畅销全球的经典 C++ 教程

SAMS

C++ Primer Plus (第五版) 中文版

〔美〕Stephen Prata 著
孙建春 韦 强 译

人民邮电出版社
POSTS & TELECOM PRESS

封面设计：胡平利

C++ Primer Plus（第五版）中文版

本书享有“程序员和开发人员学习C++的完整教程”的美誉，它经过仔细的测试，制作精细，是计算机行业的经典著作。前4版在美国的总销售量超过了10万册。该书阐述了包括结构化代码和自顶向下设计在内的编程原理，涵盖了类、继承、模板、异常以及最新的面向对象编程技术等内容。

作者Stephen Prata是受人尊敬的作者和教育家，他以清晰的语言，对C++做了具有启发性的、见解深刻的介绍。本书在解释基本概念的同时，兼顾了C++语言的细节。书中通过大量简短、易于输入的范例，每次阐述一两个概念，激励读者通过立刻应用来掌握全新的主题；而每章最后的复习题和编程练习则强调了最重要的信息，并帮助读者消化最难以理解的概念。本书是一本友好而易于使用的自学指南，适合用做编程课程的教材，也可供熟悉其他语言的开发人员参考，以更深入地理解C++语言的基本知识。

本书采用了各种教学技巧，以提高读者的学习效率：

- 兼顾基本C语言知识和C++特性；
- 就何时以及为何使用特性提供了概念性指南；
- 强调实践，通过简短、易于输入的范例每次阐述一两个概念；
- 包含几百个范例程序；
- 通过示意图、注意、提示等来组织和区分相关的概念；
- 提供了复习题和编程练习，让读者能够测试自己对所学知识的理解程度；
- 以极大的灵活性实际使用通用C++——不要求读者使用特定的计算机、操作系统或编译器；
- 以ISO/ANSI标准为基础，对模板、标准模板库、字符串类、异常、RTTI和名称空间进行了讨论。

Stephen Prata在位于加利福尼亚州Kentfield的Marin学院教授天文、物理和计算机科学。他编写的《The Waite Group's New C Primer Plus》获得了美国计算机出版协会颁发的1990年度最佳“How-to”计算机图书奖，他编写的《The Waite Group's C++ Primer Plus》获得了美国计算机出版协会颁发的1991年度最佳“How-to”计算机图书奖的提名。

“想掌握C++就选这本书吧！绝对是进入C++殿堂的踏脚石，守护你度过最危险的沼泽！”

——本书第四版网上读者评价

SAMS

ISBN 7-115-13416-2

9 787115 134165 >

ISBN7-115-13416-2/TP·4665
定价：72.00元

人民邮电出版社网址 www.ptpress.com.cn

C++ Primer Plus（第五版）中文版

[美] Stephen Prata 著

孙建春 韦 强 译

人 民 邮 电 出 版 社

图书在版编目（CIP）数据

C++ Primer Plus：第5版/（美）普拉塔（Prata, S.）著；孙建春，韦强译.
—北京：人民邮电出版社，2005.5

ISBN 7-115-13416-2

Ⅰ. C… Ⅱ. ①普…②孙…③韦… Ⅲ. C语言—程序设计 Ⅳ. TP312

中国版本图书馆CIP数据核字（2005）第042689号

版权声明

Stephen Prata:C++ Primer Plus，Fifth Edition（ISBN:0672326973）
Copyright © 2005 by Sams Publishing.
Authorized translation from the English language edition published by Sams.
All rights reserved.

本书中文简体字版由美国Sams出版公司授权人民邮电出版社出版。未经出版者书面许可，对本书任何部分不得以任何方式复制或抄袭。

版权所有，侵权必究。

C++Primer Plus（第五版）中文版

◆ 著 [美] Stephen Prata
译 孙建春 韦 强
责任编辑 李 际

◆ 人民邮电出版社出版发行 北京市崇文区夕照寺街14号
邮编 100061 电子函件 315@ptpress.com.cn
网址 http://www.ptpress.com.cn

读者热线 010-67132705
北京顺义振华印刷厂印刷
新华书店总店北京发行所经销

◆ 开本：787×1092 1/16
印张：50.25
字数：1 639千字 2005年5月第1版
印数：1－5 000册 2005年5月北京第1次印刷

著作权合同登记号 图字：01-2004-4418号
ISBN 7-115-13416-2/TP・4665

定价：72.00元

本书如有印装质量问题，请与本社联系 电话：（010）67129223

内 容 提 要

C++是在 C 语言基础上开发的一种集面向对象编程、通用编程和传统的过程化编程于一体的编程语言，是 C 语言的超集。本书是根据 2003 年的 ISO/ANSI C++标准编写的，通过大量短小精悍的程序详细而全面地阐述了 C++的基本概念和技术。

全书分 17 章和 10 个附录，分别介绍了 C++程序的运行方式、基本数据类型、复合数据类型、循环和关系表达式、分支语句和逻辑操作符、函数重载和函数模板、内存模型和名称空间、类的设计和使用、多态、虚函数、动态内存分配、继承、代码重用、友元、异常处理技术、string 类和标准模板库、输入/输出等内容。

本书针对 C++初学者，从 C 语言基础知识开始介绍，然后在此基础上详细阐述 C++新增的特性，因此不要求读者有较多 C 语言方面的背景知识。本书可作为高等院校 C++课程的教材，也可供初学者自学 C++时使用。

作 者 简 介

Stephen Prata 在加州肯特菲尔得的马林学院教授天文、物理和计算机科学。他毕业于加州理工学院，在加州大学伯克利分校获得博士学位。Stephen 本人或与他人合作编写的图书有十多本。他撰写的 *The Waite Group's New C Primer Plus* 一书获得了计算机出版联合会 1990 年度最佳“How-to”计算机图书奖；他撰写的 *The Waite Group's C++ Primer Plus* 一书获得了计算机出版联合会 1991 年度最佳“How-to”计算机图书奖的提名。

前　言

学习C++是一次探索之旅，因为这种语言容纳了好几种编程模式，其中包括面向对象编程、通用编程和传统的过程化编程。随着新特性的不断添加，C++一度成为一个活动目标，不过现在有了2003年的ISO/ANSI C++标准第二版后，已经稳定下来了。现代编译器支持该标准要求的多数或全部特性，程序员要花时间来习惯这些特性的应用。本书第五版是按ISO/ANSI标准编写的，将介绍这种成熟的C++版本。

本书在介绍C++特性的同时，讨论了基本C语言，使二者成为有机的整体。书中介绍了C++的基本概念，并通过短小精悍的程序来阐明，这些程序都很容易复制和试验。还介绍了输入和输出、如何让程序执行重复性任务、如何让程序做出选择、处理数据的多种方式以及如何使用函数等内容。另外，还讲述了C++在C语言的基础上新增的许多特性，其中包括：

- 类和对象。
- 继承。
- 多态、虚函数和RTTI（运行阶段类型识别）。
- 函数重载。
- 引用变量。
- 通用（或独立于类型的）编程，这种技术是由模板和标准模板库（STL）提供的。
- 处理错误条件的异常机制。
- 管理函数、类和变量名的名称空间。

本书在传授知识方面有几个优点。大约20年前，《C Primer Plus》一书开创了优良的初级教程传统，本书建立在这样的基础之上，吸收了其中很多成功的理念：

- 初级教程应当是友好的、便于使用的指南。
- 初级教程不要求读者已经熟悉相关的编程概念。
- 初级教程强调的是“实践性”学习，通过简短、容易输入的范例同时阐述一个或多个概念。
- 初级教程用示意图来解释概念。
- 初级教程提供问题和练习来检验读者对知识的理解，从而适于自学或课堂教学。

基于上述理念，本书帮助读者理解这种用途广泛的语言，并学习如何使用它。例如：

- 对何时使用某些特性，例如何时使用公共继承来建立*is-a*关系，提供了概念方面的指导。
- 阐释了常用的C++编程理念和技术。
- 提供了大量的附注，如提示、警告、注意等。

本书的作者和编辑尽最大的努力使本书简单、明了、生动有趣。我们的目标是，读者阅读本书后，能够编写出可靠、高效的程序，并且觉得这是一种享受。

本书使用的范例代码

本书包含大量的范例代码，其中大部分是完整的程序。和前一版一样，本书介绍的是通用C++，因此适用于任何计算机、操作系统和编译器。书中的范例在Windows XP系统、Macintosh OS X系统和Linux系统上进行了测试。只有为数不多的几个程序会受编译器不兼容问题的影响。本书前一版面世后，编译器在遵循C++标准方面更严格。

对于本书中完整的程序，其源代码可从Sams网站（www.samspublishing.com）下载得到。为此，可输入本书的英文书名，然后单击Search，再单击该书名，切换到能够下载源代码的网页。在该网站，也可找到有些编程练习的解决方案。

本书的组织结构

本书分为 17 章和 10 个附录。

第 1 章：预备知识

本章介绍了 Bjarne Stroustrup 如何在 C 语言的基础上添加对面向对象编程的支持，来创造 C++编程语言的。讨论了过程化语言（如 C 语言）与面向对象语言（如 C++）之间的区别。读者将了解 ANSI/ISO 在制定 C++标准方面所做的工作。本章还讨论了创建 C++程序的技巧，介绍了当前几种 C++编译器使用的方法。最后，本章介绍了本书的一些约定。

第 2 章：开始学习 C++

本章介绍创建简单 C++程序的步骤。读者可以学习到 main( )函数扮演的角色以及 C++程序使用的一些语句。读者将使用预定义的 cout 和 cin 对象来实现程序输出和输入，学习如何创建和使用变量。最后，本章还将介绍函数——C++的编程模块。

第 3 章：处理数据

C++提供了内置类型来存储两种数据：整数（没有小数的数字）和浮点数（带小数的数字）。为满足程序员的各种需求，C++为每一种数据都提供了几种类型。本章将要讨论这些类型，包括创建变量和编写各种类型的常量。另外，还将讨论 C++是如何处理不同类型之间的隐式和显式转换的。

第 4 章：复合类型

C++允许程序员使用基本的内置类型来创建更复杂的类型。最高级的形式是类，这将在第 9～13 章讨论。本章讨论其他形式，包括数组（存储多个同类型的值）、结构（存储多个不同类型的值）、指针（标识内存位置）。读者还将学习如何创建和存储文本字符串及如何使用 C-风格字符数组和 C++ string 类来处理文本输入和输出。最后，还将学习 C++处理内存分配的一些方法，其中包括用于显式地管理内存的 new 和 delete 操作符。

第 5 章：循环和关系表达式

程序经常需要执行重复性操作，为此 C++提供了 3 种循环结构：for 循环、while 循环和 do while 循环。这些循环必须知道何时终止，C++的关系操作符使程序员能够创建测试来引导循环。本章还将介绍如何创建逐字符地读取和处理输入的循环。最后，读者将学习如何创建二维数组以及如何使用嵌套循环来处理它们。

第 6 章：分支语句和逻辑操作符

如果程序可以根据实际情况调整执行，我们就说程序能够智能地行动。在本章，读者将了解到如何使用 if、if else 和 switch 语句及条件操作符来控制程序流程，学习如何使用逻辑操作符来表达决策测试。另外，本章还将介绍确定字符关系（如测试字符是数字还是非打印字符）的函数库 cctype。最后，还将简要地介绍文件输入/输出。

第 7 章：函数——C++的编程模块

函数是 C++的基本编程部件。本章重点介绍 C++函数与 C 函数共同的特性。具体地说，读者将复习函数定义的通用格式，了解函数原型是如何提高程序可靠性的。同时，还将学习如何编写函数来处理数组、字符串和结构。还要学习有关递归的知识（即函数在什么情况下调用自身）以及如何用它来实现分而治之的策略。最后将介绍函数指针，它使程序员能够通过函数参数来命令函数使用另一个函数。

第 8 章：函数探幽

本章将探索 C++中函数新增的特性。读者将学习内联函数，它可以提高程序的执行速度，但会增加程序的长度；还将使用引用变量，它们提供了另一种将信息传递给函数的方式。默认参数使函数能够自动为函数调用中省略的函数参数提供值。函数重载使程序员能够创建多个参数列表不同的同名函数。类设计中经常使用这些特性。另外，读者还将学习函数模板，它们使程序员能够指定相关函数族的设计。

第 9 章：内存模型和名称空间

本章讨论如何创建多文件程序，介绍分配内存的各种方式、管理内存的各种方式以及作用域、链接、名称空间，这些内容决定了变量在程序的哪些部分是可见的。

第 10 章：对象和类

类是用户定义的类型，对象（如变量）是类的实例。本章介绍面向对象编程和类设计。对象声明描述的是存储在对象中的信息以及可对对象执行的操作（类方法）。对象的某些组成部分对于外界来说是可见的（公有部分），而某些部分却是隐藏的（私有部分）。特殊的类方法（构造函数和析构函数）在对象创建和释放时发挥作用。在本章中，读者将学习所有这些内容以及其他类知识，了解如何使用类来实现 ADT，如堆栈。

第 11 章：使用类

在本章中，读者将深入了解类。首先了解操作符重载，它使程序员能够定义与类对象一起使用的操作符，如+。还将学习友元函数，这些函数可以访问外部世界不可访问的类数据。同时还将了解一些构造函数和重载操作符成员函数是如何被用来管理类的类型转换的。

第 12 章：类和动态内存分配

一般来说，让类成员指向动态分配的内存很有用。如果程序员在类构造函数中使用 new 来分配动态内存，就有责任提供适当的析构函数，定义显式复制构造函数和显式赋值操作符。本章介绍了在程序员没有提供显式定义时，将如何隐式地生成成员函数以及这些成员函数的行为。读者还将通过使用对象指针，了解队列模拟问题，扩充类方面的知识。

第 13 章：类继承

在面向对象编程中，继承是功能最强大的特性之一，通过继承，派生类可以继承基类的特性，可重用基类代码。本章讨论公有继承，这种继承模拟了 *is-a* 关系，即派生对象是基对象的特例。例如，物理学家是科学家的特例。有些继承关系是多态的，这意味着相同的方法名称可能导致依赖于对象类型的行为。要实现这种行为，需要使用一种新的成员函数——虚函数。有时，使用抽象基类是实现继承关系的最佳方式。本章讨论了这些问题，说明了公有继承在什么情况下合适，在什么情况下不合适。

第 14 章：C++中的代码重用

公有继承只是代码重用的方式之一。本章将介绍其他几种方式。如果一个类包含了另一个类的对象，则称为包含。包含可以用来模拟 *has-a* 关系，其中一个类包含另一个类的对象。例如，汽车有马达。也可以使用私有继承和保护继承来模拟这种关系。本章说明了各种方法之间的区别。同时，读者还将学习类模板，它使程序员能够使用通用类型来定义类，然后使用模板来根据具体类型创建特定的类。例如，堆栈模板使程序员能够创建整数堆栈或字符串堆栈。最后，本章还将介绍多重公有继承，使用这种方式，一个类可以从多个类派生而来。

第 15 章：友元、异常和其他

本章扩展了对友元的讨论，讨论了友元类和友元成员函数。然后从异常开始介绍了 C++的几项新特性。异常为处理程序异常提供了一种机制，如函数参数值不正确或内存耗尽等。读者还将学习 RTTI，这种机制用来确定对象类型。最后，本章还将介绍一种更安全的方法来替代不受限制的强制类型转换。

第 16 章：string 类和标准模板库

本章将讨论 C++语言中新增的一些类库。对于传统的 C-风格字符串来说，string 类是一种方便且功能强大的替代方式。auto_ptr 类帮助管理动态分配的内存。STL 提供了几种类容器（包括数组、队列、链表、集合和映射）的模板表示。它还提供了高效的通用算法库，这些算法可用于 STL 容器，也可用于常规数组。模板类 valarray 为数值数组提供了支持。

第 17 章：输入、输出和文件

本章复习 C++ I/O，并讨论如何格式化输出。读者将要学习如何使用类方法来确定输入或输出流的状态，了解输入类型是否不匹配或是否检测到了文件尾。C++使用继承来派生用于管理文件输入和输出的类。读者将学习如何打开文件，以进行输入和输出，如何在文件中追加数据，如何使用二进制文件，如何获得对文件的随机访问权。最后，还将学习如何使用标准的 I/O 方法来读取和写入字符串。

附录 A：计数系统

本附录讨论八进制数、十六进制数和二进制数。

附录 B：C++保留字

本附录列出了 C++关键字。

附录 C：ASCII 字符集

本附录列出了 ASCII 字符集及其十进制、八进制、十六进制和二进制表示。

附录 D：操作符优先级

本附录按优先级从高到低的顺序列出了 C++的操作符。

附录 E：其他操作符

本附录总结了正文中没有介绍的其他 C++操作符，如按位操作符等。

附录 F：string 模板类

本附录总结了 string 类方法和函数。

附录 G：STL 方法和函数

本附录总结了 STL 容器方法和通用的 STL 算法函数。

附录 H：精选读物和网资源

本附录列出一些参考书，帮助读者深入了解 C++。

附录 I：转换为 ANSI/ISO 标准 C++

本附录提供了从 C 和老式 C++实现到 ANSI/ISO C++的转换指南。

附录 J：复习题答案

本附录提供每章结尾的复习题的答案。

对教师的提示

本书第五版的宗旨之一是：提供一本既可以用作自学书也可以用作教材的书籍。下面是本书在支持教学方面的一些特征：

- 本书介绍的是通用 C++，不依赖于特定的实现。
- 本书内容记录了 ISO/ANSI C++标准委员会的工作，并讨论了模板、STL、string 类、异常、RTTI 和名称空间。
- 本书不要求学生了解 C 语言，但如果有一定的编程经验则更好。
- 本书内容经过了精心安排，前几章可以作为对 C 预备知识的复习章节一带而过。
- 各章都有复习题和编程练习。附录 J 提供了复习题的答案；有些编程练习的解决方案可在 Sams 网站（www.samspublishing.com）上找到。
- 本书介绍的一些主题很适于计算机科学课程，包括抽象数据类型（ADT）、堆栈、队列、简单链表、模拟、通用编程以及使用递归来实现分而治之的策略。
- 各章都非常简短，用一周甚至更短的时间就可以学完。
- 本书讨论了何时使用具体的特性以及如何使用它们。例如，把 *is-a* 关系的公有继承同组合、*has-a* 关系的私有继承联系起来，讨论了何时应使用虚函数以及何时不应使用。

本书约定

为区别不同类型的文本，我们使用了一些印刷上的约定。

- 代码行、命令、语句、变量、文件名和程序输出使用 courier new 字体：

```
#include <iostream>
int main()
{
   using namespace std;
   cout <<
   return 0;
}
```

- 用户需要输入的程序输入用粗体表示：

```
Please enter your name:
```

```
Plato
```

- 句法描述中的占位符用斜体表示。读者应使用实际的文件名、参数等替换占位符。

开发书中编程范例时使用的系统

本书的范例都是使用 Microsoft Visual C++ 7.1（Microsoft Visual Studio .NET 2003 中的版本）和 Metrowerks CodeWarrior Development Studio 9 在一台带硬盘的运行 Windows XP Professional 的 Pentium PC 上开发的。对于其中的多数程序，都在上述系统上使用 Borland C++ 5.5 命令行编译器和 GNU gpp 3.3.3，在运行 SuSE 9.0 Linux 的 IBM 兼容 Pentium 机上使用 Comeau 4.3.3 和 GNU g++3.3.1，在运行 OS 10.3 的 Macintosh G4 上使用 Metrowerks Development Studio 9 进行了测试。本书指出了由于滞后于标准而导致的差异，如“老式实现使用 ios::fixed，而不是 ios_base::fixed”。本书指出了老式编译器中的错误和令人厌烦或容易混淆的特性，不过，它们都将在最新版本中得到了解决。

C++为程序员提供了丰富多彩的内容。祝您学习愉快！

目　录

第 1 章　预备知识 ……1

1.1　C++简介 ……1
1.2　C++简史 ……2
1.2.1　C 语言 ……2
1.2.2　C 语言编程原理 ……2
1.2.3　面向对象编程 ……3
1.2.4　C++和通用编程 ……4
1.2.5　C++的起源 ……4
1.3　可移植性和标准 ……5
1.4　程序创建的技巧 ……6
1.4.1　创建源代码文件 ……7
1.4.2　编译和链接 ……8
1.5　总结 ……11

第 2 章　开始学习 C++ ……12

2.1　进入 C++ ……12
2.1.1　main() 函数 ……14
2.1.2　C++注释 ……15
2.1.3　C++预处理器和 iostream 文件 ……16
2.1.4　头文件名 ……16
2.1.5　名称空间 ……17
2.1.6　使用 cout 进行 C++输出 ……18
2.1.7　C++源代码的格式化 ……19
2.2　C++语句 ……21
2.2.1　声明语句和变量 ……21
2.2.2　赋值语句 ……22
2.2.3　cout 的新花样 ……23
2.3　其他 C++语句 ……23
2.3.1　使用 cin ……24
2.3.2　使用 cout 进行拼接 ……24
2.3.3　类简介 ……25
2.4　函数 ……26
2.4.1　使用有返回值的函数 ……26
2.4.2　函数变体 ……28
2.4.3　用户定义的函数 ……29
2.4.4　用户定义的有返回值的函数 ……31
2.4.5　在多函数程序中使用 using 编译指令 ……32
2.5　总结 ……33
2.6　复习题 ……34
2.7　编程练习 ……34

第 3 章　处理数据 ……36

3.1　简单变量 ……36
3.1.1　变量名 ……37
3.1.2　整型 ……38
3.1.3　short、int 和 long ……38
3.1.4　无符号类型 ……41
3.1.5　选择整型类型 ……43
3.1.6　整型常量 ……44
3.1.7　C++如何确定常量的类型 ……45
3.1.8　char 类型：字符和小整数 ……45
3.1.9　新的 bool 类型 ……50
3.2　const 限定符 ……51
3.3　浮点数 ……51
3.3.1　书写浮点数 ……52
3.3.2　浮点类型 ……53
3.3.3　浮点常量 ……54
3.3.4　浮点数的优缺点 ……55
3.4　C++算术操作符 ……55
3.4.1　操作符优先级和结合性 ……56
3.4.2　除法分支 ……57
3.4.3　求模操作符 ……58
3.4.4　类型转换 ……59
3.5　总结 ……62
3.6　复习题 ……63
3.7　编程练习 ……63

第 4 章　复合类型 ……65

4.1　数组 ……65
4.2　字符串 ……68
4.2.1　拼接字符串常量 ……70
4.2.2　在数组中使用字符串 ……70
4.2.3　字符串输入 ……71
4.2.4　每次读取一行字符串输入 ……72
4.2.5　混合输入字符串和数字 ……75
4.3　string 类简介 ……76
4.3.1　赋值、拼接和附加 ……77
4.3.2　string 类的其他操作 ……78
4.3.3　string 类 I/O ……79
4.4　结构简介 ……81
4.4.1　在程序中使用结构 ……82
4.4.2　结构可以将 string 类作为成员吗 ……84
4.4.3　其他结构属性 ……84

4.4.4 结构数组……85
4.4.5 结构中的位字段……86
4.5 共用体……86
4.6 枚举……88
4.6.1 设置枚举量的值……89
4.6.2 枚举的取值范围……89
4.7 指针和自由存储空间……89
4.7.1 声明和初始化指针……92
4.7.2 指针的危险……93
4.7.3 指针和数字……93
4.7.4 使用 new 来分配内存……94
4.7.5 使用 delete 来释放内存……95
4.7.6 使用 new 来创建动态数组……96
4.8 指针、数组和指针算术……98
4.8.1 程序说明……99
4.8.2 指针和字符串……101
4.8.3 使用 new 创建动态结构……104
4.8.4 自动存储、静态存储和动态存储……107
4.9 总结……108
4.10 复习题……109
4.11 编程练习……109

第 5 章 循环和关系表达式……111

5.1 for 循环……111
5.1.1 for 循环的组成部分……112
5.1.2 回到 for 循环……116
5.1.3 修改步长……118
5.1.4 使用 for 循环访问字符串……118
5.1.5 递增操作符（++）和递减操作符（--）……119
5.1.6 副作用和顺序点……119
5.1.7 前缀格式和后缀格式……120
5.1.8 递增/递减操作符和指针……120
5.1.9 组合赋值操作符……121
5.1.10 复合语句（语句块）……122
5.1.11 逗号操作符（或其他句法技巧）……123
5.2 关系表达式……125
5.2.1 可能犯的错误……126
5.2.2 C-风格字符串的比较……127
5.2.3 比较 string 类字符串……129
5.3 while 循环……130
5.3.1 for 与 while……131
5.3.2 等待一段时间：编写延时循环……132
5.4 do while 循环……134
5.5 循环和文本输入……136
5.5.1 使用原始的 cin 进行输入……136
5.5.2 使用 cin.get(char) 进行补救……137
5.5.3 使用哪一个 cin.get()……138
5.5.4 文件尾条件……138
5.5.5 另一个 cin.get() 版本……140
5.6 嵌套循环和二维数组……142
5.7 总结……145
5.8 复习题……145
5.9 编程练习……146

第 6 章 分支语句和逻辑操作符……148

6.1 if 语句……148
6.1.1 if else 语句……149
6.1.2 格式化 if else 语句……151
6.1.3 if else if else 结构……152
6.2 逻辑表达式……153
6.2.1 逻辑 OR 操作符：||……153
6.2.2 逻辑 AND 操作符：&&……154
6.2.3 逻辑 NOT 操作符：!……157
6.2.4 逻辑操作符细节……158
6.2.5 其他表示方式……159
6.3 字符函数库 cctype……159
6.4 ?:操作符……161
6.5 switch 语句……162
6.5.1 将枚举量用作标签……165
6.5.2 switch 和 if else……166
6.6 break 和 continue 语句……166
6.7 读取数字的循环……168
6.8 简单文件输入/输出……171
6.8.1 文本 I/O 和文本文件……171
6.8.2 写入到文本文件中……172
6.8.3 读取文本文件……174
6.9 总结……178
6.10 复习题……178
6.11 编程练习……180

第 7 章 函数——C++的编程模块……182

7.1 复习函数的基本知识……182
7.1.1 定义函数……183
7.1.2 函数原型和函数调用……184
7.2 函数参数和按值传递……187
7.2.1 多个参数……188
7.2.2 另外一个接受两个参数的函数……189
7.3 函数和数组……191
7.3.1 函数如何使用指针来处理数组……192
7.3.2 将数组作为参数意味着什么……192
7.3.3 更多数组函数范例……194
7.3.4 使用数组区间的函数……198
7.3.5 指针和 const……200

7.4 函数和一维数组 ……202
7.5 函数和C-风格字符串 ……203
7.5.1 将C-风格字符串作为参数的函数 ……203
7.5.2 返回C-风格字符串的函数 ……204
7.6 函数和结构 ……206
7.6.1 传递和返回结构 ……206
7.6.2 另一个处理结构的函数范例 ……207
7.6.3 传递结构的地址 ……211
7.7 函数和string对象 ……212
7.8 递归 ……214
7.8.1 包含一个递归调用的递归 ……214
7.8.2 包含多个递归调用的递归 ……215
7.9 函数指针 ……216
7.9.1 函数指针的基础知识 ……216
7.9.2 函数指针示例 ……218
7.10 总结 ……219
7.11 复习题 ……220
7.12 编程练习 ……220

第8章 函数探幽 ……223

8.1 C++内联函数 ……223
8.2 引用变量 ……225
8.2.1 创建引用变量 ……225
8.2.2 将引用用作函数参数 ……227
8.2.3 引用的属性和特别之处 ……230
8.2.4 将引用用于结构 ……232
8.2.5 将引用用于类对象 ……235
8.2.6 对象、继承和引用 ……238
8.2.7 何时使用引用参数 ……240
8.3 默认参数 ……240
8.4 函数重载 ……242
8.4.1 重载范例 ……243
8.4.2 何时使用函数重载 ……245
8.5 函数模板 ……246
8.5.1 重载的模板 ……248
8.5.2 显式具体化 ……250
8.5.3 实例化和具体化 ……253
8.5.4 编译器选择使用哪个函数版本 ……254
8.6 总结 ……258
8.7 复习题 ……258
8.8 编程练习 ……259

第9章 内存模型和名称空间 ……261

9.1 单独编译 ……261
9.2 存储持续性、作用域和链接性 ……265
9.2.1 作用域和链接 ……265
9.2.2 自动存储持续性 ……266
9.2.3 静态持续变量 ……270
9.2.4 说明符和限定符 ……277
9.2.5 函数和链接性 ……278
9.2.6 语言链接性 ……279
9.2.7 存储方案和动态分配 ……279
9.3 布局new操作符 ……280
9.4 名称空间 ……282
9.4.1 传统的C++名称空间 ……282
9.4.2 新的名称空间特性 ……284
9.4.3 名称空间范例 ……289
9.4.4 名称空间及其前途 ……292
9.5 总结 ……292
9.6 复习题 ……293
9.7 编程练习 ……295

第10章 对象和类 ……297

10.1 过程性编程和面向对象编程 ……297
10.2 抽象和类 ……298
10.2.1 类型是什么 ……298
10.2.2 C++中的类 ……299
10.2.3 实现类成员函数 ……302
10.2.4 使用类 ……306
10.2.5 小结 ……308
10.3 类的构造函数和析构函数 ……309
10.3.1 声明和定义构造函数 ……309
10.3.2 使用构造函数 ……310
10.3.3 默认构造函数 ……311
10.3.4 析构函数 ……311
10.3.5 改进Stock类 ……312
10.3.6 构造函数和析构函数小结 ……317
10.4 this指针 ……318
10.5 对象数组 ……322
10.6 接口和实现小结 ……325
10.7 类作用域 ……326
10.8 抽象数据类型 ……327
10.9 总结 ……331
10.10 复习题 ……332
10.11 编程练习 ……332

第11章 使用类 ……335

11.1 操作符重载 ……335
11.2 计算时间：一个操作符重载范例 ……336
11.2.1 添加加法操作符 ……339
11.2.2 重载限制 ……341
11.2.3 其他重载操作符 ……342
11.3 友元简介 ……345
11.3.1 创建友元 ……346
11.3.2 常用的友元：重载<<操作符 ……347

11.4 重载操作符：作为成员函数还是非成员函数……351
11.5 再谈重载：矢量类……352
11.5.1 使用状态成员……358
11.5.2 为 Vector 类重载算术操作符…359
11.5.3 对实现的说明……361
11.5.4 使用 Vector 类来模拟随机行走……361
11.6 类的自动转换和强制类型转换……363
11.6.1 程序说明……367
11.6.2 转换函数……367
11.6.3 转换函数和友元函数……371
11.7 总结……373
11.8 复习题……374
11.9 编程练习……375

第 12 章 类和动态内存分配……377

12.1 动态内存和类……377
12.1.1 复习范例和静态类成员……378
12.1.2 隐式成员函数……383
12.1.3 改进后的新 String 类……389
12.1.4 在构造函数中使用 new 时应注意的事项……397
12.1.5 有关返回对象的说明……399
12.1.6 使用指向对象的指针……401
12.1.7 复习各种技术……408
12.2 队列模拟……409
12.2.1 队列类……410
12.2.2 Customer 类……417
12.2.3 模拟……420
12.3 总结……423
12.4 复习题……424
12.5 编程练习……425

第 13 章 类继承……428

13.1 一个简单的基类……429
13.1.1 派生一个类……430
13.1.2 构造函数：访问权限的考虑…431
13.1.3 使用派生类……433
13.2 派生类和基类之间的特殊关系……435
13.3 继承——*is-a* 关系……436
13.4 多态公有继承……438
13.4.1 开发 Brass 类和 BrassPlus 类…438
13.4.2 静态联编和动态联编……447
13.4.3 指针和引用类型兼容性……448
13.4.4 虚拟成员函数和动态联编……449
13.4.5 有关虚函数注意事项……450
13.5 访问控制：protected……452
13.6 抽象基类……454
13.6.1 应用 ABC 概念……456
13.6.2 ABC 理念……459
13.7 继承和动态内存分配……459
13.7.1 第一种情况：派生类不使用 new……460
13.7.2 第二种情况：派生类使用 new……460
13.7.3 使用动态内存分配和友元的继承范例……462
13.8 类设计回顾……466
13.8.1 编译器生成的成员函数……466
13.8.2 其他的类方法……467
13.8.3 公有继承的考虑因素……469
13.8.4 类函数小结……472
13.9 总结……472
13.10 复习题……473
13.11 编程练习……474

第 14 章 C++中的代码重用……477

14.1 包含对象成员的类……477
14.1.1 valarray 类简介……478
14.1.2 Student 类的设计……478
14.1.3 Student 类范例……479
14.2 私有继承……485
14.3 多重继承……492
14.3.1 有多少 Worker……496
14.3.2 哪个方法……499
14.3.3 MI 小结……507
14.4 类模板……508
14.4.1 定义类模板……508
14.4.2 使用模板类……510
14.4.3 深入探讨模板类……512
14.4.4 数组模板范例和非类型参数…517
14.4.5 模板多功能性……518
14.4.6 模板的具体化……521
14.4.7 成员模板……523
14.4.8 将模板用作参数……525
14.4.9 模板类和友元……526
14.5 总结……532
14.6 复习题……533
14.7 编程练习……534

第 15 章 友元、异常和其他……539

15.1 友元……539
15.1.1 友元类……539
15.1.2 友元成员函数……543
15.1.3 其他友好关系……546

15.2 嵌套类……547
15.2.1 嵌套类和访问权限……548
15.2.2 模板中的嵌套……549
15.3 异常……552
15.3.1 调用 abort()……552
15.3.2 返回错误码……553
15.3.3 异常机制……554
15.3.4 将对象用作异常类型……557
15.3.5 堆栈解退……560
15.3.6 其他异常特性……564
15.3.7 exception 类……566
15.3.8 异常、类和继承……569
15.3.9 异常何时会迷失方向……573
15.3.10 有关异常的注意事项……575
15.4 RTTI……576
15.4.1 RTTI 的用途……576
15.4.2 RTTI 的工作原理……576
15.5 类型转换操作符……583
15.6 总结……585
15.7 复习题……586
15.8 编程练习……587

第 16 章 string 类和标准模板库……588

16.1 string 类……588
16.1.1 构造字符串……588
16.1.2 string 类输入……591
16.1.3 使用字符串……593
16.1.4 string 还提供了哪些功能……597
16.2 auto_ptr 类……599
16.2.1 使用 auto_ptr……599
16.2.2 有关 auto_ptr 的注意事项……601
16.3 STL……602
16.3.1 vector 模板类……602
16.3.2 可对矢量执行的操作……604
16.3.3 对矢量可执行的其他操作……608
16.4 通用编程技术……611
16.4.1 为何使用迭代器……611
16.4.2 迭代器类型……614
16.4.3 迭代器层次结构……615
16.4.4 概念、改进和模型……616
16.4.5 容器种类……620
16.4.6 联合容器……627
16.5 函数对象……631
16.5.1 函数符概念……632
16.5.2 预定义的函数符……634
16.5.3 自适应函数符和函数适配器……635
16.6 算法……637
16.6.1 算法组……637
16.6.2 算法的通用特征……638
16.6.3 STL 和 string 类……639
16.6.4 函数和容器方法……640
16.6.5 使用 STL……641
16.7 其他库……644
16.8 总结……648
16.9 复习题……649
16.10 编程练习……650

第 17 章 输入、输出和文件……651

17.1 C++输入和输出概述……651
17.1.1 流和缓冲区……652
17.1.2 流、缓冲区和 iostream 文件……653
17.1.3 重定向……655
17.2 使用 cout 进行输出……656
17.2.1 重载的<<操作符……656
17.2.2 其他 ostream 方法……658
17.2.3 刷新输出缓冲区……660
17.2.4 用 cout 进行格式化……660
17.3 使用 cin 进行输入……672
17.3.1 cin>>如何检查输入……673
17.3.2 流状态……675
17.3.3 其他 istream 类方法……678
17.3.4 其他 istream 方法……683
17.4 文件输入和输出……686
17.4.1 简单的文件 I/O……686
17.4.2 流状态检查和 is_open()……688
17.4.3 打开多个文件……689
17.4.4 命令行处理技术……689
17.4.5 文件模式……691
17.4.6 随机存取……698
17.5 内核格式化……704
17.6 接下来的任务……706
17.7 总结……707
17.8 复习题……707
17.9 编程练习……708

附录 A 计数系统……712

A.1 十进制数……712
A.2 八进制整数……712
A.3 十六进制数……712
A.4 二进制数……713
A.5 二进制和十六进制……714

附录 B C++保留字……715

B.1 C++关键字……715
B.2 替代标记……715

B.3 C++库保留名称……716

附录 C ASCII 字符集……717

附录 D 操作符优先级……721

附录 E 其他操作符……724

E.1 按位操作符……724
E.1.1 移位操作符……724
E.1.2 逻辑按位操作符……725
E.1.3 按位操作符的另一种表示……727
E.1.4 几种常用的按位操作符技术……727
E.2 成员解除引用操作符……728

附录 F string 模板类……732

F.1 13 种类型和一个常量……732
F.2 数据信息、构造函数及其他……733
F.2.1 默认构造函数……734
F.2.2 使用数组的构造函数……735
F.2.3 使用部分数组的构造函数……735
F.2.4 复制构造函数……735
F.2.5 使用一个字符的 *n* 个副本的构造函数……736
F.2.6 使用区间的构造函数……736
F.2.7 内存杂记……736
F.3 字符串存取……737
F.4 基本赋值……737
F.5 字符串搜索……738
F.5.1 find() 系列……738
F.5.2 rfind() 系列……738
F.5.3 find_first_of() 系列……738
F.5.4 find_last_of() 系列……739
F.5.5 find_first_not_of() 系列……739
F.5.6 find_last_not_of() 系列……739
F.6 比较方法和函数……740
F.7 字符串修改方法……741
F.7.1 用于追加和相加的方法……741
F.7.2 其他赋值方法……741
F.7.3 插入方法……742
F.7.4 清除方法……742
F.7.5 替换方法……742
F.7.6 其他修改方法：copy() 和 swap()……743
F.8 输出和输入……743

附录 G STL 方法和函数……744

G.1 所有容器都有的成员……744
G.2 矢量、链表和双端队列的其他成员……746
G.3 set 和 map 的其他成员……747
G.4 STL 函数……748
G.4.1 非修改式序列操作……749
G.4.2 修改式序列操作……751
G.4.3 排序和相关操作……756
G.4.4 数字操作……763

附录 H 精选读物和网上资源……765

H.1 精选读物……765
H.2 网上资源……766

附录 I 转换为 ANSI/ISO 标准 C++……767

I.1 使用一些预处理器编译指令的替代品……767
I.1.1 使用 const 而不是#define 来定义常量……767
I.1.2 使用 inline 而不是# define 来定义小型函数……768
I.2 使用函数原型……769
I.3 使用类型转换……769
I.4 逐渐熟悉 C++特性……769
I.5 使用新的头文件……769
I.6 使用名称空间……770
I.7 使用 autoptr 模板……771
I.8 使用 string 类……771
I.9 使用 STL……771

附录 J 复习题答案……772

第 2 章复习题答案……772
第 3 章复习题答案……772
第 4 章复习题答案……773
第 5 章复习题答案……774
第 6 章复习题答案……774
第 7 章复习题答案……775
第 8 章复习题答案……777
第 9 章复习题答案……778
第 10 章复习题答案……779
第 11 章复习题答案……780
第 12 章复习题答案……781
第 13 章复习题答案……783
第 14 章复习题答案……784
第 15 章复习题答案……785
第 16 章复习题答案……786
第 17 章复习题答案……786

第 1 章　预备知识

本章内容包括：

- C 语言和 C++的发展历史和基本原理。
- 过程性编程技术和面向对象编程技术。
- C++是如何在 C 语言的基础上添加面向对象概念的。
- C++是如何在 C 语言的基础上添加通用编程概念的。
- 编程语言标准。
- 创建程序的技巧。

欢迎进入 C++世界！这是一种令人兴奋的语言，它在 C 语言的基础上添加了对面向对象编程的支持，在 20 世纪 90 年代便是最重要的编程语言之一，其应用在 21 世纪仍保持强劲势头。C++继承了 C 语言高效、简洁、快速和可移植性的传统。C++面向对象的特性带来了全新的编程方法，这种方法是为应付复杂程度不断提高的现代编程任务而设计的。C++的模板特性提供了另一种全新的编程方法——通用编程。这 3 件法宝既是福也是祸，一方面让 C++语言功能强大，一方面则意味着有更多的东西需要学习。

本章先介绍 C++的背景，然后进一步介绍创建 C++程序的一些基本原则。本书其他章节将讲述如何使用 C++语言，从语言最浅显的基本知识开始，到面向对象的编程（OOP）及其支持的新术语——对象、类、封装、数据隐藏、多态和继承等，然后介绍它对通用编程的支持（当然，随着您对 C++的学习，这些词汇将从花里胡哨的词语变为论述中必不可少的术语）。

1.1　C++简介

C++融合了 3 种不同的编程传统——C 语言代表的过程性语言传统、C++在 C 语言基础上添加的类代表的面向对象语言的传统以及 C++模板支持的通用编程传统。本章将简要介绍这些传统。不过首先，我们来看看这种传统对于学习 C++来说意味着什么。使用 C++的原因之一是为了利用其面向对象的特性。要利用这种特性，必须对标准 C 语言知识有较深入的了解，因为它提供了基本类型、操作符、控制结构和句法规则。所以，如果已经对 C 有所了解，便可以学习 C++了，但这并不仅仅是学习更多的关键字和结构，从 C 过渡到 C++的学习量就像从头学习 C 语言一样大。另外，如果先掌握了 C 语言，则在过渡到 C++时，必须摈弃一些编程习惯。如果不了解 C 语言，则学习 C++时需要掌握 C 语言的知识、OOP 知识以及通用编程知识，但无须摈弃任何编程习惯。如果您认为学习 C++可能需要扩展思维，这就对了。本书将以清晰的、帮助的方式，引导读者一步一个脚印地学习，因此扩展思维的过程是温和的，不至于让您的大脑接受不了。

本书通过传授 C 语言基础知识和 C++新增的内容，带您步入 C++的世界，因此不要求读者具备 C 语言知识。首先学习 C++与 C 语言共有的一些特性。即使已经了解 C 语言，也会发现阅读本书的这一部分是一次很好的复习。另外，本章还介绍了一些对后面的学习十分重要的概念，指出了 C++和 C 之间的区别。在牢固地掌握了 C 语言的基础知识后，就可以在此基础上学习 C++方面的知识了。那时将学习对象和类以及 C++是如何实现它们的，另外还将学习模板。

本书不是完整的C++参考手册，不会探索该语言的每个细节，但将介绍所有的重要特性，包括一些新加入的特性，如模板、异常和名称空间等。

下面简要地介绍一下C++的背景知识。

1.2 C++简史

在过去的几十年，计算机技术以令人惊讶的速度发展着，当前，笔记本电脑的计算速度和存储信息的能力超过了上世纪60年代的大型机。很多程序员可能还记得，将数叠穿孔卡片提交给充斥整个房间的大型计算机系统的时代，而这种系统只有100KB的内存，连运行当今优秀的PC游戏都不够。计算机语言也得到了发展，尽管变化可能不是天翻地覆的，但也是非常重要的。体积更大、功能更强的计算机引出了更大、更复杂的程序，而这些程序在程序管理和维护方面带来了新的问题。

在20世纪70年代，C和Pascal这样的语言引领人们进入了结构化编程时代，这种机制把秩序和规程带进了迫切需要这种性质的领域中。除了提供结构化编程工具外，C还能生成简洁、快速运行的程序，并提供了处理硬件问题的能力，如管理通信端口和磁盘驱动器。这些因素使C语言成为20世纪80年代占统治地位的编程语言。同时，20世纪80年代，人们也见证了一种新的编程模式，面向对象编程（OOP）的成长。SmallTalk和C++语言具备这种功能。下面将深入地介绍C和OOP。

1.2.1 C语言

20世纪70年代早期，贝尔实验室的Dennis Ritchie致力于开发UNIX操作系统（操作系统是能够管理计算机资源、处理计算机与用户之间交互的一组程序。例如，操作系统将系统提示符显示在屏幕上，并运行程序）。为完成这项工作，Ritchie需要一种语言，它必须简洁，能够生成简洁、快速的程序，并能有效地控制硬件。传统上，程序员使用汇编语言来满足这些需求，汇编语言依赖于计算机的内部机器语言。不过，汇编语言是低级（low-level）语言，也就是说，它针对于特定的计算机处理器。因此，要将汇编程序移植到另一种计算机上，必须使用不同的汇编语言重新编写程序。这有点像每次购买新车时，都发现设计人员改变了控制系统的位置和功能，客户不得不重新学习驾驶这辆新车。不过，UNIX是为在不同的计算机（或平台）上工作而设计的，这意味着它是一种高级语言。高级（high-level）语言致力于解决问题，而不针对特定的硬件。一种被称为编译器的特殊程序将高级语言翻译成特定计算机的内部语言。这样，就可以通过对每个平台使用不同的编译器来在不同的平台上使用同一个高级语言程序了。Ritchie希望有一种语言能将低级语言的效率、硬件访问能力和高级语言的通用性、可移植性融合在一起，于是他在旧语言的基础上开发了C语言。

1.2.2 C语言编程原理

由于C++在C语言的基础上移植了新的编程理念，因此我们首先来看一看C所遵循的旧的理念。一般来说，计算机语言要处理两个概念——数据和算法。数据是程序使用和处理的信息，而算法是程序使用的方法（参见图1.1）。C语言与当前最主流的语言一样，在最初面世时也是过程性（procedural）语言，这意味着它强调的是编程的算法方面。从概念上说，过程化编程首先要确定计算机应采取的操作，然后使用编程语言来实现这些操作。程序命令计算机按一系列流程生成特定的结果，就像菜谱指定了厨师做蛋糕时应遵循的一系列步骤一样。

随着程序规模的扩大，早期的程序语言（如FORTRAN和BASIC）都会遇到组织方面的问题。例如，程序经常使用分支语句，根据某种测试的结果，执行一组或另一组指令。很多旧式程序的执行路径很混乱（被称为“意大利面条式编程”），几乎不可能通过阅读程序来理解它，修改这种程序简直是一场灾难。为了解决这种问题，计算机科学家开发了一种更有序的编程方法——结构化编程（structured programming）。C语言具有使用这种方法的特性。例如，结构化编程将分支（决定接下来应执行哪个指令）限制为一小组行为良好的结构。C语言的词汇表中就包含了这些结构（for循环、while循环、do while循环和if else语句）。

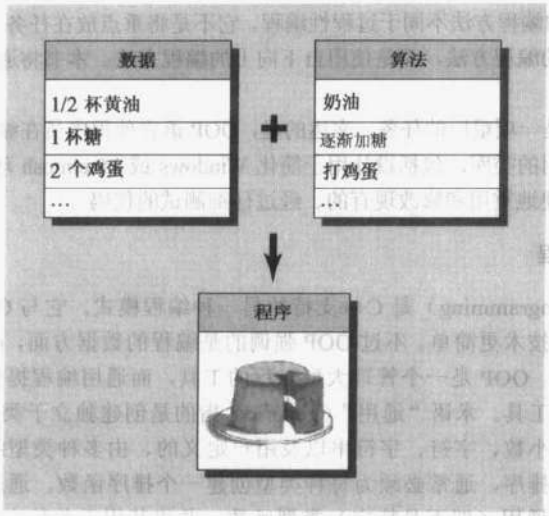


图 1.1　数据+算法=程序

另一个新原则是自顶向下（top-down）的设计。在 C 语言中，其理念是将大型程序分解成小型、便于管理的任务。如果其中的一项任务仍然过大，则将它分解为更小的任务。这一过程将一直持续下去，直到将程序划分为小型的、易于编写的模块（整理一下书房。先整理桌子、桌面、档案柜，然后整理书架。好，先从桌子开始，然后整理每个抽屉，从中间的那个抽屉开始整理。也许我都可以管理这项任务）。C 语言的设计有助于使用这种方法，它鼓励程序员开发程序单元（称为函数）来表示各个任务模块。如上所述，结构化编程技术反映了过程性编程的思想，根据执行的操作来构思一个程序。

1.2.3　面向对象编程

虽然结构化编程的理念提高了程序的清晰度、可靠性，并使之便于维护，但它在编写大型程序时，仍面临着挑战。为应付这种挑战，OOP 提供了一种新方法。与强调算法的过程性编程不同的是，OOP 强调的是数据。OOP 不像过程性编程那样，试图使问题满足语言的过程性方法，而是试图让语言来满足问题的要求。其理念是设计与问题的本质特性相对应的数据格式。

在 C++中，类是一种规范，它描述了这种新型数据格式，对象是根据这种规范构造的特定数据结构。例如，类可以描述公司管理人员的基本特征（姓名、头衔、工资、特长等），而对象则代表特定的管理人员（Guilford Sheepblat、副总裁、$325 000、知道如何恢复 Windows 注册表）。通常，类规定了可使用哪些数据来表示对象以及可以对这些数据执行哪些操作。例如，假设正在开发一个能够绘制矩形的计算机绘图程序，则可以定义一个描述矩形的类。定义的数据部分应包括顶点的位置、长和宽、4 条边的颜色和样式、矩形内部的填充颜色和图案等；定义的操作部分可以包括移动、改变大小、旋转、改变颜色和图案、将矩形复制到另一个位置上等操作。这样，当使用该程序来绘制矩形时，它将根据类定义创建一个对象。该对象保存了描述矩形的所有数据值，因此可以使用类方法来修改该矩形。如果绘制两个矩形，程序将创建两个对象，每个矩形对应一个。

OOP 程序设计方法首先设计类，它们准确地表示了程序要处理的东西。例如，绘图程序可能定义表示矩形、直线、圆、画刷以及画笔的类。类定义描述了对每个类可执行的操作，如移动圆或旋转直线。然后您便可以设计一个使用这些类的对象的程序。从低级组织（如类）到高级组织（如程序）的处理过程叫做自下向上（bottom-up）的编程。

OOP 编程并不仅仅是将数据和方法合并为类定义。例如，OOP 还有助于创建可重用的代码，这将减少大量的工作。

信息隐藏可以保护数据，使其免遭不适当的访问。多态让您能够为操作符和函数创建多个定义，通过

编程上下文来确定使用哪个定义。继承让您能够使用旧类派生出新类。正如接下来将看到的那样，OOP 引入了很多新的理念，使用的编程方法不同于过程性编程。它不是将重点放在任务上，而是放在表示概念上。有时不一定使用自顶向下的编程方法，而是使用自下向上的编程方法。本书将通过大量易于掌握的范例帮助读者理解这些要点。

设计有用、可靠的类是一项艰巨的任务，幸运的是，OOP 语言使程序员在编程中能够轻松地使用已有的类。厂商提供了大量有用的类库，包括设计用于简化 Windows 或 Macintosh 环境下编程的类库。C++真正的优点之一是：可以方便地重用和修改现有的、经过仔细测试的代码。

1.2.4 C++和通用编程

通用编程（generic programming）是 C++支持的另一种编程模式。它与 OOP 的目标相同，即使重用代码和抽象通用概念的技术更简单。不过 OOP 强调的是编程的数据方面，而通用编程强调的是算法方面。它们的侧重点不同。OOP 是一个管理大型工程的工具，而通用编程提供了执行常见任务（如对数据排序或合并链表）的工具。术语“通用”（generic）指的是创建独立于类型的代码。C++的数据表示有多种类型——整数、小数、字符、字符串以及用户定义的、由多种类型组成的复合结构。例如，要对不同类型的数据进行排序，通常必须为每种类型创建一个排序函数。通用编程需要对语言进行扩展，以便可以只编写一个通用（即不是特定）类型函数，并将其用于各种实际类型。C++模板提供了完成这种任务的机制。

1.2.5 C++的起源

和 C 语言一样，C++也是在贝尔实验室诞生的，Bjarne Stroustrup 于 20 世纪 80 年代在这里开发出了这种语言。用他自己的话来说，“C++主要是为了我的朋友和我不必再使用汇编语言、C 语言或其他现代高级语言来编程而设计的。它的主要功能是可以更方便地编写出好程序，让每个程序员更加快乐”。

Bjarne Stroustrup 的主页

Bjarne Stroustrup 设计并实现了 C++编程语言，他是权威参考手册 *The C++ Programming Language* 和 *The design and Evolution of C++* 的作者。读者应将他位于 AT&T Labs Research 上的个人网站作为首选的 C++书签：

```
http://www.research.att.com/~bs/
```

该网站包括了 C++语言有趣的发展历史、Bjarne 的传记材料和 C++ FAQ。Bjarne 被问得最多的问题是：Bjarne Stroustrup 应该如何读。您可以下载.WAV 文件，亲自听一听。

Stroustrup 比较关心的是让 C++更有用，而不是实施特定的编程原理或风格。在确定 C++语言特性方面，真正的编程需要比纯粹的原理更重要。Stroupstrup 之所以在 C 的基础上创建 C++，是因为 C 语言简洁、适合系统编程、使用广泛且与 UNIX 操作系统联系紧密。C++的 OOP 方面是受到了计算机模拟语言 Simula67 的启发。Stroustrup 加入了 OOP 特性和对 C 的通用编程支持，但并没有对 C 的组件作很大的改动。因此，C++是 C 语言的超集，这意味着任何有效的 C 程序都是有效的 C++程序。它们之间有些细微的差异，但无足轻重。C++程序可以使用已有的 C 软件库。库是编程模块的集合，可以从程序中调用它们。库对很多常见的编程问题提供了可靠的解决方法，因此能节省程序员大量的时间和工作量。这也有助于 C++的广泛传播。

名称 C++来自 C 语言中的递增操作符++，该操作符将变量加 1。名称 C++表明，它是 C 的扩充版本。

计算机程序将实际问题转换为计算机能够执行的一系列操作。OOP 部分赋予了 C++语言将问题所涉及的概念联系起来的能力，C 部分则赋予了 C++语言紧密联系硬件的能力（参见图 1.2），这种能力上的结合成就了 C++的广泛传播。从程序的一个方面转到另一个方面时，思维方式也要跟着转换（确实，有些 OOP 正统派把为 C 添加 OOP 特性看作是为猪插上翅膀，虽然这是头瘦骨嶙峋、非常能干的猪）。另外，由于 C++是在 C 语言的基础上添加 OOP 特性，因此可以忽略 C++面向对象的特性，但这将错过很多有用的东西。

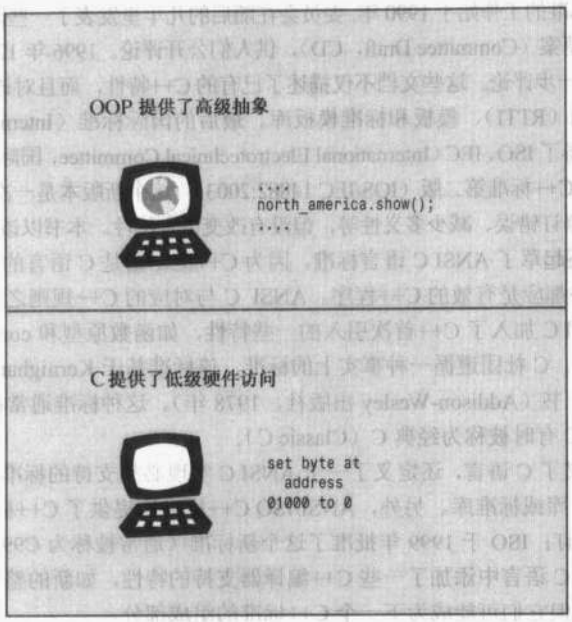


图 1.2　C++的二重性

在 C++获得一定程度的成功后，Stroustrup 才添加了模板，这使得进行通用编程成为可能。在模板特性被使用和改进后，人们才逐渐认识到，它们和 OOP 同样重要——甚至比 OOP 还重要，但有些人不这么认为。C++融合了 OOP、通用编程和传统的过程性方法，这表明 C++强调的是实用价值，而不是意识形态方法，这也是该语言获得成功的原因之一。

1.3　可移植性和标准

假设为老式奔腾 PC 编写了一个很好用的 C++程序，而此时管理人员决定用 Macintosh G5（使用不同的处理器和操作系统）替换这台机器。该程序是否可以在新平台上运行呢？当然，必须使用为新平台设计的 C++编译器对程序重新进行编译。但是否需要修改编写好的代码呢？如果在不修改代码的情况下，重新编译程序后，程序将运行良好，则该程序是可移植的。

在可移植性方面存在两个障碍，其中的一个是硬件。硬件特定的程序是不可移植的。例如，直接控制 IBM PC 视频卡的程序在涉及到 Sun 时将“胡言乱语”（将依赖于硬件的部分放在函数模块中，可以最大限度地降低可移植性问题，这样只需重新编写这些模块即可）。本书将避免这种编程。

可移植性的第二个障碍是语言上的差异。口语确实可能产生问题。约克郡的人对某个事件的描述，布鲁克林人可能就听不明白，即使这两个地方的人都说英语。计算机语言也可能出现方言。Windows XP C++的实现与 Red Hat Linux 或 Macintosh OS X 实现相同吗？虽然多数实现都希望其 C++版本与其他版本兼容，但如果没有准确描述语言工作方式的公开标准，这将很难做到。因此，美国国家标准局（American National Standards Institute，ANSI）在 1990 年设立了一个委员会（ANSI X3J16），专门负责制定 C++标准（ANSI 制定了 C 语言标准）。国际标准化组织（ISO）很快通过自己的委员会（ISO-WG-21）加入了这个行列，创建了联合组织 ANSI/ISO，致力于制定 C++标准。这些组织每年 3 次聚在一起，我们将它们统称为 ANSI/ISO 委员会。ANSI/ISO 决定制定一种标准，强调 C++是一种重要的、广泛应用的语言。它还表明 C++已经达到了一定程度的成熟，因为在语言高速发展时，它是不会引入标准的。无论如何，自从 ANSI/ISO 委员会开始工作后，C++便发生了巨大的变化。

制定ANSI/ISO C++标准的工作始于1990年。委员会在随后的几年里发表了一些阶段性的工作论文。在1995年4月，它发表了委员会草案（Committee Draft，CD），供人们公开评论。1996年12月，它发表了该草案的第2版（CD2），让公众做进一步评论。这些文档不仅描述了已有的C++特性，而且对该语言进行了扩展，添加了异常、运行阶段类型识别（RTTI）、模板和标准模板库。最后的国际标准（International Standard，ISO/IEC 14882:1998）在1998年获得了ISO、IEC（International Electrotechnical Committee，国际电工技术委员会）和ANSI的批准。2003年，发布了C++标准第二版（IOS/IEC 14882:2003）；这个新版本是一次技术性修订，这意味着它对第一版进行了整理——修订错误、减少多义性等，但没有改变语言特性。本书以该标准为基础。

ANSI/ISO C++标准还起草了ANSI C语言标准，因为C++应尽量是C语言的超集。这意味着在理想情况下，任何有效的C程序都应是有效的C++程序。ANSI C与对应的C++规则之间存在一些差别，但这种差别很小。实际上，ANSI C加入了C++首次引入的一些特性，如函数原型和const类型限定符。

在ANSI C出现之前，C社团遵循一种事实上的标准，该标准基于Kernighan和Ritchie编写的*The C Programming Language*一书（Addison-Wesley出版社，1978年）。这种标准通常被叫做K&R C；ANSI C出现后，更简单的K&R C有时被称为经典C（Classic C）。

ANSI C标准不仅定义了C语言，还定义了一个ANSI C实现必须支持的标准C库。C++也使用了这个库；本书将它称作标准C库或标准库。另外，ANSI/ISO C++标准还提供了C++标准类库。

最近，C标准又被修订；ISO于1999年批准了这个新标准（通常被称为C99），而ANSI则于2000年批准了该标准。该标准在C语言中添加了一些C++编译器支持的特性，如新的整型。虽然这些特性不是当前C++标准的组成部分，但它们可能成为下一个C++标准的组成部分。

在ANSI/ISO C++委员会开始工作之前，很多人都将贝尔实验室的C++版本作为标准。例如，编译器可能把它自己描述成能够兼容C++的Release 2.0和Release 3.0。

C++在不断发展，下一版C++标准的制定工作已经开始，该版本被称为C++ 0X，因为其预期完工时间为2009年左右。

本书介绍ISO/ANSI C++标准第二版（ISO/IEC 14882:2003），因此其中的范例适用于任何与该标准兼容的C++实现。然而，C++标准仍是新生事物，您可能遇到一些没有遵循该标准的情况。例如，如果您的编译器不是最新的版本，可能没有名称空间和最新的模板特性。较老的编译器可能不支持第16章将介绍的STL。有些老式UNIX系统使用前端转换器，后者将转换后的代码传递给不与ANSI完全兼容的C编译器，其后果是，有些语言特性没有得到实现，有些标准ANSI库函数和头文件没有得到支持。即使编译器遵循了ANSI标准，有些方面，如用于存储整数的字节数也可能随实现而异。

介绍C++语言之前，我们先来介绍一些有关创建程序和使用本书的背景知识。

1.4 程序创建的技巧

假设编写了一个C++程序。如何让它运行起来呢？具体的步骤取决于计算机环境和使用的C++编译器，但大体如下（参见图1.3）：

1. 使用文本编辑器编写程序，并将其保存到文件中，这个文件就是程序的源代码。

2. 编译源代码。这意味着运行一个程序，该程序将源代码翻译为主机使用的内部语言——机器语言。包含了翻译后的程序的文件就是程序的目标代码（object code）。

3. 将目标代码与其他代码链接起来。例如，C++程序通常使用库。C++库包含一系列计算机例程（被称为函数）的目标代码，这些函数可以执行诸如在屏幕上显示信息或计算数字的平方根等任务。链接指的是将目标代码同使用的函数的目标代码以及一些标准的启动代码（startup code）组合起来，生成程序的运行阶段版本。包含该最终产品的文件被称为可执行代码。

本书将不断使用术语“源代码”，请记住该术语。

本书的程序都是通用的，可在任何支持现代C++的系统中运行（不过，为支持名称空间和最新的模板特性，可能需要使用最新的版本）。创建程序的步骤可能各不相同，下面深入介绍这些步骤。

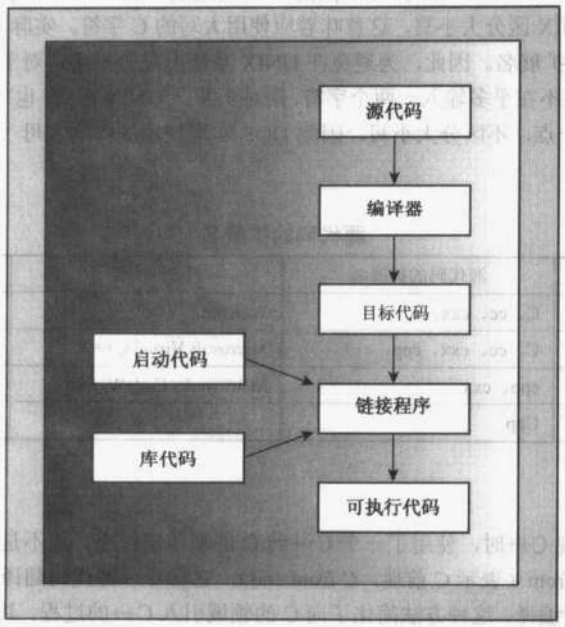


图 1.3　编程步骤

1.4.1　创建源代码文件

本书余下的篇幅讨论源代码文件中的内容，本节讨论创建源代码文件的技巧。有些 C++实现（如 Microsoft Visual C++、各种版本的 Borland C++、Watcom C++、Digital Mars 和 Metrowerks CodeWarrior）提供了集成开发环境（integrated development environments，IDE），让您能够在主程序中管理程序开发的所有步骤，包括编辑。有些实现（如 AT&T C++、UNIX 和 Linux 上的 GNU C++以及 Borland 和 Digital Mars 编译器的免费版本）只能处理编译和链接阶段，要求在系统命令行上输入命令。在这种情况下，可以使用任何文本编辑器来创建和修改源代码。例如，在 UNIX 系统上，可以使用 vi、ed、ex 或 emacs；在 DOS 系统上，可以使用 edlin、edit 或任何程序编辑器。如果将文件保存为标准的 DOS ASCII 文本文件（而不是特殊的字处理器格式），则甚至可以使用字处理器。

给源文件命名时，必须使用正确的后缀，将文件标识为 C++文件。这不仅告诉您，该文件是 C++源代码，而且将这种信息告知编译器（如果 UNIX 编译器显示信息“bad magic number”，则表明后缀不正确）。后缀由一个句点和一个或多个字符组成，这些字符被称作扩展名（参见图 1.4）。

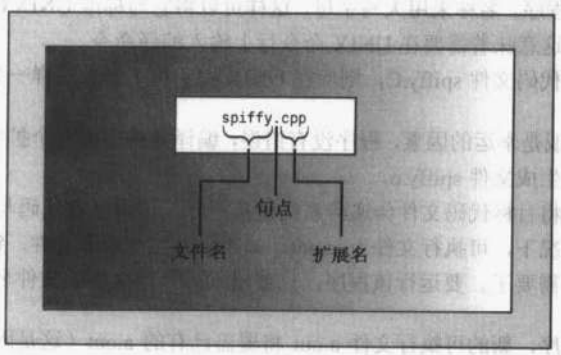


图 1.4　源代码文件名

使用什么扩展名取决于C++的实现，表1.1列出了一些常用的扩展名。例如，spiffy.C是有效的UNIX C++源代码文件名。注意，UNIX区分大小写，这意味着应使用大写的C字符。实际上，小写c扩展名也有效，但标准C才使用小写的c扩展名。因此，为避免在UNIX系统上发生混淆，对于C程序应使用c，而对于C++程序则请使用C。如果不在乎多输入一两个字符，则对于某些UNIX系统，也可以使用扩展名cc和cxx。DOS比UNIX稍微简单一点，不区分大小写，因此DOS实现使用额外的字母（如表1.1所示）来区别C和C++程序。

表1.1 源代码的扩展名

C++实现	源代码的扩展名	C++实现	源代码的扩展名
UNIX	C、cc、cxx、c	Watcom	cpp
GNU C++	C、cc、cxx、cpp、c++	Microsoft Visual C++	cpp、cxx、cc
Digital Mars	cpp、cxx	Metrowerks CodeWarrior	cp、cpp、cc、cxx、c++
Borland C++	Cpp		

1.4.2 编译和链接

最初，Stroustrup实现C++时，使用了一个C++到C的编译器程序，而不是开发直接的C++到目标代码的编译器。前者叫做cfront（表示C前端，C front end），它将C++源代码翻译成C源代码，然后使用一个标准C编译器对其进行编译。这种方法简化了向C的领域引入C++的过程。其他实现也采用这种方法将C++引入到其他平台。随着C++的日渐普及，越来越多的实现转向创建C++编译器，直接将C++源代码生成目标代码。这种直接方法加速了编译过程，并强调C++是一种独立（虽然有些相似）的语言。

对用户而言，cfront翻译器与编译器之间的差别通常是不可见的。例如，在UNIX系统上，CC命令首先将程序传递给cfront翻译器，然后自动将该翻译器的输出传递给C编译器cc。从现在开始，术语“编译器”指的是翻译并编译。编译的机理取决于实现，接下来的几节将介绍一些常见的形式。这些总结概括了基本步骤，但对于具体步骤，必须查看系统文档。

如果有cfront翻译器，并了解C语言，则可能想看一看将C++程序翻译为C程序的过程，以了解某些C++特性是如何被实现的。

1. UNIX编译和链接

对于传统的UNIX系统C++编译器，使用命令CC来启动。然而，当今的UNIX计算机可能没有编译器、有专用编译器或第三方编译器，这些编译器可能是商业的，也可能是自由软件，如GNU g++编译器。如果UNIX计算机上有C++编译器，很多情况下命令CC仍然管用，只是启动的编译器随系统而异。出于简化的目的，读者应假设命令CC可用，但必须认识到这一点，即对于下述讨论中的CC，可能必须使用其他命令来代替。

请用CC命令来编译程序。名称采用大写字母，这样可以将它与标准UNIX C编译器cc区分开来。CC编译器是命令行编译器，这意味着需要在UNIX命令行上输入编译命令。

例如，要编译C++源代码文件spiffy.C，则应在UNIX提示符下输入这样一条命令：

```
CC spiffy.C
```

如果由于技巧、努力或是幸运的因素，程序没有错误，编译器将生成一个扩展名为o的目标代码文件。在这个例子中，编译器将生成文件spiffy.o。

接下来，编译器自动将目标代码文件传递给系统链接程序，该程序将代码与库代码结合起来，生成一个可执行文件。在默认情况下，可执行文件为a.out。如果只使用一个源文件，链接程序还将删除spiffy.o文件，因为这个文件不再需要了。要运行该程序，只要输入可执行文件的文件名即可：

```
a.out
```

注意，如果编译新程序，新的可执行文件a.out将覆盖已有的a.out（这是因为可执行文件占据了大量空间，因此覆盖旧的可执行文件有助于降低存储需求）。不过，如果想保留可执行文件，只需使用UNIX

的 mv 命令来修改可执行文件的文件名即可。

与在 C 语言中一样，在 C++中，程序也可以包含多个文件（本书第 8～16 章的很多程序都是这样）。在这种情况下，可以通过在命令行上列出全部文件来编译程序：

```
CC my.C precious.C
```

如果有多个源代码文件，则编译器将不会删除目标代码文件。这样，如果只修改了 my.C 文件，则可以用下面的命令重新编译该程序：

```
CC my.C precious.o
```

这将重新编译 my.C 文件，并将它与前面编译的 precious.o 文件链接起来。

可能需要显式地指定一些库。例如，要访问数学库中定义的函数，必须在命令行中加上-lm 标记：

```
CC usingmath.C -lm
```

2. Linux 编译和链接

Linux 系统中最常用的编译器是 g++，这是来自 Free Software Foundation 的 GNU C++编译器。Linux 的多数版本都包括该编译器，但并不一定总是会安装它。g++编译器的工作方式很像标准 UNIX 编译器：

```
g++ spiffy.cxx
```

这将生成一个可执行文件 a.out。

有些版本可能要求链接 C++库：

```
g++ spiffy.cxx -lg++
```

要编译多个源文件，只需将它们全部放到命令行中即可：

```
g++ my.cxx precious.cxx
```

这将生成一个名为 a.out 的可执行文件和两个目标代码文件 my.o 和 precious.o。如果接下来修改了其中的某个源代码文件，如 mu.cxx，则可以使用 my.cxx 和 precious.o 来重新编译：

```
g++ my.cxx precious.o
```

另一个可能的选择是 Comeau C++编译器（www.comeaucomputing.com），它要求使用 GNU 编译器。然而，Comeau 编译器提供了最完备、最严格的 C++标准实现。

GNU 编译器可以在很多平台上使用，包括基于 Windows 的 PC 和在各种平台上运行的 UNIX 系统。

3. Windows 命令行编译器

要在 Windows PC 上编译 C++程序，最便宜的方法是下载一个在 Windows MS-DOS 窗口中运行的免费命令行编译器。GNU C++编译器的 MS-DOS 版本名为 gpp，可以从 www.delorie.com/djgpp/下载。Borland 公司在 www.borland.com/bcppbuilder/freecompiler/上提供了一个免费的命令行编译器。相对便宜的 Borlnd C++BuilderX 个人版自带了该编译器的新版本。Digital Mars 提供了一个免费的命令行编译器，其下载地址为 www.digitalmars.com。C++BuilderX 的安装是自动完成的；对于其他编译器，需要阅读安装指南，因为有些安装步骤不会自动完成。

要使用 gpp 编译器，首先需要打开一个 MS-DOS 窗口。要编译名为 great.cpp 的源代码文件，请在提示符下输入下面的命令：

```
gpp great.cpp
```

如果程序编译成功，则得到的可执行文件名为 a.exe。

要使用 Borland 在线（online）编译器，首先需要打开一个 MS-DOS 窗口。要编译源代码文件 great.cpp，可在提示符下输入下面的命令：

```
bcc32 great.cpp
```

如果程序编译成功，则生成的可执行文件名为 great.exe。

4. Windows 编译器

Windows 平台上的产品很多，且修订频繁，无法对它们分别进行介绍。其中最流行的包括 Microsoft、Borland、Metrowerks 和 Digital Mars 等公司提供的编译器，它们的设计和目标虽然不同，但有一些共同特征。

通常，必须为程序创建一个工程，并将组成程序的一个或多个文件添加到该工程中。每个厂商提供的

IDE（集成开发环境）都包含用于创建工程的菜单选项（可能还有自动帮助）。必须确定的非常重要的一点是，需要创建的是什么类型的程序。通常，编译器提供了很多选择，如 Windows 应用程序、MFC Windows 应用程序、动态链接库、ActiveX 控件、DOS 或字符模式的可执行文件、静态库或控制台应用程序等。其中的一些可能既有 16 位版本，又有 32 位版本。

由于本书的程序都是通用的，因此应当避免要求平台特定代码的选项，如 Windows 应用程序。相反，希望程序以字符模式运行。具体选项取决于编译器。对于 Microsoft Visual C++，应选择 Win32 Console Application（控制台应用程序）选项；如果您使用的是 Visual Studio.NET，也可以选择 Empty Project 选项 | Application Settings。Metrowerks 编译器提供了 Win32 Console C++ App 选项和 Win32 WinSIOUX C++ App 选项，使用哪一个都可以（前者在 DOS 窗口中运行编译后的程序，而后者在标准 Windows 窗口中运行这种程序）。有些 Borland 版本包含一个模拟 DOS 会话的 EasyWin 选项；有些版本则提供了 Console 选项。总的来说，应检查是否有标有 Console、字符模式或 DOS 可执行等的选项，并使用这样的选项。

创建好工程后，需要对程序进行编译和链接。IDE 通常提供了多个菜单项，如 Compile（编译）、Build（建立）、Make（生成）、Build All（全部建立）、Link（链接）、Execute（执行）和 Run（运行），不过同一个 IDE 中不一定包含所有这些选项。

- Compile 通常意味着对当前打开的文件中的代码进行编译。
- Build 和 Make 通常意味着编译工程中所有源代码文件的代码。这通常是一个递增过程，也就是说，如果工程包含 3 个文件，而只有其中一个文件被修改，则只重新编译该文件。
- Build All 通常意味着重新编译所有的源代码文件。
- Link 意味着（如前所述）将编译后的源代码与所需的库代码组合起来。
- Run 或 Execute 意味着运行程序。通常，如果您还没有执行前面的步骤，Run 将在运行程序之前完成这些步骤。

如果程序违反了语言规则，编译器将生成错误消息，指出存在问题的行。遗憾的是，如果不熟悉语言，将难以理解这些消息的含义。有时，真正的问题可能在标识行之前；有时，一个错误可能引发一连串的错误消息。

提示：改正错误时，应首先改正第一个错误。如果在标识为有错误的那一行上找不到错误，请查看前面的一行

需要注意的是，程序能够通过某个编译器的编译并不意味着它是合法的 C++程序；同样，程序不能通过某个编译器的编译也并不意味着它是非法的 C++程序。与两三年前相比，现在的编译器更严格地遵循了 C++标准。当前，Comeau 编译器最为严格、准确地遵循了 C++标准。

提示：有时，编译器在不完全地构建程序后将出现混乱，它显示无法改正的、无意义的错误消息。在这种情况下，可以选择 Build All，重新编译整个程序，以清除这些错误消息。遗憾的是，这种情况和那些更常见的情况（即错误消息只是看上去无意义，实际上有意义）很难区分。

通常，IDE 允许在辅助窗口中运行程序。程序执行完毕后，有些 IDE 将关闭该窗口，而有些 IDE 则不关闭。如果编译器关闭窗口，将难以看到程序的输出，除非程序员眼疾手快，且过目不忘。为查看输出，必须在程序的最后加上一些代码：

```
    cin.get();  // add this statement
    cin.get();  // and maybe this, too
    return 0;
}
```

cin.get()语句读取下一次键击，因此上述语句让程序等待，直到按下了 Enter 键（在按下 Enter 之前，键击将不被发送给程序，因此按其他键都不管用）。如果程序在其常规输入后留下一个没有被处理的键击，则第二条语句是必需的。例如，如果要输入一个数字，则需要输入该数字，然后按 Enter 键。程序将读取该数字，但 Enter 键不被处理，这样它将被第一个 cin.get()读取。

Borland C++ Builder 编译器与传统设计有些不同，其主要目标是 Windows 编程。要使用其老式版本来编写通用程序，请选择菜单 File（文件）/New（新建），然后选择 Console App（控制台应用程序），这样将

打开一个包括 main()框架版本的窗口，可以删除其中的一些项目，但应保留下面这两行非标准代码：

```
#include <vcl\condefs.h>
#pragma hdrstop
```

在 C++BuiderX 中，选择菜单 File/New/New Console 后，窗口中没有 main()的框架版本。您需要选择菜单 File/New File，在工程中添加一个新的.cpp 文件。

5. Macintosh 上的 C++

主要的 Macintosh C++编译器是 Metrowerks CodeWarrior，它提供了基于工程的 IDE，从基本概念上说，这种 IDE 与 Windows 编译器中的 IDE 类似。首先从 File 菜单中选择 New Project（新建工程），系统允许选择工程类型。对于老版本的 CodeWarrior，请选择 MacOS:C/C++:ANSI C++ Console；对于其新版本，请选择 MacOS:C/C++:Standard Console:Std C++ Console 或 MacOS C++ Stationery:Mac OS Carbon:Standard Console: C++ Console Carbon。当然，还有其他选择，例如可以选择 Classic 而不是 Carbon，选择 C++ Console Carbon Altivec 而不是常规 C++ Console Carbon。

CodeWarrior 在初始工程中提供了一个小型源代码文件。可以对该程序进行编译和运行，看看系统是否设置正确。不过，提供自己的代码后，应从工程中删除该文件，方法是从工程窗口中选中该文件，然后选择 Project（工程）/Remove（删除）菜单项。

接下来，必须将自己的源代码添加到工程中，方法是使用 File 菜单中的 New 选项来新建一个文件或选择 File 菜单中的 Open（打开）选项来打开一个已有的文件。使用正确的后缀（如.cp 或.cpp）保存文件，然后使用 Project 菜单将该文件添加到工程列表中。本书中的一些程序要求添加多个源代码文件。一切准备就绪后，从 Project 菜单中选择 Run（运行）菜单项。

提示： *为节省时间，可对所有范例程序使用同一个工程。方法是从工程列表中删除前一个范例程序的源代码文件，并添加当前的源代码。这样可节省磁盘空间。*

编译器包含一个调试程序，帮助您找到运行阶段问题的位置。

1.5　总结

随着计算机的功能越来越强大，计算机程序越来越庞大而复杂。为应对这种挑战，计算机语言也得到了改进，以便编程过程更为简单。C 语言新增了诸如控制结构和函数等特性，以便更好地控制程序流程，支持结构化和模块化程度更高的方法；而 C++增加了对面向对象编程和通用编程的支持，这有助于提高模块化和创建可重用代码，从而节省编程时间并提高程序的可靠性。

C++的流行导致大量使用于各种计算平台的 C++实现得以面世；ISO/ANSI C++标准为确保众多实现的相互兼容提供了基础。该标准规定了语言必须具备的特性、语言呈现出的行为、标准库函数、类和模板；它旨在实现该语言在不同计算平台和实现之间的可移植性。

要创建 C++程序，可创建一个或多个源代码文件，其中包含了以 C++语言表示的程序。这些文件是文本文件，它们经过编译和链接后将得到机器语言文件，后者构成了可执行的程序。上述任务通常是在 IDE 中完成的，IDE 提供了用于创建源代码文件的文本编辑器、用于生成可执行文件的编译器和链接器以及其他资源，如工程管理和调试功能。然而，这些任务也可以在命令行环境中通过调用合适的工具来完成。

第 2 章　开始学习 C++

本章内容包括：

- 如何创建 C++程序。
- C++程序的一般格式。
- #include 编译指令。
- main() 函数。
- 如何使用 cout 对象进行输出。
- 如何在 C++程序中加入注释。
- 何时以及如何使用 endl。
- 如何声明和使用变量。
- 如何使用 cin 对象进行输入。
- 如何定义和使用简单函数。

要建造简单的房屋，首先要打地基、搭建框架。如果一开始没有牢固的结构，后面就很难构建窗子、门框、圆屋顶和镶木地板的舞厅等。同样，学习计算机语言时，应从程序的基本结构开始学起。只有这样，才能一步一步了解其具体细节，如循环和对象等。这一章将要对 C++程序的基本结构做一概述，并预览一下后面章节将介绍的主题，如函数和类（这里的理念是，先介绍一些基本概念，这样可以激发读者接下去学习的兴趣）。

2.1　进入 C++

首先介绍一个显示消息的简单 C++程序。程序清单 2.1 使用 C++工具 cout 生成字符输出。源代码中包含一些供读者阅读的注释，这些注释都以//打头，编译器将忽略它们。C++对大小写敏感，也就是说区分大写字符和小写字符。这意味着大小写必须与范例中相同。例如，该程序使用的是 cout，如果将其替换为 Cout 或 COUT，程序将无法通过编译，并且编译器将指出使用了未知的标识符（编译器也是对拼写敏感的，因此请不要使用 kout 或 coot）。文件扩展名 cpp 是一种表示 C++程序的常用方式，您可能需要使用第 1 章介绍的其他扩展名。

程序清单 2.1　myfirst.cpp

```
// myfirst.cpp--displays a message

#include <iostream>                          // a PREPROCESSOR directive
int main()                                   // function header
{                                            // start of function body
   using namespace std;                      // make definitions visible
   cout << "Come up and C++ me some time."; // message
   cout << endl;                             // start a new line
```

```
    cout << "You won¡¯t regret it!" << endl;
    return 0;                                    // terminate main()
}                                                // end of function body
```

注意：如果使用的是老式编译器，可能需要使用#include <iostream.h>来代替#include <iostream>；在这种情况下，还应当省略代码行“using namespace std;”。也就是说，将下面的代码：

```
#include <iostream>    // the way of the future
```

替换为：

```
#include <iostream.h>  // in case the future has not yet arrived
```

并省略下面的代码行：

```
using namespace std;   // also the way of the future
```

有些非常老式的编译器使用#include <stream.h>，而不是#include <iostream.h>；如果所用的编译器就是这样老式的，则要么换一台新的编译器，要么参考以前的书籍。将 iostream.h 替换为 iostream 是最近的事，读者可能遇到还没有实现 iostream 的编译器。

一些 Windows 环境在独立的窗口中运行程序，程序运行完毕后，自动关闭该窗口。第 1 章讨论过，可以让该窗口一直打开，直到用户按下 Enter 键为止，方法是在 return 语句前添加下面这行代码：

```
cin.get();
```

对于某些程序，必须添加两行这样的代码。该代码让程序等待键击。第 4 章将更详细地介绍这行代码。

程序调整

如上所示，要在自己的系统上运行本书的范例，必须对其进行修改。两种最常见的修改请参见本章的第一条“注意”。其中的一项修改涉及到语言标准，即如果编译器不是最新的，就必须包含 iostream.h（而不是 iostream），并省略引入名称空间的一行。第二项修改涉及编程环境，可能需要添加一或两条 cin.get() 语句，使程序输出在屏幕上可见。由于这些调整适用于书中的每个范例，因此这个“注意”并不是惟一的警告　别把它们忘了！后面的“注意”将提示需要做的其他修改。

在编辑器中输入该程序（或使用 Sams Publishing 网站中的源代码，地址为 www.samspublishing.com）后，可以使用 C++编译器创建可执行代码（参见第 1 章的介绍）。下面是运行编译后的程序时得到的输出：

```
Come up and C++ me some time.
You won¡¯t regret it!
```

C 语言的输入和输出

如果已经使用过 C 语言来编程，则看到 cout 函数（而不是 printf() 函数）时可能会小吃一惊。事实上，C++能够使用 printf()、scanf() 和其他所有标准 C 输入和输出函数，只需要包含常规 C 语言的 stdio.h 文件。不过本书介绍的是 C++，所以将使用 C++的输入工具，它们在 C 版本的基础上作了很多改进。

使用函数来创建 C++程序。通常，先将程序组织为主要任务，然后设计独立的函数来处理这些任务。程序清单 2.1 中的范例非常简单，只包含一个名为 main() 的函数。myfirst.cpp 范例包含以下元素：

- 注释，由//前缀指示。
- 预处理器编译指令#include。
- 函数头：int main()。
- 编译指令 using namespace。
- 函数体，用{和}括起。
- 使用 C++的 cout 工具显示消息的语句。
- 结束 main() 函数的返回语句。

下面详细介绍这些元素。先来看看 main() 函数，因为了解了 main() 的作用后，main() 前面的一些特性（如预处理器编译指令）将更易于理解。

2.1.1 main()函数

去掉修饰后，程序清单 2.1 中的范例程序的基本结构如下：

```
int main()
{
   statements
   return 0;
}
```

这几行表明有一个名为 main()的函数，并描述了该函数的行为。这几行代码构成了函数定义（function definition）。该定义由两部分组成：第一行 int main()叫函数头（function heading），花括号({和})中包括的部分叫函数体（function body）。图 2.1 所示对 main()函数做了说明。函数头对函数与程序其他部分之间的接口进行了总结；函数体是指出函数应做什么的计算机指令。在 C++中，每条完整的指令都称为语句（statement）。所有的语句都以分号结束，因此在输入范例代码时，请不要省略分号。

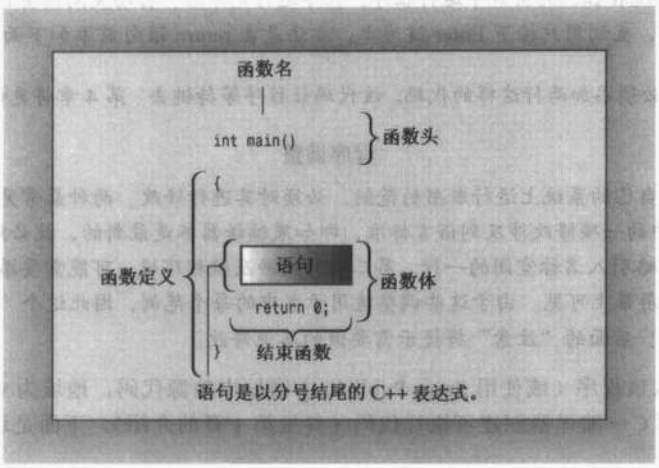


图 2.1 main()函数

main()中最后一条语句叫做返回语句（return statement），它结束该函数。本章将讲述有关返回语句的更多知识。

语句和分号

语句是一条完整的计算机指令。为理解源代码，编译器需要知道一条语句何时结束，另一条语句何时开始。有些语言使用语句分隔符。例如，FORTRAN 通过行尾将语句分隔开来，Pascal 使用分号分隔语句。在 Pascal 中，有些情况下可以省略分号，例如 END 前的语句后面，实际上并没有将两条语句分开。不过 C++与 C 一样，也使用终止符（terminator），而不是分隔符。终止符是一个分号，它是语句的结束标记，是语句的组成部分，而不是语句之间的标记。结论是：在 C++中，不能省略分号。

1. 作为接口的函数头

就目前而言，需要记住的主要一点是，C++句法要求 main()函数的定义以函数头 int main()开始。本章后面的“函数”一节将详细讨论函数头句法，不过，为满足读者的好奇心，下面先预览一下。

通常，C++函数可被其他函数激活或调用，函数头描述了函数与调用它的函数之间的接口。位于函数名前面的部分叫做函数返回类型（function return type），它描述的是从函数返回给调用它的函数的信息。函数名后括号中的部分叫做形参列表（argument list）或参数列表（parameter list）；它描述的是从调用函数传递给被调用的函数的信息。这种通用格式用于 main()时让人感到有些迷惑，因为通常并不从程序的其他部分调用 main()。

不过，通常，main()被启动代码调用，而启动代码是由编译器添加到程序中的，是程序和操作系统（UNIX、Windows XP 或其他操作系统）之间的桥梁。事实上，该函数头描述的是 main()和操作系统之间的接口。

来看一下 main()的接口，该接口从 int 开始。C++函数可以给调用函数返回一个值，这个值叫做返回值（return value）。在这里，从关键字 int 可知，main()返回一个整数值。接下来，是空括号。通常，C++函数在调用另一个函数时，可以将信息传递给该函数。括号中的函数头部分描述的就是这种信息。在这里，空括号意味着 main()函数不接受任何信息，或者 main()不接受任何参数（main()不接受任何参数并不意味着 main()是不讲道理的、发号施令的函数。相反，术语参数（argument）只是计算机人员用来表示从一个函数传递给另一个函数的信息）。

简而言之，下面的函数头

```
int main()
```

表明 main()函数可以给调用它的函数返回一个整数值，且不从调用它的函数那里获得任何信息。

很多现有的程序都使用经典 C 函数头：

```
main()    // original C style
```

在 C 语言中，省略返回类型相当于说函数的类型为 int。不过，C++逐步淘汰了这种用法。

也可以使用下面的变体：

```
int main(void)    // very explicit style
```

在括号中使用关键字 void 明确地指出，函数不接受任何参数。在 C++（不是 C）中，让括号空着与在括号中使用 void 等效（在 C 中，让括号空着意味着对是否接受参数保持沉默）。

有些程序员使用下面的函数头，并省略返回语句：

```
void main()
```

这在逻辑上是一致的，因为 void 返回类型意味着函数不返回任何值。该变体适用于很多系统，但由于它不是当前标准强制的一个选项，因此在有些系统上不能工作。因此，读者应避免使用这种格式，而应使用 C++标准格式，这不需要做太多的工作就能完成。

最后，ANSI/ISO C++标准对那些抱怨必须在 main()函数最后包含一条返回语句过于繁琐的人做出了让步。如果编译器到达 main()函数末尾时没有遇到返回语句，则认为 main()函数以如下语句结尾：

```
return 0;
```

这条隐含的返回语句只适用于 main()函数，而不适用于其他函数。

2. 为什么 main()不能使用其他名称

之所以将 myfirst.cpp 程序中的函数命名为 main()，原因是必须这样做。通常，C++程序必须包含一个名为 main()的函数（不是 Main()、MAIN()或 mane()。记住，大小写和拼写都要正确）。由于 myfirst.cpp 程序只有一个函数，因此该函数必须担负起 main()的责任。在运行 C++程序时，通常从 main()函数开始执行。因此，如果没有 main()，程序将不完整，编译器将指出未定义 main()函数。

存在一些例外情况。例如，在 Windows 编程中，可以编写一个动态链接库（DLL）模块，这是其他 Windows 程序可以使用的代码。由于 DLL 模块不是独立的程序，因此不需要 main()。用于专用环境的程序——如机器人中的控制器芯片——可能不需要 main()。但常规的独立程序都需要 main()，本书讨论的都是这种程序。

2.1.2　C++注释

C++注释以双斜杠（//）打头。注释是程序员为读者提供的说明，通常标识程序的一部分或解释代码的某个方面。编译器忽略注释，毕竟，它对 C++的了解至少和程序员一样，在任何情况下，它都不能理解注释。对编译器而言，程序清单 2.1 就像没有注释一样：

```
#include <iostream>

int main()
{
   using namespace std;
   cout <<"Come up and C++ me some time.";
```

```
    cout <<endl;
    cout<<"You won't regret it! " <<endl;
    return 0;
}
```

C++注释以（//）打头，到行尾结束。注释可以位于单独的一行上，也可以和代码位于同一行。请注意程序清单 2.1 的第一行：

```
// myfirst.cpp -- displays a message
```

本书中所有的程序都以注释开始，而注释指出了源代码的文件名并简要地总结了该程序。在第 1 章中介绍过，源代码的文件扩展名取决于所用的 C++系统。在其他系统中，文件名可能为 myfirst.C 或 myfirst.cxx。

提示：应使用注释来说明程序。程序越复杂，注释的价值越大。注释不仅有助于他人理解这些代码，也有助于程序员自己理解代码，特别是隔了一段时间没有接触程序的情况下。

C-风格注释

C++也能够识别 C 注释，C 注释包括在符号/*和*/之间：

```
#include <iostream> /* a C-style comment */
```

由于 C-风格注释以*/结束，而不是到行尾结束，因此可以跨越多行。可以在程序中使用 C 或 C++风格的注释，也可以同时使用这两种注释。但应尽量使用 C++注释，因为这不涉及到结尾符号与起始符号的正确配对，所以它产生问题的可能性很小。事实上，C99 标准也在 C 语言中添加了//注释。

2.1.3 C++预处理器和 iostream 文件

下面简要介绍一下需要知道的一些知识。如果程序要使用 C++输入或输出工具，请提供这样两行代码：

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

如果编译器不接受这几行代码（例如，它抱怨找不到文件 iostream），请将它们替换为下面的一行代码：

```
#include <iostream.h >   // compatible with older compilers
```

为使程序正常工作，只需要知道这些。下面更深入地介绍一下这些内容。

C++和 C 一样，也使用一种预处理器，该程序在进行主编译之前对源文件进行处理（第 1 章中介绍过，有些 C++实现使用翻译器程序将 C++程序转换为 C 程序。虽然翻译器也是一种预处理器，但这里不讨论这种预处理器，而只讨论处理名称以#开头的编译指令的预处理器）。不必执行任何特殊的操作来调用该预处理器，它会在编译程序时自动运行。

程序清单 2.1 使用了#include 编译指令：

```
#include <iostream>   // a PREPROCESSOR directive
```

该编译指令导致预处理器将 iostream 文件的内容添加到程序中。这是一种典型的预处理器操作：在源代码被编译之前，替换或添加文本。

这提出了一个问题：为什么要将 iostream 文件的内容添加到程序中呢？答案涉及到程序与外部世界之间的通信。iostream 中的 io 指的是输入（进入程序的信息）和输出（从程序中发送出去的信息）。C++的输入/输出方案涉及到 iostream 文件中的多个定义。为了使用 cout 来显示消息，第一个程序需要这些定义。#include 编译指令导致 iostream 文件的内容随源代码文件的内容一起被发送给编译器。实际上，iostream 文件的内容将取代程序中的代码行#include <iostream>。原始文件没有被修改，而是将源代码文件和 iostream 组合成一个复合文件，编译的下一阶段将使用该文件。

记住：使用 cin 和 cout 进行输入和输出的程序必须包含 iostream 文件（在有些系统中为 iostream.h）。

2.1.4 头文件名

像 iostream 这样的文件叫做包含文件（include file）——由于它们被包含在其他文件中；也叫头文件（header file）——由于它们被包含在文件起始处。C++编译器自带了很多头文件，每个头文件都支持一组特定的工具。C 语言的传统是，头文件使用扩展名 h，将其作为一种通过名称标识文件类型的简单方式。例如，math.h 头文件支持各种 C 语言数学函数。最初，C++也是这样做的。例如，支持输入和输出的头文

件叫做 iostream.h。不过，最近，C++的用法发生了变化。现在，对老式 C 的头文件保留了扩展名 h（C++程序仍可以使用这种文件），而 C++头文件则没有扩展名。有些 C 头文件被转换为 C++头文件，这些文件被重新命名，去掉了扩展名 h（使之成为 C++风格的名称），并在文件名称前面加上前缀 c（表明来自 C 语言）。例如，C++版本的 math.h 为 cmath 头文件。有时 C 头文件的 C 版本和 C++版本相同，而有时候新版本做了一些修改。对于纯粹的 C++头文件（如 iostream）来说，去掉 h 不只是形式上的变化，没有 h 的头文件也可以包含名称空间——本章的下一个主题。表 2.1 对头文件的命名约定进行了总结。

表 2.1　　头文件命名约定

头文件类型	约　定	范　例	说　明
C++旧式风格	以.h 结尾	iostream.h	C++程序可以使用
C 旧式风格	以.h 结尾	math.h	C、C++程序可以使用
C++新式风格	没有扩展名	iostream	C++程序可以使用，使用 namespace std
转换后的 C	加上前缀 c，没有扩展名	cmath	C++程序可以使用，可以使用不是 C 的特性，例如 namespace std

由于 C 使用不同的文件扩展名来表示不同文件类型，因此用一些特殊的扩展名（如.hx 或.hxx）表示 C++头文件是有道理的，ANSI/ISO 委员会也这样认为。问题在于究竟使用哪种扩展名，因此最终他们一致同意不使用任何扩展名。

2.1.5　名称空间

如果使用 iostream，而不是 iostream.h，则应使用下面的名称空间编译指令来使 iostream 中的定义对程序可用：

```
using namespace std;
```

这叫做 using 编译指令（directive）。最简单的办法是，现在接受这个编译指令，以后再考虑它（例如，到第 9 章再考虑它）。不过我们先介绍一下将发生的情况，以免让您感觉一头雾水。

名称空间支持是 C++中一项较新的特性，它是为了使编写将多个厂商已有的代码组合起来的程序更简单而设计的。一个潜在的问题是：可能使用两个已封装好的产品，而它们都包含一个名为 wanda()的函数。这样，使用 wanda()函数时，编译器将不知道指的是哪个版本。名称空间让厂商能够将其产品封装在一个叫做名称空间的单元中，这样就可以用名称空间的名称来指出想使用哪个厂商的产品。因此，Microflop Industries 可以将其定义放到一个名为 Microflop 的名称空间中。这样，其 wanda()函数的全称为 Microflop::wanda()；同样，Piscine 公司的 wanda()版本可以表示为 Piscine::wanda()。这样，程序就可以使用名称空间来区分不同的版本了：

```
Microflop::wanda ("go dancing?");      // use Microflop namespace version
Piscine::wanda ("a fish named Desire"); // use Piscine namespace version
```

按照这种方式，类、函数和变量便是 C++编译器的标准组件，它们现在都被放置在名称空间 std 中。仅当头文件没有扩展名 h 时，情况才是如此。这意味着在 iostream 中定义的、用于输出的 cout 变量实际上是 std::cout，而 endl 实际上是 std::endl。因此，可以省略 using 编译指令，以下述方式进行编码：

```
std::cout << "Come up and C++ me some time.";
std::cout << std::endl;
```

不过，多数用户并不喜欢将引入名称空间之前的代码（使用 iostream.h 和 cout）转换为名称空间代码（使用 iostream 和 std::cout），除非他们可以不费力地完成这种转换。于是，using 编译指令应运而生。下面的一行代码表明，可以使用 std 名称空间中定义的名称，而不必使用 std::前缀：

```
using namespace std;
```

这个 using 编译指令使得 std 名称空间中的所有名称都可用。这是一种偷懒的做法。更好的方法是，只使所需的名称可用，这可以通过使用 using 声明来实现：

```
using std::cout;  // make cout available
using std::endl;   //make endl available
```

```
using std::cin;   // make cin available
```

用这些编译指令替换下述代码后：

```
using namespace std;  // lazy approach, all names available
```

便可以使用 cin 和 cout 了，而不必加上 std::前缀。不过，要使用 iostream 中的其他名称，必须将它们分别加到 using 列表中。本书首先采用这种偷懒的方法，其原因有两个。首先，对于简单程序而言，采用何种名称空间管理方法无关紧要；其次，本书的重点是介绍C++的基本方面。本书后面将采用其他名称空间管理技术。

2.1.6 使用 cout 进行 C++输出

现在来看一看如何显示消息。myfirst.cpp 程序使用下面的 C++语句：

```
cout << "Come up and C++ me some time. ";
```

双引号括起的部分是要打印的消息。在 C++中，用双引号括起的一系列字符叫做字符串，因为它是由若干字符组合而成的。<<符号表示该语句将把这个字符串发送给 cout；该符号指出了信息流动的路径。cout 是什么呢？它是一个预定义的对象，知道如何显示字符串、数字和单个字符等（第 1 章介绍过，对象是类的特定实例，而类定义了数据的存储和使用方式）。

马上就使用对象可能有些困难，因为几章后才会介绍对象。实际上，这演示了对象的长处之一——不用了解对象的内部情况，就可以使用它。只需要知道它的接口，即可以使用它。cout 对象有一个简单的接口，如果 string 是一个字符串，则下面的代码将显示该字符串：

```
cout << string;
```

对于显示字符串而言，只需知道这些即可。不过，现在我们要看一看 C++从概念上如何解释这个过程。从概念上看，输出是一个流，即从程序流出的一系列字符。cout 对象表示这种流，其属性是在 iostream 文件中定义的。cout 的对象属性包括一个插入操作符（<<），它可以将其右侧的信息插入到流中。因此下面的语句（注意结尾的分号）：

```
cout << "Come up and C++ me some time. ";
```

将把字符串“Come up and C++ me some time.”插入到输出流中。因此，与其说程序显示了一条消息，不如说它将一个字符串插入到了输出流中。不知道为什么，后者听起来更好一点（参见图 2-2）。

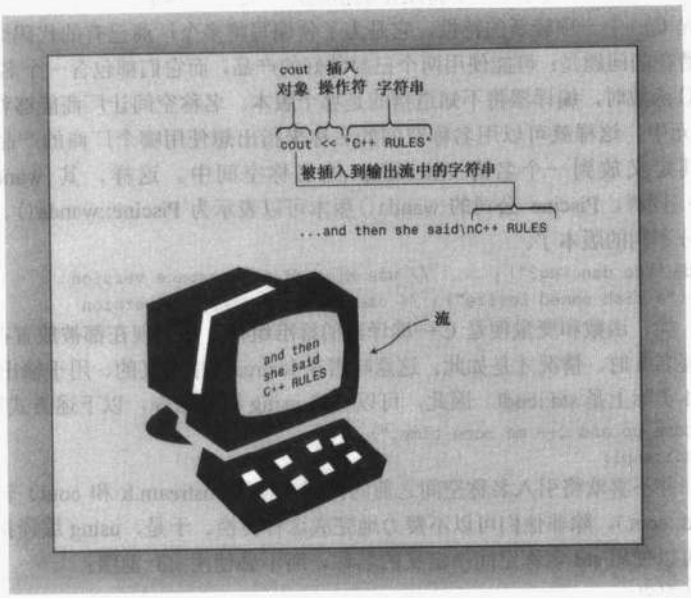


图 2.2 使用 cout 显示字符串

初识操作符重载

如果熟悉 C 后才开始学习 C++，则可能注意到了，插入操作符（<<）看上去就像按位左移操作符（<<）。

这是一个操作符重载的例子，通过重载，同一个操作符将有不同的含义。编译器通过上下文来确定操作符的含义。C 本身也有一些操作符重载的情况，例如，&符号既表示地址操作符，又表示按位 AND 操作符；*符号既表示乘法，又表示对指针解除引用。这里重要的不是这些操作符的具体功能，而是同一个符号可以有多种含义，而编译器可以根据上下文来确定其含义（这和确定“sound card”中的“sound”与“sound financial basic”中的“sound”的含义是一样的）。C++扩展了操作符重载的概念，允许为用户定义的类型（类）重新定义操作符的含义。

1. 控制符 endl

现在来看看程序清单 2.1 中第二个输出流中看起来有些古怪的符号：

```
cout << endl;
```

endl 是一个特殊的 C++符号，表示一个重要的概念：重起一行。在输出流中插入 endl 将导致屏幕光标移到下一行的开头。诸如 endl 等对于 cout 来说有特殊函数的特殊符号被称为控制符（manipulator）。和 cout 一样，endl 也是在头文件 iostream 中定义的，且位于名称空间 std 中。

打印字符串时，cout 不会自动移到下一行，因此在程序清单 2.1 中，第一条 cout 语句将光标留在输出字符串的后面。每条 cout 语句的输出从前一个输出的末尾开始，因此如果省略程序清单 2.1 中的 endl，得到的输出将如下：

```
Come up and C++ me some time.You won' t regret it!
```

从上述输出可知，Y 紧跟在句点的后面。下面来看另一个例子，假设有如下代码：

```
cout << "The Good, the";
cout << "Bad, " ;
cout << "and the Ukulele";
cout << endl;
```

其输出将如下：

```
The Good, theBad, and the Ukulele
```

同样，每个字符串紧接在前一个字符串的后面。如果要在两个字符串之间留一个空格，必须将空格包含在字符串中。注意，要尝试上述输出范例，必须将代码放到完整的程序中，该程序包含一个 main()函数头以及起始和结束花括号。

2. 换行符

C++还提供了另一种在输出中指示换行的旧式方法：C 语言符号\n：

```
cout << "What' s next?\n"; // \n means start a new line
```

\n 被视为一个字符，名为换行符。

显示字符串时，在字符串中包含换行符，而不是在末尾加上 endl，可减少输入量：

```
cout << "Jupiter is a large planet.\n"; // displays sentence, goes to next line
cout << "Jupiter is a large planet. " << endl; // displays sentence, goes to next line
```

另一方面，如果要生成一个空行，则两种方法的输入量相同，但对大多数人而言，输入 endl 更为方便：

```
cout << "\n"; // start a new line
cout << endl; // start a new line
```

本书中显示用引号括起的字符串时，通常使用换行符\n，在其他情况下则使用控制符 endl。

换行符是一种被称为“转义序列”的按键组合，有关“转义序列”将在第 3 章做更详细的讨论。

2.1.7　C++源代码的格式化

有些语言（如 FORTRAN）是面向行的语言，即每条语句占一行。对于这些语言来说，回车的作用是将语句分开。不过，在 C++中，分号标示了语句的结尾。因此，在 C++中，回车的作用就和空格或制表符相同。也就是说，在 C++中，通常可以在能够使用回车的地方使用空格，反之亦然。这说明既可以把一条语句放在几行上，也可以把几条语句放在同一行上。例如，可以将 myfirst.cpp 重新格式化为如下所示：

```
#include <iostream>
    int
main
```

```
(){  using
   namespace
       std; cout
         <<
"Come up and C++ me some time. "
;    cout <<
endl; cout<<
"You won't regret it!" <<
endl; return 0; }
```

这样虽然不太好看，但仍然是合法的代码。必须遵守一些规则，具体地说，在C和C++中，不能把空格、制表符或回车放在元素（比如名称）中间，也不能把回车放在字符串中间。下面是一个不能这样做的例子：

```
int ma  in()    // INVALID -- space in name
re
turn 0; // INVALID -- carriage return in word
cout << "Behold the Beans
  of Beauty! "; // INVALID -- carriage return in string
```

1. 标记和空白

一行代码中不可分割的元素叫做标记（token，参见图2.3）。通常，必须用空格、制表符或回车将两个标记分开，空格、制表符和回车统称为空白（white space）。有些字符（如括号和逗号）是不需要用空白分开的标记。下面的一些范例说明了什么情况下可以使用空白，什么情况下可以省略：

```
return0;              // INVALID, must be return 0;
return (0);           // VALID, white space omitted
return (0) ;          // VALID, white space used
intmain();            // INVALID, white space omitted
int main()            // VALID, white space omitted
int main()            // ALSO VALID, white space used
```

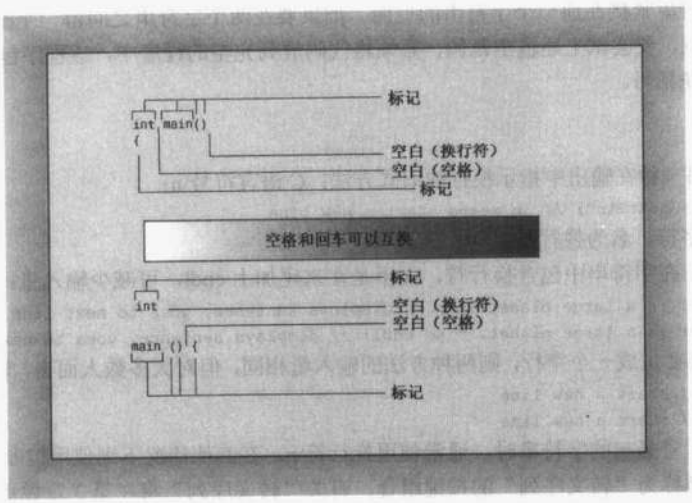


图2.3 标记和空白

2. C++源代码风格

虽然C++在格式方面赋予了人们很大的自由，但如果遵循一些切合实际的风格，程序将更便于阅读。有效但难看的代码不会令人满意。多数程序员都使用程序清单2.1的风格，它遵循了以下规则：

- 每行一条语句。
- 每个函数都有一个开始花括号和一个结束花括号，这两个花括号各占一行。
- 函数中的语句都相对于花括号进行缩进。
- 与函数名称相关的圆括号周围没有空白。

前 3 条规则旨在确保代码清晰易读；第 4 条规则帮助区分函数和一些也使用圆括号的 C++内置结构（如循环）。在涉及其他指导原则时，本书将提醒读者。

2.2　C++语句

C++程序是一组函数，而每个函数又是一组语句。C++有好几种语句，下面介绍其中的一些。程序清单 2.2 提供了两种新的语句。声明语句创建变量，赋值语句给该变量提供一个值。另外，该程序还演示了 cout 的新功能。

程序清单 2.2　carrot.cpp

```
// carrots.cpp -- food processing program
// uses and displays a variable

#include <iostream>

int main()
{
   using namespace std;

   int carrots;          // declare an integer variable

   carrots = 25;          // assign a value to the variable
   cout << "I have";
   cout << carrots;       // display the value of the variable
   cout << " carrots.";
   cout << endl;
   carrots = carrots - 1;  // modify the variable
   cout << "Crunch, crunch. Now I have" << carrots << " carrots. " << endl;
   return 0;
}
```

空行将声明语句与程序的其他部分分开。这是 C 常用的方法，但在 C++中不那么常见。下面是该程序的输出：

```
I have 25 carrots.
Crunch, crunch. Now I have 24 carrots.
```

下面探讨这个程序。

2.2.1　声明语句和变量

计算机是一种精确的、有条理的机器。要将信息项存储在计算机中，必须指出信息的存储位置和所需的内存空间。在 C++中，完成这种任务的一种相对简便的方法，是使用声明语句来指出存储类型并提供位置标签。例如，程序清单 2.2 中包含这样一条声明语句（注意其中的分号）：

```
int carrots;
```

这条语句使程序使用足够的存储空间来存储一个整数，在 C++中用 int 表示整数。编译器负责分配和标记内存的细节。C++可以处理多种类型的数据，而 int 是最基本的数据类型。它表示整数——没有小数部分的数字。C++的 int 类型可以为正，也可以为负，但是大小范围取决于实现。第 3 章将详细介绍 int 和其他基本类型。

除了指出类型外，该声明语句还指出，此后程序将使用名称 carrots 来标识存储在该内存单元中的值。carrots 被称为变量，因为它的值可以修改。在 C++中，所有变量都必须声明。如果省略了 carrots.cpp 中的声明，则当程序试图使用 carrots 时，编译器将指出错误。事实上，程序员尝试省略声明，可能只是为了看看编译器的反应。这样，以后看到这样的反应时，便知道应检查是否省略了声明。

为什么变量必须声明

有些语言（最典型的是 BASIC）在使用新名称时创建新的变量，而不用显式地进行声明。这看上去对用户比较友好，事实上从短期上说确实如此。问题是，如果错误地拼写了变量名，将在不知情的情况下创

建一个新的变量。在 BASIC 中，程序员可能会写出如下的语句：

```
CastleDark = 34
...
CastleDank = CastleDark + MoreGhosts
...
PRINT CastleDark
```

由于 CastleDank 是拼写错误（将 r 拼成了 n），因此所作的修改实际上并没有修改 CastleDark。这种错误很难发现，因为它没有违反 BASIC 中的任何规则。然而，在 C++中，将声明 CastleDark，但不会声明被错误拼写的 CastleDank，因此对应的 C++代码将违反“使用变量前必须声明它”的规则，因此编译器将捕获这种错误，发现潜在的问题。

因此，声明通常指出了要存储的数据类型和程序对存储在这里的数据使用的名称。在这个例子中，程序将创建一个名为 carrots 的变量，它可以存储整数（参见图 2.4）。

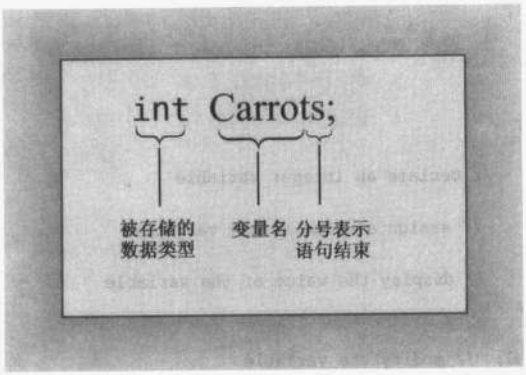


图 2.4 变量声明

程序中的声明语句叫做定义声明（defining declaration）语句，简称为定义（definition）。这意味着它将导致编译器为变量分配内存空间。在较为复杂的情况下，还可能有引用声明（reference declaration）。这些声明命令计算机使用在其他地方定义的变量。通常，声明不一定是定义，但在这个例子中，声明是定义。

如果熟悉 C 语言或 Pascal，则一定对变量声明比较熟悉。不过 C++中的变量声明也可能让人小吃一惊。在 C 和 Pascal 中，所有的变量声明通常都位于函数或过程的开始位置，但 C++没有这种限制。实际上，C++通常的做法是，在首次使用变量前声明它。这样，就不必在程序中到处查找，以了解变量的类型。本章后面将有一个这样的例子。这种风格也有缺点，它没有把所有的变量名放在一起，因此无法对函数使用了哪些变量一目了然（C99 标准使 C 声明规则与 C++非常相似）。

提示：对于声明变量，C++的做法是尽可能在首次使用变量前声明它。

2.2.2 赋值语句

赋值语句将值赋给存储单元。例如，下面的语句：

```
carrots = 25;
```

将整数 25 赋给变量 carrots 表示的内存单元。符号=叫做赋值操作符。C++（和 C）有一项不寻常的特性——可以连续使用赋值操作符。例如，下面的代码是合法的：

```
int steinway;
int baldwin;
int yamaha;
yamaha = baldwin = steinway = 88;
```

赋值将从右向左进行。首先，88 被赋给 steinway；然后，steinway 的值（现在是 88）被赋给 baldwin；然后 baldwin 的值 88 被赋给 yamaha（C++遵循 C 的爱好，允许外观奇怪的代码）。

程序清单 2.2 中的第二条赋值语句表明，可以对变量的值进行修改：

```
carrots = carrots - 1; // modify the variable
```

赋值操作符右边的表达式 carrots - 1 是一个算术表达式。计算机将变量 carrots 的值 25 减去 1，得到 24。然后，赋值操作符将这个新值存储到变量 carrots 对应的内存单元中。

2.2.3　cout 的新花样

到目前为止，本章的范例都使用 cout 来打印字符串，程序清单 2.2 使用 cout 来打印变量，该变量的值是一个整数：

```
cout << carrots;
```

程序没有打印“carrots”，而是打印存储在 carrots 中的整数值，即 25。实际上，这将两个操作合而为一了。首先，cout 将 carrots 替换为其当前值 25；然后，把值转换为合适的输出字符。

如上所示，cout 可用于数字和字符串。这似乎没有什么不同寻常的地方，但别忘了，整数 25 与字符串“25”有天壤之别。该字符串存储的是书写该数字时使用的字符，即字符 2 和 5。程序在内部存储的是字符 2 和字符 5 的编码。要打印字符串，cout 只需打印字符串中各个字符即可。但整数 25 被存储为数值，计算机不是单独存储每个数字，而是将 25 存储为二进制数（附录 A 讨论了这种表示法）。这里的要点是，在打印之前，cout 必须将整数形式的数字转换为字符串形式。另外，cout 很聪明，知道 carrots 是一个需要转换的整数。

与老式 C 语言的区别在于 cout 的聪明程度。在 C 语言中，要打印字符串“25”和整数 25，可以使用 C 的多功能输出函数 printf()：

```
printf ("Printing a string: %s\n", "25");
printf ("Printing an integer: %d\n", 25);
```

撇开 printf()的复杂性不说，必须用特殊代码（%s 和%d）来指出是要打印字符串还是整数。如果让 printf()打印字符串，但又错误地提供了一个整数，由于 printf()不够精密，因此根本发现不了错误。它将继续处理，显示一堆乱码。

cout 的智能行为源自 C++的面向对象特性。实际上，C++插入操作符（<<）将根据其后的数据类型相应地调整其行为，这是一个操作符重载的例子。在后面的章节中学习函数重载和操作符重载时，将知道如何实现这种智能设计。

cout 和 printf()

如果已经习惯了 C 语言和 printf()，可能觉得 cout 看起来很奇怪。程序员甚至可能固执地坚持使用 printf()。但与使用所有转换说明的 printf()相比，cout 的外观一点也不奇怪。更重要的是，cout 还有明显的优点。它能够识别类型的功能表明，其设计更灵活、更好用。另外，它是可扩展的（extensible）。也就是说，可以重新定义<<操作符，使 cout 能够识别和显示所开发的新数据类型。如果喜欢 printf()提供的细致的控制功能，可以使用更高级的 cout 来获得相同的效果（参见第 17 章）。

2.3　其他 C++语句

再来看几个 C++语句的例子。程序清单 2.3 中的程序对前一个程序进行了扩展，要求在程序运行时输入一个值。为实现这项任务，它使用了 cin，这是与 cout 对应的用于输入的对象。另外，该程序还演示了 cout 对象的多功能性。

程序清单 2.3　getinfo.cpp

```
// getinfo.cpp -- input and output
#include <iostream>

int main()
{
   using namespace std;
```

```
    int carrots;

    cout << "How many carrots do you have?" << endl;
    cin >> carrots;              // C++ input
    cout << "Here are two more. ";
    carrots = carrots + 2;
// the next line concatenates output
    cout << "Now you have " << carrots << " carrots." << endl;
    return 0;
}
```

下面是该程序的运行情况：

```
How many carrots do you have?
12
Here are two more. Now you have 14 carrots.
```

该程序包含两项新特性：用 cin 来读取键盘输入以及将 4 条输出语句组合成一条。下面分别介绍它们。

2.3.1 使用 cin

上面的输出表明，从键盘输入的值（12）最终被赋给变量 carrots。下面就是执行这项功能的语句：

```
cin >> carrots;
```

从这条语句可知，信息从 cin 流向 carrots。显然，对这一过程有更为正式的描述。就像 C++将输出看作是流出程序的字符流一样，它也将输入看作是流入程序的字符流。iostream 文件将 cin 定义为一个表示这种流的对象。输出时，<<操作符将字符串插入到输出流中；输入时，cin 使用>>操作符从输入流中抽取字符。通常，需要在操作符右侧提供一个变量，以接收抽取的信息（符号<<和>>被选择用来指示信息流的方向）。

与 cout 一样，cin 也是一个智能对象。它可以将通过键盘输入的一系列字符（即输入）转换为接收信息的变量能够接受的形式。在这个例子中，程序将 carrots 声明为一个整型变量，因此输入被转换为计算机用来存储整数的数字形式。

2.3.2 使用 cout 进行拼接

getinfo.cpp 中的另一项新特性是将 4 条输出语句合并为一条。iostream 文件定义了<<操作符，以便可以像下面这样合并（拼接）输出：

```
cout << "Now you have " << carrots << " carrots." << endl;
```

这样能够将字符串输出和整数输出合并为一条语句。得到的输出与下述代码生成的相似：

```
cout << "Now you have ";
cout << carrots;
cout << " carrots";
cout << endl;
```

根据有关 cout 的建议，也可以按照这样的方式重写拼接版本，即将一条语句放在 4 行上：

```
cout << "Now you have "
    << carrots
    << " carrots."
    << endl;
```

这是由于 C++的自由格式规则将标记间的换行符和空格看作是可相互替换的。当代码行很长，限制输出的显示风格时，最后一种技术很方便。

需要注意的另一点是：

```
Now you have 14 carrots.
```

和

```
Here are two more.
```

在同一行中。

这是因为前面指出过的，cout 语句的输出紧跟在前一条 cout 语句的输出后面。即使两条 cout 语句之间有其他语句，情况也将如此。

2.3.3　类简介

看了足够多的 cin 和 cout 范例后，可以学习有关对象的知识了。具体地说，本节将进一步介绍有关类的知识。正如第 1 章指出的，类是 C++中面向对象编程（OOP）的核心概念之一。

类是用户定义的一种数据类型。要定义类，需要描述它能够表示什么信息和可对数据执行哪些操作。类之于对象就像类型之于变量。也就是说，类定义描述的是数据格式及其用法，而对象则是根据数据格式规范创建的实体。换句话说，如果说类就好比所有著名演员，则对象就好比某个著名的演员，如蛙人 Kermit。我们来扩展这种类比，表示演员的类中包括该类可执行的操作的定义，如念某一角色的台词，表达悲伤、威胁恫吓、接受奖励等。如果了解其他 OOP 术语，就知道 C++类对应于某些语言中的对象类型，而 C++对象对应于对象实例或实例变量。

下面更具体一些。前文讲述过下面的变量声明：

```
int carrots;
```

上面的代码将创建一个类型为 int 的变量（carrots）。也就是说，carrots 可以存储整数，可以按特定的方式使用——例如，用于加和减。现在来看 cout。它是一个 ostream 类对象。ostream 类定义（iostream 文件的另一个成员）描述了 ostream 对象表示的数据以及可以对它执行的操作，如将数字或字符串插入到输出流中。同样，cin 是一个 istream 类对象，也是在 iostream 中定义的。

记住：类描述了一种数据类型的全部属性，对象是根据这些描述创建的实体。

知道类是用户定义的类型，但作为用户，并没有设计 ostream 和 istream 类。就像函数可以来自函数库一样，类也可以来自类库。ostream 和 istream 类就属于这种情况。从技术上说，它们没有被内置到 C++语言中，而是该语言自带的类的范例。这些类定义位于 iostream 文件中，没有被内置到编译器中。如果愿意，程序员甚至可以修改这些类定义，虽然这不是一个好主意（准确地说，这个主意很糟）。iostream 系列类和相关的 fstream（或文件 I/O）系列类是早期所有的实现都自带的惟一两组类定义。不过，ANSI/ISO C++委员会在 C++标准中添加了其他一些类库。另外，多数实现都在软件包中提供了其他类定义。事实上，C++当前之所以如此有吸引力，很大程度上是由于存在大量支持 UNIX、Macintosh 和 Windows 编程的类库。

类描述指定了可对类对象执行的所有操作。要对特定对象执行这些允许的操作，需要给该对象发送一条消息。例如，如果希望 cout 对象显示一个字符串，应向它发送一条消息，告诉它，“对象！显示这些内容！”C++提供了两种发送消息的方式：一种方式是使用类方法（本质上就是稍后将介绍的函数调用）；另一种方式是重新定义操作符，cin 和 cout 采用的就是这种方式。因此，下面的语句：

```
cout << "I am not a crook. "
```

使用重新定义的<<操作符将“显示消息”发送给 cout。在这个例子中，消息带一个参数——要显示的字符串（参见图 2.5）。

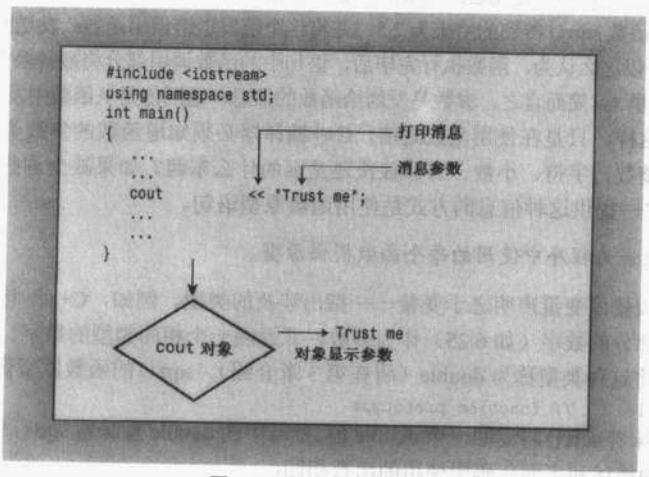


图 2.5　向对象发送消息

2.4 函数

由于函数用于创建 C++程序的模块，对 C++的 OOP 定义至关重要，因此必须熟悉它。函数的某些方面属于高级主题，将在第 7 章和第 8 章重点讨论函数。不过，现在了解函数的一些基本特征，将使得在以后的函数学习中更加得心应手。本章剩余的内容将介绍函数的一些基本知识。

C++函数分两种：有返回值的和没有返回值的。在标准 C++函数库中可以找到这两类函数的例子，您也可以自己创建这两种类型的函数。下面首先来看一个有返回值的库函数，然后介绍如何编写简单的函数。

2.4.1 使用有返回值的函数

有返回值的函数将生成一个可被赋给变量的值。例如，标准 C/C++库包含一个名为 sqrt()的函数，它返回平方根。假设要计算 6.25 的平方根，并将这个值赋给变量 *x*，则可以在程序中使用下面的语句：

```
x = sqrt(6.25); // returns the value 2.5 and assigns it to x
```

表达式 sqrt(6.25)将调用 sqrt()函数。表达式 sqrt(6.25)被称为函数调用，被调用的函数叫做被调用函数（called function），包含函数调用的函数叫做调用函数（calling function，参见图 2.6）。

