;

Package* pTemp=load.pHead;              //Saves addresses for new chain
do {
  addBox(pTemp->pBox);
}while(pTemp=pTemp->pNext);             //Assign and then test pointer to next Box

}

// Retrieve the first Box
Box* TruckLoad::getFirstBox(){
   pCurrent = pHead;
   return pCurrent->getBox();
}

// Retrieve the next Box
Box* TruckLoad::getNextBox(){
    if(pCurrent)
       pCurrent = pCurrent->getNext();    // pCurrent is not null so set to next
    else                                  // pCurrent is null
       pCurrent = pHead;                  //  so set to the first list element

   return pCurrent ? pCurrent->getBox()  : 0;
}

// Add a new Box to the list
void TruckLoad::addBox(Box* pBox) {
    Package* pPackage = new Package(pBox);    // Create a Package

    if(pHead)                                  // Check list is not empty
       pTail->setNext(pPackage);               // Add the new object to the tail
    else                                       // List is empty
       pHead = DPackage;                       // so new object is the head
    pTail = pPackage;                          // Store its address as tail
}
```

这次创建的 TruckLoad 对象只包含 3 个 Box 对象。可以使用副本构造函数来复制已有的 TruckLoad 对象，再扩展它。由于要找出几个列表中体积最大的 Box 对象，下面把完成这一任务的代码放在一个单独的函数中。代码如下：

```
// Program 13.2 Exercising the copy constructor   File: prog13_02.cpp
#include <iostream>
#include <cstdlib>                    // For random number generator
#include <ctime>                      // For time function

using std::cout;
using std::endl;

#include "Box.h"
#include "List.h"

// Function to generate a random integer 1 to count
```

```
inline int random(int count) {
   return 1 + static_cast<int>
                  (count*static_cast<double>(std::rand())/(RAND_MAX+1.0));
}

// Find the Box in the list with the largest volume
Box* maxBox(TruckLoad& Load) {
    Box* pBox = Load.getFirstBox();
    Box* pNextBox;
    while(pNextBox = Load.getNextBox())  // Assign and then test pointer to next Box
        if(pBox->compareVolume(*pNextBox)  < 0)
            pBox = pNextBox;
    return pBox;
}

int main() {
    const int dimLimit = 100;                    // Upper limit on Box dimensions
    std::srand((unsigned)std::time(0));          // Initialize the random number
                                                 // generator

    // Create a list
    TruckLoad load1;

    // Add 3 Boxes to the list
    for(int i = 0  ;  i < 3  ;  i++)
      load1.addBox(new Box(random(dimLimit),  random(dimLimit),  random
                                                        (dimLimit)));

Box* pBox = maxBox(load1);          // Find the largest Box in the first list

 cout << endl
      << "The largest box in the first list is  "
      << pBox->getLength()   <<  "by"
      << pBox->getWidth()  <<  "by"
      << pBox->getHeight()   << endl;

 TruckLoad load2(load1);             // Create a copy of the first list

 pBox = maxBox(load2);               // Find the largest Box in the second list

 cout << endl                        // Display it
           << "The largest box in the second list is"
           << pBox->getLength()   << "by"
           << pBox->getWidth()  << "by"
           << pBox->getHeight()   << endl;

// Add 5 more boxes to the second list
for(int i = 0;  i<5;  i++)
   load2.addBox(new Box(random(dimLimit),  random(dimLimit),  random
                                                        (dimLimit)));
pBox = maxBox(load2);          // Find the largest Box in the extended list
cout  << endl                  //  Display it
```

```
        << "The largest box in the extended second list is"
        << pBox->getLength()    <<  "  by"
        << pBox->getWidth()     <<  "by"
        << pBox->getHeight()    <<  endl;

// Count the number of boxes in the first list and display the count
Box*  pNextBox =  load1.getFirstBox();
int count = 0;                                  // Box count
while(pNextBox) {                               // While there is a box
   count++;                                     // Increment  the  count
   pNextBox = load1.getNextBox();               // and get the next box
 }
 cout << endl <<  "First list still contains  "  << count <<  "Box objects."<< endl;

 // Delete the Box objects in the free store
 pNextBox = load2.getFirstBox();
 while(pNextBox) {
   delete pNextBox;
   pNextBox = load2.getNextBox();
 }
 return  0;
}
```

运行这个程序，结果如下所示：

```
The largest box in the first list is 44 by 92 by 87

The largest box in the second list is 44 by 92 by 87

The largest box in the extended second list is 55 by 83 by 85

First list still contains 3 Box objects.
```

读者得到的结果肯定与此不同，而且可能需要运行好几次才能在扩展后的列表中得到与第一个列表中不同的盒子。

例子的说明

新函数 maxBox()把 TruckLoad 对象作为它的参数。它返回一个指针，指向这个 TruckLoad 对象中体积最大的 Box 对象。这个函数的代码来自于程序示例 13.1 中的 main()函数体。注意参数是一个引用，所以不必复制列表。

为了说明 TruckLoad 类副本构造函数的工作情况，本例首先创建一个空的 TruckLoad 对象 load1：

```
TruckLoad load1;
```

接着与程序示例 13.1 一样，把 Box 对象添加到列表中。每个 Box 对象都是在自由存储区中创建的：

```
for(int i=0; i<3; i++)
  load1.addBox(new Box(random(dimLimit), random(dimLimit), random
                                                    (dimLimit)));
```

为了便于检查，应调用函数 maxBox()，以获取列表中体积最大的 Box 对象：

```
Box* pBox= maxBox(load1);              //Find the largest Box in the first list
```

在显示这个 Box 对象的尺寸后，使用 TruckLoad 类的副本构造函数创建 load1 的副本：

```
TruckLoad load2(load1);                //Create a copy of the first list
```

为了说明 load2 与原列表相同，再次使用 maxBox()获取 load2 列表中体积最大的 Box 对象，并显示它的尺寸。然后在 load2 列表中再添加 5 个 Box 对象：

```
for(int i=0; i<5; i++)
  load2.addBox(new Box(random(dimLimit), random(dimLimit), random
                                                        (dimLimit)));
```

之后，再次对 load2 使用 maxBox()函数，找出列表中 8 个盒子中的最大 Box 对象，并显示它的尺寸。

```
pBox = maxBox(load2);          // Find the largest Box in the extended list
cout  <<  endl                 //  Display it
      << "The largest box in the extended second list is"
      << pBox->getLength()  << "  by"
      << pBox->getWidth() << "by"
      << pBox->getHeight() << endl;
```

添加 load2 后，load1 并没有修改，为了说明这一点，下面计算 load1 中的 Box 对象的个数：

```
Box* pNextBox= load1.getFirstBox();
int count=0;                                    //Box count
while(pNextBox) {                               //While there is a box
  count++;                                      //Increment the count
  pNextBox= load1.getNextBox();                 //and get the next box
}
```

循环由指针 pNextBox 控制。首先，该指针包含第一个 Box 对象的地址，只要 pNext 不为空，就递增 count，并在列表中移动 pNext。当到达列表末尾时，getNextBox()就返回空，循环停止。

最后，显示 count 的值，之后从自由存储区中删除 load2 中的所有 Box 对象。当然，这包括 load1 中的对象，所以不需要明确删除它们。

TruckLoad 类假定用户负责 Box 对象的存储。这些对象可以在自由存储区中创建，如本例所示，也可以是自动对象—— 此时，TruckLoad 类也能正常工作。让用户负责 Box 对象意味着，用户可以从自由存储区中删除 Box 对象，从而使 TruckLoad 对象失效。实现这一操作的一种方法是复制 TruckLoad 对象，再删除它。可以阻止用户这么做，方法非常简单：把 TruckLoad 副本构造函数声明为类的私有成员，就可以阻止在类的外部使用它，并阻止编译器生成默认的副本构造函数。由于在这种情况下可能不会使用它，所以不需要提供它的定义。

13.5 对象内部的动态内存分配

下面讨论前面提到的 TruckLoad 类的第三个问题。这是另一个严重的问题：内存泄漏，它产生的原因是 TruckLoad 类构造函数(包含副本构造函数)和 addBox()函数动态分配了内存。

动态创建的成员在自由存储区中分配内存。代码分配了内存，所以应负责释放它。我们总是在对象地址失去踪迹之前，释放对象在自由存储区中的内存。如果不知道对象位于自由存储区的什么地方，就无法释放其内存。如果在释放自由存储区的内存时总是失败(例如在循环中)，最终自由存储区就会被填满，程序就会失败。

提示：

当然，在程序执行完毕后，操作系统总是会重新分配内存，但这对程序的执行没有什么帮助。

这与动态创建新对象是不同的。在自由存储区中创建对象时，new 运算符会为它分配内存空间，并调用构造函数创建该对象。不是构造函数分配了内存空间，而是调用构造函数的代码分配了内存空间。当不再需要对象时，可以通过前面的方式使用 delete 运算符释放它占用的内存，因为创建对象的代码仍保有该对象的地址。前面是在自由存储区中为 TruckLoad 对象内部的对象分配内存空间。类的构造函数和 addBox()函数把自由存储区中的内存分配给对象，该内存的地址存储在对象的成员中。释放该内存的惟一方式是明确使用 delete 运算符和这些地址。为此，必须在释放 TruckLoad 对象时，执行释放内存所需的代码。

13.5.1 析构函数

事实上，还有一个类成员可以释放内存，即析构函数。编译器总是会提供一个默认的析构函数，析构函数总是在删除 TruckLoad 对象时调用，我们从来不需要明确调用析构函数，但常常需要定义一个析构函数。

默认析构函数不会释放在类对象中分配的自由存储区的内存，因为这是程序员的工作。实际上，默认析构函数什么也不做。但是，总是可以用自己的析构函数代替默认析构函数。

13.5.2 定义析构函数

类的析构函数总是有相同的形式。它是类的一个公共成员函数，与类同名，但名称前面有一个符号～。类的析构函数没有参数，也没有返回值。类可以有好几个构造函数，但只有一个析构函数。对于 TruckLoad 类，类的析构函数定义如下：

```
TruckLoad:: ~TruckLoad () {
  //Code to destroy the object
}
```

注意：

为析构函数指定返回类型或参数是错误的。

13.5.3 默认的析构函数

与默认的构造函数一样，如果类没有显式提供析构函数，编译器就会生成默认的析构函数。如果没有为类定义析构函数，编译器就会提供一个公共或内联析构函数。

在从自由存储区中显式删除目前使用的每个对象时，或该对象超出了作用域(此时它是自动变量)时，都会调用该对象的默认析构函数。在前面的例子中，为每个类添加一个明确的析构函数，如下所示。

程序示例 13.3—— 析构函数

修改程序示例 13.2，为每个类添加析构函数的定义。之后需要修改 Box 类定义，在文件 Box.cpp 中添加下面的析构函数定义：

```
//Box destructor
Box::~Box() {
  cout<<"Box destructor called. "<<endl;
}
```

不要忘了给<iostream>添加#include 指令，给 cout 和 endl 添加 using 声明。

在 Box.h 中，还需要在 Box 类定义的公共部分添加析构函数原型：

```
class Box {
  public:
     ~Box();       //Box destructor
     //Rest of the class as before…
};
```

以类似的方式，在 List.cpp 中为 Package 和 TruckLoad 类添加析构函数定义：

```
//TruckLoad destructor
TruckLoad::~TruckLoad () {
  Cout << " TruckLoad destructor called. " <<endl;
}

//Package destructor
TruckLoad::Package::~Package (){
  cout << " Package destructor called. " <<endl;
}
```

再在 List.h 头文件中为类定义添加原型：

```
class TruckLoad  {
  public:
     ~TruckLoad ();              // TruckLoad destructor

    //other TruckLoad public member declarations

  private:
    class Package {
```

```
    Public:
        ~Package ();                    //Destructor
        //Rest of Package class definition…
    };

    //Rest of TruckLoad class definition…
};
```

对于每个.cpp 文件，还需要包含标准头文件 iostream，并为 cout 和 endl 添加 using 声明。

程序示例 13.4—— 跟踪析构函数

这里使用的 main()函数与程序示例 13.2 完全相同：

```
//Program 13.3 Exercising the destructor                File: prog13_03.cpp
//Code as Program 13.2…
```

如果编译并运行这个例子，就会得到程序执行过程中所有的析构函数调用记录。

例子的说明

下面从输出中了解程序的执行过程。有一些内容很令人惊讶。输出的第一部分如下所示：

```
The largest box in the first list is 52 by 83 by 93
The largest box in the second list is 52 by 83 by 93
The largest box in the extended second list is 52 by 83 by 93
First list still contains 3 Box objects.
```

这没有什么可惊讶的。接着，记录了 8 个 Box 析构函数调用，其后是两个 TruckLoad 析构函数调用：

```
Box destructor called.
Box destructor called.
Box destructor called.
Box destructor called.
Box destructor called.
Box destructor called.
Box destructor called.
Box destructor called.
Box destructor called.
TruckLoad destructor called.
TruckLoad destructor called.
```

调用了 8 次 Box 析构函数，是因为我们显式删除了在自由存储区中为第一个列表 load1 创建的 Box 对象：

```
pNextBox=load2.getFirstBox();
while(pNextBox) {
  delete pNextBox;
  pNextBox=load2.getNextBox();
}
```

没有这些代码，内存在程序终止之前是不会释放的。在 main()函数中执行 return 时，两个 TruckLoad 对象超出了作用域，于是调用 TruckLoad 类析构函数。

注意这里没有 Package 析构函数调用，也就是说，这些 Package 对象在创建后，并没有删除。所有 Package 对象的内存都是动态分配的，因此用户应负责删除它们。而用户没有删除，也就不会调用它们的析构函数。这是一个严重的错误，下面就更正它。

13.5.4 实现析构函数

删除 Package 对象的责任应由创建它们的对象负责。由于 Package 对象是由 TruckLoad 对象创建的，因此要解决这个问题，应为 TruckLoad 类实现一个适当的析构函数。TruckLoad 对象需要在删除时，同时删除其列表中的每个 Package 对象。这可以在 TruckLoad 类析构函数中实现。用下面的代码替换 List.cpp 文件中的析构函数定义：

```
TruckLoad::~TruckLoad() {
  cout<<" TruckLoad destructor called. "<<endl;
  while(pCurrent=pHead->pNext) {
    delete pHead;                    //Delete the previous
    pHead= pCurrent;                 //Store address of next
  }
  delete pHead;                      //Delete the last
}
```

用 TruckLoad 析构函数的这个版本运行例子，就会得到如下结果：

```
The largest box in the first list is 98 by 79 by 78

The largest box in the second list is 93 by 68 by 99

The largest box in the extended second list is 89 by 98 by 78

First list still contains 3 Box objects.
Box destructor called.
Box destructor called.
Box destructor called.
Box destructor called.
Box destructor called.
Box destructor called.
Box destructor called.
Box destructor called.
TruckLoad destructor called.
Package destructor called.
Package destructor called.
Package destructor called.
Package destructor called.
Package destructor called.
Package destructor called.
Package destructor called.
Package destructor called.
TruckLoad destructor called.
Package destructor called.
Package destructor called.
Package destructor called.
```

在每个 TruckLoad 析构函数调用之后，都调用 Package 析构函数若干次，每次都会删除已创建的一个 Package 对象。这完全是由刚才添加到 TruckLoad 类的析构函数的代码完成的。在第一个列表中有 3 个对象，在第二个列表中有 8 个对象，所以，从输出中可以清楚地看到，TruckLoad 对象按照与创建顺序相反的顺序删除。

由此可得到一个黄金规则：

如果在对象中动态分配了内存，就必须实现该类的析构函数。

13.6 类的引用

前面介绍了把引用用作函数的参数类型是编写副本构造函数所必须的。这是编写副本构造函数的惟一方式。在许多其他情况下，使用引用参数也有很大的优点，因为它避免了在按值传送机制中的隐式复制过程。第 14 章将讨论从成员函数中返回一个引用也是很重要的。

还可以把引用用作类的数据成员。这不是很常见，但需要详细讨论一下，因为在这种情况下，引用需要仔细的考虑。

引用用作类的成员

为了说明如何把数据成员声明为引用，可以把 Box 对象的引用(而不是 Box 的指针)用作 Package 类的一个数据成员。在 TruckLoad 类中，修改后的 Package 类的基本定义如下所示：

```
class Package {
      public:
         Box& rBox;                                  //Reference to the Box
         Package* pNext;                             //Pointer to the next Package

          ~Package();                                //Destructor

         void setNext(Package* pPackage);            //Add package to end of list
         Package(Box& rNewBox);                      //Constructor
};
```

在 List.cpp 中，构造函数的定义应改为：

```
TruckLoad::Package::Package(Box& rNewBox):rBox(rNewBox),pNext(0){}
```

应总是用初始化列表中的数据初始化最初的数据成员，因为这比较有效，但对于引用数据成员，就没有其他选择。引用是另一个名称的别名，不能在构造函数中用赋值语句来初始化它，而必须使用初始化列表。

下面看看存储 Box 对象的引用会如何影响 TruckLoad 类的函数成员。首先看看 addBox() 成员。把参数改为引用，再把该引用传送给 Package 类的构造函数：

```
void TruckLoad::addBox(Box& rBox) {
  Package* pPackage = new Package(rBox);     //Create a Package

  if(pHead)                                  //Check list is not empty
```

```
    pTail->pNext=pPackage;                    //Add the new object to the tail
  else                                        //List is empty
    pHead = pPackage;                         //so new object is the head
  pTail = pPackage;                           //Store its address as tail
}
```

函数原型也应作相应的调整。这会改变 addBox()函数使用的方式。它现在把对象作为一个参数，而不是使用指针参数，这么做的一个缺点是在函数调用中，无法确定参数是通过引用传送的。在 TruckLoad 类的定义中，还必须修改这个函数中的参数声明。

我们仍旧希望把一个数组传送给 TruckLoad 类的构造函数，从而使参数列表保持不变，但现在必须把数组作为一个引用传送给 addBox()函数：

```
TruckLoad::TruckLoad(Box* pBox, int count) {
  pHead = pTail = pCurrent = 0;

  if ((count>0)&&(pBox!=0))
     for(int i =0; i<count; i++)
       addBox(*(pBox+i));
}
```

惟一的变化是解除了指针 pBox+i 的引用，给 addBox()函数传送对象，而不是地址。副本构造函数调用 addBox()成员，所以必须修改该构造函数的定义，如下所示：

```
TruckLoad:: TruckLoad(const TruckLoad& load) {
  pHead = pTail = pCurrent = 0;
  if(load.pHead==0)
    return;

  Package* pTemp=load.pHead;                  //Saves addresses for new chain
  do  {
    addBox(pTemp->rBox);
  }while(pTemp=pTemp->pNext);                 //Assign and then test pointer to next Box
}
```

getFirstBox()和 getNextBox()函数目前返回指针：应把它们改为返回引用吗？这取决于使用返回值的方式。在对 TruckLoad 类的 getNextBox()函数的后续调用中，会访问列表中后续的 Box 对象，并利用空返回值来检测列表末尾。而引用返回类型做不到这些，因为没有空引用。因此，应让这两个函数继续返回 Box 的指针。

对 getFirstBox()函数定义的调整如下所示：

```
Box* TruckLoad::getFirstBox() {
  pCurrent = pHead;
  return &pCurrent->rBox;
}
```

这返回了 Package 对象中 Box 对象的引用地址。在表达式中不需要使用括号，因为->运算符的优先级高于地址运算符。

getNextBox()函数的定义应改为：

```
Box* TruckLoad::getNextBox() {
```

```
  if(pCurrent)
    pCurrent = pCurrent ->pNext;          // pCurrent is not null so set to next
  else                                    // pCurrent is null
    pCurrent =pHead ;                     // so set to the first list element

  return pCurrent ? &pCurrent ->rBox : 0 ;
}
```

TruckLoad 类的构造函数不受影响，因此对 Package 类中使用引用所需要进行的修改已全部完成。

如果用这个版本的类运行 main()的最后一个版本，就必须在 main()中把传送给 addBox()的参数改为对象—— 只需在两个实例中解除对运算符 new 返回的地址的引用。第一个语句变成：

```
load1.addBox(*(new Box(random(dimLimit), random(dimLimit), random
                                                    (dimLimit))));
```

提示：

不要忘记修改 addBox()函数的签名，在函数原型中反映这种变化。

这说明了把引用用作类成员的一些暗示，但这不会改善类的功能。使用指针的方案是首选方案。在使用动态对象的引用时，对象总是有被删除的风险，所以引用就成为不存在的对象的一个别名。

13.7 本章小结

本章学习了在实现自己的类时可以组合在一起的基本部分。用户现在知道实现类需要做什么工作了。需要确定每个类应提供的功能的性质和范围。实现自己的类，就是定义一个数据类型—— 一致的实体，类需要反映它的本质和特性。

本章的要点如下所示：

- 只能通过构造函数的初始化列表来初始化类的引用成员。引用不能用赋值语句来初始化。
- 只要给函数按值传送对象，就会调用副本构造函数。其结果是传送给类的副本构造函数的参数必须是一个引用。
- 如果在类的成员函数中动态分配内存，就总是要执行析构函数来释放内存、实现副本构造函数和副本赋值运算符。
- 把类的所有成员都声明为 private，就可以限制对类的访问。此时，只有友元类可以创建该类类型的对象。
- 嵌套类是把自己的定义放在另一个类定义的内部。嵌套类的名称在包含类的作用域内。为了在包含类的外部引用嵌套的类类型，类型名称必须用包含类的名称来限定。
- 如果嵌套类的定义放在包含类的私有部分，嵌套类类型的对象就不能在包含类的外部创建。

13.8 练习

1. 编写一个类 Sequence，在自由存储区中按照升序存储整数值的递增序列。序列的长度和起始值在构造函数中提供。确保该序列至少有两个值，默认有 10 个值，从 0 开始(0，1，2，3，4，5，6，7，8，9)。需要有足够的内存空间来存储该序列，再用要求的值填充内存。

提供 show()函数列出该序列，以确保正确创建 Sequence 对象。

确保在销毁 Sequence 对象时，释放分配给序列的内存(注意：确保释放所有的内存！)。创建并输出 5 个随机长度(长度有限！)的序列和一个默认序列，来演示这个类的操作。

2. 编写一个函数，比较两个序列。如果 Sequence 对象有不同的长度，它们就是不同的，如果 Sequence 对象有相同的长度，但对应的值不同，它们也是不同的。只有 Sequence 对象有相同的长度，且对应的值也相同，它们才是相同的。把这个函数编写为 Sequence 类的一个成员。

3. 重新编写比较函数，把它作为 Sequence 类的一个友元。如何改变参数和调用该函数的方式？最好使用什么技术？

4a. 标准库包含一个 string 类，但创建自己的 string 类可以弄明白设计和编写 C++类的许多问题。使用基本的 char 数据类型编写一个 string 类，看看它如何隐藏使用 C 样式字符串时的复杂性。

为 MyString 类创建一个头文件，把它放在自己的命名空间中。给这个类提供两个私有数据成员：整型长度和 char*，char*指向对象所管理的字符串。为什么要把长度作为该类的一个数据成员存储？

4b. 创建类的一个实现文件(.cpp)，并提供构造函数，从下面的数据类型中构建 MyString 对象：

- 一个字符串字面量(例如 const char*类型)，以便编写 MyString s1("hello")。
- 一个重复多次的字符。默认的重复次数应是 1。使用这个构造函数的例子如 MyString s2('c', 5)。
- 一个整数值，这样 MyString s3(10)就存储了字符串"10"。

这些构造函数是显式的还是隐式的？构造函数在需要时应提供错误处理。

4c. 构造函数为存储字符串而分配内存；提供一个析构函数，在删除对象时正确地释放内存。

4d. 编写该类的副本构造函数，以便从其他字符串中创建和初始化 MyString 对象。

4e. 给类添加一些成员函数：

- 返回字符串的长度
- 输出字符串
- 索引从 0 开始，找出某个字符或子字符串在字符串中的位置，如果没有找到，就返回 - 1

现在可以编写一个测试程序，以各种方式创建并处理 mystring 对象。保证所有的成员都工作正常。

第 14 章　运算符重载

本章将探讨如何给类添加运算符的支持，使它们可以应用于类类型的对象，这会使自己定义的类型更像 C++的基本数据类型。

前面介绍了类可以包含成员函数，来操作对象的数据成员。但我们能做的决不仅仅是这些。通过运算符重载，可以让 C++的基本运算符以定义好的方式操作类对象。

本章主要内容

- 哪些 C++运算符可以用于自己的数据类型
- 如何在类中实现重载运算符的函数
- 如何把运算符函数实现为类成员和一般函数
- 何时必须实现赋值运算符
- 如何把类型转换定义为运算符函数
- 智能指针

14.1　为自己的类实现运算符

第 12、13 章开发的 Box 类可用于主要涉及盒子体积的应用程序。对于这样的应用程序，显然需要比较盒子的体积，确定盒子的相对尺寸。第 12 章在 Box 类中实现了 compareVolume()函数，用于比较两个对象的体积，但下面的语句：

```
if(Box1<Box2)
  //Do something…
```

显然比调用该函数更好一些：

```
if(Box1.compareVolume(Box2)<0)
  //Do something…
```

最好能用 Box1+Box2 这样的表达式把两个 Box 对象的体积加在一起，或者用 load1[2]这样的表达式从一组 Box 对象中选择第三个 Box 对象。这些操作都是可以实现的，只需实现重载类的基本运算符的函数即可。

14.1.1　运算符重载

运算符重载允许把标准运算符(如+、－、*、<等)应用于定制数据类型的对象。为此，要编写一个函数，重新定义每个运算符，使之在每次应用于类的对象时，都执行指定的操作。

例如，为了确定<运算符如何操作 Box 对象，可以在 Box 类中编写一个成员函数，以定义该运算符的操作。在本例中，我们对 Box 对象的体积感兴趣，可以把函数定义为：如果第一个 Box 对象的体积小于第二个 Box 对象的体积，就返回 true。这种函数的名称是 operator<()。

一般情况下，重载给定运算符的函数名由关键字 operator 和要重载的运算符组成。对于使用字母字符的运算符，例如 new 和 delete，在关键字和运算符之间至少要有一个空格。对于其他运算符，空格是可选的。

14.1.2 可以重载的运算符

运算符重载不允许发明新的运算符，也不能修改运算符的优先级或操作数的个数，所以运算符的重载版本与原运算符在计算表达式的顺序方面是相同的。附录 D 列出了所有运算符的优先级。

尽管不能重载所有的运算符，但限制不是非常严格。表 14-1 列出了不能重载的运算符：

表 14-1 不能重载的运算符

运 算 符	符 号
作用域解析运算符	::
条件运算符	?:
直接成员访问运算符	.
解除类成员指针引用的运算符	.*
sizeof 运算符	sizeof

提示：

前面没有提到类成员的指针运算符，详见第 16 章。

另外，也不能重载预处理器指令符号#和标志传送符号##。其他运算符都可以重载，这个范围相当广泛。为指定类重载运算符的函数不一定是该类的成员，它可以是一般函数。下面就介绍这两种不同函数的例子。

注意：

显然，最好确保标准运算符的重载版本与其一般的用法保持一致，或者至少在类中其含义和操作一目了然。而类的重载+运算符执行类对象中的相乘操作是不妥的。

理解运算符重载的工作原理的最佳方式是通过一个例子来学习，下面就为 Box 类实现刚才提到的小于运算符<。

14.1.3 实现重载运算符

要给类重载运算符，只需编写运算符函数。实现为类成员的二元运算符有一个参数，稍后解释它。在 Box 类定义中，重载<运算符的函数原型如下所示：

```
class Box {
  public:
    bool operator<(const Box& aBox) const;   //Overloaded 'less-than' operator

    //The rest of the Box class
};
```

这里实现的是一个比较操作，返回类型是 bool。调用运算符函数 operator<()后，会使用<比较两个 Box 对象。参数是<运算符的右操作数，左操作数对应于当前对象的 this 指针。由于该函数没有改变任何一个操作数，因此把参数和函数都指定为 const。

为了查看该重载运算符的工作情况，假定有下面的 if 语句：

```
if(box1<box2)
  cout<<"box1 is less than box2"<<endl;
```

括号中的测试表达式会调用运算符函数，这等价于函数调用 box1.operator<(box2)。事实上，还可以把上述语句改写为：

```
if(box1.operator<(box2))
  cout<<"box1 is less than box2"<<endl;
```

给 Box 对象使用<运算符，会使代码的可读性更高。

现在知道表达式 box1<box2 中的操作数映射于函数调用，就可以很容易地实现重载运算符。重载<运算符的成员函数定义如图 14-1 所示。

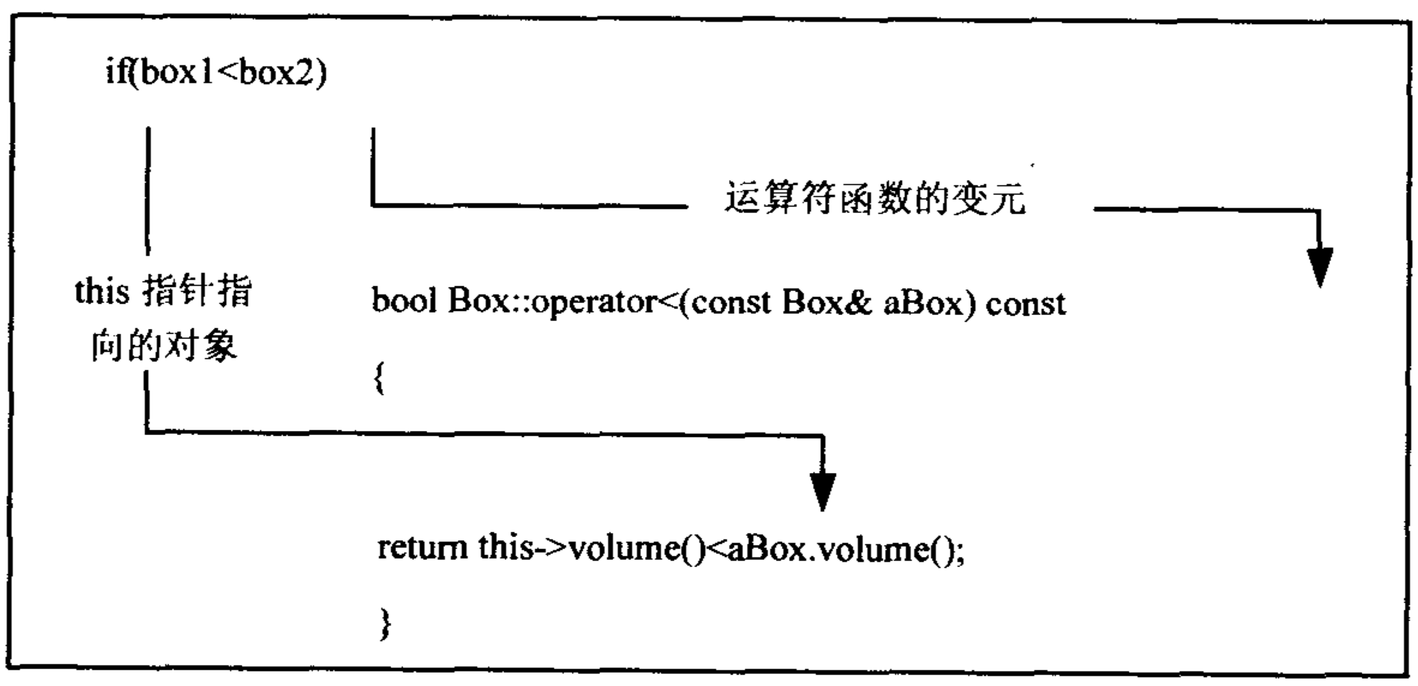


图 14-1　重载小于运算符

在调用函数时，使用函数的引用参数来避免不必要的参数复制。return 表达式使用成员函数 volume()来计算 this 指针所指向的 Box 对象的体积，其结果再与 aBox 对象的体积比较(使用基本的<运算符)。如果 this 指针所指向的 Box 对象的体积小于传送为引用参数的 aBox 对象的体积，就返回 true，否则返回 false。

注释：

this 指针用于表示与第一个操作数的关系，这里不必显式使用 this。

下面看看它是否工作。

程序示例 14.1—— 使用重载的<运算符

下面用一个例子来试验 Box 对象的重载小于运算符。与前一章一样，Box 类的定义包含在头文件 Box.h 中，这个类定义包含了重载运算符函数 operator<()的定义：

```
// Box.h - Definition of the Box class
#ifndef BOX_H
#define BOX_H
```

```
class Box {
  public:
    // Constructor
    Box(double aLength = 1.0,  double aWidth = 1.0,  double aHeight = 1.0);

    double volume()  const;                          // Calculate Box volume

    double getLength()   const;
    double getwidth() const;
    double getHeight()   const;

    bool operator<(const Box& aBox)  const   // Overloaded 'less-than' operator
      return volume()  < aBox.volume();      // Defined inline
    }
  private:
    double length;
    double width;
    double height;
};
#endif
```

注意这里把重载运算符定义为一个内联函数，编译起来，效率比较高。Box 类中其他成员的原型包含在 Box.cpp 中，如下所示：

```
// Box.cpp
#include "Box.h"

// Box constructor
Box::Box(double aLength, double aWidth, double aHeight):
        length(aLength),  width(aWidth),  height(aHeight)  {}

// Calculate Box volume
double Box::volume()  const {
  return  length*width*height;
}

// getXXX()  functions
double Box::getLength()   const  { return length;  }
double Box::getWidth()  const  { return width; }
double Box::getHeight()   const  { return height;  }
```

重载比较运算符后，compareVolume()成员就多余了，所以删除它。
利用重载的<运算符，在 Box 对象的数组中查找最大的元素，代码如下所示：

```
//  Program 14.1 Exercising the overloaded  'less-than'  operator    File:
//  prog14_01.cpp
#include <iostream>
#include <cstdlib>                          // For random number generator
#include  <ctime>                          //  For time  function
using std::cout;
using std::endl;
```

```
#include "Box.h"

// Function to generate random integers from 1 to count
inline int random(int count) {
   return 1 + static_cast<int>((count*static_cast<double>(std::rand()))/
    (RAND_MAX+1.0));
}

int main() {
    const int dimLimit = 100;          // Upper limit on Box dimensions
    std::srand((unsigned)std::time(0)); // Initialize the random number
    generator

    const int boxCount = 20;           // Number of elements in Box array
    Box boxes[boxCount];               // Array of Box objects

    for(int i = 0 ;  i < boxCount  ;  i++)
      boxes[i]  = Box(random(dimLimit),  random(dimLimit),  random(dimLimit));

// Find the largest Box object in the array
   Box* pLargest = &boxes[0];

for(int i = 1 ;  i < boxCount  ;  i++)
   if(*pLargest < boxes[i])
      pLargest = &boxes[i];

   cout << endl
        << "The box with the largest volume has dimensions:  "
        << pLargest->getLength()   <<  "by"
        << pLargest->getWidth()  <<  "by"
        << pLargest->getHeight()   << endl;
   return 0;
}
```

运行这个程序，结果如下所示：

```
The box with the largest volume had dimensions: 76 by 92 by 90
```

由于盒子的尺寸是随机数字，所以用户得到的结果可能与此不同。

例子的说明

函数 main()首先创建一个 Box 对象的数组。假定第一个数组元素是最大的，用下面的语句存储它的地址：

```
Box* pLargest =&boxes[0];
```

接着比较指针 pLargest 中包含的地址所对应的 Box 对象和后续的数组元素：

```
for(int i=1; i<boxCount; i++)
  if(*pLargest< boxes[i])
     pLargest= &boxes[i];
```

if 中的比较会调用函数 operator<()。该函数的参数是 boxes[i]，this 指针是 pLargest 所指向的 Box 对象。如果 boxes[i]的体积大于 pLargest 所指向的 Box 对象的体积，比较的结果就是 true，于是将 boxes[i]的地址作为新的最大 Box 对象。这里使用指针是因为这比其他方法更有效：

```
Box largest =boxes[0];
for(int i=1; i<boxCount; i++)
  if(largest< boxes[i])
     largest= boxes[i];       //Copies the object from the array
```

在这段代码中，与原代码相比有一个额外的系统开销。在循环中，每次在变量 largest 中存储 Box 对象时，boxes 数组中的对象都要逐个成员地复制到 largest 对象中。所需要的时间取决于对象的大小和复杂程度，但使用指针，就只复制地址。这段代码还产生了另一个系统开销：每次在 largest 中存储 Box 对象的新副本时，所存储的旧对象都必须删除，所以要调用析构函数。

在退出循环时，指针 pLargest 就包含数组中最大 Box 对象的地址。下面用指针输出最大对象的尺寸：

```
cout << endl
     << "The box with the largest volume had dimensions: "
     << pLargest->getLength() <<"by "
     << pLargest->getWidth() <<"by"
     << pLargest->getHeight() <<endl;
```

14.1.4 全局运算符函数

volume()函数是 Box 类的一个公共成员，因此可以在类的外部把重载的<运算符实现为一个一般函数，即全局运算符函数。此时，该函数的定义为：

```
inline bool operator<(const Box& Box1, const Box& Box2 ) {
  return Box1.Volume() < Box2.Volume();
}
```

这里把 operator<()声明为内联函数，因为它应像以前那样编译。以这种方式定义运算符后，前面例子中 main()函数的代码就会以相同的方式工作。当然，不能把这个版本的运算符函数声明为 const，const 只能应用于类的成员函数。

提示：

在把成员函数指定为 const 时，该函数就不能修改调用它的对象了，在这种情况下，const 只能用于它所属的对象。

即使运算符函数需要访问类的私有成员，也可以把它声明为类的友元函数，将它实现为一般函数。但一般情况下，如果函数必须访问类的私有成员，最好将它定义为类的成员。

14.1.5 提供对运算符的全部支持

为类实现诸如<这样的运算符会产生一个期望，即可以编写 Box1<Box2 这样的表达式，但可以使用 Box1<25.0 或者 10<Box2 这样的表达式吗？运算符函数 operator<()不能处理这样的表

达式。在开始实现类的重载运算符时，需要考虑运算符是否可以在类似的情形下使用。

注释：

在使用基本的<运算符时，例如比较 float 和 int，编译器会自动把一个操作数的类型转换为另一个操作数的类型，再执行比较。这是因为基本<运算符只能比较同类型的两个数值。Box 对象的情形有点不同，需要重载<运算符，使上述比较有意义。这里在比较体积(这表示每个操作数都是一个体积)时，是比较一个体积值或比较 Box 对象的体积。

很容易支持前面提到的表达式类型。首先添加一个函数，来比较 Box 对象的体积(Box 对象是第一个操作数)和 double 类型的第二个操作数。把这个函数定义为类外部的内联函数，看看它是如何工作的。在 Box 类定义的 public 部分，添加如下函数原型：

```
bool operator<(double aValue)const;        //Compare Box volume < double value
```

Box 对象传送给函数，作为隐式的指针 this，double 值则传送为一个参数。实现这个函数与第一个运算符函数一样容易，在其函数体中只有一个语句：

```
//Function to compare volume of a Box object with a constant
inline bool Box::operator<(double aValue)const  {
  return volume()<aValue;
}
```

这个定义应与 Box.h 中的类声明一致。内联函数不应在独立的.cpp 文件中定义，因为它的定义必须出现在使用该函数的每个源文件中。把它放在类定义中，可以确保它总是出现在使用该函数的每个源文件中。

处理像 10<box2 这样的表达式并不难，只是略有不同。成员运算符函数总是把 this 指针提供为左操作数。而在这个表达式中，左操作数是 double 类型，因此不能把运算符实现为成员函数。此时有两个选择；把函数实现为全局运算符函数，或实现为友元函数。由于类中不需要访问任何私有成员，因此可以把该函数实现为一般函数：

```
//Function comparing a constant with volume of a Box object
inline bool operator<(const double aValue, const Box& aBox)  {
  return aValue < aBox.volume();
}
```

现在，Box 对象的<运算符有三个重载版本，除了 box1<box2 这样的表达式之外，还可以处理 box1<2.5*box2.volume()或 0.5*(box1.Volume()+box2.Volume())<box3 等表达式。<运算符的两个操作数可以是值类型为 double 的任意表达式或 Box 对象。下面举例说明。

程序示例 14.2—— <运算符的完整重载

除了运算符函数之外，再给 Box 类添加一个函数，显示对象的尺寸，这会使 main()中的代码更紧凑。要更新程序示例 14.1 中的 Box.h 文件，首先删除旧的 operator<()函数定义，在类定义的公共部分添加如下原型：

```
bool operator<( const Box& aBox) const;        //Compare Box < Box
bool operator<( const double aValue)const;     //Compare Box < double value
```

接着在 Box.h 中添加如下内联成员函数定义：

```
//Function comparing Box object < Box object
inline bool Box::operator<(const Box& aBox) const {
  return volume() < aBox.volume();
}

//Function comparing Box object < double value
inline bool Box::operator<(const double aValue) const {
  return volume()<aValue;
}
```

最后，在 Box.h 的末尾添加全局运算符函数定义：

```
//Function comparing double value < Box object
inline bool operator<(const double aValue, const Box& aBox) {
  return aValue < aBox.volume();
}
```

这 3 个运算符函数定义应放在 Box 类定义和#endif 指令之间。把所有运算符函数的定义包含在头文件 Box.h 中，因为它们都是内联函数。只有不是内联成员函数的定义才应放在.cpp 文件中。因此，Box.cpp 文件与程序示例 14.1 相同。

应用新的运算符，查找出数组中体积在指定范围内的所有 Box 对象。下面是代码：

```
// Program 14.2 Exercising the overloaded 'less-than' operators   File:
// prog14_02.cpp
#include <iostream>
#include <cstdlib>                    // For random number generator
#include <ctime>                      // For time function
using std::cout;
using std::endl;

#include "Box.h"

// Function to generate random integers from 1 to count
inline int random(int count) {
   return 1 + static_cast<int>((count* static_cast<double>(std::rand()))  /
                               (RAND_MAX+1.0));
}

// Display box dimensions
void show(const Box& aBox) {
  cout << endl
       << aBox.getLength()  <<  "by"
       << aBox.getWidth()  <<  "by"  << aBox.getHeight();
}

int main() {
  const int dimLimit = 100;       // Upper limit on Box dimensions
  std::srand((unsigned)std::time(0));  // Initialize the random number
                                       // generator

  const int boxCount = 20;        // Number of elements in Box array
  Box boxes[boxCount];            // Array of Box objects
```

```
for(int i = 0 ;  i < boxCount ;  i++)
   boxes[i] = Box(random(dimLimit), random(dimLimit), random(dimLimit));

// Find the largest Box object in the array
Box* pLargest = &boxes[0];

for(int i = 1 ;  i < boxCount ;  i++)
    if(*pLargest < boxes[i])
       pLargest = &boxes[i];

cout << endl
     <<  "The largest box in the array has dimensions:";
show(*pLargest);

int volMin = 100000.0;                    // Lower Box volume limit
int volMax = 500000.0;                    // Upper Box volume limit
// Display details of Box objects between the limits
cout << endl << endl
     <<  "Boxes  with volumes  between  "
     <<  volMin  <<  "  and  "  <<  volMax  <<  "  are:";
for(int i = 0  ;  i < boxCount ;  i++)
    if(volMin < boxes[i]  && boxes[i]  < volMax)
       show(boxes[i]);
   cout  <<  endl;
   return 0;
}
```

运行这个例子，结果如下所示：

```
The largest box in the array has dimensions:
100 by 79 by 99

Boxes with volumes between 100000 and 500000 are:
92 by 38 by 46
76 by 83 by 44
83 by 78 by 18
31 by 87 by 83
93 by 15 by 90
62 by 64 by 88
```

例子的说明

函数 show()显示传送为参数的 Box 对象的尺寸，这只是一个在 main()中使用比较方便的函数。

main()的第一部分创建了一个 Box 对象数组，找出体积最大的 Box 对象。这个过程使用了 operator<()函数的最初版本，该版本定义为 Box 类的一个成员。为了使用两个新的运算符函数，本例添加了一些代码，列出体积在 volMin 和 volMax 之间的所有 Box 对象。这些 Box 对象在下面的循环中查找：

```
for(int i=0 ; i<boxCount; i++)
```

```
if(volMin<boxes[i] && boxes[i]< volMax)
  show(boxes[i]);
```

if 表达式调用了运算符的两个新版本。子表达式 volMin<boxes[i]等价于 operator<(volMin, boxes[i])，而 boxes[i]< volMax 等价于用表达式 boxes[i].operator<(volMax)调用成员函数。为了显示满足条件的 Box 对象的尺寸，我们把该对象传送给了 show()函数。

注释：

任何比较运算符都可以用实现小于运算符的方式来实现。它们仅在一些很小的方面有区别，实现它们的一般方法则完全相同。

14.1.6 运算符函数术语

可以重载的所有二元运算符总是与上一节介绍的运算符函数有相同的形式。在重载运算符 X 时，左操作数是重载 X 的类的对象，定义重载的成员函数的一般形式如下：

```
返回类型  operator X(类型 右操作数);
```

返回类型取决于运算符所执行的操作。对于比较和逻辑运算符来说，它通常是 bool(也可以使用 int)。像+和*等运算符，就需要返回某种形式的对象。

在使用非成员函数实现二元运算符时，其形式如下：

```
返回类型 operator X(类类型 左操作数，类型 右操作数);
```

其中类类型是重载运算符 X 的类。在二元运算符中，如果 Type 类型的左操作数不实现为 Type 类的成员，该函数就必须实现为全局运算符函数，其形式如下：

```
返回类型 operator X(类型 左操作数，类类型 右操作数);
```

在把一元运算符实现为类的成员函数时，它们一般不需要参数，但递增和递减运算符例外。例如，一元运算符 Op 实现为类 Class_Type 的成员，其形式如下所示：

```
类类型& operator Op();
```

一元运算符实现为全局运算符函数时，只有一个参数，这个参数就是操作数。如果一元运算符 Op 实现为非成员函数，其形式如下：

```
类类型& operator Op(类类型&);
```

注意，对于所有的运算符函数，包括成员函数和全局运算符函数，参数的个数都是固定的。必须使用该运算符指定的参数个数。

这里不再介绍重载每个运算符的例子，因为大多数运算符都与前面介绍的运算符类似。下面详细讨论重载时有特质的运算符。

14.1.7 重载赋值运算符

赋值运算符就是运算符=，它这样表示是为了与运算符+=、*=等区分开。赋值运算符会把

给定类型的对象(等号的右边)复制到同一类型的另一个对象中(等号的左边)。下面的语句会调用赋值运算符：

```
Box box1;
Box box2(10,10,10);
box1=box2;                    //Call the assignment operator
```

如果没有提供重载的赋值运算符函数来复制类的对象。编译器就会提供默认版本operator=()。

注意，如果定义一个极为简单的类，例如只有一个数据成员的类：

```
class Data {
  public:
    int value;
};
```

根据使用它的方式，就会得到如下结果：

```
class Data  {
  public:
     int value;

     Data(){}                                          // Default constructor
     ~Data(){}                                         // Destructor

     Data(const Data& aData)  : Value(aData.value)  {}  // Copy constructor

     Data& operator=(const Data& aData) {                    // Assignment operator
       value  =  aData.value;
       return *this;
     }
};
```

赋值运算符的默认版本会简单地进行逐个成员的复制过程，类似于默认的副本构造函数。

提示：

不要混淆副本构造函数和赋值运算符函数。它们是完全不同的。在类对象用该类的另一个已有对象进行创建和初始化时，或者对象按值传送给函数时，就会调用副本构造函数。而当赋值语句的左边和右边是同一个类的对象时，才会调用赋值运算符函数。

对于 Box 类，默认的赋值运算符工作起来没有问题，但对于成员是动态分配内存的类来说，就需要仔细考虑赋值运算符的操作了。如果没有在这些环境下实现赋值运算符，程序就会出现混乱。其问题与前面介绍副本构造函数时讨论的内容类似。假定把默认的赋值运算符应用于动态分配内存的对象，就会有两个对象共享自由存储区中的对象，这样这两个对象就是相互依赖的。改变其中一个对象，就会使另一个对象失效。

第 13 章中的 TruckLoad 类就是这种情况。把 TruckLoad 的成员逐个复制到另一个 TruckLoad 中，就会使两个对象共享同一个列表，这是第 13 章必须解决的问题。解决方案很简单：实现赋值运算符，使动态创建的对象的各个部分正确地复制。但这仅完成了一部分工作。

提示：

事实上，默认副本构造函数也有与默认赋值运算符相同的问题。如果需要实现一个副本构造函数，就需要实现另一个副本构造函数，即析构函数。

实现赋值运算符

下面看看函数要做什么。赋值运算符有两个操作数。右操作数是要复制的对象，左操作数是赋值的目标，它是右操作数的副本。在运算符函数中要执行复制，那么返回类型应是什么？是否需要返回值？

下面看看如何在实际中应用该函数。根据赋值运算符的一般用法，可以编写下面的语句：

```
load1=load2=load3;
```

这是 3 个 TruckLoad 类型的变量，load1 和 load2 是 load3 的副本。由于赋值运算符是右相关的，所以该语句等价于：

```
load1=(load2=load3);
```

最右边的赋值语句的结果是最左边赋值操作的右操作数，因此肯定需要有返回值。根据成员函数 operator=()，它等价于：

```
load1.operator=(load2.operator=(load3));
```

显然，从 operator=()返回的内容是另一个 operator=()调用的参数。operator=()的参数是一个对象引用，因此函数必须返回一个对象，该对象就是函数的左操作数。而且，如果为了避免不必要的复制操作，返回类型必须是该对象的引用。

复制右操作数的过程与副本构造函数相同，现在知道了返回类型，就可以定义 TruckLoad 类的=运算符了。考虑下面的代码，之后讨论其中有什么错误：

```
TruckLoad& TruckLoad::operator=(const TruckLoad& load) {
   pHead = pTail = pCurrent = 0;
   if(load.pHead == 0)
      return;

   Package* pTemp = load.pHead;          // Saves addresses for new chain
   do {
      addBox(pTemp->pBox);
    }while(pTemp = pTemp->pNext);

   return *this;                         // Return the left operand
}
```

this 指针包含左参数的地址，所以返回*this 就返回对象。除此以外，这段代码与副本构造函数的代码完全相同。函数看起来不错，而且在大多数情况下都会正常工作，但其中有两个问题。

第一个问题是左操作数。TruckLoad 对象潜在地包含一个列表。把其数据成员 pHead 设置为 0，就删除了 TruckLoad 对象所拥有的所有 Package 对象。首先必须删除左操作数拥有的所有 Package 对象，所以在函数定义中添加如下代码：

```
TruckLoad& TruckLoad::operator=(const TruckLoad& load) {
  while(pCurrent = pHead) {      //Copy and check pointer for null
     pHead = pHead->pNext;       //Move pHead to the address of the next object
     delete pCurrent;            //Delete current object
  }

   pHead = pTail = pCurrent = 0;
   if(load.pHead == 0)
      return;
  Package* pTemp = load.pHead;   //Saves addresses for new chain
  do {
     addBox(pTemp->pBox);
  }while(pTemp = pTemp->pNext);

  return *this;                  //Return the left operand
}
```

如果左操作数包含一个列表，pHead 就为非空。while 循环会把指针指向列表中的第一个 Package 对象，并检查它是否为空。如果它不是空指针，就执行循环体，把 pHead 移动到列表中下一个 Package 对象的地址上，然后删除列表中的当前第一个对象。这样，就可以按顺序处理列表，重复删除第一个对象，直到找到空指针为止。

这是在处理左操作数，但仍有一个问题。假定有下面的语句：

```
load1=load1;
```

这看起来不像是一个有意义的赋值语句，但在比较复杂的语句中会出现这种情况，只是不太明显。在这种情况下，由于更正了第一个问题，删除了包含在对象中的列表，这个语句就试图复制目前不存在的列表！因此，需要查看两个操作数是否相等。为此，修改函数，如下所示：

```
TruckLoad& TruckLoad::operator=(const TruckLoad& load) {
  if(this == &load)              // Compare operand addresses
    return *this;                // if equal return the lst operand

  while(pCurrent = pHead) {      // Copy and check pointer for null
    pHead = pHead->pNext;        // Move pHead to the address of the next object
    delete pCurrent;             // Delete current object
  }

   pHead = pTail = pCurrent = 0;
   if(load.pHead == 0)
     return;

  Package* pTemp = load.pHead;  // Saves addresses for new chain
  do {
    addBox(pTemp->pBox);
  }while(pTemp = pTemp->pNext);

  return *this;                 // Return the left operand
}
```

如果两个操作数是相同的对象，它们就具有相同的地址，比较 this 和传送给 operator=()的参数地址就不会有问题。

注意：

只要编写了赋值运算符，就总是要检查两个操作数是否为同一个对象。

从第 13 章的最后开始讨论的动态分配内存的类中，可以得到如下黄金规则：

如果类的函数在自由存储区中动态分配内存，就总是应实现副本构造函数、赋值运算符和析构函数。

副本赋值运算符不仅可以重载为复制对象。一般情况下，一个类可以有若干个赋值运算符的重载版本。其他赋值运算符重载版本的参数类型可以与类类型不同，它们实际上相当于类型转换。在任何情况下，返回类型都应是左操作数的引用。当然，也可以重载 op=形式的其他运算符。

在许多情况下，可以禁止对类的对象执行某些赋值运算。此时，可以把赋值运算符声明为类的私有成员。

程序示例 14.3—— 实现赋值运算符

下面用一个简单的例子来说明，类对象在定义和没有定义赋值运算符的情况下会如何操作，从而验证赋值运算符的重要性。下面定义一个类 ErrorMessage，表示一个错误消息，再定义副本构造函数、赋值运算符和析构函数：

```
// ErrorMessage.h
#ifndef ERRORMESSAGE_H
#define ERRORMESSAGE_H
#include <iostream>
using namespace std;

class ErrorMessage {
 public:
    ErrorMessage(const char* pText = "Error");          // Constructor
    ~ErrorMessage();                                    // Destructor
    void resetMessage();                                // Change the message
    ErrorMessage& operator=(const ErrorMessage& Message);   // Assignment
                                                        // operator


    char* what() const{ return pMessage;  }      // Display the message

 private:
    char* pMessage;
};
#endif
```

当然，这个类必须定义副本构造函数、副本赋值运算符和析构函数，因为它是动态分配内存的。从类中删除副本赋值运算符，就可以看出没有实现赋值运算符会发生什么。

构造函数、析构函数和其他成员函数的定义放在 ErrorMessage.cpp 中：

```
// ErrorMessage.cpp ErrorMessage class implementation
#include <cstring>
#include "ErrorMessage.h"
```

```
using std::cout;
using std::endl;

// Constructor
ErrorMessage::ErrorMessage(const char* pText) {
  pMessage = new char[ strlen(pText) + 1 ];              // Get space for message
  std::strcpy(pMessage, pText);                         // Copy to new memory
}

// Destructor to free memory allocated by new
ErrorMessage::~ErrorMessage() {
  cout << endl << "Destructor called." << endl;
  delete[] pMessage;                                    // Free memory for message
}

// Change the message
void ErrorMessage::resetMessage() {
   // Replace message text with asterisks
   for(char* temp = pMessage ; *temp != '\0' ; *(temp++) = '*')
      ;
}

// Assignment operator
 ErrorMessage& ErrorMessage::operator=(const ErrorMessage& message) {
    if(this == &message)                                // Compare addresses, if equal
       return *this;                                    // return left operand

    delete[] pMessage;                                  // Release memory for left operand
    pMessage = new char[ strlen(message.pMessage) + 1];

    // Copy right operand string to left operand
    std::strcpy(this->pMessage, message.pMessage);

    return *this;                                       // Return left operand
}
```

把类 ErrorMessage 的这个版本放在下面的源文件中：

```
// Program 14.3 Overloading the copy assignment operator  File: prog14_03.cpp
#include <iostream>
#include <cstring>
using std::cout;
using std::endl;

#include "ErrorMessage.h"

int main() {
   ErrorMessage warning ("There is a serious problem here");
   ErrorMessage standard;

   cout << endl << "warning contains - " << warning.what();
   cout << endl << "standard contains - " << standard.what();
```

```
    standard = warning;                    // Use assignment operator

    cout << endl << "After assigning the value of warning, standard contains - "
         << standard.what();

    cout << endl << "Resetting warning, not standard" << endl;
    warning.resetMessage();                // Reset the Warning message

    cout << endl << "warning now contains - " << warning.what();
    cout << endl << "standard now contains - " << standard.what();
    cout << endl;

    return 0;
}
```

编译并运行这个例子，结果如下所示：

```
warning contains - There is a serious problem here
standard contains - Error
After assigning the value of warning, standard contains -
There is a serious problem here
Resetting warning, not standard

warning now contains - ********************************
standard now contains - There is a serious problem here

Destructor called.

Destructor called.
```

可以看出，一切正常。可以重新设置包含在第一个 ErrorMessage 对象中的消息，且不影响要复制的对象。

现在删除或注释掉类定义中赋值运算符的声明，以及 ErrorMessage.cpp 中的定义。重新编译程序并运行，结果如下：

```
warning contains - There is a serious problem here
standard contains - Error
After assigning the value of warning, standard contains -
There is a serious problem here
Resetting warning, not standard

warning now contains - ********************************
standard now contains - ********************************

Destructor called.

Destructor called.
```

产生了其他错误消息，因为程序试图对同一个对象释放两次自由存储区中的内存。

例子的说明

在这个例子中，比较有趣的是没有赋值运算符的怪异操作。在 main()函数中，创建了一个对象 warning，它包含一个特定的文本字符串，如下面的语句所示：

```
ErrorMessage warning("There is a serious problem here");
```

这会调用 ErrorMessage 构造函数，并把字符串传送给它，作为一个参数。之后使用默认的消息字符串创建一个 ErrorMessage 对象：

```
ErrorMessage standard;
```

在显示这两个对象的消息后，调用 ErrorMessage 类的赋值运算符，如下面的语句所示：

```
standard= warning;          //Use assignment operator
```

第二次运行程序时，这是编译器提供的默认赋值运算符，因此两个对象的 pMessage 成员包含相同的地址。结果是，在下面的语句中修改 warning 对象时：

```
warning.resetMessage();   //Reset the Warning message
```

就会把两个对象的消息修改为相同的内容。输出结果表明，standard 字符串与 warning 字符串相同。

在程序的最后，两个 ErrorMessage 对象超出了作用域，于是调用它们的析构函数，如输出所示。第一个析构函数调用会在自由存储区中删除 pMessage 指针指向的字符串。在有缺陷的版本中，当调用第二个析构函数时，会再次删除同一个字符串，因为两个 ErrorMessage 对象的 pMessage 成员包含相同的地址，系统会产生警告。所有这些都是因为没有定义赋值运算符。

第一次运行程序时，使用了 ErrorMessage 类和已实现的赋值运算符，因此会创建右操作数的独立副本，对 warning 的修改不会影响 standard。两个对象中的 pMessage 指针指向不同的字符串，所以析构函数不会出问题。

14.1.8 重载算术运算符

下面看看如何为 Box 类重载加号运算符。这是一个有趣的过程，因为加号是一个二元运算符，涉及到创建并返回新对象。新对象是两个 Box 对象的总和(其含义由用户指定)，而这两个对象是加号运算符的两个操作数。

总和的含义是什么？这有许多可能性，但由于盒子的主要作用是保存物品，所以我们感兴趣的是其容量，则两个盒子的总和就是一个可以保存这两个盒子中所有物品的盒子。

在这个基础上，把两个 Box 对象的总和定义为一个 Box 对象，它可以容纳两个盒子内堆放的物品。这与 Box 类用于打包物品的目的一致，因为把许多 Box 对象加在一起，就会得到一个可以包含所有这些 Box 对象的大 Box 对象。

这可以用一个简单的方式来完成。新对象仍有 length 成员，它比要加在一起的对象的 length 成员大。width 成员以类似的方式派生，height 成员是两个操作数的 height 成员之和，所以得到的 Box 对象可以包含另外两个 Box 对象。修改构造函数，就可以使 Box 对象的 length 成员总是大于或等于 width 成员。

图 14-2 列出了把两个 Box 对象加在一起，得到一个新 Box 对象的过程。

因为相加的结果是一个新的 Box 对象，所以实现相加的函数必须返回一个 Box 对象。如果重载+运算符的函数是一个成员函数，则在 Box 类定义中，该函数的声明如下所示：

```
Box operator+(const Box& aBox)const;          //Adding two Box objects
```

把参数定义为 const，是因为该函数不会修改它，把它定义为引用，是为了在调用函数时避免对右操作数进行不必要的复制。可以把该函数声明为 const，因为它不改变左操作数。成员函数的定义如下所示：

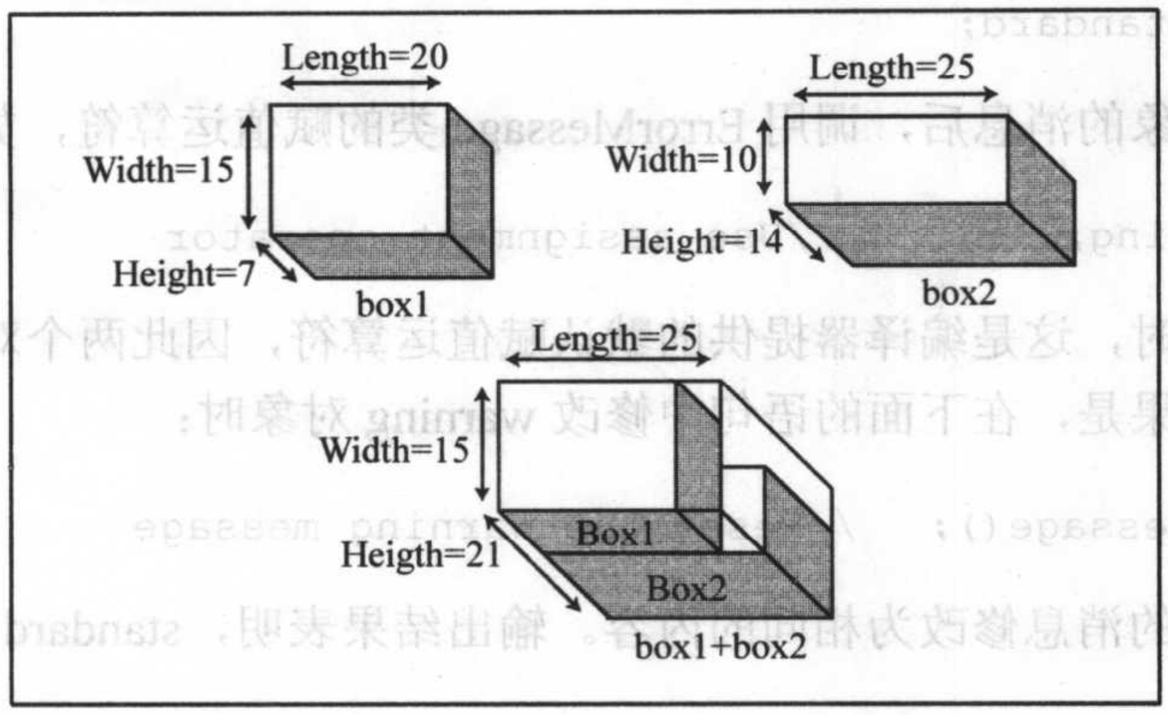


图 14-2 将两个 Box 对象加在一起

```
// Function to add two Box objects
inline Box Box::operator+(const Box& aBox) const {
   // New object has larger length and width, and sum of heights
   return Box( length  > aBox.length  ? length   : aBox.length,
               width > aBox.width ? width : aBox.width,
               height + aBox.height );
}
```

特别要注意这里并没有在自由存储区中创建一个 Box 对象，以返回给调用者。以这种方式实现函数是非常不好的，因为很难看出如何确保释放了内存。返回一个指针也会影响其他运算符如 operator=()的编写。函数应编写为创建一个本地 Box 对象，并给调用程序返回一个副本。这些都是自动变量，所以内存的管理也是自动的。

程序示例 14.4—— 重载加号运算符

下面用一个例子来说明新的加号运算符是如何工作的。在程序示例 14.3 中使用的 Box 类中添加该加号运算符。在 Box.h 中，为 Box 类定义的公共部分添加运算符函数的声明：

```
class Box {
  public:
     // Constructor
     Box(double aLength = 1.0, double aWidth = 1.0, double aHeight = 1.0);

     double volume()  const;                    // Calculate Box volume

     double getLength()   const;
     double getWidth() const;
```

```
   double getHeight()   const;

   bool operator<(const Box& aBox)  const;     // Compare Box < Box
   bool operator<(const double aValue)  const;   // Compare Box < double value
   Box operator+(const Box& aBox) const;        // Function to add two Box objects
private:
   double length;
   double width;
   double height;
};
```

在 Box.h 文件中，把 operator+()函数的内联定义添加到类定义的后面。

在 Box.cpp 中，修改构造函数的定义：

```
Box::Box(double aLength, double aWidth, double aHeight) {
   double maxSide = aLength > aWidth ? aLength :  aWidth;
   double minSide = aLength < aWidth ? aLength :  aWidth;
   length = maxSide > 0.0 ? maxSide  :  1.0;
   width = minSide > 0.0 ? minSide  :  1.0;
   height = aHeight > 0.0 ? aHeight  :  1.0;
}
```

修改后的构造函数将确保 length 成员包含长度和宽度更大的 Box 对象，从而以一致的方式表示对象，但加号运算符会产生高而窄的盒子。

下面组合使用加号运算符和<运算符，把两个 Box 对象加在一起后，从数组中查找体积最小的 Box 对象对。在有 20 个 Box 对象的数组中，需要检查 380 对 Box 对象。下面是代码：

```
// Program 14.4 Adding Box objects   File: prog14_04.cpp
#include <iostream>
#include <cstdlib>                  // For random number generator
#include <ctime>                    // For time functinn
using std::cout;
using std::endl;

#include "Box.h"

// Function to generate random integers from 1 to count
inline int random(int count) {
   return 1 + static_cast<int>((count* static_cast<double>(std::rand()))/
                               (RAND_MAX + 1.0));
}

// Display box dimensions
void show(const Box& aBox) {
    cout << endl
         << aBox.getLength() << "by"
         << aBox.getWidth() << "by" << aBox.getHeight ();
}

int main () {
    const int dimLimit = 100;        // Upper limit on Box dimensions
```

```
    std::srand((unsigned) std::time(0));  // Initialize the random number
                                          // generator

    const int boxCount = 20;              // Number of elements in Box array
    Box boxes[boxCount];                  // Array of Box objects

// Create 20 Box objects
for(int i = 0 ; i < boxCount ; i++)
 boxes[i] = Box(random(dimLimit), random(dimLimit), random(dimLimit));

int first = 0;                            // Index of first Box object of pair
int second = 1;                           // Index of second Box object of pair
double minVolume = (boxes[first] + boxes[second]).volume();

for(int i = 0  ;  i < boxCount -1 ;  i++)
   for(int j = i + 1 ; j < boxCount ; j++)
      if(boxes[i]  + boxes[j]  < minVolume) {
         first = i;
         second = j;
         minVolume =  (boxes[i]  + boxes[j]).volume();
      }

cout <<  "The objects that sum to the smallest volume are:";
cout << endl << "boxes[" << first << "]  ";
show(boxes[first]);
cout << endl << "boxes["  << second <<  "]  ";
show(boxes[second]);
cout << endl <<  "Volume of the sum is"  << minVolume << endl;

return 0;
}
```

结果如下：

```
The objects that sum to the smallest volume are:
boxes [8]
15 by 15 by 23
boxes [16]
21 by 15 by 41
Volume of the sum is 20160
```

每次运行程序时，都会得到不同的结果。

例子的说明。

我们记录下数组中的元素对，这对 Box 对象加在一起后，得到的体积最小，并把这两个 Box 对象的索引值存储在变量 first 和 second 中。把最前面的两个数组元素设置为初始对，并存储它们组合起来的体积，如下面的语句所示：

```
double minVolume=(boxes[first]+ boxes[second]).volume();
```

因为 operator+()返回一个 Box 对象，所以可以为该 Box 对象调用 volume()函数，这个 Box 对象是把 boxes[first]和 boxes[second]加在一起后得到的。两个 Box 对象之和所使用的括号是必

不可少的，因为成员访问运算符的优先级高于+。当然，如果要保存组合在一起的 Box 对象，就可以使用下面的语句：

```
Box combined= boxes[first]+ boxes[second];        //Assigns sum to new Box object
double minVolume= combined.volume();
```

在嵌套的循环中查找组成最小 Box 对象的元素对：

```
for(int i = 0 ;  i < boxCount - 1 ;  i++)
   for(int j = i + 1 ; j < boxCount ; j++)
   if(boxes[i]  + boxes[j] < minVolume) {
      first = i;
      second = j;
      minVolume =  (boxes[i]  + boxes[j]).volume();
   }
```

外层循环由 i 控制，迭代数组中从 1 到倒数第二个元素。内层循环由 j 控制，把 i 选择出来的 Box 对象和数组中在该 Box 对象后面的对象组合起来。对于每对 Box 对象，都在 if 表达式中，用重载运算符<对总和的体积与 minVolume 进行比较。这个表达式等价于：

```
(boxes[i].operator+( boxes[j])).operator<(minVolume)
```

operator+()返回一个 Box 对象，然后调用该对象的 operator<()函数。如果 operator<()函数返回 true，就在 first 和 second 中记录当前索引值，把 Box 对象之和的体积存储在 minVolume 中。

最后，使用 show()函数输出找到的 Box 对象的结果，如下面的语句所示：

```
cout << "The objects that sum to the smallest volume are:";
cout << endl << "boxes[" << first << "]  ";
show(boxes[first]);
cout << endl << "boxes[" << second << "]  ";
show(boxes[second]);
cout << endl << "Volume of the sum is "<< minVolume << endl;
```

当然，重载的+运算符可以用于更复杂的表达式，把 Box 对象加起来。例如，下面的语句：

```
Box box4=box1+ box2+ box3;
```

结果是 Box 对象 box4 包含其他三个叠加在一起的 Box 对象。

还可以把类的加法操作实现为一般函数(即非成员函数)，因为 Box 对象的尺寸可以通过公共成员函数来访问。这种函数的原型如下所示：

```
Box operator+(const Box& aBox, const Box& bBox);
```

如果不能以这种方式访问数据成员的值，仍可以把它声明为 Box 类中的一个友元函数，将它编写为一般函数。在这些选择中，友元函数通常是最不好的，因此最好选择其他方法。运算符函数是相当基本的类功能，一般把它们实现为类成员，这表示运算符操作是类型的一个组成部分。

根据一个运算符实现另一个运算符

如果为一个类实现了加法操作，就可以实现+=运算符。当然，+和+=之间并没有联系。但

如果要实现这两个运算符，就应注意，根据+=来实现+是非常经济的。

首先，定义为 Box 类实现+=的函数。因为涉及到赋值，所以需要返回一个引用。使用在加号运算符中把 Box 对象加在一起的方法，其定义如下：

```
// Overloaded += operator
inline Box& Box::operator+=(const Box& right) {
   length =   length   > right.length   ? length   :  right.length;
   width = width > right.width ? width :  right.width;
   height += right.height;
   return *this;
}
```

这非常简单，只是根据 Box 对象的加法定义，给左操作数*this 加上右操作数，修改了左操作数。得到的对象可以包含两个叠加在一起的原对象。

现在可以使用 operator+=()来实现 operator+()，函数就简化为：

```
//Function to add two Box objects
inline Box Box::operator+( const Box& aBox) const {
  return Box(*this)+=aBox;
}
```

其中，表达式 Box(*this)调用副本构造函数，创建了左操作数的副本。然后调用 operator+=()函数，给新 Box 对象加上右操作数对象 right，并返回新的 Box 对象。

可以把这一方法应用于实现类的-=、*=等重载版本。在根据一个运算符定义另一个运算符时，可以进行相同的比较。前面定义了 operator<()函数，下面根据它定义重载的>=函数：

```
inline bool Box::operator>=( const Box& aBox)const {
  return !(*this<(aBox));
}
```

小于的反就是大于或等于，因此可以定义出这个函数。

14.1.9 重载下标运算符

下标运算符[]为某些类提供了非常有趣的功能。显然，这个运算符的主要作用是从许多可解释为数组的对象中选择。但实际上，对象可以包含在任意多个不同的容器中。重载下标运算符可以访问稀疏数组(许多元素都为空的数组)、关联数组或链表中的元素。数据甚至可以存储在文件中，使用下标运算符可以隐藏文件输入和输出操作的复杂性。由于可以控制在运算符函数中的操作，这甚至可以改善标准下标运算符的工作方式，例如，检查给定的索引值是否合法。

第 13 章中的 TruckLoad 类就支持下标运算符的实现。每个 TruckLoad 对象都包含一组有序的对象，用户不必编写代码来迭代列表中的 Box 对象，而可以通过索引值来访问它们。索引为 0，就返回列表中的第一个对象，索引为 1，则返回第二个对象，依此类推。下标运算符的内部工作机制会迭代列表，找到需要的对象。

下面看看在这种情况下 operator[]()函数是如何工作的。该函数需要接受一个解释为列表中某位置的索引值，返回该位置的 Box 对象。如果要与表达式 load[3]的一般含义(它是 load 对象所表示的列表中的第 4 个对象)保持一致，该函数就必须返回一个对象，而不是一个指针。在

TruckLoad 类中，该函数的声明如下所示：

```
class TruckLoad {
  public:
      Box operator[] (int index) const;          // Overloaded subscript operator
     // Rest of the class as before...
};
```

该函数的实现代码如下所示：

```
// Subscript operator
Box TruckLoad::operator[] (int index) const {
   if(index<0) {                                 // Check for negative index
     cout << endl <<  "Negative index";
     exit(1);
   }

   Package* pPackage = pHead;                    // Address of first Package
   int count = 0;                                // Package count
   do {
     if(index == count++)                        // Up to index yet?
       return *pPackage->pBox;                   // If so return the Box
   } while(pPackage = pPackage->pNext);

   cout << endl <<  "Out of range index";        // If we get to here index is too high
   exit(1);
}
```

注释：

一些老式的编译器不支持 exit()的这种用法，它们会生成一个错误，认为函数没有返回一个值。如果出现了这个错误，就应使用 return *pTail->pBox;替换 exit(1);，它不会测试对象的索引值，但代码至少会编译成功。

如果 index 的值是负的，就显示一个消息，之后调用 exit()函数终止程序。do-while 循环会遍历列表，递增 count。当 count 的值与 index 相同时，循环就找到所需要的 Package 对象，并返回与该 Package 对象对应的 Box 对象。如果遍历了整个列表，count 的值也不与 index 相同，该 index 就一定超出了范围，于是在显示一个消息后终止程序。

提示：

这里终止程序并不是处理这个问题的好方法。如果可能，应允许程序继续运行，并发出信号，说明出现了一个错误。C++提供了处理这类问题的异常，异常在第 17 章讨论。

下面用另一个例子来试验下标运算符。

程序示例 14.5—— 重载下标运算符

这里使用程序示例 14.4 定义的 Box 类。在 List.h 中扩展 TruckLoad 类定义，使之包含重载的下标运算符函数：

```
// List.h classes supporting a linked list
```

```
#ifndef LIST_H
#define LIST_H

#include "Box.h"

class TruckLoad {
public:
     // Constructors
     TruckLoad(Box* pBox = 0, int count = 1);       // Constructor
     TruckLoad::TruckLoad(const TruckLoad& Load);  // Copy constructor

     ~TruckLoad();                                  // Destructor

     Box* getFirstBox();                            // Retrieve the first Box
     Box* getNextBox();                             // Retrieve the next Box
     void addBox(Box* pBox);                        // Add a new Box to the list
     Box operator[] (int index) const;              // Overloaded subscript operator

   private:
     class Package {
          public:
               Box* pBox;                           // Pointer to the Box
               Package* pNext;                      // Pointer to the next Package

               Package(Box* pNewBox);               // Constructor
};
          Package* pHead;                           // First in the list
          Package* pTail;                           // Last in the list
          Package* pCurrent;                        // Last retrieved from the list
};

#endif
```

该函数的实现代码放在 List.cpp 文件中。为了确保在该文件中包含下标运算符函数的完整代码，下面列出了该文件的全部代码：

```
// List.cpp Implementations for the Package and TruckLoad classes
#include <iostream>
#include "Box.h"
#include "List.h"
using std::cout;
using std::endl;

// Package class functions
// Package constructor
TruckLoad::Package::Package(Box* pNewBox) :pBox(pNewBox), pNext(0){}

// TruckLoad class functions
// Constructor
TruckLoad::TruckLoad(Box* pBox, int count) {
   pHead = pTail = pCurrent = 0;
```

```
    if(count > 0 && pBox != 0)
    for(int i = 0 ;  i<count ;  i++)
       addBox(pBox+i);
   return;
}

// Copy constructor
TruckLoad::TruckLoad(const TruckLoad& Load) {
   pHead = pTail = pCurrent = 0;
    if(Load.pHead == 0)
       return;

    Package* pTemp = Load.pHead;          // Saves addresses for new chain
    do {
       addBox(pTemp->pBox);
    }while(pTemp = pTemp->pNext);
}

// Destructor
TruckLoad::~TruckLoad() {
   while(pCurrent = pHead->pNext) {
       delete pHead;                      // Delete the previous
       pHead = pCurrent;                  // Store address of next
   }
       delete pHead;                      // Delete the last
}

// Get the first Box in the list
Box* TruckLoad::getFirstBox() {
    pCurrent = pHead;
    return pCurrent->pBox;
}

// Get the next Box in the list
Box* TruckLoad::getNextBox() {
    if(pCurrent)
       pCurrent = pCurrent->pNext;      // pCurrent is not null so set to next
    else                                // pCurrent is null
       pCurrent = pHead;                // so set to the first list element
    return pCurrent ? pCurrent->pBox :  0;
}

// Add a list element
void TruckLoad::addBox(Box* pBox) {
    Package* pPackage = new Package(pBox);          // Create a Package

   if(pHead)                                        // Check list is not empty
      pTail->pNext = pPackage;                      // Add the new object to the tail
   else                                             // List is empty
      pHead = pPackage;                             // so new object is the head
```

```
    pTail = pPackage;                          // Store its address as tail
}

// Subscript operator
Box TruckLoad::operator[](int index) const {
    if(index<0) {                                        // Check for negative index
        cout  << endl <<  "Negative index";
        exit(1);
    }

    Package* pPackage = pHead;                           // Address of first Package
    int count = 0;                                       // Package count
    do {
        if(index == count++)                             // Up to index yet?
            return *pPackage->pBox;                      // If so return the Box
    } while(pPackage = pPackage->pNext);

    cout  << endl <<  "Out of range index";    // If we get to here index is too high
    exit(1);
}
```

注意 TruckLoad 类析构函数的输出已删除，因为已不再需要它们。

为了试验下标运算符，下面创建 Box 对象的一个链表(如程序示例 13.1 所示)，然后使用下标运算符查找最大的对象：

```
// Program 14.5 Using the overloaded subscript operator   File: prog14_05.cpp
#include <iostream>
#include <cstdlib>                    // For random number generator
#include <ctime>                      // For time function
using std::cout;
using std::endl;

#include "Box.h"
#include "List.h"

// Function to generate random integers from 1 to count
inline int random(int count) {
   return 1 + static_cast<int>
              ((count*static_cast<double>(std::rand()))/ (RAND_MAX + 1.0));
}

// Display box dimensions
void show(const Box& aBox) {
    cout << endl
         << aBox.getLength()  << "by" << aBox.getWidth()
                              << "by" << aBox.getHeight();
}

int main() {
   const int dimLimit = 100;          // Upper limit on Box dimensions
   std::srand((unsigned)std::time(0)); // Initialize the random number
                                       // generator
```

```
const int boxCount = 20;          // Number of elements in Box array
Box boxes[boxCount];              // Array of Box objects

// Create 20 Box objects
for(int i = 0 ;  i < boxCount ;  i++)
   boxes[i] = Box(random(dimLimit), random(dimLimit), random(dimLimit));

TruckLoad load = TruckLoad(boxes, boxCount);

 // Find the largest Box in the list
Box maxBox = load[0];

 for(int i = 1  ;  i < boxCount  ;  i++)
   if(maxBox < load[i])
      maxBox = load[i];

 cout << endl
      <<  "The largest box in the list is ";
 show(maxBox);
 cout << endl;
 return 0;
}
```

运行这个例子，结果如下：

```
The largest box in the list is
90 by 79 by 77
```

例子的说明

main()函数现在使用下标运算符从链表中提取 Box 对象。下面的语句提取了列表中的第一个 Box 对象：

```
Box maxBox=load[0];
```

这个语句等价于：

```
Box maxBox=load.operator[](0);
```

通过下标运算符，可以使用索引迭代列表中的其他对象，以查找最大的 Box 对象：

```
for(int i=1; i<boxCount; i++)
  if(maxBox<load[i])
    maxBox=load[i];
```

再使用重载的<运算符来比较当前最大的对象 maxBox 与列表中索引位置为 i 的对象 load[i]。这不是一个非常高效的过程，因为在每次引用列表中的对象时，重载的下标运算符都必须迭代列表。为了提高效率，可以跟踪上次记录的元素的索引位置，从而得到后续的元素。但是，这总是比迭代列表慢，只是看起来比较简明。

1. lvalue 和重载的下标运算符

在一些情况下，需要重载下标运算符，并把返回的对象用作 lvalue，即赋值语句左边的值。利用前面的实现代码，如果有如下语句，程序就会有错误：

```
load[0]= load[1];
```

问题是 operator[]()的返回值。返回的对象是一个由编译器创建的临时对象，如果在赋值语句的左边使用它，就会出问题。赋值语句会运行，但不会改变列表中的第一个对象，只是改变它的一个副本，从长远来看，这并不好。当然，下次使用表达式 load[0]时，会得到列表中第一个对象的另一个副本。要解决这个问题，可以重新定义该运算符，使之返回一个引用。这样返回值就可以用作 lvalue 了(显然，在这种情况下，不能返回本地对象的引用)。把下标运算符的定义改为：

```
Box&  TruckLoad::operator[](int index) const {
  if(index<0) {                                  // Check for negative index
     cout << endl << "Negative index";
     exit(1);
  }

  Package* pPackage = pHead;                     // Address of first Package
  int count = 0;                                 // Package count
  do {
    if(index == count++)                         // Up to index yet?
       return *pPackage->pBox;                   // If so return the Box
  } while(pPackage = pPackage->pNext);

  cout << endl << "Out of range index";      // If we get to here index is too high
  exit(1);
}
```

函数体中的代码不需要修改，只需要修改函数的签名。当然，TruckLoad 类定义中的声明必须也返回一个引用。现在就可以使用下面的语句了：

```
load[0]= load[1]+ load[2];
```

理解了这个语句后，下面看看在调用函数之后，这个语句会解释为什么。该语句等价于：

```
load. operator[](0)=( load.operator[](1)).operator+( load.operator[](2));
```

如果给 Box 类重载赋值运算符，这个语句可以进一步解释为：

```
load.operator[](0). operator=(( load.operator[](1)). operator+ (load. Operator
                                                                 [](2)));
```

2. const 上的重载

在某些情况下，不允许把 Box 对象用作可修改的 lvalue。而有时又允许这么做。如何区分这两种情况？

一种方法是，在不允许修改 TruckLoad 对象包含的 Box 对象时，就使用 const 的 TruckLoad 对象；在允许修改 Box 对象时，就使用非 const 的 TruckLoad 对象。可以重载一个非 const 的

成员函数和一个 const 的成员函数，这样就有两个版本的下标运算符了。为 const 的 TruckLoad 对象调用 const 版本，为非 const 的 TruckLoad 对象调用非 const 版本。这就是 const 在应用于成员函数时的含义。非 const 版本的定义如前面的例子所示。下面是 const 版本的定义：

```
const Box& TruckLoad:: operator[](int index) const {
  //Body of the function the same as before…
}
```

函数体与前面相同，但返回类型是 const，所以在赋值语句的左边不能使用引用。其结果是为 const 的 TruckLoad 对象调用 operator[]()函数后，返回的 Box 对象不能用于赋值语句的左边。

14.1.10 重载类型转换

可以定义一个运算符函数，把类类型转换为另一种类型。要转换的类型可以是基本类型或类类型。转换任意类的对象 Object 的运算符函数的形式如下：

```
class Object {
  public:
    operator Type();          //Conversion from Object to Type
  //Rest of Object class definition…
};
```

其中 Type 是类对象的目标类型。注意这里没有指定返回类型。目标类型总是隐式的，所以函数必须返回 Type 类型的对象。

例如，定义一个运算符函数，把 TruckLoad 类型转换为 Box*类型，即 Box 数组的指针。在 TruckLoad 类中，该函数的声明如下所示：

```
class TruckLoad {
  public:
    operator Box*() const;
  //Rest of TruckLoad class definition…
};
```

返回类型 Box*在运算符函数中是隐式的，不必指定它。实现这个函数有一个问题。该函数需要在自由存储区中创建数组，所以调用程序必须负责删除它，以避免出现内存泄漏。

另一个方法是定义从 Box 类型向 double 类型的转换。根据应用程序，这个转换应得到转换过来的 Box 对象的体积。该转换的定义为：

```
class Box {
  public:
    operator double() const
      {return volume();}
  //Rest of Box class definition…
};
```

接着，给定一个 Box 对象 theBox，下面的语句会调用该运算符函数：

```
double boxVolume = theBox;
```

也可以用下面的语句显式调用运算符函数：

```
double total = 10 + static_cast<double>(theBox);
```

提示：

也可以用表达式 double(TheBox)，通过老式的强制转换进行这类转换。

转换的模糊性

注意在给类实现转换运算符时，可能出现模糊，导致编译错误。如前所述，构造函数也可以实现转换。在类 Type2 的构造函数中包含如下声明，就可以实现从 Type1 类型到 Type2 类型的转换：

```
Type2(const Type1& theObject);      //Constructor converting Type1 to Type2
```

如果在类 Type2 中实现一个转换运算符，就可以完成与这个构造函数相同的任务。该转换运算符的声明如下：

```
operator Type1();      //Conversion from type Type1 to Type2
```

在需要转换时，编译器不知道该使用哪个函数。为了去除转换运算符和构造函数之间的这种模糊性，可以把构造函数声明为 explicit，如第 12 章所述。这会阻止把构造函数用于隐式的转换，这样编译器就会选择转换运算符。

提示：

转换运算符和构造函数之间的这种模糊性说明，转换运算符在实现时要特别小心。

14.1.11 重载递增和递减运算符

++和--运算符在重载它们时有一个新问题，因为它们放在操作数之前和之后的情况是不一样的。因此每个运算符都需要两个函数：一个在运算符放在操作数之前时调用，另一个则是运算符放在操作数之后时调用。

当参数的类型是 int 时，运算符函数的后缀形式与其前缀形式是不同的。这个参数仅用于区分这两种情况，不用于其他情形。对任意类 Object 重载++的函数声明如下：

```
class Object {
  public:
    Object& operator++() ;                  //Overloaded prefix increment operator
    const Object operator++(int) ;          //Overloaded postfix increment operator
  //Rest of Object class definition…
};
```

前缀形式的返回类型一般要求是当前对象递增后的引用。

对于后缀形式，先在修改之前创建原对象的副本，再返回执行递增后的原对象的副本。后缀运算符的返回值声明为 const，可阻止编译 theObject++++这样的表达式。这种表达式会与运算符的一般形式混淆，产生不一致。但是，如果不把返回类型声明为 const，这种用法就是允许的。

注意：

对于这些重载运算符的任何类实现，前缀形式的返回类型都总是当前对象的引用，后缀形式的返回类型则是同一类型的新对象，该对象是递增操作执行之前的原对象的副本。

14.1.12 智能指针

由于可以重载解除引用运算符*和间接成员访问运算符->，就可以定义一个表示智能指针的类型，它类似于一个指针，但实际上是一个类对象。标准库广泛使用了智能指针的形式，称为类迭代器，详见第 20 章。但智能指针主要用于什么场合？

智能指针是一个对象，它可以用作以复杂方式组织起来的对象的智能指针。它可以隐藏组织对象的复杂性，递增或递减智能指针时，可以从一个对象移动到下一个对象或前一个对象上。因此在解除其引用时，智能指针可以返回它当前指向的对象，或通过->运算符访问其成员，就像一个普通的指针一样。

每个智能指针类都能适应它的环境。例如，它可以指向存储在某种容器类中的对象，该容器类以复杂的结构存储对象，例如树或列表。还可以使用智能指针访问或遍历这些对象，而不必理会其内部组织。另外，通过智能指针访问的对象甚至可以不存储在一个地方。在非常复杂的应用程序环境中，对象通过网络上的多个系统存储在文件或数据库中，但在程序中，使用智能指针对象就可以遍历这些对象，提取出需要的对象。从程序代码的角度来看，本地对象看起来与远程对象没有什么区别。访问智能指针指向的对象时，其复杂性都隐藏到定义该对象的类中。在外部，可以把智能指针当做一般指针来使用。

可以创建一个类，来定义智能指针。下面用 TruckLoad 类来访问它包含的 Box 对象。这是智能指针的一个简单实现，其中还使用了重载运算符，并演示了递增运算符的重载。

1. 为 Box 对象定义智能指针类

智能指针工作起来应与一般的 Box 指针一样，但它应指向 TruckLoad 对象中的 Box 对象。该类命名为 BoxPtr。从 TruckLoad 对象的引用中创建一个 BoxPtr 对象，使它自动指向链表中的第一个 Box 对象。这样，BoxPtr 对象就需要跟踪 TruckLoad 对象中当前的 Box 对象和 TruckLoad 对象本身。把这些放在 BoxPtr 类的定义中，而 BoxPtr 类的定义放在头文件 BoxPtr.h 中：

```
#ifndef BOXPTR_H
#define BOXPTR_H
#include "List.h"

class BoxPtr {
 public:
   BoxPtr(TruckLoad& load);                    // Constructor

 private:
   Box* pBox;                                  // Points to current Box in rLoad
   TruckLoad& rLoad;

   // Not accessible so not implemented
   BoxPtr();                                   // Default constructor
   BoxPtr(BoxPtr&);                            // Copy constructor
```

```
    BoxPtr& operator=(const BoxPtr&);        // Assignment operator
};
#endif
```

只允许从 TruckLoad 对象中创建 BoxPtr 对象，是为了确保在执行 BoxPtr 对象的函数时，TruckLoad 对象存在，但它可能是空的。不允许使用默认的构造函数来创建 BoxPtr 对象，所以把它声明为类的一个私有成员。拥有 BoxPtr 对象的副本也会使事情变得复杂起来，但可以在类的私有部分声明副本构造函数和赋值运算符，来避免出现 BoxPtr 对象的副本。

不需要修改 TruckLoad 类和 Box 类。可以使用程序示例 14.5 中的版本。如果希望 BoxPtr 对象可以像指针那样使用，就需要对 BoxPtr 类实现*和->运算符。下面在定义中添加这两个运算符：

```
class BoxPtr {
  public:
    BoxPtr(TruckLoad& load);                  // Constructor
    Box& operator*() const;                   // * overload
    Box* operator->() const;                  // -> overload

  private:
    Box* pBox;                                // Points to current Box in rLoad
    TruckLoad& rLoad;

    // Not accessible so not implemented
    BoxPtr();                                 // Default constructor
    BoxPtr(BoxPtr&);                          // Copy constructor
    BoxPtr& operator=(const BoxPtr&);         // Assignment operator
};
```

由于 BoxPtr 对象的操作与 Box*指针非常类似，因此解除 BoxPtr 对象的引用就应返回对 Box 对象的引用，这就是返回类型是 Box&的原因。*运算符是一元运算符，所以该函数没有参数。

operator->()函数有点古怪，下面看看它是如何工作的。它在下面的语句中调用：

```
BoxPtr pLoadBox(aTruckLoad);
double boxVol= pLoadBox->volume();
```

其中，aTruckLoad 是一个 TruckLoad 对象。由于 pLoadBox 表示 aTruckLoad 中第一个 Box 对象的指针，因此第二个语句应调用该对象的 volume()成员。第二个语句等价于：

```
double boxVol= (pLoadBox.operator ->())->volume();
```

插入一个额外的->运算符，用于访问成员 volume()，所以 operator ->()函数必须返回 Box* 类型的指针。

如果要递增 BoxPtr 对象，则表达式++ pLoadBox 就指向 TruckLoad 对象中的下一个 Box 对象。给 BoxPtr 类添加前缀形式和后缀形式：

```
class BoxPtr {
  public:
    BoxPtr(TruckLoad& load);                  // Constructor
```

```
    Box& operator*() const;                    // * overload
    Box* operator->() const;                   // -> overload
    Box* operator++();                         // Prefix increment
    const Box* operator++(int);                // Postfix increment

private:
    Box* pBox;                                 // Points to current Box in rLoad
    TruckLoad& rLoad;

    // Not accessible so not implemented
    BoxPtr();                                  // Default constructor
    BoxPtr(BoxPtr&);                           // Copy constructor
    BoxPtr& operator=(const BoxPtr&);          // Assignment operator
};
```

当然，这些函数都返回 Box*类型，因为递增的是指针的替代品。后缀形式返回一个 const 对象，以防止重复应用该运算符，如 pLoadBox++++。

还需要使用 BoxPtr 对象控制循环，以便编写下面的语句：

```
if(pLoadBox) {
  //Do something...
}
```

这里该对象应像一个正常的指针，并把它自动转换为在 if 测试表达式中有效的值。一种方法是对 BoxPtr 类型的对象实现到 bool 的转换。这样，在需要返回 bool 类型的逻辑表达式中使用 BoxPtr 时，编译器就会自动插入转换运算符。给 BoxPtr 类添加如下声明：

```
class BoxPtr {
  public:
    BoxPtr(TruckLoad& load);                   // Constructor
    Box& operator*() const;                        // * overload
    Box* operator->() const;                       // -> overload
    Box* operator++();                         // Prefix increment
    const Box* operator++(int);                // Postfix increment
    operator bool();                           // Conversion to bool

  private:
    Box* pBox;                                 // Points to current Box in rLoad
    TruckLoad& rLoad;

    // Not accessible so not implemented
    BoxPtr();                                  // Default constructor
    BoxPtr(BoxPtr&);                           // Copy constructor
    BoxPtr& operator=(const BoxPtr&);          // Assignment operator
};
```

注意这里不需要返回类型，也不允许有返回类型，因为这是转换运算符函数的性质。还可以采用其他方式，让对象在 if 测试表达式中使用。在流输入/输出的标准库类中就会遇到这种情况，详见第 19 章。

这就是智能指针的用法。下面看看如何使用 BoxPtr 类的成员函数定义。

2. 实现智能指针类

构造函数是非常简单的。只需初始化 TruckLoad 对象的引用成员，存储它包含的第一个 Box 对象地址即可。把这个定义放在源文件 BoxPtr.cpp 中：

```
// BoxPtr.cpp
#include <iostream>
#include "List.h"
#include "BoxPtr.h"
using std::cout;
using std::endl;

BoxPtr::BoxPtr(TruckLoad& load) : rLoad(load) {
   pBox = rLoad.getFirstBox();
}
```

rLoad 成员必须在构造函数的初始化列表中进行初始化，这是初始化引用成员的惟一方式。为了获取第一个 Box 对象的地址，调用 getFirstBox()成员，并把该对象传送为一个参数。添加<iostream>的#include 指令和 using 声明，用于下一个要添加到文件中的函数定义。

解除引用运算符函数应返回由 BoxPtr 对象的 pBox 成员指向的对象，还必须考虑没有对象的情况。该函数的定义如下所示：

```
Box& BoxPtr::operator*() {
 if(pBox)
   return *pBox;
 else {
   cout << endl << "Dereferencing null BoxPtr";
   exit(1);
 }
}
```

如果 pBox 是空，就不能以正常的方式返回一个对象。解除空指针的引用会出现灾难性问题，因此应显示一个消息，并结束程序。这样做并不是处理该问题的最佳方式。第 17 章将介绍一种更好的方式。

注释：

编译器可能不支持 exit()的这种用法。此时，应使用语句 return *pBox;替换语句 exit(1)；进行修复。

间接成员选择运算符更简单。只需返回 pBox 成员：

```
Box* BoxPtr::operator->() {
   return pBox;
}
```

这会返回 pBox 中包含的地址，当 pBox 为空时，返回值就是空。所以，BoxPtr 类的用户在使用之前，需要验证智能指针不为空—— 就像使用一般指针一样。

BoxPtr 对象的前缀递增运算符也返回一个指针，就像递增一般的指针一样：

```
Box* BoxPtr::operator++() {
```

```
    return pBox = rLoad.getNextBox();
}
```

前缀运算符在执行表达式之前进行递增，返回 TruckLoad 对象中下一个指针的地址，这是由 getNestBox()成员返回的。如果递增超出了最后一个 Box 对象，就返回空。

递增运算符的后缀形式必须在表达式中使用当前值后，再递增智能指针。这听起来很困难，实际上并非如此。它的实现代码如下所示：

```
const Box* BoxPtr::operator++(int) {
   Box* pTemp = pBox;
   pBox = rLoad.getNextBox();
   return pTemp;
}
```

在 pBox 中保存当前地址的副本，再递增 pBox，使之指向下一个 Box 对象，然后返回原地址。

最后一个要定义的成员函数是转换运算符。这也很简单：

```
BoxPtr::operator bool() {
   return pBox !=0;
}
```

如果 pBox 不为空，就返回 true，如 return 中的表达式所示。

下面运行另一个例子，确保这些运算符都有效。

程序示例 14.6—— 使用智能指针

下面创建一个 TruckLoad 对象，其中包含一组随机的 Box 对象，以验证智能指针类。接着试验运算符函数。下面是代码：

```
// Program 14.6 Using a smart pointer
#include <iostream>
#include <cstdlib>                  // For random number generator
#include <ctime>                    // For time function
using std::cout;
using std::endl;

#include "Box.h"
#include "List.h"
#include "BoxPtr.h"

// Function to generate random integers from 1 to count
inline int random(int count) {
   return 1 + static_cast<int>
              (count*static_cast<double>(std::rand())/(RAND_MAX + 1.0));
}

int main ( ) {
   const int dimLimit = 100;        // Upper limit on Box dimensions
   std::srand((unsigned)std::time(0));   // Initialize the random number generator

   const int boxCount = 20;         // Number of elements in Box array
   Box boxes[boxCount];             // Array of Box objects
```

```
// Create 20 Box objects
for(int i = 0 ; i < boxCount ; i++)
  boxes[i] = Box(random(dimLimit), random(dimLimit), random(dimLimit));

TruckLoad load = TruckLoad(boxes, boxCount);

// Find the largest Box in the list
BoxPtr pLoadBox(load);          // Create smart pointer

Box maxBox = *pLoadBox;         // Intialize maxBox object using * operator
if(pLoadBox)                    // Try the bool conversion
   cout << endl << "Volume of first Box is" << pLoadBox->volume(); // and ->

while(++pLoadBox)               // Prefix increment smart pointer
    if(maxBox < *pLoadBox)
        maxBox  =  *pLoadBox;

cout << endl
     << "The largest box in the list is"
     << maxBox.getLength()   <<  "by"
     << maxBox.getWidth()    <<  "by"
     << maxBox.getHeight()   <<  "with volume"
     << maxBox.volume()      << endl;
return 0;
}
```

这个例子的运行结果如下：

```
Volume of first Box is 110880
The largest box in the list is 100 by 74 by 91 with volume 673400
```

例子的说明

下面讨论这个例子是如何应用运算符函数的。建立一个 TruckLoad 对象，其中包含 20 个 Box 对象，之后创建一个智能指针，语句如下：

```
BoxPtr pLoadBox(load);          //Create smart pointer
```

在创建该指针时，pLoadBox 对象是 TruckLoad 对象 load 中第一个 Box 对象指针的代用品。解除它的引用，初始化 maxBox 对象：

```
Box maxBox=*pLoadBox;           //Initialize maxBox object using * operator
```

这个语句调用 Box 类的默认构造函数创建 maxBox。然后调用 pLoadBox 对象的 operator*()函数，使用返回的 Box 对象引用，作为 Box 对象 maxBox 的 operator=()函数参数。

接着，if 语句调用 BoxPtr 类的两个成员函数：

```
if(pLoadBox)            //Try the bool conversion
  cout<<endl<<"Volume of first Box is "<< pLoadBox->volume();     // and ->
```

if 表达式隐式调用了 operator bool()函数，输出语句调用了 operator->()函数，返回当前 Box 对象的地址。

while 循环在循环控制表达式中使用了前缀递增运算符：

```
while(++pLoadBox)              //Prefix increment smart pointer
  if(maxBox < *pLoadBox)
    maxBox = *pLoadBox;
```

operator++()函数返回下一个 Box 对象的指针。如果它不为空，就使用 operator*()函数解除对它的引用，再调用 maxBox 对象的 operator<()成员来比较体积。如果最后一个 Box 对象的体积较大，就把它存储在 maxBox 中——使用 pLoadBox 对象类的 operator*()函数和 maxBox 对象的 operator=()函数。

在循环的最后，pLoadBox 表示一个空指针，不能再使用。最后以通常的方式输出 maxBox 对象的信息。

14.1.13 重载运算符 new 和 delete

可以给类重载运算符 new 和 delete，而且，如果重载了 new，就必须重载 delete。重载这些运算符的原因一般是，对某个类的对象，使内存的分配和释放更快、更经济。当需要为巨量的对象分配内存空间，而且每个对象都需要少量的内存时，就应重载运算符 new 和 delete。一次为大量较小的对象分配内存时，实际上给每个对象分配的内存量会比较大，分配和释放内存的时间都比较长。

实现 new 的标准方法是使用默认的 new 运算符分配一大块内存，再按照需要把它分为许多小块。显然，类 delete 操作必须实现，以释放这些小块的内存。针对类的 new 和 delete 的一般声明如下所示：

```
class Data {
  public:
    void* operator new(size_t Size);
    void operator delete(void* Object,  size_t  size);
  // Rest of Data class definition ...
};
```

要在针对类的运算符函数中调用全局运算符，可以使用作用域解析运算符::。例如，要从针对类的 new 运算符中调用全局 new 运算符，就可以使用下面的语句：

```
void* operator new(size_t size) {
  //  ...
  pSpace =  ::new char(size);          //  size bytes allocated by global new
  //  ...
}
```

实现这些运算符并不是很必要，因为这会使管理内存不足和其他错误复杂化。除非绝对必要，最好不要重载这些运算符。因为很少需要重载这些运算符，后面就不再探讨它们了。

14.2 本章小结

本章学习了如何添加函数，使自己建立的数据类型的对象可以利用 C++中的基本运算符。

在类中，需要实现哪些运算符函数取决于应用程序的需要。必须确定每个类应提供的功能的性质和范围。类是自己定义的一个数据类型，是一个集成的实体，类需要反映它的本质和特性。还要确保重载运算符的实现不会与标准形式的运算符发生冲突。

本章的要点如下：

- 在类中可以重载任何运算符，以提供针对该类的功能。但作用域解析运算符(::)、条件运算符(?:)、成员访问运算符(.)、解除类成员指针的引用运算符(.*)和 sizeof 运算符不能重载。
- 运算符函数可以定义为类的成员或全局运算符函数。
- 如果一元运算符定义为类的成员函数，操作数就是类对象。
- 如果一元运算符定义为全局运算符函数，操作数就是函数的参数。
- 如果二元运算符定义为类的成员函数，左操作数就是类对象，右操作数就是函数的参数。
- 如果二元运算符定义为全局运算符函数，第一个参数指定左操作数，第二个参数指定右操作数。
- 要重载递增运算符，需要用两个函数分别提供运算符的前缀和后缀形式。实现后缀运算符的函数有一个 int 类型的额外参数，它仅用于与前缀函数区分。递减运算符也是这样。
- 实现+=运算符重载的函数可以用在+函数的实现上。所有 op=运算符都是这样。
- 智能指针是一个操作类似于指针的对象。智能指针的一种形式是迭代给定类型的对象的复杂集合，采用的方式与一般指针类似。标准模板库广泛使用了这种形式。

14.3 练习

1. 这些练习都建立在第 13 章练习的基础之上，首先为 MyString 类提供一个重载的赋值运算符。确保它不是自我赋值。用下面的语句测试这个运算符是否正常工作，其中 s1、s2 和 s3 都是 MyString 对象：