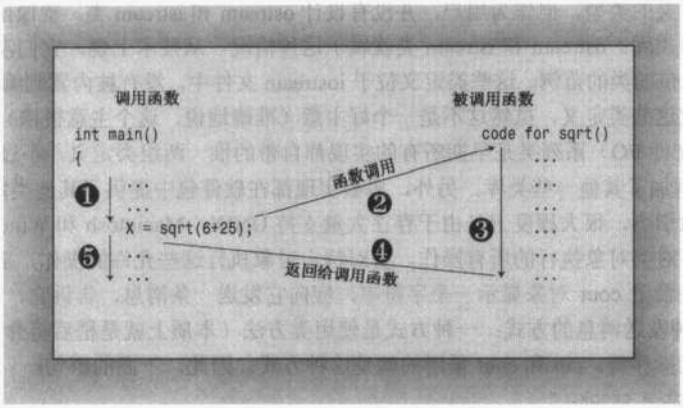


图 2.6 调用函数

圆括号中的值（这里为 6.25）是发送给函数的信息，这被称为传递给函数。以这种方式发送给函数的值叫做参数（参见图 2.7）。函数 sqrt()得到的结果为 2.5，并将这个值发送给调用函数；发送回去的值叫做函数的返回值（return value）。可以这么认为，函数执行完毕后，语句中的函数调用部分将被替换为返回的值。因此，这个例子将返回值赋给变量 *x*。简而言之，参数是发送给函数的信息，返回值是从函数中发送回去的值。

情况基本上就是这样，只是在使用函数之前，C++编译器必须知道函数的参数类型和返回值类型。也就是说，函数是返回整数、字符、小数、有罪裁决还是别的什么东西？如果缺少这些信息，编译器将不知道如何解释返回值。C++提供这种信息的方式是使用函数原型语句。

记住：C++程序应当为程序中使用的每个函数提供原型。

函数原型之于函数就像变量声明之于变量——指出涉及的类型。例如，C++库将 sqrt()函数定义成将一个（可能）带小数部分的数字（如 6.25）作为参数，并返回一个相同类型的数字。有些语言将这种数字称为实数，但是 C++将这种类型称为 double（将在第 3 章介绍）。sqrt()的函数原型像这样：

```
double sqrt(double);  // function prototype
```

第一个 double 意味着 sqrt()将返回一个 double 值。括号中的 double 意味着 sqrt()需要一个 double 参数。因此该原型对 sqrt()的描述和下面代码中使用的函数相同：

```
double x;       // declare x as a type double variable
```

```
x = sqrt(6.25);
```

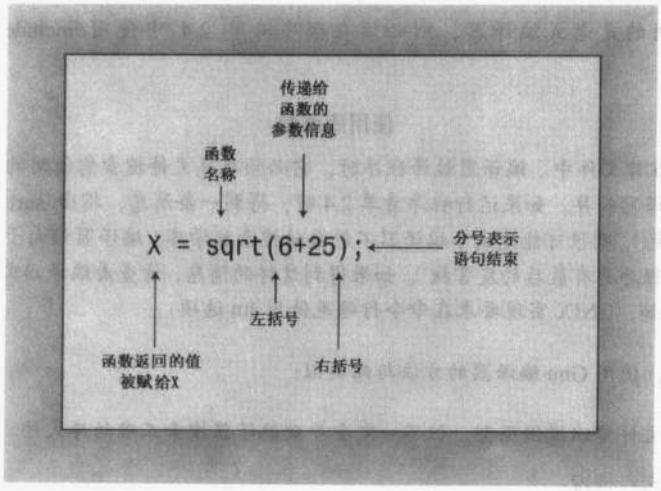


图 2.7　函数调用的句法

原型结尾的分号表明它是一条语句，这使得它是一个原型，而不是函数头。如果省略分号，编译器将把这行代码解释为一个函数头，并要求接着提供定义该函数的函数体。

在程序中使用 sqrt()时，也必须提供原型。可以用两种方法来实现：

- 在源代码文件中输入函数原型。
- 包含头文件 cmath（老系统为 math.h），其中定义了原型。

第二种方法更好，因为头文件更有可能使原型正确。对于 C++库中的每个函数，都在一个或多个头文件中提供了其原型。请通过手册或在线帮助查看函数描述来确定应使用哪个头文件。例如，sqrt()函数的说明将指出，应使用 cmath 头文件（同样，可能必须使用老式的头文件 math.h，它可用于 C 和 C++程序中）。

不要混淆函数原型和函数定义。可以看出，原型只描述函数接口。也就是说，它描述的是发送给函数的信息和返回的信息。而定义中包含了函数的代码，如计算平方根的代码。C 和 C++将库函数的这两项特性（原型和定义）分开了。库文件中包含了函数的编译代码，而头文件中则包含了原型。

应在首次使用函数之前提供其原型。通常的做法是把原型放到 main()函数定义的前面。程序清单 2.4 演示了库函数 sqrt()的用法，它通过包含 cmath 文件来提供该函数的原型：

程序清单 2.4　sqrt.cpp

```
// sqrt.cpp -- using the sqrt() function

#include <iostream>
#include <cmath>    // or math.h

int main()
{
   using namespace std;

   double area;
   cout << "Enter the floor area, in square feet, of your home: ";
   cin >> area;
   double side;
   side = sqrt(area);
   cout << "That's the equivalent of a square
      << " feet to the side." << endl;
   cout << "How fascinating!" << endl;
```

```
    return 0;
}
```

注意：如果使用的是老式编译器，则必须在程序清单 2.4 中使用#include <math.h>，而不是#include<cmath>。

使用库函数

C++库函数存储在库文件中。编译器编译程序时，它必须在库文件搜索您使用的函数。至于自动搜索哪些库文件，将因编译器而异。如果运行程序清单 2.4 时，得到一条消息，指出_sqrt 是一个没有定义的外部函数（似乎应当避免），则很可能是由于编译器不能自动搜索数学库（编译器倾向于给函数名添加下划线前缀——提示它们对程序具有最后的发言权）。如果得到这样的消息，请查看编译器文档，了解如何让编译器搜索正确的库。例如，UNIX 实现要求在命令行结尾使用-lm 选项：

```
CC sqrt.C -lm
```

在 Linux 系统中，使用 Gnu 编译器的方法与此类似：

```
g++ sqrt.C -lm
```

只包含 cmath 头文件可以提供原型，但不一定会导致编译器搜索正确的库文件。

下面是该程序的运行情况：

```
Enter the floor area, in square feet, of your home: 1536
That's the equivalent of a square 39.1918 feet to the side.
How fascinating!
```

由于 sqrt()处理的是 double 值，因此这里将变量声明为这种类型。声明 double 变量的句法与声明 int 变量相同：

```
type-name variable-name;
```

double 类型使得变量 area 和 side 能够存储带小数的值，如 536.0 和 39.1918。将看起来是整数（如 536）的值赋给 double 变量时，将以实数形式存储它，其中的小数部分为.0。在第 3 章将指出，double 类型覆盖的范围要比 int 类型大得多。

C++允许在程序的任何地方声明新变量，因此 sqrt.cpp 在要使用 side 时才声明它。C++还允许在创建变量时对它进行赋值，因此也可以这样做：

```
double side = sqrt(area);
```

这个过程叫做初始化（initialization），将在第 3 章介绍。

cin 知道如何将输入流中的信息转换为 double 类型，cout 知道如何将 double 类型插入到输出流中。前面讲过，这些对象都很智能化。

2.4.2 函数变体

有些函数需要多项信息。这些函数使用多个参数，参数间用逗号分开。例如，数学函数 pow()接受两个参数，返回值为以第一个参数为底，第二个参数为指数的幂。该函数的原型如下：

```
double pow(double, double);  // prototype of a function with two arguments
```

要计算 5 的 8 次方，可以这样使用该函数：

```
answer = pow(5.0, 8.0);      // function call with a list of arguments
```

另外一些函数不接受任何参数。例如，有一个 C 库（与 cstdlib 或 stdlib.h 头文件相关的库）包含一个 rand()函数，该函数不接受任何参数，并返回一个随机整数。该函数的原型如下：

```
int rand(void);        // prototype of a function that takes no arguments
```

关键字 void 明确指出，该函数不接受任何参数。如果省略 void，让括号为空，则 C++将其解释为一个不接受任何参数的隐式声明。可以这样使用该函数：

```
myGuess = rand();      // function call with no arguments
```

注意，与其他一些计算机语言不同，在 C++中，函数调用中必须包括括号，即使没有参数。

还有一些函数没有返回值。例如，假设编写了一个函数，它按美元、美分格式显示数字。当向它传递参数 23.5 时，它将在屏幕上显示$23.50。由于这个函数把值发送给屏幕，而不是调用程序，因此它不会返

回任何值。可以在原型中使用关键字 void 来指定返回类型，以指出函数没有返回值：

```
void bucks(double);  // prototype for function with no return value
```

由于它不返回值，因此不能将该函数调用放在赋值语句或其他表达式中。相反，应使用一条纯粹的函数调用语句：

```
bucks(1234.56);      // function call, no return value
```

在有些语言中，有返回值的函数被称为函数（function）；没有返回值的函数被称为过程（procedure）或子程序（subroutine）。但 C++与 C 一样，这两种变体都被称为函数。

2.4.3　用户定义的函数

标准 C 库提供了 140 多个预定义的函数。如果其中的函数能满足要求，则应使用它们。但用户经常需要编写自己的函数，尤其是在设计类的时候。无论如何，设计自己的函数很有意思，下面来介绍这一过程。前面已经使用过好几个用户定义的函数，它们都叫 main()。每个 C++程序都必须有一个 main()函数，用户必须对它进行定义。假设需要添加另一个用户定义的函数。和库函数一样，也可以通过函数名来调用用户定义的函数。对于库函数，在使用之前必须提供其原型，通常把原型放到 main()定义之前。新的内容是，还必须提供新函数的源代码。最简单的方法是，将代码放在 main()的后面。程序清单 2.5 演示了这些元素。

程序清单 2.5　ourfunc.cpp

```
// ourfunc.cpp -- defining your own function
#include <iostream>
void simon(int);    // function prototype for simon()

int main()
{
   using namespace std;
   simon(3);       // call the simon() function
   cout << "Pick an integer: ";
   int count;
   cin >> count;
   simon(count);   // call it again
   cout << "Done!" << endl;
   return 0;
}

void simon(int n)   // define the simon() function
{
   using namespace std;
   cout << "Simon says touch your toes " << n << " times. " << endl;
}                  // void functions don't need return statement
```

main()函数两次调用 simon()函数，一次的参数为 3，另一次的参数为变量 count。在这两次调用之间，用户输入一个整数，用来设置 count 的值。这个例子没有在 cout 提示消息中使用换行符。这样将导致用户输入与提示出现在同一行中。下面是运行情况：

```
Simon says touch your toes 3 times.
Pick an integer: 512
Simon says touch your toes 512 times.
Done!
```

1. 函数格式

在程序清单 2.5 中，simon()函数的定义与 main()的定义采用的格式相同。首先，有一个函数头；然后是花括号中的函数体。可以把函数的格式统一为如下的情形：

```
type functionname(argumentlist)
{
   statements
}
```

注意，定义 simon() 的源代码位于 main() 的后面。和 C 一样（但不同于 Pascal），C++不允许将函数定义嵌套在另一个函数定义中。每个函数定义都是独立的，所有函数的创建都是平等的（参见图 2.8）。

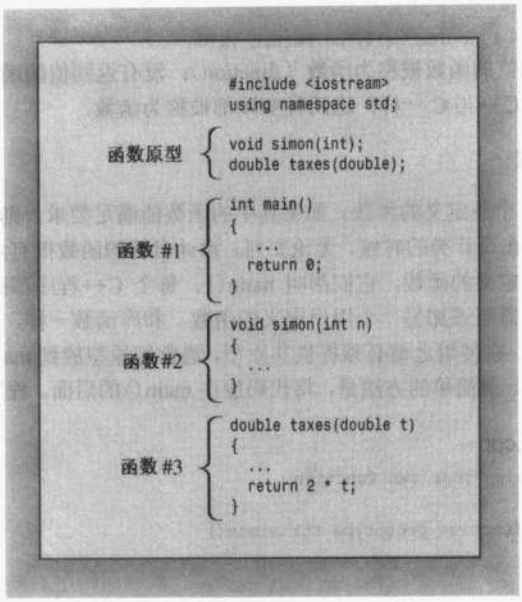


图 2.8 函数定义在文件中依次出现

2. 函数头

在程序清单 2.5 中，simon() 函数的函数头如下：

```
void simon (int n)
```

开头的 void 表明 simon() 没有返回值，因此调用 simon() 不会生成可在 main() 中将其赋给变量的数字。因此，第一个函数调用方式如下：

```
simon (3);          // ok for void functions
```

由于 simon() 没有返回值，因此不能这样使用它：

```
simple = simon (3);  // not allowed for void functions
```

括号中的 int n 表明，使用 simon() 时，应提供一个 int 参数。*n* 是一个新的变量，函数调用时传递的值将被赋给它。因此，函数调用：

```
simon (3);
```

将 3 赋给 simon() 函数头中定义的变量 *n*。当函数体中的 cout 语句使用 *n* 时，将使用函数调用时传递的值。这就是为什么 simon（3）在输出中显示 3 的原因所在。在范例运行中，函数调用 simon（count）导致函数显示 512，因为这正是赋给 count 的值。简而言之，simon() 的函数头表明，该函数接受一个 int 参数，不返回任何值。

我们复习一下 main() 的函数头：

```
int main()
```

开头的 int 表明，main() 返回一个整数值；空括号（其中可以包含 void）表明，main() 没有参数。对于有返回值的函数，应使用关键字 return 来提供返回值，并结束函数。这就是为什么要在 main() 结尾使用下述语句的原因：

```
return 0;
```

这在逻辑上是一致的：main() 返回一个 int 值，而程序员要求它返回整数 0。但可能会产生疑问，将这个值返回到哪里了呢？毕竟，程序中没有哪个地方可以看出对 main() 的调用：

```
squeeze = main();  // absent from our programs
```

答案是，可以将计算机的操作系统（如 UNIX 或 DOS）看作调用程序。因此，main() 的返回值并不是返回给程序的其他部分，而是被返回给操作系统。很多操作系统都可以使用程序的返回值。例如，UNIX 外壳脚本和 DOS 批处理文件都被设计成运行程序，并测试它们的返回值（通常叫做退出值）。通常的约定是，退出值为 0 则意味着程序运行成功，为非零则意味着存在问题。因此，如果 C++程序无法打开文件，可以将它设计为返回一个非零值。然后，便可以设计一个外壳脚本或批处理文件来运行该程序，如果该程序发出指示失败的消息，则采取其他措施。

关键字

关键字是计算机语言中的词汇。本章使用了 4 个 C++关键字：int、void、return 和 double。由于这些关键字都是 C++专用的，因此不能用作他用。也就是说，不能将 return 用作变量名，也不能把 double 用作函数名。不过可以把它们用作名称的一部分，如 painter（其中包含 int）或 return_aces。附录 B 提供了 C++关键字的完整列表。另外，main 不是关键字，由于它不是语言的组成部分。不过，它是一个必不可少的函数的名称。可以把 main 用作变量名（在一些很神秘的以致于无法在这里介绍的情况中，将 main 用作变量名会引发错误，由于它在任何情况下都是容易混淆的，因此最好不要这样做）。同样，其他函数名和变量名也都不能是关键字。不过，在程序中将同一个名称（比如 cout）用作对象名和变量名会把编译器搞糊涂。也就是说，在不使用 cout 对象进行输出的函数中，可以将 cout 用作变量名，但不能在同一个函数中同时将 cout 用作对象名和变量名。

2.4.4　用户定义的有返回值的函数

我们再深入一步，编写一个使用返回语句的函数。main() 函数已经揭示了有返回值的函数的格式：在函数头中指出返回类型，在函数体结尾处使用 return。可以用这种形式为在英国观光的人解决重量的问题。在英国，很多浴室都以英石（stone）为单位，不像美国以磅或公斤为单位。一英石等于 14 磅，程序清单 2.6 使用一个函数来完成这样的转换。

程序清单 2.6　convert.cpp

```
// convert.cpp -- converts stone to pounds
#include <iostream>
int stonetolb(int);     // function prototype
int main()
{
   using namespace std;
   int stone;
   cout << "Enter the weight in stone: ";
   cin >> stone;
   int pounds = stonetolb(stone);
   cout << stone << " stone = ";
   cout << pounds << " pounds." << endl;
   return 0;
}

int stonetolb(int sts)
{
   return 14 * sts;
}
```

下面是该程序的运行情况：

```
Enter the weight in stone: 14
14 stone = 196 pounds.
```

在 main() 中，程序使用 cin 来给整型变量 stone 提供一个值。这个值被作为参数传递给 stonetolb() 函数，在该函数中，这个值被赋给变量 sts。然后，stonetolb() 用关键字 return 将 14 * sts 返回给 main()。这表明 return 后面并非一定得跟一个简单的数字。这里通过使用较为复杂的表达式，避免了创建一个新变量，将

结果赋给该变量，然后将它返回。程序将计算表达式的值（这里为 196），并将其返回。如果返回表达式的值很麻烦，可以采取更复杂的方式：

```
int stonetolb (int sts)
{
    int pounds = 14 * sts;
    return pounds;
}
```

两个版本返回的结果相同，但第二个版本花的时间稍微长些。

通常，在可以使用一个简单常量的地方，都可以使用一个返回值类型与该常量相同的函数。例如，stonetolb()返回一个 int 值，这意味着可以以下面的方式使用该函数：

```
int aunt = stonetolb (20);
int aunts = aunt + stonetolb (10);
cout << "Ferdie weighs " << stonetolb (16) << " pounds." << endl;
```

在上述任何一种情况下，程序都将计算返回值，然后在语句中使用这个值。

这些例子表明，函数原型描述了函数接口，即函数如何与程序的其他部分交互。参数列表指出了何种信息将被传递给函数，函数类型指出了返回值的类型。程序员有时将函数比作一个由出入它们的信息所指定的黑盒子（black boxes，电工用语）。函数原型将这种观点诠释得淋漓尽致（参见图 2.9）。

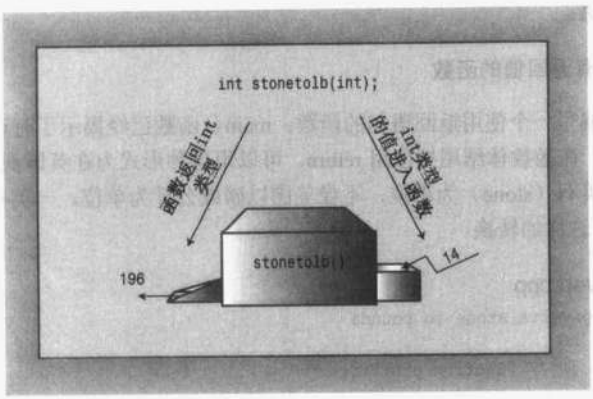


图 2.9 函数原型和作为黑盒子的函数

函数 stonetolb()短小且简单，但包含了全部的函数特性：

- 有函数头和函数体。
- 接受一个参数。
- 返回一个值。
- 需要一个原型。

可以把 stonetolb()看作函数设计的标准格式。第 7 章和第 8 章将更详细地介绍函数。而本章的内容让读者能够很好地了解函数的工作方式及其如何与 C++匹配。

2.4.5 在多函数程序中使用 using 编译指令

在程序清单 2.5 中，两个函数中都包含下面一条 using 编译指令：

```
using namespace std;
```

这是因为每个函数都使用了 cout，因此需要能够访问位于名称空间 std 中的 cout 定义。

在程序清单 2.5 中，可以采用另一种方法让两个函数都能够访问名称空间 std，即将编译指令放在函数的外面，且位于两个函数的前面：

```
// ourfunc1.cpp - repositioning the using directive
#include <iostream>
using namespace std; // affects all function definitions in this file
```

```
void simon(int);

int main()
{
   simon(3);
   cout << "Pick an integer: ";
   int count;
   cin >> count;
   simon(count);
   cout << "Done!" << endl;
   return 0;
}

void simon(int n)
{
   cout << "Simon says touch your toes " << n << " times. " << endl;
}
```

当前通行的理念是，只让需要访问名称空间 std 的函数访问它是更好的选择。例如，在程序清单 2.6 中，只有 main()函数使用 cout，因此没有必要让函数 stonetolb()能够访问名称空间 std。因此编译指令 using 被放在函数 main()中，使得只有该函数能够访问名称空间 std。

总之，让程序能够访问名称空间 std 的方法有多种，下面是其中的几种：

- 将 using std namespace;放在函数定义之前，让文件中所有的函数都能够使用名称空间 std 中所有的元素。
- 将 using std namespace;放在特定的函数定义中，让该函数能够使用名称空间 std 中的所有元素。
- 在特定的函数中使用类似 using std::cout;这样的编译指令而不是 using std namespace;，让该函数能够使用指定的元素，如 cout。
- 完全不使用编译指令 using，在需要使用名称空间 std 中的元素时，使用前缀 std::，如下所示：

```
std::cout << "I'm using cout and endl from the std namespace" <<std::endl;
```

命名约定

C++程序员给函数、类和变量命名时，可以有很多种选择。程序员对风格的观点五花八门，这些看法有时就像公共论坛上的圣战。就函数名称而言，程序员有以下选择：

- Myfunction()。
- myfunction()。
- myFunction()。
- my_function()。
- my_funct()。

选择取决于开发团体、使用的技术或库以及程序员个人的品位和喜好。因此凡是 C++语言所涉及的所有合法的风格都是正确的，都可以根据个人的判断而使用。

撇开语言是否容许不谈，个人的命名风格也是值得注意的——它有助于保持一致性和精确性。精确、让人一目了然的个人命名约定是良好的软件工程的标志，它在整个编程生涯中都会起到很好的作用

2.5 总结

C++程序由一个或多个被称为函数的模块组成。程序从 main()函数（全部小写）开始执行，因此该函数必不可少。函数由函数头和函数体组成。函数头指出函数的返回值（如果有的话）的类型和函数期望通过参数传递给它的信息的类型。函数体由一系列位于花括号（{ }）中的 C++语句组成。

有多种类型的 C++语句，包括：

- **声明语句**——定义函数中使用的变量的名称和类型。
- **赋值语句**——使用赋值操作符（=）给变量赋值。
- **消息语句**——将消息发送给对象，激发某种行动。
- **函数调用**——执行函数。被调用的函数执行完毕后，程序返回到函数调用语句后面的语句。
- **函数原型**——声明函数的返回类型、函数接受的参数数量和类型。
- **返回语句**——将一个值从被调用的函数那里返回到调用函数中。

类是用户定义的数据类型规范，它详细描述了如何表示信息以及可对数据执行的操作。对象是根据类规范创建的实体，就像简单变量是根据数据类型描述创建的实体一样。

C++提供了两个用于处理输入和输出的预定义对象（cin 和 cout），它们是 istream 和 ostream 类的实例，这两个类是在 iostream 文件中定义的。为 ostream 类定义的插入操作符（<<）使得将数据插入到输出流成为可能；为 istream 类定义的抽取操作符（>>）能够从输入流中抽取信息。cin 和 cout 都是智能对象，能够根据程序上下文自动将信息从一种形式转换为另一种形式。

C++可以使用大量的 C 库函数。要使用库函数，应当包含提供该函数原型的头文件。

至此，读者对简单的 C++程序有了大致的了解，可以进入下一章，了解程序的细节。

2.6 复习题

在附录 J 中可以找到所有复习题的答案。

1. C++程序的模块叫什么？
2. 下面的预处理器编译指令是做什么用的？

```
#include <iostream>
```

3. 下面的语句是做什么用的？

```
using namespace std;
```

4. 什么语句可以用来打印短语“Hello，world”，然后开始新的一行？
5. 什么语句可以用来创建名为 cheeses 的整数变量？
6. 什么语句可以用来将值 32 赋给变量 cheeses？
7. 什么语句可以用来将从键盘输入的值读入变量 cheeses 中？
8. 什么语句可以用来打印“We have X varieties of cheese”，用 cheeses 变量的当前值来代替 X。
9. 下面的函数原型指出了关于函数的哪些信息？

```
int froop(double t);
void rattle(int n);
int prune(void);
```

10. 定义函数时，在什么情况下不必使用关键字 return？

2.7 编程练习

1. 编写一个 C++程序，它显示您的姓名和地址。
2. 编写一个 C++程序，它要求用户输入一个以浪为单位的距离，然后将它转换为码（一浪等于 220 码）。
3. 编写一个 C++程序，它使用 3 个用户定义的函数（包括 main()），并生成下面的输出：

```
Three blind mice
Three blind mice
See how they run
See how they run
```

其中一个函数要调用两次，该函数生成前两行；另一个函数也被调用两次，并生成其余的输出。

4. 编写一个程序，其中的 main()调用一个用户定义的函数（以摄氏温度值为参数，并返回相应的华氏温度值）。该程序按下面的格式要求用户输入摄氏温度值，并显示结果：

```
Please enter a Celsius value: 20
20 degrees Celsius is 68 degrees Fahrenheit.
```

下面是转换公式：

华氏温度=1.8×摄氏温度+32.0

5. 编写一个程序，其 main()调用一个用户定义的函数（以光年值为参数，并返回对应天文单位的值）。该程序按下面的格式要求用户输入光年值，并显示结果：

```
Enter the number of light years: 4.2
4.2 light years = 265608 astronomical units.
```

天文单位是从地球到太阳的平均距离（约 150 000 000 千米或 93 000 000 英里），光年是光一年走的距离（约 10 万亿千米或 6 万亿英里。除太阳外，最近的恒星大约离地球 4.2 光年）。请使用 double 类型（参见程序清单 2.4），转换公式为：

1 光年=63 240 天文单位

6. 编写一个程序，要求用户输入小时数和分钟数。在 main()函数中，将这两个值传递给一个 void 函数，后者以下面这样的格式显示这两个值：

```
Enter the number of hours: 9
Enter the number of minutes: 28
Time: 9:28
```

第 3 章　处理数据

本章内容包括：

- C++变量的命名规则。
- C++内置的整型：unsigned long、long、unsigned int、int、unsigned short、short、char、unsigned char、signed char 和 bool。
- 表示各种整型的系统限制的 climits 文件。
- 各种整型的数字常量。
- 使用 const 限定符来创建符号常量。
- C++内置的浮点类型：float、double 和 long double。
- 表示各种浮点类型的系统限制的 cfloat 文件。
- 各种浮点类型的数字常量。
- C++的算术操作符。
- 自动类型转换。
- 强制类型转换。

面向对象编程（OOP）的本质是设计并扩展自己的数据类型。设计自己的数据类型就是让类型与数据匹配。如果正确做到了这一点，将会发现以后使用数据时会容易得多。不过，在创建自己的类型之前，必须了解并理解 C++内置的类型，因为这些类型是创建自己类型的基本组件。

内置的 C++类型分两组：基本类型和复合类型。本章将介绍基本类型，它表示整数和浮点数。似乎只有两种类型，但 C++知道，没有任何一种整型和浮点型能够满足所有的编程要求，因此对于这两种数据，它提供了多种变体。第 4 章将介绍在基本类型的基础上创建的复合类型，包括数组、字符串、指针和结构。

当然，程序还需要一种标识存储数据的方法，本章将介绍这样一种方法——使用变量。然后介绍如何在 C++中进行算术运算以及 C++如何将值从一种类型转换为另一种类型。

3.1　简单变量

程序通常都需要存储信息——如 IBM 股票当前的价格、纽约市 8 月份的平均湿度、美国宪法中使用最多的字母及其相对使用频率或猫王模仿者的数目。为把信息存储在计算机中，程序必须记录 3 个基本属性：

- 信息将存储在哪里。
- 要存储什么值。
- 存储何种类型的信息。

到目前为止，本书的范例采取的策略都是声明一个变量。声明中使用的类型描述了信息的类型和通过符号来表示其值的变量名。例如，假设实验室首席助理 Igor 使用了下面的语句：

```
int braincount;
braincount = 5;
```

这些语句告诉程序，它正在存储整数，并使用名称 braincount 来表示该整数的值（这里为 5）。实际上，程序将找到一块能够存储整数的内存，将该内存单元标记为 braincount，并将 5 复制到该内存单元中。这些语句没有告诉您（或者 Igor），这个值将被存储在内存中的什么位置，但程序确实记录了这种信息。实际上，可以使用&操作符来检索 braincount 在内存中的位置。下一章介绍另一种标识数据的方法（使用指针）时，将介绍这个操作符。

3.1.1　变量名

C++提倡使用有一定含义的变量名。如果变量表示差旅费，应将其命名为 cost_of_trip 或 costOfTrip，而不要将其命名为 x 或 cot。必须遵循几种简单的 C++命名规则：

- 在名称中只能使用字母字符、数字和下划线（_）。
- 名称的第一个字符不能是数字。
- 区分大写字符与小写字符。
- 不能将 C++关键字用作名称。
- 以两个下划线或下划线和大写字母打头的名称被保留给实现（编译器及其使用的资源）使用。以一个下划线开头的名称被保留给实现，用作全局标识符。
- C++对于名称的长度没有限制，名称中所有的字符都有意义。

倒数第二点与前面几点有些不同，因为使用像_time_stop 或_Donut 这样的名称不会导致编译器错误，而会导致行为的不确定性。换句话说，不知道结果将是什么。不出现编译器错误的原因是，这样的名称不是非法的，但要留给实现使用。全局名称指的是名称被声明的位置，这将在第 4 章讨论。

最后一点使得 C++与 ANSI C（C99 标准）有所区别，后者只保证名称中的前 63 个字符有意义（在 ANSI C 中，前 63 个字符相同的名称被认为是相同的，即使第 64 个字符不同）。

下面是一些有效和无效的 C++名称：

```
int poodle;    // valid
int Poodle;    // valid and distinct from poodle
int POODLE;    // valid and even more distinct
Int terrier;   // invalid -- has to be int, not Int
int my_stars3  // valid
int _Mystars3; // valid but reserved -- starts with underscore
int 4ever;     // invalid because starts with a digit
int double;    // invalid -- double is a C++ keyword
int begin;     // valid -- begin is a Pascal keyword
int __fools;   // valid but reserved -- starts with two underscores
int the_very_best_variable_i_can_be_version_112;  // valid
int honky-tonk;       // invalid -- no hyphens allowed
```

如果想用两个或更多的单词组成一个名称，通常的做法是用下划线字符将单词分开，如 my_onions；或者从第二个单词开始将每个单词的第一个字母大写，如 myEyeTooth（C 程序员倾向于按 C 语言的方式使用下划线，而 Pascal 程序员喜欢采用大写方式）。这两种形式都很容易将单词区分开，如 carDrip 和 cardRip 或 boat_sport 和 boats_port。

变量名

变量命名方案和函数命名方案一样，也有很多话题可供讨论。确实，该主题会引发一些最尖锐的反对意见。同样，和函数名称一样，只要变量名合法，C++编译器就不会介意，但是一致、精确的个人命名约定是很有帮助的。

与函数命名一样，大写在变量命名中也是一个关键问题（参见第 2 章的注释“命名约定”），但很多程序员可能会在变量名中加入其他的信息，即描述变量类型或内容的前缀。例如，可以将整型变量 myWeight 命名为 nMyWeight，其中前缀 n 用来表示整数值，在阅读代码或变量定义不是十分清楚的情况下，前缀很有用。另外，这个变量也可以叫做 intMyWeight，这将更精确，而且容易理解，不过它多了几个字母（对

于很多程序员来说，这是非常讨厌的事）。常以这种方式使用的其他前缀有：str 或 sz（表示以空字符结束的字符串）、b（表示布尔值）、p（表示指针）和 c（表示单个字符）。

随着对 C++的逐步了解，将发现很多有关前缀命名风格的范例（包括漂亮的 m_lpctstr 前缀——这是一个类成员值，其中包含了指向常量的长指针和以空字符结尾的字符串），还有其他更奇异、更违反直觉的风格，采不采用这些风格，完全取决于程序员。在 C++所有主观的风格中，一致性和精度是最重要的。请根据自己的需要、喜好和个人风格来使用变量名（或必要时，根据雇主的需要、喜好和个人风格来选择变量名）。

3.1.2 整型

整数就是没有小数部分的数字，如 2、98、-5286 和 0。整数有很多，如果将无限大的整数看作很大，则不可能用有限的计算机内存来表示所有的整数。因此，语言只能表示所有整数的一个子集。有些语言（如标准 Pascal）只提供一种整型（一种类型满足所有要求！），而 C++则提供好几种，这样便能够根据程序的具体要求选择最合适的整型。

不同 C++整型使用不同的内存量来存储整数。使用的内存量越大，可以表示的整数值范围也越大。另外，有的类型（符号类型）可表示正值和负值，而有的类型（无符号类型）不能表示负值。术语宽度（width）用于描述存储整数时使用的内存量。使用的内存越多，则越宽。C++的基本整型（按宽度递增的顺序排列）分别是 char、short、int 和 long，其中每种类型都有符号版本和无符号版本，因此总共有 8 种类型可供选择。下面更详细地介绍这些整数类型。由于 char 类型有一些特殊属性（它最常用来表示字符，而不是数字），因此本章将首先介绍其他类型。

3.1.3 short、int 和 long

计算机内存由一些叫做位（bit）的单元组成（参见本章后面的注释“位与字节”）。C++的 short、int 和 long 类型通过使用不同数目的位来存储值，最多能够表示 3 种不同的整数宽度。如果在所有的系统中，每种类型的宽度都相同，则使用起来将非常方便。例如，如果 short 总是 16 位，int 总是 32 位，等等。不过生活并非那么简单。然而，没有一种选择能够满足所有的计算机设计要求。C++提供了一种灵活的标准，它确保了最小长度（从 C 语言借鉴而来），如下所示：

- short 至少 16 位。
- int 至少与 short 一样长。
- long 至少 32 位，且至少与 int 一样长。

位与字节

计算机内存的基本单元是位（bit）。可以将位看作电子开关，可以开，也可以关。关表示值 0，开表示值 1。8 位的内存块可以设置出 256 种不同的组合，因为每一位都可以有两种设置，所以 8 位的总组合数为 2×2×2×2×2×2×2×2，即 256。因此，8 位单元可以表示 0～255 或者-128 到 127。每增加一位，组合数便加倍。这意味着可以把 16 位单元设置成 65 536 个不同的值，把 32 位单元设置成 4 294 672 296 个不同的值。

字节（byte）通常指的是 8 位的内存单元。从这个意义上说，字节指的就是描述计算机内存量的度量单位，1KB 等于 1 024 字节，1MB 等于 1024KB。不过，C++对字节的定义与此不同。C++字节由至少能够容纳实现的基本字符集的相邻位组成，也就是说，可能取值的数目必须等于或超过字符数目。在美国，基本字符集通常是 ASCII 和 EBCDIC 字符集，它们都可以用 8 位来容纳，所以在使用这两种字符集的系统中，C++字节通常包含 8 位。不过，国际编程可能需要使用更大的字符集，如 Unicode，因此有些实现可能使用 16 位甚至 32 位的字节。

当前很多系统都使用最小长度，即 short 为 16 位，long 为 32 位。这仍然为 int 提供了多种选择，其宽度可以是 16 位、24 位或 32 位，同时又符合标准。通常，在老式 IBM PC 的实现中，int 的宽度为 16 位（与

short 相同），而在 Windows 98、Windows NT、Windows XP、Macintosh OS X、VAX 和很多其他微型计算机的实现中，为 32 位（与 long 相同）。有些实现允许选择如何处理 int（读者所用的实现使用的是什么？下面的例子将演示如何在不打开手册的情况下，确定系统的限制）。类型的宽度随实现而异，这可能在将 C++程序从一种环境移到另一种环境时引发问题。但只要小心一点（如本章后面讨论的那样），就可以最大限度地减少这种问题。

可以像使用 int 一样，使用这些类型名来声明变量：

```
short score;          // creates a type short integer variable
int temperature;      // creates a type int integer variable
long position;        // creates a type long integer variable
```

实际上，short 是 short int 的简称，而 long 是 long int 的简称，但是程序设计者们几乎都不使用比较长的形式。

这 3 种类型（int、short 和 long）都是符号类型，这意味着每种类型的取值范围中，负值和正值几乎相同。例如，16 位的 int 的取值范围为-32 768 到+32 767。

要知道系统中整数的最大长度，可以在程序中使用 C++工具来检查类型的长度。首先，sizeof 操作符返回类型或变量的长度，单位为字节（操作符是内置的语言元素，对一个或多个数据进行运算，并生成一个值。例如，加号操作符+将两个值相加）。注意，“字节”的含义依赖于实现，因此在一个系统中，两字节的 int 可能是 16 位，而在另一个系统中可能是 32 位。其次，头文件 climits（在老式实现中为 limits.h）中包含了关于整型限制的信息。具体地说，它定义了表示各种限制的符号名称。例如，INT_MAX 为 int 的最大取值，CHAR_BIT 为字节的位数。程序清单 3.1 演示了如何使用这些工具。该程序还演示如何初始化，即使用声明语句将值赋给变量。

程序清单 3.1　limits.cpp

```
// limits.cpp -- some integer limits
#include <iostream>
#include <climits>            // use limits.h for older systems
int main()
{
   using namespace std;
   int n_int = INT_MAX;       // initialize n_int to max int value
   short n_short = SHRT_MAX;  // symbols defined in limits.h file
   long n_long = LONG_MAX;

   // sizeof operator yields size of type or of variable
   cout << "int is " << sizeof (int) << " bytes." << endl;
   cout << "short is " << sizeof n_short << " bytes." << endl;
   cout << "long is " << sizeof n_long << " bytes." << endl << endl;

   cout << "Maximum values:" << endl;
   cout << "int: " << n_int << endl;
   cout << "short: " << n_short << endl;
   cout << "long: " << n_long << endl << endl;

   cout << "Minimum int value = " << INT_MIN << endl;
   cout << "Bits per byte = " << CHAR_BIT << endl;
   return 0;
}
```

注意：头文件 climits 是 ANSI C 头文件 limits.h 的 C++版本。有些早期的 C++平台中没有头文件。如果使用的是这种系统，就不能运行该范例了。

下面是使用 Microsoft Visual C++ 7.1 时程序清单 2.1 中程序的输出：

```
int is 4 bytes.
short is 2 bytes.
long is 4 bytes.
```

```
Maximum values:
int: 2147483647
short: 32767
long: 2147483647

Minimum int value = -2147483648
Bits per byte = 8
```

下面是使用另一个系统（Borland C++ 3.1 for DOS）时的输出：

```
int is 2 bytes.
short is 2 bytes.
long is 4 bytes.

Maximum values:
int: 32767
short: 32767
long: 2147483647

Minimum int value = -32768
Bits per byte = 8
```

程序说明

我们来看一下该程序的主要编程特性。

（1）操作符 sizeof 和头文件 limits

sizeof 操作符指出，在使用 8 位字节的系统中，int 的长度为 4 个字节。可对类型名或变量名使用 sizeof 操作符。对类型名（如 int）使用 sizeof 操作符时，应将名称放在括号中；但对变量名（如 n_short）使用该操作符，括号是可选的：

```
cout << "int is " << sizeof (int) << " bytes.\n";
cout << "short is " << sizeof n_short << " bytes.\n";
```

头文件 climits 定义了符号常量（参见本章后面的注释“符号常量”）来表示类型的限制。如前所述，INT_MAX 表示类型 int 能够存储的最大值，对于 DOS 系统，为 32 767。编译器厂商提供了 climits 文件，该文件指出了其编译器中的值。例如，在使用 32 位 int 的系统（如 Windows XP）中，climits 文件将 INT_MAX 定义为 2 147 483 647。表 3.1 对该文件中定义的符号常量进行了总结，其中的一些符号常量与还没有介绍过的类型相关。

表 3.1　climits 中的符号常量

符号常量	表　示
CHAR_BIT	char 的位数
CHAR_MAX	char 的最大值
CHAR_MIN	char 的最小值
SCHAR_MAX	signed char 的最大值
SCHAR_MIN	signed char 的最小值
UCHAR_MAX	unsigned char 的最大值
SHRT_MAX	short 的最大值
SHRT_MIN	short 的最小值
USHRT_MAX	unsigned short 的最大值
INT_MAX	int 的最大值
INT_MIN	int 的最小值
UNIT_MAX	unsigned int 的最大值
LONG_MAX	long 的最大值

续表

符号常量	表 示
LONG_MIN	long 的最小值
ULONG_MAX	unsigned long 的最大值

（2）初始化

初始化将赋值与声明合并在一起。例如，下面的语句：

```
int n_int = INT_MAX;
```

声明了变量 n_int，并将 int 的最大取值赋给该变量；也可以使用常规常量来初始化。可以将变量初始化为另一个变量，条件是后者已经定义过。甚至可以使用表达式来初始化变量，条件是当程序执行到该声明时，表达式中所有的值都是已知的：

```
int uncles = 5;                    // initialize uncles to 5
int aunts = uncles;                // initialize aunts to 5
int chairs = aunts + uncles + 4;   // initialize chairs to 14
```

如果将 uncles 的声明移到语句列表的最后，则另外两条初始化语句将非法，因为这样当程序试图对其他变量进行初始化时，uncles 的值是未知的。

前面的初始化句法来自 C 语言，C++还有另一种 C 语言没有的初始化句法：

```
int owls = 101;  // traditional C initialization
int wrens(432);  // alternative C++ syntax, set wrens to 432
```

记住：如果不对函数内部定义的变量进行初始化，该变量的值将是不确定的。这意味着该变量的值将是它被创建之前，该内存单元保存的值。

如果知道变量的初始值应该是什么，则应对它进行初始化。将变量声明和赋值分开，可能会带来瞬间悬而未决的问题：

```
short year;      // what could it be?
year = 1492;     // oh
```

然而，在声明变量时对它进行初始化，可避免以后忘记给它赋值的情况发生。

符号常量——预处理器方式

climits 文件中包含与下面类似的语句行：

```
#define INT_MAX 32767
```

在 C++编译过程中，首先将源代码传递给预处理器。在这里，#define 和#include 一样，也是一个预处理器编译指令。该编译指令告诉预处理器：在程序中查找 INT_MAX，并将所有的 INT_MAX 都替换为 32767。因此#define 编译指令的工作方式与文本编辑器或字处理器中的全局搜索和替换命令相似。修改后的程序将在完成这些替换后被编译。预处理器查找独立的标记（单独的单词），跳过嵌入的单词。也就是说，预处理器不会将 PINT_MAXTM 替换为 P32767IM。也可以使用#define 来定义自己的符号常量（参见程序清单 3.2）。不过，#define 编译指令是 C 语言遗留下来的。C++有一种更好的创建符号常量的方法（使用关键字 const，将在后面的一节讨论），所以不会经常使用#define。不过，有些头文件，尤其是那些被设计成可用于 C 和 C++中的头文件，必须使用#define。

3.1.4 无符号类型

前面介绍的 3 种整型都有一种不能存储负数值的无符号变体，其优点是可以增大变量能够存储的最大值。例如，如果 short 表示的范围为–32 768 到+32 767，则无符号版本的表示范围为 0～65 535。当然，仅当数值不会为负时才应使用无符号类型，如人口、粒数等。要创建无符号版本的基本整型，只需使用关键字 unsigned 来修改声明即可：

```
unsigned short change;       // unsigned short type
unsigned int rovert;         // unsigned int type
unsigned quarterback;        // also unsigned int
unsigned long gone;          // unsigned long type
```

注意，unsigned 本身是 unsigned int 的缩写。

程序清单 3.2 演示了如何使用无符号类型，并说明了程序试图超越整型的限制时将产生的后果。最后，再看一看预处理器语句#define。

程序清单 3.2 exceed.cpp

```
// exceed.cpp -- exceeding some integer limits
#include <iostream>
#define ZERO 0     // makes ZERO symbol for 0 value
#include <climits>  // defines INT_MAX as largest int value
int main()
{
   using namespace std;
   short sam = SHRT_MAX;     // initialize a variable to max value
   unsigned short sue = sam;// okay if variable sam already defined

   cout << "Sam has " << sam << " dollars and Sue has " << sue;
   cout << " dollars deposited." << endl
   << "Add $1 to each account." << endl << "Now ";
   sam = sam + 1;
   sue = sue + 1;
   cout << "Sam has " << sam << " dollars and Sue has " << sue;
   cout << " dollars deposited.\nPoor Sam!" << endl;
   sam = ZERO;
   sue = ZERO;
   cout << "Sam has " << sam << " dollars and Sue has " << sue;
   cout << " dollars deposited." << endl;
   cout << "Take $1 from each account." << endl << "Now ";
   sam = sam - 1;
   sue = sue - 1;
   cout << "Sam has " << sam << " dollars and Sue has " << sue;
   cout << " dollars deposited." << endl << "Lucky Sue!" << endl;
   return 0;
}
```

注意：与程序清单 3.1 一样，程序清单 3.2 也使用了 climits 文件；老式编译器可能需要使用文件 limits.h，有些非常老的编译器可能根本没有这两个文件。

下面是该程序的输出：

```
Sam has 32767 dollars and Sue has 32767 dollars deposited.
Add $1 to each account.
Now Sam has -32768 dollars and Sue has 32768 dollars deposited.
Poor Sam!
Sam has 0 dollars and Sue has 0 dollars deposited.
Take $1 from each account.
Now Sam has -1 dollars and Sue has 65535 dollars deposited.
Lucky Sue!
```

该程序将一个 short 变量（sam）和一个 unsigned short 变量（sue）分别设置为最大的 short 值，在我们的系统上，是 32 767。然后，将这些变量的值都加 1。这对于 sue 来说没有什么问题，因为新值仍比无符号整数的最大值小得多；但 sam 的值从 32 767 变成了-32 768！同样，对于 sam，将其设置为 0 并减去 1，也不会有问题；但对于无符号变量 sue，将其设置为 0 并减去后，它变成了 65 535。可以看出，这些整型变量的行为就像里程表。如果超越了限制，其值将为范围另一端的取值（参见图 3.1）。C++确保了无符号类型的这种行为；但 C++并不保证符号整型超越限制（上溢和下溢）时不出错，而这正是当前实现中最为常见的行为。

超越 long

C99 添加了两种新的类型，它们很可能成为下一版 C++标准的组成部分。确实，很多 C++编译器已经支持它们了。这两种类型分别是 long long 和 unsigned long long。两种类型都至少是 64 位，且至少同 long 和 unsigned long 类型一样宽。

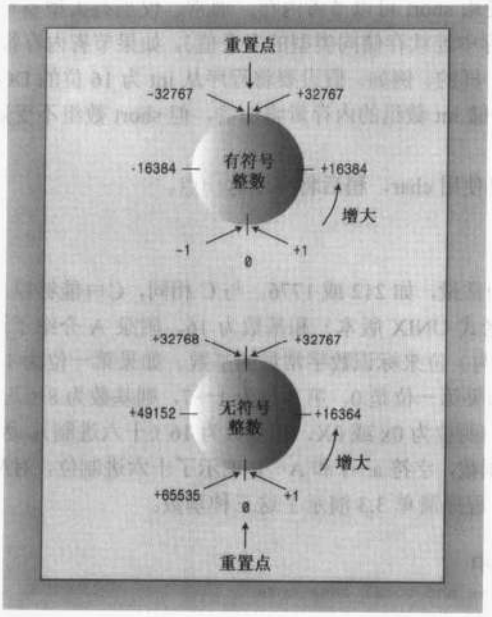


图 3.1　典型的整型溢出行为

3.1.5　选择整型类型

C++提供了大量的整型，应使用哪种类型呢？通常，int 被设置为对目标计算机而言最为“自然”的长度。自然长度（natural size）指的是计算机处理起来效率最高的长度。如果没有非常有说服力的理由来选择其他类型，则应使用 int。

现在来看看可能使用其他类型的原因。如果变量表示的值不可能为负，如文档中的字数，则可以使用无符号类型，这样变量可以表示更大的值。

如果知道变量可能表示的整数值大于 16 位整数的最大可能值，则使用 long。即使系统上 int 为 32 位，也应这样做。这样，当程序移植到 16 位的系统中时，程序就不会突然无法正常工作（参见图 3.2）。

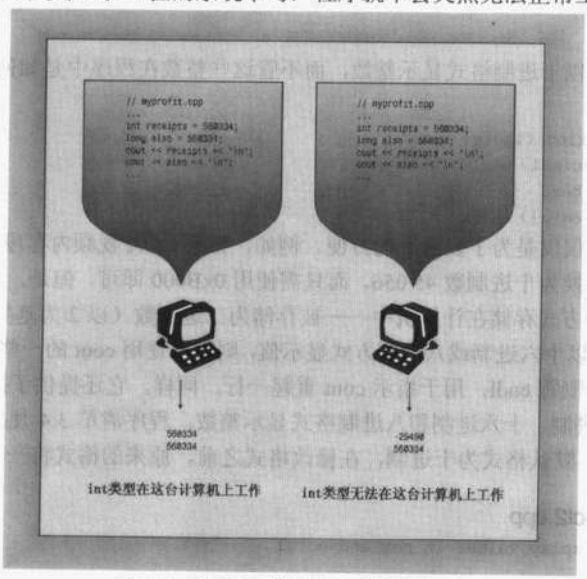


图 3.2　为提高可移植性，请使用 long

如果 short 比 int 小，则使用 short 可以节省内存。通常，仅当有大型整型数组时，才有必要使用 short（数组是一种数据结构，在内存中连续存储同类型的多个值）。如果节省内存很重要，则应使用 short 而不是使用 int，即使它们的长度是一样的。例如，假设要将程序从 int 为 16 位的 DOS PC 系统移到 int 为 32 位的 Windows XP 系统，则用于存储 int 数组的内存量将加倍，但 short 数组不受影响。记住，节省一点就是赢得一点。

如果只需要一个字节，可使用 char，稍后将介绍这一点。

3.1.6 整型常量

整型常量是显式地书写的常量，如 212 或 1776。与 C 相同，C++能够以 3 种不同的计数方式来书写整数：基数为 10、基数为 8（老式 UNIX 版本）和基数为 16。附录 A 介绍了这几种计数系统；这里将介绍 C++表示法。C++使用前一（两）位来标识数字常量的基数。如果第一位为 1～9，则基数为 10（十进制）；因此 93 是以 10 为基数的。如果第一位是 0，第二位为 1～7，则基数为 8（八进制）；因此 042 的基数是 8，它相当于十进制数 34。如果前两位为 0x 或 0X，则基数为 16（十六进制）；因此 0x42 为十六进制数，相当于十进制数 66。对于十六进制数，字符 a～f 和 A～F 表示了十六进制位，对应于 10～15。0xF 为 15，0x45 为 165（10 个 16 加 5 个 1）。程序清单 3.3 演示了这三种基数。

程序清单 3.3 hexoct.cpp

```
// hexoct.cpp -- shows hex and octal constants
#include <iostream>
int main()
{
    using namespace std;
    int chest = 42;     // decimal integer constant
    int waist = 0x42;   // hexadecimal integer constant
    int inseam = 042;   // octal integer constant

    cout << "Monsieur cuts a striking figure!\n";
    cout << "chest = " << chest << "\n";
    cout << "waist = " << waist << "\n";
    cout << "inseam = " << inseam << "\n";
    return 0;
}
```

在默认情况下，cout 以十进制格式显示整数，而不管这些整数在程序中是如何书写的，如下面的输出所示：

```
Monsieur cuts a striking figure!
chest = 42 (42 in decimal)
waist = 66 (0x42 in hex)
inseam = 34 (042 in octal)
```

记住，这些表示方法仅仅是为了表达上的方便。例如，如果 CGA 视频内存段为十六进制 B000，则不必在程序使用之前将它转换为十进制数 45 056，而只需使用 0xB000 即可。但是，不管把值书写为 10、012 还是 0xA，都将以相同的方式存储在计算机中——被存储为二进制数（以 2 为基数）。

顺便说一句，如果要以十六进制或八进制方式显示值，则可以使用 cout 的一些特殊特性。前面指出过，头文件 iostream 提供了控制符 endl，用于指示 cout 重起一行。同样，它还提供了控制符 dec、hex 和 oct，分别用于指示 cout 以十进制、十六进制和八进制格式显示整数。程序清单 3.4 使用了 hex 和 oct 以上述 3 种格式显示十进制值 42。默认格式为十进制，在修改格式之前，原来的格式将一直有效。

程序清单 3.4 hexoct2.cpp

```
// hexoct2.cpp -- display values in hex and octal
#include <iostream>
using namespace std;
int main()
```

```
{
    using namespace std;
    int chest = 42;
    int waist = 42;
    int inseam = 42;

    cout << "Monsieur cuts a striking figure!" << endl;
    cout << "chest = " << chest << " (decimal)" << endl;
    cout << hex;      // manipulator for changing number base
    cout << "waist = " << waist << " hexadecimal" << endl;
    cout << oct;      // manipulator for changing number base
    cout << "inseam = " << inseam << " (octal)" << endl;
    return 0;
}
```

下面是运行该程序时得到的输出：

```
Monsieur cuts a striking figure!
chest = 42 (decimal)
waist = 2a hexadecimal
inseam = 52 (octal)
```

诸如 cout << hex;等代码不会在屏幕上显示任何内容，而只是修改 cout 显示整数的方式。因此，控制符 hex 实际上是一条消息，告诉 cout 采取何种行为。另外，由于标识符 hex 位于名称空间 std 中，而程序使用了该名称空间，因此不能将 hex 用作变量名。然而，如果省略编译指令 using，而使用 std::cout、std::endl、std::hex 和 std::oct，则可以将 hex 用作变量名。

3.1.7　C++如何确定常量的类型

程序的声明将特定的整型变量的类型告诉了 C++编译器，但编译器是如何知道常量的类型呢？假设在程序中使用常量表示一个数字：

```
cout << "Year = " << 1492 << "\n";
```

程序将把 1492 存储为 int、long 还是其他整型呢？答案是，除非有理由存储为其他类型（如使用了特殊的后缀来表示特定的类型，或者值太大，不能存储为 int），否则 C++将整型常量存储为 int 类型。

首先来看看后缀。后缀是放在数字常量后面的字母，用于表示类型。整数后面的 l 或 L 后缀表示该整数为 long 常量，u 或 U 后缀表示 unsigned int 常量，ul（可以采用任何一种顺序，大写小写均可）表示 unsigned long 常量（由于小写 l 看上去像 1，因此应使用大写 L 作后缀）。例如，在 int 为 16 位、long 为 32 位的系统上，数字 22022 被存储为 int，占 16 位，数字 22022L 被存储为 long，占 32 位。同样，22022LU 和 22022UL 都被存储为 unsigned long。

接下来考察长度。在 C++中，对十进制整数采用的规则，与十六进制和八进制稍微有些不同。对于不带后缀的十进制整数，将使用下面几种类型中能够存储该数的最小类型来表示：int、long 或 unsigned long。在 int 为 16 位、long 为 32 位的计算机系统上，20000 被表示为 int 类型，40000 被表示为 long 类型，3000000000 被表示为 unsigned long 类型。对于不带后缀的十六进制或八进制整数，将使用下面几种类型中能够存储该数的最小类型来表示：int、unsigned int、long 或 unsigned long。在将 40000 表示为 long 的计算机系统中，十六进制数 0x9C40（40000）将被表示为 unsigned int。这是因为十六进制常用来表示内存地址，而内存地址是没有符号的，因此，usigned int 比 long 更适合用来表示 16 位的地址。

3.1.8　char 类型：字符和小整数

下面介绍最后一种整型：char 类型。顾名思义，char 类型是专为存储字符（如字母和数字）而设计的。现在，存储数字对于计算机来说算不了什么，但存储字母则是另一回事。编程语言通过使用字母的数值编码解决了这个问题。因此，char 类型是另一种整型。它足够长，能够表示目标计算机系统中的所有基本符号——所有的字母、数字、标点符号等。实际上，多数系统所支持的字符都不超过 256 种，因此用一个字节就可以表示所有的符号。因此，虽然 char 最常被用来处理字符，但也可以将它用做比 short 更小的整型。

在美国，最常用的符号集是 ASCII 字符集（参见附录 C）。字符集中的字符用数值编码（ASCII 码）表

示。例如，字符 A 的编码为 65，字符 M 的编码为 77。为方便起见，本书在范例中使用的是 ASCII 码。不过，C++实现使用的是其主机系统的编码——例如，IBM 大型机使用 EBCDIC 编码。ASCII 和 EBCDIC 都不能很好地满足国际需要，C++支持的宽字符类型可以存储更多的值，如国际 Unicode 字符集使用的值。本章稍后将介绍 wchar_t 类型。

程序清单 3.5 使用了 char 类型。

程序清单 3.5 chartype.cpp

```
// chartype.cpp -- the char type
#include <iostream>
int main()
{
   using namespace std;
   char ch;       // declare a char variable

   cout << "Enter a character: " << endl;
   cin >> ch;
   cout << "Holla! ";
   cout << "Thank you for the " << ch << " character." << endl;
   return 0;
}
```

下面是该程序的输出：

```
Enter a character:
M
Holla! Thank you for the M character.
```

有趣的是，程序中输入的是 M，而不是对应的字符编码 77。另外，程序将打印 M，而不是 77。通过查看内存可以知道，77 是存储在变量 ch 中的值。这种神奇的力量不是来自 char 类型，而是来自 cin 和 cout，这些工具为您完成了转换工作。输入时，cin 将键盘输入的 M 转换为 77；输出时，cout 将值 77 转换为所显示的字符 M；cin 和 cout 的行为都是由变量类型引导的。如果将 77 存储在 int 变量中，则 cout 将把它显示为 77（也就是说，cout 显示两个字符 7）。程序清单 3.6 说明了这一点，该程序还演示了如何在 C++中书写字符常量：将字符用单引号括起，如'M'（注意，范例中没有使用双引号。C++对字符用单引号，对字符串使用双引号。cout 对象能够处理这两种情况，但正如第 4 章将讨论的，这两者有天壤之别）。最后，程序引入了 cout 的一项特性——cout.put()函数，该函数显示一个字符。

程序清单 3.6 morechar.cpp

```
// morechar.cpp -- the char type and int type contrasted
#include <iostream>
int main()
{
   using namespace std;
   char ch = 'M';      // assign ASCII
   int i = ch;         // store same code in an int
   cout << "The ASCII code for " << ch << " is " << i << endl;

   cout << "Add one to the character code:" << endl;
   ch = ch + 1;        // change character code in c
   i = ch;             // save new character code in i
   cout << "The ASCII code for " << ch << " is " << i << endl;
   // using the cout.put() member function to display a char
   cout << "Displaying char ch using cout.put(ch): ";
   cout.put(ch);

   // using cout.put() to display a char constant
   cout.put('!');

   cout << endl << "Done" << endl;
```

```
    return 0;
}
```

下面是程序清单 3.6 中程序的输出：

```
The ASCII code for M is 77
Add one to the character code:
The ASCII code for N is 78
Displaying char c using cout.put(ch): N!
Done
```

1. 程序说明

在程序清单 3.6 中，'M' 表示字符 M 的数值编码，因此将 char 变量 c 初始化为 'M'，将把 c 设置为 77。然后，程序将同样的值赋给 int 变量 i，这样 c 和 i 的值都是 77。接下来，cout 把 c 显示为 M，而把 i 显示为 77。如前所述，值的类型将引导 cout 选择如何显示值——这是智能对象的另一个例子。

由于 c 实际上是一个整数，因此可以对它使用整数操作，例如加 1，这将把 c 的值变为 78。然后，程序将 i 重新设置为新的值（也可以将 i 加 1）。cout 再次将这个值的 char 版本显示为字符，将 int 版本显示为数字。

C++将字符表示为整数提供了方便，使得操纵字符值很容易。不必使用笨重的转换函数来在字符和 ASCII 码之间来回转换。

最后，程序使用 cout.put() 函数来显示 c 和 a 字符常量。

2. 成员函数 cout.put()

cout.put() 到底是什么东西？其名称中为何有一个句点？函数 cout.put() 是一个重要的 C++ OOP 概念——成员函数——的第一个例子。类定义了如何表示和控制数据。成员函数归类所有，描述了操纵类数据的方法。例如类 ostream 有一个 put() 成员函数，用来输出字符。只能通过类的特定对象（例如这里的 cout 对象）来使用成员函数。要通过对象（如 cout）使用成员函数，必须用句点将对象名和函数名称（put()）连接起来。句点被称为成员操作符。cout.put() 的意思是，通过类对象 cout 来使用函数 put()。我们将在第 10 章介绍类时详细介绍这一点。现在，接触过的类只有 istream 和 ostream，可以通过使用它们的成员函数来熟悉这一概念。

cout.put() 成员函数提供了另一种显示字符的方法，可以替代<<操作符。现在读者可能会问，为何需要 cout.put()。答案与历史有关。在 C++版本 2.0 之前，cout 将字符变量显示为字符，而将字符常量（如 'M' 和 'N'）显示为数字。问题是，C++的早期版本与 C 一样，也将把字符常量存储为 int 类型。也就是说，'M' 的编码 77 将被存储在一个 16 位或 32 位的单元中。而 char 变量一般占 8 位。下面的语句：

```
char c = 'M';
```

将从常量 'M' 中复制 8 位（左边的 8 位）到变量 c 中。遗憾的是，这意味着对 cout 来说，'M' 和 c 看上去有天壤之别，虽然它们存储的值相同。因此，下面的语句：

```
cout << '$';
```

将打印$字符的 ASCII 码，而不是$字符。不过下面的语句：

```
cout.put('$');
```

将打印字符。在版本 2.0 之后，C++将字符常量存储为 char 类型，而不是 int 类型。这意味着 cout 现在可以正确处理字符常量了。

cin 对象有几种不同的方式可以读取输入的字符。通过使用一个利用循环来读取几个字符的程序，读者可以更容易地领会到这一点。因此在第 5 章介绍了循环之后我们再来讨论这个主题。

3. char 常量

在 C++中，书写字符常量的方式有多种。对于常规字符（如字母、标点符号和数字），最简单的方法是将字符用单引号括起。这种表示法代表的是字符的数值编码。例如，ASCII 系统中的对应情况如下：

- 'A'为 65，即字符 A 的 ASCII 码。
- 'a' is 97，即字符 a 的 ASCII 码。
- '5' is 53，即数字 5 的 ASCII 码。

- ' ' is 32，即空格字符的 ASCII 码。
- '!' is 33，即惊叹号的 ASCII 码。

这种表示法优于数值编码，它更加清晰，且不需要知道编码方式。如果系统使用的是 EBCDIC，则 A 的编码将不是 65，但是'A'表示的仍然是字符 A。

有些字符不能直接通过键盘输入到程序中。例如，按 Enter 键并不能使字符串包含一个换行符；相反，程序编辑器将把这种键击解释为在源代码中开始新的一行。其他一些字符也无法从键盘输入，因为 C++语言赋予了它们特殊的含义。例如，双引号字符用来分隔字符串，因此不能只把一个双引号放在字符串中。对于这些字符，C++提供了一种特殊的表示方法——转义序列，如表 3.2 所示。例如，\a 表示振铃字符，它可以使终端扬声器振铃。转义序列\n 表示换行符，\"将双引号作为常规字符，而不是字符串分隔符。可以在字符串或字符常量中使用这些表示法，如下例所示：

```
char alarm = '\a';
cout << alarm << "Don't do that again!\a\n";
cout << "Ben \"Buggsie\" Hacker\nwas here!\n";
```

最后一行的输出如下：

```
Ben "Buggsie" Hacker
was here!
```

注意，应该像处理常规字符（如 Q）那样处理转义序列（如\n）。也就是说，将它们作为字符常量时，应用单引号括起；将它们放在字符串中时，不要使用单引号。

表 3.2 C++转义序列的编码

字符名称	ASCII 符号	C++代码	十进制 ASCII 码	十六进制 ASCII 码
换行符	NL (LF)	\n	10	0xA
水平制表符	HT	\t	9	0x9
垂直制表符	VT	\v	11	0xB
退格	BS	\b	8	0x8
回车	CR	\r	13	0xD
振铃	BEL	\a	7	0x7
反斜杠	\	\\	92	0x5C
问号	?	\?	63	0x3F
单引号	'	\'	39	0x27
双引号	"	\"	34	0x22

换行符可替代 endl，用于在输出中重起一行。可以以字符常量表示法（'\n'）或字符串方式（"\n"）使用换行符。下面三行代码都将光标移到下一行的开头：

```
cout << endl;    // using the endl manipulator
cout << '\n';    // using a charcter constant
cout << "\n";    // using a string
```

可以将换行符嵌入到较长的字符串中，这通常比使用 endl 方便。例如，下面两条 cout 语句的输出相同：

```
cout << endl << endl << "What next?" << endl << "Enter a number:" << endl;
cout << "\n\nWhat next?\nEnter a number:\n";
```

显示数字时，使用 endl 比输入"\n"或'\n'更容易些，但显示字符串时，在字符串末尾添加一个换行符所需的输入量要少些：

```
cout << x << endl;    // easier than cout << x << "\n";
cout << "Dr. X.\n";   // easier than cout << "Dr. X." << endl;
```

最后，可以基于字符的八进制和十六进制编码来使用转义序列。例如，Ctr+Z 的 ASCII 码为 26，对应的八进制编码为 032，十六进制编码为 0x1a。可以用下面的转义序列来表示该字符：\032 或\x1a。将这些编码用单引号括起，可以得到相应的字符常量，如'\032'，也可以将它们放在字符串中，如"hi\x1a there"。

提示：在可以使用数字转义序列或符号转义序列（如\0x8 和\b）时，应使用符号序列。数字表示与特定的编码方式（如 ASCII 码）相关，而符号表示适用于任何编码方式，其可读性也更强。

程序清单 3.7 演示了一些转义序列。它使用振铃字符来提请注意，使用换行符使光标前进，使用退格字符使光标向左退一格（Houdini 曾经在只使用转义序列的情况下，绘制了一幅关于哈得逊河的图画；他无疑是一位转义序列艺术大师）。

程序清单 3.7　bondini.cpp

```
// bondini.cpp -- using escape sequences
#include <iostream>
int main()
{
   using namespace std;
   cout << "\aOperation \"HyperHype\" is now activated!\n";
   cout << "Enter your agent code:_______\b\b\b\b\b\b\b";
   long code;
   cin >> code;
   cout << "\aYou entered " << code << "...\n";
   cout << "\aCode verified! Proceed with Plan Z3!\n";
   return 0;
}
```

注意：有些基于 ANSI C 之前的编译器的 C++系统不能识别\a。对于使用 ASCII 字符集的系统，可以用\007 替换\a。有些系统的行为可能有所不同，例如可能将\b 显示为一个小矩形，而不是退格，或者在退格时删除，还可能忽略\a。

运行程序清单 3.7 中的程序时，将在屏幕上显示下面的文本：

```
Operation "HyperHype" is now activated!
Enter your agent code:_______
```

打印下划线字符后，程序使用退格字符将光标退到第一个下划线处。读者可以输入自己的密码，并继续。下面是完整的运行情况：

```
Operation "HyperHype" is now activated!
Enter your agent code:42007007
You entered 42007007...
Code verified! Proceed with Plan Z3!
```

4. 通用字符名

C++实现支持一个基本的源字符集，即可用来编写源代码的字符集。它由标准美国键盘上的字符（大写和小写）和数字、C 语言中使用的符号（如{和=）以及其他一些字符（如换行符和空格）组成。还有一个基本的执行字符集（通过执行程序来生成的字符），它增加了一些字符，如退格和振铃。C++标准还允许实现提供扩展源字符集和扩展执行字符集。另外，那些被作为字母的额外字符也可用于标识符名称中。也就是说，德国实现可能允许使用日耳曼语的元音变音，而法国实现则允许使用重元音。C++有一种表示这种特殊字符的机制，它独立于任何特定的键盘，使用的是通用字符名（universal character name）。

通用字符名的用法类似于转义序列。通用字符名可以以\u 或\U 打头。\u 后面是 8 个十六进制位，\U 后面则是 16 个十六进制位。这些位表示的是字符的 ISO 10646 编码（ISO 10646 是一种正在制定中的国际标准，为大量的字符提供了数值编码，请参见本章后面的“Unicode 和 ISO 10646”）。

如果所用的实现支持扩展字符，则可以在标识符（如字符常量）和字符串中使用通用字符名。例如，请看下面的代码：

```
int k\u00F6rper;
cout << "Let them eat g\u00E2teau.\n";
```

ö 的 ISO 10646 编码为 00F6，而 â 的编码为 00E2。因此，上述 C++代码将变量名设置为 körper，并显示下面的输出：

```
Let them eat gâteau.
```

如果系统不支持 ISO 10646，它将显示其他字符或 gu00E2teau，而不是 â。

Unicode 和 ISO 10646

Unicode 提供了一种表示各种字符集的解决方案——为大量字符和符号提供标准数值编码，并根据类型将它们分组。例如，ASCII 码成为 Unicode 的子集，因此在这两种系统中，美国的拉丁字符（如 A 和 Z）的表示相同。不过，Unicode 还包含其他拉丁字符，如欧洲语言使用的拉丁字符、来自其他语言（如希腊语、西里尔语、希伯来语、阿拉伯语、泰语和孟加拉语）中的字符以及象形文字（如中国和日本的文字）。到目前为止，Unicode 可以表示 96000 多种符号和 49 个手写符号（script），它还在不断发展中。要了解更多有关的知识，请登录 Unicode 联盟的网站：www.unicode.org。

国际标准化组织（ISO）建立了一个工作组，专门开发 ISO 10646——这也是一个对多种语言文本进行编码的标准。ISO 10646 小组和 Unicode 小组从 1991 年开始合作，以确保他们的标准同步。

5. signed char 和 unsigned char

与 int 不同的是，char 在默认情况下既不是没有符号，也不是有符号。是否有符号由 C++实现决定，这样编译器开发人员可以最大限度地将这种类型与硬件属性匹配起来。如果 char 有某种特定的行为对您来说非常重要，则可以显式地将类型设置为 signed char 或 unsigned char：

```
char fodo;          // may be signed, may be unsigned
unsigned char bar;  // definitely unsigned
signed char snark;  // definitely signed
```

如果将 char 用作数值类型，则 unsigned char 和 signed char 之间的差异将非常重要。unsigned char 类型的表示范围通常为 0～255，而 signed char 的表示范围为-128 到 127。例如，假设要使用一个 char 变量来存储像 200 这样大的值，则在某些系统上可以，而在另一些系统上可能不可以。但使用 unsigned char 可以在任何系统上达到这种目的。另一方面，如果使用 char 变量来存储标准 ASCII 字符，则 char 有没有符号都没关系，在这种情况下，可以使用 char。

6. wcha_t

程序需要处理的字符集可能无法用一个 8 位的字节表示，如日文汉字系统。对于这种情况，C++的处理方式有两种。首先，如果大型字符集是实现的基本字符集，则编译器厂商可以将 char 定义为一个 16 位的字节或更长的字节。其次，一种实现可以同时支持一个小型基本字符集和一个较大的扩展字符集。8 位 char 可以表示基本字符集，另一种类型 wchar_t（宽字符类型）可以表示扩展字符集。wchar_t 类型是一种整数类型，它有足够的空间，可以表示系统使用的最大扩展字符集。这种类型与另一种整型（底层（underlying）类型）的长度和符号属性相同。对底层类型的选择取决于实现，因此在一个系统中，它可能是 unsigned short，而在另一个系统中，则可能是 int。

cin 和 cout 将输入和输出看作是 char 流，因此不适于用来处理 wchar_t 类型。iostream 头文件的最新版本提供了作用相似的工具——wcin 和 wcout，可用于处理 wchar_t 流。另外，可以通过加上前缀 L 来指示宽字符常量和宽字符串。下面的代码将字母 P 的 wchar_t 版本存储到变量 bob 中，并显示单词 tall 的 wchar_t 版本：

```
wchar_t bob = L'P';         // a wide-character constant
wcout << L"tall" << endl;  // outputting a wide-character string
```

在支持两字节 wchar_t 的系统中，上述代码将把每个字符存储在一个两个字节的内存单元中。本书不使用宽字符类型，但读者应知道有这种类型，尤其是在进行国际编程或使用 Unicode 或 ISO 10646 时。

3.1.9 新的 bool 类型

ANSI/ISO C++标准添加了一种名叫 bool 的新类型（对 C++来说是新的）。它的名称来源于英国数学家 George Boole，是他开发了逻辑律的数学表示法。在计算中，布尔变量的值可以是 true 或 false。过去，C++和 C 一样，也没有布尔类型。在第 5 章和第 6 章中将会看到，C++将非零值解释为 true，将零解释为 false。不过，现在可以使用 bool 类型来表示真和假了，它们分别用预定义的字面值 true 和 false 表示。也就是说，可以这样编写语句：

```
bool isready = true;
```

字面值 true 和 false 都可以通过提升转换为 int 类型，true 被转换为 1，而 false 被转换为 0：

```
int ans = true;            // ans assigned 1
int promise = false;       // promise assigned 0
```

另外，任何数字值或指针值都可以被隐式转换（即不用显式强制转换）为 bool 值。任何非零值都被转换为 true，而零被转换为 false：

```
bool start = -100;         // start assigned true
bool stop = 0;             // stop assigned false
```

在第 6 章介绍 if 语句后，范例中将经常使用数据类型 bool。

3.2　const 限定符

现在回过头来介绍常量的符号名称。符号名称指出常量表示的内容。另外，如果程序在多个地方使用同一个常量，而需要修改该常量值，则只需修改一个符号定义即可。本章前面关于#define 语句的注释（“符号常量——预处理器方法”）指出过，C++有一种更好的处理符号常量的方法，这种方法就是使用 const 关键字来修改变量声明和初始化。例如，假设需要一个表示一年中月份数的符号常量，请在程序中输入下面这行代码：

```
const int MONTHS = 12;  // Months is symbolic constant for 12
```

这样，便可以在程序中使用 MONTHS，而不是 12 了（在程序中，12 可能表示一英尺有多少英寸或一打面包圈是多少个，而名称 MONTHS 指出了值 12 表示的是什么）。常量（如 MONTHS）被初始化后，其值就被固定了，编译器将不允许再修改该常量的值。例如，如果您这样做，Borland C++将显示一条错误消息，指出需要一个 lvalue。这条消息与将值 4 赋给 3 时显示的错误消息相同（lvalue 是位于赋值操作符左边的值，如变量）。关键字 const 叫做限定符，因为它限定了声明的含义。

一种常见的做法是将名称大写，这样可以提醒您 MONTHS 是一个常量。这决不是一种通用约定，但在阅读程序时有助于区分常量和变量。另一种约定是只将名称的首字母大写，还有一种约定是以字母 k 打头，如 kmonths。当然，还有其他的约定。很多组织都有特殊的编码约定，要求其程序员遵守。

创建常量的通用格式如下：

```
const type name = value;
```

注意，应在声明中对 const 进行初始化。下面的代码不好：

```
const int toes;         // value of toes undefined at this point
toes = 10;              // too late!
```

如果在声明常量时没有提供值，则该常量的值将是不确定的，而且无法修改它。

如果以前使用过 C 语言，可能会觉得前面讨论的#define 语句已经足够完成这样的工作了。但是 const 比#defien 好。首先，它能够明确指定类型。其次，可以使用 C++的作用域规则将定义限制在特定的函数或文件中（作用域规则描述了名称在各种模块中的可知程度，这将在第 9 章讨论）。第三，可以将 const 用于更复杂的类型（如第 4 章将介绍的数组和结构）。

提示：如果读者在学习 C++之前学习过 C 语言，并打算使用#define 来定义符号常量，请不要这样做，而应使用 const。

ANSI C 也使用 const 限定符，这是从 C++借鉴来的。如果对 ANSI C 版本比较熟悉，则应注意，C++版本稍微有些不同。区别之一是作用域规则，这将在第 9 章讨论；另一个主要的区别是，在 C++（而不是 C）中可以用 const 值来声明数组长度，第 4 章将介绍一些这样的例子。

3.3　浮点数

了解各种 C++整型后，我们来看一看浮点类型，它们是 C++的第二组基本类型。浮点数能够表示带小数部分的数字，如 M1 油箱的汽油里程数（0.56MPG），它们提供的值范围也更大。如果数字很大，无法表示为 long 类型，如银河系中行星的数目（大约 4000 亿颗），则可以使用浮点类型来表示。

使用浮点类型，可以表示诸如 2.5、3.14159 和 122442.32 这样的数字，即带小数部分的数字。计算机将这样的值分成两部分存储。一部分表示值，另一部分用于对值进行放大或缩小。下面打个比方。对于数字 34.1245 和 34124.5，它们除了小数点的位置不同外，其他都是相同的。可以把第一个数表示为 0.341245（基准值）和 100（缩放因子），而将第二个数表示为 0.341245（基准值相同）和 10000（缩放因子更大）。缩放因子的作用是移动小数点的位置，术语浮点因此而得名。C++内部表示浮点数的方法与此相同，只不过它基于的是二进制数，因此缩放因子是 2 的幂，不是 10 的幂。幸运的是，程序员不必详细了解内部表示。重要的是，浮点数能够表示小数值、非常大和非常小的值，它们的内部表示方法与整数有天壤之别。

3.3.1 书写浮点数

C++有两种书写浮点数的方式。第一种是使用常用的标准小数点表示法：

```
12.34               // floating-point
939001.32           // floating-point
0.00023             // floating-point
8.0                 // still floating-point
```

即使小数部分为 0（如 8.0），小数点也将确保该数字以浮点格式（而不是整数格式）表示（C++标准允许实现表示不同的区域；例如，提供了使用欧洲方法的机制，即将逗号而不是句点用作小数点。不过，这些选项控制的是数字在输入和输出中的外观，而不是数字在代码中的外观）。

第二种表示浮点值的方法叫做 E 表示法，其外观是像这样的：3.45E6，这指的是 3.45 与 1 000 000 相乘的结果；E6 指的是 10 的 6 次方，即 1 后面 6 个 0。因此，3.45E6 表示的是 3 450 000，6 被称为指数，3.45 被称为尾数。下面是一些例子：

```
2.52e+8             // can use E or e, + is optional
8.33E-4             // exponent can be negative
7E5                 // same as 7.0E+05
-18.32e13           // can have + or - sign in front
7.123e12            // US public debt, early 2004
5.98E24             // mass of Earth in kilograms
9.11e-31            // mass of an electron in kilograms
```

读者可能注意到，E 表示法最适合于非常大和非常小的数。

E 表示法确保数字以浮点格式存储，即使没有小数点。注意，既可以使用 E 也可以使用 e，指数可以是正数也可以是负数（参见图 3.3）。不过，数字中不能有空格，因此 7.2 E6 是非法的。

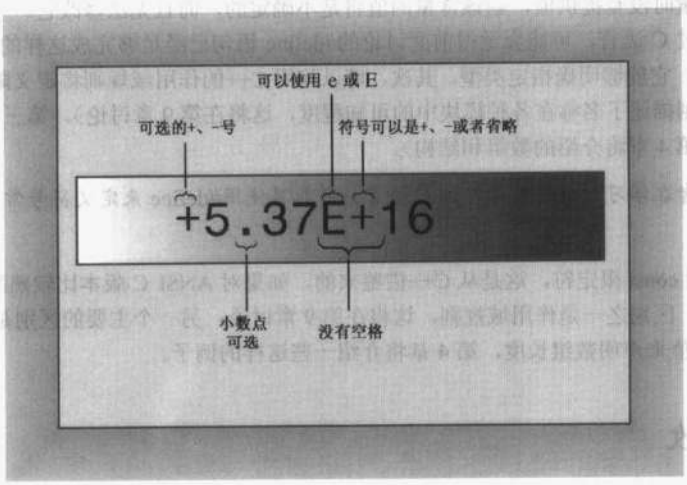


图 3.3 E 表示法

指数为负数意味着除以 10 的乘方，而不是乘以 10 的乘方。因此，8.33E-4 表示 $8.33/10^4$，即 0.000833。同样，电子质量 9.11e-31 kg 表示 0.00000000000000000000000000000911 kg。可以按照自己喜欢的方式表

小数字（911 在美国是报警电话，而电话信息通过电子传输，这是巧合还是科学阴谋呢？读者可以自己作出评判）。注意，-8.33E4 指的是-83300。前面的符号用于数值，而指数的符号用于缩放。

记住：d.dddE+n 指的是将小数点向右移 n 位，而 d.dddE-n 指的是将小数点向左移 n 位。

3.3.2　浮点类型

和 ANSI C 一样，C++也有 3 种浮点类型：float、double 和 long double。这些类型是按它们可以表示的有效数位和允许的指数最小范围来描述的。有效位（significant figure）是数字中有意义的位。例如，加利福尼亚的 Shast 山脉的高度为 14 162 英尺，该数字使用了 5 个有效位，指出了最接近的英尺数。不过，将 Shasta 山脉的高度写成约 14 000 英尺时，有效位数为 2 位，因为结果经过四舍五入精确到了千位。在这种情况下，其余的 3 位只不过是占位符而已。有效位数不依赖于小数点的位置。例如，可以将高度写成 14.162 千英尺。这样仍有 5 个有效位，因为这个值精确到了第 5 位。

事实上，C 和 C++对于有效位数的要求是，float 至少 32 位，double 至少 48 位，且不少于 float，long double 至少和 double 一样多。这三种类型的有效位数可以一样多。不过，通常，float 为 32 位，double 为 64 位，long double 为 80、96 或 128 位。另外，这 3 种类型的指数范围至少是-37 到 37。可以从头文件 cfloat 或 float.h 中找到系统的限制（cfloat 是 C 语言的 float.h 文件的 C++版本）。下面是 Borland C++ Builder 的 float.h 文件中的一些批注项：

```
// the following are the minimum number of significant digits
#define DBL_DIG 15         // double
#define FLT_DIG 6          // float
#define LDBL_DIG 18        // long double

// the following are the number of bits used to represent the mantissa
#define DBL_MANT_DIG     53
#define FLT_MANT_DIG     24
#define LDBL_MANT_DIG    64

// the following are the maximum and minimum exponent values
#define DBL_MAX_10_EXP   +308
#define FLT_MAX_10_EXP   +38
#define LDBL_MAX_10_EXP  +4932
#define DBL_MIN_10_EXP   -307
#define FLT_MIN_10_EXP   -37
#define LDBL_MIN_10_EXP  -4931
```

注意：有些 C++实现尚未添加头文件 cfloat，有些基于 ANSI C 之前的编译器的 C++实现没有提供头文件 float.h。

程序清单 3.8 演示了 float 和 double 类型及它们表示数字时在精度方面的差异（即有效位数）。该程序预览了将在第 17 章介绍的 ostream 方法——setf()。这种调用迫使输出使用定点表示法，以便更好地了解精度，它防止程序把较大的值切换为 E 表示法，并使程序显示到小数点后 6 位小数。参数 ios_base::fixed 和 ios_base::floatfield 是通过包含 iostream 来提供的常量。

程序清单 3.8　floatnum.cpp

```
// floatnum.cpp -- floating-point types
#include <iostream>
int main()
{
   using namespace std;
   cout.setf(ios_base::fixed, ios_base::floatfield); // fixed-point
   float tub = 10.0 / 3.0;     // good to about 6 places
   double mint = 10.0 / 3.0;   // good to about 15 places
   const float million = 1.0e6;
```

```
    cout << "tub = " << tub;
    cout << ", a million tubs = " << million * tub;
    cout << ",\nand ten million tubs = ";
    cout << 10 * million * tub <<endl;

    cout << "mint = " << mint << " and a million mints = ";
    cout << million * mint <<endl;
    return 0;
}
```

下面是该程序的输出：

```
tub = 3.333333, a million tubs = 3333333.250000,
and ten million tubs = 33333332.000000
mint = 3.333333 and a million mints = 3333333.333333
```

注意：C++标准用 ios_base::fixed 代替了 ios::fixed，用 ios_base::floatfield 代替了 ios::floatfield。如果编译器不接受 ios_base，请使用 ios；也就是说，用 ios::fixed 代替 ios_base::fixed。在默认情况下，老版本的 C++在显示浮点值时，将显示 6 位小数，如 2345.831541。标准 C++在默认情况下总共显示 6 位（2345.83），在值到达或超过百万位时，便切换为 E 表示法（2.34583E+06）。不过，无论是老版本还是新版本，非默认显示模式（如这里的 fixed）都显示 6 位小数。

默认设置还会删除结尾的 0，将 23.4500 显示为 23.45。使用 setf()语句来覆盖默认设置时，响应随实现而异。老版本，如 Borland C++ 3.1 for DOS，在这种模式下，也会删除结尾的 0。遵循标准的版本，如 Microsoft Visual C++ 7.0、Metrowerkr CodeWarrior 9、Gnu GCC 3.3 和 Borland C++ 5.5，将把 0 显示出来，如程序清单 3.8 所示

程序说明

通常 cout 会删除结尾的零。例如，将 3333333.250000 显示为 3333333.25。调用 cout.setf()将覆盖这种行为，至少在新的实现中是这样的。这里要注意的是，为何 float 的精度比 double 低。tub 和 mint 都被初始化为 10.0/3.0——3.3333333333333333……由于 cout 打印 6 位小数，在这种情况下，tub 和 mint 都是精确的。但当程序将每个数乘以 100 方后，tub 在第 7 个 3 之后就与正确的值有了误差。tub 在 7 位有效位上还是精确的（该系统确保 float 至少有 6 位有效位，不过这是最糟糕的情况）。然而，double 类型的变量显示了 13 个 3，因此它至少有 13 位是精确的。由于系统确保 15 位有效位，因此这就没有什么好奇怪的了。另外，将 tub 乘以 100 方，然后再乘以 10 后，得到的结果不正确，这再一次指出了 float 的精度限制。

cout 所属的 ostream 类有一个这样的类成员函数，它能够精确地控制输出的格式——字段宽度、小数位数、采用小数格式还是 E 格式，等等。第 17 章将介绍这些选项。为简单起见，本书的例子通常只使用<<操作符。有时候，这种方法显示的位数比需要的位数多，不过这只能带来感觉上的差异。如果确实很关心这种差异，可以浏览第 17 章，以了解如何使用格式化方法。不过，在这里我们就不作过多的解释了。

读取包含文件

C++源文件开头的包含编译指令总是有一种魔咒的力量，新手 C++程序员通过阅读和体验来了解哪个头文件添加哪些功能，再一一包含它们，以便程序能够运行。不要将包含文件作为神秘的知识而依赖；可以随便打开及阅读它们。它们都是文本文件，因此可以很轻松地阅读它们。被包含在程序中的所有文件都存在于计算机中，或位于计算机可以使用的地方。找到那些要使用的包含文件，看看它们包含的内容。您将会很快地知道，所使用的源文件和头文件都是知识和信息的很好来源——在很多情况下，它们都是最好的文档。当使用更复杂的包含文件，并开始在应用程序中使用其他非标准库时，这种习惯将非常有帮助。

3.3.3 浮点常量

在程序中书写浮点常量的时候，程序将把它存储为哪种浮点类型呢？在默认情况下，像 8.24 和 2.4E8 这样的浮点常量都属于 double 类型。如果希望常量为 float 类型，请使用 f 或 F 后缀。对于 long double 类

型，可使用 l 或 L 后缀（由于 l 看起来像数字 1，因此 L 是更好的选择）。下面是一些范例：

```
1.234f          // a float constant
2.45E20F        // a float constant
2.345324E28     // a double constant
2.2L            // a long double constant
```

3.3.4　浮点数的优缺点

与整数相比，浮点数有两大优点。首先，它们可以表示整数之间的值。其次，由于有缩放因子，它们可以表示的范围大得多。另一方面，浮点运算的速度比整数运算慢，至少在没有数学协处理器的计算机上是如此，而且精度将降低。程序清单 3.9 说明了最后一点。

程序清单 3.9　fltadd.cpp

```
// fltadd.cpp -- precision problems with float
#include <iostream>
int main()
{
   using namespace std;
   float a = 2.34E+22f;
   float b = a + 1.0f;

   cout << "a = " << a << endl;
   cout << "b - a = " << b - a << endl;
   return 0;
}
```

注意：有些基于 ANSI C 之前的编译器的老式 C++实现不支持浮点常量后缀 f。如果出现这样的问题，可以用 2.34E+22 代替 2.34E+22f，用(float) 1.0 代替 1.0f。

该程序将数字加 1，然后减去原来的数字。结果应该为 1。下面是在某个系统上运行时该程序的输出：

```
a = 2.34e+022
b - a = 0
```

问题在于，2.34E+22 是一个小数点左边有 23 位的数字。加上 1，就是在第 23 位加 1。但 float 类型只能表示数字中的前 6 位或前 7 位，因此修改第 23 位对这个值不会有任何影响。

将类型分类

C++对基本类型进行分类，形成了若干个族。类型 signed char、short、int 和 long 统称为符号整型；它们的无符号版本统称为无符号整型。bool、char、wchar_t、符号整数和无符号整型统称为整型。float、double 和 long double 统称为浮点型。整数和浮点型统称算术（arithmetic）类型。

3.4　C++算术操作符

读者可能还对学校里作的算术练习记忆犹新，在计算机上也能够获得同样的乐趣。C++使用操作符来运算。它提供了几种操作符来完成 5 种基本的算术计算：加法、减法、乘法、除法以及求模。每种操作符都使用两个值（操作数）来计算结果。操作符及其操作数构成了表达式。例如，在下面的语句中：

```
int wheels = 4 + 2;
```

4 和 2 都是操作数，+是加法操作符，4+2 则是一个表达式，其值为 6。

下面是 C++的 5 种基本算术操作符：

- +操作符对操作数执行加法运算。例如，4+20 等于 24。
- -操作符从第一个数中减去第二个数。例如，12-3 等于 9。
- *操作符将操作数相乘。例如，28*4 等于 112。
- /操作符用第一个数除以第二个数。例如，1000/5 等于 200。如果两个操作数都是整数，则结果为

商的整数部分。例如，17/3 等于 5，小数部分被丢弃。

- %操作符求模。也就是说，它生成第一个数除以第二个数后的余数。例如，19%6 为 1，因为 19 是 6 的 3 倍余 1。两个操作数必须都是整型，将该操作符用于浮点数将导致编译错误。如果其中一个是负数，则结果的符号依赖于实现。

当然，变量和常量都可以用作操作数，程序清单 3.10 说明了这一点。由于%的操作数只能是整数，因此我们将在后面的例子中讨论它。

程序清单 3.10 arith.cpp

```
// arith.cpp -- some C++ arithmetic
#include <iostream>
int main()
{
    using namespace std;
    float hats, heads;

    cout.setf(ios_base::fixed, ios_base::floatfield); // fixed-point
    cout << "Enter a number: ";
    cin >> hats;
    cout << "Enter another number: ";
    cin >> heads;

    cout << "hats = " << hats << "; heads = " << heads << endl;
    cout << "hats + heads = " << hats + heads << endl;
    cout << "hats - heads = " << hats - heads << endl;
    cout << "hats * heads = " << hats * heads << endl;
    cout << "hats / heads = " << hats / heads << endl;
    return 0;
}
```

注意：如果读者所用的编译器不接受 serf() 中的 ios_base，可以使用老版本的 ios；也就是说，用 ios_base::fixed 代替 ios::fixed。

下面是该程序的输出，从中可知 C++能够完成简单的算术运算：

```
Enter a number: 50.25
Enter another number: 11.17
hats = 50.250000; heads = 11.170000
hats + heads = 61.419998
hats - heads = 39.080002
hats * heads = 561.292480
hats / heads = 4.498657
```

也许读者对得到的结果心存怀疑。11.17 加上 50.25 应等于 61.42，但是输出中却是 61.419998。这不是运算问题；而是由于 float 类型表示有效位数的能力有限。记住，对于 float，C++只保证 6 位有效位。如果将 61.419998 四舍五入成 6 位，将得到 61.4200，这是保证精度下的正确值。如果需要更高的精度，请使用 double 或 long double。

3.4.1 操作符优先级和结合性

读者是否委托 C++来完成复杂的算术运算？是的，但必须知道 C++使用的规则。例如，很多表达式都包含多个操作符。这样将产生一个问题：究竟哪个操作符最先被使用呢？例如，请看下面的语句：

```
int flyingpigs = 3 + 4 * 5;  // 35 or 23?
```

操作数 4 旁边有两个操作符：+和*。当多个操作符可用于同一个操作数时，C++使用优先级规则来决定首先使用哪个操作符。算术操作符遵循通常的代数优先级，先乘除，后加减。因此 3+4*5 指的是 3+(4*5)，而不是（3+4）*5，结果为 23，而不是 35。当然，可以使用括号来执行自己定义的优先级。附录 D 介绍了所有 C++操作符的优先级。其中，*、/和%位于同一行，这说明它们的优先级相同。同样，加和减的优先级也相同，但比乘除低。

有时，优先级列表并不够用。请看下面的语句：

```
float logs = 120 / 4 * 5;    // 150 or 6?
```

操作数 4 也位于两个操作符中间，但操作符/和*的优先级相同，因此优先级本身并不能指出程序究竟是先计算 120 除以 4，还是先计算 4 乘以 5。因为第一种选择得到的结果是 150，而第二种选择的结果是 6，因此选择十分重要。当两个操作符的优先级相同时，C++将看操作数的结合性（associativity）是从左到右，还是从右到左。从左到右的结合性意味着如果两个优先级相同的操作符被同时用于同一个操作数，则首先应用左侧的操作符。从右到左的结合性则首先应用右侧的操作符。附录 D 也列出了结合性方面的信息。从中可以看出，乘除都是从左到右结合的。这说明应当先对 4 使用左侧的操作符。也就是说，用 120 除以 4，得到的结果为 30，然后再乘以 5，结果为 150。

注意，仅当两个操作符被用于同一个操作数时，优先级和结合性规则才有效。请看下面的表达式：

```
int dues = 20 * 5 + 24 * 6;
```

操作符优先级表明了两点：程序必须在做加法之前计算 20*5，必须在做加法之前计算 24*6。但优先级和结合性都没有指出应先计算哪个乘法。读者可能认为，结合性表明应先做左侧的乘法，但是在这种情况下，两个*操作符并没有用于同一个操作数，所以该规则不适用。事实上，C++把这个问题留给了实现，让它来决定在系统中的最佳顺序。对于这个例子来说，两种顺序的结果是一样的，但是也有两种顺序结果不同的情况。在第 5 章讨论递增操作符时，将介绍一个这样的例子。

3.4.2　除法分支

除法操作符（/）的行为取决于操作数的类型。如果两个操作数都是整数，则 C++将执行整数除法。这意味着结果的小数部分将被丢弃，使得最后的结果是一个整数。如果其中有一个（或两个）操作数是浮点值，则小数部分将保留，结果为浮点数。程序清单 3.11 演示了 C++除法如何处理不同类型的值。和程序清单 3.10 一样，该程序也调用 setf() 成员函数来修改结果的显示方式。

程序清单 3.11　divide.cpp

```
// divide.cpp -- integer and floating-point division
#include <iostream>
int main()
{
   using namespace std;
   cout.setf(ios_base::fixed, ios_base::floatfield);
   cout << "Integer division: 9/5 = " << 9 / 5  << endl;
   cout << "Floating-point division: 9.0/5.0 = ";
   cout << 9.0 / 5.0 << endl;
   cout << "Mixed division: 9.0/5 = " << 9.0 / 5  << endl;
   cout << "double constants: 1e7/9.0 = ";
   cout << 1.e7 / 9.0 <<  endl;
   cout << "float constants: 1e7f/9.0f = ";
   cout << 1.e7f / 9.0f <<  endl;
   return 0;
}
```

注意：如果编译器不接受 setf() 中的 ios_base，请使用 ios。

有些基于 ANSI C 之前的编译器的 C++实现不支持浮点常量的 f 后缀。如果面临这样的问题，可以用 (float) 1.e7 / (float) 9.0 代替 1.e7f / 9.0f.

有些实现会删除结尾的零

下面是使用某种实现时，程序清单 3.11 中程序的输出：

```
Integer division: 9/5 = 1
Floating-point division: 9.0/5.0 = 1.800000
Mixed division: 9.0/5 = 1.800000
double constants: 1e7/9.0 = 1111111.111111
float constants: 1e7f/9.0f = 1111111.125000
```

从第一行输出可知，整数 9 除以 5 的结果为整数 1。4/5 的小数部分（或 0.8）被丢弃。在本章后面学习求模操作符时，将会看到这种除法的实际应用。接下来的两行表明，当至少有一个操作数是浮点数时，结果为 1.8。实际上，对不同类型进行运算时，C++将把它们全部转换为同一类型。本章稍后将介绍这种自动转换。最后两行的相对精度表明，如果两个操作数都是 double 类型，则结果为 double 类型；如果两个操作数都是 float 类型，则结果为 float 类型。记住，浮点常量在默认情况下为 double 类型。

操作符重载简介

在程序清单 3.11 中，除法操作符表示了 3 种不同的运算：int 除法、float 除法和 double 除法。C++根据上下文（这里是操作数的类型）来确定操作符的含义。使用相同的符号进行多种操作叫做操作符重载（operator overloading）。C++有一些内置的重载范例。C++还允许扩展操作符重载，以便能够用于用户定义的类，因此在这里看到的是一个重要的 OOP 属性（参见图 3.4）。

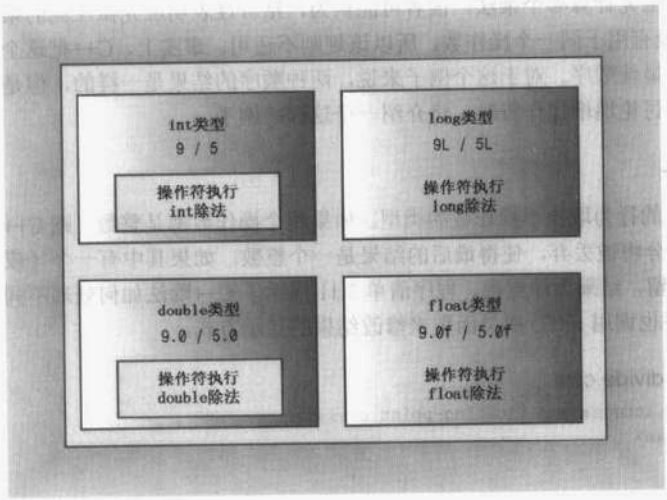


图 3.4 各种除法

3.4.3 求模操作符

比起求模操作符来说，多数人更熟悉加、减、乘、除，因此这里花些时间介绍这种操作符。求模操作符返回整数除法的余数。它与整数除法相结合，尤其适用于解决要求将一个量分成不同的整数单元的问题，例如将英寸转换为英尺和英寸，或者将美元转换为元、角、分、厘。第 2 章的程序清单 2.6 将重量单位英石转换为磅。程序清单 3.12 则将磅转换为英石。记住，一英石等于 14 磅，多数英国浴室都使用这种单位。该程序使用整数除法来计算合多少英石，再用求模操作符来计算余下多少磅。

程序清单 3.12 modulus.cpp

```
// modulus.cpp -- uses % operator to convert lbs to stone
#include <iostream>
int main()
{
    using namespace std;
    const int Lbs_per_stn = 14;
    int lbs;

    cout << "Enter your weight in pounds: ";
    cin >> lbs;
    int stone = lbs / Lbs_per_stn;      // whole stone
```

```
    int pounds = lbs % Lbs_per_stn;    // remainder in pounds
    cout << lbs << " pounds are " << stone;
       << " stone, " << pounds << " pound(s).\n";
    return 0;
}
```

下面是该程序的运行情况：

```
Enter your weight in pounds: 177
184 pounds are 13 stone, 2 pound(s).
```

在表达式 lbs/Lbs_per_stn 中，两个操作数的类型都是 int，所以计算机执行整数除法。lbs 的值为 177，所以表达式的值为 12。12 和 14 的乘积是 168，所以 177 与 14 相除的余数是 9，这就是 lbs % Lbs_per_stn 的值。现在即使在感情上还没有适应英国的质量单位，但在技术上也做好了去英国旅游时解决质量单位转换问题的准备。

3.4.4　类型转换

C++丰富的类型允许根据需求选择不同的类型，这也使计算机的操作更复杂。例如，将两个 short 值相加涉及到的硬件编译指令可能会与将两个 long 值相加不同。由于有 11 种整型和 3 种浮点类型，因此计算机需要处理大量不同的情况，尤其是对不同的类型进行运算时。为处理这种潜在的混乱，C++自动执行很多类型转换：

- 将一种算术类型的值赋给另一种算术类型的变量时，C++将对值进行转换。
- 表达式中包含不同的类型时，C++将对值进行转换。
- 将参数传递给函数时，C++将对值进行转换。

如果不知道进行这些自动转换时将发生的情况，将无法理解一些程序的结果，因此下面详细地介绍这些规则。

1. 赋值时进行的转换

C++允许将一种类型的值赋给另一种类型的变量。这样做时，值将被转换为接收变量的类型。例如，假设 so_long 的类型为 long，thirty 的类型为 short，而程序中包含这样的语句：

```
so_long = thirty;          // assigning a short to a long
```

则进行赋值时，程序将 thirty 的值（通常是 16 位）扩展为一个 long 值（通常为 32 位）。扩展后将得到一个新的值，这个值被存储在 so_long 中，而 thirty 的内容不变。

将一个值赋给取值范围更大的类型通常不会导致什么问题。例如，将 short 值赋给 long 变量并不会改变这个值，只是占用的字节更多而已。不过，将一个很大的 long 值（如 2111222333）赋给 float 变量将降低精度。因为 float 只有 6 位有效数字，因此这个值将被四舍五入为 2.11122E9。因此，有些转换是安全的，有些则会带来麻烦。表 3.3 列出了一些可能出现的转换问题。

表 3.3　　潜在的数值转换问题

转　　换	潜在的问题
将较大的浮点类型转换为较小的浮点类型，例如将 double 转换为 float	精度（有效数位）降低，值可能超出目标类型的取值范围，在这种情况下，结果将是不确定的
将浮点类型转换为整型	小数部分丢失，原来的值可能超出目标类型的取值范围，在这种情况下，结果将是不确定的
将较大的整型转换为较小的整型，如将 long 转换为 short	原来的值可能超出目标类型的取值范围，通常只复制右边的字节

将 0 赋给 bool 变量时，将被转换为 false；而非零值将被转换为 true。

将浮点值赋给整型将导致两个问题。首先，将浮点值转换为整型会将数字截短（除掉小数部分）。其次，float 值对于 int 变量来说可能太大了。在这种情况下，C++并没有定义结果应该是什么；这意味着不同的实现的反应可能不同。程序清单 3.13 演示了一些通过赋值进行的转换。

程序清单 3.13 assign.cpp

```
// assign.cpp -- type changes on assignment
#include <iostream>
int main()
{
    using namespace std;
    cout.setf(ios_base::fixed, ios_base::floatfield);
    float tree = 3;     // int converted to float
    int guess = 3.9832; // float converted to int
    int debt = 7.2E12;  // result not defined in C++
    cout << "tree = " << tree << endl;
    cout << "guess = " << guess << endl;
    cout << "debt = " << debt << endl;
    return 0;
}
```

下面是该程序在某个系统中的输出：

```
tree = 3.000000
guess = 3
debt = 1634811904
```

在这个程序中，将浮点值 3.0 赋给了 tree。将 3.9832 赋给 int 变量 guess 导致这个值被截取为 3。将浮点型转换为整型时，C++采取截取（丢弃小数部分）而不是四舍五入（查找最接近的整数）。最后，int 变量 debt 无法存储 3.0E12，这导致 C++没有对结果进行定义的情况发生。在这种系统中，debt 的结果为 1634811904，或大约 1.6E09。

当您将整数变量初始化为浮点值时，有些编译器将提出警告，指出这可能丢掉数据。另外，对于 debt 变量，不同编译器显示的值也可能不同。例如，在另一个系统上运行该程序时，得到的值为 2147483647。

2. 表达式中的转换

当同一个表达式中包含两种不同的算术类型时，将出现什么情况呢？在这种情况下，C++将执行两种自动转换：首先，一些类型在出现时便会自动转换。其次，有些类型在与其他类型同时出现在表达式中时将被转换。

先来看看自动转换。在计算表达式时，C++将 bool、char、unsigned char、signed char 和 short 值转换为 int。具体地说，true 被转换为 1，false 被转换为 0。这些转换被称为整型提升（integral promotion）。例如，请看下面的语句：

```
short chickens = 20;          // line 1
short ducks = 35;             // line 2
short fowl = chickens + ducks; // line 3
```

为执行第 3 行语句，C++程序取得 chickens 和 ducks 的值，并将它们转换为 int。然后，程序将结果转换为 short 类型，因为结果将被赋给一个 short 类型的变量。这种说法可能有点拗口，但是情况确实如此。通常将 int 类型选择为计算机最自然的类型，这意味着计算机使用这种类型时，运算速度可能最快。

还有其他一些整型提升：如果 short 比 int 短，则 unsigned short 类型将被转换为 int；如果两种类型的长度相同，则 unsigned short 类型将被转换为 unsigned int。这种规则确保了在对 unsigned short 进行提升时不会损失数据。

同样，wchar_t 被提升成为下列类型中第一个宽度足够存储 wchar_t 取值范围的类型：int、unsigned int、long 或 unsigned long。

将不同类型进行算术运算时，也会进行一些转换，例如将 int 和 float 相加时。当运算涉及到两种类型时，较小的类型将被转换为较大的类型。例如，程序清单 3.11 中的程序用 9.0 除以 5。由于 9.0 的类型为 double，因此程序在用 5 除之前，将 5 转换为 double 类型。总之，编译器通过校验表来确定在算术表达式中执行的转换。下面是一个列表，编译器将依次查阅该列表：

① 如果有一个操作数的类型是 long double，则将另一个操作数转换为 long double。

② 否则，如果有一个操作数的类型是 double，则将另一个操作数转换为 double。

③ 否则，如果有一个操作数的类型是 float，则将另一个操作数转换为 float。

④ 否则，说明操作数都是整型，因此执行整型提升。

⑤ 在这种情况下，如果有一个操作数的类型是 unsigned long，则将另一个操作数转换为 unsigned long。

⑥ 否则，如果一个操作数是 long int，而另一个操作数是 unsigned int，则转换取决于两种类型的相对长度。如果 long 能够表示 unsigned int 的所有可能值，则将 unsigned int 转换为 long。

⑦ 否则，将两个操作数都转换为 unsigned long。

⑧ 否则，如果一个操作数是 long，则将另一个操作数转换为 long。

⑨ 否则，如果一个操作数是 unsigned int，则将另一个操作数转换为 unsigned int。

⑩ 如果编译器到达此处，则说明两个操作数都是 int 类型。

ANSI C 遵循的规则与 C++相同，但传统 K&R C 的规则稍有不同。例如，传统 C 语言总是将 float 提升为 double，即使两个操作数都是 float。

3. 传递参数时的转换

正如第 7 章将介绍的，传递参数时的类型转换通常由 C++函数原型控制。不过，也可以取消原型对参数传递的控制，尽管这样做并不明智。在这种情况下，C++将对 char 和 short 类型（signed 和 unsigned）应用整型提升。另外，为保持与传统 C 语言中大量代码的兼容性，在将参数传递给取消原型对参数传递控制的函数时，C++将 float 参数提升为 double。

4. 强制类型转换

C++还允许通过强制类型转换机制显式地进行类型转换（C++认识到，必须有类型规则，而有时又需要推翻这些规则）。强制类型转换的格式有两种。例如，为将存储在变量 thorn 中的 int 值转换为 long 类型，可以使用下述表达式中的一种：

```
(long) thorn     // returns a type long conversion of thorn
long (thorn)     // returns a type long conversion of thorn
```

强制类型转换不会修改 thorn 变量本身，而是创建一个新的、指定类型的值，可以在表达式中使用这个值。

```
cout << int('Q');  // displays the integer code for 'Q'
```

强制转换的通用格式如下：

```
(typeName) value      // converts value to typeName type
typeName  (value)     // converts value to typeName type
```

第一种格式来自 C 语言，第二种格式是纯粹的 C++。新格式的想法是，要让强制类型转换就像是函数调用。这样对内置类型的强制类型转换就像是为用户定义的类设计的类型转换。

C++还引入了 4 个强制类型转换操作符，对它们的使用要求更为严格，这将在第 15 章介绍。在这 4 个操作符中，static_cast<>可用于将值从一种数值类型转换为另一种数值类型。例如，可以像下面这样将 thorn 转换为 long 类型：

```
static_cast<long> (thorn)     // returns a type long conversion of thorn
```

推而广之，可以这样做：

```
static_cast<typeName> (value)        // converts value to typeName type
```

Stroustrup 认为，C 语言式的强制类型转换由于有过多的可能性而极其危险，这将在第 15 章更深入地讨论。

程序清单 3.14 演示了这两种格式。可以将该程序第一部分是想象为一个功能强大的生态模拟程序的一部分，该程序执行浮点计算，结果被转换为鸟和动物的数目。得到的结果取决于何时进行转换。计算 auks 时，首先将浮点值相加，然后在赋值时，将总数转换为 int。但计算 bats 和 coots 时，首先通过强制类型转换将浮点值转换为 int，然后计算总和。程序的最后一部分演示了如何通过强制类型转换来显示 char 值的 ASCII 码。

程序清单 3.14 typecast.cpp

```
// typecast.cpp -- forcing type changes
#include <iostream>
int main()
{
    using namespace std;
    int auks, bats, coots;

    // the following statement adds the values as double,
    // then converts the result to int
    auks = 19.99 + 11.99;

    // these statements add values as int
    bats = (int) 19.99 + (int) 11.99;   // old C syntax
    coots = int (19.99) + int (11.99);  // new C++ syntax
    cout << "auks = " << auks << ", bats = " << bats;
    cout << ", coots = " << coots << endl;
    char ch = 'Z';
    cout << "The code for " << ch << " is ";    // print as char
    cout << int(ch) << endl;                    // print as int
    return 0;
}
```

下面是该程序的运行结果：

```
auks = 31, bats = 30, coots = 30
The code for Z is 90
```

首先，将 19.99 和 11.99 相加，结果为 31.98。将这个值赋给 int 变量 auks 时，它被截短为 31。但在进行加法运算之前使用强制类型转换时，这两个值将被截短为 19 和 11，因此 bats 和 coots 的值都为 30。最后的 cout 语句使用强制类型转换将一个 char 类型的值转换为 int，然后显示它。这样，cout 将把值打印为整数，而不是字符。

该程序指出了使用强制类型转换的两个原因。首先，可能有一些值被存储为 double 类型，但要使用它们来计算得到一个 int 类型的值。例如，可能要用浮点数来对齐网格或者模拟整数值（如人口）。程序员可能希望在计算时将值视为 int，强制类型转换允许直接这样做。注意，将值转换为 int，然后相加得到的结果，与先将值相加，然后转换为 int 是不同的，至少对于这些值来说是不同的。

程序的第二部分指出了最常见的使用强制类型转换的原因——使一种格式的数据能够满足不同的期望。例如，在程序清单 3.14 中，char 变量 ch 存储的是字母 Z 的编码。将 cout 用于 ch 将显示字符 Z，因为 ch 的类型为 char。但通过将 ch 强制转换为 int 类型，cout 将采用 int 模式，从而打印存储在 ch 中的 ASCII 码。

3.5 总结

C++的基本类型分为两组：一组由存储为整数的值组成，另一组由存储为浮点格式的值组成。整型之间通过存储值时使用的内存量及有无符号来区分。整型从最小到最大依次是：bool、char、signed char、unsigned char、short、unsigned short、int、unsigned int、long 和 unsigned long。还有一种 wchar_t 类型，它在这个序列中的位置取决于实现。C++确保了 char 足够大，能够存储系统基本字符集中的任何成员，而 wchar_t 则可以存储系统扩展字符集中的任意成员，short 至少为 16 位，而 int 至少与 short 一样长，long 至少为 32 位，且至少和 int 一样长。确切的长度取决于实现。

字符通过其数值编码来表示。I/O 系统决定了编码是被解释为字符还是数字。

浮点类型可以表示小数值以及比整型能够表示的值大得多的值。3 种浮点类型分别是 float、double 和 long double。C++确保 float 不比 double 长，而 double 不比 long double 长。通常，float 使用 32 位内存，double 使用 64 位，long double 使用 80 到 128 位。

通过提供各种长度不同、有符号或无符号的类型，C++使程序员能够根据特定的数据要求选择合适的类型。

C++使用操作符来提供对数字类型的算术运算：加、减、乘、除和求模。当两个操作符对同一个操作数进行操作时，C++的优先级和结合性规则可以确定先执行哪种操作。

对变量赋值、在运算中使用不同类型、使用强制类型转换时，C++将把值从一种类型转换为另一种类型。很多类型转换都是“安全的”，即可以在不损失和改变数据的情况下完成转换。例如，可以把 int 值转换为 long 值，而不会出现任何问题。对于其他一些转换，如将浮点类型转换为整型，则需要更加小心。

开始，读者可能觉得大量的 C++基本类型有些多余，尤其是考虑到各种转换规则时。但是很可能最终将发现，某些时候，只有一种类型是需要的，此时应感谢 C++提供了这种类型。

3.6　复习题

1. 为什么 C++有多种整型？
2. 声明与下述描述相符的变量：

a. short 整数，值为 80；

b. unsigned int 整数，值为 42110；

c. 值为 3000000000 的整数。

3. C++提供了什么措施来防止超出整型的范围？
4. 33L 与 33 之间有什么区别？
5. 下面两条 C++语句是否等价？

```
char grade = 65;
char grade = 'A';
```

6. 如何使用 C++来找出编码 88 表示的字符？指出至少两种方法。
7. 将 long 值赋给 float 会导致舍入误差，将 long 赋给 double 呢？
8. 下列 C++表达式的结果分别是多少？

a. 8 * 9 + 2

b. 6 * 3 / 4

c. 3 / 4 * 6

d. 6.0 * 3 / 4

e. 15 % 4

9. 假设 x1 和 x2 是两个类型为 double 的变量，而要将它们作为整数，并相加，然后将结果赋给一个整型变量。请编写一条完成这项任务的 C++语句。

3.7　编程练习

1. 编写一个小程序，要求用户使用一个整数指出自己的身高（单位为英寸），然后将身高转换为英尺和英寸。该程序使用下划线字符来指示输入位置。另外，使用一个 const 符号常量来表示转换因子。

2. 编写一个小程序，要求以几英尺几英寸的方式输入其身高，并以磅为单位输入其体重（使用 3 个变量来存储这些信息）。该程序报告其 BMI（Body Mass Index，体重指数）。为了计算 BMI，该程序以英寸的方式指出用户的身高（1 英尺为 12 英寸），并将以英寸为单位的身高转换为以米为单位的身高（1 英寸=0.0254 米）。然后，将以磅为单位的体重转换为以千克为单位的体重（1 千克=2.2 磅）。最后，计算相应的 BMI——体重（千克）除以身高（米）的平方。用符号常量表示各种转换因子。

3. 编写一个程序，要求用户以度、分、秒的方式输入一个纬度，然后以度为单位显示该纬度。1 度为

60 分，1 分等于 60 秒，请以符号常量的方式表示这些值。对于每个输入值，应使用一个独立的变量存储它。下面是该程序运行时的情况：

```
Enter a latitude in degrees, minutes, and seconds:
First, enter the degrees: 37
Next, enter the minutes of arc: 51
Finally, enter the seconds of arc: 19
37 degrees, 51 minutes, 19 seconds = 37.8553 degrees
```

4. 编写一个程序，要求用户以整数方式输入秒数（使用 long 变量存储），然后以天、小时、分钟和秒的方式显示这段时间。使用符号常量来表示每天有多少小时、每小时有多少分钟以及每分钟有多少秒。该程序的输出应与下面类似：

```
Enter the number of seconds: 31600000
31600000 seconds = 365 days, 46 minutes, 40 seconds
```

5. 编写一个程序，要求用户输入驱车里程（英里）和使用汽油量（加仑），然后指出汽车耗油量为一加仑的里程。如果愿意，也可以让程序要求用户以公里为单位输入距离，并以升为单位输入汽油量，然后指出欧洲风格的结果——即每 100 公里的耗油量（升）。

6. 编写一个程序，要求用户按欧洲风格输入汽车的耗油量（每 100 公里消耗的汽油量（升）），然后将其转换为美国风格的耗油量——每加仑多少英里。注意，除了使用不同的单位计量外，美国方法（距离/燃料）与欧洲风格（燃料/距离）相反。100 公里等于 62.14 英里，1 加仑等于 3.875 升。因此，19mpg 大约合 12.4l/100km，l27mpg 大约合 8.71/100km。

第 4 章　复合类型

本章内容包括：

- 如何创建和使用数组。
- 如何创建和使用 C-风格字符串。
- 如何创建和使用 string 类字符串。
- 如何使用 getline()和 get()方法来读取字符串。
- 如何混合输入字符串和数字。
- 如何创建和使用结构。
- 如何创建和使用共用体。
- 如何创建和使用枚举。
- 如何创建和使用指针。
- 如何通过 new 和 delete 来管理动态内存。
- 如何创建动态数组。
- 如何创建动态结构。
- 自动存储、静态存储和动态存储。

假设开发了一个名叫 User-Hostile 的计算机游戏，玩家需要用智慧来应对一个神秘的、险恶的计算机界面。现在，必须编写一个程序来跟踪 5 年来游戏每月的销售量，或者希望盘点一下与黑客英雄累积的较量回合。您很快发现，需要一些比 C++的简单基本类型更复杂的东西，才能满足这些数据的要求，C++也提供了这样的东西——复合类型。这种类型是基于基本整型和浮点类型创建的。影响最为深远的复合类型是类，它是我们正要学习的 OOP 的堡垒。不过，C++还支持几种更普通的复合类型，它们都来自 C 语言。例如，数组可以存储多个同类型的值。一种特殊的数组可以存储字符串（一系列字符）。结构可以存储多个不同类型的值。而指针则是一种将数据所处位置告诉计算机的变量。本章将介绍所有这些复合类型（类除外），还将介绍 new 和 delete 及如何使用它们来管理数据。另外，还将简要地介绍 string 类，它提供了另一种处理字符串的途径。

4.1　数组

数组（array）是一种数据格式，能够存储多个同类型的值。例如，数组可以存储 60 个 int 类型的值（这些值表示游戏 5 年来的销售量）、12 个 short 值（这些值表示每个月的天数）或 365 个 float 值（这些值指出一年中每天在食物方面的开销）。每个值都存储在一个独立的数组元素中，计算机在内存中依次存储数组的各个元素。

要创建数组，可使用声明语句。数组声明应指出以下三点：

- 存储在每个元素中的值的类型。

- 数组名。
- 数组中的元素数。

在C++中，可以通过修改简单变量的声明，添加中括号（其中包含元素数目）来完成数组声明。例如，下面的声明：

```
short months[12];    // creates array of 12 short
```

创建一个名为 months 的数组，该数组有 12 个元素，每个元素都可以存储一个 short 类型的值。事实上，可以将数组中的每个元素看作是一个简单变量。

声明数组的通用格式如下：

```
typeName arrayName[arraySize];
```

表达式 arraySize 指定元素数目，它必须是整型常数（如 10）或 const 值，也可以是常量表达式（如 8 * sizeof（int）），即其中所有的值在编译时都是已知的。具体地说，arraySize 不能是变量，变量的值是在程序运行时设置的。不过，本章稍后将介绍如何使用 new 操作符来避开这种限制。

作为复合类型的数组

数组之所以被称为复合类型，是因为它是使用其他类型来创建的（C 语言使用术语“派生类型”，但由于 C++对类关系使用术语“派生”，所以它必须创建一个新术语）。不能仅仅将某种东西声明为数组，它必须是特定类型的数组。没有通用的数组类型，但存在很多特定的数组类型，如 char 数组或 long 数组。例如，请看下面的声明：

```
float loans[20];
```

loans 的类型不是“数组”，而是“float 数组”。这强调了 loans 数组是使用 float 类型创建的。

数组的很多用途都是基于这样一个事实：可以单独访问数组元素。方法是使用下标或索引来对元素进行编号。C++数组从 0 开始编号（这没有商量的余地，必须从 0 开始。Pascal 和 BASIC 用户必须调整习惯）。C++使用带索引的方括号表示法来指定数组元素。例如，months[0]是 months 数组的第一个元素，months[11]是最后一个元素。注意，最后一个元素的索引比数组长度小 1（参见图 4.1）。因此，数组声明能够使用一个声明创建大量的变量，然后便可以用索引来标识和访问各个元素。

有效下标值的重要性

编译器不会检查使用的下标是否有效。例如，如果将一个值赋给不存在的元素 months[101]，编译器并不会指出错误。但是程序运行后，这种赋值可能引发问题，它可能破坏数据或代码，也可能导致程序异常终止。所以必须确保程序只使用有效的下标值。

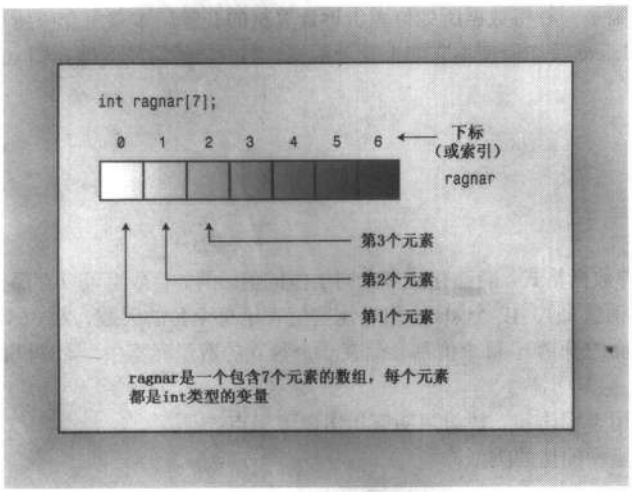


图 4.1 创建数组

程序清单 4.1 中的马铃薯分析程序说明了数组的一些属性，包括声明数组、给数组元素赋值以及初始化数组。

程序清单 4.1　arrayone.cpp

```
// arrayone.cpp -- small arrays of integers
#include <iostream>
int main()
{
    using namespace std;
    int yams[3];    // creates array with three elements
    yams[0] = 7;    // assign value to first element
    yams[1] = 8;
    yams[2] = 6;

    int yamcosts[3] = {20, 30, 5}; // create, initialize array
// NOTE: If your C++ compiler or translator can't initialize
// this array, use static int yamcosts[3] instead of
// int yamcosts[3]

    cout << "Total yams = ";
    cout << yams[0] + yams[1] + yams[2] << endl;
    cout << "The package with " << yams[1] << " yams costs ";
    cout << yamcosts[1] << " cents per yam.\n";
    int total = yams[0] * yamcosts[0] + yams[1] * yamcosts[1];
    total = total + yams[2] * yamcosts[2];
    cout << "The total yam expense is " << total << " cents.\n";

    cout << "\nSize of yams array = " << sizeof yams;
    cout << " bytes.\n";
    cout << "Size of one element = " << sizeof yams[0];
    cout << " bytes.\n";
    return 0;
}
```

注意：当前的 C++版本以及 ANSI C 都允许对函数中定义的常规数组进行初始化。不过，在一些使用 C++翻译器（而不是真正的编译器）的老式实现中，C++翻译器将为不完全与 ANSI C 兼容的 C 编译器创建 C 代码。在这种情况下，将出现一条错误消息，该消息与来自 Sun C++2.0 系统的下述消息类似：

```
"arrayone.cc", line 10: sorry, not implemented: initialization of
yamcosts (automatic aggregate) Compilation failed
```

补救办法是在数组声明中使用关键字 static：

```
// pre-ANSI initialization
static int yamcosts[3] = {20, 30, 5};
```

关键字 static 导致编译器使用另一种内存方案来存储数组，即使在 ANSI C 之前的版本中，这种方案也允许初始化。第 9 章将讨论 static 的这种用法。

下面是该程序的输出：

```
Total yams = 21
The package with 8 yams costs 30 cents per yam.
The total yam expense is 410 cents.
Size of yams array = 12 bytes.
Size of one element = 4 bytes.
```

1. 程序说明

该程序首先创建一个名为 yams 的、包含 3 个元素的数组。由于 yams 有 3 个元素，它们的编号为 0-2，因此 arrayone.cpp 使用索引 0～2 分别给这三个元素赋值。Yam 的每个元素都是 int，都有 int 类型的权力和特权，因此 arrayone.cpp 能够将值赋给元素、将元素相加和相乘，并显示它们。

程序给 yam 的元素赋值时，绕了一个大弯。C++允许在声明语句中初始化数组元素。程序清单 4.1 使

用这种捷径来给 yamcosts 数组赋值：

```
int yamcosts[3] = {20, 30, 5};
```

只需提供一个用逗号分隔的值列表（初始化列表），并将它们用花括号括起即可。列表中的空格是可选的。如果没有初始化函数中定义的数组，则其元素值将是不确定的，这意味着元素的值为以前驻留在该内存单元中的值。

接下来，程序使用数组值进行一些计算。程序的这部分由于包含了下标和括号，所以看上去有些混乱。第 5 章将介绍 for 循环，它可以提供一种功能强大的方法来处理数组，因而不用显式地书写每个索引。同时，我们仍然坚持使用小型数组。

读者可能还记得，sizeof 操作符返回类型或数据对象的长度（单位为字节）。注意，如果将 sizeof 操作符用于数组名，得到的将是整个数组中的字节数。但如果将 sizeof 用于数组元素，则得到的将是元素的长度（单位为字节）。这表明 yams 是一个数组，而 yams[1]只是一个 int 变量。

2. 数组的初始化规则

C++有几条关于初始化数组的规则，它们限制了初始化的时刻，决定了数组的元素数目与初始化器中值的数目不相同时将发生的情况。我们来看看这些规则。

只有在定义数组时才能使用初始化，此后就不能使用了，也不能将一个数组赋给另一个数组：

```
int cards[4] = {3, 6, 8, 10};          // okay
int hand[4];                           // okay
hand[4] = {5, 6, 7, 9};                // not allowed
hand = cards;                          // not allowed
```

不过，可以使用下标分别给数组中的元素赋值。

初始化数组时，提供的值可以少于数组的元素数目。例如，下面的语句只初始化 hotelTips 的前两个元素：

```
float hotelTips[5] = {5.0, 2.5};
```

如果只对数组的一部分进行初始化，则编译器将把其他元素设置为 0。因此，将数组中所有的元素都初始化为 0 非常简单——只要显式地将第一个元素初始化为 0，然后让编译器将其他元素都初始化为 0 即可：

```
long totals[500] = {0};
```

如果初始化为{1}而不是{0}，则第一个元素被设置为 1，其他元素都被设置为 0。

如果初始化数组时方括号内（[]）为空，C++编译器将计算元素个数。例如，对于下面的声明：

```
short things[] = {1, 5, 3, 8};
```

编译器将使 things 数组包含 4 个元素。

让编译器去做

通常，让编译器计算元素个数是一种很糟的做法，因为其计数可能与您想象的不一样。不过，这种方法对于将字符数组初始化为一个字符串来说比较安全，很快读者将明白这一点。如果主要关心的问题是程序，而不是自己是否知道数组的大小，则可以这样做：

```
short things[] = {1, 5, 3, 8};
int num_elements = sizeof things / sizeof (short);
```

这样做是有用还是偷懒取决于具体情况。

C++标准模板库（STL）一种数组替代品——模板类 vector，它比内置复合类型数组更复杂，也更灵活。第 16 章将讨论 STL 和模板类 vector。

4.2 字符串

字符串是存储在内存的连续字节中的一系列字符。C++处理字符串的方式有两种。第一种来自 C 语言，常被称为 C-风格字符串（C-style string）。本章将首先介绍它，然后介绍另一种基于 string 类库的方法。

存储在连续字节中的一系列字符意味着可以将字符串存储在 char 数组中，其中每个字符都位于自己的数组元素中。字符串提供了一种存储文本信息的便捷方式，如提供给用户的消息（“请告诉我您的瑞士银行账号。”）或来自用户的响应（“您肯定在开玩笑。”）。C-风格字符串具有一种特殊的性质：以空字符（null character）结尾，空字符被写作\0，其 ASCII 码为 0，用来标记字符串的结尾。例如，请看下面两个声明：

```
char dog [5] = {'b', 'e', 'a', 'u', 'x'};     // not a string!
char cat[5] = {'f', 'a', 't', 's', '\0'};     // a string!
```

这两个数组都是 char 数组，但只有第二个数组是字符串。空字符对 C-风格字符串而言至关重要。例如，C++有很多处理字符串的函数，其中包括 cout 使用的那些函数。它们都逐个地处理字符串中的字符，直到到达空字符为止。如果使用 cout 显示上面的 cat 这样的字符串，则将显示前 4 个字符，发现空字符后停止。但是，如果使用 cout 显示上面的 dog 数组（它不是字符串），cout 将打印出数组中的 5 个字母，并接着将内存中随后的各个字节解释为要打印的字符，直到遇到空字符为止。由于空字符（实际上是被设置为 0 的字节）在内存中很常见，因此这一过程将很快停止。但尽管如此，还是不应将不是字符串的字符数组当作字符串来处理。

在 cat 数组范例中，将数组初始化为字符串的工作看上去冗长乏味——使用大量单引号，且必须记住加上空字符。不必担心，有一种更好的、将字符数组初始化为字符串的方法——只需使用一个用引号括起的字符串即可，这种字符串被称为字符串常量（string constant）或字符串字面值（string literal），如下所示：

```
char bird[10] = "Mr. Cheeps";   // the \0 is understood
char fish[] = "Bubbles";        // let the compiler count
```

用引号括起的字符串隐式地包括结尾的空字符，因此不用显式地包括它（参见图 4.2）。另外，各种 C++输入工具通过键盘输入，将字符串读入到 char 数组中时，将自动加上结尾的空字符（如果在运行程序清单 4.1 中的程序时发现，必须使用关键字 static 来初始化数组，则初始化上述 char 数组时也必须使用该关键字）。

当然，应确保数组足够大，能够存储字符串中所有字符——包括空字符。使用字符串常量初始化字符数组是这样的一种情况，即让编译器计算元素数目更为安全。让数组比字符串长没有什么害处，只是会浪费一些空间而已。这是因为处理字符串的函数根据空字符的位置，而不是数组长度来进行处理。C++对字符串长度没有限制。

记住：*在确定存储字符串所需的最短数组时，别忘了将结尾的空字符计算在内。*

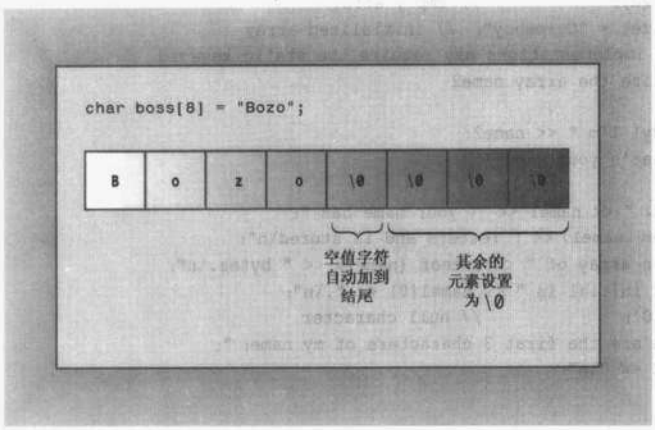


图 4.2　将数组初始化为字符串

注意，字符串常量（使用双引号）不能与字符常量（使用单引号）互换。字符常量（如'S'）是字符串编码的简写表示。在 ASCII 系统上，'S'只是 83 的另一种写法。因此，下面的语句：

```
char shirt_size = 'S';          // this is fine
```

将 83 赋给 shirt_size。但是"S"表示的是两个字符（S 和\0 字符）组成的字符串。更糟糕的是，"S"实际

上表示的是字符串所在的内存地址。因此下面的语句：

```
char shirt_size = "S";        // illegal type mismatch
```

将内存地址赋给 shirt_size！由于地址在 C++中是一种独立的类型，因此 C++编译器不允许这种不合理的做法（本章后面讨论指针后，将回过头来讨论这个问题）。

4.2.1 拼接字符串常量

有时候，字符串很长，无法放到一行中。C++允许拼接字符串常量，即将两个用引号括起的字符串合并为一个。事实上，任何两个由空白（空格、制表符和换行符）分隔的字符串常量都将自动拼接成一个。因此，下面所有的输出语句都是等效的：

```
cout << "I'd give my right arm to be" " a great violinist.\n";
cout << "I'd give my right arm to be a great violinist.\n";
cout << "I'd give my right ar"
"m to be a great violinist.\n";
```

注意，拼接时不会在被连接的字符串之间添加空格，第二个字符串的第一个字符将紧跟在第一个字符串的最后一个字符（不考虑\0）后面。第一个字符串中的\0 字符将被第二个字符串的第一个字符取代。

4.2.2 在数组中使用字符串

要将字符串存储到数组中，最常用的方法有两种——将数组初始化为字符串常量、将键盘或文件输入读入到数组中。程序清单 4.2 演示了这两种方法，它将一个数组初始化为用引号括起的字符串，并使用 cin 将一个输入字符串放到另一个数组中。该程序还使用了标准库函数 strlen()来确定字符串的长度。标准头文件 cstring（老式实现为 string.h）提供了该函数以及很多与字符串相关的其他函数的声明。

程序清单 4.2 string.cpp

```
// strings.cpp -- storing strings in an array
#include <iostream>
#include <cstring>  // for the strlen()function
int main()
{
    using namespace std;
    const int Size = 15;
    char name1[Size];             // empty array
    char name2[Size] = "C++owboy";  // initialized array
    // NOTE: some implementations may require the static keyword
    // to initialize the array name2

    cout << "Howdy! I'm " << name2;
    cout << "! What's your name?\n";
    cin >> name1;
    cout << "Well, " << name1 << ", your name has ";
    cout << strlen(name1) << " letters and is stored\n";
    cout << "in an array of " << sizeof (name1) << " bytes.\n";
    cout << "Your initial is " << name1[0] << ".\n";
    name2[3] = '\0';              // null character
    cout << "Here are the first 3 characters of my name: ";
    cout << name2 << "\n";
    return 0;
}
```

注意：如果系统没有提供 cstring 头文件，请使用老版本的 string.h。

下面是该程序的运行情况：

```
Howdy! I'm C++owboy! What's your name?
Basicman
Well, Basicman, your name has 8 letters and is stored
```

```
in an array of 15 bytes.
Your initial is B.
Here are the first 3 characters of my name: C++
```

程序说明

从程序清单 4.2 中可以学到什么呢？首先，sizeof 操作符指出整个数组的长度：15 字节，但 strlen()函数返回的是存储在数组中的字符串的长度，而不是数组本身的长度。另外，strlen()只计算可见的字符，而不把空字符计算在内。因此，对于 Basicman，返回的值为 8，而不是 9。如果 cosmic 是字符串，则要存储该字符串，数组的长度不能短于 strlen（cosmic）+1。

由于 name1 和 name2 是数组，所以可以用索引来访问数组中各个字符。例如，该程序使用 name1[0]找到数组的第一个字符。另外，该程序将 name2[3]设置为空字符。这使得字符串在第 3 个字符后即结束，虽然数组中还有其他的字符（参见图 4.3）。

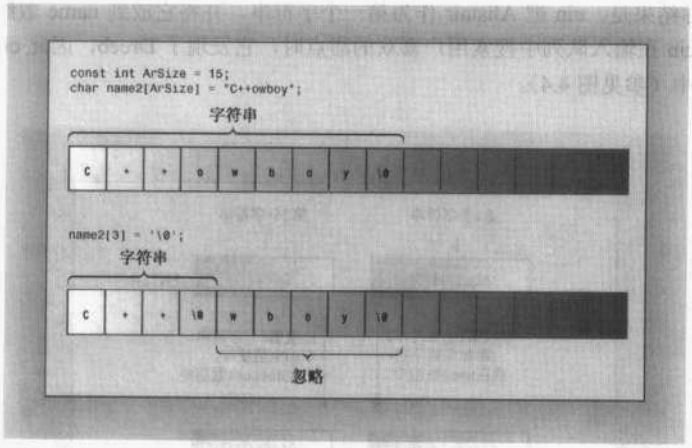


图 4.3　使用\0 截短字符串

该程序使用符号常量来指定数组的长度。程序常常有多条语句使用了数组长度。使用符号常量来表示数组长度后，当需要修改程序以使用不同的数组长度时，工作将变得更简单——只需在定义符号常量的地方进行修改即可。

4.2.3　字符串输入

程序 strings.cpp 有一个缺陷，这种缺陷通过精心选择输入被掩盖掉了。程序清单 4.3 揭开了它的面纱，揭示了字符串输入的技巧。

程序清单 4.3　instr1.cpp

```
// instr1.cpp--reading more than one string
#include <iostream>
int main()
{
    using namespace std;
    const int ArSize = 20;
    char name[ArSize];
    char dessert[ArSize];

    cout << "Enter your name: \n";
    cin >> name;
    cout << "Enter your favorite dessert: \n";
    cin >> dessert;
    cout << "I have some delicious " << dessert;
    cout << " for you, " << name << ".\n";
```

```
    return 0;
}
```

该程序的意图很简单：读取来自键盘的用户名和用户喜欢的甜点，然后显示这些信息。下面是该程序的运行情况：

```
Enter your name:
Alistair Dreeb
Enter your favorite dessert:
I have some delicious Dreeb for you, Alistair.
```

我们甚至还没有对“输入甜点的提示”做出反应，程序便把它显示出来了，然后立即显示最后一行。

cin 是如何确定已完成字符串输入呢？由于不能通过键盘输入空字符，因此 cin 需要用别的方法来确定字符串的结尾位置。cin 使用空白（空格、制表符和换行符）来定字符串的界。这意味着 cin 在获取字符数组输入时只读取一个单词。读取该单词后，cin 将该字符串放到数组中，并自动在结尾添加空字符。

这个例子的实际结果是，cin 把 Alistair 作为第一个字符串，并将它放到 name 数组中。这把 Dreeb 留在输入队列中。当 cin 在输入队列中搜索用户喜欢的甜点时，它发现了 Dreeb，因此 cin 读取 Dreeb，并将它放到 dessert 数组中（参见图 4.4）。

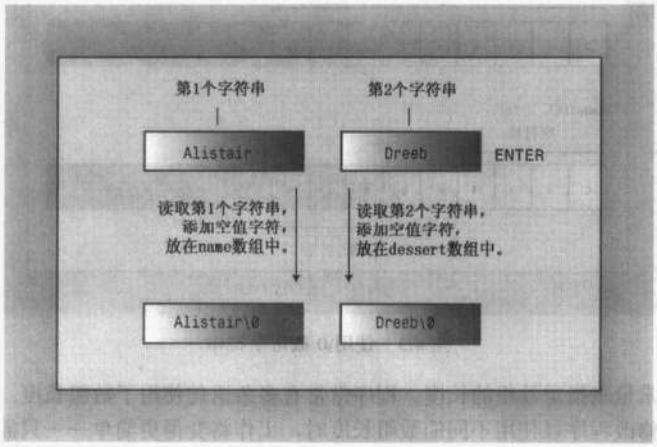


图 4.4 使用 cin 读取字符串输入时的情况

另一个问题是，输入字符串可能比目标数组长（运行中没有揭示出来）。像这个例子一样使用 cin，确实不能防止将包含 30 个字符的字符串放到 20 个字符的数组中的情况发生。

很多程序都依赖于字符串输入，因此有必要对这一主题做进一步的探讨。我们必须使用 cin 较高级的特性，这将在第 17 章介绍。

4.2.4 每次读取一行字符串输入

每次读取一个单词通常不是最好的选择。例如，假设程序要求用户输入城市名，用户输入 New York 或 Sao Paulo。您希望程序读取并存储完整的城市名，而不仅仅是 New 或 Sao。要将整条短语而不是一个单词作为字符串输入，需要采用另一种字符串读取方法。具体地说，需要采用面向行而不是面向单词的方法。幸运的是，istream 中的类（如 cin）提供了一些面向行的类成员函数：getline()和 get()。这两个函数都读取一行输入，直到到达换行符。然而，随后 getline()将丢弃换行符，而 get()将换行符保留在输入序列中。下面详细介绍它们，首先介绍 getline()。

1. 面向行的输入：getline()

getline()函数读取整行，它使用通过回车键输入的换行符来确定输入结尾。要调用这种方法，可以使用 cin.getline()。该函数有两个参数。第一个参数是用来存储输入行的数组的名称，第二个参数是要读取的

字符数。如果这个参数为 20，则函数最多读取 19 个字符，余下的空间用于存储自动在结尾处添加的空字符。getline()成员函数在读取指定数目的字符或遇到换行符时停止读取。

例如，假设要使用 getline()将姓名读入到一个包含 20 个元素的 name 数组中。可以使用这样的函数调用：

```
cin.getline (name, 20);
```

这将把一行读入到 name 数组中——如果这行包含的字符不超过 19 个（getline()成员函数还可以接受第三个可选参数，这将在第 17 章讨论）。

程序清单 4.4 将程序清单 4.3 修改为使用 cin.getline()，而不是简单的 cin。除此之外，该程序没有做其他修改。

程序清单 4.4　instr2.cpp

```
// instr2.cpp -- reading more than one word with getline
#include <iostream>
int main()
{
    using namespace std;
    const int ArSize = 20;
    char name[ArSize];
    char dessert[ArSize];

    cout << "Enter your name: \n";
    cin.getline (name, ArSize);  // reads through newline
    cout << "Enter your favorite dessert: \n";
    cin.getline (dessert, ArSize);
    cout << "I have some delicious " << dessert;
    cout << " for you, " << name << ".\n";
    return 0;
}
```

注意：有些早期的 C++版本没有实现当前 C++ I/O 程序包的全部功能。具体地说，getline()成员函数并不总是可用的。如果这对您有影响，则看一看这个例子，然后接着看下一个例子，它使用的是 getline()成员函数的前身。Turbo C++的早期版本实现 getline()的方式稍有不同，它将换行符存储在字符串中。Microsoft Visual C++ 5.0 和 6.0 在头文件 iostream 中实现 getline()时有些错误，但 ostream.h 中的实现没有问题；Service Pack 5 for Microsoft Visual C++ 6.0（可从 msdn.microsoft.com/vstdio 下载）能够修复这种错误。

下面是该程序的输出：

```
Enter your name:
Dirk Hammernose
Enter your favorite dessert:
Radish Torte
I have some delicious Radish Torte for you, Dirk Hammernose.
```

该程序现在可以读取完整的姓名以及用户喜欢的甜点！getline()函数每次读取一行。它通过换行符来确定行尾，但不保存换行符。相反，在存储字符串时，它用空字符来替换换行符（参见图 4.5）。

2. 面向行的输入：get()

我们来试试另一种方法。istream 类有另一个名为 get()的成员函数，该函数有几种变体。其中一种变体的工作方式与 getline()类似，它们接受的参数相同，解释参数的方式也相同，并且都读取到行尾。但 get 并不再读取并丢弃换行符，而是将其留在输入队列中。假设我们连续两次调用 get()：

```
cin.get (name, ArSize);
cin.get (dessert, Arsize);  // a problem
```

由于第一次调用后，换行符将留在输入队列中，因此第二次调用时看到的第一个字符便是换行符。因此 get()认为已到达行尾，而没有发现任何可读取的内容。如果不借助于帮助，get()将不能跨过该换行符。

幸运的是，get()有另一种变体。使用不带任何参数的 cin.get()调用可读取下一个字符（即使是换行符），

因此可以用它来处理换行符，为读取下一行输入做好准备。也就是说，可以采用下面的调用序列：

```
cin.get (name, ArSize) ;     // read first line
cin.get();                   // read newline
cin.get (dessert, Arsize) ; // read second line
```



图 4.5 getline()读取并替换换行符

另一种使用 get()的方式是将两个类成员函数拼接起来（合并），如下所示：

```
cin.get (name, ArSize) .get(); // concatenate member functions
```

之所以可以这样做，是由于 cin.get（name，ArSize）返回一个 cin 对象，该对象随后将被用来调用 get()函数。同样，下面的语句：

```
cin.getline (name1, ArSize) .getline (name2, ArSize) ;
```

将把输入中连续的两行分别读入到数组 name1 和 name2 中；其效果与两次调用 cin.getline()相同。

程序清单 4.5 采用了拼接方式。第 11 章将介绍如何在类定义中使用这项特性。

程序清单 4.5 instr3.cpp

```
// instr3.cpp -- reading more than one word with get()& get()
#include <iostream>
int main()
{
    using namespace std;
    const int ArSize = 20;
    char name[ArSize];
    char dessert[ArSize];

    cout << "Enter your name: \n";
    cin.get (name, ArSize) .get();    // read string, newline
    cout << "Enter your favorite dessert: \n";
    cin.get (dessert, ArSize) .get();
    cout << "I have some delicious " << dessert;
    cout << " for you, " << name << ".\n";
    return 0;
}
```

注意：有些 C++的老版本没有实现不接受任何参数的 get()变体，但实现了接受一个 char 参数的 get()变体。要使用该变体（而不是没有参数 get()），需要首先声明一个 char 变量：

```
char ch;
cin.get (name, ArSize) .get (ch) ;
```

可以用上述代码来代替程序清单 4.5 中相应的代码。第 5、6 和 17 章将深入讨论 get() 的变体。

下面是程序清单 4.5 中程序的运行情况：

```
Enter your name:
Mai Parfait
Enter your favorite dessert:
Chocolate Mousse
I have some delicious Chocolate Mousse for you, Mai Parfait.
```

需要指出的一点是，C++允许函数有多个版本，条件是这些版本的参数列表不同。如果使用的是 cin.get (name, ArSize)，则编译器知道是要将一个字符串放入数组中，因而将使用适当的成员函数。如果使用的是 cin.get()，则编译器知道是要读取一个字符。第 8 章将探索这种特性——函数重载。

为什么要使用 get()，而不是 getline()呢？首先，老式实现没有 getline()。其次，get()使输入更仔细。例如，假设用 get()将一行读入数组中。如何知道停止读取的原因是由于已经读取了整行，而不是由于数组已填满呢？查看下一个输入字符，如果是换行符，说明已读取了整行；否则，说明该行中还有其他输入。第 17 章将介绍这种技术。总之，getline()使用起来简单一些，但 get()使得检查错误更简单些。可以用其中的任何一个来读取一行输入；只是应该知道，它们的行为稍有不同。

3. 空行和其他问题

当 getline()或 get()读取空行时，将发生什么情况？最初的做法是，下一条输入语句将在前一条 getline()或 get()结束读取的位置开始读取；但当前的做法是，当 get()（不是 getline()）读取空行后将设置失效位（failbit）。这意味着接下来的输入将被阻断，但可以用下面的命令来恢复输入：

```
cin.clear();
```

另一个潜在的问题是，输入字符串可能比分配的空间长。如果输入行包含的字符数比指定的多，则 getline()和 get()将把余下的字符留在输入队列中，而 getline()还会设置失效位，并关闭后面的输入。

第 5、6 和 17 章将介绍这些属性，并探讨程序如何避免这些问题。

4.2.5　混合输入字符串和数字

混合输入数字和面向行的字符串会导致问题。请看程序清单 4.6 中的简单程序：

程序清单 4.6　numstr.cpp

```
// numstr.cpp -- following number input with line input
#include <iostream>
int main()
{
    using namespace std;
    cout << "What year was your house built?\n";
    int year;
    cin >> year;
    cout << "What is its street address?\n";
    char address[80];
    cin.getline (address, 80);
    cout << "Year built: " << year << endl;
    cout << "Address: " << address << endl;
    cout << "Done!\n";
    return 0;
}
```

该程序的运行情况如下：

```
What year was your house built?
1966
What is its street address?
Year built: 1966
```

```
Address
Done!
```

用户根本没有输入地址的机会。问题在于，当 cin 读取年份时，将回车键生成的换行符留在了输入队列中。后面的 cin.getline() 看到换行符后，将认为是一个空行，并将一个空字符串赋给 address 数组。解决之道是，在读取地址之前先读取并丢弃换行符。这可以通过几种方法来完成，其中包括使用没有参数的 get() 和使用接受一个 char 参数的 get()，如前面的例子所示。可以单独进行调用：

```
cin >> year;
cin.get();  // or cin.get(ch);
```

也可以利用表达式 cin>>year 返回 cin 对象，将调用拼接起来：

```
(cin >> year).get();  // or (cin >> year).get(ch);
```

按上述任何一种方法修改程序清单 4.6 后，它便可以正常工作：

```
What year was your house built?
1966
What is its street address?
43821 Unsigned Short Street
Year built: 1966
Address: 43821 Unsigned Short Street
Done!
```

C++程序常使用指针（而不是数组）来处理字符串。我们将在介绍指针后，再介绍字符串方面的特性。下面介绍一种较新的处理字符串的方式：C++ string 类。

4.3 string 类简介

ISO/ANSI C++标准通过添加 string 类扩展了 C++库，因此现在可以 string 类型的变量（使用 C++的话说是对象）而不是字符数组来存储字符串。读者将看到，string 类使用起来比数组简单，同时提供了将字符串作为一种数据类型的表示方法。

要使用 string 类，必须在程序中包含头文件 string。string 类位于名称空间 std 中，因此您必须提供一条 using 编译指令，或者使用 std::string 来引用它。string 类定义隐藏了字符串的数组性质，让您能够像处理普通变量那样处理字符串。程序清单 4.7 说明了 string 对象与字符数组之间的一些相同点和不同点。

程序清单 4.7 strtype1.cpp

```
// strtype1.cpp -- using the C++ string class
#include <iostream>
#include <string>               // make string class available
int main()
{
   using namespace std;
   char charr1[20];             // create an empty array
   char charr2[20] = "jaguar";  // create an initialized array
   string str1;                 // create an empty string object
   string str2 = "panther";     // create an initialized string

   cout << "Enter a kind of feline: ";
   cin >> charr1;
   cout << "Enter another kind of feline: ";
   cin >> str1;                 // use cin for input
   cout << "Here are some felines:\n";
   cout << charr1 << " " << charr2 << " "
      << str1 << ""<< str2 // use cout for output
      << endl;
   cout << "The third letter in " << charr2 << " is "
      << charr2[2] << endl;
   cout << "The third letter in " << str2 << " is "
```

```
        << str2[2] << endl;    // use array notation

    return 0;
}
```

下面是该程序的运行情况：

```
Enter a kind of feline: ocelot
Enter another kind of feline: tiger
Here are some felines:
ocelot jaguar tiger panther
The third letter in jaguar is g
The third letter in panther is n
```

从这个范例可知，在很多方面，使用 string 对象的方式与使用字符数组相同：

- 可以使用 C-风格字符串来初始化 string 对象。
- 可以使用 cin 来将键盘输入存储到 string 对象中。
- 可以使用 cout 来显示 string 对象。
- 可以使用数组表示法来访问存储在 string 对象中的字符。

程序清单 4.7 表明，string 对象和字符数组之间的主要区别是，可以将 string 对象声明为简单变量，而不是数组：

```
string str1;              // create an empty string object
string str2 = "panther";   // create an initialized string
```

类设计让程序能够自动处理 string 的大小。例如，str1 的声明创建一个长度为 0 的 string 对象，但程序将输入读取到 str1 中时，将自动调整 str1 的长度：

```
cin >> str1;              // str1 resized to fit input
```

这使得与使用数组相比，使用 string 对象更方便，也更安全。从理论上说，可以将 char 数组视为一组用于存储一个字符串的 char 存储单元，而 string 类变量是一个表示字符串的实体。

4.3.1 赋值、拼接和附加

使用 string 类时，某些操作比使用数组时更简单。例如，不能将一个数组赋给另一个数组，但可以将一个 string 对象赋给另一个 string 对象：

```
char charr1[20];          // create an empty array
char charr2[20] = "jaguar";// create an initialized array
string str1;              // create an empty string object
string str2 = "panther";  // create an initialized string
charr1 = charr2;          // INVALID, no array assignment
str1 = str2;              // VALID, object assignment ok
```

string 类简化的字符串合并操作。可以使用操作符+将两个 string 对象合并起来，还可以使用操作符+=将一个字符串附加到一个 string 对象的末尾。继续前面的代码，您可以这样做：

```
string str3;
str3 = str1 + str2;       // assign str3 the joined strings
str1 += str2;             // add str2 to the end of str1
```

程序清单 4.8 演示了这些用法。可以将 C 语言式的字符串或 string 对象与 string 对象相加，或将它们附加到 string 对象的末尾。

程序清单 4.8 strtype2.cpp

```
// strtype2.cpp -- assigning, adding, and appending
#include <iostream>
#include <string>              // make string class available
int main()
{
   using namespace std;
   string s1 = "penguin";
   string s2, s3;
```

```
    cout << "You can assign one string object to another: s2 = s1\n";
    s2 = s1;
    cout << "s1: " << s1 << ", s2: " << s2 << endl;
    cout << "You can assign a C-style string to a string object.\n";
    cout << "s2 = \"buzzard\"\n";
    s2 = "buzzard";
    cout << "s2: " << s2 << endl;
    cout << "You can concatenate strings: s3 = s1 + s2\n";
    s3 = s1 + s2;
    cout << "s3: " << s3 << endl;
    cout << "You can append strings.\n";
    s1 += s2;
    cout <<"s1 += s2 yields s1 = " << s1 << endl;
    s2 += " for a day";
    cout <<"s2 += \" for a day\" yields s2 = " << s2 << endl;

    return 0;
}
```

转义序列\"表示双引号，而不是字符串结尾。该程序的输出如下：

```
You can assign one string object to another: s2 = s1
s1: penguin, s2: penguin
You can assign a C-style string to a string object.
s2 = "buzzard"
s2: buzzard
You can concatenate strings: s3 = s1 + s2
s3: penguinbuzzard
You can append strings.
s1 += s2 yields s1 = penguinbuzzard
s2 += " for a day" yields s2 = buzzard for a day
```

4.3.2 string 类的其他操作

在C++新增string类之前，程序员也需要完成诸如给字符串赋值等工作。对于C语言式的字符串，程序员使用C语言库中的函数来完成这些任务。头文件cstring（以前为string.h）提供了这些函数。例如，可以使用函数strcpy()将字符串复制到字符数组中，使用函数strcat()将字符串附加到字符数组末尾：

```
strcpy(charr1, charr2);  // copy charr2 to charr1
strcat(charr1, charr2);  // append contents of charr2 to char1
```

程序清单4.9对用于string对象的技术和用于字符数组的技术进行了比较。

程序清单4.9 strtype3.cpp

```
// strtype3.cpp -- more string class features
#include <iostream>
#include <string>            // make string class available
#include <cstring>           // C-style string library
int main()
{
    using namespace std;
    char charr1[20];
    char charr2[20] = "jaguar";
    string str1;
    string str2 = "panther";

    // assignment for string objects and character arrays
    str1 = str2;                 // copy str2 to str2
    strcpy(charr1, charr2);      // copy charr2 to charr1

    // appending for string objects and character arrays
    str1 += " paste";            // add paste to end of str1
    strcat(charr1, " juice");    // add juice to end of charr1
```

```
    // finding the length of a string object and a C-style string
    int len1 = str1.size();    // obtain length of str1
    int len2 = strlen(charr1); // obtain length of charr1

    cout << "The string " << str1 << " contains "
       << len1 << " characters.\n";
    cout << "The string " << charr1 << " contains "
       << len2 << " characters.\n";

    return 0;
}
```

下面是该程序的输出：

```
The string panther paste contains 13 characters.
The string jaguar juice contains 12 characters.
```

处理 string 对象通常比使用 C 字符串函数简单，尤其是执行较为复杂的操作时。例如，对于下述操作：

```
str3 = str1 + str2;
```

使用 C 语言字符串时，需要使用的函数如下：

```
strcpy(charr3, charr1);
strcat(charr3, charr2);
```

另外，使用字符数组时，总是存在目标数组过小，无法存储指定信息的危险，如下面的范例所示：

```
char site[10] = "house";
strcat(site, " of pancakes"); // memory problem
```

函数 strcat()试图将全部 12 个字符复制到数组 site 中，这将覆盖相邻的内存。这可能导致程序终止，或者程序继续运行，但数据被损坏。string 类具有自动调整大小的功能，从而能够避免这种问题发生。C 函数库确实提供了与 strcat()和 strcpy()类似的函数——strncat()和 strncpy()，它们接受指出目标数组最大允许长度的第三个参数，因此更为安全，但使用它们进一步增加了编写程序的复杂度。

下面是两种确定字符串中字符数的方法：

```
int len1 = str1.size();    // obtain length of str1
int len2 = strlen(charr1); // obtain length of charr1
```

函数 strlen()是一个常规函数，它接受一个 C-风格字符串作为参数，并返回该字符串包含的字符数。函数 size()的功能基本上与此相同，但句法不同：str1 不是被用作函数参数，而是位于函数名之前，它们之间用句点连接。与第 3 章介绍的 put()方法相同，这种句法表明，str1 是一个对象，而 size()是一个类方法。方法是一个函数，只能通过其所属类的对象进行调用。在这里，str1 是一个 string 对象，而 size()是 string 类的一个方法。总之，C 函数使用参数来指出要使用哪个字符串，而 C++ string 类对象使用对象名和句点操作符来指出要使用哪个字符串。

4.3.3　string 类 I/O

正如读者知道的，可以使用 cin 和操作符<<来将输入存储到 string 对象中，使用 cout 和操作符>>来显示 string 对象，其句法与处理 C-风格字符串相同。但每次读取一行而不是一个单词时，使用的句法不同，程序清单 4.10 说明了这一点。

程序清单 4.10　strtype4.cpp

```
// strtype4.cpp -- line input
#include <iostream>
#include <string>              // make string class available
#include <cstring>             // C-style string library
int main()
{
    using namespace std;
    char charr[20];
    string str;
```

```
    cout << "Length of string in charr before input: "
    << strlen(charr) << endl;
    cout << "Length of string in str before input: "
       << str.size() << endl;
    cout << "Enter a line of text:\n";
    cin.getline(charr, 20);     // indicate maximum length
    cout << "You entered: " << charr << endl;
    cout << "Enter another line of text:\n";
    getline(cin, str);          // cin now an argument; no length specifier
    cout << "You entered: " << str << endl;
    cout << "Length of string in charr after input: "
       << strlen(charr) << endl;
    cout << "Length of string in str after input: "
       << str.size() << endl;

    return 0;
}
```

下面是一个运行该程序时的输出范例：

```
Length of string in charr before input: 27
Length of string in str before input: 0
Enter a line of text:
peanut butter
You entered: peanut butter
Enter another line of text:
blueberry jam
You entered: blueberry jam
Length of string in charr after input: 13
Length of string in str after input: 13
```

在用户输入之前，该程序指出数组 charr 中的字符串长度为 27，这比该数组的长度要大。这里有两点需要说明。首先，未初始化的数组的内容是未定义的；其次，函数 strlen() 从数组的第一个元素开始计算字节数，直到遇到空字符。在这个例子中，在数组末尾的几个字节后才遇到空字符。对于未被初始化的数据，第一个空字符的出现位置是随机的，因此读者在运行该程序时，得到的数组长度很可能与此不同。

另外，用户输入之前，str 中的字符串长度为 0。这是因为未被初始化的 string 对象的长度被自动设置为 0。

下面是将一行输入读取到数组中的代码：

```
cin.getline(charr, 20);
```

这种句点表示法表明，函数 getline() 是 istream 类的一个类方法（还记得吗，cin 是一个 istream 对象）。正如前面指出的，第一个参数是目标数组；第二个参数数组长度，getline() 使用它来避免超越数组的边界。

下面是将一行输入读取到 string 对象中的代码：

```
getline(cin,str);
```

这里没有使用句点表示法，这表明这个 getline() 不是类方法。它将 cin 作为参数，指出到哪里去查找输入。另外，也没有指出字符串长度的参数，因为 string 对象将根据字符串的长度自动调整自己的大小。

那么，为何一个 getline() 是 istream 的类方法，而另一个不是呢？在引入 string 类之前很久，C++就有 istream 类。因此 istream 的设计考虑到了诸如 double 和 int 等基本 C++数据类型，但没有考虑 string 类型，所以 istream 类中，有处理 double、int 和其他基本类型的类方法，但没有处理 string 对象的类方法。

由于 istream 类中没有处理 string 对象的类方法，因此读者可能会问，下述代码为何可行呢？

```
cin >> str;  // read a word into the str string object
```

像下面这样的代码：

```
cin >> x;  // read a value into a basic C++ type
```

使用 istream 类的一个成员函数，但前面处理 string 对象的代码使用 string 类的一个友元函数。有关友元函数及这种技术为何可行，将在第 11 章介绍。另外，读者可以将 cin 和 cout 用于 string 对象，而不用考虑其内部工作原理。

接下来介绍另一种复合类型——结构。

4.4　结构简介

假设要存储有关篮球运动员的信息，则可能需要存储他（她）的姓名、工资、身高、体重、平均得分、命中率、助攻次数等。希望有一种数据格式可以将所有这些信息存储在一个单元中。数组不能完成这项任务，因为虽然数组可以存储多个元素，但所有元素的类型必须相同。也就是说，一个数组可以存储 20 个 int，另一个数组可以存储 10 个 float，但同一个数组不能在一些元素中存储 int，在另一些元素中存储 float。

C++中的结构可以满足要求（存储篮球运动员的信息）。结构是一种比数组更灵活的数据格式，因为同一个结构可以存储多种类型的数据，这使得能够将有关篮球运动员的信息放在一个结构中，从而将数据的表示合并到一起。如果要跟踪整个球队，则可以使用结构数组。结构也是 C++ OOP 堡垒（类）的基石。学习有关结构的知识将使我们离 C++的核心 OOP 更近。

结构是用户定义的类型，而结构声明定义了这种类型的数据属性。定义了类型后，便可以创建这种类型的变量。因此创建结构包括两步。首先，定义结构描述——它描述并标记了能够存储在结构中的各种数据类型。然后按描述创建结构变量（结构数据对象）。

例如，假设 Bloataire 公司要创建一种类型来描述其生产线上充气产品的成员。具体地说，这种类型应存储产品名称、容量（单位为立方英尺）和售价。下面的结构描述能够满足这些要求：

```
struct inflatable    // structure description
{
    char name[20];
    float volume;
    double price;
};
```

关键字 struct 表明，这些代码定义的是一个结构的布局。标识符 inflatable 是这种数据格式的名称，因此新类型的名称（或标记）为 inflatable。这样，便可以像创建 char 或 int 类型的变量那样创建 inflatable 类型的变量了。接下来的大括号中包含的是结构存储的数据类型的列表，其中每个列表项都是一条声明语句。这个例子使用了一个适合用于存储字符串的 char 数组、一个 float 和一个 double。列表中的每一项都被称为结构成员，因此 infatable 结构有 3 个成员（参见图 4.6）。

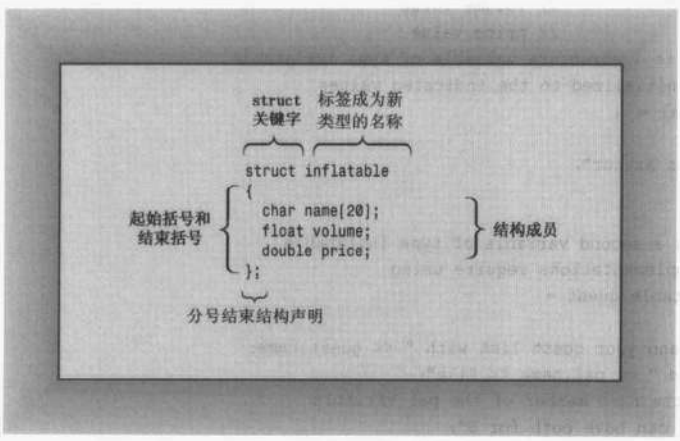


图 4.6　结构描述的组成部分

有了模板后，便可以创建这种类型的变量了：

```
inflatable hat;                 // hat is a structure variable of type inflatable
inflatable woopie_cushion;      // type inflatable variable
inflatable mainframe;           // type inflatable variable
```

如果读者熟悉 C 语言中的结构，则可能已经注意到了，C++允许在声明结构变量时省略关键字 struct：

```
struct inflatable goose;        // keyword struct required in C
inflatable vincent;             // keyword struct not required in C++
```

在 C++中，结构标记的用法与基本类型名相同。这种变化强调的是，结构声明定义了一种新类型。在 C++中，省略 struct 不会出错。

由于 hat 的类型为 inflatable，因此可以使用成员操作符（.）来访问各个成员。例如，hat.volume 指的是结构的 volume 成员，hat.price 指的是 price 成员。同样，vincent.price 是 vincent 变量的 price 成员。总之，通过成员名能够访问结构的成员，就像通过索引能够访问数组的元素一样。由于 price 成员被声明为 double 类型，因此 hat.price 和 vincent.price 相当于是 double 类型的变量，可以像使用常规 double 变量那样来使用它们。总之，hat 是一个结构，而 hat.price 是一个 double 变量。顺便说一句，访问类成员函数（如 cin.getline()）的方式是从访问结构成员变量（如 vincent.price）的方式衍生而来的。

4.4.1 在程序中使用结构

介绍结构的主要特征后，下面在一个使用结构的程序中使用这些概念。程序清单 4.11 说明了有关结构的这些问题，还演示了如何初始化结构

程序清单 4.11 structur.cpp

```
// structur.cpp -- a simple structure
#include <iostream>
struct inflatable   // structure declaration
{
    char name[20];
    float volume;
    double price;
};

int main()
{
    using namespace std;
    inflatable guest =
    {
        "Glorious Gloria",  // name value
        1.88,               // volume value
        29.99               // price value
    };  // guest is a structure variable of type inflatable
        // It's initialized to the indicated values
    inflatable pal = 
    {
        "Audacious Arthur",
        3.12,
        32.99
    };  // pal is a second variable of type inflatable
// NOTE: some implementations require using
// static inflatable guest =

    cout << "Expand your guest list with " << guest.name;
    cout << " and " << pal.name << "!\n";
// pal.name is the name member of the pal variable
    cout << "You can have both for $";
    cout << guest.price + pal.price << "!\n";
    return 0;
}
```

注意：C++的一些老版本不能对函数中定义的常规数组进行初始化，也不能对函数中定义的常规结构进行初始化。解决方法是在声明中使用关键字 static。

下面是该程序的输出：

```
Expand your guest list with Glorious Gloria and Audacious Arthur!
You can have both for $62.98!
```

程序说明

结构声明的位置很重要。对于 structur.cpp 而言，有两种选择。可以将声明放在 main() 函数中，紧跟在开始括号的后面。另一种选择是将声明放到 main() 的前面，这里采用的便是这种方式，位于函数外面的声明被称为外部声明。对于这个程序来说，两种选择之间没有实际区别。但是对于那些包含两个或更多函数的程序来说，差别很大。外部声明可以被其后面的任何函数使用，而内部声明只能被该声明所属的函数使用。通常应使用外部声明，这样所有函数都可以使用这种类型的结构（参见图 4.7）。

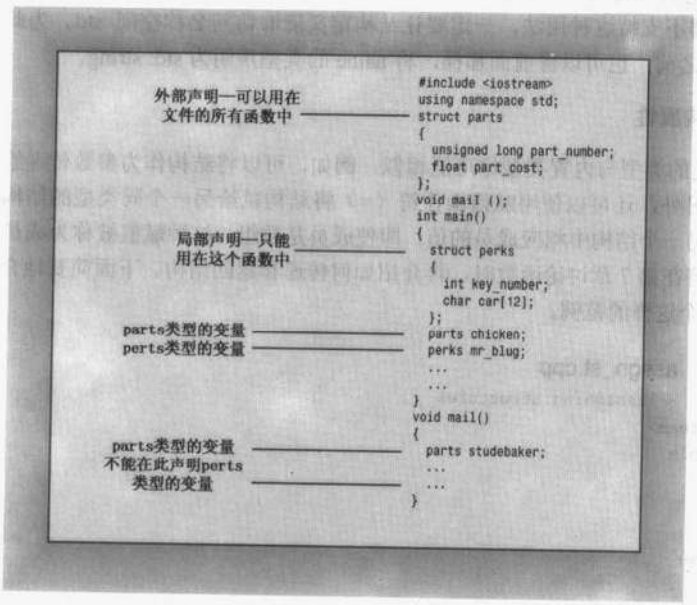


图 4.7　局部结构声明和外部结构声明

变量也可以在函数内部和外部定义，外部变量由所有的函数共享（这将在第 9 章做更详细的介绍）。C++不提倡使用外部变量，但提倡使用外部结构声明。另外，在外部声明符号常量通常更合理：

接下来，请注意初始化方式：

```
inflatable guest =
{
    "Glorious Gloria",      // name value
    1.88,              // volume value
    29.99              // price value
};
```

和数组一样，使用由逗号分隔的值列表，并将这些值用花括号括起。在该程序中，每个值占一行，但也可以将它们全部放在同一行中。只是应用逗号将它们分开：

```
inflatable duck = {"Daphne", 0.12, 9.98};
```

可以将结构的每个成员都初始化为适当类型的数据。例如，name 成员是一个字符数组，因此可以将其初始化为一个字符串。

可将每个结构成员看作是相应类型的变量。因此，pal.price 是一个 double 变量，而 pal.name 是一个 char

数组。当程序使用 cout 显示 pal.name 时，将把该成员显示为字符串。另外，由于 pal.name 是一个字符数组，因此可以用下标来访问其中的各个字符。例如，pal.name[0]是字符 A。不过 pal[0]没有意义，因为 pal 是一个结构，而不是数组。

4.4.2 结构可以将 string 类作为成员吗

可以将成员 name 指定为 string 类对象而不是字符数组吗？即可以像下面这样声明结构吗？

```
#include <string>
struct inflatable   // structure template
{
    std::string name;
    float volume;
    double price;
};
```

大体上说，答案是肯定的。实际上，答案取决于您使用的是哪种编译器，因为有些编译器（包括 Borland C++ 5.5 和 7.0 版之前的 Microsoft Visual C++）不支持对以 string 对象作为成员的结构进行初始化。

如果您的编译器不支持这种用法，一定要让结构定义能够访问名称空间 std。为此，可以将编译指令 using 移到结构定义之前；也可以像前面那样，将 name 的类型声明为 std::string。

4.4.3 其他结构属性

C++使用户定义的类型与内置类型尽可能相似。例如，可以将结构作为参数传递给函数，也可以让函数返回一个结构。另外，还可以使用赋值操作符（=）将结构赋给另一个同类型的结构，这样结构中每个成员都将被设置为另一个结构中相应成员的值，即使成员是数组。这种赋值被称为成员赋值（memberwise assignment），我们将在第 7 章讨论函数时，再介绍如何传递和返回结构。下面简要地介绍一下结构赋值，程序清单 4.12 是一个这样的范例。

程序清单 4.12 assgn_st.cpp

```
// assgn_st.cpp -- assigning structures
#include <iostream>
struct inflatable
{
    char name[20];
    float volume;
    double price;
};
int main()
{
    using namespace std;
    inflatable bouquet =
    {
        "sunflowers",
        0.20,
        12.49
    };
    inflatable choice;
    cout << "bouquet: " << bouquet.name << " for $";
    cout << bouquet.price << endl;
    choice = bouquet;  // assign one structure to another
    cout << "choice: " << choice.name << " for $";
    cout << choice.price << endl;
    return 0;
}
```

下面是该程序的输出：

```
bouquet: sunflowers for $12.49
```

```
choice: sunflowers for $12.49
```

从中可以看出，成员赋值是有效的，因为 choice 结构的成员值与 bouquet 结构中存储的值相同。

可以同时完成定义结构和创建结构变量的工作。为此，只需将变量名放在结束括号的后面即可：

```
struct perks
{
    int key_number;
    char car[12];
} mr_smith, ms_jones;   // two perks variables
```

甚至可以初始化以这种方式创建的变量：

```
struct perks
{
    int key_number;
    char car[12];
} mr_glitz =
{
    7,              // value for mr_glitz.key_number member
    "Packard"       // value for mr_glitz.car member
};
```

不过，将结构定义和变量声明分开，可以使程序更易于阅读和理解。

还可以声明没有名称的结构类型，方法是省略名称，同时定义一种结构类型和一个这种类型的变量：

```
struct      // no tag
{
    int x; // 2 members
    int y;
} position; // a structure variable
```

这样将创建一个名为 position 的结构变量。可以使用成员操作符来访问它的成员（如 position.x），但这种类型没有名称，因此以后无法创建这种类型的变量。本书将不使用这种形式的结构。

除了 C++程序可以使用结构标记作为类型名称外，C 结构具有到目前为止我们讨论的 C++结构的所有特性，但 C++结构的特性更多。例如，与 C 结构不同，C++结构除了成员变量之外，还可以有成员函数。但这些高级特性通常被用于类中，而不是结构中，因此将在讨论类的时候（从第 10 章开始）介绍它们。

4.4.4　结构数组

inflatable 结构包含一个数组（name）。也可以创建元素为结构的数组，方法和创建基本类型数组完全相同。例如，要创建一个包含 100 个 inflatable 结构的数组，可以这样做：

```
inflatable gifts[100];  // array of 100 inflatable structures
```

这样，gifts 将是一个 inflatable 数组，其中的每个元素（如 gifts[0]或 gifts[99]）都是 inflatable 对象，可以与成员操作符一起使用：

```
cin >> gifts[0].volume;          // use volume member of first struct
cout << gifts[99].price << endl; // display price member of last struct
```

记住，gifts 本身是一个数组，而不是结构，因此像 gifts.price 这样的表述是无效的。

要初始化结构数组，可以结合使用初始化数组的规则（用逗号分隔每个元素的值，并将这些值用花括号括起）和初始化结构的规则（用逗号分隔每个成员的值，并将这些值用花括号括起）。由于数组中的每个元素都是结构，因此可以使用结构初始化的方式来提供它的值。因此，最终结果为一个被括在花括号中、用逗号分隔的值列表，其中每个值本身又是一个被括在花括号中、用逗号分隔的值列表：

```
inflatable guests[2] =              // initializing an array of structs
    {
        {"Bambi", 0.5, 21.99},      // first structure in array
        {"Godzilla", 2000, 565.99}  // next structure in array
    };
```

可以按自己喜欢的方式来格式化它们。例如，两个初始化位于同一行，而每个结构成员的初始化各占一行。

程序清单 4.13 是一个使用结构数组的简短范例。由于 guests 是一个 inflatable 数组，因此 guests[0]的类型为 inflatable，可以使用它和句点操作符来访问相应 inflatable 结构的成员。

程序清单 4.13 arrstruc.cpp

```
// arrstruc.cpp -- an array of structures
#include <iostream>
struct inflatable
{
    char name[20];
    float volume;
    double price;
};
int main()
{
    using namespace std;
    inflatable guests[2] =          // initializing an array of structs
    {
        {"Bambi", 0.5, 21.99},      // first structure in array
        {"Godzilla", 2000, 565.99}  // next structure in array
    };

    cout << "The guests " << guests[0].name << " and " << guests[1].name
        << "\nhave a combined volume of "
        << guests[0].volume + guests[1].volume << " cubic feet.\n";
    return 0;
}
```

下面是该程序的输出：

```
The guests Bambi and Godzilla
have a combined volume of 2000.5 cubic feet.
```

4.4.5 结构中的位字段

与 C 语言一样，C++也允许指定占用特定位数的结构成员，这使得创建与某个硬件设备上的寄存器对应的数据结构非常方便。字段的类型应为整型或枚举（稍后将介绍），接下来是冒号，冒号后面是一个数字，它指定了使用的位数。可以使用没有名称的字段来提供间距。每个成员都被称为位字段（bit field）。下面是一个例子：

```
struct torgle_register
{
    unsigned int SN: 4;     // 4 bits for SN value
    unsigned int: 4;        // 4 bits unused
    bool goodIn: 1;         // valid input (1 bit)
    bool goodTorgle: 1;     // successful torgling
};
```

可以像通常那样初始化这些字段，还可以使用标准的结构表示法来访问位字段：

```
torgle_register tr = { 14, true, false };
...
if (tr.goodIn)  // if statement covered in Chapter 6
...
```

位字段通常用在低级编程中。一般来说，可以使用整型和附录 E 介绍的按位操作符来代替这种方式。

4.5 共用体

共用体（union）是一种数据格式，它能够存储不同的数据类型，但只能同时存储其中的一种类型。也就是说，结构可以同时存储 int、long 和 double，共用体只能存储 int、long 或 double。共用体的句法与结

构相似，但含义不同。例如，请看下面的声明：

```
union one4all
{
   int int_val;
   long long_val;
   double double_val;
};
```

可以使用 one4all 变量来存储 int、long 或 double，条件是在不同的时间进行：

```
one4all pail;
pail.int_val = 15;          // store an int
cout << pail.int_val;
pail.double_val = 1.38;     // store a double, int value is lost
cout << pail.double_val;
```

因此，pail 有时可以是 int 变量，而有时又可以是 double 变量。成员名称标识了变量的容量。由于共用体每次只能存储一个值，因此它必须有足够的空间来存储最大的成员，所以，共用体的长度为其最大成员的长度。

共用体的用途之一是，当数据项使用两种或更多种格式（但不会同时使用）时，可节省空间。例如，假设管理一个小商品目录，其中有一些商品的 ID 为整数，而另一些的 ID 为字符串。在这种情况下，可以这样做：

```
struct widget
{
char brand[20];
int type;
union id              // format depends on widget type
{
   long id_num;       // type 1 widgets
   char id_char[20];  // other widgets
} id_val;
};
...
widget prize;
...
if (prize.type == 1)           // if-else statement(chapter 6)
   cin >> prize.id_val.id_num;    // use member name to indicate mode
else
   cin >> prize.id_val.id_char;
```

匿名共用体（anonymous union）没有名称，其成员将成为位于相同地址处的变量。显然，每次只有一个成员是当前的成员：

```
struct widget
{
   char brand[20];
   int type;
   union                    // anonymous union
   {
      long id_num;          // type 1 widgets
      char id_char[20];     // other widgets
   };
};
...
widget prize;
...
if (prize.type == 1)
   cin >> prize.id_num;
else
   cin >> prize.id_char;
```

由于共用体是匿名的，因此 id_num 和 id_char 被视为 prize 的两个成员，它们的地址相同，所以不需要中间标识符 id_val。程序员负责确定当前哪个成员是活动的。

4.6 枚举

C++的 enum 工具提供了另一种创建符号常量的方式，这种方式可以代替 const。它还允许定义新类型，但必须按严格的限制进行。使用 enum 的句法与使用结构相似。例如，请看下面的语句：

```
enum spectrum {red, orange, yellow, green, blue, violet, indigo, ultraviolet};
```

这条语句完成两项工作：

- 让 spectrum 成为新类型的名称；spectrum 被称为枚举（enumeration），就像 struct 变量被称为结构一样。
- 将 red、orange、yellow 等作为符号常量，它们对应整数值 0～7。这些常量叫作枚举量（enumerator）。

在默认情况下，将整数值赋给枚举量，第一个枚举量的值为 0，第二个枚举量的值为 1，依次类推。可以通过显式地指定整数值来覆盖默认值，本章后面将介绍如何做。

可以用枚举名来声明这种类型的变量：

```
spectrum band;  // band a variable of type spectrum
```

枚举变量具有一些特殊的属性，下面来看一看。

在不进行强制类型转换的情况下，只能将定义枚举时使用的枚举量赋给这种枚举的变量，如下所示：

```
band = blue;     // valid, blue is an enumerator
band = 2000;     // invalid, 2000 not an enumerator
```

因此，spectrum 变量受到限制，只有 8 个可能的值。如果试图将一个非法值赋给它，则有些编译器将出现编译器错误，而另一些则发出警告。为获得最大限度的可移植性，应将把非 enum 值赋给 enum 变量视为错误。

对于枚举，只定义了赋值操作符。具体地说，没有为枚举定义算术运算：

```
band = orange;              // valid
++band;                     // not valid, ++ discussed in Chapter 5
band = orange + red;        // not valid, but a little tricky
...
```

不过，有些实现并没有这种限制，这有可能导致违反类型限制。例如，如果 band 的值为 ultraviolet（7），则++band（如果有效的话）将把 band 增加到 8，而对于 spectrum 类型来说，8 是无效的。另外，为获得最大限度的可移植性，应采纳较严格的限制。

枚举量是整型，可被提升为 int 类型，但 int 类型不能自动转换为枚举类型：

```
int color = blue;      // valid, spectrum type promoted to int
band = 3;              // invalid, int not converted to spectrum
color = 3 + red;       // valid, red converted to int
...
```

虽然在这个例子中，3 对应的枚举量是 green，但将 3 赋给 band 将导致类型错误。不过将 green 赋给 band 是可以的，因为它们都是 spectrum 类型。同样，有些实现方法没有这种限制。表达式 3 + red 中的加法并非为枚举量定义，但 red 被转换为 int 类型，因此结果的类型也是 int。由于在这种情况下，枚举将被转换为 int，因此可以在算术表达式中同时使用枚举和常规整数，尽管并没有为枚举本身定义算术运算。

前面范例：

```
band = orange + red;        // not valid, but a little tricky
```

非法的原因有些复杂。确实没有为枚举定义操作符+，但用于算术表达式中时，枚举将被转换为整数，因此表达式 orange + red 将被转换为 1 + 0。这是一个合法的表达式，但其类型为 int，不能将其赋给类型为 spectrum 的变量 band。

如果 int 值是有效的，则可以通过强制类型转换，将它赋给枚举变量：

```
band = spectrum(3);         // typecast 3 to type spectrum
```

如果试图对一个不适当的值进行强制类型转换，将出现什么情况呢？结果是不确定的，这意味着这样做不会出错，但不能依赖得到的结果：

```
band = spectrum(40003);     // undefined
```

请参阅本章后面的“枚举的取值范围”一节，以了解哪些值合适，哪些值不合适。

正如您看到的那样，枚举的规则相当严格。实际上，枚举更常被用来定义相关的符号常量，而不是新类型。例如，可以用枚举来定义 switch 语句中使用的符号常量（有关范例见第 6 章）。如果打算只使用常量，而不创建枚举类型的变量，则可以省略枚举类型的名称，如下面的例子所示：

```
enum {red, orange, yellow, green, blue, violet, indigo, ultraviolet};
```

4.6.1　设置枚举量的值

可以使用赋值操作符来显式地设置枚举量的值：

```
enum bits{one = 1, two = 2, four = 4, eight = 8};
```

指定的值必须是整数。也可以只显式地定义其中一些枚举量的值：

```
enum bigstep{first, second = 100, third};
```

这里，first 在默认情况下为 0。后面没有被初始化的枚举量的值将比其前面的枚举量大 1。因此，third 的值为 101。

最后，可以创建多个值相同的枚举量：

```
enum {zero, null = 0, one, numero_uno = 1};
```

其中，zero 和 null 都为 0，one 和 umero_uno 都为 1。在 C++早期的版本中，只能将 int 值（或提升为 int 的值）赋给枚举量，不过这种限制已经被取消了，因此可以使用 long 类型的值。

4.6.2　枚举的取值范围

最初，对于枚举来说，只有声明中指出的那些值是有效的。不过，C++现在通过强制类型转换，增加了可赋给枚举变量的合法值。每个枚举都有取值范围（range），通过强制类型转换，可以将取值范围中的任何整数值赋给枚举变量，即使这个值不是枚举值。例如，假设 bits 和 myflag 的定义如下：

```
enum bits{one = 1, two = 2, four = 4, eight = 8};
bits myflag;
```

则下面的代码将是合法的：

```
myflag = bits(6);    // valid, because 6 is in bits range
```

其中 6 不是枚举值，但它位于枚举定义的取值范围内。

取值范围的定义如下。首先，要找出上限，需要知道枚举量的最大值。找到大于这个最大值的、最小的 2 的幂，将它减去 1，得到的便是取值范围的上限。例如，前面定义的 bigstep 的最大值枚举值是 101。在 2 的幂中，比这个数大的最小值为 128，因此取值范围的上限为 127。要计算下限，需要知道枚举量的最小值。如果它不小于 0，则取值范围的下限为 0；否则，采用与寻找上限方式相同的方式，但加上负号。例如，如果最小的枚举量为-6，而比它小的、最大的 2 的幂是-8（加上负号），因此下限为-7。

选择用多少空间来存储枚举由编译器决定。对于取值范围较小的枚举，使用一个字节或更少的空间；而对于包含 long 类型值的枚举，则使用 4 个字节。

4.7　指针和自由存储空间

在第 3 章的开头，提到了计算机程序在存储数据时必须跟踪的 3 种基本属性。为了方便，这里再次列出了这些属性：

- 信息存储在何处。
- 存储的值为多少。
- 存储的信息是什么类型。

已经使用过一种策略来达到上述目的：定义一个简单变量。声明语句指出了值的类型和符号名，还让程序为值分配内存，并在内部跟踪该内存单元。

下面来看一看另一种策略，它在开发 C++类时非常重要。这种策略以指针为基础，指针是一个变量，其存储的是值的地址，而不是值本身。在讨论指针之前，我们先看一看如何找到常规变量的地址。只需对变量应用地址操作符（&），就可以获得它的位置：例如，如果 home 是一个变量，则&home 是它的地址。

程序清单 4.14 演示了这个操作符的用法。

程序清单 4.14 address.cpp

```
// address.cpp -- using the & operator to find addresses
#include <iostream>
int main()
{
    using namespace std;
    int donuts = 6;
    double cups = 4.5;

    cout << "donuts value = " << donuts;
    cout << " and donuts address = " << &donuts << endl;
// NOTE: you may need to use unsigned (&donuts)
// and unsigned (&cups)
    cout << "cups value = " << cups;
    cout << " and cups address = " << &cups << endl;
    return 0;
}
```

注意：cout 是一个智能对象，但在有些版本中比在另一些版本更“聪明”。也就是说，有些实现没有遵循 C++ 标准，无法识别指针类型。在这种情况下，必须将地址强制转换为可识别的类型，如 unsigned int。到底哪种强制类型转换合适取决于内存模型。默认的 DOS 内存模型使用 2 个字节的地址，因此可将地址强制转换为 unsigned int。不过，有些 DOS 内存模型使用 4 个字节的地址，在这种情况下，需要将地址强制转换为 unsigned long。

下面是该程序在某个系统上的输出：

```
donuts value = 6 and donuts address = 0x0065fd40
cups value = 4.5 and cups address = 0x0065fd44
```

显示地址时，该实现的 cout 使用十六进制表示法，因为这是常用于描述内存的表示法（有些实现可能使用十进制表示法）。在该实现中，donuts 的存储位置比 cups 要低。两个地址的差为 0x0065fd44 – 0x0065fd40（或 4）。这是有意义的，因为 donuts 的类型为 int，而这种类型使用 4 个字节。当然，不同系统给定的地址值可能不同。有些系统可能先存储 cups，再存储 donuts，这样两个地址值的差将为 8 个字节，因为 cups 的类型为 double。另外，在有些系统中，可能不会将这两个变量存储在相邻的内存单元中。

使用常规变量时，值是指定的量，而地址为派生量。下面来看看指针策略，它是 C++内存管理编程理念的核心（参见注释“指针与 C++基本原理”）。

指针与 C++基本原理

面向对象编程与传统的过程性编程的区别在于，OOP 强调的是在运行阶段（而不是编译阶段）进行决策。运行阶段指的是程序正在运行时，编译阶段指的是编译器将程序组合起来时。运行阶段决策就好比度假时，选择参观哪些景点取决于天气和当时的心情；而编译阶段决策更像不管在什么条件下，都坚持预先设定的日程安排。

运行阶段决策提供了灵活性，可以根据当时的情况进行调整。例如，考虑为数组分配内存的情况。传统的方法是声明一个数组。要在 C++中声明数组，必须指定数组的长度。因此，数组长度在程序编译时就设定好了；这就是编译阶段决策。读者可能认为，在 80%的情况下，一个包含 20 个元素的数组足够了，但程序有时需要处理 200 个元素。为了安全起见，使用了一个包含 200 个元素的数组。这样，程序在大多数情况下都浪费了内存。OOP 通过将这样的决策推迟到运行阶段进行，使程序更灵活。在程序运行后，可以这次告诉它只需要 20 个元素，而还可以下次告诉它需要 205 个元素。

总之，使用 OOP 时，您可能在运行阶段确定数组的长度。为使用这种方法，语言必须允许在程序运行时创建数组。稍后您看会到，C++采用的方法是，使用关键字 new 请求正确数量的内存以及使用指针来跟踪新分配的内存的位置。

处理存储数据的新策略刚好相反，将地址视为指定的量，而将值视为派生量。一种特殊类型的变量——指针——用于存储值的地址。因此，指针名表示的是地址。*操作符被称为间接值（indirect velue）或解除引用

（dereferencing）操作符，将其应用于指针，可以得到该地址处存储的值（这和乘法使用的符号相同；C++根据上下文来确定所指的是乘法还是解除引用）。例如，假设 manly 是一个指针，则 manly 表示的是一个地址，而*manly 表示存储在该地址处的值。*manly 与常规 int 变量等效。程序清单 4.15 说明了这几点，它还演示了如何声明指针。

程序清单 4.15　pointer.cpp

```
// pointer.cpp -- our first pointer variable
#include <iostream>
int main()
{
   using namespace std;
   int updates = 6;        // declare a variable
   int * p_updates;        // declare pointer to an int

   p_updates = &updates;   // assign address of int to pointer

// express values two ways
   cout << "Values: updates = " << updates;
   cout << ", *p_updates = " << *p_updates << endl;

// express address two ways
   cout << "Addresses: &updates = " << &updates;
   cout << ", p_updates = " << p_updates << endl;

// use pointer to change value
   *p_updates = *p_updates + 1;
   cout << "Now updates = " << updates << endl;
   return 0;
}
```

下面是该程序的输出：

```
Values: updates = 6, *p_updates = 6
Addresses: &updates = 0x0065fd48, p_updates = 0x0065fd48
Now updates = 7
```

从中可以知道，int 变量 updates 和指针变量 p_updates 只不过是同一枚硬币的两面。变量 updates 表示值，并使用&操作符来获得地址；而变量 p_updates 表示地址，并使用*操作符来获得值（参见图 4.8）。由于 p_updates 指向 updates，因此*p_updates 和 updates 完全等价。可以像使用 int 变量那样使用*p_updates。正如程序清单 4.15 表明的，甚至可以将值赋给*p_updates。这样做将修改指向的值，即 updates。

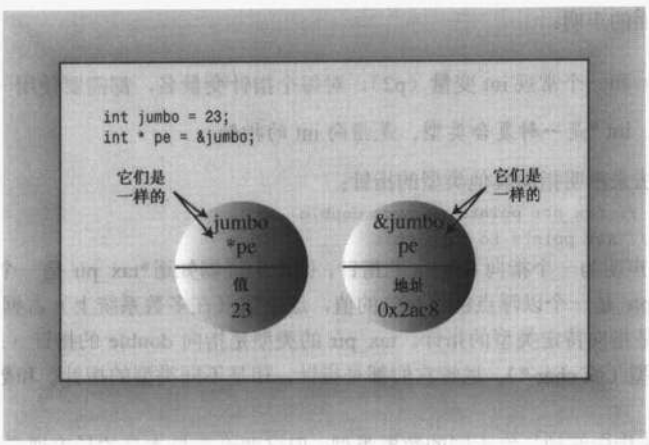


图 4.8　硬币的两面

4.7.1 声明和初始化指针

我们来看看如何声明指针。计算机需要跟踪指针指向的值的类型。例如，char 的地址与 double 的地址看上去没什么两样，但 char 和 double 使用的字节数是不同的，它们存储值时使用的内部格式也不同。因此，指针声明必须指定指针指向的数据的类型。

例如，前一个范例包含这样的声明：

```
int * p_updates;
```

这表明，* p_updates 的类型为 int。由于*操作符被用于指针，因此 p_updates 变量本身必须是指针。我们说 p_updates 指向 int 类型，我们还说 p_updates 的类型是指向 int 的指针，或 int*。可以这样说，p_updates 是指针（地址），而*p_updates 是 int，而不是指针（见图 4.9）。

顺便说一句，*操作符两边的空格是可选的。传统上，C 程序员使用这种格式：

```
int *ptr;
```

这强调*ptr 是一个 int 类型的值。而很多 C++程序员使用这种格式：

```
int* ptr;
```

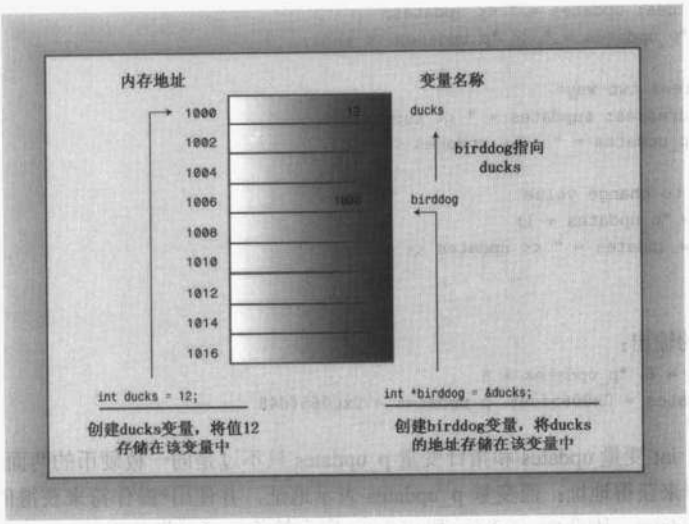


图 4.9 指针存储地址

这强调的是：int*是一种类型，是指向 int 的指针。在哪里添加空格对于编译器来说是没有任何区别的。不过要知道的是，下面的声明：

```
int* p1, p2;
```

将创建一个指针（p1）和一个常规 int 变量（p2）。对每个指针变量名，都需要使用一个*。

记住：在 C++中，int *是一种复合类型，是指向 int 的指针。

可以用同样的句法来声明指向其他类型的指针：

```
double * tax_ptr; // tax_ptr points to type double
char * str;       // str points to type char
```

由于已将 tax_ptr 声明为一个指向 double 的指针，因此编译器知道*tax_ptr 是一个 double 类型的值。也就是说，它知道*tax_ptr 是一个以浮点格式存储的值，这个值（在多数系统上）占据 8 个字节。指针变量不仅仅是指针，而且是指向特定类型的指针。tax_ptr 的类型是指向 double 的指针（或 double *类型），str 是指向 char 的指针类型（或 char *）。尽管它们都是指针，却是不同类型的指针。和数组一样，指针都是基于其他类型的。

虽然 tax_ptr 和 str 指向两种长度不同的数据类型，但这两个变量本身的长度通常是相同的。也就是说，char 的地址与 double 的地址的长度相同，这就好比 1016 可能是超市的街道地址，而 1024 可以是小村庄的

街道地址一样。地址的长度或值既不能指示关于变量的长度或类型的任何信息，也不能指示该地址上有什么建筑物。一般来说，地址需要 2 个还是 4 个字节，取决于计算机系统（有些系统可能需要更大的地址，系统可以针对不同的类型使用不同长度的地址）。

可以在声明语句中初始化指针。在这种情况下，被初始化的是指针，而不是它指向的值。也就是说，下面的语句：

```
int higgens = 5;
int * pt = &higgens;
```

将 pt（而不是*pt）的值设置为&higgens。

程序清单 4.16 演示了如何将指针初始化为一个地址。

程序清单 4.16　init_ptr.cpp

```
// init_ptr.cpp -- initialize a pointer
#include <iostream>
int main()
{
   using namespace std;
   int higgens = 5;
   int * pt = &higgens;

   cout << "Value of higgens = " << higgens
      << "; Address of higgens = " << &higgens << endl;
   cout << "Value of *pt = " << *pt
      << "; Value of pt = " << pt << endl;
   return 0;
}
```

下面是该程序的输出：

```
Value of higgens = 5; Address of higgens = 0012FED4
Value of *pt = 5; Value of pt = 0012FED4
```

从中可以知道，程序将 pi（而不是*pi）初始化为 higgens 的地址。

4.7.2　指针的危险

危险更易发生在那些使用指针不仔细的人身上。极其重要的一点是：在 C++中创建指针时，计算机将分配用来存储地址的内存，但不会分配用来存储指针所指向的数据的内存。为数据提供空间是一个独立的步骤，忽略这一步无疑是自找麻烦，如下所示：

```
long * fellow;          // create a pointer-to-long
*fellow = 223323;       // place a value in never-never land
```

fellow 确实是一个指针，但它指向哪里呢？上述代码没有将地址赋给 fellow。那么 223323 将被放在哪里呢？我们不知道。由于 fellow 没有被初始化，它可能有任何值。不管值是什么，程序都将它解释为存储 223323 的地址。如果 fellow 的值碰巧为 1200，计算机将把数据放在地址 1200 上，即使这恰巧是程序代码的地址。fellow 指向的地方很可能并不是所要存储 223323 的地方。这种错误可能会导致一些最隐匿、最难以跟踪的 bug。

警告：一定要在对指针应用解除引用操作符（*）之前，将指针初始化为一个确定的、适当的地址。这是关于使用指针的金科玉律

4.7.3　指针和数字

指针不是整型，虽然计算机通常把地址当作整数来处理。从概念上看，指针与整数是截然不同的类型。整数是可以执行加、减、除等运算的数字，而指针描述的是位置，将两个地址相乘没有任何意义。从可以对整数和指针执行的操作上看，它们也是彼此不同的。因此，不能简单地将整数赋给指针：

```
int * pt;
pt = 0xB8000000;  // type mismatch
```

在这里，左边是指向 int 的指针，因此可以把它赋给地址，但右边是一个整数。您可能知道，0xB8000000 是系统上视频内存的组合段偏移地址，但这条语句并没有告诉程序，这个数字就是一个地址。在 C99 标准发布之前，C 语言允许这样赋值。但 C++在类型一致方面的要求更严格，编译器将显示一条错误消息，通告类型不匹配。要将数字值作为地址来使用，应通过强制类型转换将数字转换为适当的地址类型：

```
int * pt;
pt = (int *) 0xB8000000; // types now match
```

这样，赋值语句的两边都是整数的地址，因此这样赋值有效。注意，pt 是 int 值的地址，并不意味着 pt 本身的类型是 int。例如，在使用 DOS 的 IBM PC 中的大型内存模型下，int 类型是一个 2 个字节的值，而地址是一个 4 个字节的值。

指针还有其他一些有趣的特性，我们将在合适的时候进行讨论。现在，看一看如何使用指针来管理运行阶段的内存空间分配。

4.7.4 使用 new 来分配内存

对指针的工作方式有一定的了解后，来看一看它如何实现重要的 OOP 技术——在程序运行时分配内存。前面，我们将指针初始化为变量的地址；变量是在编译时分配的、有名称的内存，而指针只是为可以通过名称直接访问的内存提供了一个别名。指针真正的用武之地在于，在运行阶段分配未命名的内存以存储值。在这种情况下，只能通过指针来访问内存。在 C 语言中，可以用库函数 malloc()来分配内存；在 C++中仍然可以这样做，但 C++还有更好的方法——new 操作符。

我们来试试这种新技术，在运行阶段为一个 int 值分配未命名的内存，并使用指针来访问这个值。这里的关键所在是 C++的 new 操作符。程序员要告诉 new，需要为哪种数据类型分配内存；new 将找到一个长度正确的内存块，并返回该内存块的地址。程序员的责任是将该地址赋给一个指针。下面是一个这样的范例：

```
int * pn = new int;
```

new int 告诉程序，需要适合存储 int 的内存。new 操作符根据类型来确定需要多少字节的内存。然后，它找到这样的内存，并返回其地址。接下来，将地址赋给 pn，pn 是被声明为指向 int 的指针。现在，pn 是地址，而*pn 是存储在那里的值。将这种方法与将变量的地址赋给指针进行比较：

```
int higgens;
int * pt = &higgens;
```

在这两种情况（pn 和 pt）下，都是将一个 int 变量的地址赋给了指针。在第二种情况下，可以通过名称 higgens 来访问该 int，在第一种情况下，则只能通过该指针进行访问。这引出了一个问题：pn 指向的内存没有名称，如何称呼它呢？我们说，pn 指向一个数据对象。这里的“对象”不是“面向对象编程”中的对象，而是一样“东西”。术语“数据对象”比“变量”更通用，它指的是为数据项分配的内存块。因此，变量也是数据对象，但 pn 指向的内存不是变量。乍一看，处理数据对象的指针方法可能不太好用，但它使程序在管理内存方面有更大的控制权。

为一个数据对象（可以是结构，也可以是基本类型）获得并指定分配内存的通用格式如下：

```
typeName pointer_name = new typeName;
```

需要在两个地方指定数据类型：用来指定需要什么样的内存和用来声明合适的指针。当然，如果已经声明了相应类型的指针，则可以使用该指针，而不用再声明一个新的指针。程序清单 4.17 演示了如何将 new 用于两种不同的类型。

程序清单 4.17 use_new.cpp

```
// use_new.cpp -- using the new operator
#include <iostream>
int main()
{
   using namespace std;
   int * pt = new int;          // allocate space for an int
   *pt = 1001;                  // store a value there
```

```
    cout << "int ";
    cout << "value = " << *pt << ": location = " << pt << endl;

    double * pd = new double;     // allocate space for a double
    *pd = 10000001.0;             // store a double there

    cout << "double ";
    cout << "value = " << *pd << ": location = " << pd << endl;
    cout << "size of pt = " << sizeof (pt);
    cout << ": size of *pt = " << sizeof (*pt) << endl;
    cout << "size of pd = " << sizeof pd;
    cout << ": size of *pd = " << sizeof (*pd) << endl;
    return 0;
}
```

下面是该程序的输出：

```
int value = 1001: location = 0x004301a8
double value = 1e+07: location = 0x004301d8
size of pt = 4: size of *pt = 4
size of pd = 4: size of *pd = 8
```

当然，内存位置的准确值随系统而异。

程序说明

该程序使用 new 分别为 int 类型和 double 类型的数据对象分配内存。这是在程序运行时进行的。指针 pt 和 pd 指向这两个数据对象，如果没有它们，将无法访问这些内存单元。有了这两个指针，就可以像使用变量那样使用*pt 和*pd 了。将值赋给*pt 和*pd，从而将这些值赋给新的数据对象。同样，可以通过打印 *pt 和*pd 来显示这些值。

该程序还指出了必须声明指针所指向的类型的原因之一。地址本身只指出了对象存储地址的开始，而没有指出其类型（使用的字节数）。从这两个值的地址可以知道，它们都只是数字，并没有提供类型或长度信息。另外，指向 int 的指针的长度与指向 double 的指针相同。它们都是地址，但由于 use_new.cpp 声明了指针的类型，因此程序知道*pd 是 8 个字节的 double 值，*pt 是 4 个字节的 int 值。use_new.cpp 打印*pd 的值时，cout 知道要读取多少字节以及如何解释它们。

内存被耗尽

计算机可能会由于没有足够的内存而无法满足 new 的请求。在这种情况下，new 将返回 0。在 C++中，值为 0 的指针被称为空值指针（null pointer）。C++确保空值指针不会指向有效的数据，因此它常被用来表示操作符或函数失效，如果成功，它们将返回一个有用的指针。学习 if 语句（参见第 6 章）后，读者便可以检查 new 是否返回的是空值指针，从而防止程序超界。如果无法分配内存，new 除返回空值指针外，还可能引发 bad_alloc 异常。异常机制将在第 15 章讨论。

4.7.5　使用 delete 来释放内存

当需要内存时，可以使用 new 来请求，这只是 C++内存管理数据包中有魅力的一个方面。另一个方面是 delete 操作符，它使得在使用完内存后，能够将其归还给内存池，这是通向最有效地使用内存的关键一步。归还或释放（free）的内存可供程序的其他部分使用。使用 delete 时，后面要加上指向内存块的指针（这些内存块最初是用 new 分配的）：

```
int * ps = new int;    // allocate memory with new
. . .                  // use the memory
delete ps;             // free memory with delete when done
```

这将释放 ps 指向的内存，但不会删除指针 ps 本身。例如，可以将 ps 重新指向另一个新分配的内存块。一定要配对地使用 new 和 delete；否则将发生内存泄漏（memory leak），也就是说，被分配的内存再也无法使用了。如果内存泄漏严重，则程序将由于不断寻找更多内存而终止。

不要尝试释放已经释放的内存块，C++标准指出，这样做的结果将是不确定的，这意味着什么情况都可能发生。另外，不能使用 delete 来释放声明变量所获得的内存：

```
int * ps = new int;    // ok
delete ps; ..........  // ok
delete ps; .........   // not ok now
int jugs = 5;          // ok
int * pi = & jugs;     // ok
delete pi;             // not allowed, memory not allocated by new
```

警告：只能用 delete 来释放使用 new 分配的内存。不过，对空指针使用 delete 是安全的。

注意，使用 delete 的关键在于，将它用于 new 分配的内存。这并不意味着要使用用于 new 的指针，而是用于 new 的地址：

```
int * ps = new int;    // allocate memory
int * pq = ps;         // set second pointer to same block
delete pq;             // delete with second pointer
```

一般来说，不要创建两个指向同一个内存块的指针，因为这将增加错误地删除同一个内存块两次的可能性。但稍后读者会看到，对于返回指针的函数，使用另一个指针确实有道理。

4.7.6 使用 new 来创建动态数组

如果程序只需要一个值，则可能会声明一个简单变量，因为对于管理一个小型数据对象来说，这样做比使用 new 和指针更简单，尽管给人留下的印象不那么深刻。通常，对于大型数据（如数组、字符串和结构），应使用 new，这正是 new 的用武之地。例如，假设要编写一个程序，它是否需要数组取决于运行时用户提供的信息。如果通过声明来创建数组，则在程序被编译时将为它分配内存空间。不管程序最终是否使用数组，数组都在那里，它占用了内存。在编译时给数组分配内存被称为静态联编（static binding），意味着数组是在编译时加入到程序中的。但使用 new 时，如果在运行阶段需要数组，则创建它；如果不需要，则不创建。还可以在程序运行时选择数组的长度。这被称为动态联编（dynamic binding），意味着数组是在程序运行时创建的。这种数组叫作动态数组（dynamic array）。使用静态联编时，必须在编写程序时指定数组的长度；使用动态联编时，程序将在运行时确定数组的长度。

现在，我们来看一下关于动态数组的两个基本问题：如何使用 C++的 new 操作符创建数组以及如何使用指针访问数组元素。

1. 使用 new 创建动态数组

在 C++中，创建动态数组很容易：只要将数组的元素类型和元素数目告诉 new 即可。必须在类型名后加上方括号，其中包含元素数目。例如，要创建一个包含 10 个 int 元素的数组，可以这样做：

```
int * psome = new int [10]; // get a block of 10 ints
```

new 操作符返回第一个元素的地址。在这个例子中，该地址被赋给指针 psome。当程序结束使用内存块时，应使用 delete 释放它们。

使用 new 创建数组时，应使用另一种格式的 delete，它能够指出所要释放的是一个数组：

```
delete [] psome;              // free a dynamic array
```

方括号告诉程序，应释放整个数组，而不仅仅是指针指向的元素。请注意 delete 和指针之间的方括号。如果使用 new 时，不带方括号，则使用 delete 时，也不应带方括号。如果使用 new 时带方括号，则使用 delete 时也应带方括号。C++的早期版本无法识别方括号表示法。不过，对于 ANSI/ISO 标准来说，new 与 delete 的格式不匹配导致的后果是不确定的，这意味着程序员不能依赖于某种特定的行为。下面是一个例子：

```
int * pt = new int;
short * ps = new short [500];
delete [] pt;  // effect is undefined, don't do it
delete ps;     // effect is undefined, don't do it
```

总之，使用 new 和 delete 时，应遵守以下规则：

- 不要使用 delete 来释放不是 new 分配的内存。

- 不要使用 delete 释放同一个内存块两次。
- 如果使用 new [ ]为数组分配内存，则应使用 delete [ ]来释放。
- 如果使用 new [ ]为一个实体分配内存，则应使用 delete（没有方括号）来释放。
- 对空值指针应用 delete 是安全的。

现在我们回过头来讨论动态数组。psome 是指向一个 int（数组第一个元素）的指针。您的责任是跟踪内存块中的元素个数。也就是说，由于编译器不能对 psome 是指向 10 个整数中的第 1 个这种情况进行跟踪，因此编写程序时，必须让程序跟踪元素的数目。

实际上，程序确实跟踪了分配的内存量，以便以后使用 delete [ ]操作符时能够正确地释放这些内存。但这种信息不是公用的，例如，不能使用 sizeof 操作符来确定动态分配的数组包含的字节数。

为数组分配内存的通用格式如下：

```
type_name pointer_name = new type_name [num_elements];
```

使用 new 操作符可以确保内存块足以存储 num_elements 个类型为 type_name 的元素，而 pointer_name 将指向第 1 个元素。下面将会看到，可以以使用数组名的方式来使用 pointer_name。

2. 使用动态数组

创建动态数组后，如何使用它呢？首先，从概念上考虑这个问题。下面的语句：

```
int * psome = new int [10]; // get a block of 10 ints
```

创建指针 psome，它指向包含 10 个 int 值的内存块中的第 1 个元素。可以将它看作是一根指向该元素的手指。假设 int 占 4 个字节，则将手指沿正确的方向移动 4 个字节，手指将指向第 2 个元素。总共有 10 个元素，这就是手指的移动范围。因此，new 语句提供了识别内存块中每个元素所需的全部信息。

现在从实际角度考虑这个问题。如何访问其中的元素呢？第一个元素不成问题。由于 psome 指向数组的第 1 个元素，因此*psome 是第 1 个元素的值。这样，还有 9 个元素。如果没有使用过 C 语言，下面这种最简单的方法可能会令人大吃一惊：只要把指针当作数组名使用即可。也就是说，对于第 1 个元素，可以使用 psome[0]，而不是*psome；对于第 2 个元素，可以使用 psome[1]，依此类推。这样，使用指针来访问动态数组就非常简单了，虽然还不知道为何这种方法管用。可以这样做的原因是，C 和 C++内部都使用指针来处理数组。数组和指针基本等价是 C 和 C++的优点之一。稍后将更详细地介绍这种等同性。首先，程序清单 4.18 演示了如何使用 new 来创建动态数组以及使用数组表示法来访问元素；它还指出了指针和真正的数组名之间的根本差别。

程序清单 4.18　arraynew.cpp

```
// arraynew.cpp -- using the new operator for arrays
#include <iostream>
int main()
{
    using namespace std;
    double * p3 = new double [3]; // space for 3 doubles
    p3[0] = 0.2;                  // treat p3 like an array name
    p3[1] = 0.5;
    p3[2] = 0.8;
    cout << "p3[1] is " << p3[1] << ".\n";
    p3 = p3 + 1;                  // increment the pointer
    cout << "Now p3[0] is " << p3[0] << " and ";
    cout << "p3[1] is " << p3[1] << ".\n";
    p3 = p3 - 1;                  // point back to beginning
    delete [] p3;                 // free the memory
    return 0;
}
```

下面是该程序的输出：

```
p3[1] is 0.5.
Now p3[0] is 0.5 and p3[1] is 0.8.
```

从中可以知道，arraynew.cpp 将指针 p3 当作数组名来使用，p3[0]为第 1 个元素，依次类推。下面的代

码行指出了数组名和指针之间的根本差别：

```
p3 = p3 + 1; // okay for pointers, wrong for array names
```

不能修改数组名的值。但指针是变量，因此可以修改它的值。请注意将 p3 加 1 的效果。表达式 p3[0] 现在指的是数组的第 2 个值。因此，将 p3 加 1 导致它指向第 2 个元素而不是第 1 个。将它减 1 后，指针将指向原来的值，这样程序便可以给 delete[ ]提供正确的地址。

相邻的 int 地址通常相差 2 个字节或 4 个字节，而将 p3 加 1 后，它将指向下一个元素的地址，这表明指针算术有一些特别的地方。情况确实如此。

4.8 指针、数组和指针算术

指针和数组基本等价的原因在于指针算术（pointer arithmetic）和 C++内部处理数组的方式。首先，我们来看一看算术。将整数变量加 1 后，其值将增加 1；但将指针变量加 1 后，增加的量等于它指向的类型的字节数。将指向 double 的指针加 1 后，如果系统对 double 使用 8 个字节存储，则数值将增加 8；将指向 short 的指针加 1 后，如果系统对 short 使用 2 个字节存储，则指针值将增加 2。程序清单 4.19 演示了这种令人吃惊的现象，它还说明了另一点：C++将数组名解释为地址。

程序清单 4.19 addpntrs.cpp

```
// addpntrs.cpp -- pointer addition
#include <iostream>
int main()
{
    using namespace std;
    double wages[3] = {10000.0, 20000.0, 30000.0};
    short stacks[3] = {3, 2, 1};

// Here are two ways to get the address of an array
    double * pw = wages;     // name of an array = address
    short * ps = &stacks[0]; // or use address operator
// with array element
    cout << "pw = " << pw << ", *pw = " << *pw << endl;
    pw = pw + 1;
    cout << "add 1 to the pw pointer:\n";
    cout << "pw = " << pw << ", *pw = " << *pw << "\n\n";

    cout << "ps = "" << ps << ", *ps = " << *ps << endl;
    ps = ps + 1;
    cout << "add 1 to the ps pointer:\n";
    cout << "ps = " << ps << ", *ps = " << *ps << "\n\n";

    cout << "access two elements with array notation\n";
    cout << "stacks[0] = " << stacks[0]
    << ", stacks[1] = " << stacks[1] << endl;
    cout << "access two elements with pointer notation\n";
    cout << "*stacks = " << *stacks
        << ", *(stacks + 1) = " << *(stacks + 1) << endl;

    cout << sizeof(wages) << " = size of wages array\n";
    cout << sizeof(pw) << " = size of pw pointer\n";
    return 0;
}
```

下面是该程序的输出：

```
pw = 0012FEBC, *pw = 10000
add 1 to the pw pointer:
```

```
pw = 0012FEC4, *pw = 20000

ps = 0012FEAC, *ps = 3
add 1 to the ps pointer:
ps = 0012FEAE, *ps = 2

access two elements with array notation
stacks[0] = 3, stacks[1] = 2
access two elements with pointer notation
*stacks = 3, *(stacks + 1) = 2
24 = size of wages array
4 = size of pw pointer
```

4.8.1　程序说明

在多数情况下，C++将数组名解释为数组第 1 个元素的地址。因此，下面的语句：

```
double * pw = wages;
```

将 pw 声明为指向 double 类型的指针，然后将它初始化为 wages——wages 数组中第 1 个元素的地址。wages 和所有数组一样，存在下面的等式：

```
wages = &wages[0] = address of first element of array
```

为表明情况确实如此，该程序在表达式&stacks[0]中显式地使用地址操作符来将 ps 指针初始化为 stacks 数组的第 1 个元素。

接下来，程序查看 pw 和*pw 的值。前者是地址，后者是存储该地址中的值。由于 pw 指向第 1 个元素，因此*pw 显示的值为第 1 个元素的值，即 10000。接着，程序将 pw 加 1。正如前面指出的那样，这样数字地址值将增加 8（十六进制为，fd24 + 8 = fd2c），这是因为在该系统中，double 占 8 个字节的内存。这使得 pw 的值为第 2 个元素的地址。因此，*pw 现在的值是 20000——第 2 个元素的值（参见图 4.10，为使图像更为清晰，对其中的地址值做了调整）。

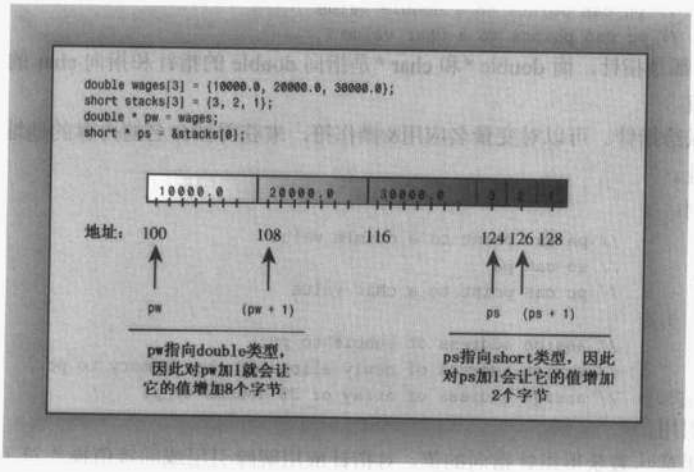


图 4.10　指针加法

此后，程序对 ps 执行相同的操作。这一次由于 ps 指向的是 short 类型，而 short 占用 2 个字节，因此将指针加 1 时，其值将增加 2。结果是，指针也指向数组中下一个元素。

记住：将指针变量加 1 后，其增加的值等于指向的类型占用的字节数。

现在来看一看数组表达式 stacks[1]。C++编译器将该表达式看作是*（stacks + 1），这意味着先计算数组第 2 个元素的地址，然后找到存储在那里的值。最后的结果便是 stacks [1]的含义（操作符优先级要求使

用括号，如果不使用括号，将给*stacks 加上 1，而不是给 stacks 加上 1）。

从该程序的输出可知，*（stacks + 1）和 stacks[1]是等价的。同样，*（stacks + 2）和 stacks[2]也是等价的。通常，使用数组表示法时，C++都执行下面的转换：

```
arrayname[i] becomes *(arrayname + i)
```

如果使用的是指针，而不是数组名，则 C++也将执行同样的转换：

```
pointername[i] becomes *(pointername + i)
```

因此，在很多情况下，可以相同的方式使用指针名和数组名。对于它们，可以使用数组方括号表示法，也可以使用解除引用操作符（*）。在多数表达式中，它们都表示地址。区别之一是，可以修改指针的值，而数组名是常量：

```
pointername = pointername + 1; // valid
arrayname = arrayname + 1;     // not allowed
```

另一个区别是，对数组应用 sizeof 操作符得到的是数组的长度，而对指针应用 sizeof 得到的是指针的长度，即使指针指向的是一个数组。例如，在程序清单 4.19 中，pw 和 wages 指的是同一个数组，但对它们应用 sizeof 操作符得到的结果如下：

```
24 = size of wages array  //displaying sizeof wages
4 = size of pw pointer    //displaying sizeof pw
```

这种情况下，C++不会将数组名解释为地址。

总之，使用 new 来创建数组以及使用指针来访问不同的元素很简单。只要把指针当作数组名对待即可。不过，要理解为何可以这样做，将是一种挑战。要想真正了解数组和指针，应认真复习它们的相互关系。

指针小结

刚才已经介绍了大量指针的知识，下面对指针和数组做一总结。

（1）声明指针

要声明指向特定类型的指针，请使用下面的格式：

```
typeName * poiterName
```

下面是一些范例：

```
double * pn:     // pn can points to a double value
char * pc:       // pc can points to a char value
```

其中，pn 和 pc 都是指针，而 double *和 char *是指向 double 的指针和指向 char 的指针。

（2）给指针赋值

应将内存地址赋给指针。可以对变量名应用&操作符，来获得被命名的内存的地址，new 操作符返回未命名的内存的地址。

下面是一些范例：

```
double * pn;          // pn can point to a double value
double * pa;          // so can pa
char * pc;            // pc can point to a char value
double bubble = 3.2;
pn = &bubble;         // assign address of bubble to pn
pc = new char;        // assign address of newly allocated char memory to pc
pa = new double[30];  // assign address of array of 30 double to pa
```

（3）解除指针引用

对指针解除引用意味着获得指针指向的值。对指针应用解除引用或间接值操作符（*）来解除引用。因此，如果像上面的例子中那样，pn 是指向 bubble 的指针，则*pn 是指向的值，即 3.2。

下面是一些范例：

```
cout << *pn; // print the value of bubble
*pc = 'S';   // place 'S' into the memory location whose address is pc
```

另一种解除指针引用的方法是使用数组表示法，例如，pn[0]与*pn 是一样的。决不要对未被初始化为适当地址的指针解除引用。

（4）区分指针和指针所指向的值

如果 pt 是指向 int 的指针，则*pt 不是指向 int 的指针，而是完全等同于一个 int 类型的变量。pt

才是指针。

下面是一些范例：

```
int * pt = new int;     // assigns an address to the pointer pt
*pt = 5;               // stores the value 5 at that address
```

（5）数组名

在多数情况下，C++将数组名视为数组的第一个元素的地址。

下面是一个范例：

```
int tacos[10];          // now tacos is the same as &tacos[0]
```

一种例外情况是，将 sizeof 操作符用于数组名时，此时将返回整个数组的长度（单位为字节）。

（6）指针算术

C++允许将指针和整数相加。加 1 的结果等于原来的地址值加上指向的对象占用的总字节数。还可以将一个指针减去另一个指针，获得两个指针的差。后一种运算将得到一个整数，仅当两个指针指向同一个数组（也可以指向超出结尾的一个位置）时，这种运算才有意义；这将得到两个元素的间隔。

下面是一些范例：

```
int tacos[10] = {5, 2, 8, 4, 1, 2, 2, 4, 6, 8};
int * pt = tacos;           // suppose pf and tacos are the address 3000
pt = pt + 1;                // now pt is 3004 if a int is 4 bytes
int *pe = &tacos[9];        // pe is 3036 if an int is 4 bytes
pe = pe -1;                 // now pe is 3032, the address of tacos[8]
int diff = pe - pt;         // diff is 7, the separation between
                            // tacos[8] and tacos[1]
```

（7）数组的动态联编和静态联编

使用数组声明来创建数组时，将采用静态联编，即数组的长度在编译时设置：

```
int tacos[10]; // static binding, size fixed at compile time
```

使用 new[ ]操作符创建数组时，将采用动态联编（动态数组），即将在运行时为数组分配空间，其长度也将在运行时设置。使用完这种数组后，应使用 delete [ ]释放其占用的内存：

```
int size;
cin >> size;
int * pz = new int [size]; // dynamic binding, size set at run time
...
delete [] pz;              // free memory when finished
```

（8）数组表示法和指针表示法

使用方括号数组表示法等同于对指针解除引用：

```
tacos[0] means *tacos means the value at address tacos
tacos[3] means *(tacos + 3) means the value at address tacos + 3
```

数组名和指针变量都是如此，因此对于指针和数组名，既可以使用指针表示法，也可以使用数组表示法。

下面是一些范例：

```
int * pt = new int [10];  // pt points to block of 10 ints
*pt = 5;                  // set element number 0 to 5
pt[0] = 6;                // reset element number 0 to 6
pt[9] = 44;               // set tenth element (element number 9) to 44
int coats[10];
*(coats + 4) = 12;        // set coats[4] to 12
```

4.8.2 指针和字符串

数组和指针的特殊关系可以扩展到 C-风格字符串。请看下面的代码：

```
char flower[10] = "rose";
cout << flower << "s are red\n";
```

数组名是第一个元素的地址，因此 cout 语句中的 flower 是包含字符 r 的 char 元素的地址。cout 对象认为 char 的地址是字符串的地址，因此它打印该地址处的字符，然后继续打印后面的字符，直到遇到空字符

（\0）为止。总之，如果给 cout 提供一个字符的地址，则它将从该字符开始打印，直到遇到空字符为止。

这里的关键不在于 flower 是数组名，而在于 flower 是一个 char 的地址。这意味着可以将指向 char 的指针变量作为 cout 的参数，因为它也是 char 的地址。当然，该指针应指向字符串的开头，我们稍后将讨论这个问题。

前面的 cout 语句中最后一部分的情况如何呢？如果 flower 是字符串第一个字符的地址，则表达式"s are red\n"是什么呢？为了与 cout 对字符串输出的处理保持一致，这个被引号括起的字符串也应当是一个地址。在 C++中，用引号括起的字符串像数组名一样，也是第一个元素的地址。上述代码不会将整个字符串发送给 cout，而只是发送该字符串的地址。这意味着对于数组中的字符串、用引号括起的字符串常量以及指针所描述的字符串，处理的方式是一样的，都将传递它们的地址。与逐个传递字符串中的所有字符相比，这样做的工作量确实要少。

记住：在 cout 和多数 C++表达式中，char 数组名、指向 char 的指针以及用引号括起的字符串常量都被解释为字符串第一个字符的地址。

程序清单 4.20 演示了如何使用不同形式的字符串。它使用了两个字符串库中的函数。函数 strlen()我们以前用过，它返回字符串的长度。函数 strcpy()将字符串从一个位置复制到另一个位置。这两个函数的原型都位于头文件 cstring（在不太新的实现中，为 string.h）中。该程序还通过注释指出了应尽量避免错误使用指针的方式。

程序清单 4.20 ptrstr.cpp

```
// ptrstr.cpp -- using pointers to strings
#include <iostream>
#include <cstring>              // declare strlen(), strcpy()
int main()
{
   using namespace std;
   char animal[20] = "bear";    // animal holds bear
   const char * bird = "wren";  // bird holds address of string
   char * ps;                   // uninitialized

   cout << animal << " and ";   // display bear
   cout << bird << "\n";        // display wren
   // cout << ps << "\n";       //may display garbage, may cause a crash

   cout << "Enter a kind of animal: ";
   cin >> animal;               // ok if input < 20 chars
   // cin >> ps; Too horrible a blunder to try; ps doesn't
   //          point to allocated space

   ps = animal;                 // set ps to point to string
   cout << ps << "s!\n";        // ok, same as using animal
   cout << "Before using strcpy(): \n";
   cout << animal << " at " << (int *) animal << endl;
   cout << ps << " at " << (int *) ps << endl;

   ps = new char[strlen(animal) + 1];  // get new storage
   strcpy(ps, animal);         // copy string to new storage
   cout << "After using strcpy(): \n";
   cout << animal << " at " << (int *) animal << endl;
   cout << ps << " at " << (int *) ps << endl;
   delete [] ps;
   return 0;
}
```

注意：如果所用的系统中没有 cstring 头文件，请使用老版本 string.h。

下面是该程序的运行情况：

```
bear and wren
Enter a kind of animal: fox
foxs!
Before using strcpy():
fox at 0x0065fd30
fox at 0x0065fd30
After using strcpy():
fox at 0x0065fd30
fox at 0x004301c8
```

程序说明

程序清单 4.20 中的程序创建了一个 char 数组（animal）和两个指向 char 的指针变量（bird 和 ps）。该程序首先将 animal 数组初始化为“bear”字符串，就像初始化数组一样。然后，程序执行了一些新的操作，将 char 指针初始化为指向一个字符串：

```
const char * bird = "wren"; // bird holds address of string
```

记住，“wren”实际表示的是字符串的地址，因此这条语句将“wren”的地址赋给了 bird 指针（一般来说，编译器在内存留出一些空间，以存储程序源代码中所有用引号括起的字符串，并将每个被存储的字符串与其地址关联起来）。这意味着可以像使用字符串“wren”那样使用指针 bird，如下面的范例所示：

```
cout << "A cocerned" << bird << "speaks\n".
```

字符串字面值是常量，这就是为什么代码在声明中使用关键字 const 的原因。以这种方式使用 const 意味着可以用 bird 来访问字符串，但不能修改它。第 7 章将详细介绍 const 指针。最后，指针 ps 未被初始化，因此不指向任何字符串（正如读者知道的，这通常是个坏主意，这里也不例外）。

接下来，程序说明了这样一点，即对于 cout 来说，使用数组名 animal 和指针 bird 是一样的。毕竟，它们都是字符串的地址，cout 将显示存储在这两个地址上的两个字符串（“bear”和“wren”）。如果激活错误地显示 ps 的代码，则将可能显示一个空行、一堆乱码，或者程序将崩溃。创建未初始化的指针有点像签发一张空头支票：无法控制它将被如何使用。

对于输入，情况有点不同。只要输入比较短，能够被存储在数组中，则使用数组 animal 进行输入将是安全的。不过，使用 bird 来进行输入并不合适：

- 有些编译器将字符串字面值视为只读常量，如果试图修改它们，将导致运行阶段错误。在 C++中，字符串字面值都将被视为常量，但并不是所有的编译器都对以前的行为做了这样的修改。
- 有些编译器只使用字符串字面值的一个副本来表示程序中所有的该字面值。

下面讨论一下第二点。C++不能保证字符串字面值被惟一地存储。也就是说，如果在程序中多次使用了字符串字面值“wren”，则编译器将可能存储该字符串的多个副本，也可能只存储一个副本。如果是后面一种情况，则将 bird 设置为指向一个“wren”，将使它只是指向该字符串的惟一一个副本。将值读入一个字符串可能会影响被认为是独立的、位于其他地方的字符串。无论如何，由于 bird 指针被声明为 const，因此编译器将禁止改变 bird 指向的位置中的内容。

试图将信息读入 ps 指向的位置将更糟。由于 ps 没有被初始化，因此并不知道信息将被存储在哪里，这甚至可能改写内存中的信息。幸运的是，要避免这种问题很容易——只要使用足够大的 char 数组来接收输入即可。请不要使用字符串常量或未被初始化的指针来接收输入。为避免这些问题，也可以使用 std::string 对象，而不是数组。

警告：在将字符串读入程序时，应使用已分配的内存的地址。该地址可以是数组名，也可以是使用 new 初始化过的指针。

接下来，请注意下面的代码完成的工作：

```
ps = animal;              // set ps to point to string
...
cout << animal << " at " << (int *) animal << endl;
cout << ps << " at " << (int *) ps << endl;
```

它将生成下面的输出：

```
fox at 0x0065fd30
fox at 0x0065fd30
```

一般来说，如果给 cout 提供一个指针，它将打印地址。但如果指针的类型为 char *，则 cout 将显示指向的字符串。如果要显示的是字符串的地址，则必须将这种指针强制转换为另一种指针类型，如 int *（上面的代码就是这样做的）。因此，ps 被显示为字符串“fox”，而（int *）ps 被显示为该字符串的地址。注意，将 animal 赋给 ps 并不会复制字符串，而只是复制地址。这样，这两个指针将指向相同的内存单元和字符串。

要获得字符串的副本，还需要做其他工作。首先，需要分配内存来存储该字符串，这可以通过声明另一个数组或使用 new 来完成。后一种方法使得能够根据字符串的长度来指定所需的空间：

```
ps = new char[strlen(animal) + 1]; // get new storage
```

字符串“fox”不能填满整个 animal 数组，因此这样做浪费了空间。上述代码使用 strlen() 来确定字符串的长度，并将它加 1 来获得包含空字符时该字符串的长度。随后，程序使用 new 来分配刚好足够存储该字符串的空间。

接下来，需要将 animal 数组中的字符串复制到新分配的空间中。将 animal 赋给 ps 是不可行的，因为这样只能修改存储在 ps 中的地址，从而失去程序访问新分配内存的惟一途径。需要使用库函数 strcpy()：

```
strcpy(ps, animal);              // copy string to new storage
```

strcpy() 函数接受 2 个参数。第一个是目标地址，第二个是要复制的字符串的地址。您应确定，分配了目标空间，并有足够的空间来存储副本。在这里，我们用 strlen() 来确定所需的空间，并使用 new 获得可用的内存。

通过使用 strcpy() 和 new，将获得“fox”的两个独立副本：

```
fox at 0x0065fd30
fox at 0x004301c8
```

另外，new 在离 animal 数组很远的地方找到了所需的内存空间。

经常需要将字符串放到数组中。初始化数组时，请使用=操作符；否则应使用 strcpy() 或 strncpy()。strcpy() 在前面已经介绍过，其工作原理如下：

```
char food[20] = "carrots"; // initialization
strcpy(food, "flan");      // otherwise
```

注意，类似下面这样的代码：

```
strcpy(food, "a picnic basket filled with many goodies");
```

这样可能会产生问题，因为 food 数组比字符串小。在这种情况下，函数将字符串中剩余的部分复制到数组后面的内存字节中，这可能会覆盖程序正在使用的其他内存。要避免这种问题，请使用 strncpy()。该函数还接受第 3 个参数——要复制的最大字符数。不过，要注意的是，如果该函数在到达字符串结尾之前，目标内存已经用完，则它将不会添加空字符。因此，应该这样使用该函数：

```
strncpy(food, "a picnic basket filled with many goodies", 19);
food[19] = '\0';
```

这样最多将 19 个字符复制到数组中，然后将最后一个元素设置成空字符。如果该字符串少于 19 个字符，则 strncpy() 将在复制完该字符串之后加上空字符，以标记该字符串的结尾。

记住：应使用 strcpy() 或 strncpy()，而不是赋值操作符来将字符串赋给数组。

读者对使用 C-风格的字符串和 cstring 库的一些方面有了了解后，便可以理解为何使用 C++ string 类型更为简单了：您不用担心字符串会导致数组越界，并可以使用赋值操作符而不是函数 strcpy() 和 strncpy()。

4.8.3 使用 new 创建动态结构

在运行时创建数组优于在编译时创建数组，对于结构也是如此。需要在程序运行时为结构分配所需的空间，这也可以使用 new 操作符来完成。通过使用 new，可以创建动态结构。同样，“动态”意味着内存是在运行时，而不是编译时分配的。由于类与结构非常相似，因此本节介绍的有关结构的技术也适用于类。

将 new 用于结构由两步组成：创建结构和访问其成员。要创建结构，需要同时使用结构类型和 new。

例如，要创建一个未命名的 inflatable 类型，并将其地址赋给一个指针，可以这样做：

```
inflatable * ps = new inflatable;
```

这将把足以存储 inflatable 结构的一块可用内存的地址赋给 ps。这种句法和 C++的内置类型完全相同。

比较棘手的一步是访问成员。创建动态结构时，不能将成员操作符句点用于结构名，因为这种结构没有名称，只是知道它的地址。C++专门为这种情况提供了一个操作符：箭头成员操作符（->）。该操作符由连字符和大于号组成，可用于指向结构的指针，就像点操作符可用于结构名一样。例如，如果 ps 指向一个 inflatable 结构，则 ps->price 是被指向的结构的 price 成员（参见图 4.11）。

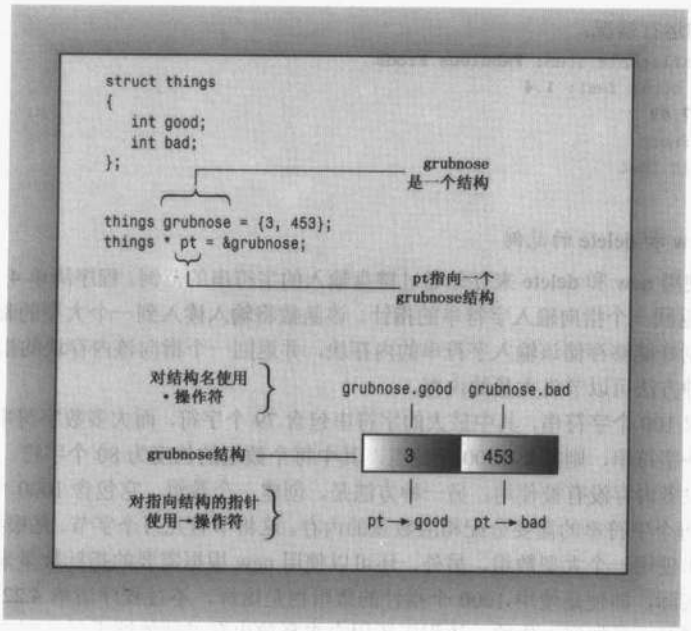


图 4.11　标识结构成员

记住：有时，C++新手在指定结构成员时，搞不清楚何时应使用句点操作符，何时应使用箭头操作符。规则非常简单。如果结构标识符是结构名，则使用句点操作符；如果标识符是指向结构的指针，则使用箭头操作符。

另一种访问结构成员的方法是，如果 ps 是指向结构的指针，则*ps 就是被指向的值——结构本身。由于*ps 是一个结构，因此（*ps）.price 是该结构的 price 成员。C++的操作符优先规则要求使用括号。

程序清单 4.21 使用 new 创建一个未命名的结构，并演示了两种访问结构成员的指针表示法。

程序清单 4.21　newstrct.cpp

```
// newstrct.cpp -- using new with a structure
#include <iostream>
struct inflatable   // structure template
{
    char name[20];
    float volume;
    double price;
};
int main()
{
    using namespace std;
    inflatable * ps = new inflatable; // allot memory for structure
    cout << "Enter name of inflatable item: ";
    cin.get(ps->name, 20);          // method 1 for member access
```

```
    cout << "Enter volume in cubic feet: ";
    cin >> (*ps).volume;          // method 2 for member access
    cout << "Enter price: $";
    cin >> ps->price;
    cout << "Name: " << (*ps).name << endl;                // method 2
    cout << "Volume: " << ps->volume << " cubic feet\n";   // method 1
    cout << "Price: $" << ps->price << endl;               // method 1
    delete ps;                    // free memory used by structure
    return 0;
}
```

下面是该程序的运行情况：

```
Enter name of inflatable item: Fabulous Frodo
Enter volume in cubic feet: 1.4
Enter price: $27.99
Name: Fabulous Frodo
Volume: 1.4 cubic feet
Price: $27.99
```

1. 一个使用 new 和 delete 的范例

下面介绍一个使用 new 和 delete 来存储通过键盘输入的字符串的范例。程序清单 4.22 定义了一个函数 getname()，该函数返回一个指向输入字符串的指针。该函数将输入读入到一个大型的临时数组中，然后使用 new [ ]创建一个刚好能够存储该输入字符串的内存块，并返回一个指向该内存块的指针。对于读取大量字符串的程序，这种方法可以节省大量的内存。

假设程序要读取 100 个字符串，其中最大的字符串包含 79 个字符，而大多数字符串都短得多。如果用 char 数组来存储这些字符串，则需要 1000 个数组，其中每个数组的长度为 80 个字符。这总共需要 80 000 个字节，而其中的很多内存没有被使用。另一种方法是，创建一个数组，它包含 1000 个指向 char 的指针，然后使用 new 根据每个字符串的需要分配相应数量的内存。这将节省几万个字节。是根据输入来分配内存，而不是为每个字符串使用一个大型数组。另外，还可以使用 new 根据需要的指针数量来分配空间。就目前而言，这有点不切实际，即使是使用 1000 个指针的数组也是这样，不过程序清单 4.22 还是演示了一些技巧。另外，为演示 delete 是如何工作的，该程序还用它来释放内存以便能够重新使用。

程序清单 4.22 delete.cpp

```
// delete.cpp -- using the delete operator
#include <iostream>
#include <cstring>     // or string.h
using namespace std;
char * getname (void);    // function prototype
int main()
{
    char * name;       // create pointer but no storage

    name = getname();  // assign address of string to name
    cout << name << " at " << (int *) name << "\n";
    delete [] name;    // memory freed

    name = getname();    // reuse freed memory
    cout << name << " at " << (int *) name << "\n";
    delete [] name;      // memory freed again
    return 0;
}

char * getname()       // return pointer to new string
{
    char temp[80];     // temporary storage
    cout << "Enter last name: ";
    cin >> temp;
```

```
    char * pn = new char[strlen(temp) + 1];
    strcpy(pn, temp);    // copy string into smaller space

    return pn;           // temp lost when function ends
}
```

下面是该程序的运行情况：

```
Enter last name: Fredeldumpkin
Fredeldumpkin at 0x004326b8
Enter last name: Pook
Pook at 0x004301c8
```

2. 程序说明

来看一下程序清单 4.22 中的函数 getname()。它使用 cin 将输入的单词放到 temp 数组中，然后使用 new 分配新内存，以存储该单词。程序需要 strlen（temp）+ 1 个字符（包括空字符）来存储该字符串，因此将这个值提供给 new。获得空间后，getname() 使用标准库函数 strcpy() 将 temp 中的字符串复制到新的内存块中。该函数并不检查内存块是否能够容纳字符串，但 getname() 通过使用 new 请求合适的字节数来完成了这样的工作。最后，函数返回 pn，这是字符串副本的地址。

在 main() 中，返回值（地址）被赋给指针 name。该指针是在 main() 中定义的，但它指向 getname() 函数中分配的内存块。然后，程序打印该字符串及其地址。

接下来，在释放 name 指向的内存块后，main() 再次调用 getname()。C++不保证新释放的内存就是下一次使用 new 时选择的内存，从程序运行结果可知，确实不是。

在这个例子中，getname() 分配内存，而 main() 释放内存。将 new 和 delete 放在不同的函数中通常并不是个好办法，因为这样很容易忘记使用 delete。不过这个例子确实把 new 和 delete 分开放置了，只是为了说明这样做也是可以的。

为了解该程序的一些更为微妙的方面，需要知道一些有关 C++是如何处理内存的知识。下面介绍一些这样的知识，这些知识将在第 9 章做全面的介绍。

4.8.4　自动存储、静态存储和动态存储

根据用于分配内存的方法，C++有 3 种管理数据内存的方式：自动存储、静态存储和动态存储（有时也叫作自由存储空间或堆）。在存在时间的长短方面，以这 3 种方式分配的数据对象各不相同。下面简要地介绍一下每种类型。

1. 自动存储

在函数内部定义的常规变量使用自动存储空间，被称为自动变量（automatic variable），这意味着它们在所属的函数被调用时自动产生，在该函数结束时消亡。例如，程序清单 4.22 中的 temp 数组仅当 getname() 函数活动时存在。当程序控制权回到 main() 时，temp 使用的内存将自动被释放。如果 getname() 返回 temp 的地址，则 main() 中的 name 指针指向的内存将很快得到重新使用。这就是在 getname() 中使用 new 的原因之一。

实际上，自动变量是一个局部变量，其作用域为包含它的代码块。代码块是被包含在花括号中的一段代码。到目前为止，我们使用的所有代码块都是整个函数。不过，在下一章将会看到，函数内也可以有代码块。如果在其中的某个代码块定义了一个变量，则该变量仅在程序执行该代码块中的代码时存在。

2. 静态存储

静态存储是整个程序执行期间都存在的存储方式。使变量成为静态的方式有两种：一种是在函数外面定义它；另一种是在声明变量时使用关键字 static：

```
static double fee = 56.50;
```

在 K&R C 中，只能初始化静态数组和静态结构，而 C++ 版本 2.0（及后续版本）和 ANSI C 中，也可以初始化自动数组和自动结构。不过，一些读者可能已经发现，有些 C++实现还不支持对自动数组和自动

结构的初始化。

第 9 章将详细介绍静态存储。自动存储和静态存储的关键在于：这些方法严格地限制了变量的寿命。变量可能存在于程序的整个生命周期（静态变量），也可能只是在特定函数被执行时存在（自动变量）。

3. 动态存储

new 和 delete 操作符提供了一种比自动变量和静态变量更灵活的方法。它们管理了一个内存池，这在 C++中被称为自由存储空间（free store）。内存池同用于静态变量和自动变量的内存是分开的。程序清单 4.22 表明，new 和 delete 允许在一个函数中分配内存，而在另一个函数中释放它。因此，数据的生命周期就不完全受到程序或函数的生存时间的控制了。与使用常规变量相比，使用 new 和 delete 使程序员对程序如何使用内存有更大的控制权。

堆栈、堆和内存泄漏

如果使用 new 操作符在自由存储空间（或堆）上创建变量后，没有调用 delete，将发生什么情况呢？如果没有调用 delete，则即使包含指针的内存由于作用域规则和对象生命周期的原因而被释放，在自由存储空间上动态分配的变量或结构也将继续存在。实际上，将会无法访问自由存储空间中的结构，因为指向这些内存的指针无效。这将导致内存泄漏。被泄漏的内存将在程序的整个生命周期内都不可使用；这些内存被分配出去，但无法收回。极端情况（不过不常见）是，内存泄漏可能会非常严重，以致于应用程序可用的内存被耗尽，出现内存耗尽错误，导致程序崩溃。另外，这种泄漏还会给一些操作系统或在相同的内存空间中运行的应用程序带来负面影响，导致它们也相应崩溃。

即使是最好的程序员和软件公司，也可能导致内存泄漏。要避免内存泄漏，最好是养成这样一种习惯，即同时使用 new 和 delete 操作符，在自由存储空间上动态分配内存，随后便释放它。

注意：指针是功能最强大的 C++工具之一，但也最危险，因为它们允许执行对计算机不友好的操作，如使用未经初始化的指针来访问内存或者试图释放同一个内存块两次。另外，在通过实践习惯指针表示法和指针概念之前，指针是容易引起迷惑的。由于指针是 C++编程的重要组成部分，本书后面将更详细地讨论它。本书多次对指针进行了讨论，就是希望读者能够越来越熟悉它。

4.9 总结

数组、结构和指针是 C++的 3 种复合类型。数组可以在一个数据对象中存储多个同种类型的值。通过使用索引或下标，可以访问数组中各个元素。

结构可以将多个不同类型的值存储在同一个数据对象中，可以使用成员关系操作符（.）来访问其中的成员。使用结构的第一步是创建结构模板，它定义结构存储了哪些成员。模板的名称将成为新类型的标识符，然后就可以声明这种类型的结构变量。

共用体可以存储一个值，但是这个值可以是不同的类型，成员名指出了使用的模式。

指针是被设计用来存储地址的变量的。我们说，指针指向它存储的地址。指针声明指出了指针指向的对象的类型。对指针应用解除引用操作符，将得到指针指向的位置中的值。

字符串是以空字符为结尾的一系列字符。字符串可用引号括起的字符串常量表示，其中隐式包含了结尾的空字符。可以将字符串存储在 char 数组中，可以用被初始化为指向字符串的 char 指针表示字符串。函数 strlen()返回字符串的长度，其中不包括空字符。函数 strcpy()将字符串从一个位置复制到另一个位置。在使用这些函数时，应当包含头文件 cstring 或 string.h。

头文件 string 支持的 C++ string 类提供了另一种对用户更为友好的字符串处理方法。具体地说，string 对象将根据要存储的字符串自动调整其大小，用户可以使用赋值操作符来复制字符串。

new 操作符允许在程序运行时为数据对象请求内存。该操作符返回获得内存的地址，可以将这个地址赋给一个指针，程序将只能使用该指针来访问这块内存。如果数据对象是简单变量，则可以使用解除引用

操作符（*）来获得其值；如果数据对象是数组，则可以像使用数组名那样使用指针来访问元素；如果数据对象是结构，则可以用指针解除引用操作符（->）来访问其成员。

指针和数组紧密相关。如果 ar 是数组名，则表达式 ar[i]被解释为*（ar + i），其中数组名被解释为数组第一个元素的地址。这样，数组名的作用和指针相同。反过来，可以使用数组表示法，通过指针名来访问 new 分配的数组中的元素。

操作符 new 和 delete 允许显式控制何时给数据对象分配内存，何时将内存归还给内存池。自动变量是在函数中声明的变量，而静态变量是在函数外部或者使用关键字 static 声明的变量，这两种变量都不太灵活。自动变量在程序执行到其所属的代码块（通常是函数定义）时产生，在离开该代码块时终止。静态变量在整个程序周期内都存在。

4.10　复习题

1．如何声明下述数据？

a．actor 是由 30 个 char 组成的数组。

b．betsie 是由 100 个 short 组成的数组。

c．chuck 是由 13 个 float 组成的数组。

d．dipsea 是由 64 个 long double 组成的数组。

2．声明一个包含 5 个元素的 int 数组，并将它初始化为前 5 个正奇数。

3．编写一条语句，将问题 2 中数组的第一个元素和最后一个元素的和赋给变量 even。

4．编写一条语句，显示 float 数组 ideas 中的第 2 个元素的值。

5．声明一个 char 的数组，并将其初始化为字符串“cheeseburger”。

6．设计一个描述鱼的结构声明。结构中应当包括品种、重量（整数，单位为盎司）和长度（英寸，包括小数）。

7．声明一个问题 6 中定义的结构的变量，并对它进行初始化。

8．用 enum 定义一个名为 Response 的类型，它包含 Yes、No 和 Maybe 等枚举量，其中 Yes 的值为 1，No 为 0，Maybe 为 2。

9．假设 ted 是一个 double 变量，请声明一个指向 ted 的指针，并使用该指针来显示 ted 的值。

10．假设 treacle 是一个包含 10 个元素的 float 数组，请声明一个指向 treacle 的第一个元素的指针，并使用该指针来显示数组的第一个元素和最后一个元素。

11．编写一段代码，要求用户输入一个正整数，然后创建一个动态的 int 数组，其中包含的元素数目等于用户输入的值。

12．下面的代码是否有效？如果有效，它将打印出什么结果？

```
cout << (int *) "Home of the jolly bytes";
```

13．编写一段代码，给问题 6 中描述的结构动态分配内存，然后读取该结构的成员的值。

14．程序清单 4.6 指出了混合输入数字和一行字符串时存储的问题。如果将下面的代码：

```
cin.getline (address, 80);
```

替换为：

```
cin >> address;
```

将对程序的运行带来什么影响？

4.11　编程练习

1．编写一个 C++程序，如下述输出范例所示的那样请求并显示信息：

```
What is your first name? Betty Sue
```

```
What is your last name? Yew
What letter grade do you deserve? B
What is your age? 22
Name: Yew, Betty Sue
Grade: C
Age: 22
```

注意，该程序应该接受的名字包含多个单词。另外，程序将向下调整成绩，即向上调一个字母。假设用户请求 A、B 或 C，所以不必担心 D 和 F 之间的空档。

2. 修改程序清单 4.4，使用 C++ string 类而不是 char 数组。

3. 编写一个程序，它要求用户首先输入其名，然后输入其姓；然后程序使用一个逗号和空格将姓和名组合起来，并存储和显示组合结果。请使用 char 数组和头文件 cstring 中的函数。下面是该程序运行时的情形：

```
Enter your first name: Flip
Enter your last name: Fleming
Here's the information in a single string: Fleming, Flip
```

4. 编写一个程序，它要求用户首先输入其名，然后输入其姓；然后程序使用一个逗号和空格将姓和名组合起来，并存储和显示组合结果。请使用 string 对象和头文件 string 中的函数。下面是该程序运行时的情形：

```
Enter your first name: Flip
Enter your last name: Fleming
Here's the information in a single string: Fleming, Flip
```

5. 结构 CandyBar 包含 3 个成员。第一个成员存储了糖块的品牌；第二个成员存储糖块的重量（可以有小数）；第三个成员存储了糖块的卡路里含量（整数）。请编写一个程序，声明这个结构，创建一个名为 snack 的 CandyBar 变量，并将其成员分别初始化为“Mocha Munch”、2.3 和 350。初始化应在声明 snack 时进行。最后，程序显示 snack 变量的内容。

6. 结构 CandyBar 包含 3 个成员，如编程练习 5 所示。请编写一个程序，创建一个包含 3 个元素的 CandyBar 数组，并将它们初始化为所选择的值，然后显示每个结构的内容。

7. William Wingate 从事比萨饼分析服务。对于每个比萨饼，他都需要记录下列信息：

- 比萨饼公司的名称，可以由多个单词组成。
- 比萨饼的直径。
- 比萨饼的重量。

请设计一个能够存储这些信息的结构，并编写一个使用这种结构变量的程序。程序将请求用户输入上述信息，然后显示这些信息。请使用 cin（或它的方法）和 cout。

8. 完成编程练习 7，但使用 new 来为结构分配内存，而不是声明一个结构变量。另外，让程序在请求输入比萨饼公司名称之前输入比萨饼的直径。

9. 完成编程练习 6，但使用 new 来动态分配数组，而不是声明一个包含 3 个元素的 CandyBar 数组。

第 5 章　循环和关系表达式

本章内容包括：

- for 循环。
- 表达式和语句。
- 递增操作符和递减操作符：++和--。
- 组合赋值操作符。
- 复合语句（语句块）。
- 逗号操作符。
- 关系操作符：>、>=、==、<=、<和!=。
- while 循环。
- typedef 工具。
- do while 循环。
- get() 字符输入方法。
- 文件尾条件。
- 嵌套循环和二维数组。