```
s1=s2;
s1=s1;
s1=s2=s3;
```

2. 重载+运算符，提供字符串连接功能。测试 s1=s2+s3；语句正确运行。提供+=运算符，这个运算符应返回什么值？

3. 重载[]，提供对字符串中单个字符的访问。于是，s1[4]返回 s1 中的第 5 个字符。如何确保它可以用于等号的两端？

4. 提供==、!=、<和>运算符的重载，用于比较 MyString 对象。这些布尔运算符应返回什么类型？检查表达式 if(s1==s2)是否工作正确？

5. (较难)重载()运算符，从 MyString 对象中返回一个子字符串，于是 s1(2,3)返回从 s1[2]开始的三个字符。

第 15 章　继　　承

本章讨论面向对象编程的一个核心主题—— 继承。通过继承，可以通过重用并扩展已有的类定义，创建新类。

继承也是实现多态性的基础。多态性是面向对象编程的核心，详见第 16 章，所以第 16 章介绍的也是继承的组成部分。

本章主要内容

- 继承如何应用于面向对象编程
- 基类和派生类的定义及其关系
- 根据已有的类定义新类
- 使用关键字 protected 为类成员定义新的访问规范
- 构造函数在派生类中如何工作，在调用构造函数时会发生什么
- 在使用派生类时，析构函数会发生什么
- 多重继承及其工作原理
- 在类体系结构中，类类型之间的转换

15.1　类和面向对象编程

首先了解如何从已经学习的内容延伸到本章所讨论的主题。

第 12 章介绍了类的概念。类是一种自己定义的数据类型，用于满足应用程序的需求。在使用面向对象的编程方式解决问题时，第一步是定义与问题相关的实体类型，根据问题的解决方案确定每个类型的特性和操作，然后编写类定义，对这些类型编码。最后，根据对象，使用直接利用这些对象的操作编写解决问题的方案。

任何类型的实体都可以用类来表示，从完全抽象的实体(如复数的数学概念)到很具体的实体(如树或卡车)，都可以用类表示。类的定义应反映一组实体的特性，这些实体用一组共同的属性来标识。所以，除了数据类型之外，类还是一个实际对象的定义(或者，至少是一个近似的对象，可以用于解决给定的问题)。

在许多真实的问题中，所涉及的实体的类型都是相关的。例如，狗是一种动物，它具有动物的所有属性，还有一些自己的属性。因此，Animal 和 Dog 的类定义就应以某种方式相关。毕竟，狗是一种动物，所以类定义应以某种方式反映这一点。另一种不同的情形是，汽车有引擎，所以在定义 Automobile 类时，就应以某种方式使用 Engine 类。但引擎肯定不是汽车，所以它们的关系应与 Animal 和 Dog 的关系不同。本章介绍这两种不同的关系如何在 C++中实现。

层次结构

前面的章节使用 Box 类来描述矩形的盒子。Box 对象的定义仅由三个正交尺寸组成。这个

基本的定义可以应用于现实中许多不同类型的矩形盒子—— 硬纸盒、木箱、糖果盒和谷物箱等。这些对象都有三个正交尺寸，在这个方面它们类似于一般的 Box 对象。另外，每个对象又都有其他属性，例如，可以装在这些对象中的物品，或制作它们的材料。实际上，可以把它们描述为 Box 对象的特殊类型。

例如，下面描述 Carton 类。它具有与 Box 对象相同的属性，即三个尺寸，还有另一个属性，即它的合成材料。再进一步，就使用 Carton 定义描述 FoodCarton 类，这是一种特殊的 Carton 类，它专门用于储存食物。它具有 Carton 对象的所有属性，还有一个额外的成员，即存储预定的物品。

类层次结构的关系如图 15-1 所示。

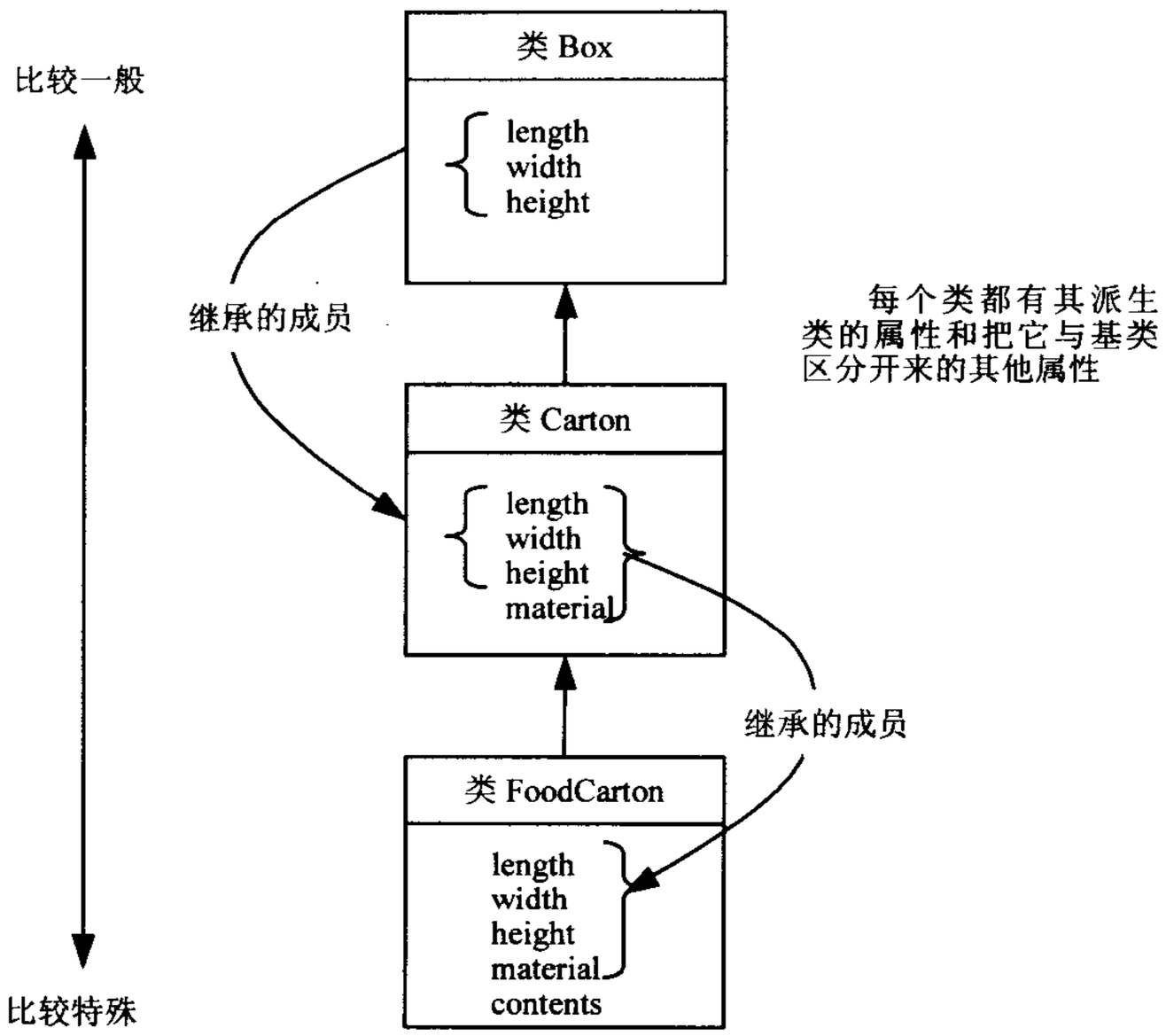


图 15-1　层次结构中的类

Carton 类是 Box 类的一个扩展—— Carton 类是从 Box 类的规范派生而来。FoodCarton 类又以类似的方式从 Carton 类派生而来。我们常常用一个箭头图形化地表示这种关系，在层次结构中，箭头从比较一般的类指向较特殊的类，图 15-1 就使用了这种箭头。

按照这个过程，可以开发出一个相互关联的类层次结构。在该层次结构中，一个类派生自另一个类，并添加了其他属性—— 换言之，就是特殊化，使新类成为较一般类的特殊版本。在图 15-1 中，每个类都有 Box 类的所有属性，这很好地演示了 C++中类的继承机制。Box、Carton 和 FoodCarton 类的定义可以是相互独立的，但把它们定义为相互关联的类会得到更好的回报。下面看看其工作原理。

15.2　类的继承

首先，解释一下相关类中要使用的术语。假定有一个类 A，要创建一个新类 B，它是类 A 的一个特殊版本。类 A 就称为基类，类 B 则称为派生类。类 A 是父，类 B 就是子。派生类自

动包含基类的所有数据成员和所有函数成员 (但有一些限制)。这称为派生类继承了基类的数据成员和函数成员。

如果类 B 是直接派生自类 A 的派生类，则类 A 就称为 B 的直接基类，B 派生自 A。在上面的例子中，类 Carton 是 FoodCarton 的直接基类。因为 Carton 类本身又是根据 Box 类定义的，所以 Box 类是 FoodCarton 的间接基类。FoodCarton 类的对象继承了 Carton 的成员，包括 Carton 类从 Box 中继承的成员。

派生类从基类中继承成员的过程如图 15-2 所示。

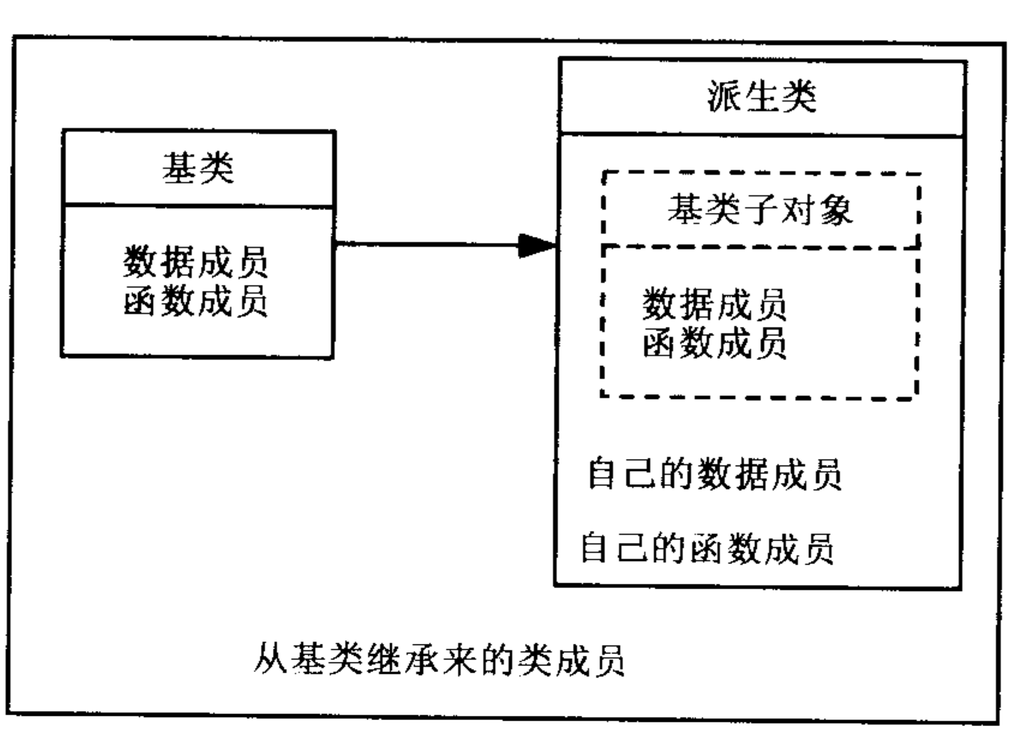


图 15-2　继承基类的派生类成员

可以看出，派生类拥有基类所有的数据成员和成员函数，还拥有自己的数据成员和成员函数。因此，每个派生类对象都包含一个完整的基类子对象，以及其他成员。

15.2.1　继承和聚合

类继承的过程不仅是把一个类的成员显示在另一个类中的方式，支撑这个概念还有一个重要的方面：派生类对象应代表有意义的基类对象。

为了说明这一点，下面进行一些简单的测试。首先是种类的测试：任何派生类对象都是基类类型的对象。换言之，派生类应描述基类所表示的对象的一个子集。例如，类 Dog 派生于类 Animal。这是有意义的，因为狗是一种动物，或者，Dog 对象是某种 Animal 对象的有意义表示。另一方面，Table 类不应派生于 Dog 类，尽管 Table 对象和 Dog 对象都有四条腿，但 Table 对象不能看作是 Dog 对象的某个特殊种类。

种类测试是极佳的第一个检查，但这不是非常可靠的。例如，假定定义一个类 Bird，它反映了大多数鸟类都可以飞翔这一事实。而鸵鸟是一种鸟类，但从 Bird 类中派生 Ostrich 类是没有意义的，因为鸵鸟不会飞。如果类通过了种类测试，就应询问下面的问题，进行第二个测试：基类中是否有特性不能应用于派生类？如果有，这种派生就是不安全的。因此 Dog 派生于 Animal 是有意义的，但 Ostrich 派生于 Bird 就没有意义—— 相反，应以更一般的方式定义 Bird，从该类中派生 Flighted_Birds 类和 Flightless_Birds 类。

如果类没有通过种类测试，就不能使用类的派生机制。在这种情况下，就应进行包含测试。如果类对象包含另一个类的实例，就通过了包含测试。当第二个类的对象被包含为第一个类的数据成员时，就要进行包含测试。这种依赖性称为集合。

例如，考虑类 Automobile。这个类把主要的汽车部件作为它的类成员。显然，汽车有一个引擎、传动装置、底盘和悬挂装置，所以 Automobile 类应包含 Engine、Transmission、Chassis 和 Suspension 类型的对象。注意，引擎是一种汽车这种说法是不对的。因此，这些类都没有通过种类测试，Automobile 继承自 Engine 就是没有意义的。

三种测试总结如下：

- 种类测试是第一步，检查继承是否是实现类的一种合适方式。
- 如果类通过了种类测试，就要询问下面的问题，进行第二个测试：基类中是否有特性不能应用于派生类？如果没有，继承通常就是可行的。
- 如果类没有通过种类测试，就试试包含测试。如果它们通过了包含测试，就应使用聚合。

当然，采用什么实现方式取决于应用程序，这些规则只是指导性的，而不是圣经。有时，类的派生机制可以用于装配一组功能，所派生出来的类是包装了给定函数聚合的封包。派生类一般表示以某种方式相关的一组函数。

下面看看从一个类中派生另一个类的代码。

15.2.2 从基类中派生新类

下面列出 Box 类的简化版本，其中有三个私有数据成员和一个公共构造函数：

```
// Box.h - defines Box class
#ifndef BOX_H
#define BOX_H
class Box {
  public:
    Box(double lv=1.0, double wv=1.0, double hv=1.0);

  private:
    double length;
    double width;
    double height;
};
#endif
```

因为在构造函数中为参数指定了默认值，所以该构造函数也称为默认构造函数。把这个定义保存在头文件 Box.h 中，并用#include 指令添加到例子中。为了保持一致，Box 构造函数的定义包含在文件 Box.cpp 中：

```
//  Box.cpp
#include  "box.h"

// Constructor
Box::Box(double lv, double wv, double hv): length(lv), width(wv), height(hv)
{}
```

现在，定义另一个类 Carton。Carton 对象如前面所述，类似于 Box 对象，但有一个额外的数据成员，用于表示对象的合成材料。把这个新数据成员声明为非空字符串，以描述制作盒子

的材料(最好使用 string 对象来表示材料，这里使用非空字符串是为了说明动态分配内存的一些特性)。把 Carton 定义为派生类，并把 Box 类用作其基类：

```
// Carton.h - defines the Carton class with the Box class as base
#ifndef CARTON_H
#define CARTON_H
#include "Box.h"                          // For Box class definition

class Carton : public Box {
  public:
    Carton(const char* pStr = "Cardboard");       // Constructor

    ~Carton();                                    // Destructor

  private:
    char* pMaterial;
};
#endif
```

必须把 Box 类定义包含在这个文件中，因为它是 Carton 的基类。文件 Carton.cpp 中不包含 Box 类的定义，但包含 Carton 类的定义：

```
// Carton.cpp
#include "Carton.h"
#include <cstring>

// Constructor
Carton::Carton(const char* pStr) {
  pMaterial = new char[strlen(pStr) +1] ;    // Allocate space for the string
  std::strcpy( pMaterial, pStr) ;            // Copy it
}

// Destructor
Carton::~Carton () {
  delete[] pMaterial;
}
```

Carton 类定义的第一行表示 Carton 类直接派生于 Box：

```
class Carton : public Box
```

关键字 public 是基类访问指定符。它表示 Box 的成员如何在 Carton 类中访问。稍后详细讨论它。

在其他方面，Carton 类定义与其他类定义相同。其中包含一个新成员 pMaterial，这是一个指向非空字符串的指针。它由类的构造函数初始化为在自由存储区中创建的字符串的地址。这将确保对象有一个描述材料的字符串。还必须提供一个析构函数，以释放 pMaterial 指向的字符串对象所占用的内存。注意构造函数包含字符串的默认值，该默认值描述了 Carton 对象的内容，所以这也是 Carton 类的默认构造函数。Carton 类的对象包含基类 Box 中的所有数据成员，还包含附加的数据成员 pMaterial。

下面看看这个类在例子中如何工作。

程序示例 15.1—— 使用派生类

下面是使用派生类的代码：

```
// Program 15.1 Defining and using a derived class
#include <iostream>
#include "Box.h"                                    // For the Box class
#include "Carton.h"                                 // For the Carton class
using std::cout;
using std::endl;

int main() {
  // Create a Box and two Carton objects
  Box myBox(40.0, 30.0, 20.0);
  Carton myCarton;
  Carton candyCarton("Thin cardboard");

  // Check them out - sizes first of all
  cout << endl
       << "myBox occupies "       << sizeof myBox       << " bytes" << endl;
  cout << "myCarton occupies "    << sizeof myCarton    << " bytes" <<endl;
  cout << "candyCarton occupies " << sizeof candyCarton << " bytes" <<endl;

// myBox.length =10.0;                          // uncomment this for an error
// candyCarton.length =10.0;                    // uncomment this for an error

  return 0;
}
```

例子的运行结果如下：

```
myBox occupies 24 bytes
myCarton occupies 32 bytes
candyCarton occupies 32 bytes
```

例子的说明

首先，把包含 Box 类和 Carton 类定义的头文件包含在内。当然，包含 Carton 类定义的文件也包含 Box.h，但#ifndef/#endif 预处理器指令可以防止 Box 类定义包含两次。

在 main()中声明一个 Box 对象和两个 Carton 对象，再输出每个对象占用的字节数。输出显示了我们期望的结果。Carton 对象占用的字节数比 Box 对象大。Box 对象有三个 double 类型的数据成员，这些数据成员都占用 8 个字节，所以总共占用 24 个字节。而两个 Carton 对象占用的字节数相同：32 个字节。每个 Carton 对象占用的额外内存是由于 Carton 对象多了一个数据成员 pMaterial(字符串的长度不会影响 Carton 对象占用的字节数，因为 pMaterial 是一个指针)。存储字符串的内存空间位于自由存储区上。

如果去掉下面两个语句的注释，程序就不会编译。第一个语句是：

```
// myBox.length=10.0;              //uncomment this for an error
```

当然，我们希望这个语句产生错误。Box 对象 myBox 的 length 成员被声明为一个私有数据成员，不能在这个输出语句中访问。去掉下面语句的注释，就可以看到，也不能访问 Carton

对象的 length 成员：

```
// candyCarton.length=10.0;     //uncomment this for an error
```

编译器会生成一个消息：不能访问基类的 length 成员。这也许有点意外，毕竟，在定义 Carton 类时，Box 类成员都能继承为 public 吗？

错误的原因是：length 在基类中是一个私有数据成员。在派生类 Carton 中，length 成员是公共继承的私有数据成员。编译器会把这个解释为：length 是 Carton 的一个私有成员。因此，在试图访问成员 candyCarton.length 时，编译器会产生错误。

提示：

如果要访问这些私有数据成员，就可以使用公共函数成员如 getLength()，如第 12 章所述。

访问派生类对象的继承成员取决于基类中数据成员的访问指定符和派生类中基类的访问指定符。

在派生类中访问继承成员的问题需要详细论述。稍后介绍第三个访问指定符 protected。下面再介绍几个例子，之后将总结基类访问指定符如何影响继承类成员的访问级别。

15.3 继承下的访问控制

基类的私有数据成员也是派生类的成员，但它们在派生类中仍是私有的。也就是说，继承的数据成员可以由从基类继承而来的函数成员访问，但不能由在派生类定义中声明的成员函数访问。

例如，在派生类 Carton 中添加一个函数 volume()。文件 Carton.h 就更新为：

```
//Carton.h-defines the Carton class with the Box class as base
#ifndef CARTON_H
#define CARTON_H
#include "Box.h"                                    // For Box class definition

class Carton : public Box {
  public:
    Carton(const char* pStr = "Cardboard"); // Constructor

    ~Carton();                                      // Destructor

    double volume() const;                          // Error - members not accessible
    private:
      char* pMaterial;
};
#endif
```

文件 Carton.cpp 更新为：

```
// Carton.cpp
#include "Carton.h"
#include <cstring>
```

```
// Constructor
Carton::Carton(const char* pStr) {
  pMaterial = new char[strlen(pStr)+1] ;   // Allocate space for the string
  std::strcpy( pMaterial, pStr);           // Copy it
}

// Destructor
Carton::~Carton() {
  delete[] pMaterial;
}

// Function to calculate the volume of a Carton object
double Carton::volume() const {
  return length*width*height;
}
```

使用这个类和程序示例 15.1 中的 Box 类的任何程序都不会编译。Carton 类中的函数 volume() 试图访问基类中的私有成员，这是不合法的，即使它们是派生类的继承成员也不行。

但是，如果函数 volume()是一个基类成员，使用它就是合法的。如果把函数 volume()的定义放在 Box 基类的公共部分，则不仅程序会编译，而且可以使用该函数获得 Carton 对象的体积。下面讲述其工作原理。

程序示例 15.2—— 基类成员函数

首先修改 Box.h 中的类定义：

```
class Box {
  public:
    Box(double lv=1.0,double wv=1.0,double hv=1.0);

// Function to calculate the volume of a Box object
    double volume() const;
  private:
    double length;
    double width;
    double height;
};
```

把函数 volume()声明为 const，是因为它不修改类的任何数据成员，只是使用它们计算对象的体积。在文件 Box.cpp 中添加该函数定义：

```
// Box.cpp
#include "box.h"

// Constructor
Box::Box(double lv,double wv,double hv):length(lv),width(wv),height(hv)
{}

// Function to calculate the volume of a Box object
double Box::volume() const {
  return length*width*height;
}
```

对于这个程序，使用程序示例 15.1 中的文件 Carton.h 和 Carton.cpp。因此，Carton 类公开继承 Box 类，还有一个构造函数、一个析构函数和一个私有数据成员 pMaterial。

现在计算在程序示例 15.1 中创建的 Box 对象和 Carton 对象的体积，看看如何在派生类中使用基类函数：

```
// Program 15.2 Using a function inherited from a base class File: prog15_02.cpp
#include <iostream>
#include "Box.h"                    //  For the Box class
#include "Carton.h"                 //  For the carton class
using std::cout;
using std::endl;

int main() {
  // Create a Box and two Carton objects
  Box myBox(40.0,   30.0,   20.0);
  Carton myCarton;
  Carton candyCarton("Thin cardboard");

  cout << endl;
  cout << "myBox volume is "         << myBox.volume()          << endl;
  cout << "myCarton volume is "      << myCarton.volume()       << endl;
  cout << "candyCarton volume is " << candyCarton.volume()   << endl;
  return 0;
}
```

结果如下所示：

```
myBox volume is 24000
myCarton volume is 1
candyCarton volume is 1
```

例子的说明

这个例子演示了在 Box 类中定义的函数 volume()也能操作 Carton 类的对象。它可以访问基类的私有数据成员，但这些私有成员不能由派生类的函数访问。两个 Carton 对象的体积都是 1，说明在基类构造函数中为它们设置了默认的尺寸。

在创建 Carton 类的对象时，就会调用该类的构造函数，但显然也必须调用基类的构造函数，因为派生类的构造函数没有提供指令(实际上也不能提供)，来初始化从基类继承而来的数据成员。添加输出语句，跟踪构造函数的每次调用，就可以验证为 Carton 对象调用了基类的构造函数。对于 Box 对象，可以在构造函数体中添加如下语句：

```
cout<<"Box constructor"<<endl;
```

也可以在 Carton 类的构造函数中添加类似的语句。不要忘记包含 iostream 头文件，并在每个.cpp 文件中为 cout 和 endl 添加 using 语句。再次运行程序，结果如下：

```
Box constructor
Box constructor
Carton constructor
Box constructor
Carton constructor
```

```
myBox volume  is  24000
myCarton volume  is  1
candyCarton volume is  1
```

输出的第一行表示创建了 Box 对象 myBox。接下来的两行表示创建了 Carton 对象 myCarton。可以看出，要创建 Carton 对象，将先自动调用 Box 类的构造函数(即默认的构造函数)，初始化基类的成员，之后执行 Carton 类的构造函数。

本章后面将详细论述基类的构造函数，特别是要讨论如何更好地控制基类数据成员的初始化。

15.4 把类的成员声明为 protected

基类的私有成员只能由基类的函数成员访问，但这并不是很方便。在许多情况下，基类的成员也需要能在派生类中访问，但不受外界的干扰。C++提供了一种方式来实现这种访问操作。

类的成员除了可以使用 public 和 private 访问指定符之外，还可以把类的成员声明为 protected。在类中，关键字 protected 与 private 有相同的效果。声明为 protected 的类成员只能由类的成员函数、友元类和类的友元函数访问。这些受保护的类成员不能在类的外部访问，它们的操作与私有类成员一样。

受保护的成员和私有成员仅在派生类中有区别。声明为 protected 的基类成员可以在派生类的函数成员中访问，而基类的私有成员则不能。

重新定义 Box 类，使之包含受保护的数据成员，如下所示：

```
class Box {
  public:
    Box(double lv=1.0, double wv=1.0, double hv=1.0);

  protected:
    double length;
    double width;
    double height;
};
```

现在 Box 的数据成员仍是私有的，因为它们不能由一般的全局函数访问，但可以在派生类的成员函数中访问。下面看一个例子。

程序示例 15.3—— 使用受保护的继承成员

使用这个版本的 Box 类派生 Carton 类的新版本，该版本可以通过自己的成员函数 volume()访问基类的成员。

从 Box 类定义中删除 volume()函数原型(如上所示)，再删除 Box.cpp 中的 volume()函数定义。接着把 volume()函数原型添加到 Carton 类的定义中，把 volume()函数定义添加到 Carton.cpp 中。把 Carton 类的数据成员也声明为 protected，这样该类的规范就与从 Box 继承来的数据成员相同了。下面是类定义：

```
class Carton : public Box  {
```

```
  public :
    Carton(const char* pStr = "Cardboard");     // Constructor

    ~Carton ();                                 // Destructor

    // Function to calculate the volume of a Carton object
    double volume () const;

  protected:
    char* pMaterial;
};
```

下面是添加到 Carton.cpp 中的函数定义：

```
//  Function to calculate the volume of a Carton object
double Carton:: volume() const {
  return length*width*height;
}
```

在下面的程序中使用新的 Carton 类：

```
// Program 15.3 Using inherited protected members in a derived class
#include <iostream>
#include "Box.h"                    // For the Box class
#include "Carton. h"                // For the Carton class
using std::cout;
using std::endl;

int main() {
  // Create a Box and two Carton objects
  Box myBox(40.0, 30.0, 20.0);
  Carton myCarton;

  cout << endl;
  cout << "myCarton volume is " << myCarton.volume () << endl;
// Uncomment either of the following statement for error
//cout << "myBox volume is " <<  myBox. volume () << endl;
//cout << "myCarton length is " << myCarton. length << endl;

  return 0 ;
}
```

注释说明了 Box 和 Carton 类的新版本。这个程序的结果如下所示：

```
Box constructor
Box constructor
Carton constructor

myCarton volume  is  1
```

例子的说明

如果去掉计算 myBox 体积的输出语句的注释，程序就不会编译：

```
//cout << "myBox volume is " << myBox.volume() << endl;
```

这是因为 Box 类现在没有定义函数 volume()。目前只能计算 Carton 对象的体积。通过调用在派生类中定义的 volume()函数，计算 Carton 对象 myCarton 的体积：

```
cout << "myCarton volume is" << myCarton.volume() << endl;
```

这个函数访问继承的成员 length、width 和 height，并计算出结果。这些成员都在基类中声明为 protected，在派生类中也继承为 protected。在输出中，myCarton 的体积计算正确。其体积是 1，因为从 Box 继承来的成员被赋予了默认值。

提示：

由于在 Carton 类中 pMaterial 指定为 protected，该类的所有数据成员现在都是 protected，因此它们都可以在另一个派生自 Carton 的类中访问。

去掉下面语句的注释，就可以证明基类中的受保护成员在派生类中也是受保护的：

```
//cout << "myCarton length is" << myCarton.length <<endl;
```

此时，编译器会产生一个错误消息，因为成员 length 是受保护的，不能在派生类的外部访问。

15.5 派生类成员的访问级别

在从基类中派生一个类时，必须选择基类访问指定符。有三种基类访问指定符：public、protected 和 private。

注释：

实际上，默认的基类访问指定符是 private。如果省略指定符，例如在程序示例 15.1 中，Carton 类定义的第一个语句是 class Carton:Box，就使用 private 访问指定符。

对于基类的数据成员，其访问指定符也可以选择使用 public、protected 和 private 中的一个。

使用基类创建派生类时，基类访问指定符会影响继承成员的访问状态。有 9 种不同的组合：本章的例子将介绍其中的几个组合。下面将讨论所有可能的组合，但有一些组合仅在第 16 章讨论多态性时才能用得上。

首先，看看派生类如何继承基类的私有成员。无论基类的访问指定符是什么(public、protected 和 private)，私有的基类成员对于基类来说总是私有的。这有两个结果：第一，继承过来的私有成员是派生类中的私有成员(因此它们不能在派生类的外部访问)。第二，它们也不能由派生类的成员函数访问(因为它们对于基类来说是私有的)。

现在，看看如何继承基类的公共成员和受保护的成员。在所有的情况下，派生类的成员函数可以访问派生的成员。下面看看成员是如何继承的。

- 继承最常见的形式是公共继承。在这种情况下，继承成员的访问状态不变。因此，继承的公共成员也是公共的，继承的受保护成员也是受保护的。
- 当基类的继承是受保护的时，继承的公共成员在派生类中就是受保护的。受保护的继承成员在派生类中保持其原来的访问级别。
- 最后，当基类的继承是私有的时，继承的公共成员和受保护成员都是派生类的私有成员，可以由派生类的成员函数访问，但如果在另一个派生类中继承它们，它们就是不能访问的。

访问级别如图 15-3 所示。

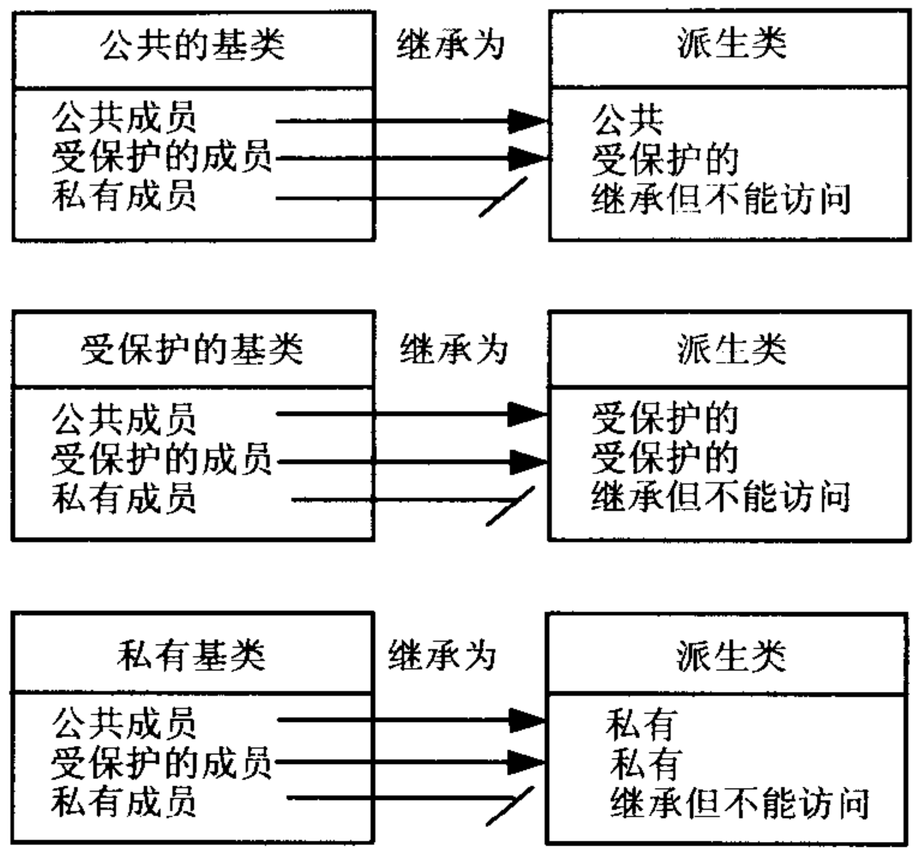


图 15-3　基类访问限定符对继承成员的影响

能改变派生类中继承成员的访问级别，会有一定程度的灵活性，但这只会使继承成员的访问级别变得非常古怪，并不能改变基类中指定的访问级别。

15.5.1　在类层次结构中使用访问指定符

在定义类的层次结构时，需要考虑两个方面：类成员的访问指定符和基类访问指定符。如第 12 章所述，类的公共成员定义了类的外部接口，它一般不应包含数据成员。如果类的成员不是类接口的一部分，就不能在类的外部直接访问，也就是说，它们应是私有成员或受保护的成员。给成员选择什么访问级别取决于是否允许它在派生类中访问。如果允许，就使用 protected，否则就使用 private。一般应尽可能把成员锁定在类的内部。

如图15-4 所示，继承成员的可访问性仅受这些成员在基类定义中的访问指定符的影响。在派生类中，公共基类成员和受保护的基类成员总是可以访问的，私有的基类成员则永远不能访问。在派生类的外部，只能访问公共的基类成员，这是基类声明为 public 的惟一情况。

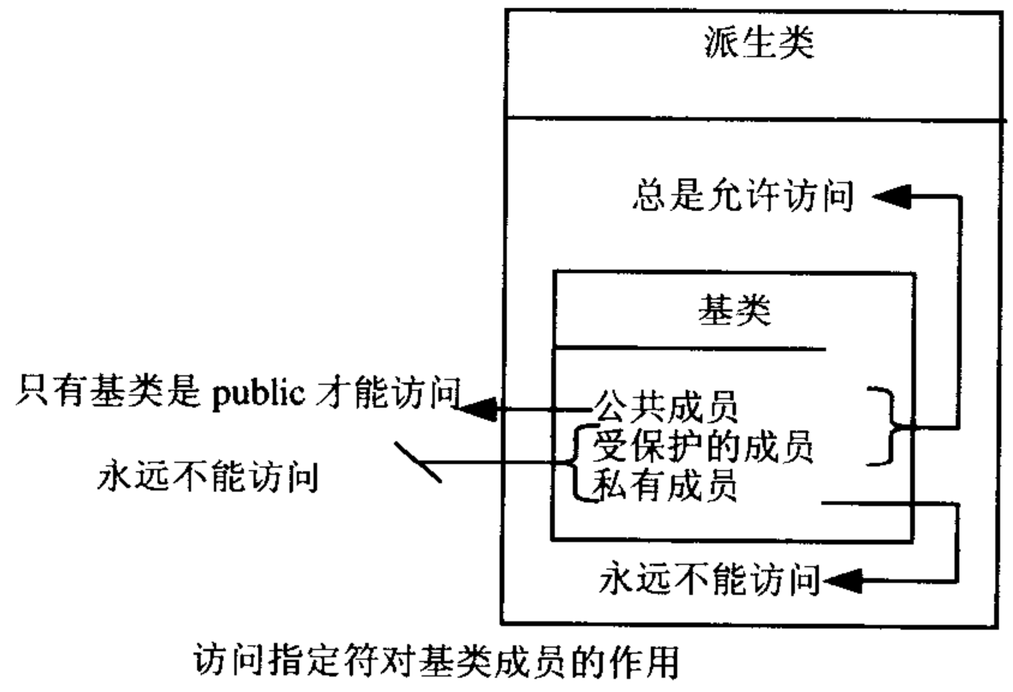


图 15-4　访问指定符对基类成员的影响

如果基类的访问指定符是 public，则继承成员的访问状态保持不变。而使用 protected 和 private 基类访问指定符，可以做两件事。

第一，可以阻止在派生类的外部访问公共的基类成员，protected 和 private 这两个基类访问指定符都可以阻止这类访问。如果基类拥有公共的函数成员，这就是重要的一步，因为基类的类接口从派生类的公共视图中删除了。

第二，可以影响派生类的继承成员如何在另一个类(这个类把该派生类作为自己的基类)中继承。

如图 15-5 所示，基类的公共成员和受保护的成员在另一个派生类中成为受保护的成员，而从私有基类中继承的成员在以后继承的派生类中都不能访问。

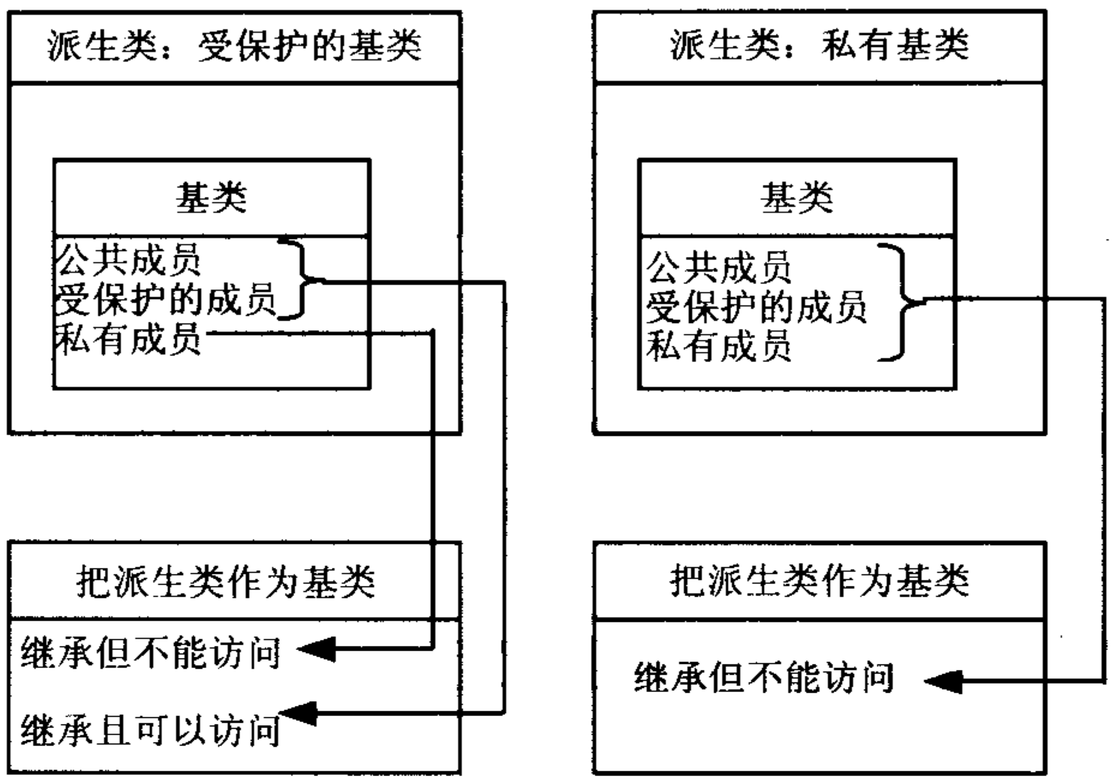


图 15-5　访问指定符对继承成员的影响

在大多数情况下，如例子中所示，public 基类访问指定符是最常见的，基类数据成员声明为 private 或 protected。使用 private 或 protected 基类访问指定符，基类子对象就是派生类的内部对象，因此不是派生的类对象的公共接口的一部分。实际上，由于派生的类对象是一种基类对象，而基类接口要在派生类中继承，因此基类必须指定为 public。

15.5.2　改变继承成员的访问指定符

假定要使某个基类成员免受 protected 或 private 基类访问指定符的影响。下面看一个例子。假定定义 Box 类，其中包含一个公共的成员函数 volume()：

```
class Box {
  public:
    // Constructor
    Box(double lv=1.0, double wv=1.0, double hv1.0);

    // Function to calculate the volume of a Box object
    double volume() const;

  protected:
    double length;
    double width;
```

```
    double height;
};
```

下面是这个类的函数定义：

```
// Constructor
Box::Box(double lv, double wv, double hv) : length(lv), width(wv), height (hv){
  cout << "Box constructor"  << endl;
}

// Function to calculate the volume of a Box object
double Box::volume() const {
  return length*width*height;
}
```

现在派生一个新类 Package，并把 Box 类作为 Package 的私有基类，但函数 volume()在派生类中是公共成员。当然，private 继承的默认结果是所有的公共成员和受保护的成员在派生类中都是私有的。使用 using 声明，可以改变某个继承成员的访问指定符。

这与第 10 章讨论命名空间时使用的 using 声明完全一样。使用下面的类定义，就可以迫使函数 volume()在派生类中是公共成员：

```
class Package : private Box  {
  public:
    using Box::volume;              // Inherit as public
    // Rest of the class definition
};
```

上面的语句定义了一个作用域，类定义中的 using 声明在类作用域中引入了一个名称。using 声明对基类成员函数 volume()重写了 private 基类访问指定符。该函数在 Package 类中继承为 public，而不是 private。

这里有两个语法要注意。第一，在对基类的成员应用 using 声明时，必须用基类名限定该成员名，如上所示，因为类名指定了成员名的上下文。第二，这里不应提供参数列表或返回类型，仅提供成员函数的限定名即可。

注释：

这里使用 using 声明来重写基类访问指定符对继承的成员函数的作用。这个技术还可以应用于继承的数据成员——在成员名前面加上基类名和作用域解析运算符即可。

还可以使用 using 声明重写一般的 public 或 protected 基类访问指定符。使用这个技术允许基类成员在派生类中的可访问性更高，或可访问性更低。例如，如果函数 volume()在 Box 基类中是受保护的成员，就可以使用 using 声明，在派生类 Package 中使它成为公共成员。

注意，不能以这种方式把 using 声明应用于基类的私有成员，因为私有成员在派生类中不能访问。

知道如何隐藏基类的成员，而在派生类中仍可以访问它们后，下面看看构造函数的问题。

15.6 派生类中的构造函数操作

在程序示例 15.2 和 15.3 中，都自动调用了基类的默认构造函数，但不一定要调用它。可以在派生类的构造函数中指定调用某个基类构造函数，用非默认的构造函数来初始化基类的数据成员。这样可以根据提供给派生类构造函数的数据，来选择不同的基类构造函数。

程序示例 15.4—— 调用基类构造函数

下面修改程序示例 15.3，演示这个过程。保留已有的类构造函数中的跟踪语句。为了说明如何显式地调用基类构造函数，给 Carton 类添加第二个构造函数，指定对象的尺寸。这需要在 Carton.h 中修改 Carton 类的定义：

```
class Carton : public  Box {
  public:
    // Constructor which can also act as default constructor -
    //  calls default base constructor automatically
    Carton(const char* pStr = "Cardboard");

    // Constructor explicitly calling the base constructor
    Carton(double lv, double wv, double hv, const char* pStr = "Cardboard");

    ~Carton();              // Destructor

    // Function to calculate the volume of a Carton object
    double volume() const;

  protected:
    char* pMaterial;
};
```

与前面一样，函数定义包含在 Carton.cpp 中：

```
// Carton. cpp
#include "Carton.h"
#include <cstring>
#include <iostream>
using std::cout;
using std::endl;

// Constructor which can also act as default constructor -
//          calls default base constructor automatically
Carton::Carton(const char* pStr)  {
  pMaterial = new char [strlen(pStr) +1] ;        // Allocate space for the string
  std::strcpy( pMaterial, pStr) ;                 // Copy it
  cout  <<  "Carton constructor 1" <<  endl;
}

// Constructor explicitly calling the base constructor
Carton::Carton(double lv, double wv, double hv, const char* pStr) :
                                                          Box(lv, wv, hv) {
```

```
  pMaterial = new char [strlen(pStr) +1] ;        // Allocate space for the string
  std::strcpy(pMaterial, pStr) ;                  // Copy it
  cout  <<  "Carton constructor 2" <<  endl;
}

// Destructor
Carton::~Carton() {
  delete[] pMaterial;
}

// Function to calculate the volume of a Carton object
double Carton::volume() const {
  return length*width*height;
}
```

现在可以跟踪何时调用 **Box** 类中的默认构造函数了，下面修改定义，提供一个独立的默认构造函数：

```
class Box {
  public:
    Box();                                              // Default  constructor
    Box (double lv,   double wv,  double hv) ;  // Constructor

  protected:
    double length;
    double width;
    double height;
};
```

把定义放在文件 **Box.cpp** 中：

```
// Box.cpp
# include "box.h"
#include <iostream>
using std::cout;
using std::endl;

// Default constructor
Box::Box() : length(1.0), width(1.0), height(1.0) {
  cout  <<  "Default Box constructor" << endl;
}

// Constructor
Box: : Box (double lv, double wv, double hv) : length (lv), width (wv) , height(hv){
  cout <<  "Box constructor"  <<  endl;
}
```

编写 **main()**的新版本，调用两个 **Carton** 类构造函数：

```
// Program 15.4 Calling a base class constructor from a derived class constructor
```

```
// File prog15_04.cpp
#include <iostream>
#include "Box.h"                                    // For the Box class
#include "Carton.h"                                 // For the Carton class
using std::cout;
using std::endl;

int main() {
  // Create two Carton objects
  Carton myCarton;
  Carton candyCarton(50.0,30.0,20.0,"Thin cardboard");

  cout << endl <<  "myCarton volume is "       << myCarton.volume ();
  cout << endl << "candyCarton volume is "    << candyCarton.volume ()
       << endl;

  return 0 ;
}
```

编译并运行这个版本的程序，结果如下：

```
Default Box constructor
Carton constructor 1
Box constructor
Carton constructor 2

myCarton volume is 1
candyCarton volume is 30000
```

例子的说明

每个构造函数的输出语句都提供了事件序列的清晰说明。对于 Carton 对象 myCarton，编译器首先调用默认的 Box 构造函数，之后调用第一个 Carton 构造函数。默认的 Box 构造函数调用是自动进行的，如前所示。

对于第二个 Carton 对象，将显式调用 Box 构造函数，之后调用第二个 Carton 构造函数。

```
Carton:: Carton(double lv, double wv, double hv, const char* pStr):Box(lv, wv, hv)
```

Box 类构造函数的显式调用放在冒号的后面，而冒号的前面是派生类的构造函数。该调用仅出现在构造函数定义中，不会出现在原型中。Carton 构造函数的参数传送给基类构造函数作为参数。

提示：

调用基类构造函数的表示法与在构造函数中初始化数据成员的表示法完全相同。在 Box 类的构造函数中就使用了这个表示法。它与这里要进行的操作完全一致，因为这里是使用传送给 Carton 构造函数的参数来初始化 Carton 对象的 Box 子对象。

表 15-1 列出了输出中的每个构造函数调用都负责做什么工作。

表 15-1　输出中的各个构造函数

程 序 输 出	要构建的对象
默认的 Box 构造函数	myCarton 对象的 Box 子对象
Carton 构造函数 1	myCarton 对象的其他成员
Box 构造函数	candyCarton 对象的 Box 子对象
Carton 构造函数 2	candyCarton 对象的其他成员

在创建派生类的对象时，必须先调用基类构造函数，再调用派生类的构造函数。这是一个一般法则。在涉及到好几级继承时，最一般的基类的构造函数最先调用，之后按照派生的顺序调用派生类的构造函数，直到最后，调用最后一级的派生类构造函数。

基类中非私有的数据成员可以在派生类中访问，但它们不能在派生类构造函数的初始化列表中初始化。例如，用下面的代码替换第二个 Carton 类构造函数(只需修改 Carton.cpp 文件)：

```
//Constructor that won't compile!
Carton::Carton(double lv, double wv,double hv, const char* pStr):
                                                  length(lv), width(wv), height(hv) {
  pMaterial = new char[strlen(pStr)+1];       //Allocate space for the string
  strcpy(pMaterial, pStr);                    //Copy it
  cout<<"Carton Constructor 2"<<endl;
}
```

初看之下，觉得这段代码可以工作，因为 length、width 和 height 是受保护的基类成员，在派生类中被公共继承，所以 Carton 类构造函数应能访问它们。但是，编译器提出，length、width 和 height 不是 Carton 类的成员，为什么会这样？

答案是派生类构造函数可以引用受保护的基类成员，但不能在初始化列表中引用，因为在初始化时，这些成员还不存在。初始化列表是在调用基类的构造函数，对象的主要部分创建之前处理。如果要显式初始化继承的数据成员，就必须在构造函数体中进行。在 Carton.cpp 文件中使用下面的构造函数定义：

```
//This constructor doesn't cause a compiler error
Carton::Carton(double lv, double wv,double hv, const char* pStr) {
  length=lv;
  width=wv;
  height=hv;
  pMaterial = new char[strlen(pStr)+1];       //Allocate space for the string
  strcpy(pMaterial, pStr);                    //Copy it
  cout << "Carton constructor 2"<<endl;
}
```

在执行构造函数体时，对象的主要部分已创建。此时，Carton 对象的主要部分已通过自动调用基类的默认构造函数创建。之后就可以引用基类中非私有的成员名了。

派生类中的副本构造函数

前面介绍了在声明用户定义的类对象，并用同一个类的另一个对象初始化该对象时的情况。下面再简单讨论一下这种情况。考虑下面的语句：

```
Box myBox(2.0, 3.0, 4.0);          //Calls constructor
Box copyBox(myBox);                //Calls copy constructor
```

这里声明了一个 Box 对象 myBox，但关键是第二个语句，在这个语句中，声明并初始化了副本对象 copyBox。第 13 章说过，对于这种初始化，编译器会自动调用副本构造函数。在程序示例 12.5 中，如果没有为 Box 类定义自己的副本构造函数，编译器就会提供一个默认的版本，逐个复制原对象成员，创建出一个新对象来。

下面看看如何在派生类中使用副本构造函数。在程序示例 15.4 中使用的类定义中添加该副本构造函数。首先，在 Box.cpp 中插入如下代码，在基类 Box 中添加一个副本构造函数：

```
// Copy constructor
Box::Box(const Box& aBox):
            length(aBox .length), width(aBox .width), height(aBox .height) {
  cout << "Box copy constructor called"<<endl;
}
```

提示：

第 13 章提到，必须把副本构造函数的参数指定为引用。

这段代码复制了原来的值，初始化了数据成员，并生成输出，以便跟踪何时调用了 Box 副本构造函数。还需要在 Box.h 中，在 Box 类声明的公共部分添加副本构造函数的原型：

```
Box(const Box& aBox);              //Copy constructor
```

对于 Carton 类，也要添加自己的副本构造函数。首先，在 Carton.cpp 中添加如下定义：

```
// Copy constructor
Carton::Carton(const Carton& aCarton) {
  pMaterial  = new char[strlen(aCarton.pMaterial) +1] ;// Allocate  space  for
                                                      // string
  strcpy (pMaterial,   aCarton. pMaterial) ;          // Copy it
  cout  <<   "Carton copy constructor"    <<   endl;
}
```

当然，还需要在 Carton.h 的类定义中添加原型：

```
Carton(const Carton& aCarton );          //Copy constructor
```

提示：

实际上，Carton 类的副本构造函数非常重要，因为该类包含数据成员 pMaterial，它是一个指针。副本构造函数必须复制 pMaterial 指向的字符串，而不仅仅复制指针。否则，原来的对象和复制的对象就各自包含一个指向同一个字符串的指针。当删除其中一个对象时，字符串也会删除，剩下的一个对象的指针就没有用了。

下面看看这是否工作。

程序示例 15.5—— 派生类中的副本构造函数

下面创建一个 Carton 对象，再创建一个副本，试一试刚才在 Box 类和 Carton 类中定义的副本构造函数：

```
// Program 15.5 Using a derived class copy constructor   File: prog15_05.cpp
# include <iostream>
#include "Box.h"                                    // For the Box class
#include "Carton.h"                                 // For the Carton class
using std::cout;
using std::endl;

int main() {
   // Declare and initialize a Carton object
   Carton candyCarton(20.0,30.0,40.0,"Glassine board");

   Carton copyCarton(candyCarton);                  // Use copy constructor

   cout <<  endl
        << "Volume of candyCarton is"  << candyCarton.volume()
        <<  endl
        << "Volume of copyCarton is"   << copyCarton.volume()
        << endl;

   return 0;
}
```

运行结果如下：

```
Box constructor
Carton constructor 2
Default Box constructor
Carton copy constructor

Volume of candyCarton is 24000
Volume of copyCarton is 1
```

例子的说明(或该程序为什么不工作)

仔细观察输出，就会发现结果不应是这样。显然，copyCarton 的体积与 candyCarton 的体积不同，这是很奇怪的，因为 copyCarton 是 candyCarton 的副本。输出还显示了其中的原因。

为了复制 candyCarton 对象，调用了 Carton 类的副本构造函数。作为复制过程的一部分，Carton 类的副本构造函数必须复制 candyCarton 的 Box 子对象，为此，就必须调用 Box 副本构造函数。但是，输出清楚地显示，这里调用的是默认的 Box 构造函数。

Carton 类的副本构造函数没有调用 Box 副本构造函数，仅仅因为我们没有告诉它要调用。编译器知道它必须为对象 candyCarton 创建一个 Box 子对象，但没有指定如何创建，编译器不可能猜测出我们的意图，就创建了默认的基类对象。

注意：

在为派生类的对象编写构造函数时，一定要确保正确地初始化派生类对象的成员。这包括所有继承的数据成员和派生类所特有的数据成员。

要修改这个程序，可以在 Carton 副本构造函数的初始化列表中调用 Box 副本构造函数。方法是修改 Carton.cpp 中的副本构造函数定义：

```
// Copy constructor
Carton::Carton(const Carton& aCarton) : Box(aCarton) {
 pMaterial = new char[strlen(aCarton.pMaterial)+1];//Allocate space for string
  strcpy(pMaterial,aCarton.pMaterial);              //   Copy it
  cout  <<     "Carton copy constructor"   <<  endl;
}
```

现在，调用的就是 Box 类的副本构造函数，并把 aCarton 对象作为一个参数。对象 aCarton 的类型是 Carton，但 Box 类副本构造函数的参数是 Box 对象的引用。编译器会插入对 aCarton 的转换操作—— 从 Carton 类型转换为 Box 类型，把 aCarton 对象中的基本部分传送给 Box 类副本构造函数。现在，再次编译并运行这个例子，输出就如下所示：

```
Box constructor
Carton constructor 2
Box copy constructor called
Carton copy constructor

Volume of candyCarton is 24000
Volume of copyCarton is 24000
```

输出中显示，构造函数以正确的顺序调用。特别是 Box 副本构造函数(用于 copyCarton 中的 Box 子对象)在 Carton 副本构造函数之前调用。检查一下，就会发现，candyCarton 和 copyCarton 对象的体积是相同的。

总之，这里的重要规则是：

可以为派生类的任何构造函数编写自己的定义，此时，应初始化派生类对象的所有成员，包括所有继承来的成员。

15.7 继承中的析构函数

在派生类对象超出作用域时，或位于自由存储区时，要释放它，总是要涉及到派生类的析构函数和基类的析构函数。

下面利用 Box 和 Carton 析构函数定义中的语句来跟踪何时调用这两个析构函数。修改程序示例 15.5 中使用的类定义。首先在 Box.cpp 文件中添加如下析构函数定义：

```
//  Destructor
Box::~Box() {
  cout <<  "Box destructor"   <<  endl;
}
```

在 Box.h 的类定义中添加原型：

```
~Box();         //Destructor
```

Carton 类已经有一个析构函数，这里仅需要在析构函数定义中添加跟踪语句。在 Carton.cpp 中添加如下代码：

```
// Destructor
```

```
Carton::~Carton() {
  cout << "Carton destructor. Material = " << pMaterial << endl;
  delete[] pMaterial;
}
```

跟踪输出显示了合成材料，说明已经把另一个合成材料赋予了 Carton 对象，这表示 Carton 对象已释放。下面看看这些类是如何工作的。

程序示例 15.6—— 类层次结构中的析构函数

在下面的 main()中，演示了类析构函数如何操作派生类的对象：

```
// Program 15.6 Destructors in a class hierarchy File:prog15_06.cpp
#include <iostream>
#include "Box.h"                                  // For the Box class
#include "Carton.h"                               // For the Carton class
using std::cout;
using std::endl;

int main() {
  Carton myCarton;
  Carton candyCarton(50.0, 30.0, 20.0, "Thin cardboard");

  cout << endl << "myCarton volume is "     << myCarton.volume ();
  cout << endl << "candyCarton volume is " << candyCarton.volume ()
       << endl << endl;

  return 0;
}
```

修改包含类定义的头文件后，重新编译这个程序，结果如下：

```
Default Box constructor
Carton constructor 1
Box constructor
Carton constructor 2

myCarton volume  is  1
candyCarton volume  is  30000

Carton destructor. Material  = Thin cardboard
Box destructor
Carton destructor. Material  =  Cardboard
Box destructor
```

例子的说明

这里创建了两个 Carton 对象：第一个是有默认尺寸的对象，第二个有显式赋予的尺寸和材料描述。接着输出每个对象的体积。

这个示例的重点是看看析构函数是如何工作的。析构函数调用的结果说明了释放对象的两个方面。第一，是对某个对象调用析构函数的顺序，第二，是对象的释放顺序。

输出所记录的析构函数调用对应于表 15-2 的操作。

表 15-2　析构函数调用对应的操作

析构函数输出	释放的对象
Carton destructor.Material=Thin cardboard	candyCarton 对象
Box destructor	candyCarton 的 Box 子对象
Carton destructor.Material=Cardboard	myCarton 对象
Box destructor	myCarton 的 Box 子对象

从表 15-2 中可以看出，对象的释放顺序与创建它们的顺序相反。对象 myCarton 先创建，最后释放，candyCarton 对象最后声明，但最先释放。

选择这个顺序是为了确保不使对象处于不合法的状态。对象只有在声明之后才能使用，也就是说，任何给定的对象都只能包含指向(引用)已建对象的指针(或引用)。在释放其他对象指向(或引用)的给定对象后，才能确保析构函数的执行不会出现无效的指针或引用。

析构函数的调用顺序

对于派生类对象，析构函数的调用顺序与为该对象调用构造函数的顺序相反。派生类的析构函数先调用，再调用基类构造函数，如本例所示。三级类层次结构如图 15-6 所示。

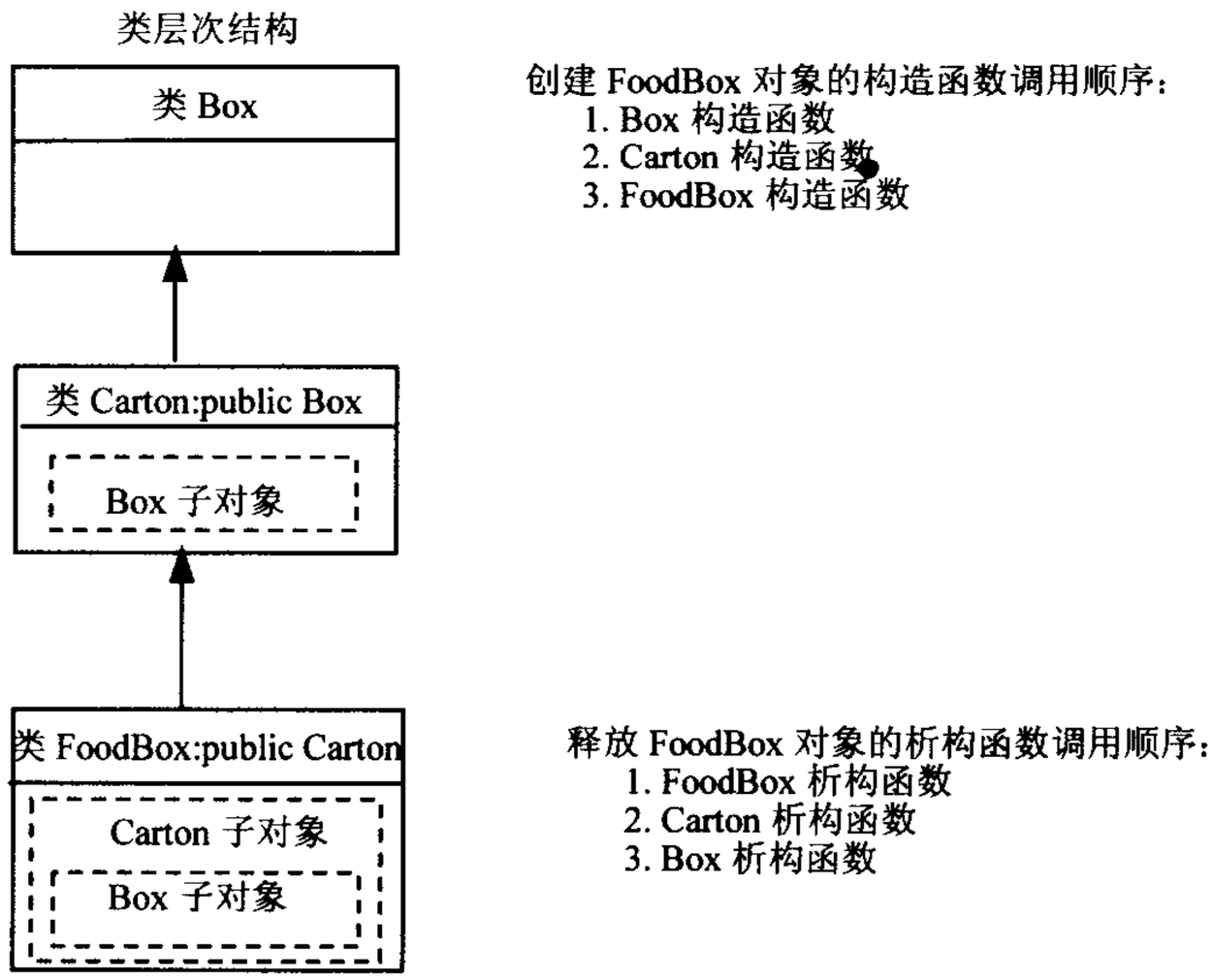


图 15-6　析构函数的调用顺序

对于具有好几级基类的对象，析构函数的调用顺序与类层次结构的顺序相同，首先是派生类的析构函数，最后是最一般的基类析构函数。

15.8　重复的成员名

有时基类和派生类有同名的数据成员，甚至可能在基类和间接基类中都有同名的数据成员。

当然，这种情形会造成混乱，在自己的类中应避免出现这种情况。但是，有时就会有同名的成员。例如，如果从由另一个程序员创建的基类中派生自己的类，肯定不知道在基类中有什么私有数据成员，只知道基类有哪些接口。如果基类和派生类使用了相同的数据成员名，该怎么办？

实际上，名称的重复并不会阻碍继承。下面看看如何区分名称相同的基类成员和派生类成员。

假定有一个类 Base 定义为：

```
class Base  {
  public:
    Base(int number  =  10){  value  = number;  }     //  Constructor
  protected:
    int value;
};
```

其中只包含一个数据成员 value 和一个构造函数。从 Base 中派生一个类 Derived：

```
class Derived:  public Base  {
  public:
    Derived(int number = 20){ value = number; } // Constructor
    int total() const;                          //  Total value of data members
  protected:
    int value;
};
```

派生类有一个数据成员 value，同时从基类中继承了成员 value。这已经开始出现混乱了。下面编写 total()函数的定义，说明编译器如何在派生类中区分这两个数据成员。

在派生类的成员函数中，名称 value 本身表示在该作用域中声明的数据成员，即 value 是一个派生类成员。基类成员在另一个作用域中声明，要在派生类成员函数中访问它，必须限定成员名(使用基类名和作用域解析运算符)。因此，total()函数的实现代码如下：

```
int Derived::total() const {
  return value  + Base::value;
}
```

表达式 Base::value 指定基类数据成员，名称 value 本身表示 Derived 类中声明的成员。

重复的函数成员名

在基类和派生类的成员函数同名时，会发生什么情况？派生类中的函数与基类的成员函数同名有两种情况。

第一种情况，函数的名称相同，但参数列表不同。尽管函数的签名不同，但这并不是函数的重载，因为重载的函数必须在同一个作用域中定义，而基类和派生类定义了不同的作用域。

实际上，作用域是这种情形的关键。派生类的函数成员隐藏了同名的继承函数成员。在基类成员函数和继承成员函数同名时，必须使用 using 声明，在派生类的作用域中引入基类成员函数的限定名。派生类对象可以调用这两个函数，如图 15-7 所示。

```
class Base
{
 public:
    void doThat(int arg);
    …
};


class Derived:public Base
{
   public:
     void doThat(double arg);
     using Base::doThat;
     …
};
```

在默认情况下，派生类函数 doThat()会隐藏同名的继承函数。using 声明把函数名 doThat 从基类的作用域引入到派生类的作用域中，这样派生类的作用域就包括函数的两个版本。

```
Derived object;
object.doThat(2);     //Call inherited base function
object.doThat(2.5);   //Call derived function
```

图 15-7　继承与已有函数同名的函数

在第二种情况下，继承的函数在所有的方面都相同，即函数的签名相同。使用类名和作用域解析运算符来调用基类函数，就可以区分继承函数和派生类函数：

```
Derived object;              // Object declaration
object.Base::doThat(3);  // Call base version of the function
```

但是这个主题还包含许多内容，而且与多态性相关，第 16 章将进一步论述这个主题。

15.9　多重继承

前面，派生类都派生自一个直接基类。但是我们并没有限制这个结构—— 派生类可以有任意多个直接基类。这个功能称为多重继承(与单一继承相对，即只使用一个基类)，使继承得到了极大的丰富。

多重继承的使用要比单一继承少多了，这里也不深入分析。只是说明多重继承的基本工作原理，了解其复杂性的根源。

15.9.1　多个基类

多重继承涉及到使用两个或多个直接基类，来派生一个新类，所以比较复杂。在这种情况下，派生类是基类的一个特例这个概念就会导出另一个概念：派生类定义的对象是两个或多个不同且独立的类类型并存的特例。

实际上，多重继承常常不以这种方式使用，而是使用多个基类，将这些基类的特性加在一起，形成一个包含所有基类功能的合成对象，有时这称为“混合编程”。这通常比较便于实现，而不便于反映对象之间的关系。例如，考虑某种类型的编程接口，如图形化编程接口。综合的接口可以打包到一组类中，这些类都定义了一个自包含的接口，该接口提供了一些特定的功能，例如绘制二维图形。接着用几个类作为基类，派生出一个新类，这个新类就提供了应用程序所需要的功能。

为了探讨多重继承中的一些特点，下面看看前面使用的层次结构，其中包含 Box 和 Carton 类。假定要定义一个类，表示包含脱水物品的箱子，例如谷物箱。这可以使用单一继承来完成，

即从 Carton 类中派生一个新类，再添加一个数据成员表示其内容。也可以使用如图 15-8 所示的层次结构来完成。

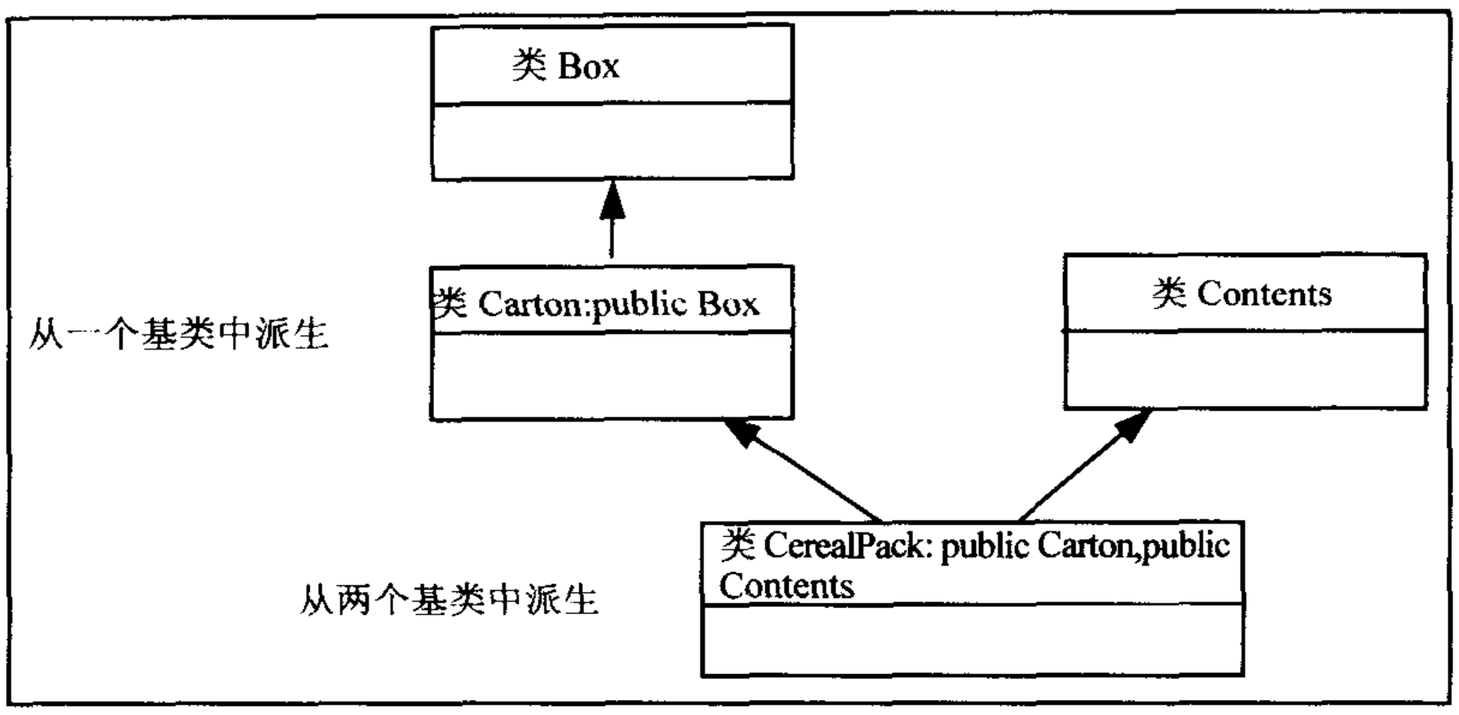


图 15-8　多重继承的例子

CerealPack 类的定义如下：

```
class CerealPack:public Carton,public Contents {
  // Details of the class…
};
```

在类的函数头中，每个基类都在冒号后指定，用逗号隔开。每个基类都有自己的访问指定符(与单一继承相同，如果省略访问指定符，就使用默认的 private)。

如图 15-9 所示，CerealPack 类继承了两个类的所有成员，所以它包含间接基类 Box 的成员。在单一继承中，每个继承成员的访问级别由两个因素决定：基类中成员的访问指定符和基类的访问指定符。CerealPack 对象包含两个子对象 Contents 和 Carton，Carton 本身又包含 Box 类型的一个子对象。

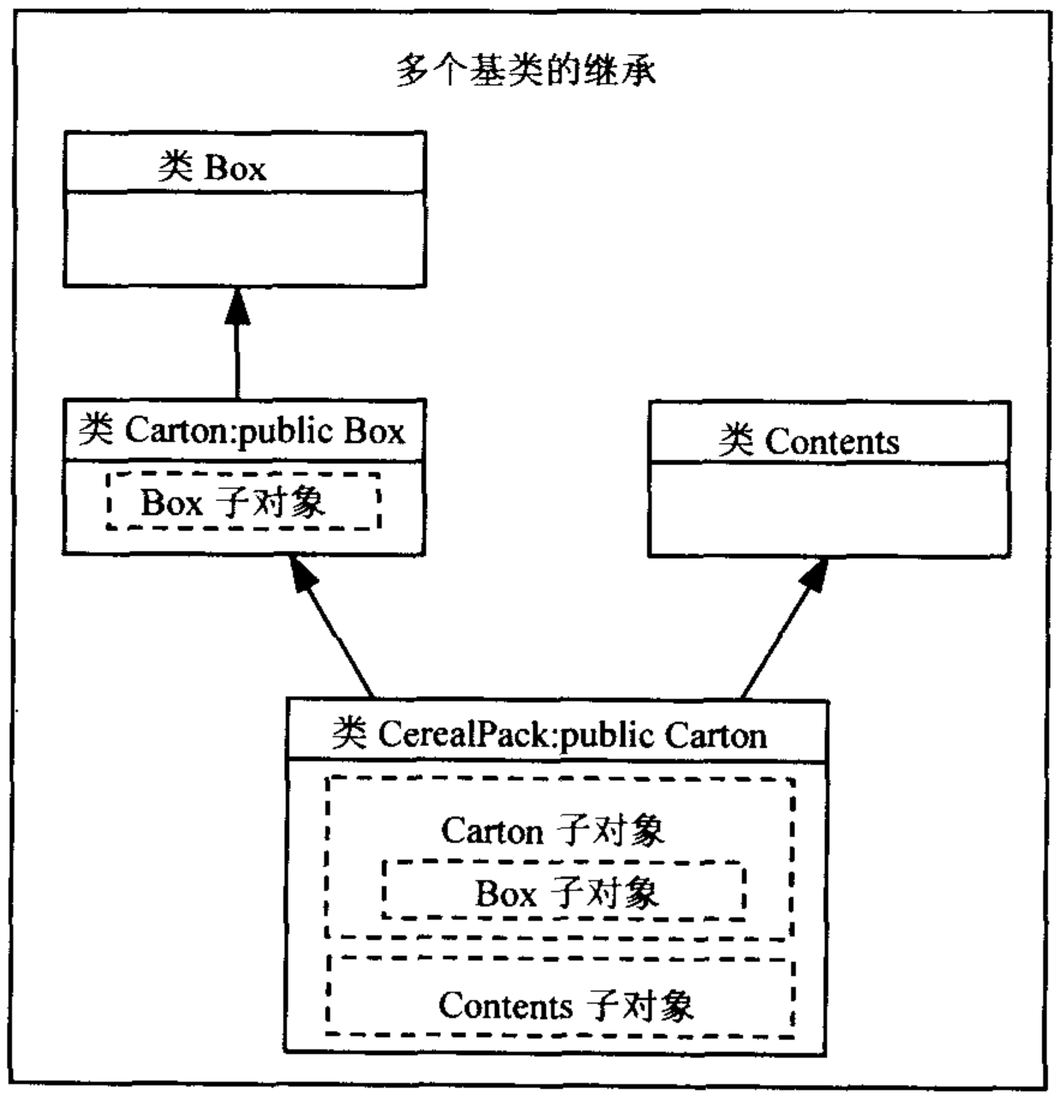


图 15-9　有多个基类的继承

15.9.2 继承成员的模糊性

下面详细定义自己的类。在 Box.h 中定义 Box 类：

```
// Box.h - defines Box class
#ifndef BOX_H
#define BOX_H
class Box {
  public:
    Box(double lv=1, double wv=1, double hv=1);     // Constructor

    Box(const Box& aBox);                           // Copy constructor

    ~Box();                                         // Destructor

    // Function to calculate the volume of a Box object
    double volume()  const;

  protected:
    double length;
    double width;
    double height;
};
#endif
```

下面的成员函数定义放在 Box.cpp 文件中：

```
// Box.cpp
#include "box.h"
#include <iostream>
using std::cout;
using std::endl;

// Default constructor
Box:: Box(double lv, double wv, double hv) : length(lv), width (wv), height (hv){
  cout<< "Box constructor"<< endl;
}

// Copy constructor
Box::Box(const Box& aBox):
        length (aBox. length) , width (aBox. width) , height ( aBox . height ){
  cout << "Box copy constructor called"<< endl;
}

// Destructor
Box::~Box() {
  cout<<"Box destructor"<< endl;
}

// Function to calculate the volume of a Box object
double Box::volume() const {
```

```
  return length*width*height;
}
```

这个版本的 Box 类非常类似于前面的版本。其中有一个构造函数，它提供了参数的默认值，所以用作默认的构造函数。volume 函数也放在基类中。

下面略微扩展一下 Carton 类定义。头文件 Carton.h 如下所示：

```
// Carton.h - defines the Carton class with the Box class as base
#ifndef CARTON_H
#define CARTON_H
#include "Box.h"                          // For Box class definition

class Carton : public Box {
 public:
   // Constructor explicitly calling the base constructor
   Carton (double lv = 1, double wv = 1, double hv = 1,
           const char* pStr = "Cardboard",
           double dense = 0.125, double thick = 0.2);

  ~Carton();                              // Destructor

   double getWeight() const;              // "Get carton weight" function

 protected:
   char* pMaterial;                       // Carton material
   double thickness;                      // Material thickness inches
   double density;                        // Material density in pounds /cubic inch
};
#endif
```

其中有两个新的数据成员，记录制造 Carton 对象的材料的浓度和密度。构造函数提供了 Carton 类的所有数据成员的默认值。在该类中还有一个新函数 getWeight()，它使用新的数据成员来计算空 Carton 对象的重量。其定义在 Carton.cpp 中：

```
// Carton.cpp
#include  "Carton.h"
#include  <cstring>
#include  <iostream>
using std::cout;
using std::endl;

// Constructor
Carton::Carton(double lv, double wv, double hv,
               const char* pStr,double dense,double thick) :
                     Box(lv, wv, hv), density(dense), thickness (thick) {
  pMaterial  = new char[strlen(pStr)+1] ;    // Allocate space for the string
  strcpy(  pMaterial, pStr) ;                // Copy it
  cout  <<  "Carton constructor"  << endl;
}

// Destructor
```

```
Carton::~Carton() {
  cout << "Carton destructor" << endl;
  delete [] pMaterial;
}

// "Get carton weight" function
double Carton::getWeight() const {
  return 2* (length*width + width*height + height*length) *thickness*density;
}
```

Contents 类描述了各种可以包装在硬纸盒中的脱水产品，例如早餐饼。Contents 类有 3 个数据成员：name、volume 和 density(每立方英寸的磅数)。实际上，可以包含一组可能的谷物类型，以及它们的密度，以便在构造函数中验证数据。但这里忽略这些，以使程序简单一些。

下面是类定义和需要放在头文件 Contents.h 中的预处理器指令：

```
// Contents.h - Dry contents
#ifndef CONTENTS_H
#define CONTENTS_H

class Contents {

  public:
    Contents (const char* pStr = "cereal", double weight =0.3, double vol  =  0);
                                          // Constructor
    ~Contents();                          //  Destructor

    double getWeight()   const;           //  "Get  contents weight"   function

  protected:
    char*  pName;                        //  Contents  type
    double volume;                       //  Cubic  inches
    double unitweight;                   //  Pounds  per  cubic   inch
};
#endif
```

除了构造函数和析构函数之外，还有一个公共函数 getWeight()，它计算各种脱水产品的重量。其定义包含在 Contents.cpp 中：

```
// Contents.cpp
#include  "contents.h"
#include <cstring>
#include <iostream>
using std::cout;
using std::endl;

// Constructor
Contents::Contents(const char* pStr, double weight, double vol):
                                        unitweight (weight ),volume(vol) {
  pName  = new char [strlen(pStr)+1] ;
  std::strcpy(pName, pStr) ;
  cout  << "Contents  constructor"  <<  endl;
```

```
}

// Destructor
Contents::~Contents () {
  delete[] pName;
  cout  <<   "Contents destructor"  <<  endl;
}

// "Get Contents weight" function
double Contents::getWeight() const {
  return volume*unitweight;
}
```

现在把 Carton 和 Contents 类作为公共基类，定义 CerealPack 类。把该定义放在头文件 CerealPack.h 中：

```
// Cerealpack.h - Class defining a carton of cereal
#ifndef CEREALPACK_H
#define CEREALPACK_H
#include "Carton.h"
#include "Contents.h"

class CerealPack: public Carton, public Contents {
  public:
    CerealPack(double length, double width, double height,
                                    const char* cerealType); // Constructor
    ~CerealPack();                                           // Destructor
};
#endif
```

Cerealpack.cpp 文件中包含如下定义：

```
//Cerealpack.cpp
#include <iostream>
#include "Carton.h"
#include "Contents.h"
#include "Cerealpack.h"
using std::cout;
using std::endl;

//Constructor
CerealPack::CerealPack(double length, double width, double height,
                                                    const char* cerealType) :
          Carton (length, width, height, "cardboard"), Contents ( cerealType){
  cout << "CerealPack constructor" << endl;
  Contents::volume = 0.9*Carton::volume( ) ;        // Set contents volume
}

// Destructor
CerealPack::~CerealPack(){
  cout << "CerealPack destructor" << endl;
}
```

这个类继承了 Carton 和 Contents 类。构造函数只需要外部尺寸和谷物类型。Carton 对象的材料在 Carton 构造函数调用的初始化列表中设置。

CerealPack 对象包含两个子对象，它们分别对应于两个基类。每个子对象都通过构造函数调用，在 CerealPack 类构造函数的初始化列表中进行初始化。注意 Contents 类的 volume 数据成员默认为 0，所以在 CerealPack 类的构造函数体中，根据 Carton 的 volume 成员来计算它的值。这里必须限定从 Contents 类继承而来的 volume 数据成员的引用，因为它与通过 Carton 从 Box 继承而来的函数同名。可以在输出语句中跟踪构造函数和析构函数的调用顺序。

程序示例 15.7—— 使用多重继承

下面创建一个 CerealPack 对象，计算它的体积和重量，程序如下：

```
//Program 15.7 Using multiple inheritance    File: prog15_07.cpp
#include <iostream>
#include "CerealPack.h"                          // For the CerealPack class
using std::endl;
using std::cout;

int main() {
   cerealPack packofFlakes(8.0,3.0,10.0,"Cornflakes");

   cout << endl;
   cout << "packOfFlakes volume is"<<packOfFlakes.volume()  << endl;
   cout << "packOfFlakes weight is" <<packOfFlakes.getWeight()
        << endl;

   return 0;
   }
```

可惜，这个程序不会编译。问题是基类中使用了一些不惟一的函数名，名称 volume(从 Box 中继承为一个函数，而从 Contents 中继承为一个数据成员)和 getWeight()函数(分别从 Carton 和 Contents 中继承)在 CerealPack 类中不是惟一的。简言之，这是一个模糊性问题。

当然，在编写用于继承的类时，首先应避免重复的成员名。理想的解决方案是重新编写类。

如果不能重新编写类，例如，如果基类是从某类库中提取出来的，就必须限定 main()函数的函数名。这个程序修改为：

```
//Program 15.7a Using multiple inheritance.  Compilable version!
//File: prog15_07a.cpp
#include <iostream>
#include "CerealPack.h"                          //For the CerealPack class
using std::cout;
using std::endl;

int main() {
   CerealPack packofFlakes(8.0,3.0,10.0,"Cornflakes");

cout  << endl;
   cout << "packOfFlakes volume is"<<packOfFlakes.Carton::volume() << endl;
   cout << "packOfFlakes weight is"
        << packOfFlakes.Carton::getWeight()+packOfFlakes.Contents::getWeight()
        << endl;

   return 0;
}
```

现在程序就可以编译了，运行的结果如下所示：

```
Box constructor
Carton constructor
Contents constructor
CerealPack constructor

packOfFlakes volume is 240
packOfFlakes weight is 71.5
CerealPack destructor
Contents destructor
Carton destructor
Box destructor
```

例子的说明

从输出中可以看出，这个谷物箱是正确的—— 一个重量超过 4 磅的箱子。构造函数和析构函数的调用顺序也与单一继承的调用顺序相同—— 构造函数的调用顺序从最一般的基类开始到最特殊的派生类，而析构函数的调用顺序与此正好相反。

CerealPack 类型的对象有其继承链上的两个子对象，这些子对象的构造函数在创建 CerealPack 对象时都调用了。

15.9.3 重复的继承

在上面的例子中，演示了基类的成员名出现重复时的情形。在多重继承中，还应注意另一个模糊性：派生的对象包含一个基类的多个子对象版本。

在使用多重继承时，不能把一个类多次用作直接基类。但是，仍旧可能出现间接基类重复的情况。假定 Box 和 Contents 类都派生自 Common 类，图 15-10 显示了这里创建的类层次结构。

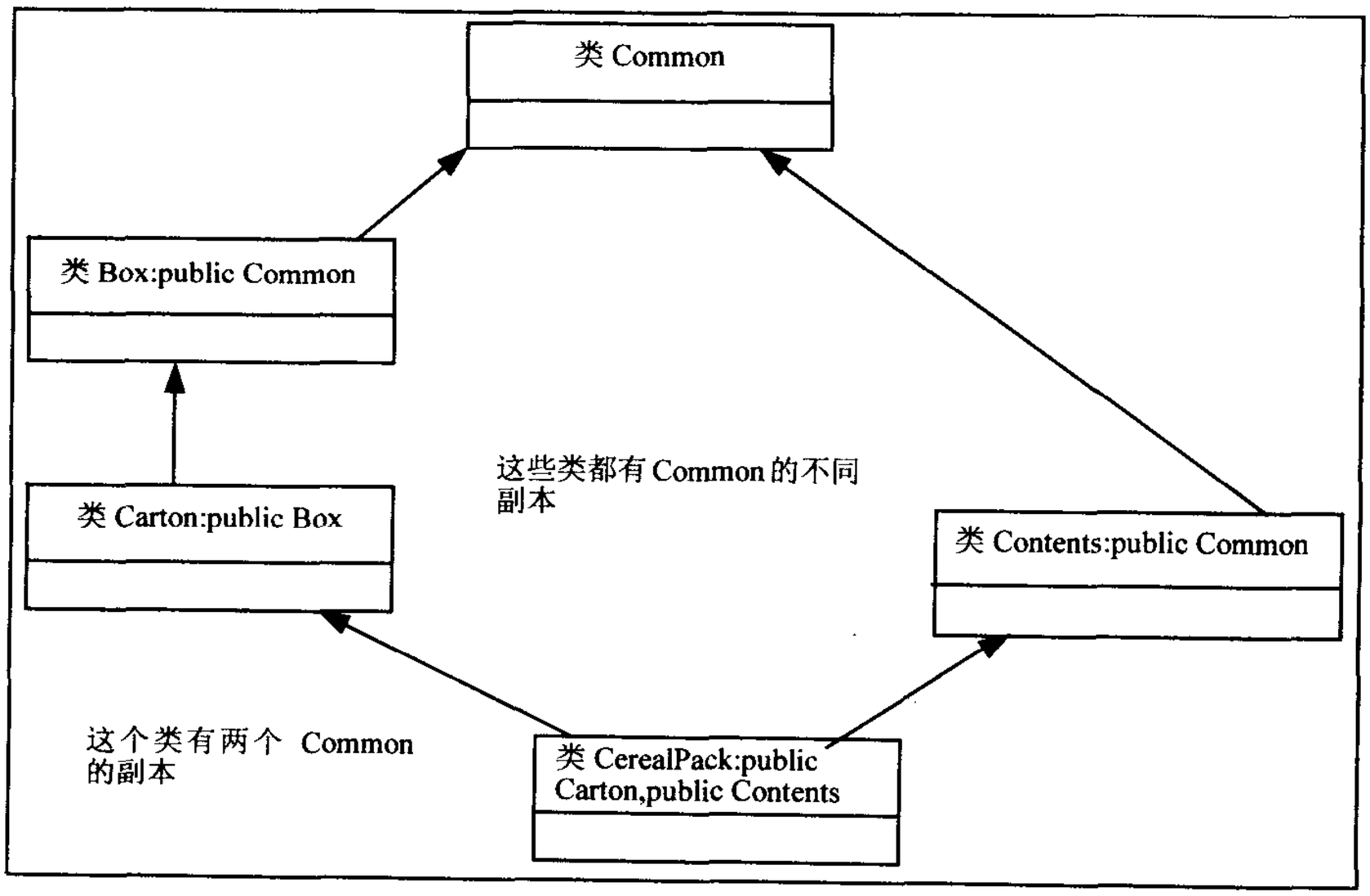


图 15-10　派生类中的重复基类

CerealPack 类继承了 Contents 类和 Carton 类的所有成员。Carton 类继承了 Box 类的所有成员，Box 和 Contents 类继承了 Common 类的成员。因此，如图 15-10 所示，Common 类在 CerealPack 类中是重复的。这在 CerealPack 类型的对象上产生的效果如图 15-11 所示。

如图 15-11 所示，每个 CerealPack 对象都有两个 Common 类型的子对象。

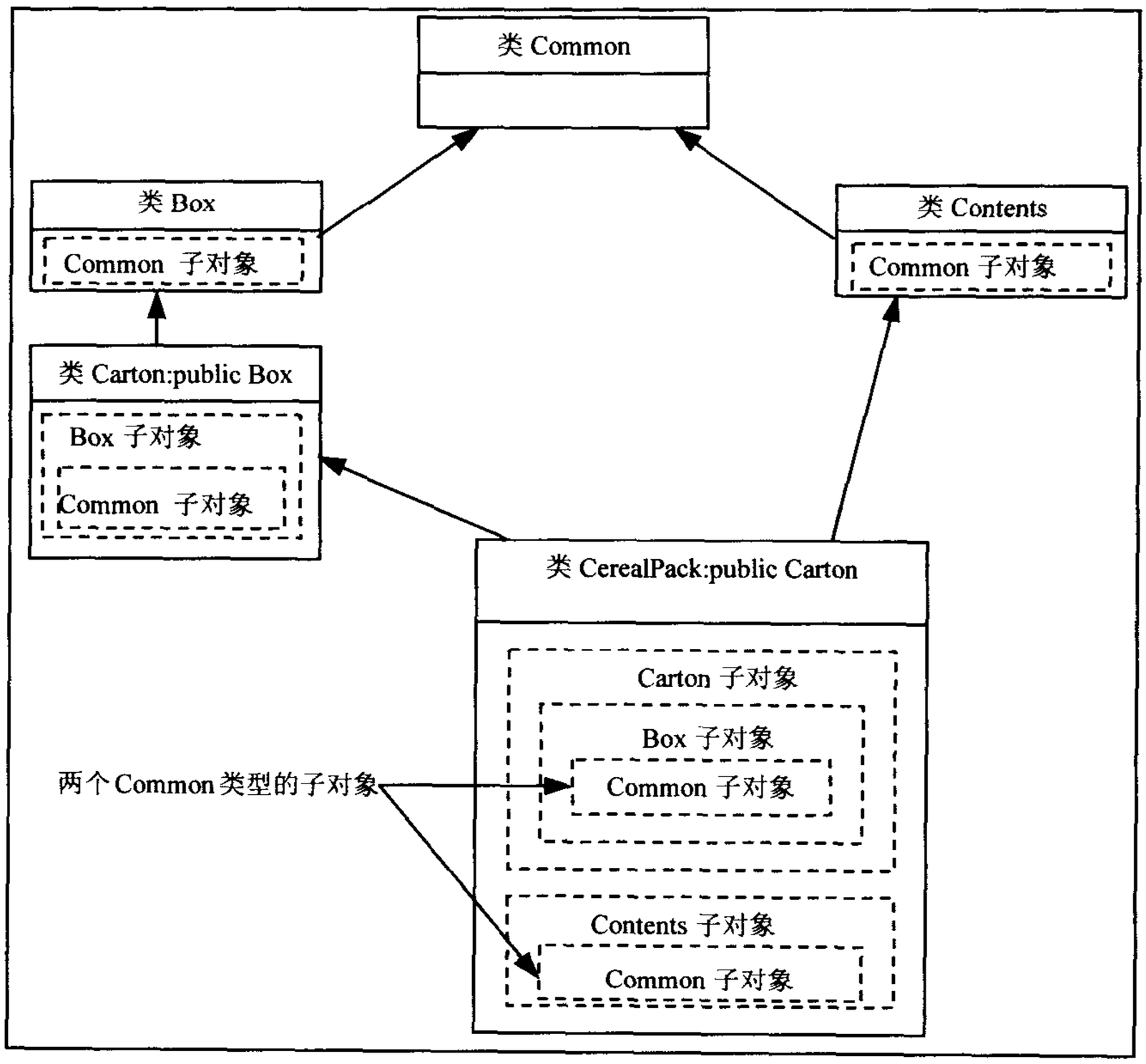


图 15-11　一个对象包含多个间接基类的子对象

允许 Common 类出现重复是可行的。在这种情况下，必须限定每个对 Common 类成员的引用，这样编译器就知道要引用的是哪个继承成员。在本例中，可以使用 Carton 和 Contents 类名作为限定符，因为这些类都包含 Common 类型的惟一子对象。当然，在创建 CerealPack 对象时，要调用 Common 类构造函数，还需要限定符来指定初始化哪个基类的对象。

但一般情况是，应避免基类的重复，下面看看如何避免。

15.9.4　虚基类

为了避免基类的重复，必须告诉编译器，在任何派生类中，基类应只出现一次。为此，可以使用关键字 virtual，把类声明为虚基类。Contents 类定义为：

```
class Contents: public virtual Common {
  ...
};
```

Box 类也应使用虚基类来定义：

```
class Box: public virtual Common {
  ...
};
```

现在，把 Contents 和 Box 类作为(直接或间接)基类的其他类会像以往一样继承基类的其他成员，但仅继承 Common 类的一个实例。在上面的例子中，派生的 CerealPack 类仅继承 Common 基类的一个实例。由于 CerealPack 类中没有重复的 Common 成员，在派生类中引用成员时，也就不需要限定成员名。

把 Common 类声明为 Contents 和 Box 类的虚基类，并没有排除另一个类把 Common 作为非虚类的可能性。这个类应是 CerealPack 的第三个基类。例如：

```
class Freebie: public Common {
  ...
};
```

CerealPack 类就变成：

```
class CerealPack: public Carton, public Contents, public Freebie {
  ...
};
```

现在 CerealPack 类有两个 Common 类型的子对象，一个继承自 Carton 和 Contents 类，另一个继承于 Freebie 类。为了对子对象引用 Common 的成员，必须使用直接基类名来限定成员名。如果把 Common 声明为 Freebie 的虚基类，CerealPack 类就只继承 Common 类型的一个子对象。

15.10 在相关的类类型之间转换

每个派生类对象都至少包含一个基类对象，把派生类型转换为基类类型总是合法和自动的。下面的声明定义了一个 Carton 对象：

```
Carton aCarton(40, 50, 60, "fiberboard");
```

下面的语句可以把这个对象转换为 Box 类型的基类对象，并存储结果：

```
Box aBox;
aBox= aCarton;
```

这个语句把 aCarton 对象转换为 Box 类型的新自动对象，并在变量 aBox 中存储它的一个副本。当然，它只是 aCarton 中的 Box 部分，Carton 部分被切开并舍弃了。这里使用的赋值运算符是 Box 类的默认赋值运算符。上面的语句不等价于：

```
Box aBox= aCarton;
```

最后的结果是相同的，但在这个语句中，涉及到 Box 类的副本构造函数，而不是赋值运算符。aCarton 对象转换为 Box 对象，并传送给 Box 类的副本构造函数。

如这个例子所示，类层次结构中向上的转换(即向基类方向的转换)只要没有模糊的成分，就是合法和自动的。当两个基类包含同一类型的子对象时，就会出现模糊性。例如，如果使用

包含两个 Common 子对象的 CerealPack 类的定义(如上一节所述)，来初始化 CerealPack 类型的对象 packOfFlakes，就会出现模糊性，如下面的语句所示：

```
Common commonObject=packOfFlakes;
```

编译器试图初始化 commonObject 的值，但不能确定 packOfFlakes 应转换为 Carton 的 Common 子对象，还是应转换为 Contents 的 Common 子对象。

在类层次结构中，对象不能自动实现向下的转换(即向比较特殊的类转换)。Box 类型的对象不包含任何派生于 Box 的类的信息，所以转换得不到合理的解释。

15.11 本章小结

本章学习了如何根据一个或多个已有的类定义新类，类继承如何确定派生类的构成。继承是面向对象编程的基本特性，也使多态性成为可能。本章的要点如下：

- 类可以派生自一个或多个基类，此时派生类在其所有的基类中继承成员。
- 单一继承就是从一个基类中派生新类。多重继承就是从两个或多个基类中派生新类。
- 访问派生类的继承成员由两个因素控制：基类中成员的访问指定符和在派生类声明中基类的访问指定符。
- 派生类的构造函数负责初始化类的所有成员，包括继承的成员。
- 创建派生类对象一般需要按顺序(从最一般的基类开始到最特殊的直接基类)调用所有直接和间接基类的构造函数，之后执行派生类的构造函数。
- 派生类构造函数可以在初始化列表中显式调用直接基类的构造函数。
- 在派生类中声明的成员名，如果与继承的成员名相同，就会遮盖继承的成员。为了访问被遮盖的成员，可以使用作用域解析运算符和类名来限定成员名。
- 如果派生类有两个或多个直接基类，就会包含同一个类的两个或多个继承子对象，此时，把重复的类声明为虚基类，就可以避免出现重复。

15.12 练习

1. 定义一个基类 Animal，它包含两个私有数据成员，一个是 string，存储动物的名称(例如"Fido"或"Yogi")，另一个是整数成员 weight，包含该动物的重量(单位是磅)。该类还包含一个公共成员函数 who()，它可以显示一个消息，给出 Animal 对象的名称和重量。把 Animal 用作公共基类，派生两个类 Lion 和 Aardvark。再编写一个 main()函数，创建 Lion 和 Aardvark 对象("Leo"，400 磅；"Algernon"，50 磅)。为派生类对象调用 who()成员，说明 who()成员在两个派生类中是继承得来的。

2. 在 Animal 类中，把 who()函数的访问指定符改为 protected，但类的其他内容不变。现在修改派生类，使原来的 main()函数仍能工作。

3. 在上一题中，把基类成员 who()的访问指定符改为 public，但把 who()函数实现为每个派生类的成员，且在输出消息中显示派生类名。现在修改 main()函数，为每个派生类对象调用 who()的基类版本和派生类版本。

4. 定义一个 Person 类，它包含数据成员 age、name 和 gender。从 Person 中派生一个类 Employee，在新类中添加一个数据成员，存储个人的 number。再从 Employee 中派生一个类 Executive，每个派生类都应定义一个函数，来显示相关的信息(名称和类型，如"Fred Smith is an Employee")。编写一个 main()函数，生成两个数组，一个数组包含 5 个 Executive 对象，另一个数组包含 5 个一般的 Employee 对象，然后显示它们的信息。另外，调用从 Employee 类继承的成员函数，显示 Executive 的信息。

第 16 章　虚函数和多态性

多态性是面向对象编程的一个强大功能，在大多数 C++程序中都要用到。多态性需要使用派生类，本章的内容主要讨论与第 15 章介绍的继承相关的概念。

本章主要内容

- 多态性的概念，如何对类实现多态性操作
- 虚函数的概念
- 在类层次结构中需要虚拟析构函数的场合和原因
- 虚函数的默认参数值如何使用
- 纯虚函数的概念，如何声明纯虚函数
- 抽象类的概念
- 如何在类层次结构中强制转换类的类型
- 如何在运行期间确定对象指针的类型
- 成员的指针是什么，如何使用这些指针

16.1　理解多态性

多态性表示可以在各种面向对象语言中获得的一种特殊机制。由于多态性在 C++的实现方式，常常用虚函数调用来描述它。多态性总是要在调用函数时使用对象的指针或对象的引用。另外，多态性仅用于共享一个公共基类的类层次结构，所以能从一个类中派生另一个类就是多态性的基本条件。

实现多态性究竟要什么条件？下面举一个例子大致说明它是如何工作的，但首先需要理解基类指针的作用。

16.1.1　使用基类指针

第 15 章介绍过，派生类的对象包含一个基类对象。换言之，每个派生类对象也是一个基类对象。因此，可以使用基类指针来存储派生类对象的地址，甚至可以使用任何直接或间接基类的指针存储派生类对象的地址。

在图 16-1 中，Carton 类通过单一继承派生于 Box 基类，CerealPack 类通过多重继承派生于 Carton 和 Contents 基类。图 16-1 描述了在这样的类结构中，基类指针可以用于存储派生类对象的地址。

但反过来就不对了。例如，不能使用 CerealPack*类型的指针存储(直接或间接)基类对象的地址。这是符合逻辑的，因为基类没有描述完整的派生类对象。派生类对象总是包含每个基类的完整子对象，但每个基类只表示派生类对象的一部分。

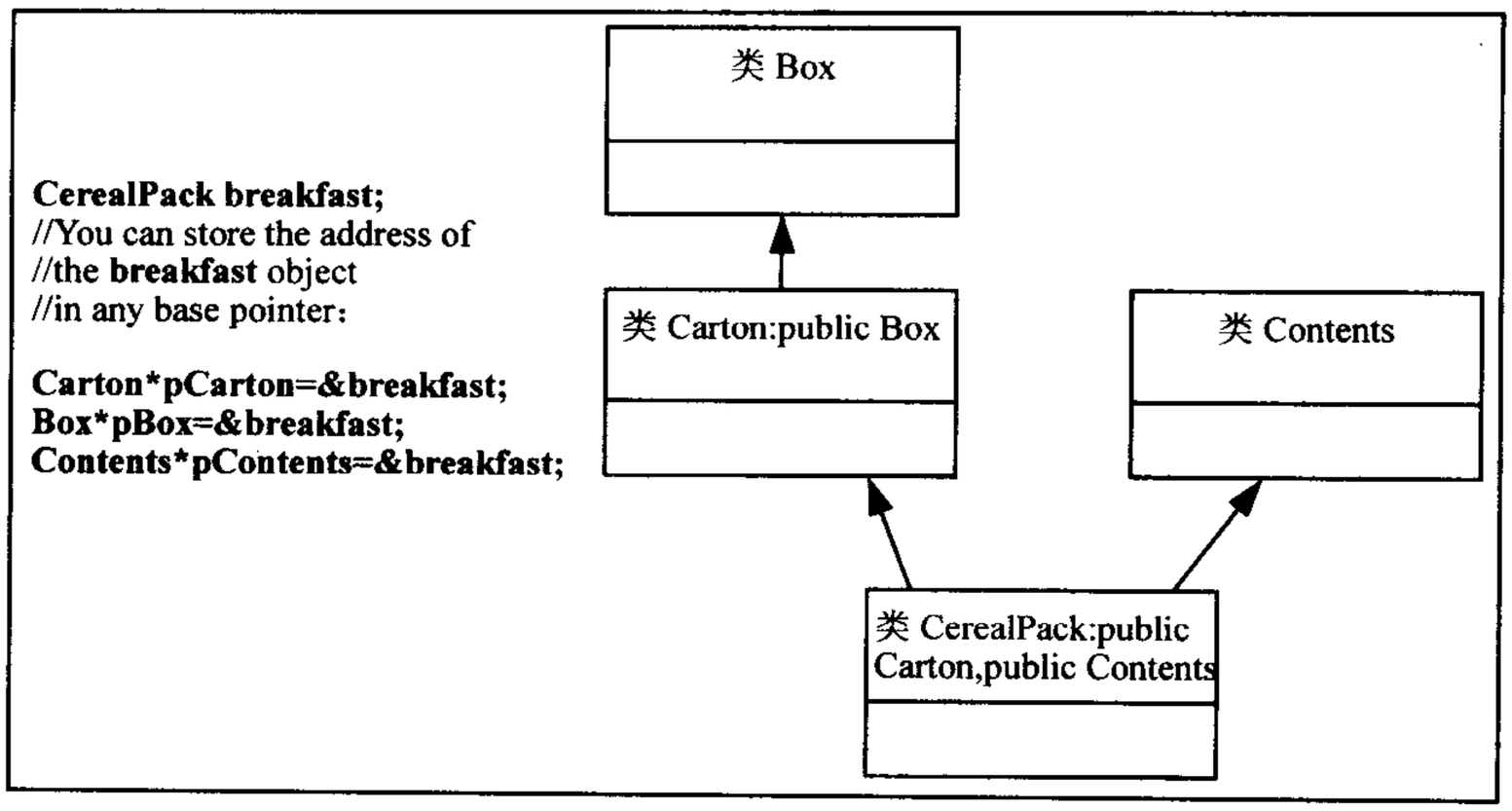


图 16-1　在基类指针中存储派生类对象的地址

下面看一个例子。假定从 Box 类中派生两个类，分别表示不同类型的容器，其特性如第 15 章所述。Carton 类定义如下所示：