计算机除了存储数据外，还可以做很多其他的工作。可以对数据进行分析、合并、重组、抽取、修改、推断、合成以及其他操作。有时甚至会歪曲和破坏数据，不过我们应当尽量防止这种行为的发生。为了发挥其强大的操控能力，程序需要有执行重复的操作和进行决策的工具。当然，C++提供了这样的工具。事实上，它使用与常规 C 语言相同的 for 循环、while 循环、do while 循环、if 语句和 switch 语句，如果读者熟悉 C 语言，可粗略地浏览本章和第 6 章；但浏览速度不要过快，否则会错过 cin 如何处理字符输入。这些程序控制语句通常都使用关系表达式和逻辑表达式来控制其行为。本章将讨论循环和关系表达式，而第 6 章将介绍分支语句和逻辑表达式。

5.1　for 循环

很多情况下都需要程序执行重复的任务，如将数组中的元素累加起来或将歌颂生产的赞歌打印 20 份，C++中的 for 循环可以轻松地完成这种任务。我们来看看程序清单 5.1 中，以了解 for 循环所做的工作，然后讨论它是如何工作的。

程序清单 5.1　forloop.cpp

```
// forloop.cpp -- introducing the for loop
#include <iostream>
int main()
{
   using namespace std;
```

```
    int i;  // create a counter
//  initialize; test; update
    for (i = 0; i < 5; i++)
        cout << "C++ knows loops.\n";
    cout << "C++ knows when to stop.\n";
    return 0;
}
```

下面是该程序的输出：

```
C++ knows loops.
C++ knows loops.
C++ knows loops.
C++ knows loops.
C++ knows loops.
C++ knows when to stop.
```

该循环首先将整数变量 i 设置为 0：

```
i = 0
```

这是循环的初始化（loop initialization）部分。然后，循环测试（loop test）部分检查 i 是否小于 5：

```
i < 5
```

如果确实小于 5，则程序将执行接下来的语句——循环体（loop body）：

```
cout << "C++ knows loops.\n";
```

然后，程序使用循环更新（loop update）部分将 i 加 1：

```
i++
```

这里使用了++操作符——递增操作符（increment operator），它将操作数的值加 1。递增操作符并不仅限于用于 for 循环。例如，在程序中，可以使用 i++;来替换语句 i = i + 1;。将 i 加 1 后，便结束了循环的第一个周期。

接下来，循环开始了新的周期，将新的 i 值与 5 进行比较。由于新值（1）也小于 5，因此循环打印另一行，然后再次将 i 加 1，从而结束这一周期。这样又进入了新的一轮测试、执行语句和更新 i 的值。这一过程将一直进行下去，直到循环将 i 更新为 5 为止。这样，接下来的测试失败，程序将接着执行循环后的语句。

5.1.1 for 循环的组成部分

for 循环为执行重复的操作提供了循序渐进的步骤。我们来具体看一看它是如何工作的。for 循环的组成部分完成下面这些步骤：

1. 设置初始值。
2. 执行测试，看看循环是否应当继续进行。
3. 执行循环操作。
4. 更新用于测试的值。

C++循环设计中包括了这些要素，将其识别出来一目了然。初始化、测试和更新操作这三部分构成了控制部分，这些操作由括号括起。其中每一部分都是一个表达式，彼此由分号隔开。控制部分后面的语句叫作循环体，只要测试表达式为 true，它便被执行：

```
for (initialization; test-expression; update-expression)
    body
```

C++语法将整个 for 看作一条语句——虽然循环体可以包含一条或多条语句。（包含多条语句时，需要使用复合语句或代码块，这将在本章后面进行讨论。）

循环只执行一次初始化。通常，程序使用该表达式将变量设置为起始值，然后用该变量计算循环周期。

test-expression（测试表达式）决定循环体是否被执行。通常，这个表达式是关系表达式，即对两个值进行比较。这个例子将 i 的值同 5 进行比较，看 i 是否小于 5。如果比较结果为真，则程序将执行循环体。实际上，C++并没有将 test-expression 的值限制为只能为真或假。可以使用任意表达式，C++将把结果强制转换为 bool 类型。因此，值为 0 的表达式将被转换为 bool 值 false，导致循环结束。如果表达式的值为非

零，则被强制转换为 bool 值 true，循环将继续进行。程序清单 5.2 通过将表达式 i 用作测试条件来演示了这一特点。更新部分的 i--与 i++相似，只是每使用一次，i 值就减 1。

程序清单 5.2　num_test.cpp

```
// num_test.cpp -- use numeric test in for loop
#include <iostream>
int main()
{
   using namespace std;
   cout << "Enter the starting countdown value: ";
   int limit;
   cin >> limit;
   int i;
   for (i = limit; i; i--)     // quits when i is 0
      cout << "i = " << i << "\n";
   cout << "Done now that i = " << i << "\n";
   return 0;
}
```

下面是该程序的输出：

```
Enter the starting countdown value: 4
i = 4
i = 3
i = 2
i = 1
Done now that i = 0
```

注意，循环在 i 变为 0 后结束。

关系表达式（如 i<5）是如何得到循环终止值 0 的呢？在引入 bool 类型之前，如果关系表达式为 true，则被判定为 1；如果为 false，则被判定为 0。因此，表达式 3<5 的值为 1，而 5<5 的值为 0。不过，C++添加了 bool 类型后，关系表达式就判定为 bool 字面值 true 和 false，而不是 1 和 0 了。这种变化不会导致不兼容的问题，因为 C++程序在需要整数值的地方将把 true 和 false 分别转换为 1 和 0，而在需要 bool 值的地方将把 0 转换为 false，非 0 转换为 true。

for 循环是入口条件（entry-condition）循环。这意味着在每轮循环之前，都将计算测试表达式的值，当测试表达式为 false 时，将不会执行循环体。例如，假设重新运行程序清单 5.2 中的程序，但将起始值设置为 0，则由于测试条件在首次被判定时便为 false，循环体将不被执行：

```
Enter the starting countdown value: 0
Done now that i = 0
```

这种在循环之前进行检查的方式可以让程序少出错。

update-expression（更新表达式）在每轮循环结束时执行，此时循环体已经执行完毕。通常，它用来对跟踪循环轮次的变量的值进行增减。不过，它可以是任何有效的 C++表达式，还可以是其他控制表达式。这使 for 循环的功能不仅仅是从 0 数到 5（这是第一个循环范例所做的工作），稍后将介绍一些例子。

for 循环体由一条语句组成，不过很快将介绍如何扩展这条规则。图 5.1 对 for 循环设计进行了总结。

for 语句看上去有些像函数调用，因为它使用一个后面跟一对括号的名称。不过，for 是一个 C++关键字，因此编译器不会将 for 视为一个函数，这将防止将函数命名为 for。

提示：C++常用的方式是，在 for 和括号之间加上一个空格，而省略函数名与括号之间的空格：

```
for (int i = 6; i < 10; i++)
    smart_function (i);
```

对于其他控制语句（如 if 和 while），处理方式与 for 相似。这样从视觉上强化了控制语句和函数调用之间的区别。另外，常见的做法是缩进 for 语句体，使它看上去比较显著。

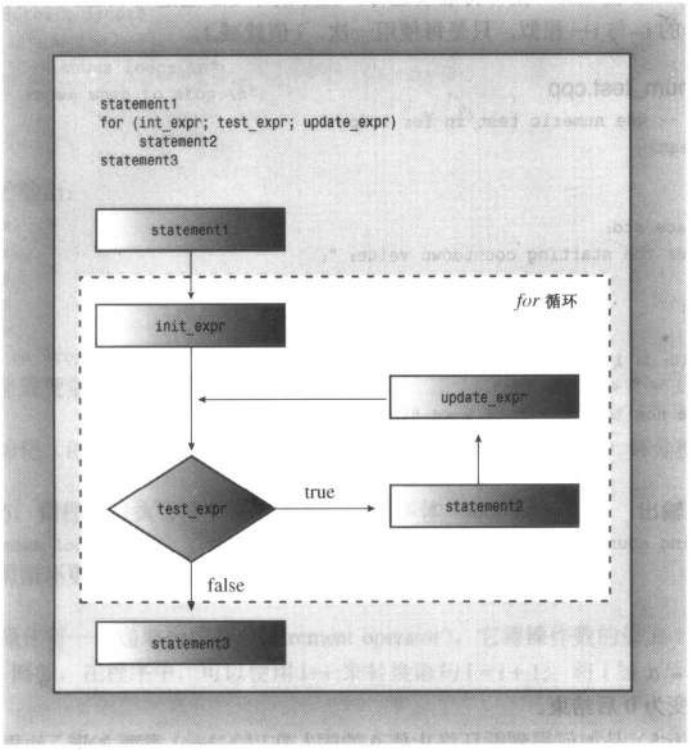


图 5.1 for 循环

1. 表达式和语句

for 语句的控制部分使用 3 个表达式。由于其自身强加的句法限制，C++成为非常具有表现力的语言。任何值或任何有效的值和操作符的组合都是表达式。例如，10 是值为 10 的表达式（一点都不奇怪），28 * 20 是值为 560 的表达式。在 C++中，每个表达式都有值。通常值是很明显的。例如，下面的表达式：

```
22 + 27
```

是由两个值和一个加号组成，它的值为 49。有时值不这么明显，例如：

```
x = 20
```

是一个表达式，因为它由两个值和一个赋值操作符组成。C++将赋值表达式的值定义为左侧成员的值，因此这个表达式的值为 20。由于赋值表达式有值，因此可以编写下面这样的语句：

```
maids = (cooks = 4) + 3;
```

表达式 cooks = 4 的值为 4，因此 maids 的值为 7。不过，C++虽然允许这样做，但并不意味着应鼓励这种做法。允许存在上述语句存在的原则也允许编写如下的语句：

```
x = y = z = 0;
```

这种方法可以快速地将若干个变量设置为相同的值。优先级表（见附录 D）表明，赋值操作符是从右向左结合的，因此首先将 0 赋给 z，然后将 z = 0 赋给 y，依此类推。

最后，正如前面指出的，像 x<y 这样的关系表达式将被判定为 bool 值 true 或 false。程序清单 5.3 中的小程序指出了有关表达式值的一些重要方面。<<操作符的优先级比表达式中使用的操作符高，因此代码使用括号来获得正确的运算顺序。

程序清单 5.3 express.cpp

```
// express.cpp -- values of expressions
#include <iostream>
using namespace std;
int main()
```

```
{
    int x;

    cout << "The expression x = 100 has the value ";
    cout << (x = 100) << endl;
    cout << "Now x = " << x << endl;
    cout << "The expression x < 3 has the value ";
    cout << (x < 3) << endl;
    cout << "The expression x > 3 has the value ";
    cout << (x > 3) << endl;
    cout.setf (ios_base:: boolalpha);  //a newer C++ feature
    cout << "The expression x < 3 has the value ";
    cout << (x < 3) << endl;
    cout << "The expression x > 3 has the value ";
    cout << (x > 3) << endl;
    return 0;
}
```

注意：老式 C++实现可能要求使用 ios: boolalpha，而不是 ios_base:: boolalpha 来作为 cout.setf() 的参数。有些老式实现甚至无法识别这两种形式。

下面是该程序的输出：

```
The expression x = 100 has the value 100
Now x = 100
The expression x < 3 has the value 0
The expression x > 3 has the value 1
The expression x < 3 has the value false
The expression x > 3 has the value true
```

通常，cout 在显示 bool 值之前将它们转换为 int，但 cout.setf（ios:: boolalpha）函数调用设置了一个标记，该标记命令 cout 显示 true 和 false，而不是 1 和 0。

记住：C++表达式是值或值与操作符的组合，每个 C++表达式都有值。

为判定表达式 x = 100，C++必须将 100 赋给 x。当判定表达式的值这种操作改变了内存中数据的值时，我们说表达式有副作用（side effect）。因此，判定赋值表达式会带来这样的副作用，即修改被赋值者的值。有可能把赋值看作预期的效果，但从 C++的构造方式这个角度来看，判定表达式才是主要作用。并不是所有的表达式都有副作用。例如，判定 x + 15 将计算出一个新的值，但不会修改 x 的值。不过，判定++x + 15 就有副作用，因为它将 x 加 1。

从表达式到语句的转变是很小的一步，只要加一个分号就可以完成。因此：

```
age = 100
```

是表达式，而：

```
age = 100;
```

是语句。只要加上分号，所有的表达式都可以成为语句，但不一定有编程意义。例如，如果 rodents 是一个变量，则：

```
rodents + 6;    // valid, but useless, statement
```

就是有效的 C++语句。编译器允许这样的语句，但它没有完成任何有用的工作。程序仅仅是计算和，而没有使用得到的结果，然后便进入下一条语句（智能的编译器甚至可能跳过这条语句。）

2. 非表达式和语句

有些概念对于理解 C++至关重要，如了解 for 循环的结构。不过句法中也有一些相对次要的内容，让认为自己理解语言的人突然觉得不知所措。下面来看看这样的内容。

对任何表达式加上分号都可以成为语句，但是这句话反过来说就不对了。也就是说，从语句中删除分号，并不一定能将它转换为表达式。就我们目前使用的语句而言，返回语句、声明语句和 for 语句都不满足“语句=表达式+分号”这种模式。例如，下面是一条语句：

```
int toad;
```

但 int toad 并不是表达式，因为它没有值。因此，下面的代码是非法的：

```
eggs = int toad * 1000;   // invalid, not an expression
cin >> int toad;          // can't combine declaration with cin
```

同样，不能把 for 循环赋给变量。在下面的范例中，for 循环不是表达式，因此没有值，也不能给它赋值：

```
int fx = for (int i = 0; i< 4; i++)
    cout >> i;   // not possible
```

3. 修改规则

C++在 C 循环的基础上添加了一项特性，要求对 for 循环句法做一些微妙的调整。

这是原来的句法：

```
for (expression; expression; expression)
    statement
```

具体地说，正如本章前面指出的，for 结构的控制部分由 3 个表达式组成，它们由分号分隔。不过，C++循环允许像下面这样做：

```
for (int i = 0; i < 5; i++)
```

也就是说，可以在 for 循环的初始化部分中声明变量。这很方便，但并不适用于原来的句法，因为声明不是表达式。这种一度是非法的行为最初是通过定义一种新的表达式——声明语句表达式（declaration-statement expression）——来合法化的，声明语句表达式不带分号声明，只能出现在 for 语句中。不过，这种调整已经被取消了，代之以将 for 语句的句法修改成下面这样：

```
for (for-init-statement condition; expression)
    statement
```

乍一看很奇怪，因为这里只有一个分号（而不是两个分号）。但是这是允许的，因为 for-init-statement 被视为一条语句，而语句有自己的分号。对于 for-init-statement 来说，它既可以是表达式语句，也可以是声明。这种句法规则用语句替换了后面跟分号的表达式，语句本身有自己的分号。总之，C++程序员希望能够在 for 循环初始化部分中声明和初始化变量，他们会做 C++句法需要和英语所允许的工作。

在 for-init-statement 中声明变量还有其实用的一面，这也是应该知道的。这种变量只存在于 for 语句中，也就是说，当程序离开循环后，这种变量将消失：

```
for (int i = 0; i < 5; i++)
   cout << "C++ knows loops.\n";
cout << i << endl;  // oops! i no longer defined
```

还应知道，有些 C++实现遵循以前的规则，对于前面的循环，将把 i 视为是在循环之前声明的，因此在循环结束后，i 仍可用。这种在 for 循环初始化部分声明变量的方式，导致的结果将随系统而异（至少目前是这样）。

警告：编写本书时，并不是所有的编译器都遵循当前的规则——在循环结束时，for 循环控制部分声明的变量将过期。例如，7.1 版之前的 Microsoft Visual C++默认遵守老式规则，这主要是由于原有的 Microsoft 库中的很多代码都是在采纳新规则之前编写的（7.1 版视图遵循新老规则，接受以两种方式编写的代码）。

5.1.2 回到 for 循环

我们来使用 for 循环完成更多的工作。程序清单 5.4 使用循环来计算并存储前 16 个阶乘。阶乘的计算方式如下：零阶乘写作 0!，被定义为 1。1!是 1*0!，即 1。2!为 2*1!，即 2。3!为 3*2!，即 6，依此类推。每个整数的阶乘都是该整数与前一个阶乘的乘积（钢琴家 Victor Borge 最著名的独白以其语音标点为特色，其中，惊叹号的发音就像 phffft pptz，带有潇湿的口音。不过，我们刚才提到的“!”读作“阶乘”）。该程序用一个循环来计算连续阶乘的值，并将这些值存储在数组中。然后，用另一个循环来显示结果。另外，该程序还在外部声明了一些值。

程序清单 5.4　formore.cpp

```
// formore.cpp -- more looping with for
#include <iostream>
using namespace std;
const int ArSize = 16;      // example of external declaration
int main()
{
    double factorials[ArSize];
    factorials[1] = factorials[0] = 1.0;
    //int i;
    for (i = 2; i < ArSize; i++)
        factorials[i] = i * factorials[i-1];
    for (i = 0; i < ArSize; i++)
        cout << i << "! = " << factorials[i] << endl;
    return 0;
}
```

下面是该程序的输出：

```
0! = 1
1! = 1
2! = 2
3! = 6
4! = 24
5! = 120
6! = 720
7! = 5040
8! = 40320
9! = 362880
10! = 3.6288e+006
11! = 3.99168e+007
12! = 4.79002e+008
13! = 6.22702e+009
14! = 8.71783e+010
15! = 1.30767e+012
```

阶乘增加得很快！

程序说明

该程序创建了一个数组来存储阶乘值。元素 0 存储 0!，元素 1 存储 1!，依此类推。由于前两个阶乘都等于 1，因此程序将 factorials 数组的前两个元素设置为 1（记住，数组第一个元素的索引值为 0）。然后，程序用循环将每个阶乘设置为索引号与前一个阶乘的乘积。该循环表明，可以在循环体中使用循环计数。

该程序演示了 for 循环如何通过提供一种访问每个数组成员的方便途径来与数组协同工作。另外，formore.cpp 还使用 const 创建了数组长度的符号表示(ArSize)。然后，它在需要数组长度的地方使用 ArSize，如定义数组以及限制循环如何处理数组时。现在，如果要将程序扩展成处理 20 个阶乘，则只需要将 ArSize 设置为 20 并重新编译程序即可。通过使用符号常量，就可以避免将所有的 10 修改为 20。

提示：通常，定义一个 const 值来表示数组中的元素个数是个好办法。在声明数组和引用数组长度时（如在 for 循环中），可以使用 const 值。

表达式 limit i < ArSize 反映了这样一个事实，包含 ArSize 个元素的数组的下标从 0 到 ArSize – 1，因此数组索引应当在 ArSize 减 1 的位置停止。也可以使用 i <= ArSize –1，但它看上去没有前面的表达式好。

该程序在 main() 的外面声明 const int 变量 ArSize。第 4 章末尾提到过，这样可以使 ArSize 成为外部数据。以这种方式声明 ArSize 的两种后果是，ArSize 在整个程序周期内存在、程序文件中所有的函数都可以使用它。在这个例子中，程序只有一个函数，因此在外部声明 ArSize 几乎没有任何实际用处，但包含多个函数的程序常常会受益于共享外部常量，因此我们现在就开始练习使用外部变量。

5.1.3 修改步长

到现在为止，循环范例每一轮循环都将循环计数加 1 或减 1。可以通过修改更新表达式来修改步长。例如，程序清单 5.5 中的程序按照用户选择的步长值将循环计数递增。它没有将 i++用作更新表达式，而是使用表达式 i = i + by，其中 by 是用户选择的步长值。

程序清单 5.5 bigstep.cpp

```
// bigstep.cpp -- count as directed
#include <iostream>
int main()
{
   using namespace std;
   cout << "Enter an integer: ";
   int by;
   cin >> by;
   cout << "Counting by " << by << "s: \n";
   for (int i = 0; i < 100; i = i + by)
      cout << i << endl;
   return 0;
}
```

下面是该程序的运行情况：

```
Enter an integer: 17
Counting by 17s:
0
17
34
51
68
85
```

当 i 的值到达 102 时，循环终止。这里的重点是，更新表达式可以是任何有效的表达式。例如，如果要求每轮递增以 i 的平方加 10，则可以使用表达式 i = i * i + 10。

5.1.4 使用 for 循环访问字符串

for 循环提供了一种依次访问字符串中每个字符的方式。例如，程序清单 5.6 让用户能够输入一个字符串，然后按相反的方向逐个字符地显示该字符串。在这个例子中，可以使用 string 对象，也可以使用 char 数组，因为它们都让您能够使用数组表示法来访问字符串中的字符。程序清单 5.6 使用的是 string 对象。string 类的 size()获得字符串中的字符数；循环在其初始化表达式中使用这个值，将 i 设置为字符串中最后一个字符的索引（不考虑空值字符）。为了反向计数，程序使用递减操作符（--），在每轮循环后将数组下标减 1。另外，程序清单 5.6 使用关系操作符大于或等于（>=）来测试循环是否到达第一个元素。稍后我们将对所有的关系操作符做一总结。

程序清单 5.6 forstr1.cpp

```
// forstr1.cpp -- using for with a string
#include <iostream>
#include <string>
int main()
{
   using namespace std;
   cout << "Enter a word: ";
   string word;
   cin >> word;

   // display letters in reverse order
   for (int i = word.size() - 1; i >= 0; i--)
```

```
    cout << word[i];
    cout << "\nBye.\n";
    return 0;
}
```

下面是该程序的运行情况：

```
Enter a word: animal
lamina
Bye.
```

程序成功地按相反的方向打印了 animal；与回文 redder 或 stats 相比，animal 能更清晰地说明这个程序的作用。

5.1.5　递增操作符（++）和递减操作符（--）

C++中有多个常被用在循环中的操作符，因此我们花一点时间来讨论它们。前面已经介绍了两个这样的操作符：递增操作符（++）（名称 C++由此得到）和递减操作符（--）。这两个操作符执行两种极其常见的循环操作：将循环计数加 1 或减 1。不过，它们还有很多特点不为读者所知。这两个操作符都有两种变体。前缀（prefix）版本位于操作数前面，如++x；后缀（postfix）版本位于操作数后面，如 x++。两个版本对操作数的影响是一样的，但是影响的时间不同。这就像对于钱包来说，清理草坪之前付钱和清理草坪之后付钱的最终结果是一样的，但支付钱的时间不同。程序清单 5.7 演示了递增操作符的这种差别。

程序清单 5.7　plus_one.cpp

```
// plus_one.cpp -- the increment operator
#include <iostream>
int main()
{
    using namespace std;
    int a = 20;
    int b = 20;

    cout << "a   = " << a << ";   b = " << b << "\n";
    cout << "a++ = " << a++ << "; ++b = " << ++b << "\n";
    cout << "a   = " << a << ";   b = " << b << "\n";
    return 0;
}
```

下面是该程序的输出：

```
a   = 20;  b = 20
a++ = 20; ++b = 21
a   = 21;  b = 21
```

粗略地讲，a++意味着使用 a 的当前值计算表达式，然后将 a 的值加 1；而++b 的意思是先将 b 的值加 1，然后使用新的值来计算表达式。例如，我们有下面这样的关系：

```
int x = 5;
int y = ++x;      // change x, then assign to y
                  // y is 6, x is 6
int z = 5;
int y = z++;      // assign to y, then change z
                  // y is 5, z is 6
```

递增和递减操作符是处理将值加减 1 这种常见任务的一种简约、方便的方法。

递增操作符和递减操作符都是漂亮的小型操作符，不过千万不要失去控制，在同一条语句对同一个值递增或递减多次。问题在于，规则“使用后修改”和“修改后使用”可能会变得模糊不清。也就是说，下面这条语句：

```
x = 2 * x++ * (3 - ++x);   // don't do it
```

在不同的系统上将生成不同的结果。对这种语句，C++没有定义正确的行为。

5.1.6　副作用和顺序点

下面更详细地介绍 C++就递增操作符何时生效的哪些方面做了规定，哪些方面没有规定。首先，副作

用（side effect）指的是在计算表达式时对某些东西（如存储在变量中的值）进行了修改；顺序点（sequence point）是程序执行过程中的一个点，在这里，进入下一步之前将确保对所有的副作用都进行了评估。在C++中，语句中的分号就是一个顺序点，这意味着程序处理下一条语句之前，赋值操作符、递增操作符和递减操作符执行的所有修改都必须完成。本章后面将讨论的有些操作也有顺序点。另外，任何完整的表达式末尾都是一个顺序点。

何为完整表达式呢？它是这样一个表达式：不是另一个更大表达式的子表达式。完整表达式的例子有：表达式语句中的表达式部分以及用作 while 循环中检测条件的表达式。

顺序点有助于阐明后缀递增何时进行。例如，请看下面的代码：

```
while (guests++ < 10)
    printf("%d \n", guests);
```

有时候，C++新手认为“使用值，然后递增”：在这里，这意味着先在 printf() 语句中使用 guests 的值，然后将其值加 1。然而，表达式 guests++ < 10 是一个完整表达式，因为它是一个 while 循环的测试条件，因此该表达式的末尾是一个顺序点。所以，C++确保副作用（将 guests 加 1）在程序进入 printf() 之前完成。然而，通过使用后缀格式，可确保将 guests 同 10 进行比较后再将其值加 1。

现在来看下面的语句：

```
y = (4 + x++) + (6 + x++);
```

表达式 4 + x++不是一个完整表达式，因此，C++不保证 x 的值在计算子表达式 4 + x++后立刻增加 1。在这个例子中，整条赋值语句是一个完整表达式，而分号标示了顺序点，因此 C++只保证程序执行到下一条语句之前，x 的值将被递增两次。C++没有规定是在计算每个子表达式之后将 x 的值递增，还是在整个表达式计算完毕后才将 x 的值递增，有鉴于此，您应避免使用这样的表达式。

5.1.7 前缀格式和后缀格式

显然，如果变量被用于某些目的（如用作函数参数或给变量赋值），使用前缀格式和后缀格式的结果将不同。然而，如果递增表达式的值没有被使用，情况又如何呢？例如，下面两条语句的作用是否不同？

```
x++;
++x;
```

下面两条语句的作用是否不同？

```
for (n = lim; n > 0; --n)
    ...;
```

和

```
for (n = lim; n > 0; n--)
    ...;
```

从逻辑上说，在上述两种情形下，使用前缀格式和后缀格式没有任何区别。表达式的值未被使用，因此只存在副作用。在上面的例子中，使用这些操作符的表达式为完整表达式，因此将 x 加 1 和 n 减 1 的副作用将在程序进入下一步之前完成，前缀格式和后缀格式的最终效果相同。

然而，虽然选择使用前缀格式还是后缀格式对程序的行为没有影响，但执行速度可能有细微的差别。对于内置类型和当代的编译器而言，这看似不是什么问题。然而，C++允许您针对类定义这些操作符，在这种情况下，用户这样定义前缀函数：将值加 1，然后返回结果；但后缀版本首先复制一个拷贝，将其加 1，然后将复制的拷贝返回。因此，对于类而言，前缀版本的效率比后缀版本高。

总之，对于内置类型，采用哪种格式不会有差别；但对于用户定义的且有用户定义的递增和递减操作符的类型，前缀格式的效率更高。

5.1.8 递增/递减操作符和指针

可以将递增操作符用于指针和基本变量。本书前面介绍过，将递增操作符用于指针时，将把指针的值增加其指向的数据类型占用的字节数，这种规则适用于对指针递增和递减：

```
double arr[5] = {21.1, 32.8, 23.4, 45.2, 37.4};
```

```
double *pt = arr;  // pt points to arr[0], i.e. to 21.1
++pt;              // pt points to arr[1], i.e. to 32.8
```

也可以结合使用这些操作符和*操作符来修改指针指向的值。将*和++同时用于指针时提出了这样的问题：将什么解除引用，将什么递增。这取决于操作符的位置和优先级。前缀递增、前缀递减和解除引用操作符的优先级相同，以从右到左的方式进行结合。后缀递增和后缀递减的优先级相同，但比前缀操作符的优先级高，这两个操作符以从左到右的方式进行结合。

前缀操作符的从右到左结合规则意味着*++pt 的含义如下：现将++应用于 pt（因为++位于*的右边），然后将*应用于被递增后的 pt：

```
*++pt;      // increment pointer, take the value; i.e., arr[2], or 23.4
```

另一方面，++*pt 意味着先取得 pt 指向的值，然后将这个值加 1：

```
++*pt;      // increment the pointed to value; i.e., change 23.4 to 24.4
```

在这种情况下，pt 仍然指向 arr[2]。

接下来请看下面的组合：

```
(*pt)++;    // increment pointed-to value
```

圆括号指出，首先对指针解除引用，得到 24.4。然后，操作符++将这个值递增到 25.4，pt 仍然指向 arr[2]。

最后，来看看下面的组合：

```
*pt++;      // dereference original location, then increment pointer
```

后缀操作符++的优先级更高，这意味着将操作符用于 pt，而不是*pt，因此对指针递增。然而后缀操作符意味着将对原来的地址（&arr[2]）而不是递增后的新地址解除引用，因此*pt++的值为 arr[2]，即 25.4，但该语句执行完毕后，pt 的值将为 arr[3]的地址。

记住：指针递增和递减遵循指针算术规则。因此，如果 pt 指向某个数组的第一个元素，++pt 将修改 pt，使之指向第二个元素。

5.1.9　组合赋值操作符

程序清单 5.5 使用了下面的表达式来更新循环计数：

```
i = i + by
```

C++有一种合并了加法和赋值操作的操作符，能够更简洁地完成这种任务：

```
i += by
```

+=操作符将两个操作数相加，并将结果赋给左边的操作数。这意味着左边的操作数必须能够被赋值，如变量、数组元素、结构成员或通过对指针解除引用来标识的数据：

```
int k = 5;
k += 3;                    // ok, k set to 8
int *pa = new int[10];     // pa points to pa[0]
pa[4] = 12;
pa[4] += 6;                // ok, pa[4] set to 18
*(pa + 4) += 7;            // ok, pa[4] set to 25
pa += 2;                   // ok, pa points to the former pa[2]
34 += 10;                  // quite wrong
```

每个算术操作符都有其对应的组合赋值操作符，表 5.1 对它们进行了总结。其中每个操作符的工作方式都和+=相似。因此，下面的语句：

```
k *= 10;
```

将 k 与 10 相乘，然后将结果赋给 k。

表 5.1　组合赋值操作符

操　作　符	作用（L 为左操作数，R 为右操作数）	操　作　符	作用（L 为左操作数，R 为右操作数）
+=	将 L+R 赋给 L	/=	将 L/R 赋给 L
-=	将 L-R 赋给 L	%=	将 L%R 赋给 L
*=	将 L*R 赋给 L		

5.1.10 复合语句（语句块）

编写C++for语句的格式（或句法）看上去可能比较严格，因为循环体必须是一条语句。如果要在循环体中包含多条语句，这将很不方便。所幸的是，C++提供了避开这种限制的方式，通过这种方式可以在循环体中包含任意多条语句。方法是用两个花括号来构造一条复合语句（代码块）。代码块由一对花括号和它们包含的语句组成，被视为一条语句，从而满足句法的要求。例如，程序清单5.8中的程序使用花括号将3条语句合并为一个代码块。这样，循环体便能够提示用户、读取输入并进行计算。该程序计算用户输入的数字的和，因此有机会使用+=操作符。

程序清单5.8 block.cpp

```
// block.cpp -- use a block statement
#include <iostream>
int main()
{
    using namespace std;
    cout << "The Amazing Accounto will sum and average ";
    cout << "five numbers for you.\n";
    cout << "Please enter five values: \n";
    double number;
    double sum = 0.0;
    for (int i = 1; i <= 5; i++)
    {                              // block starts here
        cout << "Value " << i << ": ";
        cin >> number;
        sum += number;
    }                              // block ends here
    cout << "Five exquisite choices indeed! ";
    cout << "They sum to " << sum << endl;
    cout << "and average to " << sum / 5 << ".\n";
    cout << "The Amazing Accounto bids you adieu!\n";
    return 0;
}
```

下面是该程序的运行情况：

```
The Amazing Accounto will sum and average five numbers for you.
Please enter five values:
Value 1: 1942
Value 2: 1948
Value 3: 1957
Value 4: 1974
Value 5: 1980
Five exquisite choices indeed! They sum to 9801
and average to 1960.2.
The Amazing Accounto bids you adieu!
```

假设对循环体进行了缩进处理，但省略了花括号：

```
for (int i = 1; i <= 5; i++)
     cout << "Value " << i << ": ";        // loop ends here
     cin >> number;                        // after the loop
     sum += number;
cout << "Five exquisite choices indeed! ";
```

编译器将忽略缩进，因此只有第一条语句位于循环中。因此，该循环将只打印出5条提示，而不执行其他操作。循环结束后，程序移到后面几行执行，只读取和计算一个数字。

复合语句还有一种有趣的特性。如果在语句块中定义一个新的变量，则仅当程序执行该语句块中的语句时，该变量才存在。执行完该语句块后，变量将被释放。这表明此变量仅在该语句块中才是可用的：

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

```
int main()
{
    int x = 20;
    {                           // block starts
        int y = 100;
        cout << x << endl;      // ok
        cout << y << endl;      // ok
    }                           // block ends
    cout << x << endl;          // ok
    cout << y << endl;          // invalid, won't compile
    return 0;
}
```

注意，在外部语句块中定义的变量在内部语句块中也是被定义了的。

如果在一个语句块中声明一个变量，而外部语句块中也有一个这种名称的变量，情况将如何呢？在声明位置到内部语句块结束的范围之内，新变量将隐藏旧变量；然后旧变量再次可见，如下例所示：

```
int main()
{
   int x = 20;              // original x
   {                        // block starts
      cout << x << endl;    // use original x
      int x = 100;          // new x
      cout << x << endl;    // use new x
   }                        // block ends
   cout << x << endl;       // use original x
   return 0;
}
```

5.1.11　逗号操作符（或其他句法技巧）

正如读者看到的，语句块允许把两条或更多条语句放到按 C++句法只能放一条语句的地方。逗号操作符对表达式完成同样的任务，允许将两个表达式放到 C++句法只允许放一个表达式的地方。例如，假设有一个循环，每轮都将一个变量加 1，而将另一个变量减 1。在 for 循环控制部分的更新部分中完成这两项工作将非常方便，但循环句法只允许这里包含一个表达式。在这种情况下，可以使用逗号操作符将两个表达式合并为一个：

```
++j, --i   // two expressions count as one for syntax purposes
```

逗号并不总是逗号操作符。例如，下面这个声明中的逗号将变量列表中相邻的名称分开：

```
int i, j;  // comma is a separator here, not an operator
```

程序清单 5.9 在一个程序中使用了两次逗号操作符，该程序将一个 string 类对象的内容反转。也可以使用 char 数组来编写该程序，但可输入的单词长度将受 char 数组大小的限制。注意，程序清单 5.6 按相反的顺序显示数组的内容，而程序清单 5.9 将数组中的字符顺序反转。该程序还使用了语句块将几条语句组合成一条。

程序清单 5.9　forstr2.cpp

```
// forstr2.cpp -- reversing an array
#include <iostream>
#include <cstring>
int main()
{
   using namespace std;
   cout << "Enter a word: ";
   string word;
   cin >> word;

   // physically modify string object
   char temp;
```

```
    int i, j;
    for (j = 0, i = word.size() - 1; j < i; --i, ++j)
    {                       // start block
        temp = word[i];
        word[i] = word[j];
        word[j] = temp;
    }                       // end block
    cout << word << "\nDone\n";
    return 0;
}
```

下面是该程序运行情况：

```
Enter a word: parts
strap
Done
```

顺便说一句，在反转字符串方面，string 类提供了更为简洁的方式，这将在第 16 章介绍。

1. 程序说明

来看程序清单 5.9 中的 for 循环控制部分。

首先，它使用逗号操作符将两个初始化操作放进控制部分第一部分的表达式中。然后，再次使用逗号操作符将两个更新合并到控制部分最后一部分的表达式中。

接下来看循环体。程序用括号将几条语句合并为一个整体。在循环体中，程序将数组第一个元素和最后一个元素调换，从而将单词反转过来。然后，它将 j 加 1，将 i 减 1，让它们分别指向第二个元素和倒数第二个元素，然后将这两个元素调换。注意，测试条件 j<i 使得到达数组的中间时，循环将停止。如果过了这一点后，循环仍继续下去，于是交换后的元素开始回到原来的位置（参见图 5.2）。

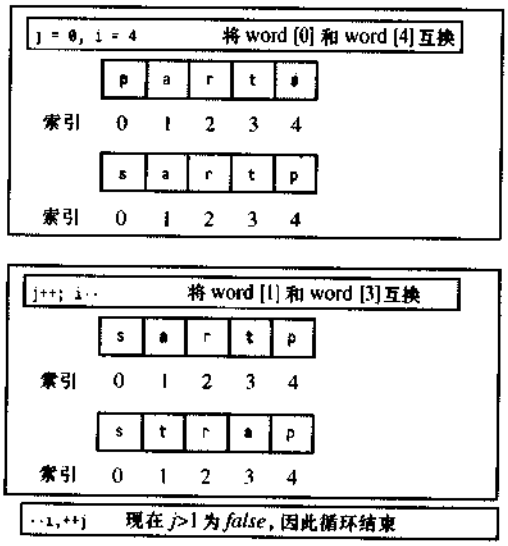


图 5.2 反转字符串

需要注意的另一点是，声明变量 temp、i 和 j 的位置。代码在循环之前声明 i 和 j，因为不能用逗号操作符将两个声明组合起来。这是因为声明已经将逗号用于其他用途——分隔列表中的变量。也可以使用一个声明语句表达式来创建并初始化两个变量，但是这样看起来有些乱：

```
int j = 0, i = word.size() - 1;
```

在这种情况下，逗号只是一个列表分隔符，而不是逗号操作符，因此该表达式对 j 和 i 进行声明和初始化。不过，看上去好像只声明了 j。

另外，可以在 for 循环内部声明 temp：

```
int temp = word[i];
```

这样，temp 在每轮循环中都将被分配和释放。这比在循环前声明 temp 的速度要慢一些。另一方面，如果在循环内部声明 temp，则它将在循环结束后被丢弃。

2. 逗号操作符花絮

到目前为止，逗号操作符最常见的用途是将两个或更多的表达式放到一个 for 循环表达式中。不过 C++还为这个操作符提供了另外两个特性。首先，它确保先计算第一个表达式，然后计算第二个表达式（换句话说，逗号操作符是一个顺序点）。如下所示的表达式是安全的：

```
i = 20, j = 2 * i      // i set to 20, j set to 40
```

其次，C++规定，逗号表达式的值是第二部分的值。例如，上述表达式的值为 40，因为 j = 2 * i 的值为 40。

在所有操作符中，逗号操作符的优先级是最低的。例如，下面的语句：

```
cata = 17, 240;
```

被解释为：

```
(cats = 17), 240;
```

也就是说，将 cats 设置为 17，240 不起作用。不过，由于括号的优先级最高，下面的表达式：

```
cats = (17, 240);
```

将把 cats 设置为 240——逗号右侧的表达式值。

5.2 关系表达式

计算机不只是机械的数字计数器。它能够对值进行比较，这种能力是计算机决策的基础。在 C++中，关系操作符是这种能力的体现。C++提供了 6 种关系操作符来对数字进行比较。由于字符用其 ASCII 码表示，因此也可以将这些操作符用于字符。不能将它们用于 C-风格字符串，但可用于 string 类对象。对于所有的关系表达式，如果比较结果为真，则其值将为 true，否则为 false，因此可将其用作循环测试表达式（老式实现认为结果为 true 的关系表达式的值为 1，而结果为 false 的关系表达式为 0）。表 5.2 对这些操作符进行了总结。

表 5.2　关系操作符

操 作 符	含　义	操 作 符	含　义
<	小于	>	大于
<=	小于或等于	>=	大于或等于
==	等于	!=	不等于

这 6 种关系操作符可以在 C++中完成对数字的所有比较。如果要对两个值进行比较，看看哪个值更漂亮或者更幸运，则这里的操作符就派不上用场了。

下面是一些测试范例：

```
for (x = 20; x > 5; x--)              // continue while x is greater than 5
for (x = 1; y != x; ++x)              // continue while y is not equal to x
for (cin >> x; x == 0; cin >> x) )    // continue while x is 0
```

关系操作符的优先级比算术操作符低。这意味着表达式：

```
x + 3 > y - 2            // expression 1
```

对应于：

```
(x + 3) > (y - 2)        // expression 2
```

而不是：

```
x + (3 > y) - 2          // expression 3
```

由于将 bool 值提升为 int 后，表达式（3>y）要么为 1，要么为 0，因此第二个和第三个表达式都是有效的。不过我们更希望第一个表达式等价于第二个表达式，而 C++正是这样做的。

5.2.1 可能犯的错误

不要混淆等于操作符（==）与赋值操作符（=）。表达式：

```
musicians == 4    // comparison
```

问了一个音乐问题——musicians 是否等于 4？表达式的值为 true 或 false。表达式：

```
musicians = 4     // assignment
```

却将 4 赋给 musicians。在这种情况下，整个表达式的值为 4，因为该表达式左边的值为 4。

for 循环的灵活设计让用户很容易出错。如果不小心遗漏了==操作符中的一个等号，则 for 循环的测试部分将是一个赋值表达式，而不是关系表达式，此时代码仍是有效的。这是因为可以将任何有效的 C++表达式用作 for 循环的测试条件。别忘了，非零值为 true，零值为 false。将 4 赋给 musicians 的表达式的值为 4，因此被视为 true。如果以前使用过用=判断是否相等的语言，如 Pascal 或 BASIC，则尤其可能出现这样的错误。

程序清单 5.10 中指出了可能出现这种错误的情况。该程序试图检查一个存储了测验成绩的数组，在遇到第一个不为 20 的成绩时停止。该程序首先演示了一个正确进行比较的循环，然后是一个在测试条件中错误地使用了赋值操作符的循环。该程序还有另一个重大的设计错误，稍后将介绍如何修复（应从错误中吸取教训，而程序清单 5.10 在这方面很有帮助）。

程序清单 5.10 equal.cpp

```
// equal.cpp -- equality vs assignment
#include <iostream>
int main()
{
    using namespace std;
    int quizscores[10] =
        { 20, 20, 20, 20, 20, 19, 20, 18, 20, 20};

    cout << "Doing it right: \n";
    int i;
    for (i = 0; quizscores[i] == 20; i++)
        cout << "quiz " << i << " is a 20\n";

    cout << "Doing it dangerously wrong: \n";
    for (i = 0; quizscores[i] = 20; i++)
        cout << "quiz " << i << " is a 20\n";

    return 0;
}
```

由于这个程序存在一个严重的问题，读者可能希望了解它，以便真正运行它。下面是该程序的一些输出：

```
Doing it right:
quiz 0 is a 20
quiz 1 is a 20
quiz 2 is a 20
quiz 3 is a 20
quiz 4 is a 20
Doing it dangerously wrong:
quiz 0 is a 20
quiz 1 is a 20
quiz 2 is a 20
quiz 3 is a 20
quiz 4 is a 20
```

```
quiz 5 is a 20
quiz 6 is a 20
quiz 7 is a 20
quiz 8 is a 20
quiz 9 is a 20
quiz 10 is a 20
quiz 11 is a 20
quiz 12 is a 20
quiz 13 is a 20
...
```

第一个循环在显示了前 5 个测验成绩后正确地终止，但第二个循环显示整个数组。更糟糕的是，显示的每个值都是 20。更加糟糕的是，它到了数组末尾还不停止。

当然，错误出在下面的测试表达式中：

```
quizscores[i] = 20
```

首先，由于它将一个非零值赋给数组元素，因此表达式始终为非零，所以始终为 true。其次，由于表达式将值赋给数组元素，它实际上修改了数据。第三，由于测试表达式一直为 true，因此程序在到达数组结尾后，仍不断修改数据。它把一个又一个 20 放入内存中！这会带来不好的影响。

发现这种错误的困难之处在于，代码在语法上是正确的，因此编译器不会将其视为错误（不过，由于 C 和 C++程序员频繁地犯这种错误，因此很多编译器都会发出警告，询问这是否是设计者的真正意图）。

警告：不要使用=来比较两个量是否相等，而要使用==。

和 C 语言一样，C++比起大多数编程语言来说，赋予程序员更大的自由。这种自由以程序员应付的更大责任为代价。只有良好的规划才能避免程序超出标准 C++数组的边界。不过，对于 C++类，可以设计一种保护数组类型来防止这种错误，第 13 章提供一个这样的例子。另外，应在需要的时候在程序中加入保护措施。例如，在程序清单 5.10 的循环中，应包括防止超出最后一个成员的测试，这甚至对于“好”的循环来说也是必要的。如果所有的成绩都是 20，“好”的循环也会超出数组边界。总之，循环需要测试数组的值和索引的值。第 6 章将介绍如何使用逻辑操作符将两个这样的测试合并为一个条件。

5.2.2　C-风格字符串的比较

假设要知道字符数组中的字符串是不是 mate。如果 word 是数组名，下面的测试可能并不能像我们预想的那样工作：

```
word == "mate"
```

请记住，数组名是数组的地址。同样，用引号括起的字符串常量也是其地址。因此，上面的关系表达式不是判断两个字符串是否相同，而是查看它们是否存储在相同的地址上。两个字符串的地址是否相同呢？答案是否定的，虽然它们包含相同的字符。

由于 C++将 C-风格字符串视为地址，因此如果使用关系操作符来比较它们，将无法得到满意的结果。相反，应使用 C-风格字符串库中的 strcmp()函数来比较。该函数接受两个字符串地址作为参数。这意味着参数可以是指针、字符串常量或字符数组名。如果两个字符串相同，该函数将返回零；如果第一个字符串按字母顺序排在第二个字符串之前，则 strcmp()将返回一个负数值；如果第一个字符串按字母顺序排在第二个字符串之后，则 strcpm()将返回一个正数值。实际上，“按系统排列顺序”比“按字母顺序”更准确。这意味着字符是根据字符的系统编码来进行比较的。例如，使用 ASCII 码时，所有大写字母的编码都比小写字母小，所以按排列顺序，大写字母将位于小写字母之前。因此，字符串“Zoo”在字符串“aviary”之前。根据编码进行比较还意味着大写字母和小写字母是不同的，因此字符串“FOO”和字符串“foo”不同。

在一些语言中，如 BASIC 和标准 Pascal，存储在不同长度的数组中的字符串彼此不相等。但是 C-风格字符串是通过结尾的空值字符定义的，而不是由其所在数组的长度定义的。这意味着两个字符串即使被存储在长度不同的数组中，也可能是相同的：

```
char big[80] = "Daffy";         // 5 letters plus \0
char little[6] = "Daffy";       // 5 letters plus \0
```

另外，虽然不能用关系操作符来比较字符串，但却可以用它们来比较字符，因为字符实际上是整型。

因此下面的代码可以用来显示字母表中的字符，至少对于 ASCII 字符集来说是有效的：

```
for (ch = 'a'; ch <='z'; ch++)
    cout << ch;
```

程序清单 5.11 在 for 循环的测试条件中使用了 strcmp()。该程序显示一个单词，修改其首字母，然后再次显示这个单词，这样循环往复，直到 strcmp()确定该单词与字符串“mate”相同为止。注意，该程序清单包含了文件 cstring，因为它提供了 strcmp()的函数原型。

程序清单 5.11 comstr1.cpp

```
// compstr.cpp -- comparing strings
#include <iostream>
#include <cstring>    // prototype for strcmp()
int main()
{
   using namespace std;
   char word[5] = "?ate";

   for (char ch = 'a'; strcmp(word, "mate"); ch++)
   {
      cout << word << endl;
      word[0] = ch;
   }
   cout << "After loop ends, word is " << word << endl;
   return 0;
}
```

注意：可能必须使用 string.h，而不是 cstring。另外，程序清单 5.11 中的代码假设系统使用的是 ASCII 字符编码集。在这种编码集中，从 a 到 z 的字母的编码是连续的，？字符的编码比 a 的编码小 1。

下面是该程序的输出：

```
?ate
aate
bate
cate
date
eate
fate
gate
hate
iate
jate
kate
late
After loop ends, word is mate
```

程序说明

该程序有几个有趣的地方。其中之一当然是测试。我们希望只要 word 不是 mate，循环就继续进行。也就是说，我们希望只要 strcmp()判断出两个字符串不相同，测试就继续进行。最显而易见的测试是这样的：

```
strcmp(word, "mate") != 0    // strings are not the same
```

如果字符串不相等，则该语句的值为 1（true），如果字符串相等，则该语句的值为 0（false）。但使用 strcmp（word，"mate"）本身将如何呢？如果字符串不相等，则它的值为非零（true）；如果字符串相等，则它的值为零（false）。实际上，如果字符串不同，该返回 true，否则返回 false。因此，可以只用这个函数，而不是整个关系表达式。这样得到的结果将相同，还可以少输入几个字符。另外，C 和 C++程序员传统上就是用这种方式使用 strcmp()的。

记住：可以使用 strcmp()来测试 C-风格字符串是否相等（排列顺序）。如果 str1 和 str2 相等，则下面

的表达式为 true：

```
strcmp(str1, str2) == 0
```

如果 str1 和 str2 不相等，则下面两个表达式都为 true：

```
strcmp(str1, str2) != 0
strcmp(str1, str2)
```

如果 str1 在 str2 的前面，则下面的表达式为 true：

```
strcmp(str1, str2) < 0
```

如果 str1 在 str2 的后面，则下面的表达式为 true：

```
strcmp(str1, str2) > 0
```

因此，根据要如何设置测试条件，strcmp() 可以扮演==、!=、<和>操作符的角色。

接下来，compstr1.cpp 使用递增操作符使变量 ch 遍历字母表：

```
ch++
```

可以对字符变量使用递增操作符和递减操作符，因为 char 类型实际上是整型，因此这种操作实际上将修改存储在变量中的整数编码。另外，使用数组索引可使修改字符串中的字符更为简单：

```
word[0] = ch;
```

5.2.3　比较 string 类字符串

如果使用 string 类字符串而不是 C-风格字符串，比较起来将简单些，因为类设计让您能够使用关系操作符进行比较。这之所以可行，是因为类函数重载（修订）了这些操作符。第 12 章将介绍如何将这种特性加入到类设计中，但从应用的角度说，读者现在只需知道可以将关系操作符用于 string 对象即可。程序清单 5.12 是在程序清单 5.11 的基础上修改而成的，它使用的是 string 对象而不是 char 数组。

程序清单 5.12　comstr2.cpp

```
// compstr2.cpp -- comparing strings using arrays
#include <iostream>
#include <string>    // string class
int main()
{
    using namespace std;
    string word = "?ate";

    for (char ch = 'a'; word !=
    {
        cout << word << endl;
        word[0] = ch;
    }
    cout << "After loop ends, word is " << word << endl;
    return 0;
}
```

该程序的输出与程序清单 5.11 相同。

程序说明

在程序清单 5.12 中，下面的测试条件：

```
word != "mate"
```

使用了一个关系操作符，该操作符的左边是一个 string 对象，右边是一个 C-风格字符串。string 类重载操作符!=的方式让您能够在下述条件下使用它：至少有一个操作数为 string 对象，另一个操作数可以是 string 对象，也可以是 C-风格字符串。

string 类的设计让您能够将 string 对象作为一个实体（在关系型测试表达式中），也可以将其作为一个聚合对象，从而使用数组表示法来提取其中的字符。

最后，和前面大多数 for 循环不同，此循环不是计数循环。也就是说，它并不对语句块执行指定的次数。相反，此循环将根据情况（word 为“mate”）来确定是否停止。对于这种测试，C++程序通常使用 while

循环，下面来看看这种循环。

5.3 while 循环

while 循环是没有初始化和更新部分的 for 循环，它只有测试条件和循环体：

```
while (test-condition)
    body
```

首先，程序计算圆括号内的测试条件（test-condition）表达式。如果该表达式为 true，则执行循环体中的语句。与 for 循环一样，循环体也由一条语句或两个花括号定义的语句块组成。执行完循环体后，程序返回测试条件，对它进行重新计算。如果该条件为非零，则再次执行循环体。测试和执行将一直进行下去，直到测试条件为 false 为止（参见图 5.3）。显然，如果希望循环最终能够结束，循环体中的代码必须完成某种影响测试条件表达式的操作。例如，循环可以将测试条件中使用的变量加 1 或从键盘输入读取一个新值。和 for 循环一样，while 循环也是一种入口条件循环。因此，如果测试条件一开始便为 false，则程序将不会执行循环体。

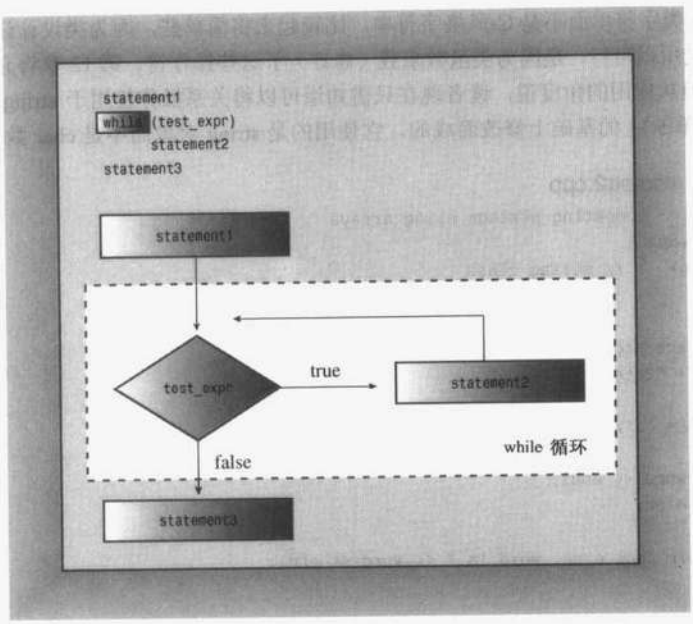


图 5.3 while 循环的结构

程序清单 5.13 使用了一个 while 循环。该循环遍历字符串，并显示其中的字符及其 ASCII 码。循环在遇到空值字符时停止。这种逐字符遍历字符串直到遇到空值字符的技术是 C++处理字符串的标准方法。由于字符串中包含了结尾标记，因此程序通常不需要知道字符串的长度。

程序清单 5.13 while.cpp

```
// while.cpp -- introducing the while loop
#include <iostream>
const int ArSize = 20;
int main()
{
    using namespace std;
    char name[ArSize];
```

```
    cout << "Your first name, please: ";
    cin >> name;
    cout << "Here is your name, verticalized and ASCIIized: \n";
    int i = 0;                    // start at beginning of string
    while (name[i] != '\0')       // process to end of string
    {
        cout << name[i] << ": " << int (name[i]) << endl;
        i++;                      // don't forget this step
    }
    return 0;
}
```

下面是该程序的运行情况：

```
Your first name, please: Muffy
Here is your name, verticalized and ASCIIized:
M: 77
u: 117
f: 102
f: 102
y: 121
```

verticalized 和 ASCIIized 并不是真正的单词，甚至将来也不会是单词。不过它们确实在输出中添加了一种“可爱”的氛围。

程序说明

程序清单 5.13 中的 while 条件像这样：

```
while (name[i] != '\0')
```

它可以测试数组中特定的字符是不是空值字符。为使该测试最终能够成功，循环体必须修改 i 的值，这是通过在循环体结尾将 i 加 1 来实现的。省略这一步将导致循环停留在同一个数组元素上，打印该字符及其编码，直到强行终止该程序。导致死循环是循环最常见的问题之一。通常，在循环体中忘记更新某个值时，便会出现这种情况。

可以这样修改 while 行：

```
while (name[i])
```

经过这种修改后，程序的工作方式将不变。这是由于 name[i]是常规字符，其值为该字符的编码——非零值或 true。不过，当 name[i]为空值字符时，其编码将为 0 或 false。这种表示法更为简洁（也更常用），但没有程序清单 5.13 中的表示法清晰。对于后一种情况，“笨拙”的编译器生成的代码的速度将更快，“聪明”的编译器对于这两个版本生成的代码将相同。

要打印字符的 ASCII 码，必须通过强制类型转换将 name[i]转换为整型。这样，cout 将把值打印成整数，而不是将它解释为字符编码。

不同于 C-风格字符串，string 类对象不使用空字符来标记字符串末尾，因此要将程序清单 5.13 转换为使用 string 类的版本，只需用 string 对象替换 char 数组即可。第 16 章将讨论可用于标识 string 对象中最后一个字符的技术。

5.3.1　for 与 while

在 C++中，for 和 while 循环本质上是相同的。例如，下面的 for 循环：

```
for (init-expression; test-expression; update-expression)
{
    statement (s)
}
```

可以改写成这样：

```
init-expression;
while (test-expression)
{
    statement (s)
```

```
    update-expression;
}
```

同样，下面的 while 循环：

```
while (test-expression)
    body
```

也可以改写成这样：

```
for (; test-expression;)
    body
```

for 循环需要 3 个表达式（从技术的角度说，它需要 1 条后面跟两个表达式的语句），不过它们可以是空表达式（语句），只有两个分号是必需的。另外，省略 for 循环中的测试表达式时，测试结果将为 true，因此下面的循环将一直运行下去：

```
for (;;)
    body
```

由于 for 循环和 while 循环几乎是等效的，因此究竟使用哪一个只是风格上的问题（如果循环体中包括 continue 语句，情况将稍有不同，continue 语句将在第 6 章讨论）。通常，程序员使用 for 循环来为循环计数，因为 for 循环格式允许将所有相关的信息——初始值、终止值和更新计数器的方法——放在同一个地方。在无法预先知道循环将执行的次数时，程序员常使用 while 循环。

提示：在设计循环时，请记住下面几条指导原则：

- 确定循环终止的条件
- 在首次测试之前初始化条件
- 在条件被再次测试之前更新条件

for 循环的一个优点是，其结构提供了一个可实现上述 3 条指导原则的地方，因此有助于程序员记住应该这样做。

错误的标点符号

for 循环和 while 循环都由用括号括起的表达式和后面的循环体（包含一条语句）组成。前面讲过，这条语句可以是语句块，其中包含多条语句。记住，语句块是由花括号，而不是由缩进定义的。例如，请看下面的循环：

```
i = 0;
while (name[i] != '\0')
    cout << name[i] << endl;
    i++;
cout << "Done\n";
```

缩进表明，该程序的作者希望 i++; 语句是循环体的组成部分。不过，由于没有花括号，因此编译器认为循环体仅由最前面的 cout 语句组成。因此，该循环将不断地打印数组的第一个字符。该程序不会执行 i++; 语句，因为它在循环的外面。

下面的例子说明了另一个潜在的缺陷：

```
i = 0;
while (name[i] != '\0');    // problem semicolon
{
    cout << name[i] << endl;
    i++;
}
cout << "Done\n";
```

这一次，代码正确地使用了花括号，但还插入了一个分号。记住，分号结束语句，因此该分号将结束 while 循环。换句话说，循环体为空语句，也就是说，分号后面没有任何内容。这样，花括号中所有的代码现在位于循环的后面，永远不会被执行。该循环不执行任何操作，是一个死循环。请注意这种分号。

5.3.2 等待一段时间：编写延时循环

有时候，让程序等待一段时间很有用。例如，读者可能遇到过这样的程序，它在屏幕上显示一条消息，

而还没来得及阅读之前，又出现了其他内容。这样读者将担心自己错过了重要的、无法恢复的消息。如果程序在显示其他内容之前等待 5 秒钟，情况将会好得多。while 循环可用于这种目的。一种用于个人计算机的早期技术是，让计算机进行计数，以等待一段时间：

```
long wait = 0;
while (wait < 10000)
   wait++;          // counting silently
```

这种方法的问题是，当计算机处理器的速度发生变化时，必须修改计数限制。例如，有些为 IBM PC 编写的游戏在速度更快的机器上运行时，其速度将快得无法控制。更好的方法是让系统时钟来完成这种工作。

ANSI C 和 C++库中有一个函数有助于完成这样的工作。这个函数名为 clock()，返回程序开始执行后所用的系统时间。这有两个复杂的问题：首先，clock()返回时间的单位不一定是秒；其次，该函数的返回类型在某些系统上可能是 long，在另一些系统上可能是 unsigned long 或其他类型。

不过头文件 ctime（较早的实现中为 time.h）提供了这些问题的解决方案。首先，它定义了一个符号常量——CLOCKS_PER_SEC，该常量等于每秒钟包含的系统时间单位数。因此，将系统时间除以这个值，可以得到秒数。或者将秒数乘以 CLOCK_PER_SEC，可以得到以系统时间单位为单位的时间。其次，ctime 将 clock_t 作为 clock()返回类型的别名（参见本章后面的注释“类型别名”），这意味着可以将变量声明为 clock_t 类型，编译器将把它转换为 long、unsigned int 或适合系统的其他类型。

注意：对于没有 ctime 头文件的系统，可以使用 time.h。如果实现的库组件不完全与 ANSI C 兼容，则在这种实现中运行 waiting.cpp 将有问题。这是因为 clock()函数是 ANSI 在传统 C 库的基础上新增的。另外，一些不成熟的 ANSI C 实现使用 CLK_TCK 或 TCK_CLK，而不是更长的 CLOCK_PER_SEC。C++的一些老版本不能识别这些常量。有些环境无法处理振铃字符\a 和协调显示延迟。

程序清单 5.14 演示了怎样使用 clock()和头文件 ctime 来创建延迟循环。

程序清单 5.14　waiting.cpp

```
// waiting.cpp -- using clock()in a time-delay loop
#include <iostream>
#include <ctime> // describes clock()function, clock_t type
int main()
{
   using namespace std;
   cout << "Enter the delay time, in seconds: ";
   float secs;
   cin >> secs;
   clock_t delay = secs * CLOCKS_PER_SEC;       // convert to clock ticks
   cout << "starting\a\n";
   clock_t start = clock();
   while (clock()- start < delay)               // wait until time elapses
     ;                                          // note the semicolon
   cout << "done \a\n";
   return 0;
}
```

该程序以系统时间单位为单位（而不是以秒为单位）计算延迟时间，避免了在每轮循环中将系统时间转换为秒。

类型别名

C++为类型建立别名的方式有两种，一种是使用预处理器：

```
#define BYTE char // preprocessor replaces BYTE with char
```

这样，预处理器将在编译程序时用 char 替换所有的 BYTE，从而使 BYTE 成为 char 的别名。

第二种方法是使用 C++（和 C）的关键字 typedef 来创建别名。例如，要将 byte 作为 char 的别名，可以这样做：

```
typedef char byte;  // makes byte an alias for char
```

下面是通用格式：

```
typedef typeName aliasName;
```

换句话说，如果要将 aliasName 作为某种类型的别名，可以声明 aliasName，如同将 aliasName 声明为这种类型的变量那样，然后在声明的前面加上关键字 typedef。例如，要让 byte_pointer 成为 char 指针的别名，可将 byte_pointer 声明为 char 指针，然后在前面加上 typedef：

```
typedef char * byte_pointer; // pointer to char type
```

也可以使用#define，不过声明一系列变量时，这种方法不适用。例如，请看下面的代码：

```
#define FLOAT_POINTER float *
FLOAT_POINTER pa, pb;
```

预处理器置换将该声明转换为这样：

```
float * pa, pb;  // pa a pointer to float, pb just a float
```

typedef方法不会有这样的问题。它能够处理更复杂的类型别名，这使得与使用#define 相比，使用 typedef 是一种更佳的选择——有时候，这也是惟一的选择。

注意，typedef 不会创建新类型，而只是为已有的类型建立一个新名称。如果将 word 作为 int 的别名，则 cout 将把 word 类型的值视为 int 类型。

5.4 do while 循环

前面已经学习了 for 循环和 while 循环。第 3 种 C++循环是 do while，它不同于另外两种循环，因为它是出口条件（exit condition）循环。这意味着这种循环将首先执行循环体，然后再判定测试表达式，决定是否应继续执行循环。如果条件为 false，则循环终止；否则，进入新一轮的执行和测试。这样的循环通常至少执行一次，因为其程序流必须经过循环体后才能到达测试条件。下面是其句法：

```
do
    body
while (test-expression);
```

循环体是一条语句或用括号括起的语句块。图 5.4 总结了 do while 循环的程序流程。

通常，入口条件循环比出口条件循环好，因为入口条件循环在循环开始之前对条件进行检查。例如，假设程序清单 5.13 使用 do while（而不是 while），则循环将打印空值字符及其编码，然后才发现已到达字符串结尾。但是有时 do while 测试更合理。例如，请求用户输入时，程序必须先获得输入，然后对它进行测试。程序清单 5.15 演示了如何在这种情况下使用 do while。

程序清单 5.15 dowhile.cpp

```
// dowhile.cpp -- exit-condition loop
#include <iostream>
int main()
{
   using namespace std;
   int n;
   cout << "Enter numbers in the range 1-10 to find ";
   cout << "my favorite number\n";
   do
   {
      cin >> n;       // execute body
   } while (n != 7);   // then test
   cout << "Yes, 7 is my favorite.\n";
   return 0;
}
```

下面是该程序的运行情况：

```
Enter numbers in the range 1-10 to find my favorite number
9
```

```
4
7
Yes, 7 is my favorite.
```

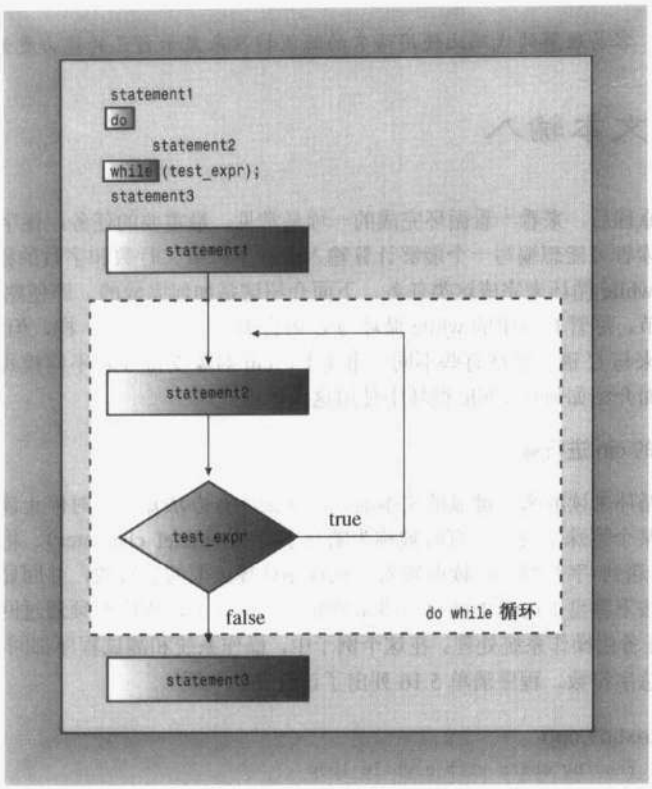


图 5.4　do while 循环的结构

奇特的 for 循环

虽然不是很常见，但有时出现下面这样的代码，：

```
for (;;) // sometimes called a "forever loop"
{
      I++;
      // do something ...
      if (30 >= I) break;   //if statement and break (Chapter 6)
}
```

或另一种变体：

```
for (;; I++)
{
      if (30 >= I) break;
      // do something ...
}
```

上述代码基于这样一个事实：for 循环中的空测试条件被视为 true。这些例子既不易于阅读，也不能用作编写循环的通用模型。第一个例子的功能在 do while 循环中将表达得更清晰：

```
do {
      I++;
      // do something;
while (30 < I);
```

同样，第二个例子使用 while 循环可以表达得更清晰：

```
while (I < 30)
{
      // do something
      I++;
}
```

通常，编写清晰、容易理解的代码比使用语言的晦涩特性来显示自己的能力更为有用

5.5 循环和文本输入

知道循环的工作原理后，来看一看循环完成的一项最常见、最重要的任务：逐字符地读取来自文件或键盘的文本。例如，读者可能想编写一个能够计算输入中的字符数、行数和字数的程序。传统上，C++和C语言一样，也使用 while 循环来完成这类任务。下面介绍这是如何完成的。即使熟悉 C 语言，也不要太快地浏览本节和下一节。尽管 C++中的 while 循环与 C 语言中的 while 循环一样，但 C++的 I/O 工具不同，这使得 C++循环看起来与 C 语言循环有些不同。事实上，cin 对象支持 3 种不同模式的单字符输入，其用户接口各不相同。下面介绍如何在 while 循环中使用这 3 种模式。

5.5.1 使用原始的 cin 进行输入

如果程序要使用循环来读取来自键盘的文本输入，则必须有办法知道何时停止读取。如何知道这一点呢？一种方法是选择某个特殊字符——有时被称为哨兵字符（sentinel character），将其作为停止标记。例如，程序清单 5.16 在遇到#字符时停止读取输入。该程序计算读取的字符数，并回显这些字符，即在屏幕上显示读取的字符。按下键盘上的键不能自动将字符显示到屏幕上，程序必须通过回显输入字符来完成这项工作。通常，这种任务由操作系统处理。在这个例子中，操作系统和测试程序都回显输入。运行完毕后，该程序将报告处理的总字符数。程序清单 5.16 列出了该程序的代码。

程序清单 5.16 textin1.cpp

```
// textin1.cpp -- reading chars with a while loop
#include <iostream>
int main()
{
   using namespace std;
   char ch;
   int count = 0;          // use basic input
   cout << "Enter characters; enter # to quit:\n";
   cin >> ch;              // get a character
   while (ch != '#')       // test the character
   {
      cout << ch;          // echo the character
      ++count;             // count the character
      cin >> ch;           // get the next character
   }
   cout << endl << count << " characters read\n";
   return 0;
}
```

下面是该程序的运行情况：

```
Enter characters; enter # to quit:
see ken run#really fast
seekenrun
9 characters read
```

程序说明

请注意该程序的结构。该程序在循环之前读取第一个输入字符，这样循环可以测试第一个字符。这很

重要，因为第一个字符可能是#。由于 textin1.cpp 使用的是入口条件循环，因此在这种情况下，能够正确地跳过整个循环。由于前面已经将变量 count 设置为 0，因此 count 的值也是正确的。

如果读取的第一个字符不是#，则程序进入该循环，显示字符，增加计数，然后读取下一个字符。最后一步是极为重要的，没有这一步，循环将反复处理第一个输入字符，一直进行下去。有了这一步后，程序就可以处理到下一个字符。

注意，该循环设计遵循了前面指出的几条指导原则。结束循环的条件是最后读取的一个字符是#。该条件是通过在循环之前读取一个字符进行初始化的，而通过循环体结尾读取下一个字符进行更新。

上面的做法合情合理。但为什么程序在输出时省略了空格呢？原因在 cin。读取 char 值时，与读取其他基本类型一样，cin 将忽略空格和换行符。因此输入中的空格没有被回显，也没有被包括在计数内。

更为复杂的是，发送给 cin 的输入被缓冲。这意味着只有在用户按下回车键后，他输入的内容才会被发送给程序。这就是在运行该程序时，可以在#后面输入字符的原因。按下回车键后，整个字符序列将被发送给程序，但程序在遇到#字符后将结束对输入的处理。

5.5.2　使用 cin.get（char）进行补救

通常，逐个字符读取输入的程序需要检查每个字符，包括空格、制表符和换行符。cin 所属的 istream 类（在 iostream 中定义）中包含一个能够满足这种要求的成员函数。具体地说，成员函数 cin.get（ch）读取输入中的下一个字符（即使它是空格），并将其赋给变量 ch。使用这个函数调用替换 cin>>ch，可以修补程序清单 5.16 的问题。程序清单 5.17 列出了修改后的代码。

程序清单 5.17　textin2.cpp

```
// textin2.cpp -- using cin.get(char)
#include <iostream>
int main()
{
    using namespace std;
    char ch;
    int count = 0;

    cout << "Enter characters; enter # to quit:\n";
    cin.get(ch);        // use the cin.get(ch) function
    while (ch != '#')
    {
        cout << ch;
        ++count;
        cin.get(ch);    // use it again
    }
    cout << endl << count << " characters read\n";
    return 0;
}
```

下面是该程序的运行情况：

```
Did you use a #2 pencil?
Did you use a
14 characters read
```

现在，该程序回显了每个字符，并将全部字符计算在内，其中包括空格。输入仍被缓冲，因此输入的字符个数仍可能比最终到达程序的要多。

如果熟悉 C 语言，可能以为这个程序存在严重的错误。cin.get（ch）调用将一个值放在 ch 变量中，这意味着将修改该变量的值。在 C 语言中，要修改变量的值，必须将变量的地址传递给函数。但程序清单 5.17 调用 cin.get()时，传递的是 ch，而不是&ch。在 C 语言中，这样的代码无效，但在 C++中有效，只要函数将参数声明为引用即可。引用是 C++中的一种新类型。头文件 iostream 将 cin.get（ch）的参数声明为引用类型，因此该函数可以修改其参数的值。我们将在第 8 章中详细介绍。同时，C 语言行家可以松一口气了——通常，在 C++中传递的参数的工作方式与在 C 语言中相同。不过，cin.get（ch）不是这样。

5.5.3 使用哪一个 cin.get()

在第 4 章的程序清单 4.5 中，使用了这样的代码：

```
char name[ArSize];
...
cout << "Enter your name: \n";
cin.get (name, ArSize) .get();
```

最后一行相当于两个连续的函数调用：

```
cin.get (name, ArSize);
cin.get();
```

cin.get()的一个版本接受两个参数：数组名（字符串（char*类型）的地址）和 ArSize（int 类型的整数）（记住，数组名是其第一个元素的地址，因此字符数组名的类型为 char*）。接下来，程序使用了不接受任何参数的 cin.get()。而最近，我们这样使用过 cin.get()：

```
char ch;
cin.get (ch);
```

这里 cin.get 接受一个 char 参数。

看到这里，熟悉 C 语言的读者将再次感到兴奋或困惑。在 C 语言中，如果函数接受 char 指针和 int 参数，则使用该函数时，不能只传递一个参数（类型不同）。但在 C++中，可以这样做，因为该语言支持被称为函数重载的 OOP 特性。函数重载允许创建多个同名函数，条件是它们的参数列表不同。例如，如果在 C++中使用 cin.get（name，ArSize），则编译器将找到使用 char*和 int 作为参数的 cin.get()版本；如果使用 cin.get（ch），则编译器将使用接受一个 char 参数的版本；如果没有提供参数，则编译器将使用不接受任何参数的 cin.get()版本。函数重载允许对多个相关的函数使用相同的名称，这些函数以不同方式或针对不同类型执行相同的基本任务。第 8 章将讨论该主题。另外，通过使用 istream 类中的 get()范例，读者将逐渐习惯函数重载。为区分不同的函数版本，我们在引用它们时提供参数列表。因此，cin.get()指的是不接受任何参数的版本，而 cin.get（char）则指的是接受一个参数的版本。

5.5.4 文件尾条件

程序清单 5.17 表明，使用诸如#等符号来表示输入结束很难令人满意，因为这样的符号可能就是合法输入的组成部分，其他符号（如@和%）也如此。如果输入来自于文件，则可以使用一种功能更强大的技术——检测文件尾（EOF）。C++输入工具和操作系统协同工作，来检测文件尾并将这种信息告知程序。

乍一看，读取文件中的信息似乎同 cin 和键盘输入没什么关系，但其实存在两个相关的地方。首先，很多操作系统（包括 UNIX 和 MS-DOS）都支持重定向，允许用文件替换键盘输入。例如，假设在 MS-DOS 中有一个名为 gofish.exe 的可执行程序和一个名为 fishtale 的文本文件，则可以在 DOS 提示符下输入下面的命令：

```
gofish <fishtale
```

这样，程序将从 fishtale 文件（而不是键盘）获取输入。<符号是 UNIX 和 DOS 上的重定向操作符。

其次，很多操作系统都允许通过键盘来模拟文件尾条件。在 UNIX 中，可以在行首按下 Ctrl+D 来实现；在 DOS 中，可以在任意位置按下 Ctrl+Z、Enter。有些 C++实现支持类似的行为，即使底层操作系统并不支持。键盘输入的 EOF 概念实际上是命令行环境遗留下来的。不过，用于 Mac 的 Symantec C++模拟了 UNIX，将 Ctrl+D 视为仿真的 EOF。Metrowerks Codewarrior 能够在 Macintosh 和 Windows 环境下识别 Ctrl+Z。用于 PC 的 Microsoft Visual C++ 7.0、Borland C++ 5.5 和 GNU C++都能够识别行首的 Ctrl + Z，但用户必须随后按下回车键。总之，很多 PC 编程环境都将 Ctrl+Z 视为模拟的 EOF，但具体细节（必须在行首还是可以在任何位置，是否必须按下回车键等）各不相同。

如果编程环境能够检测 EOF，可以在类似于程序清单 5.17 的程序中使用重定向的文件，也可以使用键盘输入，并在键盘输入中模拟 EOF。这一点似乎很有用，因此我们来看看究竟如何做。

检测到 EOF 后，cin 将两位（eofbit 和 failbit）都设置为 1。可以通过成员函数 eof()来查看 eofbit 是否被设置：如果检测到 EOF，则 cin.eof()将返回 bool 值 true，否则返回 false。同样，如果 eofbit 或 failbit 被

设置为 1，则 fail() 成员函数返回 true，否则返回 false。注意，eof() 和 fail() 方法报告最近读取的结果；也就是说，它们在事后报告，而不是预先报告。因此应将 cin.eof() 或 cin.fail() 测试放在读取后，程序清单 5.18 中的设计体现了这一点。它使用的是 fail()，而不是 eof()，因为前者可用于更多的实现中。

注意：有些系统不支持来自键盘的模拟 EOF；有些系统对其支持不完善。cin.get() 可以用来锁住屏幕，直到可以读取为止，但是这种方法在这里并不适用，因为检测 EOF 时将关闭对输入的进一步读取。不过，可以使用程序清单 5.14 中那样的计时循环来使屏幕在一段时间内是可见的。

程序清单 5.18　textin3.cpp

```
// textin3.cpp -- reading chars to end of file
#include <iostream>
int main()
{
    using namespace std;
    char ch;
    int count = 0;
    cin.get (ch);                  // attempt to read a char
    while (cin.fail()== false)  // test for EOF
    {
        cout << ch;                // echo character
        ++count;
        cin.get (ch);              // attempt to read another char
    }
    cout << endl << count << " characters read\n";
    return 0;
}
```

下面是该程序的运行情况：

```
The green bird sings in the winter.<ENTER>
The green bird sings in the winter.
Yes, but the crow flies in the dawn.<ENTER>
Yes, but the crow flies in the dawn.
<CTRL><Z>
73 characters read
```

这里在 Windows XP 系统上运行该程序，因此可以按下 Ctrl+Z 和回车键来模拟 EOF 条件。UNIX 和 Linux 用户应按 Ctrl+D 来模拟。

通过使用重定向，可以用该程序来显示文本文件，并报告它包含的字符数。下面，我们在 UNIX 系统运行该程序，并对一个两行的文件进行读取、回显和计算字数（$是 UNIX 提示符）：

```
$ textin3 < stuff
I am a UNIX file. I am proud
to be a UNIX file.
49 characters read
$
```

1. EOF 结束输入

前面指出过，cin 方法检测到 EOF 时，将设置 cin 对象中一个指示 EOF 条件的标记。设置这个标记后，cin 将不读取输入，再次调用 cin 也不管用。对于文件输入，这是有道理的，因为程序不应读取超出文件尾的内容。不过，对于键盘输入，有可能使用模拟 EOF 来结束循环，但稍后要读取其他输入。cin.clear() 方法可能清除 EOF 标记，使输入继续进行。这将在第 17 章详细介绍。不过要记住的是，在有些系统中，按 Ctrl+Z 实际上将结束输入和输出，而 cin.clear() 将无法恢复输入和输出。

2. 常见的字符输入做法

每次读取一个字符，直到遇到 EOF 的输入循环的基本设计如下：

```
cin.get (ch);                  // attempt to read a char
while (cin.fail()== false)     // test for EOF
```

```
{
   ...                          // do stuff
   cin.get(ch);                 // attempt to read another char
}
```

可以在上述代码中使用一些简捷方式。第 6 章将介绍的!操作符可以将 true 切换为 false 或将 false 切换为 true。可以使用此操作符将上述 while 测试改写成这样：

```
while (!cin.fail())   // while input has not failed
```

cin.get（char）方法的返回值是一个 cin 对象。不过，istream 类提供了一个可以将 istream 对象（如 cin）转换为 bool 值的函数：当 cin 出现在需要 bool 值的地方（如在 while 循环的测试条件中）时，该转换函数将被调用。另外，如果最后一次读取成功了，则转换得到的 bool 值为 true；否则为 false。这意味着可以将上述 while 测试改写为这样：

```
while (cin)   // while input is successful
```

这比 !cin.fail()或!cin.eof()更通用，因为它可以检测到其他失败原因，如磁盘故障。

最后，由于 cin.get（char）的返回值为 cin，因此可以将循环精简成这种格式：

```
while (cin.get(ch))   // while input is successful
{
   ...                // do stuff
}
```

这样，cin.get(char)只被调用一次，而不是两次：循环前一次、循环结束后一次。为判断循环测试条件，程序必须首先调用 cin.get（ch）。如果成功，则将值放入 ch 中。然后，程序获得函数调用的返回值，即 cin。接下来，程序对 cin 进行 bool 转换，如果输入成功，则结果为 true，否则为 false。3 条指导原则（确定结束条件、对条件进行初始化以及更新条件）全部被放在循环测试条件中。

5.5.5 另一个 cin.get()版本

“怀旧”的 C 语言用户可能喜欢 C 语言中的字符 I/O 函数——getchar()和 putchar()，它们仍然适用，只要像在 C 语言中那样包含头文件 stdio.h（或新的 cstdio）即可。也可以使用 istream 和 ostream 类中类似功能的成员函数，来看看这种方式。

注意：有些老式 C++实现不支持这里讨论的不接受任何参数的 cin.get()成员函数。

不接受任何参数的 cin.get()成员函数返回输入中的下一个字符。也就是说，可以这样使用它：

```
ch = cin.get();
```

该函数的工作方式与 C 语言中的 getchar()相似，将字符编码作为 int 值返回；而 cin.get（ch）返回一个对象，而不是读取的字符。同样，可以使用 cout.put()函数（参见第 3 章）来显示字符：

```
cout.put(ch);
```

该函数的工作方式类似 C 语言中的 putchar()，只不过其参数类型为 char，而不是 int。

注意：最初，put()成员只有一个原型——put（char）。可以传递一个 int 参数给它，该参数将被强制转换为 char。C++标准还要求只有一个原型。不过，当前的很多 C++实现都提供了 3 个原型：put（char）、put（signed char）和 put（unsigned char）。在这些实现中，给 put()传递一个 int 参数将导致错误消息，因为转换 int 的方式不止一种。使用显式强制类型转换的原型（如 cin.put（char(ch)））可使用 int 参数。

为成功地使用 cin.get()，需要知道其如何处理 EOF 条件。当该函数到达 EOF 时，将没有可返回的字符。相反，cin.get()将返回一个用符号常量 EOF 表示的特殊值。该常量是在头文件 iostream 中定义的。EOF 值必须不同于任何有效的字符值，以便程序不会将 EOF 与常规字符混淆。通常，EOF 被定义为值-1，因为没有 ASCII 码为-1 的字符，但并不需要知道实际的值，而只需在程序中使用 EOF 即可。例如，程序清单 5.18 的核心是这样：

```
char ch;
cin.get(ch);
while (cin.fail() == false)  // test for EOF
{
```

```
    cout << ch;
    ++count;
    cin.get(ch);
}
```

可以使用 int ch，并用 cin.get() 代替 cin.get(char)，用 cout.put() 代替 cout，用 EOF 测试代替 cin.fail() 测试：

```
int ch;     /// for compatibility with EOF value
ch = cin.get();
while (ch != EOF)
{
    cout.put(ch);   // cout.put(char(ch)) for some implementations
    ++count;
    ch = cin.get();
}
```

如果 ch 是一个字符，则循环将显示它。如果 ch 为 EOF，则循环将结束。

提示：需要知道的是，EOF 不表示输入中的字符，而是指出没有字符。

除了当前所做的修改外，关于使用 cin.get() 还有一个微妙而重要的问题。由于 EOF 表示的不是有效字符编码，因此可能不与 char 类型兼容。例如，在有些系统中，char 类型是没有符号的，因此 char 变量不可能为 EOF 值（-1）。由于这种原因，如果使用 cin.get()（没有参数）并测试 EOF，则必须将返回值赋给 int 变量，而不是 char 变量。另外，如果将 ch 的类型声明为 int，而不是 char，则必须在显示 ch 时将其强制转换为 char 类型。

程序清单 5.19 将程序清单 5.18 进行了修改，使用了 cin.get() 方法。它还通过将字符输入与 while 循环测试合并在一起，使代码更为简洁。

程序清单 5.19　textin4.cpp

```
// textin4.cpp -- reading chars with cin.get()
#include <iostream>
int main (void)
{
    using namespace std;
    int ch;                 // should be int, not char
    int count = 0;

    while ((ch = cin.get()) != EOF) // test for end-of-file
    {
        cout.put (char (ch));
        ++count;
    }
    cout << endl << count << " characters read\n";
     return 0;
}
```

注意：有些系统要么不支持来自键盘的模拟 EOF，要么支持地不完善，在这种情况下，上述范例将无法正常运行。如果使用 cin.get() 来锁住屏幕直到可以阅读它，这将不起作用，因为检测 EOF 时将禁止进一步读取输入。不过，可以使用程序清单 5.14 那样的计时循环来使屏幕停留一段时间。

下面是该程序的运行情况：

```
The sullen mackerel sulks in the shadowy shallows.<ENTER>
The sullen mackerel sulks in the shadowy shallows.
Yes, but the blue bird of happiness harbors secrets.<ENTER>
Yes, but the blue bird of happiness harbors secrets.
^Z
104 characters read
```

下面分析一下循环条件：

```
while ((ch = cin.get()) != EOF)
```

子表达式 ch=cin.get()两端的括号导致程序首先计算该表达式。为此，程序必须首先调用 cin.get()函数，然后将该函数的返回值赋给 ch。由于赋值语句的值为左操作数的值，因此整个子表达式变为 ch 的值。如果这个值是 EOF，则循环将结束，否则继续。该测试条件中所有的括号都是必不可少的。如果省略其中的一些括号：

```
while (ch = cin.get()!= EOF)
```

由于!=操作符的优先级高于=，因此程序将首先对 cin.get()的返回值和 EOF 进行比较。比较的结果为 false 或 true，而这些 bool 值将被转换为 0 或 1，并将值赋给 ch。

另一方面，使用 cin.get（ch）（有一个参数）进行输入时，将不会导致任何类型方面的问题。前面讲过，cin.get（char）函数在到达 EOF 时，不会将一个特殊值赋给 ch。事实上，在这种情况下，它不会将任何值赋给 ch。ch 不会被用来存储非 char 值。表 5.3 总结了 cin.get（char）和 cin.get()之间的差别。

表 5.3 cin.get（ch）与 cin.get()

属　性	cin.get（ch）	ch=cin.get()
传递输入字符的方式	赋给参数 ch	将函数返回值赋给 ch
用于字符输入时函数的返回值	istream 类对象（执行 bool 转换后为 true）	int 类型的字符编码
到达 EOF 时函数的返回值	istream 类对象（执行 bool 转换后为 false）	EOF

那么应使用 cin.get()还是 cin.get（char）呢？使用字符参数的版本更符合对象方式，因为其返回值是 istream 对象。这意味着可以将它们拼接起来。例如，下面的代码将输入中的下一个字符读入到 ch1 中，并将接下来的一个字符读入到 ch2 中：

```
cin.get (ch1).get (ch2);
```

这是可行的，因为函数调用 cin.get（ch1）返回一个 cin 对象，然后便可以通过该对象调用 get（ch2）。

get()的主要用途是能够将 stdio.h 的 getchar()和 putchar()函数转换为 iostream 的 cin.get()和 cout.put()方法。只要用头文件 iostream 替换 stdio.h，并用作用相似的方法替换所有的 getchar()和 putchar()即可（如果旧的代码使用 int 变量进行输入，而所用的实现包含 put()的多个原型，则必须做进一步的调整）。

5.6 嵌套循环和二维数组

如本章前面所述，for 循环是一种处理数组的工具。下面进一步讨论如何使用嵌套 for 循环中来处理二维数组。

首先，介绍一下什么是二维数组。到目前为止，本章使用的数组都是一维数组，因为每个数组都可以看作是一行数据。二维数组更像是一个表格——既有数据行又有数据列。例如，可以用二维数组来表示 6 个不同地区每季度的销售额，每一个地区占一行数据。也可以用二维数组来表示 RoboDork 在计算机游戏板上的位置。

C++没有提供二维数组类型，但用户可以创建每个元素本身都是数组的数组。例如，假设要存储 5 个城市在 4 年间的最高温度。在这种情况下，可以这样声明数组：

```
int maxtemps[4][5];
```

该声明意味着 maxtemps 是一个包含 4 个元素的数组，其中每个元素都是一个由 5 个整数组成的数组（参见图 5.5）。可以将 maxtemps 数组看作由 4 行组成，其中每一行有 5 个温度值。

表达式 maxtemps[0]是 maxtemps 数组的第一个元素，因此 maxtemps[0]本身就是一个由 5 个 int 组成的数组。maxtemps[0] 数组的第一个元素是 maxtemps [0] [0]，该元素是一个 int。因此，需要使用两个下标来访问 int 元素。可以认为第一个下标表示行，第二个下标表示列（参见图 5.6）。

假设要打印数组所有的内容，可以用一个 for 循环来改变行，用另一个被嵌套的 for 循环来改变列：

```
for (int row = 0; row < 4; row++)
{
    for (int col = 0; col < 5; ++col)
```

```
        cout << maxtemps[row][col] << "\t";
    cout << endl;
}
```

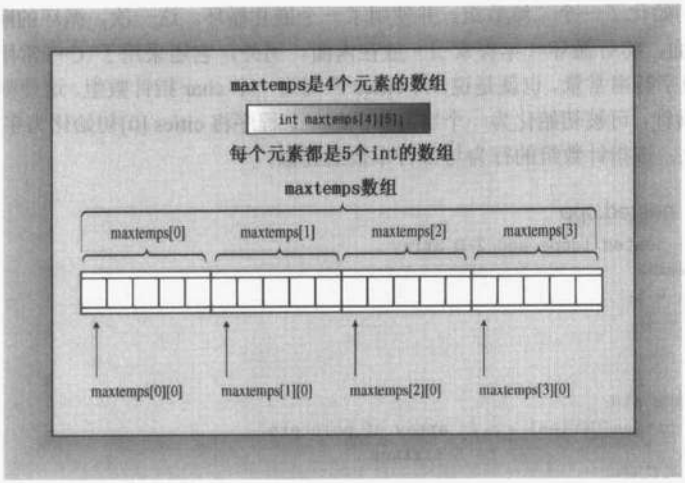


图 5.5　由数组组成的数组

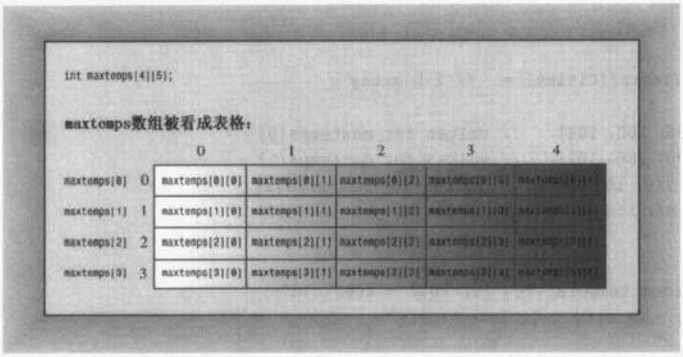


图 5.6　使用下标访问数组元素

对于每个 row 值，内部的 for 循环将遍历所有的 col 值。这个范例在每个值之后打印一个制表符（使用 C++转义字符表示时为\t），打印完每行后，打印一个换行符。

初始化二维数组

创建二维数组时，可以初始化其所有元素。这项技术建立在一维数组初始化技术的基础之上：提供由逗号分隔的用花括号括起的值列表：

```
// initializing a one-dimensional array
int btus[5] = { 23, 26, 24, 31, 28};
```

对于二维数组来说，由于每个元素本身就是一个数组，因此可以使用与上述代码类似的格式来初始化每一个元素。因此，初始化由一系列逗号分隔的一维数组初始化（用花括号括起）组成：

```
int maxtemps[4][5] =                // 2-D array
{
    {94, 98, 87, 103, 101},         // values for maxtemps[0]
    {98, 99, 91, 107, 105},         // values for maxtemps[1]
    {93, 91, 90, 101, 104},         // values for maxtemps[2]
    {95, 100, 88, 105, 103}         // values for maxtemps[3]
```

};

{94，98，87，103，101}初始化第一行，即 maxtemps [0]。作为一种风格，如果可能的话，每行数据应各占一行，这样阅读起来将更容易。

程序清单 5.20 初始化了一个二维数组，并使用了一个嵌套循环。这一次，循环的顺序相反，将列循环（城市索引）放在外面，将行循环（年份索引）放在内面。另外，它还采用了 C++常用的做法，将一个指针数组初始化为一组字符串常量。也就是说，将 cities 声明为一个 char 指针数组。这使得每个元素（如 cities [0]）都是一个 char 指针，可被初始化为一个字符串的地址。程序将 cities [0]初始化为字符串“Gribble City”的地址，等等。因此，该指针数组的行为与字符串数组类似。

程序清单 5.20 nested.cpp

```
// nested.cpp -- nested loops and 2-D array
#include <iostream>
const int Cities = 5;
const int Years = 4;
int main()
{
    using namespace std;
    const char * cities[Cities] =   // array of pointers
    {                               // to 5 strings
        "Gribble City",
        "Gribbletown",
        "New Gribble",
        "San Gribble",
        "Gribble Vista"
    };
    int maxtemps[Years][Cities] =   // 2-D array
    {
        {95, 99, 86, 100, 104},   // values for maxtemps[0]
        {95, 97, 90, 106, 102},   // values for maxtemps[1]
        {96, 100, 940, 107, 105}, // values for maxtemps[2]
        {97, 102, 89, 108, 104}   // values for maxtemps[3]
    };

    cout << "Maximum temperatures for 2002 - 2005\n\n";
    for (int city = 0; city < Cities; ++city)
    {
        cout << cities[city] << ":\t";
        for (int year = 0; year < Years; ++year)
            cout << maxtemps[year][city] << "\t";
        cout << endl;
    }

    return 0;
}
```

下面是该程序的输出：

```
Maximum temperatures for 2002 - 2005

Gribble City:   95  95  96  97
Gribbletown:    99  97  100 102
New Gribble:    86  90  940 89
San Gribble:    100 106 107 108
Gribble Vista:  104 102 105 104
```

在输出中使用制表符比使用空格可使数据排列更有规则。不过，制表符设置不相同，因此输出的外观将随系统而异。第 17 章将介绍更精确的、更复杂的、对输出进行格式化的方法。

在这个例子中，可以使用 char 数组的数组，而不是字符串指针数组。在这种情况下，声明如下：

```
char cities[25][Cities] =   // array of 5 arrays of 25 char
```

```
{
    "Gribble City",
    "Gribbletown",
    "New Gribble",
    "San Gribble",
    "Gribble Vista"
};
```

上述方法将全部五个字符串的最大长度限制为 24 个字符。指针数组存储五个字符串的地址，而使用 char 数组的数组时，将五个字符串分别复制到五个包含 25 个元素的 char 数组中。因此，从存储空间的角度说，使用指针数组更为经济；然而，如果要修改其中的任何一个字符串，则二维数组是更好的选择。令人惊讶的是，这两种方法使用相同的初始化列表，显示字符串的 for 循环代码页相同。

另外，还可以使用 string 对象数组，而不是字符串指针数组。在这种情况下，声明如下：

```
const string cities[Cities] =  // array of 5 strings
{
    "Gribble City",
    "Gribbletown",
    "New Gribble",
    "San Gribble",
    "Gribble Vista"
};
```

如果希望字符串是可修改的，则应省略限定符 const。使用 string 对象数组时，初始化列表和用于显示字符串的 for 循环代码与前两种方法中相同。在希望字符串是可修改的情况下，string 类自动调整大小的特性将使这种方法比使用二维数组更为方便。

5.7　总结

C++提供了 3 种循环：for 循环、while 循环和 do while 循环。如果循环测试条件为 true 或非零，则循环将重复执行相同的一组指令；如果测试条件为 false 或 0，则结束循环。for 循环和 while 循环都是入口条件循环，这意味着程序将在执行循环体中的语句之前检查测试条件。do while 循环是出口条件循环，这意味着其将在执行循环体中的语句之后检查条件。

每种循环的句法都要求循环体由一条语句组成。不过，这条语句可以是复合语句，也可以是语句块（由花括号括起的多条语句）。

关系表达式对两个值进行比较，常被用作循环测试条件。关系表达式是通过使用 6 种关系操作符之一构成的：<、<=、==、>=、>或!=。关系表达式的结果为 bool 类型，值为 true 或 false。

许多程序都逐字节地读取文本输入或文本文件，istream 类提供了多种可完成这种工作的方法。如果 ch 是一个 char 变量，则下面的语句：

```
cin >> ch;
```

将输入中的下一个字符读入到 ch 中。不过，它将忽略空格、换行符和制表符。下面的成员函数调用：

```
cin.get(ch);
```

读取输入中的下一个字符，而不管该字符是什么，并将其存储到 ch 中。成员函数调用 cin.get() 返回下一个输入字符——包括空格、换行符和制表符，因此，可以这样使用它：

```
ch = cin.get();
```

cin.get（char）成员函数调用通过返回转换为 false 的 bool 值来指出已到达 EOF，而 cin.get() 成员函数调用则通过返回 EOF 值来指出已到达 EOF，EOF 是在文件 iostream 中定义的。

嵌套循环是循环中的循环，适合用于处理二维数组。

5.8　复习题

1. 入口条件循环和出口条件循环之间的区别是什么？各种 C++循环分别属于其中的哪一种？

2. 如果下面的代码片段是有效程序的组成部分，它将打印什么内容？

```
int i;
for (i = 0; i < 5; i++)
     cout << i;
     cout << endl;
```

3. 如果下面的代码片段是有效程序的组成部分，它将打印什么内容？

```
int j;
for (j = 0; j < 11; j += 3)
     cout << j;
cout << endl << j << endl;
```

4. 如果下面的代码片段是有效程序的组成部分，它将打印什么内容？

```
int j = 5;
while (++j < 9)
     cout << j++ << endl;
```

5. 如果下面的代码片段是有效程序的组成部分，它将打印什么内容？

```
int k = 8;
do
     cout <<" k = " << k << endl;
while (k++ < 5);
```

6. 编写一个打印 1、2、4、8、16、32、64 的 for 循环，每轮循环都将计数变量的值乘以 2。

7. 如何在循环体中包括多条语句？

8. 下面的语句是否有效？如果无效，原因是什么？如果有效，它将完成什么工作？

```
int x = (1, 024);
```

下面的语句又如何呢？

```
int y;
y = 1, 024;
```

9. 在查看输入方面，cin >>ch 同 cin.get（ch）和 ch=cin.get()有什么不同？

5.9 编程练习

1. 编写一个要求用户输入两个整数的程序。该程序将计算并输出这两个整数之间（包括这两个整数）所有整数的和。这里假设先输入较小的整数。例如，如果用户输入的是 2 和 9，则程序将指出 2～9 之间所有整数的和为 44。

2. 编写一个要求用户输入数字的程序。每次输入后，程序都将报告到目前为止，所有输入的累计和。当用户输入 0 时，程序结束。

3. Daphne 以 10%的单利投资了 100 美元。也就是说，每一年的利润都是投资额的 10%，即每年 10 美元：

利息=0.10×原始存款

而 Cleo 以 5%的复利投资了 100 美元。也就是说，利息是当前存款（包括获得的利息）的 5%，：

利息=0.05×当前存款

Cleo 在第一年投资 100 美元的盈利是 5%——得到了 105 美元。下一年的盈利是 105 美元的 5%——即 5.25 美元，依此类推。请编写一个程序，计算多少年后，Cleo 的投资价值才能超过 Daphne 的投资价值，并显示此时两个人的投资价值。

4. 假设要销售 *C++ For Fools* 一书。请编写一个程序，输入全年中每个月的销售量（图书数量，而不是销售额）。程序通过循环，使用初始化为月份字符串的 char *数组（或 string 对象数组）逐月进行提示，并将输入的数据储存在一个 int 数组中。然后，程序计算数组中各元素的总数，并报告这一年的销售情况。

5. 完成编程练习 4，但这一次使用一个二维数组来存储输入——3 年中每个月的销售量。程序将报告每年销售量以及 3 年的总销售量。

6. 设计一个名为 car 的结构，用它存储下述有关汽车的信息：生产商（存储在字符数组或 string 对象中的字符串）、生产年份（整数）。编写一个程序，向用户询问有多少辆汽车。随后，程序使用 new 来创建一个由相应数量的 car 结构组成的动态数组。接下来，程序提示用户输入每辆车的生产商（可能由多个单词组成）和年份信息。请注意，这需要特别小心，因为它将交替读取数值和字符串（参见第 4 章）。最后，程序将显示每个结构的内容。该程序的运行情况如下：

```
How many cars do you wish to catalog? 2
Car #1:
Please enter the make: Hudson Hornet
Please enter the year made: 1952
Car #2:
Please enter the make: Kaiser
Please enter the year made: 1951
Here is your collection:
1952 Hudson Hornet
1951 Kaiser
```

7. 编写一个程序，它使用一个 char 数组和循环来每次读取一个单词，直到用户输入 done 为止。随后，该程序指出用户输入了多少个单词（不包括 done 在内）。下面是该程序的运行情况：

```
Enter words (to stop, type the word done):
anteater birthday category dumpster
envy finagle geometry done for sure
You entered a total of 7 words.
```

您应在程序中包含头文件 cstring，并使用函数 strcmp() 来进行比较测试。

8. 编写一个满足前一个练习中描述的程序，但使用 string 对象而不是字符数组。请在程序中包含头文件 string，并使用关系操作符来进行比较测试。

9. 编写一个使用嵌套循环的程序，要求用户输入一个值，指出要显示多少行。然后，程序将显示相应行数的星号，其中第一行包括一个星号，第二行包括两个星号，依此类推。每一行包含的字符数等于用户指定的行数，在星号不够的情况下，在星号前面加上句点。该程序的运行情况如下：

```
Enter number of rows: 5
....*
...**
..***
.****
*****
```

第 6 章　分支语句和逻辑操作符

本章内容包括：

- if 语句。
- if else 语句。
- 逻辑操作符：&&、||和!。
- cctype 字符函数库。
- 条件操作符：?:。
- switch 语句。
- continue 和 break 语句。
- 读取数字的循环。
- 基本文件输入/输出。

设计智能程序的一个关键是使程序具有决策能力。第 5 章介绍了一种决策方式——循环，在循环中，程序决定是否继续循环。现在，来研究一下 C++是如何使用分支语句在可选择的操作中做出决定的。程序应使用哪一种防止吸血鬼的方案（大蒜还是十字架）呢？用户选择了哪个菜单选项呢？用户是否输入了 0？C++提供了 if 和 switch 语句来进行决策，它们是本章的主要主题。另外，还将介绍条件操作符和逻辑操作符，前者提供了另一种决策方式，而后者允许将两个测试组合在一起。最后，本章将首次介绍文件输入/输出。

6.1　if 语句

当 C++程序必须决定是否执行某个操作时，通常使用 if 语句来实现选择。if 有两种格式：if 和 if else。首先看一看简单的 if。它模仿英语，例如，“If you have a Captain Cookie card，you get a free cookie（如果您有一张 Captain Cookie 卡，就可获得免费的小甜饼）”。如果测试条件为 true，则 if 语句将引导程序执行语句或语句块；如果条件是 false，程序将会跳过这条语句或语句块。因此，if 语句让程序能够决定是否应执行特定的语句。

if 语句的句法与 while 相似：

```
if (test-condition)
   statement
```

如果 test-condition（测试条件）为 true，则程序将执行 statement（语句），后者既可以是一条语句，也可以是语句块。如果测试条件为 false，则程序将跳过语句（参见图 6.1）。和循环测试条件一样，if 测试条件也将被强制转换为 bool 值，因此 0 将被转换为 false，非零为 true。整个 if 语句被视为一条语句。

通常情况下，测试条件都是关系表达式，如那些用来控制循环的表达式。例如，假设读者希望程序计算输入中的空格数和字符总数，则可以在 while 循环中使用 cin.get（char）来读取字符，然后使用 if 语句识别空格字符并计算其总数。程序清单 6.1 完成了这项工作，它使用句点（.）来确定句子的结尾。

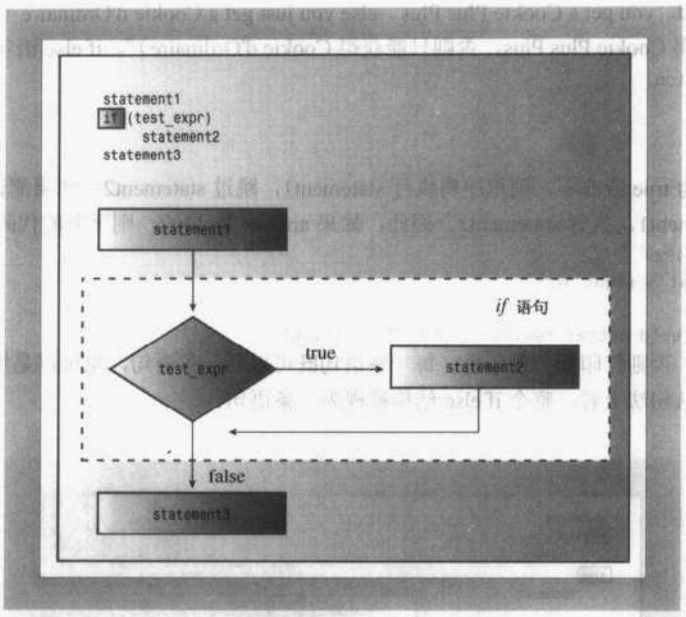


图 6.1　if 语句的结构

程序清单 6.1　if.cpp

```
// if.cpp -- using the if statement
#include <iostream>
int main()
{
    using namespace std;
    char ch;
    int spaces = 0;
    int total = 0;
    cin.get (ch);
    while (ch != '. ') // quit at end of sentence
    {
        if (ch == ' ')  // check if ch is a space
            ++spaces;
        ++total;        // done every time
        cin.get (ch);
    }
    cout << spaces << " spaces, " << total;
    cout << " characters total in sentence\n";
    return 0;
}
```

下面是该程序的输出：

```
The balloonist was an airhead
with lofty goals.
6 spaces, 46 characters total in sentence
```

正如程序中的注释指出的，仅当 ch 为空格时，语句++spaces;才被执行。因为语句++tota;位于 if 语句的外面，因此在每轮循环中都将被执行。注意，字符总数中包括按回车键生成的换行符。

6.1.1　if else 语句

if 语句让程序决定是否执行特定的语句或语句块，而 if else 语句则让程序决定执行两条语句或语句块中的哪一条，这种语句对于选择其中一种操作很有用。C++的 if else 语句模仿了简单的英语，如“If you have

a Captain Cookie card，you get a Cookie Plus Plus，else you just get a Cookie d'Ordinaire（如果您拥有 Captain Cookie 卡，将可获得 Cookie Plus Plus，否则只能获得 Cookie d'Ordinaire）”。if else 语句的通用格式如下：

```
if (test-condition)
   statement1
else
   statement2
```

如果测试条件为 true 或非零，则程序将执行 statement1，跳过 statement2；如果测试条件为 false 或 0，则程序将跳过 statement1，执行 statement2。因此，如果 answer 是 1492，则下面的代码片段：

```
if (answer == 1492)
   cout << "That's right!\n";
else
    cout << "You'd better review Chapter 1 again.\n";
```

将打印第一条信息，否则打印第二条信息。每一条语句既可以是一条语句，也可以是用花括号括起的语句块（参见图 6.2）。从句法上看，整个 if else 结构被视为一条语句。

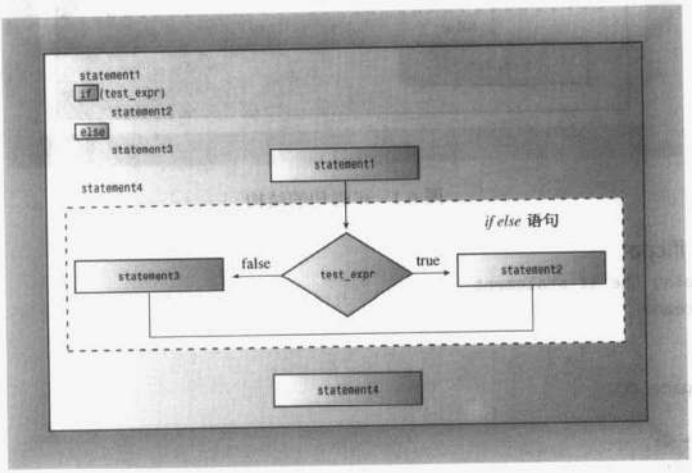


图 6.2 if else 语句的结构

例如，假设要通过对字母进行加密编码来修改输入的文本（换行符不变）。这样，每个输入行都被转换为一行输出，且长度不变。这意味着程序对换行符采用一种操作，而对其他字符采用另一种操作。正如程序清单 6.2 所表明的，if else 使得这项工作非常简单。

程序清单 6.2 ifelse.cpp

```
// ifelse.cpp -- using the if else statement
#include <iostream>
int main()
{
   using namespace std;
   char ch;

   cout << "Type, and I shall repeat.\n";
   cin.get (ch);
   while (ch != '.')
   {
      if (ch == '\n')
         cout << ch;     // done if newline
      else
         cout << ++ch;   // done otherwise
      cin.get (ch);
```

```
    }
// try ch + 1 instead of ++ch for interesting effect
    cout << "\nPlease excuse the slight confusion.\n";
    return 0;
}
```

下面是该程序的运行情况：

```
Type, and I shall repeat.
I am extraordinarily pleased
J!bn!fyusbpsejobsjmz!qmfbtfe
to use such a powerful computer.
up!vtf!tvdi!b!qpxfsgvm!dpnqvufs
Please excuse the slight confusion.
```

注意，程序清单 6.2 中的注释之一指出，将++ch 改为 ch+1 将产生一种有趣的效果。能推断出它是什么吗？如果不能，就试验一下，然后看一看是否可以解释发生的情况（提示：想一想 cout 是如何处理不同的类型的）。

6.1.2　格式化 if else 语句

if else 中的两种操作都必须是一条语句。如果需要多条语句，需要用花括号将它们括起来，组成一个块语句。和有些语言（如 BASIC 和 FORTRAN）不同的是，由于 C++不会自动将 if 和 else 之间的所有代码视为一个代码块，因此必须使用花括号将这些语句组合成一个语句块。例如，下面的代码将出现编译器错误：

```
if (ch == 'Z')
    zorro++;      // if ends here
    cout << "Another Zorro candidate\n";
else              // wrong
    dull++;
    cout << "Not a Zorro candidate\n";
```

编译器把它看作是一条以 zorro ++;语句结尾的简单 if 语句，接下来是一条 cout 语句。到目前为止，一切正常。但之后编译器发现一个独立的 else，这被视为句法错误。

请添加花括号，将代码转换为所希望的形式：

```
if (ch == 'Z')
{                     // if true block
    zorro++;
    cout << "Another Zorro candidate\n";
}
else
{                    // if false block
    dull++;
    cout << "Not a Zorro candidate\n";
}
```

由于 C++是自由格式语言，因此只要使用花括号将语句括起，对花括号的位置没有任何限制。上述代码演示了一种流行的格式，下面是另一种流行的格式：

```
if (ch == 'Z') {
    zorro++;
    cout << "Another Zorro candidate\n";
    }
else {
    dull++;
    cout << "Not a Zorro candidate\n";
    }
```

第一种格式强调的是语句的块结构，第二种格式则将语句块与关键字 if 和 else 更紧密地结合在一起。这两种风格清晰、一致，应该能够满足要求；不过，可能会有老师或雇主在这个问题上的观点强硬而固执。

6.1.3 if else if else 结构

与实际生活中发生的情况类似，计算机程序也可能提供两个以上的选择。可以将 C++的 if else 语句进行扩展来满足这种需求。正如读者知道的，else 之后应是一条语句，也可以是语句块。由于 if else 语句本身是一条语句，所以可以放在 else 的后面：

```
if (ch == 'A')
    a_grade++;              // alternative # 1
else
    if (ch == 'B')          // alternative # 2
        b_grade++;          // subalternative # 2a
    else
        soso++;             // subalternative # 2b
```

如果 ch 不是'A'，则程序将执行 else。执行到那里，另一个 if else 又提供了两种选择。C++的自由格式允许将这些元素排列成便于阅读的格式：

```
if (ch == 'A')
    a_grade++;              // alternative # 1
else if (ch == 'B')
    b_grade++;              // alternative # 2
else
    soso++;                 // alternative # 3
```

这看上去像是一个新的控制结构——if else if else 结构。但实际上，它只是一个 if else 被包含在另一个 if else 中。修订后的格式更为清晰，使程序员通过浏览代码便能确定不同的选择。整个构造仍被视为一条语句。

程序清单 6.3 使用这种格式创建了一个小型测验程序。

程序清单 6.3 ifelseif.cpp

```
// ifelseif.cpp -- using if else if else
#include <iostream>
const int Fave = 27;
int main()
{
    using namespace std;
    int n;

    cout << "Enter a number in the range 1-100 to find ";
    cout << "my favorite number: ";
    do
    {
        cin >> n;
        if (n < Fave)
          cout << "Too low -- guess again: ";
        else if (n > Fave)
          cout << "Too high -- guess again: ";
        else
          cout << Fave << " is right!\n";
    } while (n != Fave);
    return 0;
}
```

下面是该程序的输出：

```
Enter a number in the range 1-100 to find my favorite number: 50
Too high -- guess again: 25
Too low -- guess again: 37
Too high -- guess again: 31
Too high -- guess again: 28
Too high -- guess again: 27
27 is right!
```

条件操作符和错误防范

许多程序员将更直观的表达式 variable = =value 反转为 value = =variable，以此来捕获将相等操作符误写为赋值操作符的错误。例如，将条件输入为：

```
if (3 == myNumber)
```

时将是有效的，可以正常工作。但如果将它错误地输入为：

```
if (3 = myNumber)
```

则编译器将生成错误消息，因为它以为程序员试图将一个值赋给一个字面值（3 总是等于 3，而不能将另一个值赋给它）。假设犯了类似的错误，但使用的是前一种表示方法：

```
if (myNumber = 3)
```

则编译器将只是把 3 赋给 myNumber，而 if 中的语句块将包含非常常见的、而又非常难以发现的错误。一般来说，编写让编译器能够发现错误的代码，比找出导致难以理解的错误的原因要容易得多。

6.2　逻辑表达式

经常需要测试多种条件。例如，字符要是小写，其值就必须大于或者等于'a'，并且小于或者等于'z'。如果要求用户使用 *y* 或 *n* 进行响应，则希望用户无论输入大写（Y 和 N）或小写都可以。为满足这种需要，C++提供了 3 种逻辑操作符，来组合或修改已有的表达式。这些操作符分别是逻辑 OR(||)、逻辑 AND(&&)和逻辑 NOT（!）。下面介绍这些操作符。

6.2.1　逻辑 OR 操作符：||

在英语中，当两个条件中有一个或全部满足某个要求时，可以用单词 or 来指明这种情况。例如，如果您或您的配偶在 MegaMicro 公司工作，您就可以参加 MegaMicro 公司的野餐会。C++可以采用逻辑 OR 操作符（||），将两个表达式组合在一起。如果原来表达式中的任何一个或全部都为 true（或非零），则得到的表达式的值为 true；否则，表达式的值为 false。下面是一些例子：

```
5 ==5 || 5 == 9    // true because first expression is true
5 > 3 || 5 > 10    // true because first expression is true
5 > 8 || 5 < 10    // true because second expression is true
5 < 8 || 5 > 2     // true because both expressions are true
5 > 8 || 5 < 2     // false because both expressions are false
```

由于||的优先级比关系操作符低，因此不需要在这些表达式中使用括号。表 6.1 总结了||的工作原理。

表 6.1　||操作符

	expr1 || expr2 的值			expr1 || expr2 的值	
	expr1 = = true	expr1 = = false	expr2 = = false	true	false
expr2 = = true	true	true			

C++规定，||操作符是一个顺序点（sequence point）。也是说，先修改左侧的值，然后再对右侧的值进行判定。例如，请看下面的表达式：

```
i++ < 6 || i == j
```

假设 i 原来的值为 10，则在对 i 和 j 进行比较时，i 的值将为 11。另外，如果左侧的表达式为 true，则 C++将不会去判定右侧的表达式，因为只要一个表达式为 true，则整个逻辑表达式为 true（读者可能还记得，冒号和逗号操作符也是顺序点）。

程序清单 6.4 在一条 if 语句中使用||操作符来检查某个字符的大写或小写。另外，它还使用了 C++字符串的拼接特性（参见第 4 章）将一个字符串分布在 3 行中。

程序清单 6.4 or.cpp

```
// or.cpp -- using the logical OR operator
#include <iostream>
int main()
{
   using namespace std;
   cout << "This program may reformat your hard disk\n"
         "and destroy all your data.\n"
         "Do you wish to continue? <y/n> ";
   char ch;
   cin >> ch;
   if (ch == 'y' || ch == 'Y')            // y or Y
      cout << "You were warned!\a\a\n";
   else if (ch == 'n' || ch == 'N')       // n or N
      cout << "A wise choice ... bye\n";
   else
      cout << "That wasn't a y or an n, so I guess I'll "
           "trash your disk anyway.\a\a\a\n";
   return 0;
}
```

下面是该程序的运行情况：

```
This program may reformat your hard disk
and destroy all your data.
Do you wish to continue? <y/n> N
A wise choice ... bye
```

由于程序只读取一个字符，因此只读取响应的第一个字符。这意味着用户可以用 NO!（而不是 N）进行回答，程序将只读取 N。但是，如果程序后面再读取输入时，将从 O 开始读取。

6.2.2 逻辑 AND 操作符：&&

逻辑 AND 操作符（&&），也是将两个表达式组合成一个表达式。仅当原来的两个表达式都为 true 时，得到的表达式的值才为 true。下面是一些例子：

```
5 == 5 && 4 == 4   // true because both expressions are true
5 == 3 && 4 == 4   // false because first expression is false
5 > 3 && 5 > 10    // false because second expression is false
5 > 8 && 5 < 10    // false because first expression is false
5 < 8 && 5 > 2     // true because both expressions are true
5 > 8 && 5 < 2     // false because both expressions are false
```

由于&&的优先级低于关系操作符，因此不必在这些表达式中使用括号。和||操作符一样，&&操作符也是顺序点，因此将首先判定左侧，并且在右侧被判定之前产生所有的副作用。如果左侧为 false，则整个逻辑表达式必定为 false，在这种情况下，C++将不会再对右侧进行判定。表 6.2 总结了&&操作符的工作方式。

表 6.2 &&操作符

	expr1 &&expr2 的值			expr1 &&expr2 的值	
	expr1 = = true	expr1 = = false	expr2 = = false	false	false
expr2 = = true	true	false			

程序清单 6.5 演示了如何用&&来处理一种常见的情况——由于两种不同的原因而结束 while 循环。在这个程序清单中，一个 while 循环将值读入到数组。一个测试（i<ArSize）在数组被填满时循环结束，另一个测试（temp>=0）让用户通过输入一个负值来提前结束循环。该程序使用&&操作符将两个测试组合成一个条件。该程序还使用了两条 if 语句、一条 if else 语句和一个 for 循环，因此它演示了本章和第 5 章的多个主题。

程序清单 6.5　and.cpp

```
// and.cpp -- using the logical AND operator
#include <iostream>
const int ArSize = 6;
int main()
{
    using namespace std;
    float naaq[ArSize];
    cout << "Enter the NAAQs (New Age Awareness Quotients) "
         << "of\nyour neighbors. Program terminates "
         << "when you make\n" << ArSize << " entries"
         << "or enter a negative value.\n";

    int i = 0;
    float temp;
    cout << "First value: ";
    cin >> temp;
    while (i < ArSize && temp >= 0)    // 2 quitting criteria
    {
        naaq[i] = temp;
        ++i;
        if (i < ArSize)                // room left in the array,
          {
            cout << "Next value: ";
            cin >> temp;                    // so get next value
          }
    }
    if (i == 0)
        cout << "No data--bye\n";
    else
    {
        cout << "Enter your NAAQ: ";
        float you;
        cin >> you;
        int count = 0;
        for (int j = 0; j < i; j++)
          if (naaq[j] > you)
            ++count;
        cout << count;
        cout << " of your neighbors have greater awareness of\n"
             << "the New Age than you do.\n";
    }
    return 0;
}
```

注意，该程序将输入放在临时变量 temp 中。在核实输入有效后，程序才将这个值赋给数组。

下面是该程序的两次运行情况。一次在输入 6 个值后结束：

```
Enter the NAAQs (New Age Awareness Quotients) of
your neighbors. Program terminates when you make
6 entries or enter a negative value.
First value: 20
Next value: 72
Next value: 15
Next value: 6
Next value: 130
Next value: 145
Enter your NAAQ: 50
3 of your neighbors have greater awareness of
the New Age than you do.
```

另一次在输入负值后结束：

```
Enter the NAAQs (New Age Awareness Quotients) of
your neighbors. Program terminates when you make
6 entries or enter a negative value.
First value: 123
Next value: 119
Next value: 4
Next value: 89
Next value: -1
Enter your NAAQ: 123.031
0 of your neighbors have greater awareness of
the New Age than you do.
```

1. 程序说明

来看看该程序的输入部分：

```
cin >> temp;
while (i < ArSize && temp >= 0) // 2 quitting criteria
{
   naaq[i] = temp;
   ++i;
   if (i < ArSize)           // room left in the array,
   {
      cout << "Next value: ";
      cin >> temp;           // so get next value
   }
```

该程序首先将第一个输入值读入到临时变量（temp）中。然后，while 测试条件查看数组中是否还有空间（i<ArSize）以及输入值是否为非负（temp >=0）。如果满足条件，则将 temp 的值复制到数组中，并将数组索引加 1。此时，由于数组下标从 0 开始，因此 i 指示输入了多少个值。也是说，如果 i 从 0 开始，则第一轮循环将一个值赋给 naaq[0]，然后将 i 设置为 1。

当数组被填满或用户输入了负值时，循环将结束。注意，仅当 i 小于 ArSize 时，即数组中还有空间时，循环才将另外一个值读入到 temp 中。

获得数据后，如果没有输入任何数据（即第一次输入的是一个负数），程序将使用 if else 语句指出这一点，如果存在数据，就对数据进行处理。

2. 用&&来设置取值范围

&&操作符还允许建立一系列 if else if else 语句，其中每种选择都对应于一个特定的取值范围。程序清单 6.6 演示了这种方法。另外，它还演示了一种用于处理一系列消息的技术。与 char 指针变量可以通过指向一个字符串的开始位置来标识该字符串一样，char 指针数组也可以标识一系列字符串，只要将每一个字符串的地址赋给各个数组元素即可。程序清单 6.6 使用 qualify 数组来存储 4 个字符串的地址，例如，qualify [1]存储字符串“mud tug-of-war\n”的地址。然后，程序便能够将 cout、strlen()或 strcmp()用于 qualify [1]，就像用于其他字符串指针一样。使用 const 限定符可以避免无意间修改这些字符串。

程序清单 6.6 more_and.cpp

```
// more_and.cpp -- using logical AND operator
#include <iostream>
const char * qualify[4] =        // an array of pointers
{                                // to strings
   "10, 000-meter race.\n",
   "mud tug-of-war.\n",
   "masters canoe jousting.\n",
   "pie-throwing festival.\n"
};
int main()
{
   using namespace std;
```

```
    int age;
    cout << "Enter your age in years: ";
    cin >> age;
    int index;

    if (age > 17 && age < 35)
        index = 0;
    else if (age >= 35 && age < 50)
        index = 1;
    else if (age >= 50 && age < 65)
        index = 2;
    else
        index = 3;
    cout << "You qualify for the " << qualify[index];
    return 0;
}
```

注意：读者可能还记得，在一些 C++实现中，要对数组进行初始化，必须在数组声明中使用关键字 static。正如第 9 章介绍的，这种限制适用于函数体内声明的数组。在函数体外声明的数组（如程序清单 6.6 中的 qualify）被称为外部数组（external array），甚至在 ANSI C 之前的实现中也可以对这种数组进行初始化。

下面是该程序的运行情况：

```
Enter your age in years: 87
You qualify for the pie-throwing festival.
```

由于输入的年龄不与任何测试取值范围匹配，因此程序将索引设置为 3，然后打印相应的字符串。

3. 程序说明

在程序清单 6.6 中，表达式 age > 17 && age < 35 测试年龄是否位于两个值之间，即年龄是否在 18 到 34 之间。表达式 age >= 35 && age < 50 使用<=操作符将 35 包括在取值范围内。如果程序使用 age > 35 && age < 50，则 35 将被所有的测试忽略。在使用取值范围测试时，应确保取值范围之间既没有缝隙，又没有重叠。另外，应确保正确设置每个取值范围（参见本节后面的注释“取值范围测试”）。

if else 语句用来选择数组索引，而索引则标识特定的字符串。

取值范围测试

取值范围测试的每一部分都使用 AND 操作符将两个完整的关系表达式组合起来：

```
if (age > 17 && age < 35)  // OK
```

不要使用数学符号将其表示为：

```
if (17 < age < 35)         // Don't do this!
```

编译器不会捕获这种错误，因为它仍然是有效的 C++句法。<操作符从左向右结合，因此上述表达式的含义如下：

```
if ( (17 < age) < 35)
```

但 17 < age 的值要么为 true（1），要么为 false（0）。不管是哪种情况，表达式 17 < age 的值都小于 35，因此整个测试的结果总是 true!