```
class Carton: public Box {
    // Details of the class…
};
```

新类 ToughPack 有类似的定义：

```
class ToughPack: public Box {
    // Details of the class…
};
```

假定每个派生类型的体积的计算方法都不同。对于纸板做的 Carton 对象，考虑到材料的厚度，只需把体积减小一些即可。而对于 ToughPack 对象，就必须减去相当大的数，才能获得有用的内部容积，起到保护的作用。

有了这些类的定义后，声明并初始化一个指针：

```
Carton aCarton(10.0, 10.0, 5.0);
Box* pBox=&aCarton;
```

指针 pBox 指向 Box，它用 aCarton(其类型是 Carton)的地址进行初始化。这是可以的，因为 Carton 派生于 Box，包含一个 Box 类型的子对象。这个指针可用于存储 ToughPack 对象的地址，因为 ToughPack 类也派生于 Box：

```
ToughPack hardcase(12.0, 8.0, 4.0);
pBox=&hardcase;
```

在任意时刻，指针 pBox 都可以包含任何以 Box 为基类的类对象的地址。该指针在声明时的类型称为静态类型，pBox 的静态类型是“指向 Box”。因为 pBox 是指向基类的指针，所以它也具有动态类型，它会根据它指向的对象类型而变化。当 pBox 指向 Carton 对象时，其动态类型就是“指向 Carton 的指针”。当 pBox 指向 ToughPack 对象时，其动态类型就是“指向 ToughPack 的指针”。在 pBox 指向 Box 类型的对象时，其动态类型就与静态类型相同。

这就是多态性。在某些情况下，可以使用指针 pBox 调用在基类和每个派生类中定义的函

数。编译器会根据 pBox 的动态类型来选择调用哪个函数。下面的语句：

```
pBox->volume();
```

如果 pBox 包含 Carton 对象的地址，就可以使用这个语句调用 Carton 对象的 volume()函数。如果 pBox 指向 ToughPack 对象，该语句就调用 ToughPack 对象的 volume()函数。该指针也可以用于其他从 Box 派生出来的类。

表达式 pBox->volume()会根据 pBox 指向的对象执行不同的操作。更重要的是，编译器会在运行期间根据 pBox 指向的对象自动执行相应的操作，就像该指针有一个内置的 switch 语句，用于测试类型，选择要调用的函数。

这是一个极为强大的机制。我们常常不能事先确定要处理哪种类型的对象，即在设计期间或编译期间不能确定类型，只能在运行期间确定。而使用多态性可以轻松地解决这个问题。多态性一般用于交互式应用程序，输入的类型取决于用户一时的兴致。

例如，图形化应用程序需要绘制不同的图形：圆、直线、曲线等，就可以为每种图形类型定义一个派生类，这些类都有一个共同的基类 Shape。应用程序会在类型为“指向 Shape”的基类指针 pShape 中存储用户创建的对象地址，再用 pShape->draw();语句绘制相应的图形。该语句会根据指针所指向的对象调用对应图形的 draw()函数，因此，一个表达式就可以绘制任何类型的图形。

为了以这种方式操作，被调用的函数应是基类的一个成员。下面深入探讨继承函数的操作。

16.1.2 调用继承的函数

在介绍多态性的特性之前，需要先详细了解一下继承的成员函数的操作，以及它们与派生类成员函数的关系。为此，修改 Box 类，使之包含一个计算 Box 对象体积的函数，和另一个显示所得体积的函数。Box.h 和 Box.cpp 文件中的类如下所示：

```
// Box.h
#ifndef BOX_H
#define BOX_H

class Box {
  public:
    Box (double lengthValue = 1.0, double widthValue = 1.0,
                                   double heightValue = 1.0);

    // Function to show the volume of an object
    void showVolume() const ;

    // Function to calculate the volume of a Box object
    double volume() const;

  protected:
    double length;
    double width;
    double height;
};
```

```
#endif
```

Box.cpp 中的定义如下所示：

```
// Box.cpp
#include "Box.h"
#include <iostream>
using std::cout;
using std::endl;

// constructor
Box::Box(double lvalue, double wvalue, double hvalue) :
                 length(lvalue),width(wvalue) , height(hvalue) {}

// Output the volume
void Box::showVolume() const {
  cout <<"Box usable volume is "<< volume() <<endl;
}

// Calculate the volume
double Box::volume() const {
  return length * width * height;
}
```

这里不再需要把析构函数定义为默认，也不需要构造函数中的跟踪语句。以这种方式定义了 Box 类后，就可以调用 Box 对象的 showVolume()函数，显示 Box 对象的可用体积。这里使用的数据成员与以前相同(length、width 和 height)，它们都指定为 protected，所以可以在任意派生类的成员函数中访问。

接着把 Box 作为基类，定义 ToughPack 类。ToughPack 对象使用包装材料来保护其中的物品，其容积仅是基本 Box 对象的 85%。因此需要在派生类中定义另一个 volume()函数：

```
// ToughPack.h
#ifndef TOUGHPACK_H
#define TOUGHPACK_H

#include "Box.h"

class ToughPack : public Box {     // Derived class
  public:
    // Constructor
    ToughPack(double lengthValue, double widthValue, double heightValue);

    // Function to calculate volume of a ToughPack allowing 15% for packing
    double volume() const;
};
#endif
```

.cpp 文件包含下述定义：

```
// ToughPack.cpp
```

```
#include "ToughPack.h"

ToughPack::ToughPack(double lVal, double wVal, double hVal) :
                                                    Box(lVal, wVal, hVal) {}

double ToughPack::volume() const {
  return 0.85 * length * width * height;
}
```

在这个派生类中还可以有其他成员，但目前为了使例子简单一些，主要讨论继承的函数如何工作。派生类的构造函数在其初始化列表中调用了基类构造函数，以设置数据成员的值。在派生类的构造函数体中不需要任何语句。再用 volume()函数的新版本替代基类中的版本。在为 ToughPack 类的对象调用函数 showVolume()时，继承函数 showVolume()应调用 Volume()函数的派生类版本。下面看看其工作情况。

程序示例 16.1—— 使用继承的函数

创建一个 Box 对象和一个有相同尺寸的 ToughPack 对象，再验证是否计算出了正确的体积，就可以测试新的派生类。main()函数如下所示：

```
// Program 16.1 Behavior of inherited functions in a derived class
// File: prog16_01.cpp
#include <iostream>
#include "Box.h"                          // For the Box class
#include "ToughPack.h"                    // For the ToughPack class
using std::cout;
using std::endl;

int main() {
  Box myBox(20.0, 30.0, 40.0);            // Declare a base box
  ToughPack hardcase(20.0, 30.0, 40.0); // Declare derived box - same size

  cout << endl;
  myBox.showVolume();                     // Display volume of base box
  hardcase.showVolume();                  // Display volume of derived box

  return 0;
}
```

运行程序，结果令人失望：

```
Box usable volume is 24000
Box usable volume is 24000
```

例子的说明

派生类对象的容积应比基类对象小，但程序的结果显然与预期的不一样。下面看看是哪里出错了。showVolume()的第二次调用是给派生类的对象 ToughPack 调用的，但显然这没有考虑在内。ToughPack 对象的体积应是尺寸相同的基本 Box 对象的 85%。

问题是在这个程序中，在 showVolume()函数调用 Volume()函数时，编译器仅设置了一次，且设置为调用在基类中定义的 volume()函数。无论怎样调用 showVolume()，它都永远不会调用

volume()函数的 ToughPack 版本。

在执行程序之前，函数调用以这种方式固定下来，就称为函数调用的静态解析，或静态绑定，也可以使用术语“早期绑定”。在本例中，volume()函数在程序的编译期间绑定到showVolume()函数的调用上。每次调用 showVolume()时，都使用所绑定的基类 volume()函数。

注释：

如果设置相应的条件，这种解析也会发生在派生类 ToughPack 上。即给 ToughPack 类添加一个函数 showvolume()(它调用 volume())，volume()调用就静态解析到派生类函数上。

如果直接调用 ToughPack 对象的 volume()函数，会如何？下面进一步演示这个过程。添加语句，直接调用 ToughPack 对象的 volume()函数，再通过基类的指针来调用该函数：

```
cout << "hardcase volume is " << hardcase.volume() << endl;
Box *pBox=&hardcase;
cout << "hardcase volume through pBox is " << pBox->volume() << endl;
```

把这些语句放在 main()的 return 语句之前。运行程序，结果如下：

```
Box usable volume is 24000
Box usable volume is 24000
hardcase volume is 20400
hardcase volume through pBox is 24000
```

结果很容易看懂。派生类对象 hardcase 的 volume()函数调用了派生类的 volume()函数，这正是我们希望的。但是，通过基类指针 pBox 的调用解析为 volume()的基类版本，尽管 pBox 包含了 hardcase 的地址。换言之，这两个调用都是静态解析的。编译器把这些调用执行为：

```
cout << "hardcase volume is " << hardcase. ToughPack::volume() << endl;
Box *pBox=&hardcase;
cout << "hardcase volume through pBox is " << pBox-> Box::volume() << endl;
```

函数通过指针的静态调用仅取决于指针的类型，不取决于它指向的对象。指针 pBox 的类型是“指向 Box”，因此使用 pBox 进行的静态调用都仅调用 Box 的函数成员。

注意：

通过静态解析的基类指针来调用函数，都会调用基类的函数。

本例要解决的问题是，在执行程序时，在要解析的任意给定实例中会使用哪个 volume()函数。如果用派生类对象来调用 showVolume()，它就应调用派生类的 volume()函数，而不是基类的 volume()函数。同样，如果通过基类指针来调用 volume()函数，就应调用该指针指向的对象的 volume()函数。这种操作称为动态绑定或后期绑定。

程序还没有实现这个目标，因为必须告诉编译器，Box 中的 volume()函数和派生于 Box 的类中的 volume()函数是不同的，而且对该函数的调用是动态解析的。此时需要把该 volume()函数指定为虚函数，这样才能使用虚函数调用。

16.1.3 虚函数

把一个函数声明为基类中的虚函数，就是告诉编译器，在派生于这个基类的任何类中，该函数都是动态绑定。虚函数在基类中声明时使用关键字 virtual。如图 16-2 所示。

在基类中声明为 virtual 的函数，在所有的从基类中派生的类(直接或间接)中都是虚函数。为了获得多态性，每个派生类都可以执行虚函数的自有版本(但这不是强制的)。使用基类对象的指针或引用就可以调用虚函数。图 16-2 演示了通过指针调用虚函数是如何动态解析的。基类指针用于存储派生类对象的地址。它可以指向图中三个派生类中的任意一个类对象，当然也可以指向基类对象。调用哪个 volume()函数取决于执行调用时指针指向的对象类型。

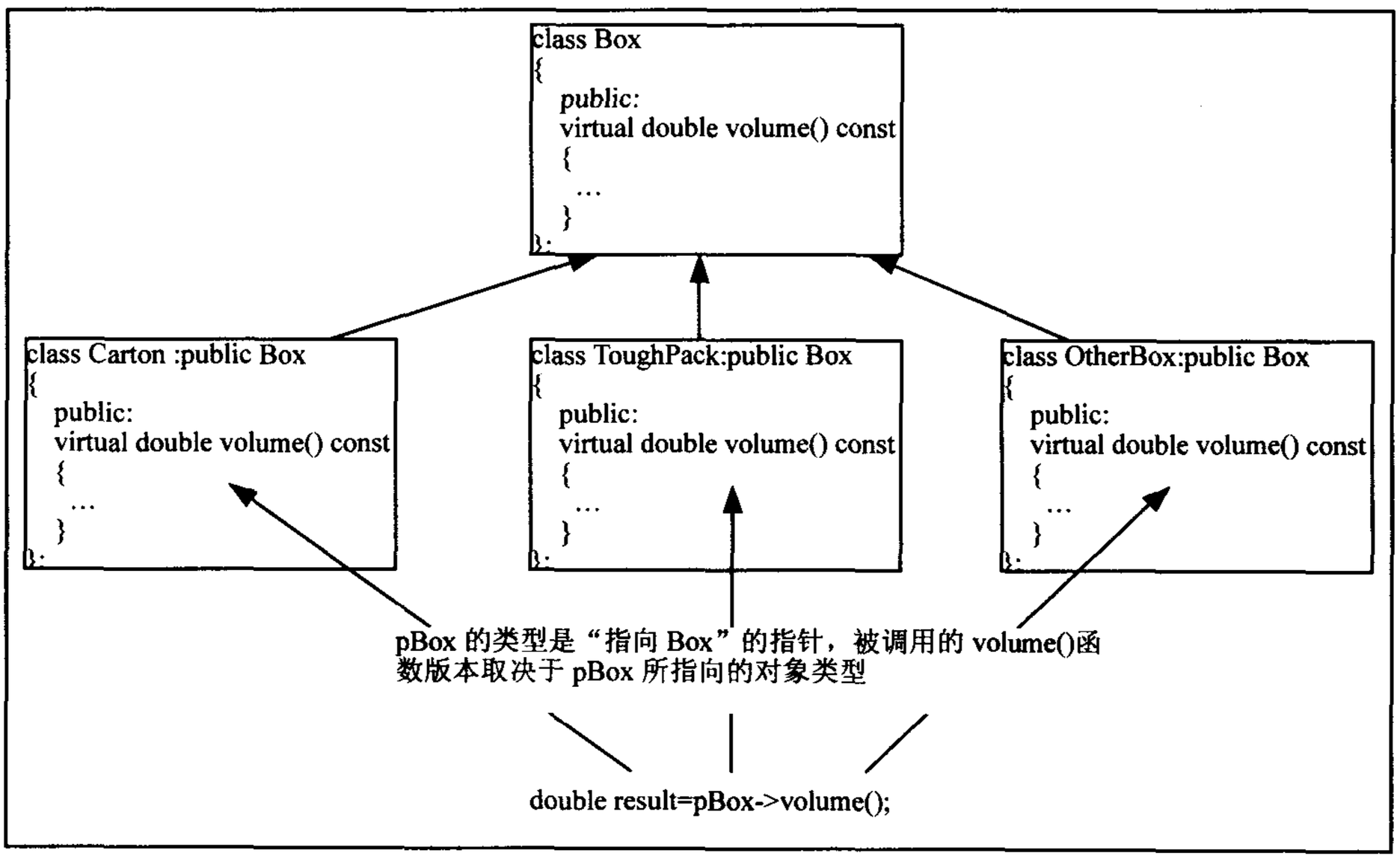


图 16-2　调用虚函数

注意：

把类描述为多态性，意味着它是一个至少包含一个虚函数的派生类。

注意使用对象调用虚函数总是进行静态解析。只有通过指针或引用调用虚函数，才会进行动态解析。也就是说，虚函数只是一个工具。

程序示例 16.2—— 使用虚函数

为了使例子像希望的那样工作，需要对 Box 类作很小的改动。在该类的 volume()函数的声明中添加关键字 virtual：

```
class Box {
  public:
    Box(double lengthValue = 1.0, double widthValue = 1.0,
                                          double heightValue = 1.0);

    // Function to show the volume of an object
```

```
    void showVolume() const;

    // Function to calculate the volume of a Box object
    virtual double volume() const;

protected:
    double length;
    double width;
    double height;
};
```

提示：

不需要在函数定义中添加关键字 virtual，这会产生一个错误。

为了使程序更有趣一些，下面在一个新类 Carton 中实现 volume()函数。首先是类定义：

```
// Carton.h
#ifndef CARTON_H
#define CARTON_H

#include <string>
#include "Box.h"
using std::string;

class Carton : public Box {
  public:
    // Constructor explicitly calling the base constructor
    Carton(double lv, double wv, double hv, string material = "Cardboard");

    // Copy constructor
    Carton(const Carton& aCarton);

    // Destructor
    ~Carton();

    // Function to calculate the volume of a Carton object
    double volume() const;

  private :
    string* pMaterial;
};

#endif
```

.cpp 文件中的定义如下：

```
// Carton.cpp
#include "Carton.h"

Carton: : Carton (double lv, double wv, double hv, string material) : Box(lv, wv, hv){
  pMaterial = new string (material );
}
```

```
Carton::Carton(const Carton& aCarton) {
  length = aCarton.length;
  width = aCarton.width;
  height = aCarton.height;
  pMaterial = new string( * aCarton. pMaterial);
}

Carton::~Carton() {
  delete pMaterial;
}

double Carton::volume() const {
  double vol = (length - 0.5) * (width - 0.5) * (height - 0.5);
  return vol > 0.0 ? vol : 0.0;
}
```

Carton 的 volume()函数假定材料的厚度是 0.25，所以要在每个尺寸中减去 0.5，作为纸板箱的边长。如果因某种原因，Carton 对象在创建时其尺寸小于 0.5，体积就会是一个负值，此时，该对象的体积设置为 0。

这里还要使用在程序示例 16.1 中定义的 ToughPack 类。修改上一个例子中的 main()函数，使用 Carton 对象，并用指针 pBox 调用 showVolume()函数：

```
// Program 16.2 Using virtual functions  File: prog16_02.cpp
#include <iostream>
#include "Box.h"                              //  For  the Box class
#include "ToughPack.h"                        //  For the ToughPack class
#include "Carton.h"                           //  For the Carton class
using std::cout;
using std::endl;

int main() {
  Box myBox(20.0, 30.0, 40.0);                // Declare a base box
  ToughPack hardcase(20.0, 30.0, 40.0);       // Declare derived box - same size
  Carton aCarton(20.0,   30.0,   40.0);       // A different kind of derived box

  cout << endl;
  myBox.showVolume();                        // Display volume of base box
  hardcase.showVolume();                     // Display volume of derived box
  aCarton.showVolume();                      // Display volume of derived box
  cout << endl;

// Now try using a base pointer for the Box object
Box* pBox = &myBox;                          // Points to type Box
cout << "myBox volume through pBox is " << pBox->volume() << endl;
pBox-> showVolume();
cout << endl;

// Now try using a base pointer for the ToughPack object
pBox = &hardcase;                            // Points to type ToughPack
cout << "hardcase volume through pBox is " << pBox->volume() << endl;
```

```
    pBox->showVolume();
    cout << endl;

    // Now try using a base pointer for the Carton object
    pBox = &aCarton;                          // Points to type Carton
    cout << "aCarton volume through pBox is " << pBox->volume() << endl;
    pBox->showVolume();

    return 0;
}
```

现在重新编译例子，运行后，输出如下所示：

```
Box usable volume is 24000
Box usable volume is 20400
Box usable volume is 22722.4

myBox volume through pBox is 24000
Box usable volume is 24000

hardcase volume through pBox is 20400
Box usable volume is 20400

aCarton volume through pBox is 22722.4
Box usable volume is 22722.4
```

例子的说明

在基类定义中，用于函数 volume()的关键字 virtual 足以确定，该函数在派生类中的所有声明都会理解为虚函数。也可以给派生类函数使用关键字 virtual，如图 16-2 所示。

注释：

在这个例子中，volume()的声明省略了关键字 virtual，说明不需要使用它。但是，最好在派生类的所有虚函数的声明中都使用关键字 virtual，因为这会使他人在查看派生类定义中，很清楚地看出该函数是虚函数，并且会动态链接。

程序现在的结果与希望的相同。对函数 showVolume()的第一次调用用于 Box 对象 myBox，如下所示：

```
myBox.showVolume();       //Display volume of base box
```

这个语句调用 volume()的基类版本，因为 myBox 的类型是 Box。对函数 showVolume()的第二次调用用于 ToughPack 对象 hardcase，如下所示：

```
hardcase.showVolume();   //Display volume of derived box
```

这个语句调用在 Box 类中定义的 showVolume()函数—— 实际上，也没有其他版本的 showVolume()。该函数继承为 ToughPack 类的一个公共成员，所以以这种方式调用它是没有问题的。但是，在 showVolume()中，对 volume()的调用解析为在派生类中定义的版本，因为 volume()是一个虚函数，计算出 ToughPack 对象的体积。

对函数 showVolume()的第三次调用用于 Carton 对象：

```
aCarton.showVolume();    //Display volume of derived Box
```

函数 showVolume()在 Carton 中继承，对 volume()的调用解析为 Carton 类的版本，所以得到了该对象的正确体积。

接着，使用指针 pBox 直接调用 volume()函数，再通过 showVolume()函数间接调用。指针首先包含 Box 对象 myBox 的地址，然后依次包含两个派生类对象的地址。每个对象的输出说明，在每种情况下，编译器都自动选择了正确的 volume()函数的版本，这样就对多态性有了清晰的认识。

1. 要求虚函数的情形

对于执行为“虚拟”的函数，在任意派生类和基类中声明和定义它时，必须要有相同的名称和参数列表。而且，如果把基类函数声明为 const，就必须把派生类函数也声明为 const。一般情况下，函数在派生类中的返回类型也必须与基类中的相同，但当基类中的返回类型是类类型的指针或引用时例外，在这种情况下，虚函数的派生类版本可以返回更特殊的类型的指针或引用。这里不深入探讨这个问题，但万一在其他地方遇到这种情形，这些返回类型使用的技术术语是“协变性”。

虚函数定义的规则是，如果要在派生类中对虚函数使用与基类不同的参数，虚函数机制就不起作用。此时，派生类中的函数会利用在编译期间建立和固定的静态绑定。如果基类函数是 const，但没有把派生类函数也声明为 const，则也会出现静态绑定的情形。

要测试这个规则，可以在 Carton 类中删除 volume()声明中的 const 关键字，再次运行该例子。则 Carton 中的 volume()函数不再与 Box 中声明的虚函数匹配，所以派生类 volume()函数不是虚函数。结果是，解析是静态的，通过基类指针对 Carton 对象调用 volume()函数，甚至间接通过 showVolume()函数调用 volume()函数，都会调用基类版本。

如果在派生类中，函数的名称和参数列表与在基类中声明的虚函数相同，返回类型也必须与虚函数一致。如果不一致，派生类函数就不会编译。另一个限制是，虚函数不能是模板函数(模板函数详见第 9 章)。

2. 虚函数和类层次结构

如果要通过基类指针把函数用作虚函数，就必须在基类中把它声明为 virtual。在基类中，可以声明任意多个虚函数，但在有若干层的类层次结构中，并不是所有的虚函数都需要在最一般的基类中声明。如图 16-3 所示。

在一个类中把某函数声明为虚函数后，该函数在直接或间接继承自该类的所有派生类中都是虚函数。例如在图 16-3 中，所有派生于 Box 类的子类都继承了 volume()函数的虚拟特性。通过 Box*类型的指针 pBox，可以调用任何类的 volume()函数，因为该指针可以包含层次结构中任何类对象的地址：

```
double result=pBox-> volume();        //Call for any class in the hierarchy
```

由于 Crate 类没有声明虚函数 volume()，因此 Crate 对象就会调用从 Carton 继承来的 volume()版本。该函数继承为虚函数，因此可以进行多态调用。

Carton*类型的指针 pCarton 也可以用于调用 volume()函数，但仅能为 Carton 类的对象和以 Carton 为基类派生的两个类 Crate 和 Packet 的对象调用该函数：

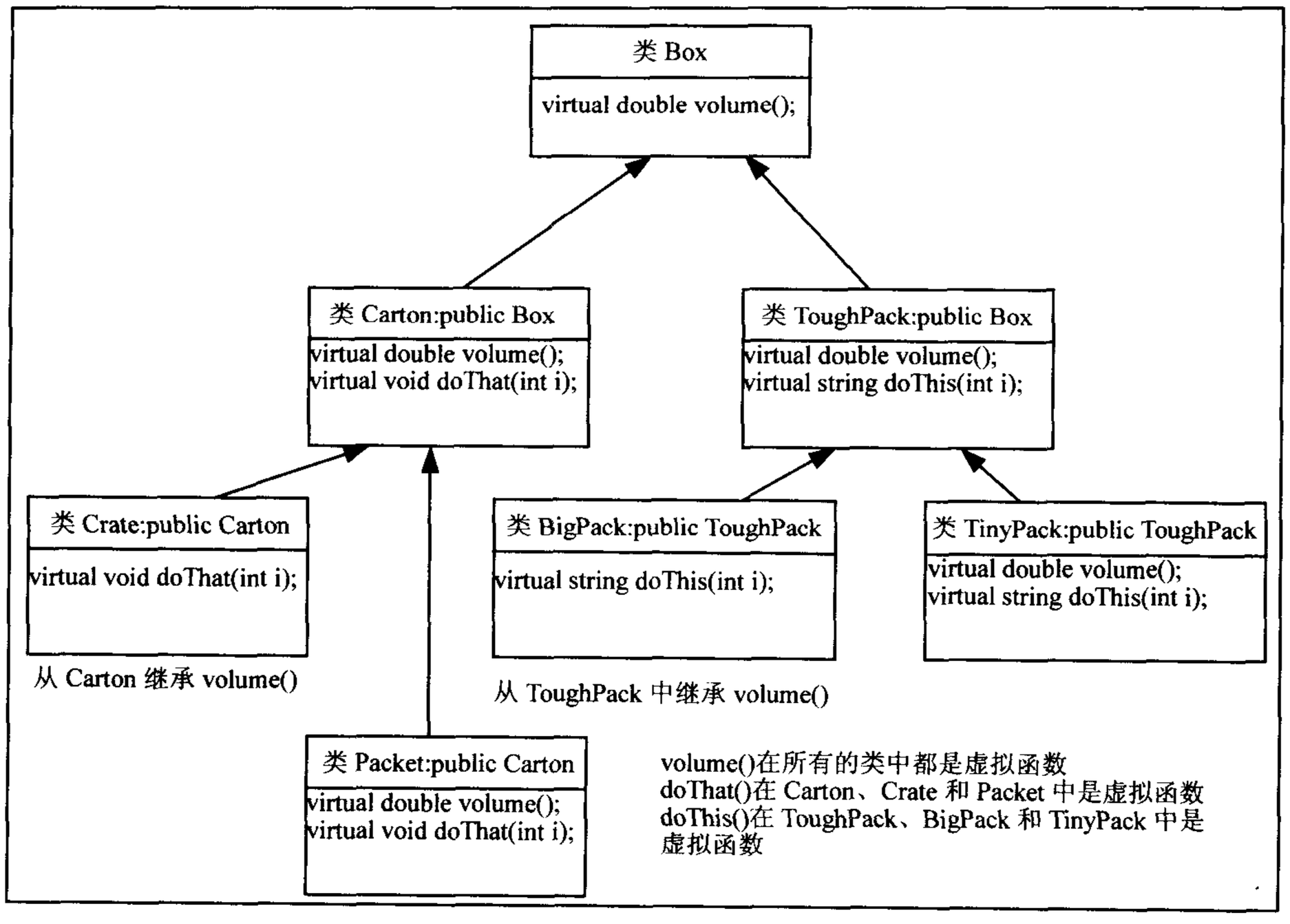


图 16-3　层次结构中的虚函数

```
result=pCarton-> volume();       //Call for Carton, Crate or Packet
```

Carton 类和派生于它的类也包含虚函数 doThat()。用 Carton*类型的指针也可以多态调用该函数：

```
pCarton->doThat(12);             //Call for Carton, Crate, or Packet
```

注意不能使用指针 pBox 为这些类调用 doThat()，因为 Box 类不包含函数 doThat()。

同样，也可以用基类指针 pToughPack 为 ToughPack、Bigpack 和 TinyPack 类型的对象调用虚函数 doThis()：

```
string answer=pToughPack-> doThis(3);  // for ToughPack,  Bigpack, or TinyPack
```

当然，该指针也可以用于为这些类的对象调用函数 volume()。

3. 访问指定符和虚函数

虚函数在派生类中声明时，所用的访问指定符可以不同于其在基类中声明时的访问指定符。在通过指针调用虚函数时，基类中的访问指定符将确定该函数是否可以在派生类中访问。如果虚函数在基类中声明为 public，则无论它在派生类中的访问指定符是什么，都可以通过基类的指针(或引用)为所有的派生类调用该函数。

下面修改前面的例子，演示这个主题。

程序示例 16.3—— 访问指定符对虚函数的影响

修改前一个例子中的 ToughPack 类定义，把 volume()函数改为 protected，并在它的声明中

添加关键字 virtual：

```
class ToughPack : public Box {     // Derived class
  public:
    // Constructor
    ToughPack(double lengthValue, double widthValue, double heightValue);

  protected:
    // Function to calculate volume of a ToughPack allowing 15% for packing
    virtual double volume() const;
};
```

还要略微修改一下 main()函数：

```
// Program 16.3 Access specifiers and virtual functions   File: prog16_03.cpp
#include <iostream>
#include "Box.h"                          // For the Box class
#include "ToughPack.h"                    // For the ToughPack class
#include "Carton.h"                       // For the Carton class
using std::cout;
using std::endl;

int main() {
  Box myBox(20.0, 30.0, 40.0);            // Declare a base box
  ToughPack hardcase(20.0, 30.0, 40.0)    // Declare derived box - same
size
  Carton aCarton(20.0, 30.0, 40.0); // A different kind of derived
box

  cout << endl;
  myBox.showVolume();      // Display volume of base box
  hardcase.showVolume();      // Display volume of derived
box
  aCarton.showVolume();       // Display volume of derived
box
  cout << endl;

// Uncomment he following statement for an error
// cout << "hardcase volume is * << hardcase.volume() << endl;

  // Now try using a base pointer for the Box object
Box *pBox = &myBox;                       // Points to type Box
cout << "myBox volume through pBox is " << pBox->volume() << endl;
pBox->showVolume();
cout << endl;

// Now try using a base pointer for the ToughPack object
pBox = &hardcase;       // Points to type ToughPack
cout << "hardcase volume through pBox is " << pBox->volume() << endl;
pBox->showVolume();
cout << endl;
```

```
  // Now try using a base pointer for the Carton object
  pBox = &aCarton;       // Points to type Carton
  cout << "aCarton volume through pBox is " << pBox->volume() << endl;
  pBox->showVolume();

  return 0;
}
```

这些代码的运行结果与前一个例子完全相同。

例子的说明

尽管 volume()函数在 ToughPack 类中声明为 protected，仍可以通过从 Box 类中继承来的 showVolume()函数为 hardcase 对象调用 volume()函数。也可以通过基类指针 pBox 直接调用它。但是，如果去掉用 hardcase 对象直接调用 volume()函数的那行代码的注释，代码就不会编译。

这里的问题是调用是动态解析的还是静态解析的。在使用类对象时，调用将进行静态解析(即由编译器解析)，因为 volume()函数在 ToughPack 类中声明为 protected，使用 hardcase 对象的调用就不会编译。其他调用则在程序执行时解析—— 它们是多态性调用。在这种情况下，虚函数在基类中的访问指定符会被其所有的派生类继承。它与派生类中的访问指定符无关，派生类中的显式访问指定符仅影响静态解析的调用。

16.1.4 虚函数中的默认参数值

因为默认值是在编译期间处理，所以在虚函数的参数中使用默认值会得到意想不到的结果。如果虚函数在基类的声明时带有默认参数值，则通过基类指针调用该函数时，就总是从函数的基类版本中接受默认的参数值。而该函数在派生类版本中的默认参数值就不起作用。

下面修改前一个例子，在 volume()函数中包含一个默认参数值，以演示这个过程。

程序示例 16.4—— 默认参数值

在文件 Box.cpp 中修改 volume()函数的定义：

```
// Box.cpp
#include "Box.h"
#include <iostream>
using std::cout;
using std::endl;

// constructor
Box::Box(double lvalue, double wvalue, double hvalue) :
                        length(lvalue), width(wvalue), height(hvalue){}

// Output the volume
void Box::showVolume() const {
  cout <<"Box usable volume is "<< volume() <<endl;
}

// Calculate the volume
double Box::volume(const int i) const {
```

```
  cout<<"Parameter = "<<i<<endl;
  return length*width*height;
}
```

还需要在 Box 类中调整函数成员的声明：

```
// Box.h
#ifndef BOX_H
#define BOX_H

class Box {
  public:
    Box(double lengthValue = 1.0, double widthValue = 1.0,
                                          double heightValue = 1.0);

    // Function to show the volume of an object
    void showVolume ( ) const ;

    // Function to calculate the volume of a Box object
    virtual double volume(const int i=5) const;

  protected:
    double length;
    double width;
    double height;
};
#endif
```

这里的参数值仅用于演示默认值是如何赋予的。以同样的方式修改 Carton，但默认的参数值设置为 50，Carton.h 如下所示：

```
// Carton.h
#ifndef CARTON_H
#define CARTON_H

#include  <string>
#include   "Box.h"
using std::string;

class Carton  :  public  Box {
  public:
// Constructor explicitly calling the base constructor
Carton(double lv, double wv, double hv, string material = "Cardboard");

// Copy constructor
Carton(const Carton& aCarton);

// Destructor
~Carton();

// Function to calculate the volume of a Carton object
virtual double volume(const int i = 50) const;
```

```
  private :
    string* pMaterial;
};

#endif
```

在 ToughPack 类中，把默认参数值设置为 500。把 volume()函数的访问指定符恢复为 public，修改后，ToughPack.h 文件的内容如下：

```
// ToughPack.h
#ifndef TOUGHPACK_H
#define TOUGHPACK_H

#include "Box.h"

class ToughPack : public Box {                    // Derived class
  public:
    // Constructor
    ToughPack(double lengthValue, double widthValue, double heightValue);

    // Function to calculate volume of a ToughPack allowing 15% for packing
    virtual double volume(const int i = 500) const;
};
#endif
```

ToughPack.cpp 文件包含下述定义：

```
// ToughPack.cpp
#include "ToughPack.h"
#include <iostream>
using std::cout;
using std::endl;

ToughPack::ToughPack(double lVal, double wVal, double hVal) :
                                                  Box(lVal, wVal, hVal){}

double ToughPack::volume() const {
  cout<<"Parameter = " << i << endl;
  return 0.85 * length * width * height;
}
```

对类定义进行了这些修改后，就可以在修改过的 main()函数中试验它们了。在 main()函数中，去掉为 hardcase 对象直接调用 volume()成员的那行代码的注释，代码如下所示：

```
// Program 16.4 Default parameter values in virtual functions
// File: prog16_04.cpp
#include <iostream>
#include "Box.h"           // For the Box class
#include "ToughPack.h"     // For the ToughPack class
#include "Carton.h"        // For the Carton class
using std::cout;
using std::endl;
```

```
int main() {
  Box myBox(20.0, 30.0, 40.0);    // Declare a base box
  ToughPack hardcase(20.0, 30.0, 40.0)   // Declare derived box - same size
  Carton aCarton(20.0, 30.0, 40.0);      // A different kind of derived box

  cout << endl;
  myBox.showVolume();                    // Display volume of base box
  hardcase.showVolume();                 // Display volume of derived box
  aCarton.showVolume();                  // Display volume of derived box
  cout << endl;

  cout << "hardcase volume is *<< hardcase.volume() << endl;

  // Now try using a base pointer for the Box object
  Box *pBox = &myBox;                    // Points to type Box
  cout << "myBox volume through pBox is " << pBox->volume() << endl;
  pBox->showVolume();
  cout << endl;

  // Now try using a base pointer for the ToughPack object
  pBox = &hardcase;      // Points to type ToughPack
  cout << "hardcase volume through pBox is " << pBox->volume() << endl;
  pBox->showVolume();
  cout << endl;

  // Now try using a base pointer for the Carton object
  pBox = &aCarton;       // Points to type Carton
  cout << "aCarton volume through pBox is " << pBox->volume() << endl;
  pBox->showVolume();

  return 0;
}
```

结果如下所示：

```
Parameter = 5
Box usable volume is 24000
Parameter = 5
Box usable volume is 20400
Parameter = 5
Box usable volume is 22722.4

Parameter = 500
hardcase volume is 20400
Parameter = 5
myBox volume through pBox is 24000
Parameter = 5
Box usable volume is 24000

Parameter = 5
hardcase volume through pBox is 20400
```

```
Parameter = 5
Box usable volume is 20400

Parameter = 5
aCarton volume through pBox is 22722.4
Parameter = 5
Box usable volume is 22722.4
```

例子的说明

在调用 volume()函数的每个实例中，默认值的结果都是基类函数的默认值，但有一个例外，即使用 hardcase 对象直接调用 volume()成员，这个调用是静态解析的，所以使用了 ToughPack 类的默认参数值。其他调用都是动态解析的，所以使用基类的默认值。

16.1.5 通过引用来调用虚函数

也可以通过引用来调用虚函数，引用参数是应用多态性的一个强大工具。通过引用变量来调用虚函数与通过指针来调用是一样的，因为引用变量仅初始化一次，只能为所引用的对象调用虚函数。但函数的引用参数是另外一回事。

假定定义一个函数，它的一个参数是基类的引用。可以把派生类对象作为一个参数传送给该函数。在该函数中，可以使用引用参数调用虚函数。在该函数执行时，会自动为所传送的对象选择合适的虚函数。下面修改程序示例 16.2 中的函数 main()，调用一个参数为“引用 Box”类型的函数。

程序示例 16.5—— 对虚函数使用引用

在这个例子中，要添加一个新函数 showVolume()，作为一个独立的全局函数，它输出对象的体积。可以给它传送引用参数，为对象调用函数 showVolume()。该函数的定义如下所示：

```
void showVolume(const Box& rBox) {
  rBox.showVolume();
}
```

在 main()中，可以用一些派生类参数来调用这个函数，看看它是如何工作的。对类定义的惟一改动是 volume()成员函数，在该成员函数的定义中，应删除 i 参数和输出 i 的那行代码。

对包含 main()的源文件修改如下：

```
// Program 16.5 Using virtual functions through a reference to the base class
// File: prog16_05.cpp
#include <iostream>
#include "Box.h"           // For the Box class
#include "ToughPack.h"     // For the ToughPack class
#include "Carton.h"        // For the Carton class
using std::cout;
using std::endl;

void showVolume(const Box& rBox);          // Prototype for global function

int main() {
```

```
Box myBox(20.0, 30.0, 40.0);              // Declare a base box
ToughPack hardcase(20.0, 30.0, 40.0);  // Declare derived box - same size
Carton aCarton(20.0, 30.0, 40.0);       // A different kind of derived box

cout << endl;
showVolume(myBox);          // Display volume of base box
showVolume(hardcase);       // Display volume of derived box
showVolume(aCarton);        // Display volume of derived box

// Lines deleted

  return 0;
}

// Global function to display the volume of a box
void showVolume(const Box& rBox) {
  rBox.showVolume();
}
```

运行这个程序，结果如下：

```
Box usable volume is 24000
Box usable volume is 20400
Box usable volume is 22722.4
```

例子的说明

在函数 main()中，创建了一个基类对象 myBox 和两个不同的派生类对象 hardcase 和 aCarton。接着把这些对象作为参数，调用 showVolume()全局函数。从输出中可以看出，在每次调用时都使用了正确的 volume()函数，证明多态性可以通过引用参数正确工作。

每次调用函数时，引用参数都用传送为参数的对象进行初始化。由于参数是基类的一个引用，编译器就会在运行期间绑定虚函数 volume()。如果把参数指定为派生类的引用，调用就进行静态解析，因为它已被完全确定了。只有通过基类引用来调用虚函数，才会进行动态绑定。

16.1.6 调用虚函数的基类版本

通过派生类对象的指针或引用调用虚函数的派生类版本是很简单的—— 该调用是动态进行的。但是，在相同的情形下，如何为派生类对象调用虚函数的基类版本？

Box 类可以说明为什么需要这样的调用。在 Carton 或 ToughPack 对象中计算总体积的损失是很有用的，而计算该损失的一种方法是计算 volume()函数的基类版本和派生类版本所得结果之差。

使用类名和作用域解析运算符来指定要使用的函数，就可以强制静态调用虚函数的基类版本。假定有一个指针 pBox 定义如下：

```
Carton aCarton(40.0, 30.0, 20.0);
Box* pBox=&aCarton;
```

下面的语句就会计算出 Carton 对象的总体积损失：

```
double difference=pBox->Box::volume()-pBox->volume();
```

表达式 pBox->Box::volume()调用 volume()函数的基类版本。类名和作用域解析运算符标识了特定的 volume()函数，所以这是在编译期间静态解析的调用。

注释：

使用作用域解析运算符，可以调用任何成员函数的基类版本，如果访问指定符允许访问该函数的话。

不能使用这个技术在通过基类指针进行的调用中选择特定的派生类函数版本。表达式 pBox-> Carton :: volume()不会编译，因为 Carton :: volume()不是 Box 类的成员。通过指针进行的函数调用要么是该指针所指向的类函数成员的静态调用，要么是对虚函数的动态调用。

通过派生类的对象调用虚函数的基类版本也很简单。可以使用静态强制转换，把派生类对象转换为基类对象，再使用该转换结果来调用基类函数。下面的语句也可以计算 aCarton 对象的体积损失：

```
double difference=static_cast<Box>(aCarton).volume() - aCarton.volume();
```

这个语句中的两个调用都是静态解析的。把 aCarton 强制转换为 Box，会得到一个 Box 类型的对象，所调用的函数就变成 volume()的 Box 版本。使用对象调用虚函数总是静态解析的。

16.1.7 在指针和类对象之间转换

如果程序包含派生类的指针，就可以把指针隐式转换为基类指针，该基类指针可以是直接基类指针，也可以是间接基类指针。例如，下面声明一个 Carton 对象的指针：

```
Carton* pCarton =new Carton(30, 40, 10);
```

可以把这个指针隐式转换为 Carton 的直接基类指针(Box 是 Carton 的直接基类)：

```
Box* pBox = pCarton;
```

结果是“指向 Box”的指针，它初始化为指向新的 Carton 对象。还可以把派生类的指针隐式转换为间接基类的指针。假定有如图 16-4 所示的类层次结构。

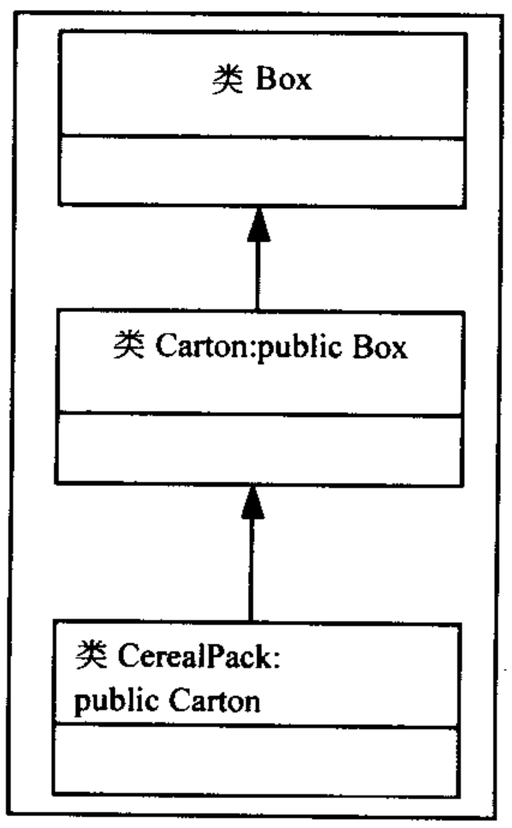


图 16-4 类层次结构

其中，Box 是 Carton 的直接基类，也是 CerealPack 的间接基类。下面的语句：

```
Box* pBox = pCerealPack;
```

这个语句把 pCerealPack 中的地址从“指向 CerealPack”的指针类型转换为“指向 Box”的指针类型。如果需要指定显式转换，可以使用 static_cast<>()运算符：

```
Box* pBox = static_cast<Box*>(pCerealPack);
```

编译器通常可以加速这个强制转换过程，因为它可以确定 Box 是 CerealPack 的基类。因为 CerealPack 对象包含 Box 对象，所以这种转换是可行的。如果 Box 类不能访问，或 Box 类是虚拟基类，就不允许进行这种转换，但编译器应能指出这一点。图 16-5 显示了指针从最特殊的派生类 CerealPack 沿着层次结构向上进行的所有静态转换。

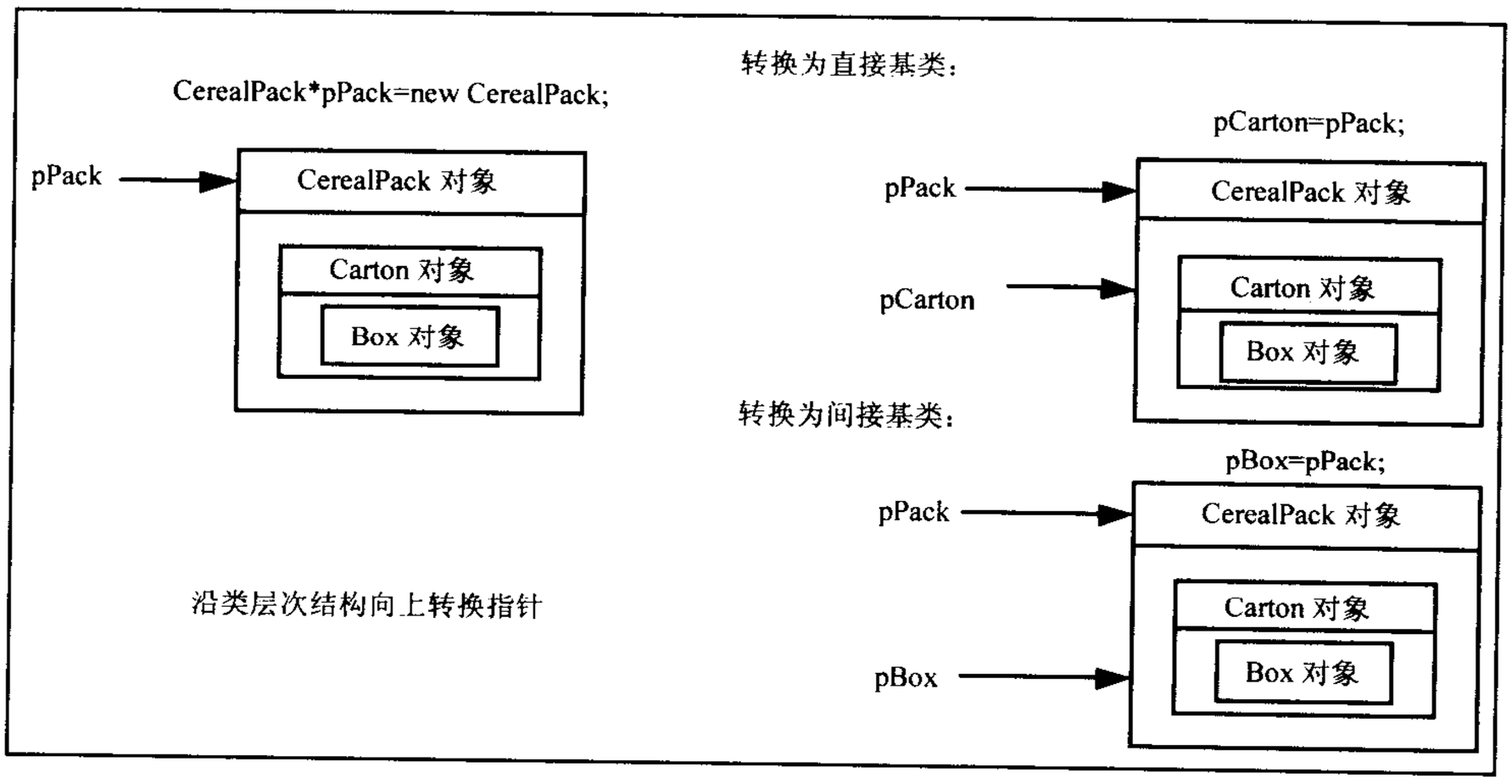


图 16-5　沿着类层次结构向上转换指针

如图 16-5 所示，每种转换的结果都是对应于目标类型的子对象的指针。在把指针强制转换为类类型时，很容易出现混乱。类类型的指针只能指向该类的对象，或派生类的对象，不能指向其他对象。例如，指针 pCarton 可以包含 Carton 对象的地址(它可以是 CerealPack 对象的一个子对象)或 CerealPack 对象的地址，但不能包含 Box 对象的地址。这是因为 CerealPack 对象是 Carton 的一种特殊类型，但 Box 对象不是。

有时还可以进行相反方向的强制转换。将指针沿着类层次结构向下进行强制转换，即从基类转换为派生类，这与从派生类转换为基类是不同的，因为强制转换是否可行，取决于基类指针指向什么对象。要把基类指针 pBox 静态强制转换为派生类指针 pCarton，基类指针必须指向 Carton 对象的 Box 子对象。否则，强制转换的结果就是不定的。如图 16-6 所示。

图 16-6 显示了静态转换指针 pBox(它包含对象 Carton 的地址)的过程。强制转换为 Carton*类型是可行的，因为该对象的类型是 Carton。但转换为 CerealPack*类型的结果是不定的，因为不存在这种类型的对象。

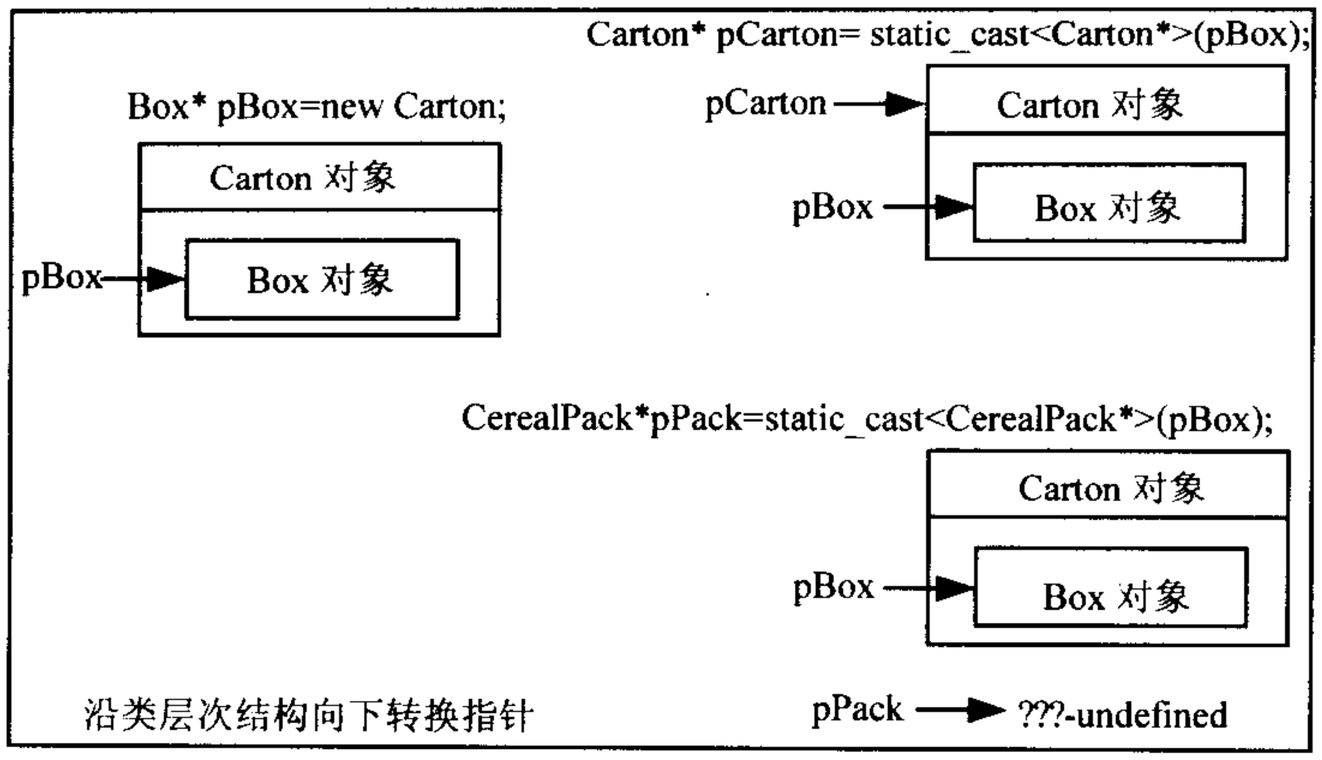


图 16-6　沿着类层次结构向下转换指针

如果对其合法性有所怀疑，就不应使用静态强制转换。沿着类层次结构向下转换指针是否成功，取决于指针是否包含目标类型的对象地址。静态强制转换不会对指针进行这种检查，如果要在不知道指针指向哪个对象的情况下进行这种静态强制转换，就可能得到不定的结果。因此，在沿着类层次结构向下转换时，需要用另外一种方式来进行，即强制转换可以在运行期间检查指针。

16.1.8　动态强制转换

动态强制转换在运行期间进行。要指定动态强制转换，应使用 dynamic_cast<>()运算符。这个运算符只能应用于多态类类型的指针和引用，即至少包含一个虚函数的类类型。原因是只有指向多态类类型的指针，才包含 dynamic_cast<>()运算符检查强制转换是否有效所需的信息。这个运算符专门用于转换类层次结构中类类型的指针或引用。

注意要强制转换的类型必须是同一类层次结构中的类的指针或引用。dynamic_cast<>()运算符不能用于其他情形。对于该运算符，首先介绍如何动态转换指针。

1. 动态转换指针

动态强制转换有两种基本类型。第一种是沿着类层次结构向下进行强制转换，即从直接或间接基类的指针转换为派生类型的指针。这称为 downcast。第二种是跨类层次结构的强制转换，这称为 crosscast。这两种强制转换如图 16-7 所示。

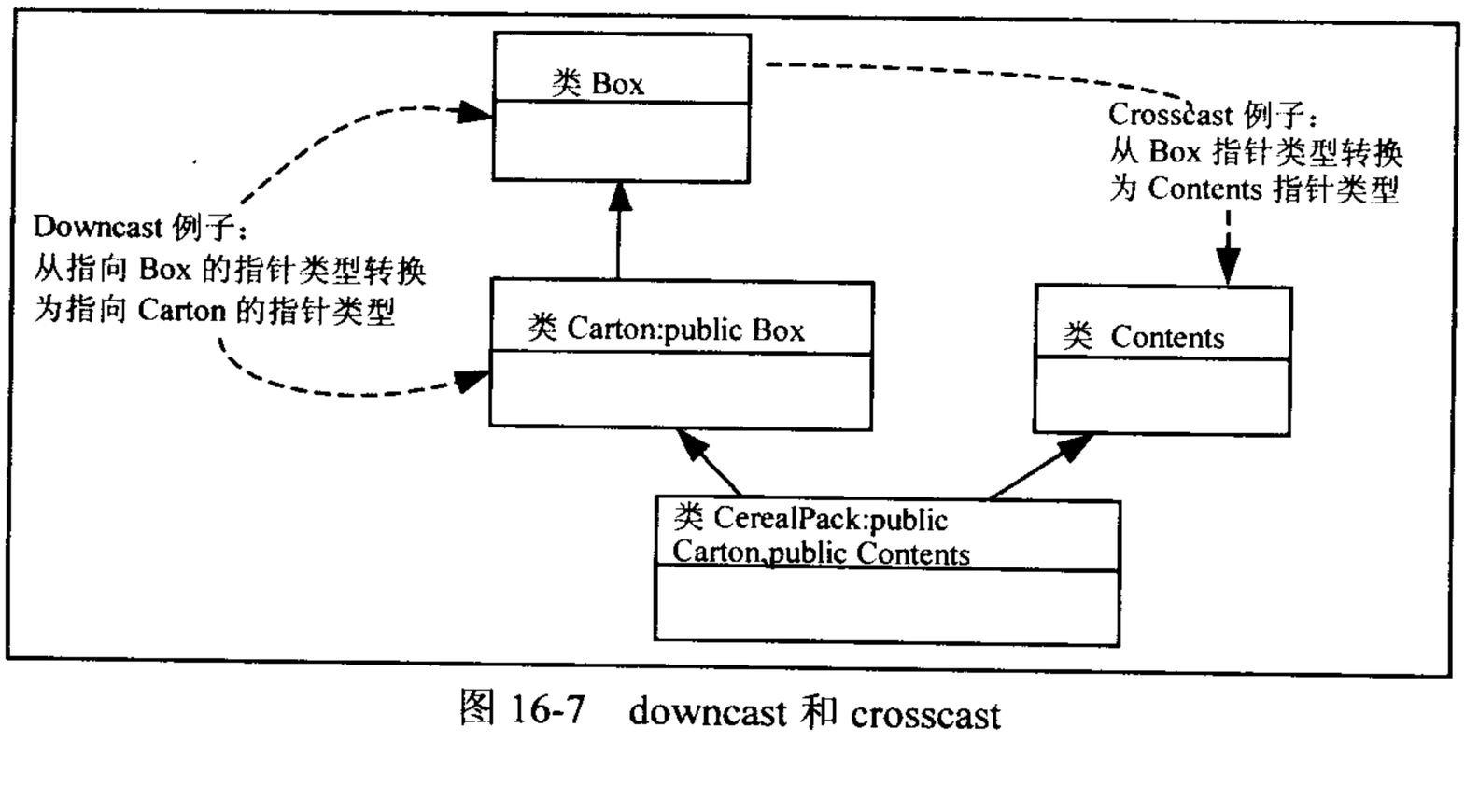


图 16-7　downcast 和 crosscast

对于 pBox 指针，其类型是“指向 Box”的指针，可以把图 16-7 中的 downcast 写为如下语句：

```
Carton* pCarton= dynamic_cast<Carton*>(pBox);
```

可以看出，dynamic_cast<>()运算符的使用方式与 static_cast<>()运算符的相同。目标类型放在 dynamic_cast 后面的尖括号中，要强制转换为新类型的表达式放在圆括号中。要使这个强制转换合法，类 Carton 和 Box 就必须包含虚函数，该虚函数可以是声明的成员或继承的成员。要使这个强制转换能正常工作，pBox 就必须指向 Carton 对象或 CerealPack 对象，因为只有这些类型的对象包含 Carton 子对象。如果强制转换不成功，指针 pCarton 就会设置为 0。

图 16-7 中的 crosscast 可以写为：

```
Contents* pContents=dynamic_cast<Contents*>(pBox);
```

在上面的例子中，Contents 和 Box 类都必须是多态性的，这个强制转换才是合法的。只有 pBox 包含 CerealPack 对象的地址，这个强制转换才会成功。因为只有 CerealPack 类型包含 Contents 对象，而且可以使用 Box*类型的指针表示。另外，如果转换没有成功，pContents 就会设置为 0。

使用 dynamic_cast<>()运算符沿着类层次结构向下进行强制转换可能会失败，但与静态强制转换不同，动态转换的结果是一个空指针，而不是“不定”。这就提供了使用该动态转换运算符的一种方式。假定一个 Box 指针指向某个对象，现在要调用 Carton 类的一个非虚函数成员。基类指针只允许调用派生类的虚函数，而 dynamic_cast<>()运算符可以调用非虚函数。假定 surface()是 Carton 类的一个非虚函数成员。则下面的语句可以调用该 surface()函数：

```
dynamic_cast<Carton*>(pBox)-> surface();
```

这显然是很危险的。仍需要确保 pBox 指向 Carton 对象或指向把 Carton 作为基类的类的对象。否则，dynamic_cast<>()运算符就会返回空，调用就会失败。为了更正这个错误，可以使用 dynamic_cast<>()运算符确定要完成的工作是否有效。例如：

```
if(Carton* pCarton=dynamic_cast<Carton*>(pBox))
  pCarton-> surface();
```

如果强制转换的结果不为空，就可以调用函数了。

注意不能用 dynamic_cast<>()去除指针的 const 性质。如果要强制转换的指针类型是 const，则目标指针类型就必须也是 const。如果要把 const 指针转换为非 const 指针，就必须先使用 const_cast<>()运算符，把 const 指针转换为同类型的非 const 指针。

2. 转换引用

也可以把 dynamic_cast<>()运算符应用于函数中的引用参数，沿着类层次结构向下进行强制转换，生成另一个引用。在下面的例子中，函数 doThat()的参数是基类(Box)对象的一个引用。在函数体中，可以把参数强制转换为派生类型的引用：

```
double doThat(Box& rBox) {
  ...
  Carton& rCarton=dynamic_cast<Carton&>(rBox);
  ...
}
```

这个语句把对 Box 的引用转换为对 Carton 的引用。当然，一般情况下，传送为参数的对象不是 Carton 对象，此时强制转换就不会成功。此时返回的不是空引用，它的失败方式不同于指针强制转换：函数的执行会停止，并产生 bad_cast 类型的异常。我们还没有讲到异常，但第 17 章会详细论述这个主题。

16.2 多态性的成本

天下没有免费的午餐，多态性也是这样。必须以两种方式为多态性买单：它需要更多的内存，虚函数调用也需要额外的系统开销。其原因是虚函数调用的一般实现方式。

假定两个类 A 和 B 包含相同的数据成员，但 A 包含虚函数，而 B 的函数是非虚函数。于是，A 类型的对象需要的内存就比 B 类型的对象多。

提示：

可以用这两个类对象创建一个简单的例子，使用 sizeof()查看它们的内存使用情况。另外，包含虚函数的.exe 程序文件也比包含非虚函数的程序文件大。

内存需求增多的原因是，在创建多态性类的对象(如上述的类 A)时，要在对象中创建一个特殊的指针。这个指针用于调用对象中的虚函数。这个特殊的指针指向一个为类创建的函数指针表，这个表通常称为 vtable，它为类中的每个虚函数建立一个数据项。如图 16-8 所示。

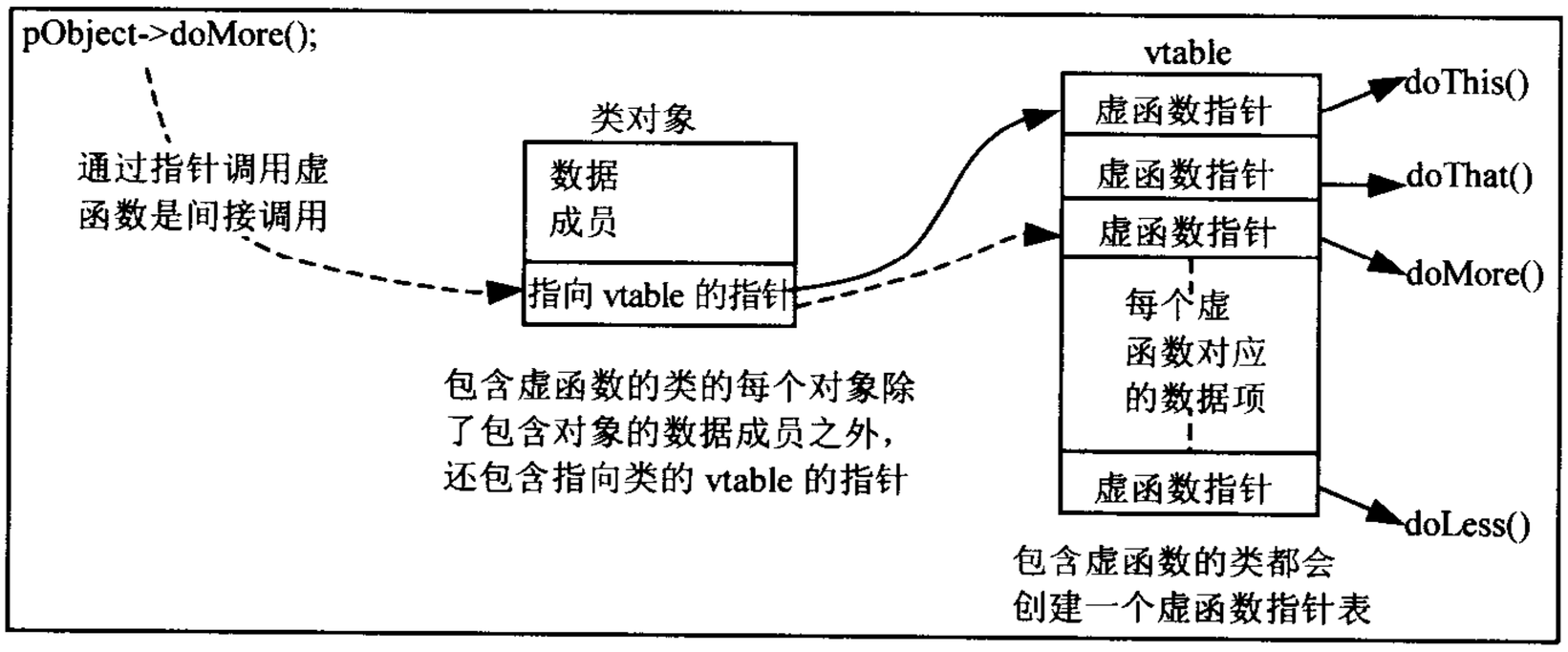


图 16-8 多态性函数调用的工作原理

在通过基类对象的指针调用函数时，就会按顺序发生下列事件：

- 首先，使用指向 vtable 的对象指针查找类中 vtable 的开头。
- 接着在类的 vtable 中，查找被调用的函数所对应的数据项，这通常使用偏移量来实现。
- 最后，通过 vtable 中的函数指针间接地调用函数。这个间接调用要比直接调用非虚函数慢一些，因此虚函数的每次调用都会在执行时间中占用额外的系统开销。

但是，这个系统开销是相当小的，不必担心它。每个对象只需要几个额外的字节，函数调用的速度略慢，这与多态性提供的功能和灵活性相比，代价是比较小的。这里做出这样的解释，是为了说明有虚函数的对象所占用的字节数要比没有虚函数的对象多这种情形。

16.3 纯虚函数

有时，一个基类有大量的派生类，在每个派生类中都要重新定义虚函数，以满足各个派生类的要求，但在基类中定义虚函数却没有什么意义。

例如，定义一个基类 Shape，从该类中派生一些新类 Circle、Ellipse、Rectangle、Curve 等，来定义特定的图形。Shape 类包含一个虚函数 draw()，调用它可以绘制特定的图形，但 Shape 类本身是抽象的：Shape 类的 draw()函数没有实现代码。这种虚函数称为纯虚函数。

纯虚函数的主要作用是允许函数的派生类版本进行多态性的调用。要把函数声明为纯虚函数，而不是一般的虚函数，可以使用与声明虚函数相同的语法，但在类的声明中要加上=0。纯虚函数通常没有实现代码。

如果这些看上去难以理解，下面定义刚才描述的 Shape 类，演示如何声明纯虚函数：

```
// Generic base class for shapes
class Shape {
  public:
    // Pure virtual function to draw a shape
    virtual void draw() const = 0;

    // Pure virtual function to move a shape
    virtual void move(const Point& newPosition) = 0;

  protected:
    Point position;          // Position of a shape

    Shape(const Point& shapePosition) : position(shapePosition) {}
};
```

Shape 类包含一个 Point 类型的数据成员(这是另一个类类型)，它存储了图形的位置。该数据成员放在基类中，是因为每个图形都必须有一个位置，而且构造函数要初始化它。draw()函数是一个虚函数，因为该函数使用了关键字 virtual，draw()还是一个纯虚函数，因为‘=0’(在参数列表的后面)表示该函数不是为这个类定义的。换言之，draw()函数是一个纯虚函数。move()也是一个纯虚函数。

包含纯虚函数的类称为抽象类。在本例中，Shape 类包含两个纯虚函数 draw()和 move()，它肯定是一个抽象类。下面详细论述一下其含义。

16.3.1 抽象类

即使 Shape 类有一个数据成员和一个构造函数，它也没有完整地描述对象，因为 draw()和 move()函数没有定义。因此不允许创建 Shape 类的实例。该类存在的惟一原因是为了定义派生于它的其他类。因为不能创建抽象类的对象，所以不能把它用作函数的参数类型或返回类型。但要注意，抽象类的指针或引用可以用作参数或返回类型。

这就提出了一个问题：如果不能创建抽象类的实例，为什么抽象类还包含一个构造函数？抽象类的构造函数是用于初始化其数据成员的。为此，抽象类的构造函数可以在派生类构造函

数的初始化列表中调用。如果要在其他地方调用抽象类的构造函数，编译器就会产生一个错误消息。

因为抽象类的构造函数一般不能被使用，所以最好把它声明为类的受保护成员，如 Shape 类所示。注意抽象类的构造函数不能调用纯虚函数，调用纯虚函数的结果是不定的。

派生于 Shape 类的任何类如果也不是抽象类，就必须定义 draw()和 move()函数。更明确地说，如果抽象基类的纯虚函数没有在派生类中定义，纯虚函数就会原封不动地继承下来，派生类也就成为一个抽象类。

为了说明这一点，下面定义一个新类 Circle，它把 Shape 类作为基类：

```
// Class defining a circle
class Circle: public Shape {
  public:
    Circle(Point center, double circleRadius) : Shape ( center ),
                                                radius (circleRadius) {}

    virtual void draw() const {
      // Circle center is at point 'position', inherited from the base class
      cout << " Circle center "<< position << " radius " <<radius << endl;
    }

    virtual void move(const Point& newCenter) {position = newCenter; }

  private:
    double  radius;       // Radius of a circle
};
```

这段代码定义了 draw()和 move()函数，所以这个类不是抽象类。如果没有定义这两个类中的任何一个，Circle 类就是抽象类。该类包含一个构造函数，它调用基类的构造函数来初始化基类的子对象。

注意：

在派生类构造函数的初始化列表中，可以调用抽象基类的构造函数。

当然，抽象类还包含非纯虚函数和非虚函数。它还可以包含任意数量的纯虚函数，如本例所示。类中只要有一个纯虚函数，该类就是抽象类。派生类必须定义基类中的每个纯虚函数，否则，该派生类也是一个抽象类。下面介绍一个使用抽象类的例子。

程序示例 16.6—— 抽象类

定义 Box 类的新版本，其中把 volume()函数声明为纯虚函数：

```
class Box {
  public:
    Box(double lengthValue = 1.0, double widthValue = 1.0,
                                                  double heightValue = 1.0);

    // Function to calculate the volume of a Box object
    virtual double volume() const = 0;

  protected:
```

```
    double length;
    double width;
    double height;
};
```

在文件 Box.cpp 中不再需要 volume()的定义，也可以在这个例子中删除 showVolume()函数。Box 现在是一个抽象类，不再需要创建它的对象。Carton 和 ToughPack 类定义了 volume()函数，所以它们不是抽象类，可以使用这两个类的对象证明虚函数 volume()仍像以前那样工作：

```
// Program 16.6 Using an abstract base class   File: prog16_06.cpp
#include <iostream>
#include "Box.h"   // For the Box class
#include "ToughPack.h"// For the ToughPack class
#include "Carton.h" // For the Carton class
using std::cout;
using std::endl;

int main() {
  cout << endl;

  ToughPack hardcase(20.0, 30.0, 40.0);
  Box*  pBox = &hardcase;  //  Store address of  ToughPack object
  cout << "Volume of hardcase is " << pBox->volume() << endl;

  Carton aCarton(20.0, 30.0, 40.0);
  pBox = &aCarton;     // Store address of a Carton object
  cout << "Volume of aCarton is " << pBox->volume()<< endl;

  return 0;
}
```

运行结果如下所示：

```
Volume of hardcase is 20400
Volume of aCarton is 22722.4
```

例子的说明

volume()在 Box 类中的纯虚函数声明，确保了 Carton 和 ToughPack 类中的 volume()函数也是虚函数。因此，可以通过基类的指针调用它们。该调用是动态解析的。首先定义并初始化第一个 ToughPack 对象，再定义指针 pBox：

```
ToughPack hardcase(20.0, 30.0, 40.0);
Box* pBox=&hardcase ;              //Store address of ToughPack object
```

pBox 现在包含 ToughPack 对象 hardcase 的地址。使用基类 Box 的指针调用 volume()函数的 ToughPack 类版本：

```
cout<< "Volume of hardcase is "<<pBox-> volume()<<endl;
```

从输出中可以看出，函数调用动态解析为 volume()函数的 ToughPack 类版本。接着，在 pBox 中存储 Carton 对象的地址：

```
Carton aCarton(20.0, 30.0, 40.0);
pBox=&aCarton;              //Store address of a Carton object
```

包含在 pBox 中的 ToughPack 对象的地址被 Carton 对象的地址改写了。接着使用 pBox 显示 Carton 对象的体积：

```
cout<< "Volume of aCarton is"<<pBox-> volume()<<endl;
```

由于 pBox 包含 Carton 对象的地址，因此该调用动态解析为 Carton 类中的 volume()函数。

用作接口的抽象类

有时，创建抽象类的原因仅是函数在类中定义不合理，只能在派生类中实现。但是，使用抽象类还有另一种方式。

只包含一个纯虚函数的抽象类可以用于定义标准类接口。它一般表示一组支持特定功能的相关函数声明，这组函数通过调制解调器进行通信。前面说过，派生于这种抽象基类的类必须为每个虚函数定义实现代码，但每个虚函数的实现方式由实现派生类的人决定。抽象类固定了接口，但实现代码(通过派生类)是灵活的。

16.3.2 间接的抽象基类

本章前面简要介绍了间接继承。给定一个基类和一个派生类，基类本身也可以派生于另一个更一般的基类。在这种情况下，最特殊的派生类间接继承了最一般的基类。可以创建任意多级派生。下面对前一个例子进行略微的扩展，演示涉及到抽象类的间接继承，同时说明虚函数在第二级继承中的用法。

程序示例 16.7—— 多级继承

下面定义一个 Vessel 类，表示一般的容器，用作 Box 类的抽象基类。这样，表示其他存储容器的类(如 Bottle 或 Can)就派生于 Vessel，并可以对这些类型的对象体积进行多态性的计算。把 Vessel 类的定义放在新的头文件 Vessel.h 中：

```
// Vessel.h - Abstract class defining a vessel
#ifndef VESSEL_H
#define VESSEL_H

class Vessel {
  public:
    virtual double volume() const = 0;
};
#endif
```

这是一个抽象类，因为它包含纯虚函数 volume()。现在修改 Box 类，把 Vessel 类作为基类定义：

```
// Box.h
#ifndef BOX_H
#define BOX_H
```

```
#include "Vessel.h"

class Box : public Vessel {
  public:
    Box(double lengthValue = 1.0, double widthValue = 1.0,
                                                  double heightValue = 1.0);

    // Function to calculate the volume of a Box object
    virtual double volume() const;

  protected:
    double length;
    double width;
    double height;
};
#endif
```

除了包含 Vessel 类的头文件，使 Box 派生于 Vessel 之外，还要把 volume()函数设置为虚拟状态(不是纯虚函数)，这样就可以把函数定义放在 Box.cpp 中了：

```
double Box::volume()   const {
  return length * width * height;
}
```

接着，从 Vessel 中派生另一个类 Can，该类定义了一个铁罐，并把类的定义放在 Can.h 中，把实现代码放在 Can.cpp 中。下面是类定义：

```
// Can.h Class defining a cylindrical can of a given height and diameter
#ifndef CAN_H
#define CAN_H

#include "Vessel.h"

class Can  :  public Vessel {
  public:
    Can(double canDiameter, double canHeight);
    virtual double volume() const;

  protected:
    double diameter;
    double height;
    static const double pi;
};
#endif
```

这个类定义的 Can 对象表示规则的圆柱铁罐，例如啤酒罐。该类把 pi 定义为 const 静态成员，因为在类的函数成员中需要它。即使从来不调用类的构造函数，静态类成员也存在，所以必须在全局命名空间作用域中初始化 pi。为此，可以把其定义和 volume()函数的定义一起放在 Can.cpp 中。所以这个.cpp 文件包含如下内容：

```
// Can.cpp
#include "Can.h"
```

```
Can::Can(double canDiameter,  double canHeight)  :
                                    diameter(canDiameter),height(canHeight) {}

// Function to calculate the volume of a Can object
double Can::volume()  const {
  return pi  * diameter * diameter * height  /  4;
}

// Definitions for the Can class
const  double Can::pi  =  3.14159265;  // Initialize  static member
```

在头文件中，可以把 pi 的定义放在类定义之后，但需要确保该定义位于#ifndef/#endif 指令对之间，因为在程序中，每个静态数据成员只定义一次。通常最好在.cpp 文件中初始化类的静态成员。

下面用一个非常短的 main()函数试验所有这些类：

```
// Program 16.7 Using an indirect base class   File:prog16_07.cpp
#include <iostream>
#tinclude "Box.h"          // For the Box class
#include "ToughPack.h"     // For the  ToughPack class
#include "Carton.h"        // For the Carton class
#include "Can.h"           // for the Can class
using std::cout;
using std::endl;

int main() {
  Box aBox(40, 30, 20);
  Can aCan(10, 3);
  Carton aCarton(40, 30, 20);
  ToughPack hardcase(40, 30, 20);

  Vessel* pVessels[ ] = {&aBox, &aCan, &aCarton,&hardcase };

  cout << endl;
  for(int i = 0 ; i < sizeof pVessels / sizeof(pVessels[0]) ; i++)
    cout << "Volume is " << pVessels[i]->volume() << endl;

  return 0;
}
```

结果如下所示：

```
Volume is 24000
Volume is 235.619
Volume is 22722.4
Volume is 20400
```

例子的说明

在这个例子中，有三级类层次结构，如图 16-9 所示。

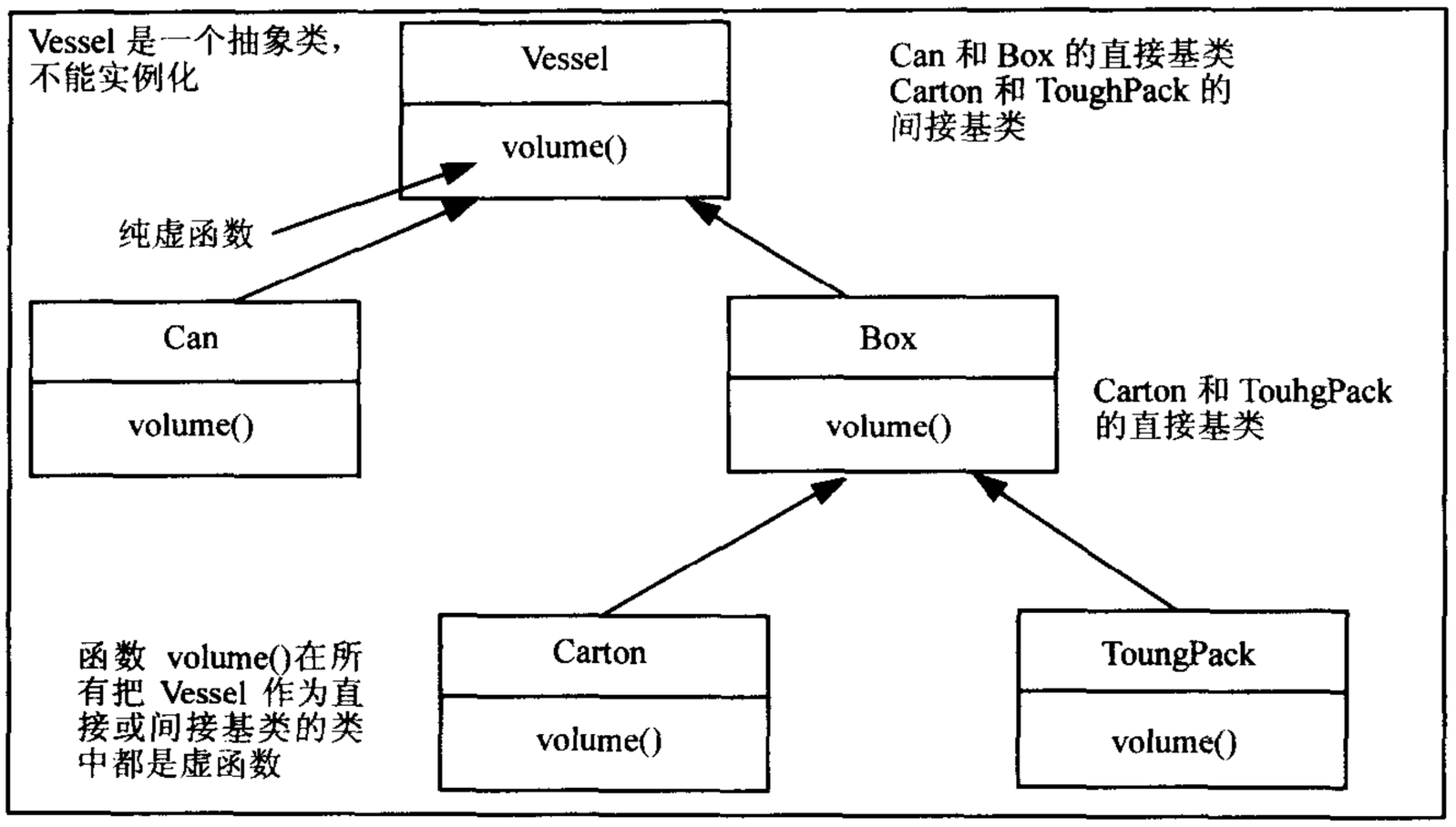


图 16-9　三级类层次结构

在所有以 Vessel 为直接基类或间接基类而派生出来的类中，volume()函数都是虚函数。如果派生类没有定义基类中声明为纯虚拟的函数，该函数就会继承为纯虚函数，这会使该派生类也变成抽象类。把 Can 或 Box 类中的 const 声明删除，就可以看到这个结果。这会使该函数与基类中的纯虚函数不同，派生类继承函数的基类版本，程序不会编译。

在本例中，使用指向 Vessel 的指针的数组调用虚函数：

```
Box aBox(40, 30, 20);
Can aCan(10, 3);
Carton aCarton(40, 30, 20);
ToughPack hardcase(40, 30, 20);

Vessel* pVessels[]={&aBox, &aCan, &aCarton, &hardcase };
```

数组元素用 4 个不同类型的 Vessel 对象的地址进行初始化：这会声明并初始化包含 4 个元素的数组。接着在 for 循环中调用虚函数 Volume()：

```
for(int i=0; i<sizeof pVessels/ sizeof (pVessels[0]); i++)
  cout<<"Volume is "<< pVessels[i]-> volume()<<endl;
```

这个语句调用指针数组 pVessels 中每个元素指向的对象的函数 volume()。从结果中可以看出，函数调用都是动态解析的。

16.4　通过指针释放对象

在处理派生类对象时使用基类指针是非常常见的，因为这是利用虚函数的方式。但在通过基类指针释放对象时有一个问题。如果在前面的例子中用显示一个消息的析构函数实现类，就可以看到这个问题。

程序示例 16.8—— 通过指针释放对象

在 Vessel 类中添加一个析构函数，在调用它时，显示一个消息：

```
class Vessel {
  public:
    virtual double volume() const = 0;
    ~Vessel();
};
```

需要在 Vessel.cpp 文件中创建下面的代码：

```
// Vessel.cpp
#include <iostream>
#include "Vessel.h"

Vessel::~Vessel() {
  std::cout << "Vessel destructor" << std::endl;
}
```

对 Can、Box 和 ToughPack 类作相同的修改，在 Carton 类的析构函数中添加一个输出语句。同样也需要在几个.cpp 文件中包含<iostream>头文件。

现在需要修改前面例子中的 main()，用在自由存储区中创建的对象地址初始化 pVessels 数组。在完成后，还需要使用运算符 delete 释放内存。

```
// Program 16.8 Destroying objects through a pointer   File: prog16_08.cpp
#include <iostream>
#include "Box.h"              // For the Box class
#include "ToughPack.h"        // For the ToughPack class
#include "Carton.h"           // For the Carton class
#include "Can.h"              // For the Can Class
using std::cout;
using std::endl;

int main() {
  Vessel* pVessels[] = { new Box(40, 30, 20),   new Can(10, 3),
                         new Carton(40, 30, 20), new ToughPack(40, 30, 20) };

  cout << endl;
  for(int i = 0 ; i < sizeof pVessels / sizeof(pVessels[0]) ; i++)
    cout << "Volume is " << pVessels[i]->volume() << endl;

  // Delete the objects from the free store
  for(int i = 0 ; i < sizeof pVessels / sizeof(pVessels[0]) ; i++)
    delete pVessels[i];

  return 0;
}
```

运行这个程序，结果如下所示：

```
Volume is 24000
Volume is 235.619
Volume is 22722.4
Volume is 20400
Vessel destructor
```

```
Vessel destructor
Vessel destructor
Vessel destructor
```

例子的说明

显然又遇到另一个故障：在每个对象中调用了错误的析构函数。发生这个故障的原因是对析构函数的绑定是在编译期间设置的，而且对 Vessel 指针指向的对象应用了 delete 运算符，于是每次都会调用 Vessel 类析构函数。为了修复这个故障，需要析构函数的动态绑定。

虚析构函数

为了确保为在自由存储区中创建的对象调用正确的析构函数，需要使用虚类析构函数。要实现虚类析构函数，只需在类的析构函数声明中添加关键字 virtual。这就告诉编译器，通过指针或引用参数调用的析构函数应是动态绑定的，这样析构函数就在运行期间选择。即使析构函数有不同的名称，这个机制也会工作；虚析构函数就是为这个目的而设计的。

程序示例 16.9——调用虚析构函数

修改上一个例子就可以呈现这个效果。只需在基类的析构函数声明中添加关键字 virtual 即可：

```
class Vessel {
  public:
    virtual double volume() const = 0;
    virtual ~Vessel();
};
```

在声明虚基类的析构函数之后，所有派生类的析构函数就都自动成为虚析构函数了。再次运行这个例子，就会得到如下结果：

```
Volume is 24000
Volume is 235.619
Volume is 22722.4
Volume is 20400
Box destructor
Vessel destructor
Can destructor
Vessel destructor
Carton destructor
Box destructor
Vessel destructor
ToughPack destructor
Box destructor
Vessel destructor
```

例子的说明

一个小小的关键字就会造成这么大的区别！输出显示，删除 Box 对象将调用两个析构函数：一个是 Box 析构函数，另一个是 Vessel 析构函数。删除 Can 对象也调用了两个析构函数。删除 Carton 和 ToughPack 对象则分别调用了三个析构函数：对象的类析构函数、直接基类的析构

函数和间接基类的析构函数。

本书前面说过，对象总是以这种方式释放：按照类层次结构的顺序逐个调用类的析构函数，直到调用最一般的基类析构函数。在创建派生类对象时，调用构造函数的顺序与此相反。

注意：

在使用继承，且类至少包含一个虚函数时，应总是把基类析构函数声明为 virtual。在执行类的析构函数时，这需要一点额外的系统开销，但在许多情况下这并不是很明显。使用虚析构函数，可以确保对象正确的释放，并避免发生其他可能的错误。

16.5 在运行期间标识类型

基类指针可以包含任何派生类型的对象地址，也就是说，在任意时刻，都不需要指明指针指向的对象是什么类型。指针可以传送为参数，也可以是类成员，在这两种情况下，地址是在其他地方设置的。引用参数也是这样——“引用 Base”类型的参数可以和派生于 Base 的类型的对象一起使用。

在这些情况下，知道指针或引用的类型是有帮助的—— 操作取决于指针或引用的类型。当类层次结构中有两个或多个分支时，这会特别有用，例如在前面使用的类层次结构中，一个分支包含 Can 类，而 Carton 类在另一个分支中。

typeid()运算符允许在运行期间确定类型。要使用该运算符，必须在源文件中包含标准头文件<typeinfo>。这个头文件包含 type_info 类的定义。typeid()运算符返回 type_info 类型的对象，更严格地说，是 const std:: type_info 类型的对象。type_info 类实现了==运算符，允许比较两个 type_info 对象。

对于特定的类型，使用表达式 typeid(type)就会获得 type_info 对象。例如，表达式 typeid(Box*)返回表示 Box*类型的 type_info 对象。下面的语句检查指针 pVessel 是否指向 Carton 类型的派生类对象：

```
if(typeid(*pVessel) == typeid(Carton))
  cout << "Pointer is type Carton" << endl;
else
  cout << "Pointer is not type Carton" << endl;
```

实际上，这个语句使用 typeid()运算符执行下述操作：例如，为某个类调用非虚函数。这里忽略了顶级的 const 限定符，因为在函数重载中该限定符用于参数类型，所以类型 const Carton 会得到与 Carton 相同的 type_info 对象。

在使用 typeid()确定动态类型时，通常需要解除基类指针的引用，以测试该指针所指向的对象类型，如上面的代码所示。当然，对于引用参数，则使用参数名就可以获取参数类型的 type_info 对象。例如：

```
void Box: doThat(Vessel& rVessel) {
  ...

if(typeid(rVessel) == typeid(Carton))
  cout << "Type Carton was passed" << endl;
else
```

```
    cout << "Type Carton was not passed" << endl;

    ...
}
```

这里，doThat()是一个假设的函数，在 Box 类中定义，它接受派生于 Vessel 的类的对象。代码确定作为参数传送给函数的对象是否为 Carton 类型的对象。当然，它也可以测试派生于 Vessel 的所有类的类型。

注意：

如果要使用 typeid()运算符，就必须在源文件中包含<typeinfo>头文件。

在这个程序中，并不需要使用 typeid()运算符。如果滥用 typeid()运算符，就可能在需要的时候无法使用多态性。typeid()运算符使用得过于频繁，就会使程序非常僵化—— 要在程序中引入类类型的测试，如果在类层次结构中引入了新的类，这种测试就需要改变。

相反，虚函数是非常灵活的—— 可以在层次结构中添加新类，而无需改变在层次结构中调用虚函数的现有代码。因此，实现已有虚函数的新类都会自动融入类层次结构。

16.6 类成员的指针

前面讨论了如何定义任意类型的变量的指针，以及如何定义函数指针。甚至可以使用指针存储给定类对象的数据成员的地址(假定它是可以访问的)。但是，不能使用函数指针存储成员函数的地址，甚至在指针类型表示的函数签名与成员函数相同时也不能。这是因为成员函数把类类型用作其类型的一部分，所以，成员函数的类型总是不同于具有相同参数列表的一般函数的类型。

但是，可以定义类成员的指针，这种指针可以指向数据成员或成员函数。

类成员的指针为程序中的类层次结构提供了另一层。类成员的指针可以指向任意几个兼容成员，也可以为类的对象访问它指向的成员。下面首先介绍数据成员的指针，它们比成员函数指针简单。

16.6.1 数据成员指针

一般的指针包含某个变量的地址，该变量可以是一般的变量，也可以是类对象的数据成员，但在这两种情况下，指针都包含内存中某块空间的地址。

类数据成员的指针的工作原理有所不同。它包含类数据成员的地址，且仅在与类对象组合使用时，才指向内存的某个位置。

为了说明其含义，下面返回程序示例 16.5 中的代码，修改 Box 类，了解数据成员指针是如何工作的。把数据成员的访问指定符改为 public(这仅用于演示，稍后就会把它们改回 protected)。类定义如下所示：

```
class Box {
  public:
    Box(double lengthValue = 1.0, double widthValue = 1.0,
```

```
                                                    double heightValue = 1.0);
    // Function to show the volume of an object
    void showVolume() const;

    // Function to calculate the volume of a Box object
    virtual double volume() const;

public:
    double length;
    double width;
    double height;
};
```

由于这三个数据成员都是 double 类型，因此可以声明一个数据成员的指针，来指向这三个数据成员中的任一个。指针的类型是 double Box::*。声明如下所示：

```
double Box::* pData;
```

提示：

像 double Box::*这样比较古怪的类型名，最好使用 typedef 创建一个同义词。为此，可以使用下面的语句：

```
typedef double Box::*pBoxMember;
```

创建同义词 pBoxMember，以后就可以使用它代替 double Box::*了。

下面的语句把类成员 width 的地址赋予这个指针：

```
pData=&Box::width;
```

当然，pData 不指向某个数据项，因为它不是类对象的数据成员的指针，而是类成员的指针。它只在和 Box 类型的对象一起使用时，才指向内存中的某个位置。

在类包含相同类型的几个数据成员时，就可以使用数据成员的指针。如果该类型的类成员只有一个，也可以直接引用它。要使用类数据成员的指针，需要使用新的运算符。

“成员指针”选择运算符

前面建立了类的数据成员指针。但只能使用它访问特定对象的成员。该指针总是要和对象、对象的引用或对象的指针一起使用。下面声明并定义一个 Box 类型的对象：

```
Box myBox(20.0, 30.0, 40.0);          //Declare a box
```

假定 pData 仍像前面那样定义，就可以使用它引用 myBox 对象的 width 成员。首先，直接把 myBox 对象名和指针 pData 放在一起使用。使用表达式 myBox.*pData 就可以引用 myBox 对象的 width 成员。在下面的语句中使用这个表达式，可以显示 myBox 对象的 width 数据成员值：

```
std::cout << "Data member value is " << myBox.*pData << std::endl;
```

这个语句使用了“成员的直接指针”选择运算符.*。这个运算符组合了成员选择运算符(.)和解除引用运算符(*)，并应用于数据成员的指针。.*运算符总是把类对象(或类对象的引用)和成员指针组合使用，以选择类对象的成员。这里，指针 pData 被解除引用，以访问 Box 类的

width 成员，成员选择运算符把它应用于 myBox 对象，就得到了 myBox.width。

接着，应用 myBox 指针。为此，在下面的声明语句中，需要声明并初始化 Box 指针：

```
Box* pBox=&myBox;
```

下面，利用数据成员指针 pData，通过这个指针访问 myBox 的数据成员。为此，必须使用“间接成员指针”选择运算符->*，如下所示：

```
cout << "Data member value is " <<pBox->* pData << endl;
```

这个语句组合了间接成员选择运算符(->)和解除引用运算符(*)。下面在一个例子中把这些都组合起来。

程序示例 16.9—— 使用数据成员指针

把 Box 类的数据成员声明为 public，就可以在下面的程序中使用数据成员的指针：

```
// Program 16.9 Using a pointer to a data member    File: prog16_09.cpp
#include <iostream>
#include "Box.h"              // For the Box class
#include "ToughPack.h"        // For the ToughPack class
#include "Carton.h"           // For the Carton class
using std::cout;
using std::endl;

typedef double Box::* pBoxMember;          // Define pointer to data member type

int main() {
  Box myBox(20.0, 30.0, 40.0);             // Declare a base box
  ToughPack hardcase(35.0, 45.0, 55.0); // Declare a derived box
  Carton aCarton(48.0, 58.0, 68.0);        // A different kind of derived box

  pBoxMember pData = &Box::length;         // Define pointer to Box data member

  cout << endl;

  // Using a pointer to class data member with class objects
  cout << "length member of myBox is " << myBox.*pData << endl;
  pData = &Box::width;

  cout << "width member of myBox is" << myBox.*pData << endl;
  pData=&Box::height;

  cout << "height member of myBox is " << myBox.*pData << endl;
  cout << "height member of hardcase is " << hardcase.*pData << endl;
  cout << "height member of aCarton is " << aCarton.*pData << endl;

  Box* pBox = &myBox; // Define pointer to Box

  // Using a pointer to class data member with a pointer to the base class
  cout << "height member of myBox is " << pBox->*pData << endl;
  pBox = &hardcase;
```

```
  cout << "height member of hardcase is " << pBox->*pData << endl;

  cout << endl;
  return 0;
}
```

编译并运行这个例子，结果如下所示：

```
length member of myBox is 20
width member of myBox is 30
height member of myBox is 40
height member of hardcase is 55
height member of aCarton is 68
height member of myBox is 40
height member of hardcase is 55
```

例子的说明

在 main()的定义之前，下面的语句定义了类型 pBoxMember：

```
typedef double Box::* pBoxMember;          //Define pointer to data member type
```

这个语句把 pBoxMember 定义为“Box 的 double 成员指针”类型的同义词。

在定义了类 Box、Carton 和 ToughPack 的对象后，下面的语句声明并初始化了 Box 类的 double 成员指针：

```
pBoxMember pData=&Box::length;          //Define pointer to Box data member
```

pData 现在指向 Box 类的 length 成员，使用它可以引用任何 Box 对象的 length 成员。下面的语句显示 myBox 的 length 成员：

```
cout << "length member of myBox is " << myBox.* pData << endl;
```

pData 是一个指针，可以使它指向 Box 类中的其他成员：

```
pData=&Box::width;
```

pData 现在指向 Box 类的 width 成员。pData 只能存储 double 类型的数据成员的地址。如果数据成员的类型不是 double，例如是 string，就必须声明另一个类数据成员的指针，来存储其地址。

用与前面非常类似的语句显示 myBox 的 width 成员的值：

```
cout << " width member of myBox is " << myBox.* pData << endl;
```

为了证明这不是侥幸成功，在 pData 中存储 Box 的 height 成员的地址，并显示 myBox 对象的 height 成员的值，如下面的语句所示：

```
pData=&Box::height;
cout << " height member of myBox is " << myBox.* pData << endl;
```

当然，派生于 Box 的类的对象都包含 Box 子对象，所以也可以对 Carton 和 Toughpack 对象使用数据成员指针。

在声明并初始化 pBox 后，使用成员选择的间接指针运算符显示 myBox 的 height 成员值：

```
Box* pBox=&myBox;
cout << " height member of myBox is " << pBox->*pData << endl;
```

当然，当基类指针包含派生类对象的地址时，也可以使用这个方法：

```
pBox=&hardcase;
cout << " height member of hardcase is " << pBox->*pData << endl;
```

这个语句显示了 pBox 指向的对象的 height 成员值，得到 hardcase. height 的值。

16.6.2 成员函数指针

“类的成员函数指针”的类型涉及到类类型、函数的参数列表和返回类型。这意味着这种指针是专用于类的，不能用于存储其他类的函数成员的地址。除此之外，它们也遵循第 9 章提出的函数指针的原则。成员函数指针的声明有点啰嗦，下面用一个具体的实例来说明。

把 Box 类数据成员重新设置为 protected，添加一些 public 函数，提取数据成员的值：

```
class Box {
  public:
    Box(double lengthValue = 1.0, double widthValue = 1.0,
                                        double heightValue = 1.0);

    // Function to show the volume of an object
    void showVolume() const;

    // Function to calculate the volume of a Box object
    virtual double volume() const;

    // Get values of data members
    double getLength() const { return length; }
    double getWidth() const { return width; }
    double getHeight() const { return height; }

  protected:
    double length;
    double width;
    double height;
};
```

下面声明一个 Box 的成员函数指针，用于存储前面添加的三个函数的地址。该指针声明为：

```
double(Box::*pGet)()const;
```

这个语句声明了指针 pGet。与数据成员指针相同，这个指针指向类的一个函数成员，且只有在与类对象一起使用时，才能转换为特定的函数。括号中的类名限定了指针名，用类标识了指针。一般情况下，要为类 class_type 的函数声明一个指针，应使用下面的语句：

```
return_type(class_type::*pointer_name)(parameter_type_list);
```

这是一个复杂的声明。首先是指针指向的成员函数的返回类型。第一对括号包含指针名称

pointer_name 的规范，第二对括号包含指针指向的成员函数通用的参数列表。这不是很简单，而且出现多次，可能降低代码的可读性，因此，在程序代码中，常常使用 typedef 声明这种类型。要替代上面的声明，可以使用下面的语句为 pGet 指针类型定义一个同义词：

```
typedef double(Box::*pBoxFunction)()const;
```

现在使用上面定义的 pBoxFunction 类型声明一个指针：

```
pBoxFunction pGet=&Box::getLength;
```

除了声明指针 pGet 之外，这个语句还用函数 getLength()的地址初始化了 pGet。当然，pGet 只能指向 Box 类的成员函数，其返回类型和参数列表在 typedef 语句中指定。这也表示，pGet 只能指向 const 成员函数。

声明成员函数指针的同义词的一般形式如下：

```
typedef return_type(class_type::*ptr_typename)(parameter_type_list);
```

这个语句定义了类型 ptr_typename。这个类型的指针可以存储 class_type 类的任何成员函数的地址，该成员函数的返回类型是 return_type，参数类型列在 parameter_type_list 中。

成员函数的指针应与类对象、对象引用或指针组合使用，再加上类数据成员的指针所使用的运算符。下面在另一个例子中使用成员函数的指针。

程序示例 16.10——成员函数指针

Box 类的定义如前所述，其他类与前面的例子相同，通过成员函数的指针来调用 Box 类的成员，如下面的代码所述：

```
// Program 16.10 Using a pointer to a function member   File: prog16_10.cpp
#include <iostream>
#include "Box.h"           // For the Box class
#include "ToughPack.h"     // For the ToughPack class
#include "Carton.h"        // For the Carton class
using std::cout;
using std::endl;

typedef double (Box::*pBoxFunction)() const;  // Pointer to member function type

int main() {
  Box myBox(20.0, 30.0, 40.0);                // Declare a base box
  ToughPack hardcase(35.0, 45.0, 55.0);       // Declare a derived box
  Carton aCarton(48.0, 58.0, 68.0);           // A different kind of derived box

  pBoxFunction pGet = &Box::getLength;        // Pointer to member function

  cout << endl;

  // Call member function for an object through the pointer
  cout << "length member of myBox is " << (myBox.*pGet)() << endl;

  pGet = &Box::getWidth;
  cout << "width member of myBox is " << (myBox.*pGet)() << endl;
```

```
    pGet = &Box::getHeight;
    cout << "height member of myBox is " << (myBox.*pGet)() << endl;

    //It works for derived class objects too
    cout << "height member of hardcase is " << (hardcase.*pGet)() << endl;
    cout << "height member of aCarton is " << (aCarton.*pGet)() << endl;

    Box* pBox = &myBox;     // Pointer to the base class

    // Calling a function with a pointer to a class object
    cout << "height member of myBox is " << (pBox->*pGet)() << endl;

    pBox = &hardcase;
    cout << "height member of hardcase is " << (pBox->*pGet)() << endl;

    cout << endl;
    return 0;
}
```

这个程序的结果如下：

```
length member of myBox is 20
width  member of myBox is 30
height member of myBox is 40
height member of hardcase is 55
height member of aCarton is 68
height member of myBox is 40
height member of hardcase is 55
```

例子的说明

typedef 为 Box 成员函数类型的指针定义了一个同义词：

```
typedef double(Box::*pBoxFunction)()const;  //Pointer to member function type
```

这种类型的指针只能指向 Box 类中没有参数、返回类型为 double 的 const 函数成员。与此不同的函数成员需要另一个指针类型。

使用在 main()中定义的类型，声明指针 pGet：

```
pBoxFunction pGet=&Box::getLength;          //Pointer to member function
```

使用 pGet 可以存储 Box 中提取数据成员值的三个函数中任何一个函数的地址，因为它们的签名和返回类型都相同。这里，该指针初始化为 getLength()函数成员的地址。

为给 myBox 对象调用 getLength()函数，只需使用直接指针成员选择运算符：

```
cout << "length member of myBox is " << (myBox.*pGet)() << endl;
```

表达式 myBox.*pGet 外面的括号是必须的，没有它们，该语句就不会编译。这是因为函数调用运算符()的优先级比运算符.*的优先级高。没有括号，表达式就等价于 myBox.*(pGet())。此时，就要对在全局命名空间中调用 pGet()函数所返回的值应用直接成员选择运算符。在程序的后面，通过类对象指针，使用间接成员选择运算符来调用函数成员时，也会出现这种情况。

```
cout << "height member of myBox is " << (pBox->*pGet)() << endl;
```

函数调用运算符的优先级比间接成员选择运算符高，所以必须加上括号。从例子其余代码的结果来看，通过指针 pGet 为派生类对象调用函数，与使用类数据成员的指针完全相同。

把成员指针传送给函数

成员函数的参数可以是成员的指针。它们可以是数据成员的指针，也可以是函数成员的指针，下面看一个例子。

下面编写一个函数，计算 Box 对象中任何一面的面积，并把该函数作为 Box 类的一个函数成员，该函数带有两个成员函数指针的参数：

```
class Box {
  public:
    double sideArea(double (Box::*pGetSidel)() const,
                                                double (Box::*pGetSide2)() const) {
      return (this->*pGetSidel)() * (this->*pGetSide2)();
    }

  // Rest of the class as before
};
```

上面的代码显示了参数的类型，为了使这些代码更简单，下面先在上面的例子中，使用 typedef 定义类型 pBoxFunction：

```
typedef double(Box::*pBoxFunction)()const;      //Pointer to member function type
```

则类中的函数定义如下：

```
class  Box {
  public:
    double sideArea(pBoxFunction pGetSidel,   pBoxFunction pGetSide2) {
      return (this->*pGetSidel)()  *  (this->*pGetSide2)();
    }

  // Rest of the class as before
};
```

有一个 Box 对象 myBox，下面的语句将计算出一面的面积：

```
std::cout << "Side area is "
          << myBox.sideArea(&Box::getHeight, &Box::getLength)
          << std::endl;
```

这个语句把成员函数 getHeight()和 getLength()的地址作为 sideArea()函数的参数，用于在面积计算中获取边长的值。

也可以为派生类对象调用这个函数，因为该函数是继承的。当然，也可以通过 Box 对象的基类指针来调用它，或通过 Carton 或 ToughPack 类对象的指针来调用它。

16.7 本章小结

本章介绍了使用继承时涉及到的一些规则。本章的要点如下：

- 多态性是通过指针或引用调用函数，而且调用是动态解析的，即在程序执行时确定调用哪个函数。
- 基类中的函数可以声明为 virtual。在派生于该基类的所有类中，这会迫使该函数总是虚函数。在通过指针或引用调用虚函数时，函数调用就是动态解析的。函数调用的对象类型将确定所使用的函数版本。
- 使用对象和直接成员选择运算符来调用虚函数，该函数调用就是静态解析的，即在编译期间解析。
- 如果基类包含虚函数，就应把基类的析构函数声明为 virtual。这会确保为动态创建的派生类对象选择正确的析构函数。
- 纯虚函数没有定义。基类中的虚函数在函数声明的最后加上=0，就变成了纯虚函数。
- 包含一个或多个纯虚函数的类称为抽象类，这种类不能创建对象。在该类的任何派生类中，必须定义所继承的所有纯虚函数。否则，该派生类也是抽象类，也不能创建该类的对象。
- 虚函数的默认参数值是静态赋予的，如果虚函数的基类版本有默认参数值，在派生类中指定的默认参数值就会被忽略，因为虚函数调用是动态解析的。
- 可以声明类成员的指针。它们可以是数据成员的指针，也可以是函数成员的指针。这类指针要与对象、对象的引用或指针一起使用，来引用成员指针定义的对象的类成员。

16.8 练习

1. 在第 15 章的练习中，定义了一个基类 Animal，它包含两个私有数据成员，一个是 string 成员，存储动物的名称("Fido")，一个是整数成员 weight，存储了动物的重量(单位是磅)。该基类还包含一个公共的虚拟成员函数 who()和一个纯虚函数 sound()，公共的虚拟成员函数 who()返回一个 string 对象，该对象包含了 Animal 对象的名称和重量，纯虚函数 sound()在派生类中应返回一个 string 对象，表示该动物发出的声音。把 Animal 类作为一个公共基类，派生至少三个类 Sheep、Dog 和 Cow，在每个类中实现 sound()函数。

定义一个类 Zoo，它至多可以在一个数组中存储 50 种不同类型的动物(使用指针数组)。编写一个 main()函数，创建给定数量的派生类对象的随机序列，在 Zoo 对象中存储这些对象的指针。使用 Zoo 对象的一个成员函数，输出 Zoo 中每个动物的信息，以及每个动物发出的声音。

2. 定义一个类 BaseLength，它把长度存储为一个整数值，单位是毫米，该类有一个成员函数 length()，该函数返回一个指定长度的 double 值。从 BaseLength 中派生一些类 Inches、Meters、Yards 和 Perches，并重写基类的 length()函数，把长度返回为相应单位的 double 值(1 英寸＝25.4 毫米，1 米＝1000 毫米，1 码＝36 英寸，1 杆(US)＝5.5 码)。定义一个 main()函数，读取一系列不同单位的长度，创建相应的派生类对象，把它们的地址存储在 BaseLength*类型的数组中。以毫米和原单位输出每个长度。

3. 定义转换运算符函数，把第 2 题中的每个派生类型转换为其他派生类型。例如，在 Inches 类中，定义成员 operator Meters()、operator Perches()和 operator Yards()。在 main()中添加代码，以 4 种不同的单位输出每个长度值(转换运算符不需要指定返回值，因为返回类型在名称中隐含了)。

4. 使用转换的构造函数代替转换运算符，完成第 3 题。

第 17 章　程序错误和异常处理

异常是对 C++程序中出现错误或未预料到的情况发出信号的方式。使用异常来警示错误不是强制的，有时以其他方式处理它们会更方便。但是，理解异常的工作原理是很重要的，因为它们可以在使用标准语言功能如运算符 new 和 dynamic_cast 时发生，而且在标准库中得到广泛的应用。

本章主要内容

- 异常的概念
- 如何使用异常警示程序中的错误
- 如何处理异常
- 在标准库中定义了哪些异常
- 如何限制函数生成的异常类型
- 如何处理在构造函数中生成的异常
- 生成的异常如何影响类的析构函数

17.1　处理错误

错误处理是成功编程的一个基本元素。程序需要具备处理潜在的错误和异常事件的能力，而对此付出的努力要多于编写在正常情况下运行的代码。错误处理代码的质量将决定程序的健壮程度，通常也是程序友好性的一个主要因素。它还对更正代码中的错误以及增加应用程序功能有非常重大的影响。

但并不是所有的错误都是相同的，错误的本质决定了如何在程序中处理它们。在许多情况下，可以在出现错误的地方直接处理它们。例如，考虑从键盘上读取输入的例子：错误的键入会导致错误的输入，但这并不是一个严重的问题。检测输入错误通常比较容易，最合适的处理方法常常是舍弃该输入，提示用户再次输入数据。在这种情况下，错误处理代码常常与处理整个输入过程的代码集成在一起。

比较严重的错误常常是可以修复的，在处理时也不会损害程序中的其他操作。在函数中发现错误时，常常很方便给调用者返回某种类型的错误代码，告诉它该错误的有关信息，并允许调用者决定如何处理。

异常提供了处理错误的另一种方式——它们不会替代前面所述的错误处理方式。使用异常来警示错误的主要优点是错误处理代码与导致出错的代码被完全隔离开。

17.2 理解异常

异常是一个任意类型的临时对象，用于警示错误。异常可以是基本类型的对象，如 int 或 char*，但它通常是专门为处理错误而定义的类对象。异常对象可以把错误发生的信息传送给处理错误的代码，而且涉及到的信息不只一项，显然，这用类对象处理是最好不过了。

在标识出代码中的错误时，可以通过“抛出”异常来警示错误。术语“抛出”表示发生了错误。异常会传送给捕获并处理它的代码块中。抛出异常的代码必须包含在一个用花括号括起来的特殊块中，称为 try 块。如果不在 try 块中的语句抛出了异常，程序就会中断。稍后讨论这个问题。

try 块的后面是一个或多个 catch 块。每个 catch 块包含处理某种异常的代码，因此，catch 块有时称为处理程序。如果在发生错误时抛出异常，处理该错误的所有代码就都放在 catch 块中，与未发生错误时执行的代码完全隔离开。也就是说，处理“正常”事件的代码与处理“异常”事件的代码完全隔离开。

如图 17-1 所示，try 块是花括号中的正常块，前面用关键字 try 来标识。每次执行 try 块时，都有可能抛出几种不同类型的异常之中的一种，所以，try 块的后面有许多不同的 catch 块，每个 catch 块都可以处理一种不同类型的异常。处理程序所处理的异常类型用圆括号中的一个参数标识，前面是 catch 关键字，后面是 catch 块的花括号，最后是处理异常的代码。

```
try
{
 //Code that may throw exceptions...
 }

catch(parameter specifying exception type 1)
{
   //Code to handle the exception...
}
catch(parameter specifying exception type 2)
{
   //Code to handle the exception...
}
...
catch(parameter specifying exception type n)
{
   //Code to handle the exception...
}
```

可能抛出异常的代码必须放在 try 块中

在抛出异常时，将按顺序执行 catch 块。控制权传递给参数与异常类型匹配的第一个 catch 块，如果需要，还会自动转换异常的类型

图 17-1　try 块及其 catch 块

catch 块仅在抛出匹配类型的异常时才执行。如果 try 块没有抛出异常，就不执行相应的 catch 块。不能利用分支结构进入 try 块，例如使用 goto 语句。执行 try 块的惟一方式是从头开始执行，即从左花括号后面的第一个语句开始执行。

17.2.1 抛出异常

我们常常会抛出异常，看看会发生什么。尽管异常总是应使用类对象来处理(稍后讨论)，但这里先使用基本类型，因为在研究可能出现的情况时，使用基本类型会使代码非常简单。

抛出异常要使用 throw 表达式，该表达式用关键字 throw 来表示。下面是抛出异常的一个例子：

```
try {
  // Code that may throw exceptions...

  if(test > 5)
    throw "test is greater than 5";  // Throws an exception

  // This code executes if the exception is not thrown
}
catch(const char* message) {
  // Code to handle the exception...
  // This code executes if an exception of type 'char*' or 'const char*' is thrown
  cout<< message<< endl;
}
```

如果 test 的值大于 5，throw 语句就抛出一个异常。在本例中，异常是"test is greater than 5"。控制权会立即从 try 块传送给用于处理抛出的该类型异常的第一个处理程序 const char*。这里只有一个处理程序，它正巧捕获 const char*类型的异常，所以执行该 catch 块中的语句，显示异常。

注释：

编译器在匹配异常类型和 catch 参数类型时，会忽略关键字 const。稍后解释其原因。

程序示例 17.1——抛出和捕获异常

下面用一个例子来试验异常。在这个例子中，会抛出 int 和 const char*类型的异常，并编写几个输出语句，以查看控制权的流动：

```
// Program 17.1 Throwing and catching exceptions    File: prog17_01.cpp
#include <iostream>
using std::cout;
using std::endl;

int main() {
  cout << endl ;
  for(int i = 0 ; i < 7 ; i++) {
    try {
      if(i<3)
      throw i ;
    cout << " i not thrown - value is " << i << endl;

    if(i > 5)
      throw "Here is another!";
    cout << " End of the try block." << endl;
  }
  catch(const int i) {                    // Catch exceptions of type int
    cout << " i caught - value is " << i << endl;
  }
  catch(const char* pmessage) {           // Catch exceptions of type char*
    cout << " \"" << pmessage<< "\" caught" << endl;
  }

cout << "End of the for loop (after the catch blocks) -i is"
```

```
        << i << endl;
  }

  return 0;
}
```

这个例子的运行结果如下所示：

```
i caught - value is 0
End of the for loop body (after the catch blocks) - i is 0
i caught - value is 1
End of the for loop body (after the catch blocks) - i is 1
i caught - value is 2
End of the for loop body (after the catch blocks) - i is 2
i not thrown - value is 3
End of the try block.
End of the for loop body (after the catch blocks) - i is 3
i not thrown - value is 4
End of the try block.
End of the for loop body (after the catch blocks) - i is 4
i not thrown - value is 5
End of the try block.
End of the for loop body (aftar the catch blocks) - i is 5
i not thrown - value is 6
"Here is another !" caught
End of the for loop body (after the catch blocks) - i is 6
```

例子的说明

在 for 循环中，有一个 try 块，其中的代码可以抛出 int 类型和 const char*类型的异常：如果循环计数器 i 小于 3，就抛出 int 类型的异常；如果 i 大于 5，就抛出 const char*类型的异常。

```
try {
  if(i<3)
    throw i ;
  cout << "  i not thrown - value is " << i << endl;

  if(i>5)
    throw "Here is another!";
  cout << "  End of the try block." << endl;
}
```

异常的抛出会立即把控制权从 try 块中转出，所以该块末尾的输出语句仅在未抛出异常的情况下才会执行。这可以从输出中看出来。当 i 的值是 3、4 或 5 时，才会从这个语句中获得输出。对于 i 的其他值，都会抛出异常，所以输出行不会执行。

try 块后面的第一个 catch 块如下所示：

```
catch(const  int  i) {                    // Catch exceptions of type int
  cout << "  i  caught  - value  is   " << i << endl ;
}
```

try 块的处理程序必须紧跟在 try 块之后。如果在 try 块和 catch 块之间有其他代码，或 try

块的各个 catch 块之间有其他代码，程序就不会编译。这个 catch 块处理 int 类型的异常，从输出中可以看出，当第一个 throw 语句执行时，这个 catch 块就会执行。在这种情况下，不执行下一个 catch 块。在这个处理程序执行完后，控制权就会直接传送给循环的最后一个语句。

第二个处理程序处理 char*类型的异常：

```
catch (const char*  pmessage) {          // Catch exceptions of type char*
  cout << " \""<< pmessage << "\" caught" << endl;
    }
```

在抛出异常"Here is another!"时，控制权会从 throw 语句直接传送给这个处理程序，跳过前一个 catch 块。如果没有抛出异常，则这两个 catch 块都不会执行。可以把这个 catch 块放在前一个处理程序的前面，程序仍会正常运行。在本例中，处理程序的顺序并不重要，但并不总是这样。本章后面将举出处理程序的顺序比较重要的例子。

标记循环迭代结束的语句如下：

```
cout << "End of the for loop(after the catch blocks) - i is"
     << i<<endl;
```

无论是否执行了处理程序，这行代码都会执行。抛出异常并不会结束程序，除非想结束程序。如果可以在处理程序中修复导致异常的问题，程序就可以继续执行。

1. 异常处理过程

在前面的例子中，抛出异常时事件发生的顺序是相当清晰的。但在后台会发生其他事件，如果考虑一下控制权如何从 try 块传递到 catch 块，就能猜出其中的一些事件。事件的 throw/catch 顺序如图 17-2 所示。

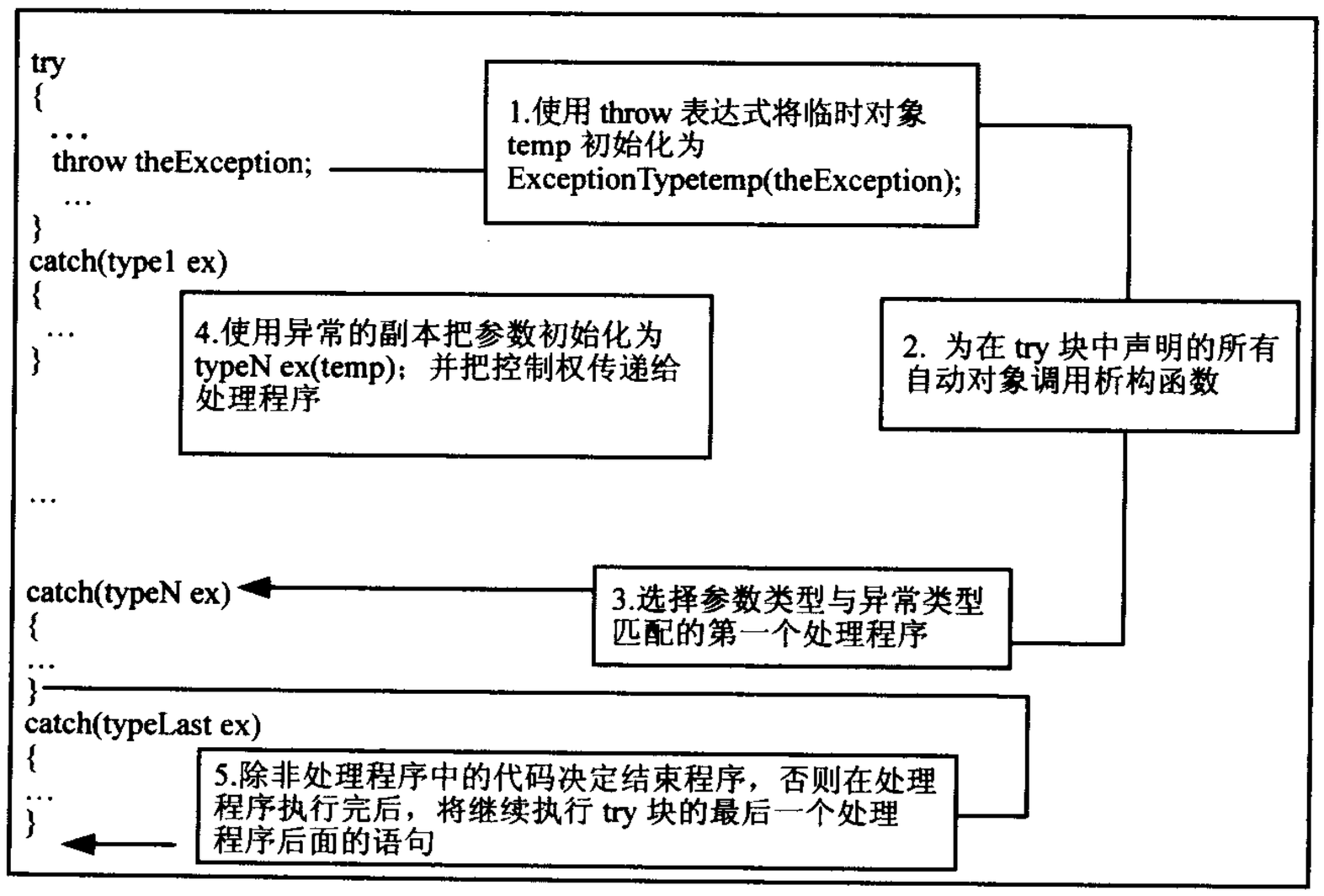


图 17-2　抛出和捕获异常的机制

当然，try 块是一个语句块，语句块总是定义了一个作用域。抛出异常会立即退出 try 块，此时，在 try 块中(到抛出异常的地方为止)声明的所有自动对象都会释放。在处理程序的代码

执行时，这些对象都不存在了。这是非常重要的，所抛出的异常对象不能是指向 try 块的局部对象的指针。这也是复制异常对象的原因。

注意：

异常对象的类型必须是可以复制的。如果对象的类副本构造函数是私有的，该对象就不能用作异常。

由于 throw 表达式用于初始化临时对象，因此创建了异常的一个副本，就可以抛出 try 块的本地异常对象。接着，就使用抛出对象的副本来初始化 catch 块的参数，执行该 catch 块的处理程序。

catch 块也是一个语句块，在执行完 catch 块后，其所有的本地自动对象(包括参数)也都会释放。除非使用 goto 或 return 语句把控制权传送到 catch 块的外部，否则程序会继续执行 try 块的最后一个 catch 块后面的语句。

为异常选择了处理程序后，控制权就会传递给它，所抛出的异常就会被认为在这个处理程序中已处理过。即使 catch 块为空，什么也不做，也是如此。

2. 未处理的异常

如果在 try 块中抛出的异常没有由 catch 块处理(这涉及到稍后介绍的嵌套 try 块)，就会调用标准库函数 terminate()。这个函数在<exception>头文件中声明，它调用预先定义的默认中断处理函数，该函数再调用在<cstdlib>头文件中声明的标准库函数 abort()。未捕获异常的事件顺序如图 17-3 所示。

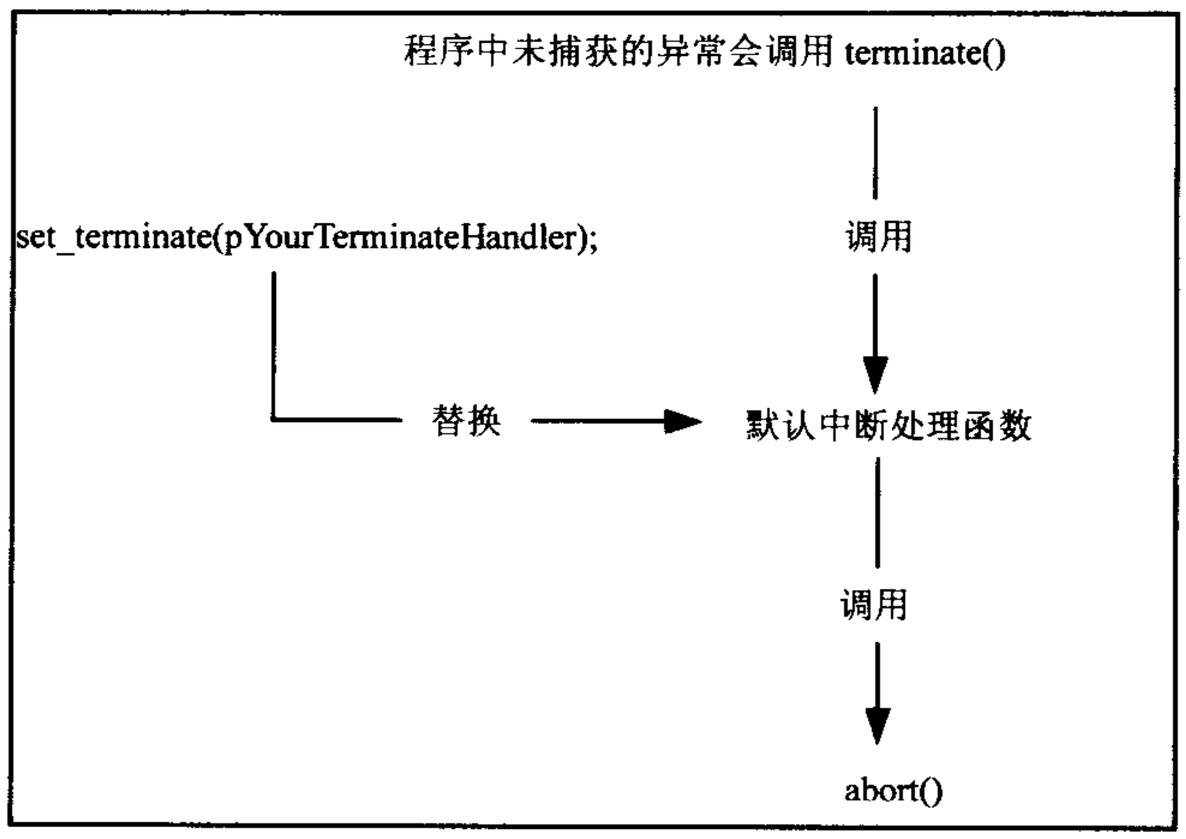


图 17-3　未捕获的异常

注意：

函数 abort()会立即中断整个程序。与 exit()不同，它不为静态建立的对象调用析构函数。

默认中断处理程序所提供的默认操作在某些情况下会造成灾难性的后果，例如，它会使文件处于不满意的状态，或者与电话线的连接依然是打开的。在这种情况下，应确保数据的整齐有序。为此，可以用自己的函数替代默认的中断处理函数，如图 17-3 所示。为了替换默认的处理程序，应调用标准库函数 set_terminate()，它接受 terminate_handler 类型的参数，返回该类型的值。这个类型在 exception 标准头文件中定义：

```
typedef void(*terminate_handler)();
```

terminate_handler 是一个函数指针，该函数没有参数，也没有返回值，所以用于替换它的函数也应没有参数和返回值。在自己的中断处理程序中可以执行需要的操作，但它不能有返回值，而最后必须中断程序。该函数的定义可以采用下面的形式：

```
void myHandler() {
  //Do necessary clean-up to leave things in an orderly state…
  exit(1);
}
```

调用 exit()标准库函数是比调用 abort()更好的中断程序的方式。调用 exit()可确保调用全局对象的析构函数，清空所有打开的输入输出流，并关闭它们；还会删除使用标准库函数创建的所有临时文件。传送给 exit()函数的整数参数会作为一个状态代码返回给操作系统。非 0 值表示程序是非正常中止的。

要把 myHandler ()函数设置为中断处理程序，可以使用下面的语句：

```
terminate_handler pOldhandler = set_terminate(myHandler);
```

返回值是一个指针，它指向所设置的旧处理程序，把它存储起来，就可以在以后需要时恢复它。第一次调用 set_ terminate()时，返回值是指向默认处理程序的指针。以后的每次调用都会建立一个新处理程序，并返回实际起作用的处理程序的指针。也就是说，可以让处理程序在程序的某一部分起作用，再在自己的处理程序不再起作用时，恢复默认的处理程序。

当然，可以在程序的不同地方设置不同的中断处理程序，提供在任意给定的时候都可应用的、适合于特定情况的关闭操作。例如，在程序涉及到数据库操作时，需要确保在发生致命错误时，按顺序关闭数据库，因此应为此专门定义一个中断处理程序。程序的另一个部分则需要使用调制解调器管理通信。这里的中断处理程序应关闭通信链接。不同的中断处理程序适合于不同的关闭请求，只要需要，就可以使用它们。

17.2.2 导致抛出异常的代码

本节开始时提到过，try 块包含可能抛出异常的代码。但是，这并不是说，抛出异常的代码实际上必须放在 try 块的花括号对中。这些代码只需在逻辑上位于 try 块中即可。也就是说，如果在 try 块中调用一个函数，在该函数中抛出的任何异常都会被该 try 块的某个 catch 块捕获。图 17-4 是一个例子。

在 try 块中有两个函数调用，即 fun1()和 fun2()。在这两个函数中抛出的 exceptionType 类型的异常，可以被该 try 块后面的 catch 块捕获。如果在函数中抛出的异常没有被该函数捕获，则该异常就会传递到下一级的调用函数。如果调用函数也没有捕获这个异常，这个异常就会继续向下一级传送，如图 17-4 中 fun3()中抛出的异常。如果异常到达没有 catch 处理程序的一级，且仍未被捕获，就调用中断处理程序，结束程序。

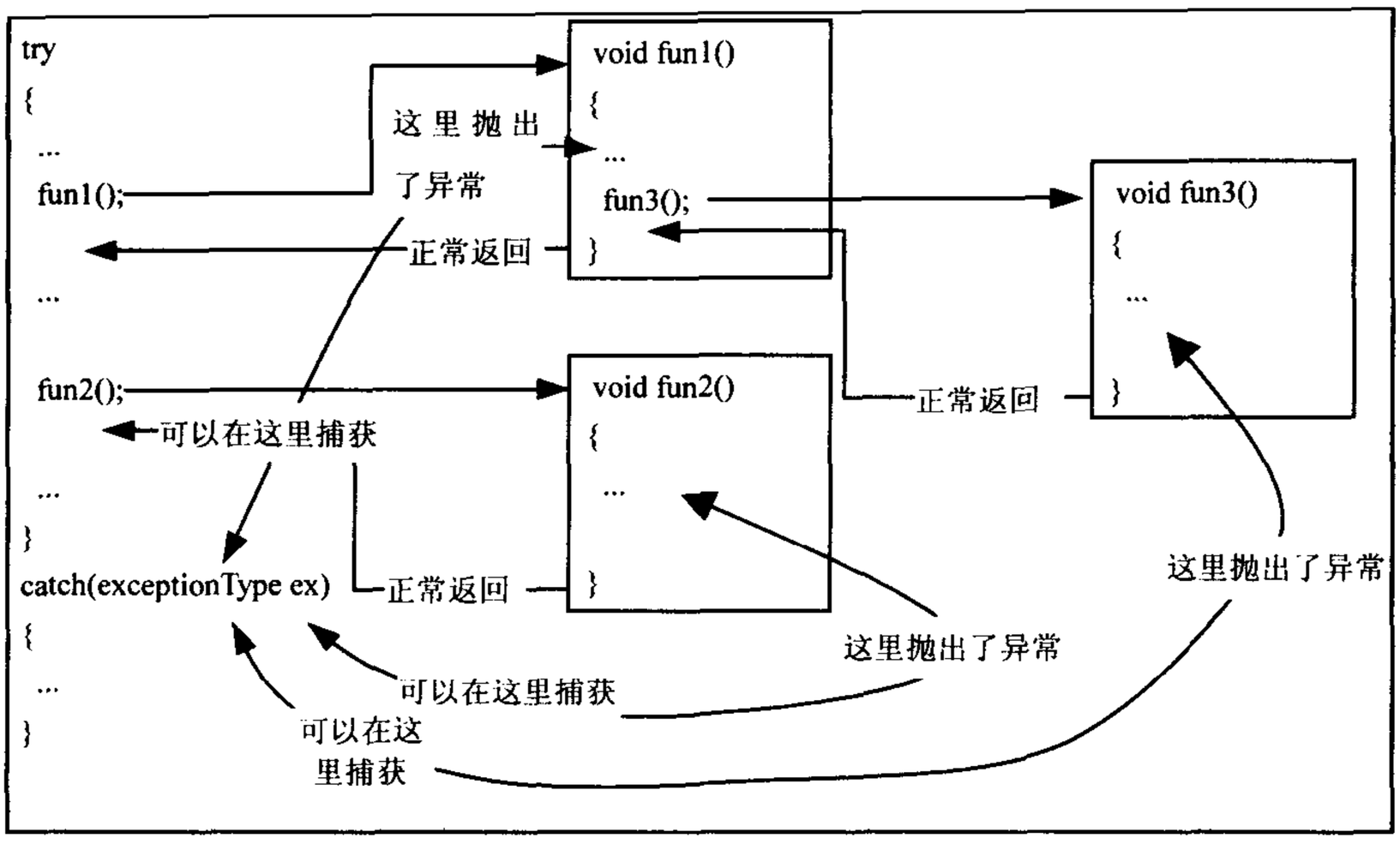


图 17-4　在 try 块中调用的函数抛出的异常

当然，如果在程序的不同位置调用了同一个函数，函数体中代码抛出的异常有可能在不同的时间由不同的 catch 块处理。如图 17-5 所示。

在第一个 try 块中调用函数 fun1()时，fun1()函数抛出的 exceptionType 类型的异常，可以由该 try 块的 catch 块处理。在第二个 try 块中调用函数 fun1()时，该 try 块的 catch 处理程序会处理所抛出的所有 exceptionType 类型的异常。

从图 17-5 中可以看出，应在对程序结构和操作最方便的地方处理异常。在极端情况下，应在 main()中捕获程序的所有异常，即把 main()中的代码都放在 try 块中，其后带数量合适的 catch 块。

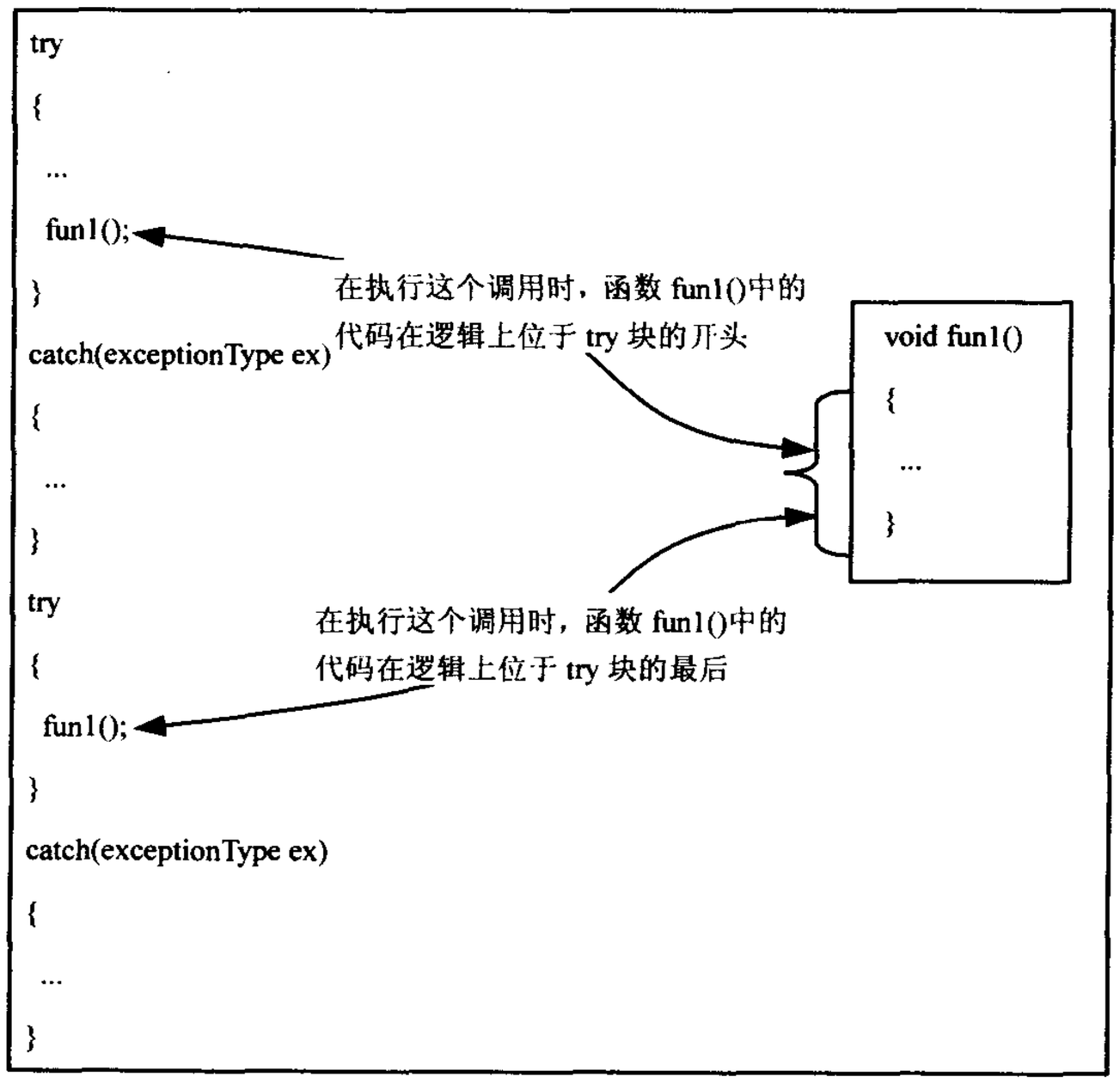


图 17-5　在不同的 try 块中调用相同的函数

17.2.3 嵌套的 try 块

在一个 try 块中可以嵌套另一个 try 块。每个 try 块都有自己的一组 catch 块，来处理在该 try 块中抛出的异常。try 块的 catch 块只能处理在该 try 块中抛出的异常。如图 17-6 所示。

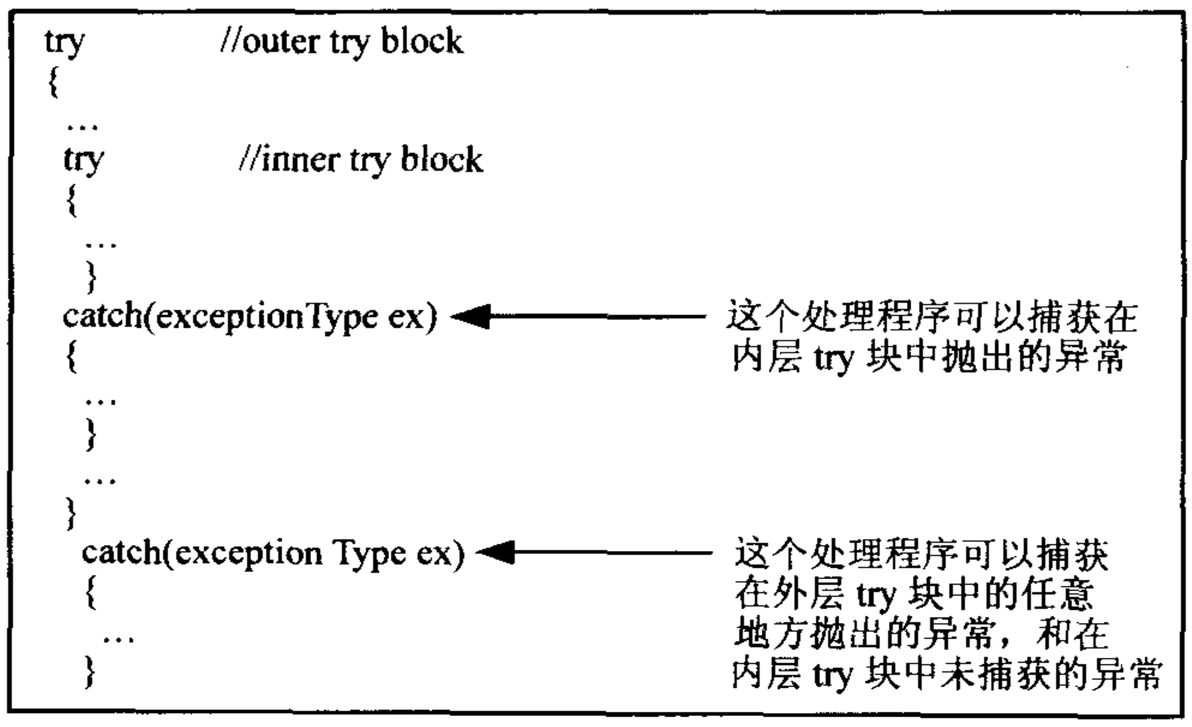


图 17-6 嵌套的 try 块

如图 17-6 所示，每个 try 块都有一个处理程序，但一般情况下，每个 try 块都有好几个处理程序。在内层 try 块中的代码抛出一个异常时，其处理程序会首先处理它。内层 try 块的每个处理程序都会检查匹配的异常类型，如果这些处理程序都不匹配，外层 try 块的处理程序就会捕获该异常。以这种方式嵌套 try 块，嵌套的深度可以是应用程序需要的任意深度。

在外层 try 块的代码抛出异常时，即使抛出异常的语句位于内层 try 块的前面，也由外层 try 块的 catch 处理程序处理该异常。内层 try 块的 catch 处理程序永远都不会处理外层 try 块的代码抛出的异常。

一般情况下，两个 try 块的代码都可以调用函数，在执行函数时，函数体中的代码在逻辑上位于调用它的 try 块中。函数体中的代码也可以位于自己的 try 块中。此时，函数体中的 try 块就嵌套在调用该函数的 try 块中。

程序示例 17.2——嵌套的 try 块

这听起来相当复杂，但实践起来很简单。下面举一个例子，在该例子中要抛出一个异常，看看该异常如何捕获。在目前这个阶段，我们仅是解释异常的捕获，而不是实际应用，所以抛出的异常类型是 int 和 long。本例包含两个嵌套的 try 块，并在一个函数中抛出异常，代码如下所示：

```
// Program 17.2 Throwing exceptions in nested try blocks    File: prog17_02.cpp
#include <iostream>
using std::cout;
using std::endl;

void throwIt(int i) {
  throw i;          // Throws the parameter value
}

int main() {
```

```
for(int i=0;i<=5;i++) {
  try {
    cout << endl << "outer try: ";
    if(i == 0)
      throw i;                          // Throw int exception

    if(i == 1)
      throwIt(i);                       // Call the function that throws int

    try {        // Nested try block
       cout << endl << " inner try: "
    if(i == 2)
      throw static_cast<long>(i); // Throw long exception

    if(i == 3)
      throwIt(i);                       // Call the function that throws int
                                        // End nested try block
  }
  catch(int n) {
    cout << endl << "Catch int for inner try. "
         << "Exception " << n;
  }

cout << endl << "  outer try: ";
    if(i == 4)
      throw i;                  // Throw int
    throwIt(i);     // Call the function that throws int
  }
  catch(int n) {
    cout << endl << "Catch int for outer try. "
                 << "Exception " << n;
  }
  catch (long n) {
    cout << endl << "Catch long for outer try. "
         << "Exception " << n;
   }
  }
  cout << endl;
  return  0 ;
}
```

程序的输出结果如下所示：

```
outer try:
Catch int for outer try. Exception 0
outer try:
Catch int for outer try. Exception 1
outer try:
 inner try:
Catch long for outer try. Exception 2
outer try:
 inner try:
```

```
Catch int for inner try. Exception 3
  outer try:
Catch int for outer try. Exception 3
outer try:
 inner try:
  outer try:
Catch int for outer try. Exception 4
outer try:
 inner try:
  outer try:
Catch int for outer try. Exception 5
```

例子的说明

throwIt()函数显示了它的参数值。如果在 try 块的外部调用这个函数，就会立即结束程序，因为异常未被捕获，于是调用了默认的中断处理程序。

所有的异常都是在 for 循环中抛出的。在该循环中，将在 if 语句中测试循环变量 i 的值，确定何时抛出异常，抛出什么类型的异常。在每次迭代中，至少会抛出一个异常。进入每个 try 块都会记录在输出中，因为每个异常都有一个惟一的值，所以可以看出每个异常是在哪里抛出和捕获的。

第一个异常是在循环变量 i 等于 0 时从外层的 try 块中抛出的：

```
if(i==0)
  throw i;                  //Throw int exception
```

从输出中可以看出，外层 try 块后面捕获 int 类型异常的 catch 块捕获了这个异常。内层 try 块的 catch 块与此没有关系，因为它只能捕获内层 try 块抛出的异常。

下一个异常在外层 try 块中因调用 throwIt()函数而抛出：

```
if(i==1)
  throwIt(i);                          //Call the function that throws int
```

这也是由外层 try 块后面捕获 int 类型异常的 catch 块捕获的。但是，下面两个异常是在内层 try 块中抛出的：

```
if(i==2)
  throw static_cast<long>(i);    //Throw long exception

if(i==3)
  throwIt(i);                          //Call the function that throws int
```

第一个异常是 long 类型。内层 try 块中没有捕获这类异常的 catch 块，所以把它传递给了外层的 try 块。这里，long 类型的 catch 块会处理它，如输出所示。第二个异常是 int 类型，且在 throwIt()函数的函数体中抛出。在这个函数中没有 try 块，所以该异常传递到在内层 try 块中调用该函数的位置。接着，内层 try 块后面处理 int 异常的 catch 块捕获了它。

在内层 try 块的一个处理程序捕获了异常后，程序会继续执行外层 try 块的剩余代码。因此，在 i 等于 3 时，会得到内层 try 块中 catch 块的输出，以及外层 try 块中处理 int 异常的处理程序的输出。后一个异常是在内层 try 块的最后调用 throwIt()函数而抛出的。

最后，在外层 try 块中再抛出两个异常：

```
if(i==4)
  throw i;                          //Throw int
throwIt(i);                         //Call the function that throws int
```

外层 try 块中处理 int 异常的处理程序会捕获这两个异常。第二个异常是在 throwIt()函数体中抛出的，而该函数是在外层 try 块中调用的，所以由外层 try 块的 catch 块来处理异常。

尽管这些异常都不是实际的异常——实际程序中的异常一般是类对象，但它们说明了抛出和捕获异常的机制，以及在嵌套的 try 块中抛出异常的情形。下面详细论述异常对象。

17.3 用类对象作为异常

可以把任意类型的类对象作为抛出的异常。但是，异常对象的核心是与处理程序交流错误信息。因此，通常要定义一个特殊的异常类，来表示某种类型的问题。每个应用程序的异常类都是该应用程序所特有的，但异常类对象总是包含某种类型的消息，也可能包含某种类型的错误代码。还可以让异常对象以合适的方式提供错误源的信息。

下面定义一个简单的异常类，把它放在一个头文件中，该头文件的名称相当普通 MyTroubles.h，以后还可以在这个文件中添加新的内容：

```
// MyTroubles.h Exception class definition
#ifndef MYTROUBLES_H
#define MYTROUBLES_H

class Trouble {
   public:
      Trouble(const char* pStr = "There's a problem") : pMessage(pStr) {}
      const char* what() const {return pMessage;}

   private :
      const char* pMessage;
};

#endif
```

这个类只定义了一个对象，表示一个异常，该异常存储了一个表示有问题的消息。在构造函数的参数列表中定义一个默认的消息，使用该默认构造函数可以创建包含默认消息的对象。what()成员函数返回当前的消息。由于没有在自由存储区中分配内存，在这种情况下，默认的副本构造函数就足够用了。为了使异常处理的逻辑易于管理，需要确保异常类的成员函数不抛出异常。本章的后面将说明如何明确禁止成员函数抛出异常。

下面看看在抛出异常类对象时会发生什么。与前面的例子一样，在大多数情况下不应对出现错误感到烦恼。本例只是抛出异常对象，看看在各种情况下会发生什么。首先要知道如何抛出异常对象。

程序示例 17.3——抛出异常对象

下面用一个简单的例子来执行异常类，在一个循环中抛出一些异常对象：

```
// Program 17.3 Throw an exception object    File: prog17_03.cpp
```

```
#include <iostream>
#include "MyTroubles.h"
using std::cout;
using std::endl;

int main() {
   for(int i = 0; i < 2; i++){
     try {
       if(i == 0)
         throw Trouble();
       else
         throw Trouble("Nobody knows  the trouble  I've  seen...");
     }
     catch(const Trouble& t) {
       cout << endl << "Exception: " << t.what();
     }
  }
  return 0;
}
```

程序的结果如下所示：

```
Exception: There's a problem
Exception: Nobody knows the trouble I've seen…
```

例子的说明

在 for 循环中抛出了两个异常对象。第一个抛出的对象是由 Trouble 类的默认构造函数创建的，它包含默认的消息字符串。第二个异常对象是在 if 的 else 子句中抛出的，它包含给构造函数传送为参数的消息。catch 块捕获了这两个异常对象。

异常对象总是在抛出时复制，如果在 catch 块中没有把参数指定为引用，异常对象会再次复制，这显然是不必要的。抛出异常对象的事件顺序是：先复制异常对象(创建一个临时对象)，再释放原对象，因为退出了 try 块，该对象超出了作用域。之后，把对象副本传送给 catch 处理程序——如果参数是一个引用，就按引用传送。如果要观察所发生的事件，只需给 Trouble 类添加包含一些输出语句的一个副本构造函数和一个析构函数即可。

17.3.1 匹配 Catch 处理程序和异常

跟在 try 块后面的处理程序按照它们在代码中的顺序进行检查，并执行第一个参数类型与异常类型匹配的处理程序。对于基本类型的异常(不是类类型)，需要参数与异常类型完全匹配的 catch 块。而对于类对象的异常，会自动进行类型转换，以匹配异常类型和处理程序的参数类型。

在匹配参数(被捕获的)类型和异常(被抛出的)类型时，出现下列情形就是匹配的：

- 参数类型与异常类型完全匹配，忽略 const
- 参数类型是异常类类型的直接或间接基类，或是异常类的直接或间接基类的引用，忽略 const
- 异常和参数都是指针，异常类型可以自动转换为参数类型，忽略 const

这里列出了可能的类型转换，暗示了 try 块的处理程序的排序方式。如果对同一个类层次结构中的异常类型建立了几个处理程序，则最特殊的派生类类型排在第一，最一般的类类型排在最后。如果基类类型的处理程序排在派生类类型的处理程序之前，则总是选择基类类型的处理程序，来处理派生类的异常。换言之，派生类型的处理程序永远不会执行。

下面在包含 Trouble 类的头文件中，以 Trouble 类为基类再添加两个异常类。在定义了新类后，MyTroubles.h 头文件的内容如下：

```
// MyTroubles.h Exception class definition
#ifndef MYTROUBLES_H
#define MYTROUBLES_H

// Base exception class
class Trouble {
  public :
    Trouble(const char* pStr="There's a problem");
    virtual ~Trouble();
    virtual const char* what()const;

  private:
    const char* pMessage;
};

// Derived exception class
class MoreTrouble : public Trouble {
  public:
    MoreTrouble(const char* pStr = "There's more trouble");
};

// Derived exception class
Class BigTrouble : public MoreTrouble {
  public:
    BigTrouble (const char* pStr = "Really big trouble");
};

#endif
```

注意 what()成员和基类的析构函数都声明为 virtual。因此，在派生于 Trouble 的类中，what()函数也是虚函数。这不会有什么区别，但最好在基类中声明虚析构函数。

派生类除了为消息定义不同的默认字符串外，并没有给基类添加新内容。我们常常用不同的类名来区分不同类型的错误。在一种错误发生时，就抛出对应于该问题的异常类型。类的内部并没有不同。使用不同的 catch 块捕获不同的类类型，可以区分不同类型的错误。

在 MyTroubles.cpp 中，放置三个类的成员函数定义：

```
// MyTroubles.cpp
#include "MyTroubles.h"

// Constructor for Trouble
Trouble::Trouble(const char* pStr) : pMessage(pStr) {}
```

```
// Destructor for Trouble
Trouble::~Trouble() {}
// Returns the message
const char* Trouble::what() const {
  return pMessage;
}

// Constructor for MoreTrouble
MoreTrouble::MoreTrouble(const char* pStr) : Trouble(pStr) {}

// Constructor for BigTrouble
BigTrouble::BigTrouble (const char* pStr) : MoreTrouble(pStr) {}
```

现在用例子来试验一下。

程序示例 17.4——抛出层次结构中的异常类型

下面的代码抛出了 Trouble、MoreTrouble 和 BigTrouble 类型的异常，并包含捕获它们的处理程序：

```
// Program 17.4 Throwing and Catching Objects in a Hierarchy  File: prog17_04.cpp
#include <iostream>
#include "MyTroubles.h"
using std::cout;
using std::endl;

int main() {
  Trouble trouble;
  MoreTrouble moreTrouble ;
  BigTrouble bigTrouble;

  cout << endl;
  for(int i=0; i < 7 ; i++) {
    try {
      if(i<3)
        throw trouble;
      if(i < 5)
        throw moreTrouble;
      else
        throw bigTrouble;
    }
catch(BigTrouble& rT) {
  cout << "  BigTrouble object caught: " << rT.what() << endl;
}
catch(MoreTrouble& rT) {
  cout <<"MoreTrouble object caught: " << rT.what() << endl;
}
catch(Trouble& rT) {
  cout << "Trouble object caught: " << rT.what()<< endl;
}

cout << "End of the for loop (after the catch blocks) - i is " << i << endl;
```

```
  }
  cout << endl;
  return 0 ;
}
```

这个例子的结果如下所示：

```
Trouble object caught: There's a problem
End of the for loop (after the catch blocks) - i is 0
Trouble object caught: There's a problem
End of the for loop (after the catch blocks) - i is 1
Trouble object caught: There's a problem
End of the for loop (after the catch blocks) - i is 2
 MoreTrouble object caught: There's more trouble
End of the for loop (after the catch blocks) - i is 3
 MoreTrouble object caught: There's more trouble
End of the for loop (after the catch blocks) - i is 4
 BigTrouble object caught: Really big trouble
End of the for loop (after the catch blocks) - i is 5
 BigTrouble object caught: Really big trouble
End of the for loop (after the catch blocks) - i is 6
```

例子的说明

在创建好每个类类型的对象后，在一个 for 循环中包含下面的 try 块：

```
try {
  if(i < 3)
    throw trouble;
  if (i < 5)
    throw moreTrouble;
  else
    throw bigTrouble;
}
```

如果循环变量 i 的值小于 3，就抛出 Trouble 类型的异常。如果 i 等于 3 或 4，就抛出 MoreTrouble 类型的异常。如果 i 等于 5 或大于 5，就抛出 BigTrouble 类型的异常。

每个类类型都有一个处理程序，首先是 BigTrouble 类型异常的处理程序：

```
catch(BigTrouble&  rT) {
  cout << " BigTrouble object caught: " <<  rT.what() << endl;
}
```

其他的处理程序与此相同，但它们输出的消息略有不同。在两个派生类型的处理程序中，仍由继承而来的 what()函数返回消息。注意每个 catch 块的参数类型都是一个引用，与上一个例子相同。使用引用的一个原因是避免复制异常对象。在下一个例子中，会说明在处理程序中总是使用引用参数的另一个原因。

每个处理程序都显示包含在被抛出对象中的消息。从输出中可以看出，每个处理程序都会针对所抛出异常的类型进行调用。处理程序的顺序非常重要，因为这与异常和处理程序的匹配相关，与异常类的类型也相关。下面深入探讨这个问题。

17.3.2 用基类处理程序捕获派生类异常

因为派生类类型的异常会自动转换为基类类型，以匹配处理程序的参数，所以前一个例子用一个处理程序就捕获了所有的异常。下面修改前一个例子。

程序示例 17.5——使用基类处理程序

从前一个例子中删除(或注释掉)两个派生类处理程序：

```
// Program 17.5 Catching exceptions with a base class handler   File: prog17_05.cpp
#include <iostream>
#include "MyTroubles.h"
using std::cout;
using std::endl;

int main() {
  Trouble trouble;
  MoreTrouble moreTrouble;
  BigTrouble bigTrouble;

  cout << endl;
  for(int i = 0 ; i < 7 ; i++) {
    try {
      if(i < 3)
        throw trouble;
      if(i < 5)
        throw moreTrouble;
      else
        throw bigTrouble;
    }
    catch(Trouble& rT) {    // Base class handler only
      cout << "Trouble object caught: " << rT.what() << endl;
    }
     cout << "End of the for loop (after the catch blocks) - i is " << i << endl;
    }

  cout << endl;
  return 0;
}
```

这个程序现在的结果如下：

```
Trouble object caught: There's a problem
End of the for loop (after the catch blocks) - i is 0
Trouble object caught: There's a problem
End of the for loop (after the catch blocks) - i is 1
Trouble object caught: There's a problem
End of the for loop (after the catch blocks) - i is 2
Trouble object caught: There's more trouble
End of the for loop (after the catch blocks) - i is 3
Trouble object caught: There's more trouble
End of the for loop (after the catch blocks) - i is 4
```

```
Trouble object caught: Really big trouble
End of the for loop (after the catch blocks) - i is 5
Trouble object caught: Really big trouble
End of the for loop (after the catch blocks) - i is 6
```

例子的说明

Trouble&处理程序现在捕获了所有的异常。如果 catch 块中的参数是基类的一个引用，该块就会匹配所有派生类的异常。尽管每个异常的输出都是 Trouble object caught，但最后 4 个 catch 块捕获的是对应于派生于 Trouble 的异常对象。

因为在按引用传送异常时，其动态类型保持不变，所以也可以使用 typeid()运算符获取和显示动态类型。为此，处理程序的代码应修改为：

```
catch(Trouble& rT) {
  cout << typeid(rT).name() << " object caught: " << rT.what() << endl;
}
```

一些编译器在默认情况下不支持运行期间类型的标识，所以如果这个代码不能正常工作，就应打开相应的编译器选项。对代码进行这样的修改后，输出就会显示，即使使用了基类的引用，派生类异常仍保留了其动态类型。typeid()运算符返回 type_info 类的对象，类的 name()成员把类名返回为 const char*。

对于记录，这个版本的程序输出应如下所示：

```
class Trouble object caught: There's a problem
End of the for loop (after the catch blocks) - i is 0
class Trouble object  caught: There's a problem
End of the for loop (after the catch blocks) - i is 1
class Trouble object caught: There's a problem
End of the for loop (after the catch blocks) - i is 2
class MoreTrouble object caught: There's more trouble
End of the for loop (after the catch blocks) - i is 3
class MoreTrouble object caught: There's more trouble
End of the for loop (after the catch blocks) - i is 4
class BigTrouble object caught: Really big trouble
End of the for loop (after the catch blocks) - i is 5
class BigTrouble object caught: Really big trouble
End of the for loop (after the catch blocks) - i is 6
```

现在可以把处理程序的参数类型改为 Trouble，这样异常就是按值传送，而不是按引用传送：

```
catch(Trouble t) {
  cout << typeid(t).name() << " object caught: " << t.what() << endl;
}
```

运行程序的这个版本，结果如下：

```
class Trouble object caught: There's a problem
End of the for loop (after the catch blocks) - i is 0
class Trouble object caught: There's a problem
End of the for loop (after the catch blocks) - i is 1
class Trouble object caught: There's a problem
End of the for loop (after the catch blocks) - i is 2
```

```
class Trouble object caught: There's more trouble
End of the for loop (after the catch blocks) - i is 3
class Trouble object caught: There's more trouble
End of the for loop (after the catch blocks) - i is 4
class Trouble object caught: Really big trouble
End of the for loop (after the catch blocks) - i is 5
class Trouble object caught: Really big trouble
End of the for loop (after the catch blocks) - i is 6
```

对于派生类的异常对象，仍会选择 Trouble 处理程序，但动态类型不再保持不变了。这是因为参数用基类的副本构造函数进行初始化，与派生类相关的所有属性都丢失了。

在这种情况下，只保留了原派生类对象的基类子对象。所有的派生类成员都从对象中删除了。这是对象切片的一个例子，其原因是基类的副本构造函数对派生对象一无所知。对象切片是按值传送对象的一个常见错误原因，它可以发生在一般函数和异常处理程序中。这就引出了另一个黄金规则：

在 catch 块中应总是使用引用参数。

17.3.3 重新抛出异常

在处理程序捕获一个异常时，可以重新抛出该异常，让外层 try 块的处理程序捕获它。下面的语句就可以重新抛出当前的异常，该语句只包含 throw 关键字，不包含 throw 表达式：

```
throw;          //Rethrow the exception
```

这个语句会重新抛出已有的异常对象，而不是复制它。如果处理程序发现异常需要传送到另一级 try 块中，就可以重新抛出异常。还可以记录下抛出异常的程序位置，在程序的某个中心位置重新抛出它，例如 main()。

注意从内层 try 块中重新抛出异常，并不能使该异常由内层 try 块的其他处理程序捕获。在执行一个处理程序时，所抛出的所有异常(包括当前的异常)都需要由包含当前处理程序的 try 块捕获，如图 17-7 所示。

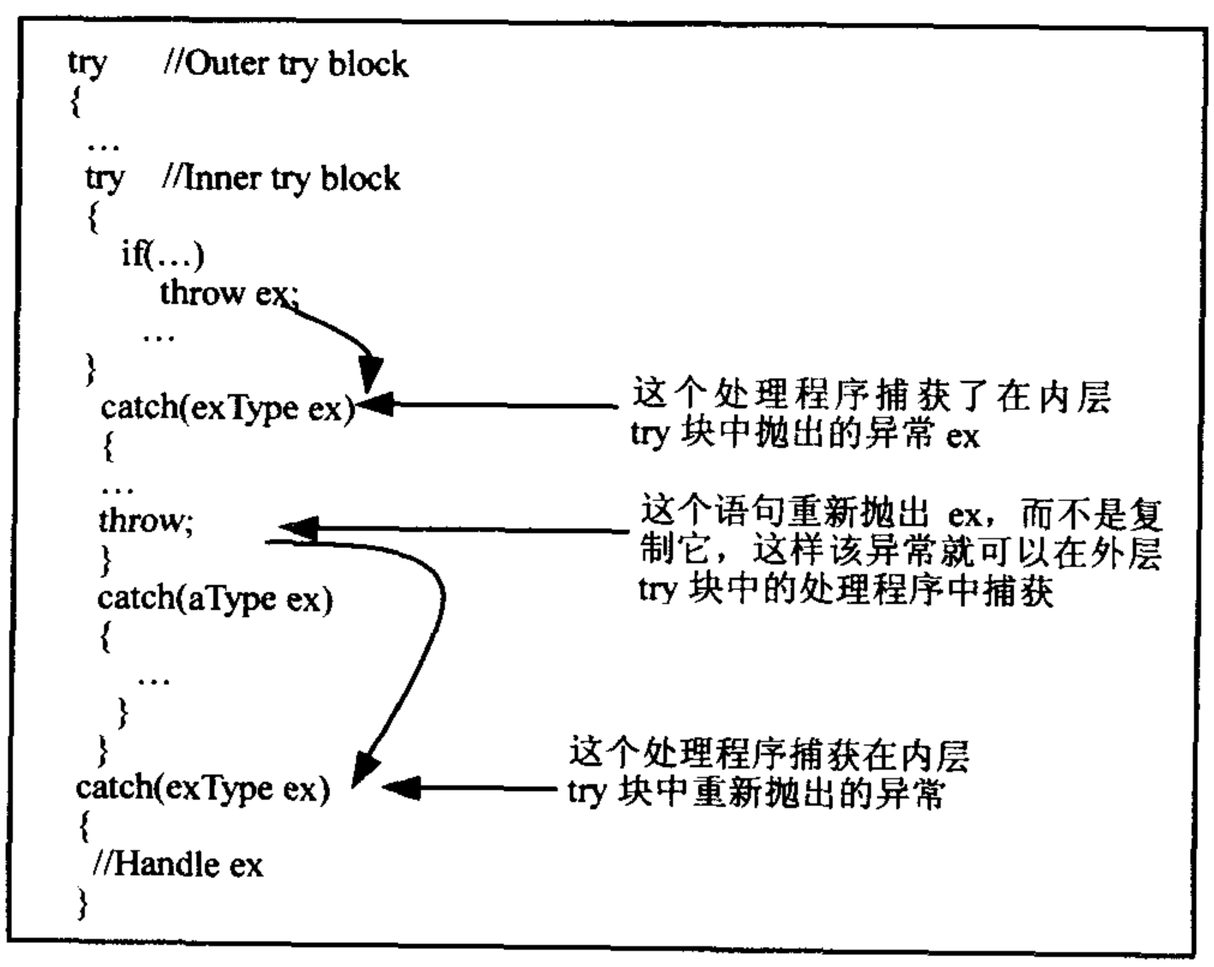


图 17-7 重新抛出异常

重新抛出的异常不是复制的，这一点非常重要，特别是在异常是一个派生类对象，它初始化了基类引用参数时就更重要。下面用一个例子来说明。

程序示例 17.6——重新抛出异常

我们可以抛出一些 Trouble、MoreTrouble 和 BigTrouble 异常对象，再重新抛出它们，看看该机制是如何工作的。这是上一个例子的修订版本：

```
// Program 17.6 Rethrowing exceptions          File: prog17_06.cpp
#include <iostream>
#include "MyTroubles.h"
using std::cout;
using std::endl;

int main() {
  Trouble trouble;
  MoreTrouble moreTrouble;
  BigTrouble bigTrouble;

  cout << endl;
  for(int i = 0 ; i < 7 ; i++) {
    try {
      try {
        if(i < 3)
          throw trouble;
        if(i < 5)
          throw moreTrouble;
        else
          throw bigTrouble;
}
      catch(Trouble& rT) {                    // Inner handler
        if(typeid(rT) == typeid(Trouble))
          cout <<"Trouble object caught: "<< rT.what() << endl;
        else
          throw;       // Rethrow current exception
      }
  }
  catch(Trouble& rT)  {           //Outer handler
     cout << typeid(rT) .name () << " object caught: " << rT.what() << endl;
  }

  cout << "End of the for loop (after the catch blocks) - i is " << i << endl;
  }
  cout << endl;
  return 0;
}
```

这个例子的结果如下所示：

```
Trouble object caught: There's a problem
End of the for loop (after the catch blocks) - i is 0
Trouble object caught: There's a problem
End of the for loop (after the catch blocks) - i is 1
```

```
Trouble object caught: There's a problem
End of the for loop (after the catch blocks) - i is 2
class MoreTrouble object caught: There's more trouble
End of the for loop (after the catch blocks) - i is 3
class MoreTrouble object caught: There's more trouble
End of the for loop (after the catch blocks) - i is 4
class BigTrouble object caught: Really big trouble
End of the for loop (after the catch blocks) - i is 5
class BigTrouble object caught: Really big trouble
End of the for loop (after the catch blocks) - i is 6
```

例子的说明

for 循环与前一个例子相同，但这次有一个 try 块的嵌套。异常对象在内层 try 块中抛出：

```
try {
  if(i < 3)
    throw trouble;
  if(i < 5)
    throw moreTrouble;
  else
    throw bigTrouble;
}
```

这段代码按照与前一个例子相同的顺序抛出异常对象。处理程序会捕获所有的异常对象，因为参数是基类的一个引用：

```
catch(Trouble&  rT) {          // Inner handler
  if(typeid(rT) == typeid(Trouble))
     cout << "Trouble object  caught:  " << rT.what() << endl;
  else
     throw;                                    // Rethrow  current  exception
}
```

这个处理程序捕获了 Trouble 对象，以及派生于 Trouble 的所有类对象。if 语句测试所传送对象的类类型，如果其类型是 Trouble，就执行输出语句。对于其他类型的异常，会重新抛出异常。在这个 catch 块中可以区分输出结果，因为它没有以单词 class 开头。

重新抛出的异常会被外层 try 块的处理程序捕获：

```
catch(Trouble&  rT) {          //Outer handler
  cout << typeid(rT).name() << " object  caught: " << rT.what() << endl;
}
```

其参数也是 Trouble 的引用，所以可以捕获所有的派生类对象。从输出中可以看出，它捕获了重新抛出的对象，结果仍与第一次抛出的情形相同。

内层 try 块的处理程序中的 throw 语句就等价于下面的语句：

```
throw rT;          //Rethrow current exception
```

这里仅是重新抛出了异常吗？不是，实际上，这有一个很大的区别。对程序代码作如下修改，并再次运行，结果如下：

```
Trouble object caught: There's  a problem
End of the for loop (after the catch blocks) - i  is  0
Trouble object caught: There's a problem
End of the for loop (after the catch blocks) - i is 1
Trouble object caught: There's a problem
End of the for loop (after the catch blocks) - i is 2
class Trouble object caught: There's more trouble
End of the for loop (after the catch blocks) - i is 3
class Trouble object caught: There's more trouble
End of the for loop (after the catch blocks) - i is 4
class Trouble object caught: Really big trouble
End of the for loop (after the catch blocks) - i is 5
class Trouble object caught: Really big trouble
End of the for loop (after the catch blocks) - i is 6
```

以这种方式抛出异常会利用 Trouble 类的副本构造函数复制异常，于是又出现了对象切片问题。每个对象的派生部分都被切掉了，只剩下派生类中的基类子对象。从输出中可以看出，typeid()运算符把所有的异常都标识为 Trouble 类型。

17.3.4 捕获所有的异常

对于 catch 块，可以把省略号(三个句点)用作参数说明，表示该块应处理所有的异常：

```
catch(...) {
  //Code to handle any exception...
}
```

这个 catch 块可以处理任何类型的异常，像这样的处理程序必须总是放在 try 块中处理程序的最后。当然，我们对异常的类型一无所知，但至少可以防止程序因为未捕获的异常而中断。注意即使对异常一无所知，也可以重新抛出异常，如前一个例子所示。

程序示例 17.7——捕获所有的异常

修改上一个例子，用省略号代替参数，捕获内层 try 块的所有异常：

```
// Program 17.7 Catching any exception    File: prog17_07.cpp
#include <iostream>
#include "MyTroubles.h"
using std::cout;
using std::endl;

int main() {
  Trouble trouble;
  MoreTrouble moreTrouble;
  BigTrouble bigTrouble;

  cout << endl;
  for(int i = 0 ; i < 7 ; i++) {
    try {
      try {
        if(i < 3)
```

```
        throw trouble;
      if(i < 5)
        throw moreTrouble;
      else
        throw bigTrouble;
    }
    catch(...) {              // Inner handler
      cout << "We caught something! Let's rethrow it." << endl;
      throw;        // Rethrow current exception
    }
  }
    }
    catch(Trouble& rT) {      // Outer handler
      cout << typeid(rT).name() << " object caught: " << rT.what() << endl;
    }

    cout << "End of the for loop (after the catch blocks) - i is " << i << endl;
  }

  cout << endl;
  return 0;
}
```

这个程序的结果如下所示：

```
We caught something! Let's rethrow it.
class Trouble object caught: There's a problem
End of the for loop (after the catch blocks) - i is 0
We caught something! Let's rethrow it.
class Trouble object caught: There's a problem
End of the for loop (after the catch blocks) - i is 1
We caught something! Let's rethrow it.
class Trouble object caught: There's a problem
End of the for loop (after the catch blocks) - i is 2
We caught something! Let's rethrow it.
class MoreTrouble object caught: There's more trouble
End of the for loop (after the catch blocks) - i is 3
We caught something! Let's rethrow it.
class MoreTrouble object caught: There's more trouble
End of the for loop (after the catch blocks) - i is 4
We caught something! Let's rethrow it.
class BigTrouble object caught: Really big trouble
End of the for loop (after the catch blocks) - i is 5
We caught something! Let's rethrow it.
class BigTrouble object caught: Really big trouble
End of the for loop (after the catch blocks) - i is 6
```

例子的说明

与上一个例子的不同之处已经用阴影突出显示了。内层 try 块的 catch 块改为：

```
catch(...) {              // Inner handler
  cout << "We caught something! Let's rethrow it." << endl;
  throw;                                          // Rethrow current exception
}
```

这是一个 catch-all 块，它可以捕获所有抛出的异常。每次捕获到一个异常时，就会显示一个消息，并重新抛出该异常，让外层 try 块的 catch 块捕获它。外层 try 块的 catch 块会正确标识异常的类型，并显示其 what()成员返回的字符串。

17.4 抛出异常的函数

函数可以抛出在调用函数中捕获的异常，如程序示例 17.2 所示。对于这种情况，只需使异常在函数内抛出，但不在该函数中捕获。当然，如果不希望程序中断，就需要在某个地方捕获异常，为此，函数调用必须包含在 try 块中。然后，这个 try 块的处理程序就能捕获该异常。

当然，函数体可以包含自己的 try 块来处理自己的异常。未捕获的所有异常会传递到调用该函数的地方。有时把整个函数体放在带有一组处理程序的 try 块中比较方便，而这可以通过函数 try 块来实现。

17.4.1 函数 try 块

在函数体的左花括号前面加上关键字 try，就定义了一个函数 try 块。然后在函数体的右花括号后面加上函数 try 块的处理程序。例如：

```
void doThat(int argument)
try {
  // Code for the function...
}
catch(BigTrouble& ex) {
  // Handler code for BigTrouble exceptions...
}
catch(MoreTrouble& ex) {
  // Handler code for MoreTrouble exceptions...
}
catch(Trouble& ex) {
// Handler code for Trouble exceptions...
}
```

整个函数体是一个 try 块，而 catch 块放在函数体的右花括号后面。

当然，抛出异常的函数不一定有函数 try 块，或者其他 try 块。但是，对抛出异常的函数调用应放在 try 块中，否则未捕获的异常就会中断程序。还应说明在这种情况下，函数可能会抛出异常。

1. 指定函数可能抛出的异常

在默认情况下，函数可以抛出任何类型的异常。这不是非常好，因为如果要确保捕获所有的异常，就要对每个调用该函数的 try 块使用带省略号的“catch-all” catch 块。有时抛出异常是非常不方便的，必须确保(并强制)函数不抛出异常。

在函数头中添加异常说明，可以指定函数可能抛出的异常集。异常说明有 3 个选项。如表 17-1 所示。

表 17-1　异常说明

异 常 说 明	可能抛出的异常
没有异常说明	函数允许抛出任何类型的异常
throw()	不能抛出异常
throw(异常类型列表)	可以抛出括号中指定的异常类型，这些异常类型用逗号隔开

异常说明限制了函数可以抛出的异常类型。如果函数有异常说明，就必须在函数声明和函数定义中列出它们。

下面的函数定义包含异常说明：

```
void doThat(int argument) throw(Trouble, MoreTrouble)
try {
  // Code for the function...
}
catch(BigTrouble& ex) {
  // Handler code for BigTrouble exceptions...
}
```

这个函数可以处理 BigTrouble 类型的异常，但 Trouble 和 MoreTrouble 类型的异常必须由调用函数来捕获。上述函数的声明也需要包含异常说明：

```
void doThat(int argument) throw(Trouble, MoreTrouble);
```

如果声明一个包含异常说明的函数指针，也必须包含异常说明。例如：

```
void (*pFunction)(int) throw(Trouble, MoreTrouble);// Pointer to function declaration
pFunction = doThat;                                // Store function address
```

如果使用 typedef 来定义函数指针的类型，就不能包含异常说明，因为异常说明不是类型的一部分。但是，在指针声明中使用该类型时，必须包含它，如下所示：

```
typedef void (*FunctionPtrType)(int);    //  Define pointer to function type
FunctionPtrType pFunction throw(Trouble, MoreTrouble); // Pointer to function
                                                       // declaration
```

第一个语句定义了类型，该类型在第二个语句中使用。只能在指针声明中包含异常说明。

2. 未预料到的异常

函数应仅抛出其异常说明允许抛出的异常类型，但编译器一般无法确认抛出的异常是允许的类型。带有异常说明的函数限制了它可以抛出的异常类型，但这种函数可以调用没有异常说明的其他函数。这些函数所抛出的异常可以不在异常说明中指定的类型集中。C++通过未预料到的异常来识别这种可能性。

如果函数抛出的异常不在其异常说明中指定的类型集合中，它就不能传递到函数的外部，因为这会使异常说明无效。该异常会引发一系列事件，首先是调用标准库函数 unexpected()。完整的事件序列如图 17-8 所示。

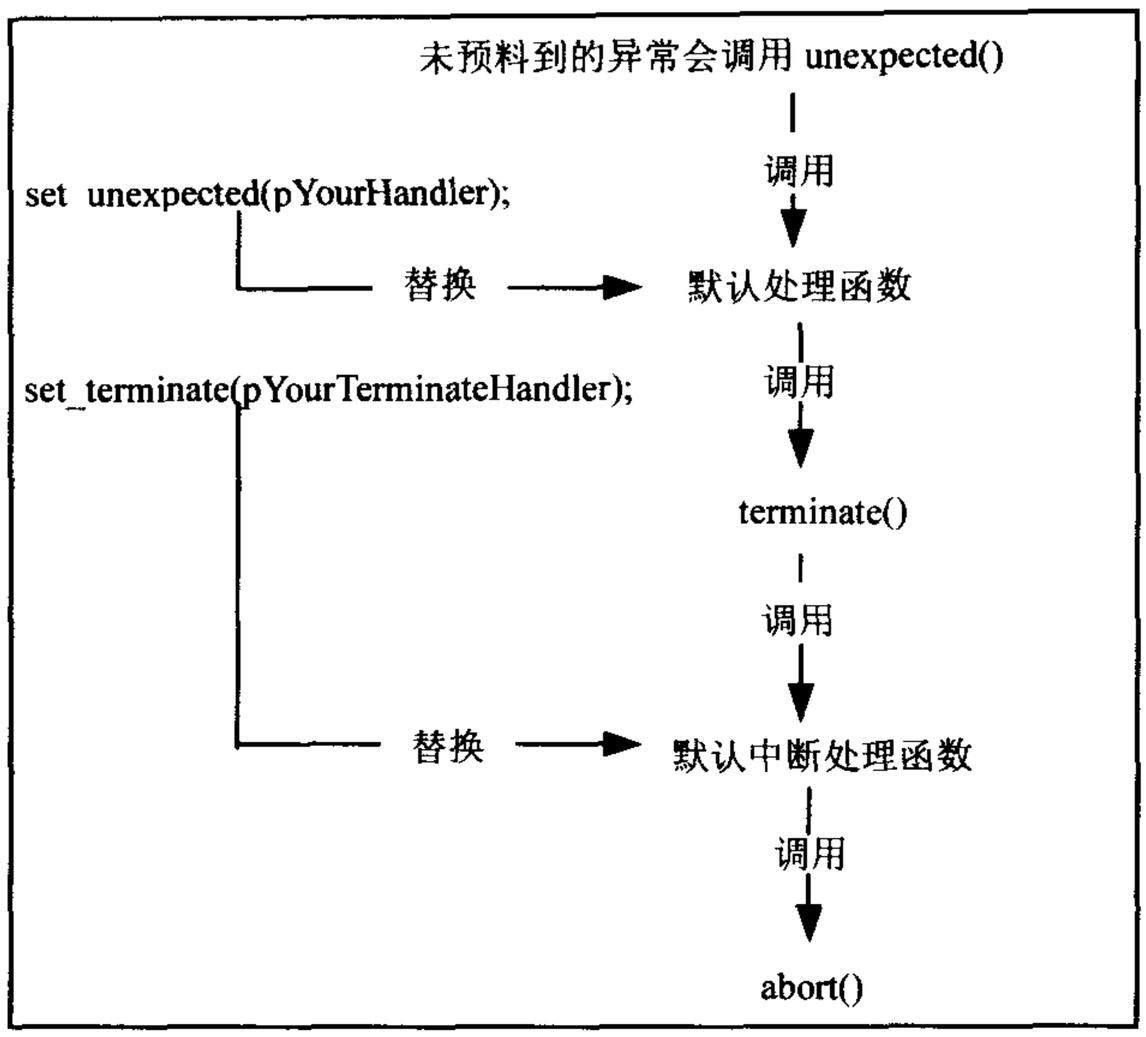


图 17-8　处理未预料到的异常

这一系列的调用，主要是为了更好地处理未预料到的异常，而不仅仅是使程序突然暂停。unexpected()函数调用一个处理函数，其默认操作是调用标准库函数 terminate()(该函数用于未捕获的异常)。如前所述，terminate()再调用默认的中断处理函数，该函数又调用 abort()，以结束程序。

可以用自己编写的函数来替代 unexpected()调用的默认处理函数，其技术类似于替换 terminate()调用的默认中断处理函数所采用的技术。具体方法是把处理函数的地址作为参数，调用 set_unexpected()函数。set_unexpected()函数的参数类型定义如下：

```
typedef void(*unexpected_handler)();
```

从 unexpected_handler 类型的定义来看，函数必须不带参数，也不能有返回值。在 unexpected_handler()函数中，可以执行需要的操作，但必须以下列方式结束函数：

- 抛出一个异常，该异常遵循抛出未预料到异常的函数的异常声明。
- 抛出 bad_exception 类型的异常，这是一个标准类型，在<exception>头文件中定义
- 调用 terminate()、exit()或 abort()

与中断处理程序一样，可以用不同的处理程序来处理不同的情况，以自己的方式自由处理未预料到的异常。std::bad_exception 类型的作用是，允许为这种类型的异常提供一个处理程序，处理未预料到的异常。有异常说明的所有函数都需要把这个类型包括进来，作为一个允许的异常。这样，就可以控制未预料的异常如何处理，而不是通过调用 terminate()进行默认的处理(即调用 abort())。但是，如果处理未预料到的异常的处理函数抛出了 bad_exception 类型的异常，而这种类型又不包含在函数的异常说明中，就一定要调用 terminate()了。

17.4.2　在构造函数中抛出异常

构造函数没有返回值，但可以抛出异常。在创建对象的过程中，抛出异常允许警示出现了

错误。前几章使用的 Box 类就是这种情况。如果给构造函数参数提供了无效的尺寸，就可以抛出异常。本章稍后会讨论这个 Box 类构造函数。

但是，在构造函数中抛出异常时需小心，特别是构造函数动态分配内存的情形。如果从构造函数中抛出了异常，所构造的对象就不会正常创建，因此对象的析构函数也就不会调用。最好的情形是，调用所有完整的子对象的析构函数。最糟糕的情形是，在自由存储区中分配的所有内存都不会以正常方式释放。此时，自由存储区的内存必须在异常处理过程中释放。

在构造函数中抛出异常的一个原因是运算符 new 的默认操作。如果运算符 new 没有成功地分配需要的内存，它就会抛出 bad_alloc 类型的异常。

构造函数是一个函数，所以它可以有函数 try 块。而且，构造函数中的 try 块可以包含初始化列表，因此也可以捕获子对象的构造函数抛出的异常。如果构造函数可以抛出异常，处理它的一种方式就是为构造函数提供一个函数 try 块，并添加处理程序，在出现错误时清理内存。这个类构造函数的形式如下所述：

```
Example::Example(int count) throw(bad_alloc)
try : BaseClass(count) {
  // Allocate some memory...
  // Rest of the constructor...
}
catch(...) {
  // Release memory as necessary...
}
```

其中，Example 类的构造函数在初始化列表中调用基类 BaseClass 的构造函数。这个调用和构造函数体放在函数 try 块中，因此由基类构造函数抛出异常，或当前对象的构造函数抛出异常时，就会调用处理程序。注意这里不需要在 catch 块中重新抛出异常。在构造函数 try 块中抛出的异常会自动重新抛出。

17.4.3 异常和析构函数

因为在抛出异常时，自动对象会释放，所以，一些类析构函数会在执行异常的处理程序之前调用。在析构函数中，知道因抛出了异常(而不是因为对象超出了作用域)而调用析构函数是非常重要的。此时，在析构函数中抛出异常，就会调用 terminate()，立即结束程序。这将防止执行源异常的 catch 块，在许多情况下，执行源异常的 catch 块会导致灾难。在这种情况下，需要确保在析构函数中不抛出异常，以执行源异常的处理程序。

一般的规则是，析构函数不应抛出异常，但如果它们必须抛出异常，就应在因抛出了异常而调用析构函数时，调用一个函数。如果抛出了异常，且没有执行对应的 catch 块，函数 uncaught_exception()就会返回 true，在析构函数中执行合适的操作。当然，为了防止异常超出析构函数的界限，可以把代码放在 try 块中，再使用处理程序捕获异常。

17.5 标准库异常

在标准库中定义了几个异常类型。它们都派生于标准类 std::exception，而 exception 在头文

件<exception>中定义。exception 类包含一个默认的构造函数和副本构造函数、副本赋值运算符和虚函数 what()，该函数返回一个描述异常的非空字符串。这些函数都不会抛出异常。派生于 exception 的标准异常类如图 17-9 所示。

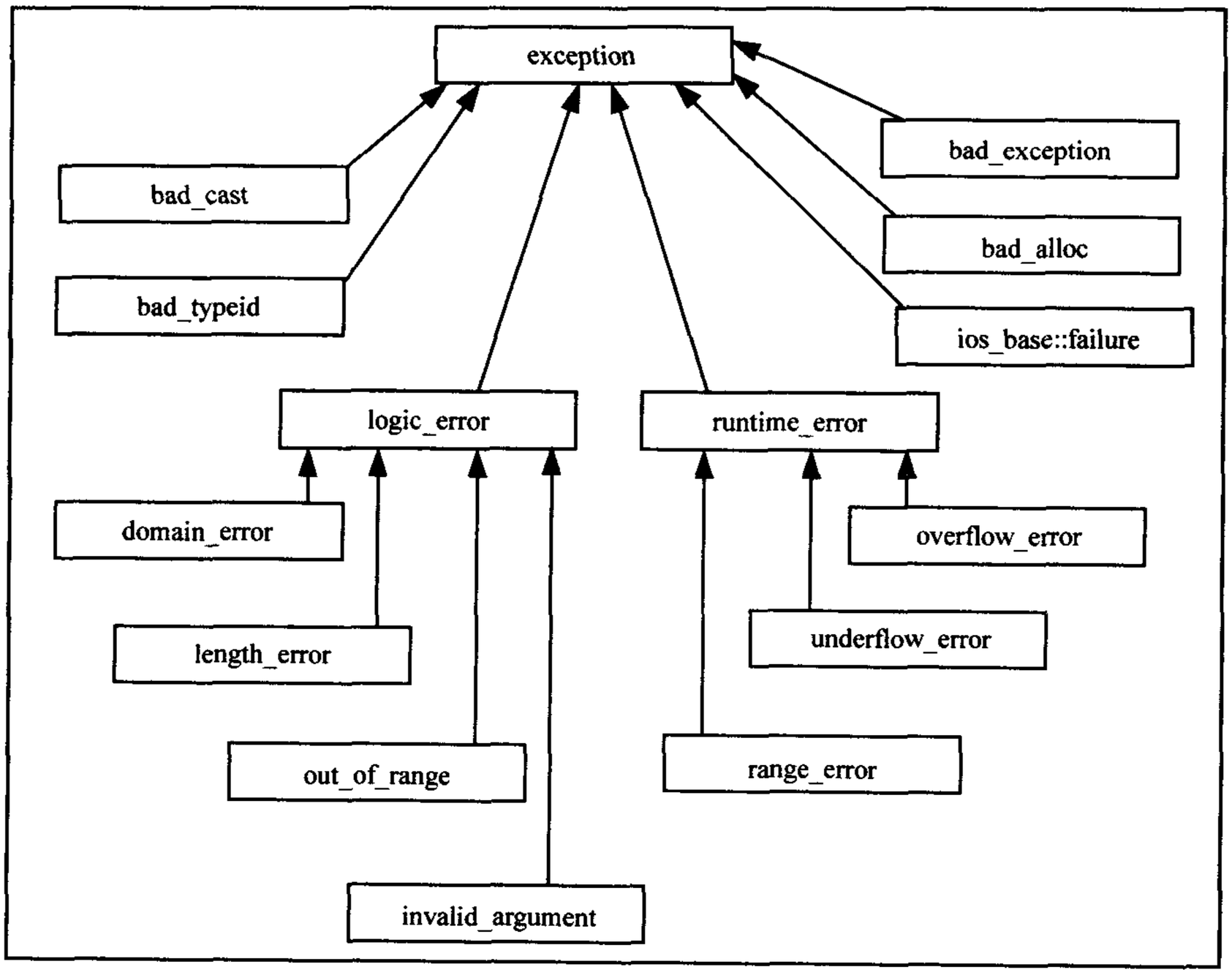


图 17-9　标准异常类

前面介绍了 dynamic_cast<>()运算符抛出的 bad_cast 异常。在第 7 章讨论运算符 new 时提到了 bad_alloc 异常。本章前面又介绍了 bad_exception 类。如果对空指针使用 typeid()运算符，就会抛出 bad_typeid 异常。ios_base::failure 异常由标准库中支持流输入输出的函数抛出。第 19 章会讨论流。

其他类型的异常分为两组，每一组都有一个派生于 exception 的基类。它们都在头文件<stdexcept>中定义。把 logic_error 作为基类的异常类型是在程序执行前为可以检测到的错误抛出的异常(至少在原则上可以检测到)，因为它们是因程序逻辑中的错误而抛出的。另一组异常派生于 runtime_error，它们通常是为与数据相关、且只能在运行期间检测出来的错误而抛出的。标准库函数抛出的异常类型表示各种类型的错误。例如，如果使用 at()成员函数访问 string 对象中的字符，且索引值在对象的合法范围之外，就会抛出 out_of_range 类型的异常。

如果 catch()的基类参数匹配任意派生类异常，就可以使用 catch 块的 exception&类型的参数，捕获任何类型的标准异常。当然，也可以使用 logic_error&类型或 runtime_error&类型的参数，捕获标准库函数抛出的一组或另一组异常。

17.5.1　标准库异常类

所有标准异常类都以 exception 为基类，所以应理解这个类有哪些成员，因为它们可以被

其他异常类继承。exception 类的定义在标准库头文件 exception 中，如下所示：

```
class exception {
  public:
    exception() throw();                                    // Default constructor
    exception (const exception&) throw();                   // Copy constructor
    exception& operator=(const exception&) throw(); // Assignment operator
    virtual ~exception() throw();     // Destructor
    virtual const char* what () const throw();      // Return a message string
};
```

与其他 C++标准库类一样，exception 类在命名空间 std 中定义。在每个成员声明中都有的 throw()是前面讨论的函数异常说明。这可以确保 exception 类的成员函数不会抛出异常。

注意 exception 类没有数据成员。成员 what()返回的非空字符串在函数定义体内定义，其实现方式是相互关联的。这个函数声明为 virtual，因此在派生于 exception 的类中也是虚函数。虚函数可以输出对应于每种异常类型的消息，提供了一种记录所抛出异常的经济方式。可以为 main()函数提供一个函数 try 块，再加上一个处理 exception 类型异常的 catch 块：

```
int main()
try {
  // Code for main...
}
catch(exception& rEx) {
  cout << endl << typeid (rEx) .name () << " caught in main: " << rEx. what ();
}
```

catch 块会捕获所有以 exception 作为基类的异常，并显示类类型和 what()函数返回的消息。因此，这个简单的机制给出了程序中抛出的、但没有捕获的异常信息。如果程序使用不是派生于 exception 的异常类，用省略号代替参数类型的 catch 块就会捕获其他异常，但此时得不到异常的信息。

这是一个很方便的 catch-all 机制，但本地 try 块可以直接本地化抛出异常的源代码。

17.5.2 使用标准异常

使用在标准库中定义的异常类有几个非常好的原因。使用标准库异常有两种方式：可以在自己的程序中抛出标准类型的异常；也可以把标准异常类作为自己异常类的基类。

显然，如果打算抛出标准异常，就需要在适当的情况下抛出它们。也就是说，不应抛出 bad_cast 异常，因为它们已经有了非常明确的目的。但是，可以在程序中直接使用派生于 logic_error 和 runtime_error 的异常类。在下面的例子中，当给 Box 类的构造函数提供无效的尺寸参数时，会抛出 range_error 异常：

```
Box::Box(double lv, double wv, double hv) throw(std::range_error) {
  if(lv<= 0.0 || wv <=0.0 || hv <= 0.0)
    throw std::range_error();

  length =lv;
  width = wv;
```

```
  height = hv;
}
```

当然，源文件需要包含定义了 range_error 类的<stdexcept>头文件。如果任何一个参数的值是 0 或负数，构造函数体都会抛出 range_error 异常。构造函数定义包含一个异常说明，只允许抛出 range_error 异常。构造函数的类定义必须也包含相同的异常说明。

派生自己的异常类

从一个标准异常类中派生自己的类时，一个要点是自己的类会成为标准异常系列中的一员。这样就可以在相同的 catch 块中捕获标准异常和自己的异常了。例如，如果异常类派生于 logic_error，则参数类型为 logic_error&的 catch 块就可以捕获该异常和标准 logic_error 异常。参数类型为 exception&的 catch 块总是捕获标准异常，包括自己的标准异常，只要该异常把 exception 作为基类即可。

让 Trouble 异常类派生于 exception 类，就可以轻松地使用它(和派生于它的异常类)。具体方法是修改类定义，如下所示：

```
class Trouble : public std::exception {
  public:
    Trouble(const char* pStr = "There's a problem") throw();
    virtual ~Trouble() throw();
    virtual const char*what()const throw();
  private:
    const char*pMessage:
};
```

这为在基类中定义的虚拟成员 what()提供了实现代码。这个版本会显示类对象中的消息。利用前面介绍的函数异常说明，为每个成员函数添加一个异常说明，这样在函数内部就不会抛出异常了。还需要以类似的方式更新派生于 Trouble 的类 MoreTrouble 和 BigTrouble 的成员函数。成员函数的定义必须包含与类定义中的函数相同的异常说明。

17.6 本章小结

异常是 C++的一个组成部分。几个运算符会抛出异常，它们可以用于标准库中，以警示错误。因此即使不打算使用自己的异常类，掌握它们的工作原理也是非常重要的。本章的要点如下所示：

- 异常是用于在程序中警示错误的对象。
- 可能抛出异常的代码通常包含在 try 块中。
- 处理在 try 块中抛出的各种类型异常的代码放在该 try 块后面的一个或多个 catch 块中。
- try 块及其 catch 块可以嵌套在另一个 try 块中。
- 参数为基类类型的处理程序可以捕获派生类类型的异常。
- 如果异常没有被 catch 块捕获，就调用 terminate()函数，该函数会接着调用 abort()。
- 标准库定义了一组标准异常。

- 异常说明限制了函数可以抛出的异常类型。
- 如果函数抛出的异常类型不在该函数的异常说明的允许范围之内，就会调用 unexpected()函数。
- 可以改变 unexpected()函数的默认操作，方法是实现自己的 unexpected()处理程序，把该函数指针传送给 set_unexpected()函数，以建立该处理程序。
- 构造函数的函数 try 块可以包含初始化列表和构造函数体。
- uncaught_exception()函数允许检测因抛出异常而调用的析构函数。

17.7 练习

1. 从标准 exception 类中派生自己的异常类 CurveBall，表示一个随机错误，再编写一个函数，在时间过了大约 25%时抛出这个异常(一种方式是生成一个 1 到 20 之间的随机数，如果该数小于或等于 5，就抛出异常)。定义函数 main()，调用这个函数 1000 次，记录并显示异常抛出的次数。

2. 定义另一个异常类 TooManyExceptions，在第 1 题中，当捕获的异常数超过 10 次时，就从 CurveBall 异常的 catch 块中抛出这个类型的异常。

3. 在第 1 题的代码中实现自己的中断处理程序，在抛出 TooManyExceptions 异常时显示一个消息。

4. 在稀疏数组中，大多数元素的值都是 0 或空。给类型为 string 指针的一维稀疏数组元素定义一个类，仅存储数组中的非 0 元素。元素的个数应指定为构造函数参数，因此至多可以存储 100 个 string 对象的稀疏数组声明如下：

```
SparseArray words(100);
```

为 SparseArray 类实现下标运算符，以使用数组表示法提取或存储元素。如果在下标运算符函数中超出了合法的索引范围，就抛出异常(提示：在内部使用链表，让每个节点存储一个元素及其下标)。

第 18 章　类　模　板

模板是自动生成新类类型的一种强大机制。标准库的一个重要部分完全建立在定义类模板的功能基础之上，所以理解类模板所涉及的技术就非常重要了。

本章主要内容

- 类模板的概念和定义
- 类模板的实例是什么，如何创建它
- 如何在类模板定义体的外部，为类模板的成员函数定义模板
- 类型参数与非类型参数的区别
- 类模板的静态成员如何初始化
- 类的部分说明是什么，如何定义它
- 类模板如何嵌套在另一个类模板中

18.1　理解类模板

类模板与第 9 章介绍的函数模板类似。类模板是一个参数化类型，也就是使用一个或多个参数创建一系列类的方式，其中对应于每个参数的变元一般是(但并不总是)一个类型。在使用类模板声明变量时，编译器会使用类模板创建类的定义，该定义对应于在声明中使用的模板参数。以这种方式使用类模板，可以生成任意多个不同的类。

与普通类一样，类模板有一个名称和一组参数。在命名空间中，类模板的名称必须惟一，在声明该模板的命名空间中，不能有其他类或模板与之同名。只要为类模板的每个参数提供一个参数，就可以从类模板中生成类定义。如图 18-1 所示。

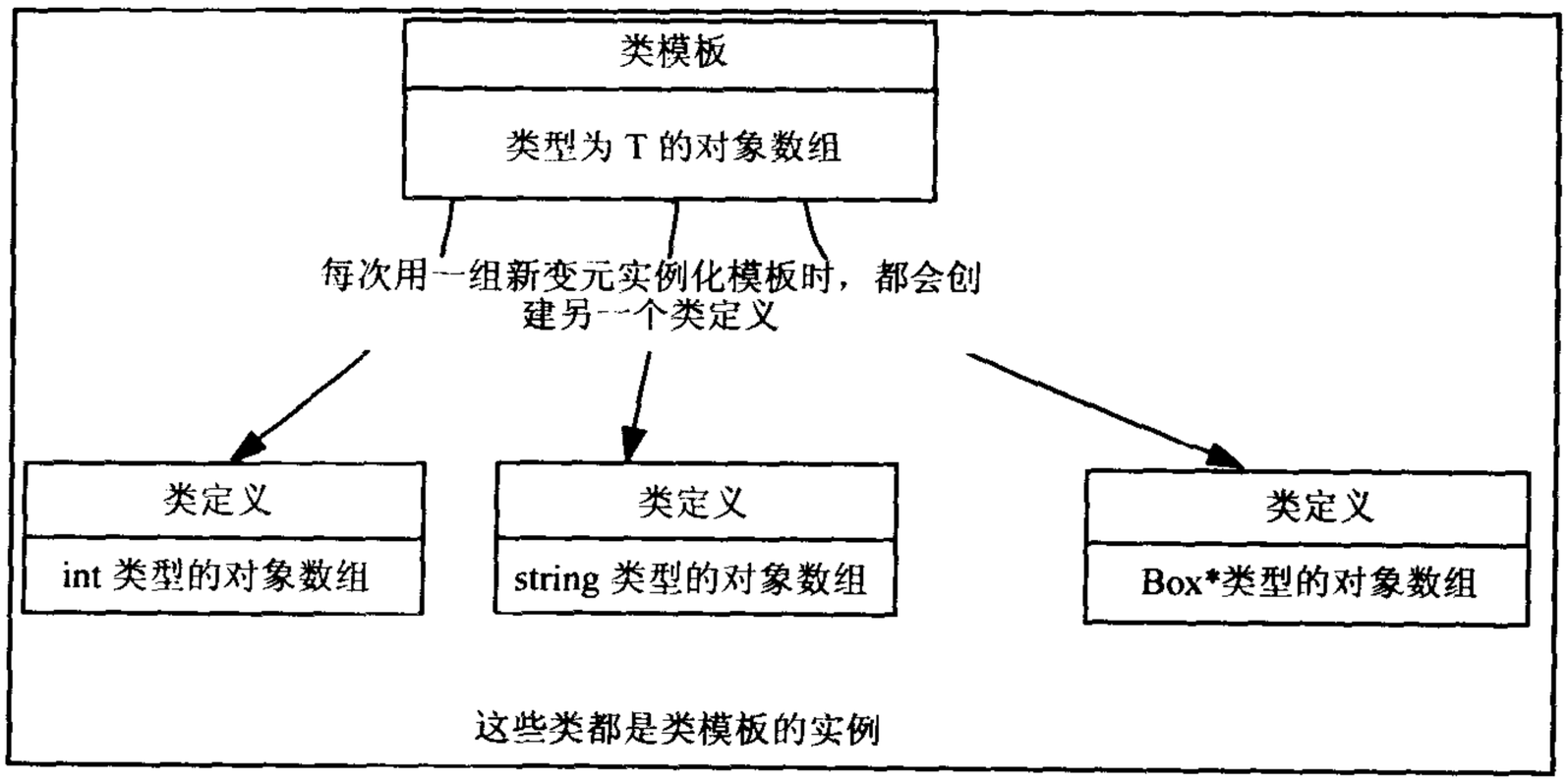


图 18-1　实例化模板

编译器从类模板中生成的每个类都称为该类模板的一个实例。使用模板类型声明变量会创

建该模板的一个实例，也可以显式声明类模板的实例，而不需要在同时声明一个变量。从模板中实例化的类不会重复，因此一旦创建了模板实例，它就可以用于声明该类型的变量。

类模板的应用

类模板有许多应用，最常见的应用是定义容器类。这些类可以包含以特定方式组织起来的给定类型的对象集。例如，对象数组、叠加堆栈或对象的链表。重要的是所使用的存储方法独立于所存储的对象类型。

类模板提供的工具可以定义能存储任意类型的对象的容器。模板参数可以用于指定容器要存储的对象类型。标准模板库是标准 C++实现的一部分，其中包含的许多模板定义了各种类型的容器。第 20 章将介绍如何使用它们。

下面考虑一个可以使用类模板的情形。假定 C++中的数组不能检查用户提供的索引值是否合法，因为这可能会在无意中改写不在数组界限之内的内存位置。当然，一种解决方案是编写自己的 Array 类，检查索引值是否在合法的界限内。这需要为类定义 operator[]()函数来检查数组索引值，如果索引值不合法，就抛出异常。

但这个 Array 类表示什么类型的数组？我们有时需要 double 类型的数组，有时则需要 string 对象的数组，或任何类型的对象数组。这样，就需要为要使用的每种数组类型定义一个类，即使类非常类似，也要定义。编写了这样几个类之后，这些实际上相同的类看起来相当烦琐，也不必要。而且，每个类还需要使用不同的名称，最后会得到名称为 ArrayOfDouble、ArrayOfString、ArrayOfBox 等的类。

此时使用类模板最合适不过了，因为类模板可以根据所需要的类型生成一个 Array 类。一旦把 Array 定义为一个类模板，就可以自动创建新的 Array 类，以管理所需类型的对象。实际上，不必自己定义 Array 模板。标准库已经提供了一个包含上述功能的模板。

18.2 定义类模板

在第一次看到类模板的定义时，它们看起来比实际的要复杂，这主要是因为定义它们所使用的表示法看起来比较复杂，定义中的参数也比较烦琐。类模板定义基本上类似于一般类的定义，但与其他事物一样，其中的一些细节要特别注意。

用关键字 template 定义类模板，把模板的参数放在关键字 template 后面的尖括号中。之后，编写模板类定义，先使用关键字 class，之后是类模板名和放在花括号中的定义代码。与普通类一样，整个定义以分号结束。类模板的一般形式如下所示：