6.2.3　逻辑 NOT 操作符：!

!操作符将它后面的表达式的真值取反。也是说，如果 expression 为 true，则!expression 是 false；如果 expression 为 false，则!expression 是 true。更准确地说，如果 expression 为 true 或非零，则!expression 为 false。

通常，不使用这个操作符可以更清楚地表示关系：

```
if (!(x > 5))              // if (x <= 5) is clearer
```

但是，!操作符对于返回 true-false 值或可以被解释为 true-false 值的函数来说很有用。例如，如果 C-风格字符串 s1 和 s2 不同，则 strcmp（s1，s2）将返回非零（true）值，否则返回 0。这意味着如果这两个字符串相同，则!strcmp（s1，s2）为 true。

程序清单 6.7 使用这种技术（将!操作符用于函数返回值）来筛选可赋给 int 变量的数字输入。如果用户定义的函数 is_int()（稍后将详细）的参数位于 int 类型的取值范围内，则它将返回 true。然后，程序使用 while（!is-int（num））测试来拒绝不在该取值范围内的值。

程序清单 6.7 not.cpp

```
// not.cpp -- using the not operator
#include <iostream>
#include <climits>
bool is_int(double);
int main()
{
    using namespace std;
    double num;

    cout << "Yo, dude! Enter an integer value: ";
    cin >> num;
    while (!is_int(num))   // continue while num is not int-able
    {
        cout << "Out of range -- please try again: ";
        cin >> num;
    }
    int val = int(num);   // type cast
    cout << "You've entered the integer " << val << "\nBye\n";
    return 0;
}

bool is_int(double x)
{
    if (x <= INT_MAX && x >= INT_MIN)  // use climits values
        return true;
    else
        return false;
}
```

注意：如果系统没有提供 climits，请使用 limits.h。

下面是该程序在 int 占 32 位的系统上的运行情况：

```
Yo, dude! Enter an integer value: 6234128679
Out of range -- please try again: -8000222333
Out of range -- please try again: 99999
You've entered the integer 99999
Bye
```

程序说明

如果给读取 int 值的程序输入一个过大的值，很多 C++实现只是将这个值截短为合适的大小，并不会通知丢失了数据。程序清单 6.7 中的程序避免了这样的问题，它首先将可能的 int 值作为 double 值来读取。double 类型的精度足以存储典型的 int 值，而且取值范围更大。

布尔函数 *is_int*()使用了 *climits* 文件（第 3 章讨论过）中定义的两个符号常量（INT_MAX 和 INT_MIN）来确定其参数是否位于适当的范围内。如果是，该函数将返回 true，否则返回 false。

main()程序使用 while 循环来拒绝无效输入，直到用户输入有效的值为止。可以在输入超出取值范围时显示 int 的界限，这样程序将更为友好。确认输入有效后，程序将其赋给一个 int 变量。

6.2.4 逻辑操作符细节

正如本章前面指出的，C++逻辑 OR 和逻辑 AND 操作符的优先级都低于关系操作符。这意味着下面的表达式。

```
x > 5 && x < 10
```

将被解释为：

```
(x > 5) && (x < 10)
```

另一方面，!操作符的优先级高于所有的关系操作符和算术操作符。因此，要对表达式求反，必须用括号将其括起，如下所示：

```
!(x > 5)    // is it false that x is greater than 5
!x > 5      // is !x greater than 5
```

第二个表达式总是为 false，因为!x 的值只能为 true 或 false，而它们将被转换为 1 或 0。

逻辑 AND 操作符的优先级高于逻辑 OR 操作符。因此，表达式：

```
age > 30 && age < 45 || weight > 300
```

被解释为：

```
(age > 30 && age < 45) || weight > 300
```

也是说，一个条件是 age 位于 31 到 44 之间，另一个条件是 weight 大于 300。如果这两个条件中的一个或全部都为 true，则整个表达式为 true.

当然，还可以用括号将所希望的解释告诉程序。例如，假设要用&&将 age 大于 50 或 weight 大于 300 的条件与 donation 大于 1000 的条件组合在一起，则必须使用括号将 OR 部分括起：

```
(age > 50 || weight > 300) && donation > 1000
```

否则，编译器将把 weight 条件与 donation 条件（而不是 age 条件）组合在一起。

虽然 C++操作符的优先级规则常可能不使用括号便可以编写复合比较的语句，但最简单的方法还是用括号将测试进行分组，而不管是否需要括号。这样代码容易阅读，避免读者查看不常使用的优先级规则，并减少由于没有准确记住所使用的规则而出错的可能性。

C++确保程序从左向右进行计算逻辑表达式，并在知道答案后立刻停止。例如，假设有下面的条件：

```
x != 0  && 1.0 / x > 100.0
```

如果第一个条件为 false，则整个表达式肯定为 false。这是因为要使整个表达式为 true，每个条件都必须为 true。知道第一个条件为 false 后，程序将不判定第二个条件。这个例子非常幸运，因为计算第二个条件将导致被 0 除，这是计算机没有定义的操作。

6.2.5　其他表示方式

并不是所有的键盘都提供了用作逻辑操作符的符号，因此 C++标准提供了另一种表示方式，如表 6.3 所示。标识符 and、or 和 not 都是 C++保留字，这意味着不能将它们用作变量名等。它们不是关键字，因为它们都是已有语言特性的另一种表示方式。另外，它们并不是 C 语言中的保留字，但 C 语言程序可以将它们用作操作符，只要在程序中包含了头文件 iso646.h。C++不要求使用头文件。

表 6.3　逻辑操作符：另一种表示方式

操 作 符	替 换 代 表	操 作 符	替 换 代 表
&&	and	!	not
||	or		

6.3　字符函数库 cctype

C++从 C 语言继承了一个与字符相关的、非常方便的函数软件包，它可以简化诸如确定字符是否为大写字母、数字、标点符号等工作，这些函数的原型是在头文件 cctype（老式的风格中为 ctype.h）中定义的。例如，如果 ch 是一个字母，则 isalpha（ch）函数返回一个非零值，否则返回 0。同样，如果 ch 是标点符号（如逗号或句号），函数 ispunct（ch）将返回 true（这些函数的返回类型为 int，而不是 bool，但通常 bool 转换让您能够将它们视为 bool 类型）。

使用这些函数比使用 AND 和 OR 操作符更方便。例如，下面是使用 AND 和 OR 来测试字符 ch 是不

是字母字符的代码：

```
if ( (ch >= 'a' && ch <= 'z') || (ch >= 'A' && ch <= 'Z') )
```

与使用 isalpha() 相比：

```
if (isalpha (ch) )
```

isalpha() 不仅更容易使用，而且更通用。AND/OR 格式假设 A～Z 的字符编码是连续的，其他字符的编码不在这个范围内。这种假设对于 ASCII 码来说是成立的，但通常并非总是如此。

程序清单 6.8 演示一些 ctype 库函数。具体地说，它使用 isalpha() 来检查字符是否为字母字符，使用 isdigits() 来测试字符是否为数字字符，如 3；使用 isspace() 来测试字符是否为空白，如换行符、空格和制表符；使用 ispunct() 来测试字符是否为标点符号。该程序还复习了 if else if 结构，并在一个 while 循环中使用了 cin.get (char)。

程序清单 6.8 cctype.cpp

```
// cctypes.cpp -- using the ctype.h library
#include <iostream>
#include <cctype>           // prototypes for character functions
int main()
{
   using namespace std;
   cout << "Enter text for analysis, and type @"
        " to terminate input.\n";
   char ch;
   int whitespace = 0;
   int digits = 0;
   int chars = 0;
   int punct = 0;
   int others = 0;

   cin.get (ch);                // get first character
   while (ch != '@')            // test for sentinal
   {
      if (isalpha (ch))         // is it an alphabetic character?
         chars++;
      else if (isspace (ch))    // is it a whitespace character?
         whitespace++;
      else if (isdigit (ch))    // is it a digit?
         digits++;
      else if (ispunct (ch))    // is it punctuation?
         punct++;
      else
         others++;
      cin.get (ch);             // get next character
   }
   cout << chars << " letters, "
       << whitespace << " whitespace, "
       << digits << " digits, "
       << punct << " punctuations, "
       << others << " others.\n";
   return 0;
}
```

下面是该程序的运行情况。注意，空白字符计数中包括换行符：

```
Enter text for analysis, and type @ to terminate input.
Jody "Java-Java" Joystone, noted restaurant critic,
celebrated her 39th birthday with a carafe of 1982
Chateau Panda.@
89 letters, 16 whitespace, 6 digits, 6 punctuations, 0 others.
```

表 6.4 对 cctype 软件包中的函数进行了总结。有些系统可能没有表中列出的一些函数，也可能还有在

表中没有列出的一些函数。

表 6.4　cctype 中的字符函数

函数名称	返回值
isalnum()	如果参数是字母数字，即字母或数字，该函数返回 true
isalpha()	如果参数是字母，该函数返回 true
isblank()	如果参数是空格或水平制表符，该函数返回 true
iscntrl()	如果参数是控制字符，该函数返回 true
isdigit()	如果参数是数字（0～9），该函数返回 true
isgraph()	如果参数是除空格之外的打印字符，该函数返回 true
islower()	如果参数是小写字母，该函数返回 true
isprint()	如果参数是打印字符（包括空格），该函数返回 true
ispunct()	如果参数是标点符号，该函数返回 true
isspace()	如果参数是标准空白字符，如空格、进纸、换行符、回车、水平制表符或者垂直制表符，该函数返回 true
isupper()	如果参数是大写字母，该函数返回 true
isxdigit()	如果参数是十六进制的数字，即 0～9、a～f 或 A～F，该函数返回 true
tolower()	如果参数是大写字符，则返回其小写，否则返回该参数
toupper()	如果参数是小写字符，则返回其大写，否则返回该参数

6.4　?:操作符

C++有一个常被用来代替 if else 语句的操作符，这个操作符被称为条件操作符（?:），它是 C++中惟一一个需要 3 个操作数的操作符。该操作符的通用格式如下：

```
expression1 ? expression2 : expression3
```

如果 expression1 为 true，则整个条件表达式的值为 expression2 的值；否则，整个表达式的值为 expression3 的值。下面的两个范例演示了该操作符是如何工作的：

```
5 > 3 ? 10: 12  // 5 > 3 is true, so expression value is 10
3 == 9? 25: 18  // 3 == 9 is false, so expression value is 18
```

我们可以这样解释第一个范例：如果 5 大于 3，则整个表达式的值为 10，否则为 12。当然，在实际的编程中，这些表达式中将包含变量。

程序清单 6.9 使用条件操作符来确定两个值中较大的一个。

程序清单 6.9　condit.cpp

```
// condit.cpp -- using the conditional operator
#include <iostream>
int main()
{
   using namespace std;
   int a, b;
   cout << "Enter two integers: ";
   cin >> a >> b;
   cout << "The larger of " << a << " and " << b;
   int c = a > b ? a: b;   // c = a if a > b, else c = b
   cout << " is " << c << endl;
   return 0;
}
```

下面是该程序的运行情况：

```
Enter two numbers: 25 28
The larger of 25 and 28 is 28
```

该程序的关键部分是下面的语句：

```
int c = a > b ? a: b;
```

它与下面的语句等效：

```
int c;
if (a > b)
   c = a;
else
   c = b;
```

与 if else 序列相比，条件操作符更简洁，但第一次使用时不那么容易理解。这两种方法之间的区别是，条件操作符生成一个表达式，因此是一个值，可以将其赋给变量或将其放到一个更大的表达式中，程序清单 6.9 中的程序正是这样做的，它将条件表达式的值赋给变量 c。条件操作符格式简洁、句法奇特、外观与众不同，因此在欣赏这些特点的程序员中广受欢迎。其中一个技巧（它完成一个应被谴责的任务——隐藏代码）是将条件表达式嵌套在另一个条件表达式中，如下所示：

```
const char x[2] [20] = {"Jason ", "at your service\n"};
const char * y = "Quillstone ";

for (int i = 0; i < 3; i++)
   cout << ((i < 2) ? !i ? x [i]: y: x[1]);
```

这是一种费解的方式（但绝不是最难理解的），它按下面的顺序打印 3 个字符串：

```
Jason Quillstone at your service
```

从可读性来说，条件操作符最适合于简单关系和简单表达式的值：

```
x = (x > y) ? x: y;
```

当代码变得更复杂时，使用 if else 语句来表达可能更为清晰。

6.5 switch 语句

假设要创建一个屏幕菜单，要求用户从 5 种选项中选择一个，例如，便宜、适中、昂贵、奢侈与过度。虽然可以扩展 if else if else 序列来处理这 5 种情况，但 C++的 switch 语句能够更容易地从大型列表中进行选择。下面是 switch 语句的通用格式：

```
switch (integer-expression)
{
     case label1 :  statement (s)
     case label2 :  statement (s)
     ...
     default     :  statement (s)
}
```

C++的 switch 语句就像指路牌，告诉计算机接下来应执行哪行代码。执行到 switch 语句时，程序将跳到使用 integer-expression 的值标记的那一行。例如，如果 integer-expression 的值为 4，则程序将执行标签为 case 4：那一行。顾名思义，integer-expression 必须是一个结果为整数值的表达式。另外，每个标签都必须是整数常量表达式。最常见的标签是 int 或 char 常量（如 1 或'q'），也可以是枚举量。如果 integer-expression 不与任何标签匹配，则程序将跳到标签为 default 的那一行。Default 标签是可选的，如果被省略，而又没有匹配的标签，则程序将跳到 switch 后面的语句处执行（参见图 6.3）。

switch 语句与 Pascal 等语言中类似的语句之间存在重大的差别。C++中的 case 标签只是行标签，而不是选项之间的界线。也是说，程序跳到 switch 中特定代码行后，将依次执行之后的所有语句，除非有明确的其他指示。程序不会在执行到下一个 case 处自动停止，要让程序执行完一组特定语句后停止，必须使用 break 语句。这将导致程序跳到 switch 后面的语句处执行。

程序清单 6.10 演示了如何使用 switch 和 break 来让用户选择简单菜单。该程序使用 showmenu()函数显示一组选项，然后使用 switch 语句，根据用户的反应执行相应的操作。

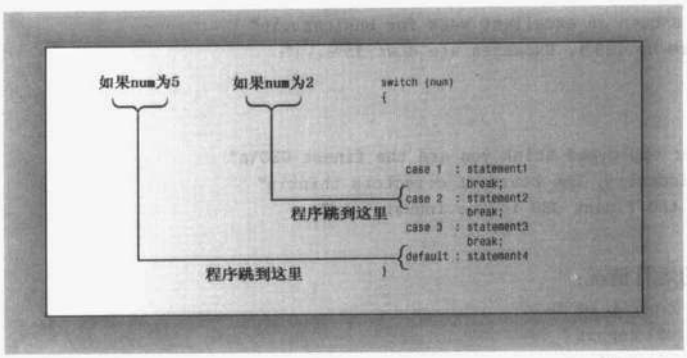


图 6.3　switch 语句的结构

注意：有些实现不会将（程序清单 6.10 的 case 1 中使用的）转义序列\a 解释为振铃。

程序清单 6.10　switch.cpp

```
// switch.cpp -- using the switch statement
#include <iostream>
using namespace std;
void showmenu();   // function prototypes
void report();
void comfort();
int main()
{
    showmenu();
    int choice;
    cin >> choice;
    while (choice != 5)
    {
        switch (choice)
        {
          case 1 :  cout << "\a\n";
                    break;
          case 2 :  report();
                    break;
          case 3 :  cout << "The boss was in all day.\n";
                    break;
          case 4 :  comfort();
                    break;
          default:  cout << "That's not a choice.\n";
        }
        showmenu();
        cin >> choice;
    }
    cout << "Bye!\n";
    return 0;
}

void showmenu()
{
    cout << "Please enter 1, 2, 3, 4, or 5: \n"
         "1) alarm          2) report\n"
         "3) alibi          4) comfort\n"
         "5) quit\n";
}
void report()
```

```
{
    cout << "It's been an excellent week for business.\n"
        "Sales are up 120%. Expenses are down 35%.\n";
}
void comfort()
{
    cout << "Your employees think you are the finest CEO\n"
        "in the industry. The board of directors think\n"
        "you are the finest CEO in the industry.\n";
}
```

下面是该程序的运行情况：

```
Please enter 1, 2, 3, 4, or 5:
1) alarm          2) report
3) alibi          4) comfort
5) quit
4
Your employees think you are the finest CEO
in the industry. The board of directors think
you are the finest CEO in the industry.
Please enter 1, 2, 3, 4, or 5:
1) alarm          2) report
3) alibi          4) comfort
5) quit
2
It's been an excellent week for business.
Sales are up 120%. Expenses are down 35%.
Please enter 1, 2, 3, 4, or 5:
1) alarm          2) report
3) alibi          4) comfort
5) quit
6
That's not a choice.
Please enter 1, 2, 3, 4, or 5:
1) alarm          2) report
3) alibi          4) comfort
5) quit
5
Bye!
```

当用户输入了 5 时，while 循环结束。输入 1 到 4 将执行 switch 列表中相应的操作，输入 6 将执行默认语句。

如前所述，该程序需要 break 语句来确保只执行 switch 语句中的特定部分。为检查情况是否如此，可以删除程序清单 6.10 中的 break 语句，然后看看其运行情况。例如，读者将发现，输入 2 后，将执行 case 标签为 2、3、4 和 defualt 中的所有语句。C++之所以这样，是由于这种行为很有用。例如，它使得使用多个标签很简单。例如，假设重新编写程序清单 6.10，使用字符（而不是整数）作为菜单选项和 switch 标签，则可以为大写标签和小写标签提供相同的语句：

```
char choice;
cin >> choice;
while (choice != 'Q' && choice != 'q')
{
    switch (choice)
    {
        case 'a':
        case 'A': cout << "\a\n";
                  break;
        case 'r':
        case 'R': report();
                  break;
```

```
        case 'l':
        case 'L': cout << "The boss was in all day.\n";
                  break;
        case 'c'
        case 'C': comfort();
                  break;
        default: cout << "That's not a choice.\n";
    }
    showmenu();
    cin >> choice;
}
```

由于 case 'a'后面没有 break 语句，因此程序将接着执行下一行——case 'A'后面的语句。

6.5.1　将枚举量用作标签

程序清单 6.11 使用 enum 定义了一组相关的常量，然后在 switch 语句中使用这些常量。通常，cin 无法识别枚举类型（它不知道程序员是如何定义它们的），因此该程序要求用户选择选项时输入一个整数。当 switch 语句将 int 值和枚举量标签进行比较时，将枚举量提升为 int。另外，在 while 循环测试条件中，也会将枚举量提升为 int 类型。

程序清单 6.11　enum.cpp

```
// enum.cpp -- using enum
#include <iostream>
// create named constants for 0 - 6
enum {red, orange, yellow, green, blue, violet, indigo};

int main()
{
    using namespace std;
    cout << "Enter color code (0-6): ";
    int code;
    cin >> code;
    while (code >= red && code <= indigo)
    {
        switch (code)
        {
          case red    : cout << "Her lips were red.\n"; break;
          case orange : cout << "Her hair was orange.\n"; break;
          case yellow : cout << "Her shoes were yellow.\n"; break;
          case green  : cout << "Her nails were green.\n"; break;
          case blue   : cout << "Her sweatsuit was blue.\n"; break;
          case violet : cout << "Her eyes were violet.\n"; break;
          case indigo : cout << "Her mood was indigo.\n"; break;
        }
        cout << "Enter color code (0-6): ";
        cin >> code;
    }
    cout << "Bye\n";
    return 0;
}
```

下面是该程序的输出：

```
Enter color code (0-6) : 3
Her nails were green.
Enter color code (0-6) : 5
Her eyes were violet.
Enter color code (0-6) : 2
Her shoes were yellow.
Enter color code (0-6) : 8
Bye
```

6.5.2 switch 和 if else

switch 语句和 if else 语句都允许程序从选项中进行选择。相比之下，if else 更通用。例如，它可以处理取值范围，如下所示：

```
if (age > 17 && age < 35)
   index = 0;
else if (age >= 35 && age < 50)
   index = 1;
else if (age >= 50 && age < 65)
   index = 2;
else
   index = 3;
```

然而，switch 并不是为处理取值范围而设计的。switch 语句中的每一个 case 标签都必须是一个单独的值。另外，这个值必须是整数（包括 char），因此 switch 无法处理浮点测试。另外 case 标签值还必须是常量。如果选项涉及到取值范围、浮点测试或两个变量的比较，则应使用 if else 语句。

不过，如果所有的选项都可以使用整数常量来标识，则可以使用 switch 语句或 if else 语句。由于 switch 语句是专门为这种情况设计的，因此，如果选项超过两个，则就代码长度和执行速度而言，switch 语句的效率更高。

提示：如果既可以使用 if else if 语句，也可以使用 switch 语句，则当选项不少于 3 个时，应使用 switch 语句

6.6 break 和 continue 语句

break 和 continue 语句都使程序能够跳过部分代码。可以在 switch 语句或任何循环中使用 break 语句，使程序跳到 switch 或循环后面的语句处执行。continue 语句用于循环中，让程序跳过循环体中余下的代码，并开始新一轮循环（参见图 6.4）。

程序清单 6.12 演示了这两条语句是如何工作的。该程序让用户输入一行文本。循环将回显每个字符，如果该字符为句点，则使用 break 结束循环。这表明，可以在某种条件为 true 时，使用 break 来结束循环。接下来，程序计算空格数，但不计算其他字符。当字符不为空格时，循环使用 continue 语句跳过计数部分。

程序清单 6.12 jump.cpp

```
// jump.cpp -- using continue and break
#include <iostream>
const int ArSize = 80;
int main()
{
   using namespace std;
   char line[ArSize];
   int spaces = 0;

   cout << "Enter a line of text: \n";
   cin.get (line, ArSize);
   cout << "Complete line:\n" << line << endl;
   cout << "Line through first period:\n";
   for (int i = 0; line[i] != '\0'; i++)
   {
      cout << line[i];    // display character
      if (line[i] == '.') // quit if it's a period
        break;
      if (line[i] != ' ') // skip rest of loop
        continue;
```

```
        spaces++;
    }
    cout << "\n" << spaces << " spaces\n";
    cout << "Done.\n";
    return 0;
}
```

下面是该程序的运行情况：

```
Enter a line of text:
Let's do lunch today. You can pay!
Complete line:
Let's do lunch today. You can pay!
Line through first period:
Let's do lunch today.
3 spaces
Done.
```

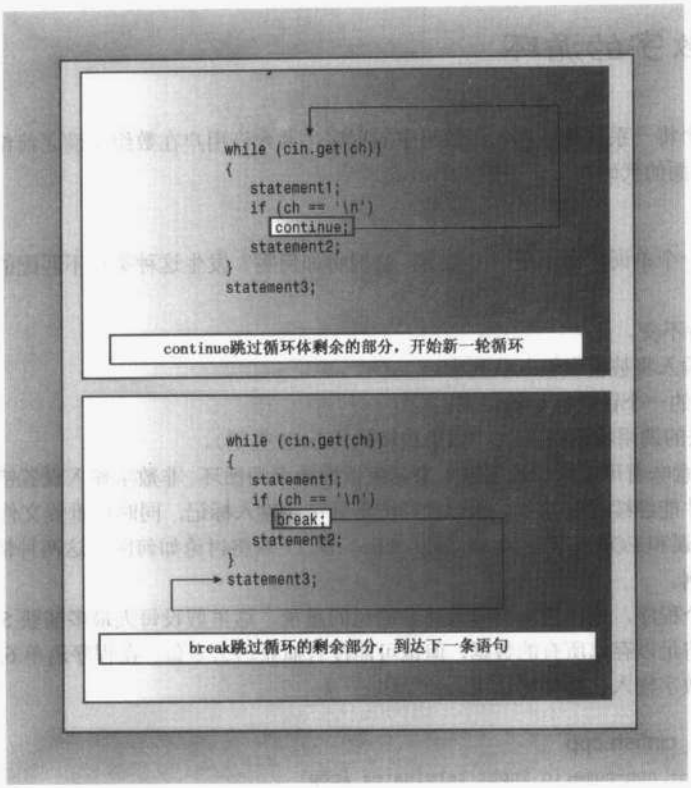


图 6.4　break 和 continue 语句的结构

程序说明

虽然 continue 语句导致该程序跳过循环体的剩余部分，但不会跳过循环的更新表达式。在 for 循环中，continue 语句使程序直接跳到更新表达式处，然后跳到测试表达式处。不过，对于 while 循环来说，continue 将使程序直接跳到测试表达式处，因此 while 循环体中位于 continue 之后的更新表达式都将被跳过。在某些情况下，这可能是一个问题。

该程序可以不使用 continue 语句，而使用下面的代码：

```
if (line[i] == ' ')
    spaces++;
```

不过，当 continue 之后有多条语句时，continue 语句可以提高程序的可读性。这样，就不必将所有这些语句放在 if 语句中。

和 C 语言一样，C++也有 goto 语句。下面的语句将跳到使用 paris: 作为标签的位置：

```
goto paris
```

也就是说，可以有下面这样的代码：

```
char ch;
cin >> ch;
if (ch == 'P')
    goto paris;
cout << ...
...
paris: cout << "You've just arrived at Paris.\n";
```

在大多数情况下（有些人认为，在任何情况下），使用 goto 语句不好，而应使用结构化控制语句（如 if else、switch、continue 等）来控制程序的流程。

6.7 读取数字的循环

假设要编写一个将一系列数字读入到数组中的程序，并允许用户在数组填满之前结束输入。一种方法是利用 cin。请看下面的代码：

```
int n;
cin >> n;
```

如果用户输入一个单词，而不是一个数字，情况将如何呢？发生这种类型不匹配的情况时，将发生 4 种情况：

- *n* 的值保持不变。
- 不匹配的输入将被留在输入队列中。
- cin 对象中的一个错误标记被设置。
- 对 cin 方法的调用将返回 false（如果被转换为 bool 类型）。

方法返回 false 意味着可以用非数字输入来结束读取数字的循环。非数字输入设置错误标记意味着必须重置该标记，程序才能继续读取输入。clear()方法重置错误输入标记，同时也重置文件尾（EOF 条件，参见第 5 章）。输入错误和 EOF 都将导致 cin 返回 false，第 17 章将讨论如何区分这两种情况。下面来看两个演示这些技术的范例。

假设要编写一个程序，来计算平均每天捕获的鱼的重量。这里假设每天最多捕获 5 条鱼，因此一个包含 5 个元素的数组将足以存储所有的数据，但也可能没有捕获这么多鱼。在程序清单 6.13 中，如果数组被填满或者输入了非数字输入，循环将结束。

程序清单 6.13 cinfish.cpp

```
// cinfish.cpp -- non-numeric input terminates loop
#include <iostream>
const int Max = 5;
int main()
{
   using namespace std;
// get data
   double fish[Max];
   cout << "Please enter the weights of your fish.\n";
   cout << "You may enter up to " << Max
        << " fish <q to terminate>.\n";
   cout << "fish #1: ";
   int i = 0;
   while (i < Max && cin >> fish[i]) {
```

```
        if (++i < Max)
            cout << "fish #" << i+1 << ": ";
    }
// calculate average
    double total = 0.0;
    for (int j = 0; j < i; j++)
        total += fish[j];
// report results
    if (i == 0)
        cout << "No fish\n";
    else
        cout << total / i << " = average weight of "
            << i << " fish\n";
    cout << "Done.\n";
    return 0;
}
```

注意：对于下面的代码，一些早期的 Borland 编译器将会给出警告：

```
cout << "fish #" << i+1 << ": ";
```

指出对于二义性操作符，必须使用括号。不过不必担心。它们只是警告——如果<<按其最初的含义使用（左移操作符），将出现分组错误。

程序清单 6.13 中的表达式 cin>>fish [i]实际上一个是 cin 方法函数调用，该函数返回 cin。如果 cin 位于测试条件中，则将被转换为 bool 类型。如果输入成功，则转换后的值为 true，否则为 false。如果表达式的值为 false，则循环结束。下面是该程序的运行情况：

```
Please enter the weights of your fish.
You may enter up to 5 fish <q to terminate>.
fish #1: 30
fish #2: 35
fish #3: 25
fish #4: 40
fish #5: q
32.5 = average weight of 4 fish
Done.
```

请注意下面的代码行：

```
while (i < Max && cin >> fish[i]) {
```

前面讲过，如果逻辑 AND 表达式的左侧为 false，则 C++将不会判断右侧的内容。在这种情况下，对右侧的内容进行判定意味着用 cin 将输入放到数组中。如果 i 等于 Max，则循环将结束，而不会将一个值读入到数组后面的位置中。

当用户输入的不是数字时，该程序将不再读取输入。下面来看一个继续读取的例子。假设程序要求用户提供 5 个高尔夫得分，以计算平均成绩。如果用户输入非数字输入，程序将拒绝，并要求用户继续输入数字。可以看到，可以使用 cin 输入表达式的值来检测输入是不是数字。程序发现用户输入了错误内容时，应采取 3 个步骤：

1. 重置 cin 以接受新的输入。
2. 删除错误输入。
3. 提示用户再输入。

请注意，程序必须先重置 cin，然后才能删除错误输入。程序清单 6.14 演示了如何完成这些工作。

程序清单 6.14　cingolf.cpp

```
// cingolf.cpp -- non-numeric input skipped
#include <iostream>
const int Max = 5;
int main()
{
    using namespace std;
```

```
// get data
   int golf[Max];
   cout << "Please enter your golf scores.\n";
   cout << "You must enter " << Max << " rounds.\n";
   int i;
   for (i = 0; i < Max; i++)
   {
      cout << "round #" << i+1 << ": ";
      while (!(cin >> golf[i])) {
         cin.clear();    // reset input
         while (cin.get()!= '\n')
            continue;    // get rid of bad input
         cout << "Please enter a number: ";
      }
   }
// calculate average
   double total = 0.0;
   for (i = 0; i < Max; i++)
      total += golf[i];
// report results
   cout << total / Max << " = average score "
        << Max << " rounds\n";
   return 0;
}
```

注意：对于下面的代码，一些早期的 Borland 编译器将会给出警告：

```
cout << "round #" << i+1 << ": ";
```

指出对于二义性操作符，必须使用括号。不过不必担心。它们只是警告——如果<<按其最初的含义使用（左移操作符），将出现分组错误。

下面是该程序的运行情况：

```
Please enter your golf scores.
You must enter 5 rounds.
round #1: 88
round #2: 87
round #3: must i?
Please enter a number: 103
round #4: 94
round #5: 86
91.6 = average score 5 rounds
```

程序说明

在程序清单 6.14 中，错误处理代码的关键部分如下：

```
while (!(cin >> golf[i])) {
   cin.clear();    // reset input
   while (cin.get()!= '\n')
      continue; // get rid of bad input
   cout << "Please enter a number: ";

}
```

如果用户输入 88，则 cin 表达式将为 true，因此将一个值被放到数组中；而表达式!（cin >> golf [i]）为 false，因此结束内部循环。但是，如果用户输入 must i?，则 cin 表达式将为 false，因此不会将任何值放到数组中；而表达式!（cin >> golf [i]）将为 true，因此进入内部的 while 循环。该循环的第一条语句使用 clear()方法重置输入，如果省略这条语句，程序将拒绝继续读取输入。接下来，程序在 while 循环中使用 cin.get()来读取行尾之前的所有输入，从而删除这一行中的错误输入。另一种方法是读取到下一个空白字符，这样将每次删除一个单词，而不是一次删除整行。最后，程序告诉用户，应输入一个数字。

6.8　简单文件输入/输出

有时候，通过键盘输入并非最好的选择。例如，假设您编写了一个股票分析程序，并下载了一个包含1000 种股票价格的文件。在这种情况下，让程序直接读取文件，而不是手工输入文件中所有的值，将方便得多。同样，让程序将输出写入到文件将更为方便，这样可得到有关结果的永久性记录。

幸运的是，C++使得将读取键盘输入和在屏幕上显示输出（统称为控制台输入/输出）的技巧用于文件输入/输出（文件 I/O）非常简单。第 17 章将更详细地讨论这些主题，这里只介绍简单的文本文件 I/O。

6.8.1　文本 I/O 和文本文件

这里再来介绍一些文本 I/O 的概念。使用 cin 进行输入时，程序将输入视为一系列的字节，其中每个字节都被解释为字符编码。不管目标数据类型是什么，输入一开始都是字符数据——文本数据。然后，cin 对象负责将文本转换为其他类型。为说明这是如何完成的，来看一些处理同一个输入行的代码。

假设有如下范例输入行：

```
38.5 19.2
```

来看一下使用不同数据类型的变量进行存储时，cin 是如何处理该输入行的。首先，来看使用 char 数据类型的情况：

```
char ch;
cin >> ch;
```

输入行中的第一个字符被赋给 ch。在这里，第一个字符是数字 3，其字符编码（二进制）被存储在变量 ch 中。输入和目标变量都是字符，因此不需要进行转换。注意，这里存储的不是数值 3，而是字符 3 的编码。执行上述输入语句后，输入队列中的下一个字符为字符 8，下一个输入操作将对其进行处理。

接下来看看 int 类型：

```
int n;
cin >> n;
```

在这种情况下，cin 将不断读取，直到遇到非数字字符。也就是说，它将读取 3 和 8，这样句点将成为输入队列中的下一个字符。cin 通过计算发现，这两个字符对应数值 38，因此将 38 的二进制编码复制到变量 *n* 中。

接下来看看 double 类型：

```
double x;
cin >> x;
```

在这种情况下，cin 将不断读取，直到遇到第一个不属于浮点数的字符。也就是说，cin 读取 3、8、句点和 5，使得空格成为输入队列中的下一个字符。cin 通过计算发现，这 4 个字符对应于数值 38.5，因此将 38.5 的二进制编码（浮点格式）复制到变量 x 中。

接下来看看 char 数组的情况：

```
char word[50];
cin >> word;
```

在这种情况下，cin 将不断读取，直到遇到空白字符。也就是说，它读取 3、8、句点和 5，使得空格成为输入队列中的下一个字符。然后，cin 将这 4 个字符的字符编码存储到数组 word 中，并在末尾加上一个空字符。这里不需要进行任何转换。

最后，来看一下另一种使用 char 数组来存储输入的情况：

```
char word[50];
cin.geline(word,50);
```

在这种情况下，cin 将不断读取，直到遇到换行符。所有字符都将被存储到数组 word 中，并在末尾加上一个空字符。换行符被丢弃，输入队列中的下一个字符是下一行中的第一个字符。这里不需要进行任何转换。

对于输入，将执行相反的转换。即整数被转换为数字字符序列，浮点数被转换为数字字符和其他字符组成的字符序列（如 284.53 或-1.587E+06）。字符数据不需要做任何转换。

这里的要点是，输入一开始为文本。因此，控制台输入的文件版本是文本文件，即每个字节都存储了一个字符编码的文件。并非所有的文件都是文本文件，例如，数据库和电子表格以数值格式（即二进制整数或浮点格式）来存储数值数据。另外，字处理文件中可能包含文本信息，但也可能包含用于描述格式、字体、打印机等的非文本数据。

本章讨论的文件 I/O 相当于控制台 I/O，因此仅适用于文本文件。要创建文本文件，用于提供输入，可使用文本编译器，如 DOS 中的 EDIT、Windows 中的“记事本”和 UNIX/Linux 系统中的 vi 或 emacs。也可以使用字处理程序来创建，但必须将文件保存为文本格式。IDE 中的源代码编辑器生成的也是文本文件，事实上，源代码文件就属于文本文件。同样，可以使用文本编辑器来查看通过文本输出创建的文件。

6.8.2 写入到文本文件中

对于文件输入，C++使用类似于 cout 的东西。下面来复习一些有关将 cout 用于控制台输出的基本事实，为文件输入做准备：

- 必须包含头文件 iostream。
- 头文件 iostream 定义了一个用处理输出的 ostream 类。
- 头文件 iostream 声明了一个名为 cout 的 ostream 变量（对象）。
- 必须指明名称空间 std；例如，为引用元素 cout 和 endl，必须使用编译指令 using 或前缀 std::。
- 可以结合使用 cout 和操作符>>来显示各种类型的数据。

文件输出与此极其相似：

- 必须包含头文件 fstream。
- 头文件 fstream 定义了一个用于处理输出的 ofstream 类。
- 需要声明一个或多个 ofstream 变量（对象），并以自己喜欢的方式对其进行命名，条件是遵守常用的命名规则。
- 必须指明名称空间 std；例如，为引用元素 ofstream，必须使用编译指令 using 或前缀 std::。
- 需要将 ofstream 对象与文件关联起来。为此，方法之一是使用 open()方法。
- 使用完文件后，应使用 close()方法将其关闭。
- 可结合使用 ofstream 对象和操作符>>来输出各种类型的数据。

注意，虽然头文件 iostream 提供了一个预先定义好的名为 cout 的 ostream 对象，但您必须声明自己的 ofstream 对象，为其命名，并将其同文件关联起来。下面演示了如何声明这种对象：

```
ofstream outFile;          // outFile an ofstream object
ofstream fout;             // fout an ofstream object
```

下面演示了如何将这种对象与特定的文件关联起来：

```
outFile.open("fish.txt");   // outFile used to write to the fish.txt file
char filename[50];
cin >> filename;            // user specifies a name
fout.open(filename);        // fout used to read specified file
```

注意，方法 open()接受一个 C-风格字符串作为参数，这可以是一个字面字符串，也可以是存储在数组中的字符串。

下面演示了如何使用这种对象：

```
double wt = 125.8;
outFile << wt;         // write a number to fish.txt
char line[81] = "Objects are closer than they appear.";
fin << line << endl;   // write a line of text
```

重要的是，声明一个 ofstream 对象并将其同文件关联起来后，便可以像使用 cout 那样使用它。所有可用于 cout 的操作和方法（如<<、endl 和 setf()）都可用于 ofstream 对象（如前述范例中的 outFile 和 fout）。

总之，使用文件输出的主要步骤如下：

1. 包含头文件 fstream。
2. 创建一个 ofstream 对象。
3. 将该 ofstream 对象同一个文件关联起来。
4. 就像使用 cout 那样使用该 ofstream 对象。

程序清单 6.15 中的程序演示了这种方法。它要求用户输入信息，然后将信息显示到屏幕上，再将这些信息写入到文件中。读者可以使用文本编辑器来查看该输出文件的内容。

程序清单 6.15　outfile.cpp

```
// outfile.cpp -- writing to a file
#include <iostream>
#include <fstream>              // for file I/O

int main()
{
   using namespace std;

   char automobile[50];
   int year;
   double a_price;
   double d_price;

   ofstream outFile;            // create object for output
   outFile.open("carinfo.txt");   // associate with a file

   cout << "Enter the make and model of automobile: ";
   cin.getline(automobile, 50);
   cout << "Enter the model year: ";
   cin >> year;
   cout << "Enter the original asking price: ";
   cin >> a_price;
   d_price = 0.913 * a_price;

// display information on screen with cout

   cout << fixed;
   cout.precision(2);
   cout.setf(ios_base::showpoint);
   cout << "Make and model: " << automobile << endl;
   cout << "Year: " << year << endl;
   cout << "Was asking $" << a_price << endl;
   cout << "Now asking $" << d_price << endl;

// now do exact same things using outFile instead of cout

   outFile << fixed;
   outFile.precision(2);
   outFile.setf(ios_base::showpoint);
   outFile << "Make and model: " << automobile << endl;
   outFile << "Year: " << year << endl;
   outFile << "Was asking $" << a_price << endl;
   outFile << "Now asking $" << d_price << endl;

   outFile.close();             // done with file
   return 0;
}
```

该程序的最后一部分与 cout 部分相同，只是将 cout 替换为 outFile 而已。下面是该程序的运行情况：

```
Enter the make and model of automobile: Flitz Pinata
Enter the model year: 2001
Enter the original asking price: 28576
Make and model: Flitz Pinata
Year: 2001
Was asking $28576.00
Now asking $26089.89
```

屏幕输出是使用 cout 的结果。如果您查看该程序的可执行文件所在的目录，将看到一个名为 carinfo.txt 的新文件，其中包含使用 outFile 生成的输出。如果使用文本编辑器打开该文件，将发现其内容如下：

```
Make and model: Flitz Pinata
Year: 2001
Was asking $28576.00
Now asking $26089.89
```

正如读者看到的，outFile 将 cout 显示到屏幕上的内容写入到了文件 carinfo.txt 中。

程序说明

在程序清单 6.15 的程序中，声明一个 ofstream 对象后，便可以使用方法 open() 来将该对象同一个特定的文件关联起来：

```
ofstream outFile;              // create object for output
outFile.open("carinfo.txt");   // associate with a file
```

程序使用完该文件后，应该将其关闭：

```
outFile.close();
```

注意，方法 close() 不需要使用文件名作为参数，这是因为 outFile 已经同特定的文件关联起来了。如果您忘记关闭文件，程序正常终止时将自动关闭它。

outFile 可使用 cout 可使用的任何方法。它不但能够使用操作符<<，还可以使用各种格式化方法，如 setf() 和 precision()。这些方法只影响调用它们的对象。例如，对于不同的对象，可以提供不同的值：

```
cout.precision(2);      // use a precision of 2 for the display
outFile.precision(4);   // use a precision of 4 for file output
```

读者需要记住的重点是，创建好 ofstream 对象（如 outFile）后，便可以像使用 cout 那样使用它。

回到 open() 方法：

```
outFile.open("carinfo.txt");
```

在这里，该程序运行之前，文件 carinfo.txt 并不存在。在这种情况下，方法 open() 将新建一个名为 carinfo.txt 的文件。如果在此运行该程序，文件 carinfo.txt 将存在，此时情况将如何呢？默认情况下，open() 将首先截断该文件，即将其长度截短到零——丢其原有的内容，然后将新的输出加入到该文件中。第 17 章将介绍如何修改这种默认行为。

警告：打开已有的文件，以接受输出时，默认将它其长度截短为零，因此原来的内容将丢失。

打开文件用于接受输入时可能失败。例如，指定的文件可能已经存在，但禁止对其进行访问。因此细心的程序员将检查打开文件的操作是否成功，这将在下一个例子中介绍。

6.8.3 读取文本文件

接下来介绍文本文件输入，它是基于控制台输入的，后者涉及多个方面。我们首先来总结这些方面：

- 必须包含头文件 iostream。
- 头文件 iostream 定义了一个用处理输入的 istream 类。
- 头文件 iostream 声明了一个名为 cin 的 istream 变量（对象）。
- 必须指明名称空间 std；例如，为引用元素 cin，必须使用编译指令 using 或前缀 std::。
- 可以结合使用 cin 和操作符>>来读取各种类型的数据。
- 可以使用 cin 和 get() 方法来读取一个字符，使用 cin 和 getline() 来读取一行字符。
- 可以结合使用 cin 和 eof()、fail() 方法来判断输入是否成功。
- 对象 cin 本身被用作测试条件时，如果最后一个读取操作成功，它将被转换为布尔值 true，否则

被转换为 false。

文件输出与此极其相似：

- 必须包含头文件 fstream。
- 头文件 fstream 定义了一个用于处理输入的 ifstream 类。
- 需要声明一个或多个 ifstream 变量（对象），并以自己喜欢的方式对其进行命名，条件是遵守常用的命名规则。
- 必须指明名称空间 std；例如，为引用元素 ifstream，必须使用编译指令 using 或前缀 std::。
- 需要将 ifstream 对象与文件关联起来。为此，方法之一是使用 open() 方法。
- 使用完文件后，应使用 close() 方法将其关闭。
- 可结合使用 ifstream 对象和操作符<<来读取各种类型的数据。
- 可以使用 ifstream 对象和 get() 方法来读取一个字符，使用 ifstream 对象和 getline() 来读取一行字符。
- 可以结合使用 ifstream 和 eof()、fail() 等方法来判断输入是否成功。
- ifstream 对象本身被用作测试条件时，如果最后一个读取操作成功，它将被转换为布尔值 true，否则被转换为 false。

注意，虽然头文件 iostream 提供了一个预先定义好的名为 cin 的 istream 对象，但您必须声明自己的 ifstream 对象，为其命名，并将其同文件关联起来。下面演示了如何声明这种对象：

```
ifstream inFile;          // inFile an ifstream object
ifstream fin;             // fin an ifstream object
```

下面演示了如何将这种对象与特定的文件关联起来：

```
inFile.open("bowling.txt");  // inFile used to read bowling.txt file
char filename[50];
cin >> filename;             // user specifies a name
fin.open(filename);          // fin used to read specified file
```

注意，方法 open() 接受一个 C-风格字符串作为参数，这可以是一个字面字符串，也可以是存储在数组中的字符串。

下面演示了如何使用这种对象：

```
double wt;
inFile >> wt;          // read a number from bowling.txt
char line[81];
fin.getline(line, 81); // read a line of text
```

重要的是，声明一个 ifstream 对象并将其同文件关联起来后，便可以像使用 cin 那样使用它。所有可用于 cin 的操作和方法都可用于 ifstream 对象（如前述范例中的 inFile 和 fin）。

如果试图打开一个不存在的文件用于输入，情况将如何呢？这种错误将导致后面使用 ifstream 对象进行输入时失败。检查文件是否被成功打开的首先方法是使用方法 is_open()，为此，可以使用类似于下面的代码：

```
inFile.open("bowling.txt");
if (!inFile.is_open())
{
   exit(EXIT_FAILURE);
}
```

如果文件被成功地打开，方法 is_open() 将返回 true；因此如果文件没有被打开，表达式!inFile.isopen() 将为 true。函数 exit() 的原型是在头文件 cstdlib 中定义的，在该头文件中，还定义了一个用于同操作系统通信的参数值 EXIT_FAILURE。函数 exit() 终止程序。

方法 is_open() 是 C++中相对较新的内容。如果读者的编译器不支持它，可使用较老的方法 good() 来代替。正如第 17 章将讨论的，方法 good() 在检查可能存在的问题方面，没有 is_open() 那么广泛。

程序清单 6.16 中的程序打开用户指定的文件，读取其中的数字，然后指出文件中包含多少个值以及它们的和与平均值。正确地设计输入循环至关重要，详细请参阅接下来的“程序说明”。注意，该程序大量地使用了 if 语句。

程序清单 6.16 sumafile.cpp

```
// sumafile.cpp -- functions with an array argument
#include <iostream>
#include <fstream>          // file I/O support
#include <cstdlib>          // support for exit()
const int SIZE = 60;
int main()
{
    using namespace std;
    char filename[SIZE];
    ifstream inFile;        // object for handling file input

    cout << "Enter name of data file: ";
    cin.getline(filename, SIZE);
    inFile.open(filename);  // associate inFile with a file
    if (!inFile.is_open())  // failed to open file
    {
        cout << "Could not open the file " << filename << endl;
        cout << "Program terminating.\n";
        exit(EXIT_FAILURE);
    }
    double value;
    double sum = 0.0;
    int count = 0;          // number of items read

    inFile >> value;        // get first value
    while (inFile.good())   // while input good and not at EOF
    {
        ++count;            // one more item read
        sum += value;       // calculate running total
        inFile >> value;    // get next value
    }
    if (inFile.eof())
        cout << "End of file reached.\n";
    else if (inFile.fail())
        cout << "Input terminated by data mismatch.\n";
    else
        cout << "Input terminated for unknown reason.\n";
    if (count == 0)
        cout << "No data processed.\n";
    else
    {
        cout << "Items read: " << count << endl;
        cout << "Sum: " << sum << endl;
        cout << "Average: " << sum / count << endl;
    }
    inFile.close();         // finished with the file
    return 0;
}
```

注意：有些老式编译器不支持方法 is_open()。对于这些编译器，可以将下面的代码：

```
if (!inFile.is_open())
```

替换为：

```
if (!inFile.good())
```

后者在检查程序是否成功地打开了文件方面没有前者严格。

要运行程序清单 6.16 中的程序，首先必须创建一个包含数字的文本文件。为此，可以使用文本编辑器（如用于编写源代码的文本编辑器）。假设该文件名为 scores.txt，包含的内容如下：

```
18 19 18.5 13.5 14
```

```
16 19.5 20 18 12 18.5
17.5
```

警告：对于 DOS/Windows 文本文件，每行的末尾必须有一个换行符。有些文本编辑器，如 Metrowerks CodeWarrior IDE 编辑器，不会自动在最后一行末尾加上换行符。因此，如果读者使用的是这种编辑器，请在输入最后的文本后按下回车键，然后再保存文件。

下面是该程序的运行情况：

```
Enter name of data file: scores.txt
End of file reached.
Items read: 12
Sum: 204.5
Average: 17.0417
```

程序说明

该程序没有使用硬编码文件名，而是将用户提供的文件名存储到字符数组 filename 中，然后将该数组用作 open() 的参数：

```
inFile.open(filename);
```

正如本章前面讨论的，检查文件是否被成功打开至关重要。下面是一些可能出问题的地方：指定的文件可能不存在；文件可能位于另一个目录（文件夹）中；访问可能被拒绝；用户可能输错了文件名或省略了文件扩展名。很多初学者花了大量的时间检查文件读取循环的哪里出了问题后，最终却发现问题在于程序没有打开文件。检查文件是否被成功打开可避免将这种精力放在错误地方的情况发生。

读者需要特别注意的是文件读取循环的正确设计。读取文件时，有几点需要检查。首先，程序读取文件时不应超过 EOF。如果最后一次读取数据时遇到 EOF，方法 eof() 将返回 true。其次，程序可能遇到类型不匹配的情况。例如，程序清单 6.16 期望文件中只包含数字。如果最后一次读取操作中发生了类型不匹配的情况，方法 fail() 将返回 true（如果遇到了 EOF，该方法也将返回 true）。最后，某些预期外的东西可能出现问题，例如文件受损或硬件故障。如果最后一次读取文件时发生了这样的问题，方法 bad() 将返回 true。不要分别检查这些情况，一种更简单的方法是使用 good() 方法，该方法在没有发生任何错误时返回 true：

```
while (inFile.good())  // while input good and not at EOF
{
   ...
}
```

然后，如果愿意，可以使用其他方法来确定循环终止的真正原因：

```
if (inFile.eof())
   cout << "End of file reached.\n";
else if (inFile.fail())
   cout << "Input terminated by data mismatch.\n";
else
   cout << "Input terminated for unknown reason.\n";
```

这些代码紧跟在循环的后面，用于判断循环为何终止。由于 eof() 只能判断是否到达 EOF，而 fail() 可用于检查 EOF 和类型不匹配，因此上述代码首先判断是否到达 EOF。这样，如果执行到了 else if 测试，便可排除 EOF，因此，如果 fail() 返回 true，便可断定导致循环终止的原因是类型不匹配。

方法 goo() 指出最后一次读取输入的操作是否成功，这一点至关重要。这意味着应该在执行读取输入的操作后，立刻应用这种测试。为此，一种标准方法是，在循环之前（首次执行循环测试前）放置一条输入语句，并在循环的末尾（下次执行循环测试之前）放置另一条输入语句：

```
// standard file-reading loop design
inFile >> value;       // get first value
while (inFile.good())  // while input good and not at EOF
{
   // loop body goes here
   inFile >> value;    // get next value
}
```

鉴于以下事实，可以对上述代码进行精简：表达式 inFile >> value 的结果为 inFile，而在需要一个 bool 值的情况下，inFile 的结果为 inFile.goo()，即 true 或 false。因此，可以将两条输入语句用一条用作循环测试的输入语句代替。也就是说，可以将上述循环结构替换为如下循环结构：

```
// abbreviated file-reading loop design
// omit pre-loop input
while (inFile >> value)   // read and test for success
{
   // loop body goes here
   // omit end-of-loop input
}
```

这种设计仍然遵循了在测试之前进行读取的规则，因为要计算表达式 inFile >> value 的值，程序必须首先试图将一个数字读取到 value 中。

至此，读者对文件 I/O 有了初步的认识。

6.9 总结

使用引导程序选择不同操作的语句后，程序和编程将更有趣（这是否也能引起程序员们的兴趣，我没有做过全面研究）。C++提供了 if 语句、if else 语句和 switch 语句来管理选项。if 语句使程序有条件地执行语句或语句块，也就是说，如果满足特定的条件，程序将执行特定的语句或语句块。if else 语句程序选择执行两个语句或语句块之一。可以在这条语句后再加上 if else，以提供一系列的选项。switch 语句引导程序执行一系列选项之一。

C++还提供了帮助决策的操作符。第 5 章讨论了关系表达式，这种表达式对两个值进行比较。if 和 if else 语句通常使用关系表达式作为测试条件。通过使用逻辑操作符（&&、||和!），可以组合或修改关系表达式，创建更细致的测试。条件操作符（?:）提供了一种选择两个值之一的简洁方式。

cctype 字符函数库提供了一组方便的、功能强大的工具，可用于分析字符输入。

对于文件 I/O 来说，循环和选择语句是很有用的工具：文件 I/O 与控制台 I/O 极其相似。声明 ifstream 和 ofstream 对象，并将它们同文件关联起来后，便可以像使用 cin 和 cout 那样使用这些对象。

使用循环和决策语句，便可以编写有趣的、智能的、功能强大的程序。不过我们刚开始涉足 C++的强大功能，接下来将介绍函数。

6.10 复习题

1. 请看下面两个计算空格和换行符数目的代码片段：

```
// Version 1
while (cin.get (ch))     // quit on eof
{
     if (ch == ' ')
          spaces++;
     if (ch == '\n')
          newlines++;
}
// Version 2
while (cin.get (ch))     // quit on eof
{
     if (ch == ' ')
          spaces++;
     else if (ch == '\n')
          newlines++;
}
```

第二种格式比第一种格式好在哪里呢？

2．在程序清单 6.2 中，用 ch+1 替换++ch 将发生什么情况呢？

3．请认真考虑下面的程序：

```
#include <iostream>
using namespace std;
int main()
{
    char ch;
    int ct1, ct2;

    ct1 = ct2 = 0;
    while ((ch = cin.get()) != '$')
    {
        cout << ch;
        ct1++;
        if (ch = '$')
            ct2++;
        cout << ch;
    }
    cout <<"ct1 = " << ct1 << ", ct2 = " << ct2 << "\n";
    return 0;
}
```

假设输入如下，其中→表示回车：

```
Hi! →
Send $10 or $20 now! →
```

则输出将是什么（还记得吗，输入被缓冲）？

4．创建表示下述条件的逻辑表达式：

a．weight 大于或等于 115，但小于 125。

b．ch 为 q 或 Q。

c．x 为偶数，但不是 26。

d．x 为偶数，但不是 26 的倍数。

e．donation 位于 1000～2000 中或 guest 为 1。

f．ch 是小写字母或者大写字母（假设小写字母是依次编码的，大写字母也是依次编码的，但在大小写字母间编码不是连续的）。

5．在英语中，“I will not not speak（我不会不说）”的意思与“I will speak（我要说）”相同。在 C++中，!!x 是否与 x 相同呢？

6．创建一个条件表达式，其值为变量的绝对值。也是说，如果变量 x 为正，则表达式的值为 x；但如果 x 为负，则表达式的值为-x——这是一个正值。

7．用 switch 改写下面的代码片段：

```
if (ch == 'A')
    a_grade++;
else if (ch == 'B')
    b_grade++;
else if (ch == 'C')
    c_grade++;
else if (ch == 'D')
    d_grade++;
else
    f_grade++;
```

8．对于程序清单 6.10，与使用数字相比，使用字符（如 a 和 c）表示菜单选项和 case 标签有何优点呢？（提示：想想用户输入 q 和输入 5 的情况。）

9．请看下面的代码片段：

```
int line = 0;
```

```
char ch;
while (cin.get (ch))
{
    if (ch == 'Q')
        break;
    if (ch != '\n')
        continue;
    line++;
}
```

请重写该代码片段，不要使用 break 和 continue 语句。

6.11 编程练习

1. 编写一个程序，读取键盘输入，直到遇到@符号为止，并回显输入（数字除外），同时将大写字符转换为小写，将小写字符转换为大写（别忘了 cctype 函数系列）。

2. 编写一个程序，最多将 10 个 donation 值读入到一个 double 数组中。程序遇到非数字输入时将结束输入，并报告这些数字的平均值以及数组中有多少个数字大于平均值。

3. 编写一个菜单驱动程序的雏形。该程序显示一个提供 4 个选项的菜单——每个选项用一个字母标记。如果用户使用有效选项之外的字母进行响应，程序将提示用户输入一个有效的字母，直到用户这样做为止。然后，该程序使用一条 switch 语句，根据用户的选择执行一个简单操作。该程序的运行情况如下：

```
Please enter one of the following choices:
c) carnivore          p) pianist
t) tree               g) game
f
Please enter a c, p, t, or g: q
Please enter a c, p, t, or g: t
A maple is a tree.
```

4. 加入 Benevolent Order of Programmer 后，在 BOP 大会上，人们便可以通过加入者的真实姓名、头衔或秘密 BOP 姓名来了解他（她）。请编写一个程序，可以使用真实姓名、头衔、秘密姓名或成员偏好来列出成员。编写该程序时，请使用下面的结构：

```
// Benevolent Order of Programmers name structure
struct bop {
    char fullname[strsize];    // real name
    char title[strsize];       // job title
    char bopname[strsize];     // secret BOP name
    int preference;            // 0 = fullname, 1 = title, 2 = bopname
};
```

该程序创建一个由上述结构组成的小型数组，并将其初始化为适当的值。另外，该程序使用一个循环，让用户在下面的选项中进行选择：

```
a. display by name     b. display by title
c. display by bopname  d. display by preference
q. quit
```

注意，“display by preference”并不意味着显示成员的偏好，而是意味着根据成员的偏好来列出成员。例如，如果偏好号为 1，则选择 d 将显示程序员的头衔。该程序的运行情况如下：

```
Benevolent Order of Programmers Report
a. display by name     b. display by title
c. display by bopname  d. display by preference
q. quit
Enter your choice: a
Wimp Macho
Raki Rhodes
```

```
Celia Laiter
Hoppy Hipman
Pat Hand
Next choice: d
Wimp Macho
Junior Programmer
MIPS
Analyst Trainee
LOOPY
Next choice: q
Bye!
```

5. 在 Neutronia 王国，货币单位是 tvarp，收入所得税的计算方式如下：

5000 tvarps：不收税

5001-15000 tvarps：10%

15001-35000 tvarps：15%

35000 tvarps 以上：20%

例如，某人的收入为 38000 tvarps，则其所得税为 5000 × 0.00 + 10000 × 0.10 + 20000 × 0.15 + 3000 × 0.20，即 4600 tvarps。请编写一个程序，使用循环来要求用户输入收入，并报告所得税。当用户输入负数或非数字时，循环将结束。

6. 编写一个程序，记录捐助给“维护合法权利团体”的钱。该程序要求用户输入捐献者数目，然后要求用户输入每一个捐献者的姓名和款项。这些信息被储存在一个动态分配的结构数组中。每个结构有两个成员：用来存储姓名的字符数组（或 string 对象）和用来存储款项的 double 成员。读取所有的数据后，程序将显示所有捐款超过 10000 的捐款者的姓名及其捐款数额。该列表前应包含一个标题，指出下面的捐款者是重要捐款人（Grand Patrons）。然后，程序将列出其他的捐款者，该列表要以 Patrons 开头。如果某种类别没有捐款者，则程序将打印单词“none”。该程序只显示这两种类别，而不进行排序。

7. 编写一个程序，它每次读取一个单词，直到用户只输入 q。然后，该程序指出有多少个单词以元音打头，有多少个单词以辅音打头，还有多少个单词不属于这两类。为此，方法之一是，使用 isalpha() 来区分以字母和其他字符打头的单词，然后对于通过了 isalpha() 测试的单词，使用 if 或 switch 语句来确定哪些以元音打头。该程序的运行情况如下：

```
Enter words (q to quit):
The 12 awesome oxen ambled
quietly across 15 meters of lawn. q
5 words beginning with vowels
4 words beginning with consonants
2 others
```

8. 编写一个程序，它打开一个文本文件，逐个字符地读取该文件，直到到达文件末尾，然后指出该文件中包含多少个字符。

9. 完成编程练习 6，但从文件中读取所需的信息。该文件的第一项应为捐款人数，余下的内容应为成对的行。在每一对中，第一行为捐款人姓名，第二行为捐款数额。即该文件应类似于下面所示：

```
4
Sam Stone
2000
Freida Flass
100500
Tammy Tubbs
5000
Rich Raptor
55000
```

第 7 章　函数——C++的编程模块

本章内容包括：

- 函数基本知识。
- 函数原型。
- 如何按值传递函数参数。
- 如何设计处理数组的函数。
- 如何使用 const 指针参数。
- 如何设计处理文本字符串的函数。
- 如何设计处理结构的函数。
- 如何设计处理 string 类对象的函数。
- 调用自身的函数（递归）。
- 指向函数的指针。

乐趣在于发现。仔细研究，读者将在函数中找到乐趣。C++自带了一个包含函数的大型库（标准 ANSI C 库加上多个 C++类），但真正的编程乐趣在于编写自己的函数。本章和第 8 章将介绍如何定义函数、给函数传递信息以及从函数那里获得信息。本章首先复习函数是如何工作的，然后着重介绍如何使用函数来处理数组、字符串和结构，最后介绍递归和函数指针。如果读者熟悉 C 语言，将发现本章的很多内容是熟悉的。但是，不要因此而掉以轻心，产生错误认识。在函数方面，C++在 C 语言的基础上新增了一些功能，这将在第 8 章介绍。现在，我们把注意力放在基础知识上。

7.1　复习函数的基本知识

来复习一下介绍过的有关函数的知识。要使用 C++函数，必须完成如下工作：

- 提供函数定义。
- 提供函数原型。
- 调用函数。

库函数是已经定义和编译好的函数，同时可以使用标准库头文件提供其原型，因此只需正确地调用这种函数即可。本书前面的范例已经多次这样做了。例如，标准 C 库中有一个 strlen()函数，可用来确定字符串的长度。相关的标准头文件 cstring 包含了 strlen()和其他一些与字符串相关的函数的原型。这些预备工作使程序员能够在程序中随意使用 strlen()函数。

不过，创建自己的函数时，必须自行处理这 3 个方面——定义、提供原型和调用。程序清单 7.1 用一个简短的范例演示了这 3 个步骤。

程序清单 7.1　calling.cpp

```
// calling.cpp -- defining, prototyping, and calling a function
#include <iostream>
```

```
void simple();    // function prototype

int main()
{
    using namespace std;
    cout << "main()will call the simple()function: \n";
    simple();     // function call
    return 0;
}

// function definition
void simple()
{
    using namespace std;
    cout << "I'm but a simple function.\n";
}
```

下面是该程序的输出：

```
main()will call the simple()function:
I'm but a simple function.
```

这个程序在每个函数定义中使用了一条 using 编译指令，因为每个函数都使用了 cout。另一种方法是，在函数定义之前放置一条 using 编译指令。

下面详细介绍这 3 个步骤。

7.1.1　定义函数

可以将函数分成两类：没有返回值的函数和有返回值的函数。没有返回值的函数被称为 void 函数，其通用格式如下：

```
void functionName (parameterList)
{
    statement (s)
    return;          // optional
}
```

其中，parameterList 指定了传递给函数的参数类型和数量，本章后面将更详细地介绍该列表。可选的返回语句标记了函数的结尾；否则，函数将在右花括号处结束。void 函数相当于 Pascal 中的过程、FORTRAN 中的子程序和现代 BASIC 中的子程序过程。通常，可以用 void 函数来执行某种操作。例如，将 Cheers!打印指定次数（*n*）的函数如下：

```
void cheers (int n)          // no return value
{
    using namespace std;
    for (int i = 0; i < n; i++)
        cout << "Cheers! ";
    cout << endl;
}
```

参数列表 int n 意味着调用函数 cheers()时，应将一个 int 值作为参数传递给它。

有返回值的函数将生成一个值，并将它返回给调用函数。换句话来说，如果函数返回 9.0 的平方根（sqrt(9.0)），则该函数调用的值为 3.0。这种函数的类型被声明为返回值的类型，其通用格式如下：

```
typeName functionName (parameterList)
{
     statements
     return value;   // value is type cast to type typename
}
```

对于有返回值的函数，必须使用返回语句，以便将值返回给调用函数。值本身可以是常量、变量，也可以是表达式，只是其结果的类型必须为 typeName 类型或可以被转换为 typeName（例如，如果声明的返回类型为 double，而函数返回一个 int 表达式，则该 int 值将被强制转换为 double 类型）。然后，函数将最

终的值返回给调用函数。C++对于返回值的类型有一定的限制：不能是数组，但可以是其他任何类型——整数、浮点数、指针，甚至可以是结构和对象（有趣的是，虽然C++函数不能直接返回数组，但可以将数组作为结构或对象组成部分来返回）。

作为一名程序员，并不需要知道函数是如何返回值的，但是对这个问题有所了解将有助于澄清概念（另外，还有助于与朋友和家人交换意见）。通常，函数通过将返回值复制到指定的CPU寄存器或内存单元中来将其返回。随后，调用程序将查看该内存单元。返回函数和调用函数必须就该内存单元中存储的数据的类型达成一致。函数原型将返回值类型告知调用程序，而函数定义命令被调用函数应返回什么类型的数据（参见图7.1）。在原型中提供与定义中相同的信息似乎有些多余，但这样做确实有道理。要让信差从办公室的办公桌上取走一些物品，则向信差和办公室中的同事交代自己的意图，将提高信差顺利完成这项工作的概率。

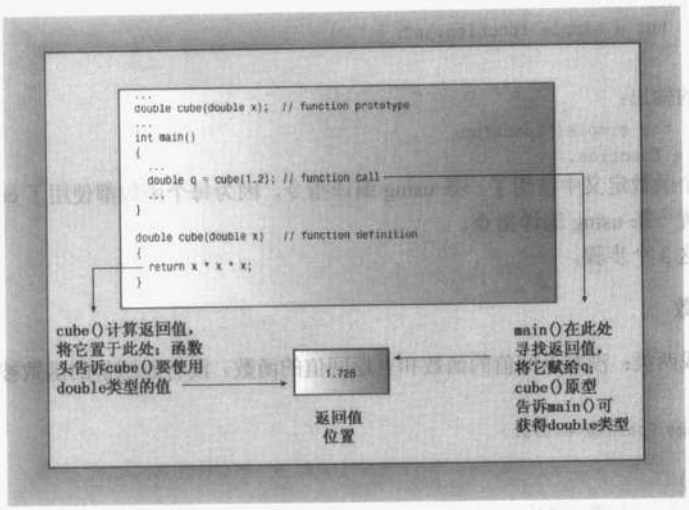


图7.1 典型的返回值机制

函数在执行返回语句后结束。如果函数包含多条返回语句（例如，它们位于不同的if else选项中），则函数在执行遇到的第一条返回语句后结束。例如，在下面的例子中，else并不是必需的，但它可以帮助马虎的读者理解程序员的意图：

```
int bigger (int a, int b)
{
   if (a > b)
      return a;  // if a > b, function terminates here
   else
      return b;  // otherwise, function terminates here
}
```

有返回值的函数与Pascal、FORTRAN和BASIC中的函数相似，它们向调用程序返回一个值，然后调用程序可以将其赋给变量、显示或将其用于别的用途。下面是一个简单的例子，函数返回double值的立方：

```
double cube (double x)    // x times x times x
{
   return x * x * x; // a type double value
}
```

例如，函数调用cube（1.2）将返回1.728。请注意，上述返回语句使用了一个表达式，函数将计算该表达式的值（这里为1.728），并将其返回。

7.1.2 函数原型和函数调用

至此，读者已熟悉了函数调用，但对函数原型可能不太熟悉，因为它经常隐藏在include文件中。程序

清单 7.2 在一个程序中使用了函数 cheer()和 cube()。请留意其中的函数原型。

程序清单 7.2　protos.cpp

```
// protos.cpp -- using prototypes and function calls
#include <iostream>
void cheers (int);          // prototype: no return value
double cube (double x);     // prototype: returns a double
int main (void)
{
   using namespace std;
   cheers (5);              // function call
   cout << "Give me a number: ";
   double side;
   cin >> side;
   double volume = cube (side);    // function call
   cout << "A " << side <<"-foot cube has a volume of ";
   cout << volume << " cubic feet.\n";
   cheers (cube (2));    // prototype protection at work
   return 0;
}

void cheers (int n)
{
   using namespace std;
   for (int i = 0; i < n; i++)
      cout << "Cheers! ";
   cout << endl;
}

double cube (double x)
{
   return x * x * x;
}
```

在程序清单 7.2 的程序中，只需要在使用名称空间 std 的成员函数中使用了编译指令 using。下面是该程序的运行情况：

```
Cheers! Cheers! Cheers! Cheers! Cheers!
Give me a number: 5
A 5-foot cube has a volume of 125 cubic feet.
Cheers! Cheers! Cheers! Cheers! Cheers! Cheers! Cheers! Cheers!
```

main()使用函数名和参数（后面跟一个分号）来调用 void 类型的函数：cheers (5);，这是一个函数调用语句。但由于 cube()有返回值，因此 main()可以将其用在赋值语句中：

```
double volume = cube (side);
```

不过正如前面指出的，读者应将重点放在原型上。那么，应了解有关原型的哪些内容呢？首先，需要知道 C++要求提供原型的原因。其次，由于 C++要求提供原型，因此还应知道正确的句法。最后，应当感谢原型所做的一切。下面依次介绍这几点，将程序清单 7.2 作为讨论的基础。

1. 为什么需要原型

原型描述了函数到编译器的接口，也就是说，它将函数返回值的类型（如果有的话）以及参数的类型和数量告诉编译器。例如，请看原型将如何影响程序清单 7.2 中下述函数调用：

```
double volume = cube (side);
```

首先，原型告诉编译器，cube()有一个 double 参数。如果程序没有提供这样的参数，原型将让编译器能够捕获这种错误。其次，cube()函数完成计算后，将把返回值放置在指定的位置——可能是 CPU 寄存器，也可能是内存中。然后调用函数（这里为 main()）将从这个位置取得返回值。由于原型指出了 cube()的类型为 double，因此编译器知道应检索多少个字节以及如何解释它们。如果没有这些信息，编译器将只能进行猜测，而编译器是不会这样做的。

读者可能还会问，为何编译器需要原型，难道它就不能在文件中进一步查找，以了解函数是如何定义的吗？这种方法的一个问题是效率不高。编译器在搜索文件的剩余部分时将必须停止对 main() 的编译。一个更严重的问题是，函数甚至可能并不在文件中。C++允许将一个程序放在多个文件中，单独编译这些文件，然后再将它们组合起来。在这种情况下，编译器在编译 main() 时，可能无权访问函数代码。如果函数位于库中，情况也将如此。避免使用函数原型的惟一方法是，在首次使用函数之前定义它，但这并不总是可行的。另外，C++的编程风格是将 main() 放在最前面，因为它通常提供了程序的整体结构。

2. 原型的句法

函数原型是一条语句，因此必须以分号结束。获得原型最简单的方法是，复制函数定义中的函数头，并添加分号。对于 cube()，程序清单 7.2 中的程序正是这样做的：

```
double cube (double x) ; // add; to header to get prototype
```

不过，函数原型不要求提供变量名，有类型列表就足够了。对于 cheer() 的原型，该程序只提供了参数类型：

```
void cheers (int) ; // okay to drop variable names in prototype
```

通常，在原型的参数列表中，可以包括变量名，也可以不包括。原型中的变量名相当于占位符，因此不必与函数定义中的变量名相同。

C++原型与 ANSI C 原型

ANSI C 借鉴了 C++中的原型，但这两种语言还是有区别的。其中最重要的区别是，为与基本 C 兼容，ANSI C 中的原型是可选的，但在 C++中，原型是必不可少的。例如，请看下面的函数声明：

```
void say_hi();
```

在 C++中，括号为空与在括号中使用关键字 void 是等效的——意味着函数没有参数。在 ANSI C 中，括号为空意味着不指出参数——这意味着将在后面定义参数列表。在 C++中，不指定参数列表时应使用省略号：

```
void say_bye (...) ;  // C++ abdication of resposibility
```

通常，仅当与接受可变参数的 C 函数（如 printf()）交互时才需要这样做。

3. 原型的功能

可以看出，原型可以帮助编译器完成许多工作；但它对程序员有什么帮助呢？它们可以极大地降低程序出错的几率。具体来说，原型确保以下几点：

- 编译器正确处理函数返回值。
- 编译器检查使用的参数数目是否正确。
- 编译器检查使用的参数类型是否正确。如果不正确，则转换为正确的类型（如果可能的话）。

前面已经讨论了如何正确处理返回值。下面来看一看参数数目不对时将发生的情况。例如，假设进行了如下调用：

```
double z = cube();
```

如果没有函数原型，编译器将允许它通过。当函数被调用时，它将找到 cube() 调用存放值的位置，并使用这里的值。这正是 ANSI C 从 C++借鉴原型之前，C 语言的工作方式。由于对于 ANSI C 来说，原型是可选的，因此有些 C 语言程序正是这样工作的。但在 C++中，原型不是可选的，因此可以确保不会发生这类错误。

接下来，假设提供了一个参数，但其类型不正确。在 C 语言中，这将造成奇怪的错误。例如，如果函数需要一个 int 值（假设占 16 位），而程序员传递了一个 double 值（假设占 64 位），则函数将只检查 64 位中的前 16 位，并试图将它们解释为一个 int 值。但 C++自动将传递的值转换为原型中指定的类型，条件是两者都是算术类型。例如，程序清单 7.2 将能够应付下述语句中两次出现的类型不匹配的情况：

```
cheers (cube (2) ) ;
```

首先，程序将 int 的值 2 传递给 cube()，而后者期望的是 double 类型。编译器注意到，cube() 原型指定了一个 double 类型参数，因此将 2 转换为 2.0——一个 double 值。接下来，cube() 返回一个 double 值（8.0），

这个值被用作 cheer() 的参数。编译器将再一次检查原型，并发现 cheer() 要求一个 int 参数，因此它将返回值转换为整数 8。通常，原型自动将被传递的参数强制转换为期望的类型（但第 8 章将介绍的函数重载可能导致二义性，因此不允许某些自动强制类型转换）。

自动类型转换并不能避免所有可能的错误。例如，如果将 8.33E27 传递给期望一个 int 值的函数，则这样大的值将不能被正确转换为 int 值。当较大的类型被自动转换为较小的类型时，有些编译器将发出警告，指出这可能会丢失数据。

仅当有意义时，原型化才会导致类型转换。例如，原型不会将整数转换为结构或指针。

在编译阶段进行的原型化被称为静态类型检查（static type checking）。可以看出，静态类型检查可捕获许多在运行阶段非常难以捕获的错误。

7.2　函数参数和按值传递

下面详细介绍一下函数参数。C++通常按值传递参数，这意味着将数值参数传递给函数，而后者将其赋给一个新的变量。例如，程序清单 7.2 包含下面的函数调用：

```
double volume = cube(side);
```

其中，side 是一个变量，在前面的程序运行中，其值为 5。cube() 的函数头如下：

```
double cube(double x)
```

被调用时，它将创建一个新的名为 x 的 double 变量，并将 5 赋给它。这样，cube() 执行的操作将不会影响 main() 中的数据，因为 cube() 使用的是 side 的拷贝，而不是原来的数据。稍后将介绍一个实现这种保护的例子。用于接收传递值的变量被称为形参。传递给函数的值被称为实参。出于简化的目的，C++标准使用参数（argument）来表示实参，使用参量（parameter）来表示形参，因此参数传递将参量赋给参数（参见图 7.2）。

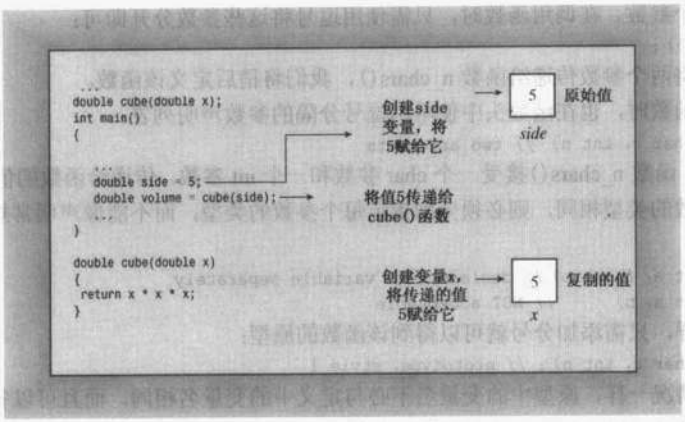


图 7.2　按值传递

在函数中声明的变量（包括参数）是该函数私有的。在函数被调用时，计算机将为这些变量分配内存；在函数结束时，计算机将释放这些变量使用的内存（有些 C++文献将分配和释放内存称为创建和毁坏变量，这样似乎更激动人心）。这样的变量被称为局部变量，因为它们被限制在函数中。前面提到过，这样做有助于确保数据的完整性。这还意味着，如果在 main() 中声明了一个名为 x 的变量，同时在另一个函数中也声明了一个名为 x 的变量，则它们将是两个完全不同的、毫无关系的变量，这与加利福尼亚州的 Albany 与纽约的 Albany 是两个完全不同的地方是一样的道理（参见图 7.3）。这样的变量也被称为自动变量，因为它们是在程序执行过程中自动被分配和释放的。

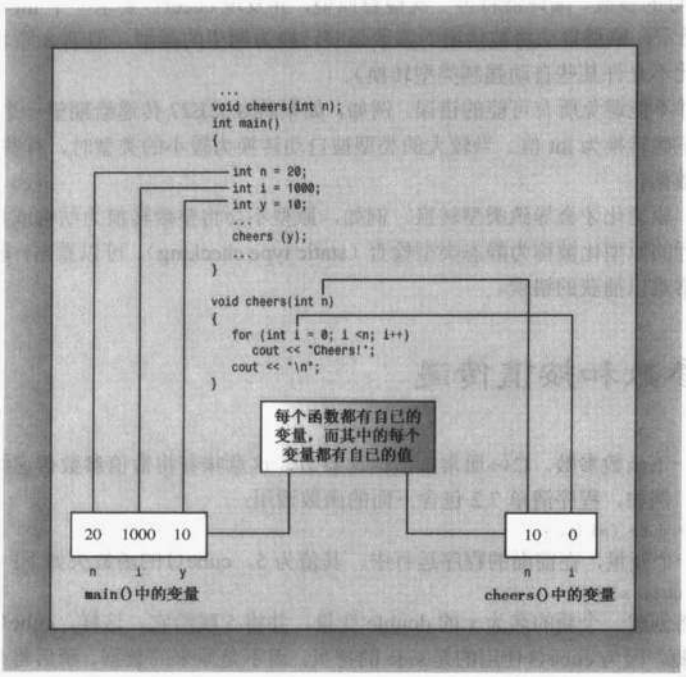


图 7.3 局部变量

7.2.1 多个参数

函数可以有多个参数。在调用函数时，只需使用逗号将这些参数分开即可：

```
n_chars ('R', 25);
```

上述函数调用将两个参数传递给函数 n_chars()，我们将稍后定义该函数。

同样，在定义函数时，也在函数头中使用由逗号分隔的参数声明列表：

```
void n_chars (char c, int n)  // two arguments
```

该函数头指出，函数 n_chars()接受一个 char 参数和一个 int 参数。传递给函数的值被赋给参数 *c* 和 *n*。如果函数的两个参数的类型相同，则必须分别指定每个参数的类型，而不能像声明常规变量那样，将声明组合在一起：

```
void fifi (float a, float b) // declare each variable separately
void fufu (float a, b)       // NOT acceptable
```

和其他函数一样，只需添加分号就可以得到该函数的原型：

```
void n_chars (char c, int n); // prototype, style 1
```

和一个参数的情况一样，原型中的变量名不必与定义中的变量名相同，而且可以省略：

```
void n_chars (char, int);     // prototype, style 2
```

不过，提供变量名将使原型更容易理解，尤其是两个参数的类型相同时。这样，变量名可以提醒参量和参数间的对应关系：

```
double melon_density (double weight, double volume);
```

程序清单 7.3 演示了一个接受两个参数的函数，它还表明，在函数中修改形参的值不会影响调用程序中的数据。

程序清单 7.3 twoarg.cpp

```
// twoarg.cpp -- a function with 2 arguments
#include <iostream>
using namespace std;
```

```
void n_chars (char, int);
int main()
{
    int times;
    char ch;

    cout << "Enter a character: ";
    cin >> ch;
    while (ch != 'q')       // q to quit
    {
        cout << "Enter an integer: ";
        cin >> times;
        n_chars (ch, times); // function with two arguments
        cout << "\nEnter another character or press the"
             " q-key to quit: ";
        cin >> ch;
    }
    cout << "The value of times is " << times << ".\n";
    cout << "Bye\n";
    return 0;
}

void n_chars (char c, int n) // displays c n times
{
    while (n-- > 0)         // continue until n reaches 0
        cout << c;
}
```

在程序清单 7.3 的程序中，将编译指令 using 放在函数定义的前面，而不是函数中。下面是该程序的运行情况：

```
Enter a character: W
Enter an integer: 50
WWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWW
Enter another character or press the q-key to quit:  a
Enter an integer: 20
aaaaaaaaaaaaaaaaaaaa
Enter another character or press the q-key to quit:  q
The value of times is 20.
Bye
```

程序说明

程序清单 7.3 中的 main() 函数使用一个 while 循环提供重复输入（并让读者温习使用循环的技巧），它使用 cin>>ch，而不是 cin.get（ch）或 ch = cin.get() 来读取一个字符。这样做是有原因的。前面讲过，这两个 cin.get() 函数读取所有的输入字符，包括空格和换行符，而 cin>>跳过空格和换行符。当用户对程序提示做出响应时，必须在每行的最后按 Enter 键，以生成换行符。cin>>ch 方法可以轻松地跳过这些换行符，但当输入的下一个字符为数字时，cin.get() 将读取后面的换行符。可以通过编程来避开这种麻烦，但比较简便的方法是像该程序那样使用 cin。

n_char() 函数接受两个参数：一个是字符 c，另一个是整数 n。然后，它使用循环来显示该字符，显示次数为 n：

```
while (n-- > 0)         // continue until n reaches 0
    cout << c;
```

程序通过将 *n* 变量递减来计数，其中 *n* 是参数列表的形参，main() 中 times 变量的值被赋给该变量。然后，while 循环将 *n* 递减到 0，但前面的运行情况表明，修改 *n* 的值对 times 没有影响。

7.2.2　另外一个接受两个参数的函数

下面创建另一个功能更强大的函数，它执行重要的计算任务。另外，函数将演示局部变量的用法，而

不是形参的用法。

目前，美国许多州都采用某种纸牌游戏的形式来发行彩票，让参与者从卡片中选择一定数目的选项。例如，从 51 个数字中选取 6 个。随后，彩票管理者将随机抽取 6 个数。如果参与者选择的数字与这 6 个完全相同，将赢得大约几百万美元的奖金。我们的函数将计算中奖的几率（是的，能够成功预测获奖号码的函数将更有用，但虽然 C++的功能非常强大，目前还不具备超自然能力）。

首先，需要一个公式。假设必须从 51 个数中选取 6 个，而获奖的概率为 1/R，则 R 的计算公式如下：

$$R = \frac{51 \times 50 \times 49 \times 48 \times 47 \times 46}{6 \times 5 \times 4 \times 3 \times 2 \times 1}$$

选择 6 个数时，分母为前 6 个整数的乘积或 6 的阶乘。分子也是 6 个连续整数的乘积，从 51 开始，依次减 1。推广开来，如果从 numbers 个数中选取 picks 个数，则分母是 picks 的阶乘，分子为 numbers 开始向前的 picks 个整数的乘积。可以用 for 循环来进行计算：

```
long double result = 1.0;
for (n = numbers, p = picks; p > 0; n--, p--)
   result = result * n / p;
```

循环不是首先将所有的分子项相乘，而是首先将 1.0 与第一个分子项相乘，然后除以第一个分母项。然后，下一轮循环乘以第二个分子项，并除以第二个分母项。这样得到的乘积将比先进行乘法运算得到的小。例如，对于（10 * 9）/（2 * 1）和（10 / 2）*（9 / 1），前者将计算 90/2，得到 45，后者将计算为 5×9，得到 45。这两种方法得到的结果相同，但前者的中间值（90）大于后者。因子越多，中间值的差别就越大。当数字非常大时，这种交替进行乘除运算的策略可以防止中间结果超出最大的浮点数。

程序清单 7.4 在 probability() 函数中使用了这个公式。由于选择的数目和总数目都为正，因此该程序将这些变量声明为 unsigned int 类型（简称 unsigned）。将若干整数相乘可以得到相当大的结果，因此 lotto.cpp 将该函数的返回值声明为 long double 类型。另外，如果使用整型，则像 49/6 这样的运算将出现舍入误差。

注意：有些 C++实现不支持 long double 类型，如果所用的 C++实现属于这种类型，请使用 double 类型。

程序清单 7.4 lotto.cpp

```
// lotto.cpp -- probability of winning
#include <iostream>
// Note: some implementations require double instead of long double
long double probability(unsigned numbers, unsigned picks);
int main()
{
   using namespace std;
   double total, choices;
   cout << "Enter the total number of choices on the game card and\n"
      "the number of picks allowed:\n";
   while ((cin >> total >> choices) && choices <= total)
   {
      cout << "You have one chance in ";
      cout << probability(total, choices);     // compute the odds
      cout << " of winning.\n";
      cout << "Next two numbers (q to quit): ";
   }
   cout << "bye\n";
   return 0;
}

// the following function calculates the probability of picking picks
// numbers correctly from numbers choices
long double probability(unsigned numbers, unsigned picks)
{
   long double result = 1.0;  // here come some local variables
```

```
    long double n;
    unsigned p;

    for (n = numbers, p = picks; p > 0; n--, p--)
        result = result * n / p ;
    return result;
}
```

下面是该程序的运行情况：

```
Enter total number of choices on the game card and
the number of picks allowed:
49 6
You have one chance in 1.39838e+007 of winning.
Next two numbers (q to quit): 51 6
You have one chance in 1.80095e+007 of winning.
Next two numbers (q to quit): 38 6
You have one chance in 2.76068e+006 of winning.
Next two numbers (q to quit): q
bye
```

请注意，增加游戏卡中可供选择的数字数目，获奖的可能性将急剧降低。

程序说明

程序清单 7.4 中的 probability()函数演示了可以在函数中使用的两种局部变量。首先是形参（number 和 picks），这是在左括号前面的函数头中声明的；其次是其他局部变量（result、*n* 和 *p*），它们是在将函数定义括起的括号内声明的。形参与其他局部变量的主要区别是，形参从调用 probability ()的函数那里获得自己的值，而其他变量是从函数中获得自己的值。

7.3　函数和数组

到目前为止，本书的函数范例都很简单，参数和返回值的类型都是基本类型。但是，函数是处理更复杂的类型（如数组和结构）的关键。下面来看看如何将数组和函数结合在一起。

假设使用一个数组来记录家庭野餐中每人吃了多少个甜饼（每个数组索引都对应一个人，元素值对应于这个人所吃的甜饼数量）。现在想知道总数。这很容易，只需使用循环将所有数组元素累积起来即可。将数组元素累加是一项非常常见的任务，因此设计一个完成这项工作的函数很有意义。这样就不必在每次计算数组总和时都编写新的循环了。

考虑函数接口所涉及的内容。由于函数计算总数，因此应返回答案。如果不分吃甜饼，则可以让函数的返回类型为 int。另外，函数需要知道要对哪个数组进行累计，因此需要将数组名作为参数传递给它。为使函数通用，而不限于特定长度的数组，还需要传递数组长度。这里惟一的新内容是，需要将一个形参声明为数组名。下面来看一看函数头及其其他部分：

```
int sum_arr (int arr[], int n) // arr = array name, n = size
```

这看起来似乎合理。方括号指出 arr 是一个数组，而方括号为空则表明，可以将任何长度的数组传递给该函数。但实际情况并非如此：arr 实际上并不是数组，而是一个指针！好消息是，在编写函数的其余部分时，可以将 arr 看作是数组。首先，来看一看这种方法是如何工作的，然后看看它为什么适用。

程序清单 7.5 演示如同使用数组名那样使用指针的情况。程序将数组初始化为某些值，并使用 sum_arr()函数计算总数。注意到 sum_arr()函数使用 arr 时，就像是使用数组名一样。

程序清单 7.5　arrfun1.cpp

```
// arrfun1.cpp -- functions with an array argument
#include <iostream>
const int ArSize = 8;
```

```
int sum_arr (int arr[], int n);       // prototype
int main()
{
    using namespace std;
    int cookies[ArSize] = {1, 2, 4, 8, 16, 32, 64, 128};
// some systems require preceding int with static to
// enable array initialization

    int sum = sum_arr (cookies, ArSize);
    cout << "Total cookies eaten: " << sum << "\n";
    return 0;
}

// return the sum of an integer array
int sum_arr (int arr[], int n)
{
    int total = 0;
    for (int i = 0; i < n; i++)
        total = total + arr[i];
    return total;
}
```

下面是该程序的输出：

```
Total cookies eaten: 255
```

从中可知，该程序可用。下面讨论该程序为何适用。

7.3.1 函数如何使用指针来处理数组

在大多数情况下，C++和 C 语言一样，也将数组名视为指针。第 4 章介绍过，C++将数组名解释为其第一个元素的地址：

```
cookies == &cookies[0]  // array name is address of first element
```

（这一规则有两种例外的情况：第一种情况是数组声明使用数组名来标记存储位置；第二种情况是对数组名使用 sizeof 将得到整个数组的长度（以字节为单位））。

程序清单 7.5 执行下面的函数调用：

```
int sum = sum_arr (cookies, ArSize);
```

其中，cookies 是数组名，而根据 C++规则，cookies 是其第一个元素的地址，因此函数传递的是地址。由于数组的元素的类型为 int，因此 cookies 的类型必须是 int 指针，即 int *。这表明，正确的函数头应该是这样的：

```
int sum_arr (int * arr, int n) // arr = array name, n = size
```

其中用 int * arr 替换了 int arr []。这证明这两个函数头都是正确的，因为在 C++中，当（且仅当）用于函数头或函数原型中，int *arr 和 int arr [ ]的含义才是相同的。它们都意味着 arr 是一个 int 指针。不过，数组表示法（int arr[]）提醒用户，arr 不仅指向 int，还指向 int 数组的第一个 int。当指针指向数组的第一个元素时，本书使用数组表示法；而当指针指向一个独立的值时，使用指针表示法。别忘了，在其他的上下文中，int * arr 和 int arr []的含义并不相同。例如，不能在函数体中使用 int tip[ ]来声明指针。

鉴于变量 arr 实际上就是一个指针，函数的其余部分是合理的。第 4 章在介绍动态数组时指出过，同数组名或指针一样，也可以用方括号数组表示法来访问数组元素。无论 arr 是指针还是数组名，表达式 arr [3] 都指的是数组的第 4 个元素。就目前而言，提请读者记住下面两个恒等式，将不会有任何坏处：

```
arr[i] == *(ar + i)    // values in two notations
&arr[i] == ar + I      // addresses in two notations
```

记住，将指针（包括数组名）加 1，实际上是加上了一个与指针指向的类型的长度（以字节为单位）相等的值。对于遍历数组而言，使用指针加法和数组下标是等效的。

7.3.2 将数组作为参数意味着什么

我们来看一看程序清单 7.5 暗示了什么。函数调用 sum_arr（coolies，ArSize）将 cookies 数组第一个元

素的地址和数组中的元素数目传递给 sum_arr() 函数。sum_arr() 函数将 cookies 的地址赋给指针变量 arr，将 ArSize 赋给 int 变量 *n*。这意味着，程序清单 7.5 实际上并没有将数组内容传递给函数，而是将数组的位置（地址）、包含的元素种类（类型）以及元素数目（*n* 变量）提交给函数（参见图 7.4）。有了这些信息后，函数便可以使用原来的数组。传递常规变量时，函数将使用该变量的拷贝；但传递数组时，函数将使用原来的数组。实际上，这种区别并不违反 C++按值传递的方法，sum_arr() 函数仍传递了一个值，这个值被赋给一个新的变量，但这个值是一个地址，而不是数组的内容。

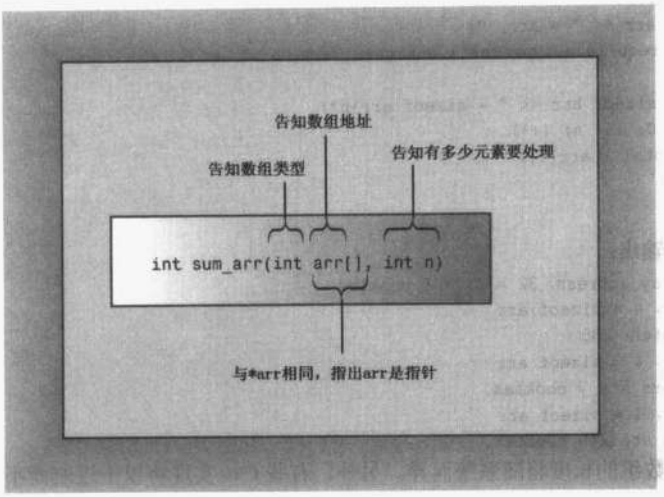


图 7.4　告知函数有关数组的信息

数组名与指针对应是好事吗？确实是一件好事。将数组地址作为参数可以节省复制整个数组所需的时间和内存。如果数组很大，则使用拷贝的系统开销将非常大；程序不仅需要更多的计算机内存，还需要花费时间来复制大块的数据。另一方面，使用原始数据增加了破坏数据的风险。在经典的 C 语言中，这确实是一个问题，但 ANSI C 和 C++中的 const 限定符提供了解决这种问题的办法。稍后将介绍一个这样的范例。不过。我们首先来修改程序清单 7.5，以演示数组函数是如何运作的。程序清单 7.6 表明，cookies 和 arr 的值相同。它还演示了指针概念如何使 sum_arr 函数比以前更通用。该程序使用限定符 std::而不是编译指令 using 来提供对 cout 和 endl 的访问权。

程序清单 7.6　arrfun2.cpp

```
// arrfun2.cpp -- functions with an array argument
#include <iostream>
const int ArSize = 8;
int sum_arr(int arr[], int n);
// use std:: instead of using directive
int main()
{
    int cookies[ArSize] = {1,2,4,8,16,32,64,128};
//  some systems require preceding int with static to
//  enable array initialization

    std::cout << cookies << " = array address, ";
//  some systems require a type cast: unsigned (cookies)

    std::cout << sizeof cookies << " = sizeof cookies\n";
    int sum = sum_arr(cookies, ArSize);
    std::cout << "Total cookies eaten: " << sum << std::endl;
    sum = sum_arr(cookies, 3);       // a lie
```

```
    std::cout << "First three eaters ate " << sum << " cookies.\n";
    sum = sum_arr(cookies + 4, 4);    // another lie
    std::cout << "Last four eaters ate " << sum << " cookies.\n";
    return 0;
}

// return the sum of an integer array
int sum_arr(int arr[], int n)
{
    int total = 0;
    std::cout << arr << " = arr, ";
// some systems require a type cast: unsigned (arr)

    std::cout << sizeof arr << " = sizeof arr\n";
    for (int i = 0; i < n; i++)
        total = total + arr[i];
    return total;
}
```

下面是该程序的输出：

```
0x0065fd24 = array address, 32 = sizeof cookies
0x0065fd24 = arr, 4 = sizeof arr
Total cookies eaten: 255
0x0065fd24 = arr, 4 = sizeof arr
First three eaters ate 7 cookies.
0x0065fd34 = arr, 4 = sizeof arr
Last four eaters ate 240 cookies.
```

注意，地址值和数组的长度将随系统而异。另外，有些C++实现将以十进制而不是十六进制格式显示地址。

程序说明

程序清单7.6说明了数组函数的一些有趣的地方。首先，cookies和arr指向同一个地址。但sizeof cookies的值为16，而sizeof arr为4。这是由于sizeof cookies是整个数组的长度，而sizeof arr只是指针变量的长度（上述程序运行结果是从一个使用4字节地址的系统中获得的）。另外，这就是必须显式传递数组长度，而不能在sum_arr()中使用sizeof arr的原因。

由于sum_arr()只能通过第二个参数获知数组中的元素数量，因此可以对函数“说谎”。例如，程序第二次使用该函数时，这样调用它：

```
sum = sum_arr(cookies, 3);
```

通过告诉该函数cookies有3个元素，可以让它计算前3个元素的总和。

为什么在这里停下了呢？还可以提供假的数组起始位置：

```
sum = sum_arr(cookies + 4, 4);
```

由于cookies是第一个元素的地址，因此cookies + 4是第5个元素的地址。这条语句将计算数组第5、6、7、8个元素的总和。请注意输出中第三次函数调用选择将不同于前两个调用的地址赋给arr的。是的，可以将&cookies[4]，而不是cookies + 4作为参数；它们的含义是相同的。

记住：为将数组类型和元素数量告诉数组处理函数，请通过两个不同的参数来传递它们：

```
void fillArray(int arr[], int size);  // prototype
```

而不要试图使用方括号表示法来传递数组长度：

```
void fillArray(int arr[size]);        // NO -- bad prototype
```

7.3.3 更多数组函数范例

选择使用数组来表示数据时，实际上是在进行一次设计方面的决策。但设计决策不仅仅是确定数据的存储方式，还涉及到如何使用数据。程序员常会发现，编写特定的函数来处理特定的数据操作是有好处的（这里讲的好处指的是程序的可靠性更高、修改和调试更为方便）。另外，构思程序时将存储属性与操作结

合起来，便是朝 OOP 思想迈进了重要的一步；以后将证明这是很有好处的。

我们来看一个简单的案例。假设要使用一个数组来记录房地产的价值（假设拥有房地产）。在这种情况下，程序员必须确定要使用哪种类型。当然，double 的取值范围比 int 和 long 大，并且提供了足够多的有效位数来精确地表示这些值。接下来必须决定数组元素的数目（对于使用 new 创建的动态数组来说，可以稍后再决定，但我们希望使事情简单一点）。如果房地产数目不超过 5 个，则可以使用一个包含 5 个元素的 double 数组。

现在，考虑要对房地产数组执行的操作。两个基本的操作分别是，将值读入到数组中和显示数组内容。我们再添加另一个操作：重新评估每种房地产的值。为简单起见，假设所有房地产都以相同的比率增加或者减少（别忘了，这是一本关于 C++的书，而不是关于房地产管理的书）。接下来，为每项操作编写一个函数，然后编写相应的代码。下面首先介绍这些步骤，然后将其用于一个完整的范例中。

1. 填充数组

由于接受数组名参数的函数访问的是原始数组，而不是其拷贝，因此可以通过调用该函数将值赋给数组元素。该函数的一个参数是要填充的数组的名称。通常，程序可以管理多个人的投资，因此需要多个数组，因此不能在函数中设置数组长度，而要将数组长度作为第二个参数传递，就像前一个范例那样。另外，用户也可能希望在数组被填满之前停止读取数据，因此需要在函数中建立这种特性。由于用户输入的元素数目可能少于数组的长度，因此函数应返回实际输入的元素数目。因此，该函数的原型如下：

```
int fill_array(double ar[], int limit);
```

该函数接受两个参数，一个是数组名，另一个指定了要读取的最大元素数；该函数返回实际读取的元素数。例如，如果使用该函数来处理一个包含 5 个元素的数组，则将 5 作为第二个参数。如果只输入 3 个值，则该函数将返回 3。

可以使用循环连续地将值读入到数组中，但如何提早结束循环呢？一种方法是，使用一个特殊值来指出输入结束。由于所有的属性都不为负，因此可以使用负数来指出输入结束。另外，该函数应对错误输入作出反应，如停止输入等。这样，该函数的代码如下所示：

```
int fill_array(double ar[], int limit)
{
    using namespace std;
    double temp;
    int i;
    for (i = 0; i < limit; i++)
    {
        cout << "Enter value #" << (i + 1) << ": ";
        cin >> temp;
        if (!cin)    // bad input
        {
            cin.clear();
            while (cin.get()!= '\n')
                continue;
            cout << "Bad input; input process terminated.\n";
            break;
        }
        else if (temp < 0)    // signal to terminate
            break;
        ar[i] = temp;
    }
    return i;
}
```

注意，代码中包含了对用户的提示。如果用户输入的是非负值，则这个值将被赋给数组，否则循环结束。如果用户输入的都是有效值，则循环将在读取最大数目的值后结束。循环完成的最后一项工作是将 i 加 1，因此循环结束后，i 将比最后一个数组索引大 1，即等于填充的元素数目。然后，函数返回这个值。

2. 显示数组及用 const 保护数组

创建显示数组内容的函数很简单。只需将数组名和填充的元素数目传递给函数，然后该函数使用循环来显示每个元素。不过，还有另外一个问题——确保显示函数不修改原始数组。除非函数的目的就是修改传递给它的数据，否则应避免发生这种情况。使用普通参数时，这种保护将自动实现，这是由于 C++ 按值传递数据，而且函数使用数据的拷贝。不过，接受数组名的函数将使用原始数据，这正是 fill_array() 函数能够完成其工作的原因。为防止函数无意中修改数组的内容，可在声明形参时使用关键字 const（参见第 3 章）：

```
void show_array (const double ar[], int n);
```

该声明表明，指针 ar 指向的是常量数据。这意味着不能使用 ar 修改来该数据，也就是说，可以使用像 ar[0]这样的值，但不能修改。注意，这并不是意味着原始数组必须是常量，而只是意味着不能在 show_array() 函数中使用 ar 来修改这些数据。因此，show_array() 将数组视为只读数据。假设无意间在 show_array() 函数中执行了下面的操作，从而违反了这种限制：

```
ar[0] += 10;
```

编译器将禁止这样做。例如，Borland C++将给出一条错误消息，如下所示（稍作编辑）：

```
Cannot modify a const object in function
      show_array (const double *, int)
```

其他编译器可能用其他措词表示其不满。

这条消息提醒用户，C++将声明 const double ar [ ]解释为 const double *ar。因此，该声明实际上是说，ar 指向的是一个常量值。结束这个例子后，我们将详细讨论这个问题。下面是 show_array() 函数的代码：

```
void show_array (const double ar[], int n)
{
    using namespace std;
    for (int i = 0; i < n; i++)
    {
        cout << "Property #" << (i + 1) << ": $";
        cout << ar[i] << endl;
    }
}
```

3. 修改数组

在这个例子中，对数组进行的第三项操作是将每个元素与同一个重新评估因子相乘。需要给函数传递 3 个参数：因子、数组和元素数目。该函数不需要返回值，因此其代码如下：

```
void revalue (double r, double ar[], int n)
{
    for (int i = 0; i < n; i++)
        ar[i] *= r;
}
```

由于这个函数将修改数组的值，因此在声明 ar 时，不能使用 const。

4. 将上述代码组合起来

至此，我们根据数据的存储方式（数组）和使用方式（3 个函数）定义了数据的类型，因此可以将它们组合成一个程序。由于已经建立了所有的数组处理工具，因此 main() 的编程工作非常简单。余下的大部分编程工作只是让 main() 调用前面开发的函数。程序清单 7.7 列出了最终的代码，它将编译指令 using 放在那些需要 iostream 工具的函数中。

程序清单 7.7 arrfun3.cpp

```
// arrfun3.cpp -- array functions and const
#include <iostream>
const int Max = 5;

// function prototypes
```

```
int fill_array (double ar[], int limit);
void show_array (const double ar[], int n);  // don't change data
void revalue (double r, double ar[], int n);

int main()
{
    using namespace std;
    double properties[Max];

    int size = fill_array (properties, Max);
    show_array (properties, size);
    cout << "Enter revaluation factor: ";
    double factor;
    cin >> factor;
    revalue (factor, properties, size);
    show_array (properties, size);
    cout << "Done.\n";
    return 0;
}

int fill_array (double ar[], int limit)
{
    using namespace std;
    double temp;
    int i;
    for (i = 0; i < limit; i++)
    {
        cout << "Enter value #" << (i + 1) << ": ";
        cin >> temp;
        if (!cin)    // bad input
        {
            cin.clear();
            while (cin.get()!= '\n')
                continue;
          cout << "Bad input; input process terminated.\n";
          break;
        }
        else if (temp < 0)     // signal to terminate
            break;
        ar[i] = temp;
    }
    return i;
}

// the following function can use, but not alter,
// the array whose address is ar
void show_array (const double ar[], int n)
{
    using namespace std;
    for (int i = 0; i < n; i++)
    {
        cout << "Property #" << (i + 1) << ": $";
        cout << ar[i] << endl;
    }
}

// multiplies each element of ar[] by r
void revalue (double r, double ar[], int n)
{
    for (int i = 0; i < n; i++)
```

```
        ar[i] *= r;
}
```

下面是两次运行该程序时的输出：

```
Enter value #1: 100000
Enter value #2: 80000
Enter value #3: 222000
Enter value #4: 240000
Enter value #5: 118000
Property #1: $100000
Property #2: $80000
Property #3: $222000
Property #4: $240000
Property #5: $118000
Enter reassessment rate: 1.10
Property #1: $110000
Property #2: $88000
Property #3: $244200
Property #4: $264000
Property #5: $129800
Done.

Enter value #1: 200000
Enter value #2: 84000
Enter value #3: 160000
Enter value #4: -2
Property #1: $200000
Property #2: $84000
Property #3: $160000
Enter reassessment rate: 1.20
Property #1: $240000
Property #2: $100800
Property #3: $192000
Done.
```

函数 fill_array() 指出，当用户输入 5 项房地产值或负值后，将结束输入。第一次运行演示了输入 5 项房地产值的情况，第二次运行演示了输入负值的情况。

5. 程序说明

前面已经讨论了与该范例相关的重要编程细节，因此这里回顾一下整个过程。我们首先考虑的是通过数据类型和设计适当的函数来处理数据，然后将这些函数组合成一个程序。有时也称为自下而上的程序设计（bottom-up programming），因为设计过程从组件到整体进行。这种方法非常适合于 OOP——它首先强调的是数据表示和操纵。而传统的过程性编程倾向于自顶向下的程序设计（top-down programming），首先指定模块化设计方案，然后再研究细节。这两种方法都很有用，最终的产品都是模块化程序。

7.3.4 使用数组区间的函数

可以看出，对于处理数组的 C++函数，必须将数组中的数据种类、数组的起始位置和数组中元素数量提交给它；传统的 C/C++方法是，将指向数组起始处的指针作为一个参数，将数组长度作为第二个参数（指针指出数组的位置和数据类型），这样便给函数提供了找到所有数据所需的信息。

还有另一种给函数提供所需信息的方法，即指定元素区间（range），这可以通过传递两个指针来完成：一个指针标识数组的开头，另一个指针标识数组的尾部。例如，C++标准模板库（STL，将在第 16 章介绍）将区间方法广义化了。STL 方法使用“超尾”概念来指定区间。也就是说，对于数组而言，标识数组结尾的参数将是指向最后一个元素后面的指针。例如，假设有这样的声明：

```
double elbuod[20];
```

则指针 elboud 和 elboud + 20 定义了区间。首先，数组名 elboub 指向第一个元素。表达式 elboud + 19

指向最后一个元素（即 elboud[19]），因此，elboud + 20 指向数组结尾后面的一个位置。将区间传递给函数将告诉函数应处理哪些元素。程序清单 7.8 对程序清单 7.6 做了修改，使用两个指针来指定区间。

程序清单 7.8　arrfun4.cpp

```
// arrfun4.cpp -- functions with an array range
#include <iostream>
const int ArSize = 8;
int sum_arr (const int * begin, const int * end);
int main()
{
    using namespace std;
    int cookies[ArSize] = {1, 2, 4, 8, 16, 32, 64, 128};
//  some systems require preceding int with static to
//  enable array initialization

    int sum = sum_arr (cookies, cookies + ArSize);
    cout << "Total cookies eaten: "<< sum <<  endl;
    sum = sum_arr (cookies, cookies + 3);       // first 3 elements
    cout << "First three eaters ate " << sum << " cookies.\n";
    sum = sum_arr (cookies + 4, cookies + 8);    // last 4 elements
    cout << "Last four eaters ate " << sum << " cookies.\n";
    return 0;
}

// return the sum of an integer array
int sum_arr (const int * begin, const int * end)
{
    const int * pt;
    int total = 0;
    for (pt = begin; pt != end; pt++)
        total = total + *pt;
    return total;
}
```

下面是该程序的输出：

```
Total cookies eaten: 255
First three eaters ate 7 cookies.
Last four eaters ate 240 cookies.
```

程序说明

请注意程序清单 7.8 中 sum_array() 函数中的 for 循环：

```
for (pt = begin; pt != end; pt++)
    total = total + *pt;
```

它将 pt 设置为指向要处理的第一个元素（begin 指向的元素）的指针，并将*pt（元素的值）加入到 total 中。然后，循环通过递增操作来更新 pt，使之指向下一个元素。只要 pt 不等于 end，这一过程就将继续下去。当 pt 等于 end 时，它将指向区间中最后一个元素后面的一个位置，此时循环将结束。

其次，请注意不同的函数调用是如何指定数组中不同的区间的：

```
int sum = sum_arr (cookies, cookies + ArSize);
...
sum = sum_arr (cookies, cookies + 3);       // first 3 elements
...
sum = sum_arr (cookies + 4, cookies + 8);    // last 4 elements
```

指针 cookies + ArSize 指向最后一个元素后面的一个位置（数组有 ArSize 个元素，因此 cookies[ArSize - 1]是最后一个元素，其地址为 cookies + ArSize – 1）。因此，区间[cookies，cookies + ArSize]指定的是整个数组。同样，cookies，cookies + 3 指定了前 3 个元素，依此类推。

请注意，根据指针减法规则，在 sum_arr() 中，表达式 end – begin 是一个整数值，等于数组的元素数目。

7.3.5 指针和 const

将 const 用于指针有一些很微妙的地方（指针看起来总是很微妙），我们来详细探讨一下。可以用两种不同的方式将 const 关键字用于指针。第一种方法是让指针指向一个常量对象，这样可以防止使用该指针来修改所指向的值。第二种方法是将指针本身声明为常量，这样可以防止改变指针指向的位置。下面来看细节。

首先，我们声明一个指向常量的指针 pt：

```
int age = 39;
const int * pt = &age;
```

该声明指出，pt 指向一个 const int（这里为 39），因此不能使用 pt 来修改这个值。换句话来说，*pt 的值为 const，不能被修改：

```
*pt += 1;          // INVALID because pt points to a const int
cin >> *pt;        // INVALID for the same reason
```

现在看一个微妙的问题。pt 的声明并不意味着它指向的值实际上就是一个常量，而只是意味着对 pt 而言，这个值是常量。例如，pt 指向 age，而 age 不是 const。可以直接通过 age 变量来修改 age 的值，但不能使用 pt 指针来修改它：

```
*pt = 20;          // INVALID because pt points to a const int
age = 20;          // VALID because age is not declared to be const
```

以前我们将常规变量的地址赋给过常规指针；而现在又将常规变量的地址赋给了指向 const 的指针。因此还有两种可能：将 const 变量的地址赋给指向 const 的指针、将 const 的地址赋给常规指针。这两种操作都可行吗？第一种可行，但第二种不可行：

```
const float g_earth = 9.80;
const float * pe = &g_earth;   // VALID

const float g_moon = 1.63;
float * pm = &g_moon;          // INVALID
```

对于第一种情况来说，既不能使用 g_earth 来修改值 9.80，也不能使用 pe 来修改。C++禁止第二种情况的原因很简单——如果将 g_moon 的地址赋给 pm，则可以使用 pm 来修改 g_moon 的值，这使得 g_moon 的 const 状态很荒谬，因此 C++禁止将 const 的地址赋给非 const 指针。如果读者非要这样做，可以使用强制类型转换来突破这种限制，详情请参阅第 15 章中对操作符 const_cast 的讨论。

如果将指针指向指针，则情况更复杂。前面已经讲过，假如涉及的是一级间接关系，则将非 const 指针赋给 const 指针是可以的：

```
int age = 39;          // age++ is a valid operation
int * pd = &age;       // *pd = 41 is a valid operation
const int * pt = pd;   // *pt = 42 is an invalid operation
```

不过，进入两级间接关系时，与一级间接关系一样将 const 和非 const 混合的指针赋值方式将不再安全。如果允许这样做，则可以编写这样的代码：

```
const int **pp2;
int *p1;
const int n = 13;
pp2 = &p1;  // not allowed, but suppose it were
*pp2 = &n;  // valid, both const, but sets p1 to point at n
*p1 = 10;   // valid, but changes const n
```

上述代码将非 const 地址（&p1）赋给了 const 指针（pp2），因此可以使用 p1 来修改 const 数据。因此，仅当只有一层间接关系（如指针指向基本数据类型）时，才可以将非 const 地址或指针赋给 const 指针。

记住：如果数据类型本身并不是指针，则可以将 const 数据或非 const 数据的地址赋给指向 const 的指针，但只能将非 const 数据的地址赋给非 const 指针。

假设有一个由 const 数据组成的数组：

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

则禁止将常量数组的地址赋给非常量指针将意味着不能将数组名作为参数传递给使用非常量形参的函数：

```
int sum (int arr[], int n);  // should have been const int arr[]
```

```
...
int j = sum (months, 12);    // not allowed
```

上述函数调用试图将 const 指针（months）赋给非 const 指针（arr），编译器将禁止这种函数调用。

尽可能使用 const

将指针参数声明为指向常量数据的指针有两条理由：

- 这样可以避免由于无意间修改数据而导致的编程错误。
- 使用 const 使得函数能够处理 const 和非 const 实参，否则将只能接受非 const 数据。

如果条件允许，则应将指针形参声明为指向 const 的指针。

为说明另一个微妙之处，请看下面的声明：

```
int age = 39;
const int * pt = &age;
```

第二个声明中的 const 只能防止修改 pt 指向的值（这里为 39），而不能防止修改 pt 的值。也就是说，可以将一个新地址赋给 pt：

```
int sage = 80;
pt = &sage; // okay to point to another location
```

但是仍然不能使用 pt 来修改它所指向的值（现在为 80）。

第二种使用 const 的方式使得无法修改指针的值：

```
int sloth = 3;
const int * ps = &sloth;      // a pointer to const int
int * const finger = &sloth;  // a const pointer to int
```

最后一个声明中，关键字 const 的位置与以前不同。这种声明格式使得 finger 只能指向 sloth，但允许使用 finger 来修改 sloth 的值。中间的声明不允许使用 ps 来修改 sloth 的值，但允许将 ps 指向另一个位置。简而言之，finger 和*ps 都是 const，而*finger 和 ps 不是（参见图 7.5）。

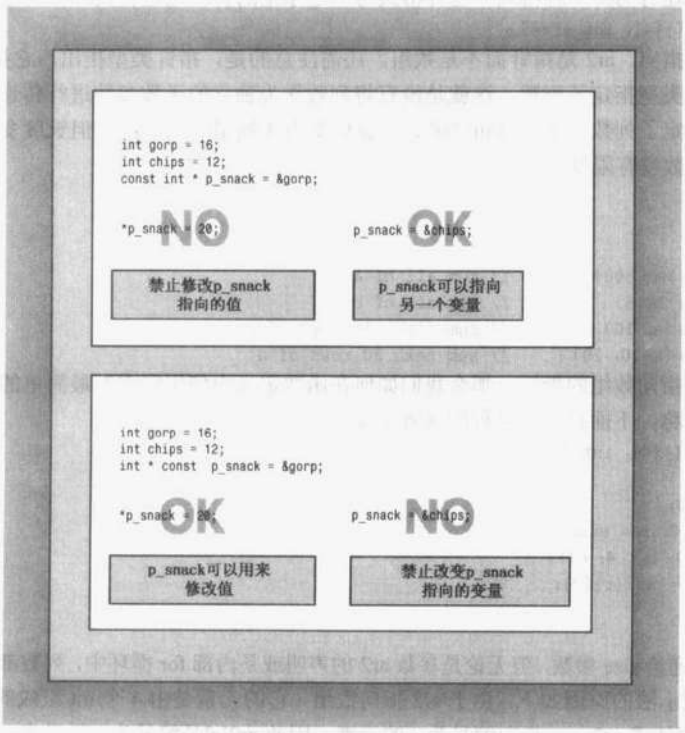


图 7.5　指向 const 的指针和 const 指针

如果愿意，还可以声明指向 const 对象的 const 指针：

```
double trouble = 2.0E30;
const double * const stick = &trouble;
```

其中，stick 只能指向 trouble，而 stick 不能用来修改 trouble 的值。简而言之，stick 和*stick 都是 const。

通常，将指针作为函数参数来传递时，可以使用指向 const 的指针来保护数据。例如，程序清单 7.5 中的 show_array() 的原型：

```
void show_array (const double ar[], int n);
```

在该声明中使用 const 意味着 show_array() 不能修改传递给它的数组中的值。只要有一层间接关系，就可以使用这种技术。例如，这里的数组元素是基本类型。但如果它们是指针或指向指针的指针，则不能使用 const。

7.4 函数和二维数组

为编写将二维数组作为参数的函数，必须牢记，数组名被视为其地址，因此，相应的形参是一个指针，就像一维数组一样。比较难处理的是如何正确地声明指针。例如，假设有下面的代码：

```
int data[3][4] = {{1, 2, 3, 4}, {9, 8, 7, 6}, {2, 4, 6 8}};
int total = sum(data, 3);
```

则 sum() 的原型是什么样的呢？函数为何将行数（3）作为参数，而将列数（4）作为参数呢？

data 是一个数组名，该数组有 3 个元素。第一个元素本身是一个数组，有 4 个 int 值组成。因此 data 的类型是指向由 4 个 int 组成的数组的指针，因此正确的原型如下：

```
int sum(int (*ar2) [4], int size);
```

其中的括号是必不可少的，因为声明：

```
int *ar2[4]
```

将声明一个由 4 个指向 int 的指针组成的数组，而不是由一个指向由 4 个 int 组成的数组的指针。另外，函数参数不能是数组。还有另外一种格式，这种格式与上述原型的含义完全相同，但可读性更强：

```
int sum(int ar2[][4], int size);
```

上述两个原型都指出，ar2 是指针而不是数组。还需注意的是，指针类型指出，它指向由 4 个 int 组成的数组。因此，指针类型指定了列数，这就是没有将列数作为独立的函数参数进行传递的原因。

由于指针类型指定了列数，因此 sum() 函数只能接受由 4 列组成的数组。但长度变量指定了行数，因此 sum() 对数组的行数没有限制：

```
int a[100][4];
int b[6][4];
...
int total1 = sum(a, 100);       // sum all of a
int total2 = sum(b, 6);         // sum all of b
int total3 = sum(a, 10);        // sum first 10 rows of a
int total4 = sum(a+10, 20);     // sum next 20 rows of a
```

由于参数 ar2 是指向数组的指针，那么我们如何在函数定义中使用它呢？最简单的方法是将 ar2 看作是一个二维数组的名称。下面是一个可行的函数定义：

```
int sum(int ar2[][4], int size)
{
    int total = 0;
    for (int r = 0; r < size; r++)
        for (c = 0; c < 4; c++)
            total += ar2[r][c];
    return total;
}
```

同样，行数被传递给 size 参数，但无论是参数 ar2 的声明或是内部 for 循环中，列数都是固定的——4 列。

可以使用数组表示法的原因如下。由于 ar2 指向数组（它的元素是由 4 个 int 组成的数组）的第一个元素（元素 0），因此表达式 ar2 + r 指向编号为 r 的元素。因此 ar2[r]是编号为 r 的元素。由于该元素本身就是一个由 4 个 int 组成的数组，因此 ar2[r]是由 4 个 int 组成的数组的名称。将下标用于数组名将得到一个

数组元素，因此 ar2[r][c]是由 4 个 int 组成的数组中的一个元素，是一个 int 值。必须对指针 ar2 执行两次解除引用，才能得到数据。最简单的方法是使用方括号两次：ar2[r][c]。然而，如果不考虑难看的话，也可以使用操作符*两次：

```
ar2[r][c] == *(*(ar2 + r) + c)  // same thing
```

为理解这一点，读者可以从内向外解析各个子表达式的含义：

```
ar2             // pointer to first row of an array of 4 int
ar2 + r         // pointer to row r (an array of 4 int)
*(ar2 + r)      // row r (an array of 4 int, hence the name of an array,
                // thus a pointer to the first int in the row, i.e., ar2[r]

*(ar2 +r) + c   // pointer int number c in row r, i.e., ar2[r] + c
*(*(ar2 + r) + c // value of int number c in row r, i.e. ar2[r][c]
```

sum()的代码在声明参数 ar2 时，没有使用 const，因为这种技术只能用于指向基本类型的指针，而 ar2 是指向指针的指针。

7.5　函数和 C-风格字符串

C-风格字符串由一系列字符组成，以空值字符结尾。前面介绍的大部分有关设计数组函数的知识也适用于字符串函数。

例如，将字符串作为参数时意味着传递的是地址，但可以使用 const 来禁止对字符串参数进行修改。不过，我们首先介绍一些有关字符串的特殊知识。

7.5.1　将 C-风格字符串作为参数的函数

假设要将字符串作为参数传递给函数，则表示字符串的方式有 3 种：

- char 数组。
- 用引号括起的字符串常量（也称字符串字面值）。
- 被设置为字符串的地址的 char 指针。

但上述 3 种选择的类型都是 char 指针（准确地说是 char*），因此可以将其作为字符串处理函数的参数：

```
char ghost[15] = "galloping";
char * str = "galumphing";
int n1 = strlen(ghost);             // ghost is &ghost[0]
int n2 = strlen(str);               // pointer to char
int n3 = strlen("gamboling");       // address of string
```

可以说是将字符串作为参数来传递，但实际传递的是字符串第一个字符的地址。这意味着字符串函数原型应将其表示字符串的形参声明为 char *类型。

C-风格字符串与常规 char 数组之间的一个重要区别是，字符串有内置的结束字符（前面讲过，包含字符，但不以空值字符结尾的 char 数组只是数组，而不是字符串）。这意味着不必将字符串长度作为参数传递给函数，而函数可以使用循环依次检查字符串中的每个字符，直到遇到结尾的空值字符为止。程序清单 7.9 演示了这种方法，使用一个函数来计算特定的字符在字符串中出现的次数。

程序清单 7.9　strgfun.cpp

```
// strgfun.cpp -- functions with a string argument
#include <iostream>
int c_in_str(const char * str, char ch);
int main()
{
   using namespace std;
   char mmm[15] = "minimum";    // string in an array
// some systems require preceding char with static to
// enable array initialization
```

```
    char *wail = "ululate";    // wail points to string

    int ms = c_in_str (mmm, 'm');
    int us = c_in_str (wail, 'u');
    cout << ms << " m characters in " << mmm << endl;
    cout << us << " u characters in " << wail << endl;
    return 0;
}

// this function counts the number of ch characters
// in the string str
int c_in_str (const char * str, char ch)
{
    int count = 0;

    while (*str)       // quit when *str is '\0'
    {
        if (*str == ch)
            count++;
        str++;         // move pointer to next char
    }
    return count;
}
```

下面是该程序的输出：

```
3 m characters in minimum
2 u characters in ululate
```

程序说明

由于程序清单 7.9 中的 c_int_str()函数不应修改原始字符串，因此它在声明形参 str 时使用了限定符 const。这样，如果错误地址函数修改了字符串的内容，编译器将捕获这种错误。当然，可以在函数头中使用数组表示法，而不声明 str：

```
int c_in_str (const char str[], char ch) // also okay
```

不过，使用指针表示法提醒读者注意，参数不一定必须是数组名，也可以是其他形式的指针。

该函数本身演示了处理字符串中字符的标准方式：

```
while (*str)
{
    statements
    str++;
}
```

开始，str 指向字符串的第一个字符，因此*str 表示的是第一个字符。例如，第一次调用该函数后，*str 的值将为 *m*——“minimum”的第一个字符。只要字符不为空值字符（\0），*str 就为非零值，因此循环将继续。在每轮循环的结尾处，表达式 str++将指针增加一个字节，使之指向字符串中的下一个字符。最终，str 将指向结尾的空值字符，使得*str 等于 0——空值字符的数字编码，从而结束循环。

7.5.2 返回 C-风格字符串的函数

现在，假设要编写一个返回字符串的函数。是的，函数无法返回一个字符串，但可以返回字符串的地址，这样做的效率更高。例如，程序清单 7.10 定义了一个名为 buildstr()的函数，该函数返回一个指针。该函数接受两个参数：一个字符和一个数字。函数使用 new 创建一个长度与数字参数相等的字符串，然后将每个元素都初始化为该字符。然后，返回指向新字符串的指针。