```
template <模板参数列表> class ClassName {
  //Template class definition…
};
```

在这个定义中，ClassName 是模板的名称。为模板体编写代码的方式与为普通类编写代码相同，但一些成员的声明和定义要根据模板参数来编写。模板的参数放在尖括号中，用逗号隔开。要从这个模板中创建类，需要指定列表中的每个参数。

18.2.1 模板参数

模板参数列表可以包含两种参数：类型参数和非类型参数，列表中可以有任意多个参数。对应于类型参数的变元是一个类型，例如 int、string、Box 等，而对应于非类型参数的变元是给定类型的值，如 200，或给定类型的变量，如 ivalue。模板中的类型参数比非类型参数普遍得多，所以非类型参数在本章后面介绍。

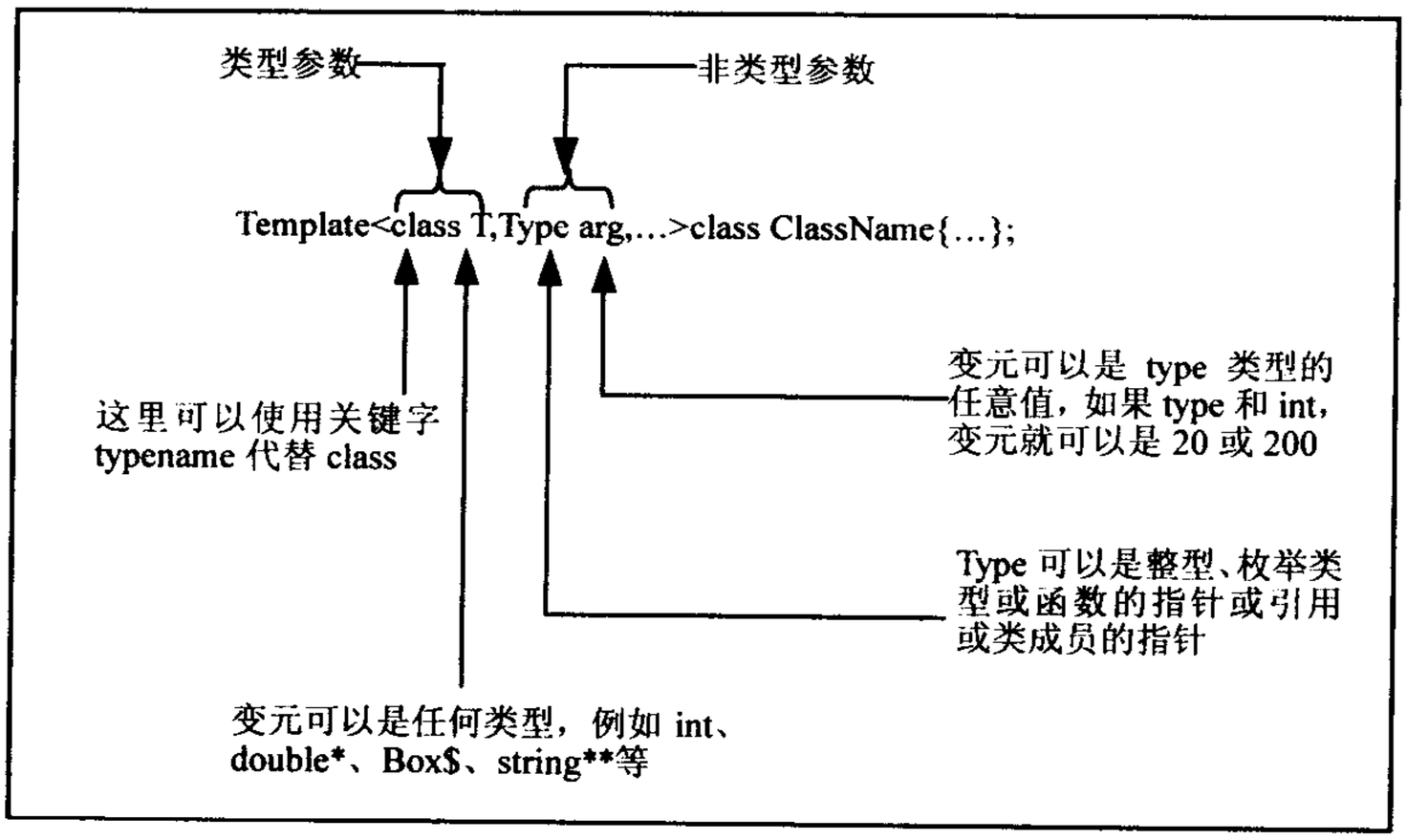


图 18-2 类模板参数

类型参数通常使用关键字 class 后跟参数名来表示(如图 18-2 中的 class T)，也可以使用关键字 typename 代替 class，即这里也可以使用 typename T。其中 T 常常用作类型参数名(当模板有多个类型参数时，则使用 T1、T2 等)，也可以使用自己喜欢的名称。

提示：

尽管关键字 class 似乎暗示，类型参数的参数必须是一个类，但实际上可以给参数提供任意类型，因此，如果使用 typename，可能会使其含义更清楚一些。

下面介绍一个简单的类模板，它有一个类型参数，以此说明类模板的本质。

18.2.2 简单的类模板

下面用前面的一个例子来说明，为数组定义一个类模板，该数组要对索引值进行边界检查，确保索引值是合法的。尽管标准库提供了数组模板的完整实现方式，但建立自己的数组模板有助于理解模板的工作原理。我们已经很清楚数组的工作原理了，因此下面集中讨论模板的特性。这也更容易使用第 20 章介绍的标准库中的 Array 模板。

数组模板只有一个类型参数，所以该模板的定义如下：

```
template<typename T>class Array {
  // definition of the template…
}
```

Array 模板只有一个类型参数 T。说它是类型参数，是因为它的前面有关键字 typename。在实例化模板时，为这个参数指定的参数，如 int、double*、string 等，确定了存储在相关类对象中的元素类型。模板体中的定义与类定义非常类似，其数据成员和成员函数可以声明为 public、protected 或 private，类模板一般有构造函数和析构函数。可以使用 T 或类型指针 T*声明变量或指定成员函数的参数或返回类型，而且，还可以把模板名称(在本例中是 Array)用作类型名称，也可以在构造函数和析构函数的声明中使用模板名称。

要利用类接口，至少需要构造函数、副本构造函数(因为要为数组动态地分配内存)、副本赋值运算符(因为如果没有提供该运算符，编译器就会提供一个)，重载的下标运算符和析构函数。因此，模板的最初定义如下所示：

```
template <typename T> class Array {
  private :
    T* elements;          // Array of type T
    size_t size;          // Number of elements in the array

  public:
    explicit Array<T>(size_t arraySize);      // Constructor
    Array<T>(const Array<T>& theArray) ;      // Copy Constructor
    ~Array<T>() ;                             // Destructor
    T& operator[](size_t index);    // Subscript operator
   const T& operator[](size_t index) const;     // Subscript operator-const arrays
    Array<T>& operator=(const Array<T>& rhs); // Assignment operator
};
```

模板体看起来类似于普通的类定义，只是在许多地方都使用了 T。例如，有一个数据成员 elements，其类型是 T 的指针(等价于 T 的数组)。在实例化类模板，生成一个类定义时，T 就会被用于实例化模板的类型代替。如果为 double 类型创建模板的一个实例，elements 的类型就是 double 数组。

给成员 size 使用 size_t 类型，它存储了数组中的元素个数。这是在标准头文件<cstddef>中定义的标准整数类型，对应于 sizeof()运算符返回的类型值。它是指定数组维数的首选类型。

注意第一个构造函数声明为 explicit。因为这个函数带有一个整数参数，所以编写构造函数调用有两种方式：

```
Array<int> data(5);      //explicit constructor notation
Array<int> numbers=5;   //assignment-like notation
```

把构造函数声明为 explicit，可以防止出现第二种语法形式(很不直观)，还可以防止给需要 Array<int>类型参数的函数传送整数。构造函数中如果没有 explicit 声明，编译器就会插入一个构造函数调用，把整数参数转换为 Array<int>类型。

下标运算符重载为 const。第 9 章介绍了带有 const 参数和非 const 参数的重载函数。下标运算符的非 const 版本应用于非 const 数组对象，可以返回数组元素的一个非 const 引用。因此这个版本可以放在等号的左边。const 版本用于调用 const 对象，返回元素的 const 引用。显然它不能放在等号的左边。

在副本赋值运算符的声明中，使用了类型 Array<T>&。这个类型是 Array<T>的引用。在类从模板中综合处理时，例如 T 是 double 类型，Array<T>&就是该类的类名引用，也就是

Array<double>。一般来说，模板某个实例的类名是由模板名后跟尖括号中的类型参数组成的。模板名后跟尖括号中的参数名列表称为模板 ID。

在模板定义中，不需要使用完整的模板 ID。在类模板体中，Array 本身就表示 Array<T>，而 Array&表示 Array<T>&，所以，可以把类模板定义简化为：

```
template <typename T> class Array {
  private:
    T* elements;                              // Array of type T
    size_t size;                              // Number of elements in the array
  public:
    explicit Array(size_t arraySize);  // Constructor
    Array(const Array& theArray);      // Copy Constructor
    ~Array();                                 // Destructor
    T& operator[](size_t index);       // Subscript operator
   const T& operator[](size_t index) const;  // Subscript operator-const arrays
   Array& operator=(const Array& rhs); // Assignment operator
};
```

提示：

如果需要在模板体的外部标识模板，必须使用模板 ID。在本章后面定义类模板的成员函数时，就要用到模板 ID。

赋值运算符允许把一个数组对象赋予另一个数组对象，而 C++中的一般数组不能这样做。如果因某种原因要保留这个功能，仍旧需要把 operator=()函数声明为模板的一个成员。否则，在需要对模板实例进行这个操作时，就会创建一个默认的公共赋值运算符。为了不使用赋值运算符，可以把它声明为类的私有成员，这样就不能访问它了。当然，在这种情况下，该成员函数不需要实现代码，因为除非要使用成员函数，否则 C++不需要实现它，而这个成员函数是从来都不会使用的。

定义类模板的成员函数

可以在类模板体中包含它的成员函数的定义。在这种情况下，成员函数隐含为模板所有实例的内联函数，这与普通的类一样。但是，有时需要在模板体的外部定义成员函数，特别是在成员函数包含许多代码时，就更是如此。在模板体的外部定义成员函数时，语法有些不同，初看时会觉得很令人沮丧。下面就介绍该语法。

理解该语法的线索是模板类的成员函数的定义本身就是函数模板。定义成员函数的函数模板中的参数列表必须与类模板的参数列表完全相同。这听起来有点混乱，下面举例说明。我们为 Array 模板的成员函数编写定义，首先编写构造函数。

在类模板定义的外部定义构造函数时，构造函数的名称必须用类模板名称来限定，所采用的方式与普通类成员函数相同。但是，这不是函数定义，而是函数定义的模板，所以也必须表示出来。下面是构造函数的定义：

```
template <typename T>                          //This is a template with parameter T
Array<T>::Array(size_t arraySize):size(arraySize)  {
  elements= new T[size];
}
```

其中，第一行把这个函数标识为模板，还把模板参数指定为 T。把模板函数声明放在两行上，只是为了演示得更清晰，如果整个声明可以放在一行上，就不必放在两行上。

在限定构造函数的名称 Array<T>时，模板参数是必须的，因为它把函数定义和类模板联系起来。注意这里没有使用关键字typename，该关键字仅用于模板参数列表。在构造函数名的后面不需要列出参数。在为类模板的实例实例化构造函数时，例如为类型 double 实例化构造函数，类型名会替换掉构造函数限定符中的 T，于是类 Array<double>的限定构造函数名应是 Array<double>::Array()。

在构造函数中，必须在自由存储区中为 elements 数组分配内存，该数组包含 size 个类型 T 的元素。如果 T 是类类型，就必须在类 T 中包含一个默认的公共构造函数。如果 T 不是类类型，就不会编译这个构造函数的实例。如果不能分配内存，运算符 new 就会抛出 bad_alloc 异常。Array 构造函数应总是放在 try 块中。

析构函数必须释放 elements 数组的内存，其定义如下所示：

```
template <typename T> Array<T>::~Array() {
  delete[] elements;
}
```

要释放数组专用的内存，必须使用 delete 运算符的正确形式。

副本构造函数必须为要创建的对象创建一个数组，其大小与参数相同，接着把后者的数据成员复制到前者中。其定义代码如下：

```
template <typename T> Array<T>::Array(const Array& theArray) {
  size=theArray.size;
  elements=new T[size];
  for(int i=0; i<size; i++)
    elements[i]= theArray.elements[i];
}
```

这段代码假定赋值运算符处理类型 T。这对为动态分配内存的类定义赋值运算符是非常重要的。如果类 T 没有定义副本构造函数，就使用 T 的默认副本构造函数，但这样动态分配内存的类会出现不希望的负面效应，如第 13 章所述。在使用之前不查看模板的代码，就可能认识不到副本构造函数对赋值运算符的依赖关系。

operator[]()函数相当简单，但应确保不能使用不合法的索引值。对于超出范围的索引值，可以抛出一个异常：

```
template <typename T> T& Array<T>::operator[](size_t index) {
  if(index < 0 || index >= size)
    throw std::out_of_range(index<0 ? "Negative index" :"Index too large");

  return elements[index];
}
```

这里可以定义自己的异常类，但借用在标准库的<stdexcept>头文件中定义的 out_of_range 类会更容易。例如，如果用超出范围的索引值引用 string 对象，就会抛出该异常。这样用法就一致了。如果索引的值不在 0 到 size-1 之间，就抛出 out_of_range 类型的异常。构造函数的参数是一个描述错误的 string 对象。对应于 string 对象的非空字符串(类型为 const char*)由异常对

象的 what()成员返回。传送给 out_of_range 构造函数的参数是一个简单的消息，但可以在 string 对象中包含索引值和数组大小等信息，以易于跟踪问题的源头。

下标运算符函数的 const 版本和非 const 版本大致相同：

```
template <typename T> const T& Array<T>::operator[](size_t index)const {
  if(index < 0 || index >= size)
    throw std::out_of_range(index<0 ? "Negative index" : "Index too large");

  return elements[index];
}
```

最后一个需要定义的函数模板是赋值运算符的函数模板。该函数模板需要释放在目标对象中分配的内存，再执行副本构造函数的操作——这当然要在检查对象是不同的之后进行。下面是定义：

```
template <typename T> Array<T>& Array<T>::operator=(const Array& rhs) {
  if(&rhs == this)          // If lhs == rhs
    return *this;           // just return lhs

  if(elements)              // If lhs array exists
    delete[]elements;       // release the free store memory

  size = rhs.size;
  elements = new T[rhs.size];
  for(int i = 0 ; i < size;i++)
    elements[i] = rhs.elements[i];
}
```

检查左操作数与右操作数是否相同是必不可少的，否则就是释放普通 elements 成员的内存，然后在它已不存在的情况下复制它。如果操作数不同，就释放左操作数占用的内存，之后创建右操作数的副本。

这里编写的所有定义都是函数模板的定义，它们都绑定到类模板上。但它们不是函数定义，而是在需要生成某个类成员函数的代码时由编译器使用的函数模板。因此需要在使用该模板的源文件中可用。为此，一般应将类模板的所有成员函数的定义都放在包含类模板的头文件中。

即使把模板的成员函数定义为独立的函数模板，它们仍可以是内联函数。为了让编译器把它们看作内联实现代码，只需在定义开头的 template<>之后加上关键字 inline，如下所示：

```
template <typename T> inline const T& Array<T>::operator[](size_t index)const {
  if(index < 0 || index >= size)
    throw out_of_range(index<0 ? "Negative index": "Index too large");

  return elements[index];
}
```

18.2.3 创建类模板的实例

声明一个对象，其类型是根据模板创建的，其结果是编译器创建的类模板的一个实例。例如：

```
Array<int> data(40);
```

要编译这个语句，需要两件事：必须声明类型 Array<int>，才能标识它；构造函数必须存在，因为要调用该构造函数来创建对象。这个语句创建了类模板的一个实例，即类 Array<int>，以及该类的一个构造函数实例。这些都是创建对象 data 所必须的，也是编译器目前提供的全部内容。

在模板定义中用 int 替代 T，就会生成包含在程序中的类定义。但这是一个编译。编译器只编译程序使用的成员函数，不会编译从模板参数的一次替代中生成的整个类。在声明对象 data 后，它等价于：

```
class Array<int> {
  private:
    int* elements;   // Array of type int
    size_t size;     // Number of elements in the array

  public:
    Array(size_t arraySize); // Constructor
};
```

除了构造函数之外，这里忽略了类的成员函数。编译器不会生成创建对象时不需要的成员实例，也不包含程序中不需要的模板部分。这表示，即使类模板中存在编码错误，程序也能成功编译、链接和运行。如果错误位于程序不需要的模板部分，编译器就不会发现它们，因为它们不会包含在已编译的代码中。显然，在程序中，除了使用其他成员函数的对象声明之外，肯定会包含其他语句。在任何情况下，都需要调用析构函数来释放对象。所以，程序中类的最终版本包含的内容要比上述的多。但是，从模板中生成的类的最终内容与实际在程序中使用的代码相同。

在声明中实例化类模板，称为模板的隐式实例化，因为它是声明对象的一个副产品。这个术语也把它与模板的显式实例化区分开来，稍后介绍模板的显式实例化。

data 的声明还调用了类构造函数 Array<int>::Array()，因此，编译器会使用定义构造函数的函数模板，为类的构造函数创建定义：

```
Array<int>::Array(long arraySize):size(arraySize) {
  elements=new int[size];
}
```

每次使用带另一个类型参数的类模板声明一个变量时，都会在程序中定义并包含一个新类。类对象的创建需要调用构造函数，并生成类构造函数的定义。当然，在创建模板的新实例时，并不需要创建以前已经创建的对象类型。编译器会在需要时使用以前创建的模板实例。

在使用某个类模板实例中的成员函数时，例如，在用模板定义的对象上调用函数，就会生成要使用的每个成员函数的代码。如果不使用某个成员函数，就不会创建该函数的模板实例。每个函数定义都创建为隐式模板实例化，因为它在函数的使用之外。模板本身并不是可执行代码的一部分。编译器只是自动生成需要的代码。如图 18-3 所示。

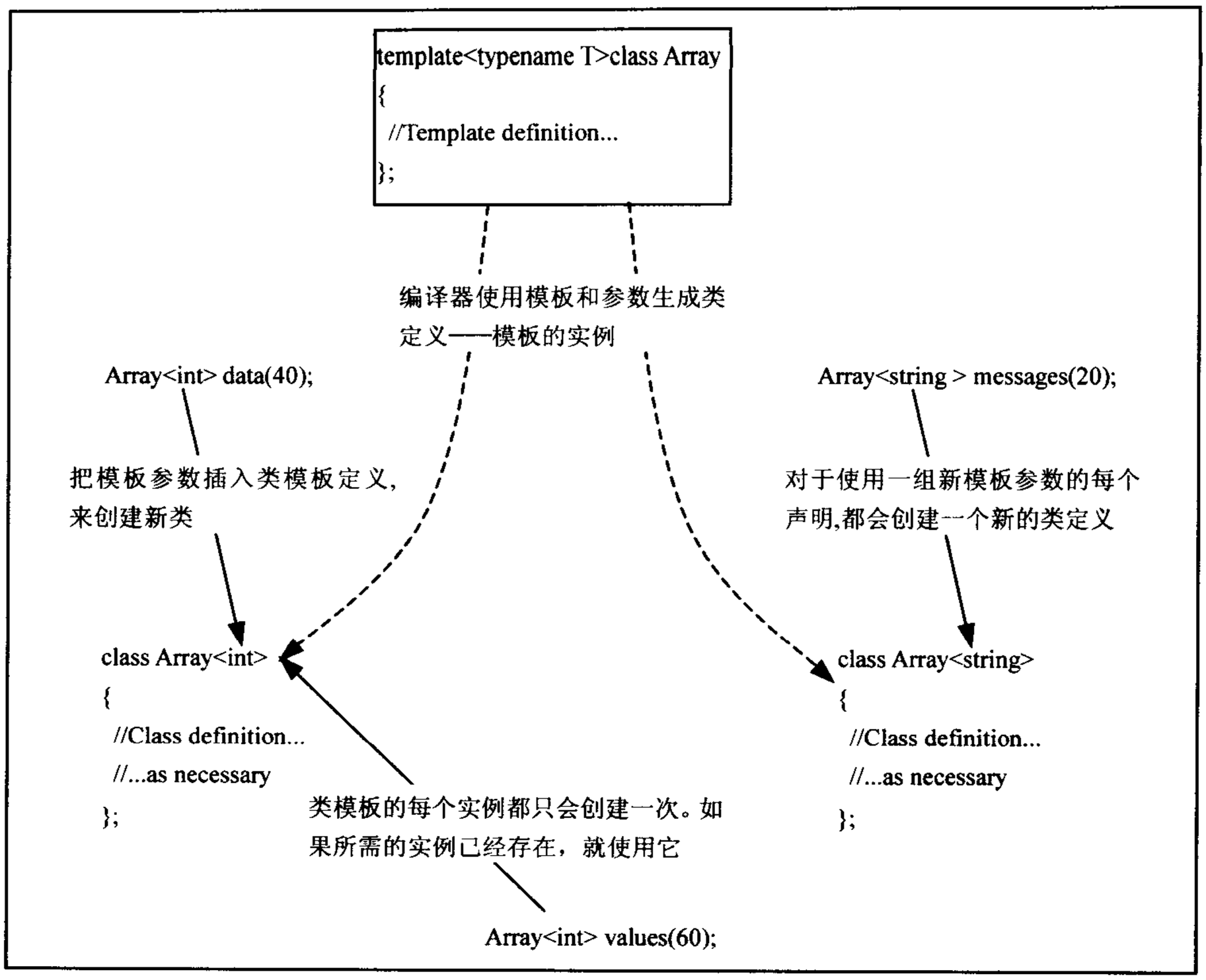


图 18-3　类模板的隐式实例化

注意，在需要创建对象类型时，类模板只能隐式实例化。声明对象类型的指针，并不会创建模板的实例。例如：

```
Array<string>* pObject;
```

这个语句把 pObject 声明为“类型 Array<string>的指针”类型。这个语句不会创建 Array<string>类型的对象，也不会创建模板实例。与此相反，下面的声明：

```
Array<string*> pMessages(10);
```

这个语句会创建类模板的实例。它声明了类型为 Array<string*>的对象，因此 pMessages 的每个元素都存储了 string 对象的指针。同时生成了定义类构造函数的模板实例。

下面用一个例子来试验 Array 模板。

程序示例 18.1——使用类模板

把类模板和定义模板的成员函数的模板都放在头文件 Array.h 中：

```
// Array class template definition
#ifndef ARRAY _H
#define ARRAY_H
#include <stdexcept>                          // For the exception classes

template <typename T> class Array {
  private :
```

```
    T* elements;                              // Array of type T
    size_t size;                              // Number of elements in the array

  public :
    explicit Array(size_t arraySize) ;  // Constructor
    Array(const Array& theArray);        // Copy Constructor
    ~Array();                             // Destructor
    T& operator[](size_t index);         // Subscript operator
    const T& operator[](size_t index) const;  // Subscript operator
    Array& operator = (const Array& rhs) ;    // Assignment operator
};

// Constructor
template <typename T>   // This is a template with parameter T
Array<T>::Array(size_t arraySize) : size(arraySize) {
  elements = new T[size];
}

// Copy Constructor
template <typename T>
Array<T>::Array(const Array& theArray) {
  size = theArray.size;
  elements = new T[size];
  for(int i = 0 ; i < size ; i++)
    elements[i] = theArray.elements[i] ;
}

// Destructor
template <typename T>
Array<T>::~Array() {
  delete[] elements;
}

// Subscript operator
template <typename T>
T& Array<T>::operator[](size_t index) {
  if(index < 0 || index >= size)
    throw std::out_of_range (index < 0 ? "Negative index" : "Index too large");

  return elements [index] ;
}

// Subscript operator for const objects
template <typename T>
const T& Array<T>::operator[](size_t index) const {
  if(index < 0 || index >= size)
    throw std::out_of_range (index < 0 ? "Negative index" : "Index too large");

  return elements [index] ;
}

// Assignment operator
```

```
template <typename T>
Array<T>& Array<T>::operator=(const Array& rhs) {
  if(&rhs == this)       // If lhs == rhs
    return *this;        // just return lhs

  if(elements)           // If lhs array exists
    delete[]elements;    // release the free store memory

  size = rhs.size;
  elements = new T[rhs.size];
    for(int i = 0 ; i < size ; i + +)
      elements[i] = rhs.elements[i];
}
#endif
```

要使用该模板，只需编写一个程序，用模板声明一些数组，并试验它们。我们还要试验一些超出范围的索引值，看看会发生什么情形：

```
// Program 18.1 Using a class template    File: prog18_01.cpp
#include "Box.h"
#include "Array.h"
#include <iostream>
#include <iomanip>
using std::cout;
using std::endl;

int main() {
  const int doubleCount =50;
  Array<double> values(doubleCount);        // Class constructor instance created

  try {
    for(int i = 0 ; i < doubleCount ; i++)
      values[i] = i + 1;      // Member function instance created

    cout << endl << "Sums of pairs of elements:";
    int lines = 0;
    for(int i=doubleCount - 1 ; i >= 0 ; i--)
    cout << (lines++ % 5==0 ? "\n" : "") << std::setw(5)
         << values[i] + values[i - 1];
  }
  catch(const std::out_of_range& ex) {
    cout << endl <<"out_of_range exception object caught! " << ex.what();
  }

  try {
    const int boxCount = 10;
    Array<Box> boxes(boxCount);              // Template instance created
    for(int i = 0 ; i <= boxCount ; i++) // Member instance created in loop
      cout << endl << "Box volume is " << boxes[i].volume();
  }
  catch(const std::out_of_range& ex) {
    cout << endl << "out_of_range exception object caught ! " << ex.what();
```

```
    }

    cout << endl;
    return 0;
}
```

头文件 Box.h 包含第 16 章的 Box 类定义，这个例子也需要在 Box.cpp 中包含 Box 类成员函数的定义。

这个例子的结果如下所示：

```
Sums of pairs of elements:
   99   97   95   93   91
   89   87   85   83   81
   79   77   75   73   71
   69   67   65   63   61
   59   57   55   53   51
   49   47   45   43   41
   39   37   35   33   31
   29   27   25   23   21
   19   17   15   13   11
    9    7    5    3
out_of_range exception object caught! Negative index
Box volume is 1
Box volume is 1
Box volume is 1
Box volume is 1
Box volume is 1
Box volume is 1
Box volume is 1
Box volume is 1
Box volume is 1
out_of_range exception object caught! Index too large
```

例子的说明

在 main()的开头，使用类模板和 double 参数类型，创建 Array<double>类型的对象：

```
Array<double> values(doubleCount);              //Template instance created
```

这个语句把 values 声明为类型 Array<double>，数组中的元素个数指定为 doubleCount。在编译器处理这个语句时，就会从类模板中为 Array<double>类创建一个定义。要创建对象 values，就必须调用构造函数(Array<double>:: Array(doubleCount))，因此编译器使用构造函数的函数模板，在程序中创建构造函数的定义。

在 try 块中，在 for 循环中，用 1 到 doubleCount 的值初始化 values 的元素：

```
for(int i=0; i<doubleCount; i++)
  values[i]=i+1;                          //Member function instance created
```

表达式 values[i]会创建下标运算符函数的一个实例。这个表达式会隐式地调用 values.operator[](i)。因为 values 不是 const，所以调用的是非 const 版本。

在 try 块中使用第二个 for 循环，输出连续两对元素的和，从数组的最后一个元素开始：

```
for(int i=doubleCount-1; i>=0; i--)
  cout << (lines++ % 5 ==0 ? "\n":"") << std::setw(5)
       << values[i]+values[i-1];
```

这段代码也调用下标运算符函数，但函数模板的实例也已创建，所以不会生成新实例。显然，在 i 等于 0 时，表达式 values[i-1]的索引值并不合法，这会让 operator[]()函数抛出一个异常。处理程序会捕获这个异常：

```
catch(const std::out_of_range& ex) {
  cout << endl << "out_of_range exception object caught!" << ex.what();
}
```

捕获 out_of_range 异常的 what()函数返回一个非空字符串，该字符串对应于创建异常对象时传送给构造函数的 string 对象。从输出中可以看出，重载的运算符函数抛出了一个负的索引值。

在下标运算符函数抛出异常时，控制权会立即传送给处理程序，这样就不会使用非法的元素引用了，也不会在非法索引所在的位置存储信息了。当然，循环也会在此结束。

在下一个 try 块中，定义了一个可以存储 Box 对象数组的对象：

```
Array<Box> boxes(boxCount);          // Template instance created
```

这次，编译器生成了类模板的一个实例 Array<Box>，它存储了一个 Box 对象的数组，因为以前没有用模板实例化 Box 对象。这个语句还调用 Box 类的构造函数，创建对象 boxes，因此会为构造函数创建函数模板的一个实例。在自由存储区中创建 Box 类的 elements 成员时，Array<Box>类的构造函数会调用 Box 类的默认构造函数。当然，elements 数组中的所有 Box 对象的默认尺寸都是 1×1×1。

在一个 for 循环中显示每个 Box 对象的体积：

```
for(int i=0; i<=boxCount; i++)    // Member instance created in loop
  cout << endl << "Box volume is " << boxes[i].volume();
```

表达式 boxes[i]调用了重载的下标运算符，因此编译器再次使用函数模板的实例生成该函数的定义。当 i 等于 boxCount 时，下标运算符函数会抛出一个异常，因为 boxCount 超出了 elements 数组的边界。try 块后面的 catch 块捕获该异常：

```
catch(const std::out_of_range& ex) {
  cout << endl << " out_of_range exception object caught! " << ex.what();
}
```

由于退出了 try 块，因此本地声明的所有对象都释放了，包括 boxes 对象。此时 values 对象仍存在，因为它不是在前面的 try 块中创建的，且仍在其作用域之内。

导出模板

把类模板的成员函数的代码放在头文件中有一个缺点：编译器必须在每个包含该头文件的源文件中处理这些代码。在广泛使用模板的大型程序中，这会大大增加处理的系统开销。而另

一个方法是把成员函数的模板放在一个单独的源文件中，并使之可用于需要访问它的其他源文件。此时，只需在每个模板中使用关键字 export，就可以导出函数模板定义。例如：

```
// ArrayTemplate.cpp file
#include <stdexcept>                    // For the exception classes
#include "Array.h"

// Constructor
export template <typename T>            // This is a template with parameter T
Array<T>::Array(size_t arraySize) : size(arraySize) {
  elements = new T[size];
}

// Copy Constructor
export template <typename T>
Array<T>::Array(const Array& theArray) {
  size = theArray.size;
  elements = new T[size];
  for(int i = 0 ; i < size ; i++)
    elements[i] = theArray.elements[i];
}

// Plus templates for other member functions as before, but with export keyword...
```

这个文件现在可以单独编译，生成的对象文件可以用作使用该类模板的程序的一部分。

与以前一样，头文件 Array.h 包含类模板，但只需要包含函数模板的声明：

```
// Array class template definition
#ifndef ARRAY_H
#define ARRAY_H

// Class template definition as before
template <typename T> class Array {
  private:
    T* elements;                              // Array of type T
    size_t size;                              // Number of elements in the array

public:
  explicit Array(size_t arraySize);           // Constructor
  Array(const Array& theArray);               // Copy Constructor
  ~Array();                                   // Destructor
  T& operator[](size_t index);                // Subscript operator
  const T& operator[](size_t index) const;    // Subscript operator
  Array& operator=(const Array& rhs);         // Assignment operator
};

#endif
```

如果程序要使用类模板，只需在需要该类模板的源文件中包含 Array.h 的这个简短版本，并把 Array.cpp 作为一个源文件，或者访问从该源文件中生成的对象文件(当它链接时)。

只有不是内联的函数模板才能导出。如果对内联的函数模板使用关键字 export，该关键字

会被忽略。如果一定要导出模板的定义，就不能在程序的其他地方重复该定义。导出模板的声明只能放在其他源文件中。不能导出在未指定的命名空间中定义的模板。

也可以把 export 关键字用于类模板。此时，会导出类模板中所有的非内联函数成员和函数成员模板，我们可以在单独的源文件中自由定义它们。把 export 关键字应用于类模板，也会导出类模板的静态数据成员、成员类和成员类模板，它们的定义可以放在单独的文件中。

18.2.4 类模板的静态成员

类模板可以包含静态成员，就像普通类一样。模板类的静态成员函数是相当简单的。类模板的每个实例会根据需要实例化类的静态成员函数。这种成员函数没有 this 指针，所以不能引用类的非静态成员。定义类模板的静态成员函数的规则与定义类的静态成员函数的规则相同，类模板的静态成员函数在模板的每个实例中都存在，就好像它在普通的类中一样。

静态数据成员比较有趣，因为它需要在模板定义的外部初始化。假定 Array 模板包含一个静态数据成员，则成员的声明和初始化它的模板如下所示：

```
template <typename T> class Array {
  private:
     static T value;                              // Static data member
    T* elements;                                  // Array of type T
    size_t size;                                  // Number of elements in the array

public:
  explicit Array(size_t arraySize);               // Constructor
  Array(const Array& theArray);                   // Copy Constructor
  ~Array();                                       // Destructor
  T& operator[](size_t index);                    // Subscript operator
  const T& operator[](size_t index) const;        // Subscript operator
  Array& operator=(const Array& rhs);             // Assignment operator
};

template <typename T > T Array<T>::value;    //Initialize static data member
```

初始化是通过模板完成的。静态数据成员总是依赖于模板的参数，所以必须用参数 T 把 value 初始化为一个模板。静态变量名也必须用类型名 Array<T>来限定，所以它用类模板的实例来标识。这里不能单独使用 Array，因为这个定义在模板体的外部，而模板 ID 是 Array<T>。

如果把 export 关键字应用于类模板，就可以把类模板的静态数据成员定义放在一个单独的文件中。

18.2.5 非类型的类模板参数

非类型参数看起来像函数的参数，它也是类型名后跟参数名。非类型参数的变元是给定类型的值。但是，不能给类模板中的非类型参数使用任意类型。非类型参数主要用于定义对指定容器有效的值，例如数组的维数或其他大小说明，或者是索引值的上下限。

非类型参数只能是整型类型，例如 int 或 long、枚举类型、对象的指针或引用(如 string*或

Box&)、函数的指针或引用、类成员的指针等。因此，非类型参数不能是浮点数类型或类类型，不允许使用 double、Box 和 string 类型，也不允许使用 string**。非类型参数的主要用途是允许指定容器的大小和上下限。当然，只要非类型参数的类型是一个引用，对应于非类型参数的变元就可以是类类型的对象。例如，对于类型为 Box&的参数，就可以使用 Box 类型的对象作为其参数。

非类型参数的代码类似于函数参数，即类型名后跟参数名。如下所示：

```
template<typename T, size_t size> class ClassName {
  //Definition using T and size…
};
```

这个模板有一个类型参数 T 和一个非类型参数 size。其定义根据这两个参数和模板名来设置。如果需要，也可以把类型参数的名称用作非类型参数的类型。例如：

```
template <typename T,          //T is the name of the type parameter
          size_t size,
          T value>             //T is also the type of this non-type parameter
class ClassName{
  //Definition using T ,size,and value…
};
```

这个模板有一个非类型参数 value，其类型是 T。在参数列表中，参数 T 必须放在使用它的非类型参数前面，因此，value 不能放在类型参数 T 的前面。注意，对类型参数和非类型参数使用相同的符号，会把 typename 参数的变元限制为非类型参数允许使用的类型(换言之，T 必须是整型类型)。

为了说明如何使用非类型参数，假定为数组定义了如下类模板：

```
template<typename T, int arraySize, T value> class Array {
  //Definition using T size,and value…
};
```

现在可以在构造函数中，用非类型参数 value 初始化数组的每个元素：

```
template <typename T, int arraySize, T value>
Array<T, size, value>::Array(size_t arraySize) : size(arraySize) {
  elements = new T[arraySize];
  for(int i = 0 ; i < arraySize, i++)
    elements[i] = value;
}
```

因为非类型参数只能是整型类型、指针或引用，所以不能创建存储 double 值的 Array 对象，模板的用途也就受到了限制。

18.2.6 非类型参数示例

在下面这个更切实的例子中，要给 Array 模板添加一个非类型参数，为数组的索引增加一些灵活性：

```
template <typename T, long startIndex> class Array {
```

```
  private:
    T* elements;                              // Array of type T
    size_t size;                              // Number of elements in the array

public:
  explicit Array(size_t arraySize);           // Constructor
  Array(const Array& theArray);               // Copy Constructor
  ~Array();                                   // Destructor
  T& operator[](size_t index);                // Subscript operator
  const T& operator[](size_t index) const;    // Subscript operator
  Array& operator=(const Array& rhs);         // Assignment operator
};
```

这段代码添加了一个 long 类型的非类型参数 startIndex。这样，就可以指定使用给定范围的索引值，例如从 - 10 到+10，此时，可以指定非类型参数值为 - 10，传送给构造函数的参数为 21，来声明一个数组，因为这个数组需要 21 个元素。

类模板现在有两个参数，定义类模板中成员函数的函数模板必须也有两个参数。即使一些函数不会使用非类型参数，也必须指定两个参数。参数是模板标识的一部分，要匹配类模板，它们必须有相同的参数列表。

前面的代码有一些严重的缺陷。添加新模板参数 startIndex 的一个后果是，参数的不同值会生成不同的模板实例。这意味着，从 0 开始索引的 double 数组与从 1 开始索引的 double 数组具有不同的类型。如果在一个程序中使用这两个 double 数组，就会从模板中创建两个独立的类定义，每个类定义都包含要使用的成员函数。这样至少会导致两个我们不希望看到的后果：首先，程序中编译的代码比预期的要复杂得多(这种情形常常称为“代码膨胀”)；其次(也是比较糟糕的)，不能在一个表达式中混合使用两种类型的元素。如果给构造函数添加一个参数，为索引值指定比较灵活的范围，就要比使用非类型模板参数好得多。例如：

```
template <typename T> class Array {
  private:
    T* elements;                                  // Array of type T
    size_t size;                                  // Number of array elements
    long start;                                   // Starting index value

public:
  explicit Array(size_t arraySize, long startIndex = 0); // Constructor
  Array(const Array& theArray);                   // Copy Constructor
  ~Array();                                       // Destructor
  T& operator[](long index);                      // Subscript operator
  const T& operator[](long index) const;          // Subscript operator for const
  Array& operator=(const Array& rhs);             // Assignment operator
};.
```

额外的成员 start 主要用于存储构造函数中第二个参数指定的数组的起始索引值。startIndex 参数的默认值是 0，所以在默认情况下得到的是一般的索引方式。

在理解了有一个非模板参数的情况下如何定义成员函数后，下面完成 Array 类模板所需要的其他函数模板集。

1. 成员函数的模板

由于前面在类模板定义中添加了一个非类型参数，因此，构造函数的函数模板的代码和其他成员函数的模板代码也需要修改。构造函数的模板如下所示：

```
template <typename T, long startIndex>
Array<T, startIndex>::Array(size_t arraySize) : size(arraySize) {
  elements   =  new T[size];
}
```

模板 ID 现在是 Array<T, startIndex>，用于限定构造函数名。除了在模板中添加新的模板参数之外，这是惟一需要修改的地方。

对于副本构造函数，需要对函数模板作类似的修改：

```
template <typename T, long startIndex>
Array<T, startIndex>::Array(const Array& theArray) {
  size = theArray.size;
  elements = new T[size];
  for(int i = 0 ; i < size ; i++)
    elements[i] = theArray.elements[i];
}
```

当然，数组的外部索引不影响其内部管理方式。

析构函数也只需添加额外的模板参数：

```
template <typename T, long startIndex>
Array<T, startIndex>::~Array() {
  delete[] elements;
}
```

需要给非 const 的下标运算符函数修改模板定义：

```
template <typename T, long startIndex>
T& Array<T, startIndex>::operator[](long index) {
  if(index < startIndex || index > startIndex + static_cast<long>(size) - 1)
    throw out_of_range(
                    index < startIndex ?"Index too small" : "Index too large");

  return elements[index -  startIndex];
}
```

这里进行了重大的修改。Index 参数现在是 long 类型，允许使用负值。对 index 值的有效性检查现在验证该值是否在非类型模板参数和数组的元素个数确定的界限之内。索引值只能从 startIndex 到 startIndex+size-1。因为 size_t 通常是不带符号的整数类型，需要显式将它强制转换为 long，否则其他值就会自动转换为 size_t，如果该值为负，就会产生错误的结果。异常的消息和选择它的表达式也改变了。

以类似的方式修改 const 版本：

```
template  <typename T, long startIndex>
const T& Array<T, startIndex>::operator[](long index)  const {
  if(index < startIndex || index > startIndex + static_cast<long>(size) -1)
```

```
    throw out_of_range(
                    index < startIndex ?"Index too small" : "Index too large");

  return elements[index - startIndex];
}
```

最后，修改赋值运算符的函数模板，但只需要修改模板参数列表和限定运算符名的模板 ID：

```
template <typename T, long startIndex>
Array<T, startIndex>& Array<T, startIndex>::operator=(const Array& rhs) {
  if(&rhs == this)                          // If lhs == rhs
    return *this;                           // just return lhs

  if(elements)                              // If lhs array exists
    delete[]elements;                       // then release the free store memory

  size = rhs.size;
  elements = new T[rhs.size] ;
  for(int i = 0 ; i < size ; i++)
    elements[i] = rhs.elements[i];
}
```

在模板中使用非类型参数有一些限制。特别是，不能在模板定义中修改参数的值。因此，非类型参数不能放在等号的左边，也不能对它应用递增或递减运算符，换言之，它会被当做一个常量。

第 9 章介绍了传送给函数模板的模板参数可以从函数参数中推断出来，但这不适用于类模板。类模板中的所有参数都必须指定，除非参数有默认值，本章后面会讨论类模板中的参数默认值。

2. 非类型参数的变元

如果非类型参数不是引用或指针，则其参数就必须是在编译期间编译的常量表达式。也就是说，不能把包含非常量整型变量的表达式用作参数，这有一点缺陷，但编译器可以验证参数的有效性，从而弥补了这个缺陷。例如，下面的语句就不会编译：

```
long start = -10;
Array<double, start> values(21);      //Won't compile
```

编译器会因为第二个参数无效而生成一个消息。这两个语句的正确版本如下：

```
const long start = -10;
Array<double, start> values(21);
```

start 现在声明为 const，编译器就可以依赖其值，两个模板参数也是合法的。

如果参数需要与参数类型相匹配，编译器还提供了参数的标准转换。例如，如果非类型参数声明为 const size_t 类型，编译器就可以把整数字面量如 10 转换为需要的参数类型。

3. 把指针和数组用作非类型参数

如果非类型参数是一个指针，其参数就必须是一个地址，但不能是任何旧地址。它必须是

对象的地址或带有外部链接的函数地址，例如，不能把数组元素的地址或非静态类成员的地址用作参数。这也说明，如果非类型参数的类型是 const char*，在实例化模板时，就不能把字符串字面量用作参数。在这种情况下，如果一定要把字符串字面量用作参数，就必须用字符串字面量的地址初始化一个指针变量，再把该指针传送为模板参数。

指针是一个合法的非类型模板参数，可以把数组指定为参数，但在给模板提供参数时，数组和指针不能互换。例如，一个模板定义如下：

```
template <long* pNumber>class MyClass {
  //Template definition…
};
```

下面的代码可以创建这个模板的实例：

```
long data[10];                          //Global
long* pData=data;                       //Global

MyClass<pData> values;
MyClass<data> values;
```

数组名和合适类型的指针都可以用作类型为指针的参数的对应变元。但是，反过来就不行。假定定义了如下模板：

```
template <long number[10]>class AnotherClass {
  //Template definition…
};
```

这个模板的参数是一个包含 10 个元素的数组，参数的类型必须与它相同。在这种情况下，使用上面声明的数组 data，可以编写下面的代码：

```
AnotherClass<data> numbers;          //OK
```

但是，不能使用指针，下面的语句就不会编译：

```
AnotherClass<pData> numbers;         //Not allowed!
```

了解了 Array 模板的这些缺点后，下面用一个例子来演示非类型参数。

程序示例 18.2——使用非类型参数

把前面所作的修改插入到包含 Array 模板定义的头文件中，然后用下面的例子试验新特性：

```
// Program 18.2 Using non-type parameters in a class template   File: prog18_02.cpp
#include "Box.h"
#include " Array.h"
#include <iostream>
#include <iomanip>
using std::cout;
using std::endl;

int main() {
  try {
    const int size = 21;                          // Number of array elements
```

```
const int start = -10;                    // Index for first element
const int end = start+size-1;             // Index for last element

Array<double, start> values (size) ; // Declare array for double values

for(int i = start;  i<= end ; i++)    // Initialize the elements
  values [i] = i - start + 1;

cout << endl << "Sums of pairs of elements: ";
int lines = 0;
  for( int i = end ; i >= start ; i--)
    cout << (lines++ % 5 == 0 ? "\n" : "")
         << std::setw(5) << values[i] + values[i-1] ;
}
catch(const std::out_of_range& ex) {
  cout << endl << "out_of_range exception object caught!" << ex.what();
}
catch(const std::exception& ex) {
  cout << endl << ex.what();
}

try {
  const int start = 0;
  const int size = 11;

  Array<Box, start - 5> boxes(size);

  for(int i = start - 5 ; i <= start + size - 5 ; i++)
     cout << endl << "Box volume is " << boxes[i].volume() ;
  }
  catch (const std::exception& ex) {
    cout << endl << typeid(ex).name() << " exception caught! "<< ex.what();
  }

  cout << endl;
  return 0;
}
```

这个例子的结果如下所示：

```
Sums of pairs of elements:
   41   39   37   35   33
   31   29   27   25   23
   21   19   17   15   13
   11    9    7    5    3
out_of_range exception object caught ! Index too small
Box volume is 1
Box volume is 1
Box volume is 1
Box volume is 1
Box volume is 1
Box volume is 1
Box volume is 1
```

```
Box volume is 1
Box volume is 1
Box volume is 1
Box volume is 1
Box volume is 1
Box volume is 1
class std::out_of_range exception caught! Index too large
```

例子的说明

在第一个 try 块中，首先定义了一些常量，指定索引值的范围和数组的大小：

```
const int size=21;                          //Number of array elements
const int start=-10;                        //Index for first element
const int end= start + size -1;             //Index for last element
```

接着创建模板的一个实例，存储 21 个 double 类型的值：

```
Array<double, start> values(size);      // Declare array for double values
```

第二个参数对应于模板的非类型参数，指定了数组索引值的下限。数组的大小指定为构造函数的参数。

在 for 循环中为 values 对象的元素赋值：

```
for(int i=start; i<=end; i++)               //Initialize the elements
  values[i]=i-start+1;
```

索引值 i 的范围是从下限 start(其值为 - 10)开始，到上限 end(其值为+10)。在循环中，为数组元素定义了初始值，它们的值就是从 1 到 21。

初始化数组后，输出连续两个元素的和，从数组中的最后一个元素开始递增：

```
int lines=0;
for(i=end;i>=start; i--)
  cout << (lines++ % 5 == 0 ? "\n" : "")
       << std::setw(5) << values[i]+values[i-1];
```

lines 变量仅用于在一行中输出 5 个数。在前面的例子中，对索引值的过度控制导致表达式 values[i - 1]抛出 out_of_range 异常。try 块的第一个处理程序会捕获它：

```
catch(const std::out_of_range& ex) {
  cout<<endl<<" out_of_range exception object caught!"<<ex.what();
}
```

这段代码会显示程序输出中的消息。try 块还有第二个处理程序：

```
catch(const std::exception& ex) {
  cout<<endl<<ex.what();
}
```

这个处理程序会捕获 exception 类型的所有异常，实际上是把 exception 作为基类的所有异常，所以它会捕获所有的标准异常。如果 Array<double>构造函数抛出了 bad_alloc 异常，该处理程序就会捕获它。在这里，参数必须是一个引用，否则派生类异常就会转换为基类，出现第 17 章提到的对象切片问题。

因为 out_of_range 异常也把 exception 类作为基类，所以也可以只用一个处理程序来捕获所有的异常。例如，对第三个 try 块就使用了这个处理程序：

```
catch(const std::exception& ex) {
  cout<<endl<<typeid(ex).name()<<"exception caught!"<<ex.what();
}
```

实际上，如第 17 章所述，可以对这三个 try 块都使用同一个处理程序，减少源代码，减小可执行模块的文件尺寸，且仍能获得所抛出异常的全部信息。希望这不久就会变成现实！

下一个 try 块会创建一个数组来存储 Box 对象：

```
Array<Box,start-5> boxes(size);
```

可以看出，在模板的实例化过程中，这个表达式可以用作非类型参数的变元值。这种表达式必须等于对应参数的类型，或者必须通过标准转换，把结果转换为对应的类型。如果该表达式包含>字符，就要特别小心。例如：

```
Array<Box, start> 5 ? start : 5> boxes;        // Will not compile!
```

第二个参数的表达式使用条件运算符的目的是，提供最小为 5 的值，但这个语句是不会编译的。表达式中的>与前面的左尖括号配对，关闭了参数列表。要使该语句有效，需要使用括号：

```
Array<Box,(start> 5 ? start : 5)> boxes;
```

注释：

涉及间接成员访问运算符(->)或按位右移运算符(>>)也需要进行相同的修改。

下一个 for 循环抛出了另一个异常，这与前一个例子类似，并用一个处理程序来捕获它。

程序示例 18.3——更好的解决方案

必须记住，类模板中的非类型参数是对应于模板实例的类型的一部分。模板参数的每次独特组合都会生成另一个类类型。如前所述，在 Array<T>模板中，其效率非常低，模板的使用受到了限制。例如，如果数组的起始索引不同，就不能把一个 double 数组赋予另一个 double 数组，因为这两个数组的类型不同。而如果类模板带有额外的数据成员和额外的构造函数参数，效率就会提高很多。下面对前面的模板进行修改，并突出显示了与原模板类不同的代码：

```
template <typename T> class Array {
  private:
    T* elements;        // Array of type T
    size_t size;        // Number of elements in the array
    long start;         // Starting index value

  public:
    explicit Array(size_t arraySize, long startIndex = 0); // Constructor
    Array(const Array& theArray);                 // Copy Constructor
    ~Array();                                     // Destructor
    T& operator[](long index);                    // Subscript operator
    const T& operator[](long index) const; // Subscript operator for const
    Array& operator=(const Array& rhs);    // Assignment operator
};
```

构造函数作轻微的修改，以初始化新的数据成员：

```
template <typename T>
Array<T>::Array(size_t arraySize, long startIndex) :
                                        size(arraySize),start(startIndex) {
  elements = new T[size];
}
```

副本构造函数也要考虑到额外的数据成员：

```
template <typename T>
Array<T>::Array(const Array& theArray) {
  size = theArray.size;
  start=theArray.start;
  elements = new T[size];
  for(int i = 0 ; i < size ; i++)
    elements[i] = theArray.elements[i];
}
```

赋值运算符也是一样：

```
template <typename T>
Array<T>& Array<T>::operator=(const Array& rhs) {
  if(&rhs == this)              // If lhs == rhs
    return *this;               // just return lhs

  if(elements)                  // If lhs array exists
    delete [] elements ;        // then release the free store memory

  size = rhs.size;
  start = rhs.start;
  elements = new T[rhs . size] ;
  for(int i = 0 ; i < size ; i++)
    elements[i] = rhs.elements[i];
}
```

下标运算符函数也需要进行类似的修改，下面是非 const 版本：

```
template <typename T>
T& Array<T>::operator[](long index) {
  if (index < start || index > static_cast<long>(size) + start - 1)
    throw std::out_of_range(
                          index < start ? "Index too small" : "Index too large");

  return elements[index - start];
}
```

用下面的例子试验一下：

```
//  Program 18.3 A better Array class template  File: prog18_03.cpp
#include "Box.h"
#include "Array.h"
#include <iostream>
```

```
#include <iomanip>
using std::cout;
using std::endl;

int main() {
  try {
    const int size = 21;                                  // Number of array elements
    const int startValues = -10;                          // Index for first element
    const int endValues = startValues + size - 1;  // Index for last element

    Array<double> values(size, startValues);       // values[-10] to values[10]

    for(int i = startValues; i <= endValues; i + +) // Initialize the elements
      values [i]=i - startValues +1;
    const int startData = startValues+5;                  // Index for first element
    const int endData = endValues +5;                     // Index for last element

    Array<double> data(size, startData);   // Data[-5] to Data[15]

    // Initialize  the array
    for(int j = startData, i = startValues ; i <= endValues ; i++, j++)
      data[j] = values[i];

    cout << endl << "Sums of pairs of elements: ";
    int lines = 0;
    for(int i = endData ; i >= startData ; i--)
      cout << (lines++ % 5== 0 ? "\n" : "") << std::setw(5) << data[i] + data[i - 1];
  }
  catch (const std::exception& ex) {
    cout << endl << typeid(ex).name() << " exception caught! "<< ex.what();
  }

  cout << endl;
  return 0;
}
```

例子的运行结果如下：

```
Sums of pairs of elements:
   41   39   37   35   33
   31   29   27   25   23
   21   19   17   15   13
   11    9    7    5    3
class std:: out_of _range exception caught! Index too small
```

例子的说明

因为这里是通过构造函数的参数来设置 start 索引的，而不是通过类模板参数设置的，所以只要索引存储相同类型的值，就可以使用索引范围不同的数组。因为构造函数把数组的起始索引设置为默认值 0，所以，在使用从 0 开始索引的数组时，该模板的使用方式与源模板相同。

main()中的代码创建了对象 values 和对象 data，values 的索引从 - 10 到+10，data 的索引从 - 15 到+10。这两个对象都存储 double 类型的值。用 values 对象的元素初始化 data 对象的元素，

说明可以在表达式中混用这两个数组。而如果类模板使用非类型参数生成了不同类型的对象，这些对象就不能在表达式中混用。

在类模板中使用非类型参数时要三思而后行，以确保这些参数是必须的。此时，常常有另一种方法，能提供更灵活的模板和更高效的代码。

18.2.7 默认的模板参数值

可以为类模板中的类型参数和非类型参数提供默认值。如果某个类模板参数有默认值，该列表中的所有后续参数也都必须指定默认值。如果省略了类模板参数的值，而该类模板参数有默认值，就使用该默认值，就像函数中的默认参数值一样。同样，如果省略了列表中给定参数的值，则所有后续的参数也必须都省略。

给类模板参数指定默认值的方式和函数参数一样，也是在参数名的后面加上一个等号=。下面为带有非类型参数的 Array 模板提供参数默认值。例如

```
template <typename T=int, long startIndex=0>class Array {
  //Template definition as before…
};
```

当然，成员函数的模板也必须使用相同的默认值。这样，声明一个从 0 开始的 int 数组时，就可以省略所有的模板参数：

```
Array<> numbers(101);
```

合法的索引值是从 0 开始到 100，这是由非类型模板参数的默认值和构造函数的参数确定的。在这种情况下，即使不需要任何参数，也必须写上尖括号。另一个可能的情形是省略第二个参数，或提供所有的参数。例如：

```
Array<string, -100> messages(200);          //Array of 200 string objects
Array<Box> boxes(101);                      //Array of 101 Box objects
```

注意：

不能只省略第一个参数。一般情况下，第一个参数右边的所有模板参数也必须省略。

如果类模板的所有参数都有默认值，在源文件中，就只在模板的第一个声明中指定它们。(当然，它们也可以是模板的定义)。

18.3 模板的显式实例化

前面都是通过声明模板类型的变量，来隐式创建类模板的实例。也可以显式实例化类模板和函数模板。显式实例化模板时，编译器会用指定的参数值来创建实例。

第 9 章介绍了如何显式实例化函数模板。要实例化类模板，只需在关键字 template 的后面加上模板类的名称和要使用的模板参数。下面的声明就显式创建了 Array 模板的一个实例：

```
template class Array<double,1>;
```

这个语句创建了模板的一个实例，它可以存储 double 类型的值，且从 1 开始索引。显式实例化类模板，会生成类类型定义，并从模板中实例化类的所有成员函数。

18.4 类模板的友元

由于类可以有友元，因此类模板也可以有友元，其友元可以是类、函数或其他模板。如果类是模板的一个友元，其所有成员函数就是该模板每个实例的友元。如果函数是模板的友元，该函数就是模板的每个实例的友元。如图 18-4 所示。

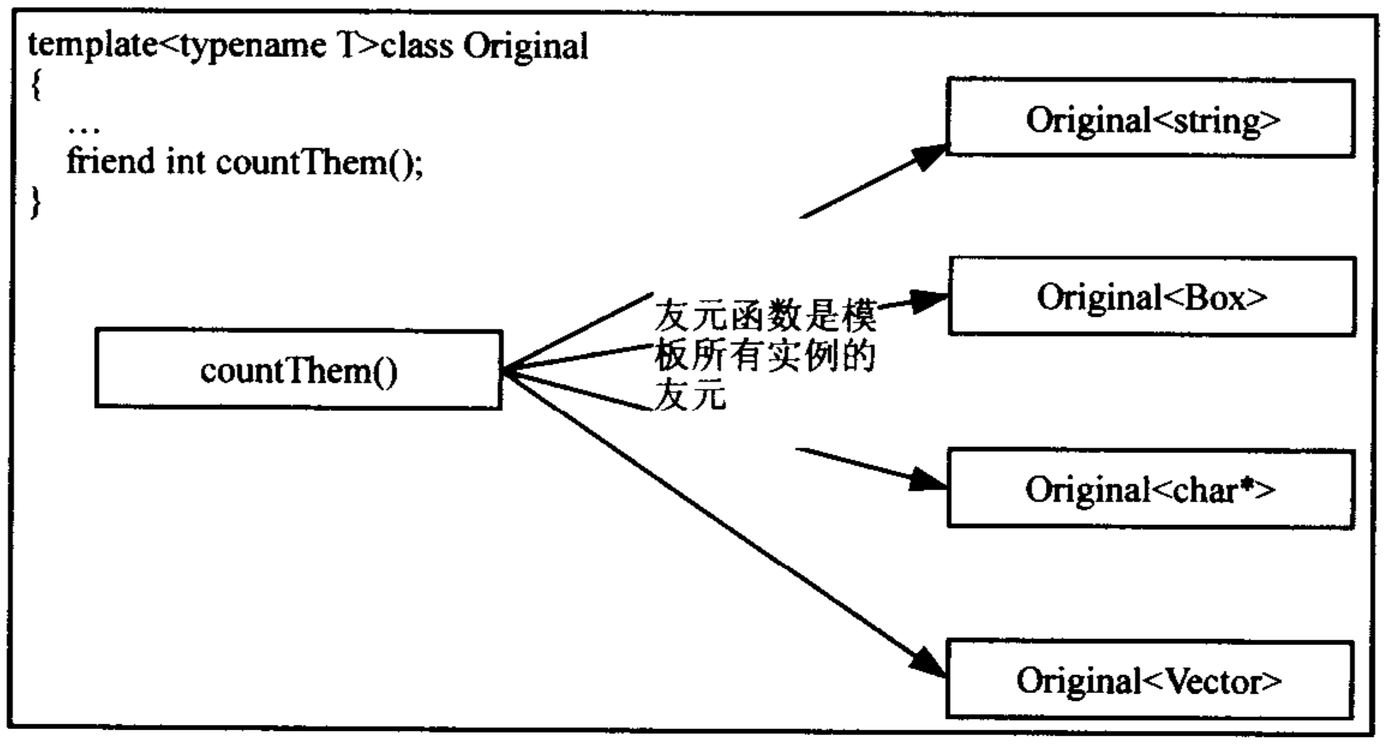


图 18-4 类模板的友元函数

如果一个模板是另一个模板的友元，情况则有些不同。由于模板有参数，模板类的参数列表通常包含定义友元模板的所有参数。因此有必要标识友元模板的实例，该友元模板又是原类模板的某个实例的友元。但是，只有在代码中使用友元函数时，才实例化友元的函数模板。在图 18-5 中，getBest()是一个函数模板。

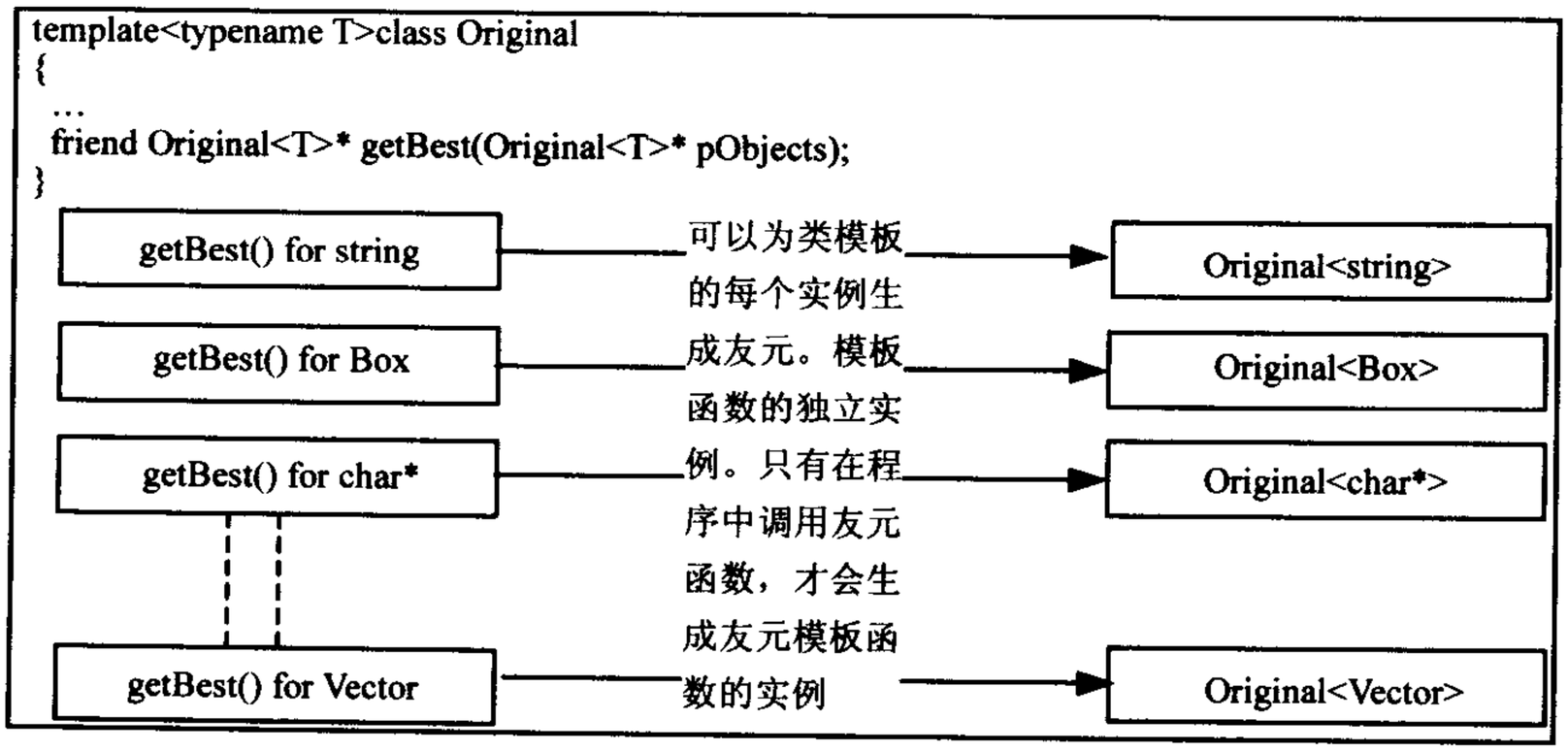


图 18-5 函数模板是类模板的一个友元

在上面的例子中，每个类模板实例都有一个惟一的友元模板实例，但这是不必要的。如果类模板的一些参数在友元模板中没有，则友元模板的一个实例就会用于类模板的几个实例。

注意，普通的类可以把类模板或函数模板声明为它的一个友元。此时，模板的所有实例都

是这个类的友元。在图 18-6 的例子中，Thing 模板的每个实例中的每个成员函数都是类 Box 的友元，因为该模板声明为 Box 类的友元。

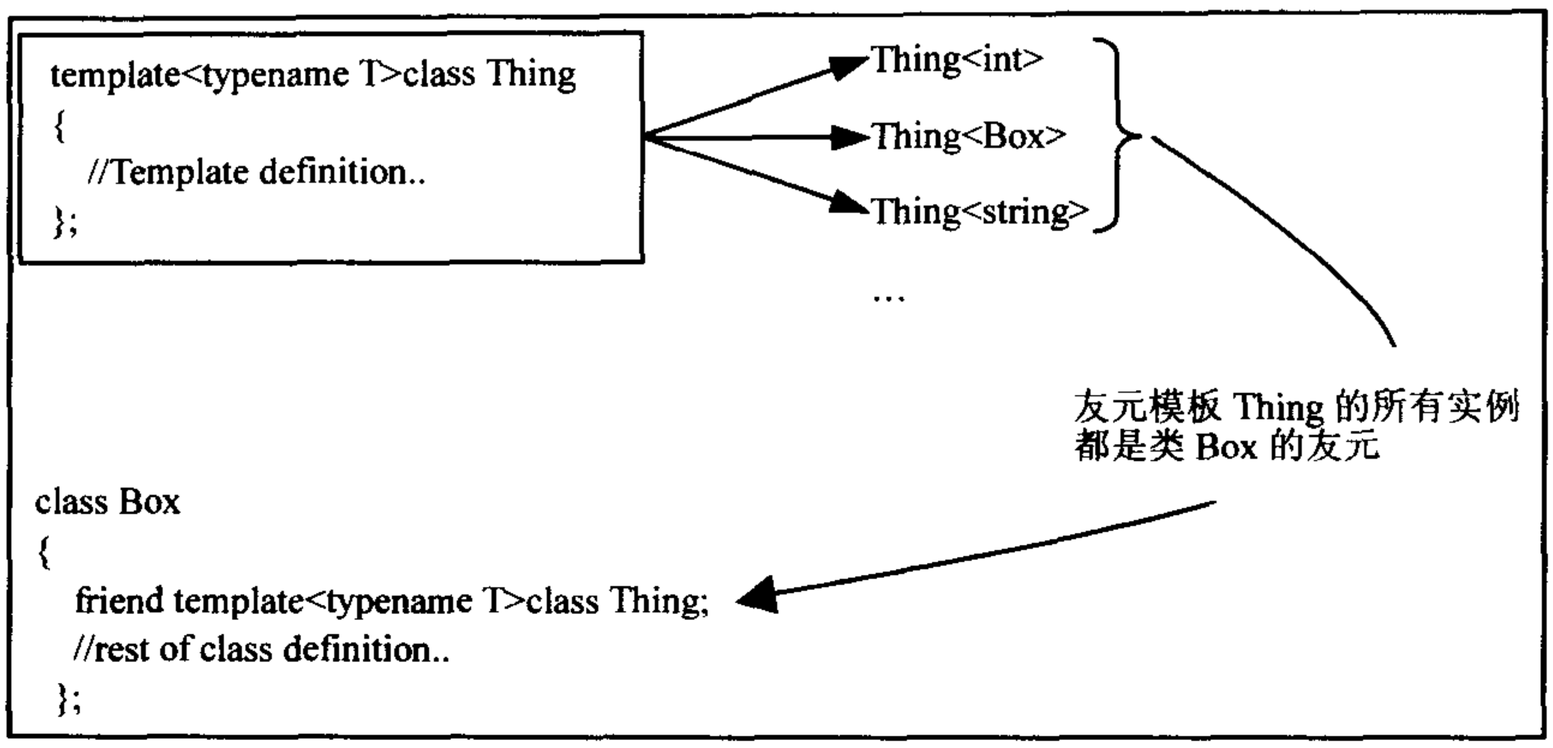


图 18-6　类模板是类的一个友元

18.5　特殊情形

在许多情况下，类模板定义不能适用于所有的参数类型。例如，可以使用重载的比较运算符比较 string 对象，但不能对非空字符串进行这类比较。如果模板使用比较运算符比较对象，则该比较运算符可用于比较 string 类型，但不能比较 char*类型。要比较 char*类型的对象，需要使用在<cstring>头文件中声明的比较函数。

为了处理这类问题，可以定义类模板说明。类模板说明提供的类定义专门针对模板参数的给定参数集。注意，这是一个类定义，而不是类模板。编译器使用类模板说明而不是使用模板为特定类型，如 char*，生成类。类模板说明允许定义类的特定版本，让编译器使用该版本处理特定的类模板参数值。

假定要为 char*类型创建 Array 模板的第一个版本说明。再假定它包含的成员函数可以比较模板类型的对象，这些成员函数会逐个元素地比较两个对象的 elements 成员，但其细节是不重要的。类模板定义如下所示：

```
template <> class Array<char*> {
  //Definition of a class to suit type char*…
};
```

为 char*类型的 Array 模板所定义的说明，必须放在原模板定义的前面，或放在原模板的声明前面。

因为在说明中指定了所有的参数，所以称为模板的完整说明，这也是第一组尖括号为空的原因。由于指定了所有的参数，就不能再指定模板参数了。这没有任何灵活性——类型 char*已被指定为 Array 模板的参数，编译器会使用该说明来编译，而不是把参数应用于模板。

类模板中可能有一两个成员函数需要为特定类型编码。如果成员函数由独立的函数模板定义，而不是在类模板体中定义，就只能为该函数模板提供说明。

部分模板说明

如果特殊化带有两个参数的模板版本，就只能为说明指定类型参数，不能指定非类型参数。这是用 Array 模板的部分说明完成的，其定义如下所示：

```
template <long start> class Array<char*,start> {
  //Definition to suit type char*…
};
```

template 关键字后面的参数列表表示，需要为这个模板说明的实例指定参数，在本例中只需要指定一个参数。第一个参数被省略了，因为它现在是固定的。模板名后面的尖括号指定原模板定义中的参数如何特殊化。该参数列表必须与原来未特殊化的模板有相同的参数个数。这个说明的第一个参数是 char*。另一个参数指定为这个模板中的对应参数名，因此不能特殊化。

除了给类型参数使用 char*，给所生成的模板实例特别的考虑之外，指针一般也是特殊化的一个子集，需要采用与对象和引用不同的方式来处理。在使用指针类型实例化模板时，为了进行合适的比较，在比较变量之前需要解除对它们的引用，否则就只比较了地址，而没有比较存储在这些地址中的对象或值。

对于这种情况，可以定义模板的另一个部分说明。在这种情况下，第一个参数不完全固定，但它必须匹配在模板名后面的列表中指定的特定模式。例如，Array 模板给指针指定的部分说明如下所示：

```
template <typename T, long start>class Array<T*, start> {
  //Definition to suit pointer types other than char*…
};
```

第一个参数仍旧是 T，但模板名后面的尖括号中的 T*表示，这个定义用于将 T 指定为指针的实例。其他两个参数仍旧可变，所以这个说明可应用于第一个参数为指针的实例。

从多个部分说明中选择

假定给刚才讨论的 Array 模板创建了两个部分说明，一个用于类型 char*，另一个用于指针类型。在类型 char*的版本适合某个实例时，如何确保编译器选择这个版本？例如，下面的声明：

```
Array<Box*, -5> boxes(11);
```

显然，一般这只能使用指针类型的说明，但对于下面的语句，两个部分说明都适合：

```
Array<char*, 1> messages(100);
```

在这种情况下，编译器会认为 char*部分说明比较合适，因为它指定得比较清楚。char*类型的部分说明模板一般比指针类型的部分说明模板更清楚，是因为对于可以选择 char*说明的情形(仅仅是 char*这一种情形)，也可以选择 T*说明，但反过来则不行。当匹配给定说明的每个参数也匹配另一个说明时，给定的说明比另一个说明更特殊一些，但反过来则不行。因此可

以为模板指定一组说明，按照最特殊的说明到最一般的说明排序。当几个模板说明适合于给定的声明时，编译器会选择并应用其中最特殊的说明。

18.6 带有嵌套类的类模板

类模板定义可以包含嵌套的类或嵌套的类模板。嵌套的类模板进行独立的参数化，这样就可以生成二维的类。这种情形超出了本书的范围，这里仅探讨带有嵌套类的类模板。

下面举一个例子。假定要实现一个堆栈，它采用后进先出的存储机制。“推”操作会把数据项存储在堆栈的顶部，而“拉”操作会从堆栈中提取出顶部的数据项。该堆栈可以存储任意给定类型的对象，这就是模板的本质。如图 18-7 所示。

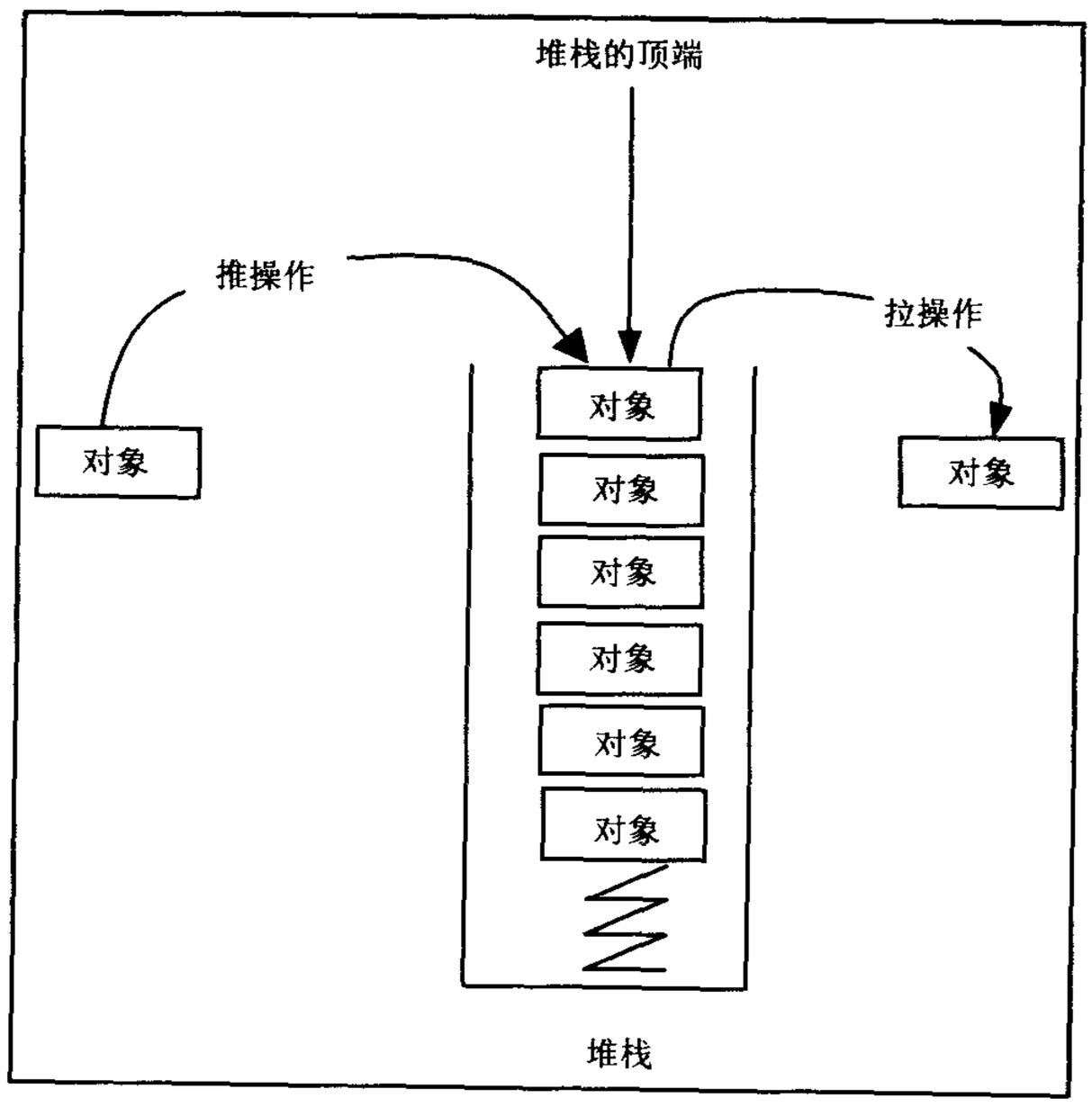


图 18-7 堆栈的概念

Stack 模板的模板参数是一个类型参数，它指定了堆栈中的对象类型，于是，最初的模板定义如下所示：

```
template <typename T> class Stack {
  //Detail of the Stack definition…
};
```

如果希望堆栈的容量自动增加，就不能为堆栈中的对象使用固定的存储空间。如果在对象进入或离开堆栈时，要自动增减堆栈的存储空间，一种方法是把堆栈实现为链表。这个链表中的节点可以在自由存储区中创建，堆栈只需要记住堆栈顶部的节点即可，如图 18-8 所示。

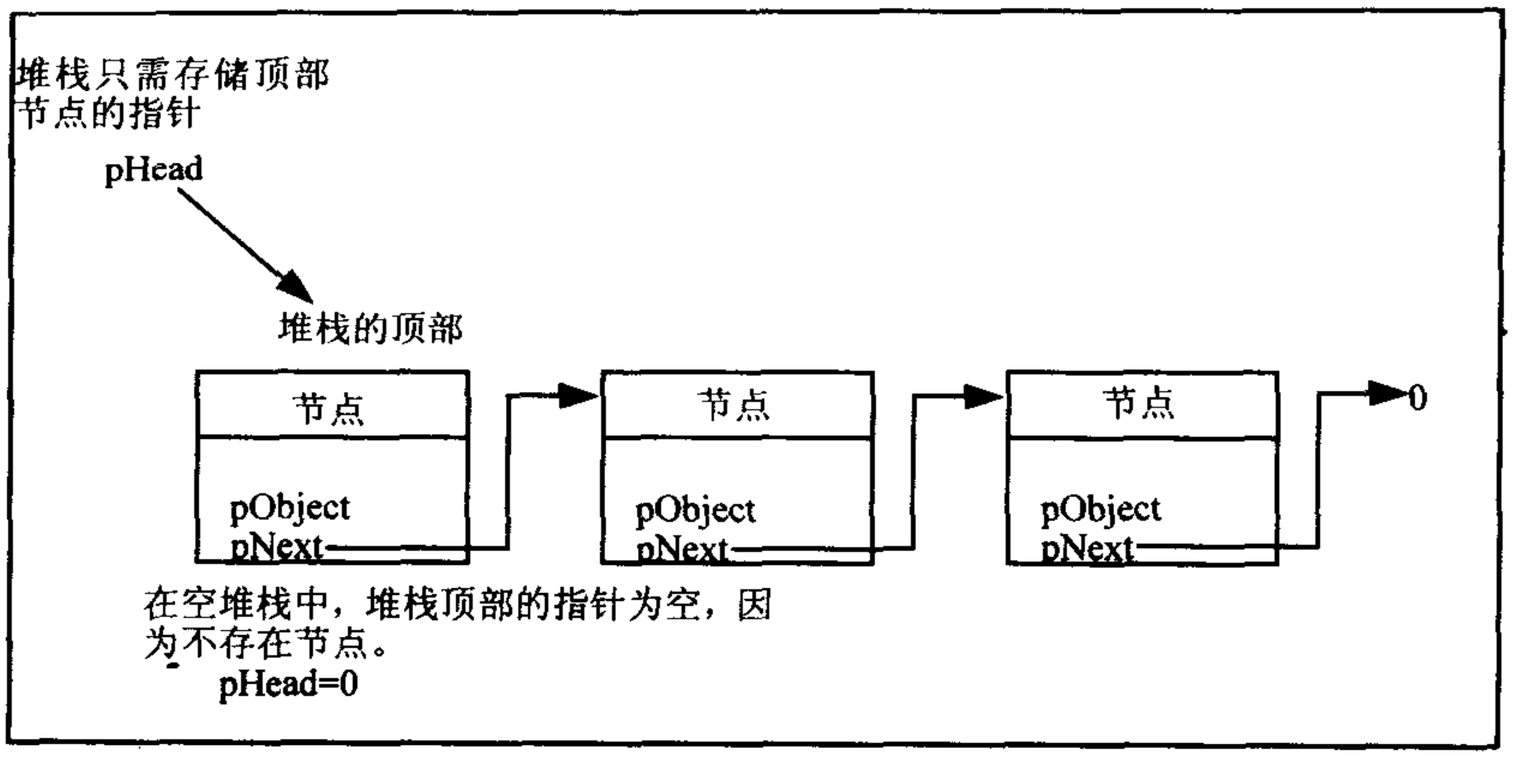


图 18-8　堆栈实现为链表

在创建空堆栈时，列表顶部的指针为空，于是，如果堆栈没有包含任何 Node 对象，就表示该堆栈为空。

在 Stack 模板的每个实例中，都需要一个嵌套类来定义列表中的节点，而且节点必须包含 T 类型的对象，其中 T 是 Stack 模板参数类型，所以可以把它定义为嵌套模板。在 Stack 模板的初始定义中添加这个嵌套模板：

```
template <typename T> class Stack {
  private:
    class Node {
      public:
        T* pItem;                                    // Pointer to object stored
        Node* pNext;                                 // Pointer to next node

        Node(T& rItem) : pItem(&rItem), pNext(0) {} // Create node from an object
        Node() : pItem(0), pNext(0) {}               // Create an empty node
    };

  // Rest of the Stack definition...
};
```

因为 Node 类声明为 private，所以可以把它所有的成员都声明为 public，可以由 Stack 模板的成员函数直接访问。假定 T 类型的对象由用户负责，则在 Node 对象中就只存储 T 类型对象的指针。再假定 Stack 的用户还要全权负责释放已存储的对象。不带参数的 Node 构造函数只把数据成员设置为空，用于在空堆栈中创建 Node。另一个构造函数在对象推入堆栈时使用。该构造函数的参数是 T 类型对象的一个引用。

现在编写 Stack 类模板的其他内容，以支持图 18-8 中 Node 对象的链表：

```
template <typename T> class Stack {
  private:
    class Node {
      public:
        T* pItem;                                    // Pointer to object stored
        Node* pNext;                                 // Pointer to next node

        Node(T& rItem) : pItem(&rItem), pNext(0) {} // Create node from an object
```

```
    Node() : pItem(0), pNext(0) {}   // Create an empty node
  };

  Node* pHead;      // Points to the top of the stack

public:
  Stack():pHead(0){}                          // Default constructor
  Stack(const Stack& aStack);                 // Copy constructor
  ~Stack();                                   // Destructor
  Stack& operator=(const Stack& aStack);   // Assignment operator

    void push(T& rItem);                      // Push an object onto the stack
    T& pop();                                 // Pop an object off the stack
    bool isEmpty() {return pHead == 0;} // Empty test
};
```

如前所述，堆栈只需“记住”顶部的节点，因此只有一个 Node 类型的数据成员 pHead。Stack 类模板有一个默认的构造函数、一个副本构造函数、一个析构函数和赋值运算符(因为节点是动态创建的)。它还有 push()和 pop()成员，用于在堆栈中来回传送对象，最后，Stack 类模板还有一个 isEmpty()函数，如果堆栈是空，则返回 true。

为了实现堆栈，还需要 Stack 模板的成员函数模板。

定义成员函数的模板

默认的构造函数在模板中定义，它只是把 pHead 初始化为 0。副本构造函数必须复制 Stack<T>对象，方法是遍历节点，并复制它们：

```
template <typename T> Stack<T>::Stack(const Stack& aStack) {
  pHead = 0;
  if(aStack.pHead) {
    pHead = new Node(*aStack.pHead);     // Copy the top node of the original
    Node* pOldNode = aStack.pHead;       // Points to the top node of the original
    Node* pNewNode = pHead;              // Points to the node in the new stack

   while(pOldNode = pOldNode->pNext ) {  // If it is null, it is the last node
     pNewNode->pNext = new Node(*pOldNode);      // Duplicate it
     pNewNode = pNewNode->pNext ;        // Move to the node just created
    }
  }
}
```

赋值运算符类似于副本构造函数，但要完成另外两件事：首先，必须检查所涉及到的对象是否惟一；其次，必须释放等号左边对象中的节点所占用的内存。下面是定义赋值运算符的模板：

```
template <typename T> Stack<T>& Stack<T>::operator = (const Stack& aStack) {
  if(this  == &aStack)          // If objects are  identical
    return *this;               // return the left object

  // Release memory for nodes in the left object
  Node* pTemp;
```

```
while(pHead) {                  // While current pointer is not null
  pTemp = pHead->pNext;         // Get the pointer to the next
  delete pHead;                 // Delete the current
  pHead = pTemp;                // Make the next current
}

if(aStack.pHead) {
 pHead = new Node(*aStack.pHead) ;   // Copy the top node of the original
 Node* pOldNode = aStack.pHead;      // Points to the top node of the original
 Node* pNewNode = pHead; // Points to the node in the new stack

  while(pOldNode = pOldNode->pNext) {  // If it is null, it is the last node
    pNewNode->pNext = new Node(*pOldNode );    // Duplicate it
    pNewNode = pNewNode->pNext;      // Move to the node just created
   }
 }
 return *this                                  // Return the left object
}
```

如果赋值语句中的对象相同，就解除 this 指针的引用，获取左边的对象，并返回它。如果对象不相同，第一步是删除左边对象中的所有节点，再用右边对象的节点副本替换它们。之后，用与副本构造函数相同的代码复制右边的对象。在副本构造函数和赋值运算符中，执行复制的代码是相同的，所以把它们放在一个单独的成员函数中。

在析构函数中，删除节点的代码与赋值运算符函数中的代码相同：

```
template <typename T> Stack<T>::~Stack() {
  Node* pTemp;
  while(pHead) {
    pTemp = pHead->pNext;
    delete pHead;
    pHead = pTemp;
  }
}
```

注释：

与复制代码相同，这些代码也可以放在单独的帮助函数中，该函数是 Stack 类模板的私有成员，再在需要时调用它们。这会减小可执行文件的尺寸。

push()操作的模板非常简单：

```
template <typename T> void Stack<T>::push(T& rItem) {
  Node* pNode = new Node(rItem);     // Create the new node
  pNode->pNext = pHead;              // Point to the old top node
  pHead = pNode;                     // Make the new node the top
}
```

要创建节点，把对象的引用传送给 Node 构造函数。这个节点的 pNext 成员需要指向顶部节点。然后使新的节点成为堆栈顶部节点。

pop()操作的工作略多，因为必须删除顶部节点：

```
template <typename T> T& Stack<T>::pop() {
  T* pItem= pHead->pItem;                    // Get pointer to the top node object
  if(!pltem)                                 // If it is empty
    throw std::logic_error("Stack empty"); // Pop is not valid so throw exception

  Node* pTemp = pHead;                       // Save address of top node
  pHead = pHead->pNext;                      // Make next node the top
  delete pTemp;                              // Delete the previous top node
  return *pItem;                             // Return the top object
}
```

有人可能尝试在空的堆栈中执行拉操作。由于返回的是一个引用，不能通过返回值警示错误，所以必须抛出一个异常。

在顶部节点中检索出对象的指针后，就删除顶部节点，并让下一个节点指向顶部，返回该对象。完成了定义堆栈所需要的模板后，下面在一个例子中试验嵌套的模板。

程序示例 18.4——使用嵌套的类模板

把所有的模板都放在一个头文件 Stack.h 中，该头文件还包含了前面提到的帮助函数，如下所示：

```
// Stack.h Templates to define stacks
#ifndef STACKS_H
#define STACKS_H
#include <stdexcept>

template <typename T> class Stack {
 private:
   class Node {
     public:
       T* pItem;          // Pointer to object stored
       Node* pNext;       // Pointer to next node

       Node(T& rItem) : pItem(&rItem), pNext(0) {}  // Create node from an object
       Node() : pItem(0), pNext(0) {}       // Create an empty node
   };

   Node* pHead;           // Points to the top of the stack
   void copy(const Stack& aStack);         // Helper to copy a stack
   void freeMemory();  // Helper to release free store memory

 public:
   Stack():pHead(0){}  // Default constructor
   Stack(const Stack& aStack);             // Copy constructor
   ~Stack();           // Destructor
   Stack& operator=(const Stack& aStack); // Assignment operator

   void push(T& rItem);// Push an object onto the stack
   T& pop();           // Pop an object off the stack
   bool isEmpty() {return pHead == 0;}    // Empty test
};
```

```
// Copy constructor
template <typename T> Stack<T>::Stack(const Stack& aStack) {
  copy(aStack);
}

// Helper to copy a stack
template <typename T> void Stack<T>::copy(const Stack& aStack) {
  pHead = 0;
    if(aStack.pHead) {
      pHead = new Node(*aStack.pHead);  // Copy the top node of the original
      Node* pOldNode = aStack.pHead;      // Points to the top node of the original
      Node* pNewNode = pHead;  // Points to the node in the new stack

      while(pOldNode=pOldNode->pNext)   // If it is null, it is the last node
        pNewNode->pNext = new Node(*pOldNode);  // Duplicate it
        PNewNode = pNewNode->pNext;     // Move to the node just created
      }
    }
}

// Assignment operator
template <typename T> Stack<T>& Stack<T>::operator=(const Stack& aStack) {
  if(this == &aStack)          // If objects are identical
    return *this;              // return the left object

  freeMemory();                // Release memory for nodes in lhs
  copy(aStack);                // Copy rhs to lhs

  return *this                 // Return the left object
}

// Helper to release memory for a stack
template <typename T> void Stack<T>::freeMemory () {
  Node* pTemp;
  while (pHead) {                  // While current pointer is not null
    pTemp = pHead->pNext;          // Get the pointer to the next
    delete pHead;                  // Delete the current
    pHead =pTemp;                  // Make the next current
  }
}

// Destructor
template <typename T> Stack<T>::~Stack() {
  freeMemory();
}

// Push an object onto the stack
template <typename T> void Stack<T>::push(T& rItem) {
  Node* pNode = new Node(rItem);    // Create the new node
  pNode->pNext = pHead;             // Point to the old top node
  pHead = pNode;                    // Make the new node the top
}
```

```
// Pop an object off the stack
template <typename T> T& Stack<T>::pop() {
T* pItem = pHead->pItem;     // Get pointer to the top node object
if(!pItem)                                        // If it is empty
 throw std::logic_error("Stack empty");    // Pop is not valid:throw exception

 Node* pTemp = pHead;                          // Save address of top node
 pHead = pHead->pNext;                         // Make next node the top
 delete pTemp;                                 // Delete the previous top node
 return *pItem;                                // Return the top object
}
#endif
```

接着在下面使用堆栈的例子中使用这些模板：

```
// Program 18.4 Using a stack defined by nested class templates   File: prog18_04.cpp
#include " Stack.h"
#include <iostream>
#include <string>
using std::cout;
using std::cin;
using std::endl;
using std::string;

int main() {
const char* words[]={"The", "quick", "brown", "fox", "jumps"};
Stack<const char*> wordStack;    //A stack of null terminated strings

for(size_t i= 0 ; i < 5 ; i++)
  wordStack.push(words[i]);

Stack<const char*> newStack(wordStack); // Create a copy of the stack

// Display the words in reverse order
while(!newStack.isEmpty())
  cout << newStack.pop () << " ";
cout << endl;

// Reverse wordStack onto newStack
while(!wordStack.isEmpty())
  newStack.push(wordStack.pop());

// Display the words in original order
while(!newStack.isEmpty())
  cout << newStack.pop()<<" ";
cout <<endl;

cout << endl << "Enter a line of text:" << endl;
string text;
getline(cin, text);              // Read a line into the string object
```

```
Stack<const char> characters;   // A stack for characters

for(size_t i=0 ; i < text . length () ; i++)
  characters.push(text[i]);     // Push the string characters onto the stack

cout << endl;
while(!characters.isEmpty())
  cout << characters.pop();     // Pop the characters off the stack

cout <<endl;
return 0;
}
```

这个例子的结果如下所示：

```
jumps fox brown quick The
The quick brown fox jumps

Enter a line of text:
A nod is as good as a wink to a blind horse

esroh dnilb a ot kniw a sa doog sa si don A
```

例子的说明

首先定义一个数组，它包含 5 个对象，这些对象都是非空字符串，用一些单词初始化它们。接着声明一个堆栈对象，存储 const char*对象，如下面的语句所示：

```
Stack< const char*> wordStack;                    //A stack of null terminated strings
```

这个语句会创建 Stack 模板的一个实例，并为 Stack< const char*>创建一个构造函数的实例。在一个 for 循环中，把数组元素推入堆栈：

```
for(size_t i=0; i<5; i++)
  wordStack.push(words[i]);
```

在 wordStack 堆栈的底部存储了第一个单词，最后一个单词则存储在堆栈的顶部。接着创建该堆栈的一个副本，如下面的语句所示：

```
Stack< const char*> newStack(wordStack);                    //Create a copy of the stack
```

这个语句调用了副本构造函数，为此，需要创建函数模板的一个实例。newStack 是 wordStack 的一个副本。在下一个 while 循环中，把单词拉出堆栈，以逆序方式显示它们：

```
while(!newStack.isEmpty())
  cout<< newStack.pop()<<" ";
```

这个语句使用 isEmpty()继续从堆栈中拉出对象，只要堆栈不为空，这个函数就返回 false。使用 isEmpty()函数来获取堆栈中的所有内容是比较安全的。在循环的最后，newStack 为空，但 wordStack 仍包含最初的内容。

在下一个 while 循环中，从 wordStack 中提取单词，并把它们推入 newStack：

```
while(!wordStack.isEmpty())
```

```
    newStack.push(wordStack .pop());
```

拉出和推入操作组合到一个语句中，把 wordStack 的 pop()返回的对象用作 newStack()的 push()的参数。在这个循环的最后，wordStack 为空，而 newStack 包含 5 个单词，且顺序不变，即第一个单词在堆栈的顶部。接着把单词从 newStack 中拉出，并输出它们，在这个循环的最后，两个堆栈都为空：

```
while(!newStack.isEmpty())
  cout<< newStack.pop()<<" ";
```

程序的下一部分使用 getline()函数，把一行文本读入一个字符串对象：

```
cout<<endl<<endl<<"Enter a line of text:"<<endl;
string text;
getline(cin,text);              //Read a line into the string object
```

这个语句把输入放在 string 对象 text 中。接着创建一个堆栈，以存储字符：

```
Stack<const char> characters;              //A stack for characters
```

这个语句创建了 Stack 模板的一个新实例 Stack< const char>，以及这类堆栈的构造函数的新实例。此时，程序包含从 Stack 模板中创建出来的两个类，每个类包含一个嵌套的 Node 类。

在一个 for 循环中，从 text 中提取字符，把它们推入新堆栈：

```
for(size_t i=0; i<text.length(); i++)
  characters.push(text[i]);    //push the string characters onto the stack
```

text 对象的 length()函数用于确定循环何时结束，现在从堆栈中拉出字符，以逆序方式显示输入的字符串：

```
cout<<endl;
while(!characters.isEmpty())
  cout<< characters.pop();                  //Pop the characters off the stack
```

从运行结果中可以看出，输入的内容有点回文的意味，读者可以试试输入"Ned, I am a maiden"或者"Are we not drawn onward, we few, drawn onward to new era."。

18.7 更高级的类模板

类模板的更高级应用超出了本书的范围，全面讨论这个问题需要一整本书的篇幅，但这里仅提及其中的两个功能，而且不深入探讨。

类模板可以有基类，这些基类可以是一般的类，也可以是模板。例如，从 Stack 模板中派生一个新模板，提供 Stack 基模板中没有的功能。该模板的定义如下：

```
template <typename T> class SpecialStack: public Stack<T> {
  public:
    SpecialStack();
    ~SpecialStack();
    SpecialStack(const SpecialStack& aStack);
```

```
    int ObjectCount();              //Count the objects
};
```

这是一个小例子，仅添加了一个函数，确定堆栈中有多少个对象，但该例子说明了把模板指定为基模板是相当简单的。这个模板的实例派生于 Stack<T>实例，所以它的工作方式与普通的派生类一样。

还要注意，模板中的类型参数也可以是一个模板，下面定义一个模板：

```
template <typename T1, template <typename T2> Array> class ClassName {
  //Template definition…
};
```

第一个模板参数 T1 是一个类型参数，第二个参数也是一个类型参数，但这次它是一个模板。参数 T2 确定 Array 模板的一个特定实例，该实例用作 ClassName 模板的第二个参数。

18.8 本章小结

理解类模板如何定义和使用，是理解如何应用第 20 章论述的标准模板库功能的基础。它也是对定义类的基本语言特性的一个强大补充。本章的要点如下：

- 类模板定义了一系列类类型。
- 类模板的实例是根据给定的模板参数集，从该模板中生成的类定义。
- 声明类模板类型的对象，就会对类模板进行隐式实例化。
- 类模板的显式实例化，会根据类模板的给定参数集定义一个类。
- 对应于类模板中类型参数的参数，其类型可以是基本类型、类类型、指针或引用类型。
- 非类型参数可以是整型类型、枚举类型、指针或引用。
- 类模板的部分说明可以根据原类模板中参数的一组限定子集，定义一个新模板。
- 类模板的完全说明可以根据原类模板的所有参数参数，定义一个新模板。
- 类模板的友元可以是函数、类、函数模板或类模板。
- 普通的类可以把类模板或函数模板声明为友元。

18.9 练习

1. 第 17 章的练习题要求创建一个稀疏数组类。这里，则要求定义一个一维稀疏数组的模板，存储任意类型的对象，而且只有存储在数组中的元素才能占用内存。元素的个数可以由模板的一个实例存储，该个数是不应有限制的。该模板可以用于定义一个稀疏数组，该数组包含 double 类型的元素指针，语句如下：

```
SparseArray<double> values;
```

为模板定义下标运算符，以便像普通数组那样提取和设置元素值。如果在某个索引位置不存在元素，下标运算符就应返回由对象类的默认构造函数创建的对象。在 main()函数中使用这个模板，在一个稀疏数组的随机位置上存储 20 个 int 类型的随机元素值，随机数的范围是 32

到 212，索引值的范围是 0 到 499，输出非 0 的元素值及其索引位置。

2. 为链表定义一个模板，允许从列表的最后开始向前遍历列表，再从列表的开始向后遍历列表(每个节点都需要前一个节点的指针和下一个节点的指针)。在一个程序中使用该模板，把某个散文或诗歌中的单词存储为字符串对象，再以逆序方式显示它们，一行显示 5 个单词。

3. 使用链表和稀疏数组模板创建一个程序，在至多有 26 个链表的稀疏数组中，存储散文或诗歌中的单词，每个列表都包含首字母相同的单词。输出这些单词，在输出时，对这些单词分组，使每一组单词具有相同的首字母，并在一个新行上显示该组中的单词(在指定模板参数时，应在连续的>字符之间加上空格，否则>>会被解释为按位右移运算符)。

4. 在 SparseArray 模板中添加 insert()函数，该函数在数组的最后一个元素后面添加一个元素。使用这个函数和一个 SparseArray 实例完成上一题，其中，该实例的元素是存储了 string 对象的 SparseArray 对象。

第 19 章　输入输出操作

C++语言本身不提供输入输出操作。本章的主题是在标准库中可用的输入输出功能，这些功能支持程序中独立于设备的输入输出操作。前面已经在所有的例子中使用这些功能从键盘上读取数据，在屏幕上输出数据。本章将扩展这些功能，讨论如何读写磁盘文件。

本章主要内容

- 流的概念
- 标准流的概念
- 二进制流与文本流的区别
- 如何创建和使用文件流
- 流操作中的错误如何记录，如何管理它们
- 如何使用未格式化的流操作
- 如何把数据值作为二进制数据写到文件中
- 如何从流中读写对象
- 如何重载类的插入和提取运算符
- 如何实现模板类的流支持
- 如何创建字符串流

19.1　C++中的输入输出

在创建 C++程序时，需要许多不同类型的输入输出功能。应用程序可能需要在数据库中存储和提取数据，在屏幕上创建和显示图形，利用电话线通过调制解调器进行通信，通过网络进行通信等。所有这些例子都有一个共同点：它们都与 C++语言和库功能无关。

这说明，在大多数情况下，要使用的输入输出功能并不是 C++标准的一部分，但它们是 C++开发环境的一部分。有时 C++提供的一些功能与程序执行的环境并不一致。计算机的操作系统控制着与屏幕和键盘的通信，它并不能通过 cin 和 cout 进行读写。例如，在 PC 上为 Microsoft Windows 编程时，尽管在许多 Windows 的 C++开发系统上模拟了输入输出功能，可以在命令行上把读写当作控制台操作，但也不能通过 cin 和 cout 进行读写。

当然，在 C++中定义的功能仍旧非常重要，因为它们代表具有扩展功能的一个重要标准库。它们不仅提供了文件输入输出功能，还可以使用基于字符串的 I/O 进行数据格式化。

19.1.1　理解流

标准库提供的输入输出功能涉及到流的使用。流是程序中输入或输出设备的抽象表示，输入或输出设备是数据的来源或目的地。可以把流看作在外部设备和计算机内存之间流动的一系列字节。基本上，所有的输入和输出都是在某个外部设备上读取或写入的一系列字节。

从外部设备上读取数据时，应直接解释数据。从外部源上读取字节时，字节可以是8位字符系列、UCS字符系列、各种类型的二进制值或它们的混合。仅数据本身看不出其来源。必须事先知道数据的结构和类型，才能读取和解释它们。

1. 数据传输模式

在流中传入和传出数据有两种模式：文本模式和二进制模式。在文本模式中，数据解释为一系列字符，这些字符组织为一行或多行文本，并用换行符'\n'断开。在文本模式中，当字符在物理设备上读写时，流可以传送换行符。流是否传送字符，以及字符如何修改都与系统有关。例如在一些系统如Microsoft Windows上，写到流中的一个新行字符由两个字符替代：回车符和换行符。从流中读取回车符和换行符时，它们会映射为一个字符'\n'。在其他系统如Unix上，新行字符是一个字符。在二进制模式中，不允许在流中传送字符，而是传送没有经过转换的原字节。

2. 流的读写操作

对流的读写有两种方法。首先，可以使用提取和插入运算符读写各种类型的数据，本书在从键盘上读取数据和在屏幕上输出数据时，就使用了这两个运算符。这称为格式化的输入输出操作，在文本模式下进行。程序中的二进制数值数据，例如整数和浮点数值，会转换为字符表示，之后再写到流中，在读取数据值时，则进行这个逆过程。本书中的写入cout和读取cin的操作都是在文本模式下的格式化输入输出操作。

处理流的第二种方式是读写字符。读写操作可以针对一个字节、给定数量的字节、用某种分隔符隔开的一组字节进行，但这种方法最重要的一点是只能读写字节，不能读写其他类型的数据。这些称为未格式化的输入输出操作，应在二进制模式下使用这些操作。写入流中的数据实际上由字符串和各种类型的二进制数值数据组成，但无论数据是什么，都是由字节组成了内存中的数据值，它们直接写入流中。

二进制流的一个要点是，如果数据的源或目的地是不同的系统，就会进行编译。二进制值在内存中的表示方法将随着系统的不同而不同，所以在读取另一台计算机中的二进制数据时，需要考虑这一点。如果读者想更多地了解这类问题，可以参阅附录E。

19.1.2 使用流的优点

在C++中，把流用作输入输出操作的基础，主要原因是使这些操作的源代码独立于所涉及到的物理设备。这有两个优点：首先，不必担心每个设备的具体机制，因为这些都是在后台进行的。第二，程序可以处理各种不同的输入输出设备，而不必对源代码作任何修改。

输出流(换言之，在写入数据时，数据所到达的目的地)的物理现实可以是能传送字节序列的任何设备。一般情况下，输出流是屏幕、硬盘上的文件或打印机。标准库定义了三个输出流：cout、cerr和clog。这些输出流一般与显示屏幕有关。cout是标准的输出流，cerr和clog与标准错误流相关，标准错误流用于记录程序中的错误。后两个输出流的区别是：cerr不放在缓存中(所以数据会立即写到输出设备上)，而clog放在缓存中(所以数据仅在缓存已填满后才写入)。图19-1显示了一些设备和它们表示的流。

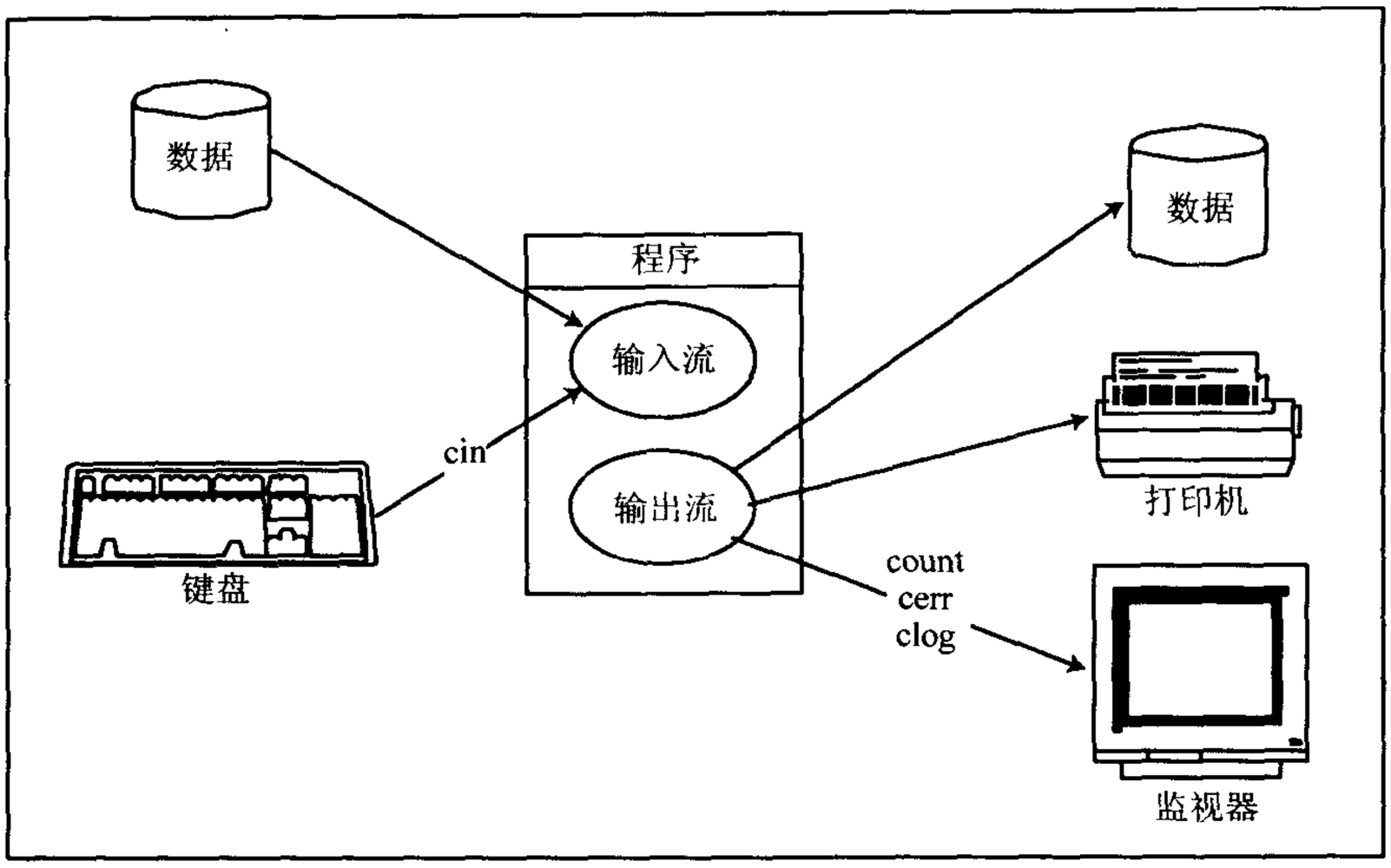


图 19-1　I/O 设备和流

在原则上，输入流也可以是任何序列的数据源，但一般是磁盘文件或键盘。标准库定义了一个标准输入流 cin，它通常与键盘相关。

C++中的流是类的对象，标准流是与系统上特定外部设备相关的预定义对象。在使用提取运算符>>读取 cin 中的对象，或使用插入运算符<<把对象写入 cout 时，实际上使用了这些对象的 operator<<()和 operator>>()重载函数。对于标准流，输入输出功能已经定义好了。但是，在使用非标准流时，例如文件的输入输出，必须创建需要的流对象，将它们与数据的物理源或目的地关联起来。下面介绍定义流的类。

19.2　流类

流输入输出涉及到许多类，但我们只对主要的类及其它们之间的关系感兴趣，如图 19-2 所示。

这是一个简化了的结构图，但足以理解其原则。ios_base 是一个普通的类，其他都是模板的实例，例如，istream 类就是 basic_istream 模板的一个实例，ios 类是 basic_ios 模板的一个实例。但是，我们感兴趣的是类，而不是它们的模板，因为程序中使用的是类。流类共享一个共同的基类 ios，它从 ios_base 类中继承了记录流状态的标志和格式化模式。因此，所有提供输入输出操作的流类都共享同一组状态和格式化标志，以及查询和设置它们的函数。

标准输入流 cin 是 istream 类型的对象，标准输出流 cout、cerr 和 clog 是 ostream 对象。处理文件的流类 ifstream、fstream 和 ofstream 都把 istream、ostream 或两者作为其基类，所以标准流 cin 和 cout 的功能也可用于文件流。

模板类一般用类型参数来指定某个流的字符集，图 19-2 中的类应用于处理 char 类型字符的流。这些模板的实例定义了可以处理 wchar_t 类型字符的流，这些流是 wistream、wostream、wiostream、wifstream、wfstream 和 wofstream。本章不讨论这些多字节字符流类，但它们的工作方式与字节流类相同。

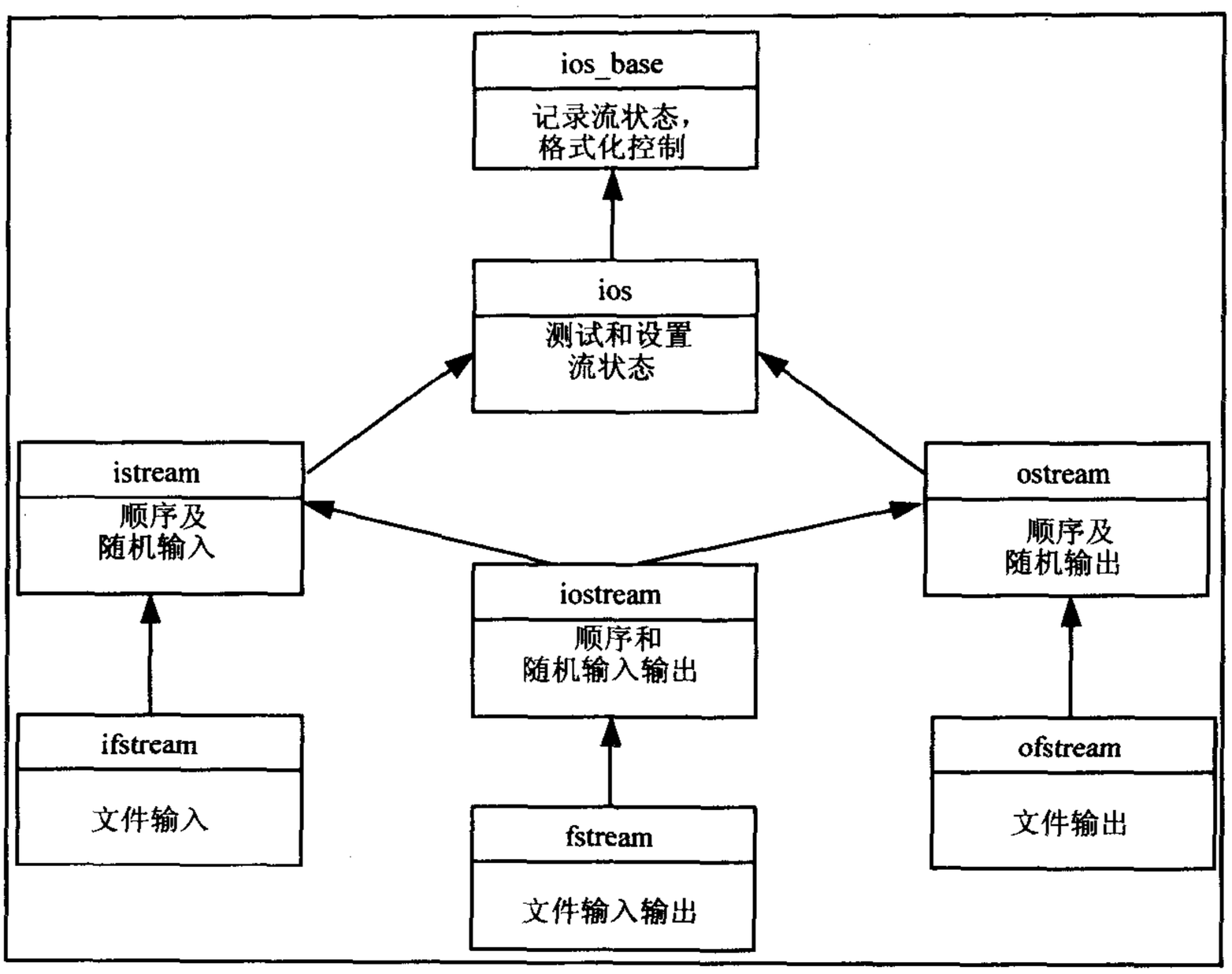


图 19-2　主要的流类

有时会看到原模板的引用，而不是图 19-2 中缩写的类名引用，此时需要注意模板名。流类使用 typedef 定义，如下所示：

```
typedef basic_ios<char>       ios;
typedef basic_istream<char>   istream;
typedef basic_ostream<char>   ostream;
typedef basic_iostream<char>  iostream;
typedef basic_ifstream<char>  ifstream;
typedef basic_ofstream<char>  ofstream;
typedef basic_fstream<char>   fstream;
```

对应的多字节字符流只是把 wchar_t 作为模板的类型参数，来替换 char。本章的其他地方将使用缩写的类名来代替完整的模板名，因为它们拼写起来比较简单。

19.2.1　标准流

标准流在 std 命名空间中定义为流类对象。这些定义在头文件 iostream 中：

```
extern istream cin;
extern ostream cout;
extern ostream cerr;
extern ostream clog;
```

头文件 iostream 也定义了对应的多字节字符流对象：

```
extern wistream wcin;
extern wostream wcout;
extern wostream wcerr;
```

```
extern wostream wclog;
```

前面已经广泛使用了标准输入流 cin 和标准输出流 cout。cerr 和 clog 流与 cout 的使用方式完全相同。本章不再重复前面章节中处理标准流的读写方式，而主要论述它们起作用的背景知识，以及如何把这些技术和机制应用于其他流类型。

首先，复习前面已介绍过的格式化流操作，再探讨如何把它们用于文件和标准流。接着阐述未格式化的流操作如何进行，如何和何时使用它们。

19.2.2 流的插入和提取操作

前面用于标准流对象的插入和提取运算符也可以用于其他类型的流对象。所有的标准流操作都在文本模式下进行，因为这个模式能确保数据在输出中正确显示。

插入和提取运算符在原则上与数据值的内部二进制表示和它们的外部字符表示之间的转换相关。在使用这些运算符和非标准的流时，通常会在文本模式下使用它们，因为它们适合于处理数据的基于文本的表示。文本模式与确保数据的可视化表示是正确的有关。

1. 流的提取操作

函数 operator>>()实现为 istream 类的一组重载成员，可以从输入流中读取基本类型的数据。在 C++中，每个基本数据类型都有这个运算符的一个重载版本。下面通过编写一些语句，介绍这些代码如何与这些运算符函数联系在一起：

```
int i=0;
double x=0.0;
std::cin>>i>>x;
```

cin 是 istream 类型的对象。最后一个语句从标准流 cin 中读取了两个变量，该语句可以转换为：

```
(std::cin.operator>>(i)).operator>>(x);
```

每次使用提取运算符时，都要调用一次 operator>>()函数。流的 operator>>()函数返回调用它的流对象的引用(在本例中是 cin)，因此返回值用于调用下一个运算符函数。operator>>()函数的参数必须是一个引用，才能让函数在传送为函数参数的变量中存储从流中读取的数据值。

空白字符总是看作值之间的分隔符，因此不能使用 istream 类的 operator>>()成员把空白字符读入程序。过量的空白字符一般会被忽略。在从键盘上读取一行文本时，必须使用 cin 的 getline()函数。

对于下述基本类型，operator>>()函数实现为 istream 类的一个重载的成员集。

short	int	long
unsigned short	unsigned int	unsigned long
float	double	long double
bool	void*	

支持最后那些类型的函数 void*可以把地址值读入任意类型的指针，但 char 类型的指针除外，该指针引用的是非空字符串，是一种特殊情形。任意类型的指针都可以作为参数，传送给参数类型为 void*引用的函数，但 operator>>()函数有一些非成员版本，它们的参数可以是非空字符串的指针，其原型如下：

```
istream& operator>>(istream& in, signed char* pStr);
istream& operator>>(istream& in, unsigned char* pStr);
```

在提取运算符中使用非空字符串的指针时，总是会选择其中一个函数。如果要读取一个地址，把它存储在 char*类型的指针中，就必须把它读取为 void*类型，再把地址强制转换为 char*类型，以存储它。

istream 类的非成员 operator>>()函数还支持使用提取运算符读取一个字符，因为这些函数可以使用为 istream 对象定义的 get 函数来实现。读取一个字符的函数有三个版本，分别对应于类型 char、signed char 和 unsigned char。本章后面将讨论 get()函数。

2. 流的插入操作

对基本类型的数据值的格式化流输出操作而言，ostream 类中的 operator<<()函数是重载的。输出到 cout 类似于 cin 的输入操作。下面的语句把数据值 i 和 x 输出到 cout 中：

```
std::cout<<i<<' '<<x;
```

这个语句调用了三次 operator<<()函数：

```
operator<<(std::cout.operator<<(i), ' ').operator<<(x);
```

operator<<()函数的所有版本都返回所调用的流对象的引用，所以返回值总是可以用于调用下一个 operator<<()函数。把单个字符和非空字符串写入流中的 operator<<()函数作为非成员函数实现的。这就是为什么把 i 写入流中的运算符函数调用是写入空格的运算符函数调用的第一个参数。后者返回流对象，该流对象又调用了写入 x 值的成员函数。

ostream 类中的 operator<<()函数也为一些基本类型进行了重载，这些类型与 istream 类中的 operator>>()函数所重载的类型相同。另外，单个字符或非空字符串的输出由 operator<<()的非成员版本创建。给输出流写入单个字符的函数原型如下：

```
ostream& operator<<(ostream& out, char ch);
ostream& operator<<(ostream& out, signed char ch);
ostream& operator<<(ostream& out, unsigned char ch);
```

输出非空字符串的函数原型如下：

```
ostream& operator<<(ostream& out, const char* pStr);
ostream& operator<<(ostream& out, const signed char* pStr);
ostream& operator<<(ostream& out, const unsigned char* pStr);
```

当其他类型的指针总是在流中写入为地址时，就可以使用指针输出字符串。由于存在这些函数，给输出流发送 const char*类型的变量时，就会把字符串写入指针指向的流中，而不是写入存储在指针变量中的地址。如果因某个原因要输出包含在指针中的地址，而不是输出字符串，就必须将它显式强制转换为 void*类型。这样就会调用 ostream 中带有该参数类型的成员函数，

把地址发送到流中。下面的语句：

```
const char* pMessage ="More is less and less is more.";
cout << pMessage;
```

会显示消息。要输出包含在 pMessage 中的地址，必须使用下面的输出语句：

```
cout << static_cast<void*>(pMessage);
```

19.2.3 流操纵程序

前面广泛使用了操纵程序，来控制流的格式。表 19-1 所示的基本操纵程序可以插入到流中。

表 19-1 基本流操纵程序

操 纵 程 序	含 义
dec	把整数的默认进制设置为十进制
oct	把整数的默认进制设置为八进制
hex	把整数的默认进制设置为十六进制
fixed	以没有指数的固定点表示法输出浮点数值
scientific	以带有指数的科学表示法输出浮点数值
boolalpha	把 bool 值表示为字母，其英文是 true 和 false
noboolalpha	把 bool 值表示为 1 和 0
showbase	表示八进制(前缀是 0)和十六进制整数(前缀是 0x)的基数
noshowbase	省略八进制和十六进制整数的基数表示
showpoint	总是把浮点数值输出到流中，且带有小数点
noshowpoint	输出整型浮点数值，不带小数点
showpos	对正整数加上+前缀
noshowpos	正整数不加+前缀
skipws	跳过输入中的空白
noskipws	不跳过输入中的空白
uppercase	对十六进制数字 A 到 F、以及指数 E 使用大写字母
nouppercase	对十六进制数字 a 到 f、以及指数 e 使用小写字母
internal	插入填充字符，使输出占满整个字段宽度
left	输出字段中的值左对齐
right	输出字段中的值右对齐
endl	在流缓存中写入一个换行符，把缓存中的内容写到流中
flush	把流缓存中的数据写入流中

这些操纵程序都可以直接放在流中。例如：

```
int i=1000;
```

```
std::cout<< std::hex<< std::uppercase<<i;
```

这个语句把整数 i 的值输出为十六进制值，其中十六进制数字使用大写字母。换言之，在屏幕上，i 的值是 3E8。

下面看看其工作原理。使用插入运算符会调用 operator<<()函数，但这里没有涉及任何操纵程序，因为它们仅处理已输出到流中的数据。这些操纵程序并不会把数据发送到流中，所以它们也不是数据值。实际上，表 19-1 中的所有操纵程序都是同一类型的函数指针。在使用某个操纵程序时，就会调用接受对应函数指针的特定 operator<<()函数版本，再把操纵程序传送为一个参数。

例如，操纵程序 hex 是下述函数的名称：

```
ios_base& hex(ios_base& str);
```

在语句中可以使用这个操纵程序：

```
std::cout<<std::hex<< i;
```

该语句转换为：

```
(std::cout.operator<<( std::hex)).operator<<(i);
```

operator<<()的第一个调用把函数的指针 hex 作为一个参数。在运算符函数中，调用了函数 hex()，来设置输出格式，并把 i 的值以十六进制格式传送给流。

如本章开头所述，hex()原型中的 ios_base 类是 ios 的基类。因为所有的流类都继承了这个基类，所以类型 ios_base&可以引用任何流对象。所有的操纵程序都是函数的指针，该函数有一个参数，其返回类型是“ios_base 的引用”，所以它们都会调用同一个版本的 operator<<()函数，该函数再调用参数指向的函数。ios_base 定义了控制流的标志，使用操纵程序时调用的函数会修改对应的标志，以生成需要的结果。使用流对象的 setf()和 unsetf()函数可以直接修改这些标志，但使用操纵程序修改它们会更简单一些。

带有参数的操纵程序

有一些操纵程序在使用它们时可以接受参数。为了访问这些操纵程序，必须在源文件中包含<iomanip>头文件，因为该头文件包含了它们的声明。使用这些操纵程序与使用其他操纵程序的方式相同，都是直接把函数调用插入到流中。这些操纵程序如表 19-2 所示。

表 19-2　接受参数的操作程序

操 纵 程 序	含　义
setprecision(int n)	把输出的浮点数的精确度设置为 n 位。这将一直有效，直到修改了它
setw(int n)	把下一个输出值的字段宽度设置为 n 个字符。在每个输出中，字符宽度都会重新设置为默认值，即输出值的字段宽度刚好可容纳该值
setfill(char ch)	把输出字段中的填充字符设置为 ch。这是模式化的，所以会一直有效，直到修改了它
setbase(int base)	根据参数的不同，把整数的输出表示方式设置为八进制、十进制或十六进制。其他参数不会改变当前的进制

这些操纵程序的返回类型都已定义好。使用它们的一个例子如下：

```
std::cout<< std::endl<< std::setw(10)<< std::setfill('*')<< std::left<<i;
```

这个语句在 10 字符宽的字段中以左对齐的方式输出 i 的值。字段中未使用的字符位置用*填充。填充字符对输出值后面的字符位置起作用，但在每个输出值前面必须显式设置字符宽度。

注意：

给本身是函数指针的操纵程序(如 left)加上括号是错误的，因此不要把它们与这里讨论的 4 个操纵程序相混淆。

<iomanip>头文件也声明了函数 setiosflags()和 resetiosflags()，这两个函数可以通过指定一个掩码来设置或重新设置控制流格式的标志。构建传送为参数的掩码时，可以使用按位或运算符，把在 ios_base 类中定义的标志组合起来。每个标志的名称都与设置它的操纵程序名相同，下面就为左对齐、十六进制输出设置了标志：

```
std::cout<< std::endl<< std::setw(10)
         << std::setiosflags(std::ios::left | std::ios::hex)<<i;
```

这个语句在 10 字符宽的字段中以左对齐的方式输出 i 的十六进制值。这里可以把 ios 用作标志名的限定符，因为标志在 ios 类中继承于 ios_base 类。

19.3 文件流

有三种流类对象可以用于处理文件：ifstream、ofstream 和 fstream。如前所述，这三个类的基类分别是 istream、ostream 和 iostream。istream 对象表示一个可以读取的文件流，ofstream 对象表示一个可以写入的文件输出流，fstream 对象表示一个可以读写的文件流。

在创建文件流对象时，可以把它与磁盘上的物理文件关联起来。另外，还可以创建与特定文件没有关联的文件流对象，以后再使用成员函数建立与特定文件的关联。为了读写物理文件，该文件必须是打开的，并通过操作系统把它关联到程序上，再设置一组许可，来描述如何使用该文件。如果在创建文件流对象时，使之与特定文件相关联，该文件就是打开的，并可以立即在程序中使用。注意可以修改与文件流对象关联的物理文件，所以在不同的时间可以用一个 ofstream 对象写入不同的文件。

文件流有一些重要的属性。它有长度，也就是流中的字符数。它还有开头，也就是流中的第一个字符。它也有结尾，也就是流中最后一个字符后面的位置。它有当前位置，也就是流中开始执行下一个读写操作的字符的索引位置。文件流中的第一个字符的索引位置是 0。这些属性允许在文件中浏览，读取感兴趣的部分，或改写文件的选定区域。

19.3.1 写入文件

在研究文件流时，首先看看如何写入文件输出流。输出文件由 ofstream 对象表示，创建 ofstream 对象的语句如下所示：

```
ofstream outFile("filename");
```

文件 filename 会自动打开，准备写入，接着就可以以默认模式即文本模式写入数据。如果文件 filename 不存在，就会创建它。如果该文件已经存在，文件中的所有数据就会删除，要写入该文件的数据将构成它的新内容。即使在打开文件后不写入数据，原来的内容也会删除，变成一个空文件。对象 outFile 包含 ostream 子对象，所以标准流 cout 的所有流操作都可以应用于 outFile。下面用第 7 章的一个程序示例 7.6 来说明，该例子生成了质数。

程序示例 19.1——写入文件

这里把质数写入文件，而不是写到屏幕上。只需给文件定义一个 ostream 对象，用它代替例子中的 cout 即可。下面是修改后的代码，并用黑体字突显修改过的语句。

```
// Program 19.1 Writing primes to a file          File: prog19_01.cpp
#include <fsteam>                                 // For file streams
#include <iomanip>

int main() {
  const int max = 100;                            // Number of primes required
  long primes[max] ={2, 3,  5};                   // First three primes defined
  int count = 3;                                  // Count of primes found
  long trial = 5;                                 // Candidate prime
  bool isprime = true;                            // Indicates when a prime is found

  do {
    trial += 2;                                   // Next value for checking
    int i = 0;                                    // Index to primes array

    // Try dividing the candidate by all the primes we have
    do {
      isprime = trial % *(primes + i)  > 0;// False for exact division
    } while(++i < count && isprime);

    if(isprime)                                   // We got one...
      *(primes + count++)  = trial;               //  ...so save it in primes array
  } while(count < max);

std::ofstream outFile("C:\\JunkData\\primes.txt"); // Define file stream object

// Output primes 5 to a line
for(int i = 0; i< max ; i++) {
    if(i % 5 ==0)                                 // New line after every 5th prime
      outFile << std::endl;
    outFile << std::setw(10) << *(primes +i);
  }
  return 0;
}
```

这个程序没有在屏幕上显示任何输出。惟一的输出是已写入数据的文件。注意，如果使用程序中指定的路径，就必须先创建 JunkData 目录，再运行程序。该程序只创建文件，不创建目录，所以在运行程序之前必须先创建目录。使用任何文本编辑器都可以查看文件的内容。

例子的说明

首先，给<fstream>头文件指定一个#include 指令，该头文件定义了文件流类：

```
# include <fstream>          //For file streams
```

这里不再需要<iostream>头文件，因为本例没有使用标准流。所有的计算都与第 7 章的例子相同，对 main()的第一个修改是添加文件输出流对象的声明：

```
std::ofstream outFile("C:\\JunkData\\primes.txt");        //Define file stream object
```

这个语句定义了一个 ofstream 对象 outFile，并把它与 C 盘中 JunkData 目录下的 primes.txt 文件关联起来。这行代码使用了 MS-DOS 表示法，因此读者应使用与操作系统环境对应的文件路径和文件名。路径中的反斜杠\要使用\\，因为一个反斜杠\表示转义序列的开头。在本例中，文件名中使用了扩展名.txt，因为我们在文本模式下编写文件，处理.txt 文件的应用程序可以查看该文件。这个程序没有添加检查功能，但它是可以添加的，稍后介绍。

如果没有指定路径，文件就假定位于当前目录。如果它不存在于当前目录，就创建它。如果要避免出错，可以采用这里的做法，在用于检查文件操作的地方建立一个目录。只要在程序中使用了完全合格的文件名，就不会有损坏重要文件的危险。

对程序的最后一处修改是把数据写入文件，其方式与以前写入 cout 完全相同，也是使用插入运算符<<：

```
for(int i=0; i<max; i++) {
  if(i%5==0)                                    //New line after every  5th prime
    outFile<< std::endl;
  outFile<< std::setw(10)<< *(primes+i);
}
```

在程序执行完后，C:\JunkData 目录(或指定的其他路径)下应有文件 primes.txt。如果使用文本编辑器查看文件的内容，就会看到该文件包含的数据与程序最初版本显示的数据相同。

当然，如果仅把文件用作中间存储介质，而且不想直接查看内容，就可以用换行符分发文件的内容。如果要使用提取运算符把数据读取出来，仍旧需要在各个值之间插入空白，但可以把空白减少为一个空格。因此，输出语句就变成：

```
for(int i=0; i<max; i++)
    outFile<<' '<< *(primes+i);
```

注意要关闭文件，不必做任何工作。在释放 ostream 对象时，与该文件关联的文件会自动关闭，因此，在 outFile 对象超出作用域时，文件就会关闭。如果要显式关闭文件，可以调用该对象的 close()函数，下面的语句会关闭 outFile 对象所表示的文件：

```
outFile.close();
```

在系统上，ostream 对象会改写已有的文件内容。要验证这一点，可以给质数的个数指定另一个值，再次运行程序，然后查看文件的内容。

提示：

不是每次都需要改写文件，这仅是默认设置下的情况。本章的后面会说明如何控制文件的写入方式。

19.3.2 读取文件

要读取文件，可以创建 ifstream 类型的对象，并把它与磁盘上的文件关联起来。例如：

```
const char* filename="C:\\JunkData\\primes.txt";
std::ifstream inFile(filename);
```

这个语句定义了对象 inFile，并把它与 C 盘上 JunkData 目录下的文件 primes.txt 关联起来，然后打开了该文件。如果要读取一个文件，该文件必须存在，但有时这个文件是不存在的。如果试图读取一个还没有准备好的文件，会发生什么？

就这个定义而言，答案是什么也不做：文件流对象不工作。要确定一切是否按预想的那样工作，必须测试文件的状态，这有几种方式。一种方式是调用 ifstream 对象的 is_open()成员函数，如果文件已打开，该函数就返回 true，否则就返回 false。另一种方式是调用文件流类中从 ios 类继承而来的 fail()函数，如果发生文件错误，该函数就返回 true。另外，还可以对文件流对象使用!运算符。这个运算符在 ios 类中进行了重载，可以检查在该类中定义的流状态指示符。在应用于流对象时，如果流不处于满意的状态，它就返回 true。使用重载的！运算符，等价于调用 fail()函数，以确保流对象处于满意的状态，可以使用下面的语句：

```
if(!inFile) {
  std::cout<< std::endl<<"Failed to open file"<<filename;
  return 1;
}
```

可以用相同的方式测试输出文件流对象是否可用。因为 ofstream 类也继承了重载的！运算符。它还继承了 fail()函数，实现了 is_open()函数。本章后面会进一步论述流错误状态。

以文本模式读取文件类似于从 cin 中读取——使用提取运算符的方式是相同的。但是，如果不知道文件中有多少个数据值，如何判断何时会到达文件末尾？在 ofstream 类中，继承自 basic_ios 的 eof()函数提供了一个简洁的解决方案。当到达文件末尾时，该函数返回 true，因此可以一直读取数据，直到 eof()函数返回 true 为止。

程序示例 19.2——读取文件

现在知道输入文件流如何工作了，下面的程序读取前一个例子所编写的文件。本例只是读取该文件，把值显示到屏幕上。下面是代码：

```
// Program 19.2 Reading the primes file    File:  prog19_02.cpp
#include <fstream>
#include <iostream>
#include <iomanip>
using std::cout;
using std::endl;

int main() {
   const char* filename = "C:\\JunkData\\primes.txt";  // Name of the file to read
   std::ifstream inFile(filename);           // Create input stream object

   // Make sure the file stream is good
```

```
if(!inFile) {
   cout << endl << "Failed to open file" << filename;
   return 1;
}

long aprime = 0;
int count  = 0;
while(!inFile.eof()) {                    // Continue until EOF is found
    inFile >> aprime;                     // Read a value from the file
    cout << (count++ % 5 == 0 ? "\n" : "") << std::setw(10) << aprime;
    }
    cout << endl;
    return 0;
}
```

这个例子的输出结果如下：

```
  2          3          5          7         11
 13         17         19         23         29
 31         37         41         43         47
 53         59         61         67         71
 73         79         83         89         97
101        103        107        109        113
127        131        137        139        149
151        157        163        167        173
179        181        191        193        197
199        211        223        227        229
233        239        241        251        257
263        269        271        277        281
283        293        307        311        313
317        331        337        347        349
353        359        367        373        379
383        389        397        401        409
419        421        431        433        439
443        449        457        461        463
467        479        487        491        499
503        509        521        523        541
```

例子的说明

下面的语句从 filename 中创建文件流对象：

```
const char* filename="C:\\JunkData\\primes.txt"; //Name of the file to read
std::ifstream inFile(filename);                  //Create input stream object
```

在使用文件之前，需要检查它是否已成功打开：

```
if(! inFile) {
  cout<<endl<<"Failed to open file"<<filename;
  return 1;
}
```

inFile 对象的 operator!()函数继承于基类 ios，如果构造函数没能正确创建流对象，并打开文件，该函数就返回 true。例如，如果文件不存在，该函数就返回 true。如果出现了错误，就显示

错误消息，结束程序。

文件打开后，就从文件中读取数值，把它们输出到标准流 cout 中：

```
long aprime=0;
int count=0;
while(!inFile.eof()) {                    //Continue until EOF is found
  inFile>>aprime;                        //Read a value from the file
  cout<<(count++ %5 ==0 ? "\n" :"")<<setw(10)<<aprime;
}
```

在 inFile 对象的 eof()成员函数返回 true 值之前，while 循环会一直迭代下去。在该循环中，我们从 inFile 流中读取一个值，放在 aprime 中，再把该值输出到 cout 中。条件运算符在输出 5 个值后就输出一个换行符。在从文件中读取最后一个值后，eof()成员函数返回 true，因为当前文件位置在文件的末尾。

19.3.3 设置文件打开模式

ifstream 或 ofstream 对象的文件打开模式确定了处理文件的方式。它由在 ios_base 类中定义的位掩码值和 ios 类中继承为 openmode 类型的掩码值的组合来确定。可以设置的掩码值如表 19-3 所示。

表 19-3 文件流位掩码值

值	含 义
ios::app	在每次写入操作之前移动到文件的末尾(追加)。这可以确保仅在文件中添加数据，而不能改写数据
ios::ate	在打开文件时，移动到文件的末尾(在文件尾)。之后，可以把当前位置移动到文件的其他地方
ios::binary	设置二进制模式，而不是文本模式。在二进制模式中，所有的字符在输入输出文件时都不改变
ios::in	打开文件，进行读取
ios::out	打开文件，进行写入
ios::trunc	把已有文件的长度变成 0

因为这些都是位掩码，所以对这些值进行按位或组合，就会生成打开模式的一个说明。如果文件要以二进制模式打开，进行写入，且数据只能添加到文件的末尾，则应使用表达式 ios::out | ios::app | ios::binary 来指定文件打开模式。通过指定 ios::in 和 ios::out 就可以打开文件，进行读写。对此必须使用 fstream 类型的对象，稍后介绍。

把文件打开模式指定为文件流类构造函数的第二个参数，就可以设置文件打开模式。ifstream 和 ofstream 构造函数的第二个参数类型都是 openmode，并指定了其默认值。对于 ifstream 对象，文件打开模式的默认值是 ios::in，它指定文件打开后，进行输入。ofstream 对象的默认值是 ios::out | ios::trunc，它指定文件打开后进行输出。如果文件已经存在，其长度就设

置为 0，以确保改写已有的内容。

假定要给文件输出流指定文件打开模式，仅允许给文件追加数据。可以使用下面的语句：

```
const char* filename="C:\\JunkData\\primes.txt";
std::ofstream outFile(filename, std::ios::out|std::ios::app);
```

如果调用流对象的 close()函数显式关闭文件流，就可以通过调用流对象的 open()函数再次以新的打开模式打开文件。open()函数接受两个参数，第一个参数是文件名，第二个参数是打开模式掩码。在文件流类的 open()成员函数中，第二个参数的默认值与文件流类的构造函数相同。下面的语句关闭了 outFile 流，再用另一个打开模式重新打开它：

```
outFile.close();
outFile. open(filename);
```

这个语句重新打开文件，以改写原来的内容，因为 ios::out | ios::trunc 是第二个参数的默认值。

程序示例 19.3——指定文件打开模式

再次修改质数程序示例，生成并显示指定个数的质数，但这些质数存储在文件中，以备将来再次使用。这样，程序只需生成还没有存储在文件中的质数。文件中已经有的所有质数可以立即显示。下面是代码：

```
// Program 19.3 Reading and writing the primes file   File: prog19_03.cpp
#include <fstream>
#include <iostream>
#include <iomanip>
#include <cmath>
#include <string>
using std::ios;
using std::cout;
using std::cin;
using std::endl;
using std::string;

// Function to find the prime after lastprime
long nextprime(long lastprime, const char* filename);

int main() {
   const char* filename = "C:\\JunkData\\primes.txt";
   int nprimes = 0;                                    // Number of primes required
   int count = 0;                                      // Count of primes found
   long lastprime = 0;                                 // Last prime found

   // Get number of primes required
   int tries = 0;                                      // Number of input tries
   cout << "How many primes would you like (at least 3)?: ";
   do {
     if(tries)
        cout << endl << "You must request at least 3, try again: ";
     cin >> nprimes;
```

```
    if(++tries == 5) {                              // Five tries is generous
      cout << endl <<" I give up!" << endl;
      return 1;
    }
  } while(nprimes < 3);

  std::ifstream inFile;                          // Create input file stream object
  inFile.open(filename);                         // Open the file as an input stream

  cout << endl;
  if(!inFile.fail()) {
   do {
     inFile >> lastprime;
     cout << (count++ % 5== 0 ? "\n" : " ") << std::setw(10) << lastprime;
   } while(count < nprimes && !inFile.eof());
   inFile.close();
  }
  inFile.clear();                                // Clear any errors

  try {
    std::ofstream outFile;
    if(count == 0) {
      outFile.open(filename);                    // Open file to create it
      if(!outFile.is_open())
        throw ios::failure(string("Error opening output file ") +
                           string(filename) +
                           string(" in main()"));
      outFile <<" 2 3 5";
      outFile.close();
      cout << std::setw(10) << 2 << std::setw(10) << 3 << std::setw(10) << 5;
      lastprime = 5;
      count = 3;
  }

  while(count < nprimes) {
    lastprime = nextprime(lastprime, filename);
    outFile.open(filename, ios::out | ios::app);     // Open file to append data
    if(!outFile.is_open())
       throw ios::failure(string("Error opening output file ") +
                          string(filename) +
                          string(" in main()"));
      outFile <<" "<< lastprime;
      outFile.close();
      cout << (count++ % 5 == 0 ? "\n" : "") << std::setw(10) << lastprime;
   }
   cout << endl;
   return 0;
  }
  catch(std::exception& ex) {
    cout << endl << typeid(ex).name() << ": " << ex.what();
    return 1;
```

```
    }
  }
```

这个程序使用 nextprime()函数，返回第一个参数值后面的下一个质数。第二个参数是包含目前已找到的所有质数的文件名。该函数的定义如下：

```
// Find the next prime after the argument
long nextprime(long lastprime, const char* filename) {
  bool isprime = false;                          // Indicator that we have a prime
  long aprime = 0;                               // Stores primes from the file
  std::ifstream inFile;                          // Local input stream object

  // Find the next prime
  for( ; ; ) {
    lastprime += 2;                              // Next value for checking
    long limit = static_cast<long>(std::sqrt(static_cast<double>(lastprime)));

    // Try dividing the candidate by all the primes up to limit
    inFile.open(filename);                       // Open the primes file
    if(!inFile.is_open())
      throw ios::failure(string("Error opening input file ") +
                         string(filename) +
                         string(" in nextprime()"));
    do {
      inFile >> aprime;
    } while(aprime <= limit && !inFile.eof() &&
            (isprime = lastprime % aprime > 0));

    inFile.close();
    if(isprime)                                  // We got one...
      return lastprime;                          // ...so return it
  }
}
```

这个程序输出所请求的质数个数，并把它们写到文件 primes.txt 中。

例子的说明

这个程序目前的结构有所不同。由于本例把所有的质数都放在一个文件中，不再需要把它们存储在内存中。质数是由 nextprime()函数找出来的，但在研究该函数之前，先看看 main()中代码的工作原理。

在用户输入质数的个数后，就打开文件 primes.txt，作为一个输入流，如下面的语句所示：

```
std::ifstream inFile;                //Create input file stream object
inFile.open(filename);               //Open the file as an input stream
```

ifstream 对象 inFile 使用默认的构造函数创建。目前，该对象没有与特定文件关联起来。要使用流对象打开文件 primes.txt，可以调用 open()函数，把文件名作为一个参数。这个函数还接受第二个参数——文件打开模式，但这里没有指定它，所以使用默认值 ios::in。

当然，文件还不存在，因此在读取它之前，必须验证文件流是否处于好的状态。使用 fail()函数测试文件流对象：

```
if(!inFile.fail()) {
  do {
    inFile>>lastprime;
    cout<<(count++ % 5 ==0 ? "\n" : "")<<std::setw(10)<< lastprime;
  }while(count<nprimes && ! inFile.eof());
  inFile.close();
}
```

如果 fail()返回 false，就读取文件。在 if 块的 do-while 循环中，从文件中读取 nprimes 个数字。在循环条件中，调用 inFile 对象的 eof()函数来检查是否到达文件末尾，再把返回的 bool 值与 count 和 nprimes 比较的结果进行逻辑与操作，如果从文件中读取了所需的质数，或到达了文件末尾，循环就结束。

在 if 语句的后面，是文件流对象的错误集合，错误的原因有文件不存在、到达了文件末尾等。在关闭文件时没有重新设置这些错误。因为要再次使用对象，可以调用流对象的 clear()函数，重新设置错误标志：

```
inFile.clear();       //Clear any errors
```

本章的后面将讨论 clear()函数还有什么其他用途。

main()中的其他代码都放在一个 try 块，因为如果在打开文件时遇到错误，就要抛出异常。Catch 块会捕获以 exception 为基类的异常类型，并在同一个地方处理同一类型的错误。

现在，可以把 count 的值用作一个指示器，说明是否从文件中读取了质数。如果没有读取，就可以把前三个质数写入文件，如下面的代码所示：

```
std::ofstream outFile;
if(count ==0) {
  outFile.open(filename);          // Open file to create it
  if(!outFile.is_open())
    throw ios::failure(string("Error opening output file")+
                       string (filename) +
                       string("in main()"));
    outFile <<" 2 3 5";
    outFile.close();
    cout << std::setw(10) <<2<< std::setw(10) << 3 << std::setw(10) <<5;
    lastprime =5;
    count =3;
}
```

创建一个输出文件流对象 outFile，再调用其 open()函数打开 primes.txt 文件。打开该文件时，所使用的打开模式指定为 ios::out | ios::trunc，所以文件以前的内容会被删除，我们将从文件的开头写入数据。在代码中，在 if 语句中调用 outFile 的 is_open()函数成员来验证文件已打开，并可以写入数据。如果一切正常，就把前三个质数写入文件，并关闭文件。然后把这三个质数写到 cout 中，设置 lastprime 值和质数的个数。如果因某种原因，不能打开文件进行写入操作，就抛出 ios::failure 类型的异常。这个异常类在 ios_base 类中定义，因此 ios 和所有的流类都继承了这个异常类。当然，作为一个标准的异常类，它把 exception 类作为其基类。构造函数接受 string 类型的参数，给该构造函数传送的参数会被对象的 what()成员函数返回，所以可以使用这个消息来标识 catch 块中的异常，如 main()所示。

现在，如果文件中的质数个数少于需要的质数个数，就需要计算更多的质数。这在一个 while 循环中进行，该循环在 count 等于 nprimes 时结束：

```
while(count < nprimes) {
   lastprime = nextprime(lastprime,  filename);
   outFile.open(filename, ios::out|ios::app);      // Open file to append data
   if(!outFile.is_open())
      throw ios::failure(string("Error opening output file ")  +
                         string(filename)  +
                         string(" in main()"));
   outFile << " "  << lastprime;
   outFile.close();
   cout <<  (count++ % 5 == 0 ? "\n"  :  " " ) << std::setw(10)  << lastprime;
}
```

nextprime()函数计算第一个参数值后面的下一个质数。第二个参数指定文件名，nextprime()函数要从这个文件中提取要在计算中使用的已有质数。接着打开文件，进行输出操作，但这次，文件打开模式确定写入文件的内容总是追加到文件的末尾。如果打开文件时出了问题，就抛出一个异常。

在写入 nextprime()函数返回的质数后，就关闭文件。这是必须的，因为在下一次迭代中，nextprime()需要再次打开文件，读取它。在每次迭代时，找到的质数也会写入 cout。在显示了 nprimes 个质数后，循环就会停止。循环结束之后，文件至少包含开始时的质数集合，因为已经在文件中添加了新的质数。

在 nextprime()函数中查找新质数，需要把文件中的质数用作除数，因此创建一个本地的 ifstream 对象。查找下一个质数的过程在一个 for 无限循环中进行。要检查的第一个值是给 lastprime 递增 2 后得到：

```
lastprime +=2;                    //Next value for checking
```

作为 lastprime 传送给函数的值应是我们找到的最后一个质数，因此不需要检查它是否为奇数。要检查一个值是否为质数，就必须用小于或等于该值的平方根的所有质数去除这个值，下面的语句就进行这个计算，并得到整数值 limit：

```
long limit=static_cast<long>(std::sqrt(static_cast<double>(lastprime)));
```

这个语句使用在 cmath 头文件中声明的标准库函数 sqrt()。必须把参数强制转换为 double，编译器才能选择要使用的函数重载版本，然后把 sqrt()函数返回的 double 结果转换回 long。

下面的语句打开文件流，验证该文件已成功打开：

```
inFile.open(filename);                  //Open the primes file
if(!inFile.is_open())
  throw ios::failure(string("Error opening input file")+
                     string(filename)+
                     string("in nextprime()"));
```

如果打开文件的过程失败，就抛出一个异常，其类型与在 main()中为文件问题抛出的异常类型相同。这里，构造函数的参数有点不同，可以标识出程序中的哪个地方会出问题。main()中的 catch 块会捕获这类异常，因此所有的文件问题都在同一个地方处理。

打开输入文件流后，在 do-while 循环中检查待选的质数：

```
do {
  inFile>>aprime;
}while(aprime<= limit && ! inFile.eof() &&
                                        (isprime=lastprime % aprime >0));
```

在循环中，质数从文件中读取，但其他工作都在循环条件中完成。在循环条件中有三个逻辑表达式用逻辑与运算符组合在一起。如果任何一个表达式为 false，循环就停止。

第一个条件是除数小于 limit，即候选值的平方根。如果除数大于等于 limit，就用小于等于这个 limit 的所有质数除这个候选值，如果相除的结果不是整数，该候选值就一定是质数。第二个条件是没有到达文件末尾，这是永远不应发生的。如果到达了文件末尾，程序或文件中就会出现严重的错误。第三个条件检查相除操作后的余数。如果余数是 0，就进行了整除操作，说明 lastprime 不是质数，需要对下一个候选值进行测试。

循环结束时，关闭文件，以便在下一个迭代的开始再次打开它。

```
inFile.close();
```

是否找到质数由循环条件中设置的 isprime 来表示。如果 isprime 是 true，就找到了一个质数，并返回该质数。否则，就再次执行 for 循环。

```
if(isprime)                          //We got one…
  return lastprime;                  //…so return it
```

这个程序可以正常工作，但每次执行循环时都要打开和关闭文件，这是非常低效的。如果每次都可以从文件的开头开始读取文件，就会好得多。下面看看如何从文件的开头开始读取。

管理当前流位置

使用在 istream 和 ostream 类中定义的函数可以控制当前流的位置，但这些函数不能应用于标准流，因为标准流与物理设备无关，而流的位置必须放在物理设备中才有意义。但是，它们可以应用于与物理文件相关的文件流类的对象。istream 和 ostream 类中的函数分别由 ifstream 和 ofstream 类继承，这两组函数也通过 iostream 类由 fstream 类继承。

基本上，在流位置方面可以做两件事：获取和记录流中的当前位置；改变当前位置。对于输入流对象，当前位置由 tellg()函数返回，对于输出流对象，当前位置由 tellp()函数返回。tellg()函数中的 g 表示 get，tellg()函数中的 p 表示 put，也就是从与函数相关的流中获取数据或在流中放置数据。这两个函数都返回 pos_type 类型的值，表示流中的一个绝对位置。

例如，下面的语句可以在输入文件流对象 inFile 中获取当前位置：

```
pos_type here=inFile.tellg();                    //Record current file position
```

可以把前面使用 tellg()或 tellp()函数记录下来的位置传送给输入流对象的 seekg()成员函数，或输出流对象的 seekp()成员函数，在流中定义一个新位置。只需用前面记录的流位置调用对应于流类型的函数，就可以把 inFile 的流位置重新设置为 here，如下面的语句所示：

```
inFile.seekg(here);                              //Set current position to here
```

pos_type 类型的值 here 是一个整数，它对应于流中的字符索引位置，第一个字符的索引位

置是 0。因此，可以使用数值移动到流中的某个指定位置。这在文本模式下是非常危险的，因为存储在流中的字节数可能与所编写的字符数不同。但是，搜索位置 0 总是会移动到流的开头。下面就在 nextprime()函数中使用它：

```
long nextprime(long lastprime, const char* filename) {
  bool isprime = false;                          // Indicator that we have a prime
  long aprime = 0;                               // Stores primes from the file
  std::ifstream inFile(filename);                // Local input stream object

  if(!inFile.is_open())
    throw ios::failure(string("Error opening input file ") +
                       string(filename) +
                       string("in nextprime()"));
// Find the next prime
for( ; ; ) {
  lastprime += 2;                                        // Next value for checking
  long limit = static_cast<long>(sqrt(static_cast<double>(lastprime)));

  // Try dividing the candidate by all the primes up to limit
  do {
    inFile >> aprime;
  } while(aprime <= limit && !inFile.eof() &&
                   (isprime = lastprime % aprime > 0));

  if(isprime) {                                          // We got one...
    inFile.close();                                      // ...so close the file...
    return lastprime;                                    // ...and return the prime
  }
  inFile.seekg(0);                                       // Move to beginning of
                                                         // file
  }
}
```

使用这个版本的函数，在创建 ifstream 对象时，就打开文件，在 for 循环的最后，把文件位置重新设置为文件中的第一个字符。

要利用 pos_type 类型的正值移动到流中指定的一个新位置，还可以使用相对于流中三个特定位置的偏移值来移动。偏移值可以是正数，也可以是负数。可以在流中定义一个相对于第一个字符的新位置值(偏移值必须为正)，或者相对于流中最后一个字符的新位置值(偏移值必须为负)，或者相对于当前位置的新位置值。在最后一种情形下，偏移值可以是正数，也可以是负数，只要不在流的端部即可。如图 19-3 所示。

要设置相对位置，可以使用接受两个参数的 seekg()或 seekp()函数。第一个参数是偏移值，这是一个 off_type 类型的整数值，第二个参数必须是下述在 ios 类中定义的值。如表 19-4 所示。

表 19-4　文件流位置值

值	描　　述
ios::beg	相对于文件中的第一个字符的偏移
ios::cur	相对于文件中的当前位置的偏移
ios::end	相对于文件中的最后一个字符的偏移

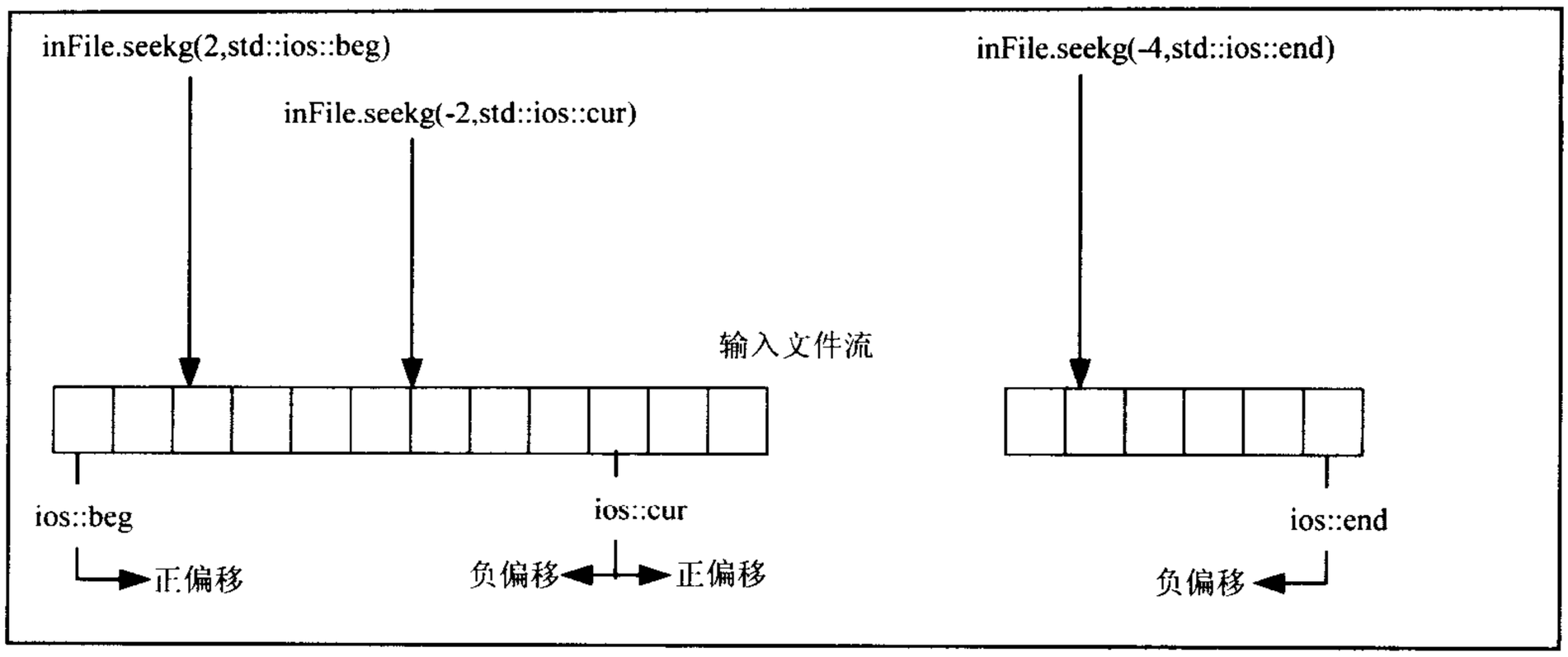


图 19-3　定义相对的流位置

相对于 ios::beg，偏移值必须为正。相对于 ios::end，偏移值必须为负。而相对于 ios::cur，偏移值可正可负。可以为偏移值指定显式的整型常量，也可以为偏移值提供结果是整数的表达式。

seekg()函数返回文件流对象的一个引用，因此可以使用提取运算符把搜索操作和输入操作组合起来。例如：

```
inFile.seekg(10, std::ios::beg)>>value;
```

这个语句把文件位置移动到与文件开头偏移 10 个字符的位置，并从该位置开始，把数据读入 value。

同样，可以使用 seekp()函数移动到要开始下一个输出操作的位置，函数参数与 seekg()完全相同。对流的下一次写入会改写字符，从处于新位置的字符开始改写。

相对搜索操作在文本模式中是一种不确定的操作，特别是在把字符写入流中时，就更是如此。这是因为所存储的字符数可能与已写入的字符数不同。

提示：

这里没有演示所讨论的操作，本章后面在讨论对流进行随机读写操作时会演示它。

19.4　未格式化的流操作

除了用于格式化流输入输出的插入和提取运算符之外，流类还有一些成员函数，可以把基于字符的数据传送给流，或从流传送基于字符的数据，而不需要对数据进行任何格式化。提取运算符把空白看作分隔符，但不会忽略它们。一般情况下，未格式化的流输入函数不会跳过空白字符，它们会像其他字符那样进行读和存储。

一些未格式化的流输入输出函数的一个重要用法是以二进制格式读写文件流。这些函数仅提供了一种方式来写入基于字符的数据，但可以使用其中的几个函数来读写二进制数值数据。首先，介绍未格式化的输入函数。

19.4.1 未格式化的流输入函数

在 istream 类中定义了丰富的未格式化输入函数，它们都在 ifstream、iostream 和 fstream 类中进行了继承。首先介绍成员函数 get()的四个变体，前两个变体从流中读取一个字符：

- int_type get();

这个版本从流中读取一个字符，并把它返回为 int_type 类型。int_type 类型是作为实现定义的，这个整数类型可以存储任何字符，它通常对应于类型 int。如果到达文件末尾，该函数就返回字符 EOF，这是在 iostream 头文件中定义的。如果某个字符因某种原因不能从流中读出，就设置错误标志 ios::failbit(本章后面在讨论 I/O 错误时，会讨论这个错误标志和其他错误标志)。

- istream& get(char& ch);

这个函数也从流中读取一个字符，但所读取的字符存储在 ch 中。该函数返回流对象的引用，因此可以把该函数与其他成员函数调用组合起来。与前一个函数一样，如果不能从流中读取一个字符，就设置错误标志 failbit，并在 ch 中存储 EOF。

还有另一个成员函数 peek()，它等价于上述第一个 get()函数，可以从流中读取下一个字符，但该字符仍留在流中，因此可以再次读取。peek()把从流中读取的字符返回为 int_type 类型的值，与第一个 get()函数一样。

使用成员函数 unget()可以把从流中读取的最后一个字符返回给流。这个函数返回 istream 对象的引用。在解析从流中读取的输入时，它通常和读取一个字符的 get()函数一起使用。例如，使用一个函数跳过文件输入流中的非数字字符，定位流到下一个数字上：

```
void skipnondigits(std::ifstream& in) {
  int_type ch=0;
  while((ch=in.get()) != EOF)          //Read while not EOF
   if(isdigit(ch)) {                   //If a digit is read…
     in.unget();                       //…put back the digit…
     return;                           //…and return
   }
}
```

putback()成员函数与 unget()的作用类似，但 putback()函数把要返回给流的字符作为一个参数。在上面的例子中，去掉调用 unget()函数的语句，代之以下面的语句：

```
in.putback(ch);                        //…put back the digit…
```

指定为 putback()函数参数的字符必须与最后一个读取的字符相同，否则结果就是不确定的。这两个函数都返回流的一个引用。

另外两个 get()函数把一组字符读取为一个非空字符串：

- istream& get(char* pArray, streamsize n);

这个函数从流中读取至多 n - 1 个字符，把它们存储在数组 pArray 中，末尾处添加一个空字符终止符，使总字符数达到 n 个。读取字符时，如果遇到换行符或到达文件末尾就停止读取，否则就读取 n - 1 个字符并存储起来。如果遇到换行符，换行符就不会存储到数组中的，但在所读取的字符序列的最后，总是要追加一个空字节。这个函数的作用是读取一整行文本，但不存储标志行尾的'\n'字符，而是将该字符串与一个空字符终止符一起存储。它还把'\n'字符当作

流中要读取的下一个字符。由于在数组中存储了总共 n 个字符，pArray 所指向的数组应至少有 n 个元素。如果没有存储字符，就设置 ios::failbit。第二个参数的类型 streamsize 是一个带符号的整数类型，它是作为实现定义的，通常定义为 long。

- istream& get(char* pArray, streamsize n, char delim);

这个函数与上一个函数类似，区别是这个函数还可以指定分隔符 delim，它用于代替换行符，结束输入过程。如果找到了分隔符，它不会存储在数组中，而是留在流中。

另外，还有两个 getline()函数成员，它们几乎等价于读取一行文本的两个 get()函数：

```
istream& getline(char* pArray, streamsize n);
istream& getline(char* pArray, streamsize n, char delim);
```

getline()和对应的 get()函数之间的区别是，getline()从输入流中删除了分隔符，下一个要读取的字符是分隔符后面的字符。

在使用上述未格式化的输入函数时，可以通过调用成员函数 gcount()，确定要从流中读取的字符数。该函数返回上一个未格式化输入操作读取的字符数，其类型为 streamsize。

使用 read()成员函数还可以从流中读取指定个数的字符(假定有这么多可读的字符)：

```
istream& read(char* pArray, streamsize n);
```

这个函数把任意类型的 n 个字符读入 pArray，包括换行符和空字符。如果还未读取 n 个字符就到达了文件末尾，就在输入对象中设置 ios::failbit。

在希望流中有 n 个可用字符时，一般使用 read()函数。如果流中没有这么多字符，就应使用另一个成员函数 readsome()。该函数的操作类似于 read()函数，但返回所读取的字符个数：

```
streamsize readsome(char* pstr, streamsize n);
```

如果流中可用的字符数少于 n，该函数就设置标志 ios::eofbit。

使用下面的函数，可以跳过输入流中的若干个字符：

```
istream& ignore(streamsize n, int_type delim);
```

该函数从流中至多读取 n 个字符并舍弃。如果读取了 delim 字符或读取了 n 个字符，就结束输入过程。参数 n 和 delim 的默认值分别是 1 和 EOF。所以，下面的语句可以跳过一个字符：

```
inFile.ignore();                    //Skip one character
```

要跳过 20 个字符，可以使用下面的语句：

```
inFile.ignore(20);                  //Skip 20 characters up to the end of the file
```

如果到达文件末尾，就会停止输入过程，但下面的语句会跳过当前行中的 20 个字符：

```
inFile.ignore(20, '\n');        //Skip 20 characters up to the end of the current
                                //line
```

19.4.2 未格式化的流输出函数

与输入函数不同，流的未格式化输出只有两个可用函数：put()和 write()。put()函数的形式

如下：

```
ostream& put(char ch);
```

这个函数把一个字符 ch 写入流中，并返回流对象的一个引用。要给流写入一组字符，应使用 write()函数，其形式如下：

```
ostream& write(const char* pArray, streamsize n);
```

这个函数把数组 pArray 中的 n 个字符写入流中。可以写入任何类型的字符，包括空字符。

一般情况下，输出到流中的内容要缓存起来，有时无论流缓存中的内容是否已满，都要写入到流中。此时，可以调用 ostream 的 flush()成员函数。这个函数把流缓存中的内容写入流中，并返回流对象的一个引用。

19.5 流输入输出中的错误

所有的流类都把流的状态存储在 3 个标志中，这 3 个标志记录了流的不同错误。它们在基类 ios 中定义为 iostate 类型的位掩码。这 3 个标志的含义如表 19-5 所示。

表 19-5 流状态标志

标　志	含　义
ios::badbit	当流不能继续使用时设置该标志，例如发生了 I/O 错误。这是不可恢复的
ios::eofbit	当到达文件末尾时设置该标志
ios::failbit	如果输入操作没有读取希望的字符，或输出操作不能成功地写入字符，就设置该标志。这一般是因为出现了转换或格式化的错误。设置了这个位掩码后，后续的操作就会失败，但该情形是可以恢复的

如果读取了流中的 EOF，流状态就是 ios::eofbit。如果在读取流时发生了一个严重的错误，没能从中读取字符，就设置 ios::badbit 和 ios::failbit 标志，因此流状态应是 ios::badbit | ios::failbit 的结果。

下面测试流的状态。在 if 语句或循环条件中使用流对象，此时该流对象会调用重载的 void*() 运算符。例如：

```
while(inFile) {
  //Read from inFile…
}
```

前面没有讨论重载的 operator void*()，因为它是相当复杂的。该运算符函数没有指定的返回类型，因为它是隐含的，当然就是 void*。流对象的 operator void*()函数返回一个指针，该指针用作对象状态的布尔测试。

由于对象用作 if 测试表达式，因此会自动调用 inFile 的 operator void*()成员。如果设置了 failbit 或 badbit，该函数就返回空指针，否则返回非空指针。注意所返回的非空指针并不会解除引用。可以使用这个循环读取到文件末尾，因为在到达文件末尾时，读取操作会设置 failbit。

如前所述，也可以使用重载的！运算符来测试流的状态，如果设置了 failbit 或 badbit，该运算符就会返回 true。

流类继承的函数成员可以用于测试对象的标志，它们分别返回 bool 类型的值。如表 19-6 所示。

表 19-6　测试流状态标志的函数

函　　数	操　　作
bad()	如果在流对象中设置了 badbit，就返回 true
eof()	如果在流对象中设置了 eofbit，就返回 true
fail()	如果在流对象中设置了 failbit 或 badbit，就返回 true
good()	如果在流对象中没有设置任何位掩码，就返回 true

一旦设置了标志，该标志就会一直保持下去，除非对它重新进行了设置。有时需要重新设置标志——例如到达文件末尾，作为停止读取文件流的一种方式，以便以后再次读取该文件流。调用对象的 clear()成员函数就可以重新设置这三个标志。实际上，clear()接受一个 iostate 类型的参数，它是对这三个标志进行按位或操作的结果，而其默认值是 0(实际上，其默认值是 ios::goodbit，它定义为 0)。一旦到达了输入流 inFile 的文件末尾，就可以用下面的语句重新设置流状态：

```
inFile.clear();
```

clear()函数的返回类型是 void。

I/O 错误和异常

在流输入和输出操作中发生错误时，就可以抛出异常。流错误的异常类型是 ios::failure，如前所述，这是继承自 ios_base 的 ios 类的一个嵌套类。掩码是流对象的一个成员，它确定了当设置某个流状态标志时是否抛出异常。把该掩码传送给流对象的 exceptions()成员，就可以设置这个掩码，所设置的位掩码指定要对哪个标志抛出异常。例如，如果要在为 inFile 流设置任何标志位时都抛出异常，就可以使用下面的语句：

```
inFile.exceptions(ios::badbit | ios::eofbit | ios::failbit);
```

只要出现了错误，甚至在到达了文件末尾时，都会抛出 ios::failure 类型的异常。

一般情况下，最好以前面讨论的方式测试错误标志，而不是使用异常来处理 I/O 错误，至少对 eofbit 和 failbit 标志要测试错误标志。在大多数情况下，都要处理 eofbit 和 failbit 标志，因为这是处理流输入输出的正常过程。在大多数开发环境下，默认操作是不为流错误抛出异常。调用不带参数的 exceptions()函数版本，可以检查是否抛出了异常，返回一个 iostate 类型的值。返回的值表示抛出异常后是否设置了错误标志。下面的语句测试设置了给定的标志是否会抛出异常：

```
iostate willthrow=inFile.exceptions();
if(willthrow & std::ios::badbit)
```

```
std::cout<<"setting badbit will throw an exception";
```

对 ios::badbit 和 willthrow 执行逻辑与的结果是 0，除非 ios::badbit 在 willthrow 中设置为 1。

19.6 使用二进制模式流操作

在一些情况下，使用文本模式并不合适或不方便，而且有时会增加困难。在一些系统中，可以把换行符转换为两个字符，但在其他一些系统中则不能转换，这会使在这两类系统上运行的程序进行的相对搜索操作不可靠。而使用二进制模式，就可以避免这种情况，使流操作更简单。前面介绍过如何在二进制模式下打开流；只需指定适当的打开模式标志即可。下面举一个例子。

程序示例 19.4——在二进制模式下复制文件

使用 get()和 put()函数可以复制任何文件，这两个函数可读写单个字符：

```
// Program 19.4 Copying files   File: prog19_04.cpp
#include <iostream>                                    // For standard streams
#include <cctype>                                      // For character functions
#include <fstream>                                     // For file streams
#include <string>                                      // For strings
#include <stdexcept>                                   // For standard exceptions
using std::cout;
using std::cin;
using std::endl;
using std::string;
using std::ios;
using std::invalid_argument;

int main(int argc, char* argv[]) {
   try {
      // Verify correct number of arguments
      if(argc != 3)
        throw invalid_argument("Input and output file names required.");

      const string source = argr[1];
      const string target = argv[2];

      // Check for output file identical to input file
      if(source == target)
         throw invalid_argument(string("Cannot copy ") +
                                string(source)  +
                                string(" to itself."));

      std::ifstream in(source.c_str(), ios::in | ios::binary);
       if(!in)                                       // Stream object OK?
         throw ios::failure(string("Input file ") +
                            string(source) +
                            string(" not found"));

      // Check if output file exists
```

```
        std::ifstream temp(target.c_str(), ios::in | ios::binary);
        char ch = 0;                          // Stores a character
        if(temp) {                            // If the file stream object is ok
                                              // then the output file exists
           temp.close();                      // Close the stream
           cout << endl
              << target << "exists, do you want to overwrite it?  (y or n):  ";
           ch = cin.get();
           if(std::toupper(ch) ! = 'Y')
               return 0;
        }

        // Create output file stream
        std::ofstream out(target.c_str(), ios::out | ios::binary | ios::trunc);

        // Copy the file
        while(in.get(ch))
           out.put(ch);

        if(in.eof())
           cout << endl << source << "copied to" << target <<" successfully.";
        else
           cout << endl << "Error copying file";
        return 0;
      }
      catch(std::exception& ex) {
        cout << endl << typeid(ex).name() << ": " << ex.what();
        return 1;
      }
    }
```

这个程序需要把输入文件名和输出文件名作为命令行参数，用它复制程序的可执行模块，则其输出如下：

```
FileCopy.exe copied to FullCopy.exe successfully.
```

例子的说明

数组 argv 有 argc 个元素，第一个元素包含程序名。因此，数组 argv 应有三个元素，适于分别包含程序名和两个文件名，这里首先验证一下，如下述语句所示：

```
if(argc != 3)
   throw invalid_argument("Input and output file names required.");

const string source = argv[1];
const string target = argv[2];
```

如果没有命令行参数，就抛出 invalid_argument 类型的标准异常，main()中最后的 catch 块会捕获这个异常。检查完参数后，把它们赋予一对 string 对象，使后续的代码更容易处理和辨识。

我们不想把文件复制给它自己，所以要检查文件，确保原文件名和复制后的文件名不同。如果这两个文件名相同，就不再继续，而是抛出另一个异常：

```
// Check for output file identical to input file
if(source == target)
  throw invalid_argument(string("Cannot copy ")  +
                         string(source) +
                         string(" to itself."));
```

命令行参数的有效性检查一旦通过，就准备创建对应于输入文件的文件流对象：

```
std::ifstream in(source.c_str(), ios::in | ios::binary);
```

这个语句使用了 string::c_str()函数，该函数返回包含在 source 对象中的非空字符串。文件打开为一个二进制输入文件，这是由构造函数的第二个参数指定的。在使用它之前，必须验证流对象是否正确创建：

```
if(!in)                                    // Stream object OK?
  throw ios::failure(string("Input file ")  +
                     string(source)  +
                     string(" not found"));
```

这个语句使用了流对象的重载！运算符。如果文件没有找到并打开，istream 构造函数就设置 failbit。

接着，看看输出文件是否存在。如果存在，就验证它是否要被改写：

```
std::ifstream temp(target.c_str(),  ios::in  |  ios::binary);
char ch = 0;                              // Stores a character
if(temp) {                                // If the file stream object is ok
                                          // then the output file exists
   temp.close();                          // Close the stream
   cout << endl
        << target << "exists,  do you want to overwrite it?  (y or n):  ";
   ch = cin.get();
   if(std::toupper(ch)  !=  'Y')
     return 0;
}
```

在为文件创建 ifstream 对象时，该文件必须存在。如果文件没有找到并打开，构造函数就设置 failbit，因此在 if 条件中使用流对象会隐式地调用 operator void*()成员，该函数会返回空。因此，if 块中的代码仅在文件存在时执行。一旦建立这个对象，就可以关闭文件，因为这里不打算读取它。只是提供了不改写它的选项。

然后创建输出文件流对象：

```
std::ofstream out(target.c_str(), ios::out | ios::binary | ios::trunc);
```

现在可以进行复制，这是在一个非常简单的循环中进行的：

```
while(in.get(ch))
  out.put(ch);
```

在循环条件中，使用 get()函数读取 in 中的字符。在到达文件末尾时，该函数返回 EOF，停止循环。在每次迭代时，存储在 ch 中的字符都会写入输出文件流 out。

最后，验证的确到达了文件末尾，因此确保正确复制了整个文件：

```
if(in.eof())
  cout<<endl<<source<<"copied to "<<target<<"successfully.";
else
  cout<<endl<<"Error copying file";
```

如果在文件流对象 in 中设置了 eofbit，eof()函数就返回 true。

以二进制模式写入数值数据

在使用二进制模式时，使用未格式化的流输入输出操作常常比使用插入和提取运算符更方便。未格式化的流输入输出操作可以把数值写为二进制值，不需要转换为数值的文本字符串表示。这样可以避免在把二进制浮点数值转换为十进制数值时可能出现的错误，而且在文件中占用的空间也比较少。另外，不需要把空白写入文件，以隔开各个数值，这也会使文件更短小，读取起来也就更快。只要知道写入了什么类型的数据，就可以读取出已写入的内容。

但是要注意，只有在同一类型的计算机上进行这些读写操作，才有这些优点。不同机器生成的二进制数据有许多不同。字符的表示方法也千差万别：IBM 大型计算机使用 EBCDIC，而 PC 使用 ASCII。同样，二进制浮点数值的表示方法也有区别，甚至二进制整数在不同的机器体系中存储的方式也不同。

标准库中没有以二进制模式读写数据值的流函数，但我们可以自己编写。使用 read()和 write()函数就可以实现一组把任意数值类型写为二进制数据的函数。下面用一个例子来说明。

假定要把 double 类型的数值写入一个文件。为此，实现自己的 write()函数，如下所示：

```
void write(std::ostream& out, double value) {
  out.write(reinterpret_cast<char*>(&value), sizeof(double));
}
```

要把浮点数值写入文件，只需写入该值在内存中占用的字节序列。如图 19-4 所示，假定类型 double 占用 8 个字节。

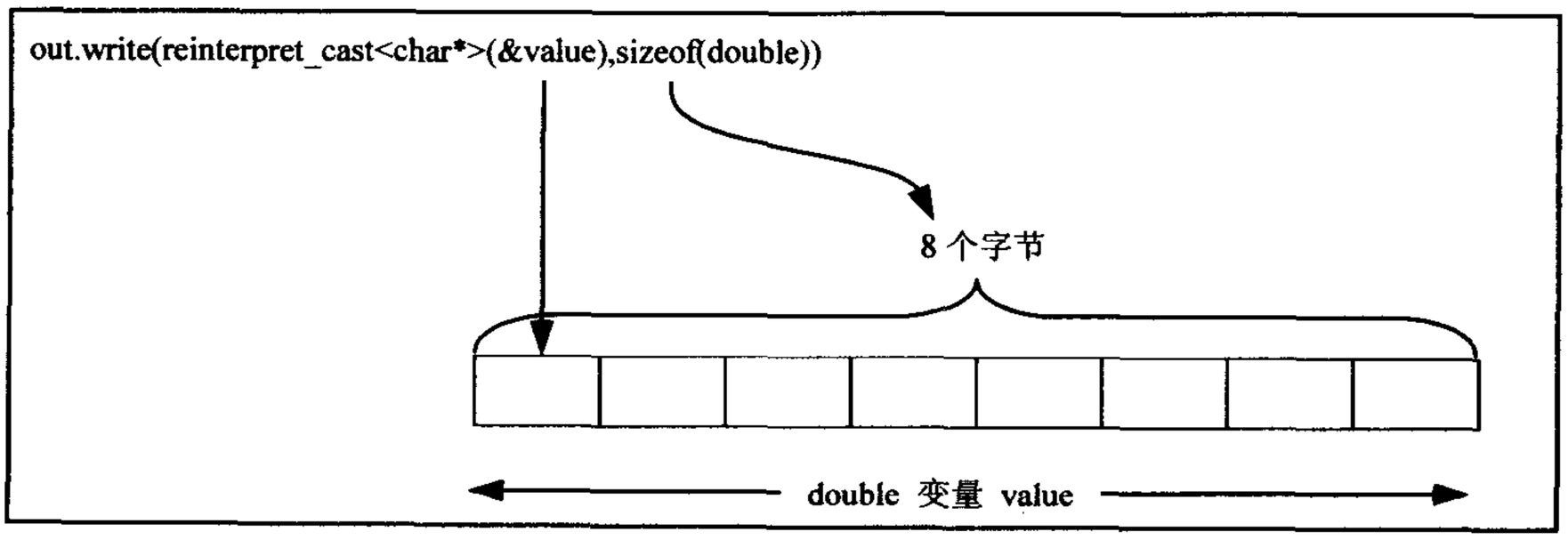


图 19-4　以二进制模式把 double 值写入流

把 double 值中第一个字节的地址强制转换为 char*，再把它传送给流对象 out 的 write()成员函数。reinterpret_cast<>()运算符仅改变指针的解释，不转换它指向的值。sizeof()运算符提供写入类型 double 所需要的字节数，把这个值传送给流对象 out 的 write()成员函数，作为要写入的字节数。

显然，任何类型的数值都可以用这种方式写入文件，因此考虑定义一个模板，该模板可以

根据需要生成这些函数。但是，这个模板还应为类类型创建函数，也就是创建处理任意类对象的模式。可惜，这是不可行的。任何包含指针数据成员的类对象在从文件中读取时都是无效的，因为指针成员中的地址总是无效的。为了不违背封装原则，只能使用类提供的复制功能。因此，必须为每个支持的数值类型编写一个重载函数，避免出现误用函数模板的情况。

回到前面的主题，从文件中读取二进制值的函数可以实现为：

```
void read(std::istream& in, double& value) {
  in.read(reinterpret_cast<char*>(&value), sizeof(double));
}
```

这个函数把 double 变量在内存中占用的字节数读入 double 变量 value 占用的内存位置。第二个参数必须是一个引用，因为要在这个位置存储从文件中读取的数据。

注意这里把 istream 对象的引用用作第一个参数类型，而没有使用 ifstream 对象的引用。同样，write()函数把 ostream&作为流参数的类型。这里仅对文件流使用这两个函数，但把文件流的引用作为参数有一个显著的缺点。fstream 类派生于 iostream，因此 istream 和 ostream 是其间接基类。它不派生于 ifstream 或 ofstream。把 istream&和 ostream&作为参数的类型，可以确保这两个函数处理的是 fstream，read()函数处理 ifstream，write()函数处理 ofstream。

实现读写基本类型的数值数组的函数也不是很困难，但需要给每个函数传送数组的长度，因为不能推断出传送给函数的数组长度。

对程序示例 19.3 中处理二进制文件的函数进行修改，可以得到该函数的其他版本。

程序示例 19.5——把二进制数值数据写入文件

原程序中惟一需要修改的地方是，把文件流对象创建为二进制流，用自己的读写函数替代格式化的 I/O 操作：

```
// Program 19.5 Reading and writing a binary File: prog19_05.cpp
#include <fstream>
#include <iostream>
#include <iomanip>
#include <cmath>
#include <string>
using std::ios;
using std::cout;
using std::cin;
using std::endl;
using std::string;
// Find the prime after lastprime
long nextprime(long lastprime,  const char* filename);

void write(std::ostream& out, long value);          // Write binary long value
void read(std::istream& in, long& value);           // Read binary long value

int main() {
   const char* filename = "C:\\JunkData\\primes.bin";
   int nprimes = 0;                                 // Number of primes required
   int count = 0;                                   // Count of primes found
   long lastprime = 0;                              // Last prime found
```

```
// Get number of primes required
int tries = 0;                                    // Number of input tries
cout << "How many primes would you like  (at least 3)?:  ";
do
{
   if(tries)
     cout << endl << "You must request at least 3,  try again:  ";
   cin >> nprimes;

   if(++tries == 5) {                             // Five tries is generous
     cout << endl << " I give up!"  << endl;
     return 1;
   }
} while(nprimes < 3);

std::ifstream inFile;                             // Create input file stream object
inFile.open(filename, ios::in | ios::binary) ;

cout << endl;
if(!inFile.fail()) {
    do {
       read(inFile, lastprime);
        cout << (count++ % 5 == 0 ? "\n" : "" )  << std::setw(10) << lastprime;
    } while(count < nprimes && !inFile.eof());
    inFile.close();
}
inFile.clear();                                   // Clear any errors

try {
    std::ofstream outFile;
    if(count == 0) {
      // Open file to create it
      outFile.open(filename, ios::out | ios::binary | ios::app);
      if(!outFile.is_open())
        throw ios::failure(string("Error opening output file ")  +
                           string(filename) +
                           string(" in main()"));
      write(outFile, 2L);                         // Write 2 as binary long
      write(outFile, 3L);                         // Write 3 as binary long
      write(outFile, 5L);                         // Write 5 as binary long
      outFile.close();
      cout << std::setw(10)  << 2 << std::setw(10)  << 3 << std::setw(10) << 5;
      lastprime = 5;
      count  =  3;
}

while(count < nprimes) {
  lastprime = nextprime(lastprime,  filename);

  // Open file to append data
  outFile.open(filename, ios::out | ios::binary | ios::app);
  if(!outFile.is_open())
```

```
        throw ios::failure(string("Error opening output file ")  +
                           string(filename) +
                           string(" in main()"));
      write(outFile, lastprime);                    // Write prime aa binary
      outFile.close();
     cout <<  (count++ % 5 == 0 ? "\n"  :  ""  )  << std::setw(10)  << lastprime;
    }
    cout << endl;
    return 0;
}
    catch(std::exception& ex) {
        cout << endl << typeid(ex).name()  << ":  "<< ex.what()  << endl;
        return 1;
    }
}
```

nextprime()函数需要修改的地方不多。

```
long nextprime(long lastprime, const char* filename) {
  bool isprime = false;                         // Indicator that we have a prime
  long aprime = 0;                              // Stores primes from the file
  std::ifstream inFile;                         // Local input stream object

  // Find the next prime
  for( ; ; ) {
  lastprime += 2;                               // Next value for checking
  long limit = static_cast<long>( std::sqrt(static_cast<double>
                                                (lastprime)));
  // Try dividing the candidate by all the primes up to limit
inFile.open(filename, ios::in | ios::binary);       // Open the primes file
if(!inFile.is_open())
  throw ios::failure(string("Error opening input file ")  +
                string(filename)  +
                string("  in nextprime()"));
do {
  read(inFile, aprime);                         // Read prime as binary
} while(aprime <= limit && !inFile.eof() &&
        (isprime = lastprime % aprime > 0));

inFile.close();
if(isprime)                                     // We got one...
    return lastprime;                           //  ...so return it
  }
 }
```

当然，必须把 read()和 write()函数的定义添加到源文件中：

```
// Read a long value from a file as binary
void read(std::istream& in, long& value) {
  in.read(reinterpret_cast<char*>(&value), sizeof(value));
}

// Write a long value to a file as binary
```

```
void write(std::ostream& out, long value) {
   out.write(reinterpret_cast<char*>(&value), sizeof(value));
}
```

这个程序运行后，应输出指定个数的质数。

例子的说明

这里仅讨论文件操作的部分，因为其他代码前面已经讨论过了。首先，是二进制 read()和 write()函数的原型：

```
void write(std::ostream& out, long value);          //write binary long value
void read(std::istream& in, long& value);           //Read binary long value
```

文件的扩展名与前面创建的 primes.txt 文件的扩展名不同：

```
const char* filename="C:\\JunkData\\primes.bin";
```

如果把文本文件读取为二进制文件，程序中就会出现大量的混乱情况。
在打开输入文件时，把打开模式指定为二进制：

```
inFile.open(filename, ios::in | ios::binary);     //Open the file as an input stream
```

构造函数的第二个参数现在是显式的，而不是假定在文本模式下输入文件的默认值。如果文件不存在，就创建它，并把文件打开为二进制输出模式：

```
outFile.open(filename, ios::out | ios::binary | ios::app);
```

接着，把前三个质数写入新文件，作为二进制值：

```
write(outFile, 2L);              //write 2 as binary long
write(outFile, 3L);              //write 3 as binary long
write(outFile, 5L);              //write 5 as binary long
```

在给文件添加质数时，还要把文件打开为二进制输出模式：

```
//Open file to append data
outFile.open(filename, ios::out | ios::binary | ios::app);
```

当然，写入文件现在应使用自己的 write()函数，写入字节：

```
write(outFile, lastprime);              //Write prime as binary
```

在 nextprime()函数定义中，文件打开为二进制输入模式，我们使用自己的 read()函数版本读取每一个质数值。如果类型 long 占用 4 个字节，则文件中的每个值无论是多少，都仅占用 4 个字符位置(字节)，且各个值之间没有空格。因此，大于 999 的数占用的文件空间比较少，如果在文件中写入浮点数值，文件的尺寸减小得就更多。二进制文件一般比文本文件更紧凑，读取速度更快，也没有因格式化而产生的额外系统开销。

19.7 对流的读写操作

可以打开流进行输入输出操作。对于一般的流，iostream 类实现了输入输出功能，但我们常常使用 fstream 类对象，因为它们专门用于文件的输入输出操作。如本章前面所述，fstream 继承自 iostream，而 iostream 又继承自 istream 和 ostream，所以前面讨论过的所有输入输出函数都可用于 fstream 对象。

在声明 fstream 对象时，可以把文件名作为一个参数，这与 ifstream 和 ofstream 对象一样。例如：

```
const char* filename="C:\\JunkData\\primes.txt";
std::fstream bothways(filename);
```

默认的打开模式是 ios::in | ios::out，与其他流一样，fstream 在默认情况下也处于文本模式下。如果要把它指定为二进制流，只需使用第二个参数，方法与前面的一样，例如：

```
std::fstream bothways(filename, std::ios::in|std::ios::out|
                      std::ios::binary);
```

这个语句在二进制模式下打开文件，进行输入和输出操作。如果文件因某种原因打不开，就设置 ios::failbit。注意 fstream 对象不能使用 ios::app，但可以使用 ios::trunc，舍弃文件中以前的内容。实际上，不能联合使用 ios::app 和 ios::in，对 ifstream 对象也不能使用该联合。这不是没有道理的，因为 ios::app 表示要在文件的末尾写入数据，这对于读操作来说没有意义。

默认的构造函数创建了一个没有关联文件的流对象。接着使用 fstream 对象的 open()成员函数打开某个物理文件。例如：

```
std::fstream inout;
inout.open(filename, std:: ios::in | std::ios::out | std::ios::binary |
           std::ios::trunc);
```

第一个语句创建了一个没有关联文件的流对象 fstream。第二个语句打开 filename 指定的文件，用于在二进制模式下进行输入和输出操作，并舍弃已有的文件内容。如果省略指定打开文件模式的第二个参数，其默认值与构造函数相同：ios::in | ios::out。

如果要读写文件，根据定义应可以访问文件中的随机位置，下面看看如何实现这个访问操作。

对流的随机访问

在 fstream 对象中对文件进行随机访问时，可以使用读取 ifstream 和写入 ofstream(但可能性不大)的技术。一旦打开一个用于输入输出的文件流，就可以在流中的任意位置进行读写。但是，我们需要知道目前在流中的什么位置，下一步要去什么位置。如果流是使用插入运算符进行格式化写操作，实现随机访问就相当困难。

困难就在于：除非总是设置字段的宽度，否则根本不知道流中写入了多少字节。如果给流中写入一个整数，则写入的字节数取决于十进制数字的个数，以及是否包含符号。宽度说明仅设置了最小字节数，除非把它设置为一个字符数至少和要输出的最大宽度一样多的值，否则仍

旧无法确定字节数。文本模式则更增加了一层复杂性，因为对于给定的输出，一些系统中写入的字节数要比另一些系统多。所以，使用未格式化的读写操作进行随机访问，在二进制模式下比较简单、安全。下面看看如何随机访问二进制文件。

对二进制流的随机访问

修改把质数写入二进制文件的程序示例 19.5，请求输出某个特定的质数，例如第 25 个质数或第 432 个质数。如果该质数在文件中，程序就可以找到并显示它。否则，就应计算出所请求的质数，并把它添加到文件中。

这次，文件有点不同。如果跟踪文件中的质数个数，就很容易确定所请求的质数是否在文件中。对此，最简单的方式是把指定个数的质数作为最后一个数据项写入文件。

文件的输入是所请求的质数的序数值：3 表示第三个质数，101 表示第 101 个质数，等等。该程序的基本逻辑如图 19-5 所示。

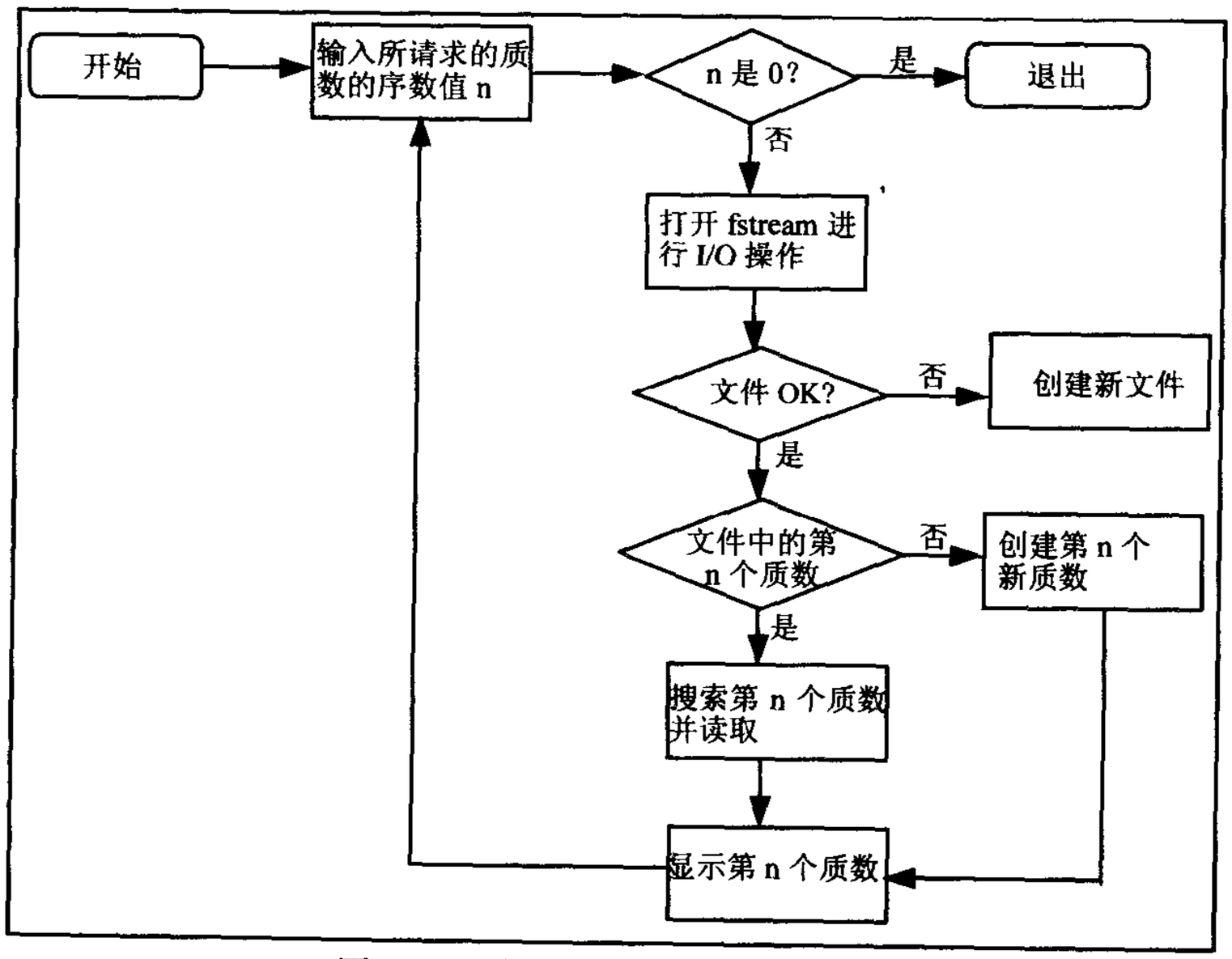


图 19-5　读写质数文件的程序逻辑

下面实现这个逻辑，并把代码组合在一起。在设置与用户通信的对话框之后，就打开文件，如果文件不存在，就创建它。这个文件使用 fstream 对象。因为 fstream 对象可以读写，所以一旦在程序的开头打开它，就可以在任何地方使用它进行输入或输出：

```
int main() {
  const char* filename="C:\\JunkData\\nuprimes.bin";
  std::fstream primes;                                    //Create file stream object
  primes.open(filename, ios::in | ios::out | ios: :binary);     //Open the file
  long count=0;                                           //Count of primes found

  //Rest of the code…
}
```

对 ios 添加一个 using 声明，以便使用未限定的名称。除了建立一个 long 变量来存储要查

找的质数个数这个简单而重要的任务之外，这段代码还以二进制模式打开文件，用于进行读写操作。由于打开模式指定为可以输入和输出，如果文件不存在，这就会失败，因此必须检查流是否正常：

```
int main() {
  try {
    const char* filename = "C:\\JunkData\\nuprimes.bin";
    std::fstream primes;                          // Create file stream object
    primes.open(filename,  ios::in |ios::out|ios::binary);  // Open the file
    long count = 0;                               // Count of primes found

  if(!primes) {
    primes.clear();
    cout << endl << "File doesn't exist - creating..." << endl;
    primes.open(filename, ios::out | ios::binary);  // Create binary file

    if(!primes)
      throw ios::failure(string("Failed to create output file ") +
                         string(filename) +
                         string(" in main()"));
    write(primes, 2L);                                  // Write 2 as binary long
    write(primes, 3L);                                  // Write 3 as binary long
    write(primes, 5L);                                  // Write 5 as binary long
    write(primes, count = 3L);                          // Write primo count
    primes. close();
      primes.open(filename, ios::in | ios::out | ios::binary);
    }

     // Rest of the code...
  }
  catch(std::exception& ex) {
    cout << endl << typeid(ex).name() << ": "<< ex.what() << endl;
    return 1;
  }
}
```

如果文件没有成功打开，就清除 ios 错误标志，并尝试再次打开文件，但这次仅以二进制输出模式打开它。这会创建一个新的文件，但仍要进行检查，以确保文件已打开。如果设置了错误标志，就抛出 ios::failure 异常，该异常包含解释错误的字符串。

通过这个检查后，就在文件中写入前三个质数和质数的个数。接着关闭流，以便以后重新打开它，进行输入输出操作。下面准备开始搜索所需质数的过程。

首先，必须读取所需要的质数个数。如果把整个过程放在一个无限 for 循环中，就可以把查找指定质数的过程重复需要的次数，并使用 0 或负值作为停止执行程序的信号：

```
int main() {
  try {
    // Code to open the file...

    long nprime = 0;                          // Sequence no. of prime required
    long lastprime = 0;                       // Last prime found
```

```
    for( ; ; ) {
       cout << "Which prime (e.g. enter 15 for the 15th prime, zero to end)?: ";
        cin >> nprime;
        if(nprime <= 0)                    // Zero or negative?
            return 0;                      //  ...yes, so we are done

        // Rest of the code for finding the prime to follow...
    }
    cout << endl;
    return 0;
}
    }
    catch(std::exception& ex) {
       cout << endl << typeid(ex).name()  << ": "<< ex.what()  << endl;
       return 1;
    }
}
```

其他代码放在 for 循环中已有代码的后面。要确定所请求的质数是否在文件中，应获取文件中的质数个数：

```
//Go to start of last item
primes.seekg(-static_cast<int>(sizeof(long)), ios::end);
read(primes, count);                       //Read the last item
```

count 是文件中的最后一个数据项。把它放在开头似乎更简单，但这样会使一些函数较难实现——必须相对于文件末尾进行搜索。count 值位于距离文件末尾 sizeof(long)个字节的位置，但 sizeof()返回 size_t 类型的值，该类型通常定义为无符号的整数，需要把它强制转换为 int，之后才能改变其符号。在搜索到该位置后，使用程序示例 19.5 中读取二进制 long 值的 read()函数版本来读取 count。

接着，检查 nprime 是否小于这个值，如果是，就在文件中的适当位置读取质数：

```
if(nprime <= count) {
  cout<<endl<<"Prime in file";
  primes.seekg((nprime-1) * sizeof(long), ios::beg);     //Seek to position of nth
  read(primes, lastprime);                               //...and read it
}
```

可以直接到文件中相对于开头的特定位置获取某个质数。第一个质数相对于 ios::beg 的偏移量是 0 个字节，第二个质数相对于 ios::beg 的偏移量是 sizeof(long)个字节，第三个质数相对于 ios::beg 的偏移量是 2*sizeof(long)个字节，依此类推。因此要从文件的开头查找(nprime - 1) *sizeof(long)个字节，到达这个位置后，就使用二进制的 read()函数读取该质数。

如果 nprime 大于 count，就必须计算新的质数，并把它们添加到文件中，直到文件中的质数总数为 nprime 为止。这是在 if 块的 else 部分进行的：

```
if(nprime <= count) {
   cout << endl << "Prime in file";
   primes.seekg((nprime - 1)* sizeof(long), ios::beg); // Seek to position of nth
   read(primes, lastprime);                            //  ...and read it
}
else
```

```
    while(count < nprime) {
       lastprime = nextprime(primes);
       primes.seekp(-static_cast<int>(sizeof(long)), ios::end);   // Move to the end
       write(primes, lastprime);                                  // Write prime as binary
       write(primes, ++count);                                    // Write prime as binary
}
```

这段代码使用 nextprime()的修订版本，它接受一个参数：文件流。这个函数会计算并返回文件中最后一个质数后面的质数，稍后讨论其实现过程。在计算出了下一个质数后，就在文件末尾搜索 count 的位置，并用新的质数改写它。接着把递增后的 count 写入文件。注意这里使用 seekp()函数，而不是 seekg()函数，因为是写入文件。这是必须的，因为 fstream 有几个内部的、同步的缓存，用于输入和输出；seekg()使用其中一个缓存，而 seekp()使用另一个缓存。

for 循环的最后一步是显示所请求的质数，之后进行下一次的迭代。如下面的语句所示：

```
cout << endl << "The" << nprime << " prime is " << lastprime<<endl;
```

如果仔细研究一下代码，就会发现这里也可以使用条件运算符，在输出中 nprime 值的后面添加"st"、"nd"、"rd"和"th"。

```
cout << endl << "The "
     << nprime << ((nprime%10 ==1) && (nprime != 11) ? "st" :
                   (nprime%10 ==2) && (nprime != 12) ? "nd" :
                   (nprime%10 ==3) && (nprime != 13) ? "rd" : "th")
     << " prime is " << lastprime<<endl;
```

nextprime()函数的实现代码如下：

```
long nextprime(std::fstream& primes) {
   bool isprime =false;                 // Indicator that we have a prime
   long aprime = 0;                     // Stores primes from the file
   long candidate = 0;                  // Value to be tested

   primes.seekg(-static_cast<int>(2*sizeof(long)), ios::end);  // Go to last
                                        // in file
   read(primes, candidate);             // ...and read it

   // Find the next prime
   for( ; ; ) {
      candidate += 2;                   // Next value for checking
      // Calculate upper limit for divisors
      long limit = std::sqrt(static_cast<double>(candidate));
      primes.seekg(0, ios::beg);  // Go to the start of the file

      // Try dividing the candidate by all the primes up to limit
      do {
         read(primes, aprime);     // Read prime as binary
      } while(aprime <= limit && (isprime = candidate % aprime > 0));

      if(isprime)                       // We got one...
        return candidate;               // ...and return the prime
   }
}
```

这里有趣的地方是文件访问部分，计算是必须的，这与前面一样。首先搜索文件中的最后一个质数在什么地方。每个数据项都占用了 sizeof(long)个字节，所以使用 seekg()函数从文件的末尾开始，回退两倍 sizeof(long)个字节。这主要是考虑到文件中质数的个数是存储在文件中的最后一个数据项，它也是 long 类型。接着使用自己的二进制 read()函数在 candidate 中存储最后一个质数。

要检查 candidate，应搜索文件的开头，即相对于 ios::beg 偏移 0 个字节，再从文件中连续读取质数作为除数。在循环条件中使用这些质数测试 candidate。limit 检查可以确保不会到达文件末尾，因此也就不会设置流的 ios 标志。

程序示例 19.6——文件的随机访问

把刚才讨论的代码都放在一起，main()的完整版本如下所示：

```
// Program 19.6 Reading and writing binary primes file File: prog19_06.cpp
#include <fstream>
#include <iostream>
#include <cmath>
#include <string>
using std::cout;
using std::cin;
using std::endl;
using std::string;
using std::ios;

long nextprime(std::fstream& primes);       // Find the prime after lastprime
void write(std::ostream& out, long value); // Write binary long value
void read(std::istream& in, long& value);  // Read binary long value

int main() {
  try {
    const char* filename = "C:\\JunkData\\nuprimes.bin";
    std::fstream primes;                          // Create file stream object
    primes.open(filename, ios::in | ios::out | ios::binary); // Open the file
    long count = 0;                               // Count of primes found

    if(!primes) {
      primes.clear();
      cout << endl << "File doesn't exist - creating..." << endl;
      primes.open(filename, ios::out | ios::binary);     // Create binary file

if(!primes)
  throw ios::failure(string("Failed to create output file ") +
                     string(filename) +
                     string(" in main()"));
write(primes, 2L);                              // Write 2 as binary long
write(primes, 3L);                              // Write 3 as binary long
write(primes, 5L);                              // Write 5 as binary long
write(primes, count=3L);                        // Write prime count
primes.close();
primes.open(filename, ios::in | ios::out | ios::binary);
```

```
}

long nprime = 0;                                  // Sequence no. of prime required
long lastprime = 0;                               // Last prime found
for( ; ; ) {
   cout << "Which prime (e.g. enter 15 for the 15th prime, zero to end)?:";
   cin >> nprime;
    if(nprime <= 0)                               // Zero or negative?
       return 0;                                  // ...yes, so we are done

   primes.seekg(-static_cast<int>(sizeof(long)), ios::end); // Go to last
    read(primes, count);                          // Read the last

    if(nprime <= count) {
      cout << endl << "Prime in file";
      primes.seekg((nprime - 1) * sizeof(long), ios::beg); // Seek to nth
      read(primes, lastprime);                    //  ...and read it
    }
    else
       while(count < nprime) {
          lastprime = nextprime(primes);
          primes.seekp(-static_cast<int>(sizeof(long)), los::end);   //Move
                                                                     // to end
          write(primes, lastprime);               // Write prime as binary
          write(primes, ++count);                 // Write prime as binary
       }
          cout << endl << "The "
               << nprime << ((nprime%10 == 1) && (nprime != 11) ? "st" :
                     (nprime%10 == 2) && (nprime != 12) ? "nd" :
                     (nptime%10 == 3) && (nprime != 13) ? "rd" : "th")
               << " prime is " << lastprime << endl;
       }
       cout << endl;
       return 0;
    }
    catch(std::exception& ex) {
       cout << endl << typeid(ex).name() << ": "<< ex.what() << endl;
       return 1;
    }
}
```

需要添加 nextprime()、read()和 write()函数的定义。这个程序的输出如下所示：

```
File doesn't exist - creating...
Which prime (e.g. enter 15 for the 15th prime, zero to end)?: 9

The 9th prime is 23
Which prime (e.g. enter 15 for the 15th prime, zero to end)?: 99

The 99th prime is 523
Which prime (e.g. enter 15 for the 15th prime, zero to end)?: 1001
```

```
The 1001 st prime is 7927
Which prime (e.g. enter 15 for the 15th prime, zero to end)?: 5000

The 5000th prime ie 48611
Which prime (e.g. enter 15 for the 15th prime, zero to end)?:3456

Prime in file
The 3456th prime is 32213
Which prime (e.g. enter 15 for the 15th prime, zero to end)?: 0
```

例子的说明

前面已论述了开发代码的所有细节。所有的输入和输出操作都对同一个文件流对象进行，直接搜索到需要的位置。由于使用的是在二进制模式下的未格式化的 I/O 操作，程序会突然终止，因为我们总是知道文件中的每个数据项占用多少字节。即使文件中有各种类型的数据，只要知道写入了什么类型的数据项，数据项的序数是多少，就可以确定该数据项在未格式化的二进制文件中的位置。

19.8 字符串流

有三个字符串流类可以把流与 string 对象连接起来，它们是 istringstream、ostringstream 和 stringstream，它们的基类分别是 istream、ostream 和 iostream。这些类的操作与文件流相同，但输入输出操作是针对 string 对象而言。

尽管它们可以使用从对应基类继承而来的所有输入输出函数，但字符串流主要与插入和提取运算符一起使用。原因是它们的主要应用是在内存中格式化数据或分析输出。例如，有一个应用程序，其输入的格式事先是未知的。在这种情况下，可以把数据作为一系列字符读入 string 对象，再使用流输入操作，以及与 string 对象关联在一起的 istringstream 对象，对数据进行格式化的读取操作。这样可以对输入读取任意多次，确定其格式。

假定下面的语句从 cin 中读取一行输入：

```
string buffer;
getline(cin, buffer);
```

把输入读到 buffer 中，再用下面的语句创建一个 istringstream 对象：

```
std::istringstream inStr(buffer);
```

现在可以通过流 inStr 读取 buffer，就像使用其他流一样，并利用转换功能把字符转换为二进制格式：

```
long value=0;
double data=0.0;
inStr>>value>>data;
```

可以使用 ostringstream 对象把数据格式化为一个字符串。例如，下面的语句创建一个 string 对象和一个输出字符串流：

```
string outBuffer;
```

```
std::ostringstream outStr(outBuffer);
```

通过 outStr，利用插入运算符通过 outStr 写入 outBuffer：

```
double number=2.5;
outStr<<"number="<<(number/2.0);
```

写入字符串流的结果是，outBuffer 包含"number=1.25"。字符串 outBuffer 会自动扩展，以容纳写入流中的字符，所以这是构建字符串或复杂输出消息的一种非常灵活的方式。

在任何情况下，字符串流构造函数的 string 参数都是一个引用，所以给 ostringstream 和 stringstream 对象执行写入操作会直接操作 string 对象。这也是每个字符串流类的默认构造函数。在使用它们时，字符串流对象会在内部包含一个 string 对象，使用 str()成员可以获得该对象的副本。例如：

```
std::ostringstream outStr;
double number=2.5;
outStr<<"number="<<(3*number/2);
string output=outStr.str();
```

执行这些语句后，output 就包含字符串"number=3.75"。

19.9 对象和流

前面介绍过如何在基本数据类型和流之间来回转换。但是，前面的章节讨论过面向对象编程的优点，那么如何把对象写入文件？这有一个问题。

对于 C++标准库而言，不同的开发环境会提供不同的 C++标准库。使用标准库并不是很方便，因为在本质上，类完全是可扩展的，按照定义，它是未知量。输入和输出操作对于每个类来说不会相同。除了有这个困难之外，一些 C++开发系统还提供了读写对象的一个框架，只是有时需要使用某个特殊的基类。但是，如果读者使用的开发系统没有提供读写对象的框架，也不必担心，可以自己设计这个框架。而且，这并不需要派生自己的流类。

理想情况下，应可以使用基本数据类型的提取和插入运算符来读写类对象。我们只需提供函数，为类类型重载这两个运算符即可。重载该运算符的难易程度取决于类的复杂程度。一旦知道如何对格式化的 I/O 重载运算符，就可以实现用于二进制模式的未格式化读写操作。下面先来看看如何为类对象实现格式化的 I/O。

19.9.1 重载类对象的插入运算符

下面实现 operator<<()运算符的一个重载版本，把给定类的对象写入流中，作为类的一个友元。以本书前面使用的 Box 类为例，将 Box 对象写入一个流。

首先，需要把函数声明为 Box 类的一个友元，以访问 Box 类的数据成员：

```
// Box.h
#ifndef BOX_H
#define BOX_H
#include <iostream>
```

```
class Box {
  public:
    Box(double lv = 1, double wv=1, double hv = 1); // Constructor

    virtual ~Box();                  // Virtual Destructor
    void showVolume() const;         // Show the volume of an object
    virtual double volume() const; // Calculate the volume of a Box object

    // Friend insertion operator
    friend std::ostream& operator<<(std::ostream& out, const Box& rBox);
  protected;
     double length;
     double width;
     double height;
};
#endif
```

由于 operator<<()函数返回流对象的一个引用，因此可以使用与基本数据类型相同的方式，把 Box 对象写入流中。这可以应用于文件流和 cout。对标准类型使用重载的插入运算符来实现这个函数：

```
std::ostream& operator<<( std::ostream& out, const Box& rBox) {
  return out<<' '<<rBox.length<<' '<<rBox.width<<' '<<rBox.height;
}
```

这个函数仅把 3 个数据成员写入流中，每个数据成员的前面加一个空格，返回传送给函数的流对象 out。如果运算符只需要把对象输出到 cout 中，就可以用更富有描述性的信息修饰输出，例如成员名。但是，如果要使用运算符把对象写入文件，这种格式化就会使对象的读取比较麻烦，所以最好使其简单一些。

如果类包含查询函数，用于检索数据成员的值(例如其他章节使用的 getLength()函数等)，operator<<()函数就不需要成为类的友元，使用 public 查询函数即可实现这一功能。下面看看如何处理得到的数据。

程序示例 19.7——把对象写入流

Box.h 包含了 operator<<()函数的友元声明，如前面所示。还需要在 Box.cpp 中定义该函数，如下所示：

```
// Box.cpp
#include <iostream>
#include "Box.h"
using std::cout;
using std::endl;

// Constructor
Box::Box(double lv, double wv, double hv) :
                                        length(lv),width(wv),height(hv){}

// Destructor
```

```
Box::~Box() {}
// Function to show the volume of an object
void Box::showVolume() const {
  cout << endl << "Box usable volume is" << volume();
}

// Function to calculate the volume of a Box object
double Box::volume() const {
   return length * width * height;
}

// Friend operator function for Box
std::ostream& operetor<<(std::ostream& out, const Box& rBox) {
   return out <<  '  '  << rBox.length << ' ' << rBox.width << ' ' << rBox.height;
}
```

在 main()中，创建了两个 Box 对象，并输出它们：

```
// Program 19.7 Writing Box object to cout    File: prog19_07.cpp
#include <iostream>
#include "Box.h"
using std::cout;
using std::endl;

int main() {
   Box bigBox(50, 60, 70);
   Box smallBox(2, 3, 4);
   cout << endl << "bigBox is" << bigBox;
   cout << endl << "smallBox is" << smallBox;
   cout << endl;
   return 0;
}
```

这个例子的输出如下所示：

```
bigBox is 50 60 70
smallBox is 2 3 4
```

例子的说明

在声明两个 Box 对象后，下面的语句输出了它们：

```
cout<<endl<<"bigBox is"<<bigBox;
cout<<endl<<"smallBox is"<< smallBox;
```

第一个语句等价于语句：

```
operator<<((cout.operator<<(endl)).operator<<("bigBox is"), bigBox);
```

换言之，我们调用了 operator<<()的友元版本，且把 cout.operator<<(endl)).operator<<("bigBox is ")作为第一个参数，bigBox 作为第二个参数。第一个参数的表达式又调用了 cout 的成员函数来输出换行符，其后是输出字符串的成员函数。后一个调用返回流对象 out，它又作为第一个参数传送给友元函数。第二个语句与第一个语句的操作类似。

19.9.2 重载类对象的提取运算符

从流中读取对象只需要给类实现 operator>>()函数，如果数据成员是 private 或 protected，该函数必须是类的友元函数。当然，如果执行 public 成员函数来设置数据成员的值，就不需要把 operator>>()函数实现为类的友元函数。但为了讨论的方便，使类尽可能简单。类对象的插入和提取运算符应保持一致，且成员值的读取顺序应与它们的写入顺序相同。Box 类的运算符实现如下所示：

```
std::istream& operator>>(std::istream& in, Box& rBox) {
  return in>>rBox.length>>rBox.width>>rBox.height;
}
```

由于修改了 rBox 的成员，所以第二个参数不能是 const。这是一个非常简单的实现代码，没有考虑输入错误。实际上，在读取需要的值之前，不仅要处理像“文件末尾”这样的错误，还要对值进行某些有效性检查，以确保得到有效的 Box 对象。但是，这个简单的版本足以说明运算符的机制。下面试验提取运算符和插入运算符。

程序示例 19.8——读写对象

当然，可以从键盘上读取一个 Box 对象，也可以将 Box 对象写入一个文件，再读取它。这需要在 Box.h 中给 Box 类定义添加 operator>>()函数的友元声明：

```
class Box {
  ...