程序清单 7.10 strgback.cpp

```
// strgback.cpp -- a function that returns a pointer to char
#include <iostream>
```

```
char * buildstr (char c, int n);     // prototype
int main()
{
    using namespace std;
    int times;
    char ch;

    cout << "Enter a character: ";
    cin >> ch;
    cout << "Enter an integer: ";
    cin >> times;
    char *ps = buildstr (ch, times);
    cout << ps << endl;
    delete [] ps;                  // free memory
    ps = buildstr ('+', 20);        // reuse pointer
    cout << ps << "-DONE-" << ps << endl;
    delete [] ps;                  // free memory
    return 0;
}

// builds string made of n c characters
char * buildstr (char c, int n)
{
    char * pstr = new char[n + 1];
    pstr[n] = '\0';        // terminate string
    while (n-- > 0)
        pstr[n] = c;       // fill rest of string
    return pstr;
}
```

下面是该程序的运行情况：

```
Enter a character: V
Enter an integer: 46
VVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVV
++++++++++++++++++++-DONE-++++++++++++++++++++
```

程序说明

要创建包含 *n* 个字符的字符串，需要能够存储 *n*+1 个字符的空间，以便能够存储空值字符。因此，程序清单 7.10 中的函数请求分配 *n*+1 个字节的内存来存储该字符串，并将最后一个字节设置为空值字符，然后从后向前对数组进行填充。在程序清单 7.10 中，下面的循环：

```
while (n-- > 0)
    pstr[n] = c;
```

将循环 *n* 次，直到 *n* 减少到 0，这将填充 *n* 个元素。在最后一轮循环开始时，*n* 的值为 1。由于 *n*--意味着先使用这个值，然后将其递减，因此 while 循环测试条件将对 1 和 0 进行比较，发现测试为 true，循环继续。测试后，函数将 *n* 减为 0，因此 pstr[0]是最后一个被设置为 c 的元素。之所以从后向前（而不是从前向后）填充字符串，是为了避免使用额外的变量。从前向后填充的代码将与下面类似：

```
int i = 0;
while (i < n)
    pstr[i++] = c;
```

注意，变量 pstr 的作用域为 buildstr 函数内，因此该函数结束时，pstr（而不是字符串）使用的内存将被释放。但由于函数返回了 pstr 的值，因此程序仍可以通过 main()中的指针 ps 来访问新建的字符串。

当该字符串不再需要时，程序清单 7.10 中的程序使用 delete 释放该字符串占用的内存。然后，将 ps 指向为下一个字符串分配的内存块，然后释放它们。这种设计（让函数返回一个指针，该指针指向 new 分配的内存）的缺点是，程序员必须记住使用 delete。在第 12 章中，读者将知道 C++类如何使用构造函数和析构函数负责为您处理这些细节。

7.6 函数和结构

现在将注意力从数组转到结构。为结构编写函数比为数组编写函数要简单得多。虽然结构变量和数组一样，都可以存储多个数据项，但在涉及到函数时，结构变量的行为更接近于基本的单值变量。也就是说，与数组不同，结构将其数据组合成一个单独的实体，该实体被视为一个整体。前面讲过，可以将一个结构赋给另外一个结构。同样，也可以按值传递结构，就像普通变量那样。在这种情况下，函数将使用原始结构的拷贝。另外，函数也可以返回结构。与数组名就是数组第一个元素的地址不同的是，结构名只是结构的名称，要获得结构的地址，必须使用地址操作符&。在 C 语言和 C++中，都使用符号&来表示地址操作符；另外，C++还使用该操作符来表示引用变量，这将在第 8 章讨论。

使用结构编程时，最直接的方式是像处理基本类型那样来处理结构：也就是说，将结构作为参数传递，并在需要时将结构用作返回值使用。不过，按值传递结构有一个缺点。如果结构非常大，则复制结构将增加内存要求，降低系统运行的速度。出于这些原因（同时由于最初 C 语言不允许按值传递结构），许多 C 程序员倾向于传递结构的地址，然后使用指针来访问结构的内容。C++提供了第三种选择——按引用传递（将在第 8 章介绍）。下面介绍其他两种传递方式，首先介绍传递和返回整个结构。

7.6.1 传递和返回结构

当结构比较小时，按值传递结构最合理，下面来看两个使用这种技术的范例。第一个例子处理行程时间。有些地图将指出，从 Thunder Falls 到 Bingo 城需要 3 小时 50 分钟，而从 Bingo 城到 Gotesquo 需要 1 小时 25 分钟。对于这种时间，可以使用结构来表示——一个成员表示小时值，另一个成员表示分钟值。将两个时间加起来需要一些技巧，因为可能需要将分钟值转换为小时。例如，前面列出的两个时间的总和为 4 小时 75 分钟，应将它转换为 5 小时 15 分钟。下面开发用于表示时间值的结构，然后再开发一个函数，它接受两个这样的结构为参数，并返回表示参数的和的结构。

定义结构的工作很简单：

```
struct travel_time
{
    int hours;
    int mins;
};
```

接下来，看一下返回两个这种结构的总和的 sum()函数的原型。返回值的类型应为 travel_time，两个参数也应为这种类型。因此，原型应如下所示：

```
travel_time sum(travel_time t1, travel_time t2);
```

要将两个时间相加，应首先将分钟成员相加。然后通过整数除法（除数为 60）得到小时值，通过求模操作符（%）得到剩余的分钟数。程序清单 7.11 将在 sum()函数中使用了这种计算方式，并使用 show_time()函数来显示 travel_time 结构的内容。

程序清单 7.11 travel.cpp

```
// travel.cpp -- using structures with functions
#include <iostream>
struct travel_time
{
    int hours;
    int mins;
};
const int Mins_per_hr = 60;

travel_time sum(travel_time t1, travel_time t2);
void show_time(travel_time t);
```

```
int main()
{
    using namespace std;
    travel_time day1 = {5, 45};    // 5 hrs, 45 min
    travel_time day2 = {4, 55};    // 4 hrs, 55 min

    travel_time trip = sum (day1, day2);
    cout << "Two-day total: ";
    show_time (trip);

    travel_time day3= {4, 32};
    cout << "Three-day total: ";
    show_time (sum (trip, day3));

    return 0;
}

travel_time sum (travel_time t1, travel_time t2)
{
    travel_time total;

    total.mins = (t1.mins + t2.mins) % Mins_per_hr;
    total.hours = t1.hours + t2.hours +
            (t1.mins + t2.mins) / Mins_per_hr;
    return total;
}

void show_time (travel_time t)
{
    using namespace std;
    cout << t.hours << " hours, "
        << t.mins << " minutes\n";
}
```

其中，travel_time 就像是一个标准的类型名，可被用来声明变量、函数的返回类型和函数的参数类型。由于 total 和 t1 变量是 travel_time 结构，因此可以对它们使用句点成员操作符。由于 sum() 函数返回 travel_time 结构，因此可以将其用作 show_time() 函数的参数。由于在默认情况下，C++函数按值传递参数，因此函数调用 show_time（sum（trip，day3））将执行函数调用 sum（trip，day3），以获得其返回值。然后，show_time() 调用将 sum() 的返回值（而不是函数自身）传递给 show_time()。下面是该程序的输出：

```
Two-day total: 10 hours, 40 minutes
Three-day total: 15 hours, 12 minutes
```

7.6.2　另一个处理结构的函数范例

前面介绍的有关函数和 C++结构的大部分知识都可用于 C++类中，因此有必要介绍另一个范例。这次要处理的是空间，而不是时间。具体地说，这个例子将定义两个结构，用于表示两种不同的描述位置的方法，然后开发一个函数，将一种格式转换为另一种格式，并显示结果。这个例子用到的数学知识比前一个要多，但并不需要像学习数学那样学习 C++。

假设要描述屏幕上某点的位置，或地图上某点相对于原点的位置，则一种方法是指出该点相对于原点的水平偏移量和垂直偏移量。传统上，数学家使用 x 表示水平偏移量，使用 y 表示垂直偏移量（参见图 7.6）。x 和 y 一起构成了直角坐标（rectangular coordinates）。可以定义由两个坐标组成的结构来表示位置：

```
struct rect
{
    double x;          // horizontal distance from origin
    double y;          // vertical distance from origin
};
```

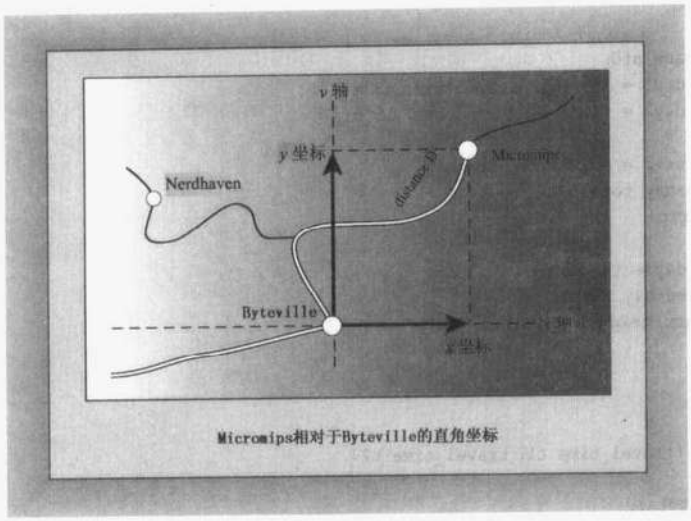


图 7.6 直角坐标

另一种描述点的位置的方法是，指出它偏离原点的距离和方向（例如，东偏北 40 度）。传统上，数学家从正水平轴开始按逆时针方向度量角度（参见图 7.7）。距离和角度一起构成了极坐标（polar coordinates）。可以定义另一个结构来表示这种位置：

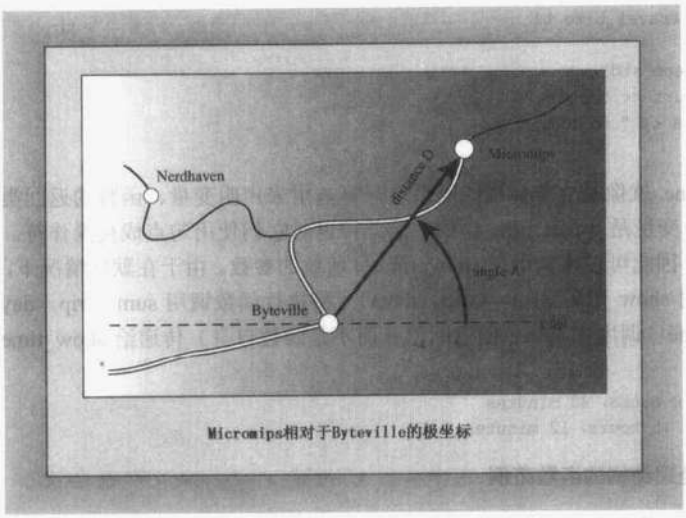


图 7.7 极坐标

```
struct polar
{
    double distance;   // distance from origin
    double angle;      // direction from origin
};
```

下面来创建一个显示 polar 结构的内容的函数。C++库中的数学函数假设角度的单位为弧度，因此应以弧度为单位来测量角度。但为了便于显示，我们将弧度值转换为角度值。这意味着需要将弧度值乘以 180/π——约为 57.29577951。该函数的代码如下：

```
// show polar coordinates, converting angle to degrees
void show_polar (polar dapos)
{
```

```
    using namespace std;
    const double Rad_to_deg = 57.29577951;

    cout << "distance = " << dapos.distance;
    cout << ", angle = " << dapos.angle * Rad_to_deg;
    cout << " degrees\n";
}
```

请注意，形参的类型为 polar。将一个 polar 结构传递给该函数时，该结构的内容将被复制到 dapos 结构中，函数随后将使用该拷贝完成工作。由于 dapos 是一个结构，因此该函数使用成员操作符句点（参见第 4 章）来标识结构成员。

接下来，让我们试着再前进一步，编写一个将直角坐标转换为极坐标的函数。该函数接受一个 rect 参数，并返回一个 polar 结构。这需要使用数学库中的函数，因此程序必须包含头文件 math.h。另外，在有些系统中，还必须命令编译器载入数学库（参见第 1 章）。可以根据毕达哥拉斯定理，使用水平和垂直坐标来计算距离：

```
distance = sqrt (x * x + y * y)
```

数学库中的 atan2() 函数可根据 x 和 y 的值计算角度：

```
angle = atan2 (y, x)
```

（还有一个 atan() 函数，但它不能区分 180 度之内和之外的角度。在数学函数中，这种不确定性与在生存手册中一样不受人欢迎）。

有了这些公式后，便可以这样编写该函数：

```
// convert rectangular to polar coordinates
polar rect_to_polar (rect xypos)  // type polar
{
    polar answer;

    answer.distance =
        sqrt (xypos.x * xypos.x + xypos.y * xypos.y);
    answer.angle = atan2 (xypos.y, xypos.x);
    return answer;      // returns a polar structure
}
```

编写好函数后，程序的其他部分编写起来就非常简单了。程序清单 7.12 列出了程序的代码。

程序清单 7.12　atrctfun.cpp

```
// strctfun.cpp -- functions with a structure argument
#include <iostream>
#include <cmath>
// structure declarations
struct polar
{
    double distance;   // distance from origin
    double angle;      // direction from origin
};
struct rect
{
    double x;          // horizontal distance from origin
    double y;          // vertical distance from origin
};

// prototypes
polar rect_to_polar (rect xypos);
void show_polar (polar dapos);

int main()
{
    using namespace std;
    rect rplace;
```

```
    polar pplace;

    cout << "Enter the x and y values: ";
    while (cin >> rplace.x >> rplace.y)  // slick use of cin
    {
        pplace = rect_to_polar (rplace);
        show_polar (pplace);
        cout << "Next two numbers (q to quit): ";
    }
    cout << "Done.\n";
    return 0;
}

// convert rectangular to polar coordinates
polar rect_to_polar (rect xypos)
{
    using namespace std;
    polar answer;

    answer.distance =
        sqrt (xypos.x * xypos.x + xypos.y * xypos.y);
    answer.angle = atan2 (xypos.y, xypos.x);
    return answer;      // returns a polar structure
}

// show polar coordinates, converting angle to degrees
void show_polar (polar dapos)
{
    using namespace std;
    const double Rad_to_deg = 57.29577951;

    cout << "distance = " << dapos.distance;
    cout << ", angle = " << dapos.angle * Rad_to_deg;
    cout << " degrees\n";
}
```

注意：有些 C++实现仍使用头文件 math.h，而不是新的 cmath。有些编译器仅当被明确指示后，才会搜索数学库。例如，较早的 g++版本使用下面这样的命令行：

```
g++ structfun.C -lm
```

下面是该程序的运行情况：

```
Enter the x and y values: 30 40
distance = 50, angle = 53.1301 degrees
Next two numbers (q to quit): -100 100
distance = 141.421, angle = 135 degrees
Next two numbers (q to quit): q
```

程序说明

程序清单 7.12 中的两个函数已经在前面讨论了，因此下面复习一下该程序如何使用 cin 来控制 while 循环：

```
while (cin >> rplace.x >> rplace.y)
```

前面讲过，cin 是 istream 类的一个对象。抽取操作符（>>）被设计成使得 cin>>rplace.x 也是一个 istream 对象。正如第 11 章将介绍的，类操作符是使用函数实现的。使用 cin>>rplace.x 时，程序将调用一个函数，该函数返回一个 istream 值。将抽取操作符用于 cin>>rplace.x 对象（就像 cin>>rplace.x>>rplace.y 这样），也将获得一个 istream 对象。因此，整个 while 循环的测试表达式的最终结果为 cin，而 cin 被用于测试表达式中时，将根据输入是否成功，被转换为 bool 值 true 或 false。例如，在程序清单 7.12 中的循环中，cin 期望用户输入两个数字，如果用户输入了 q（前面的输出范例就是这样做的），cin>>将知道 q 不是数字，从而

将 q 留在输入队列中，并返回一个将被转换为 fasle 的值，导致循环结束。

请将这种读取数字的方法与下面更为简单的方法进行比较：

```
for (int i = 0; i < limit; i++)
{
     cout << "Enter value #" << (i + 1) << ": ";
     cin >> temp;
     if (temp < 0)
      break;
     ar[i] = temp;
     }
```

要提早结束该循环，可以输入一个负值。这样将输入限制为非负值。这种限制符合某些程序的需要，但通常需要一种不会将某些数值排除在外的、终止循环的方式。将 cin>>用作测试条件消除了这种限制，因为它接受任何有效的数字输入。在需要使用循环来输入数字时，别忘了考虑使用这种方式。另外请记住，非数字输入将设置一个错误条件，禁止进一步读取输入。如果程序在输入循环后还需要进行输入，则必须使用 cin.clear() 重置输入，然后还可能需要通过读取不合法的输入来丢弃它们。程序清单 7.7 演示了这些技术。

7.6.3　传递结构的地址

假设要传递结构的地址而不是整个结构以节省时间和空间，则需要重新编写前面的函数，使用指向结构的指针。首先来看一看如何重新编写 show_polar() 函数。需要修改 3 个地方：

- 调用函数时，将结构的地址（&pplace）而不是结构本身（pplace）传递给它。
- 将形参声明为指向 polar 的指针，即 polar *类型。由于函数不应该修改结构，因此使用 const 修饰符。
- 由于形参是指针而不是结构，因此应使用间接成员操作符（->），而不是成员操作符（句点）。

完成上述修改后，该函数如下所示：

```
// show polar coordinates, converting angle to degrees
void show_polar (const polar * pda)
{
   using namespace std;
   const double Rad_to_deg = 57.29577951;

   cout << "distance = " << pda->distance;
   cout << ", angle = " << pda->angle * Rad_to_deg;
   cout << " degrees\n";
}
```

接下来对 rect_to_polar 进行修改。由于原来的 rect_to_polar 函数返回一个结构，因此修改工作更复杂些。为了充分利用指针的效率，应使用指针，而不是返回值。为此，需要将两个指针传递给该函数。第一个指针指向要转换的结构，第二个指针指向存储转换结果的结构。函数不返回一个新的结构，而是修改调用函数中已有的结构。因此，虽然第一个参数是 const 指针，但第二个参数却不是。也可以像修改函数 show_polar() 修改这个函数。程序清单 7.13 列出了修改后的程序。

程序清单 7.13　strctptr.cpp

```
// strctptr.cpp -- functions with pointer to structure arguments
#include <iostream>
#include <cmath>

// structure templates
struct polar
{
   double distance;   // distance from origin
   double angle;      // direction from origin
};
struct rect
{
```

```
    double x;          // horizontal distance from origin
    double y;          // vertical distance from origin
};

// prototypes
void rect_to_polar (const rect * pxy, polar * pda);
void show_polar (const polar * pda);

int main()
{
    using namespace std;
    rect rplace;
    polar pplace;

    cout << "Enter the x and y values: ";
    while (cin >> rplace.x >> rplace.y)
    {
        rect_to_polar (&rplace, &pplace);    // pass addresses
        show_polar (&pplace);                // pass address
        cout << "Next two numbers (q to quit): ";
    }
    cout << "Done.\n";
    return 0;
}

// show polar coordinates, converting angle to degrees
void show_polar (const polar * pda)
{
    using namespace std;
    const double Rad_to_deg = 57.29577951;

    cout << "distance = " << pda->distance;
    cout << ", angle = " << pda->angle * Rad_to_deg;
    cout << " degrees\n";
}

// convert rectangular to polar coordinates
void rect_to_polar (const rect * pxy, polar * pda)
{
    using namespace std;
    pda->distance =
        sqrt (pxy->x * pxy->x + pxy->y * pxy->y);
    pda->angle = atan2 (pxy->y, pxy->x);
}
```

注意：有些 C++实现仍使用头文件 math.h，而不是新的 cmath。有些编译器需要明确指示，才会搜索数学库。

从用户的角度来说，程序清单 7.13 的行为与程序清单 7.12 相同。它们之间的差别在于，程序清单 7.12 使用的是结构拷贝，而程序清单 7.13 使用的是指向原始结构的指针。

7.7 函数和 string 对象

虽然 C-风格字符串和 string 对象的用途几乎相同，但与数组相比，string 对象与结构的更相似。例如，可以将一个结构赋给另一个结构，也可以将一个对象赋给另一个对象。可以将结构作为完整的实体传递给函数，也可以将对象作为完整的实体进行传递。如果需要多个字符串，可以声明一个 string 对象数组，而

不是二维 char 数组。

程序清单 7.14 提供了一个小型范例，它声明了一个 string 对象数组，并将该数组传递给一个函数以显示其内容。

程序清单 7.14　topfive.cpp

```
// topfive.cpp -- handling an array of string objects
#include <iostream>
#include <string>
using namespace std;
const int SIZE = 5;
void display(const string sa[], int n);
int main()
{
    string list[SIZE];     // an array holding 5 string object
    cout << "Enter your " << SIZE << " favorite astronomical sights:\n";
    for (int i = 0; i < SIZE; i++)
    {
        cout << i + 1 << ": ";
        getline(cin,list[i]);
    }

    cout << "Your list:\n";
    display(list, SIZE);

    return 0;
}

void display(const string sa[], int n)
{
    for (int i = 0; i < n; i++)
        cout << i + 1 << ": " << sa[i] << endl;
}
```

注意：有些老式 Visual C++存在一个 bug：输入语句和输出语句不能同步。使用这些版本时，用户可能必须在计算机提示之前输入响应。

下面是该程序的运行情况：

```
Enter your 5 favorite astronomical sights:
1: Orion Nebula
2: M13
3: Saturn
4: Jupiter
5: Moon
Your list:
1: Orion Nebula
2: M13
3: Saturn
4: Jupiter
5: Moon
```

对于该范例，需要指出的一点是，除函数 getline()外，该程序像对待内置类型（如 int）一样对待 string 对象。如果需要 string 数组，只需使用通常的数组声明格式即可：

```
string list[SIZE];     // an array holding 5 string object
```

这样，数组 list 的每个元素都是一个 string 对象，可以像下面这样使用它：

```
getline(cin,list[i]);
```

同样，形参 sa 是一个指向 string 对象的指针，因此 sa[i]是一个 string 对象，可以像下面这样使用它：

```
cout << i + 1 << ": " << sa[i] << endl;
```

7.8 递归

下面介绍一些完全不同的内容。C++函数有一种有趣的特点——可以调用自己（不过，与C语言不同的是，C++不允许main()调用自己），这种功能被称为递归。尽管递归在特定的编程（例如人工智能）中是一种重要的工具，但这里只简单地介绍一下它是如何工作的。

7.8.1 包含一个递归调用的递归

如果递归函数调用自己，则被调用的函数也将调用自己，这将无限循环下去，除非代码中包含终止调用链的内容。通常的方法将递归调用放在if语句中。例如，void类型的递归函数recurs()的代码如下：

```
void recurs(argumentlist)
{
    statementsl
    if(test)
        recurs(arguments)
    statements2
}
```

test最终将为false，调用链将断开。

递归调用将导致一系列有趣的事件。只要if语句为true，每个recurs()调用都将执行statements 1，然后再调用recurs()，而不会执行statements 2。当if语句为false时，当前调用将执行statements2。当前调用结束后，程序控制权将返回给调用它的recurs()，而该recurs()将执行其statements2部分，然后结束，并将控制权返回给前一个调用，依此类推。因此，如果recurs()进行了5次递归调用，则第一个statements1部分将按函数调用的顺序执行5次，然后statements2部分将以与函数调用相反的顺序执行5次。进入5层递归后，程序将沿进入的路径返回。程序清单7.15演示了这种行为。

程序清单7.15 recur.cpp

```
// recur.cpp -- using recursion
#include <iostream>
void countdown(int n);

int main()
{
   countdown(4);          // call the recursive function
   return 0;
}

void countdown(int n)
{
   using namespace std;
   cout << "Counting down ... " << n << endl;
   if(n > 0)
      countdown(n-1);     // function calls itself
   cout << n << ": Kaboom!\n";
}
```

下面是该程序的输出：

```
Counting down ... 4    //level 1: adding levels of recursion
Counting down ... 3    //level 2
Counting down ... 2    //level 3
Counting down ... 1    //level 4
Counting down ... 0    //level 5: final recursive call
0: Kaboom!             //level 5: beginning to back out
1: Kaboom!             //level 4
```

```
2: Kaboom!              //level 3
3: Kaboom!              //level 2
4: Kaboom!              //level 1
```

注意，每个递归调用都创建自己的一套变量，因此当程序到达第 5 次调用时，将有 5 个独立的 *n* 变量，其中每个变量的值都不同。为验证这一点，读者可以修改程序清单 7.15，使之显示 *n* 的地址和值：

```
cout << "Counting down ... " << n << " (n at " << &n << ")" << endl;
...
cout << n << ": Kaboom!"; << "        (n at " << &n << ")" << endl;
```

经过上述修改后，该程序的输出将与下面类似：

```
Counting down ... 4 (n at 0012FE0C)
Counting down ... 3 (n at 0012FD34)
Counting down ... 2 (n at 0012FC5C)
Counting down ... 1 (n at 0012FB84)
Counting down ... 0 (n at 0012FAAC)
0: Kaboom!          (n at 0012FAAC)
1: Kaboom!          (n at 0012FB84)
2: Kaboom!          (n at 0012FC5C)
3: Kaboom!          (n at 0012FD34)
4: Kaboom!          (n at 0012FE0C)
```

注意，在一个内存单元（内存地址为 0012FE0C），存储的 *n* 值为 4；在另一个内存单元（内存地址为 0012FD34），存储的 *n* 值为 3；等等。

7.8.2　包含多个递归调用的递归

在需要将一项工作不断分为两项较小的、类似的工作时，递归非常有用。例如，请考虑使用这种方法来绘制标尺的情况。标出两端，找到中点并将其标出。然后将同样的操作用于标尺的左半部分和右半部分。如果要进一步细分，可将同样的操作用于当前的每一部分。递归方法有时被称为分而治的策略（divide-and-conquer strategy）。程序清单 7.16 使用递归函数 subdivide()演示了这种方法，该函数使用一个字符串，该字符串除两端为 | 字符外，其他全部为空格。Main 程序使用循环调用 subdivide()函数 6 次，每次将递归层编号加 1，并打印得到的字符串。这样，每行输出表示一层递归。该程序使用限定符 std::而不是编译指令 using，以提醒读者还可以采取这种方式。

程序清单 7.16　ruler.cpp

```
// ruler.cpp -- using recursion to subdivide a ruler
#include <iostream>
const int Len = 66;
const int Divs = 6;
void subdivide(char ar[], int low, int high, int level);
int main()
{
   char ruler[Len];
   int i;
   for (i = 1; i < Len - 2; i++)
      ruler[i] = ' ';
   ruler[Len - 1] = '\0';
   int max = Len - 2;
   int min = 0;
   ruler[min] = ruler[max] = '|';
   std::cout << ruler << std::endl;
   for (i = 1; i <= Divs; i++)
   {
      subdivide(ruler,min,max, i);
      std::cout << ruler << std::endl;
      for (int j = 1; j < Len - 2; j++)
         ruler[j] = ' '; // reset to blank ruler
   }
```

```
    return 0;
}

void subdivide(char ar[], int low, int high, int level)
{
    if (level == 0)
        return;
    int mid = (high + low) / 2;
    ar[mid] = '|';
    subdivide(ar, low, mid, level - 1);
    subdivide(ar, mid, high, level - 1);
}
```

下面是程序清单 7.16 中程序的输出：

```
|                                                               |
|                               |                               |
|               |               |               |               |
|       |       |       |       |       |       |       |       |
|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
```

程序说明

在程序清单 7.16 中，subdivide()函数使用变量 level 来控制递归层。函数调用自身时，将把 level 减 1，当 level 为 0 时，该函数将不再调用自己。注意，subdivide()调用自己两次，一次针对左半部分，另一次针对右半部分。最初的中点被用作一次调用的右端点和另一次调用的左端点。请注意，调用次数将呈几何级数增长。也就是说，调用一次导致两个调用，然后导致 4 个调用，再导致 8 个调用，依此类推。这就是 6 层调用能够填充 64 个元素的原因（2^6=64）。这将不断导致函数调用数（以及存储的变量数）翻倍，因此如果要求的递归层次很多，这种递归方式将是一种糟糕的选择；然而，如果递归层次较少，这将是一种精致而简单的选择。

7.9 函数指针

如果未提到函数指针，则对 C 或 C++函数的讨论将是不完整的。我们将大致介绍一下这个主题，将完整的介绍留给更高级的图书。

与数据项相似，函数也有地址。函数的地址是存储其机器语言代码的内存的开始地址。通常，这些地址对用户而言，既不重要，也没有什么用处，但对程序而言，却很有用。例如，可以编写将另一个函数的地址作为参数的函数。这样第一个函数将能够找到第二个函数，并运行它。与直接调用另一个函数相比，这种方法很笨拙，但它允许在不同的时间传递不同函数的地址，这意味着可以在不同的时间使用不同的函数。

7.9.1 函数指针的基础知识

首先通过一个例子来阐释这一过程。假设要设计一个名为 estimate()的函数，估算编写指定行数的代码所需的时间，并且希望不同的程序员都将使用该函数。对于所有的用户来说，estimate()中一部分代码都是相同的，但该函数允许每个程序员提供自己的算法来估算时间。为实现这种目标，采用的机制是，将程序员要使用的算法函数的地址传递给 estimate()。为此，必须能够完成下面的工作：

- 获取函数的地址。

- 声明一个函数指针。
- 使用函数指针来调用函数。

1. 获取函数的地址

获取函数的地址很简单：只要使用函数名（后面不跟参数）即可。也就是说，如果 think() 是一个函数，则 think 就是该函数的地址。要将函数作为参数进行传递，必须传递函数名。一定要区分传递的是函数的地址还是函数的返回值：

```
process (think) ;   // passes address of think()to process()
thought (think()) ; // passes return value of think()to thought()
```

process() 调用使得 process() 函数能够在其内部调用 think() 函数。thought() 调用首先调用 think() 函数，然后将 think() 的返回值传递给 thought() 函数。

2. 声明函数指针

声明指向某种数据类型的指针时，必须指定指针指向的类型。同样，声明指向函数的指针时，也必须指定指针指向的函数类型。这意味着声明应指定函数的返回类型以及函数的特征标（参数列表）。也就是说，声明应像函数原型那样指出有关函数的信息。例如，假设 Pam leCoder 编写了一个估算时间的函数，其原型如下：

```
double pam (int) ; // prototype
```

则正确的指针类型声明如下：

```
double (*pf) (int);     // pf points to a function that takes
                        // one int argument and that
                        // returns type double
```

这与 pam() 声明类似，这是将 pam 替换为了（*pf）。由于 pam 是函数，因此（*pf）也是函数。而如果（*pf）是函数，则 pf 就是函数指针。

提示：通常，要声明指向特定类型的函数的指针，可以首先编写这种函数的原型，然后用（*pf）替换函数名。这样 pf 就是这类函数的指针。

为提供正确的操作符优先级，必须在声明中使用括号将*pf 括起。括号的优先级比*操作符高，因此*pf（int）意味着 pf() 是一个返回指针的函数，而（*pf）（int）意味着 pf 是一个指向函数的指针：

```
double (*pf) (int) ; // pf points to a function that returns double
double *pf (int) ;   // pf()a function that returns a pointer-to-double
```

正确地声明 pf 后，便可以将相应函数的地址赋给它：

```
double pam (int) ;
double (*pf) (int) ;
pf = pam;           // pf now points to the pam()function
```

注意，pam() 的特征标（signatures）和返回类型必须与 pf 相同。如果不相同，编译器将拒绝这种赋值：

```
double ned (double) ;
int ted (int) ;
double (*pf) (int) ;
pf = ned;         // invalid -- mismatched signature
pf = ted;         // invalid -- mismatched return types
```

现在回过头来看一下前面提到的 estimate() 函数。假设要将将要编写的代码行数和估算算法（如 pam() 函数）的地址传递给它，则其原型将如下：

```
void estimate (int lines, double (*pf) (int) ) ;
```

上述声明指出，第二个参数是一个函数指针，它指向的函数接受一个 int 参数，并返回一个 double 值。要让 estimate() 使用 pam() 函数，需要将 pam() 的地址传递给它：

```
estimate (50, pam) ; // function call telling estimate()to use pam()
```

显然，使用函数指针时，比较棘手的是编写原型，而传递地址则非常简单。

3. 使用指针来调用函数

现在进入最后一步，即使用指针来调用被指向的函数。线索来自指针声明。前面讲过，（*pf）扮演的

角色与函数名相同，因此使用（*pf）时，只需将它看作函数名即可：

```
double pam(int);
double (*pf)(int);
pf = pam;              // pf now points to the pam()function
double x = pam(4);   // call pam()using the function name
double y = (*pf)(5); // call pam()using the pointer pf
```

实际上，C++也允许像使用函数名那样使用 pf：

```
double y = pf(5);    // also call pam()using the pointer pf
```

第一种格式虽然不太好看，但它给出了强有力的提示——代码正在使用函数指针。

历史与逻辑

真是非常棒的句法！为何 pf 和（*pf）等价呢？一种学派认为，由于 pf 是函数指针，而*pf 是函数，因此应将（*pf）()用作函数调用。另一种学派认为，由于函数名是指向该函数的指针，指向函数的指针的行为应与函数名相似，因此应将 pf()用作函数调用使用。C++进行了折衷——这 2 种方式都是正确的，或者至少是允许的，虽然它们在逻辑上是互相冲突的。在认为这种折衷粗糙之前，应该想到，容忍逻辑上无法自圆其说的观点正是人类思维活动的特点。

7.9.2 函数指针示例

程序清单 7.17 演示了如何使用函数指针。它两次调用 estimate()函数，一次传递 betsy()函数的地址，另一次则传递 pam()函数的地址。在第一种情况下，estimate()使用 betsy()计算所需的小时数；在第二种情况下，estimate()使用 pam()进行计算。这种设计有助于今后的程序开发。当 Ralph 为估算时间而开发自己的算法时，他将不需要重新编写 estimate()。相反，他只需提供自己的 ralph()函数，并确保该函数的特征标和返回类型正确即可。当然，重新编写 estimate()也并不是一件非常困难的工作，但同样的原则也适用于更复杂的代码。另外，函数指针方式使得 Ralph 能够修改 estimate()的行为，虽然他接触不到 estimate()的源代码。

程序清单 7.17 fun_ptr.cpp

```
// fun_ptr.cpp -- pointers to functions
#include <iostream>
double betsy(int);
double pam(int);

// second argument is pointer to a type double function that
// takes a type int argument
void estimate(int lines, double (*pf)(int));

int main()
{
    using namespace std;
    int code;

    cout << "How many lines of code do you need? ";
    cin >> code;
    cout << "Here's Betsy's estimate:\n";
    estimate(code, betsy);
    cout << "Here's Pam's estimate:\n";
    estimate(code, pam);
    return 0;
}

double betsy(int lns)
{
    return 0.05 * lns;
}
```

```
double pam (int lns)
{
   return 0.03 * lns + 0.0004 * lns * lns;
}

void estimate (int lines, double (*pf) (int))
{
   using namespace std;
   cout << lines << " lines will take ";
   cout << (*pf) (lines) << " hour (s) \n";
}
```

下面是两次运行该程序的情况：

```
How many lines of code do you need? 30
Here's Betsy's estimate:
30 lines will take 1.5 hour (s)
Here's Pam's estimate:
30 lines will take 1.26 hour (s)

How many lines of code do you need? 100
Here's Betsy's estimate:
100 lines will take 5 hour (s)
Here's Pam's estimate:
100 lines will take 7 hour (s)
```

7.10　总结

函数是 C++的编程模块。要使用函数，必须提供定义和原型，并调用该函数。函数定义是实现函数功能的代码；函数原型描述了函数的接口：传递给函数的值的数目和种类以及函数的返回类型。函数调用使得程序将参数传递给函数，并执行函数的代码。

在默认情况下，C++函数按值传递参数。这意味着函数定义中的形参是新的变量，它们被初始化为函数调用所提供的值。因此，C++函数通过使用拷贝，保护了原始数据的完整性。

C++将数组名参数视为数组第一个元素的地址。从技术上讲，这仍然是按值传递的，因为指针是原始地址的拷贝，但函数将使用指针来访问原始数组的内容。当且仅当声明函数的形参时，下面两个声明才是等价的：

```
TypeName arr[];
TypeName * arr;
```

这两个声明都表明，arr 是指向 typeName 的指针，但在编写函数代码时，可以像使用数组名那样使用 arr 来访问元素：arr[i]。即使在传递指针时，也可以将形参声明为 const 指针，来保护原始数据的完整性。由于传递数据的地址时，并不会传输有关数组长度的信息，因此通常将数组长度作为独立的参数来传递。

C++提供了 3 种表示 C-风格字符串的方法：字符数组、字符串常量和字符串指针。它们的类型都是 char*（char 指针），因此被作为 char*类型参数传递给函数。C++使用空值字符（\0）来结束字符串，因此字符串函数检测空值字符来确定字符串的结尾。

C++还提供了 string 类，用于表示字符串。函数可以接受 string 对象作为参数以及将 string 对象作为返回值。string 类的方法 size() 可用于判断其存储的字符串的长度。

C++处理结构的方式与基本类型完全相同，这意味着可以按值传递结构，并将其用作函数返回类型。不过，如果结构非常大，则传递结构指针的效率将更高，同时函数能够使用原始数据。

C++函数可以是递归的，也就是说，函数代码中可以包括对函数本身的调用。

C++函数名与函数地址的作用相同。通过将函数指针作为参数，可以传递要调用的函数的名称。

7.11 复习题

1. 使用函数的3个步骤是什么？

2. 请创建与下面的描述匹配的函数原型：

a. igor()没有参数，且没有返回值。

b. tofu()接受一个int参数，并返回一个float。

c. mpg()接受两个double参数，并返回一个double。

d. summation()将long数组名和数组长度作为参数，并返回一个long值。

e. doctor()接受一个字符串参数（不能修改该字符串），并返回一个double值。

f. ofcourse()将boss结构作为参数，不返回值。

g. plot()将map结构的指针作为参数，并返回一个字符串。

3. 编写一个接受3个参数的函数：int数组名、数组长度和一个int值，并将数组的所有元素都设置为该int值。

4. 编写一个接受3个参数的函数：指向数组区间中第一个元素的指针、指向数组区间最后一个元素后面的指针以及一个int值，并将数组中的每个元素都设置为该int值。

5. 编写将double数组名和数组长度作为参数，并返回该数组中最大值的函数。该函数不应修改数组的内容。

6. 为什么不对类型为基本类型的函数参数使用const限定符？

7. C++程序可使用哪3种C-风格字符串格式？

8. 编写一个函数，其原型如下：

int replace（char * str，char c1，char c2）；

该函数将字符串中所有的c1都替换为c2，并返回替换次数。

9. 表达式*"pizza"的含义是什么？"taco"[2]呢？

10. C++允许按值传递结构，也允许传递结构的地址。如果glitz是一个结构变量，如何按值传递它？如何传递它的地址？这两种方法有何利弊？

11. 函数judge()的返回类型为int，它将这样一个函数的地址作为参数：将const char指针作为参数，并返回一个int值。请编写judge()函数的原型。

7.12 编程练习

1. 编写一个程序，不断要求用户输入两个数，直到其中的一个为0。对于每两个数，程序将使用一个函数来计算它们的调和平均数，并将结果返回给main()，而后者将报告结果。调和平均数指的是倒数平均值的倒数，计算公式如下：

调和平均数=2.0 * x * y /（x+y）

2. 编写一个程序，要求用户输入最多10个高尔夫成绩，并将其存储在一个数组中。程序允许用户提早结束输入，并在一行上显示所有成绩，然后报告平均成绩。请使用3个数组处理函数来分别进行输入、显示和计算平均成绩。

3. 下面是一个结构声明：

```
struct box
{
    char maker[40];
    float height;
    float width;
```

```
    float length;
    float volume;
};
```

a．编写一个函数，按值传递 box 结构，并显示每个成员的值。

b．编写一个函数，传递 box 结构的地址，并将 volume 成员设置为其他三维长度的乘积。

c．编写一个使用这两个函数的简单程序。

4．许多州的彩票发行机构都使用如程序清单 7.4 所示的简单彩票玩法的变体。在这些玩法中，玩家从一组被称为域号码（field number）的号码中选择几个。例如，可以从域号码 1～47 中选择 5 个号码；还可以从第二个区间（如 1～27）中选择一个号码（称为特选号码）。要赢得头奖，必须正确猜中所有的号码。中头奖的几率是选中所有域号码的几率与选中特选号码几率的乘积。例如，在这个例子中，中头奖的几率是从 47 个号码中正确选取 5 个号码的几率与从 27 个号码中正确选择 1 个号码的几率的乘积。请修改程序清单 7.4，以计算中得这种彩票头奖的几率。

5．定义一个递归函数，接受一个整数参数，并返回该参数的阶乘。前面讲过，3 的阶乘写作 3!，等于 3*2!，依此类推；而 0!被定义为 1。通用的计算公式是，如果 n 大于零，则 $n!=n*(n-1)!$。在程序中对该函数进行测试，程序使用循环让用户输入不同的值，程序将报告这些值的阶乘。

6．编写一个程序，它使用下列函数：

Fill_array()将一个 double 数组的名称和长度作为参数。它提示用户输入 double 值，并将这些值存储到数组中。当数组被填满或用户输入了非数字时，输入将停止，并返回实际输入了多少个数字。

Show_array()将一个 double 数组的名称和长度作为参数，并显示该数组的内容。

Reverse-array()将一个 double 数组的名称和长度作为参数，并将存储在数组中的值的顺序反转。

程序将使用这些函数来填充数组，然后显示数组；反转数组，然后显示数组；反转数组中除第一个和最后一个元素之外的所有元素，然后显示数组。

7．修改程序清单 7.7 中的 3 个数组处理函数，使之使用两个指针参数来表示区间。fill_array()函数不返回实际读取了多少个数字，而是返回一个指针，该指针指向最后被填充的位置；其他的函数可以将该指针作为第二个参数，以标识数据结尾。

8．这个练习让您编写处理数组和结构的函数。下面是程序的框架，请提供其中描述的函数，以完成该程序。

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

const int SLEN = 30;
struct student {
    char fullname[SLEN];
    char hobby[SLEN];
    int ooplevel;
};
// getinfo()has two arguments: a pointer to the first element of
// an array of student structures and an int representing the
// number of elements of the array. The function solicits and
// stores data about students. It terminates input upon filling
// the array or upon encountering a blank line for the student
// name. The function returns the actual number of array elements
// filled.
int getinfo (student pa[], int n);

// display1()takes a student structure as an argument
// and displays its contents
void display1 (student st);

// display2()takes the address of student structure as an
// argument and displays the structure's contents
void display2 (const student * ps);
```

```
// display3()takes the address of the first element of an array
// of student structures and the number of array elements as

// arguments and displays the contents of the structures
void display3 (const student pa[], int n);

int main()
{
    cout << "Enter class size: ";
    int class_size;
    cin >> class_size;
    while (cin.get()!= '\n')
        continue;

   student * ptr_stu = new student[class_size];
   int entered = getinfo (ptr_stu, class_size);
   for (int i = 0; i < entered; i++)
   {
      display1 (ptr_stu[i]);
      display2 (&ptr_stu[i]);
   }
   display3 (ptr_stu, entered);
   delete [] ptr_stu;
   cout << "Done\n";
   return 0;
}
```

9. 设计一个名为 calculate()的函数，它接受两个 double 值和一个指向函数的指针，而被指向的函数接受两个 double 参数，并返回一个 double 值。calculate()函数的类型也是 double，并返回被指向的函数使用 calculate()的两个 double 参数计算得到的值。例如，假设 add()函数的定义如下：

```
double add (double x, double y)
{
     return x + y;
}
```

则下述代码中的函数调用：

```
double q = calculate (2.5, 10.4, add);
```

将导致 calculate()把 2.5 和 10.4 传递给 add()函数，并返回 add()的返回值（12.9）。

请编写一个程序，它调用上述两个函数和至少另一个与 add()类似的函数。该程序使用循环来让用户成对地输入数字。对于每对数字，程序都使用 calculate()来调用 add()和至少一个其他的函数。如果读者爱冒险，可以尝试创建一个指针数组，其中的指针指向 add()样式的函数，并编写一个循环，使用这些指针连续让 calculate()调用这些函数。提示：下面是声明这种指针数组的方式，其中包含 3 个指针：

```
double (*pf[3]) (double, double);
```

可以采用数组初始化句法，并将函数名作为地址来初始化这样的数组。

第 8 章　函数探幽

本章内容包括：

- 内联函数。
- 引用变量。
- 如何按引用传递函数参数。
- 默认参数。
- 函数重载。
- 函数模板。
- 函数模板具体化。

通过第 7 章，读者了解到很多有关 C++函数的知识，但需要学习的知识还很多。C++还提供许多新的函数特性，使之有别于 C 语言。新特性包括内联函数、按引用传递变量、默认的参数值、函数重载（多态）以及模板函数。本章介绍的 C++在 C 语言基础上新增的特性，比前面各章都多，这是读者进入加加（++）领域的重要一步。

8.1　C++内联函数

内联函数是 C++为提高程序运行速度所做的一项改进。常规函数和内联函数之间的主要区别不在于编写方式，而在于 C++编译器如何将它们组合到程序中。要了解内联函数与常规函数之间的区别，必须深入到程序的内部。

编译的最终产品是可执行程序——由一组机器语言指令组成。运行程序时，操作系统将这些指令载入到计算机的内存中，因此每条指令都有特定的内存地址。计算机随后将逐步执行这些指令。有时（如有循环或分支语句时），将跳过一些指令，向前或向后跳到特定的地址。常规函数调用也使程序跳到另外一个地址（函数地址），并在函数结束时返回。下面更详细地介绍这一过程的典型实现。执行到函数调用指令时，程序将在函数调用后立即存储该指令的内存地址，并将函数参数复制到堆栈（为此保留的内存块），跳到标记函数起点的内存单元，执行函数代码（也许还需将返回值放入到寄存器中），然后跳回到地址被保存的指令处（这与阅读文章时停下来看看脚注，并在阅读完脚注后返回到以前阅读的地方类似）。来回跳跃并记录跳跃位置意味着以前使用函数时，需要一定的开销。

C++内联函数提供了另一种选择。内联函数的编译代码与其他程序代码“内联”起来了。也就是说，编译器将使用相应的函数代码替换函数调用。对于内联代码，程序无需跳到另一个位置处执行代码，然后再跳回来。因此，内联函数的运行速度比常规函数稍快，但代价是需要占用更多的内存。如果程序在 10 个不同的地方调用同一个内联函数，则该程序将包含该函数的 10 个代码拷贝（参见图 8.1）。

应有选择地使用内联函数。如果执行函数代码的时间比处理函数调用机理的时间长，则所节省的时间将只占整个过程的很小一部分。如果代码执行时间很短，则内联调用就可以节省非内联调用使用的大部分时间。另一方面，由于这个过程相当快，因此尽管节省了该过程的大部分时间，但节省的时间绝对值并不大，除非该函数经常被调用。

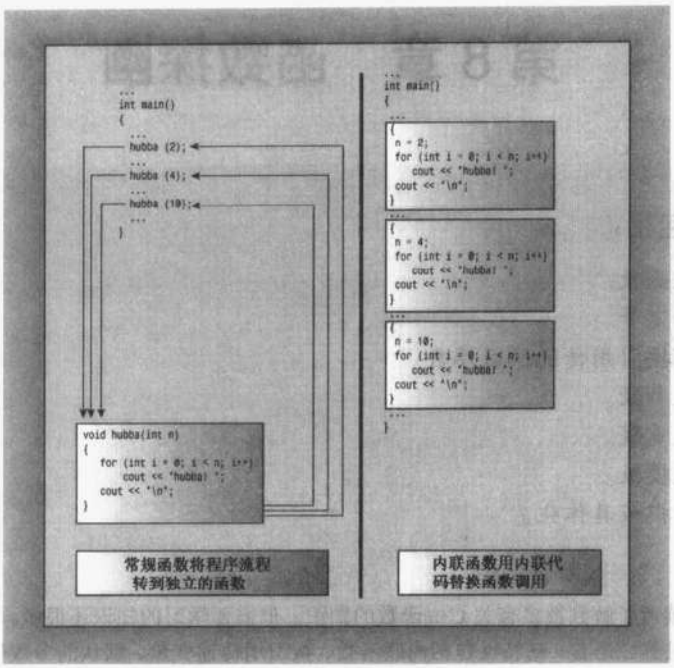


图 8.1 内联函数与常规函数

要使用这项特性，必须采取下述措施之一：

- 在函数声明前加上关键字 inline。
- 在函数定义前加上关键字 inline。

通常的做法是省略原型，将整个定义（即函数头和所有函数代码）放在本应提供原型的地方。

程序员请求将函数作为内联函数时，编译器并不一定会满足这种要求。它可能认为该函数过大或注意到函数调用了自己（内联函数不能递归），因此不将其作为内联函数；而有些编译器没有启用或实现这种特性。

程序清单 8.1 通过内联函数 square()（计算参数的平方）演示了内联技术。我们将整个函数定义放在一行中，但并不一定非得这样做。不过，如果函数定义占用多行，则将其作为内联函数就不太合适。

程序清单 8.1 inline.cpp

```
// inline.cpp -- using an inline function
#include <iostream>

// an inline function definition
inline double square (double x) { return x * x; }

int main()
{
   using namespace std;
   double a, b;
   double c = 13.0;

   a = square (5.0);
   b = square (4.5 + 7.5);   // can pass expressions
   cout << "a = " << a << ", b = " << b << "\n";
   cout << "c = " << c;
   cout << ", c squared = " << square (c++) << "\n";
   cout << "Now c = " << c << "\n";
```

```
    return 0;
}
```

下面是该程序的输出：

```
a = 25, b = 144
c = 13, c squared = 169
Now c = 14
```

输出表明，内联函数和常规函数一样，也是按值来传递参数的。如果参数为表达式，如 4.5 + 7.5，则函数将传递表达式的值（这里为 12）。这使得 C++的内联功能远远超过 C 语言的宏定义，请参见注释“内联与宏”。

尽管程序没有提供独立的原型，但 C++原型特性仍在起作用。这是因为在函数首次被使用之前出现的整个函数定义充当了原型。这意味着可以给 square() 传递 int 或 long 值，将值传递给函数之前，程序自动将这个值强制转换为 double 类型。

内联与宏

inline 工具是 C++新增的特性。C 语言使用预处理器语句#define 来提供宏——内联代码的原始实现。例如，下面是一个计算平方的宏：

```
#define SQUARE(X) X*X
```

这并不是通过传递参数实现的，而是通过文本替换来实现的——X 是“参数”的符号标记。

```
a = SQUARE(5.0); is replaced by a = 5.0*5.0;
b = SQUARE(4.5 + 7.5); is replaced by b = 4.5 + 7.5 * 4.5 + 7.5;
d = SQUARE(c++); is replaced by d = c++*c++;
```

上述范例只有第一个能正常工作。可以通过使用括号来进行改进：

```
#define SQUARE(X) ((X)*(X))
```

但仍然存在这样的问题，即宏不能按值传递。即使使用新的定义，SQUARE(C++) 仍将 c 递增两次，但是程序清单 8.1 中的内联函数 square() 计算 c 的结果，传递它，以计算其平方值，然后将 c 递增一次。

这里的目的不是演示如何编写 C 宏，而是要指出，如果使用 C 语言的宏执行了类似函数的功能，应考虑将它们转换为 C++内联函数。

8.2　引用变量

C++新增了一种复合类型——引用变量。引用是已定义的变量的别名（另一个名称）。例如，如果将 twain 作为 clement 变量的引用，则可以交替使用 twain 和 clement 来表示该变量。那么，这种别名有何作用呢？是否能帮助那些不知道如何选择变量名的人呢？有可能，但引用变量的主要用途是用作函数的形参。通过将引用变量用作参数，函数将使用原始数据，而不是其拷贝。这样除指针之外，引用也为函数处理大型结构提供了一种非常方便的途径，同时对于设计类来说，引用也是必不可少的。不过，介绍如何将引用用于函数之前，先介绍一下定义和使用引用的基本知识。请记住，下述讨论旨在说明引用是如何工作的，而不是其典型用法。

8.2.1　创建引用变量

前面讲过，C 和 C++使用&符号来指示变量的地址。C++给&符号赋予了另一个含义，将其用来声明引用。例如，要将 rodents 作为 rats 变量的别名，可以这样做：

```
int rats;
int & rodents = rats;   // makes rodents an alias for rats
```

其中，&不是地址操作符，而是类型标识符的一部分。就像声明中的 char*指的是指向 char 的指针一样，int &指的是指向 int 的引用。上述引用声明允许将 rats 和 rodents 互换——它们指向相同的值和内存单元，程序清单 8.2 表明了这一点。

程序清单 8.2 firstref.cpp

```
// firstref.cpp -- defining and using a reference
#include <iostream>
int main()
{
    using namespace std;
    int rats = 101;
    int & rodents = rats;   // rodents is a reference

    cout << "rats = " << rats;
    cout << ", rodents = " << rodents << endl;
    rodents++;
    cout << "rats = " << rats;
    cout << ", rodents = " << rodents << endl;

// some implementations require type casting the following
// addresses to type unsigned
    cout << "rats address = " << &rats;
    cout << ", rodents address = " << &rodents << endl;
    return 0;
}
```

请注意，下述语句中的&操作符：

```
int & rodents = rats;
```

不是地址操作符，而是将 rodents 的类型声明为 int &，即 int 变量的引用。但下述语句中的&操作符：

```
cout<<", rodents address ="<< &rodents << endl;
```

是地址操作符，其中&rodents 表示 rodents 引用的变量的地址。下面是程序清单 8.2 中程序的输出：

```
rats = 101, rodents = 101
rats = 102, rodents = 102
rats address = 0x0065fd48, rodents address = 0x0065fd48
```

从中可知，rats 和 rodents 的值和地址都相同（具体的地址和显示格式随系统而异）。将 rodents 加 1 将影响这两个变量。更准确地说，rodents++操作将一个有两个名称的变量加 1（同样，虽然该范例演示了引用是如何工作的，但并没有说明引用的典型用途，即作为函数参数，具体地说是结构和对象参数，稍后将介绍这些用法）。

对于 C 语言用户而言，首次接触到引用时可能也会有些困惑，因为这些用户很自然地会想到指针，但它们之间还是有区别的。例如，可以创建指向 rats 的引用和指针：

```
int rats = 101;
int & rodents = rats;   // rodents a reference
int * prats = &rats;    // prats a pointer
```

这样，表达式 rodents 和*prats 都可以同 rats 互换，而表达式&rodents 和 prats 都可以同&rats 互换。从这一点来说，引用看上去很像伪装表示的指针（其中，*解除引用操作符被隐式理解）。实际上，引用还是不同于指针的。除了表示法不同外，还有其他的差别。例如，差别之一是，必须在声明引用时将其初始化，而不能像指针那样，先声明，再赋值：

```
int rat;
int & rodent;
rodent = rat;   // No, you can't do this.
```

记住：必须在声明引用时进行初始化。

引用更接近 const 指针，必须在创建时进行初始化，一旦与某个变量关联起来，就将一直效忠于它。也就是说：

```
int & rodents = rats;
```

实际上是下述代码的伪装表示：

```
int * const pr = &rats;
```

其中，引用 rodents 扮演的角色与表达式*pr 相同。

程序清单 8.3 演示了试图将 rats 变量的引用改为 bunnies 变量的引用时，将发生的情况。

程序清单 8.3　sceref.cpp

```
// secref.cpp -- defining and using a reference
#include <iostream>
int main()
{
   using namespace std;
   int rats = 101;
   int & rodents = rats;   // rodents is a reference

   cout << "rats = " << rats;
   cout << ", rodents = " << rodents << endl;

   cout << "rats address = " << &rats;
   cout << ", rodents address = " << &rodents << endl;

   int bunnies = 50;
   rodents = bunnies;       // can we change the reference?
   cout << "bunnies = " << bunnies;
   cout << ", rats = " << rats;
   cout << ", rodents = " << rodents << endl;

   cout << "bunnies address = " << &bunnies;
   cout << ", rodents address = " << &rodents << endl;
   return 0;
}
```

下面是程序清单 8.3 中程序的输出：

```
rats = 101, rodents = 101
rats address = 0x0065fd44, rodents address = 0x0065fd44
bunnies = 50, rats = 50, rodents = 50
bunnies address = 0x0065fd48, rodents address = 0x0065fd4
```

最初，rodents 引用的是 rats，但随后程序试图将 rodents 作为 bunnies 的引用：

```
rodents = bunnies;
```

咋一看，这种意图暂时是成功的，因为 rodents 的值从 101 变为了 50。但仔细研究将发现，rats 也变成了 50，同时 rats 和 rodents 的地址相同，而该地址与 bunnies 的地址不同。由于 rodents 是 rats 的别名，因此上述赋值语句与下面的语句等效：

```
rats = bunnies;
```

也就是说，这意味着“将 bunnies 变量的值赋给 rat 变量”。简而言之，可以通过初始化声明来设置引用，但不能通过赋值来设置。

假设程序员试图这样做：

```
int rats = 101;
int * pt = &rats;
int & rodents = *pt;
int bunnies = 50;
pt = &bunnies;
```

将 rodents 初始化为*pt 使得 rodents 指向 rats。接下来将 pt 改为指向 bunnies，并不能改变这样的事实，即 rodents 引用的是 rats。

8.2.2　将引用用作函数参数

引用经常被用作函数参数，使得函数中的变量名成为调用程序中的变量的别名。这种传递参数的方法称为按引用传递。按引用传递允许被调用的函数能够访问调用函数中的变量。C++新增的这项特性是对 C 语言的超越，C 语言只能按值传递。按值传递导致被调用函数使用调用程序中的值的拷贝（参见图 8.2）。当然，C 语言也允许避开按值传递的限制，采用按指针传递的方式。

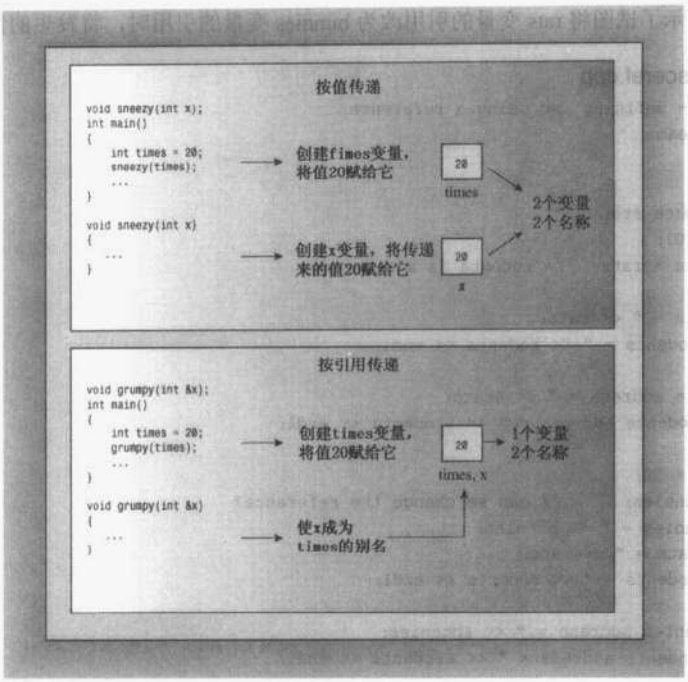


图 8.2 按值传递和按引用传递

现在我们通过一个常见的的计算机问题——交换两个变量的值，对使用引用和使用指针做一下比较。交换函数必须能够修改调用程序中的变量的值。这意味着按值传递变量将不管用，因为函数将交换原始变量拷贝的内容，而不是变量本身的内容。但传递引用时，函数将可以使用原始数据。另一种方法是，传递指针来访问原始数据。程序清单 8.4 演示了这 3 种方法，其中包括一种不管用的方法，以便读者能对这些方法进行比较。

程序清单 8.4 swaps.cpp

```
// swaps.cpp -- swapping with references and with pointers
#include <iostream>
void swapr (int & a, int & b);   // a, b are aliases for ints
void swapp (int * p, int * q);   // p, q are addresses of ints
void swapv (int a, int b);       // a, b are new variables
int main()
{
    using namespace std;
    int wallet1 = 300;
    int wallet2 = 350;

    cout << "wallet1 = $" << wallet1;
    cout << " wallet2 = $" << wallet2 << endl;

    cout << "Using references to swap contents: \n";
    swapr (wallet1, wallet2);   // pass variables
    cout << "wallet1 = $" << wallet1;
    cout << " wallet2 = $" << wallet2 << endl;

    cout << "Using pointers to swap contents: \n";
    swapp (&wallet1, &wallet2); // pass addresses of variables
    cout << "wallet1 = $" << wallet1;
    cout <<" wallet2 = $" << wallet2 << endl;
```

```
    cout << "Trying to use passing by value: \n";
    swapv (wallet1, wallet2);   // pass values of variables
    cout << "wallet1 = $" << wallet1;
    cout << " wallet2 = $" << wallet2 << endl;
    return 0;
}

void swapr (int & a, int & b)   // use references
    int temp;

    temp = a;       // use a, b for values of variables
    a = b;
    b = temp;
}

void swapp (int * p, int * q)   // use pointers
{
    int temp;

    temp = *p;       // use *p, *q for values of variables
    *p = *q;
    *q = temp;
}

void swapv (int a, int b)       // try using values
{
    int temp;
    temp = a;      // use a, b for values of variables
    a = b;
    b = temp;
}
```

下面是程序清单 8.4 中程序的输出：

```
wallet1 = $300 wallet2 = $350          //original values
Using references to swap contents:
wallet1 = $350 wallet2 = $300          // values swapped
Using pointers to swap contents again:
wallet1 = $300 wallet2 = $350          // values swapped again
Trying to use passing by value:
wallet1 = $300 wallet2 = $350          // swap failed
```

正如读者预想的，引用和指针方法都成功地交换了两个钱夹（wallet）中的内容，而按值传递的方法没能完成这项任务。

程序说明

首先来看程序清单 8.4 中每个函数是如何被调用的：

```
swapr (wallet1, wallet2);       // pass variables
swapp (&wallet1, &wallet2);     // pass addresses of variables
swapv (wallet1, wallet2);       // pass values of variables
```

按引用传递（swapr（wallet1，wallet2））和按值传递（swapv（wallet1，waller2））看起来相同。只能通过原型或函数定义才能知道 swapr() 是按引用传递的。不过，地址操作符（&）使得按地址传递（swapp（&wallet1，&wallet2））一目了然（类型声明 int * p 表明，p 是一个 int 指针，因此与 p 对应的参数应为地址，如&wallet1）。

接下来，比较函数 swapr()（按引用传递）和 swapv()（按值传递）的代码，惟一的外在区别是声明函数参数的方式不同：

```
void swapr (int & a, int & b)
void swapv (int a, int b)
```

当然还有内在的区别：在 swapr()中，变量 a 和 b 是 wallet1 和 wallet2 的别名，所以交换 a 和 b 的值相当于交换 wallet1 和 wallet2 的值。但在 swapv()中，变量 a 和 b 是复制了 wallet1 和 waller2 的值的新变量，因此交换 a 和 b 的值并不会影响 wallet1 和 wallet2 的值。

最后，比较函数 swapr()（传递引用）和 swapp()（传递指针）。第一个区别是声明函数参数的方式不同：

```
void swapr (int & a, int & b)
void swapp (int * p, int * q)
```

另一个区别是指针版本需要在函数使用 p 和 q 的整个过程中使用解除引用操作符*。

前面指出过，应在定义引用变量时对其进行初始化。可以将引用函数的参数视为被初始化为函数调用传递的参数。也就是说，下面的函数调用：

```
swapr (wallet1, wallet2);
```

将形参 a 和 b 分别初始化为 wallet1 和 wallet2。

8.2.3 引用的属性和特别之处

使用引用参数时，需要了解其一些特点。首先，请看程序清单 8.5。它使用两个函数来计算参数的立方，其中一个函数接受 double 类型的参数，另一个接受 double 引用。为了说明这一点，我们有意将计算立方的代码编写得比较奇怪。

程序清单 8.5 cubes.cpp

```
// cubes.cpp -- regular and reference arguments
#include <iostream>
double cube (double a);
double refcube (double &ra);
int main()
{
    using namespace std;
    double x = 3.0;

    cout << cube (x);
    cout << " = cube of " << x << endl;
    cout << refcube (x);
    cout << " = cube of " << x << endl;
    return 0;
}

double cube (double a)
{
    a *= a * a;
    return a;
}

double refcube (double &ra)
{
    ra *= ra * ra;
    return ra;
}
```

下面是该程序的输出：

```
27 = cube of 3
27 = cube of 27
```

refcube()函数修改了 main()中的 x 值，而 cube()没有，这提醒我们为何通常按值传递。变量 a 位于 cube()中，它被初始化为 x 的值，但修改 a 并不会影响 x。但由于 refcube()使用了引用参数，因此修改 ra 实际上就是修改 x。如果程序员的意图是让函数使用传递给它的信息，而不对这些信息进行修改，同时又想使用引用，则应使用常量引用。例如，在这个例子中，应在函数原型和函数头中使用 const：

```
double refcube (const double &ra);
```

如果这样做，当编译器发现代码修改了 ra 的值时，将生成错误消息。

另外，如果要编写类似于上述范例的函数，应采用按值传递的方式，而不要采用按引用传递的方式。当数据比较大（如结构和类）时，引用参数将很有用，读者稍后便会明白这一点。

按值传递的函数，如程序清单 8.5 中的函数 cube()，可使用多种类型的实参。例如，下面的调用都是合法的：

```
double z = cube (x + 2.0);          // evaluate x + 2.0, pass value
z = cube (8.0);                     // pass the value 8.0
int k = 10;
z = cube (k);                       // convert value of k to double, pass value
double yo[3] = { 2.2, 3.3, 4.4};
z = cube (yo[2]);                   // pass the value 4.4
```

如果将与上面类似的参数传递给接受引用参数的函数，将会发现，传递引用的限制更严格。毕竟，如果 ra 是一个变量的别名，则实参应是该变量。下面的代码：

```
double z = refcube (x + 3.0);  // may not compile
```

不合理，因为表达式 x + 3.0 并不是变量。例如，不能将值赋给该表达式：

```
x + 3.0 = 5.0;  // nonsensical
```

如果试图使用像 refcube（x + 3.0）这样的函数调用，将发生什么情况呢？在现代的 C++中，这是错误的，有些编译器将指出这一点；而有些编译器将发出这样的警告：

```
Warning: Temporary used for parameter 'ra' in call to refcube (double &)
```

之所以做出这种比较温和的反应是由于早期的 C++确实允许将表达式传递给引用变量。有些情况下，仍然是这样做的。这样做的结果如下：由于 x + 3.0 不是 double 类型的变量，因此程序将创建一个临时的无名变量，并将其初始化为表达式 x + 3.0 的值。然后，ra 将成为该临时变量的引用。下面详细讨论这种临时变量，看看什么时候创建它们，什么时候不创建。

临时变量、引用参数和 const

如果实参与引用参数不匹配，C++将生成临时变量。现在，仅当参数为 const 引用时，C++才允许这样做，但这是一种新的限制。下面来看看何种情况下，C++将生成临时变量，以及为何对 const 引用的限制是合理的。

首先，什么时候将创建临时变量呢？如果引用参数是 const，则编译器将在下面两种情况下生成临时变量：

- 实参的类型正确，但不是左值。
- 实参的类型不正确，但可以转换为正确的类型。

左值参数是可被引用的数据对象，例如，变量、数组元素、结构成员、引用和被解除引用的指针都是左值。非左值包括字面常量和包含多项的表达式。例如，假设我们重新定义 refcube()，使之接受一个常量引用参数：

```
double refcube (const double &ra)
{
    return ra * ra * ra;
}
```

现在考虑下面的代码：

```
double side = 3.0;
double * pd = &side;
double & rd = side;
long edge = 5L;
double lens[4] = { 2.0, 5.0, 10.0, 12.0};
double c1 = refcube (side);          // ra is side
double c2 = refcube (lens[2]);       // ra is lens[2]
double c3 = refcube (rd);            // ra is rd is side
double c4 = refcube (*pd);           // ra is *pd is side
double c5 = refcube (edge);          // ra is temporary variable
```

```
double c6 = refcube (7.0);          // ra is temporary variable
double c7 = refcube (side + 10.0);  // ra is temporary variable
```

参数 side、lens[2]、rd 和*pd 都是有名称的、double 类型的数据对象，因此可以为其创建引用，而不需要临时变量（还记得吗，数组元素的行为与同类型的变量类似）。但是，edge 虽然是变量，类型却不正确，double 引用不能指向 long。另一方面，参数 7.0 和 side + 10.0 的类型都正确，但没有名称，在这些情况下，编译器都将生成一个临时匿名变量，并让 ra 指向它。这些临时变量只在函数调用期间存在，此后编译器便可以随意将其删除。

那么为什么对于常量引用，这种行为是可行的，其他情况下却不行的呢？对于程序清单 8.4 中的函数 swapr()：

```
void swapr (int & a, int & b) // use references
{
    int temp;

    temp = a;     // use a, b for values of variables
    a = b;
    b = temp;
}
```

如果在早期 C++较宽松的规则下，执行下面的操作将发生什么情况呢？

```
long a = 3, b = 5;
swapr (a, b);
```

这里的类型不匹配，因此编译器将创建两个临时的 int 变量，将它们初始化为 3 和 5，然后交换临时变量的内容，而 a 和 b 保持不变。

简而言之，如果接受引用参数的函数的意图是修改作为参数传递的变量，则创建临时变量将阻止这种意图的实现。解决方法是，禁止创建临时变量，现在的 C++标准正是这样做的（不过，在默认情况下，有些编译器仍将发出警告，而不是错误消息，因此如果看到了有关临时变量的警告，请不要忽略）。

现在来看 refcube()函数。该函数的目的只是使用传递的值，而不是修改它们，因此临时变量不会造成任何不利的影响，反而会使函数在可处理的参数种类方面更通用。因此，如果声明将引用指定为 const，C++将在必要时生成临时变量。实际上，对于形参为 const 引用的 C++函数，如果实参不匹配，则其行为类似于按值传递，为确保原始数据不被修改，将使用临时变量来存储值。

记住：如果函数调用的参数不是左值或与相应的 const 引用参数的类型不匹配，则 C++将创建类型正确的匿名变量，将函数调用的参数的值传递给该匿名变量，并让参数来引用该变量。

应尽可能使用 const

将引用参数声明为常量数据的引用的理由有 3 个：

- 使用 const 可以避免无意中修改数据的编程错误。
- 使用 const 使函数能够处理 const 和非 const 实参，否则将只能接受非 const 数据。
- 使用 const 引用使函数能够正确生成并使用临时变量。

因此，应尽可能将引用形参声明为 const。

8.2.4 将引用用于结构

引用非常适合用于结构和类（C++的用户定义类型）。确实，引入引用主要是为了用于这些类型的，而不是基本的内置类型。

使用结构引用的方式与使用基本变量引用相同，只需在声明结构参数时使用引用操作符&即可，程序清单 8.6 中的程序正是这样做的。它还通过让函数返回指向结构的引用添加了一个有趣的特点，这与返回结构有所不同。对此，有一些需要注意的地方，另外通常将返回的引用声明为 const 是最好的选择。有关这几点将在接下来的几个范例中讨论。该程序包含一个 use()函数，该函数显示结构的两个成员，并将第三个成员递增。因此，第三个成员可以记录 use()函数对特定结构处理的次数。

程序清单 8.6　strtref.cpp

```
// strtref.cpp -- using structure references
#include <iostream>
using namespace std;
struct sysop
{
    char name[26];
    char quote[64];
    int used;
};

const sysop & use(sysop & sysopref);  // function with a reference return type

int main()
{
// NOTE: some implementations require using the keyword static
// in the two structure declarations to enable initialization
    sysop looper =
    {
        "Rick \"Fortran\" Looper",
        "I'm a goto kind of guy.",
        0
    };

    use(looper);          // looper is type sysop
    cout << "Looper: " << looper.used << " use(s)\n";
    sysop copycat;
    copycat = use(looper);
    cout << "Looper: " << looper.used << " use(s)\n";
    cout << "Copycat: " << copycat.used << " use(s)\n";
    cout << "use(looper): " << use(looper).used << " use(s)\n";
    return 0;
}

// use() returns the reference passed to it
const sysop & use(sysop & sysopref)
{
    cout << sysopref.name << " says:\n";
    cout << sysopref.quote << endl;
    sysopref.used++;
    return sysopref;
}
```

下面是该程序的输出：

```
Rick "Fortran" Looper says:
I'm a goto kind of guy.
Looper: 1 use(s)
Rick "Fortran" Looper says:
I'm a goto kind of guy.
Looper: 2 use(s)
Copycat: 2 use(s)
Rick "Fortran" Looper says:
I'm a goto kind of guy.
use(looper): 3 use(s)
```

1. 程序说明

该程序大胆地尝试了 3 个新的领域。第一个是使用指向结构的引用——由第一个函数调用演示：

```
use (looper) ;
```

该函数调用将结构 looper 按引用传递给 use() 函数，使得 sysopref 成为 looper 的别名。当 use() 函数显

示 sysopref 的 name 和 quote 成员时，实际上显示的是 looper 的成员。另外，当该函数将 sysopref.used 加 1 时，实际上是将 looper.used 加 1，如程序输出所示：

```
Rick "Fortran" Looper says:
I'm a goto kind of guy.
1 use(s)
```

第二个新领域是将引用作为返回值。通常，返回机制将返回值复制到临时存储区域中，随后调用程序将访问该区域。然而，返回引用意味着调用程序将直接访问返回值，而不需要拷贝。通常，引用将指向传递给函数的引用，因此调用函数实际上是直接访问自己的一个变量。例如，在这个例子中，sysopref 是 looper 的引用，因此返回值是 main() 中的原始 looper 变量。请看下面一行代码：

```
copycat = use(looper);
```

如果函数 use() 返回一个结构，sysopref 的内容将被复制到一个临时返回存储单元中，然后该临时返回存储单元的内容将被复制到 copycast 中。然而，由于 use() 返回一个指向 looper 的引用，在这种情况下，looper 的内容将被直接复制到 copycast 中。这展示了返回指向结构的引用而不是结构的优点之一：效率更高。

记住：返回引用的函数实际上是被引用的变量的别名。

该程序探索的第三个新领域是，使用函数调用来访问结构的成员：

```
cout << "use(looper): " << use(looper).used << " use(s)\n";
```

由于函数 use() 返回一个指向 looper 的引用，因此上述代码行与下面两行代码等效：

```
use(looper);
cout << "use(looper): " << looper.used << " use(s)\n";
```

表示法 use(looper).used 访问 looper 的成员 used。如果该函数返回一个结构而不是指向结构的引用，则这些代码访问的将是 looper 的临时拷贝的 used 成员。

2. 返回引用时需要注意的问题

返回引用时最重要的一点是，应避免返回当函数终止时不再存在的内存单元引用。读者应避免编写下面这样的代码：

```
const sysop & clone2(sysop & sysopref)
{
    sysop newguy;           // first step to big error
    newguy = sysopref;      // copy info
    return newguy;          // return reference to copy
}
```

该函数返回一个指向临时变量（newguy）的引用，函数运行完毕后，它将不再存在。第 9 章将讨论各种变量的持续存活性。同样，也应避免返回指向临时变量的指针。

为避免这种问题，最简单的方法是，返回一个作为参数传递给函数的引用。作为参数的引用将指向调用函数使用的数据，因此返回的引用也将指向这些数据。

另一种方法是用 new 来分配新的存储空间。前面见过这样的函数，它使用 new 为字符串分配内存空间，并返回指向该内存空间的指针。下面是使用引用来完成类似工作的方法：

```
const sysop & clone(sysop & sysopref)
{
    sysop * psysop = new sysop;
    *psysop = sysopref;   // copy info
    return *psysop;       // return reference to copy
}
```

第一条语句创建一个无名的 sysop 结构，并让指针 psysop 指向该结构，因此*psysop 就是该结构。上述代码似乎会返回该结构，但函数声明表明，该函数实际上将返回这个结构的引用。这样，便可以这样使用该函数：

```
sysop & jolly = clone(looper);
```

这使得 jolly 成为新结构的引用。这种方法存在一个问题：在不再需要 new 分配的内存时，应使用 delete 来释放它们。调用 clone() 隐藏了对 new 的调用，这使得以后很容易忘记使用 delete 来释放内存。第 16 章讨论的 auto_ptr 模板可帮助程序员自动完成释放工作。

3. 为何将 const 用于引用返回类型

在前面的范例中，函数 use() 的返回类型为 const sysop &。读者可能会问，在这里 const 有何用途？它并不意味着结构 sysop 本身为 const，而只意味着您不能使用返回的引用来直接修改它指向的结构。例如，如果省略了 const，则可以使用像下面这样的代码：

```
use(looper).used = 10;
```

由于 use() 返回一个指向 looper 的引用，上述代码将与下面的代码等效：

```
use(looper);        // display structure, increment used member
looper.used = 10;   // reset used member to 10
```

您还可以使用下面这样的代码：

```
sysop newgal = {"Polly Morf", "Polly's not a hacker. ", 0};
use(looper) = newgal;
```

上述代码与下面的代码等效：

```
sysop newgal = {"Polly Morf", "Polly's not a hacker. ", 0};
use(looper);       // display structure, increment used member
looper = newgal;   // replace looper contents with newgal contents
```

总之，省略 const 后，便可以编写更简短但含义更模糊的代码。

通常，应避免在设计中添加模糊的特性，因为模糊特性增加了犯错的机会。将返回类型声明为 const 引用，可避免您犯糊涂。有时候，省略 const 确实是有道理的，第 11 章将讨论的重载操作符<<就是一个这样的例子。

8.2.5　将引用用于类对象

将类对象传递给函数时，C++通常的做法是使用引用。例如，可以通过使用引用，让函数将类 string、ostream、istream、ofstream 和 ifstream 等类的对象作为参数。

下面来看一个例子，它使用了 string 类，并演示了一些不同的设计方案，其中的一些是糟糕的。这个例子的基本思想是，创建一个函数，它将指定的字符串加入到另一个字符串的前面和后面。程序清单 8.7 提供了 3 个这样的函数，然而其中的一个存在非常大的缺陷，可能导致程序崩溃甚至不能通过编译。

程序清单 8.7　strquote.cpp

```
// strquote.cpp  -- different designs
#include <iostream>
#include <string>
using namespace std;
string version1(const string & s1, const string & s2);
const string & version2(string & s1, const string & s2);  // has side effect
const string & version3(string & s1, const string & s2);  // bad design

int main()
{
   string input;
   string copy;
   string result;

   cout << "Enter a string: ";
   getline(cin, input);
   copy = input;
   cout << "Your string as entered: " << input << endl;
   result = version1(input, "***");
   cout << "Your string enhanced: " << result << endl;
   cout << "Your original string: " << input << endl;

   result = version2(input, "###");
   cout << "Your string enhanced: " << result << endl;
   cout << "Your original string: " << input << endl;
```

```
    cout << "Resetting original string.\n";
    input = copy;
    result = version3(input, "@@@");
    cout << "Your string enhanced: " << result << endl;
    cout << "Your original string: " << input << endl;
    return 0;
}

string version1(const string & s1, const string & s2)
{
    string temp;

    temp = s2 + s1 + s2;;
    return temp;
}

const string & version2(string & s1, const string & s2)   // has side effect
{
    s1 = s2 + s1 + s2;
// safe to return reference passed to function
    return s1;
}

const string & version3(string & s1, const string & s2)   // bad design
{
    string temp;

    temp = s2 + s1 + s2;
// unsafe to return reference to local variable
    return temp;
}
```

下面是该程序的运行情况：

```
Enter a string: It's not my fault.
Your string as entered:
Your string enhanced:
Your original string: It's
Your string enhanced:
Your original string:
Resetting original string.
```

此时，该程序已经崩溃。

程序说明

在程序清单 8.7 的 3 个函数中，version1 最简单：

```
string version1(const string & s1, const string & s2)
{
    string temp;

    temp = s2 + s1 + s2;;
    return temp;
}
```

它接受两个 string 参数，并使用 string 类的相加功能来创建一个满足要求的新字符串。这两个函数参数都是 const 引用。如果使用 string 对象作为参数，最终结果将不变：

```
string version4(string s1, string s2)  // would work the same
```

在这种情况下，s1 和 s2 将为 string 对象。使用引用的效率更高，因为函数不需要创建新的 string 对象，并将原来对象中的数据复制到新对象中。限定符 const 指出，该函数将使用原来的 string 对象，但不会修改它。

temp 是一个新的 string 对象，只在函数 version1()中有效，该函数执行完毕后，它将不再存在。因此，将返回指向 temp 的引用不可行，因此该函数的返回类型为 string，这意味着 temp 的内容将被复制到一个临时返回存储单元中，然后在 main()中，该返回存储单元的内容将被复制到一个名为 result 的 string 中：

```
result = version1(input, "***");
```

将 C-风格字符串用作 string 对象引用参数

对于函数 version1()，读者可能注意到了很有趣的一点：该函数的两个形参（s1 和 s2）的类型都是 const string &，但实参（input 和"***"）的类型分别是 string 和 const char *。由于 input 的类型为 string，因此让 s1 指向它没有任何问题。然而，程序怎么能够接受将 char 指针赋给 string 引用呢？

这里有两点需要说明。首先，string 类定义了一种 char *到 string 的转换功能，这使得可以使用 C-风格字符串来初始化 string 对象。其次是本章前面讨论过的类型为 const 引用的形参的一个属性。假设实参的类型与引用参数类型不匹配，但可被转换为引用类型，程序将创建一个正确类型的临时变量，使用转换后的实参值来初始化它，然后传递一个指向该临时变量的引用。例如，在本章前面，将 int 实参传递给 const double &形参时，就是以这种方式进行处理的。同样，也可以将实参 char *或 const char *传递给形参 const string &。

这种属性的结果是，如果形参类型为 const string &，在调用函数时，使用的实参可以是 string 对象或 C-风格字符串，如用引号括起的字符串字面量、以空字符结尾的 char 数组或指向 char 的指针变量。因此，下面的代码是可行的：

```
result = version1(input, "***");
```

函数 version2()不创建临时 string 对象，而是直接修改原来的 string 对象：

```
const string & version2(string & s1, const string & s2)   // has side effect
{
    s1 = s2 + s1 + s2;
// safe to return reference passed to function
    return s1;
}
```

该函数可以修改 s1，因为不同于 s2，s1 没有被声明为 const。

由于 s1 是指向 main()中一个对象（input）的引用，因此将 s1 最为引用返回是安全的。由于 s1 是指向 input 的引用，因此，下面一行代码：

```
result = version2(input, "###");
```

与下面的代码等价：

```
version2(input, "###");     // input altered directly by version2()
result = input;             // reference to s1 is reference to input
```

然而，同样是由于 s1 是指向 input 的引用，调用该函数将带来修改 input 的副作用：

```
Your original string: It's not my fault.
Your string enhanced: ### It's not my fault.###
Your original string: ### It's not my fault.###
```

因此，如果要保留原来的字符串不变，这将是一种错误的设计。

程序清单 8.7 中的第三个函数版本指出了什么不能做：

```
const string & version3(string & s1, const string & s2)   // bad design
{
    string temp;

    temp = s2 + s1 + s2;
// unsafe to return reference to local variable
    return temp;
}
```

它存在一个致命的缺陷：返回一个指向 version3()中声明的变量的引用。这个函数能够通过编译（但编译器会发出警告），但当程序试图执行该函数时将崩溃。具体地说，问题是由下面的复制语句引发的：

```
result = version3(input, "@@@");
```

程序试图引用已经释放的内存。

8.2.6 对象、继承和引用

ostream 和 ofstream 类凸现了引用的一个有趣属性。正如第 6 章介绍的，ofstream 对象可以使用 ostream 类的方法，这使得文件输入/输出的格式与控制台输入/输出相同。使得能够将特性从一个类传递给另一个类的语言特性被称为继承，这将在第 13 章详细讨论。简单地说，ostream 是基类（因为 ofstream 是建立在它的基础之上的），而 ofstream 是派生类（因为它是从 ostream 派生而来的）。派生类继承了基类的方法，这意味着 ofstream 对象可以使用基类的特性，如格式化方法 precision() 和 setf()。

继承的另一个特征是，基类引用可以指向派生类对象，而无需进行强制类型转换。这种特征的一个实际结果是，可以定义一个接受基类引用作为参数的函数，调用该函数时，可以将基类对象作为参数，也可以将派生类对象作为参数。例如，参数类型为 ostream &的函数可以接受 ostream 对象（如 cout）或您声明的 ofstream 对象作为参数。

程序清单 8.8 通过调用同一个函数（只有函数调用参数不同）将数据写入文件和显示到屏幕上来说明了这一点。该程序要求用户输入望远镜物镜和一些目镜的焦距，然后计算并显示每个目镜的放大倍数。放大倍数等于物镜的焦距除以目镜的焦距，因此计算起来很简单。该程序还使用了一些格式化方法，这些方法用于 cout 和 ofstream 对象（在这个例子中为 fout）时作用相同。

程序清单 8.8 filefunc.cpp

```
//filefunc.cpp -- function with ostream & parameter
#include <iostream>
#include <fstream>
#include <cstdlib>
using namespace std;

void file_it(ostream & os, double fo, const double fe[],int n);
const int LIMIT = 5;
int main(void)
{
   ofstream fout;
   const char * fn = "ep-data.txt";
   fout.open(fn);
   if (!fout.is_open())
   {
      cout << "Can't open " << fn << ". Bye.\n";
      exit(EXIT_FAILURE);
   }
   double objective;
   cout << "Enter the focal length of your "
      "telescope objective in mm: ";
   cin >> objective;
   double eps[LIMIT];
   cout << "Enter the focal lengths, in mm, of " << LIMIT << " eyepieces:\n";
   for (int i = 0; i < LIMIT; i++)
   {
      cout << "Eyepiece #" << i + 1 << ": ";
      cin >> eps[i];
   }
   file_it(fout, objective, eps, LIMIT);
   file_it(cout, objective, eps, LIMIT);
   cout << "Done\n";
   return 0;
}
void file_it(ostream & os, double fo, const double fe[],int n)
{
   ios_base::fmtflags initial;
   initial = os.setf(ios_base::fixed); // save initial formatting state
```

```
    os.precision(0);
    os << "Focal length of objective: " << fo << " mm\n";
    os.setf(ios::showpoint);
    os.precision(1);
    os.width(12);
    os << "f eyepiece";
    os.width(15);
    os << "magnification" << endl;
    for (int i = 0; i < n; i++)
    {
        os.width(12);
        os << fe[i];
        os.width(15);
        os << int (fo/fe[i] + 0.5) << endl;
    }
    os.setf(initial);  // restore initial formatting state
}
```

下面是该程序的运行情况：

```
Enter the focal length of your telescope objective in mm: 1800
Enter the focal lengths, in mm, of 5 eyepieces:
Eyepiece #1:    30
Eyepiece #2:    19
Eyepiece #3:    14
Eyepiece #4:    8.8
Eyepiece #5:    7.5
Focal length of objective: 1800 mm
 f eyepiece  magnification
 30.0            60
 19.0            95
 14.0            129
 8.8             205
 7.5             240
Done
```

下述代码行将目镜数据写入到文件 ep-data.txt 中：

```
file_it(fout, objective, eps, LIMIT);
```

而下述代码行将同样的信息以同样的格式显示到屏幕上：

```
file_it(cout, objective, eps, LIMIT);
```

程序说明

对于该程序，最重要的一点是，参数 os（其类型为 ostream &）可以指向 ostream 对象（如 cout），也可以指向 ofstream 对象（如 fout）。该程序还演示了如何使用 ostream 类中的格式化方法。下面复习（介绍）其中的一些方法，更详细的讨论请参阅第 17 章。

方法 setf() 让您能够设置各种格式化状态。例如，方法调用 setf(ios_base::fixed)将对象置于使用定点表示法的模式；setf(ios_base::showpoint)将对象置于显示小数点的模式，即使小数部分为零。方法 precision() 指定显示多少位小数（假定对象处于定点模式下）。所有这些设置都将一直保持不变，直到再次调用相应的方法重新设置它们。方法 width() 设置下一次输出操作使用的字段宽度，这种设置只在显示下一个值时有效，然后将恢复到默认设置。默认的字段宽度为零，这意味着刚好能容纳下要显示的内容。

函数 file_it() 使用了两个有趣的方法调用：

```
ios_base::fmtflags initial;
initial = os.setf(ios_base::fixed); // save initial formatting state
...
os.setf(initial);  // restore initial formatting state
```

方法 setf() 返回调用它之前有效的所有格式化设置。ios_base::fmtflags 是存储这种信息所需的数据类型名称。因此，将返回值赋给 initial 将存储调用 file_it() 之前的格式化设置，然后便可以使用变量 initial 作为参数来调用 setf()，将所有的格式化设置恢复到原来的值。因此，该函数将对象返回到传递给 file_it() 之前

的状态。

了解更多有关类的知识将有助于更好地理解这些方法的工作原理，以及为何在代码中使用 ios_base。然而，读者不用等到第 17 章才使用这些方法。

需要说明的最后一点是，每个对象都存储了自己的格式化设置。因此，当程序将 cout 传递给 file_it() 时，cout 的设置将被修改，然后被恢复。当程序将 fout 传递给 file_it() 时，fout 的设置将被修改，然后被恢复。

8.2.7 何时使用引用参数

使用引用参数的主要原因有两个：

- 程序员能够修改调用函数中的数据对象。
- 通过传递引用而不是整个数据对象，可以提高程序的运行速度。

当数据对象较大时（如结构和类对象），第二个原因最重要。这些也是使用指针参数的原因。这是有道理的，因为引用参数实际上是基于指针的代码的另一个接口。那么，什么时候应使用引用、什么时候应使用指针呢？什么时候又应按值传递呢？下面是一些指导原则：

对于使用传递的值而不作修改的函数：

- 如果数据对象很小，如内置数据类型或小型结构，则按值传递。
- 如果数据对象是数组，则使用指针，因为这是惟一的选择，并将指针声明为指向 const 的指针。
- 如果数据对象是较大的结构，则使用 const 指针或 const 引用，以提高程序的效率。这样可以节省复制结构所需的时间和空间。
- 如果数据对象是类对象，则使用 const 引用。类设计的语义常常要求使用引用，这是 C++新增这项特性的主要原因。因此，传递类对象参数的标准方式是按引用传递。

对于修改调用函数中数据的函数：

- 如果数据对象是内置数据类型，则使用指针。如果看到诸如 fixit（&x）这样的代码（其中 x 是 int 型），则很明显，该函数将修改 x。
- 如果数据对象是数组，则只能使用指针。
- 如果数据对象是结构，则使用引用或指针。
- 如果数据对象是类对象，则使用引用。

当然，这只是一些指导原则，很可能有充分的理由做出其他的选择。例如，对于基本类型，cin 使用引用，因此可以使用 cin>>n，而不是 cin >> &n。

8.3 默认参数

下面介绍 C++的另一项新内容——默认参数。默认参数指的是当函数调用中省略了实参时自动使用的一个值。例如，如果将 void wow（int n）设置成 *n* 有默认值为 1，则函数调用 wow() 相当于 wow（1）。这极大地提高了使用函数的灵活性。假设有一个名为 left() 的函数，它将字符串和 *n* 作为参数，并返回该字符串的前 *n* 个字符。更准确地说，该函数返回一个指针，该指针指向由原始字符串中被选中的部分组成的字符串。例如，函数调用 left（“theory”，3）将创建新字符串“the”，并返回一个指向该字符串的指针。现在假设第二个参数的默认值被设置为 1，则函数调用 left（“theoty”，3）仍像前面讲述的那样工作，3 将覆盖默认值。但函数调用 left（“theory”）不会出错，它认为第二个参数的值为 1，并返回指向字符串“t”的指针。如果程序经常需要抽取一个字符组成的字符串，而偶尔需要抽取较长的字符串，则这种默认值将很有帮助。

如何设置默认值呢？必须通过函数原型。由于编译器通过查看原型来了解函数所使用的参数数目，因此函数原型也必须将可能的默认参数告知程序。方法是将值赋给原型中的参数。例如，left() 的原型如下：

```
char * left (const char * str, int n = 1);
```

读者希望该函数返回一个新的字符串，因此将其类型设置为 char *（指向 char 的指针）；读者希望原始字符串保持不变，因此对第一个参数使用了 const 限定符；读者希望 *n* 的默认值为 1，因此将这个值赋给 *n*。默认参数值是初始化值，因此上面的原型将 *n* 初始化为 1。如果省略参数 *n*，则它的值将为 1；否则，传递的值将覆盖 1。

对于带参数列表的函数，必须从右向左添加默认值。也就是说，要为某个参数设置默认值，则必须为它右边的所有参数提供默认值：

```
int harpo (int n, int m = 4, int j = 5);          // VALID
int chico (int n, int m = 6, int j);              // INVALID
int groucho (int k = 1, int m = 2, int n = 3);    // VALID
```

例如，harpo()原型允许调用该函数时提供 1 个、2 个或 3 个参数：

```
beeps = harpo (2);          // same as harpo (2, 4, 5)
beeps = harpo (1, 8);       // same as harpo (1, 8, 5)
beeps = harpo (8, 7, 6);    // no default arguments used
```

实参按从左到右的顺序依次被赋给相应的形参，而不能跳过任何参数。因此，下面的调用是不允许的：

```
beeps = harpo (3, , 8);     // invalid, doesn't set m to 4
```

默认参数不是编程方面的重大突破，而只是提供了一种便捷的方式。在设计类时读者将发现，通过使用默认参数，可以减少要定义的析构函数、方法以及方法重载的数量。

程序清单 8.9 使用了默认参数。请注意，只有原型指定了默认值。函数定义与没有默认参数时完全相同。

程序清单 8.9　left.cpp

```
// left.cpp -- string function with a default argument
#include <iostream>
const int ArSize = 80;
char * left (const char * str, int n = 1);
int main()
{
   using namespace std;
   char sample[ArSize];
   cout << "Enter a string: \n";
   cin.get (sample, ArSize);
   char *ps = left (sample, 4);
   cout << ps << endl;
   delete [] ps;       // free old string
   ps = left (sample);
   cout << ps << endl;
   delete [] ps;       // free new string
   return 0;
}

// This function returns a pointer to a new string
// consisting of the first n characters in the str string.
char * left (const char * str, int n)
{
   if (n < 0)
      n = 0;
   char * p = new char[n+1];
   int i;
   for (i = 0; i < n && str[i]; i++)
      p[i] = str[i];  // copy characters
   while (i <= n)
      p[i++] = '\0';  // set rest of string to '\0'
   return p;
}
```

下面是该程序的运行情况：

```
Enter a string:
forthcoming
fort
f
```

程序说明

该程序使用 new 创建一个新的字符串，以存储被选择的字符。一种可能出现的尴尬情况是，不合作的用户要求的字符数目可能为负。在这种情况下，函数将字符计数设置为 0，并返回一个空字符串。另一种可能出现的尴尬情况是，不负责任的用户要求的字符数目可能多于字符串包含的字符数，为预防这种情况，函数使用了一个组合测试：

```
i < n && str[i]
```

$i < n$ 测试让循环复制了 n 个字符后终止。测试的第二部分——表达式 str[i]，是要复制的字符的编码。遇到空值字符（其编码为 0）后，循环将结束。这样，while 循环将使字符串以空值字符结束，并将余下的空间（如果有的话）设置为空值字符。

另一种设置新字符串长度的方法是，将 n 设置为传递的值和字符串长度中较小的一个：

```
int len = strlen(str);
n = (n < len) ? n: len;    // the lesser of n and len
char * p = new char[n+1];
```

这将确保 new 分配的空间不会多于存储字符串所需的空间。如果用户执行像 left（“Hi!”，32767）这样的调用，则这种方法很有用。第一种方法将把“Hi!”复制到由 32767 个字符组成的数组中，并将除前 3 个字符之外的所有字符设置为空值字符；第二种方法将“Hi!”复制到由 4 个字符组成的数组中。但由于添加了另外一个函数调用（strlen()），因此程序将更长，运行速度将降低，同时还必须包含头文件 cstring（或 string.h）。C 程序员倾向于选择运行速度更快、更简洁的代码，因此需要程序员在正确使用函数方面承担更多责任。不过，C++的传统是更强调可靠性。毕竟，速度较慢但能正常运行的程序，要比运行速度虽快但无法正常运行的程序好。如果调用 strlen()所需的时间很长，则可以让 left()直接确定 n 和字符串长度哪个小。例如，当 m 的值等于 n 或到达字符串结尾时，下面的循环都将终止：

```
int m = 0;
while (m <= n && str[m] != '\0')
    m++;
char * p = new char[m+1];
// use m instead of n in rest of code
```

8.4 函数重载

函数多态是 C++新增的功能。默认参数让您能够使用不同数目的参数调用同一个函数，而函数多态（函数重载）让您能够使用多个同名的函数。术语多态（polymorphism）指的是有多种形式，因此函数多态允许函数可以有多种形式。类似地，术语函数重载（function overloading）指的是可以有多个同名的函数，因此对名称进行了重载。两种术语指的是同一回事，但我们通常使用函数重载——似乎更努力。可以通过函数重载来设计一系列函数——它们完成相同的工作，但使用不同的参数列表。

重载函数就像是有多种含义的动词。例如，Piggy 小姐可以在棒球场为家乡球队助威（root），也可以在地里种植（root）菌类作物。根据上下文可以知道在每一种情况下，root 的含义是什么。同样，C++使用上下文来确定要使用的重载函数版本。

函数重载的关键是函数的参数列表——也称为函数特征标（function signature）。如果两个函数的参数数目和类型相同，同时参数的排列顺序也相同，则它们的特征标相同，而变量名是无关紧要的。C++允许定义名称相同的函数，条件是它们的特征标不同。如果参数数目和/或参数类型不同，则特征标也不同。例如，可以定义一组原型如下的 print()函数：

```
void print(const char * str, int width);  // #1
void print(double d, int width);          // #2
void print(long l, int width);            // #3
```

```
void print (int i, int width);          // #4
void print (const char *str);           // #5
```

使用 print() 函数时，编译器将根据所采取的用法使用有相应特征标的原型：

```
print ("Pancakes", 15);          // use #1
print ("Syrup");                 // use #5
print (1999.0, 10);              // use #2
print (1999, 12);                // use #4
print (1999L, 15);               // use #3
```

例如，print（“Pancakes”，15）使用一个字符串和一个整数作为参数，这与#1 原型匹配。

使用被重载的函数时，需要在函数调用中使用正确的参数类型。例如，对于下面的语句：

```
unsigned int year = 3210;
print (year, 6);                 // ambiguous call
```

print() 调用与哪个原型匹配呢？它不与任何原型匹配！没有匹配的原型并不会自动停止使用其中的某个函数，因为 C++将尝试使用标准类型转换强制进行匹配。如果#2 原型是 print() 惟一的原型，则函数调用 print（year，6）将把 year 转换为 double 类型。但在上面的代码中，有 3 个将数字作为第一个参数的原型，因此有 3 种转换 year 的方式。在这种情况下，C++将拒绝这种函数调用，并将其视为错误。

一些看起来彼此不同的特征标是不能共存的。例如，请看下面的两个原型：

```
double cube (double x);
double cube (double & x);
```

读者可能认为可以在此处使用函数重载，因为它们的特征标看起来不同。但是，请从编译器的角度来考虑这个问题。假设有下面这样的代码：

```
cout << cube (x);
```

参数 x 与 double x 原型和 double &x 原型都匹配，因此编译器无法确定究竟应使用哪个原型。为避免这种混乱，编译器在检查函数特征标时，将把类型引用和类型本身视为同一个特征标。

匹配函数时，并不区分 const 和非 const 变量。请看下面的原型：

```
void dribble (char * bits);          // overloaded
void dribble (const char *cbits);    // overloaded
void dabble (char * bits);           // not overloaded
void drivel (const char * bits);     // not overloaded
```

下面列出了各种函数调用对应的原型：

```
const char p1[20] = "How's the weather? ";
char p2[20] = "How's business? ";
dribble (p1);      // dribble (const char *);
dribble (p2);      // dribble (char *);
dabble (p1);       // no match
dabble (p2);       // dabble (char *);
drivel (p1);       // drivel (const char *);
drivel (p2);       // drivel (const char *);
```

dribble() 函数有两个原型，一个用于 const 指针，另一个用于常规指针，编译器将根据实参是否为 const 来决定使用哪个原型。dribble() 函数只与带非 const 参数的调用匹配，而 drivel() 函数可以与带 const 或非 const 参数的调用匹配。drivel() 和 dabble() 之所以在行为上有这种差别，主要是由于将非 const 值赋给 const 变量是合法的，但反之则是非法的。

请记住，是特征标，而不是函数类型使得可以对函数进行重载。例如，下面的两个声明是互斥的：

```
long gronk (int n, float m);         // same signatures,
double gronk (int n, float m);       // hence not allowed
```

因此，C++不允许以这种方式重载 gronk()。返回类型可以不同，但特征标也必须不同：

```
long gronk (int n, float m);         // different signatures,
double gronk (float n, float m);     // hence allowed
```

在本章稍后讨论过模板后，我们将进一步讨论函数匹配的问题。

8.4.1　重载范例

本章前面创建了一个 left() 函数，它返回一个指针，该指针指向字符串的前 *n* 个字符。下面添加另一

个 left() 函数，它返回整数的前 *n* 位。例如，可以使用该函数来查看被存储为整数的、美国邮政编码的前 3 位——如果要根据城区分拣邮件，则这种操作很有用。

该函数的整数版本编写起来比字符串版本更困难些，因为并不是整数的每一位被存储在相应的数组元素中。一种方法是，先计算数字包含多少位。将数字除以 10 便可以去掉一位，因此可以使用除法来计算数位。更准确地说，可以用下面的循环完成这种工作：

```
unsigned digits = 1;
while (n /= 10)
    digits++;
```

上述循环计算每次删除 *n* 中的一位时，需要多少次才能删除所有的位。前面讲过，*n* /= 10 是 *n* = *n* / 10 的缩写。例如，如果 *n* 为 8，则该测试条件将 8/10 的值（0，由于这是整数除法）赋给 *n*。这将结束循环，digits 的值仍然为 1。但如果 *n* 为 238，第一轮循环测试将 *n* 设置为 238/10，即 23。这个值不为零，因此循环将 digits 增加到 2。下一轮循环将 *n* 设置为 23/10，即 2。这个值还是不为零，因此 digits 将增加到 3。下一轮循环将 *n* 设置为 2/10，即 0，从而结束结束，而 digits 被设置为正确的值——3。

现在假设知道数字共有 5 位，并要返回前 3 位，则将这个数除以 10 后再除以 10，便可以得到所需的值。每除以 10 次就删除数字的最后一位。要知道需要删除多少位，只需将总位数减去要获得的位数即可。例如，要获得 9 位数的前 4 位，需要删除后面的 5 位。可以这样编写代码：

```
ct = digits - ct;
while (ct--)
    num /= 10;
return num;
```

程序清单 8.10 将上述代码放到了一个新的 left() 函数中。该函数还包含一些用于处理特殊情况的代码，如用户要求显示 0 位或要求显示的位数多于总位数。由于新 left() 的特征标不同于旧的 left()，因此可以在同一个程序中使用这两个函数。

程序清单 8.10 leftover.cpp

```
// leftover.cpp -- overloading the left()function
#include <iostream>
unsigned long left (unsigned long num, unsigned ct);
char * left (const char * str, int n = 1);
int main()
{
    using namespace std;
    char * trip = "Hawaii!! ";   // test value
    unsigned long n = 12345678;  // test value
    int i;
    char * temp;

    for (i = 1; i < 10; i++)
    {
        cout << left (n, i) << endl;
        temp = left (trip, i);
        cout << temp << endl;
        delete [] temp; // point to temporary storage
    }
    return 0;

}

// This function returns the first ct digits of the number num.
unsigned long left (unsigned long num, unsigned ct)
{
    unsigned digits = 1;
    unsigned long n = num;

    if (ct == 0 || num == 0)
```

```
        return 0;          // return 0 if no digits
    while (n /= 10)
        digits++;
    if (digits > ct)
    {
    ct = digits - ct;
    while (ct--)
        num /= 10;
    return num;            // return left ct digits
    }
    else                   // if ct >= number of digits
        return num;        // return the whole number
}

// This function returns a pointer to a new string
// consisting of the first n characters in the str string.
char * left (const char * str, int n)
{
    if (n < 0)
        n = 0;
    char * p = new char[n+1];
    int i;
    for (i = 0; i < n && str[i]; i++)
        p[i] = str[i];  // copy characters
    while (i <= n)
        p[i++] = '\0';  // set rest of string to '\0'
    return p;
}
```

下面是该程序的输出：

```
1
H
12
Ha
123
Haw
1234
Hawa
12345
Hawai
123456
Hawaii
1234567
Hawaii!
12345678
Hawaii!!
12345678
Hawaii!!
```

8.4.2　何时使用函数重载

虽然函数重载很吸引人，但也不要滥用。仅当函数基本上执行相同的任务，但使用不同形式的数据时，才应采用函数重载。另外，读者可能还想知道，是否可以通过使用默认参数来实现同样的目的。例如，可以用两个重载函数来代替面向字符串的 left() 函数：

```
char * left (const char * str, unsigned n);      // two arguments
char * left (const char * str);                  // one argument
```

使用一个带默认参数的函数要简单些。只需编写一个函数（而不是两个函数），程序也只需为一个函数（而不是两个）请求内存；需要修改函数时，只需修改一个。不过，如果需要使用不同类型的参数时，则默认参数便不管用了，在这种情况下，应该使用函数重载。

什么是名称修饰

C++如何跟踪每一个重载函数呢？它给这些函数指定了秘密身份。使用 C++开发工具中的编辑器编写和编译程序时，C++编译器将执行一些神奇的操作——名称修饰（name decoration）或名称矫正（name mangling），它根据函数原型中指定的形参类型对每个函数名进行加密。请看下述未经修饰的函数原型：

```
long MyFunctionFoo(int, float);
```

这种格式对于我们人类来说很适合；我们知道函数接受两个参数（一个为 int 类型，另一个为 float 类型），并返回一个 long 值。而编译器将名称转换为不太好看的内部表示，来描述该接口，如下所示：

```
?MyFunctionFoo@@YAXH@Z
```

对原始名称进行的表面看来无意义的修饰（或矫正，因人而异）将对参数数目和类型进行编码。添加的一组符号随函数特征标而异，而修饰时使用的约定随编译器而异。

8.5 函数模板

现在的 C++编译器实现了 C++新增的一项特性——函数模板。函数模板是通用的函数描述，也就是说，它们使用通用类型来定义函数，其中的通用类型可用具体的类型（如 int 或 double）替换。通过将类型作为参数传递给模板，可使编译器生成该类型的函数。由于模板允许以通用类型（而不是具体类型）的方式编写程序，因此有时也被称为通用编程。由于类型是用参数表示的，因此模板特性有时也被称为参数化类型（parameterized types）。下面介绍为何需要这种特性以及其工作原理。

前面定义了一个交换两个 int 值的函数。假设要交换两个 double 值，则一种方法是复制原来的代码，并用 double 替换所有的 int。如果需要交换两个 char 值，可以再次使用同样的技术。进行这种修改将浪费宝贵的时间，且容易出错。如果进行手工修改，则可能会漏掉一个 int。如果进行全局查找和替换，如用 double 替换 int，则可能会将：

```
int x;
short interval;
```

转换为：

```
double x;              // intended change of type
short doubleerval;     // unintended change of variable namee
```

C++的函数模板功能能自动完成这一过程，可以节省时间，而且更可靠。

函数模板允许以任意类型的方式来定义函数。例如，可以这样建立一个交换模板：

```
template <class Any>
void Swap(Any &a, Any &b)
{
   Any temp;
   temp = a;
   a = b;
   b = temp;
}
```

第一行指出，要建立一个模板，并将类型命名为 Any。关键字 template 和 class 是必需的，除非可以使用关键字 typename 代替 class。另外，必须使用尖括号。类型名可以任意选择（这里为 Any），只要遵守 C++命名规则即可；许多程序员都使用简单的名称，如 T。余下的代码描述了交换两个 Any 值的算法。模板并不创建任何函数，而只是告诉编译器如何定义函数。需要交换 int 的函数时，编译器将按模板模式创建这样的函数，并用 int 代替 Any。同样，需要交换 double 的函数时，编译器将按模板模式创建这样的函数，并用 double 代替 Any。

关键字 typename 是最近才添加到 C++中的，在上述情况下，可用来替换关键字 class。也就是说，可以这样编写模板定义：

```
template <typename Any>
void Swap(Any &a, Any &b)
```

```
{
    Any temp;
    temp = a;
    a = b;
    b = temp;
}
```

typename 关键字使得参数 Any 表示类型这一点更为明显；不过，有大量的代码库是使用关键字 class 开发的。在这种上下文中，这两个关键字是等价的。

提示： 如果需要多个将同一种算法用于不同类型的函数，请使用模板。如果不考虑向后兼容的问题，并愿意键入较长的单词，则声明类型参数时，应使用关键字 typename 而不使用 class。

要让编译器知道程序需要一个特定形式的交换函数，只需在程序中使用 Swap() 函数即可。编译器将检查所使用的参数类型，并生成相应的函数。程序清单 8.11 演示为何可以这样做。该程序的布局和使用常规函数是相同的，只是在文件的开始位置提供了模板函数的原型，而在 main() 的前面提供了模板函数的定义。

注意： 如果 C++编译器不是最新版本，则可能不支持模板。新版本允许用关键字 typename 替换 class。早期的 g++版本要求在使用模板之前提供模板原型和模板定义，而在新发布的版本中，必须这样做。

程序清单 8.11　funtemp.cpp

```
// funtemp.cpp -- using a function template
#include <iostream>
// function template prototype
template <class Any>  // or typename Any
void Swap (Any &a, Any &b);

int main()
{
    using namespace std;
    int i = 10;
    int j = 20;
    cout << "i, j = " << i << ", " << j << ".\n";
    cout << "Using compiler-generated int swapper: \n";
    Swap (i, j);  // generates void Swap (int &, int &)
    cout << "Now i, j = " << i << ", " << j << ".\n";

    double x = 24.5;
    double y = 81.7;
    cout << "x, y = " << x << ", " << y << ".\n";
    cout << "Using compiler-generated double swapper: \n";
    Swap (x, y);  // generates void Swap (double &, double &)
    cout << "Now x, y = " << x << ", " << y << ".\n";

    return 0;
}

// function template definition
template <class Any>  // or typename Any
void Swap (Any &a, Any &b)
{
    Any temp;   // temp a variable of type Any
    temp = a;
    a = b;
    b = temp;
}
```

程序清单 8.11 中的第一个 Swap() 函数接受两个 int 参数，因此编译器生成该函数的 int 版本。也就是说，用 int 替换所有的 Any，生成下面这样的定义：

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

程序员看不到这些代码，但编译器确实生成并在程序中使用了它们。第二个 Swap() 函数接受两个 double 参数，因此编译器将生成 double 版本。也就是说，用 double 替换 Any，生成下述代码：

```
void Swap (double &a, double &b)
{
    double temp;
    temp = a;
    a = b;
    b = temp;
}
```

下面是程序清单 8.11 中程序的输出，从中可以知道，这种处理方式是可行的：

```
i, j = 10, 20.
Using compiler-generated int swapper:
Now i, j = 20, 10.
x, y = 24.5, 81.7.
Using compiler-generated double swapper:
Now x, y = 81.7, 24.5.
```

注意，函数模板不能缩短可执行程序。对于程序清单 8.11，最终仍将由两个独立的函数定义，就像以手工方式定义了这些函数一样。最终的代码不包含任何模板，而只包含了为程序生成的实际函数。使用模板的好处是，它使生成多个函数定义更简单、更可靠。

8.5.1 重载的模板

需要多个对不同类型使用同一种算法的函数时，可使用模板，如程序清单 8.11 所示。不过，并非所有的类型都使用相同的算法。为满足这种需求，可以像重载常规函数定义那样重载模板定义。和常规重载一样，被重载的模板的函数特征标必须不同。例如，程序清单 8.12 新增了一个交换模板，用于交换两个数组中的元素。原来的模板的特征标为（Any &，Any &），而新模板的特征标为（Any [ ]，Any [ ]，int）。注意，在后一个模板中，最后一个参数的类型为具体类型（int），而不是通用类型。并非所有的模板参数都必须是模板参数类型。

编译器见到 twotemps.cpp 中第一个 Swap() 函数调用时，发现它有两个 int 参数，因此将它与原来的模板匹配。但第二次调用将两个 int 数组和一个 int 值用作参数，这与新模板匹配。

程序清单 8.12 twotemps.cpp

```
// twotemps.cpp -- using overloaded template functions
#include <iostream>
template <class Any>     // original template
void Swap (Any &a, Any &b);

template <class Any>     // new template
void Swap (Any *a, Any *b, int n);

void Show (int a[]);
const int Lim = 8;
int main()
{
    using namespace std;
    int i = 10, j = 20;
    cout << "i, j = " << i << ", " << j << ".\n";
    cout << "Using compiler-generated int swapper: \n";
    Swap (i, j);              // matches original template
```

```
    cout << "Now i, j = " << i << ", " << j << ".\n";

    int d1[Lim] = {0, 7, 0, 4, 1, 7, 7, 6};
    int d2[Lim] = {0, 6, 2, 0, 1, 9, 6, 9};
    cout << "Original arrays: \n";
    Show (d1);
    Show (d2);
    Swap (d1, d2, Lim);       // matches new template
    cout << "Swapped arrays: \n";
    Show (d1);
    Show (d2);

    return 0;
}

template <class Any>
void Swap (Any &a, Any &b)
{
    Any temp;
    temp = a;
    a = b;
    b = temp;
}

template <class Any>
void Swap (Any a[], Any b[], int n)
{
    Any temp;
    for (int i = 0; i < n; i++)
    {
        temp = a[i];
        a[i] = b[i];
        b[i] = temp;
    }
}

void Show (int a[])
{
    using namespace std;
    cout << a[0] << a[1] << "/";
    cout << a[2] << a[3] << "/";
    for (int i = 4; i < Lim; i++)
        cout << a[i];
    cout << endl;
}
```

注意：如果 C++编译器不是最新的版本，则可能不支持模板。新版本可能支持用关键字 typename 代替 class。早期的 C++版本对类型匹配的要求更为苛刻，为使 const int Lim 与常规 int 的模板匹配，必须使用下面的代码：

```
Swap (xd1, d2, int (Lim) );       // typecast Lim to non-const int
```

早期的 g++版本要求将模板定义放在 main() 之前。

下面是程序清单 8.12 中程序的输出：

```
i, j = 10, 20.
Using compiler-generated int swapper:
Now i, j = 20, 10.
Original arrays:
07/04/1776
07/20/1969
Swapped arrays:
```

```
07/20/1969
07/04/1776
```

8.5.2 显式具体化

假设定义了如下结构：

```
struct job
{
    char name[40];
    double salary;
    int floor;
};
```

另外，假设希望能够交换两个这种结构的内容。原来的模板使用下面的代码来完成交换：

```
temp = a;
a = b;
b = temp;
```

由于C++允许将一个结构赋给另一个结构，因此即使 Any 是一个 job 结构，上述代码也适用。但是，假设只想交换 salary 和 floor 成员，而不交换 name 成员，则需要使用不同的代码，但 Swap() 的参数将保持不变（两个 job 结构的引用），因此无法使用模板重载来提供其他的代码。

不过，可以提供一个具体化函数定义——称为显式具体化（explicit specialization），其中包含所需的代码。当编译器找到与函数调用匹配的具体化定义时，将使用该定义，而不再寻找模板。

具体化机制随着C++的演变而不断变化。下面先介绍C++标准定义的形式，然后再介绍较早的编译器支持的两种形式。

1. 第三代具体化（ISO/ANSI C++标准）

C++试验了本章后面将介绍的具体化方法后，C++标准选择了下面的方法：

- 对于给定的函数名，可以有非模板函数、模板函数和显式具体化模板函数以及它们的重载版本。
- 显式具体化的原型和定义应以 template<>打头，并通过名称来指出类型。
- 具体化将覆盖常规模板，而非模板函数将覆盖具体化和常规模板。

下面是用于交换 job 结构的非模板函数、模板函数和具体化的原型：

```
// non-template function prototype
void Swap (job &, job &);

// template prototype
template <class Any>
void Swap (Any &, Any &);

// explicit specialization for the job type
template <> void Swap<job> (job &, job &);
```

前面指出，如果有多个原型，则编译器在选择原型时，非模板版本将优先于显式具体化和模板版本，而显式具体化将优先于使用模板生成的版本。例如，在下面的代码中，第一次调用 Swap() 时使用通用版本，而第二次调用使用基于 job 类型的显式具体化版本。

```
...
template <class Any>        // template
void Swap (Any &, Any &);

// explicit specialization for the job type
template <> void Swap<job> (job &, job &);
int main()
{
   double u, v;
   ...
   Swap (u, v);  // use template
   job a, b;
```

```
    ...
    Swap (a, b);  // use void Swap<job> (job &, job &)
}
```

Swap<job>中的<job>是可选的，因为函数的参数类型表明，这是 job 的一个具体化。因此，该原型也可以这样编写：

```
template <> void Swap (job &, job &);  // simpler form
```

稍后将介绍 C++标准之前使用的方法，以应对必须使用老式编译器的情况，不过首先来看一下显式具体化的工作方式。

显式具体化范例

程序清单 8.13 演示了显式具体化的工作方式，它是按 C++标准进行的。

程序清单 8.13　twoswap.cpp

```
// twoswap.cpp -- specialization overrides a template
#include <iostream>
template <class Any>
void Swap (Any &a, Any &b);

struct job
{
    char name[40];
    double salary;
    int floor;
};

// explicit specialization
template <> void Swap<job> (job &j1, job &j2);
void Show (job &j);

int main()
{
    using namespace std;
    cout.precision (2);
    cout.setf (ios:: fixed, ios:: floatfield);
    int i = 10, j = 20;
    cout << "i, j = " << i << ", " << j << ".\n";
    cout << "Using compiler-generated int swapper: \n";
    Swap (i, j);    // generates void Swap (int &, int &)
    cout << "Now i, j = " << i << ", " << j << ".\n";

    job sue = {"Susan Yaffee", 73000.60, 7};
    job sidney = {"Sidney Taffee", 78060.72, 9};
    cout << "Before job swapping: \n";
    Show (sue);
    Show (sidney);
    Swap (sue, sidney); // uses void Swap (job &, job &)
    cout << "After job swapping: \n";
    Show (sue);
    Show (sidney);

    return 0;
}

template <class Any>
void Swap (Any &a, Any &b)   // general version
{
    Any temp;
    temp = a;
```

```
    a = b;
    b = temp;
}

// swaps just the salary and floor fields of a job structure

template <> void Swap<job> (job &j1, job &j2) // specialization
{
    double t1;
    int t2;
    t1 = j1.salary;
    j1.salary = j2.salary;
    j2.salary = t1;
    t2 = j1.floor;
    j1.floor = j2.floor;
    j2.floor = t2;
}

void Show (job &j)
{
    using namespace std;
    cout << j.name << ": $" << j.salary
        << " on floor " << j.floor << endl;
}
```

注意：程序清单 8.13 中的程序需要 ISO/ANSI C++支持。

下面是该程序的输出：

```
i, j = 10, 20.
Using compiler-generated int swapper:
Now i, j = 20, 10.
Before job swapping:
Susan Yaffee: $73000.60 on floor 7
Sidney Taffee: $78060.72 on floor 9
After job swapping:
Susan Yaffee: $78060.72 on floor 9
Sidney Taffee: $73000.60 on floor 7
```

2. 早期的具体化方法

如果程序清单 8.13 在读者使用的编译器中无法运行，则必须采用早期的用法。最简单的方法是，为要处理的类型定义一个常规函数。也就是说，将程序清单 8.13 中的：

```
template <> void Swap<job> (job &j1, job &j2);
```

替换为：

```
void Swap (int & n, int & m);   // regular prototype
```

并将：

```
template <> void Swap<job> (job &j1, job &j2) // specialization
{
...
}
```

替换为：

```
void Swap (job &j1, job &j2)  // regular function
{
...// code unchanged
}
```

当编译器遇到函数调用 Swap（sue，sidney）时，必须选择是使用模板来生成函数定义，还是使用非模板函数 Swap（job &，job &）。原来的模板工具（和当前标准）让编译器使用非模板版本。

如果不是这样，则使用的可能是安装了非官方草案版模板的编译器，在这种编译器中，模板优先于常规函数。如果采用的是这种规则，编译器将使用模板 Swap()，而不是 job 版本。因此，要获得理想的效果，

必须使用形式不那么现代的显式具体化。也就是说，不能使用：

```
template <> void Swap<job> (job &j1, job &j2);  // ISO/ANSI C++
```

而应使用：

```
void Swap<job> (job &, job &);  // earlier form of specialization
```

注意，这里没有 template <>前缀。应对函数头做相同的调整，也就是说，将下面的代码：

```
template <> void Swap<job> (job &j1, job &j2) // specialization
{
...
}
```

改为：

```
void Swap<job> (job &j1, job &j2) // old form  of specialization
{
...// code unchanged
}
```

如果使用的是最新的 C++编译器（希望如此），就不必做这些调整。

8.5.3　实例化和具体化

为进一步了解模板，必须理解术语实例化和具体化。记住，在代码中包含函数模板本身并不会生成函数定义，它只是一个用于生成函数定义的方案。编译器使用模板为特定类型生成函数定义时，得到的是模板实例（instantiation）。例如，在程序清单 8.13 中，函数调用 Swap（i，j）导致编译器生成 Swap()的一个实例，该实例使用 int 类型。模板并非函数定义，但使用 int 的模板实例是函数定义。这种实例化方式被称为隐式实例化（implicit instantiation），因为编译器之所以知道需要进行定义，是由于程序调用 Swap()函数时提供了 int 参数。

最初，编译器只能通过隐式实例化，来使用模板生成函数定义，不过现在 C++还允许显式实例化（explicit instantiation）。这意味着可以直接命令编译器创建特定的实例，如 Swap<int>()。其句法是，声明所需的种类——用<>符号指示类型，并在声明前加上关键字 template：

```
template void Swap<int> (int, int);  // explicit instantiation
```

实现了这种特性的编译器看到上述声明后，将使用 Swap()模板生成一个使用 int 类型的实例。也就是说，该声明的意思是“使用 Swap()模板生成 int 类型的函数定义”。

与显式实例化不同的是，显式具体化使用下面两个等价的声明之一：

```
template <> void Swap<int> (int &, int &);  // explicit specialization
template <> void Swap (int &, int &);       // explicit specialization
```

区别在于，这些声明的意思是“不要使用 Swap()模板来生成函数定义，而应使用独立的、专门的函数定义显式地为 int 类型生成函数定义”。这些原型必须有自己的函数定义。显式具体化声明在关键字 template 后包含<>，而显式实例化没有。

警告：试图在同一个编程单元中使用同一种类型的显式实例和显式具体化将出错。

隐式实例化、显式实例化和显式具体化统称为具体化（specialization）。它们的相同之处在于，它们表示的都是使用具体类型的函数定义，而不是通用描述。

引入显式实例化后，必须使用新的句法——在声明中使用前缀 template 和 template <>，以区分显式实例化和显式具体化。通常，功能越多，句法规则也越多。下面的代码片段总结了这些概念：

```
...
template <class Any>
void Swap (Any &, Any &);  // template prototype

template <> void Swap<job> (job &, job &);  // explicit specialization for job
int main (void)
{
   template void Swap<char> (char &, char &);  // explicit instantiation for char
   short a, b;
   ...
```

```
    Swap (a, b);   // implicit template instantiation for short
    job n, m;
    ...
    Swap (n, m); // use explicit specialization for job
    char g, h;
    ...
     Swap (g, h); // use explicit template instantiation for char
     ...
}
```

编译器看到 char 的显式实例化后，将使用模板定义来生成 Swap() 的 char 版本。对于其他 Swap() 调用，编译器根据函数调用中实际使用的参数，生成相应的版本；例如，当编译器看到函数调用 Swap（a，b）后，将生成 Swap() 的 short 版本，因为两个参数的类型都是 short。当编译器看到 Swap（n，m）后，将使用为 job 类型提供的独立定义（显式具体化）；当编译器看到 Swap（g，h）后，将使用处理显式实例化时生成的模板具体化。

8.5.4 编译器选择使用哪个函数版本

对于函数重载、函数模板和函数模板重载，C++需要（且有）一个定义良好的策略，来决定为函数调用使用哪一个函数定义，尤其是有多个参数时。这个过程称为重载解析（overloading resolution）。详细解释这个策略需要将近一章的篇幅，因此我们先大致了解一下这个过程是如何进行的：

- 第 1 步：创建候选函数列表。其中包含与被调用函数的名称相同的函数和模板函数。
- 第 2 步：使用候选函数列表创建可行函数列表。这些都是参数数目正确的函数，为此有一个隐式转换序列，其中包括实参类型与相应的形参类型完全匹配的情况。例如，使用 float 参数的函数调用可以将该参数转换为 double，从而与 double 形参匹配，而模板可以为 float 生成一个实例。
- 第 3 步：确定是否有最佳的可行函数。如果有，则使用它，否则该函数调用出错。

考虑只有一个函数参数的情况，如下面的调用：

```
may('B');   // actual argument is type char
```

首先，编译器将寻找候选者，即名称为 may() 的函数和函数模板。然后寻找那些可以用一个参数调用的函数。例如，下面的函数符合要求，因为其名称与被调用的函数相同：

```
void may(int);                          // #1
float may(float, float = 3);            // #2
void may(char);                         // #3
char * may(const char *);               // #4
char may(const char &);                 // #5
template<class T> void may(const T &);  // #6
template<class T> void may(T *);        // #7
```

注意，只考虑特征标，而不考虑返回类型。其中的两个候选函数（#4 和#7）不可行，因为整数类型不能被隐式地转换（即没有显式强制类型转换）为指针类型。剩余的一个模板可用来生成具体化，其中 T 被替换为 char 类型。这样剩下 5 个可行的函数，其中的每一个函数，如果它是声明的惟一一个函数，都可以被使用。

接下来，编译器必须确定哪个可行函数是最佳的。它查看为使函数调用参数与可行的候选函数的参数匹配所需要进行的转换。通常，从最佳到最差的顺序如下：

1. 完全匹配，但常规函数优先于模板。
2. 提升转换（例如，char 和 shorts 自动转换为 int，float 自动转换为 double）。
3. 标准转换（例如，int 转换为 char，long 转换为 double）。
4. 用户定义的转换，如类声明中定义的转换。

例如，函数#1 优于函数#2，因为 char 到 int 的转换是提升转换（参见第 3 章），而 char 到 float 的转换是标准转换（参见第 3 章）。函数#3、函数#5 和函数#6 都优于函数#1 和#2，因为它们都是完全匹配的。#3 和#5 优于#6，因为#6 函数是模板。这种分析引出了两个问题。什么是完全匹配？如果两个函数都完全匹配，情况将如何？通常，有两个函数完全匹配是一种错误，但这一规则有两个例外。显然，我们需要对这一点

做更深入的探讨。

1. 完全匹配和最佳匹配

进行完全匹配时，C++允许某些“无关紧要的转换”。表 8.1 列出了这些转换——Type 表示任意类型。例如，int 实参与 int &形参完全匹配。注意，Type 可以是 char &这样的类型，因此这些规则包括从 char & 到 const char &的转换。Type（argument-list）意味着用作实参的函数名与用作形参的函数指针只要返回类型和参数列表相同，就是匹配的（第 7 章介绍了函数指针以及为何可以将函数名作为参数传递给接受函数指针的函数）。本章后面将介绍关键字 volatile。

表 8.1　完全匹配允许的无关紧要转换

从 实 参	到 形 参	从 实 参	到 形 参
Type	Type &	Type	const Type
Type &	Type	Type	volatile Type
Type [ ]	* Type	Type *	const Type
Type (argment-list)	Type (*) (argument-list)	Type *	volatile Type *

假设有下面的函数代码：

```
struct blot {int a; char b[10]; };
blot ink = {25, "spots"};
...
recycle (ink);
```

在这种情况下，下面所有的原型都是完全匹配的：

```
void recycle (blot);            // #1 blot-to-blot
void recycle (const blot);      // #2 blot-to- (const blot)
void recycle (blot &);          // #3 blot-to- (blot &)
void recycle (const blot &);    // #4 blot-to- (const blot &)
```

可以预料到，如果有多个匹配的原型，则编译器将无法完成重载解析过程；如果没有最佳的可行函数，则编译器将生成一条错误消息，该消息可能会使用诸如“ambiguous（二义性）”这样的词语。

不过，有时候，即使两个函数都完全匹配，仍可完成重载解析。首先，指向非 const 数据的指针和引用优先与非 const 的指针和引用参数匹配。也就是说，在 recycle()范例中，如果只定义了函数#3 和#4 是完全匹配的，则将选择#3，因为 ink 没有被声明为 const。不过，const 和非 const 之间的区别只适用于指针和引用指向的数据。也就是说，如果只定义了#1 和#2，则将出现二义性错误。

另一种一个完全匹配优于另一个的情况是，其中一个是非模板函数，而另一个不是。在这种情况下，非模板函数将优先于模板函数（包括显式具体化）。

如果两个完全匹配的函数都是模板函数，则较具体的模板函数优先。例如，这意味着显式具体化将优于使用模板隐式生成的具体化：

```
struct blot {int a; char b[10]; };
template <class Type> void recycle (Type t); // template
template <> void recycle<blot> (blot & t);   // specialization for blot
...
blot ink = {25, "spots"};
...
recycle (ink);  // use specialization
```

术语“最具体（most specialized）”并不一定意味着显式具体化，而是指编译器推断使用哪种类型时执行的转换最少。例如，请看下面两个模板：

```
template <class Type> void recycle (Type t);     //#1
template <class Type> void recycle (Type * t);   //#2
```

假设包含这些模板的程序也包含如下代码：

```
struct blot {int a; char b[10]; };
blot ink = {25, "spots"};
...
```

```
recycle(&ink); // address of a structure
```

recycle（&ink）调用与#1 模板匹配，匹配时将 Type 解释为 blot *。recycle（&ink）函数调用也与#2 模板匹配，这次 Type 被解释为 ink。因此将两个隐式实例——recycle<blot *>（blot *）和 recycle <blot>（blot *）发送到可行函数池中。

在这两个模板函数中，recycle<blot *>（blot *）被认为是更具体的，因为在生成过程中，它需要进行的转换更少。也就是说，#2 模板已经显式指出，函数参数是指向 Type 的指针，因此可以直接用 blot 标识 Type；而#1 模板将 Type 作为函数参数，因此 Type 必须被解释为指向 blot 的指针。也就是说，在#2 模板中，Type 已经被具体化为指针，因此说它“更具体”。

用于找出最具体的模板的规则被称为函数模板的部分排序规则（partial ordering rules）。和显式实例一样，这也是 C++新增的特性。

部分排序规则范例

我们先看一个完整的程序，它使用部分排序规则来确定要使用哪个模板定义。程序清单 8.14 有两个用来显示数组内容的模板定义。第一个定义（模板 A）假设作为参数传递的数组中包含了要显示的数据；第二个定义（模板 B）假设数组参数中包含了一个指针，该指针指向要显示的数据。

程序清单 8.14 temptempover.cpp

```
// tempover.cpp --- template overloading
#include <iostream>
template <typename T>           // template A
void ShowArray(T arr[], int n);

template <typename T>           // template B
void ShowArray(T * arr[], int n);

struct debts
{
    char name[50];
    double amount;
};

int main(void)
{
    using namespace std;
    int things[6] = {13, 31, 103, 301, 310, 130};
    struct debts mr_E[3] =
    {
        {"Ima Wolfe", 2400.0},
        {"Ura Foxe", 1300.0},
        {"Iby Stout", 1800.0}
    };
    double * pd[3];
// set pointers to the amount members of the structures in the arr mr_E
    for (int i = 0; i < 3; i++)
        pd[i] = &mr_E[i].amount;

    cout << "Listing Mr. E's counts of things:\n";
// things is an array of int
    ShowArray(things, 6);  // uses template A
    cout << "Listing Mr. E's debts:\n";
// pd is an array of pointers to double
    ShowArray(pd, 3);      // uses template B (more specialized)
    return 0;
}

template <typename T>
```

```
void ShowArray(T arr[], int n)
{
    using namespace std;
    cout << "template A\n";
    for (int i = 0; i < n; i++)
        cout << arr[i] << ' ';
    cout << endl;
}
template <typename T>
void ShowArray(T * arr[], int n)
{
    using namespace std;
    cout << "template B\n";
    for (int i = 0; i < n; i++)
        cout << *arr[i] << ' ';
    cout << endl;
}
```

来看一下下面的函数调用：

```
ShowArray(things, 6);
```

标识符 things 是一个 int 数组的名称，因此与下面的模板匹配：

```
template <typename T>           // template A
void ShowArray(T arr[], int n);
```

其中 T 被替换为 int 类型。

接下来，看一下下面的函数调用：

```
ShowArray(pd, 3);
```

其中，pd 是一个 double *数组的名称。这与模板 A 匹配：

```
template <typename T>           // template A
void ShowArray(T arr[], int n);
```

其中，T 被替换为类型 double *。在这种情况下，模板函数将显示 pd 数组的内容，即 3 个地址。该函数调用也与模板 B 匹配：

```
template <typename T>           // template B
void ShowArray(T * arr[], int n);
```

在这里，T 被替换为类型 double，而函数将显示被解除引用的元素*arr[i]，即数组内容指向的 double 值。在这两个模板中，模板 B 更具体，因为它做了特定的假设——数组内容是指针，因此被使用。

下面是程序清单 8.14 中程序的输出：

```
Listing Mr. E's counts of things:
template A
13 31 103 301 310 130
Listing Mr. E's debts:
template B
2400 1300 1800
```

如果将模板 B 从程序中删除，则编译器将使用模板 A 来显示 pd 的内容，因此显示的将是地址，而不是值。

简而言之，重载解析将寻找最匹配的函数。如果只存在一个这样的函数，则选择它；如果存在多个这样的函数，但其中只有一个是非模板函数，则选择该函数；如果存在多个适合的函数，并且它们都为模板函数，但其中有一个函数比其他函数更具体，则选择该函数。如果有多个同样合适的非模板函数或模板函数，但没有一个函数比其他函数更具体，则函数调用将是不确定的，因此是错误的；当然，如果不存在匹配的函数，则也是错误。

2. 有多个参数的函数

将有多个参数的函数调用与有多个参数的原型进行匹配时，情况将非常复杂。编译器必须考虑所有参数的匹配情况。如果找到比其他可行函数都合适的函数，则选择该函数。一个函数要比其他函数都合适，其所有参数的匹配程度都必须不比其他函数差，同时至少有一个参数的匹配程度比其他函数都高。

本书并不是要解释复杂范例的匹配过程，这些规则只是为了让任何一组函数原型和模板都存在确定的结果。

8.6 总结

C++扩展了 C 语言的函数功能。通过将 inline 关键字用于函数定义，并在首次调用该函数前提供其函数定义，可以使得 C++编译器将该函数视为内联函数。也就是说，编译器不是让程序跳到独立的代码段，以执行函数，而是用相应的代码替换函数调用。只有在函数很短时才能采用内联方式。

引用变量是一种伪装指针，它允许为变量创建别名（另一个名称）。引用变量主要被用作处理结构和类对象的函数的参数。通常，被声明为特定类型引用的标识符只能指向这种类型的数据；然而，如果一个类（如 ofstream）是从另一个类（如 ostream）派生出来的，则基类引用可以指向派生类对象。

C++原型让您能够定义参数的默认值。如果函数调用省略了相应的参数，则程序将使用默认值；如果函数调用提供了参数值，则程序将使用这个值（而不是默认值）。只能在参数列表中从右到左提供默认参数。因此，如果为某个参数提供了默认值，则必须为该参数右边所有的参数提供默认值。

函数的特征标是其参数列表。程序员可以定义两个同名函数，只要其特征标不同。这被称为函数多态或函数重载。通常，通过重载函数来为不同的数据类型提供相同的服务。

函数模板自动完成重载函数的过程。只需使用通用类型和具体算法来定义函数，编译器将为程序中使用的特定参数类型生成正确的函数定义。

8.7 复习题

1．哪种函数适合定义为内联函数？

2．假设 song() 函数的原型如下：

```
void song (char * name, int times);
```

a．如何修改原型，使 times 的默认值为 1？

b．函数定义需要做哪些修改？

c．能否为 name 提供默认值“O. My Papa”？

3．编写 iquote() 的重载版本——显示其用双引号括起的参数。编写 3 个版本：一个用于 int 参数，一个用于 double 参数，另一个用于 string 参数。

4．下面是一个结构模板：

```
struct box
{
    char maker[40];
    float height;
    float width;
    float length;
    float volume;
};
```

a．请编写一个函数，它将 box 结构的引用作为形参，并显示每个成员的值。

b．请编写一个函数，它将 box 结构的引用作为形参，并将 volume 成员设置为其他 3 边的乘积。

5．指出下面每个目标是否可以使用默认参数和/或函数重载完成，或者这两种方法都无法完成，并提供合适的原型。

a．mass（density，volume）返回密度为 density、体积为 volume 的物体的质量，而 mass（denstity）返回密度为 density、体积为 1.0 立方米的物体的质量。这些值的类型都为 double。

b．repeat（10，“I'm OK”）将指定的字符串显示 10 次，而 repeat（“But you're kind of stupid”）将指定

的字符串显示 5 次。

c. average（3，6）返回两个 int 参数的平均值（int 类型），而 average（3.0，6.0）返回两个 double 值的平均值（double 类型）。

d. mangle（"I'm glad to meet you"）根据是将值赋给 char 变量还是 char*变量，分别返回字符 I 和指向字符串"I'm mad to gleet you"的指针。

6. 编写返回两个参数中较大值的函数模板。

7. 对于复习题 6 的模板和复习题 4 的 box 结构，提供一个模板具体化，它接受两个 box 参数，并返回体积较大的一个。

8.8　编程练习

1. 编写通常接受一个参数（字符串的地址），并打印该字符串的函数。不过，如果提供了第二个参数（int 类型），且该参数不为 0，则该函数打印字符串的次数将为该函数被调用的次数（注意，字符串的打印次数不等于第二个参数的值，而等于函数被调用的次数）。是的，这是一个非常可笑的函数，但它让读者能够使用本章介绍的一些技术。在一个简单的程序中使用该函数，以演示该函数是如何工作的。

2. CandyBar 结构包含 3 个成员。第一个成员存储 candy bar 的品牌名称；第二个成员存储 candy bar 的重量（可能有小数）；第三个成员存储 candy bar 的热量（整数）。请编写一个程序，它使用一个这样的函数，即将 CandyBar 的引用、char 指针、double 和 int 作为参数，并用最后 3 个值设置相应的结构成员。最后 3 个参数的默认值分别为"Millennium Munch"、2.85 和 350。另外，该程序还包含一个以 CandyBar 的引用为参数，并显示结构内容的函数。请尽可能使用 const。

3. 编写一个函数，它接受一个指向 string 对象的引用作为参数，并将该 string 对象的内容转换为大写，为此可使用表 6.4 描述的函数 toupper()。然后编写一个程序，它通过使用一个循环让您能够用不同的输入来测试这个函数，该程序的运行情况如下：

```
Enter a string (q to quit): go away
GO AWAY
Next string (q to quit): good grief!
GOOD GRIEF!
Next string (q to quit): q
Bye.
```

4. 下面是一个程序框架：

```
#include <iostream>
using namespace std;
#include <cstring>    // for strlen(), strcpy()
struct stringy {
   char * str;        // points to a string
   int ct;            // length of string (not counting '\0')
   };

// prototypes for set(), show(), and show()go here
int main()
{
   stringy beany;
   char testing[] = "Reality isn't what it used to be.";

    set (beany, testing);    // first argument is a reference,
          // allocates space to hold copy of testing,
          // sets str member of beany to point to the
          // new block, copies testing to new block,
          // and sets ct member of beany
   show (beany);      // prints member string once
   show (beany, 2);   // prints member string twice
```

```
    testing[0] = 'D';
    testing[1] = 'u';
    show(testing);    // prints testing string once
    show(testing, 3); // prints testing string thrice
    show("Done! ");
    return 0;
}
```

请提供其中描述的函数和原型，从而完成该程序。注意，应有两个 show() 函数，每个都使用默认参数。请尽可能使用 cosnt 参数。set() 使用 new 分配足够的空间来存储指定的字符串。这里使用的技术与设计和实现类时使用的相似。（可能还必须修改头文件的名称，删除 using 编译指令，这取决于所用的编译器。）

5. 编写模板函数 max5()，它将一个包含 5 个 T 类型元素的数组作为参数，并返回数组中最大的元素（由于长度固定，因此可以在循环中使用硬编码，而不必通过参数来传递）。在一个程序中使用该函数，将 T 替换为一个包含 5 个 int 值的数组和一个包含 5 个 dowble 值的数组，以测试该函数。

6. 编写模板函数 maxn()，它将由一个 T 类型元素组成的数组和一个表示数组元素数目的整数作为参数，并返回数组中最大的元素。在程序对它进行测试，该程序使用一个包含 6 个 int 元素的数组和一个包含 4 个 double 元素的数组来调用该函数。程序还包含一个具体化，它将 char 指针数组和数组中的指针数量作为参数，并返回最长的字符串的地址。如果有多个这样的字符串，则返回其中第一个字符串的地址。使用由 5 个字符串指针组成的数组来测试该具体化。

7. 修改程序清单 8.14，使模板函数返回数组元素的总和，而不是显示数组的内容。程序应显示 thing 的总和以及所有 debt 的总和。

第 9 章　内存模型和名称空间

本章内容包括：

- 单独编译。
- 存储持续性、作用域和链接性。
- 布局（placement）new 操作符。
- 名称空间。

C++为在内存中存储数据方面提供了多种选择。可以选择数据保留在内存中的时间长度（存储持续性）以及程序的哪一部分可以访问数据（作用域和链接）等。可以使用 new 来动态地分配内存，而布局 new 操作符提供了这种技术的一种变种。C++名称空间是另一种控制访问权的方式。通常，大型程序都由多个源代码文件组成，这些文件可能共享一些数据。这样的程序涉及到程序文件的单独编译，本章将首先介绍这个主题。

9.1　单独编译

和 C 语言一样，C++也允许甚至鼓励程序员将组件函数放在独立的文件中。第 1 章介绍过，可以单独编译这些文件，然后将它们链接成可执行的程序（通常，C++编译器既编译程序，也管理链接器）。如果只修改了一个文件，则可以只重新编译该文件，然后将它与其他文件的编译版本链接。这使得大程序的管理更便捷。另外，大多数 C++环境都提供了其他工具来帮助管理。例如，UNIX 和 Linux 系统都具有 make 程序，可以跟踪程序依赖的文件以及这些文件的最后修改时间。运行 make 时，如果它检测到上次编译后修改了源文件，make 将记住重新构建程序所需的步骤。Borland C++、Microsoft Visual C++和 Metrowerks CodeWarrior IDE（integrated development enviroment，集成开发环境）在 Project 菜单中提供了类似的工具。

现在看一个简单的范例。我们不是要从中了解编译的细节（这取决于实现），而是要重点介绍更通用的方面，如设计。

例如，假设程序员决定分解程序清单 7.12 中的程序，将支持函数放在一个独立的文件中。清单 7.12 将直角坐标转换为极坐标，然后显示结果。不能简单地以 main()之后的虚线为界，将原来的文件分为两个。问题在于，main()和其他两个函数使用了同一个结构声明，因此两个文件都应包含该声明。简单地将它们输入进去无疑是自找麻烦。即使正确地复制了结构声明，如果以后要作修改，则必须记住对这两组声明都进行修改。简而言之，将一个程序放在多个文件中将引出新的问题。

谁希望出现更多的问题呢？C 和 C++的开发人员都不希望，因此他们提供了#include 来处理这种情况。与其将结构声明加入到每一个文件中，不如将其放在头文件中，然后在每一个源代码文件中包含该头文件。这样，要修改结构声明时，只需在头文件中做一次改动即可。另外，也可以将函数原型放在头文件中。因此，可以将原来的程序分成三部分：

- 头文件：包含结构声明和使用这些结构的函数的原型。
- 源代码文件：包含与结构有关的函数的代码。
- 源代码文件：包含调用与结构相关的函数的代码。

这是一种非常有用的组织程序的策略。例如，如果编写另一个程序时，也需要使用这些函数，则只需包含头文件，并将函数文件添加到工程列表或 make 列表中即可。另外，这种组织方法也反映了 OOP 方法。一个文件（头文件）包含了用户定义类型的定义；另一个文件包含操纵用户定义类型的函数的代码。这两个文件组成了一个软件包，可用于各种程序中。

请不要将函数定义或变量声明放到头文件中。这样做对于简单的情况可能是可行的，但通常会引来麻烦。例如，如果在头文件包含一个函数定义，然后在其他两个文件（属于同一个程序）中包含该头文件，则同一个程序中将包含同一个函数的两个定义，除非函数是内联的，否则这将出错。下面列出了头文件中常包含的内容：

- 函数原型。
- 使用#define 或 const 定义的符号常量。
- 结构声明。
- 类声明。
- 模板声明。
- 内联函数。

将结构声明放在头文件中是可以的，因为它们不创建变量，而只是在源代码文件中声明结构变量时，告诉编译器如何创建该结构变量。同样，模板声明不是将被编译的代码，它们指示编译器如何生成与源代码中的函数调用相匹配的函数定义。被声明为 const 的数据和内联函数有特殊的链接属性（稍后将介绍），因此可以将其放在头文件中，而不会引起问题。

程序清单 9.1、9.2 和 9.3 是将程序清单 7.12 分成几个独立部分后得到的结果。注意，在包含头文件时，我们使用“coordin.h”，而不是<coodin.h>。如果文件名包含在尖括号中，则 C++编译器将在存储标准头文件的主机系统的文件系统中查找；但如果文件名包含在双引号中，则编译器将首先查找当前的工作目录或源代码目录（或其他目录，这取决于编译器）。如果没有在那里找到头文件，则将在标准位置查找。因此在包含自己的头文件时，应使用引号而不是尖括号。

图 9.1 简要地说明了在 UNIX 系统中将该程序组合起来的步骤。注意，只需执行编译命令 CC 即可，其他步骤将自动完成。g++和 gpp 命令行编译器以及 Borland C++命令行编译器（bcc32.exe）的行为类似。Borland C++、Turbo C++、Metrowerks CodeWarrior 和 Microsoft Visual C++基本上执行同样的步骤，但正如第 1 章介绍的，启动这个过程的方式不同——使用能够创建工程并将其与源代码文件关联起来的菜单。注意，只需将源代码文件加入到工程中，而不用加入头文件。这是因为#include 指令管理头文件。另外，不要使用#include 来包含源代码文件，这样做将导致多重声明。

警告：在 IDE 中，不要将头文件加入到工程列表中，也不要在源代码文件中使用#include 来包含其他源代码文件。

程序清单 9.1 coordin.h

```
// coordin.h -- structure templates and function prototypes
// structure templates
#ifndef COORDIN_H_
#define COORDIN_H_

struct polar
{
    double distance;    // distance from origin
    double angle;       // direction from origin
};
struct rect
{
    double x;       // horizontal distance from origin
    double y;       // vertical distance from origin
};
// prototypes
```

```
polar rect_to_polar (rect xypos) ;
void show_polar (polar dapos) ;

#endif
```

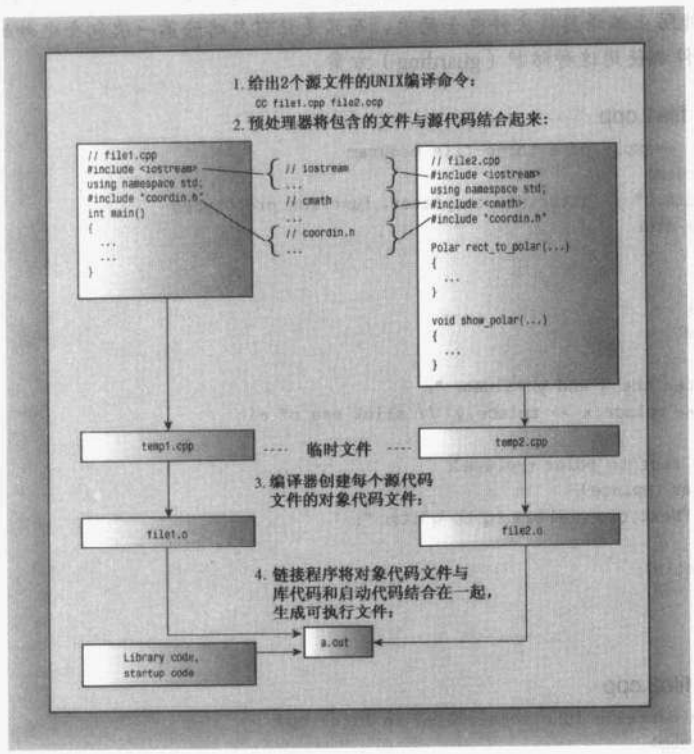


图 9.1　在 UNIX 系统中编译由多个文件组成的 C++程序

头文件管理

在同一个文件中只能将同一个头文件包含一次。记住这个规则很容易，但很可能在不知情的情况下将头文件包含多次。例如，可能使用包含了另外一个头文件的头文件。有一种标准的 C/C++技术可以避免多次包含同一个头文件。它是基于预处理器编译指令#ifndef（即 if not defined）的。下面的代码片段：

```
#ifndef COORDIN_H_
...
#endif
```

意味着仅当以前没有使用预处理器编译指令#define 定义名称 COORDIN_H_时，才处理#ifndef 和#endif 之间的语句。

通常，使用#define 语句来创建符号常量，如下所示：

```
#define MAXIMUM 4096
```

但只要将#define 用于名称，就足以完成该名称的定义，如下所示：

```
#define COORDIN_H_
```

程序清单 9.1 使用这种技术是为了将文件内容包含在#ifndef 中：

```
#ifndef COORDIN_H_
#define COORDIN_H_
// place include file contents here
#endif
```

编译器首次遇到该文件时，名称 COORDIN_H 没有定义（我们根据 include 文件名来选择名称，并加上一些下划线，以创建一个在其他地方不太可能被定义的名称）。在这种情况下，编译器将查看#ifndef 和#endif 之间的内容（这正是我们希望的），并读取定义 COORDIN_H 的一行。如果在同一个文件中遇到其他包含 coordin.h 的代码，编译器将知道 COORDIN_H 已经被定义了，从而跳到#endfi 后面的一行上。注意，这种方法并不能防止编译器将文件包含两次，而只是让它忽略除第一次包含之外的所有内容。大多数标准 C 和 C++头文件都使用这种防护（guarding）方案。

程序清单 9.2 file1.cpp

```
// file1.cpp -- example of a three-file program
#include <iostream>
#include "coordin.h" // structure templates, function prototypes
using namespace std;
int main()
{
    rect rplace;
    polar pplace;

    cout << "Enter the x and y values: ";
    while (cin >> rplace.x >> rplace.y) // slick use of cin
    {
        pplace = rect_to_polar (rplace);
        show_polar (pplace);
        cout << "Next two numbers (q to quit): ";
    }
    cout << "Bye!\n";
    return 0;
}
```

程序清单 9.3 file2.cpp

```
// file2.cpp -- contains functions called in file1.cpp
#include <iostream>
#include <cmath>
#include "coordin.h" // structure templates, function prototypes

// convert rectangular to polar coordinates
polar rect_to_polar (rect xypos)
{
    using namespace std;
    polar answer;

    answer.distance =
        sqrt (xypos.x * xypos.x + xypos.y * xypos.y);
    answer.angle = atan2 (xypos.y, xypos.x);
    return answer;      // returns a polar structure
}

// show polar coordinates, converting angle to degrees
void show_polar (polar dapos)
{
    using namespace std;
    const double Rad_to_deg = 57.29577951;

    cout << "distance = " << dapos.distance;
    cout << ", angle = " << dapos.angle * Rad_to_deg;
    cout << " degrees\n";
}
```

将这两个源代码文件和新的头文件一起进行编译和链接，将生成一个可执行程序。下面是该程序的运

行情况：

```
Enter the x and y values: 120 80
distance = 144.222, angle = 33.6901 degrees
Next two numbers (q to quit): 120 50
distance = 130, angle = 22.6199 degrees
Next two numbers (q to quit): q
```

顺便说一句，虽然我们讨论的是根据文件进行单独编译，但为保持通用性，C++标准使用了术语翻译单元（translation unit），而不是文件；文件并不是计算机组织信息时的惟一方式。

多个库的链接

C++标准允许每个编译器设计人员以他认为合适的方式实现名称修饰（参见第 8 章），因此由不同编译器创建的二进制模块（对象代码文件）很可能无法正确地链接。也就是说，两个编译器将为同一个函数生成不同的修饰名称。名称的不同将使链接器无法将一个编译器生成的函数调用与另一个编译器生成的函数定义匹配。在链接编译模块时，请确保所有对象文件或库都是由同一个编译器生成的。如果有源代码，通常可以用自己的编译器重新编译源代码来消除链接错误。

9.2　存储持续性、作用域和链接性

介绍过多文件程序后，接下来扩展第 4 章对内存方案的讨论，即存储类别如何影响信息在文件间的共享。现在读者阅读第 4 章已经有一段时间了，因此先复习一下有关内存的知识。C++使用 3 种不同的方案来存储数据，这些方案的区别就在于数据保留在内存中的时间。

- **自动存储持续性**：在函数定义中声明的变量（包括函数参数）的存储持续性为自动的。它们在程序开始执行其所属的函数或代码块时被创建，在执行完函数或代码块时，它们使用的内存被释放。C++有两种存储持续性为自动的变量。
- **静态存储持续性**：在函数定义外定义的变量和使用关键字 static 定义的变量的存储持续性都为静态。它们在程序整个运行过程中都存在。C++有 3 种存储持续性为静态的变量。
- **动态存储持续性**：用 new 操作符分配的内存将一直存在，直到使用 delete 操作符将其释放或程序结束为止。这种内存的存储持续性为动态，有时又称为自由存储（free store）。

下面介绍其他内容，包括关于各种变量何时在作用域内或可见（可被程序使用）以及链接性的细节。链接性决定了哪些信息可在文件间共享。

9.2.1　作用域和链接

作用域（scope）描述了名称在文件（翻译单元）的多大范围内可见。例如，函数中定义的变量可在该函数中使用，但不能在其他函数中使用；而在文件中的函数定义之前定义的变量则可在所有函数中使用。链接性（linkage）描述了名称如何在不同单元间共享。链接性为外部的名称可在文件间共享，链接性为内部的名称只能由一个文件中的函数共享。自动变量的名称没有链接性，因为它们不能共享。

C++变量的作用域有多种。作用域为局部的变量只在定义它的代码块中可用。代码块是由花括号括起的一系列语句。例如函数体就是代码块，但可以在函数体中嵌入其他代码块。作用域为全局（也叫文件作用域）的变量在定义位置到文件结尾之间都可用。自动变量的作用域为局部，静态变量的作用域是全局还是局部取决于它是如何被定义的。在函数原型作用域（function prototype scope）中使用的名称只在包含参数列表的括号内可用（这就是为什么这些名称是什么以及是否出现都不重要的原因）。在类中声明的成员的作用域为整个类（参见第 10 章）。在名称空间中声明的变量的作用域为整个名称空间（由于名称空间已经引入到 C++语言中，因此全局作用域是名称空间作用域的特例）。

C++函数的作用域可以是整个类或整个名称空间（包括全局的），但不能是局部的（因为不能在代码块内定义函数，如果函数的作用域为局部，则只对它自己是可见的，因此不能被其他函数调用。这样的函数

将无法运行）。

不同的C++存储方式是通过存储持续性、作用域和链接性来描述的。下面来看看各种C++存储方式的这些特征。首先介绍引入名称空间之前的情况，然后看一看名称空间带来的影响。

9.2.2 自动存储持续性

在默认情况下，在函数中声明的函数参数和变量的存储持续性为自动，作用域为局部，没有链接性。也就是说，如果在main()中声明了一个名为texas的变量，并在函数oil()中也声明了一个名为texas变量，则创建了两个独立的变量——只有在定义它们的函数中才能使用它们。对oil()中的texas执行的任何操作都不会影响main()中的texas，反之亦然。另外，当程序开始执行这些变量所属的代码块时，将为其分配内存；当函数结束时，这些变量都将消失（注意，执行到代码块时，将为变量分配内存，但其作用域的起点为其声明位置）。

如果在代码块中定义了变量，则该变量的存在时间和作用域将被限制在该代码块内。例如，假设在main()的开头定义了一个名为teledeli的变量，然后在main()中开始一个新的代码块，并在其中定义了一个新的变量websight，则teledeli在内部代码块和外部代码块中都是可见的，而websight就只在内部代码块中可见，它的作用域是从定义它的位置到该代码块的结尾：

```
int main()
{
   int teledeli = 5;
   {                        // websight allocated
      cout << "Hello\n";
      int websight = -2;    // websight scope begins
      cout << websight << ' ' << teledeli << endl;
   }                        // websight expires
   cout << teledeli << endl;
   ...
}
```

然而，如果将内部代码块中的变量命名为teledeli，而不是websight，使得有两个同名的变量（一个位于外部代码块中，另一个位于内部代码块中），情况将如何呢？在这种情况下，程序执行内部代码块中的语句时，将teledeli解释为局部代码块变量。我们说，新的定义隐藏了（hide）以前的定义，新定义可见，旧定义暂时不可见。在程序离开该代码块时，原来的定义又重新可见（参见图9.2）。

程序清单9.4表明，自动变量只在包含它们的函数或代码块中可见。

程序清单9.4 auto.cpp

```
// auto.cpp -- illustrating scope of automatic variables
#include <iostream>
void oil (int x);
int main()
{
   using namespace std;

   int texas = 31;
   int year = 1999;
   cout << "In main(), texas = " << texas << ", &texas =";
   cout << &texas << endl;
   cout << "In main(), year = " << year << ", &year =";
   cout << &year << endl;
   oil (texas);
   cout << "In main(), texas = " << texas << ", &texas =";
   cout << &texas << endl;
   cout << "In main(), year = " << year << ", &year =";
   cout << &year << endl;
   return 0;
}
```

```
void oil (int x)
{
    int texas = 5;

    cout << "In oil(), texas = " << texas << ", &texas =";
    cout << &texas << endl;
    cout << "In oil(), x = " << x << ", &x =";
    cout << &x << endl;
    {                           // start a block
        int texas = 113;
        cout << "In block, texas = " << texas;
        cout << ", &texas = " << &texas << endl;
              cout << "In block, x = " << x << ", &x =";
        cout << &x << endl;
    }                           // end a block
    cout << "Post-block texas = " << texas;
    cout << ", &texas = " << &texas << endl;
}
```

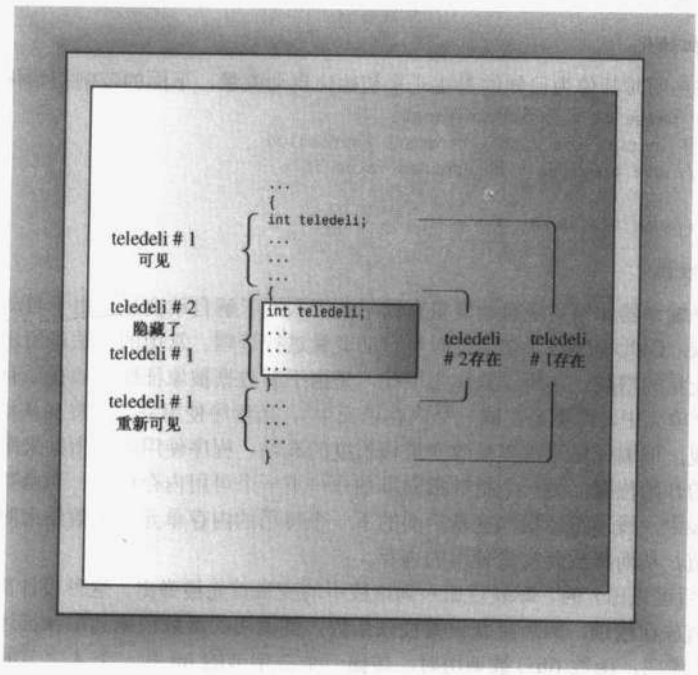


图 9.2　代码块和作用域

下面是该程序的输出：

```
In main(), texas = 31, &texas = 0012FED4
In main(), year = 1999, &year = 0012FEC8
In oil(), texas = 5, &texas = 0012FDE4
In oil(), x = 31, &x = 0012FDF4
In block, texas = 113, &texas = 0012FDD8
In block, x = 31, &x = 0012FDF4
Post-block texas = 5, &texas = 0012FDE4
In main(), texas = 31, &texas = 0012FED4
In main(), year = 1999, &year = 0012FEC8
```

在程序清单 9.4 中，3 个 texas 变量的地址各不相同，而程序使用当前可见的那个变量，因此将 113 赋给 oil()中的内部代码块中的 texas，对其他同名变量没有影响。同样，实际的地址值和地址格式随系

统而异。

现在总结一下整个过程。执行到 main() 时，程序为 texas 和 year 分配空间，使得这些变量可见。当程序调用 oil() 时，这些变量仍留在内存中，但不可见。为两个新变量（x 和 texas）分配内存，从而使它们可见。在程序执行到 oil() 中的内部代码块时，新的 texas 将不可见，它被一个更新的定义代替。不过，变量 x 仍然可见，这是因为该代码块没有定义 x 变量。当程序流程离开该代码块时，将释放最新的 texas 使用的内存，而第二个 texas 再次可见。当 oil() 函数结束时，texas 和 x 都将过期，而最初的 texas 和 year 再次变得可见。

可以使用 C++（和 C）关键字 auto 来显式地指出存储类别：

```
int froob(int n)
{
    auto float ford;
    ...
}
```

由于只能将关键字 auto 用于默认状态下为自动的变量，因此程序员几乎不使用它。有时候，它被用来向读者澄清代码的含义。例如，可以借助于表明，程序中特意创建了一个自动变量，它将覆盖全局定义（如本章后面的“静态持续性、外部连接性”一节将讨论的变量）。

1. 自动变量的初始化

可以使用任何在声明时其值为已知的表达式来初始化自动变量，下面的范例初始化变量 x、y 和 z：

```
int w;          // value of w is indeterminate
int x = 5;      // initialized with a constant expression
int y = 2 * x;  // use previously determined value of x
cin >> w;
int z = 3 * w;  // use new value of w
```

2. 自动变量和堆栈

了解典型的 C++ 编译器如何实现自动变量有助于更深入地了解自动变量。由于自动变量的数目随函数的开始和结束而增减，因此程序必须在运行时对自动变量进行管理。常用的方法是留出一段内存，并将其视为堆栈，以管理变量的增减。之所以被称为堆栈，是由于新数据被象征性地放在原有数据的上面（也就是说，在相邻的内存单元中，而不是在同一个内存单元中），当程序使用完后，将其从堆栈中删除。堆栈的默认长度取决于实现，但编译器通常提供改变堆栈长度的选项。程序使用两个指针来跟踪堆栈，一个指针指向栈底——堆栈的开始位置，另一个指针指向堆顶——下一个可用内存单元。当函数被调用时，其自动变量将被加入到堆栈中，栈顶指针指向变量后面的下一个可用的内存单元。函数结束时，栈顶指针被重置为函数被调用前的值，从而释放新变量使用的内存。

堆栈是 LIFO（后进先出）的，即最后加入到堆栈中的变量首先被弹出。这种设计简化了参数传递。函数调用将其参数的值放在栈顶，然后重新设置栈顶指针。被调用的函数根据其形参描述来确定每个参数的地址。例如，图 9.3 表明，函数 fib() 被调用时，传递一个 2 字节的 int 和一个 4 字节的 long。这些值被加入到堆栈中。当 fib() 开始执行时，它将名称 real 和 tell 同这两个值关联起来。当 fib() 结束时，栈顶指针重新指向以前的位置。新值没有被删除，但不再被标记，它们所占据的空间将被下一个将值加入到堆栈中的函数调用所使用（图 9.3 做了简化，因为函数调用可能传递其他信息，如返回地址）。

3. 寄存器变量

和 C 语言一样，C++ 也支持使用 register 关键字来声明局部变量。寄存器变量是另一种形式的自动变量，因此其存储持续性为自动，作用域为局部，但没有链接性。关键字 register 提醒编译器，用户希望它通过使用 CPU 寄存器，而不是堆栈来处理特定的变量，从而提供对变量的快速访问。这里的理念是，CPU 访问寄存器中的值的速度比访问堆栈中内存快。要声明寄存器变量，请在类型前加上关键字 register：

```
register int count_fast; // request for a register variable
```

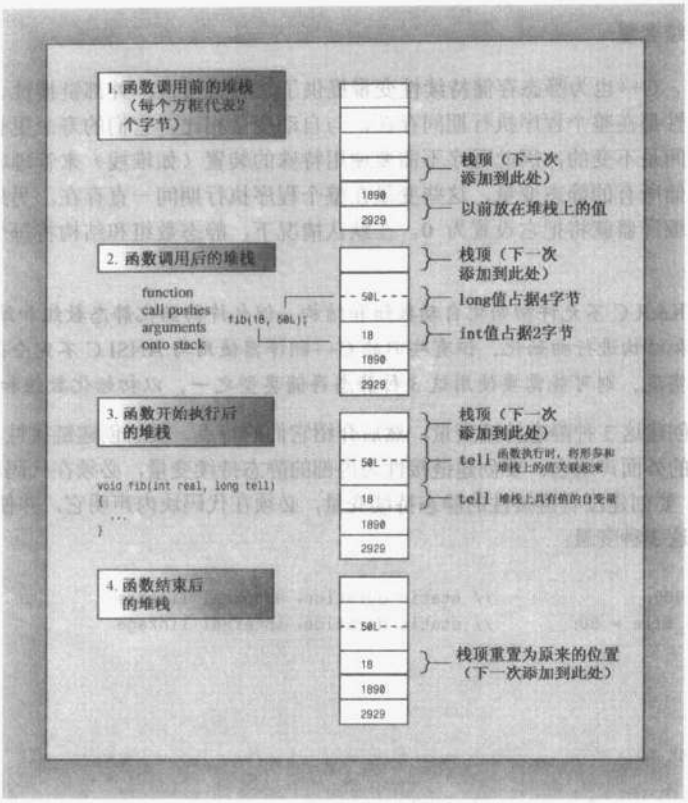


图 9.3　使用堆栈传递参数

读者可能注意到了，这里使用了“提醒”和“请求”。编译器并不一定会满足上述请求。例如，寄存器可能已经被占用，或者寄存器无法存储所请求的类型。许多程序员觉得，现代的编译器已经足够聪明，不需要提醒。例如，编写 for 循环时，编译器可能自动使用寄存器来存储循环计数。

如果变量被存储在寄存器中，则没有内存地址，因此不能将地址操作符用于寄存器变量。在下面的代码中，获取变量 x 的地址是可行的，但获取变量 y 的地址是错误的：

```
void gromb (int *);
int main()
{
   int x;
   register int y;
   gromb (&x);       // ok
   gromb (&y);       // not allowed
...
```

在声明中使用 register 后，这种限制便会生效，即使编译器实际上并没有使用寄存器来存储该变量。

简而言之，常规局部变量、使用 auto 声明的局部变量以及使用 register 声明的局部变量的存储持续性都是自动的，作用域都是局部的，也都没有链接性。下面的代码说明了这 3 种情况：

```
int main()
{
   short waffles;          // auto variable by default
   auto short pancakes;    // explicitly auto
   register int muffins;   // register variable
```

声明局部变量时，如果没有使用说明符，则与使用 auto 声明变量等效。通常，处理这种变量的方式是将其放置到内存堆栈中。使用 register 说明符指出该变量将被频繁使用，编译器可能会选择使用内存堆栈之外的其他方式（如使用 CPU 寄存器）来存储它。

9.2.3 静态持续变量

和C语言一样，C++也为静态存储持续性变量提供了3种链接性：外部链接性、内部链接性和无链接性。这3种链接性都在整个程序执行期间存在，与自动变量相比，它们的寿命更长。由于静态变量的数目在程序运行期间是不变的，因此程序不需要使用特殊的装置（如堆栈）来管理它们。编译器将分配固定的内存块来存储所有的静态变量，这些变量在整个程序执行期间一直存在。另外，如果没有显式地初始化静态变量，编译器就将把它设置为0。在默认情况下，静态数组和结构将每个元素或成员的所有位都设置为0。

注意：传统的K&R C不允许初始化自动数组和结构，但允许初始化静态数组和结构。ANSI C和C++允许对这两种数组和结构进行初始化，但有些旧的C++翻译器使用与ANSI C不完全兼容的C编译器。如果使用的是这样的实现，则可能需要使用这3种静态存储类型之一，以初始化数组和结构。

下面介绍如何创建这3种静态持续变量，然后介绍它们的特点。要想创建链接性为外部的静态持续变量，必须在代码块的外面声明它；要创建链接性为内部的静态持续变量，必须在代码块的外面声明它，并使用static限定符；要创建没有链接性的静态持续变量，必须在代码块内声明它，并使用static限定符。下面的代码片段说明这3种变量：

```
...
int global = 1000;              // static duration, external linkage
static int one_file = 50;       // static duration, internal linkage
int main()
{
...
}
void funct1 (int n)
{
   static int count = 0;        // static duration, no linkage
   int llama = 0;
...
}
void funct2 (int q)
{
...
}
```

正如前面指出的，所有静态持续变量（上述范例中的global、one_file和count）在整个程序执行期间都存在。在funct1()中声明的变量count的作用域为局部，没有链接性，这意味着只能在funct1()函数中使用它，就像自动变量llama一样。但是，与llama不同的是，即使在funct1()函数没有被执行时，count也留在内存中。global和one_file的作用域都为整个文件，即在从声明位置到文件结尾的范围内都可以被使用。具体地说，可以在main()、funct1()和funct2()中使用它们。由于one_file的链接性为内部，因此只能在包含上述代码的文件中使用它；由于global的链接性为外部，因此可以在程序的其他文件中使用它。

所有的静态持续变量都有下面的两个初始化特征：

- 未被初始化的静态变量的所有位都被设置为0。
- 只能使用常量表达式来初始化静态变量。

常量表达式可以使用字面值常量、const常量和enum常量以及sizeof操作符。下面的代码片段说明了这几点：

```
int x;                          // x set to 0
int y = 49;                     // 49 is a constant expression
int z = 2 * sizeof (int) + 1;   // also a constant expression
int m = 2 * z;                  // invalid, z not a constant
int main(){...}
```

表9.1总结了引入名称空间之前使用的存储特性。下面详细介绍各种静态持续变量。

表 9.1　5 种变量储存方式

存储描述	持续性	作用域	链接性	如何声明
自动	自动	代码块	无	在代码块中（可使用关键字 auto）
寄存器	自动	代码块	无	在代码块中，使用关键字 register
静态，无链接性	静态	代码块	无	在代码块中，使用关键字 static
静态，外部链接性	静态	文件	外部	在函数外面
静态，内部链接性	静态	文件	内部	在函数外面，使用关键字 static

1. 静态持续性、外部链接性

链接性为外部的变量通常简称为外部变量，它们的存储持续性为静态，作用域为整个文件。外部变量是在函数外部定义的，因此对所有函数而言都是外部的。例如，可以在 main() 函数的前面定义它们。可以在文件中位于外部变量定义后面的任何函数中使用它，因此外部变量也称全局变量（相对于局部的自动变量）。不过，如果定义了与外部变量同名的自动变量，则当程序执行其所属的函数时，该自动变量将隐藏同名的外部变量。程序清单 9.5 说明了这几点，它还演示了如何使用关键字 extern 来重新声明以前定义过的外部变量，以及如何使用 C++的作用域解析操作符来访问被隐藏的外部变量。由于该程序只包含一个文件，因此无法说明外部链接性；本章后面的范例将说明外部链接性。

程序清单 9.5　external.cpp

```
// external.cpp -- external variables
#include <iostream>
using namespace std;
// external variable
double warming = 0.3;

// function prototypes
void update (double dt);
void local();

int main()              // uses global variable
{
    cout << "Global warming is " << warming << " degrees.\n";
    update (0.1);           // call function to change warming
    cout << "Global warming is " << warming << " degrees.\n";
    local();              // call function with local warming
    cout << "Global warming is " << warming << " degrees.\n";
    return 0;
}

void update (double dt)    // modifies global variable
{
    extern double warming;  // optional redeclaration
    warming += dt;
    cout << "Updating global warming to " << warming;
    cout << " degrees.\n";
}

void local()            // uses local variable
{
    double warming = 0.8;    // new variable hides external one

    cout << "Local warming = " << warming << " degrees.\n";
        // Access global variable with the
        // scope resolution operator
```

```
    cout << "But global warming = " <<::warming;
    cout << " degrees.\n";
}
```

下面是该程序的输出：

```
Global warming is 0.3 degrees.
Updating global warming to 0.4 degrees.
Global warming is 0.4 degrees.
Local warming = 0.8 degrees.
But global warming = 0.4 degrees.
Global warming is 0.4 degrees.
```

2. 程序说明

程序清单 9.5 中程序的输出表明，main() 和 update() 都可以访问外部变量 warming。注意，update() 修改了 warming，这种变化在接下来使用该变量时显现出来。

update() 函数使用关键字 extern 对 warming 变量进行重新声明。该关键字的意思是“通过以前被外部定义的名称使用该变量”。由于即使省略该声明，update() 的功能也相同，因此该声明是可选的。它指出该函数被设计成使用外部变量。原来的声明：

```
double warming = 0.3;
```

称为定义声明（defining declaration）或简称为定义（definition）。它给该变量分配存储空间。重新声明：

```
extern double warming;
```

称为引用声明（referencing declaration），或简称为声明（declaration）。它不给变量分配存储空间，因为它引用已有的变量。只能在引用其他地方（或函数）定义的变量的声明中使用关键字 extern。基本上，该声明指出，使用外部定义的变量 warming。在引用声明中指定的类型应与定义声明中相同；另外，不能在引用声明中初始化变量：

```
extern double warming = 0.5;   // INVALID
```

仅当声明将为变量分配存储空间时（即定义声明），才能在声明中初始化变量。毕竟，初始化指的是在分配内存单元时给它赋值。

local() 函数表明，定义与全局变量同名的局部变量后，局部变量将隐藏全局变量。例如，local() 函数显示 warming 的值时，将使用 warming 的局部定义。

C++比 C 语言进了一步——它提供了作用域解析操作符（::）。当放在变量名称前面时，该操作符表示使用变量的全局版本。因此，local() 将 warming 显示为 0.8，但将::warming 显示为 0.4。后面介绍名称空间和类时，将再次介绍该操作符。

全局变量和局部变量

既然可以选择使用全局变量或局部变量，那么到底应使用哪种呢？首先，全局变量很有吸引力——因为所有的函数能访问全局变量，因此不用传递参数，但易于访问的代价很大——程序不可靠。计算经验表明，程序越能避免对数据进行不必要的访问，就越能保持数据的完整性。通常情况下，应使用局部变量，应在需要知晓时才传递数据，而不应不加区分地使用全局变量来使数据可用。读者将会看到，OOP 在数据隔离方面又向前迈进了一步。

不过，全局变量也有它们的用处。例如，可以让多个函数使用同一个数据块（如月份名数组或原子量数组）。外部存储尤其适于表示常量数据，因为这样可以使用关键字 const 来防止数据被修改。

```
const char * const months[12] =
{
      "January", "February", "March", "April", "May",
      "June", "July", "August", "September", "October",
      "November", "December"
};
```

在上述范例中，第一个 const 防止字符串被修改，第二个 const 确保数组中每个指针始终指向它最初指向的字符串

3. 静态持续性、内部链接性

将 static 限定符用于作用域为整个文件的变量时，该变量的链接性将为内部的。在多文件程序中，内部链接性和外部链接性之间的差别很有意义。链接性为内部的变量只能在其所属的文件中使用；但常规外部变量都具有外部链接性，即可以在其他文件中使用。对于外部链接性变量，有且只有一个文件中包含了该变量的外部定义。其他文件要使用该变量，必须在引用声明中使用关键字 extern（参见图 9.4）。

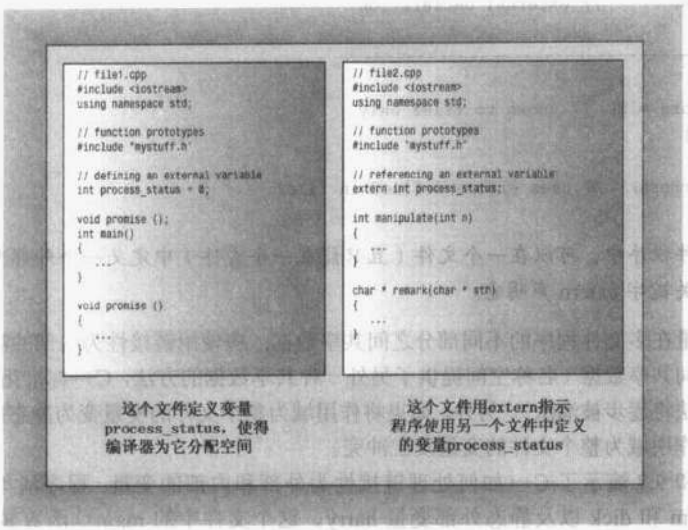


图 9.4　定义声明和引用声明

如果文件没有提供变量的 extern 声明，则不能使用在其他文件中定义的外部变量：

```
// file1
int errors = 20;      // global declaration
...
-----------------------------------------------
// file 2
...                // missing an extern int errors declaration
void froobish()
{
    cout << errors;  // doomed attempt to use errors
...
```

如果文件试图定义另一个同名的外部变量将出错：

```
// file1
int errors = 20;         // external declaration
...
-----------------------------------------------
// file 2
int errors;            // invalid declaration
void froobish()
{
    cout << errors;     // doomed attempt to use errors
..
```

正确的方法是使用关键字 extern：

```
 // file1
int errors = 20;         // external declaration
...
-----------------------------------------------
```

```
// file 2
extern int errors;      // refers to errors from file1
void froobish()
{
    cout << errors;    // uses errors defined in file1
```

但如果文件定义了一个静态外部变量，其名称与另一个文件中声明的常规外部变量相同，则在该文件中，静态变量将隐藏常规外部变量：

```
// file1
int errors = 20;        // external declaration
...
-----------------------------------------
// file2
static int errors = 5;  // known to file2 only
void froobish()
{
    cout << errors;   // uses errors defined in file2
    ...
```

记住：在多文件程序中，可以在一个文件（且只能在一个文件）中定义一个外部变量。使用该变量的其他文件必须使用关键字 extern 声明它。

应使用外部变量在多文件程序的不同部分之间共享数据；应使用链接性为内部的静态变量在同一个文件中的多个函数之间共享数据（名称空间提供了另外一种共享数据的方法，C++标准指出，使用 static 来创建内部链接性的方法将逐步被淘汰）。另外，如果将作用域为整个文件的变量变为静态的，就不必担心其名称与其他文件中的作用域为整个文件的变量发生冲突。

程序清单 9.6 和 9.7 演示了 C++如何处理链接性为外部和内部的变量。程序清单 9.6（twofile1.cpp）定义了外部变量 tom 和 dick 以及静态外部变量 harry。这个文件中的 main()函数显示这 3 个变量的地址，然后调用 remote_access()函数，该函数是在另一个文件中定义的。程序清单 9.7（twofile2.cpp）列出了该文件。除了定义 remote_access()之外，该文件还使用 extern 关键字来与第一个文件共享 tom。接下来，该文件定义一个名为 dick 的静态变量。static 限定符使该变量被限制在这个文件内，并覆盖相应的全局定义。然后，该文件定义了一个名为 harry 的外部变量，这不会与第一个文件中的 harry 发生冲突，因为后者的链接性为内部的。随后，remote-access()函数显示这 3 个变量的地址，以便于将它们与第一个文件中相应变量的地址进行比较。别忘了编译这两个文件，并将它们链接起来，以得到完整的程序。

程序清单 9.6 twofile1.cpp

```
// twofile1.cpp -- variables with external and internal linkage
#include <iostream>          // to be compiled with two file2.cpp
int tom = 3;                 // external variable definition
int dick = 30;               // external variable definition
static int harry = 300;      // static, internal linkage

// function prototype
void remote_access();

int main()
{
   using namespace std;
   cout << "main()reports the following addresses: \n";
   cout << &tom << " = &tom, " << &dick << " = &dick, ";
   cout << &harry << " = &harry\n";
   remote_access();
   return 0;
}
```

程序清单 9.7　twofile2.cpp

```
// twofile2.cpp -- variables with internal and external linkage
#include <iostream>
extern int tom;             // tom defined elsewhere
static int dick = 10;       // overrides external dick
int harry = 200;            // external variable definition,
                            // no conflict with twofile1 harry

void remote_access()
{
    using namespace std;
    cout << "remote_access()reports the following addresses: \n";
    cout << &tom << " = &tom, " << &dick << " = &dick, ";
    cout << &harry << " = &harry\n";
}
```

下面是编译程序清单 9.6 和 9.7 生成的程序的输出：

```
main()reports the following addresses:
0x0041a020 = &tom, 0x0041a024 = &dick, 0x0041a028 = &harry
remote_access()reports the following addresses:
0x0041a020 = &tom, 0x0041a450 = &dick, 0x0041a454 = &harry
```

从上述地址可知，这两个文件使用了同一个 tom 变量，但使用了不同的 dick 和 harry 变量。具体的地址和格式可能随系统而异，但两个 tom 变量的地址将相同，而两个 dick 和 harry 变量的地址不同。

4. 静态存储持续性、无链接性

至此，我们介绍了链接性分别为内部和外部、作用域为整个文件的变量。接下来介绍静态持续家族中的第三个成员——无链接性的局部变量。这种变量是这样创建的，将 static 限定符用于在代码块中定义的变量。在代码块中使用 static 时，将导致局部变量的存储持续性为静态的。这意味着虽然该变量只在该代码块中可用，但它在该代码块不处于活动状态时仍然存在。因此在两次函数调用之间，静态局部变量的值将保持不变(静态变量适用于再生——可以用它们将瑞士银行的秘密账号传递到下一个要去的地方)。另外，如果初始化了静态局部变量，则程序只在启动时进行一次初始化。以后再调用函数时，将不会像自动变量那样再次被初始化。程序清单 9.8 说明了这几点。

该程序演示了一种处理行输入可能长于目标数组的方法。前面讲过，cin.get（input，ArSize）方法将一直读取输入，直到到达行尾或读取了 ArSize-1 个字符为止。它把换行符留在输入队列中。该程序读取行输入后的字符。如果是换行符，则说明整行被读取；否则说明行中还有字符没有被读取。随后，程序使用一个循环来丢弃余下的字符，不过读者可以修改代码，让下一轮输入读取行中余下的字符。该程序还利用了这样一个事实，即试图使用 get（char *，int）读取空行将导致 cin 为 false。

程序清单 9.8　static.cpp

```
// static.cpp -- using a static local variable
#include <iostream>
// constants
const int ArSize = 10;

// function prototype
void strcount (const char * str);

int main()
{
    using namespace std;
    char input[ArSize];
    char next;

    cout << "Enter a line: \n";
    cin.get (input, ArSize);
```

```
    while (cin)
    {
        cin.get (next);
        while (next != '\n')  // string didn't fit!
          cin.get (next);
        strcount (input);
        cout << "Enter next line (empty line to quit): \n";
        cin.get (input, ArSize);
    }
    cout << "Bye\n";
    return 0;
}

void strcount (const char * str)
{
    using namespace std;
    static int total = 0;          // static local variable
    int count = 0;                 // automatic local variable

    cout << "\"" << str <<"\" contains ";
    while (*str++)                 // go to end of string
        count++;
    total += count;
    cout << count << " characters\n";
    cout << total << " characters total\n";
}
```

注意：早期的一些编译器没有实现这样的功能，即当 cin.get（char *，int）读取一个空行后，将设置错误标记 failbit，从而导致 cin 测试为 false。在这种情况下，可以将测试：

```
while (cin)
```

替换为：

```
while (input[0])
```

或者提供一个可同时用于新旧实现的测试：

```
while (cin && input[0])
```

下面是该程序的输出：

```
Enter a line:
nice pants
"nice pant" contains 9 characters
9 characters total
Enter next line (empty line to quit):
thanks
"thanks" contains 6 characters
15 characters total
Enter next line (empty line to quit):
parting is such sweet sorrow
"parting i" contains 9 characters
24 characters total
Enter next line (empty line to quit):
ok
"ok" contains 2 characters
26 characters total
Enter next line (empty line to quit):

Bye
```

注意，由于数组长度为 10，因此程序从每行读取的字符数都不超过 9 个。另外还需要注意的是，每次函数被调用时，自动变量 count 都被重置为 0。不过，静态变量 total 只在程序运行时被设置为 0，以后在两次函数调用之间，其值将保持不变，因此能够记录读取的字符总数。

9.2.4　说明符和限定符

一些被称为存储说明符（storage class specifier）或 cv-限定符（cv-qualifier）的 C++关键字提供了其他有关存储的信息。下面是存储说明符：

- auto。
- register。
- static。
- extern。
- mutable。

其中的大部分已经介绍过了，在同一个声明中不能使用多个说明符。前面讲过，可以在声明中使用关键字 auto 将变量声明为自动变量；关键字 register 用于在声明中指示寄存器存储类型。关键字 static 被用在作用域为整个文件的声明中时，表示内部链接性；被用于局部声明中，表示局部变量的存储持续性为静态的。关键字 extern 表明是引用声明，即声明引用在其他地方定义的变量。关键字 mutable 的含义将根据 const 来解释，因此先来介绍 cv-限定符，然后再解释它。

下面就是 cv 限定符：

- const。
- volatile。

（读者可能猜到了，cv 表示 const 和 volatile）其中最常用的是 const，而读者已经知道其用途。它表明，内存被初始化后，程序便不能再对它进行修改。我们稍后再回过头来介绍它。

volatile 关键字表明，即使程序代码没有对内存单元进行修改，其值也可能发生变化。听起来似乎很神秘，实际上并非如此。例如，可以将一个指针指向某个硬件位置，其中包含了来自串行端口的时间或信息。在这种情况下，硬件（而不是程序）可能修改其中的内容。或者两个程序可能互相影响，共享数据。该关键字的作用是为了改善编译器的优化能力。例如，假设编译器发现，程序在几条语句中两次使用了某个变量的值，则编译器可能不是让程序查找这个值两次，而是将这个值缓存到寄存器中。这种优化假设变量的值在这两次使用之间不会变化。如果不将变量声明为 volatile，则编译器将进行这种优化；将变量声明为 volatile，相当于告诉编译器，不要进行这种优化。

现在回到 mutable。可以用它来指出，即使结构（或类）变量为 const，其某个成员也可以被修改。例如，请看下面的代码：

```
struct data
{
   char name[30];
   mutable int accesses;
   ...
};
const data veep = {"Claybourne Clodde", 0, ... };
strcpy (veep.name, "Joye Joux"); // not allowed
veep.accesses++;                 // allowed
```

veep 的 const 限定符禁止程序修改 veep 的成员，但 access 成员的 mutable 说明符使得 access 不受这种限制。

本书不使用 volatile 或 mutable，但将进一步介绍 const。

再谈 const

在 C++（但不是在 C 语言）中，const 限定符对默认存储类型稍有影响。在默认情况下全局变量的链接性为外部的，但 const 全局变量的链接性为内部的。也就是说，在 C++看来，全局 const 定义（如下述代码段所示）就像使用了 static 说明符一样。

```
const int fingers = 10;    // same as static const int fingers;
int main (void)
{
```

...

C++修改了常量类型的规则，让程序员更轻松。例如，假设将一组常量放在头文件中，并在同一个程序的多个文件中使用该头文件。那么，预处理器将头文件的内容包含到每个源文件中后，所有的源文件都将包含类似下面这样的定义：

```
const int fingers = 10;
const char * warning = "Wak! ";
```

如果全局 const 声明的链接性像常规变量那样是外部的，则这将出错，因为不能在多个文件中定义同一个全局变量。也就是说，只能有一个文件可以包含前面的声明，而其他文件必须使用 extern 关键字来提供引用声明。另外，只有未使用 extern 关键字的声明才能进行初始化：

```
// extern would be required if const had external linkage
extern const int fingers;       // can't be initialized
extern const char * warning;
```

因此，需要为某个文件使用一组定义，而其他文件使用另一组声明。然而，由于外部定义的 const 数据的链接性为内部的，因此可以在所有文件中使用相同的声明。

内部链接性还意味着，每个文件都有自己的一组常量，而不是所有文件共享一组常量。每个定义都是其所属文件私有的，这就是能够将常量定义放在头文件中的原因。这样，只要在两个源代码文件中包括同一个头文件，则它们将获得同一组常量。

如果出于某种原因，程序员希望某个常量的链接性为外部的，则可以使用 extern 关键字来覆盖默认的内部链接性：

```
extern const int states = 50;   // external linkage
```

在这种情况下，必须在所有使用该常量的文件中使用 extern 关键字来声明它。这与常规外部变量不同，定义常规外部变量时，不必使用 extern 关键字，但在使用该变量的其他文件中必须使用 extern。另外，与常规变量不同的是，可以初始化 extern const 变量。实际上，必须这样做，因为 const 数据必须初始化。

在函数或代码块中声明 const 时，其作用域为代码块，即仅当程序执行该代码块中的代码时，该常量才是可用的。这意味着在函数或代码块中创建常量时，不必担心其名称与其他地方定义的常量发生冲突。

9.2.5 函数和链接性

和变量一样，函数也有链接性，虽然可选择的范围比变量小。和 C 语言一样，C++不允许在一个函数中定义另外一个函数，因此所有函数的存储持续性都自动为静态的，即在整个程序执行期间都一直存在。在默认情况下，函数的链接性为外部的，即可以在文件间共享。实际上，可以在函数原型中使用关键字 extern 来指出函数是在另一个文件中定义的，不过这是可选的（要让程序在另一个文件中查找函数，该文件必须作为程序的组成部分被编译，或者是由链接程序搜索的库文件）。还可以使用关键字 static 将函数的链接性设置为内部的，使之只能在一个文件中使用。必须同时在原型和函数定义中使用该关键字：

```
static int private(double x);
...
static int private(double x)
{
   ...
}
```

这意味着该函数只在这个文件中可见，还意味着可以在其他文件中定义同名的的函数。和变量一样，在定义静态函数的文件中，静态函数将覆盖外部定义，因此即使在外部定义了同名的函数，该文件仍将使用静态函数。

C++有一个“单定义规则”，即对于每个非内联函数，程序中只能包含一个定义。对于链接性为外部的函数来说，这意味着在多文件程序中，只能有一个文件包含该函数的定义，但使用该函数的每个文件都应包含其函数原型。

内联函数不受这项规则的约束，这允许程序员能够将内联函数的定义放在头文件中。这样，包含了头文件的每个文件都有内联函数的定义。不过，C++要求同一个函数的所有内联定义都必须相同。

C++在哪里查找函数

假设在程序的某个文件中调用一个函数，C++将到哪里去寻找该函数的定义呢？如果该文件中的函数原型指出该函数是静态的，则编译器将只在该文件中查找函数定义；否则，编译器（包括链接程序）将在所有的程序文件中查找。如果找到两个定义，编译器将发出错误消息，因为每个外部函数只能有一个定义。如果在程序文件中没有找到，编译器将在库中搜索。这意味着如果定义了一个与库函数同名的函数，编译器将使用程序员定义的版本，而不是库函数（不过，C++保留了标准库函数的名称，即程序员不应使用它们）。一些编译器-链接程序要求显式地指出要搜索哪些库。

9.2.6　语言链接性

另一种形式的链接性——称为语言链接性（language linking）也对函数有影响。首先介绍一些背景知识。链接程序要求每个不同的函数都有不同的符号名。在 C 语言中，一个名称只对应一个函数，因此这很容易实现。因此，为满足内部需要，C 语言编译器可能将 spiff 这样的函数名翻译为_spiff。这种方法被称为 C 语言链接性（C language linkage）。但在 C++中，同一个名称可能对应多个函数，必须将这些函数翻译为不同的符号名称。因此，C++编译器执行名称矫正或名称修饰（参见第 8 章），为重载函数生成不同的符号名称。例如，可能将 spiff（int）转换为_spoff_i，而将 spiff（double，double）转换为_spiff_d_d。这种方法被称为 C++语言链接（C++ language linkage）。

链接程序寻找与 C++函数调用匹配的函数时，使用的方法与 C 语言不同。但如果要在 C++程序中使用 C 库中预编译的函数，将出现什么情况呢？例如，假设有下面的代码：

```
spiff(22); // want spiff(int) from a C library
```

它在 C 库文件中的符号名称为_spiff，但对于我们假设的链接程序来说，C++查询约定是查找符号名称_spiff_i。为解决这种问题，可以用函数原型来指出要使用的约定：

```
extern "C" void spiff(int);     // use C protocol for name look-up
extern void spoff(int);         // use C++ protocol for name look-up
extern "C++" void spaff(int);   // use C++ protocol for name look-up
```

第一个原型使用 C 语言链接性；而后面的两个使用 C++语言链接性。第二个原型是通过默认方式指出这一点的，而第三个显式地指出了这一点。

9.2.7　存储方案和动态分配

前面介绍 C++用来为变量（包括数组和结构）分配内存的 5 种方案，它们不适用于使用 C++操作符 new（或 C 函数 malloc()）分配的内存，这种内存被称为动态内存。第 4 章介绍过，动态内存由操作符 new 和 delete 控制，而不是由作用域和链接性规则控制。因此，可以在一个函数中分配动态内存，而在另一个函数中将其释放。与自动内存不同，动态内存不是 LIFO，其分配和释放顺序要取决于 new 和 delete 在何时以何种方式被使用。通常，编译器使用 3 块独立的内存：一块用于静态变量（可能再细分），一块用于自动变量，另外一块用于动态存储。

虽然存储方案概念不适用于动态内存，但适用于用来跟踪动态内存的自动和静态指针变量。例如，假设在一个函数中包含下面的语句：

```
float * p_fees = new float [20];
```

由 new 分配的 80 个字节（假设 float 为 4 个字节）的内存将一直保留在内存中，直到使用 delete 操作符将其释放。但当包含该声明的函数结束时，p_fees 指针将消失。如果希望另一个函数能够使用这 80 个字节中的内容，则必须将其地址传递或返回给该函数。另一方面，如果将 p_fees 的链接性声明为外部的，则文件中位于该声明后面的所有函数都可以使用它。另外，通过在另一个文件中使用：

```
extern float * p_fees;
```

可以在这个文件中使用该指针。不过，请注意，使用 new 来设置 p_fees 的语句必须位于函数中（如下面的代码段所示），这是因为只能使用常量表达式来初始化静态存储变量：

```
float * p_fees;    // ok to create a static storage pointer
// float * p2 = new float[20]; // initialization with non-const not allowed here
```

```
int main()
{
   p_fees = new float [20];
...
```

注意：在程序结束时，由 new 分配的内存通常都将被释放，不过情况也并不总是这样。例如，在不那么健壮的操作系统中，在某些情况下，请求大型内存块将导致该代码块在程序结束不会被自动释放。最佳的做法是，使用 delete 来释放 new 分配的内存。

9.3 布局 new 操作符

通常，new 负责在堆（heap）中找到一个足以能够满足要求的内存块。new 操作符还有另一种变体，被称为布局（placement）new 操作符，它让您能够指定要使用的位置。程序员可能使用这种特性来设置其内存管理规程或处理需要通过特定地址进行访问的硬件。

要使用布局 new 特性，首先需要包含头文件 new，它提供了这种版本的 new 操作符的原型；然后将 new 操作符用于提供了所需地址的参数。除需要指定参数外，句法与常规 new 操作符相同。具体地说，使用布局 new 操作符时，变量后面可以有方括号，也可以没有。下面的代码段演示了 new 操作符的 4 种用法：

```
#include <new>
struct chaff
{
   char dross[20];
   int slag;
};
char buffer1[50];
char buffer2[500];
int main()
{
   chaff *p1, *p2;
   int *p3, *p4;
// first, the regular forms of new
   p1 = new chaff;               // place structure in heap
   p3 = new int[20];             // place int array in heap
// now, the two forms of placement new
   p2 = new (buffer1) chaff;    // place structure in buffer1
   p4 = new (buffer2) int[20];  // place int array in buffer2
...
```

出于简化的目的，这个范例使用两个静态数组来为布局 new 操作符提供内存空间。因此，上述代码从 buffer1 中分配空间给结构 chaff，从 buffer2 中分配空间给一个包含 20 个元素的 int 数组。

读者了解布局 new 操作符后，来看一个范例程序。程序清单 9.9 使用常规 new 操作符和布局 new 操作符创建动态分配的数组。该程序说明了常规 new 操作符和布局 new 操作符之间的一些重要差别，在查看该程序的输出后，我们将对此进行讨论。

程序清单 9.9 newplace.cpp

```
// newplace.cpp -- using placement new
#include <iostream>
#include <new>          // for placement new
const int BUF = 512;
const int N = 5;
char buffer[BUF];       // chunk of memory
int main()
{
   using namespace std;
```

```
    double *pd1, *pd2;
    int i;
    cout << "Calling new and placement new:\n";
    pd1 = new double[N];          // use heap
    pd2 = new (buffer) double[N];  // use buffer array
    for (i = 0; i < N; i++)
        pd2[i] = pd1[i] = 1000 + 20.0 * i;
    cout << "Buffer addresses:\n" << "  heap: " << pd1
        << "  static: " << (void *) buffer  <<endl;
    cout << "Buffer contents:\n";
    for (i = 0; i < N; i++)
    {
        cout << pd1[i] << " at " << &pd1[i] << "; ";
        cout << pd2[i] << " at " << &pd2[i] << endl;
    }

    cout << "\nCalling new and placement new a second time:\n";
    double *pd3, *pd4;
    pd3= new double[N];
    pd4 = new (buffer) double[N];
    for (i = 0; i < N; i++)
        pd4[i] = pd3[i] = 1000 + 20.0 * i;
    cout << "Buffer contents:\n";
    for (i = 0; i < N; i++)
    {
        cout << pd3[i] << " at " << &pd3[i] << "; ";
        cout << pd4[i] << " at " << &pd4[i] << endl;
    }

    cout << "\nCalling new and placement new a third time:\n";
    delete [] pd1;
    pd1= new double[N];
    pd2 = new (buffer + N * sizeof(double)) double[N];
    for (i = 0; i < N; i++)
        pd2[i] = pd1[i] = 1000 + 20.0 * i;
    cout << "Buffer contents:\n";
    for (i = 0; i < N; i++)
    {
        cout << pd1[i] << " at " << &pd1[i] << "; ";
        cout << pd2[i] << " at " << &pd2[i] << endl;
    }
    delete [] pd1;
    delete [] pd3;

    return 0;
}
```

下面是该程序在某个系统上运行时的输出：

```
Calling new and placement new:
Buffer addresses:
   heap: 0xc9d34  static: 0x42e10
Buffer contents:
1000 at 0xc9d34; 1000 at 0x42e10
1020 at 0xc9d3c; 1020 at 0x42e18
1040 at 0xc9d44; 1040 at 0x42e20
1060 at 0xc9d4c; 1060 at 0x42e28
1080 at 0xc9d54; 1080 at 0x42e30

Calling new and placement new a second time:
Buffer contents:
1000 at 0xc9d64; 1000 at 0x42e10
```

```
1020 at 0xc9d6c; 1020 at 0x42e18
1040 at 0xc9d74; 1040 at 0x42e20
1060 at 0xc9d7c; 1060 at 0x42e28
1080 at 0xc9d84; 1080 at 0x42e30

Calling new and placement new a third time:
Buffer contents:
1000 at 0xc9d34; 1000 at 0x42e38
1020 at 0xc9d3c; 1020 at 0x42e40
1040 at 0xc9d44; 1040 at 0x42e48
1060 at 0xc9d4c; 1060 at 0x42e50
1080 at 0xc9d54; 1080 at 0x42e58
```

程序说明

有关程序清单 9.9，首先要指出的一点是，布局 new 操作符确实将数组 p2 放在了数组 buffer 中，p2 和 buffer 的地址都是 0x42e10。同时，常规 new 操作将数组 p1 放在很远的地方，其地址为 0xc9d34，位于动态管理的堆中。

需要指出的第二点是，第二个常规 new 操作符查找一个新的内存块，其起始地址为 0xc9d64；但第二个布局 new 操作符分配与以前相同的内存块：起始地址为 0x42e10 的内存块。布局 new 操作符使用传递给它的地址，它不跟踪哪些内存单元已被使用，也不查找未使用的内存块。这将一些内存管理的负担交给了程序员。例如，在第三次调用布局 new 操作符时，提供了一个从数组 buffer 开头算起的偏移量，因此将分配新的内存：

```
pd2 = new (buffer + N * sizeof(double)) double[N]; // offset of 40 bytes
```

第三点差别是，是否使用 delete 来释放内存。对于常规 new 操作符，下面的语句：

```
delete [] pd1;
```

释放起始地址为 0xc9d34 的内存块，因此接下来再次调用 new 操作符时，该内存块是可用的。然而，程序清单 9.9 中的程序没有使用 delete 来释放使用布局 new 操作符分配的内存。事实上，在这个例子中不能这样做。buffer 指定的内存是静态内存，而 delete 只能用于这样的指针：指向常规 new 操作符分配的堆内存。也就是说，数组 buffer 位于 delete 的管辖区域之外，下面的语句：

```
delete [] pd2;
```

将引发运行阶段错误。另一方面，如果 buffer 是使用常规 new 操作符创建的，便可以使用常规 delete 操作符来释放整个内存块。

将布局 new 操作符用于类对象时，情况将更复杂，这将在第 12 章介绍。

9.4 名称空间

在 C++中，名称可以是变量、函数、结构、枚举、类以及类和结构的成员。当随着工程的增大，名称相互冲突的可能性也将增加。使用多个厂商的类库时，可能导致名称冲突。例如，两个库可能都定义了名为 List、Tree 和 Node 的类，但定义的方式不兼容。用户可能希望使用一个库的 List 类，而使用另一个库的 Tree 类。这种冲突被称为名称空间问题。

C++标准提供了名称空间工具，以便更好地控制名称的作用域。经过了一段时间后，编译器才支持名称空间，但现在这种支持很普遍。

9.4.1 传统的 C++名称空间

介绍 C++中新增的名称空间特性之前，先复习一下 C++中已有的名称空间属性，并介绍一些术语，让读者熟悉名称空间的概念。

第一个需要知道的术语是声明区域（declaration region）。声明区域是可以在其中进行声明的区域。例如，可以在函数外面声明全局变量，对于这种变量，其声明区域为其声明所在的文件。对于在函数中声明的变量，其声明区域为其声明所在的代码块。

第二个需要知道的术语是潜在作用域（potential scope）。变量的潜在作用域从声明点开始，到其声明区域的结尾。因此潜在作用域比声明区域小，这是由于变量必须定义后才能使用。

然而，变量并非在其潜在作用域内的任何位置都是可见的。例如，它可能被另一个在嵌套声明区域中声明的同名变量隐藏。例如，在函数中声明的局部变量（对于这种变量，声明区域为整个函数）将隐藏在同一个文件中声明的全局变量（对于这种变量，声明区域为整个文件）。变量对程序而言可见的范围被称为作用域（scope），前面正是以这种方式使用该术语的。图 9.5 和图 9.6 对术语声明区域、潜在作用域和作用域进行了说明。

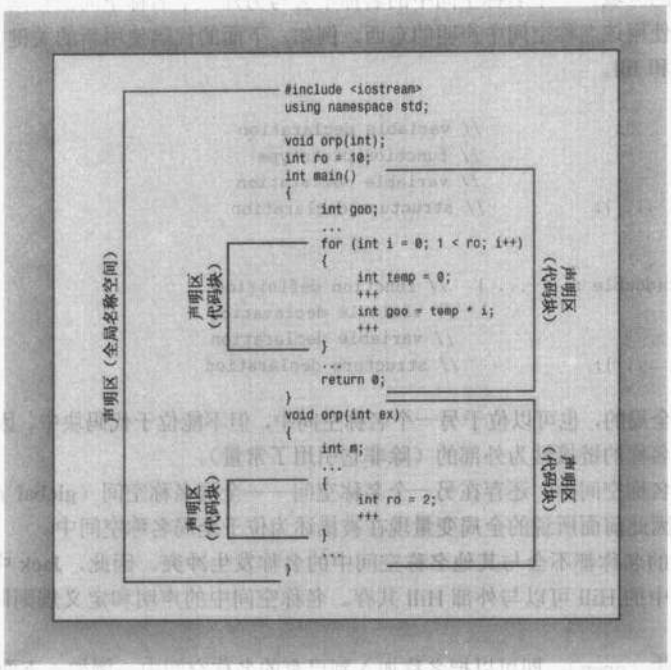


图 9.5　声明区域

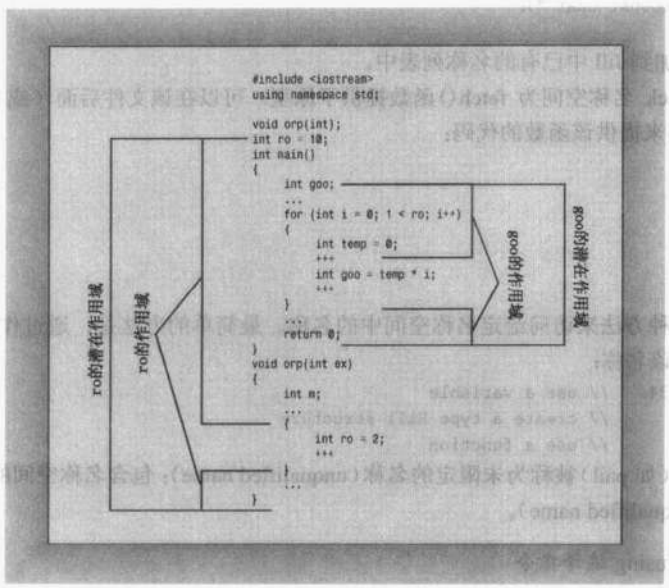


图 9.6　潜在作用域和作用域

C++关于全局变量和局部变量的规则定义了一种名称空间层次。每个声明区域都可以声明名称，这些名称独立于在其他声明区域中声明的名称。在一个函数中声明的局部变量不会与在另一个函数中声明的局部变量发生冲突。

9.4.2 新的名称空间特性

C++新增了这样一种功能，即通过定义一种新的声明区域来创建命名的名称空间，这样做的目的之一是提供一个声明名称的区域。一个名称空间中的名称不会与另外一个名称空间的相同名称发生冲突，同时允许程序的其他部分使用该名称空间中声明的东西。例如，下面的代码使用新的关键字 namespace 创建了两个名称空间：Jack 和 Jill。

```
namespace Jack {
    double pail;                    // variable declaration
    void fetch();                   // function prototype
    int pal;                        // variable declaration
    struct Well { ... };            // structure declaration
}
namespace Jill {
    double bucket(double n) { ... }  // function definition
    double fetch;                    // variable declaration
    int pal;                         // variable declaration
    struct Hill { ... };             // structure declaration
}
```

名称空间可以是全局的，也可以位于另一个名称空间中，但不能位于代码块中。因此，在默认情况下，在名称空间中声明的名称的链接性为外部的（除非它引用了常量）。

除了用户定义的名称空间外，还存在另一个名称空间——全局名称空间（global namespace）。它对应于文件级声明区域，因此前面所说的全局变量现在被描述为位于全局名称空间中。

任何名称空间中的名称都不会与其他名称空间中的名称发生冲突。因此，Jack 中的 fetch 可以与 Jill 中的 fetch 共存，Jill 中的 Hill 可以与外部 Hill 共存。名称空间中的声明和定义规则同全局声明和定义规则相同。

名称空间是开放的（open），即可以把名称加入到已有的名称空间中。例如，下面这条语句：

```
namespace Jill {
    char * goose (const char *);
}
```

将名称 goose 添加到 Jill 中已有的名称列表中。

同样，原来的 Jack 名称空间为 fetch() 函数提供了原型。可以在该文件后面（或另外一个文件中）再次使用 Jack 名称空间来提供该函数的代码：

```
namespace Jack {
    void fetch()
    {
        ...
    }
}
```

当然，需要有一种方法来访问给定名称空间中的名称。最简单的方法是，通过作用域解析操作符::，使用名称空间来限定该名称：

```
Jack::pail = 12.34;    // use a variable
Jill::Hill mole;       // create a type Hill structure
Jack::fetch();         // use a function
```

未被装饰的名称（如 pail）被称为未限定的名称（unqualified name）；包含名称空间的名称（如 Jack::pail）被称为限定的名称（qualified name）。

1. using 声明和 using 编译指令

我们并不希望每次使用名称时都对它进行限定，因此 C++提供了两种机制（using 声明和 using 编译

指令）来简化对名称空间中名称的使用。using 声明使特定的标识符可用，using 编译指令使整个名称空间可用。

using 声明由被限定的名称和它前面的关键字 using 组成：

```
using Jill::fetch;    // a using declaration
```

using 声明将特定的名称添加到它所属的声明区域中。例如 main() 中的 using 声明 Jill::fetch 将 fetch 添加到 main() 定义的声明区域中。完成该声明后，便可以使用名称 fetch 代替 Jill::fetch。下面的代码段说明了这几点：

```
namespace Jill {
   double bucket (double n) { ... }
   double fetch;
   struct Hill { ... };
}
char fetch;
int main()
{
   using Jill::fetch;      // put fetch into local namespace
   double fetch;           // Error! Already have a local fetch
   cin >> fetch;           // read a value into Jill::fetch
   cin >>::fetch;          // read a value into global fetch
   ...
}
```

由于 using 声明将名称添加到局部声明区域中，因此这个范例避免了将另一个局部变量也命名为 fetch。另外，和其他局部变量一样，fetch 也将覆盖同名的全局变量。

在函数的外面使用 using 声明时，将把名称添加到全局名称空间中：

```
void other();
namespace Jill {
   double bucket (double n) { ... }
   double fetch;
   struct Hill { ... };
}
using Jill::fetch;   // put fetch into global namespace
int main()
{
   cin >> fetch;       // read a value into Jill::fetch
   other()
...
}

void other()
{
   cout << fetch;  // display Jill::fetch
...
}
```

using 声明使一个名称可用，而 using 编译指令使所有的名称都可用。using 编译指令由名称空间名和它前面的关键字 using namespace 组成，它使名称空间中的所有名称都可用，而不需要使用作用域解析操作符：

```
using namespace Jack;  // make all the names in Jack available
```

在全局声明区域中使用 using 编译指令，将使该名称空间的名称全局可用。这种情况已出现过多次：

```
#include <iostream>    // places names in namespace std
using namespace std;   // make names available globally
```

在函数中使用 using 编译指令，将使其中的名称在该函数中可用，下面是一个例子：

```
int main()
{
   using namespace jack; // make names available in vorn()
...
}
```

在本书中，经常将这种格式用于名称空间 std。

有关 using 编译指令和 using 声明，需要记住的一点是，它们增加了名称冲突的可能性。也就是说，如果有名称空间 jack 和 jill，并在代码中使用作用域解析操作符，则不会存在二义性：

```
jack::pal = 3;
jill::pal =10;
```

变量 jack::pal 和 jill::pal 是不同的标识符，表示不同的内存单元。然而，如果使用 using 声明，情况将发生变化：

```
using jack::pal;
using jill::pal;
pal = 4;          // which one? now have a conflict
```

事实上，编译器将不允许您同时使用上述两个 using 声明，因为这将导致二义性。

2. using 编译指令和 using 声明之比较

使用 using 编译指令导入一个名称空间中所有的名称与使用多个 using 声明是不一样的，而更像是大量使用作用域解析操作符。使用 using 声明时，就好像声明了相应的名称一样。如果某个名称已经在函数中声明了，则不能用 using 声明导入相同的名称。然而，使用 using 编译指令时，将进行名称解析，就像在包含 using 声明和名称空间本身的最小声明区域中声明了名称一样。在下面的范例中，名称空间为全局的。如果使用 using 编译指令导入一个已经在函数中声明的名称，则局部名称将隐藏名称空间名，就像隐藏同名的全局变量一样。不过仍可以像下面的范例中那样使用作用域解析操作符：

```
namespace Jill {
   double bucket (double n) { ... }
   double fetch;
   struct Hill { ... };
}
char fetch;                      // global namespace
int main()
{
   using namespace Jill;         // import all namespace names
   Hill Thrill;                  // create a type Jill::Hill structure
   double water = bucket (2);    // use Jill::bucket();
   double fetch;                 // not an error: hides Jill::fetch
   cin >> fetch;                 // read a value into the local fetch
   cin >>::fetch;                // read a value into global fetch
   cin >> Jill::fetch;           // read a value into Jill::fetch
   ...
}

int foom()
{
   Hill top;                     // ERROR
   Jill::Hill crest;             // valid
}
```

在 main()中，名称 Jill::fetch 被放在局部名称空间中，但其作用域不是局部的，因此不会覆盖全局的 fetch。然而，局部声明的 fetch 将隐藏 Jill::fetch 和全局 fetch。不过，如果使用作用域解析操作符，则后两个 fetch 变量都是可用的。读者应将这个范例与前面使用 using 声明的范例进行比较。

需要指出的另一点是，虽然函数中的 using 编译指令将名称空间的名称视为在函数之外声明的，但它不会使得该文件中的其他函数能够使用这些名称。因此，在前一个例子中，foom()函数不能使用未限定的 Hill 标识符。

记住：假设名称空间和声明区域定义了相同的名称。如果试图使用 using 声明将名称空间的名称导入该声明区域，则这两个名称会发生冲突，从而出错。如果使用 using 编译指令将该名称空间的名称导入该声明区域，则局部版本将隐藏名称空间版本。

一般说来，使用 using 声明比使用 using 编译指令更安全，这是由于它只导入指定的名称。如果该名称

与局部名称发生冲突，编译器将发出指示。using 编译指令导入所有名称，包括可能并不需要的名称。如果与局部名称发生冲突，则局部名称将覆盖名称空间版本，而编译器并不会发出警告。另外，名称空间的开放性意味着名称空间的名称可能分散在多个地方，这使得难以准确知道添加了哪些名称。

下面是本书的大部分范例采用的方法：

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

首先，#include 语句将头文件 iostream 放到名称空间 std 中。然后，using 编译指令是该名称空间在 main() 函数中可用。有些范例采取下述方式：

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

这将名称空间 std 中的所有内容导出到全局名称空间中。使用这种方法的主要原因是方便。它完成起来比较简单，同时如果系统不支持名称空间，可以将前两行替换为：

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

不过，名称空间的支持者希望有更多的选择，既可以使用解析操作符，也可以使用 using 声明。也就是说，不要这样做：

```
using namespace std;  // avoid as too indiscriminate
```

而应这样做：

```
int x;
std::cin >> x;
std::cout << x << std::endl;
```

或者这样做：

```
using std::cin;
using std::cout;
using std::endl;
int x;
cin >> x;
cout << x << endl;
```

可以用嵌套式名称空间（将在下一节介绍）来创建一个包含常用 using 声明的名称空间。

3. 名称空间的其他特性

可以将名称空间声明进行嵌套：

```
namespace elements
{
   namespace fire
   {
      int flame;
      ...
   }
   float water;
}
```

这里，flame 指的是 element::fire::flame。同样，可以使用下面的 using 编译指令使内部的名称可用：

```
using namespace elements::fire;
```

另外，也可以在名称空间中使用 using 编译指令和 using 声明，如下所示：

```
namespace myth
{
   using Jill::fetch;
   using namespace elements;
   using std::cout;
   using std::cin;
}
```

假设要访问 Jill::fetch。由于 Jill::fetch 现在位于名称空间 myth（在这里，它被叫作 fetch）中，因此可

以这样访问它：

```
std::cin >> myth::fetch;
```

当然，由于它也位于 Jill 名称空间中，因此仍然可以称作 Jill::fetch：

```
std::cout << Jill::fetch;  // display value read into myth::fetch
```

如果没有与之冲突的局部变量，则也可以这样做：

```
using namespace myth;
cin >> fetch;      // really std::cin and Jill::fetch
```

现在考虑将 using 编译指令用于 myth 名称空间的情况。using 编译指令是可传递的。如果 A *op* B 且 B *op* C，则 A *op* C，则我们说操作 *op* 是可传递的。例如，>操作符是可传递的（也就是说，如果 A>B 且 B>C，则 A>C）。在这个情况下，下面的语句：

```
using namespace myth;
```

将添加 elements 名称空间（也是通过一个 using 编译指令），因此它和下面的语句等价：

```
using namespace myth;
using namespace elements;
```

可以给名称空间创建别名。例如，假设有下面的名称空间：

```
namespace my_very_favorite_things { ... };
```

则可以使用下面的语句让 mvft 成为 my_very_favorite_things 的别名：

```
namespace mvft = my_very_favorite_things;
```

可以使用这种技术来简化对嵌套名称空间的使用：

```
namespace MEF = myth::elements::fire;
using MEF::flame;
```

4. 未命名的名称空间

可以通过省略名称空间的名称来创建未命名的名称空间：

```
namespace       // unnamed namespace
{
   int ice;
   int bandycoot;
}
```

这就像后面跟着 using 编译指令一样，也就是说，在该名称空间中声明的名称的潜在作用域为：从声明点到该声明区域末尾。从这个方面看，它们与全局变量相似。不过，由于这种名称空间没有名称，因此不能显式地使用 using 编译指令或 using 声明来使它在其他位置都可用。具体地说，不能在未命名名称空间所属文件之外的其他文件中，使用该名称空间中的名称，因此这种方法可以替代链接性为内部的静态变量。实际上，C++标准不赞成在名称空间和全局作用域中使用关键字 static（该标准使用“不赞成”表明，这种做法目前合法，但以后修订标准时，很可能将其视为非法）。例如，假设有这样的代码：

```
static int counts;  // static storage, internal linkage
int other();
int main()
{
   ...
}
int other()
{
   ...
}
```

C++标准的意思是，程序员应当这样做：

```
namespace
{
   int counts;  // static storage, internal linkage
}
int other();
int main()
{
   ...
```

```
}
int other()
{
   ...
}
```

9.4.3　名称空间范例

现在来看一个多文件范例，该范例说明了名称空间的一些特性。该程序的第一个文件（参见程序清单 9.10）是头文件，其中包含头文件中常包含的内容：常量、结构定义和函数原型。在这个例子中，这些内容被放在两个名称空间中。第一个名称空间叫 pers，其中包含 Person 结构的定义和两个函数的原型——一个函数用人名填充结构，另一个函数显示结构的内容；第二个名称空间叫作 debts，它定义了一个结构，该结构用来存储人名和金额。该结构使用了 Person 结构，因此，debts 名称空间使用一条 using 编译指令，让 pers 中的名称在 debts 名称空间可用。debts 名称空间也包含一些原型。

程序清单 9.10　namesp.h

```
// namesp.h
// create the pers and debts namespaces
namespace pers
{
   const int LEN = 40;
   struct Person
   {
      char fname[LEN];
      char lname[LEN];
   };
   void getPerson (Person &);
   void showPerson (const Person &);
}

namespace debts
{
   using namespace pers;
   struct Debt
   {
      Person name;
      double amount;
   };

   void getDebt (Debt &);
   void showDebt (const Debt &);
   double sumDebts (const Debt ar[], int n);
}
```

第二个文件（见程序清单 9.11）是源代码文件，它提供了头文件中的函数原型对应的定义。在名称空间中声明的函数名的作用域为整个名称空间，因此定义和声明必须位于同一个名称空间中。这正是名称空间的开放性发挥作用的地方。通过包含 namesp.h（参见程序清单 9.10）导入了原来的名称空间。然后该文件将函数定义添加入到两个名称空间中，如程序清单 9.11 所示。另外，文件 names.cpp 演示了如何使用 using 声明和作用域解析操作符来使名称空间 std 中的元素可用。

程序清单 9.11　namesp.cpp

```
// namesp.cpp -- namespaces
#include <iostream>
#include "namesp.h"

namespace pers
{
```

```
    using std::cout;
    using std::cin;
    void getPerson(Person & rp)
    {
        cout << "Enter first name: ";
        cin >> rp.fname;
        cout << "Enter last name: ";
        cin >> rp.lname;
    }

    void showPerson(const Person & rp)
    {
        std::cout << rp.lname << ", " << rp.fname;
    }
}

namespace debts
{
    void getDebt(Debt & rd)
    {
        getPerson(rd.name);
        std::cout << "Enter debt: ";
        std::cin >> rd.amount;
    }

        void showDebt(const Debt & rd)
        {
            showPerson(rd.name);
            std::cout <<": $" << rd.amount << std::endl;
        }

        double sumDebts(const Debt ar[], int n)
        {
            double total = 0;
            for (int i = 0; i < n; i++)
            total += ar[i].amount;
            return total;
        }
}
```

最后，该程序的第三个文件（参见程序清单 9.12）是一个源代码文件，它使用了名称空间中声明和定义的结构和函数。程序清单 9.12 演示了多种使名称空间标识符可用的方法。

程序清单 9.12　namessp.cpp

```
// usenmsp.cpp -- using namespaces
#include <iostream>
#include "namesp.h"

void other(void);
void another(void);
int main(void)
{
    using debts::Debt;
    using debts::showDebt;
    Debt golf = { {"Benny", "Goatsniff"}, 120.0 };
    showDebt(golf);
    other();
    another();

    return 0;
```

```
}

void other(void)
{
    using std::cout;
    using std::endl;
    using namespace debts;
    Person dg = {"Doodles", "Glister"};
    showPerson(dg);
    cout << endl;
    Debt zippy[3];
    int i;

    for (i = 0; i < 3; i++)
        getDebt(zippy[i]);
    for (i = 0; i < 3; i++)
        showDebt(zippy[i]);
    cout << "Total debt: $" << sumDebts(zippy, 3) << endl;

    return;
}

void another(void)
{
    using pers::Person;;

    Person collector = { "Milo", "Rightshift" };
    pers::showPerson(collector);
    std::cout << std::endl;
}
```

在程序清单 9.12 中，main()函数首先使用了两个 using 声明：

```
using debts::Debt;      // makes the Debt structure definition available
using debts::showDebt;  // makes the showDebt function available
```

注意，using 声明只使用了名称，例如，第二个 using 声明没有描述 showDebt 的返回类型或函数特征标，而只给出了名称；因此，如果函数被重载，则一个 using 声明将导入所有的版本。另外，虽然 Debt 和 showDebt 都使用了 Person 类型，但不必导入任何 Person 名称，因为 debt 名称空间有一条包含 pers 名称空间的 using 编译指令。

接下来，other()函数采用了一种不太好的方法，即使用一条 using 编译指令导入整个名称空间：

```
using namespace debts;  // make all debts and pers names available to other()
```

由于 debts 中的 using 编译指令导入了 pers 名称空间，因此 other()函数可以使用 Person 类型和 showPerson()函数。

最后，another()函数使用 using 声明和作用域解析操作符来访问具体的名称：

```
using pers::Person; ;
pers::showPerson (collector) ;
```

下面是程序清单 9.10～9.12 组成的程序的运行情况：

```
Goatsniff, Benny: $120
Glister, Doodles
Enter first name: Arabella
Enter last name: Binx
Enter debt: 100
Enter first name: Cleve
Enter last name: Delaproux
Enter debt: 120
Enter first name: Eddie
Enter last name: Fiotox
Enter debt: 200
```

```
Binx, Arabella: $100
Delaproux, Cleve: $120
Fiotox, Eddie: $200
Total debt: $420
Rightshift, Milo
```

9.4.4 名称空间及其前途

随着程序员逐渐熟悉名称空间，将出现统一的编程理念。下面是当前的一些指导原则：

- 使用在已命名的名称空间中声明的变量，而不是使用外部全局变量。
- 使用在已命名的名称空间中声明的变量，而不是使用静态全局变量。
- 如果开发了一个函数库或类库，将其放在一个名称空间中。事实上，C++当前提倡将标准函数库放在名称空间 std 中，这种做法扩展到了来自 C 语言中的函数。例如，头文件 math.h 是与 C 语言兼容的，没有使用名称空间，但 C++头文件 cmath 应将各种数学库函数放在名称空间 std 中。实际上，并非所有的编译器都完成了这种过渡。
- 仅将编译指令 using 作为一种将旧代码转换为使用名称空间的权宜之计。
- 不要在头文件中使用 using 编译指令。首先，这样做掩盖了要让哪些名称可用；另外，包含头文件的顺序可能影响程序的行为。如果非要使用编译指令 using，应将其放在所有预处理器编译指令#include 之后。
- 导入名称时，首选使用作用域解析操作符或 using 声明的方法。
- 对于 using 声明，首选将其作用域设置为局部而不是全局。

别忘了，使用名称空间的主旨是简化大型编程项目的管理工作。对于只有一个文件的简单程序，使用 using 编译指令并非什么大逆不道的事。

正如前面指出的，头文件名的变化反映了这些变化。老式头文件（如 iostream.h）没有使用名称空间，但新头文件 iostream 使用了 std 名称空间。

9.5 总结

C++鼓励程序员在开发程序时使用多个文件。一种有效的组织策略是，使用头文件来定义用户类型，为操纵用户类型的函数提供函数原型；并将函数定义放在一个独立的源代码文件中。头文件和源代码文件一起定义和实现了用户定义的类型及其使用方式。最后，将 main()和其他使用这些函数的函数放在第三个文件中。

C++的存储方案决定了变量保留在内存中的时间（储存持续性）以及程序的哪一部分可以访问它（作用域和链接性）。自动变量是在代码块（如函数体或函数体中的代码块）中定义的变量，仅当程序执行到包含定义的代码块时，它们才存在，并且可见。自动变量可以通过使用存储类型说明符 auto 或 register 或者根本不使用说明符（与使用 auto 相同）来声明。register 说明符提示编译器，该变量的使用频率很高。

静态变量在整个程序执行期间都存在。对于在函数外面定义的变量，其所属文件中位于该变量的定义后面的所有函数都可以使用它（文件作用域），并可在程序的其他文件中使用（外部链接性）。另一个文件要使用这种变量，必须使用 extern 关键字来声明它。对于文件间共享的变量，应在一个文件中包含其定义声明（不使用 extern），并在其他文件中包含引用声明（使用 extern）。在函数的外面使用关键字 static 定义的变量的作用域为整个文件，但是不能用于其他文件（内部链接性）。在代码块中使用关键字 static 定义的变量被限制在该代码块内（局部作用域、无链接性），但在整个程序执行期间，它都一直存在并且保持原值。

在默认情况下，C++函数的链接性为外部，因此可在文件间共享；但使用关键字 static 限定的函数的链接性为内部的，被限制在定义它的文件中。

名称空间允许定义一个可在其中声明标识符的命名区域。这样做的目的是减少名称冲突，尤其当程序非常大，并使用多个厂商的代码时。可以通过使用作用域解析操作符、using 声明或 using 编译指令，来使

名称空间中的标识符可用。

9.6　复习题

1．对于下面的情况，应使用哪种存储方案？

a．homer 是函数的形参。

b．secret 变量由两个文件共享。

c．topsecret 变量由一个文件中的所有函数共享，但对于其他文件来说是隐藏的。

d．beencalled 记录包含它的函数被调用的次数。

2．using 声明和 using 编译指令之间有何区别？

3．重新编写下面的代码，使之不使用 using 声明和 using 编译指令。

```
#include <iostream>
using namespace std;
int main()
{
    double x;
    cout << "Enter value: ";
    while (! (cin >> x))
    {
        cout << "Bad input. Please enter a number: ";
        cin.clear();
       while (cin.get()!= '\n')
         continue;
    }
    cout << "Value = " << x << endl;
    return 0;
}
```

4．重新编写下面的代码，使之使用 using 声明，而不是 using 编译指令。

```
#include <iostream>
using namespace std;
int main()
{
    double x;
    cout << "Enter value: ";
    while (! (cin >> x))
    {
        cout << "Bad input. Please enter a number: ";
        cin.clear();
       while (cin.get()!= '\n')
         continue;
    }
    cout << "Value = " << x << endl;
    return 0;
}
```

5．在一个文件中调用 average（3，6）函数时，它返回两个 int 参数的平均值（int），在同一个程序的另一个文件中调用时，它返回两个 int 参数的平均值（double）。应如何实现？

6．下面的程序由两个文件组成，该程序显示什么内容？

```
// file1.cpp
#include <iostream>
using namespace std;
void other();
void another();
int x = 10;
int y;
```

```
int main()
{
    cout << x << endl;
    {
        int x = 4;
        cout << x << endl;
        cout << y << endl;
    }
    other();
    another();
    return 0;
}

void other()
{
    int y = 1;
    cout << "Other: " << x << ", " << y << endl;
}
// file 2.cpp
#include <iostream>
using namespace std;
extern int x;
namespace
{
    int y = -4;
}

void another()
{
    cout << "another(): " << x << ", " << y << endl;
}
```

7. 下面的代码将显示什么内容？

```
#include <iostream>
using namespace std;
void other();
namespace n1
{
    int x = 1;
}

namespace n2
{
    int x = 2;
}
int main()
{
    using namespace n1;
    cout << x << endl;
    {
        int x = 4;
        cout << x << ", " << n1::x << ", " << n2::x << endl;
    }
    using n2::x;
    cout << x << endl;

    other();
    return 0;
}
```

```
void other()
{
    using namespace n2;
    cout << x << endl;
    {
        int x = 4;
        cout << x << ", " << n1::x << ", " << n2::x << endl;
    }
    using n2::x;
    cout << x << endl;
}
```

9.7　编程练习

1. 下面是一个头文件：

```
// golf.h -- for pe9-1.cpp

const int Len = 40;
struct golf
{
    char fullname[Len];
    int handicap;
};

// non-interactive version:
//  function sets golf structure to provided name, handicap
//  using values passed as arguments to the function
void setgolf (golf & g, const char * name, int hc);

// interactive version:
//  function solicits name and handicap from user
//  and sets the members of g to the values entered
//  returns 1 if name is entered, 0 if name is empty string
int setgolf (golf & g);

// function resets handicap to new value
void handicap (golf & g, int hc);

// function displays contents of golf structure
void showgolf (const golf & g);
```

setgolf() 被重载，可以这样使用其第一个版本：

```
golf ann;
setgolf (ann, "Ann Birdfree", 24);
```

上述函数调用提供了存储在 ann 结构中的信息。可以这样使用其第二个版本：

```
golf andy;
setgolf (andy);
```

上述函数将提示用户输入姓名和等级，并将它们存储在 andy 结构中。这个函数可以（但是不一定必须）在内部使用第一个版本。

根据这个头文件，创建一个多文件程序。其中的一个文件名为 golf.cpp，它提供了与头文件中的原型匹配的函数定义；另一个文件应包含 main()，并演示原型化函数的所有特性。例如，包含一个让用户输入的循环，并使用输入的数据来填充一个由 golf 结构组成的数组，数组被填满或用户将高尔夫选手的姓名设置为空字符串时，循环将结束。main() 函数只使用头文件中原型化的函数来访问 golf 结构。

2. 修改程序清单 9.8，用 string 对象代替字符数组。这样，该程序将不再需要检查输入的字符串是否过长，同时可以将输入字符串同字符串""进行比较，以判断是否为空行。

3．下面是一个结构声明：

```
struct chaff
{
    char dross[20];
    int slag;
};
```

编写一个程序，使用布局 new 操作符将一个包含两个这种结构的数组放在一个缓冲区中。然后，给结构的成员赋值（对于 char 数组，使用函数 strcpy()），并使用一个循环来显示内容。方法 1 是像程序清单 9.9 那样将一个静态数组用作缓冲区；方法 2 是使用常规 new 操作符来分配缓冲区。

4．请基于下面这个名称空间编写一个由 3 个文件组成的程序：

```
namespace SALES
{
    const int QUARTERS = 4;
    struct Sales
    {
        double sales[QUARTERS];
        double average;
        double max;
        double min;
    };

    // copies the lesser of 4 or n items from the array ar
    // to the sales member of s and computes and stores the
    // average, maximum, and minimum values of the entered items;
    // remaining elements of sales, if any, set to 0
    void setSales (Sales & s, const double ar[], int n);

    // gathers sales for 4 quarters interactively, stores them
    // in the sales member of s and computes and stores the
    // average, maximum, and minumum values
    void setSales (Sales & s);

    // display all information in structure s
    void showSales (const Sales & s);
}
```

第一个文件是一个头文件，其中包含名称空间。第二个文件是一个源代码文件，它对这个名称空间进行扩展，以提供这 3 个函数的定义。第三个文件声明两个 Sales 对象，并使用 setSales() 的交互式版本为一个结构提供值，然后使用 setSales() 的非交互式版本为另一个结构提供值。另外它还使用 showSales() 来显示这两个结构的内容。

第 10 章　对象和类

本章内容包括：

- 过程性编程和面向对象编程。
- 类概念。
- 如何定义和实现类。
- 公有类访问和私有类访问。
- 类的数据成员。
- 类方法（类函数成员）。
- 创建和使用类对象。
- 类的构造函数和析构函数。
- const 成员函数。
- this 指针。
- 创建对象数组。
- 类作用域。
- 抽象数据类型。

面向对象编程（OOP）是一种特殊的、设计程序的概念性方法，C++通过一些特性改进了 C 语言，使得应用这种方法更容易。下面是最重要的 OOP 特性：

- 抽象。
- 封装和数据隐藏。
- 多态。
- 继承。
- 代码的可重用性。

为了实现这些特性并将它们组合在一起，C++所做的最重要的改进是提供了类。本章首先介绍类，将解释抽象、封装、数据隐藏，并演示类是如何实现这些特性的。本章还将讨论如何定义类、如何为类提供公有部分和私有部分以及如何创建使用类数据的成员函数。另外，还将介绍构造函数和析构函数，它们是特殊的成员函数，用于创建和删除其所属类的对象。最后介绍 this 指针，对于有些类编程而言，它是至关重要的。后面的章节还将把讨论扩展到操作符重载（另一种多态）和继承，它们是代码重用的基础。

10.1　过程性编程和面向对象编程

虽然本书前面偶尔探讨过 OOP 在编程方面的前景，但讨论的更多的还是诸如 C、Pascal 和 BASIC 等语言的标准过程性方法。下面来看一个例子，它揭示了 OOP 的观点与过程性编程的差别。

Genre Giants 垒球队的一名新成员被要求记录球队的统计数据。很自然，会求助于计算机来完成这项工作。如果是一位过程性程序员，可能会这样考虑：

我要输入每名选手的姓名、击球次数、击中次数、命中率（命中率指的是选手正式的击球次数除以击中次数；当选手在垒上或被罚出局时，击球停止，但某些情况不计作正式击球次数，如选手走步时）以及其他重要的基本统计数据。之所以使用计算机，是为了简化工作，因此让它来计算某些数据，如命中率。另外，我还希望程序能够显示这些结果。应如何组织呢？我想我能正确地完成这项工作，并使用了函数。是的，我让 main() 调用一个函数来获取输入，调用另一个函数来进行计算，然后再调用第三个函数来显示结果。那么，获得下一场比赛的数据后，又该做什么呢？我不想再从头开始，可以添加一个函数来更新统计数据。可能需要在 main() 中提供一个菜单，选择是输入、计算、更新还是显示数据。则如何表示这些数据呢？可以用一个字符串数组来存储选手的姓名，用另一个数组存储每一位选手的击球数，再用一个数组存储击中次数等等。这种方法太不灵活了，可以设计一个结构来存储每位选手的所有信息，然后用这种结构组成的数组来表示整个球队。

简而言之，采用过程性编程方法时，首先考虑要遵循的步骤，然后考虑如何表示这些数据（注意：并不需要程序一直运行，用户可能希望能够将数据存储在一个文件中，然后从这个文件中读取数据）。

如果换成一位 OOP 程序员，又将如何呢？首先考虑数据——不仅要考虑如何表示数据，还要考虑如何使用数据：

我要跟踪的是什么？当然是选手。因此要有一个对象表示整个选手的各个方面（而不仅仅是命中率或击球次数）。是的，这将是基本数据单元——一个表示选手的姓名和统计数据的对象。我需要一些处理该对象的方法。首先需要一种将基本信息加入到该单元中的方法；其次，计算机应计算一些东西，如命中率，因此需要添加一些执行计算的方法。程序应自动完成这些计算，而无需用户干涉。另外，还需要一些更新和显示信息的方法。所以，用户与数据交互的方式有 3 种：初始化、更新和报告——这就是用户接口。

简而言之，采用 OOP 方法时，首先从用户的角度考虑对象——描述对象所需的数据以及描述用户与数据交互所需的操作。完成对接口的描述后，需要确定如何实现接口和数据存储。最后，使用新的设计方案创建出程序。

10.2 抽象和类

生活中充满复杂性，处理复杂性的方法之一是简化和抽象。人的身体是由无数个原子组成的，而一些学者认为人的思想是由半自主的主体组成的。但将人自己看作一个实体将简单得多。在计算中，为了根据信息与用户之间的接口来表示它，抽象是至关重要的。也就是说，将问题的本质特征抽象出来，并根据特征来描述解决方案。在垒球统计范例中，接口描述了用户如何初始化、更新和显示数据。抽象是通往用户定义类型的捷径，在 C++中，用户定义类型指的是实现抽象接口的类设计。

10.2.1 类型是什么

我们来看看是什么构成了类型。例如，讨厌鬼是什么？受流行的固定模式影响，可能会指出讨厌鬼的一些外表特点：胖、戴黑宽边眼镜、兜里插满钢笔等。稍加思索后，又可能觉得从行为上定义讨厌鬼可能更合适，例如他（或她）是如何应对尴尬的社交场面的。如果将这种类比扩展到过程性语言（如 C 语言），我们得到类似的情形。首先，倾向于根据数据的外观（在内存中如何存储）来考虑数据类型。例如，char 占用 1 个字节的内存，而 double 通常占用 8 个字节的内存。但是稍加思索就会发现，也可以根据要对它执行的操作来定义数据类型。例如，int 类型可以使用所有的算术运算，可对整数执行加、减、乘、除运算，还可以对它们使用求模操作符（%）。

而指针需要的内存数量很可能与 int 相同，甚至可能在内部被表示为整数。但不能对指针执行与整数相同的运算。例如，不能将两个指针相乘，这种运算没有意义的，因此 C++没有实现这种运算。因此，将变量声明为 int 或 float 指针时，不仅仅是分配内存，还规定了可对变量执行的操作。简而言之，指定基本类型完成了 3 项工作：

- 决定数据对象需要的内存数量。
- 决定如何解释内存中的位（long 和 float 在内存中占用的位数相同，但将它们转换为数值的方法不同）。
- 决定可使用数据对象执行的操作或方法。

对于内置类型来说，有关操作的信息被内置到编译器中。但在 C++中定义用户自定义的类型时，必须自己提供这些信息。付出这些劳动换来了根据实际需要定制新数据类型的强大功能和灵活性。

10.2.2　C++中的类

类是一种将抽象转换为用户定义类型的 C++工具，它将数据表示和操纵数据的方法组合成一个整洁的包。下面来看一个表示股票的类。

首先，必须考虑如何表示股票。可以将一股作为基本单元，定义一个表示一股股票的类。不过，这意味着需要 100 个对象才能表示 100 股，这是不现实的。相反，可以将某人当前持有的某种股票作为一个基本单元，数据表示中包含他持有的股票数量。一种比较现实的方法是，必须记录最初购买价格和购买日期（用于计算纳税）等内容。另外，还必须管理诸如拆股等事件。首次定义类就考虑这么多因素有些困难，因此我们对其进行简化。具体地说，应该将可执行的操作限制为：

- 获得股票。
- 增持。
- 卖出股票。
- 更新股票价格。
- 显示关于所持股票的信息。

可以根据上述清单定义 stock 类的公有接口（如果读者有兴趣，还可以添加其他特性）。为支持该接口，需要存储一些信息。我们再次进行简化。例如，不考虑标准的美式股票计价方式（八分之一美元的倍数。显然，纽约证券交易所一定看到过本书以前的版本中关于简化的论述，因为它已经决定将系统转换为书中采用的方式）。我们将存储下面的信息：

- 公司名称。
- 所持股票的数量。
- 每股的价格。
- 股票总值。

接下来定义类。一般来说，类规范由两个部分组成：

- 类声明：以数据成员的方式描述数据部分，以成员函数（被称为方法）的方式描述公有接口。
- 类方法定义：描述如何实现类成员函数。

简单地说，类声明提供了类的蓝图，而方法定义则提供了细节。

什么是接口

接口是一个共享框架，供两个系统（如在计算机和打印机之间或者用户或计算机程序之间）交互时使用；例如，用户可能是您，而程序可能是字处理器。使用字处理器时，您不能直接将脑子中想到的词传输到计算机内存中，而必须同程序提供的接口交互。您敲打键盘时，计算机将字符显示到屏幕上；您移动鼠标时，计算机移动屏幕上的光标；您无意间单击鼠标时，计算机对您输入的段落进行奇怪的处理。程序接口将您的意图转换为存储在计算机中的具体信息。

对于类，我们说公共接口。在这里，公众（public）是使用类的程序，交互系统由类对象组成，而接口由编写类的人提供的方法组成。接口让程序员能够编写与类对象交互的代码，从而让程序能够使用类对象。例如，要计算 string 对象中包含多少个字符，您无需打开对象，而只需使用 string 类提供的 size()方法。类设计禁止公共用户直接访问类，但公众可以使用 size()方法。size()方法是用户和 string 类对象之间的公共接口的组成部分。通常，方法 getline()是 istream 类的公共接口的组成部分，使用 cin 的程序不是直接与 cin 对象内部交互来读取一行输入，而是使用 getline()。

如果希望更人性化，不要将使用类的程序视为公共用户，而将编写程序的人视为公共用户。然而，要使用某个类，必须了解其公共接口；要编写类，必须创建其公共接口。

为开发一个类并编写一个使用它的程序，需要完成多个步骤。这里将开发过程分成多个阶段，而不是一次性完成。然后，将每个阶段的代码合并在一起，得到程序清单 10.3。程序清单 10.1 是第一个阶段的代码，它是 Stock 类的类声明（为帮助识别类，本书遵循一种常见但不通用的约定——将类名首字母大写）。读者将发现，程序清单 10.1 看起来就像一个结构声明，只是还包括成员函数、公有部分和私有部分等内容。稍后将对该声明进行改进（所以不要将它用作模型），但首先让我们来看一看该定义的工作方式。

程序清单 10.1 stocks.cpp 的第一部分

```
// beginning of stocks.cpp file
#include <iostream>
#include <cstring>

class Stock  // class declaration
{
private:
    char company[30];
    int shares;
    double share_val;
    double total_val;
    void set_tot() { total_val = shares * share_val; }
public:
    void acquire (const char * co, int n, double pr);
    void buy (int num, double price);
    void sell (int num, double price);
    void update (double price);
    void show();
};   // note semicolon at the end
```

稍后将详细介绍类的细节，但先看一下更为通用的特性。首先，C++关键字 class 指出这些代码定义了一个类设计，这种句法指出，Stock 是这个新类的类型名。该声明让我们能够声明 Stock 类型的变量——称为对象或实例。每个对象都表示一支股票。例如，下面的声明：

```
Stock sally;
Stock solly;
```

创建两个 Stock 对象，它们分别名为 sally 和 solly。例如，sally 对象可以表示 Sally 持有的某公司的股票。

接下来，要存储的数据以类数据成员（如 company 和 shares）的形式出现。例如，sally 的 company 成员存储了公司名称，share 成员存储了 Sally 持有的股票数量，share_val 成员存储了每股的价格，total_val 成员存储了股票总价格。同样，要执行的操作以类函数成员（方法，如 sell()和 update()）的形式出现。成员函数可以就地定义（如 set_tot()），也可以用原型表示（如其他成员函数）。其他成员函数的完整定义稍后将介绍，但对于描述函数接口而言，原型足够了。将数据和方法组合成一个单元是类最吸引人的特性。有了这种设计，创建 Stock 对象时，将自动制定使用对象的规则。

istream 和 ostream 类有成员函数，如 get()和 getline()，而 Stock 类声明中的函数原型说明了成员函数是如何建立的。例如，头文件 iostream 将 getline()的原型放在 istream 类的声明中。

关键字 private 和 public 也是新的，它们描述了对类成员的访问控制。使用类对象的程序都可以直接访问公有部分，但只能通过公有成员函数（或友元函数，参见第 11 章）来访问对象的私有成员。例如，要修改 Stock 类的 shares 成员，只能通过 Stock 的成员函数。因此，公有成员函数是程序和对象的私有成员之间的桥梁，提供了对象和程序之间的接口。防止程序直接访问数据被称为数据隐藏（参见图 10.1）。（C++还提供了第三个访问控制关键字 protected，我们将在第 13 章介绍类继承时，讨论该关键字。）虽然在证券基金章程中，隐藏数据是非法的，但在计算方面却是一个好主意，因为它确保了数据的完整性。

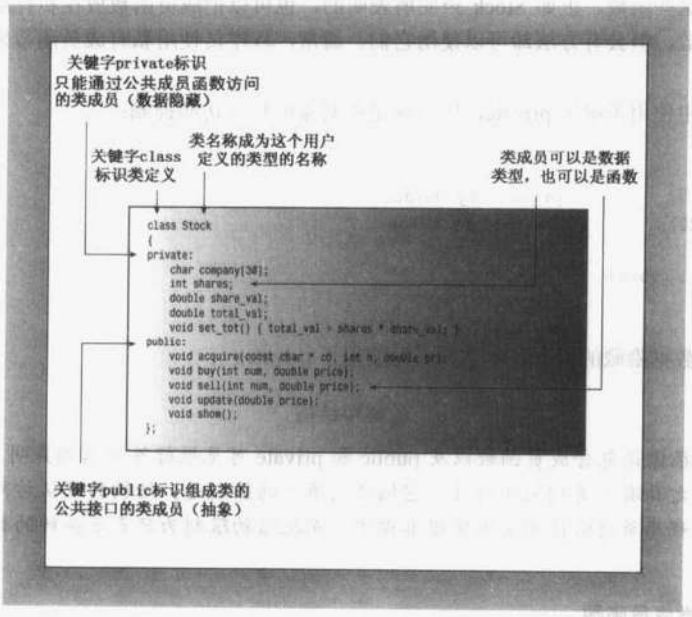


图 10.1　Stock 类

类设计尽可能将公有接口与实现细节分开。公有接口表示设计的抽象组件。将实现细节放在一起并将它们与抽象分开被称为封装。数据隐藏（将数据放在类的私有部分中）是一种封装，将实现的细节隐藏在私有部分中，就像 Stock 类对 set_tot()所做的那样，也是一种封装。封装的另一个例子是，将类函数定义和类声明放在不同的文件中。

OOP 和 C++

OOP 是一种编程风格，从某种程度说，它用于任何一种语言中。当然，可以将 OOP 思想融合到常规的 C 语言程序中。例如，在第 9 章的一个范例（程序清单 9.1、9.2、9.3）中，头文件中包含结构原型和操纵该结构的函数的原型，便是这样的例子。因此，main()函数只需定义这个结构类型的变量，并使用相关函数处理这些变量即可；main()不直接访问结构成员。实际上，该范例定义了一种抽象类型，它将存储格式和函数原型置于头文件中，对 main()隐藏了实际的数据表示。不过，C++中包括了许多专门用来实现 OOP 方法的特性，因此它使程序员更进一步。首先，将数据表示和函数原型放在一个类声明中（而不是放在一个文件中），通过将所有内容放在一个类声明中，来使描述成为一个整体。其次，让数据表示成为私有，使得数据只能被授权的函数访问。在 C 语言的例子中，如果 main()直接访问了结构成员，则违反了 OOP 的精神，但没有违反 C 语言的规则。然而，试图直接访问 Stock 对象的 shares 成员便违反了 C++语言的规则，编译器将捕获这种错误。

数据隐藏不仅可以防止直接访问数据，还让开发者（类的用户）无需了解数据是如何被表示的。例如，show()成员将显示某支股票的总价格（还有其他内容），这个值可以存储在对象中（上述代码正是这样做的），也可以在需要时通过计算得到。从用户的角度看，使用哪种方法没有什么区别。所需要知道的只是各种成员函数的功能；也就是说，需要知道成员函数接受什么样的参数以及返回什么类型的值。原则是将实现细节从接口设计中分离出来。如果以后找到了更好的、实现数据表示或成员函数细节的方法，可以对这些细节进行修改，而无须修改程序接口，这使程序维护起来更容易。

控制对成员的访问：公有还是私有

无论类成员是数据成员还是成员函数，都可以在类的公有部分或私有部分中声明它。但由于隐藏数据是 OOP 主要的目标之一，因此数据项通常放在私有部分，组成类接口的成员函数放在公有部分；否则，就

无法从程序中调用这些函数。正如 Stock 声明所表明的，也可以把成员函数放在私有部分中。不能直接从程序中调用这种函数，但公有方法却可以使用它们。通常，程序员使用私有成员函数来处理不属于公有接口的实现细节。

不必在类声明中使用关键字 private，因为这是类对象的默认访问控制：

```
class World
{
    float mass;          // private by default
    char name[20];       // private by default
public:
    void tellall (void);
    ...
};
```

然而，为强调数据隐藏的概念，本书显式地使用了 private。

类和结构

类描述看上去很像是包含成员函数以及 public 和 private 可见性标签的结构声明。实际上，C++对结构进行了扩展，使之具有与类相同的特性。它们之间惟一的区别是，结构的默认访问类型是 public，而类为 private。C++程序员通常使用类来实现类描述，而把结构限制为只表示纯粹的数据对象或没有私有部分的类。

10.2.3 实现类成员函数

还需要创建类描述的第三部分：为那些由类声明中的原型表示的成员函数提供代码。成员函数定义与常规函数定义非常相似，它们有函数头和函数体，也可以有返回类型和参数。但是它们还有两个特殊的特征：

- 定义成员函数时，使用作用域解析操作符（::）来标识函数所属的类。
- 类方法可以访问类的 private 组件。

首先，成员函数的函数头使用作用域解析操作符（::）来指出函数所属的类。例如，update()成员函数的函数头如下：

```
void Stock::update (double price)
```

这种表示法意味着我们定义的 update()函数是 Stock 类的成员。这不仅将 update()标识为成员函数，还意味着我们可以将另一个类的成员函数也命名为 update()。例如，Buffoon()类的 update()函数的函数头如下：