    // Friend insertion and extraction operators
    friend std::ostream& operator<<(std::ostream& out,  const Box& rBox);
    friend std::istream& operator>>(std::istream& in, Box& rBox);

  ...
};
```

还必须在 Box.cpp 中添加 operator>>()函数的定义。使用新运算符函数的程序如下所示：

```
// Program 19.8 Writing Box objects to a file   File: prog19_08.cpp
#include <fstream>
#include <iostream>
#include <string>
using std::cout;
using std::endl;
using std::string;
using std::ios;

#include "Box.h"

int main() {
   try {
      const string filename = "C:\\JunkDeta\\boxes.txt";

      std::ofstream out(filename.c_str());
```

```
if(!out)
  throw(ios::failure(string("Failed to open output file ") + filename));

Box bigBox(50, 60, 70);
Box smallBox(2,3,4);

out << bigBox << smallBox;
out.close();

cout << endl << "Wrote two Box objects to the file:";
cout << endl << "bigBox is" << bigBox;
cout << endl << "smallBox is" << smallBox;
cout << endl;

std::ifstream in(filename.c_str());
if(!in)
   throw(ios::failure(string("Failed to open input file ") + filename));

cout << endl << "Reading objects from the file:";
Box newBox;                                    // Default Box object

in >> newBox;
cout << endl << "First Box read is" << newBox;
in >> newBox;
cout << endl << "Second Box reed is" << newBox;
coub << endl;
return 0;
   }
   catch(std::exception& ex) {
      cout << endl << typeid(ex).name() << ": "<< ex.what() << endl;
      return 0;
   }
}
```

这个程序的结果如下所示：

```
Wrote two Box objects to the file:
bigBox is  50 60 70
smallBox is  2 3 4

Reading objects from the file:
First Box read is  50 60 70
Second Box read is  2 3 4
```

例子的说明

这段代码没有什么可解释的，因为它的运行过程与读写基本类型完全相同。程序使用一个新文件来保存 Box 对象：

```
const string filename="C:\\JunkData\\Boxes.txt";
```

创建两个 Box 对象后，把它们写入文件，如下面的语句所示：

```
out<<bigBox<<smallBox;
```

这等价于语句：

```
operator<<( operator<<(out, bigBox), smallBox);
```

因为 ofstream 把 ostream 作为基类，所以该函数会把 ofstream 对象 out 返回为外层 operator<<()调用中第一个参数的类型 ostream&。写入文件后，就关闭文件，再使用 operator<<()函数和标准流 cout：

```
cout<<endl<<"bigBox is "<<bigBox;
cout<<endl<<" smallBox is "<< smallBox;
```

这个语句把两个 Box 对象输出到屏幕上，与前面的例子相同。为了证明这里没有使用什么新技术，可以从文件中把 Box 对象读入一个新的 Box 对象 newBox，如下面的语句所示：

```
in>>newBox;
```

这个语句利用新的提取运算符从文件流 in 中检索 Box 对象。它等价于语句：

```
operator>>(in, newBox);
```

分开使用输入语句并不重要，可以用一个语句把两个 Box 对象从文件中读入内存的两个不同 Box 对象中。每个 Box 对象都从文件中读取，再显示在 cout 中，说明程序工作正常。

19.9.3 流中更复杂的对象

对于 Box 对象，我们选择了比较简单的情形。处理派生类的对象比较复杂，因为 operator>>()和 operator<<()函数都不是类的成员，不能是虚拟函数。这里需要一个虚拟的输入和输出机制，来确保派生的类对象在使用基类指针或基类引用时能正常处理，但无法使运算符函数成为类的成员。

插入和提取运算符都是二元运算符，而如果二元运算符函数是类的成员，就只能有一个参数。无法把流对象和右操作数传送给函数。但运算符函数可以调用虚拟的类成员，这为解决该问题提供了一个途径。对于 Box 类，可以把它实现为：

```
class Box {
  public:
    // Constructor
    Box(double lv = 1, double wv = 1, double hv = 1);

    // Virtual Destructor
    virtual ~Box();

    // Function to show the volume of an object
    void showVolume() const;
    // Function to calculate the volume of a Box object
    virtual double volume()  const;

    // Member stream I/O functions
    virtual std::ostream& put(std::ostream& out) const;
    virtual std::istream& get(std::istream& in);

protected:
```

```
    double length;
    double width;
    double height;
};

// Insertion and extraction operators
std::ostream& operator<<(std::ostream& out, const Box& rBox);
std::istream& operator>>(std::istream& in, Box& rBox);
```

给类添加了两个虚拟成员，用于执行流 I/O 操作，而由于插入和提取运算符现在调用类公共接口中的函数，所以不再需要成为友元函数。我们只需实现运算符函数，以利用新的成员。put()成员函数的定义如下：

```
std::ostream &Box::put(std::ostream& out) const {
  return out<<' '<< length<<' '<< breath<<' '<< height;
}
```

这个函数与 operator<<()函数的早期版本相同，operator<<()函数的实现如下：

```
std::ostream& operator<<(std::ostream& out, const Box& rBox) {
  return rBox.put(out);
}
```

这个函数调用虚拟函数 put()执行输出操作，返回流对象。同样，get()函数的定义如下：

```
std::istream& Box::get(std::istream& in) {
  return in>>length>>width>>height;
}
```

在友元函数 operator>>()的实现中使用它，如下所示：

```
std::istream& operator>>(std::istream& in, Box& rBox) {
  return rBox.get(in);
}
```

为了提供对 Box 的派生类如 Carton 的流支持，只需定义成员函数，重写基类中的虚拟函数 get()和 put()。派生类的成员函数可以在需要时调用对应的基类版本。

Carton 类继承自 Box，它实现了 volume()函数的另一个版本，添加了一个 string*数据成员，其中包含 string 对象的指针，而该 string 对象包含制造 Carton 的材料名。这个类的 get()和 put()成员函数的原型如下所示：

```
class Carton: public Box {
  public:
     virtual std::ostream& put(std::ostream& out) const;
     virtual std::istream& get(std::istream& in);

  //Rest of the class definition…
};
```

put()函数的实现如下：

```
std::ostream& Carton::put(std::ostream& out) const {
  out<<' '<<*pMaterial;
```

```
  return Box::put(out);
}
```

get()函数的实现如下：

```
std::istream& Carton::get(std::istream& in) {
  pMaterial= new string;                    //Allocate for new string
  in>>*pMaterial;                           //Read the string
  return Box::get(in);
}
```

现在，无论是输出基类 Box 的对象，还是输出派生类 Carton 的对象，基类成员 operator<<() 都会选择对应对象的虚拟 put()函数，即使对象由基类引用来表示，也是如此。这可以用于以 Box 为基类的所有类，只要实现了该类的 get()和 put()成员。

如果类包含指向其他类对象的指针，就需要进一步编译。一个基本的前提条件是，指针指向的对象类型有完成该操作的重载运算符。使用这些运算符可以写入或提取指针指向的对象。另一个前提条件是，必须有一种方式能根据从流中读取的数据构建对象。一种方式是指针指向的类必须有一个默认的构造函数。在从流中读取对象时使用该构造函数，在自由存储区中同步对象，再用流中的数据初始化它们。另一种方式是提供一个类构造函数，它把输入流作为一个参数接受，使用流中的数据构造对象。

如果类成员指针所指向的对象也有类对象的指针，事情就会变得非常复杂。下面用一个例子来说明其中的一些问题，为第 18 章定义的 Stack 模板提供一定的流支持。这里不列举内容全面的例子，但从这个简单的例子中也可以看出其中的原则。这个例子提供了模板的一些流操作，以及包含其他类对象指针的类。

1. 模板类的流支持

首先考虑一下 Stack<T>对象如何输出到流中。模板实例的惟一数据成员是指针 pHead，它包含第一个节点的地址，如果没有节点，它就包含 0。为了把 Node 对象写入 pHead 所指向的文件中，可以使用内层类 Node 的一个友元函数，但这个类是 Stack 的私有成员，最好仍使之为私有成员。另一种方法是声明一个帮助函数 writeNodes()，作为 Stack 模板的一个成员，它把所有的 Node 对象写入文件。其工作原理如图 19-6 所示。

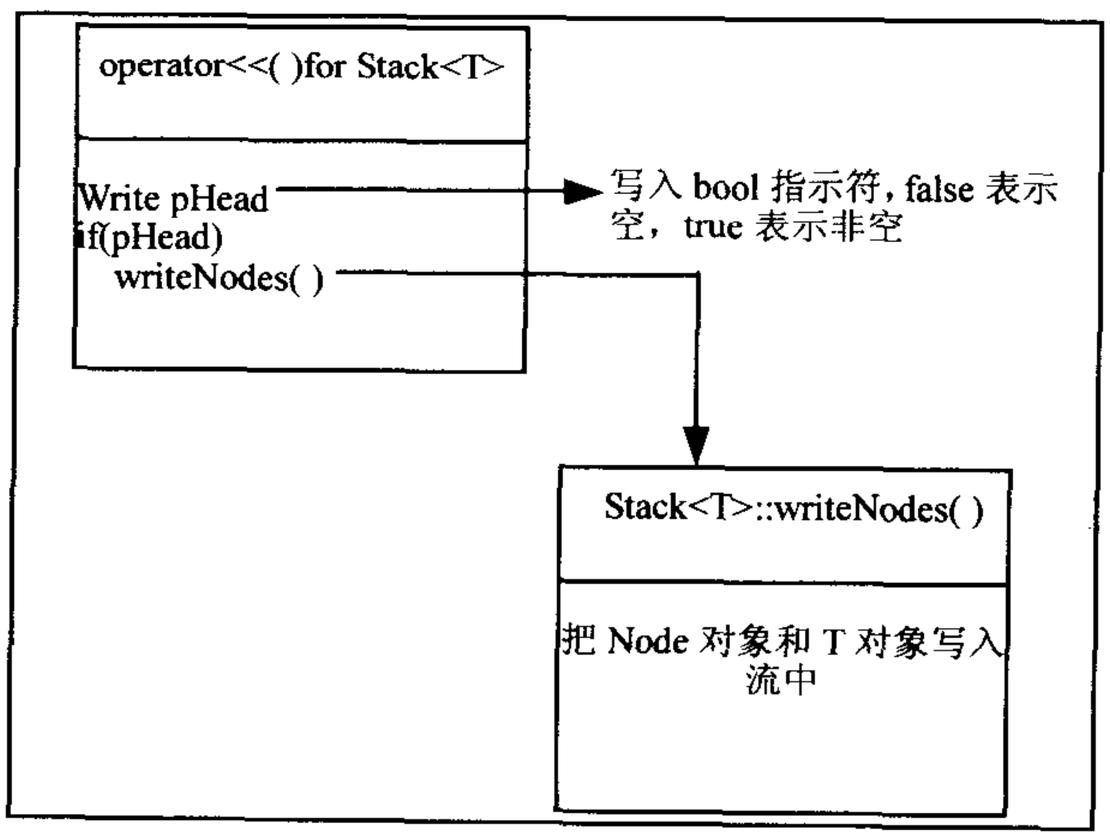


图 19-6　把 Stack 对象写入文件的帮助函数

无法把指针中包含的地址写入流中，因为它们在读取时都是无效的。重要的是，指针是否为空，因为这说明指针指向的文件中是否包含对象。把这个指针的信息写入流中，作为一个bool 值。

writeNodes()函数还需要考虑 Node 数据成员 pItem 指向的对象，因为其他函数都不考虑这个对象。指针指向的对象是 T 类型，由于必须把它写入流中，所以假定 T 类的对象有一个插入运算符版本。显然，函数要能工作，必须有这样一个插入运算符版本。

现在，我们对如何把 Stack<T>对象和随机数量的节点写入流中有了更清楚的认识。如图19-7 所示。

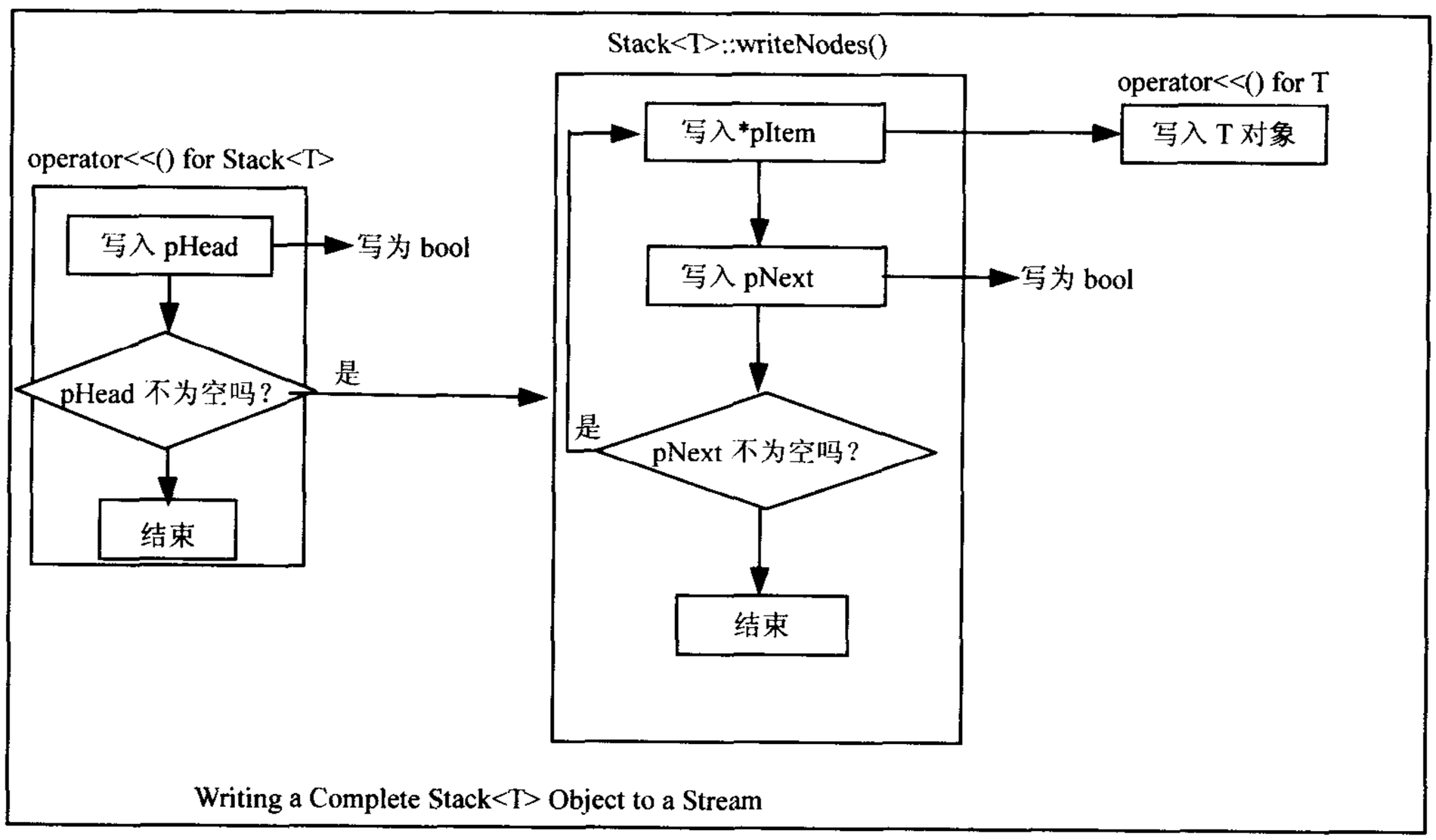


图 19-7　把一个完整的 Stack<T>对象写入流中

Stack<T>插入运算符把 pHead 指针写入流中，如果它不为空，就把 pHead 指针写为 true，否则就写为 false。接着，调用 Stack<T>的 writeNodes()成员函数把 Node 对象写入文件。对于每个 Node 对象，writeNodes()函数还使用 T 类型对象的 operator<<()函数写入了指针 pItem 指向的 T 对象。注意不需要把 pItem 成员的信息写入流中。如果有一个 Node 对象，就写入 T 对象，指针永远不为空。

大致理解了如何把 Stack<T>对象写入流中后，下面详细说明代码中应注意的问题：

- 每个 Stack<T>类实例都需要一个友元函数来重载插入运算符。
- 每个 Stack<T>类实例都需要一个私有成员函数 writeNodes()，把所有的 Node 对象写入流中。
- Stack<T>对象的插入运算符的友元函数和 writeNodes()成员函数需要根据模板来定义。

2. 实现 Stack 类模板的插入操作

首先，在 Stack 模板中，给 operator<<()模板添加友元声明，给 writeNodes()成员函数添加声明。由于是把一个函数模板添加为类模板的友元，所以还必须在类模板定义之前声明函数模板，而且，函数模板在定义之前引用了 Stack 名，所以也必须把 Stack 声明为一个模板：

```
template<typename T> class Stack;          //Class template declaration

// Prototype for friend function template
template<typename T>
  std::ostream& operator<<(std::ostream& out, const Stack<T>& rStack);

template <typename T> class Stack {
  public:
    Stack():pHead(0){}                              //Default constructor
    Stack(const Stack& aStack);                     //Copy constructor
    ~Stack();                                       //Destructor
    Stack& operator=(const Stack& aStack);  //Assignment operator

    void push(T& rItem);                            //Push an object onto the stack
    T& pop();                                       //Pop an object off the stack
    bool isEmpty()  {return pHead == 0;}    //Empty test

  friend std::ostream& operator<< <T> (std::ostream& out, const Stack& rStack);

  private:
    class Node {
       public:
          T* pItem;                                 //Pointer to object stored
          Node* pNext;                              //Pointer to next node
//Construct a node from an object
    Node(T& rItem) : pItem(&rItem), pNext(0){}
    Node()  : pItem(0), pNext(0)  {}          //Construct an empty node
    };

    Node* pHead;                                // Points to the top of the stack
    void copy(const Stack& aStack);     // Helper to copy a stack
    void freeMemory();                          // Helper to release free store memory

    // Write Node objects to a stream
    std::ostream& writeNodes(std::ostream& out, Node* pNode) const;
};
```

友元函数的形式与前面非成员函数的实现方式相同。但模板参数说明<T>的后面是函数模板名。这表示函数模板的参数与类模板的参数相同。注意函数模板名后面的空格是必须有的，这样代码才能编译。

operator<< <T>()的第一个参数是 ostream 对象的引用。第二个参数是要写入的对象的 const 引用。如果要在一个语句的多个<<操作中使用该友元函数，其返回值就必须是 ostream 对象的引用。注意这个函数不是成员函数，第二个参数的类型使用 Stack&，实际上表示 Stack<T>&，所以这是一个函数模板，是类模板的一个友元。

writeNodes()函数有两个参数。第一个参数是流的引用。第二个参数是 Node 对象的指针。指针很容易操作，因为 Node 对象总是通过一个指针来引用。为了方便，函数返回流对象的引用。

在 Stack.h 文件中添加 operator<<()函数的函数模板：

```
template <typename T>
```

```
std::ostream& operator<<( std::ostream& out, const Stack<T>& rStack) {
  out<<' '<<(rStack.pHead != 0);
  return rStack.writeNodes(out, rStack.pHead );
}
```

这个函数先把一个 bool 值写入流中，如果 rStack 的 pHead 成员不为空，该 bool 值就是 true。之后，把同一个指针传送给 writeNodes()函数，将 Node 对象写入流中。

注意如果数据项是通过提取运算符从流中读取的，就需要用空白分隔它们。输出时在每个数据项的前面加一个空格，就可以确保数据项用空格分隔开，如 bool 数据所示。

定义 writeNodes()成员函数的模板也放在 Stack.h 中：

```
template <typename T>
std::ostream& Stack<T>::writeNodes(std::ostream& out, Node* pNode)const {
  while(pNode) {
    out<<' '<<*(pNode->pItem);
    out<<' '<<*(pNode->pNext != 0);
    pNode= pNode->pNext;
  }
  return out;
}
```

指针 pNode 确定 Node 对象是否写入。首先把 pItem 指向的 T 对象写入流中，再写入 pNext 指针的 bool 指示符。接着在 pNode 中存储当前节点的 pNext 指针，用于下一次迭代。

这看起来好像很难，实际上只需把 Stack<T>对象写入流中。下面看看如何读取它们。

3. 理解 Stack 类模板的提取操作

首先看看在使用刚才创建的函数把 Stack 对象写入流中后，流中会包含什么内容。流中应包含一个 Stack<T>对象和 4 个节点，如图 19-8 所示。

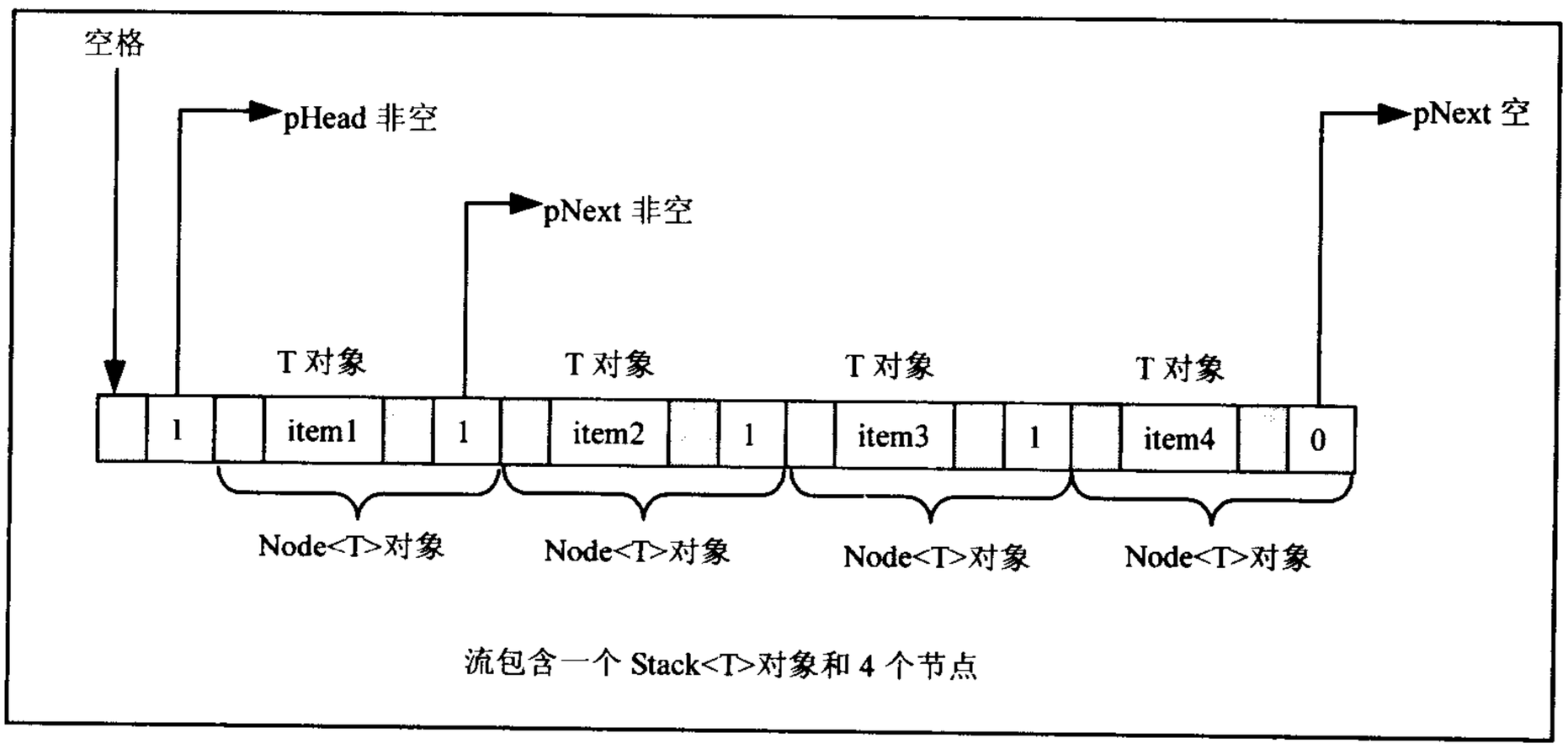


图 19-8　流包含一个 Stack<T>对象和 4 个节点

流中的第一个值是一个空格，其后是 pHead 的 bool 值，如果其后有一个 Node 对象，该 bool 值就应是 true。每个 Node<T>对象都只是一个 pItem 成员指向的 T 对象，其后是一个 bool

值，如果有下一个 Node 对象，该 bool 值就应是 true。在最后一个 Node 对象中，pNext 的 bool 指示符设置为 false，因为后面再也没有节点了。

Stack<T>中的友元函数 operator>>()可以读取 pHead 指示符，如果该指示符是 true，就调用一个私有帮助函数 readNodes()，所以输入过程是输出操作的逆过程。

在从流中读取 Node 对象时，需要创建数据成员 pItem 指向的对象。这有几个严重的问题需要仔细考虑。现在创建了 T 类型的对象，必须在使用完后释放它们。也就是说，对堆栈中的对象必须采用另一种策略来处理。如果提供了把 Stack 写入流中的功能，即使对象是推入堆栈中的，也应在自由存储区中创建对象。接着返回一个副本而不是引用，在对象从堆栈中拉出时，或在删除一个节点时，从自由存储区中释放它们。

4. 管理堆栈中的对象

必须在模板中添加 Node 类析构函数，修改构造函数，使之接受一个指针参数，而不是引用。还需要修改 pop()成员函数，使之返回一个对象，而不是返回引用：

```
template<typename T> class Stack;                    //Class template declaration

// Prototype for friend function template
template<typename T>
  std::ostream& operator<<(std::ostream& out, const Stack<T>& rStack);

template <typename T> class Stack {
public:
    Stack():pHead(0){}                                // Default constructor
    Stack(const Stack& aStack);                       // Copy constructor
    ~Stack();                                         // Destructor
    Stack& operator=(const Stack& aStack);            // Assignment operator

    void push(T& rItem);                              // Push an object onto the stack
    T pop();                                          // Pop an object off the stack
     bool isEmpty()  {return pHead == 0;}             // Empty test

friend std::ostream& operator<< <T> (std::ostream& out, const Stack& rStack);

private:
    class Node {
      public:
        T* pItem;                                     // Pointer to object stored
        Node* pNext;                                  // Pointer to next node
        // Construct a node from an object
        Node(T* pNew):pItem(pNew), pNext(0){}
        Node() : pItem(0), pNext(0)  {}  // Construct an empty node
        ~Node() {delete pItem;}
};
      Node* pHead;                                    // Points to the top of the stack
      void copy(const Stack& aStack); // Helper to copy a stack
      void freeMemory();                              // Helper to release free store memory

      // Write a Node o// to a stream
      std::ostream& writeNodes(std::ostream& out, Node* pNode) const;
};
```

析构函数删除了 pItem 指向的对象，不需要测试 pItem 是否为空，该指针应永远不为空，在空指针上调用 delete 也总是安全的。在 Stack 模板中，把 pop()成员的返回类型改为 T，从而返回堆栈中对象的副本。

必须修改 Stack 模板中 push()和 pop()的实现。下面先修改 push()。必须修改 Stack<T>::push()模板定义，在自由存储区中创建一个 T 对象，它是传送为引用的对象副本，而不仅仅存储原对象的地址：

```
template <typename T> void Stack<T>::push(T& rItem) {
  Node* pNode = new Node(newT(rTtem));  // Create node from object copy
  pNode->pNext = pHead;                 // Point to the old top node
  pHead = pNode;                        // Make the new node the top
 }
```

Stack<T>:: pop()模板必须给自由存储区中的对象制作一个本地副本，再删除自由存储区中的对象：

```
template <typename T> T Stack<T>::pop() {
  if(!pHead)                                  // If it is empty
    throw std::logic_error("Stack empty");    // Pop is not valid so throw exception

    T item(*pHead->pItem);                    // Local copy of top object
    Node* pTemp = pHead;                      // Save address of top node
    pHead = pHead->pNext;                     // Make next node the top
    delete pTemp;                             // Delete the previous top node
    return item;                              // Return copy of top object
}
```

当然，返回机制也会复制对象 item，然后释放本地对象。Stack 模板现在是自包含的，因为它在自由存储区中创建并管理着堆栈中的所有对象副本，无论这些对象是推入堆栈的，还是从流中创建的，都是如此。

5. 实现 Stack 类模板的提取操作

现在，在 Stack<T>模板中，给 operator>>()函数添加一个友元声明，给 Stack<T>模板添加 readNodes()的声明：

```
template<typename T> class Stack; {          //Class template declaration

// Prototype for friend function template
template<typename T>
  std::ostream& operator<<(std:: ostream& out, const Stack<T>& rStack);
template<typename T>
std::istream& operator<<(std:: istream& in, Stack& rStack);

template <typename T> class Stack {
  public:
    Stack():pHead(0){}                          // Default constructor
    Stack(const Stack& aStack);                 // Copy constructor
    ~Stack();                                   // Destructor
    Stack& operator=(const Stack& aStack);      // Assignment operator
```

```
  void push(T& rItem);                          // Push an object onto the stack
  T pop();                                      // Pop an object off the stack
  bool isEmpty()  {return pHead == 0;}          // Empty test

friend std::ostream& operator<< <T> (std::ostream& out, const Stack& rStack);
friend std::istream& operetor>> <T> (std::istream& in, Stack& rStack);

private:
  class Node {
    public:
       T* pItem;                          // Pointer to object stored
       Node* pNext;                       // Pointer to next node

       //Construct a node from an object
       Node(T* pNew) : pItem(pNew), pNext(0){}
       Node(): pItem(0), pNext(0)  {}     //Construct an empty node
       ~Node() {delete pItem;}
  };

  Node* pHead;                             // Points to the top of the stack
  void copy(const Stack& aStack);          // Helper to copy a stack
  void freeMemory();                       // Helper to release free store memory

  // Write Node objects to a stream
       std::ostream& writeNodes(std::ostream& out, Node* pNode)  const;
       Node* readNodes(std::istream& in);  // Read Node objects from stream
};
```

readNode()函数返回所读取的第一个 Node 的指针，使之可以存储在 operator>>()函数的 pHead 中。Stack<T>对象的提取运算符的函数模板如下：

```
template <typename T>
std::istream& operator>>(std::istream& in, Stack<T>& rStack) {
  bool notEmpty;
  in >> notEmpty;
  in(notEmpty)
     rStack.pHead = rStack.readNodes(in);
  else
     rStack.pHead = 0;
  return in;
}
```

流中的第一个 bool 值存储在变量 notEmpty 中。如果这个 bool 值为 true，则表示其后是一个 Node 对象，因此可以调用 readNodes()函数从流中读取 Node 对象。如果这个 bool 值为 false，就把 pHead 设置为空。

readNodes()函数模板的实现代码如下所示：

```
template <typename T>
typename Stack<T>::Node* Stack<T>::readNodes(std::istream& in) {
   Node* pNode = new Node;              // Create a Node object
```

```
    pNode->pItem = new T;                    // Create the T object and store its address
    in >> *pNode->pItem;                     // Read the T object from the stream

    bool isNext;
    in >> isNext;
    if(isNext)
      pNode- >pNext = readNodes(in);
    else pNode->pNext = 0;
    return pNode;
}
```

返回类型是在模板中定义的 Node 类型指针。因为 Node 类在模板定义中，所以它称为关联类型。在这种情况下，编译器需要一点帮助，我们必须在关联类型的返回类型说明前面加上 typename 关键字，告诉编译器这是一个类型，而不是其他名称。

只要确保从流中读取的顺序与写入流中的顺序相同，所有的对象都会正确读取出来。我们创建新的Node对象和类型为T的新对象，把T对象的地址存储在刚才创建的Node对象的pItem指针中。接着读取 pNext 指针的 bool 指示符。如果该指示符为 true，就表示有另一个 Node 对象要读取，因此调用 readNodes()函数从流中读取该 Node 对象，并把返回的指针存储在 Node 对象的 PNext 成员中。这个过程一直继续下去，直到读取 pNext 指示符为 false 的 Node 对象为止，此时函数调用序列停止。

程序示例 19.9——对模板类实例执行流操作

下面对一组 Box 对象试用修改后的模板类。本章前面使用的 Box 类有一个默认的构造函数。Box 对象还实现了插入和提取运算符，默认的副本构造函数也可以使用，所以该模板类完全可应用于堆栈中的流操作。下面是测试它的代码：

```
// Program 19.9 Writing a stack to a stream    File: prog19_09.cpp
#include <fstream>
#include <iostream>
#include <string>
using std::cout;
using std::endl;
using std::string;

#include "Stack.h"
#include "Box.h"

int main() {
   Box Boxes[10];                            // 10 default boxes

   for(int i = 0 ; i < 10 ; i++)             // Create different objects
     Boxes[i] = Box(10 * (i + 1), 10 * (i + 2), 10 * (i + 3));

   Stack<Box> boxstack;                       // A stack for Box objects

   // Push all Box objects onto the stack
   for(int i = 0 ; i < 10 ; i++)
     boxStack.push(Boxes[i]);
```

```
    const string boxFileName = "C:\\JunkData\\boxes.txt";    // Stack file
    std::ofstream outBoxFile(boxFileName.c_str());

    outBoxFile << boxStack;                          // Write the stack
    outBoxFile.close();                              // Close the stream

    // Display volumes for original set
    while(!boxStack.isEmpty())
       cout << endl << "Volume =" << boxStack.pop().volume();

       Stack<Box> copyBoxStack;                      // New stack for Box objects

       Std::ifstream inBoxFile(boxFileName.c_str());
       inBoxFile >> copyBoxStack;                    // Read the stack

       // Output volumes of Box objects off the stack from the stream
       int i = 0;
       while(!copyBoxStack.isEmpty())
          cout << endl
               << "Volume of Box["<< (i++) <<"] is"
               << copyBoxStack.pop().volume();

       cout << endl;
       return 0;
    }
```

这个程序的结果如下所示：

```
Volume = 1.32e+006
Volume = 990000
Volume = 720000
Volume = 504000
Volume = 336000
Volume = 210000
Volume = 120000
Volume = 60000
Volume = 24000
Volume = 6000
Volume of Box[0] is 1.32e+006
Volume of Box[1] is 990000
Volume of Box[2] is 720000
Volume of Box[3] is 504,000
Volume of Box[4] is 336000
Volume of Box[5] is 210000
Volume of Box[6] is 120000
Volume of Box[7] is 60000
Volume of Box[8] is 24000
Volume of Box[9] is 6000
```

例子的说明

从输出中可以看出，Box 对象从文件中读取出来之后，再从堆栈中拉出，其体积与原对象的体积完全相同。在 main()中，创建了默认 Box 对象的一个数组：

```
Box Boxes[10];                    //10 default boxes
```

在循环中，把数组元素设置为不同的 Box 对象：

```
for(int i=0; i<10; i++)                              //Create different objects
  Boxes[i]=Box(10*(i+1), 10*(i+2), 10*(i+3));
```

第一个对象的尺寸是(10, 20, 30)，下一个对象的尺寸是(20, 30, 40)，依此类推。接着创建一个可以存储 Box 对象的空堆栈，如下面的语句所示：

```
Stack<Box> boxStack;              //A stack for Box objects
```

接着，在一个 for 循环中把 Boxes 数组的元素推入堆栈：

```
for(int i=0; i<10; i++)
  boxStack.push(Boxes[i]);
```

boxStack 的 push()成员给传送为引用参数的对象制作了一个副本，在堆栈内部使用它。下一步是把 boxStack 写入一个文件流：

```
const string boxFileName="C:\\JunkData\\boxes.txt";          //Stack file
std::ofstream outBoxFile(boxFileName.c_str());

outBoxFile << boxStack;                                  //Write the stack
outBoxFile.close();                                      //Close the stream
```

这段代码使用了给 Stack 模板实现的<<运算符。在写入 Stack<Box>对象后，关闭文件。为了引用它们，把 Box 对象从原堆栈中拉出，显示它们的体积：

```
while(!boxStack.isEmpty())
  cout<<endl<<"Volume = "<< boxStack.pop().volume();
```

当然，这里处理的是堆栈内部的 Box 对象的副本。pop()成员会删除对应于已拉出的 Box 对象的节点，Node<Box>的析构函数则从自由存储区中删除 Box 对象。

现在要从流中读取堆栈，因此创建另一个堆栈，来存储 Box 对象，再给刚才写入的文件创建一个输入文件流：

```
Stack<Box> copyBoxStack;                    //New stack for Box objects
Std::ifstream inBoxFile(boxFileName.c_str());
```

要读取堆栈，可使用提取运算符：

```
inBoxFile >> copyBoxStack;                  //Read the stack
```

最后一步是从堆栈中拉出对象，显示其体积，以说明它们与原对象的体积是相同的：

```
int I = 0;
while(!copyBoxStack.isEmpty())
  cout << endl<<"Volume of Box[" << (i++) << "] is"
       << copyBoxStack.pop().volume();
```

实现流操作从整体上看是一个技术性非常强的操作，如果 C++提供了对流操作的支持，就可以省略许多工作。

19.10 本章小结

本章介绍了流操作的基本知识，说明了如何在C++程序中把它们应用于文件的使用。本章的要点如下：

- 标准库支持字符流、二进制(字节)流和字符串流的输入输出操作。
- 输入和输出的标准流是 cin 和 cout，还有错误流 cerr 和 clog。
- 提取和插入运算符提供了格式化的流输入输出操作。
- 文件流可以与磁盘上的文件关联起来，进行输入、输出或输入输出。
- 文件打开模式决定了是从流中读取数据，还是给流写入数据。
- 如果创建了一个文件输出流，并把它与未有文件本身使用的的文件名关联起来，就会创建该文件。
- 文件有开头、结尾和当前位置。
- 可以把文件流的当前位置改为以前记录的某个位置。这个位置与流开头的偏移量为正，与流结尾的偏移量为负，而与流当前位置的偏移量可以为正，也可以为负。
- 为了支持类对象的流操作，可以重载插入和提取运算符，使这些运算符函数成为类的友元。
- 字符串流类提供了 string 对象的流输入输出操作。

19.11 练习

1. 编写一个 Time 类，把小时、分钟和秒存储为整数。提供一个重载的插入运算符(<<)，把时间以 hh::mm::ss 的格式输出到任意输出流上。

2. 给 time 类提供一个简单的提取运算符(operator>>())，它可以以 hh::mm::ss 格式读取时间值。如何复制“:”符号？(提示：用什么类型的变量存储“:”？)

3. 编写一个程序，把时间值记录到文件中。编写一个匹配程序，读取文件中的时间值，把它们显示到屏幕上。

4. 编写一个程序，从标准输入中读取文本行，再把它们写到标准输出上，删除所有的前导空白，把多个空格转换为一个空格。对键盘输入测试该程序，再对从文件中读取的字符测试该程序。编写第二个程序，把小写字母转换为大写形式，再测试它们。

第 20 章　标准模板库

C++提供了一个扩展的标准库，简化了许多编程任务。除了前面介绍的内容之外，这个库还提供了一组包含更高级功能的模板。只要程序需要这些功能，就可以使用这些模板给自己的数据类型创建标准的容器、算法和函数。这些模板总称为标准模板库(STL)。本章介绍基本的 STL 容器以及同这些容器一起使用的算法。

本章主要内容

- 标准模板库的基本架构
- 如何创建和使用序列容器 vector<>和 list<>，以及关联容器 map<>和 multimap<>
- 迭代器的概念和用法
- 算法的概念，在算法中迭代器非常重要的原因
- 迭代器如何把容器跟过时的标准 C++数组关联起来
- 如何创建和使用流迭代器
- 如何定义自己的迭代器类
- 如何创建和使用关联容器 map<>和 multimap<>

20.1　STL 架构简介

STL 提供了一组基本工具，用于操作类对象。例如，使用 STL 可以创建一个类，为任意数据类型定义链表。如果需要链表来存储 Box 对象或 FootballPlayer 对象，就可以使用 STL。实际上，STL 还可以完成更多的工作。STL 还提供了处理对象的标准方式。如果需要以升序方式排列 Box 对象，STL 则可以生成一个排序函数。实际上， STL 能生成一个可处理任意数据类型的排序函数，该函数会为该类型定义一个重载的比较运算符。无论程序使用什么类型的对象，如果需要采用标准方式对对象进行组织或分析，就可以使用 STL。

STL 有非常多的功能，这里不可能全部介绍。STL 的用法需要一整本书的篇幅来论述，而本章的目的是讨论 STL 的工作方式，这些内容足以让读者自己领会 STL 的其他功能。我们将对 STL 提供的功能范围作一个简要的介绍，然后在例子中演示如何应用 STL 中最常用的功能。

20.1.1　STL 组件

STL 主要提供了三类工具：容器、迭代器和算法，如图 20-1 所示。

容器是以各种方式存储其他对象的类对象。链表就是一种容器。STL 提供了一组具有不同特性的容器，在应用程序中使用哪种容器取决于访问容器中内容的方式。

迭代器是智能指针的一种形式，例如第 14 章为 TruckLoad 容器使用的 BoxPtr 对象。在以某种方式处理对象时，就可以使用迭代器访问容器中的对象。

算法是一个函数，它以特殊的方式处理源(如容器)中的元素。例如，一些算法可以对容器

中的内容进行排序和搜索，其他算法可以处理数值数据。使用迭代器可以从容器中提取对象，把它们提供给算法，进行处理。

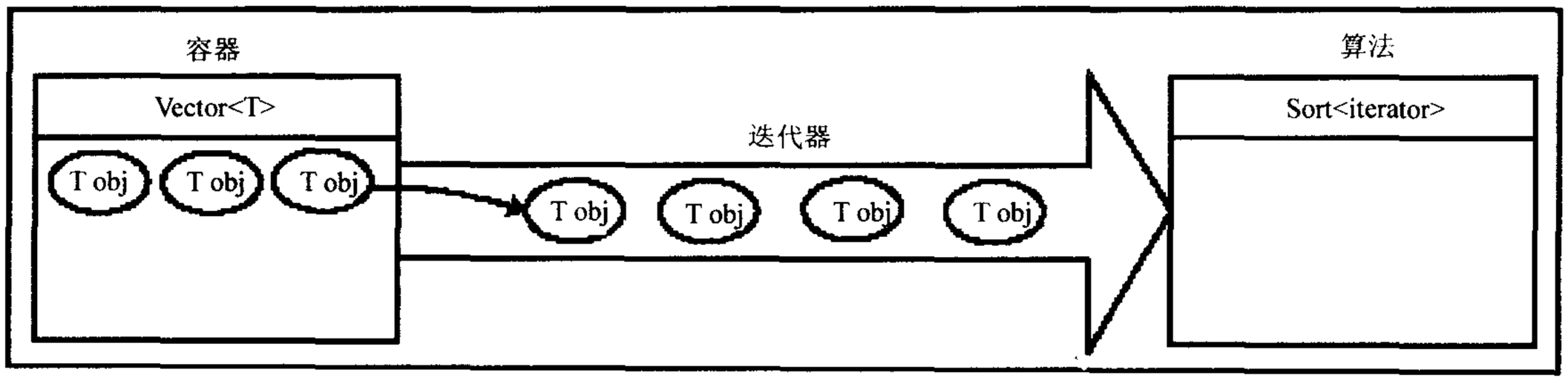


图 20-1　容器通过迭代器与算法关联在一起

在介绍使用 STL 编写代码之前，先概述一下 STL 提供的功能。首先介绍各种容器类型和它们的功能，了解每个容器类型处理其内容的特定方式。然后介绍各种迭代器的功能以及它们与容器的关系，最后讨论 STL 算法中可以使用的函数。

1. STL 容器

容器是 STL 功能中最有趣的一个，它们使用得最频繁。前面说过，如果使用容器，就要使用迭代器，因此需要很好地理解容器和迭代器。STL 提供了两类容器：序列容器以线性序列方式组织对象；关联容器用关联的键把对象组织在一起。还可以通过迭代器从关联容器中提取对象。

所有的 STL 容器都存储了其对象的副本。也就是说，如果给容器添加一个对象，并修改原对象，原对象和容器中的对象就是不同的。同样，从容器中提取一个对象时，会获得容器中对象的副本。如果要修改容器中的对象，就必须用新版本替代它。副本使用该对象类型的副本构造函数来创建，因此复制过程需要大量的系统开销。此时，在容器中存储对象的指针，而把原对象放在容器外面比较好。

序列容器

序列容器的三种基本类型分别对应于模板 vector<T>、deque<T>和 list<T>。它们都存储了一组特定类型 T 的对象，这些对象以线性序列方式组织，但每个容器都可以访问以不同方式优化的内容。因此，在特定实例中选择哪个序列容器取决于使用其中存储的实体的方式。所存储的实体类型可以是基本类型如 int 或 double、类类型，或者基本类型或类型类型的指针。

vector<T>容器以类似于普通数组的方式存储类型 T 的对象，只是它的容量可以根据需要存储的对象个数进行调整。在希望以数组方式处理对象时，就可以选择使用 vector<T>容器。使用下标运算符[]可以象数组那样访问 vector 容器中的元素。vector 容器的大小是其中元素的个数，其容量是当前它可以包含的最大元素数。在创建特定容量的 vector 对象时，其大小和容量是相同的。例如，下面创建一个 vector 对象，它的容量是 10 个 int 类型的元素：

```
std::vector<int> a(10);      // a vector with 10 elements
```

以这种方式创建的 vector 中的元素，使用指定为模板参数的类型的默认构造函数来初始化。本例的元素类型是 int，所以元素设置为 0。由于元素都已初始化，所以 vector 的大小和容量都是 10。

如果给 vector 添加一个或多个元素，其大小就会以某个比例增长，以容纳新元素，所以其容量就会大于(肯定不会小于)其大小。调用 size()和 capacity()成员分别可以获得 vector 容器的大小和容量。这两个成员返回的值都是 size_type 类型，它由 typedef 定义为无符号的整数类型，通常与 size_t 相同。名称为 a 的 vector 容器存储 int 类型的值，如图 20-2 所示。

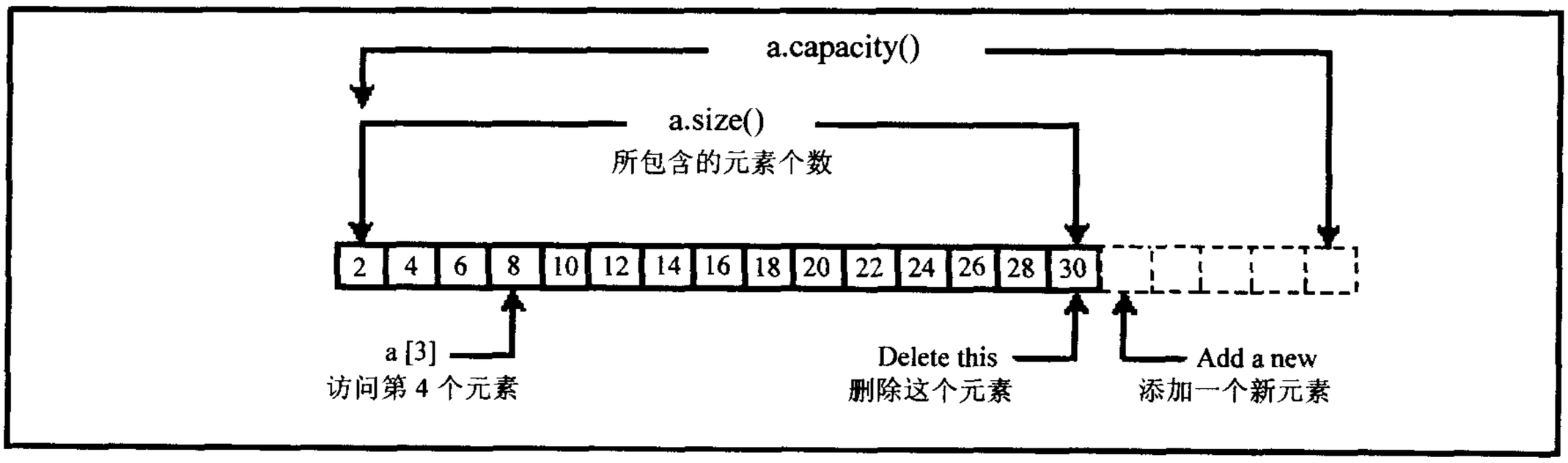


图 20-2　vector 容器的例子

只添加或删除序列尾部的元素时，vector 是最高效的。当然，如果增大容量，vector 会自动增长。可以在 vector 的开头和中间插入元素，但其效率相当低，因为需要移动插入点后面的所有元素，并在自由存储区中分配一个新内存空间。从 vector 的开头和中间删除元素也比较慢，因为元素也需要移动。如果要在序列容器的中间添加或删除元素，就不应使用 vector，而应使用 list 容器。

deque<T>容器可以组织一列 T 类型的元素。deque 是 double-ended queue 的缩写，这是其主要特性。它优于 vector 容器的方面是可以在序列开头和末尾高效地添加或删除对象，因此在需要这些功能时，就应选择这种类型的容器。创建给定大小的 deque 对象的方式与 vector 容器相同：

```
std::deque<int> b(10);      // a deque with 10 elements
```

名称为 b 的 deque 容器存储 int 类型的值，如图 20-3 所示。

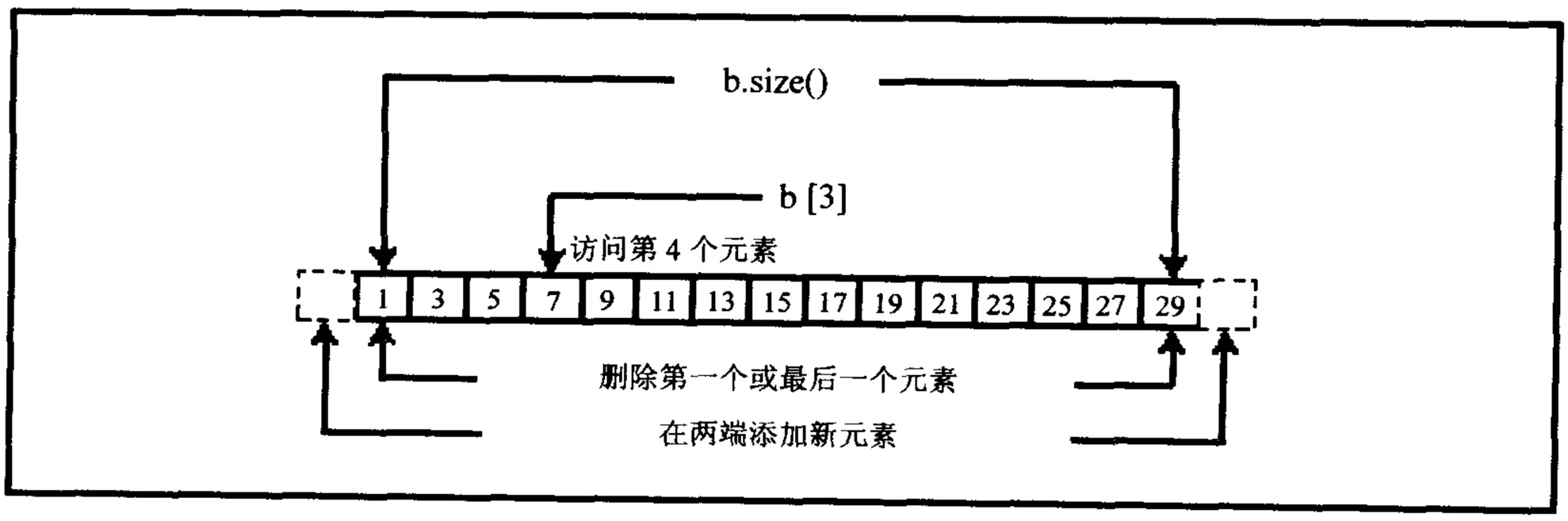


图 20-3　deque 容器的例子

deque 容器类似于 vector 容器，可以使用下标运算符访问元素。但是，deque 容器元素的内部的组织方式与 vector 容器完全不同，也比较复杂，deque 容器的大小总是等于其容量。因

此，deque 容器没有 capacity()成员函数，但仍有 size()成员，它返回 size_type 类型的当前大小。由于内部组织方式不同，所以 deque 容器使用起来比 vector 容器略慢。与 vector 一样，在 deque 序列的中间可以添加或删除元素，但其过程比较慢，因为总是要复制已有的元素。

list<T>容器在链表中存储类型 T 的元素。在序列中存储元素，并能在序列的中间插入或删除元素时，就应使用 list 容器。list<T>容器在概念上类似于第 14 章开发的链表，但要复杂得多。例如，list<T>容器为列表中的元素存储了向前和向后的指针，因此 list 容器可以向两个方向传输。list 容器中的组织方式如图 20-4 所示。

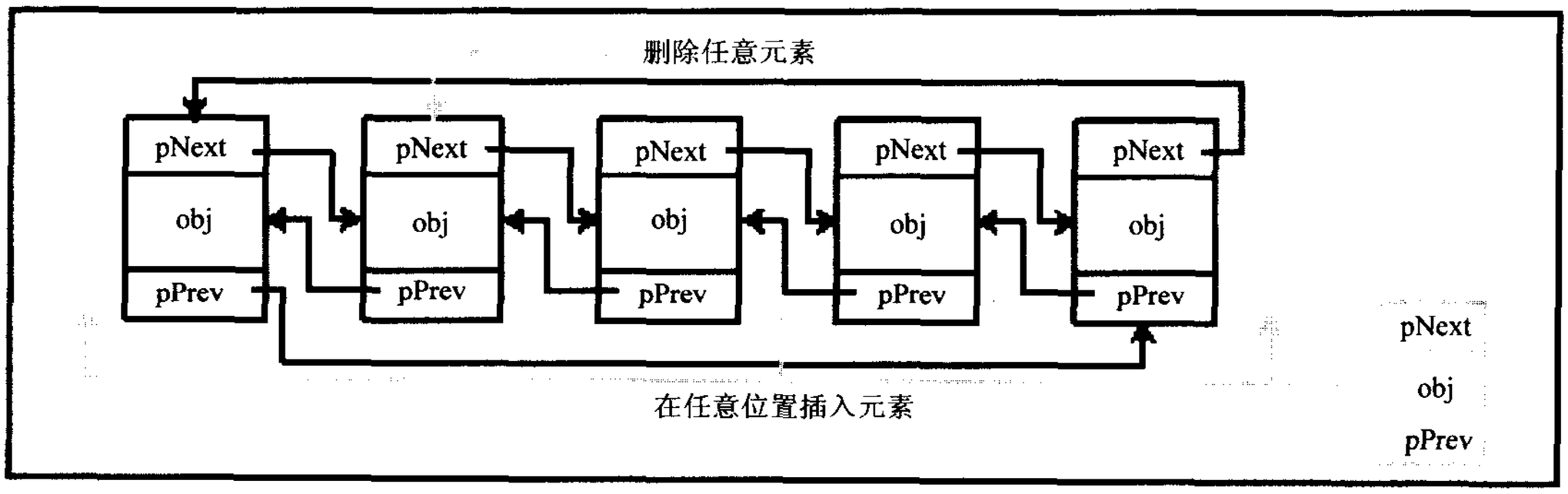


图 20-4　list 容器的例子

在使用 list 容器时，可以在序列的任意位置高效地插入元素，无需移动已有的元素。如图 20-4 所示，只需把新元素插入位置两边的元素的 pNext 和 pPrev 指针设置为指向新元素。删除 list 中的元素也比较快。list 容器的主要限制是随机访问元素比较慢，因为这需要从 list 的第一个元素或最后一个元素开始遍历整个 list，查找每个元素。一般的 STL sort()算法不能应用于 list 容器，因为它需要一个提供了随机访问元素功能的迭代器。但是，仍可以使用内置的 sort()函数对 list 容器的内容排序。

容器适配器是一个模板，它把序列容器模板作为它的一个参数，为基本序列容器的修改版本创建模板。容器适配器为 stack 容器类、queue 容器类和 priority_queue 容器类提供了模板。stack 容器类采用先进后出的存储方式，queue 容器类采用先进先出的存储方式，而 priority_queue 容器类的存储方式是，下一个要提取的对象总是容器中最大的对象。STL 还定义了 vector 模板的规范，用于优化 bool 类型元素的存储方式。

关联容器

关联容器允许使用键存储和提取给定类型的对象。关联容器中的对象按照键确定的顺序来存储。至少要定义所使用的 Key 对象，以便使用<运算符。稍后介绍其工作原理。关联容器有四个基本类型：map<Key, T>、multimap<Key, T>、set<Key >和 multiset<Key >。

在 map 或 multimap 容器中，类型 T 的每个对象都与其另一个类型 Key 的关联键对象一起存储在容器中。要从容器中提取一个对象，只需提供其键即可。例如，在关联容器中，把个人号码用作对象的键，其类型是 long，以存储 Employee 对象，为此，可以使用 map<Employee, long>容器实例。接着只要提供相应的个人号码作为键，就可以从容器中提取 Employee 对象。存储在容器中的 Employee 对象按照个人号码的升序进行排序。

map 容器一个非常有用的特性是，可以把它用作关联数组。关联数组允许通过任意对象的

索引访问其元素。map 实现了下标运算符[]，其中索引是键类型，所以对于 map 对象 mymap，就可以使用表达式 mymap[key]访问对应于给定 key 对象的元素。当然，键可以是任意类型，只要提供了 Compare 对象，就可以比较键。

对于 set 和 multiset 容器，对象就是它们的键。这两个容器的原则值是自动附加给元素的顺序。可以在 set 中存储 Employee 对象，它们将根据 Compare 对象进行排序。接着使用迭代器访问容器的内容，而且 Employee 对象肯定会按顺序处理。

对于 set 和 map 容器，所有的键都必须是惟一的，在 multiset 和 multimap 容器中，键可以重复。bitset 容器模板可以创建类，来保存和操作固定大小的位序列。

2. STL 迭代器

如前所述，迭代器是一种智能指针。在以某种方式处理容器中存储的元素时，特别是把 STL 算法应用于它们时，要使用迭代器访问它们。把递增运算符应用于迭代器，从一个对象移动到下一个对象，就可以从容器中获得一个迭代器对象，从而遍历存储在容器中的元素。也可以对迭代器使用其他操作，但这取决于所使用的容器类型。

STL 还有其他迭代器，用于给流传送对象或从流中传出对象。STL 算法需要从迭代器中提取输入。因此可以把算法应用于源中可通过迭代器访问的对象。例如，算法可以直接应用于流中的对象和容器中的对象。只要提供了可以访问的迭代器，它们就可以应用于任何情形。本章后面将介绍这种迭代器。

有四种迭代器，它们的功能各不相同。这些迭代器按照从简单到复杂的顺序排列，如下：

1. 输入输出迭代器：用于读取一系列对象，且只使用一次。如果需要第二次读写序列，就必须创建一个新的输入或输出迭代器。可以应用于这两个迭代器的操作有++iter 或 iter++、iter1==iter2、iter1!=iter2 和*iter。但输入迭代器只能进行读取访问，输出迭代器只能进行写入访问。只有输入迭代器可以使用 iter->member。

2. 向前迭代器：合并了输入和输出迭代器的功能。还可以多次使用向前迭代器读写一系列对象。

3. 双向迭代器：与向前迭代器的功能相同，但允许向前和向后遍历一系列对象。因此可以对这些迭代器执行--iter 和 iter--运算符。

4. 随机访问迭代器：提供的功能与双向迭代器相同，还允许随机读写元素。除了双向迭代器可以使用的操作之外，随机访问迭代器还支持下述操作：iter+n 或 iter－n；iter[n](等价于*(iter+n))；iter+=n 或 iter－=n；iter1－iter2；以及 iter1<iter2，iter1>iter2，iter1<=iter2 或 iter1>=iter2。

在这个列表中，迭代器的功能是累积的，每种迭代器的功能都会累加到后面的迭代器上。因此，输入输出迭代器的功能最弱，随机访问迭代器的功能最强。如果算法需要给定类型的迭代器，就不能使用差一级的迭代器。但是，总是可以使用高级的迭代器。向前迭代器、双向迭代器和随机访问迭代器也可以是可变的或固定的，这取决于解除迭代器的引用是产生一个引用还是产生对常量的引用。不能把解除常量迭代器的引用结果放在等号的左边。

从容器中获得的迭代器的特性取决于容器的类型。随机访问迭代器由 vector 和 deque 容器支持。List、map、multimap、set 和 multiset 容器提供了双向迭代器。

3. STL 算法

算法为使用迭代器访问的对象组提供了计算和分析函数。它们是通过迭代器访问数据元

素，所以可以把算法应用于容器的内容，还可以把它们应用于可以通过迭代器访问的任何序列，因此可以把算法应用于流。

算法是 STL 中最大的工具集合。但是，并不是所有的算法都与应用程序相关，其中的一些算法还有专门的用途。本章不详细介绍 STL 算法，但会应用一些算法。

算法有三大类：

1. 不修改序列的操作：不以任何方式修改应用它们的序列。这一类的操作包括 find()、find_end()、find_first()、adjacent_find()、count()、mismatch()、search()和 equal()。

2. 修改序列的操作：修改序列中的元素，这一类的操作包括 swap()、copy()、transform()、replace()、remove()、reverse()、rotate()、fill()和 random_shuffle()。

3. 排序、合并和相关的操作：在一些情况下会改变序列的顺序。这一类的操作包括 sort()、stable_sort()、binary_search()、merge()、min()、max()和 lexicographical_compare()。

注意这几类中的操作并没有囊括所有的操作。

一些操作如 transform()需要函数对象才能执行。函数对象是重载了函数调用运算符 operator()的类对象，专门用于作为参数传送给函数，其效率要比使用原始的函数指针更高。

数值算法

<numeric>头文件为数值操作的算法声明了函数模板。这些算法不是 STL 的一部分，但可以和容器一起使用，因为它们使用迭代器。

表 20-1 列出了的函数模板可以执行<numeric>头文件声明的数值算法。

表 20-1　数值算法的模板

模　板	说　明
accumulate<>	这个模板生成的函数计算迭代器指定的范围内的元素和
adjacent_difference<>	这个模板生成的函数计算指定范围内的相邻元素的差，并把结果输出到另一个指定的范围内
inner_product<>	这个模板生成的函数计算两个范围的内积，并加到指定的初始值上
partial_sum<>	这个模板生成的函数计算指定范围内第一个元素到第 n 个元素的和，并把结果输出到另一个指定的范围内

如果不熟悉这些算法中排序问题的重要性，它们可能没有什么意义，此时可以忽略它们。

20.1.2　STL 头文件

STL 提供的功能在表 20-2 所列的头文件中声明。

表 20-2　STL 头文件

头　文　件	用　途
<vector>	单端口数组容器
<deque>	双端口数组容器
<list>	双向链表容器

(续表)

头 文 件	用 途
<map>	map 和 multimap 关联数组容器
<set>	set 和 multiset 已排序的设置容器
<bitset>	表示位序列的对象
<queue>	双端口队列(容器适配器)
<stack>	堆栈(容器适配器)

支持迭代器所需要的声明在<iterator>头文件中，但提供容器声明的头文件也包含这个头文件。因此，如果不使用容器，就只需在源文件中包含<iterator>头文件。<utility>头文件包含定义和管理 map 和 multimap 容器所使用的键/值对所需的声明，所以这个头文件会自动由<map>头文件包含进来。

可以应用于容器和能通过迭代器访问的其他序列的算法声明在<algorithm>头文件中。函数对象的支持由<functional>头文件的声明提供。

数值处理的算法库严格说来不是 STL 的一部分，但这些算法与 STL 兼容，因为它们通过迭代器工作。数值算法所需要的声明在<numeric>头文件中。另外两个头文件提供了数值处理环境中一些很有用的功能。<complex>头文件提供了复杂数字的支持，<valarray>头文件提供了数值数组的支持。

另外，在流头文件集合中还声明了一些迭代器，如<iostream>、<istream>、<ostream>和<sstream>。这些头文件利用 STL 支持基于基本 STL 算法的输入输出。

20.2 使用 vector 容器

vector 容器跟普通的 C++数组相似。实际上，vector 容器可以在能使用 C++数组的任意地方使用。但是，vector 容器使用起来比 C++数组简单多了—— 使用数组时，必须管理数组的大小和容量，而使用 vector 容器时，这些管理工作在容器中都是自动完成的。普通数组的大小是固定的，而 vector 容器会根据需要自动增大其容量。一旦习惯使用 vector 容器，就不会再使用普通的数组了。

20.2.1 创建 vector 容器

下面的语句创建了一个空的 vector 容器，它可以存储 int 类型的元素：

```
std::vector<int> numbers;
```

这个语句使用模板参数 int 调用容器类的默认构造函数，默认构造函数创建了一个容器，其初始容量和大小都是 0。如前所述，容量是在任意给定时刻容器可以存储的元素个数，而大小是实际存储的元素个数。这不是创建 vector 容器的高效方式。每次添加新元素时，都需要增大容量，这涉及到把已有的元素复制到区域扩大了的新内存中。一般情况下，应在创建 vector 容器时，指定与所需空间大致相当的容量。例如，下面创建了一个 vector 容器，它包含 10 个

int 类型的元素：

```
std::vector<int> samples(10);
```

这个 vector 容器有 10 个初始化为 0 的元素。

当然，也可以创建一个 vector 容器，包含 Box 对象：

```
std::vector<Box> boxes(20);
```

这个语句创建了一个 vector 容器，它包含 20 个 Box 对象，每个元素都调用默认的 Box 类构造函数，用创建出来的 Box 对象来初始化。

这说明，Box 类必须有一个默认构造函数，才能使该语句生效。因此就有了一个问题：类型 T 需要满足什么要求，容器才可以接受它？类型 T 的最低要求如下：

```
class T{
  public:
  T();                          //default constructor
  T(const T& t);                //Copy constructor
  ~T();                         //Destructor
  T& operator=(const T& t); //Assignment operator
};
```

注意 operator<()不需要，因此没有包含在 T 的上述定义中。但是，最好提供有意义的元素排序方式的定义。否则，元素类就不能在关联容器如 map 和 set 中用作键，元素序列就不能使用任何排序算法。编译器在许多情况下都为这些算法提供了默认的实现方式，所以大多数类类型都应在容器中能使用排序算法。

注意调用 resize()函数，其参数指定为新的元素个数，就可以为 vector 容器修改大小(不是容量)。如果新的大小比原来的小，就从序列的尾部删除元素。如果新的大小比原来的大，就在序列的尾部添加元素，新元素用所存储元素的类型默认构造函数来创建。对于基本类型，元素值就是 0。调用 reserve()成员，把元素个数用作参数，就可以设置 vector 的最小容量。如果传送给 reserve()的值小于当前的容量，容量就保持不变。

不能直接用一组静态的初始化器初始化 vector 容器，而可以使用 C++数组声明一组初始化器，用于初始化容器的内容：

```
int values[]={99, 136, 247, 459, 679, 871, 928, 1045, 1300, 1302};
```

现在就可以把这些值复制到 int 元素的 vector 容器中，如下所示：

```
for(int I =0; i<sizeof values/sizeof value[0]; i++)
  simple[i]=value[i];
```

这个语句使用下标运算符和容器存储每个元素值，其方法与访问普通数组 values 中的元素相同。

另一种方法是使用构造函数，接受一系列初始值，这些值在某个外部元素源中指定为间隔：

```
std::vector<int> sample(values, values+sizeof values/sizeof value[0]);
```

这个构造函数需要两个迭代器参数指定的间隔，但因为指针是最好的迭代器，所以可以使用指针指向数组的元素。两个参数把值的范围指定为半开间隔。半开间隔就是在一组值中包含

一个范围界限，不包含另一个范围界限，在本例中包含第一个范围界限。

创建 vector 容器的另一个方法是把所有的元素初始化为一个给定的值。下面创建的 vector 容器有 50 个 double 类型的元素。每个值都初始化为 3.14159：

```
std::vector<double>(50, 3.14159);
```

第一个参数是容器中的元素个数，这些元素都初始化为第二个参数指定的值。

下面在一个例子中使用 vector 容器。

程序示例 20.1——比较 vector<>和 C++数组

下面的程序使用了 int 数组和元素类型为 int 的 vector，并比较它们的工作方式。

```
// Program 20.1 A quick comparison of array and vector    File: prog20_01.cpp
#include <iostream>
#include <vector>
using std::cout;
using std::endl;
using std::vector;

int main() {
  int a[10];              // C++ array declaration
  vector<int> v(10);      // Equivalent STL vector declaration

  cout << "size of 10 element array: "<<sizeof a <<endl;
  cout << "size of 10 element vector: "<<sizeof v <<endl;

for (int i = 0; i < 10; ++i)
  a[i] = v[i] = i;

int a_sum =0, v_sum =0;
for (int i = 0; i < 10; ++i) {
  a_sum +=a[i];
  v_sum +=v[i];
}

cout <<"sum of 10 array elements: "<<a_sum <<endl;
cout <<"sum of 10 vector elements: "<<v_sum <<endl;

vector<int> vnew(a, a+sizeof a/sizeof a[0]);     //Initialized from an interval
int vnew_sum = 0;
for (int i = 0; i < vnew.size(); ++i)
  vnew_sum += vnew[i];
  cout <<"sum of 10 new vector elements: "<<vnew_sum <<endl;

  return 0;
}
```

运行这个程序，得到如下结果：

```
size of 10 element array: 40
size of 10 element vector: 16
sum of 10 array elements: -8
sum of 10 vector elements: -8
sum of 10 new vector elements: -8
```

例子的说明

所有的 STL 对象都在标准库中，因此它们都在命名空间 std 中定义。当然，在使用需要的 STL 功能时，必须包含对应的头文件。如果不想使用带有限定符 std 的名称，则可以为每个要使用的名称提供 using 声明。

在代码中，使用数组和 vector 的惟一区别是在声明中：

```
int a[10];                          //C++ array declaration
vector<int> v(10);                  //Equivalent STL vector declaration
```

C++数组是一种内置的语言类型，用[]语法声明。而 vector 是一个 STL 类型，实现为一个类模板。模板参数指定了 vector 所包含的对象类型。在本例中，vector 包含 int 类型的对象。

在这个例子中，数组 a 和 vector 对象 v 都是自动变量。对于数组，这表示在自动存储区中创建包含 10 个整数的存储空间。在一些机器上，int 类型的变量占用 4 字节的内存，所以数组 a 要占用 40 个字节，如输出所示。

vector 对象 v 也存储在自动作用域中，它在自由存储区中分配存储空间，以存储元素。在内部，它至少会获取存储元素所需要的足够存储空间。vector 对象的内部机制处理存储空间的分配，并获取足够多的自由存储区，来存储其内容(这取决于自由存储区的大小)。

vector 对象使用 sizeof 运算符占用的空间不包括存储元素所需要的空间。因此，这种类型的对象常常称为句柄。它只包含 vector 对象管理其元素的存储空间所需要的内存空间。vector 对象 v 仅占用 16 个字节。v 的内容(包含它存储的值或对象所需要的空间)是动态分配内存的，在自由存储区中占用另外 40 个字节。

显然，数组对象使用的存储空间少于 vector 容器，但是，每个 vector 对象的系统开销非常低。数组对象的一个缺点是它在程序堆栈中分配内存空间，在编译期间，它占用的空间和元素个数是固定的。vector 的小系统开销在两个方面补足了这一点：首先，内存管理是自动进行的，其次，可以在需要时给 vector 添加元素。

最后一行输出表示，构造函数工作正常，它使用由半开间隔指定的初始值。Vector 容器 vnew 的元素和与数组 a 和 vector 容器 v 相同。

20.2.2 访问 vector 容器中的元素

可以用各种方式访问 vector 容器中的元素。例如，可以使用以前介绍的下标运算符或迭代器(智能指针)，在这两种机制中，元素都可以按顺序访问或随机访问。还可以通过容器的函数成员访问特定的元素，如图 20-5 所示。

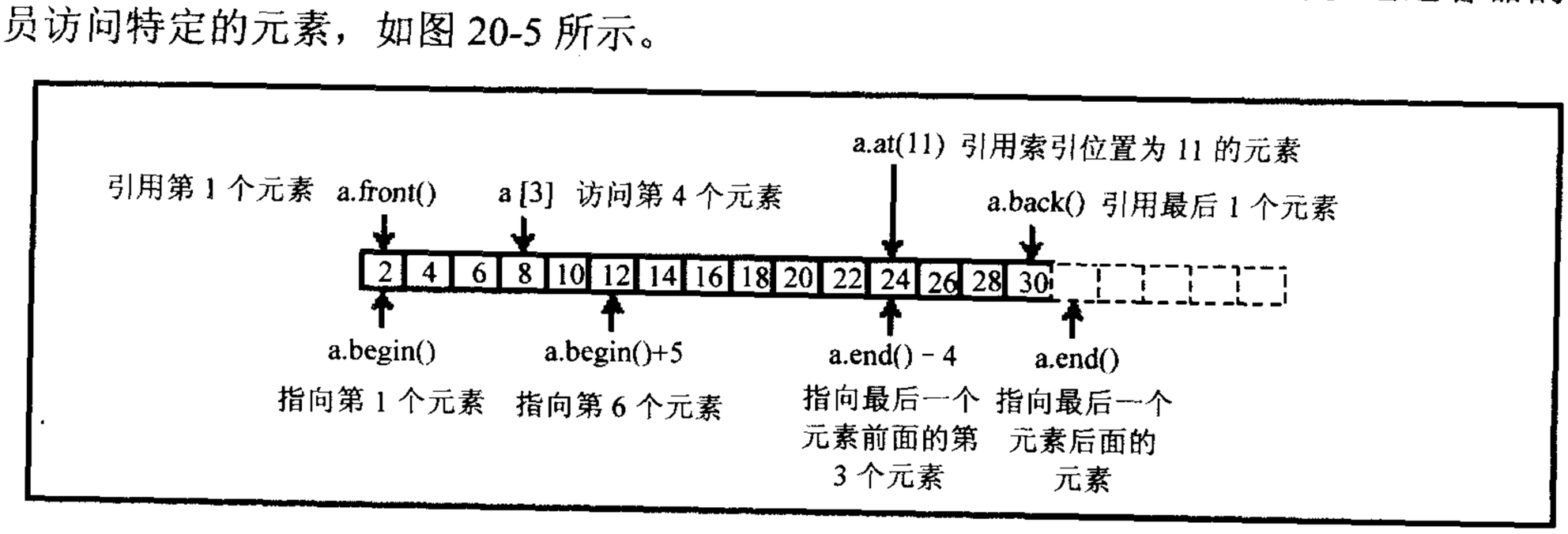


图 20-5 访问 vector 容器中的元素

图 20-5 显示的 vector 容器存储 int 类型的元素。在该容器中，前 15 个元素存储了值，而其容量是 20。vector 对象的 begin()成员返回 iterator 类型的对象，它指向第一个元素。iterator 类型在 vector 对象中定义为指向所存储对象的类型指针。end()成员返回一个迭代器，指向 vector 中最后一个元素后面的元素。使用这些迭代器可以遍历 vector 中的元素——只需递增 begin()成员返回的迭代器，直到它等于 End()成员返回的迭代器为止。下面是一个例子：

```
std::vector<double> pies(20, 3.14159);    //Vector with 20 elements
double sum_pies = 0.0;
for(std::vector<double>::iterator iter = pies.begin(); iter != pies.end(); iter++)
  sum_pies += *iter;                      //Add the object pointed to by iter
```

这段代码使用 begin()成员函数返回的迭代器，求出了所创建的 Vector 对象 pier 的元素总和。变量 iter 用于存储元素，其类型是 std::vector<double>::iterator，即在容器类中定义的迭代器类型。在 for 循环中递增迭代器，直到它等于 End()成员返回的迭代器为止。为了访问元素，并把它加到 sum_pies 中，只需解除迭代器的引用，采用的方式与解除普通指针的引用相同。

当然，还可以使用下标运算符[]来实现上面的循环：

```
for(int i = 0; i<pies.size(); i++)
  sum_pies += pies[i];
```

对从 vector 容器中获得的迭代器，还可以进行算术运算，其结果与对普通指针执行算术运算相同。给 begin()函数返回的迭代器加 5，得到的指针会指向第五个位置的元素。还可以对迭代器使用递增和递减运算符，其方式与普通指针相同。要访问一个元素，可以使用下标运算符。在图 20-5 中，a[4]访问 vector 容器中的第五个元素，表达式*(a.begin()+4)也会访问 vector 容器中的第五个元素，因为 begin()成员返回的迭代器指向 vector 容器中的第一个元素。

front()、back()和 at()成员函数返回元素的引用；在希望直接使用元素时，可以使用它们或下标运算符。下面进行演示。重新编写计算 pies 元素的和的循环，如下所示：

```
double sum_pies = pies.front();
for(int i = 0; i<pies.size(); i++)
  sum_pies += pies.at[i];
```

反向迭代器

vector 容器还可以提供反向迭代器，从后向前遍历元素。成员 rbegin()返回 reverse_iterator 类型的迭代器，它指向 vector 容器中的最后一个元素。递增这个迭代器会把指针移向上一个元素。rend()成员函数返回的迭代器指向 vector 容器中第一个元素前面的位置。使用反向迭代器重新编写 pies 循环：

```
for(std::vector<double>::reverse_iterator iter = pies.rbegin()
                                          ; iter != pies.rend(); iter++)
  sum_pies += *iter;                      //Add the object pointed to by iter
```

这看上去非常类似于前面的循环，但过程完全不同，从最后一个元素开始，向前累加元素。

20.2.3 vector 容器的基本操作

与一般数组相比，使用 vector 容器有非常大的灵活性。例如，可以在 vector 的最后添加新元素，删除最后一个元素。容纳元素所需要的存储空间总是自动管理的。还可以在 vector 容器的中间插入或删除元素——只要能接受所涉及的系统开销。vector 添加或删除元素的基本操作如表 20-3 所示。

表 20-3 添加或删除元素的操作

操　作	说　明
push_back()	把传送为参数的对象添加到 vector 的最后
pop_back()	删除 vector 尾部的对象
erase()	删除一个或多个元素
clear()	删除所有的元素
insert()	插入一个或多个对象。这个功能是通过两个成员函数和一个函数模板提供的。函数模板可以把另一个容器的元素插入 vector。为了插入一个元素，第一个参数必须是一个指定插入位置的迭代器，第二个参数必须是要插入的对象。如前所述，在使用 vector 容器时，最好避免这个操作

empty()成员函数可以避免访问空 vector 中的元素。如果 vector 不包含元素，该成员就返回 true。

要理解这些操作的工作原理，可以在一个例子中试用它们。

程序示例 20.2——试用 Vector 容器

这个程序执行了前面介绍过的所有简单 vector 操作。在 vector 对象中添加并删除一些元素，以及查看 vector 内容的各种方式。

在本例中，函数 show_sequence()用于演示使用 copy()算法和流迭代器来输出结果。inspect_vector()函数用于在不同的地方输出 vector 的内容，它通过迭代器参数调用 show_sequence()函数：

```
// Program 20.2 Manipulations of the vector<> container   File: prog20_02.cpp
#include <iostream>
#include <vector>     // For the vector container
#include <algorithm>     // For the copy() function
using std::cout;
using std::endl;
using std::vector;

// Display a sequence of elements
void show_sequence(vector<int>::const_iterator first,
                                vector<int>::const_ iterator last) {
  cout<< "{ ";
  std::copy(first, last, std::ostream_iterator<int>(cout, " "));
```

```
  cout <<"}" <<endl;
}

// Display the contents of a vector
void inspect_vector( const vector<int>& v) {
  cout<< " vector has " <<v.size() <<" elements:"
  show_sequence(v.begin() , v.end());
}

int main() {
vector<int> v;          // Create empty vector
cout<<"new vector created"<<endl;
inspect_vector(v);

cout << "filling vector from array" << endl;
int values[]={1, 3, 7, 5};
v.insert(v.end() , values+1, values+3);       // Insert two elements
inspect_vector(v);

cout <<"appending value 5"<<endl;
v.push_back(5);         // Add an element at the end
inspect_vector(v);

cout << "erasing element at offset 1"<< endl;
v.erase(&v[1]);        // Remove the second element
  inspect_vector(v);

  cout <<"inserting element 4 at offset 1" <<endl;
  v.insert(v.begin() +1, 4);      // Insert an element
  inspect_vector(v);

  cout <<"clearing all elements"<<endl;
  v.clear();                  // Delete all elements
  inspect_vector(v);

  return 0 ;
}
```

此输出列出了在每个 vector 操作后 vector 对象的内容：

```
new vector created
  vector has 0 elements: { }
filling vector from array
  vector has 2 elements: {3 7}
appending value 5
  vector has 3 elements: {3 7 5}
erasing element at offset 1
  vector has 2 elements: {3 5}
inserting element 4 at offset 1
  vector has 3 elements: {3 4 5}
clearing all elements
  vector has 0 elements: { }
```

可以看出，vector 容器适合于跟踪它自己的内容。而使用数组，就必须以比较困难的方式来显示这些元素。

例子的说明

代码首先是包含两个 STL 头文件的#include 指令：

```
#include <vector>                    //For the vector container
#include <algorithm>                 //For the copy() function
```

vector 头文件定义了 vector 容器及其所有的内置操作。algorithm 头文件提供了可以应用于容器的操作，这里使用 copy()函数。

接着，定义函数 show_sequence()，以显示一组元素：

```
void show_sequence(vector<int>::const_iterator first,
                                   vector<int>::const_iterator last) {
  cout<<"{";
  std::copy(first, last, std::ostream_iterator<int>(cout," "));
  cout<<"}"<<endl;
}
```

传送给函数的值由两个 vector<int>const_iterator 类型的迭代器定义为一个半开间隔。这个类型是 vector 中元素的一个 const 迭代器类型，如果 inspect_vector()函数的参数没有声明为 const，就可以为 show_sequence()的参数使用 vector<int>iterator 类型。iterator 类型在 vector<T>模板中用 typedef 定义为一个指向 T 类型对象的指针。const_iterator 类型定义为指向 const T 的迭代器类型。

show_sequence()函数使用 STL 算法 copy()，给 cout 输出通过一对指针引用的值间隔符。copy()的前两个参数是迭代器，它们定义了要复制元素的容器中的源间隔符，复制操作从 first 开始，一直向前，直到 last 为止，但不包括 last。copy()函数的前两个参数接受任何迭代器。

copy()的第三个参数是指定元素复制的目标位置的迭代器。在这个例子中，为目标位置提供了表达式 ostream_iterator<int>(cout, " ")创建的输出迭代器。第一个参数 cout 是应用这个迭代器的流，第二个参数是用于分隔所传送元素的分隔符。copy()算法把每个源元素赋予目标对象，但并不知道它处理的目标对象是什么类型。copy()函数仅把要复制的对象传送给第三个参数指定的迭代器，并假定迭代器知道应把它们放在什么地方。这样，copy()函数就独立于源对象和它复制的目标对象。在本例中，目标对象会回送复制到标准输出流 cout 中的每个值。

inspect_vector()函数调用 show_sequence()函数，输出 vector 的全部内容：

```
void inspect_vector(const vector<int>& v) {
  cout<<" vector has"<<v.size()<<" elements:";
  show_sequence(v.begin(), v.end());
}
```

注意这里把 vector 对象传送为惟一的参数。不需要传送第二个参数，说明 vector 包含多少个元素，因为调用 size()成员总是可以获得 vector 的元素个数。size()函数返回的值是 size_type 类型，是一个无符号的整数类型。

调用 vector 对象的 begin()和 end()成员可以获得该对象中第一个元素和最后一个元素的迭代器，把这些迭代器传送给 show_sequence()函数。

如前所述，v.begin()指向序列中的第一个元素。使用表达式*v.begin()、v[0]或 v.front()都可以访问序列的第一个元素。一般应选择适合当前工作情况的表达式。

当然，不能访问 end()返回的迭代器所指向的元素，因为它指向容器中最后一个元素后面的地址。但是，表达式*(v.end() - 1)是访问序列中最后一个值的一种合法而有效的方式。也可以使用表达式*(v.begin()+v.size() - 1)、v[v.size() - 1]或 v.back()访问序列中的最后一个元素。

在尝试访问元素之前，要确保容器有元素。使用 v.empty()成员可以检查容器是否有元素。当 vector 对象 v 不包含元素时，它返回 true。

前面的程序使用 vector 实例 vector<int>，得到一个存储 int 值的 vector 容器：

```
vector<int> v;              //Create empty vector
```

这里调用默认的构造函数，把 v 声明为类模板 vector<int>的一个实例。从程序输出中可以看出，新的 vector 对象开始时处于空状态。

把值放到新 vector 对象中有许多方式。添加元素的一种最高效的方式是使用 insert()成员函数，该函数可以把整个序列复制到 v 中，如下面的例子：

```
cout<<"filling vector from array"<<endl;
int values[]={1,3,7,5};
v.insert(v.end(), values+1, values+3);          //Insert two elements
inspect_vector(v);
```

这里 STL 再次把序列定义为间隔，用一对迭代器分隔开。insert()函数的第一个参数 v.end()是要插入新元素的位置指针。注意，这里使用了 past-the-end 值，在添加到 vector 的尾部时使用这个地址是合法的。

insert()函数的第二个和第三个参数是表达式 values+1 和 values+3，它们指定了要插入的原序列的范围。这说明可以使用普通的指针代替迭代器。原序列是数组 values，这里仅复制第二和第三个元素，即从 values+1 开始，到 values+3 前面的那个元素为止。间隔在 STL 中一般像这样指定：间隔包括第一个位置，但不包含最后一个位置，这称为半开间隔。可以写作[begin, end)，其中方括号表示间隔在左端是封闭的，圆括号表示在右端是打开的。因此，间隔包括 begin 位置，但不包括 end 位置。

注意：

在指定[begin, end]间隔时，例如用表达式 values+1 和 values+3 指定的间隔，应确保 end 表达式比 begin 表达式大，两个迭代器或两个指针都关联着同一个源对象。这是使用迭代器指定间隔时产生错误的常见原因。

下面回过头来讨论程序示例 20.2。使用 vector 的 push_back()成员添加一个元素：

```
v.push_back(5);             //Add an element at the end
```

这个语句把 5 添加为一个新元素，放在 vector 对象 v 的最后。

接着试验 erase()操作：

```
v.erase(&v[1]);             //Remove the second element
```

这个语句把要删除的元素地址传送给 erase()函数，从 vector 对象 v 中删除第二个元素。这

里使用了 erase()的最简单形式，带一个迭代器参数，指定要从 v 中删除的一个元素。一般情况下，可以使用指针参数来代替迭代器，表达式&v[1]是 v 中第二个元素的地址。执行这个语句后，第二个元素就从 v 中删除了，其值为 7。

接着，下面的语句插入一个新元素：

```
v.insert(v.begin()+1, 4);              //Insert an element
```

前面已经使用过 insert()，那个 insert()函数带有两个迭代器参数。在上述这种形式的 insert()中，两个参数分别是迭代器和整数。我们使用表达式 v.begin()+1 指定容器的内部位置，然后在这个目标位置上插入新元素，即数值 4。该元素目前占据着目标位置，其后的所有元素必须向后移动合适的字节数，以容纳新元素。vector 对象会根据需要分配新的空间，复制元素，以使元素有组织地排列起来。

注释：

这个自动复制是非常方便的。但是，如果 vector 非常大，在容器中比较靠前的位置执行的插入和删除操作又比较多，则程序肯定会慢如蜗牛。

最后使用 clear()清除 vector 对象中的所有内容：

```
v.clear();                  //Delete all elements
```

但是，这是不必要的。在 vector 实例 v 超出了作用域(位于封闭块的端部)时，v 及其内容会由 v 的析构函数自动清除掉。实际上，v.clear()是等价于表达式 v.erase(v.begin(), v.end())。

20.2.4 使用 vector 容器进行数组操作

下面讨论第 8 章在论述函数时提到的一个例子。程序示例 8.5 中有一个 average()函数，它带有 array[]参数：

```
//Function to compute an average
double average(double array[], int count) {
  double sum=0.0;                  //Accumulate total in here
  for(int i=0; i<count; i++)
   sum += array[i];                //Sum array elements

  return sum/count;                //Return average
}
```

在 STL 出现之前，所有的 C 和 C++程序都以这种方式编写。第 8 章论述过这个函数，double array[]和 double* array 在这种情况下表示相同的含义，所以 average()函数的第一个参数是一个指针。由于数组不知道它包含了多少个元素，因此需要传送另一个参数，表示数组的元素个数(如果数组的元素没有完全填满，函数需要给数组添加元素，就需要给函数传送数组的大小和已有的元素个数)。

下面再看看程序示例 20.2 中的 show_sequence()函数。这个函数也把指针以迭代器的形式作为第一个参数，但使用完全不同的形式。第二个参数也是一个指针：

```
void show_sequence(vector<int>::const_iterator first,
```

```
                                    vector<int>::const_iterator last) {
  cout<<"{";
  copy(first, last, ostream_iterator<int>(cout,' '));
  cout<<"}"<<endl;
}
```

first 和 last 之间的间隔是半开间隔：first 包含在间隔中，而 last 不包含在间隔中。但是，这与使用一对索引值完全不同，索引值只表示一个位置，而迭代器包含了所指向的对象类型及其位置。

使用间隔指定 vector 中的一个子集不同于使用数组和变量 count 表示元素个数，但仍旧很容易确定其中包含多少个元素：

```
int count= last- first;
```

为什么 STL 要使用间隔，而不使用数组和元素个数？这有几个原因：第一，在使用数组完成许多操作时，利用指针比较方便；第二，使用指针比使用索引高效得多。第三，也是最重要的一点，迭代器包含了对象的类型和位置，所以算法可以使用迭代器，应用于任何源中任意类型的对象。

1. 使用迭代器计算平均值

给 average()函数定义一个模板，使用迭代器代替数组。该模板的代码如下：

```
template <typename Iter>
double average(Iter begin, Iter end) {
  double sum=0.0;
  int count=end-begin;
  for(begin != end)
   sum += *begin++;
  return sum/(count == 0.0 ? 1.0 : count);
}
```

注意在参数 begin 上执行的几个操作，如表 20-4 所示：

表 20-4　在参数 begin 上执行的操作

操　作	作　用
end-begin	计算差，即对象个数
begin != end	检查最后一个值
*begin	获取当前值
begin++	前进到下一个值

在上面的操作中，后两个操作可以合并为一个表达式*begin++。这样编写模板代码的方式暗示，运算符*和后缀++必须应用于 Iter 类型，即指针类型或迭代器。使用迭代器可以大大简化代码。下面看一个例子。

程序示例 20.3——用模板迭代器计算平均值

下面的程序使用模板计算平均值：

```
// Program 20.3 Computing the average function with template iterators
//File: prog20_03.cpp
#include <iostream>
#include <vector>
using std::cout;
using std::endl;
using std::vector;

template <typename Iter>
double average (Iter begin, Iter end) {
  double sum = 0.0;
  int count = end - begin;
  while( begin != end)
    sum += *begin++;
  return sum/(count == 0.0 ? 1.0 : count);
}

int main() {
  double temperature [] = { 10.5, 20.0, 8.5 };
  cout << "array average<< "
       << average (temperature,
                   temperature+sizeof temperature/ sizeof temperature [0] )
       << endl ;

  vector<int> sunny;
  sunny.push_back(7);
  sunny.push_back(12);
  sunny.push_back(15);
  cout << sunny.size()<< " months on record" <<endl;
  cout << "average number of sunny days: ";
  cout << average(sunny.begin(), sunny.end())<< endl;

  return 0;
}
```

这个程序的结果如下：

```
array average = 13
3 months on record
average number of sunny days: 11.3333
```

例子的说明

模板的一个优点是可以获得函数的任意多个版本。这个例子生成了函数模板 average 的两个实例。

average 的第一个实例化是为存储在一般 C++数组中的值计算平均值：

```
double temperature[]={ 10.5, 20.0, 8.5 };
cout << "array average = "
     << average(temperature,
                temperature + sizeof temperature/ sizeof temperature[0])
     << endl;
```

实例在输出语句中创建。模板实例的类型在函数参数中隐式确定。第二个参数必须是要包含的最后一个元素后面的元素指针——把数组中的元素个数加到指针 temperature 上，计算出该元素指针。

average 的第二个实例化是为存储在 int 类型的 vector 容器中的值计算平均值。首先声明 vector<int>对象 sunny：

```
vector<int> sunny;
```

接着下面的语句生成函数模板的实例：

```
cout<<average(sunny.begin(), sunny.end())<<endl;
```

sunny.begin()和 sunny.end()返回的值是 vector<int>::iterator 类型，这个类型用于实例化模板的第二个版本。结果，模板函数 average()的两个实例的函数签名如下：

```
double average<double*>(double* begin, double* end);
double average<vector<int>::iterator>(vector<int>::iterator begin,
                                      vector<int>::iterator end);
```

从输出中可以看出，平均值的计算按照预期的那样进行。函数处理 vector 容器中的元素，与处理普通数组的方式相同。

函数模板 average()的这个版本在大多数情况下工作正常，但不能用于流迭代器，因为可以从流中获取的对象个数在事先是未知的，所以不能计算出 end - begin。此时，必须在循环中给 count 累加总和。当然，这等价于这里的代码。

2. 模板数组

在程序示例 20.3 中，使用了模板的间隔约定。当然，也可以通过数组表示法来利用模板。

程序示例 20.4——用模板数组计算平均值

下面是程序示例 20.3 的另一个版本：

```
// Program 20.4 Computing an average with template arrays   File: prog20_04.cpp
#include <iostream>
#include <vector>
using std::cout;
using std::endl;
using std::vector;

template <typename Array>
double average (Array a, long count) {   // Array can be a pointer or an iterator
  double sum = 0.0;
  for (long i = 0L; i < count; ++i)
    sum += a[i];
  return sum/(count == 0.0 ? 1.0 : count);
}

int main() {
  double temperature[] = { 10.5, 20.0, 8.5 };
```

```
// second argument is now the count
cout << "array average = "
     << average(temperature, sizeof temperature/sizeof temperature[0])
     << endl;

vector<int> sunny;
sunny.push_back(7);
sunny.push_back(12);
sunny.push_back(15);
cout << sunny.size()<< " months on record"<<endl;
cout << "average number of sunny days: " ;
cout << average(sunny.begin(), sunny.end() - sunny.begin())<< endl;
return 0;
}
```

这个程序的结果跟程序示例 20.3 相同。代码看起来更简单。注意，现在对元素 a 只需要执行一个操作，即 a.operator[](long)。从这个表示法可以看出，下标运算符用 long 类型的参数来表示索引值。在这个版本中，还必须创建自己的循环变量 i——在每次迭代时必须递增它，并与 count 比较。在以前的版本中，根本不需要这个变量。

20.2.5 使用输入流迭代器

下一个例子将说明如何把模板算法同不寻常的迭代器类型合并起来。前面已经介绍过 ostream_iterator，这次使用其补码 istream_iterator，从输入流中获取数据。这两个流迭代器都是在<iostream>头文件中定义的。

程序示例 20.5——istream_iterator<>

在程序示例 20.3 和 20.4 中，对数组或容器中已有的值执行了 average()函数。这次，average()函数将使用函数模板的另一个实例，直接从输入流中获取数值。本例再次使用程序示例 20.3 中的 average()函数模板：

```
// Program 20.5 Taking the average of values from a stream    File: prog20_05.cpp
#include <iostream>
#include <iterator>      // For the istream_iterator <> template
using std::cout;
using std::endl;
using std::cin;
using std:: istream_iterator;

template <typename Iter>
double average (Iter begin, Iter end) {
  double sum = 0.0;
  int count = 0 ;
  for ( ; begin != end ; ++count)
    sum += *begin++;
  return sum/(count == 0.0 ? 1.0 : count);
}
```

```
int main () {
  cout <<"Enter some real numbers separated by whitespace - spaces, "<< endl
       <<"tabs or newline. Then press the special key sequence " <<endl
       <<"that marks the end-of-file (Ctrl-Z on a PC)" <<endl;
  double av = average(istream_iterator<double>(cin) , istream_iterator<double> ());
  cout<< "The average value is " <<av<< endl;
  return 0 ;
}
```

注释:

在运行这个例子时，注意要对关闭输入流使用正确的约定。在 PC 中，完成这一操作的组合键是 Ctrl+Z，在 Unix 中，应使用 Ctrl+D，接着按下回车键。公司在开发通用的编程环境时，需要考虑各个方面的问题。如果第一次没有成功地关闭流，就应多按几下 Ctrl+Z。

下面是程序的输出：

```
Enter some real numbers separated by whitespace - spaces,
tabs or newline. Then press the special key sequence
That marks the end-of-file (Ctrl-Z on a PC)
 3.6
 4.5
 5.7
 ^Z
The average value is 4.6
```

一些系统的显示结果可能与此不同。

例子的说明

下面的语句生成了模板的一个实例：

```
double av=average(istream_iterator<double>(cin), istream_iterator<double> ());
```

这个语句把一些看起来很古怪的表达式传送到 average()函数中。下面把这个语句用另一种方式表达，更清楚地表示出来：

```
istream_iterator<double> begin(cin);
istream_iterator<double> end;
double av=average(begin, end);
```

对象 begin 是模板类 istream_iterator<double>的一个实例。流对象 cin 传送给构造函数，这样迭代器就从标准输入流中读取。

对象 end 也是 istream_iterator<double>的一个实例。用类 istream_iterator<double>的默认构造函数创建 end。变量 end 以这种方式创建时，其运行方式类似于对应于 end-of-file 的 past-the-end 值。

begin 和 end 的新版本肯定不是指针，它们是类对象，那么为什么这是可行的？因为这些类对象是迭代器，这表示它们可以像指针那样运行。它们还根据 average()函数提供了三个运算符，如表 20-5 所示：

表 20-5　average()函数需要的运算符

运 算 符	作　　用
bool Iter::operator!=(Iter end_value);	比较
double Iter::operator*();	解除引用
iter Iter::operator++(int);	递增

这次，把它们显示为类函数，因为它们的确是类函数。它们是 istream_iterator<double>类中的重载运算符函数。operator++(int)并不带有 int 参数——这只是为了让编译器知道指定了递增运算符的后缀版本。istream_iterator<double>类型的任何对象都可以执行这些操作。

当然，不能使用程序 20.3 中 average 函数模板的版本计算 count 的机制。在调用模板函数时，输入值的个数是未知的，所以 begin 和 end 迭代器的差不能在此时计算。实际上，这类操作是不会编译的，因为没有使用这些迭代器的 operator-()函数。这里必须在循环中递增变量的值来确定 count。这个版本在函数中的工作略多，但等价于程序 20.3 中的代码，所以这个版本更一般化。

1. 带有功能的 istream_iterator 迭代器

上面的 istream_iterator 对象假装为指针，就像程序示例 14.6 中的 BoxPtr 类对象一样。但是，这些对象的功能很强——它们可以提供与流 I/O 运算符一样复杂的功能。这是一个基本概念，有助于使 STL 成为一个功能强大的库。一旦把模板函数与迭代器联合起来，就可以从任何源中捕获间隔。而使用旧的数组表示法无法捕获间隔。

程序示例 20.6——istream_iterator<>的更多功能

为了强调迭代器的一般本质，下面是最后一个例子。这次对包含在一个字符串中的值使用半开间隔，通过迭代器访问为一个流。

```
// Program 20.6 Taking the average of values from a string stream.
//File: prog20_06.cpp
#include <iostream>
#include <iterator>
#include <sstream>      // For the istringstream class
using std::cout;
using std::endl;
using std:: istream_iterator;
using std:: istringstream;
using std::string;

template <typename Iter>
double average(Iter begin, Iter end) {
  double sum = 0.0;
  int count = 0 ;
  for ( ; begin ! = end ; ++count)
    sum + = *begin++;
  return sum/(count == 0 ? 1 : count);
}
```

```
int main () {
  string stock_ticker = "4.5 6.75 8.25 7.5 5.75";
  istringstream ticker(stock_ticker);
  istream_iterator<double> begin(ticker);
  istream_iterator<double> end;

  cout << "Readings: " <<stock_ticker
       << "Today's average is"
       << average (begin, end)<< endl;
  return 0 ;
}
```

这个程序的结果如下：

```
Readings: 4.5 6.75 8.25 7.5 5.75.
Today's average is 6.55
```

例子的说明

仔细查看，这个例子几乎跟从流 cin 中读取数据的程序示例 20.5 完全相同。区别是，为了替代 cin，用类 istringstream 创建了自己的流 ticker：

```
string stock_ticker="4.5 6.75 8.25 7.5 5.75";
istringstream ticker(stock_ticker);
```

初始化输入流 ticker，它直接从程序字符串 stock_ticker 中提取其值。接着把流对象传送给 istream_iterator<double>构造函数：

```
istream_iterator<double> begin(ticker);
```

新流 ticker 的一个优点是，它可以在使用 cin 的大多数场合中替换 cin—— istream_iterator 非常烦琐。它可以代替文件流 cin，提取 istringstream 对象 ticker 或字符源。其余代码保持不变，包括 average()模板。

2. 迭代器 average()

average()函模板数的最新版本非常灵活。如果在程序示例 20.3~20.6 中都使用了相同的定义，就创建了模板的三个实例：

```
double average<double*>(double*, double*);
double average<int*>(int*, int*);
double average<istream_iterator<double>>
                    (istream_iterator<double>, istream_ iterator<double>);
```

显然，前两个实例处理简单的指针，而第三个版本处理 istream_iterator 类对象。例如，在程序示例 20.3 中，把 vector<int>容器中的元素传送给 average<int*>版本。在程序示例 20.5 和 20.6 中还使用了 istream_iterator，它允许从控制台上和字符串中直接读取数值。指针版本包括基本 C++数组和 vector 容器，而 istream_iterator 版本包括文件和字符串。对于迭代器来说，我们还仅谈及它的皮毛。

迭代器类对象看起来仍相当神秘，下面暂停一下，先深入了解迭代器类对象的工作方式。之后就不会觉得创建自己的迭代器类非常困难了。

20.3 创建自己的迭代器

第 14 章在 BoxPtr 类中介绍了一个非常简单的迭代器，但在涉及 STL 时，迭代器会比较复杂。从头开始建立自己的迭代器，将有助于理解 STL 迭代器的工作方式。首先从最简单的迭代器开始，再扩展它，直到达到需要的程度为止——使自己的迭代器能成为 STL 迭代器的一个成员，这样自己的迭代器就可以和 STL 迭代器一起使用了。

程序示例 20.7——创建自己的迭代器

首先从最简单的迭代器开始，迭代整数。下面是 Integer 类定义的代码：

```
// Integer.h  Integer class definition
#ifndef INTEGER_H
#define INTEGER_H
#include <iostream>
using std::cout;
using std::endl;

class Integer {
  public:
    Integer (int arg = 0) : x(arg) {}

    bool operator != (const Integer& arg) const {    // Comparison !=
      if (x == arg.x)       // Debugging output
        cout << endl
             << "operator != returns false"
             << endl;  // Just to show that we are here
      return x != arg.x;
    }

    int operator* () const { return x; }        // De-reference operator

    Integers operator ++() {                    // Prefix increment operator
       ++x;
       return * this ;
    }

  private:
    int x;
};
#endif   //INTEGER_H
```

下面是使用 Integer 对象的程序代码：

```
// Program 20.7 Using the Integer iterator class    File: prog20_07.cpp
#include <iostream>
using std::cout;
using std::endl;

int main () {
```

```
  Integer begin(3);
  Integer end(7);
  cout <<"Today's integers are: ";
  for( ; begin != end ; ++begin)
    cout<<*begin<< " ";
  cout<<endl;
  return 0;
}
```

运行这个例子，得到如下结果：

```
Today's integers are: 3 4 5 6
operator! = returns false
```

例子的说明

这个程序的工作方式是：把 begin 和 end 声明为普通的整数，并给它们赋予相同的值。一个主要的区别是，在重载的!=运算符返回 false 时，Integer 要创建一个输出行。给 Integer 类定义构造函数：

```
Integer(int arg=0):x(arg){}
```

Integer 类有一个数据成员 x，它存储了当前值。如果在声明 Integer 对象时没能初始化它，数据成员 x 就默认为 0。

这个类给!=定义运算符函数：

```
bool operator!=(const Integer& arg) const {      //Comparison !=
  if (x==arg.x)                                  //Debugging output
    cout<<endl
        <<"operator! = returns false"
        <<endl;        //Just to show that we are here
  return x != arg.x;
}
```

这个函数把 Integer 对象作为一个参数，返回一个布尔值。从程序输出可以看出，只要 begin 等于 end，!=运算符就返回 false，循环也停止了。

解除引用运算符的定义如下：

```
int operator*() const{return x;}          //Dereference operator
```

这个运算符返回当前 Integer 对象的数据成员 x，即解除 Integer 对象的引用。这个函数与 operator!=()函数都声明为 const，是因为它没有改变对象的内部状态。

前缀递增运算符的定义如下：

```
Integer& operator++() {                   //Prefix increment operator
  ++X;
  return *this;
}
```

提示：

在前面例子的 average()算法中，使用了表达式*begin++，它替代了后缀 operator++(int)。

Integer 类现在有 4 个简单的函数。实际上，有了这 4 个函数，就可以使该类像指针那样执行某些操作了。在函数 main()中，使用了下面的语句：

```
for(; begin != end; ++begin)
  cout<<*begin<" ";
```

这个语句看起来很熟悉。在 average()算法中也使用了类似的语法，但在 average()算法中使用的是*begin++。实际上，这里在输出语句中用*begin++代替*begin 是不行的，因为没有定义运算符函数 operator++(int)，该函数支持 Integer 类的后缀递增表示法。在输出行中使用*begin，而在循环控制中使用++begin，其含义是：

```
for(; begin != end;)
  cout<<*++begin<" ";
```

循环输出 4 到 7 的值。*++begin 的返回值是数据成员 begin.x 执行递增后的值。

20.3.1 给算法传送迭代器

理想情况下，我们希望使用 Integer 类型定义可以传送给算法的迭代器，例如 average 模板提供的算法。average 模板的实例要求为传送给它的迭代器实现后缀递增运算符。该类的运算符函数定义如下：

```
const Integer operator++(int) {      // Postfix ++ operator
  Integer temp(*this);               // save our current value
  ++x;                               // change to new value
  return temp;                       // return unchanged saved value
}
```

这个运算符的实现非常简单，这足够 Integer 对象使用 average 模板的实例了。下面举例说明。

程序示例 20.8——给 average<>传送 Integer 对象

下面是 Integer 类的改进版本：

```
// Integer.h  Integer class definition
#ifndef INTEGER_H
#define INTEGER_H
#include <iostream>
using std::cout;
using std::endl;

class Integer {
  public:
    Integer (int arg =0) : x(arg) {}

    bool operator!=(const Integer& arg) const {   // Comparision !=
      return x != arg.x;
    }

    int operator*() const { return x; }     // Dereference operator
```

```
Integer& operator++() {       // Prefix increment operator
  ++x;
  return *this;
}

  const Integer operator++(int) {     // Postfix ++ operator
  Integer temp(*this);       // save our current value
  ++x;          // change to new value
  return temp;         // return unchanged saved value
}
private:
  int x;
};
#endif       //INTEGER_H
```

使用这个 Integer 类版本的程序如下所示：

```
// Program 20.8 Averaging values from Integer    File:prog20_08.cpp
#include <iostream>
#include "Integer.h"
using std::cout;
using std::endl;

template <typename Iter>
double average(Iter a, Iter end) {
  double sum = 0.0;
  int count = 0;
  for( ; a != end ; ++count)
    sum += *a++;
  return sum/count;         // Lets bad things happen when count==0
}

int main() {
  Integer first(1);
  Integer last(11);
  cout<<"The average of the integers from" <<*first<<" to "<<*last-1;
  cout<<" is "<< average(first, last) << endl;
  return 0;
}
```

这个程序的结果如下：

```
The average of the integers from 1 to 10 is 5.5
```

例子的说明

Integer 类的改动不大。除了删除 operator!=()的调试语句之外，只是添加了 operator++()的后缀版本：

要实现后缀递增运算符，必须创建当前对象的副本，而前缀形式则不必创建该副本。当然，这个函数需要执行额外的操作，因为它调用了当前对象类型的副本构造函数。所以，后面的大多数例子尽可能使用前缀形式++iter，因为这比较高效。这对 Integer 类并没有什么影响，但在通过模板算法执行操作时，创建表达式 iter++所涉及的对象副本可能比较昂贵。

在下面的语句中使用 Integer 类：

```
Integer first(1);
Integer last(11);
cout<<"The average of the integers from "<<*first<<"to"<<*last-1;
cout<<"is"<<average(first, last)<<endl;
```

Last 是一个 past-the-end 值。必须从中减去 1，才能得到包含范围。

如果不再继续，则通过新迭代器 Integer 调用 average()会执行完全不同的操作。在其他例子中，用于平均的值都跟一个源相关——它们都存储在数组、vector 容器或流中。在这个例子中，值是不存在的。在 average()算法调用 operator!=()、operator*()和 operator++()时，它们在 Integer 类的内部进行计算。

20.3.2 STL 迭代器类型的要求

对于简单的要求，类 Integer 作为迭代器已足够了。但是，STL 对成为迭代器的对象提出了许多特殊的要求，以确保接受迭代器的所有算法都可以按预期的方式工作。不同的算法要求迭代器有不同的功能，每个迭代器对于不同的类别(输入和输出迭代器、向前迭代器、双向迭代器和随机访问迭代器)都有不同的要求。在需要某种功能的迭代器时，我们总是使用功能比较强的迭代器。下面把类 Integer 变成一个随机访问迭代器。

一般情况下和在 STL 中，模板编程的一个问题是，在使用之前并不总是知道需要使用的所有类型。考虑下面的例子：

```
template <typename Iter>
void swap(Iter& a, Iter& b) {
  tmp= *a;    //error--variable tmp undeclared
  *a=*b;
  *b=tmp;
}
```

这个模板函数要交换迭代器 a 和 b 表示的两个对象。tmp 应是什么类型？无从知晓。我们知道它的类型是迭代器指向的类型，但不知道它究竟是什么类型，因为这是在运行期间确定的。如何声明一个不知道其类型的变量？

一个简单的解决方法是确保每个迭代器都为 value_type 类型的返回值包含一个 public 类型定义，对应于和迭代器相关的对象类型。在 swap 模板函数中使用这个方法，如下所示：

```
template <typename Iter>
void swap(Iter& a, Iter& b) {
  typename Iter::value_type tmp= *a;     // Much better - but still not good enough
  *a=*b;
  *b=tmp;
}
```

对于大多数 STL 迭代器来说，这是可行的。但是，如果 Iter 是指针类型，如 int*(常常会遇到这种情形)，这种方法就失效了。问题是不能简单地编写 int*::value_type——指针是内置的语言类型，不包含内部类型定义。

STL 通过模板 iterator_traits 解决了这个问题以及其他相关问题，该模板使用了迭代器标记和一组需要的迭代器类型。如果把它应用于 swap 模板，如下所示：

```
template <typename Iter>
void swap(Iter& a, Iter& b) {
  //cumbersome, but always works
  typename iterator_traits<Iter>::value_type tmp= *a;
  *a=*b;
  *b=tmp;
}
```

这把 tmp 的类型指定为用参数类型 Iter 实例化的模板 iterators_traits 定义的 value_type。这样，模板 iterators_traits 是否工作取决于参数 Iter 是否为一个指针。该模板为指针定义模板 iterators_traits 的一个规范，允许参数是一个普通的指针或迭代器。如果 Iter 是 T 类型的指针，就应用模板的规范，iterator_traits<T*>:: value_type 形式等价于对象类型 T。这是在模板规范中定义的。如果 Iter 是一个迭代器，该迭代器是类型为 T 的类对象，就采用标准模板。此时，iterator_traits<T>::value_type 形式等价于 T:: value_type，也就是迭代器对象中 value_type 的定义。

因此，要使 Integer 迭代器处理 iterator_traits，就必须包含 value_type 的一个内部 public 类型定义。最简单的方式是使用 STL 提供的 iterator 模板，作为 Integer 类的基础。这还涉及到其他类型的定义。下面详细论述。

1. 使用 iterator 模板

STL 把 iterator 模板定义为结构的模板，它有 5 个参数，如下所示：

```
template<class Category, class T, class Distance = ptrdiff_t,
                  class Pointer = T*, class Reference = T&>
struct iterator{
typedef T value_type;
typedef Distance difference_type;
typedef Pointer pointer;
typedef Reference reference;
typedef Category iterator_category;
};
```

结构与类相同，只是结构的成员在默认情况下都是 public。这个模板定义了 STL 需要定义定制迭代器的所有类型。例如，如果有一个未知的模板参数 Iter，在需要声明一个该类型的指针时，可以编写 Iter::pointer，在解除该指针的引用时，迭代器就会提供该类型。表 20-6 列出了这些类型的含义。

表 20-6　iterator<>模板定义的类型

T 的 Iter 迭代器类型	含　义
value_type	所指向的对象类型
difference_type	迭代器的减法操作
pointer	T*，对象类型的指针
reference	T&，对象类型的引用
iterator_category	输入、输出、向前、双向或随机访问

iterator_category 的值从一组固定的类别标记中提取，指定迭代器的类别，它确定了迭代器可以执行什么类的算法。表 20-7 列出了迭代器标记的值。

表 20-7　迭代器标记的值

迭代器类型	需要的分类标记
输入	input_iterator_tag
输出	output_iterator_tag
向前	forward_iterator_tag
双向	bidirectional_iterator_tag
随机访问	random_access_iterator_tag

随机访问迭代器具有 C++指针的所有功能。其他迭代器类别比较受限制。稍后介绍的 list 容器就提供了双向迭代器。这种迭代器可以在一步中向前或向后移动，但不能使用下标运算符[]进行随机访问。

把 iterator 模板的一个实例作为类的基础，就可以使 Integer 类满足 STL 的类型定义要求：

```
class Integer : public iterator< random_access_iterator_tag, int, int, int*, int>
```

这行语句为迭代器定义了 STL 需要的所有类型。模板的第一个参数把迭代器的类型指定为随机访问迭代器。第二个参数是迭代器指向的对象类型 int。第三个参数是两个迭代器的差的类型，它也是 int。第四个模板参数是对象指针的类型，所以它是 int*。最后一个模板参数指定引用的类型，也是 int。

20.3.3　STL 迭代器成员函数的要求

STL 还定义了一组每种迭代器都必须支持的成员函数。我们对随机访问迭代器的实现比较感兴趣，所以介绍完整的函数组。这有助于把它们搜集到组中。

第一组是构造函数，其中包含一些所有复杂类都需要的重要函数：默认构造函数、副本构造函数和赋值运算符函数。一个普遍的规则是：如果给迭代器编写了这些函数中的一个，就必须编写一个显式的析构函数。对于 Integer 类，不需要定义析构函数，因为 Integer 的析构函数什么也不做，有默认构造函数就可以了。这些函数的原型如下：

```
Integer(int n=0);                          // Default constructor
Integer(const Integer& y);                 // Copy constructor
~Integer();                                // Destructor
Integer& operator=(const Integer& y);      // Assignment operator
```

在 Integer 类中，默认构造函数使用了构造函数参数的默认值。STL 在容器中创建新元素时，会使用默认构造函数，因此必须定义它。

STL 要求为随机访问迭代器类实现全部等于和关系运算符。实际上，可以使用标准库的其他模板实现这些运算符，如下所示：

```
bool operator==(const Integer& y) const;
bool operator<(const Integer& y) const;
```

这里假定包含了 utility 头文件的#include 指令，以及 std::rel_ops 命名空间的 using 声明：

```
#include <utility>
using namespace std::rel_ops;
```

如果为类类型提供了 operator==()和 operator<()函数，在 std 命名空间中声明的 rel_ops 命名空间就包含为类类型的!=、>、>=和<=自动生成运算符函数的函数模板。使用 using 声明激活 rel_ops，就不必手工定义这 4 个运算符了。如果提供了某个运算符函数，而该函数本来应由 std::rel_ops 命名空间中的模板生成的，自己的运算符函数的优先级要高于 rel_ops 命名空间中的模板生成的函数。

运算符 operator<()比较特殊，它称为排序关系运算符。它在搜索和比较算法中非常重要。给定了 operator<()的一个定义，rel_ops 命名空间中的模板会自动创建一个 operator>()函数。

运算符 operator==()用于测试两个容器或对象是否包含相同的内容。其工作原理比较有趣。对于任意一对操作数 x 和 y，表达式(x<y || y<x || x==y)总是返回 true，因为三个组件表达式中总有一个为 true。实际上，它不一定以这种方式工作。显然，如果(x==y)为 true，则(x<y)和(y<x)就不能是 true。于是，相等的元素不能不一样。

但是，如果(x!=y)为 true，就不能假定(x<y)和(y<x)中的一个为 true——(!(x<y)) &&(!(y<x))是 true 时，元素 x 和 y 就是不相等的，这表示排序是不分先后的。一个常见的例子是对字符串排序，但忽略大小写。在不区分大小写时，字符串"A123"和"a123"是不相等的(哪个字符串都不能放在最前面)，但它们是不同的，也不相等。

STL 还要求为随机访问迭代器实现访问运算符：

```
int operator*() const;
int operator[](int n) const;
```

它们用于从序列中返回值。运算符函数 operator[]仅在 random_access 类别的迭代器中使用。

随机访问迭代器还必须支持双向迭代器实现的双向运算符：

```
Integer& operator++();
Integer& operator--();
Integer operator++(int);
Integer operator--(int);
```

这些运算符允许在某一刻向前或向后移动一个元素。向前迭代器不能向后移动，所以没有提供递减形式。

支持随机访问的运算符如下：

```
Integer operator+(int n) const;
Integer operator-(int n) const;
```

只要 operator[]出现在序列容器中，就可以使用这两个运算符。只要指定合适的偏移值，它们就可以直接从任意迭代器位置访问序列中的任意元素。

如果把这些都放在 Integer 类中，就达到了 STL 的要求。

程序示例 20.9——完全符合要求的迭代器

下面给 Integer 类添加 STL 要求随机访问迭代器具有的所有功能：

```
// Integer.h  Integer class definition
#ifndef INTEGER_H
#define INTEGER_H
#include <iostream>
using std::cout;
using std::endl;
using std::iterator;
#include <utility>
using namespace std: :rel_ops;     //Access to templates for relational ops

class Integer :
    public iterator<random_access_iterator_tag, int, int, int*, int> {
 public:
   Integer(int n=0) : x(n) {}                    // Default constructor
   Integer(const Integer& y) : x(y.x) {}     // Copy constructor
   ~Integer() {}        // Destructor

   Integer& operator = (const Integer& y) {   // Assignment operator
     x = y.x;
     return *this;
   }

   // Relational operators
   bool operator== (const Integer& y) const { return x ==y.x; }
   bool operator ! = (const Integer& y) const { return !(*this == y) ; }
   bool operator < (const Integer& y)  const { return x < y.x; }

   int operator* () const { return x; }
   int operator[](int n) const { return x+n; }

   // Bidirectional operators
   Integer& operator++() {
     ++y;
     return *this;
   }

   Integers operator--() {
     --x;
     return *this;
   }

   Integer operator ++(int) {
     Integer temp(*this);
     ++x;
     return temp;
   }
   Integer operator--(int) {
     Integer temp( *this ) ;
     --x;
     return temp;
   }
```

```
    // Random access operators
    Integer operator+( int n) const { return Integer(x+n); }
    Integer operator-(int n) const { return Integer(x-n) ; }

  private:
    int x;
};
#endif    //INTEGER_H
```

下面的程序使用 Integer 类的这个版本：

```
// Program 20.09 Full "random access" Integer iterator File: prog20_09.cpp
#include <iostream>
#include <algorithm>
#include "Integer.h"
using std::cout;
using std::endl;

int main() {
  Integer F1(-1);
  Integer L1(10) ;
  cout << "The values [-1..10] in forward order: " << endl;
  copy (F1, L1, std::ostream_iterator<int>(cout, " "));
  cout << endl;

  typedef std::reverse_iterator<Integer> CountDown;
  CountDown F2(10);
  CountDown L2(-1);
  cout << "the values (10..-1) in reverse order: " << endl;
  copy (F2, L2, std::ostream_iterator<int>( cout, " "));
  cout << endl;

  return 0 ;
}
```

Integer 类看起来不再渺小，它现在拥有 STL 迭代器所需要的所有功能。

运行这个程序，结果如下：

```
The values [-1..10) in forward order:
-1 0 1 2 3 4 5 6 7 8 9
The values (10..-1] in reverse order:
9 8 7 6 5 4 3 2 1 0 -1
```

例子的说明

这里为 Integer 类实现了上一节描述的所有功能。给这个类类型提供了 operator==()和 operator<()，所以 rel_ops 命名空间中的函数模板会自动生成需要的其他运算符函数。在本例中，为 Integer 类提供了自己的 operator!=()实现代码，所以它不会由 rel_ops 命名空间中的模板创建。

在 main()中创建迭代器类 CountDown，很好地演示了 Integer 类拥有一个功能完整的 STL 迭代器：

```
typedef reverse_iterator<Integer> CountDown;
```

```
CountDown F2(10), L2(-1);
cout<<"the values (10..-1) in reverse order: "<<endl;
copy(F2, L2, ostream_iterator<int>(cout, " "));
```

CountDown 是使用 STL 适配器类从 Integer 类中创建的一个新类。reverse_iterator 类型是一个模板类，把迭代器作为其参数，创建一个新的迭代器。新迭代器与刚开始的迭代器完全相同，只是有一点不同：所有的值现在都是逆序排列的。因此，可以使用 CountDown 对象输出间隔(10,-1)中的整数，其中不包括 10，但包括 - 1。reverse_iterator 模板只能用于双向迭代器和随机访问迭代器，所以 Integer 至少是双向迭代器，也可以肯定是随机访问迭代器。

20.3.4 插入迭代器

输入迭代器只能用于读取值。相反，输出迭代器只能用于写入值。这些迭代器类别不支持随机访问和向后移动。实际上，使用这两个迭代器只能在每个位置读写一次。这两个迭代器的主要用途是允许 STL 算法如 copy()操作流。前面已经使用了 ostream_iterator 和 istream_iterator。

还有一种非常有用的输出迭代器，可以称为插入迭代器系列。在使用 copy()等算法把元素插入容器时，就可以使用这种迭代器。例如，可以使用 back_inserter()函数模板创建一个可以在容器的最后添加元素的迭代器，下面使用它定义一个模板函数 append()：

```
template <typename Container, class Iter>
void append(Container& C, Iter src, Iter src_end) {
  copy(src, src_end, back_inserter(C));
}
```

这段代码定义了一个函数模板，它把元素追加到第一个模板参数指定的容器中。要追加的元素源是由第二个模板参数指定的类型的迭代器。因此，这个函数可以从迭代器定义的任何源中把元素追加到任意容器中。

元素复制的目标容器由 append()函数的第一个参数指定。要复制到容器中的元素由第二和第三个参数指定，这两个参数都是 Iter 类型的迭代器。back_inserter()函数把容器作为其参数，返回一个 output_iterator，把接收到的每个值都追加到目标容器中。如果直接复制到 C.end()中，copy 就会以改写模式进行操作，这可能会把数据写入 vector 中不能用于写入的区域，从而超出了 vector 的边界。由 back_inserter()函数生成的输出迭代器以插入模式进行操作，在必要的时候会使容器增大。

front_inserter()补足了 back_inserter()，它允许在容器的开头插入元素。注意，不能对 vector 容器使用这个函数，因为 front_inserter()依靠容器的一个成员 push_front()在开头插入元素，而 vector 容器没有声明这个成员函数。

最后，inserter()函数生成的输出迭代器可以在任意位置插入元素。例如：

```
copy(src, src_end, inserter(C, any_iterator_position_within_C));
```

这个插入器插入一个元素后，会前进到下一个位置。所插入的元素从开始插入的位置开始，按照先后顺序组成一组，这样，容器就会扩展，以包含插入的元素。

程序示例 20.10——使用插入器

下面试验 front_inserter()。由于不能对 vector 容器使用这个函数，我们就对 list 容器使用该

函数。下面是代码：

```
// Program 20.10 - Using an inserter    File: prog20_10.cpp
#include <iostream>
#include <iterator>
#include <algorithm>
#include <list>
using std::cout;
using std::endl;
using std:: front_inserter;
using std::ostream_iterator;

// Front insert
template <typename Container, class Iter>
void pre_insert(Container& C, Iter src, Iter src_end) {
  std::copy(src, src_end, front_inserter(C));
}

int main() {
  int values[] = { 1, 9, 7, 5, 15 };
  std::list<int> numbers;              //Create a list container of integers

  // Append elements of values array to the front of the numbers list
  pre_insert(numbers, values, values+sizeof values/sizeof values[0]);

  //Copy the list to the output stream
  std::copy(numbers.begin(), numbers.end(), ostream_iterator<int>(cout," "));
  cout <<endl;
  return 0;
}
```

这个程序的结果是以逆序方式显示插入的值：

```
15 5 7 9 1
```

例子的说明

数组中的值的顺序被颠倒了，因为 pre_insert()函数模板中使用了 front_inserter()。从 list<int> 容器 L 的开头开始插入值，先插入数组中的第一个元素，最后插入数组的最后一个元素，所以数组的最后一个元素就成为容器中的第一个值。前面说过，不能对 vector 容器使用 front_inserter() 函数，但可以对 vector 和 list 容器使用 back_inserter()。

下面深入论述 List 容器。

20.4 list 容器

STL 设计者的一个重要理念是为可能出现的任务提供有效的机制。vector 可以用于使用 C++数组的地方，它是一个非常有效、一般意义上的容器。但是，如果需要在元素序列中频繁地插入或删除元素，使用 vector 容器并不理想。List 容器的主要优点是所有的插入和删除操作所需的时间都是固定的。其缺点是不能随机访问元素。

list 容器支持双向迭代器。也就是说，可以向前或向后移动到下一个元素上，但不能进行大的跳跃。即使没有 operator[]提供的随机访问功能，大多数算法也都可以在 list 容器上执行，下面介绍 list 容器提供的一些功能。

20.4.1 创建 list 容器

list 容器的构造函数组类似于 vector 容器。下面的语句可以创建一个空的 list 容器，以存储 double 类型的值：

```
std::list<double> data;
```

如果要指定 list 容器中的元素个数，可以编写下面的代码：

```
std::list<double> values(20);
```

这个语句创建了一个包含 20 个元素的列表，每个元素都初始化为 0.0。如果模板参数是一个类类型，就调用该类的默认构造函数，为每个元素创建初始值。

下面的语句可以为这个 list 容器中的元素设置值：

```
std::list<double>::iterator first = values.begin();
double x = 1.0;
while(first != values.end())
   *first++ = x++;
```

这些代码把 list 容器中的元素设置为 1.0 到 20.0 的数值。

还可以用一个特定的值初始化 list 容器中的所有元素：

```
std::list<double> data(20, 3.14159);
```

这个 list 容器中的所有 20 个元素都设置为 3.14159。

可以从两个迭代器指定的间隔中构造 list 容器。例如，list 容器 values 的值初始化为从 1.0 到 20.0 的数值，下面从 list 容器 values 中的元素里构造一个新 list 容器：

```
std::list<double>::iterator begin = ++values.begin();        // Begin()+1
std::list<double>::iterator end = begin++;                    // end = begin, begin += 1
for(int I = 0; i<7; i++,++end);                               // end += 6
std::list<double>: some_data(begin, end);
```

这里创建了两个迭代器 begin 和 end，它们分别指向 values 列表中的第 3 和第 9 个元素。使用递增运算符和循环把这些迭代器设置为上述位置。设置迭代器的位置，是因为这些迭代器在一步中只能递增一次。从 values 的元素中使用它们以间隔[begin, end)创建新 list，于是 list 容器 some_data 包含的元素值为 3、4、5、6、7 和 8。

list 中的元素个数由 size()函数返回。调用 resize()函数，参数指定为 list 容器的新大小，就可以改变元素的个数。如果 list 容器的新大小大于原来的大小，就使用 list 容器中对象的默认构造函数添加新元素。如果新大小小于原来的大小，就从 list 容器的尾部删除元素。

20.4.2 访问 list 容器中的元素

按顺序处理 list 容器中的元素有两个方法。前面介绍了如何使用 begin()和 end()获取迭代器，按顺序访问元素。还可以使用 rbegin()和 rend()函数返回反向迭代器，逆序遍历元素。这些迭代器都是双向的，所以只能给它们递增或递减 1。

front()和 back()成员返回第一个和最后一个元素的引用。

20.4.3 list 容器上的操作

push_front()和 push_back()成员函数分别在 list 容器的开头和结尾插入一个元素。pop_front()和 pop_back()函数分别删除第一个和最后一个元素。insert()成员函数用于在 list 容器中由迭代器指定的位置上插入一个或多个元素。insert()函数有三个版本：

```
iterator insert(iterator position, const T& x);    //Insert a single element
void insert(iterator position, size_type n const T& x);    //Insert n copies
template<class InputIterator>
    void insert(iterator position, InputIterator first, InputIterator last);
```

第一个版本在第一个参数指定的位置插入 x，返回一个指向该位置的迭代器。第二个版本在 list 容器中的指定位置插入 x 的 n 个副本。模板版本在第一个参数指定的位置插入第二和第三个参数指定的间隔中的元素。指定半开间隔的迭代器是 InputIterator 类型，所以它们可以来自提供这种迭代器的任何外部源。

使用 erase()成员函数可以删除元素。它的一个版本接受一个指定要删除的元素的迭代器。另一个版本接受两个迭代器参数，指定要删除的元素的半开间隔。这两个版本都返回一个迭代器，它指向已删除的元素后面的元素。还可以使用 remove()函数删除具有给定值的元素。其参数是要删除的值。要删除 list 容器中的所有元素，只需调用 clear()函数。调用 empty()函数可以测试 list 容器是否为空。

list 容器还有一个特殊的函数 splice()，它删除一个 list 容器中的元素，并把它们移植到另一个类型相同的 list 容器中。该函数合并了 insert 和 erase 函数执行的步骤。这个函数不会移动 list 容器中的任何元素，只是把要接合的第一个和最后一个元素的指针改为指向下一个和前一个元素，使它们成为新 list 容器的一部分，然后修改插入位置两端的指针，使它们分别指向第一个和最后一个插入的元素。函数 splice()有三个版本，第一个版本有一个参数，它总是当前 list 容器中要插入元素的位置。在带有两个参数的版本中，第二个参数是要整个插入当前 list 容器的 list 容器。在带有三个参数的版本中，第二个参数是要插入的元素源，第三个参数是指向第一个插入元素的迭代器。从这个迭代器位置到最后的所有元素都会移植到当前 list 容器中。使用第三个版本时，后两个参数是指定半开间隔的迭代器，半开间隔中的元素从第三个参数指定的 list 容器中移植。

在 algorithm 头文件中声明的一般意义的 sort()算法不能用于 list<>容器，因为它需要随机访问迭代器。list<>容器提供了一个特殊的 sort()成员函数，用于给 list 容器中的元素排序。

程序示例 20.11——使用 list 容器

第 13 章在为存储货车所载的箱子开发 Package 类时，详细论述了 list 数据结构。现在则介

绍拥有 STL list 容器所有功能的 TruckLoad 类的实现情形：

```
// Box.h  definition of the Box class
#ifndef BOX_H
#define BOX_H
#include <iostream>
using std::cout;
using std::ostream;

class Box {
  public:
    Box(double lv = 1.0, double wv = 1.0, double hv = 1.0) :
                                        lengh(lv), width(wv), height(hv) {}

    double volume () const { return lengh * width * height ; }
    bool operator <(const Box& x) const { return volume () < x.volume(); }

    friend ostream& operator<<(ostream& out, const Box& box) {
      out << "("<< box.length <<","<< box.width <<","<< box.height <<")";
      return out;
    }

  private :
    double length;
    double width;
    double height;
};
#endif
```

在 TruckLoad.h 头文件中定义 TruckLoad 类，如下所示：

```
// TruckLoad.h  definition of the TruckLoad class
#ifndef TRUCKLOAD_H
#define TRUCKLOAD_H
#include <list>
#include "Box.h"

class TruckLoad {
    typedef std::list<Box> Contents;

  public:
typedef Contents::const_iterator const_iterator;
TruckLoad() {}
TruckLoad(const Box& one_box) : Load (1, one_box) {}

template<typename FwdIter>
TruckLoad(FwdIter first, FwdIter last) : Load (first, last) {}

void add_box(const Box& new_box) { Load.push _back (new_box) ; }

const_iterator begin() const { return Load.begin (); }
const_iterator end() const { return Load.end () ; }
```

```
  private :
    Contents Load;
}
#endif
```

下面的程序使用新的 TruckLoad 和 Box 类：

```
// Program 20.11 A TruckLoad container implemented using an STL list container
// Recapitulates Program 13.1
#include <iostream>
#include <algorithm>
using std::cout;
using std::endl;

// Random number generation 1 to count
inline int random(int count) {
  return 1 + static_cast<int>
                (count *static_cast<double>(rand())/(RAND_MAX+1.0));
}

// Create a Box with random dimensions in a range
inline Box random_box(int range) {
  return Box(random(range) , random(range) , random(range));
}

int main() {
  TruckLoad rig1(Box(30, 30, 30));
  for(int i = 0; i < 8; ++i)
    rig1.add_box(random_box(100));

  cout << "Contents of rig1" << endl;
  std::copy(rig1.begin() , rig1.end(), std::ostream_iterator<Box> (cout, "\n"));
  cout << endl;

  typedef TruckLoad::const_iterator BoxIter;
  BoxIter big_one=max_element(rig1.begin(), rig1.end());

  cout << "The biggest box in rig1 is " << *big_one
       << " with volume " << big_one->volume() << endl;
  cout << endl;

  cout << "Copying all boxes starting at big box to rig2" << endl;
  TruckLoad rig2(big_one, rig1. end()) ;
  cout << "Contents of rig2" << endl;
  std::copy(rig2.begin() , rig2.end(), std::ostream_iterator<Box> (cout, "\n"));
  cout << endl;

  return 0;
}
```

运行这个程序，结果如下：

```
Contents of rigl
```

```
(30,30,30)
(20,57,1)
(48,59,81)
(83,90,36)
(86,18,75)
(31,52,72)
(37,10,2)
(99,17,15)
(1,12,45)

The biggest box in rig1 is (83,90,36) with volume 268920

Copying all boxes starting at big box to rig2
Contents of rig2
(83,90,36)
(86,18,75)
(31,52,72)
(37,10,2)
(99,17,15)
(1,12,45)
```

例子的说明

在讲述 main()中的代码时，先讨论一下基础工作。我们已经很熟悉类 Box 及其使用方法。Box 类的这个版本提供了两个运算符函数。第一个运算符是：

```
bool operator<(const Box& x) const {return volume() < x.volume();}
```

提供了 operator<()后，就可以比较盒子，在使用 STL 中的 max_element()函数时，该运算符是必需的。

再重载流的插入运算符：

```
friend ostream& operator<<(ostream& out, const Box& box) {
  out<<"("<<box.length<<","<<box.width<<","<<box.height<<")";
  return out;
```

定义用于 ostream 对象的 operator<<()函数，可以简化把 Box 对象写入输出流的过程。类的友元函数不是类的成员函数，所以不能接收 this 指针。这个函数输出的 Box 对象显式传送给 operator<<()函数，作为第二个参数。该函数的返回类型是 ostream&，使函数更便于在复合语句中使用，因为这样可以在一系列流写入操作中使用该函数。

TruckLoad 的定义非常简单：

```
class TruckLoad {
  typedef list<Box> Contents;

  //public members…

  private:
    Contents load;
};
```

这个定义提供了自己的接口。TruckLoad 的内容包含在一个成员变量 load 中，它是 Box 的

一个列表。我们定义的每个函数都使用 load 提供的一个等价函数来实现。再使用 Contents 的 typedef 来声明其他类函数。在使用 STL 时这是一个标准方式。load 成员的声明与我们自己编写的完全相同：

```
private:
list<Box> Load;
```

TruckLoad 类中有三个构造函数：

```
TruckLoad() {}
TruckLoad(const Box& one_box) : load(1, one_box) {}

template<typename FwdIter>
TruckLoad(FwdIter first, FwdIter last) : load(first, last) {}
```

第一个构造函数是默认构造函数，第二个构造函数把 TruckLoad 对象初始化为包含一个 Box 对象。具体方法是，调用 list<Box>构造函数，把要插入的元素个数作为第一个参数，要插入指定次数的对象作为第二个参数，初始化 list<Box>类型的 load 对象。

第三个构造函数是一个模板构造函数提供的，可以给这个构造函数传送迭代器，把 TruckLoad 初始化为包含已创建的任意 Box 对象序列的副本。迭代器使用类模板来声明，因此迭代器的类型是非常灵活的，可以是 Box 对象数组的指针、基于 Box 对象集合的 vector<>或 list<>的迭代器，甚至可以是另一个 TruckLoad 内容的迭代器。

这是很容易实现的。load 对象是一个 list 容器，提供了一个模板序列构造函数。我们只需给 load 构造函数传送 first 和 last 参数，就像接收这两个参数一样，load 构造函数就会完成剩余的工作。

函数 add_box()给 TruckLoad 对象添加一个 Box 对象：

```
void add_box(const Box& new_box) {Load.push_back(new_box);}
```

也可以使用 push_front()把新的 Box 对象放在 Load 列表的前面。货车通常不以这种方式装载货物，所以这里使用 push_back()，把新的 Box 对象添加到列表的尾部。

接着在类定义的 public 部分，是另一个 typedef:

```
typedef Contents::const_iterator const_iterator;
```

其中 Contents 是类型 list<Box>的同义词，因此 Contents::const_iterator 就是 list<Box>::const_iterator 的同义词。如果可以记住所有同义词的含义，这就是一种很有用的简写方式。

在 typedef 语句之后，类 TruckLoad 的客户就可以用 begin()和 end()成员函数访问货车内容的迭代器：

```
const_iterator begin() const{return load.begin();}
const_iterator end() const{return load.end();}
```

这些都是常量迭代器。在清除客户之前，我们不希望客户暗地里修改 TruckLoad 的内容。

两个函数 begin()和 end()都直接返回 Load 迭代器。这个设计并不很完美，因为需要提供一些实现代码——事实上我们在内部使用一个 list 容器装载内容。但是这非常方便。如果客户的操作正确，就不必知道这些迭代器都指向一个 list 容器，更不用说其他类型的容器了。

因为我们返回的是迭代器，所以 TruckLoad 的操作非常类似于容器。这不是件坏事，因为 TruckLoad 一开始就很像一个容器。

程序使用全局函数 random()和 random_box()，创建在程序示例中使用的随机盒子：

```
inline int random(int count) {
 return 1+static_cast<int>((count* static_cast<double>(rand())) /RAND_max+1.0));
}

inline Box random_box(int range) {
 return Box (random(range), random(range), random(range));
}
```

在前面的 TruckLoad 例子中，random()也是这样定义的——其细节已在第 13 章论述过了。

下面看看函数 main()中比较有趣的部分。下面的语句：

```
cout<<"Contents of Rig1"<<endl;
std::copy(rig1.begin(), rig1.end(), std::ostream_iterator<Box>(cout,"\n"));
cout<<endl;
```

copy 算法的开始两个参数是复制的 Box 对象的范围，在这个例子中是从 begin()到 end()的整个 TruckLoad。copy()函数的第三个参数按照以前的方式使用 ostream_iterator，只是输出值之间的分隔符是换行符"\n"。

代码非常简洁，这是 STL 对象的一般工作方式。注意，ostream_iterator 使用 Box::operator<<(ostream,Box)控制 Box 对象写入流 cout 的方式。这是“输出所有盒子”的一种紧凑方式。

下面是根据 TruckLoad 类中 public 部分定义的 const_iterator 类型来定义 BoxIter 的 typedef 语句：

```
typedef TruckLoad::const_iterator BoxIter;
```

接着使用 BoxIter 声明一个常量迭代器 big_one：

```
BoxIter big_one=max_element(rig1.begin(), rig1.end());
```

TruckLoad::const_iterator 是 list<Box>::const_iterator 的同义词，这次有一个重要的区别：客户不知道这个，也不需要知道。对于客户来说，const_iterator 是 TruckLoad 的公共接口提供的匿名类型，用于从 TruckLoad::begin()和 TruckLoad::end()中接收的所有迭代器。

这里还使用了另一个 STL 算法 max_element()。它的参数定义了搜索的范围，返回该范围内最大值的迭代器。如果范围是空的，返回的迭代器就是 rig.end()，或者 rig.end()之后，那里没有元素。

提示：

使用 STL 的一个关键特性是可以使用程序中对象定义的类型。在这个例子中，BoxIter 就是类 TruckLoad 提供的一种迭代器，但目前并不知道它实际上是 STL list 容器中的迭代器。只有 TruckLoad 知道该迭代器的内部情况。

要复制 TruckLoad 的一部分，从找到的最大盒子开始复制，应使用下面的语句：

```
cout<<"Copying all boxes starting at big box to rig2"<<endl;
TruckLoad rig2(big_one, rig1.end());
```

这里使用副本构造函数进行复制，副本构造函数的参数是迭代器范围。这说明从算法max_element()函数中返回的迭代器跟其他迭代器是一样的，可以用于引用 Box 元素的TruckLoad 序列中的位置。

20.5 关联 map 容器

前面仅介绍了 vector<>和 list<>容器，它们是序列容器。序列容器中的元素顺序是由插入元素的方法确定的。关联容器的工作方式则不同，每个对象都与一个关联键(可能是对象本身)一起存储，对象在内部存储在按键排序的序列上。在插入一个新元素时，会自动使用键在容器的适当位置上放置该元素，使排序顺序保持不变。

也可以利用键访问关联容器中的元素。键的数据类型必须提供一个排序关系，排序关系可以通过 operator<()来提供，也可以通过函数对象 Compare 来提供，如果第一个对象小于第二个对象，该函数对象就返回 true，否则返回 false。前面没有介绍过函数对象，后面将详细介绍它。未定义排序关系的对象不能用作关联容器中的键。用作键的数据类型没有其他限制，所以关联值可以是已定义了顺序的任意数据类型。

如本章开头所述，有 4 种不同类型的关联容器：map、set、multimap 和 multiset。当键是惟一的时，可以使用 map 和 set 容器；当需要重复的键时，应使用 multimap 和 multiset 容器。在确定使用哪个关联容器时，还必须考虑另外一个因素：map 和 multimap 容器的键有关联值(一般是对象)，在 set 和 multiset 容器中，对象就是它们自己的键。

下面举例说明其中的含义。例如，可以使用 map 或 multimap 容器存储姓名和电话号码。这两个容器会存储一个元素集合，把一个键与一个对象关联起来——在上面的例子中，姓名是键，电话号码就是关联对象，那么应使用哪个容器呢？如果使用 map，键就必须是惟一的。也就是说，两个人不能同名，例如 Jeremiah hackenbush，而且每个人必须只有一个电话号码。但许多人可能有相同的名称，而且许多人都有多个电话号码，这就需要有多个 person 键，次数只能选择 multimap 容器。

提示：

在某些情况下，multimap 是最常见的关联容器。但是，使用 multimap 不一定是默认的。在使用较常见的 multimap 时，就不能使用 map 容器提供的、非常方便的下标运算符了，丧失了关联数组的功能。

set 和 multiset 容器没有独立的键，所以比 map 容器简单。元素的自动排序是 set 和 multiset 容器主要优点。

使用 map 容器

map 模板是用四个模板参数定义的，但正常情况下，只需要指定前两个参数。第一个参数是键的类型，第二个参数是要存储的对象类型。第三和第四个模板参数定义了用于比较键的函数对象类型和用于在 map 容器中分配内存空间的对象类型。在大多数情况下，这两个参数使用默认值就可以了。偶尔需要为比较键定义不同的对象类型。后面“理解函数对象”一节将介绍

一个例子。

map 容器的默认工作函数创建一个空的 map 容器。例如，可以创建一个 map 容器，存储 Person 对象，其键是 string 类型，如下所示：

```
std::map<std::string, Person> personnel;
```

第一个模板参数指定键的类型为 string，第二个模板参数指定与键关联的对象类型是 Person。这里，键可能是表示为字符串的人名。当然，前两个模板参数也可以是基本类型。

map 容器中的每个元素都存储为对象，该对象包含了键和与键关联的对象。元素类型是第四个 map<>模板参数指定的模板类型。如果这个参数使用默认类型，map 容器 personnel 中的元素类型就是< string, Person>。在本例中，这个模板类型也可以使用。如果需要把两个对象打包为一个对象，使用它就比较方便。

除了复制已有 map 容器的副本构造函数之外，创建 map 容器的另一个方法是从另一个 map<>容器的一组元素中创建。元素由两个迭代器定义的半开间隔来指定。如果已使用元素填充了 personnel 容器，就可以创建一个新容器，其中包含这些元素的一个子集，如下面的语句所示：

```
std::map<std::string, Person> department(iter1, iter2);
```

迭代器 iter1 和 iter2 给 map 容器 personnel 中的元素指定了键的半开间隔[iter1, iter2)，这个语句构造了一个空的 map 容器，然后把间隔中的元素插入新的 map 容器。

1. 访问 map 容器中的元素

map 容器实现了下标运算符，所以可以把键用作索引，提取关联的对象。下面是一个例子：

```
Person chosen_one = personnel["666"];
```

这个语句提取对应于个人号码"666"的 Person 对象，其中个人号码"666"提供为键。注意使用下标运算符不只是一个提取机制。如果没有对应于"666"的 Person 对象，就使用默认的 Person 类构造函数创建一个 Person 对象。因此，只有确保 map 容器中存在这个键时，这才是一个有意义的提取机制。所以，一般应在更新 map 容器中的元素，或插入 map 容器中不存在的元素时，使用下标运算符。下标运算符的另一个主要用途是放在等号的左边，修改已有的数据项。

在对 map 容器使用下标运算符存储或修改数据项中的对象时，键要放在方括号中，表示该数据项，把关联的对象作为等号的右操作数，如下所示：

```
personnel["999"] = chief_exec;          //Store the Person chief_exec
```

这个语句在 map 容器 personnel 中存储了 Person 对象 chief_exec，其个人号码"999"用作键。

检查对应于键的数据项是否存在，一种方式是使用 find()方法。这个方法返回指向 map 容器中数据项位置的迭代器，如果该数据项不存在，返回的迭代器就指向 map 容器最后一个元素后面的位置。因此，下面的语句可以检查 map 容器 personnel 中元素是否存在：

```
if(personnel.find("666") 1=personnel.end())
  std::cout<<"Number 666 is in there!" <<std::endl;
```

另一个方法是使用 map 容器的 count()方法。这个方法返回 map 容器中对应于传送为参数的键的元素个数。对于 map 容器，它可以是 0 或 1，但对于 multimap 容器，一个给定的键值可能对应几个数据项。下面的语句可以检查数字"666"是否存在：

```
if(personnel.count("666"))
  std::cout<<"Number 666 is in there!" <<std::endl;
```

返回值 1 将强制转换为 bool 值 true，所以本例会执行输出语句。

除了使用下标运算符之外，还可以通过迭代器访问 map 容器中的元素。map 容器也有 begin()、end()、rbegin()和 rend()函数，它们返回 iterator 类型的双向迭代器，iterator 类型在 map<> 类中定义。对于 map 容器 personnel，迭代器类型应指定为 map<string, Person>::iterator。可以使用这些迭代器遍历 map 容器中的元素，其方法与 list 容器相同。解除迭代器的引用会得到所存储的对象。map 容器中的对象按照它们的键来排序，所以从 begin()遍历到 end()会按照键的升序访问元素，这是由键必须提供的排序关系确定的。排序关系可以由 operator<()函数或 compare 函数对象提供。

map 迭代器指向 pair<>类型的元素。解除迭代器的引用可以直接访问键/对象对中的对象，还可以获得该元素的键。例如，下面列出了 map 容器 personnel 中的键：

```
map<string, Person>::iterator iter = personnel.begin();
while(iter!= personnel.end())
  std::cout<< iter->second <<std::endl;
```

迭代器指向的 pair<>对象中的 second 成员存储了键，与键关联的对象存储在 first 成员中。

程序示例 20.12——使用 map 搜集词的搭配

搭配词的一种常见形式是计算文本中的每个词出现了多少次。下面使用 map 容器来实现：

```
// Program 20.12 A simple word collocation  File: prog20_12.cpp
#include <iostream>
#include <iomanip>
#include <string>
#include <sstream>
#include <map>
using std::cout;
using std::endl;
using std::string;

const string twister =
  "How much wood would a woodchuck chuck if a woodchuck "
  "could chuck wood? A woodchuck would chuck as much wood "
  "as a woodchuck could chuck if a woodchuck could chuck wood.";

int main() {
  typedef std::map<string, int> Collocation;      // Type for our map
  typedef Collocation::const_iterator WordIter;  // Iterator type for our map

  Collocation words;       //Map to store words and word counts

  std::istringstream text(twister);                // Text string as a stream
```

```
std::istream_iterator<string> begin(text);      // Stream iterator
std::istream_iterator<string> end;    //End stream iterator

for( ; begin != end ; ++begin)          //Iterate over the words in the stream
  words[*begin]++;                      //Store and increment word count

//Output the words and their counts
for(WordIter iter = words.begin() ; iter != words.end() ; ++iter)
  cout<<std::setw(6)<< iter->second<<" " <<iter->first<<endl;

return 0;
}
```

这个程序的运行结果如下：

```
1 A
1 How
4 a
2 as
5 chuck
3 could
2 if
2 much
2 wood
1 wood.
1 wood?
5 woodchuck
2 would
```

例子的说明

头文件 map 有一个#include 指令：

```
#include <map>
```

map 和 multimap 容器都在头文件<map>中声明。我们不想频繁地键入 map<string, int>::const_iterator。最好利用 typedef：

```
typedef std::map<string, int>::Collocation;       // Type for our map
typedef Collocation::const_iterator WordIter;    // Iterator type for our map
```

map<>模板带两个参数。第一个参数是键的数据类型，第二个参数是关联对象的数据类型。在这个程序中，键是 string 类型的对象，表示在示例文本中找到的单词。每个字符串的关联值都是 int 类型的值，用于计算键出现的次数。这里使用 WordIter 类型的迭代器显示搭配词的内容。

搭配词的 map 容器定义非常简单：

```
Collocation words;        //Map to store words and word counts
```

接着，从 string 对象中创建 string 流对象，再为流创建一些迭代器：

```
std::istringstream text(twister);                 // Text string as a stream
std::istream_iterator<string> begin(text);  // Stream iterator
std::istream_iterator<string> end;                // End stream iterator
```

程序使用了以前的 istringstream。创建一个 istream_iterator 对象 begin，它最初指向 text 流的开头，而 text 流用闲谈的内容初始化。istream_iterator 的默认构造函数创建一个流尾迭代器，所以 end 表示 text 的结尾，可以使用它们迭代 text 中的每个词。这里说的是“词”，而不是“单词”，是因为 istream_iterator 不是很聪明，只能把用空白断开的可打印字符分隔出来，如这部分的输出所示。

在搭配词中插入并计算单词出现的次数是很简单的，这是用 for 循环体中的一个语句完成的：

```
for(;begin!=end;++begin)     //Iterate over the words in the stream
  words[*begin]++;           //Store and increment word count
```

表达式*begin 返回一个字符串，它包含从闲谈内容中提取出来的一个词。容器 words 用 string 类型的键作为下标。关联容器中的元素可以关联访问，这表示在给 map 指定一个键时，它会返回关联的 int 值，然后递增该值。

对于 map 容器，operator[]()实际上是 insert()的一种缩写方式。也就是说，有时要特别小心。考虑下面的语句：

```
if(words["brawn"]>0)
cout<<"too much muscle"<<endl;
```

这看起来像是测试字符串"brawn"的计数是否为正，但要特别小心，因为 operator[]()是为 map 容器定义的，这个语句是在容器中不存在"brawn"元素的情况下插入该元素。以这种方式插入元素时，值部分由默认构造函数进行初始化。对于整数值，这表示值由表达式 int()初始化。同内置类型的所有初始化器一样，int()返回 int 的默认值 0。所以用这种方式插入的新键都从 0 开始计数。

如果要检查键是否存在，可以使用 count()方法：

```
if(words.count("brawn"))
  cout<<"too much muscle<<endl;
```

如果键"brawn"存在，表达式 words.count()就返回 1，否则返回 0。因此，这个表达式确定是否有过多的内容，且不会在 map 中创建原来不存在的新数据项。

在 for 循环中填满了搭配词后，就可以在一个简单的循环中用 map 迭代器显示它：

```
for(WordIter iter=words.begin();iter!=words.end();++iter)
cout<<std::setw(6)<<iter->second<<" "<<iter->first<<endl;
```

2. 使用 multimap 容器

multimap 容器是允许数据项有重复键的 map 容器。其他方面它与 map 容器相同，并有一组类似的操作。数据项的键可以重复就表示，不能对 multimap 容器使用下标运算符。要给 multimap 容器添加元素，可以使用 insert()成员函数，这要求参数是一个适合于容器实例的 pair<Key, T>对象。

当然，一旦有了重复的键，就需要确定哪些元素对应于给定的键。multimap 容器的 find()成员函数返回一个迭代器，它指向与给定键对应的第一个元素。upper_bound()函数成员也返回一个迭代器，它指向的与键对应的第一个元素大于传送为参数的键。因此，可以组合使用这些函数确定包含与给定键对应的所有元素的半开间隔。下面是一个例子：

```
iter1 = mymultimap.find(aKey);
iter2 = mymultimap.upper_bound(aKey);
while(iter1 != iter2)                    //List objects for aKey
  std::cout << *iter1++ <<std::endl;
```

使用 multimap 以单词出现次数的降序方式从上一个例子中获取搭配词。这也可以从迭代器中获取一个键。由于这需要把 multimap 的第三个模板参数指定为函数对象，在 multimap 中获取元素的不同排序，所以最好先理解函数对象的工作方式。

3. 理解函数对象

函数对象是重载函数调用运算符()类型的对象。函数对象的惟一目的是封装一个函数，使之可以传送给另一个函数，作为参数。这么做比使用函数指针高效得多。对于那些要频繁调用的函数来说，例如执行计算的函数，这是非常重要的。

一般，定义函数对象类型的类模板如下所示：

```
template<class T>
struct less : public binary_function <T, T, bool>{
  bool operator() ( const T& left, const T& right) const {
    return left<right;
  }
}
```

模板用于把第三个 map 和 multimap 模板参数的默认值定义为 less<Key>，其中 Key 是键的类型。函数调用运算符看起来有点怪异，因为函数名称是 operator()，运算符是()。结果，函数是 operator()()。Less 类型的基类 binary_function 只是定义了一些使用 typedef 的类型，它们可以用于表示二元函数的参数类型和返回类型。这些类型在 less 模板的所有实例中都继承下来了。

显然，在 less<>类模板中，重载的函数调用运算符函数为 T 类型的对象定义了小于比较操作，只要类型 T 支持<运算符即可。less<>类模板重载了函数调用运算符，所以 less<T>类型的对象就可以直接用于执行比较。模板可以用于实例化对象，如下所示：

```
less<Box> compare_boxes;   // Function object for comparing Box objects
```

要比较 Box 对象，可以使用下面的语句：

```
if(compare_boxes(box1, box2))
  std::cout << "box1 is less than bos2";
```

compare_boxes 对象包含重载的函数调用运算符，所以可以象函数那样使用该对象。表达式 compare_boxes(box1, box2)等价于 compare_boxes.operator()(box1, box2)。使用这个操作比使用函数指针快得多，因为不需要解除指针的引用。这是对象的函数成员的常规调用方式。函数对象的概念可以用于任何操作，而不仅仅是比较。

STL 在<functional>头文件中定义了大量的标准函数对象类型，其中包括不同类别的函数对象类型，如表 20-7 所示。

表 20-7　函数对象的类别

类　　别	函 数 对 象
算术操作	plus<T>,minus<T>,multiplies<T>,divides<T>,modulus<T>,negates<T>
比较	equal_to<T>,not_equal_to<T>,greater<T>,less<T>, greater_equals<T>, less_equals<T>
逻辑操作	logical_and<T>, logical_or<T>,logical_not<T>

在这个头文件中还定义了许多其他的函数对象类型，详见编译器说明文档。

STL 在算法和关联容器中广泛使用了函数对象。显然，如果不喜欢 map 容器中由默认函数对象类型 less<T>生成的对象顺序，就可以为第三个 map 模板参数指定一个合适的值，改变该顺序。如下所示。

程序示例 20.13——用 multimap 颠倒搭配词的顺序

使用 multimap 容器扩展程序示例 20.12：

```
// Program 20.13 An inverted word collocation    File: prog20_13.cpp
#include <iostream>
#include <iomanip>
#include <string>
#include <sstream>
#include <map>
using std::cout;
using std::endl;
using std::string;
const string twister =
  "How much wood would a woodchuck chuck if a woodchuck "
  "could chuck wood? A woodchuck would chuck as much wood "
  "as a woodchuck could chuck if a woodchuck could chuck wood.";

int main() {
 typedef std::map<string, int> Collocation;     // Type for our map
 typedef Collocation::const_iterator WordIter;  // Iterator type for our map

 Collocation words;       //Map to store words and word counts

 std::istringstream text(twister);              // Text string as a stream
 std::istream_iterator<string> begin(text);     // Stream iterator
 std::istream_iterator<string> end;             // End stream iterator

 for( ; begin != end ; ++begin)                 //Iterate over the words in the stream
  words[*begin]++;                              //Store and increment word count

typedef std::multimap<int,string, std::greater<int> > WordRank;
typedef WordRank::const_iterator RankIter;

WordRank rank;

for(WordIter iter = words.begin () ; iter != words.end(); ++iter)
  rank.insert(std::make_pair(iter->second, iter->first));
```

```
  for(RankIter iter = rank.begin(); iter != rank.end(); ++iter)
    cout<< std::setw(6)<<iter->first<<" "<<iter->second<<endl;

  return 0;
}
```

这个程序的运行结果如下：

```
5 chuck
5 woodchuck
4 a
3 could
2 as
2 if
2 much
2 wood
2 would
1 A
1 How
1 wood.
1 wood?
```

例子的说明

仍把最初的搭配词放在 map 容器中。需要这么做的原因是 map 的键是惟一的，这可以确保文本中的每个词字符串只有一个数据项。

在 map 容器 words 中创建了搭配词后，添加两个 typedef：

```
typedef std::multimap<int, string, std::greater<int>> WordRank;
typedef WordRank::const_iterator RankIter;
```

第一个语句定义了一个 multimap 类型。注意颠倒了 map 模板实例中模板参数的顺序。这里键是 int 类型，对象是 string 类型。还在第三个模板参数中指定了自己的比较条件。

greater<>是一个标准 STL 函数对象类型，它在<functional>头文件中定义。这个头文件会自动包含在<map>头文件中。使用 greater<int>而不是默认的 less<>，就会以降序方式而不是升序方式显示等级。当然，使用 rbegin()和 rend()也可以以逆序方式迭代 WordRank 表，但这两个函数前面已介绍过了。

第二个 typedef 为 multimap 中的迭代器定义了一个方便的类型。

创建了 multimap 实例 rank 后，使用 insert()函数给它插入元素：

```
for(WordIter iter = words.begin () ; iter != words.end(); ++iter)
rank.insert(std::make_pair(iter->second, iter->first));
```

在使用 map 容器组合搭配词时，可以使用 operator[]()插入元素。这里不能这么做，因为 multimap 没有 operator[]()函数，这是由于在 multimap 中，下标运算符没有意义，键可以是非惟一的。insert()函数需要的参数是一个元素类型的对象，它组合了键和对象。本例是 pair<int, string>类型的对象。调用 make_pair()函数就可以从元素迭代器中为 map 创建这个对象。这是一个创建在<utility>头文件中定义的 pair<>对象的方便函数。<utility>头文件还定义了 pair<>模板，该模板定义了 map 元素的类型，会自动包含在<map>头文件中。make_pair()函数把第一个参数

用作键，第二个参数用作关联的对象，返回一个它创建的 pair<>对象。显然，pair<>模板的类型参数对应于传送给 make_pair()函数的参数类型。使用迭代器选择 pair<>对象的 second 和 first 成员，就可以从 map 中已提取出来的元素中获取这些类型。

在程序中，可以用下面的语句替换 make_pair()调用：

```
for(WordIter iter = words.begin () ; iter != words.end(); ++iter)
  rank.insert(std::pair<int, string>(iter->second, iter->first));
```

这里在 pair<>模板的实例中使用构造函数。读者可以自己选择更喜欢使用这种方法还是 make_pair()函数。

仔细研究一下两个 for 循环中的语句：

```
for(WordIter iter = words.begin () ; iter != words.end(); ++iter)
  rank.insert(std::make_pair(iter->second, iter->first));

for(WordIter iter = rank.begin () ; iter != rank.end(); ++iter)
  cout<<std::setw(6)<<iter-> first <<" "<<iter-> second <<endl;
```

第一个循环在 Collocation 迭代器中进行，第二个循环在 WordRank 迭代器中进行。map 容器 WordRank 是 Collocation 把顺序倒过来的结果，因此这非常容易出问题。在第一行中，iter->first 是一个词，但在第二行中，iter->second 是一个词，这是很容易混淆的。在这个例子中使用这两个语句是比较简单的，但在拥有许多相互关联的 map 容器的大型程序中，first 和 second 很容易令人烦心。在这种情况下，应在代码中添加一些注释，说明要执行的操作，并为迭代器使用不同的名称，说明它们是不同的。

20.6 性能和规范

数值算法库包含的函数执行的任务比较简单，很容易自己编码实现，所以应研究一下建立这些函数库的原因。本章的开头第一次遇到迭代器和算法时，介绍了如何编写函数 average()的许多例子。下面是 average()的一个经典实现，它使用了指针接口，非常简单，效率也很高。

```
// Simple average  function written without using any STL algorithms
double average(float*  first,   float*  last) {
  double sum = 0.0;
  for ( ; first != last ; ++first)
    sum += *first;
  return sum/(last-first);
}
```

或者，也可以使用内置的 STL 模板算法 accumulate()：

```
//An average  function based on the STL accumulate algorithm
template <typename RndIter>
double average (RndIter  first, RndIter  last) {
  return std::accumulate(first, last, 0)/(last-first);
}
```

accumulate()函数是一个在<numeric>头文件中定义的算法，它只是计算 first 到 last 范围内

所有元素的和。代码相当简单，但我们仅仅是为了用一个库函数代替三行简单的代码吗？为了回答这个问题，下面列出 STL 的集合实现：

```
// Possible  specialization of template accumulate<>  for float* iterators
#include <numeric>

template<>float std::accumulate(float* first, float*  last,  float  init) {
  double s0 = 0.0;
  double s1 = 0.0;
  double s2 = 0.0
  double s3 = 0.0;
  int burst_blocks= (last - first) / 4;
  float* burst = first + 4 * burst_blocks;

  for( ; first < burst ; first +=4) {
    s0 += first [0] ;
    s1 += first [1];
    s2 += first [2];
    s3 += first [3];
  }

  for( ; burst < last ; ++burst) {
    s0 += * burst;          // Capture up to three tail elements
  }
    return init + (s0 + s1 + s2 + s3);
  }
```

这是模板特殊化的一个例子。这个特殊化的目的是汇总通过指针引用的 float 值序列。这看起来比简单的循环要复杂，但其计算结果是相同的：

```
template<> float accumulate<float*>(float* first, flost* last, float init)
```

这个特殊化的关键和包含它的原因是 4 个语句块：

```
s0 += first[0];
s1 += first[1];
s2 += first[2];
s3 += first[3];
```

其理念是在计算中创建更多的并行。这个版本不是把所有的总计都放在一个变量中，这会产生瓶颈，而是同时拥有 4 个小计。运行速度比较快的机器可以同时进行这 4 个相加运算。

特殊化如何跟简单的循环相比较？这取决于编译器、STL 的实现、vector 容器的大小和处理器的能力。特殊化可能并不是最快的，但至多可以快 3 倍。当然，如果计算只需要几微秒的时间，其优势就不非常明显了。但是，如果计算涉及到大量的数据，执行时间是几分钟或几小时，节省下来的时间就比较可观了。

注意，特殊化还提供了更高的精确度。在第 2 章第一次遇到浮点数数据类型时说过，汇总浮点数值是很冒险的。其结果可能非常不准确，因为在汇总过程中会累加圆整错误。这个特殊化使用更精确的 double 类型进行内部计算，所以比简单的循环更准确。真正优秀的特殊化(比

这个例子复杂)可以利用更高明的技巧消除几乎所有的圆整错误，同时仍保持高性能。自己编写这种代码是非常困难的。

即使最简单的算法也可以用令人惊讶的智能方式编写。STL 的实现总是在不断地改进。如果性能非常重要，则只要有机会，就应使用 STL 算法。它们比自己编写的任何代码都复杂。

20.7 本章小结

本章简要介绍了一些新技术，指出了其中有趣的部分。尽管我们只是讨论了 STL 的皮毛，但读者应能自己扩展这部分知识。

本章的要点如下：

- STL 通过三类模板提供了功能：容器、迭代器和算法。
- 容器提供了存储和组织任意类型的对象的各种方式，但要存储的对象类型必须满足元素的基本要求。
- 迭代器是操作方式像指针的对象。迭代器是智能指针的示例。
- 迭代器可以访问容器中的对象，或从流中提取对象。
- 迭代器成对使用时，可以在序列的半开间隔中定义一组对象。第一个对象包含在间隔中，而最后一个对象不包含在间隔中。
- 算法是一般化的标准函数，可以处理迭代器指定的对象集。
- 算法独立于容器，但可以通过迭代器应用于任何容器中的对象。

20.8 练习

1. 编写一个程序，使用 vector 存储任意个城市，这些城市借助键盘读取为 string 对象，再输出它们。

2. 在上一题中添加代码，使用 sort()算法按照升序对城市排序，之后输出它们。

3. 编写一个程序，借助键盘读取任意个名称和关联的电话号码(其格式是"Laurel, Stan" 5431234)，把它们存储在 map 容器中，给定一个名称，就可以提取电话号码。在输入一系列名称和电话号码后，对 map 进行随机访问，提取一个随机电话号码。

4. 利用上一题的代码，使用一个迭代器列出 map 的内容。

5. 给上一题添加代码，用 multimap 替换 map 容器，为给定的名称列出所有的电话号码，作为查询的响应。

附录 A　ASCII　码

前 32 个 ASCII(American Standard Code for Information Interchange，美国信息互换标准代码)字符提供了控制功能。其中有许多字符本书并没有使用，这里只是为了完整起见，才列出它们。在表 A-1 中，只包含前 128 个字符。剩余的 128 个字符包括其他特殊符号和国家字符集的字母。

表 A-1　前 128 个 ASCII 字符

十　进　制	十六进制	字　　符	控　　制
000	00	空	NUL
001	01	☺	SOH
002	02	•	STX
003	03	♥	ETX
004	04	♦	EOT
005	05	♣	ENQ
006	06	♠	ACK
007	07	•	BEL(可以听到的铃声)
008	08		退格
009	09		HT
010	0A		LF(Line Feed)
011	0B		VT(Vertical feed)
012	0C		FF(Form free)
013	0D		CR(Carriage return)
014	0E		SO
015	0F		SI
016	10		DLE
017	11		DC1
018	12		DC2
019	13		DC3
020	14		DC4
021	15		NAK
022	16		SYN
023	17		ETB
024	18		CAN
025	19		EM
026	1A	→	SUB

(续表)

十 进 制	十六进制	字 符	控 制
027	1B	←	ESC(Escape)
028	1C	L	FS
029	1D		GS
030	1E		RS
031	1F		US
032	20	空格	
033	21	!	
034	22	"	
035	23	#	
036	24	$	
037	25	%	
038	26	&	
039	27	'	
040	28	(	
041	29	)	
042	2A	*	
043	2B	+	
044	2C	,	
045	2D	-	
046	2E	.	
047	2F	/	
048	30	0	
049	31	1	
050	32	2	
051	33	3	
052	34	4	
053	35	5	
054	36	6	
055	37	7	
056	38	8	
057	39	9	
058	3A	:	
059	3B	;	
060	3C	<	

(续表)

十 进 制	十六进制	字 符	控 制
061	3D	=	
062	3E	>	
063	3F	?	
064	40	@	
065	41	A	
066	42	B	
067	43	C	
068	44	D	
069	45	E	
070	46	F	
071	47	G	
072	48	H	
073	49	I	
074	4A	J	
075	4B	K	
076	4C	L	
077	4D	M	
078	4E	N	
079	4F	O	
080	50	P	
081	51	Q	
082	52	R	
083	53	S	
085	55	U	
086	56	V	
087	57	W	
088	58	X	
089	59	Y	
090	5A	Z	
091	5B	[	
092	5C	\	
093	5D	]	
094	5E	^	
095	5F	_	

(续表)

十 进 制	十六进制	字 符	控 制
096	60	'	
097	61	a	
098	62	b	
099	63	c	
100	64	d	
101	65	e	
102	66	F	
103	67	g	
104	68	h	
105	69	I	
106	6A	J	
107	6B	k	
108	6C	l	
109	6D	m	
110	6E	n	
111	6F	o	
112	70	p	
113	71	q	
114	72	r	
115	73	s	
116	74	t	
117	75	u	
118	76	v	
119	77	w	
120	78	x	
121	79	y	
122	7A	z	
123	7B	{	
124	7C	|	
125	7D	}	
126	7E	~	
127	7F	DEL(delete)	

附录 B　C++关键字

关键字是在 C++语言中赋予特殊含义的字，因此在程序中不能把它们用作名称。已定义的关键字如表 B-1 所列：

表 B-1　已定义的关键字

关　键　字	关　键　字	关　键　字
asm	false	sizeof
auto	float	static
bool	for	static_cast
break	friend	struct
case	goto	switch
catch	if	template
char	inline	this
class	int	throw
const	long	true
const_cast	mutable	try
continue	namespace	typedef
default	new	typeid
delete	operator	typename
do	private	union
double	protected	unsigned
dynamic_cast	public	using
Else	register	virtual
Enum	reinterpret_cast	void
explicit	return	volatile
export	short	wchar_t
extern	signed	while

还有一些关键字是为 C++中的按位运算符和逻辑运算符保留的，也不能用于其他目的。如表 B-2 所列。

表 B-2　保留关键字

关　键　字	关　键　字	关　键　字
and	compl	or_eq
and_eq	not	xor
bitand	not_eq	xor_eq
bitor	or	

附录C　标准库头文件

C++标准库的所有头文件都没有扩展名。C++标准库的内容在总共 50 个标准头文件中定义，其中的 18 个提供了标准 C 库的功能。<cname>形式的标准头文件(<complex>例外)其内容与 ISO 标准 C 包含的 name.h 头文件相同，但在一些实例中采用，以容纳 C++中更大的扩展语言功能。在<cname>形式的标准头文件中，与宏相关的名称在全局作用域中定义，其他名称在 std 命名空间中声明。在 C++程序中还可以使用 name.h 形式的标准 C 库头文件名，因为它们可以在标准 C++中使用，与标准 C 兼容。此时，其内容与对应的<cname>头文件等价，但所有的名称都可以在全局作用域中使用。

C++标准库的内容分为 10 类，如下所示。

语言支持	输入/输出	诊断	一般工具
字符串	容器	迭代器支持	算法
数值操作	本地化		

这些类别提供了许多功能，有时其定义和声明散布在几个头文件中。

C.1　语言支持

标准库中与语言支持功能相关的头文件如表 C-1 所列。

表 C-1　标准库中与语言支持功能相关的头文件

头　文　件	描　　述
<cstddef>	定义宏 NULL 和 offsetof，以及其他标准类型 size_t 和 ptrdiff_t。与对应的标准 C 头文件的区别是，NULL 是 C++空指针常量的补充定义，宏 offsetof 接受结构或联合类型参数，只要它们没有成员指针类型的非静态成员即可
<limits>	提供与基本数据类型相关的定义。例如，对于每个数值数据类型，它定义了可以表示出来的最大值和最小值以及二进制数字的位数。第 3 章使用了这些库功能
<climits>	提供与基本整型数据类型相关的 C 样式定义。这些信息的 C++样式在<limits>中提供
<cfloat>	提供与基本浮点型数据类型相关的 C 样式定义。这些信息的 C++样式在<limits>中提供
<cstdlib>	提供支持程序启动和终止的宏和函数。这个头文件还声明了许多其他杂项函数，例如搜索和排序函数，从字符串转换为数值等函数。它与对应的标准 C 头文件 stdlib.h 不同，定义了 abort(void)。Abort()函数还有额外的功能，它不为静态或自动对象调用析构函数，也不调用传送给 atexit()函数的函数。它还定义了 exit()函数的额外功能，可以释放静态对象，以注册的逆序调用用 atexit()注册的函数。清除并关闭所有打开的 C 流，把控制权返回给主机环境

(续表)

头 文 件	描 述
<new>	支持动态内存分配
<typeinfo>	支持变量在运行期间的类型标识
<exception>	支持异常处理，这是处理程序中可能发生的错误的一种方式
<cstdarg>	支持接受数量可变的参数的函数。即在调用函数时，可以给函数传送数量不等的数据项。它定义了宏 va_arg、va_end 和 va_start，以及 va_list 类型
<csetjmp>	为 C 样式的非本地跳跃提供函数。这些函数在 C++程序中不常用
<csignal>	为中断处理提供 C 样式支持

C.2 输入/输出

表 C-2 中的头文件支持流输入/输出。

表 C-2 支持流输入/输出的头文件

头 文 件	描 述
<iostream>	支持标准流 cin、cout、cerr 和 clog 的输入输出，它还支持多字节字符标准流 wcin、wcout、wcerr 和 wclog
<iomanip>	提供操纵程序，允许改变流的状态，从而改变输出的格式
<ios>	定义 iostream 的基类
<istream>	为管理输入流缓存区的输入定义模板类
<ostream>	为管理输出流缓存区的输入定义模板类
<sstream>	支持字符串的流输入输出
<fstream>	支持文件的流输入输出
<iosfwd>	为输入输出对象提供向前的声明
<streambuf>	支持流输入输出的缓存
<cstdio>	为标准流提供 C 样式的输入输出
<cwchar>	支持多字节字符的 C 样式输入输出

C.3 诊断

C++诊断功能在三个头文件中定义。如表 C-3 所示。

表 C-3 与诊断功能相关的头文件

头 文 件	描 述
<stdexcept>	定义标准异常。异常是处理错误的方式
<cassert>	定义断言宏，用于检查运行期间的情形
<cerrno>	支持 C 样式的错误信息

C.4 一般工具

表 C-4 中列出的头文件组定义了 C++库的其他组件使用的工具函数。这些工具函数也可以在自己的程序中使用。

表 C-4 定义工具函数的头文件

头 文 件	描 述
<utility>	定义重载的关系运算符，简化关系运算符的写入，它还定义了 pair 类型，该类型是一种模板类型，可以存储一对值。这些功能在库的其他地方使用
<functional>	定义了许多函数对象类型和支持函数对象的功能，函数对象是支持 operator()()函数调用运算符的任意对象
<memory>	给容器、管理内存的函数和 auto_ptr 模板类定义标准内存分配器
<ctime>	支持系统时钟函数

C.5 字符串

表 C-5 列出的头文件提供了处理字符串对象和 C 样式字符串的功能。

表 C-5 支持字体串处理的头文件

头 文 件	描 述
<string>	为字符串类型提供支持和定义，包括单字节字符串(由 char 组成)的 string 和多字节字符串(由 wchar_t 组成)
<cctype>	单字节字符类别
<cwctype>	多字节字符类别
<cstring>	为处理非空字节序列和内存块提供函数。这不同于对应的标准 C 库头文件，几个 C 样式字符串的一般 C 库函数被返回值为 const 和非 const 的函数对替代了
<cwchar>	为处理、执行 I/O 和转换多字节字符序列提供函数，这不同于对应的标准 C 库头文件，几个多字节 C 样式字符串操作的一般 C 库函数被返回值为 const 和非 const 的函数对替代了
<cstdlib>	为把单字节字符串转换为数值、在多字节字符和多字节字符串之间转换提供函数

C.6 容器

表 C-6 中的头文件定义了用于创建容器类的模板。

表 C-6　定义容器类的模板的头文件

头　文　件	描　　述
<vector>	定义 vector 序列模板，这是一个大小可重新设置的数组类型，比普通数组更安全、更灵活
<list>	定义 list 序列模板，这是一个序列的链表，常常在任意位置插入和删除元素
<deque>	定义 deque 序列模板，支持在开始和结尾的高效插入和删除操作
<queue>	为队列(先进先出)数据结构定义序列适配器 queue 和 priority_queue
<stack>	为堆栈(后进先出)数据结构定义序列适配器 stack
<map>	map 是一个关联容器类型，允许根据键值搜索值，其中键值是惟一的，且按照升序存储。multimap 类似于 map，但键不必是惟一的
<set>	set 是一个关联容器类型，用于以升序方式存储惟一值。multiset 类似于 set，但值不必是惟一的
<bitset>	为固定长度的位序列定义 bitset 模板，它可以看作固定长度的紧凑型 bool 数组

C.7　迭代器支持

只有一个头文件支持迭代器的定义，如表 C-7 所示。

表 C-7　支持迭代器的头文件

头　文　件	描　　述
<iterator>	给迭代器提供定义和支持

C.8　一般用途的算法

算法有两个头文件。如表 C-8 所示。

表 C-8　有关算法的头文件

头　文　件	描　　述
<algorithm>	提供一组基于算法的函数，包括置换、排序、合并和搜索
<cstdlib>	声明 C 标准库函数 bsearch()和 qsort()，进行搜索和排序
<ciso646>	允许在代码中用 and 代替&&

C.9　数值操作

这个组包括操作复杂的数值和数学函数，共有 5 个头文件。如表 C-9 所示。

表 C-9　有关数值操作的头文件

头　文　件	描　　述
<complex>	支持复杂数值的定义和操作

(续表)

头 文 件	描 述
<valarray>	支持数值矢量的操作
<numeric>	在数值序列上定义一组一般数学操作，例如 accumulate 和 inner_product
<cmath>	这是 C 数学库，其中还附加了重载函数，以支持 C++约定
<cstdlib>	提供的函数可以提取整数的绝对值，对整数进行取余数操作

C.10 本地化

本地化提供了处理根据区域而变化的事件的功能，例如货币符号，日期表示和排序序列。它有两个头文件。如表 C-10 所示。

表 C-10 有关本地化的头文件

头 文 件	描 述
<locale>	提供的本地化包括字符类别、排序序列以及货币和日期表示
<clocale>	对本地化提供 C 样式支持

附录 D　运算符的优先级和相关性

在表达式中，不同运算符的执行次序取决于该运算符的优先级。但一些表达式的执行顺序是不确定的。C++的 ISO/ANSI 标准没有显式定义运算符的优先级，但可以从语法规则推算出来。运算符的优先级是大多数表达式确定执行顺序的一种简单方式，每个运算符的优先级都列在表 D-1 中。优先级较高的运算符先执行，优先级较低的运算符后执行。在表 D-1 中，运算符按照优先级的降序排列，优先级最高的运算符排在最前面。同一组中的运算符有相同的优先级。

表 D-1　运算符优先级

组	描　　述	运　算　符
1	范围解析运算符	::
2	直接成员选择运算符	.
	间接成员选择运算符	->
	下标运算符	[]
	函数调用运算符	()
	后缀递增运算符	++
	后缀递减运算符	--
3	一元加运算符	+
	一元减运算符	-
	前缀递增运算符	++
	前缀递减运算符	--
	逻辑非运算符	!
	按位补运算符	~
	地址运算符	&
	解除引用运算符	*
	显式强制转换运算符(旧样式)	(type)
	对象或类型占用字节数运算符	sizeof
	分配内存运算符	new
	释放内存运算符	delete
	在编译期间进行的强制转换运算符	static_cast
	在运行期间进行的动态强制转换运算符	dynamic_cast
	强制转换为 const 的运算符	const_cast
	未检查的强制转换运算符	reinterpret_cast
	类型标识运算符	typeid

(续表)

组	描　　述	运　算　符
4	直接指向成员选择运算符	.*
	间接指向成员选择运算符	->*
5	相乘运算符	*
	相除运算符	/
	取模运算符	%
6	二元加运算符	+
	二元减运算符	-
7	左移运算符	<<
	右移运算符	>>
8	小于运算符	<
	小于等于运算符	<=
	大于运算符	>
	大于等于运算符	>=
9	等于运算符	==
	不等于运算符	!=
10	按位与运算符	&
11	按位异或运算符	^
12	按位或运算符	|
13	逻辑与运算符	&&
14	逻辑或运算符	||
15	赋值运算符	=
	应用运算符再赋值	*= /= %= += -= &= ^= |= <<= >>=
16	条件运算符	?:
17	抛出异常运算符	Throw
18	逗号运算符	,

运算符的相关性确定了它在表达式中如何与其操作数组合在一起。所有的一元运算符和所有的赋值运算符都是右相关的。其他运算符都是左相关的。

赋值运算符的右相关性是指语句

```
x=y=z=t;
```

等价于下面的语句：

```
x=(y=(z=t));
```

赋值语句 z=t 最先执行。该赋值表达式的值与 t 的值相同，再把该值赋予 y，最后把 y 的值赋予 x。

二元加运算符的左相关性是指，语句

```
result=x+y+z+t;
```

等价于下面的语句：

```
result=((x+y)+z)+t;
```

把 x 和 y 加起来，所得的结果再与 z 相加，最后再把 t 加进来。

注意，表达式的优先级和相关性并不能完全确定表达式的计算顺序。如果计算顺序还没有确定，就由编译器确定。例如，表达式(x * y) - (z* t)中，括号中表达式的执行顺序是不确定的，在不同的编译器上执行顺序可能不同。因此，最好不要编写结果取决于括号中表达式的执行顺序的表达式。

附录E　理解二进制和十六进制数

计算机以二进制格式存储数字，使用二进制算术操作它们。在程序代码中使用十六进制数表示法是表示二进制值的一种方便而紧凑的方式。所以理解二进制和十六进制数对全面理解C++是非常重要的。

E.1　二进制数

首先考虑一下在表示常见的十进制数(如324或911)时会做什么。显然，324是表示三百二十四，911表示九百一十一。更明确地说，这两个数表示：

324是：$3\times10^2+2\times10^1+4\times10^0$，也就是$3\times10\times10+2\times10+4$
911是：$9\times10^2+1\times10^1+1\times10^0$，也就是$9\times10\times10+1\times10+1$

这称为十进制表示法，因为这是建立在10的幂的基础之上(来源于拉丁词decimalis，其含义是什一税，即税的10%)。

以这种方式表示数值非常方便，因为人有10根手指或10根脚趾或者10个任何类型的附属物。但是，这对PC就不太方便了，因为PC主要以开关为基础，即开和关，加起来只有2，而不是10。这就是计算机用基数2来表示数值，而不是用基数10的主要原因。这称为二进制计数系统。用基数10表示数字，数字可以是0到9，而二进制数字只能是0或1，当只用开/关来表示数字时，这是很理想的。按照基数为10的计数系统的方法，二进制数1101就可以分解为：

$1\times2^3+1\times2^2+0\times2^1+1\times2^0$，也就是$1\times2\times2\times2+1\times2\times2+0\times2+1$

计算得13(十进制系统)。在表E-1中，列出了用8个二进制数字表示的对应的十进制值(二进制数字常常称为位)。

表E-1　与二进制数字对应的十进制值

二　进　制	十　进　制	二　进　制	十　进　制
0000 0000	0	1000 0000	128
0000 0001	1	1000 0001	129
0000 0010	2	1000 0010	130
…	…	…	…
0001 0000	16	1001 0000	144
0001 0001	17	1001 0001	145
…	…	…	…
0111 1100	124	1111 1100	252

(续表)

二 进 制	十 进 制	二 进 制	十 进 制
0111 1101	125	1111 1101	253
0111 1110	126	1111 1110	254
0111 1111	127	1111 1111	255

注意使用前 7 位可以表示从 0 到 127 的数，一共 2^7 个数，使用全部 8 位可以表示 256(即 2^8)个数。一般情况下，如果有 n 位，就可以表示 2^n 个整数，其值从 0 到 2^n-1。

在计算机中，二进制数相加是非常容易的，因为对应数字加起来的进位只能是 0 或 1，所以处理过程会非常简单。图 E-1 中的例子演示了两个 8 位二进制数相加的过程。

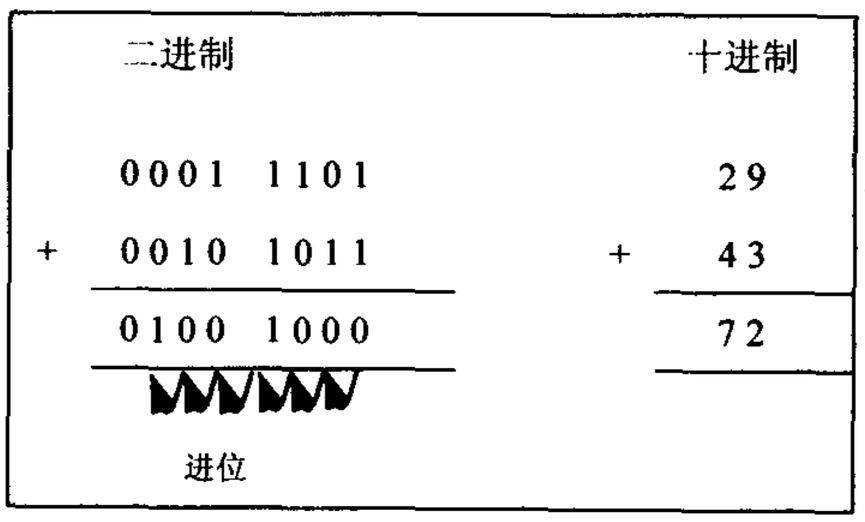


图 E-1 二进制数的相加

E.2 十六进制数

在处理很大的二进制数时，就会有一个小问题。如：

```
1111 0101 1011 1001 1110 0001
```

在实际应用中，二进制表示法显得比较烦琐，如果把这个二进制数表示为十进制数，结果为 16,103,905，这个 8 位的十进制数没有什么价值。还可以用更长的二进制位数表示更大的十进制数。显然，我们需要一种更高效的方式来表示这个数，但十进制并不总是合适的。有时(如第 3 章所示)需要能够指定从右开始算起的第 10 位和第 24 位的数字为 1(不考虑二进制表示法的系统开销)。用十进制整数来完成这个任务是非常麻烦的，而且很容易出现计算错误。比较简单的解决方案是使用十六进制表示法，即数字以 16 为基数表示。

基数为 16 的算术就方便得多，它与二进制也相得益彰。每个十六进制的数字可以是从 0 到 15 的值(从 10 到 15 的数字用 A 到 F 表示，如表 E-2 所示)，从 0 到 15 的数值就分别对应于用 4 个二进制数字表示的值。

表 E-2 十六进制数表示为二进制数

十 六 进 制	十 进 制	二 进 制
0	0	0000
1	1	0001
2	2	0010

(续表)

十六进制	十进制	二进制
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
A	10	1010
B	11	1011
C	12	1100
D	13	1101
E	14	1110
F	15	1111

因为一个十六进制数对应于4个二进制数，所以可以把较大的二进制数表示为一个十六进制数，方法是从右开始，把每4个二进制数字组成一组，再用对应的十六进制数表示每个组。例如，二进制数：

```
1111 0101 1011 1001 1110 0001
```

如果依次提取每4个二进制数字，用对应的十六进制数表示每个组，这个数字用十六进制表示，就得到：

```
F 5 B 9 E 1
```

所得的6个十六进制数字分别对应于6组4个二进制数字。为了证明这适用于所有的情况，下面用十进制表示法把这个数值直接从十六进制转换为十进制。

这个十六进制数的计算如下。F5B9E1转换为十进制值：

$$15\times16^5+5\times16^4+11\times16^3+9\times16^2+14\times16^1+1\times16^0$$

计算得：

```
15,728,640+327,680+45,056+2,304+224+1
```

最后相加的结果与把二进制数转换为十进制数的结果相同：16,103,905。

E.3 负的二进制数

二进制算术需要理解的另一个方面是负数。前面一直假定所有的数字都是正的。从乐观的角度来看是这样，所以我们目前已对二进制数有了一半的认识。但在实际中还会遇到负数，从

悲观的角度来看，我们对二进制数的认识仅仅是一半。在计算机中，是如何表示负数的？我们只能按照自己的意愿来处理二进制数字，所以解决方案必须是使用其中的一个二进制数字。

对于允许是负数的数值(称为带符号的数值)，必须先确定一个固定的长度(换言之，就是二进制数字的位数)，再把最左边的二进制数字设置为符号位。必须固定位数，这样才能避免符号位与其他位的混淆。

因为计算机的内存由 8 位字节组成，所以二进制数字要存储在多个 8 位中(通常是 2 的幕)，即有些数字是 8 位，有些数字是 16 位，有些数字是 32 位等。只要知道每个数值的位数，就可以找到符号位，它应是最左边的那一位。如果符号位是 0，该数值就是正的，如果它是 1，该数值就是负的。

似乎这就解决了问题，但实际上并非如此。每个数字都是由一个符号位和给定的位数组成，其中符号位为 0 表示正数，符号位为 1 表示负数，给定的位数指定数字的绝对值，换言之，就是无符号的数字。把+6 改为 - 6 只需要把符号位从 0 改为 1。但是，这个表示方法需要大量的系统开销，而且系统开销的大小取决于对这个数字表示方法执行的算术运算的复杂程度。因此，大多数计算机都采用了另一种方法。

理想情况下，当两个整数相加时，计算机不应不检查两个数字是否为负。我们希望使用常规的“加”来生成相应的结果，而不考虑操作数的符号。相加操作把相应的二进制数字合并在一起，得到相应的位，在必要时把 1 进到下一位上。如果把二进制的 - 8 加到+12 上，答案就应是+4。如果把+3 加到+8 上，也采用相同的方式操作。

如果用简化的解决方案来执行这一操作，也就是把正数的符号位设置为 1，使它变成负数，再执行算术运算，并进行常规的进位，答案就是错误的：

```
12 转换为二进制:          0000 1100
- 8 转换为二进制:         1000 1000
```

如果把它们加起来，结果是 1001 0100。

答案是 - 20，这可不是我们希望的结果+4，它的二进制应是 0000 0100。此时读者会认为，“没有把符号作为另一个位”。但这里就是这么做的。

下面看看计算机如何表示 - 8，即从+4 中减去+12，得到正确的结果：

```
+4 转换为二进制:          0000 0100
+12 转换为二进制:         0000 1100
```

从前者中减去后者，结果是 1111 1000。

对于右边的 4 位数字，必须借 1，才能进行减法，这正是我们在执行十进制算术时所进行的操作。结果就是 - 8，即使它看起来不是 - 8，但其值的确是 - 8。再用二进制把该值与+12 或+15 相加，就会得到正确的结果。当然，如果想得到 - 8，总是可以从 0 中减去+8。

在从 4 中减去 12 或从 0 中减去+8 时，究竟进行了什么操作？实际上是对负二进制数值采用了 2 的补码形式。通过一个可以在头脑中进行的简单过程就可以从任何正的二进制数中获得这种形式。这里需要做一个约定，以避免解释它为什么有效。下面看看如何从正数中构建负数的 2 的补码形式，读者也可以自己证明这是有效的。

现在回到前面的例子，给 - 8 构建 2 的补码形式。首先把+8 转换为二进制：

```
0000 1000
```

现在反转每个二进制数字，即把 0 变成 1，把 1 变成 0：

```
1111 0111
```

这称为 1 的补码形式，如果给这个数加上 1，就得到了 2 的补码形式：

```
1111 1000
```

这就是从+4 中减去+12，得到的 - 8 的二进制表示。为了确保正确，下面对 - 8 和+12 进行正常的相加操作：

```
+12 转换为二进制          0000 1100
 - 8 转换为二进制         1111 1000
```

把这两个数加在一起，得到： 0000 0100

答案就是 4。这是正确的。左边所有的 1 都向前进位，这样该位的数字就是 0。最左边的数字应进位到第 9 位，即第 9 位应是 1，但这里不必担心这个第 9 位，因为在前面计算-8 时从前面借了一位，在此正好抵消。实际上，这里做了一个假定，符号位 1 或 0 永远放在最左边。读者可以自己试验几个例子，就会发现这种方法总是有效的。最妙的是，使用负数的 2 的补码形式使计算机上的算术计算非常简单快速。

E.3.1 Big-Endian 和 Little-Endian 系统

如前所述，一般的整数在内存的一系列连续字节中存储为二进制值，通常存储为 2、4 或 8 个字节。字节采用什么顺序是非常重要的。

把十进制数 262657 存储为 4 字节二进制值。选择这个值是因为它的二进制值是

```
0000 0000 0000 0100 0000 0010 0000 0001
```

每个字节的位模式都很容易与其他字节区分开来。

如果使用 Inter PC，该数字就存储为：

字节地址	00	01	02	03
数据位	0000 0001	0000 0010	0000 0100	0000 0000

可以看出，值中最重要的 8 位是都为 0 的那些位，它们都存储在地址最高的字节中，换言之，就是最右边的字节。最不重要的 8 位存储在地址最低的字节中，即最左边的字节中。这种安排形式称为 Little-Endian。

如果使用大型计算机、risc 工作站或基于 Motorola 处理器的 Mac 机器，该数值在内存中存储为：

字节地址	00	01	02	03
数据位	0000 0000	0000 0100	0000 0010	0000 0001

字节现在的顺序是反的，最重要的 8 位存储在最左边的字节中，即地址最低的字节中。这种安排形式称为 Big-Endian。

注意：

无论字节顺序是 Big-Endian 还是 Little-Endian，在每个字节中，最重要的位都放在左边，最不重要的位都放在右边。

这非常有趣，但为什么这很重要？在大多数情况下，这并不重要。即使不知道执行代码的计算机是采用 Big-Endian 还是 Little-Endian，都可以编写出有效的 C++程序。但是，在处理来自另一台机器的二进制数据值，这就很重要了。二进制数据会写入文件或通过网络传送为一系列字节，此时必须解释它们。如果数据源所在的机器使用的 Endian 形式与运行代码的机器不同，就必须反转每个二进制值的字节顺序，否则就会出错。

注意：

对于那些了解一些有趣背景信息的读者来说，大概知道术语 Big-Endian 和 Little-Endian 取自 Jonathan Swift 编著的《格利佛游记》。Lilliput 的国王命令所有的国民必须在鸡蛋的小端磕破鸡蛋。这是因为国王的儿子按照在鸡蛋的大端磕破鸡蛋的传统方式玩耍划破了自己的手指。守法的普通 Lilliput 国民称为 Little-Endian，Lilliputian 王国中一些坚持在鸡蛋的大端磕破鸡蛋的反传统主义者就是 Big-Endian。许多人因此被判死刑。

附录F 项目示例

现在读者已通读了本书。下面就应用书中的知识建立一个结构合理的小项目。本附录将提出问题，并给出一些提示。

注释：

本项目的源代码可在 apress 网站(http://www.appress.com)上获得。

F.1 提纲

本项目示例的目的是创建一个面向对象的程序，跟踪教育机构中教师和学生的信息。这些信息存储在一系列 Teacher 和 Student 记录中，这些记录包含了下面的通用属性。如表 F-1 所示。

表 F-1 项目示例的通用属性

属 性	类 型	最大长度	限制/注释
姓	字母	20 字符	
名	字母	20 字符	
地址 1	混合字符	30 字符	第一行是街道地址
地址 2	混合字符	30 字符	第二行
地址 3	混合字符	30 字符	最后一行
城市	字母	20 字符	
州	字母	3 字符	
邮政编码	数字	6 字符	
电话号码	数字	8 字符	格式必须是###-####

所有的 Student 记录都必须有下述属性。如表 F-2 所示。

表 F-2 Student 记录的附加属性

属 性	类 型	最大长度	限制/注释
学生 ID	混合字符	6 字符	
成绩(或 GPA)	数字	-	必须在 0 到 100 之间

所有的 Teacher 记录都有下述属性。如表 F-3 所示。

表 F-3　Teacher 记录的附加属性

属　　性	类　　型	最 大 长 度	限制/注释
教学经验(年)	数字	-	必须是正整数
薪水	数字	-	必须是正整数

该程序应是菜单驱动的，用户应可以执行下述操作：

- 添加记录
- 删除记录
- 搜索记录
- 显示记录
- 清除所有的记录
- 把记录保存到数据库文件中
- 从数据库文件中提取记录集

在添加记录时，应正确输入 Student 和 Teacher 记录的所有属性。街道地址至多由三行组成，如果少于三行，在行中输入句点(.)就可以停止输入。

在删除记录时，应提示用户输入要删除的 Teacher 或 Student 记录的姓。一旦删除，就不能搜索或显示该记录。

在搜索记录时，应提示用户输入要查找的记录的姓。接着，程序就显示包含指定姓的记录的所有属性(例如，如果搜索的是 Student 记录，就应显示 GPA 和学生 ID 字段)。删除的记录不能搜索。

在显示记录时，应给用户提供下述选项：

- 显示所有的 Student 记录及其属性
- 显示所有的 Teacher 记录及其属性
- 显示所有的 Teacher 记录和 Student 记录及其属性

记录应以适当的格式保存在一个纯文本文件中。在保存时，应提示用户输入文件名，应确认在保存之前可以创建该文件。

在读取文件时，程序应提示用户输入文件名，并验证该文件可以打开。记录应从文件中读取，并添加到当前记录集中。

改进项目的规范

提出了问题后，就可以看看规范中的一些灰色区域：

- 由于是按姓搜索数据，因此应把姓字段用作搜索键。考虑一下如何处理重名的情况——如果数据库中有两个 Smith，如何对 Smith 进行搜索？
- 每个人都有一个地址，但如果某个人没有电话号码，该怎么办？

F.2　开发人员应注意的问题

提出问题后，再给出实现该程序的一些建议。

这个项目的整体设计基于下述三个简单的规则：

- 让对象自己照顾自己
- 实现一个容器类来组织对象
- 使用 main()函数作为“中枢”。它指导着程序的执行，但不对各个对象发号施令，更不用说添加和删除容器中的对象了。

Person 类是项目的核心。它实现了项目中所有对象的基本功能，是这个项目中其他类如 Student 类和 Teacher 类的基类。这个程序中的所有对象都是 Teacher 或 Student 类型，这些派生类只根据其类型实现特定的功能。例如，Teacher 类包含一个对教师来说是惟一的成员变量，表示“做教学工作的时间”。

每个类都执行自己的有效性检查。类成员函数可检查用户输入的有效性，因为按照规则，对象需要负责保持其数据的完整性。如果在类的外部进行了有效性检查，就表示把对象放在外部实体中，这样，代码的维护会非常麻烦，并会违反 OO 设计的原则。

用于封装 Student 和 Teacher 对象的容器来自标准模板库 deque。这个程序并没有扩展其功能，因为本项目只需添加、删除和搜索对象。选择 deque 容器是因为它易于使用，但也可以使用 map 或 multiset 容器。我们使用 deque 容器存储 Person*类型的对象，因为根据多态性，该容器可以存储 Student 和 Teacher 对象，它们都派生于 Person 类。

main()函数和其他包含在 MainProg.cpp 文件中的函数没有太多的责任。首先也是比较重要的，main()指导程序的执行。它也通过菜单验证用户的选择，而且利用了容器类。main()函数不包含与对象相关的代码，但指导程序的执行。

F.2.1 Person 类

Person 类包含 Student 和 Teacher 派生类的基本结构。它是 Student 和 Teacher 类的基类，与类相关的大多数代码都放在这里。其结构充分利用了成员变量的数据隐藏功能：

- 成员函数 setup()声明为 private，因为它仅在构造函数内部调用。
- 虚拟函数 set_other_info()也是私有函数，因为它在派生类的公共成员函数中调用。
- 所有的成员变量都是私有的，以充分利用数据隐藏功能以及该功能带来的安全性。

F.2.2 Student 和 Teacher 派生类

Student 和 Teacher 类派生于 Person 类，这些类都包含该类型特有的成员变量和成员函数。每个类还实现了虚拟函数 set_other_info()的独特版本。

F.2.3 容器

容器类用 Person 类的指针创建，以充分利用多态性。既然采用了这种实现方式，容器就可以包含 Student 和 Teacher 派生类的元素。维护很简单，因为只有一个容器需要维护。只要需要容器中的记录，就可以使用简单的 for 循环来完成。在这个容器中执行下述函数：

- 添加元素
- 删除元素

- 搜索元素

在退出程序之前，容器需要清空，就像我们自己总是要整理一样。

F.2.4 保存和恢复数据

从程序中退出时，会丢失所输入的所有数据，除非采取措施把它们存储在磁盘上。因为我们不希望项目过于复杂，所以采用了一种非常简单的方法，允许用户以文本形式把记录保存在标准磁盘文件中。这不仅使得使用 iostream 类实现起来非常简单，而且有助于调试，因为可以看到把什么数据写到了文件中。

可以设计自己的记录格式，但建议在文件的第一行中包含某些特殊的标记，以验证是否读取了正确类型的文件。

改进这个程序有许多方式，但这是建立和扩展其功能的基础(例如，可以添加一个 Principal 类)。它还把 C++的许多特性组合到了一起。总之，C++可以以简洁的方式使用，这对于代码的维护非常重要。应总是假定下一个开发人员要查看代码，理解其工作方式——这也会对长时间的运行有帮助。

书名
版权
前言
目录
正文