```
void Buffoon::update()
```

因此，作用域解析操作符确定了方法定义对应的类的身份。我们说，标识符 update()具有类作用域（class scope）。Stock 类的其他成员函数不必使用作用域解析操作符，就可以使用 update()方法，这是因为它们属于同一个类，因此 update()是可见的。不过，在类声明和方法定义之外使用 update()时，需要采取特殊的措施，稍后将作介绍。

类方法的完整名称中包括类名。我们说，Stock::update()是函数的限定名（qualified name）；而简单的 update()是全名的缩写（非限定名，unqualified name），它只能在类作用域中使用。

方法的第二个特点是，方法可以访问类的私有成员。例如，show()方法可以使用这样的代码：

```
cout << "Company: " << company
     << "  Shares: " << shares << endl
     << "  Share Price: $" << share_val
     << "  Total Worth: $" << total_val << endl;
```

其中，company、shares 等都是 Stock 类的私有数据成员。如果试图使用非成员函数访问这些数据成员，编译器禁止这样做（不过，第 11 章中将介绍的友元函数例外）。

了解这两点后，就可以实现类方法了，如程序清单 10.2 所示。这些方法定义可以放在单独一个文件中，也可以位于类声明所在的文件中。因为刚刚开始，所以我们假设这些定义位于类声明的后面。这是让方法定义可以使用类声明的最简单的方法，虽然不是最好的方法（本章稍后将使用最好的方法，即将类声明放

在头文件中，而将类成员函数定义放在单独的一个源代码文件中）。为让读者获得更多有关名称空间的经验，在有些方法中使用了限定符 std::，在其他方法中则使用了 using 声明。

程序清单 10.2　stocks.cpp

```
//more stocks.cpp -- implementing the class member functions
void Stock::acquire(const char * co, int n, double pr)
{
    std::strncpy(company, co, 29);  // truncate co to fit company
    company[29] = '\0';
    if (n < 0)
    {
        std::cerr << "Number of shares can't be negative. "
            << company << " shares set to 0.\n";
        shares = 0;
    }
    else
        shares = n;
    share_val = pr;
    set_tot();
}
void Stock::buy(int num, double price)
{
    if (num < 0)
    {
        std::cerr << "Number of shares purchased can't be negative. "
            << "Transaction is aborted.\n";
    }
    else
    {
        shares += num;
        share_val = price;
        set_tot();
    }
}
void Stock::sell(int num, double price)
{
    using std::cerr;
    if (num < 0)
    {
        cerr << "Number of shares sold can't be negative. "
            << "Transaction is aborted.\n";
    }
    else if (num > shares)
    {
        cerr << "You can't sell more than you have! "
        << "Transaction is aborted.\n";
    }
    else
    {
        shares -= num;
        share_val = price;
        set_tot();
    }
}
void Stock::update(double price)
{
    share_val = price;
    set_tot();
}
```

```
void Stock::show()
{
    using std::cout;
    using std::endl;
    cout << "Company: " << company
        << "  Shares: " << shares << endl
        << "  Share Price: $" << share_val
        << "  Total Worth: $" << total_val << endl;
}
```

1. 成员函数说明

acquire()函数管理对某个公司股票的首次购买，而 buy()和 sell()管理增加或减少持有的股票。方法 buy()和 sell()确保买入或卖出的股数不为负。另外，如果用户试图卖出超过他（或她）持有的股票数量，则 sell()函数将结束这次交易。这种使数据私有并限于对公有函数访问的技术允许我们能够控制数据如何被使用：在这个例子中，它允许我们加入这些安全防护措施，避免不适当的交易。

4 个成员函数设置或重新设置了 total_val 成员值。这个类并非将计算代码编写 4 次，而是让每个函数都调用 set_tot()函数。由于 set_tot()只是实现代码的一种方式，而不是公有接口的组成部分，因此这个类将其声明为私有成员函数（即编写这个类的人可以使用它，但编写代码来使用这个类的人不能使用）。如果计算代码很长，则这种方法还可以省去许多输入代码的工作，并可节省空间。不过，这种方法的主要价值在于，通过使用函数调用，而不是每次重新输入计算代码，可以确保执行的计算完全相同。另外，如果必须修订计算代码（在这个例子中，这种可能性不大），则只需在一个地方进行修改即可。

acquire()方法使用 strncpy()来复制字符串。函数调用 strncpy（*s2*，*s1*，*n*）将 *s1* 复制到 *s2* 或从 *s1* 复制 *n* 个字符到 *s2*。如果 *s1* 包含的字符少于 *n*，则 strncpy()函数将用空值字符填充 *s2*。也就是说，strncpy（firstname，“Tim”，6）将字符 *T*、*i*、*m* 复制到 firstname 中，然后添加 3 个空值字符，使总字符数为 6 个。但如果 *s1* 长于 *n*，则不需要添加空值字符。也就是说，strncpy（firstname，“Priscilla”，4）仅将字符 *P*、*r*、*i* 和 *s* 复制到 firstname，使其成为一个字符数组，但由于其末尾没有空值字符，因此不是字符串。因此，acquire()在数组结尾处放一个空值字符，以确保它是字符串。

记住：和 cout 一样，cerr 也是一个 ostream 对象。它们之间的区别在于，操作系统重定向只影响 cout，而不影响 cerr。cerr 对象用于错误消息。因此，如果将程序输出重定向到文件，并且发生了错误，则屏幕上仍然会出现错误消息。在 UNIX 系统中，可以分别对 cout 和 cerr 进行重定向，命令行操作符>用于对 cout 进行重定向，操作符 2>对 cerr 进行重定向。

2. 内联方法

其定义位于类声明中的函数都将自动成为内联函数，因此 Stock::set_tot()是一个内联函数。类声明常将短小的成员函数作为内联函数，set_tot()符合这样的要求。

如果愿意，也可以在类声明之外定义成员函数，并使其成为内联函数。为此，只需在类实现部分中定义函数时使用 inline 限定符即可：

```
class Stock
{
private:
    ...
    void set_tot(); // definition kept separate
public:
    ...
};

inline void Stock::set_tot() // use inline in definition
{
    total_val = shares * share_val;
}
```

内联函数的特殊规则要求在每个使用它们的文件中都对其进行定义。确保内联定义对多文件程序中的所有文件都可用的、最简便的方法是：将内联定义放在定义类的头文件中（有些开发系统包含智能链接程序，允许将内联定义放在一个独立的实现文件）。

根据改写规则（rewrite rule），在类声明中定义方法等同于用原型替换方法定义，然后在类声明的后面将定义改写为内联函数。也就是说，程序清单 10.2 中 set_tot() 的内联定义与上述代码（定义紧跟在类声明之后）是等价的。

3. 方法使用哪个对象

下面介绍使用对象时最重要的一个方面：如何将类方法应用于对象。下面的代码：

```
shares += num;
```

使用了一个对象的 shares 成员。是哪一个对象呢？这是一个非常好的问题！要回答这个问题，首先来看看如何创建对象。最简单的方式是声明类变量：

```
Stock kate, joe;
```

这将创建 2 个 Stock 类对象，一个为 kate，另一个为 joe。

接下来，看看如何使用对象的成员函数。和使用结构成员一样，通过成员操作符：

```
kate.show();    // the kate object calls the member function
joe.show();     // the joe object calls the member function
```

第 1 条语句调用 kate 对象的 show() 成员。这意味着 show() 方法将把 shares 解释为 kate.shares，将 share_vla 解释为 kate.share_val。同样，函数调用 joe.show() 使 show() 方法将 shares 和 share_val 分别解释为 joe.shares 和 joe.share_val。

记住：调用成员函数时，它将使用被用来调用它的对象的数据成员。

同样，函数调用 kate.sell() 在调用 set_tot() 函数时，相当于调用 kate.set_tot()，这样该函数将使用 kate 对象的数据。

所创建的每个新对象都有自己的存储空间，用于存储其内部变量和类成员；但同一个类的所有对象共享同一组类方法，即每种方法只有一个副本。例如，假设 kate 和 joe 都是 Stock 对象，则 kate.shares 将占据一个内存块，而 joe.shares 占用另一个内存块，但 kate.show() 和 joe.show() 都调用同一个方法，也就是说，它们将执行同一个代码块，只是将这些代码用于不同的数据。在 OOP 中，调用成员函数被称为发送消息，因此将同样的消息发送给 2 个不同的对象将调用同一个方法，但该方法被用于 2 个不同的对象（参见图 10.2）。

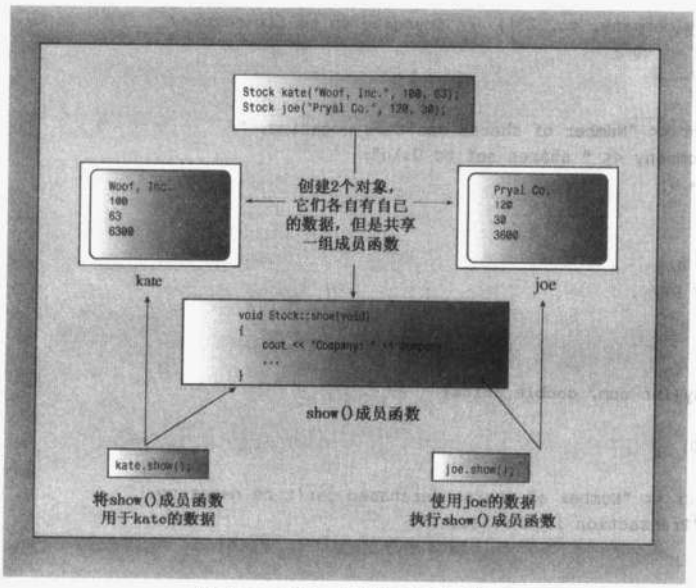


图 10.2　对象、数据和成员函数

10.2.4 使用类

知道如何定义类及其方法后，我们来创建一个程序，该程序创建并使用类对象。C++的目标是使得使用类与使用基本的内置类型（如 int 和 char）尽可能相同。要创建类对象，可以声明类变量，也可以使用 new 为类对象分配存储空间。可以将对象作为函数的参数和返回值，也可以将一个对象赋给另一个。C++提供了一些工具，可用于初始化对象、让 cin 和 cout 识别对象，甚至在相似的类对象之间进行自动类型转换。虽然要做到这些工作还需要一段时间，不过我们可以先从比较简单的属性着手。实际上，读者已经知道如何声明类对象和调用成员函数。程序清单 10.3 将这些技术与类声明和成员函数定义组合在一起，创建了一个完整的程序。该程序创建了一个名为 stock1 的 Stock 对象。程序非常简单，但确实测试了这个类的特性。

程序清单 10.3 完整的 stocks.cpp 程序

```
// stocks.cpp -- the whole program
#include <iostream>
#include <cstring>

class Stock  // class declaration
{
private:
    char company[30];
    int shares;
    double share_val;
    double total_val;
    void set_tot() { total_val = shares * share_val; }
public:
    void acquire(const char * co, int n, double pr);
    void buy(int num, double price);
    void sell(int num, double price);
    void update(double price);
    void show();
};   // note semicolon at the end

void Stock::acquire(const char * co, int n, double pr)
{
    std::strncpy(company, co, 29);  // truncate co to fit company
    company[29] = '\0';
    if (n < 0)
    {
        std::cerr << "Number of shares can't be negative. "
            << company << " shares set to 0.\n";
        shares = 0;
    }
    else
      shares = n;
    share_val = pr;
    set_tot();
}

void Stock::buy(int num, double price)
{
    if (num < 0)
    {
        std::cerr << "Number of shares purchased can't be negative. "
            << "Transaction is aborted.\n";
    }
    else
    {
```

```
        shares += num;
        share_val = price;
        set_tot();
    }
}

void Stock::sell(int num, double price)
{
    using std::cerr;
    if (num < 0)
    {
        cerr << "Number of shares sold can't be negative. "
            << "Transaction is aborted.\n";
    }
    else if (num > shares)
    {
        cerr << "You can't sell more than you have! "
            << "Transaction is aborted.\n";
    }
    else
    {
        shares -= num;
        share_val = price;
        set_tot();
    }
}

void Stock::update(double price)
{
    share_val = price;
    set_tot();
}

void Stock::show()
{
    using std::cout;
    using std::endl;
    cout << "Company: " << company
        << "  Shares: " << shares << endl
        << "  Share Price: $" << share_val
        << "  Total Worth: $" << total_val << endl;
}

int main()
{
    using std::cout;
    using std::ios_base;
    Stock stock1;
    stock1.acquire("NanoSmart", 20, 12.50);
    cout.setf(ios_base::fixed);          // #.## format
    cout.precision(2);                   // #.## format
    cout.setf(ios_base::showpoint);      // #.## format
    stock1.show();
    stock1.buy(15, 18.25);
    stock1.show();
    stock1.sell(400, 20.00);
    stock1.show();
    return 0;
}
```

程序清单 10.3 中的程序使用了 3 个格式化命令：

```
cout.setf(ios_base::fixed);    // use fixed decimal point format
cout.precision(2);             // two places to right of decimal
cout.setf(ios_base::showpoint); // show trailing zeros
```

还使用了下述 using 声明：

```
using std::ios_base;
```

这是一种嵌套名称空间的情形。标识符 fixed 和 showpoint 位于名称空间 ios_base 中，而名称空间 ios_base 位于名称空间 std 中。程序的效果是显示两位小数，包括结尾的 0。实际上，根据当前的实际情况，只需前两个命令就足够了；而早期 C++实现则只需第一个和第三个命令。使用这 3 个命令在这两种实现中产生的输出相同。具体细节请参阅第 17 章。下面是该程序的输出：

```
Company: NanoSmart  Shares: 20
   Share Price: $12.50  Total Worth: $250.00
Company: NanoSmart  Shares: 35
   Share Price: $18.25  Total Worth: $638.75
You can't sell more than you have! Transaction is aborted.
Company: NanoSmart  Shares: 35
   Share Price: $18.25  Total Worth: $638.75
```

注意，main()只是用来测试 Stock 类的设计。当 Stock 类的运行情况与预期的相同后，便可以在其他程序中将 Stock 类作为用户定义的类型使用。要使用新类型，最关键的是要了解成员函数的功能，而不必考虑其实现细节。请参阅后面的注释“客户/服务器模型”。

客户/服务器模型

OOP 程序员常依照客户/服务器模型来讨论程序设计。在这个概念中，客户是使用类的程序。类声明（包括类方法）构成了服务器，它是程序可以使用的资源。客户只能通过以公有方式定义的接口使用服务器，这意味着客户（客户程序员）惟一的责任是了解该接口。服务器（服务器设计人员）的责任是确保服务器根据该接口可靠并准确地执行。服务器设计人员只能修改类设计的实现细节，而不能修改接口。这样程序员独立地对客户和服务器进行改进，对服务器的修改不会客户的行为造成意外的影响。

10.2.5 小结

指定类设计的第一步是提供类声明。类声明类似结构声明，可以包括数据成员和函数成员。声明有私有部分，在其中声明的成员只能通过成员函数进行访问；声明还具有公有部分，在其中声明的成员可被使用类对象的程序直接访问。通常，数据成员被放在私有部分中，成员函数被放在公有部分中，因此典型的类声明的格式如下：

```
class className
{
private:
   data member declarations
public:
   member function prototypes
};
```

公有部分的内容构成了设计的抽象部分——公有接口。将数据封装到私有部分中可以保护数据的完整性，这被称为数据隐藏。因此，C++通过类使得实现抽象、数据隐藏和封装等 OOP 特性很容易。

指定类设计的第二步是实现类成员函数。可以在类声明中提供完整的函数定义，而不是函数原型，但是通常的做法是单独提供函数定义（除非函数很小）。在这种情况下，需要使用作用域解析操作符来指出成员函数属于哪个类。例如，假设 Bozo 有一个名为 Retort()的成员函数，该函数返回 char 指针，则其函数头如下所示：

```
char * Bozo::Retort()
```

换句话来说，Retort()不仅是一个 char *类型的函数，而是一个属于 Bozo 类的 char *函数。该函数的全名（或限定名）为 Bozo::Retort()。而名称 Retort()是限定名的缩写，只能在某些特定的环境中使用，如类方法的代码中。

另一种描述这种情况的方式是，名称 Retort 的作用域为整个类，因此在类声明和类方法之外使用该名

称时，需要使用作用域解析操作符进行限定。

要创建对象（类的实例），只需将类名视为类型名即可：

```
Bozo bozetta;
```

这样做是可行的，因为类是用户定义的类型。

类成员函数（方法）可通过类对象来调用。为此，需要使用成员操作符句点：

```
cout << Bozetta.Retort();
```

这将调用 Retort() 成员函数，每当其中的代码引用某个数据成员时，该函数都将使用 bozetta 对象中相应成员的值。

10.3　类的构造函数和析构函数

对于 Stock 类，还有其他一些工作要做。应为类提供被称为构造函数和析构函数的标准函数。下面来看一看为什么需要这些函数以及如何使用这些函数。

C++的目标之一是让使用类对象就像使用标准类型一样，然而，到现在为止，本章提供的代码还不能让您像初始化 int 或结构那样来初始化 Stock 对象。也就是说，常规的初始化句法不适用于类型 Stock：

```
int year = 2001;                                    // valid initialization
struct thing
{
    char * pn;
    int m;
};
thing amabob = {"wodget", -23};                     // valid initialization
Stock hot = {"Sukie's Autos, Inc.", 200, 50.25};    // NO! compile error
```

不能像上面这样初始化 Stock 对象的原因在于，数据部分的访问状态是私有的，这意味着程序不能直接访问数据成员。读者已经看到，程序只能通过成员函数来访问数据成员，因此需要设计合适的成员函数，才能成功地将对象初始化（如果使数据成员成为公有，而不是私有，就可以按刚才介绍的方法初始化类对象，但使数据成为公有的违背了类的一个主要初衷：数据隐藏）。

一般来说，最好是在创建对象时对它进行初始化。例如，请看下面的代码：

```
Stock gift;
gift.buy(10, 24.75);
```

就 Stock 类当前的实现而言，gift 对象的 company 成员是没有值的。类设计假设用户在调用任何其他成员函数之前调用 acquire()，但无法强加这种假设。避开这种问题的方法之一是在创建对象时，自动对它进行初始化。为此，C++提供了一个特殊的成员函数——类构造函数，专门用于构造新对象、将值赋给它们的数据成员。更准确地说，C++为这些成员函数提供了名称和使用句法，而程序员需要提供方法定义。名称与类名相同。例如，Stock 类一个可能的构造函数是名为 Stock() 的成员函数。构造函数的原型和函数头有一个有趣的特征——虽然没有返回值，但没有被声明为 void 类型。实际上，构造函数没有声明类型。

10.3.1　声明和定义构造函数

现在需要创建 Stock 的构造函数。由于需要为 Stock 对象提供 3 个值，因此应为构造函数提供 3 个参数（第 4 个值，total_val 成员，是根据 shares 和 share_val 计算得到的，因此不必为构造函数提供这个值）。程序员可能只想设置 company 成员，而将其他值设置为 0；这可以使用默认参数来完成（参见第 8 章）。因此，原型如下所示：

```
// constructor prototype with some default arguments
Stock(const char * co, int n = 0, double pr = 0.0);
```

第一个参数是指向字符串的指针，该字符串用于初始化类成员 company 字符数组。*n* 和 *pr* 参数为 shares 和 share_val 成员提供值。注意，没有返回类型。原型位于类声明的公有部分。

下面是构造函数的一种可能定义：

```
// constructor definition
Stock::Stock(const char * co, int n, double pr)
{
    std::strncpy(company, co, 29);
    company[29] = '\0';

    if (n < 0)
    {
        std::cerr << "Number of shares can't be negative;"
            << company << " shares set to 0.\n";
        shares = 0;
    }
    else
        shares = n;
    share_val = pr;
    set_tot();
}
```

上述代码和本章前面的 acquire() 函数相同。区别在于，程序声明对象时，将自动调用构造函数。

警告：不熟悉构造函数的读者会试图将类成员名称用作构造函数的参数，如下例所示：

```
// NO!
Stock::Stock (const char * company, int shares, double share_val)
{
...
}
```

这是错误的。构造函数的参数表示的不是类成员，而是赋给类成员的值。因此，参数名不能与类成员相同，否则最终的代码将是这样的：

```
shares = shares;
```

为避免这种混乱，通常的做法是在数据成员名中使用 m_ 前缀：

```
class Stock
{
private:
        string m_company;
        int m_shares;
  ....
```

10.3.2 使用构造函数

C++提供了两种使用构造函数来初始化对象的方式。第一种方式是显式地调用构造函数：

```
Stock food = Stock ("World Cabbage", 250, 1.25);
```

这将 food 对象的 company 成员设置为字符串“World Cabbage”，将 shares 成员设置为 250，依此类推。另一种方式是隐式地调用构造函数：

```
Stock garment ("Furry Mason", 50, 2.5);
```

这种格式更紧凑，它与下面的显式调用等价：

```
Stock garment = Stock ("Furry Mason", 50, 2.5) );
```

每次创建类对象（甚至使用 new 动态分配内存）时，C++都将使用类构造函数。下面是将构造函数与 new 一起使用的方法：

```
Stock *pstock = new Stock ("Electroshock Games", 18, 19.0);
```

这条语句创建一个 Stock 对象，将其初始化为参数提供的值，并将该对象的地址赋给 pstock 指针。在这种情况下，对象没有名称，但可以使用指针来管理该对象。我们将在第 11 章进一步讨论对象指针。

构造函数的使用方式不同于其他类方法。一般来说，使用对象来调用方法：

```
stock1.show();  // stock1 object invokes show()method
```

但无法使用对象来调用构造函数，因为在构造函数构造出对象之前，对象是不存在的。因此构造函数被用来创建对象，而不能通过对象来调用。

10.3.3　默认构造函数

默认构造函数是在未提供显式的初始化值时，被用来创建对象的构造函数。也就是说，它是用于下面这种声明的构造函数：

```
Stock stock1;  // uses the default constructor
```

程序清单 10.3 就是这样做的！这条语句管用的原因在于，如果没有提供任何构造函数，则 C++将自动提供默认构造函数。它是默认构造函数的隐式版本，不做任何工作。对于 Stock 类来说，默认构造函数可能如下：

```
Stock::Stock(){}
```

因此将创建 stock1 对象，但不初始化其成员，这和下面的语句：

```
int x;
```

创建 x，但没有提供值给它一样。默认构造函数没有参数，因为声明中不包含值。

奇怪的是，当且仅当没有定义任何构造函数时，编译器才会提供默认构造函数。为类定义了构造函数后，程序员就必须为它提供默认构造函数。如果提供了非默认构造函数（如 Stock（const char * co，int n，double pr）），但没有提供默认构造函数，则下面的声明：

```
Stock stock1;  // not possible with current constructor
```

将出错。这样做的原因可能是想禁止创建未初始化的对象。不过，如果要创建对象，而不显式地初始化，则必须定义一个不接受任何参数的默认构造函数。定义默认构造函数的方式有两种。一种是给已有构造函数的所有参数提供默认值：

```
Stock(const char * co = "Error", int n = 0, double pr = 0.0);
```

另一种方式是通过函数重载来定义另一个构造函数——一个没有参数的构造函数：

```
Stock();
```

由于只能有一个默认构造函数，因此不要同时采用这两种方式（在早期 C++版本中，只能使用第二种方式来创建默认构造函数）。

实际上，通常应初始化所有的对象，以确保所有成员一开始就有已知的、合理的值。因此，默认构造函数通常给所有成员提供隐式初始值。例如，下面是为 Stock 类定义的一个默认构造函数：

```
Stock::Stock()    //default constructor
{
    std::strcpy(company, "no name");
    shares = 0;
    share_val = 0.0;
    total_val = 0.0;
}
```

提示：在设计类时，通常应提供对所有类成员做隐式初始化的默认构造函数。

使用上述任何一种方式（没有参数或所有参数都有默认值）创建了默认构造函数后，便可以声明对象变量，而不对它们进行显式初始化：

```
Stock first;                    // calls default constructor implicitly
Stock first = Stock();          // calls it explicitly
Stock *prelief = new Stock;     // calls it implicitly
```

不过，不要被非默认构造函数的隐式形式所误导：

```
Stock first("Concrete Conglomerate");    // calls constructor
Stock second();                          // declares a function
Stock third;                             // calls default constructor
```

第一个声明调用非默认构造函数，即接受参数的构造函数；第二个声明指出，second()是一个返回 Stock 对象的函数。隐式地调用默认构造函数时，不要使用圆括号。

10.3.4　析构函数

用构造函数创建对象后，程序负责跟踪该对象，直到其过期为止。对象过期时，程序将自动调用一个特殊的成员函数，该函数的名称令人生畏——析构函数。析构函数完成清理工作，因此实际上很有用。例

如，如果构造函数使用 new 来分配内存，则析构函数将使用 delete 来释放这些内存。Stock 的构造函数没有使用 new，因此析构函数实际上没有需要完成的任务。在这种情况下，只需让编译器生成一个什么要不做的隐式析构函数即可，Stock 类第一版正是这样做的。然而，了解如何声明和定义析构函数是绝对必要的，下面为 Stock 类提供一个析构函数。

和构造函数一样，析构函数的名称也很特殊：在类名前加上(~)。因此，Stock 类的析构函数为~Stock()。另外，和构造函数一样，析构函数也可以没有返回值和声明类型。与构造函数不同的是，析构函数没有参数，因此 Stock 析构函数的原型必须是这样的：

```
~Stock();
```

由于 Stock 的析构函数不承担任何重要的工作，因此可以将它编写为不执行任何操作的函数：

```
Stock::~Stock()
{
}
```

不过，为让读者能看出析构函数何时被调用，我们将这样编写其代码：

```
Stock::~Stock()   // class destructor
{
   cout << "Bye, " << company << "!\n";
}
```

什么时候应调用析构函数呢？这由编译器决定，通常不应在代码中显式地调用析构函数（有关例外情形，请参阅第 12 章的“再谈布局 new 操作符”）。如果创建的是静态存储类对象，则其析构函数将在程序结束时自动被调用。如果创建的是自动存储类对象（就像前面的范例中那样），则其析构函数将在程序执行完代码块时（该对象是在其中定义的）自动被调用。如果对象是通过 new 创建的，则它将驻留在堆栈内存或自由存储区中，当使用 delete 来释放内存时，其析构函数将自动被调用。最后，程序可以创建临时对象来完成特定的操作，在这种情况下，程序将在结束对该对象的使用时自动调用其析构函数。

由于在类对象过期时析构函数将自动被调用，因此必须有一个析构函数。如果程序员没有提供析构函数，编译器将隐式地声明一个默认析构函数，并在发现导致对象被删除的代码后，提供默认析构函数的定义。

10.3.5 改进 Stock 类

下面将构造函数和析构函数加入到类和方法的定义中。这次我们按 C++通常的做法，将程序组织成独立的文件。我们将类声明放在头文件 stock1.h 中（这个文件名表明，今后可能还将进行修订）。类方法被放在文件 stock1.cpp 中。一般来说，包含类声明的头文件和包含方法定义的源代码文件的文件名应相同，以便跟踪哪些文件与哪些文件是一起的。将类声明和成员函数放在不同的文件中，可以将接口的抽象定义（类声明）和实现细节（成员函数定义）分开。例如，可以以文本头文件的方式来发布类声明，而以编译代码的方式来发布函数定义。最后，我们将使用这些资源的程序放在第三个文件中，这个文件名为 usestock1.cpp。

1. 头文件

程序清单 10.4 列出了头文件。它将构造函数和析构函数的原型加入到原来的类声明中。另外，它还删除了 acquire()函数——现在已经不再需要它了，因为有构造函数。该文件还使用第 9 章介绍的#ifndef 技术来防止多重包含。

程序清单 10.4 stock1.h

```
// stock1.h -- Stock class declaration with constructors, destructor added
#ifndef STOCK1_H_
#define STOCK1_H_

class Stock
{
private:
   char company[30];
   int shares;
```

```
    double share_val;
    double total_val;
    void set_tot() { total_val = shares * share_val; }
public:
    Stock();        // default constructor
    Stock(const char * co, int n = 0, double pr = 0.0);
    ~Stock();       // noisy destructor
    void buy(int num, double price);
    void sell(int num, double price);
    void update(double price);
    void show();
};

#endif
```

2. 实现文件

程序清单 10.5 提供了方法的定义。它包含了 stock1.h 文件，以提供类声明（将文件名放在双引号而不是方括号中意味着编译器将源文件所在的目录中搜索它）。另外，程序清单 10.5 还包含了头文件 iostream 和 cstring，以提供对 I/O 以及函数 strcpy() 和 strncpy() 的支持。该程序清单还使用 using 声明和限定名称（如 std::string）来访问头文件中的各种声明。该文件将构造函数和析构函数的方法定义添加到以前的方法定义中。为让读者知道这些方法何时被调用，它们都显示一条消息。这并不是构造函数和析构函数的常规功能，但有助于读者更好地了解类是如何使用它们的。

程序清单 10.5　stock1.cpp

```
// stock1.cpp -- Stock class implementation with constructors, destructor added
#include <iostream>
#include "stock1.h"

// constructors (verbose versions)
Stock::Stock()          // default constructor
{
    std::cout << "Default constructor called\n";
    std::strcpy(company, "no name");
    shares = 0;
    share_val = 0.0;
    total_val = 0.0;
}

Stock::Stock(const char * co, int n, double pr)
{
    std::cout << "Constructor using " << co << " called\n";
    std::strncpy(company, co, 29);
    company[29] = '\0';

    if (n < 0)
    {
        std::cerr << "Number of shares can't be negative; "
            << company << " shares set to 0.\n";
        shares = 0;
    }
    else
        shares = n;
    share_val = pr;
    set_tot();
}
// class destructor
Stock::~Stock()       // verbose class destructor
{
```

```
    std::cout << "Bye, " << company << "!\n";
}

// other methods
void Stock::buy(int num, double price)
{
    if (num < 0)
    {
        std::cerr << "Number of shares purchased can't be negative. "
            << "Transaction is aborted.\n";
    }
    else
    {
        shares += num;
        share_val = price;
        set_tot();
    }
}
void Stock::sell(int num, double price)
{
    using std::cerr;
    if (num < 0)
    {
        cerr << "Number of shares sold can't be negative. "
            << "Transaction is aborted.\n";
    }
    else if (num > shares)
    {
        cerr << "You can't sell more than you have! "
            << "Transaction is aborted.\n";
    }
    else
    {
        shares -= num;
        share_val = price;
        set_tot();
    }
}

void Stock::update(double price)
{
    share_val = price;
    set_tot();
}

void Stock::show()
{
    using std::cout;
    using std::endl;
    cout << "Company: " << company
        << "  Shares: " << shares << endl
        << "  Share Price: $" << share_val
        << "  Total Worth: $" << total_val << endl;
}
```

3. 客户文件

程序清单 10.6 提供了一个测试这些新方法的小程序；由于它只是使用了 Stock 类，因此是 Stock 类的客户。和 stock1.cpp 一样，它也包含了 stock1.h 文件以提供类声明。该程序显示了构造函数和析构函数，它还使用了程序清单 10.3 调用的格式化命令。要编译整个程序，必须使用第 1 章和第 8 章介绍的多文件程

序技术。

程序清单 10.6　usestock1.cpp

```
// usestok1.cpp -- using the Stock class
#include <iostream>
#include "stock1.h"
int main()
{
    using std::cout;
    using std::ios_base;
    cout.precision (2);                                    // #.## format
    cout.setf (ios_base::fixed, ios_base::floatfield);     // #.## format
    cout.setf (ios_base::showpoint);                       // #.## format

    cout << "Using constructors to create new objects\n";
    Stock stock1 ("NanoSmart", 12, 20.0);                  // syntax 1
    stock1.show();
    Stock stock2 = Stock ("Boffo Objects", 2, 2.0);        // syntax 2
    stock2.show();

    cout << "Assigning stock1 to stock2: \n";
    stock2 = stock1;
    cout << "Listing stock1 and stock2: \n";
    stock1.show();
    stock2.show();

    cout << "Using a constructor to reset an object\n";
    stock1 = Stock ("Nifty Foods", 10, 50.0);    // temp object
    cout << "Revised stock1: \n";
    stock1.show();
    cout << "Done\n";
    return 0;
}
```

注意：可能需要使用 ios::，而不是 ios_base::。

编译程序清单 10.4、10.5 和 10.6 表示的程序，得到一个可执行程序。下面是使用某个编译器得到的可执行程序的输出：

```
Using constructors to create new objects
Constructor using NanoSmart called
Company: NanoSmart  Shares: 12
   Share Price: $20.00  Total Worth: $240.00
Constructor using Boffo Objects called
Company: Boffo Objects  Shares: 2
   Share Price: $2.00  Total Worth: $4.00
Assigning stock1 to stock2:
Listing stock1 and stock2:
Company: NanoSmart  Shares: 12
   Share Price: $20.00  Total Worth: $240.00
Company: NanoSmart  Shares: 12
   Share Price: $20.00  Total Worth: $240.00
Using a constructor to reset an object
Constructor using Nifty Foods called
Bye, Nifty Foods!
Revised stock1:
Company: Nifty Foods  Shares: 10
   Share Price: $50.00  Total Worth: $500.00
Done
Bye, NanoSmart!
Bye, Nifty Foods!
```

使用某些编译器编译该程序时，该程序输出的前半部分可能如下（比前面多了一行）：

```
Using constructors to create new objects
Constructor using NanoSmart called
Company: NanoSmart  Shares: 12
   Share Price: $20.00  Total Worth: $240.00
Constructor using Boffo Objects called
Bye, Boffo Objects!                    //additional line
Company: Boffo Objects  Shares: 2
   Share Price: $2.00  Total Worth: $4.00
...
```

下一小节将解释输出行“Bye，Boffo Objects!”。

4. 程序说明

程序清单 10.6 中的下述语句：

```
Stock stock1 ("NanoSmart", 12, 20.0);
```

创建一个名为 stock1 的 Stock 对象，并将其数据成员初始化为指定的值：

```
Constructor using NanoSmart called
Company: NanoSmart  Shares: 12
```

下面的语句：

```
Stock stock2 = Stock ("Boffo Objects", 2, 2.0);
```

使用另一种句法创建并初始化一个名为 stock2 的对象。C++标准允许编译器使用两种方式来执行第二种句法。一种是使其行为和第一种句法完全相同：

```
Constructor using Boffo Objects called
Company: Boffo Objects  Shares: 2
```

另一种方式是允许调用构造函数来创建一个临时对象，然后将该临时对象复制到 stock2 中，并丢弃它。如果编译器使用的是这种方式，则将为临时对象调用析构函数，因此生成下面的输出：

```
Constructor using Boffo Objects called
Bye, Boffo Objects!
Company: Boffo Objects  Shares: 2
```

生成上述输出的编译器可能立刻删除临时对象，但也可能会等一段时间，在这种情况下，析构函数的消息将会过一段时间才显示。

下面的语句表明：

```
stock2 = stock1;     // object assignment
```

可以将一个对象赋给同类型的另一个对象。和给结构赋值一样，在默认情况下，给类对象赋值时，将把一个对象的成员复制给另一个。在这个例子中，stock2 原来的内容将被覆盖。

记住：在默认情况下，将一个对象赋给同类型的另一个对象时，C++将源对象的每个数据成员的内容复制到目标对象中相应的数据成员中。

构造函数不仅仅可用于初始化新对象。例如，该程序的 main() 中包含下面的语句：

```
stock1 = Stock ("Nifty Foods", 10, 50.0);
```

stock1 对象已经存在，因此这条语句不是对 stock1 进行初始化，而是将新值赋给它。这是通过让构造程序创建一个新的、临时的对象，然后将其内容复制给 stock1 来实现的。随后程序调用析构函数，以删除该临时对象，如下面经过注释后的输出所示：

```
Using a constructor to reset an object
Constructor using Nifty Foods called //temporary object created
Bye, Nifty Foods!                    //temporary object destroyed
Revised stock1:
Company: Nifty Foods  Shares: 10     //data now copied to stock1
   Share Price: $50.00  Total Worth: $500.00
```

有些编译器可能要过一段时间才删除临时对象，因此析构函数的调用将延迟。

最后，程序显示了下面的内容：

```
Done
```

```
Bye, NanoSmart!
Bye, Nifty Foods!
```

函数 main()结束时，其局部变量（stock1 和 stock2）将消失。由于这种自动变量被放在堆栈中，因此最后创建的对象将最先被删除，最先创建的对象将最后被删除（“NanoSmart”最初位于 stock1 中，但随后被传输到 stock2 中，然后 stock1 被重置为“Nifty Food”）。

输出表明，下面两条语句有根本性的差别：

```
Stock stock2 = Stock ("Boffo Objects", 2, 2.0);
stock1 = Stock ("Nifty Foods", 10, 50.0); // temporary object
```

第一条语句是初始化，它创建有指定值的对象，可能会创建临时对象（也可能不会）；第二条语句是赋值。像这样在赋值语句中使用构造函数总会导致在赋值前创建一个临时对象。

提示：如果既可以通过初始化，也可以通过赋值来设置对象的值，则应采用初始化方式。通常这种方式的效率更高。

5. const 成员函数

请看下面的代码片段：

```
const Stock land = Stock ("Kludgehorn Properties");
land.show();
```

对于当前的 C++来说，编译器将拒绝第二行。这是什么原因呢？因为 show()的代码无法确保调用对象不被修改——调用对象和 const 一样，不应被修改。我们以前通过将函数参数声明为 const 引用或指向 const 的指针来解决这种问题。但在这里，存在句法上的问题：show()方法没有任何参数。相反，它所使用的对象是由方法调用隐式地提供的。我们需要一种新的句法——保证函数不会修改调用对象。C++的解决方法是将 const 关键字放在函数的括号后面。也就是说，show()声明应像这样：

```
void show()const;        // promises not to change invoking object
```

同样，函数定义的开头应像这样：

```
void stock::show()const  // promises not to change invoking object
```

以这种方式声明和定义的类函数被称为 const 成员函数。就像应尽可能将 const 引用和指针用作函数形参一样，只要类方法不修改调用对象，就应将其声明为 const。从现在开始，我们将遵守这一规则。

10.3.6　构造函数和析构函数小结

我们已经介绍一些构造函数和析构函数的例子，读者可能希望停下来，整理一下学到的知识。为此，下面对这些方法进行总结。

构造函数是一种特殊的类成员函数，在创建类对象时被调用。构造函数的名称和类名相同，但通过函数重载，可以创建多个同名的构造函数，条件是每个函数的特征标（参数列表）都不同。另外，构造函数没有声明类型。通常，构造函数用于初始化类对象的成员，初始化应与构造函数的参数列表匹配。例如，假设 Bozo 类的构造函数的原型如下：

```
Bozo (const char * fname, const char * lname);  // constructor prototype
```

则可以使用它来初始化新对象：

```
Bozo bozetta = bozo ("Bozetta", "Biggens");     // primary form
Bozo fufu ("Fufu", "O'Dweeb");                  // short form
Bozo *pc = new Bozo ("Popo", "Le Peu");         // dynamic object
```

如果构造函数只有一个参数，则将对象初始化为一个与参数的类型相同的值时，该构造函数将被调用。例如，假设有这样一个构造函数原型：

```
Bozo (int age);
```

则可以使用下面的任何一种形式来初始化对象：

```
Bozo dribble = bozo (44);       // primary form
Bozo roon (66);                 // secondary form
Bozo tubby = 32;                // special form for one-argument constructors
```

实际上，第三个范例是新内容，不属于复习内容，但现在正是介绍它的好时机。第 11 章将介绍了一种可关闭这项特性的方式。

记住：接受一个参数的构造函数允许使用赋值句法来将对象初始化为一个值：

```
Classname object = value;
```

默认构造函数没有参数，因此如果创建对象时没有进行显式地初始化，则将调用默认构造函数。如果程序中没有提供任何构造函数，则编译器会为程序定义一个默认构造函数；否则，必须自己提供默认构造函数。默认构造函数可以没有任何参数；如果有，则必须给所有参数都提供默认值：

```
Bozo();                                    // default constructor prototype
Bistro(const char * s = "Chez Zero");      // default for Bistro class
```

对于未被初始化的对象，程序将使用默认构造函数来创建：

```
Bozo bubi;          // use default
Bozo *pb = new Bozo;  // use default
```

就像对象被创建时程序将调用构造函数一样，当对象被删除时，程序将调用析构函数。每个类都只能有一个析构函数。析构函数没有返回类型（连 void 都没有），也没有参数，其名称为类名称前加上~。例如，Bozo 类的析构函数的原型如下：

```
~Bozo();    // class destructor
```

如果构造函数使用了 new，则必须提供使用 delete 的析构函数。

10.4 this 指针

对于 Stock 类，还有很多工作要做。到目前为止，每个类成员函数都只涉及到一个对象，即调用它的对象。但有时候方法可能涉及到两个对象，在这种情况下需要使用 C++的 this 指针。

虽然 Stock 类声明可以显示数据，但它缺乏分析能力。例如，从 show()的输出我们可以知道持有的哪一支股票价格最高，但由于程序无法直接访问 total_val，因此无法作出判断。要让程序知道存储的数据，最直接的方式是让方法返回一个值。为此，通常使用内联代码，如下例所示：

```
class Stock
{
private:
    ...
    double total_val;
    ...
public:
    double total()const { return total_val; }
    ...
};
```

就直接程序访问而言，上述定义实际上是使 total_val 为只读的。也就是说，可以使用方法 total_val()来获得 total_val 的值，但这个类没有提供专门用于重新设置 total_val 的值的方法（作为一种副产品，其他方法，如 buy()、sell()和 update()确实在重新设置成员 shares 和 share_val 的值的同时修改了 total_val 的值）。

通过将该函数添加到类声明中，可以让程序查看一系列股票，找到价格最高的那一支。不过，可以采用另一种方法——一种帮助读者了解 this 指针的方法。这种方法是，定义一个成员函数，它查看两个 Stock 对象，并返回股价较高的那个对象的引用。实现这种方法时，将出现一些有趣的问题，下面就来讨论这些问题。

首先，如何将两个要比较的对象提供给成员函数呢？例如，假设将该方法命名为 topval()，则函数调用 stock1.topval()将访问 stock1 对象的数据，而 stock2.topval()将访问 stock2 对象的数据。如果希望该方法对两个对象进行比较，则必须将第二个对象作为参数传递给它。出于效率方面的考虑，可以按引用来传递参数，也就是说，topval()方法使用一个类型为 const Stock &的参数。

其次，如何将方法的答案传回给调用程序呢？最直接的方法是让方法返回一个引用，该引用指向股价总值较高的对象。因此，用于比较的方法的原型如下：

```
const Stock & topval(const Stock & s) const;
```

该函数隐式地访问一个对象，而显式地访问另一个对象，并返回其中一个对象的引用。括号中的 const 表明，该函数不会修改被显式地访问的对象；而括号后的 const 表明，该函数不会修改被隐式地访问的对象。由于该函数返回了两个 const 对象之一的引用，因此返回类型也应为 const 引用。

假设要对 Stock 对象 stock1 和 stock2 进行比较，并将其中股价总值较高的那一个赋给 top 对象，则可以使用下面两条语句之一：

```
top = stock1.topval (stock2) ;
top = stock2.topval (stock1) ;
```

第一种格式隐式地访问 stock1，而显式地访问 stock2；第二种格式显式地访问 stock1，而隐式地访问 stock2（参见图 10.3）。无论使用哪一种方式，都将对这两个对象进行比较，并返回股价总值较高的那一个对象。

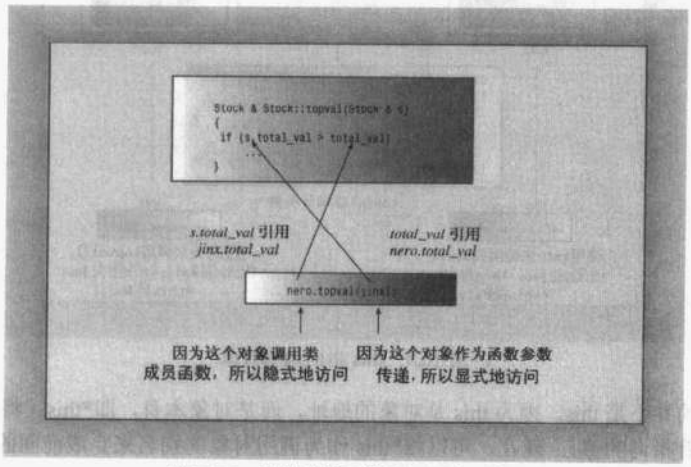


图 10.3　使用成员函数访问两个对象

实际上，这种表示法有些混乱。如果可以使用关系操作符>来比较这两个对象，将更为清晰。可以使用操作符重载（参见第 11 章）完成这项工作。

同时，还要注意的是 topval() 的实现，它将引发一个小问题。下面的部分实现强调了这个问题：

```
const Stock & Stock::topval (const Stock & s) const
{
   if (s.total_val > total_val)
      return s;              // argument object
    else
      return ?????;          // invoking object
}
```

其中，s.total_val 是作为参数传递的对象的总值，total_val 是用来调用该方法的对象的总值。如果 s.total_val 大于 toatl_val，则函数将返回 s；否则，将返回用来调用该方法的对象（在 OOP 中，是 topval 消息要发送给的对象）。问题在于，如何称呼这个对象？如果调用 stock1.topval（stock2），则 s 是 stock2 的引用（即 stock2 的别名），但 stock1 没有别名。

C++解决这种问题的方法是：使用被称为 this 的特殊指针。this 指针指向用来调用成员函数的对象（this 被作为隐藏参数传递给方法）。这样，函数调用 stock1.topval（stock2）将 this 设置为 stock1 对象的地址，使得这个指针可用于 topval() 方法。同样，函数调用 stock2.topval（stock1）将 this 设置为 stock2 对象的地址。一般来说，所有的类方法都将 this 指针设置为调用它的对象的地址。确实，topval() 中的 total_val 只不过是 this->total_val 的简写（第 4 章使用->操作符，通过指针来访问结构成员。这也适用于类成员）（参见图 10.4）。

this 指针

每个成员函数（包括构造函数和析构函数）都有一个 this 指针。this 指针指向调用对象。如果方法需

要引用整个调用对象，则可以使用表达式*this。在函数的括号后面使用 const 限定符将 this 限定为 const，这样将不能使用 this 来修改对象的值。

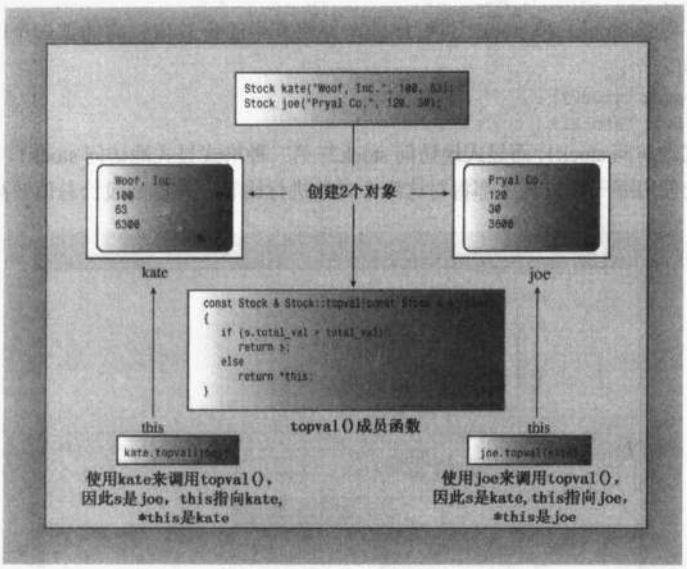


图 10.4 this 指向调用对象

不过，要返回的并不是 this，因为 this 是对象的地址，而是对象本身，即*this（将解除引用操作符*用于指针，将得到指针指向的值）。现在，可以将*this 作为调用对象的别名来完成前面的方法定义。

```
const Stock & Stock::topval (const Stock & s) const
{
   if (s.total_val > total_val)
     return s;          // argument object
   else
     return *this;      // invoking object
}
```

返回类型为引用意味着返回的是调用对象本身，而不是其拷贝。程序清单 10.7 列出了新的头文件。

程序清单 10.7 stock2.h

```
// stock2.h -- augmented version
#ifndef STOCK2_H_
#define STOCK2_H_

class Stock
{
private:
   char company[30];
   int shares;
   double share_val;
   double total_val;
   void set_tot(){ total_val = shares * share_val; }
public:
   Stock();        // default constructor
   Stock (const char * co, int n = 0, double pr = 0.0);
   ~Stock(){}     // do-nothing destructor
   void buy (int num, double price);
   void sell (int num, double price);
   void update (double price);
   void show()const;
```

```
    const Stock & topval (const Stock & s) const;
};

#endif
```

程序清单 10.8 列出了修订后的类方法文件，其中包括新的 topval() 方法。另外，现在读者已经了解了构造函数和析构函数的工作原理，因此这里没有显示消息。

程序清单 10.8　stock2.cpp

```
// stock2.cpp -- improved version
#include <iostream>
#include "stock2.h"

// constructors
Stock::Stock()          // default constructor
{
    std::strcpy(company, "no name");
    shares = 0;
    share_val = 0.0;
    total_val = 0.0;
}

Stock::Stock(const char * co, int n, double pr)
{
    std::strncpy(company, co, 29);
    company[29] = '\0';

    if (n < 0)
    {
        std::cerr << "Number of shares can't be negative. "
            << company << " shares set to 0.\n";
        shares = 0;
    }
    else
        shares = n;
    share_val = pr;
    set_tot();
}

// class destructor
Stock::~Stock()       // quiet class destructor
{
}

// other methods
void Stock::buy(int num, double price)
{
    if (num < 0)
    {
        std::cerr << "Number of shares purchased can't be negative. "
            << "Transaction is aborted.\n";
    }
    else
    {
        shares += num;
        share_val = price;
        set_tot();
    }
}
void Stock::sell(int num, double price)
```

```
{
    using std::cerr;
    if (num < 0)
    {
        cerr << "Number of shares sold can't be negative. "
            << "Transaction is aborted.\n";
    }
    else if (num > shares)
    {
        cerr << "You can't sell more than you have! "
            << "Transaction is aborted.\n";
    }
    else
    {
        shares -= num;
        share_val = price;
        set_tot();
    }
}

void Stock::update(double price)
{
    share_val = price;
    set_tot();
}

void Stock::show() const
{
    using std::cout;
    using std::endl;
    cout << "Company: " << company
        << "  Shares: " << shares << endl
        << "  Share Price: $" << share_val
        << "  Total Worth: $" << total_val << endl;
}

const Stock & Stock::topval(const Stock & s) const
{
    if (s.total_val > total_val)
        return s;
    else
        return *this;
}
```

当然，我们想知道 this 指针是否有用。显然，应在一个包含对象数组的程序中使用这种新方法。因此接下来介绍对象数组这一主题。

10.5 对象数组

和 Stock 范例一样，用户通常要创建同一个类的多个对象。可以创建独立对象变量，就像本章前面的范例所做的，但创建对象数组将更合适。这似乎是在介绍一个未知领域，但实际上，声明对象数组的方法与声明标准类型数组相同：

```
Stock mystuff[4]; // creates an array of 4 Stock objects
```

前面讲过，当程序创建未被显式初始化的类对象时，总是调用默认构造函数。上述声明要求，这个类要么没有显式地定义任何构造函数（在这种情况下，将使用不执行任何操作的隐式默认构造函数），要么定义了一个显式默认构造函数（就像这个例子那样）。每个元素（mystuff[0]、mystuff[1]等）都是 Stock 对象，

可以使用 Stock 方法：

```
mystuff[0].update();        // apply update()to 1st element
mystuff[3].show();          // apply show()to 4th element
Stock tops = mystuff[2].topval (mystuff[1]);
                            // compare 3rd and 2nd elements
```

可以用构造函数来初始化数组元素。在这种情况下，必须为每个元素调用构造函数：

```
const int STKS = 4;
Stock stocks[STKS] = {
    Stock ("NanoSmart", 12.5, 20),
    Stock ("Boffo Objects", 200, 2.0),
    Stock ("Monolithic Obelisks", 130, 3.25),
    Stock ("Fleep Enterprises", 60, 6.5)
    };
```

这里的代码使用标准格式对数组进行初始化：用括号括起的、以逗号分隔的值列表。其中，每次构造函数调用表示一个值。如果类包含多个构造函数，则可以对不同的元素使用不同的构造函数：

```
const int STKS = 10;
Stock stocks[STKS] = {
    Stock ("NanoSmart", 12.5, 20),
    Stock(),
    Stock ("Monolithic Obelisks", 130, 3.25),
};
```

上述代码使用 Stock（const char * co，int n，double pr）来初始化 stock[0]和 stock[2]，使用 Stock() 构造函数来初始化 stock[1]。由于该声明只初始化了数组的部分元素，因此余下的 7 个元素将使用默认构造函数进行初始化。

初始化对象数组的方案是，首先使用默认构造函数创建数组元素，然后花括号中的构造函数将创建临时对象，然后将临时对象的内容复制到相应的元素中。因此，要创建类对象数组，则这个类必须有默认构造函数。

警告：要创建类对象数组，则这个类必须有默认构造函数。

程序清单 10.9 在一个小程序中使用了这些原理，该程序对 4 个数组元素进行初始化，显示它们的内容，并找出这些元素中总值最高的一个。由于 topval() 每次只检查两个对象，因此程序使用 for 循环来检查整个数组。该程序使用程序清单 10.7 中的头文件和 10.8 中的方法文件。

程序清单 10.9　usestock2.cpp

```
// usestok2.cpp -- using the Stock class
// compile with stock2.cpp
#include <iostream>
#include "stock2.h"

const int STKS = 4;
int main()
{
    using std::cout;
    using std::ios_base;

// create an array of initialized objects
    Stock stocks[STKS] = {
        Stock("NanoSmart", 12, 20.0),
        Stock("Boffo Objects", 200, 2.0),
        Stock("Monolithic Obelisks", 130, 3.25),
        Stock("Fleep Enterprises", 60, 6.5)
    };

    cout.precision(2);                          // #.## format
    cout.setf(ios_base::fixed, ios_base::floatfield);// #.## format
```

```
    cout.setf(ios_base::showpoint);              // #.## format

    cout << "Stock holdings:\n";
    int st;
    for (st = 0; st < STKS; st++)
        stocks[st].show();

    Stock top = stocks[0];
    for (st = 1; st < STKS; st++)
        top = top.topval(stocks[st]);
    cout << "\nMost valuable holding:\n";
    top.show();

    return 0;
}
```

注意：可能需要使用 ios::，而不是 ios_base::。

下面是该程序的输出：

```
Stock holdings:
Company: NanoSmart  Shares: 12
    Share Price: $20.00  Total Worth: $240.00
Company: Boffo Objects  Shares: 200
    Share Price: $2.00  Total Worth: $400.00
Company: Monolithic Obelisks  Shares: 130
    Share Price: $3.25  Total Worth: $422.50
Company: Fleep Enterprises  Shares: 60
    Share Price: $6.50  Total Worth: $390.00

Most valuable holding:
Company: Monolithic Obelisks  Shares: 130
    Share Price: $3.25  Total Worth: $422.50
```

有关程序清单 10.9，需要注意的一点是，大部分工作是在类设计中完成的。完成类设计后，编写程序的工作本身便相当简单。

知道 this 指针就可以更深入了解 C++的工作方式。例如，C++前端 cfront 将 C++程序转换为 C 程序。处理方法的定义时，只需将下面这样的 C++方法定义：

```
void Stock::show()const
{
    cout << "Company: " << company
        << "  Shares: " << shares << '\n'
        << "  Share Price: $" << share_val
        << "  Total Worth: $" << total_val << '\n';
}
```

转换为下面这样的 C-风格定义：

```
void show (const Stock * this)
{
    cout << "Company: " << this->company
        << "  Shares: " << this->shares << '\n'
        << "  Share Price: $" << this->share_val
        << "  Total Worth: $" << this->total_val << '\n';
}
```

即将 Stock::限定符转换为函数参数（指向 Stock 的指针），然后用这个指针来访问类成员。

同样，该前端将把下面的函数调用：

```
top.show();
```

转换为：

```
show (&top) ;
```

这样，将调用对象的地址赋给了 this 指针（实际情况可能更复杂些）。

10.6　接口和实现小结

通过使用一个包含 30 个元素的字符数组，Stock 类将其能够存储的名称限制为最长 29 个字符。例如，下面的语句将导致存储在 firm 对象中的为截短后的字符串“Dunkelmister, Dostoyevsky, a”：

```
Stock firm ("Dunkelmeister, Dostoyevsky, and Delderfield Construction", 8, 2.5);
```

可以通过修改类实现来消除对名称的长度限制。方法之一是增加数组长度，但这将导致存储空间被浪费。另一种方法是使用 string 而不是字符数组，因而依赖 string 对象自动调整大小的功能。这要求修改 3 个地方。

首先，在头文件 stock2.h 中包含头文件 string，以提供对 string 类的支持。由于在 stock2.cpp 中包含了 stock2.h，因此 stock2.cpp 也将获得对 string 类的支持。

第二个需要修改的地方是 stock2.h 中类定义的私有部分，将下面的代码：

```
class Stock
{
private:
    char company[30];
...
```

修改成如下所示：

```
class Stock
{
private:
    std::string company;
...
```

第三个需要修改的地方是 stock2.cpp 中的构造函数定义，将下面的代码：

```
Stock::Stock(const char * co, int n, double pr)
{
    std::strncpy(company, co, 29);
    company[29] = '\0';
    ...
```

修改成如下所示：

```
Stock::Stock(const char * co, int n, double pr)
{
    company = co;    // assign C-style string to string object
    ...
```

这些修改都属于修改实现。您修改了类声明的私有部分，这种修改是在实现文件中完成的。然而，您没有修改类的公有部分，可供使用这个类的人选择的方法仍然相同，因此类接口没变。现在，Stock 对象可以存储整个字符串“Dunkelmeister, Dostoyevsky, and Delderfield Construction”了，但创建 Stock 对象的代码不变：

```
Stock firm ("Dunkelmeister, Dostoyevsky, and Delderfield Construction", 8, 2.5);
```

还可以新增一个构造函数：

```
Stock(const std::string & co, int n, double pr);
```

这修改了接口。可供使用的方法增多了，使用这个类的程序员有新的选择。例如，可以在程序中使用这样的代码：

```
string business;
getline(cout, business);
Stock mine(business, 10, 120.5);   // use new constructor
```

总之，修改类的私有部分和实现文件属于实现更变；修改类的公有部分属于接口变更。实现变更改变了类的内部工作原理，接口变更改变了使用类的人可用的编码方式。

读者可能会问，是否可以将一个类的对象用作另一个类的成员。答案是肯定的，毕竟类的目标之一是使用户定义的类型与内置类型尽可能相似。类的构造函数负责确保调用对象成员的构造函数。名为成员初

始化列表的特性可提高构造函数设计的效率，第 14 章将更深入地讨论这个主题。

10.7 类作用域

第 9 章介绍了全局（或文件）作用域和局部（或代码块）作用域。可以在全局变量所属文件的任何地方使用它，而局部变量只能在其所属的代码块中使用。函数名称的作用域也可以是全局的，但不能是局部的。C++类引入了一种新的作用域：类作用域。

在类中定义的名称（如类数据成员名和类成员函数名）的作用域都为整个类，作用域为整个类的名称只在该类中是已知的，在类外是不可知的。因此，可以在不同类中使用相同的类成员名而不会引起冲突。例如，Stock 类的 shares 成员不同于 JobRide 类的 shares 成员。另外，类作用域意味着不能从外部直接访问类的成员，公有成员函数也是如此。也就是说，要调用公有成员函数，必须通过对象：

```
Stock sleeper ("Exclusive Ore", 100, 0.25); // create object
sleeper.show();      // use object to invoke a member function
show();              // invalid -- can't call method directly
```

同样，在定义成员函数时，必须使用作用域解析操作符：

```
void Stock::update (double price)
{
    ...
}
```

简而言之，在类声明或成员函数定义中，可以使用未修饰的成员名称（未限定的名称），就像 sell()调用 set_tot()成员函数时那样。构造函数名称在被调用时，才能被识别，因为它的名称与类名相同。在其他情况下，使用类成员名时，必须根据上下文使用直接成员操作符（.）、间接成员操作符（->）或作用域解析操作符（::）。下面的代码片段演示了如何访问具有类作用域的标识符：

```
class Ik
{
private:
     int fuss;                          // fuss has class scope
public:
     Ik (int f  = 9) {fuss = f; }       // fuss is in scope
     void ViewIk()const;          // ViewIk has class scope
};

void Ik::ViewIk()const            // Ik::places ViewIk into scope
{
     cout << fuss << endl;              // fuss in scope within class methods
}
...
int main()
{
     Ik * pik = new Ik;
     Ik ee = Ik (8);   // constructor in scope because has class name
     ee.ViewIk();      // class object brings ViewIk into scope
     pik->ViewIk();    // pointer-to-Ik brings ViewIk into scope
...
```

作用域为整个类的常量

有时候，使符号常量的作用域为整个类很有用。例如，Stock 类声明使用字面值 30 来指定 company 数组的长度。另外，由于常量对于所有对象来说都是相同的，因此创建一个由所有对象共享的常量是个不错的主意。读者可能以为这样做可行：

```
class Stock
{
private:
```

```
const int Len = 30;    // declare a constant? FAILS
char company[Len];
...
```

但这是行不通的，因为声明类只是描述了对象的形式，并没有真正创建对象。因此，在被对象创建之前，将没有用于存储值的空间。不过，有两种方式可以实现我们的目的，并且效果相同。

第一种方式是在类中声明一个枚举。在类声明中声明的枚举的作用域为整个类，因此可以用枚举为整型常量提供作用域为整个类的符号名称。也就是说，可以这样开始 Stock 声明：

```
class Stock
{
private:
    enum {Len = 30};  // class-specific constant
    char company[Len];
    ...
```

注意，用这种方式声明枚举并不会创建类数据成员。也就是说，所有对象中都不包含枚举。另外，Len 只是一个符号名称，在作用域为整个类的代码中遇到它时，编译器将用 30 来替换它。

由于这里使用枚举只是为了创建符号常量，并不打算创建枚举类型的变量，因此不需要提供枚举名。在许多实现中，ios_base 类在其公有部分中完成了类似的工作，诸如 ios_base::fixed 等标识符就来自这里。其中，fixed 是 ios_base 类中定义的典型的枚举量。

最近，C++引入了另一种在类中定义常量的方式——使用关键字 static：

```
class Stock
{
private:
    static const int Len = 30;    // declare a constant! WORKS
    char company[Len];
    ...
```

这将创建一个名为 Len 的常量，该常量将与其他静态变量存储在一起，而不是存储在对象中。因此，只有一个 Len 常量，它被所有 Stock 对象共享。第 12 章将深入介绍静态类成员。只能使用这种技术声明值为整数或枚举的静态常量，而不能存储 double 常量。

10.8　抽象数据类型

Stock 类非常具体。不过，程序员常常通过定义类来表示更通用的概念。例如，就实现计算机专家们所说的抽象数据类型（abstract data type，ADT）而言，使用类是一种非常好的方式。顾名思义，ADT 以通用的方式描述数据类型，而没有引入语言或实现细节。例如，通过使用堆栈，可以以这样的方式存储数据，即总是从堆顶添加或删除数据。例如，C++程序使用堆栈来管理自动变量。当新的自动变量被生成后，它们被添加到堆顶；消亡时，从堆栈中删除它们。

下面简要地介绍一下堆栈的特征。首先，堆栈存储了多个数据项（该特征使得堆栈成为一个容器——一种更为通用的抽象）；其次，堆栈由可对它执行的操作来描述。

- 可创建空堆栈。
- 可将数据项添加到堆顶（压入）。
- 可从栈顶删除数据项（弹出）。
- 可查看堆栈是否填满。
- 可查看堆栈是否为空。

可以将上述描述转换为一个类声明，其中公有成员函数提供了表示堆栈操作的接口，而私有数据成员负责存储堆栈数据。类概念非常适合于 ADT 方法。

私有部分必须表明数据存储的方式。例如，可以使用常规数组、动态分配数组或更高级的数据结构（如链表）。不过，公有接口应隐藏数据表示，而以通用的术语来表达，如创建堆栈、压入等。程序清单 10.10 演示了一种方法，它假设系统实现了 bool 类型。如果读者使用的系统没有实现，可以使用 int、0 和 1 代替

bool、false 和 true。

程序清单 10.10 stack.h

```
// stack.h -- class definition for the stack ADT
#ifndef STACK_H_
#define STACK_H_

typedef unsigned long Item;

class Stack
{
private:
    enum {MAX = 10};   // constant specific to class
    Item items[MAX];   // holds stack items
    int top;           // index for top stack item
public:
    Stack();
    bool isempty()const;
    bool isfull()const;
    // push()returns false if stack already is full, true otherwise
    bool push (const Item & item);    // add item to stack
    // pop()returns false if stack already is empty, true otherwise
    bool pop (Item & item);           // pop top into item
};
#endif
```

注意：如果所用的系统没有实现 bool 类型，则可以使用 int、0 和 1 代替 bool、false 和 true。另外，系统可能支持早期的、非标准格式，如 boolean 或 Boolean。

在程序清单 10.10 所示的范例中，私有部分表明，堆栈是使用数组实现的；而公有部分隐藏了这一点。因此，可以使用动态数组来代替数组，而不会改变类的接口。这意味着修改堆栈的实现后，不需要重新编写使用堆栈的程序，而只需重新编译堆栈代码，并将其与已有的程序代码链接起来即可。

接口是冗余的，因为 pop()和 push()返回有关堆栈状态的信息（满或空），而不是 void 类型。在如何处理超出堆栈限制或者清空堆栈方面，这为程序员提供了两种选择。他可以在修改堆栈前使用 isempty()和 isfull()来查看，也可以使用 push()和 pop()的返回值来确定操作是否成功。

这个类不是根据特定的类型来定义堆栈，而是根据通用的 Item 类型来描述。在这个例子中，头文件使用 typedef 用 Item 代替 unsigned long。如果需要 double 堆栈或结构类型的堆栈，则只需修改 typedef 语句，而类声明和方法定义保持不变。类模板（参见第 14 章）提供了功能更强大的方法，来将存储的数据类型与类设计隔离开来。

接下来需要实现类方法，程序清单 10.11 提供了一种可行的实现。

程序清单 10.11 stack.cpp

```
// stack.cpp -- Stack member functions
#include "stack.h"
Stack::Stack()  // create an empty stack
{
    top = 0;
}

bool Stack::isempty()const
{
    return top == 0;
}

bool Stack::isfull()const
{
```

```
    return top == MAX;
}

bool Stack::push (const Item & item)
{
    if (top < MAX)
    {
        items[top++] = item;
        return true;
    }
    else
        return false;
}

bool Stack::pop (Item & item)
{
    if (top > 0)
    {
        item = items[--top];
        return true;
    }
    else
        return false;
}
```

默认构造函数确保所有堆栈被创建时都为空。pop()和 push()的代码确保栈顶被正确地处理。这种保证措施是 OOP 更可靠的原因之一。假设要创建一个独立数组来表示堆栈，创建一个独立变量来表示栈顶索引。则每次创建新堆栈时，都必须确保代码是正确的。没有私有数据提供的保护，则很可能由于无意修改了数据而导致程序出现非常严重的故障。

下面来测试该堆栈。程序清单 10.12 模拟了售货员的行为——使用堆栈的后进先出方式，从购物筐的最上面开始处理购物订单。

程序清单 10.12　stacker.cpp

```
// stacker.cpp -- testing the Stack class
#include <iostream>
#include <cctype>  // or ctype.h
#include "stack.h"
int main()
{
    using namespace std;
    Stack st; // create an empty stack
    char ch;
    unsigned long po;
    cout << "Please enter A to add a purchase order,\n"
        << "P to process a PO, or Q to quit.\n";
    while (cin >> ch && toupper(ch) != 'Q')
    {
        while (cin.get()!='\n')
            continue;
        if (!isalpha (ch))
        {
            cout << '\a';
            continue;
        }
        switch (ch)
        {
            case 'A':
            case 'a': cout << "Enter a PO number to add: ";
                cin >> po;
```

```
                if (st.isfull())
                    cout << "stack already full\n";
                else
                    st.push (po);
                break;
            case 'P':
            case 'p': if (st.isempty())
                    cout << "stack already empty\n";
                else {
                    st.pop (po);
                    cout << "PO #" << po << " popped\n";
                }
                break;
        }
        cout << "Please enter A to add a purchase order, \n"
           << "P to process a PO, or Q to quit.\n";
    }
    cout << "Bye\n";
    return 0;
}
```

程序清单 10.12 中的 while 循环删除输入行中剩余部分，就现在而言这并非是必不可少的，但它使程序的修改更方便（第 14 章将对这个程序进行修改）。下面是该程序的运行情况：

```
Please enter A to add a purchase order,
P to process a PO, or Q to quit.
A
Enter a PO number to add: 17885
Please enter A to add a purchase order,
P to process a PO, or Q to quit.
P
PO #17885 popped
Please enter A to add a purchase order,
P to process a PO, or Q to quit.
A
Enter a PO number to add: 17965
Please enter A to add a purchase order,
P to process a PO, or Q to quit.
A
Enter a PO number to add: 18002
Please enter A to add a purchase order,
P to process a PO, or Q to quit.
P
PO #18002 popped
Please enter A to add a purchase order,
P to process a PO, or Q to quit.
P
PO #17965 popped
Please enter A to add a purchase order,
P to process a PO, or Q to quit.
P
stack already empty
Please enter A to add a purchase order,
P to process a PO, or Q to quit.
Q
Bye
```

仔细选择数据类型，使类最小

在设计类时，应认真考虑类成员的数据类型。贸然使用非标准或依赖于平台的数据类型将使类急剧增大，从而增加所需的内存或程序的工作量。这种做法既低效又非常糟糕。

有一个经典范例说明了这一点，它使用非标准的 BOOL typedef，而不是标准 bool 数据类型。请看这些简单的类：

```
typedef int BOOL;
class BadClassDesign
{
   BOOL m_b1;
   BOOL m_b2;
   BOOL m_b3;
   BOOL m_b4;
   BOOL m_b5;
   BOOL m_b6;
};

class GoodClassDesign
{
   bool m_b1;
   bool m_b2;
   bool m_b3;
   bool m_b4;
   bool m_b5;
   bool m_b6;
};
```

与 bool（在多数平台上通常只占 1 个字节）不同的是，每个 BOOL 通常都将占据 4 个字节。如果类成员的用途是管理 Boolean 值（true 或 false），则实际上只需要 1 个字节。在典型的 Intel 平台机器中，类 BadClassDesign 占据 24 个字节，而类 ClassGoodDesign 仅占用 6 个字节。节省了 400%！。

应尽量使用正确的数据类型，使程序所需的内存量最小。

10.9　总结

面向对象编程强调的是程序如何表示数据。使用 OOP 方法解决编程问题的第一步是根据它与程序之间的接口来描述数据，从而指定如何使用数据。然后，设计一个类来实现该接口。一般来说，私有数据成员存储信息，公有成员函数（又称为方法）提供访问数据的惟一途径。类将数据和方法组合成一个单元，其私有性实现数据隐藏。

通常，将类声明分成两部分组成，这两部分通常保存在不同的文件中。类声明（包括由函数原型表示的方法）应放到头文件中。定义成员函数的源代码放在方法文件中。这样便将接口描述与实现细节分开了。从理论上说，只需知道公有接口就可以使用类。当然，可以查看实现方法（除非只提供了编译形式），但程序不应依赖于其实现细节，如知道某个值被存储为 int。只要程序和类只通过定义接口的方法进行通信，程序员就可以随意地对任何部分做独立的改进，而不必担心这样做会导致意外的不良影响。

类是用户定义的类型，对象是类的实例。这意味着对象是这种类型的变量，例如由 new 按类描述分配的内存。C++试图让用户定义的类型尽可能与标准类型类似，因此可以声明对象、指向对象的指针和对象数组。可以按值传递对象、将对象作为函数返回值、将一个对象赋给同类型的另一个对象。如果提供了构造函数，则在创建对象时，可以初始化对象。如果提供了析构函数方法，则在对象消亡后，程序将执行该函数。

每个对象都存储自己的数据，而共享类方法。如果 mr_object 是对象名，try_me() 是成员函数，则可以使用成员操作符句点调用成员函数：mr_object.try_me()。在 OOP 中，这种函数调用被称为将 try_me 消息发送给 mr_object 对象。在 try_me() 方法中引用类数据成员时，将使用 mr_object 对象相应的数据成员。同样，函数调用 i_object.try_me() 将访问 i_object 对象的数据成员。

如果希望成员函数对多个对象进行操作，可以将额外的对象作为参数传递给它。如果方法需要显式地引用调用它的对象，则可以使用 this 指针。由于 this 指针被设置为调用对象的地址，因此*this 是该对象的

别名。

类很适合用于描述 ADT。公有成员函数接口提供了 ADT 描述的服务，类的私有部分和类方法的代码提供了实现，这些实现对类的客户隐藏。

10.10 复习题

1. 什么是类？
2. 类如何实现抽象、封装和数据隐藏？
3. 对象和类之间的关系是什么？
4. 除了是函数之外，类函数成员与类数据成员之间的区别是什么？
5. 定义一个类来表示银行账户。数据成员包括储户姓名、账号（使用字符串）和存款。成员函数执行如下操作：
   - 创建对象，并将其初始化。
   - 显示储户姓名、账号和存款。
   - 存入参数指定的存款。
   - 取出参数指定的款项。

   请提供类声明，而不用给出方法实现（编程练习 1 将要求编写实现）。
6. 类构造函数在何时被调用？类析构函数呢？
7. 给出复习题 5 中的银行账户类的构造函数的代码。
8. 什么是默认构造函数，拥有默认构造函数有何好处？
9. 修改 Stock 类（stock2.h 中的版本），使之包含返回各个数据成员值的成员函数。注意：返回公司名的成员函数不应为修改数组提供便利，也就是说，不能仅仅返回 char *。它可以返回 const 指针，或返回一个指向 new 生成的数组的拷贝的指针。
10. this 和*this 是什么？

10.11 编程练习

1. 为复习题 5 描述的类提供方法定义，并编写一个小程序来演示所有的特性。
2. 下面是一个非常简单的类定义：

```
class Person {
private:
   static const LIMIT = 25;
   string lname;       // Person's last name
   char fname[LIMIT];  // Person's first name
public:
   Person() {lname = ""; fname[0] = '\0';
   Person(const string & ln, const char * fn = "Heyyou");   // #2
// the following methods display lname and fname
   void Show() const;        // firstname lastname format
   void FormalShow() const;  // lastname, firstname format
};
```

通过提供未定义的方法的代码来完成这个类的实现。然后，编写一个使用这个类的程序，它使用了三种可能的构造函数调用（没有参数、一个参数和两个参数）以及两种显示方法。下面是一个使用这些构造函数和方法的例子：

```
Person one;                    // use default constructor
Person two("Smythecraft");     // use #2 with one default argument
```

```
Person three("Dimwiddy", "Sam");  // use #2, no defaults
one.Show();
cout << endl;
one.FormalShow();
// etc. for two and three
```

3. 完成第 9 章的编程练习 1，但要用正确的 golf 类声明替换那里的代码。用带合适参数的构造函数替换 setgolf（golf &，const char *，int），以提供初始值。保留 setgolf() 的交互版本，但要用构造函数来实现它（例如，setgolf() 的代码应该获得数据，将数据传递给构造函数来创建一个临时对象，并将其赋给调用对象，即*this）。

4. 完成第 9 章的编程练习 4，但将 Sales 结构及相关的函数转换为一个类及其方法。用构造函数替换 setSales（sales &，double []，int）函数。用构造函数实现 setSales（Sales &）方法的交互版本。将类保留在名称空间 SALES 中。

5. 考虑下面的结构声明：

```
struct customer {
    char fullname[35];
    double payment;
};
```

编写一个程序，它从堆栈中添加和删除 customer 结构（堆栈用 Stack 类声明表示）。每次 customer 结构被删除时，其 payment 的值都被加入到总数中，并报告总数。注意：应该可以直接使用 Stack 类而不作修改；只需修改 typedef 声明，使 Item 的类型为 customer，而不是 unsigned long 即可。

6. 下面是一个类声明：

```
class Move
{
private:
    double x;
    double y;
public:
    Move (double a = 0, double b = 0); // sets x, y to a, b
    showmove()const;                   // shows current x, y values
    Move add (const Move & m) const;
// this function adds x of m to x of invoking object to get new x,
// adds y of m to y of invoking object to get new y, creates a new
// move object initialized to new x, y values and returns it
    reset (double a = 0, double b = 0);  // resets x, y to a, b
};
```

请提供成员函数的定义和测试这个类的程序。

7. Betelgeusean plorg 有这些特征：

数据：

- plorg 的名称不超过 19 个字符。
- plorg 有满意指数（CI），这是一个整数。

操作：

- 新的 plorg 将有名称，其 CI 值为 50。
- plorg 的 CI 可以修改。
- plorg 可以报告它的名称和 CI。
- plorg 的默认名称为“Plorga”。

请编写一个 Plorg 类声明（包括数据成员和成员函数原型）来表示 plorg，并编写成员函数的函数定义。然后编写一个小程序，以演示 Plorg 类的所有特性。

8. 可以将简单列表描述成下面这样：

- 可存储 0 或多个某种类型的列表。
- 可创建空列表。
- 可在列表中添加数据项。

- 可确定列表是否为空。
- 可确定列表是否为满。
- 可访问列表中的每一个数据项，并对它执行某种操作。

可以看到，这个列表确实很简单，例如，它不允许插入或删除数据项。

请设计一个 List 类来表示这种抽象类型。您应提供头文件 list.h 和实现文件 list.cpp，前者包含类定义，后者包含类方法的实现。您还应创建一个简短的程序来使用这个类。

可以选择使用数组或链表来实现该列表，但公有接口不应依赖于所做的选择。也就是说，公有接口不应有数组索引、节点指针等。应使用通用概念来表达创建列表、在列表中添加数据项等操作。对于访问数据项以及执行操作，通常应使用将函数指针作为参数的函数来处理：

```
void visit(void(*pf)(Item &));
```

其中，pf 指向一个将 Item 引用作为参数的函数（不是成员函数），Item 是列表中数据项的类型。visit() 函数将该函数用于列表中的每个数据项。

第 11 章　使用类

本章内容包括：

- 操作符重载。
- 友元函数。
- 重载<<操作符，以便用于输出。
- 状态成员。
- 使用 rand() 生成随机值。
- 类的自动转换和强制类型转换。
- 类转换函数。

C++类特性丰富、复杂、功能强大。在第 9 章，读者通过学习定义和使用简单的类，已踏上了面向对象编程之旅。通过定义用于表示对象的数据的类型以及（通过成员函数）定义可对数据执行的操作，读者知道了类是如何定义数据类型的。我们还学习了两个特殊的成员函数——构造函数和析构函数，其作用是管理类对象的创建和删除。本章将进一步探讨类的特征，重点是类设计技术，而不是通用原理。读者可能发现，本章介绍的一些特性很容易，而另一些很微妙。要更好地理解这些新特性，应使用这些范例进行练习。如果函数使用常规参数而不是引用参数，将发生什么情况呢？如果忽略了析构函数，又将发生什么情况呢？不要害怕犯错误，因为在解决问题的过程中学到的知识，比生搬硬套而不犯错误时要多得多（不过，不要认为所有的错误就都会让人增长见识）。这样，读者将更全面地了解 C++是如何工作的以及它可以为我们做哪些工作。

本章首先介绍操作符重载，它允许将标准 C++操作符（如=和+）用于类对象。然后介绍友元，这种 C++机制使得非成员函数可以访问私有数据。最后介绍如何命令 C++对类执行自动类型转换。学习本章和第 12 章后，读者将对类构造函数和类析构函数所起的作用有更深入的了解。另外，读者还将知道开发和改进类设计时，需要执行的步骤。

学习 C++的难点之一是需要记住大量的东西，但在拥有丰富的实践经验之前，根本不可能全部记住这些东西。从这种意义上说，学习 C++就像学习功能复杂的字处理程序或电子制表程序一样。任何特性都不可怕，但多数人只掌握了那些经常使用的特性，如查找文本或设置为斜体等。读者可能在那里曾经学过如何生成替换字符或者创建目录，除非经常使用它们，否则这些技能可能根本与日常工作无关。也许，学习本章知识的最好方法是，在我们自己开发的 C++程序中使用其中的新特性。对这些新特性有了充分的认识后，就可以添加其他 C++特性了。正如 Bjarne Stroustrup（C++的创建者）在一次 C++专业程序员大会上所建议的：“轻松地使用这种语言。不要觉得必须使用所有的特性，不要在第一次学习时就试图使用所有的特性”。

11.1　操作符重载

下面介绍一种使对象操作更美观的技术。操作符重载（operator overloading）是一种形式的 C++多态。

第 8 章介绍了 C++是如何使用户能够定义多个名称相同但特征标（参数列表）不同的函数的。这被称为函数重载（function overloading）或函数多态（functional polymorphism），旨在让您能够用同名的函数来完成相同的基本操作，即使这种操作被用于不同的数据类型（想象一下，如果必须对不同的物体使用不同的动词，例如抬起左脚（lift_lft），拿起汤匙（lift_sp），英语将会多么笨拙）。操作符重载将重载的概念扩展到操作符上，允许赋予 C++操作符多种含义。实际上，很多 C++（也包括 C 语言）操作符已经被重载。例如，将*操作符用于地址，将得到存储在这个地址中的值；但将它用于两个数字时，得到的将是它们的乘积。C++根据操作数的数目和类型来决定采用哪种操作。

C++允许将操作符重载扩展到用户定义的类型，例如，允许使用+将两个对象相加。编译器将根据操作数的数目和类型决定使用哪种加法定义。重载操作符可使代码看起来更自然。例如，将两个数组相加是一种常见的运算。通常，需要使用下面这样的 for 循环来实现：

```
for (int i = 0; i < 20; i++)
    evening[i] = sam[i] + janet[i];  // add element by element
```

但在 C++中，可以定义一个表示数组的类，并重载+操作符。于是便可以有这样的语句：

```
evening = sam + janet;              // add two array objects
```

这种简单的加法表示法隐藏了内部机理，并强调了实质，这是 OOP 的另一个目标。

要重载操作符，需使用被称为操作符函数的特殊函数形式。操作符函数的格式如下：

```
operator op (argument-list)
```

其中，*op* 是将要重载的操作符。例如，operator +()重载+操作符，operator *()重载*操作符。*op* 必须是有效的 C++操作符，不能虚构一个新的符号。例如，不能有 operator@()这样的函数，因为 C++中没有@操作符。但是，operator [ ]()函数将重载[ ]操作符，因为[ ]是数组索引操作符。例如，假设有一个 Salesperson 类，并为它定义了一个 operator +()成员函数，以重载+操作符，以便能够将两个 Saleperson 对象的销售额相加，则如果 district2、sid 和 sara 都是 Salesperson 类对象，便可以编写这样的等式：

```
district2 = sid + sara;
```

编译器发现，操作数是 Salesperson 类对象，因此使用相应的操作符函数替换上述操作符：

```
district2 = sid.operator+ (sara);
```

然后该函数将隐式地使用 sid（因为它调用了方法），而显式地使用 sara 对象（因为它被作为参数传递），来计算总和，并返回这个值。当然最重要的是，可以使用简便的+操作符表示法，而不必使用笨拙的函数表示法。

虽然 C++对操作符重载做了一些限制，但了解重载的工作方式后，这些限制就很容易理解了。因此，下面首先通过一些范例对操作符重载进行阐述，然后再讨论这些限制。

11.2 计算时间：一个操作符重载范例

如果今天早上在 Priggs 的账户上花费了 2 小时 35 分钟，下午又花费了 2 小时 40 分钟，则总共花了多少时间呢？这个范例与加法概念很吻合，但要相加的单位（小时与分钟的混合）与内置类型不匹配。第 7 章通过定义一个 travel_time 结构和将这种结构相加的 sum()函数来处理类似的情况。现在我们将其推广，采用一个使用方法来处理加法的 Time 类。我们首先使用一个名为 Sum()的常规方法，然后介绍如何将其转换为一个重载操作符。程序清单 11.1 列出了这个类的声明。

程序清单 11.1 mytime0.h

```
// mytime0.h -- Time class before operator overloading
#ifndef MYTIME0_H_
#define MYTIME0_H_
class Time
{
private:
    int hours;
    int minutes;
```

```
public:
    Time();
    Time (int h, int m = 0);
    void AddMin (int m);
    void AddHr (int h);
    void Reset (int h = 0, int m = 0);
    Time Sum (const Time & t) const;
    void Show()const;
};
#endif
```

Time 类提供了用于调整和重新设置时间、显示时间并将两个时间相加的方法。程序清单 11.2 列出了方法定义。请注意，当总的分钟数超过 59 时，AddMin()和 Sum()方法是如何使用整数除法和求模操作符来调整 minutes 和 hours 值的。另外，由于这里只使用了 iostream 特性 cout，且只使用了一次，因此使用 std::cout 比导入整个名称空间更经济。

程序清单 11.2　mytime0.cpp

```
// mytime0.cpp  -- implement Time methods
#include <iostream>
#include "mytime0.h"

Time::Time()
{
    hours = minutes = 0;
}

Time::Time (int h, int m)
{
    hours = h;
    minutes = m;
}

void Time::AddMin (int m)
{
    minutes += m;
    hours += minutes / 60;
    minutes %= 60;
}
void Time::AddHr (int h)
{
    hours += h;
}

void Time::Reset (int h, int m)
{
    hours = h;
    minutes = m;
}

Time Time::Sum (const Time & t) const
{
    Time sum;
    sum.minutes = minutes + t.minutes;
    sum.hours = hours + t.hours + sum.minutes / 60;
    sum.minutes %= 60;
    return sum;
}

void Time::Show()const
```

```
{
    std::cout << hours << " hours, " << minutes << " minutes";
}
```

来看一下 Sum() 函数的代码。注意参数是引用，但返回类型却不是引用。将参数声明为引用的目的是为了提高效率。如果按值传递 Time 对象，代码的功能将相同，但传递引用，速度将更快，使用的内存将更少。

不过，返回值不能是引用。因为函数将创建一个新的 Time 对象（sum），来表示另外两个 Time 对象的和。返回对象（如代码所做的那样）将创建对象的拷贝，而调用函数可以使用它。不过，如果返回类型为 Time &，则引用的将是 sum 对象。但由于 sum 对象是局部变量，在函数结束时将被删除，因此引用将指向一个不存在的对象。然而，使用 Time 返回类型意味着程序将在删除 sum 之前构造它的拷贝，调用函数将得到该拷贝。

警告：不要返回指向局部变量或临时对象的引用。函数执行完毕后，局部变量和临时对象将消失，引用将指向不存在的数据。

最后，程序清单 11.3 对 Time 类中计算时间总和的部分进行了测试。

程序清单 11.3 usetime0.cpp

```
// usetime0.cpp - using the first draft of Time class
// compile usetime0.cpp and mytime0.cpp together
#include <iostream>
#include "mytime0.h"

int main()
{
    using std::cout;
    using std::endl;
    Time planning;
    Time coding(2, 40);
    Time fixing(5, 55);
    Time total;

    cout << "planning time = ";
    planning.Show();
    cout << endl;

    cout << "coding time = ";
    coding.Show();
    cout << endl;

    cout << "fixing time = ";
    fixing.Show();
    cout << endl;

    total = coding.Sum(fixing);
    cout << "coding.Sum(fixing) = ";
    total.Show();
    cout << endl;

    return 0;
}
```

下面是程序清单 11.1、11.2 和 11.3 组成的程序的输出：

```
planning time = 0 hours, 0 minutes
coding time = 2 hours, 40 minutes
fixing time = 5 hours, 55 minutes
coding.Sum(fixing) = 8 hours, 35 minutes
```

11.2.1　添加加法操作符

将 Time 类转换为重载的加法操作符很容易，只要将 Sum() 的名称改为 operator +() 即可。这样做是对的，只要把操作符（这里为+）放到 operator 的后面，并将结果用作方法名即可。在这里，可以在标识符中使用字母、数字或下划线之外的其他字符。程序清单 11.4 和 11.5 反映了这些细微的修改。

程序清单 11.4　mytime1.h

```
// mytime1.h -- Time class after operator overloading
#ifndef MYTIME1_H_
#define MYTIME1_H_

class Time
{
private:
    int hours;
    int minutes;
public:
    Time();
    Time (int h, int m = 0);
    void AddMin (int m);
    void AddHr (int h);
    void Reset (int h = 0, int m = 0);
    Time operator+ (const Time & t) const;
    void Show()const;
};
#endif
```

程序清单 11.5　mytime1.cpp

```
// mytime1.cpp -- implementing Time methods
#include <iostream>
#include "mytime1.h"

Time::Time()
{
    hours = minutes = 0;
}

Time::Time (int h, int m)
{
    hours = h;
    minutes = m;
}

void Time::AddMin (int m)
{
    minutes += m;
    hours += minutes / 60;
    minutes %= 60;
}
void Time::AddHr (int h)
{
    hours += h;
}

void Time::Reset (int h, int m)
{
    hours = h;
    minutes = m;
```

```
}

Time Time::operator+ (const Time & t) const
{
    Time sum;
    sum.minutes = minutes + t.minutes;
    sum.hours = hours + t.hours + sum.minutes / 60;
    sum.minutes %= 60;
    return sum;
}

void Time::Show()const
{
    std::cout << hours << " hours, " << minutes << " minutes";
}
```

和 Sum() 一样，operator +()也是由 Time 对象调用的，它将第二个 Time 对象作为参数，并返回一个 Time 对象。因此，可以像调用 Sum()那样来调用 operator +()方法：

```
total = coding.operator+(fixing);    // function notation
```

不过将该方法命令为 operator +()后，也可以使用操作符表示法：

```
total = coding + fixing;             // operator notation
```

这两种表示法都将调用 operator +()方法。注意，在操作符表示法中，操作符左侧的对象（这里为 coding）是调用对象，操作符右边的对象（这里为 fixing）是作为参数被传递的对象。程序清单 11.6 说明了这一点。

程序清单 11.6 usetime1.cpp

```
// usetime1.cpp -- using the second draft of the Time class
// compile usetime1.cpp and mytime1.cpp together
#include <iostream>
#include "mytime1.h"

int main()
{
    using std::cout;
    using std::endl;
    Time planning;
    Time coding(2, 40);
    Time fixing(5, 55);
    Time total;

    cout << "planning time = ";
    planning.Show();
    cout << endl;

    cout << "coding time = ";
    coding.Show();
    cout << endl;

    cout << "fixing time = ";
    fixing.Show();
    cout << endl;

    total = coding + fixing;
    // operator notation
    cout << "coding + fixing = ";
    total.Show();
    cout << endl;

    Time morefixing(3, 28);
    cout << "more fixing time = ";
```

```
    morefixing.Show();
    cout << endl;
    total = morefixing.operator+(total);
    // function notation
    cout << "morefixing.operator+(total) = ";
    total.Show();
    cout << endl;

    return 0;
}
```

下面是程序清单 11.4～11.6 组成的程序的输出：

```
planning time = 0 hours, 0 minutes
coding time = 2 hours, 40 minutes
fixing time = 5 hours, 55 minutes
coding + fixing = 8 hours, 35 minutes
more fixing time = 3 hours, 28 minutes
morefixing.operator+(total) = 12 hours, 3 minutes
```

简而言之，operator+()函数的名称使得可以使用函数表示法或操作符表示法来调用它。编译器将根据操作数的类型来确定如何做：

```
int a, b, c;
Time A, B, C;
c = a + b;      // use int addition
C = A + B;      // use addition as defined for Time objects
```

可以将两个以上的对象相加吗？例如，如果 t1、t2、t3 和 t4 都是 Time 对象，可以这样做吗：

```
t4 = t1 + t2 + t3;                  // valid?
```

为回答这个问题，来看一些上述语句将被如何转换为函数调用。由于+是从左向右结合的操作符，因此上述语句首先被转换成下面这样：

```
t4 = t1.operator+(t2 + t3);          // valid?
```

然后，函数参数本身被转换成一个函数调用，结果如下：

```
t4 = t1.operator+(t2.operator+(t3));   // valid? YES
```

上述语句合法吗？是的。函数调用 t2.operator+(t3)返回一个 Time 对象，后者是 t2 和 t3 的和。然而，该对象成为函数调用 t1.operator+()的参数，该调用返回 t1 与表示 t2 和 t3 之和的 Time 对象的和。总之，最后的返回值为 t1、t2 和 t3 之和，这正是我们期望的。

11.2.2　重载限制

多数 C++操作符（参见表 11.1）都可以用这样的方式重载。重载的操作符（有些例外情况）不必是成员函数，但必须至少有一个操作数是用户定义的类型。下面详细介绍 C++对用户定义的操作符重载的限制：

1．重载后的操作符必须至少有一个操作数是用户定义的类型，这将防止用户为标准类型重载操作符。因此，不能将减法操作符（-）重载为计算两个 double 值的和，而不是它们的差。虽然这种限制将对创造性有所影响，但可以确保程序正常运行。

2．使用操作符时不能违反操作符原来的句法规则。例如，不能将求模操作符（%）重载成使用一个操作数：

```
int x;
Time shiva;
% x;      // invalid for modulus operator
% shiva;  // invalid for overloaded operator
```

同样，不能修改操作符的优先级。因此，如果将加号操作符重载成将两个类相加，则新的操作符与原来的加号具有相同的优先级。

3．不能定义新的操作符。例如，不能定义 operator **()函数来表示求幂。

4．不能重载下面的操作符：

- sizeof——sizeof 操作符。

- . ——成员操作符。
- .* ——成员指针操作符。
- :: ——作用域解析操作符。
- ?: ——条件操作符。
- typeid ——一个 RTTI 操作符。
- const_cast ——强制类型转换操作符。
- dynamic_cast ——强制类型转换操作符。
- reinterpret_cast ——强制类型转换操作符。
- static_cast ——强制类型转换操作符。

但是，表 11.1 中所有的操作符都可以被重载。

表 11.1　　可被重载的操作符

+	-	*	/	%	^
&	|	~	!	=	<
>	+=	-=	*=	/=	%=
^=	&=	|=	<<	>>	>>=
<<=	==	!=	<=	>=	&&
||	++	--	,	->*	->
()	[]	new	delete	new []	delete []

5. 表 11.1 中的大多数操作符都可以通过成员或非成员函数进行重载，但下面的操作符只能通过成员函数进行重载：

- = ——赋值操作符。
- () ——函数调用操作符。
- [ ] ——下标操作符。
- -> ——通过指针访问类成员的操作符。

注意：本章不介绍这里列出的所有操作符，但附录 E 对本书正文中没有介绍的操作符进行了总结。

除了这些正式限制之外，还应在重载操作符时遵循一些明智的限制。例如，不要将*操作符重载成交换两个 Time 对象的数据成员。表示法中没有任何内容可以表明操作符完成的工作，因此最好定义一个其名称具有说明性的类方法，如 Swap()。

11.2.3　其他重载操作符

还有一些其他的操作对 Time 类来说是有意义的。例如，可能要将两个时间相减或将时间乘以一个因子，这需要重载减法和乘法操作符。这和重载加法操作符采用的技术相同，即创建 operator-() 和 operator*() 方法。也就是说，将下面的原型添加到类声明中：

```
Time operator-(const Time & t) const;
Time operator*(double n) const;
```

程序清单 11.7 列出了新的头文件。

程序清单 11.7　mytime2.h

```
// mytime2.h -- Time class after operator overloading
#ifndef MYTIME2_H_
#define MYTIME2_H_

class Time
{
private:
    int hours;
```

```
    int minutes;
public:
    Time();
    Time(int h, int m = 0);
    void AddMin(int m);
    void AddHr(int h);
    void Reset(int h = 0, int m = 0);
    Time operator+(const Time & t) const;
    Time operator-(const Time & t) const;
    Time operator*(double n) const;
    void Show()const;
};
#endif.
```

然后将新增方法的定义添加到实现文件中，如程序清单 11.8 所示。

程序清单 11.8　mytime2.cpp

```
// mytime2.cpp  -- implementing Time methods
#include <iostream>
#include "mytime2.h"

Time::Time()
{
    hours = minutes = 0;
}

Time::Time(int h, int m)
{
    hours = h;
    minutes = m;
}

void Time::AddMin(int m)
{
    minutes += m;
    hours += minutes / 60;
    minutes %= 60;
}
void Time::AddHr(int h)
{
    hours += h;
}

void Time::Reset(int h, int m)
{
    hours = h;
    minutes = m;
}

Time Time::operator+(const Time & t) const
{
    Time sum;
    sum.minutes = minutes + t.minutes;
    sum.hours = hours + t.hours + sum.minutes / 60;
    sum.minutes %= 60;
    return sum;
}

Time Time::operator-(const Time & t) const
{
```

```
    Time diff;
    int tot1, tot2;
    tot1 = t.minutes + 60 * t.hours;
    tot2 = minutes + 60 * hours;
    diff.minutes = (tot2 - tot1) % 60;
    diff.hours = (tot2 - tot1) / 60;
    return diff;
}

Time Time::operator*(double mult) const
{
    Time result;
    long totalminutes = hours * mult * 60 + minutes * mult;
    result.hours = totalminutes / 60;
    result.minutes = totalminutes % 60;
    return result;
}

void Time::Show()const
{
    std::cout << hours << " hours, " << minutes << " minutes";
}
```

完成上述修改后，就可以使用程序清单 11.9 中的代码来测试新定义了。

程序清单 11.9 usetime2.cpp

```
// usetime2.cpp -- using the third draft of the Time class
// compile usetime2.cpp and mytime2.cpp together
#include <iostream>
#include "mytime2.h"

int main()
{
    using std::cout;
    using std::endl;
    Time weeding(4, 35);
    Time waxing(2, 47);
    Time total;
    Time diff;
    Time adjusted;

    cout << "weeding time = ";
    weeding.Show();
    cout << endl;

    cout << "waxing time = ";
    waxing.Show();
    cout << endl;

    cout << "total work time = ";
    total = weeding + waxing;       // use operator+()
    total.Show();
    cout << endl;

    diff = weeding - waxing;        // use operator-()
    cout << "weeding time - waxing time = ";
    diff.Show();
    cout << endl;

    adjusted = total * 1.5;      // use operator*()
```

```
    cout << "adjusted work time = ";
    adjusted.Show();
    cout << endl;

    return 0;
}
```

下面是程序清单 11.7～11.9 组成的程序的输出：

```
weeding time = 4 hours, 35 minutes
waxing time = 2 hours, 47 minutes
total work time = 7 hours, 22 minutes
weeding time - waxing time = 1 hours, 48 minutes
adjusted work time = 11 hours, 3 minutes
```

11.3　友元简介

读者知道，C++控制对类对象私有部分的访问。通常，公有类方法提供惟一的访问途径，但是有时候这种限制太严格，以致于不适合特定的编程问题。在这种情况下，C++提供了另外一种形式的访问权限：友元。友元有 3 种：

- 友元函数。
- 友元类。
- 友元成员函数。

通过让函数成为类的友元，可以赋予该函数与类的成员函数相同的访问权限。下面介绍友元函数，其他两种友元将在第 15 章介绍。

介绍如何成为友元之前，先介绍为何需要友元。在为类重载二元操作符时（带两个参数的操作符）常常需要友元。将 Time 对象乘以实数就属于这种情况，下面来看看。

在 Time 类范例中，重载的乘法操作符与其他两种重载操作符的差别在于，它使用了两种不同的类型。也就是说，加法和减法操作符都结合两个 Time 值，而乘法操作符将一个 Time 值与一个 double 值结合在一起。这限制了该操作符的使用方式。记住，左侧的操作数是调用对象。也就是说，下面的语句：

```
A = B * 2.75;
```

将被转换为下面的成员函数调用：

```
A = B.operator*(2.75);
```

但下面的语句又如何呢？

```
A = 2.75 * B;     // cannot correspond to a member function
```

从概念上说，2.75 * B 应与 B *2.75 相同，但第一个表达式不对应于成员函数，因为 2.75 不是 Time 类型的对象。记住，左侧的操作数应是调用对象，但 2.75 不是对象。因此，编译器不能使用成员函数调用来替换该表达式。

解决这个难题的一种方式是，告知每个人（包括程序员自己），只能按 B * 2.75 这种格式编写，不能写成 2.75 * B。这是一种对服务器友好-客户警惕的（server-friendly, client-beware）解决方案，与 OOP 无关。

不过，还有另外一种解决方式——非成员函数（记住，大多数操作符都可以通过成员或非成员函数来重载）。非成员函数不是由对象调用的，它使用的所有值（包括对象）都是显式参数。这样，编译器能够将下面的表达式：

```
A = 2.75 * B;     // cannot correspond to a member function
```

与下面的非成员函数调用匹配：

```
A = operator*(2.75, B);
```

该函数的原型如下：

```
Time operator*(double m, const Time & t);
```

对于非成员重载操作符函数来说，操作符表达式左边的操作数对应于操作符函数的第一个参数，操作

符表达式右边的操作数对应于操作符函数的第二个参数。而原来的成员函数则按相反的顺序处理操作数，也就是说，double 值乘以 Time 值。

使用非成员函数可以按所需的顺序获得操作数（先是 double，然后是 Time），但引发了一个新问题：非成员函数不能直接访问类的私有数据，至少常规非成员函数不能访问。但是，有一类特殊的非成员函数可以访问类的私有成员，它们被称为友元函数。

11.3.1 创建友元

创建友元函数的第一步是将其原型放在类声明中，并在原型声明前加上关键字 friend：

```
friend Time operator*(double m, const Time & t); // goes in class declaration
```

该原型意味着下面两点：

- 虽然 operator*()函数是在类声明中声明的，但它不是成员函数，因此不能使用成员操作符来调用。
- 虽然 operator*()函数不是成员函数，但它与成员函数的访问权限相同。

第二步是编写函数定义。因为它不是成员函数，所以不要使用 Time::限定符。另外，不要在定义中使用关键字 friend，定义应该如下：

```
Time operator*(double m, const Time & t) // friend not used in definition
{
    Time result;
    long totalminutes = t.hours * mult * 60 +t.minutes * mult;
    result.hours = totalminutes / 60;
    result.minutes = totalminutes % 60;
    return result;
}
```

有了上述声明和定义后，将下面的语句：

```
A = 2.75 * B;
```

转换为：

```
A = operator*(2.75, B);
```

从而调用刚才定义的非成员友元函数。

简而言之，类的友元函数是非成员函数，其访问权限与成员函数相同。

友元是否有悖于 OOP

乍一看，读者可能会认为友元违反了 OOP 数据隐藏的原则，因为友元机制允许非成员函数访问私有数据。不过，这个观点太片面了。相反，应将友元函数看作类的扩展接口的组成部分。例如，从概念上看，double 乘以 Time 和 Time 乘以 double 是完全相同的。也就是说，前一个要求有友元函数，后一个使用成员函数，这是 C++句法的结果，而不是概念上的差别。通过使用友元函数和类方法，可以用同一个用户接口表达这两种操作。另外请记住，只有类声明可以决定哪一个函数是友元，因此类声明仍然控制了哪些函数可以访问私有数据。简而言之，类方法和友元只是表达类接口的两种不同机制。

实际上，按下面的方式对定义进行修改（交换乘法操作数的顺序），可以将这个友元函数编写为非友元函数：

```
Time operator*(double m, const Time & t)
{
    return t * m;     // use a member function
}
```

原来的版本显式地访问 t.minutes 和 t.hours，所以它必须是友元。这个版本将 Time 对象 t 作为一个整体使用，让成员函数来处理私有值，因此不必是友元。不过，将该版本作为友元也是一个好主意。最重要的是，它将该作为正式类接口的组成部分。其次，如果以后发现需要函数直接访问私有数据，则只要修改函数定义即可，而不必修改类原型。

提示：如果要为类重载操作符，并将非类的项作为其第一个操作数，则可以用友元函数来反转操作数的顺序。

11.3.2　常用的友元：重载<<操作符

一个很有用的类特性是，可以对<<操作符进行重载，使之能与 cout 一起来显示对象的内容。与前面介绍的范例相比，这种重载要复杂些，因此我们分两步（而不是一步）来完成。

假设 trip 是一个 Time 对象。为显示 Time 的值，我们前面使用的是 Show()。不过，如果可以像下面这样操作会更好：

```
cout << trip;  // make cout recognize Time class?
```

之所以可以这样做，是因为<<是可被重载的 C++操作符之一。实际上，它已经被重载很多次了。最初，<<操作符是 C 和 C++的位操作符，将值中的位左移（参见附录 E）。ostream 类对该操作符进行了重载，将其转换为一个输出工具。前面讲过，cout 是一个 ostream 对象，它是智能的，能够识别所有的 C++基本类型。这是因为对于每种基本类型，ostream 类声明中都包含了相应的重载的 operator<<() 定义。也就是说，一个定义使用 int 参数，一个定义使用 double 参数，等等。因此，要使 cout 能够识别 Time 对象，一种方法是将一个新的函数操作符定义添加到 ostream 类声明中。但修改 iostream 文件是个危险的主意，这样做会在标准接口上浪费时间。更好的办法是，通过 Time 类声明来让 Time 类知道如何使用 cout。

1. <<的第一种重载版本

要使 Time 类知道使用 cout，必须使用友元函数。这是什么原因呢？因为下面这样的语句：

```
cout << trip;
```

使用两个对象，其中第一个是 ostream 类对象（cout）。如果使用一个 Time 成员函数来重载<<，Time 对象将是第一个操作数，就像使用成员函数重载*操作符那样。这意味着必须这样使用<<：

```
trip << cout;  // if operator<<()were a Time member function
```

这样会令人迷惑。但通过使用友元函数，可以像下面这样重载操作符：

```
void operator<<(ostream & os, const Time & t)
{
   os << t.hours << " hours, " << t.minutes << " minutes";
}
```

这样可以使用下面的语句：

```
cout << trip;
```

按下面这样的格式打印数据：

```
4 hours, 23 minutes
```

友元还是非友元

新的 Time 类声明使 operatro<<() 函数成为 Time 类的一个友元函数。但该函数不是 ostream 类的友元（尽管对 ostream 类并无害处）。operator<<() 函数接受一个 ostream 参数和一个 Time 参数，因此表面看来它必须同时是这两个类的友元。但是，看看函数代码就会发现，尽管该函数访问了 Time 对象的各个成员，但从始至终都将 ostream 对象作为一个整体使用。因为 operator<<() 直接访问 Time 对象的私有成员，所以它必须是 Time 类的友元。但由于它并不直接访问 ostream 对象的私有成员，所以并不一定必须是 ostream 类的友元。这很好，因为这就意味着不必修订 ostream 的定义。

注意，新的 operator<<() 定义使用 ostream 引用 os 作为它的第一个参数。通常情况下，os 引用 cout 对象，如表达式 cout << trip 所示。但也可以将这个操作符用于其他 ostream 对象，在这种情况下，os 将引用相应的对象。

提示：如果不知道还有其他 ostream 对象，请回忆第 10 章介绍的 cerr。另外，第 6 章介绍的 ofstream 对象可用于将输出写入到文件中。通过继承（参见第 13 章），ofstream 对象可以使用 ostream 的方法。这样，便可以用 operator<<() 定义来将 Time 的数据写入到文件和屏幕上，为此只需传递一个经过适当初始化的 ofstream 对象（而不是 cout 对象）。

调用 cout << trip 应使用 cout 对象本身，而不是它的拷贝，因此该函数按引用（而不是按值）来传递该

对象。这样，表达式 cout << trip 将导致 os 成为 cout 的一个别名；而表达式 cerr << trip 将导致 os 成为 cerr 的一个别名。Time 对象可以按值或按引用来传递，因为这两种形式都使函数能够使用对象的值。按引用传递使用的内存和时间都比按值传递少。

2. <<的第二种重载版本

前面介绍的实现存在一个问题。像下面这样的语句：

```
cout << trip;
```

可以正常工作，但将重新定义的<<操作符与 cout 一起使用时，不允许将输出拼接起来，而通常是可以拼接的：

```
cout << "Trip time: " << trip << " (Tuesday) \n"; // can't do
```

要理解这样做不可行的原因以及必须如何做才能使其可行，首先需要了解关于 cout 操作的一点知识。请看下面的语句：

```
int x = 5;
int y = 8;
cout << x << y;
```

C++从左至右读取输出语句，意味着它等同于：

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

正如 iosream 中定义的那样，<<操作符要求左边是一个 ostream 对象。显然，因为 cout 是 ostream 对象，所以表达式 cout << x 满足这种要求。但是，因为表达式 cout << x 位于<< y 的左侧，所以输出语句也要求该表达式是一个 ostream 类型的对象。因此，ostream 类将 operator<<()函数实现为返回一个 ostream 对象。具体地说，在这个例子中，它返回调用对象 cout。因此，表达式（cout << x）本身也是一个 ostream 对象，从而可以位于<<操作符的左侧。

可以对友元函数采用相同的方法。只要修改 operator<<()函数，让它返回 ostream 对象的引用即可：

```
ostream & operator<< (ostream & os, const Time & t)
{
   os << t.hours << " hours, " << t.minutes << " minutes";
   return os;
}
```

注意，返回类型是 ostream &。这意味着该函数返回 ostream 对象的引用。因为函数开始执行时，程序传递了一个对象引用给它，这样做的最终结果是，函数的返回值就是传递给它的对象。也就是说，下面的语句：

```
cout << trip;
```

将被转换为下面的调用：

```
operator<< (cout, trip);
```

而该调用返回 cout 对象。因此，下面的语句可以正常工作：

```
cout << "Trip time: " << trip << " (Tuesday) \n"; // can do
```

我们将这条语句分成多步，来看看它是如何工作的。首先，

```
cout << "Trip time: "
```

调用 ostream 中的<<定义，它显示字符串并返回 cout 对象。因此表达式 cout << “Trip time: ”将显示字符串，然后被它的返回值 cout 所替代。原来的语句被简化为下面的形式：

```
cout << trip << " (Tuesday) \n";
```

接下来，程序使用<<的 Time 声明显示 trip 值，并再次返回 cout 对象。这将语句简化为：

cout << "（Tuesday）\n";

现在，程序使用 ostream 中用于字符串的<<定义，来显示最后一个字符串，并结束运行。

有趣的是，这个 operator<<()版本还可用于将输出写入到文件中：

```
#include <fstream>
...
ofstream fout;
fout.open("savetime.txt");
Time trip(12, 40);
fout << trip;
```

其中最后一条语句将被转换为这样：

```
operator<<(fout, trip);
```

另外，正如第 8 章指出的，类继承属性让 ostream 引用能够指向 ostream 对象和 ofstream 对象。

提示：一般来说，要重载<<操作符来显示 c_name 的对象，可使用一个友元函数，其定义如下：

```
ostream & operator<< (ostream & os, const c_name & obj)
{
      os << ...;  // display object contents
      return os;
}
```

程序清单 11.10 列出了修改后的类定义，其中包括 operator*()和 operator<<()这两个友元函数。它将第一个友元函数作为内联函数，因为其代码很短（当定义同时也是原型时，就像这个例子中那样，要使用 friend 前缀）。

记住：只有在类声明中的原型中才能使用 friend 关键字。除非函数定义也是原型，否则不能在函数定义中使用该关键字。

程序清单 11.10　mytime3.h

```
// mytime3.h -- Time class with friends
#ifndef MYTIME3_H_
#define MYTIME3_H_
#include <iostream>

class Time
{
private:
   int hours;
   int minutes;
public:
   Time();
   Time (int h, int m = 0);
   void AddMin (int m);
   void AddHr (int h);
   void Reset (int h = 0, int m = 0);
   Time operator+ (const Time & t) const;
   Time operator- (const Time & t) const;
   Time operator* (double n) const;
   friend Time operator* (double m, const Time & t)
        { return t * m; }   // inline definition
   friend std::ostream & operator<< (std::ostream & os, const Time & t);
};
#endif
```

程序清单 11.11 列出了修改后的定义。方法使用了 Time::限定符，而友元函数不使用该限定符。另外，由于在 mytime3.h 中包含了 iostream 并提供了 using 声明 std::ostream，因此在 mytime3.cpp 中包含 mytime3.h 后，便提供了在实现文件中使用 ostream 的支持。

程序清单 11.11　mytime3.cpp

```
// mytime3.cpp  -- implementing Time methods
#include "mytime3.h"

Time::Time()
{
   hours = minutes = 0;
}

Time::Time (int h, int m)
{
   hours = h;
   minutes = m;
```

```
}

void Time::AddMin (int m)
{
    minutes += m;
    hours += minutes / 60;
    minutes %= 60;
}
void Time::AddHr (int h)
{
    hours += h;
}

void Time::Reset (int h, int m)
{
    hours = h;
    minutes = m;
}

Time Time::operator+ (const Time & t) const
{
    Time sum;
    sum.minutes = minutes + t.minutes;
    sum.hours = hours + t.hours + sum.minutes / 60;
    sum.minutes %= 60;
    return sum;
}

Time Time::operator- (const Time & t) const
{
    Time diff;
    int tot1, tot2;
    tot1 = t.minutes + 60 * t.hours;
    tot2 = minutes + 60 * hours;
    diff.minutes = (tot2 - tot1) % 60;
    diff.hours = (tot2 - tot1) / 60;
    return diff;
}

Time Time::operator* (double mult) const
{
    Time result;
    long totalminutes = hours * mult * 60 + minutes * mult;
    result.hours = totalminutes / 60;
    result.minutes = totalminutes % 60;
    return result;
}

std::ostream & operator<< (std::ostream & os, const Time & t)
{
    os << t.hours << " hours, " << t.minutes << " minutes";
    return os;
}
```

程序清单 11.12 是一个范例程序。从技术上说，在 usetime3.cpp 中不必包含头文件 iostream，因为在 mytime3.h 中已经包含了该文件。然而，作为 Time 类的用户，您并不知道在类代码文件中已经包含了哪些文件，因此您应负责将您编写的代码所需的头文件包含进来。

程序清单 11.12 usetime3.cpp

```
//usetime3.cpp -- using the fourth draft of the Time class
```

```
// compile usetime3.cpp and mytime3.cpp together
#include <iostream>
#include "mytime3.h"
int main()
{
    using std::cout;
    using std::endl;
    Time aida(3, 35);
    Time tosca(2, 48);
    Time temp;

    cout << "Aida and Tosca:\n";
    cout << aida<<"; " << tosca << endl;
    temp = aida + tosca;     // operator+()
    cout << "Aida + Tosca: " << temp << endl;
    temp = aida* 1.17;  // member operator*()
    cout << "Aida * 1.17: " << temp << endl;
    cout << "10 * Tosca: " << 10 * tosca << endl;

    return 0;
}
```

下面是程序清单 11.10～11.12 组成的程序的输出：

```
Aida and Tosca:
3 hours, 35 minutes; 2 hours, 48 minutes
Aida + Tosca: 6 hours, 23 minutes
Aida * 1.17: 4 hours, 11 minutes
10 * Tosca: 28 hours, 0 minutes
```

11.4　重载操作符：作为成员函数还是非成员函数

对于很多操作符来说，可以选择使用成员函数或非成员函数来实现操作符重载。一般来说，非成员函数应是友元函数，这样它才能直接访问类的私有数据。例如，Time 类的加法操作符在 Time 类声明中的原型如下：

```
Time operator+(const Time & t) const;                    // member version
```

这个类也可以使用下面的原型：

```
friend Time operator+(const Time & t1, const Time & t2); // non-member version
```

加法操作符需要两个操作数。对于成员函数版本来说，一个操作数通过 this 指针隐式地传递，另一个操作数作为函数参数显式地传递；对于友元版本来说，两个操作数都作为参数来传递。

记住：非成员版本的重载操作符函数所需的形参数目与操作符使用的操作数数目相同；而成员版本所需的参数数目少一个，因为其中的一个操作数是被隐式地传递的调用对象。

这两个原型都与表达式 T2 + T3 匹配，其中 T2 和 T3 都是 Time 类型对象。也就是说，编译器将下面的语句：

```
T1 = T2 + T3;
```

转换为下面两个的任何一个：

```
T1 = T2.operator+(T3);   // member function
T1 = operator+(T2, T3);  // nonmember function
```

记住，在定义操作符时，必须选择其中的一种格式，而不能同时选择这两种格式。因为这两种格式都与同一个表达式匹配，同时定义这两种格式将被视为二义性错误，导致编译错误。

那么哪种格式最好呢？对于某些操作符来说（如前所述），成员函数是惟一合法的选择。在其他情况下，这两种格式没有太大的区别。有时，根据类设计，使用非成员函数版本可能更好（尤其是为类定义类型转换时）。本章后面的“转换和友元”一节将更深入地讨论这种情形。

11.5 再谈重载：矢量类

下面介绍另一种使用了操作符重载和友元的类设计——一个表示矢量的类。这个类还说明了类设计的其他方面，例如，在同一个对象中包含两种描述同一样东西的不同方式等。即使并不关心矢量，也可以在其他情况下使用这里介绍的很多新技术。矢量（vector），是工程和物理中使用的一个术语，它是一个有大小和方向的量。例如，推东西时，推的效果将取决于推力的大小和推的方向。从某个方向推可能会省力，而从相反的方向推则要费很大的劲。为完整地描述汽车的运动情况，应指出其运动速度（大小）和运动方向；如果逆行，则向高速公路的巡警辩解没有超速、超载是徒劳的（免疫学家和计算机专家使用术语矢量的方式不同，请不要考虑这一点，至少在第 16 章介绍计算机科学版本——vector 模板类之前应如此）。下面的注释介绍了更多有关矢量的知识，不过对于下面的 C++范例来说，并不必完全理解这些知识。

矢 量

工蜂发现了一个非凡的花蜜储藏处，它匆忙返回蜂巢，告知其他蜜蜂，该花蜜储藏处离蜂巢 120 码。“这种信息是不完整的”，其他蜜蜂感到很茫然——“还必须告知方向!”，该工蜂答道：“日光方向偏北 30 度”。知道了距离（大小）和方向，其他的蜜蜂能很快找到蜜源。蜜蜂懂得矢量。

许多数量都有大小和方向。例如，推的效果取决于力气的大小和方向。在计算机屏幕上移动对象时也涉及到距离和方向。可以使用矢量来描述这类问题。例如，可以用矢量来描述如何在屏幕上移动（放置）对象，即用箭头从起始位置画到终止位置，来对它作形象化处理。矢量的长度是其大小——描述了移动的距离；箭头的指向描述了方向（参见图 11.1）。表示这种位置变化的矢量称为位移矢量（displacement vector）。

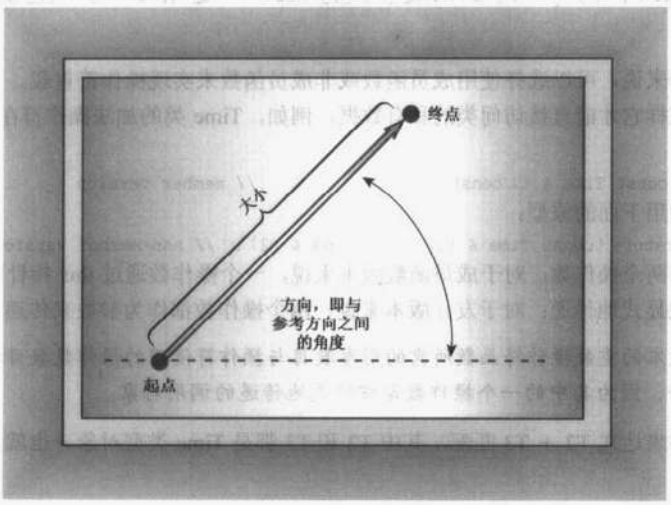


图 11.1 使用矢量描述位移

现在，假设您是 Lhanappa——伟大的毛象猎手。猎狗报告毛象群位于西北 14.1 公里处。但由于当时刮的是东南风，您不想从东南方向接近毛象群，因此先向西走了 10 公里，再向北走了 10 公里，最终从南面接近毛象群。您知道这两个位移矢量与指向西北的 14.1 公里的矢量的方向相同。伟大的毛象猎手 Lhanappa 也知道如何将两个矢量相加。

将两个矢量相加有一种简单的几何解释。首先，画一个矢量，然后从第一个矢量的尾部开始画第二个矢量。最后从第一个矢量的开始处向第二个矢量的结尾处画一个矢量。第三个矢量表示前两个矢量的和（参

见图 11.2）。注意，两个矢量之和的长度可能小于它们的长度之和。

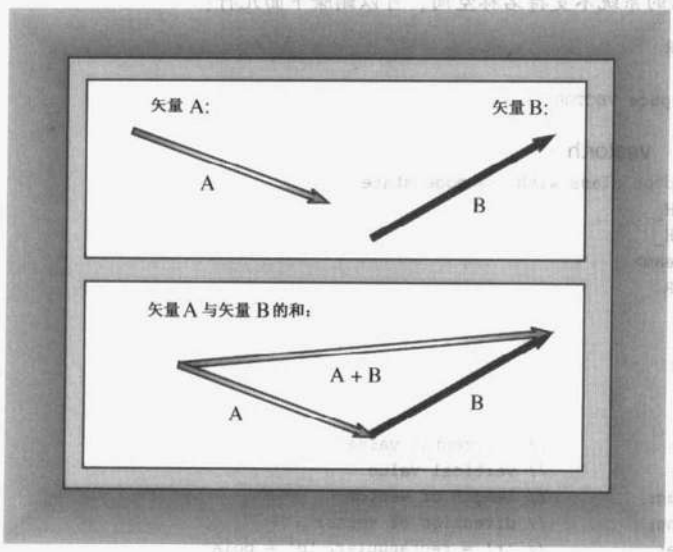


图 11.2　将两个矢量相加

显然，应为矢量操作符重载。首先，无法用一个数来表示矢量，因此应创建一个类来表示矢量。其次，矢量与普通数学运算（如加法、减法）有相似之外。这种相似表明，应重载操作符，使之能用于矢量。

出于简化的目的，本节将实现一个二维矢量（如屏幕位移），而不是三维矢量（如表示直升机或体操运动员的运动情况）。描述二维矢量只需两个数，但可以选择到底使用哪两个数：

- 可以用大小（长度）和方向（角度）描述矢量。
- 可以用分量 *x* 和 *y* 表示矢量。

两个分量分别是水平矢量（*x* 分量）和垂直矢量（*y* 分量），将其相加可以得到最终的矢量。例如，可以这样描述点的运动：向右移动 30 个单位，再向上移动 40 个单位（参见图 11.3）。这将把该点沿与水平方向呈 53.1 度的方向移动 50 个单位，因此，水平分量为 30 个单位、垂直分量为 40 个单位的矢量，与长度为 50 个单位、方向为 53.1 度的矢量相同。位移矢量指的是从何处开始、到何处结束，而不是经过的路线。这种表示基本上和第 7 章在直角坐标与极坐标之间转换的程序中介绍的相同。

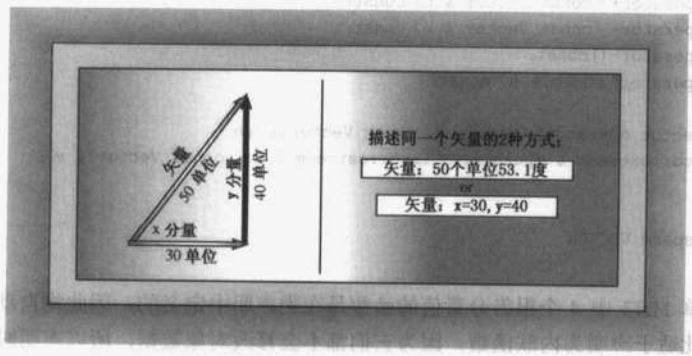


图 11.3　矢量的 x 和 y 分量

有时一种表示形式更方便，而有时另一种更方便，因此我们的类描述中将包含这两种表示形式（参见本章后面的注释“多种表示方式和类”）。另外，我们在设计这个类时，将使得用户修改了矢量的一种表示后，对象将自动更新另一种表示。使对象有这种智能，是 C++类的另一个优点。程序清单 11.13 列出了这

个类的声明。为复习名称空间，该清单将类声明放在 VECTOR 名称空间中。

注意：如果所用的系统不支持名称空间，可以删除下面几行：

```
namespace VECTOR
{
}  // end namespace VECTOR
```

程序清单 11.13 vector.h

```
// vect.h -- Vector class with <<, mode state
#ifndef VECTOR_H_
#define VECTOR_H_
#include <iostream>
namespace VECTOR
{

   class Vector
   {
   private:
      double x;            // horizontal value
      double y;            // vertical value
      double mag;          // length of vector
      double ang;          // direction of vector
      char mode;           // 'r' = rectangular, 'p' = polar
   // private methods for setting values
      void set_mag();
      void set_ang();
      void set_x();
      void set_y();
   public:
      Vector();
      Vector (double n1, double n2, char form = 'r');
      void set (double n1, double n2, char form = 'r');
      ~Vector();
      double xval()const {return x; }          // report x value
      double yval()const {return y; }          // report y value
      double magval()const {return mag; } // report magnitude
      double angval()const {return ang; } // report angle
      void polar_mode();                       // set mode to 'p'
      void rect_mode();                        // set mode to 'r'
   // operator overloading
      Vector operator+ (const Vector & b) const;
      Vector operator- (const Vector & b) const;
      Vector operator-()const;
      Vector operator* (double n) const;
   // friends
      friend Vector operator* (double n, const Vector & a);
      friend std::ostream & operator<< (std::ostream & os, const Vector & v);
   };

}  // end namespace VECTOR
#endif
```

注意，程序清单 11.13 中 4 个报告分量值的函数是在类声明中定义的，因此将自动成为内联函数。这些函数非常短，因此适于声明为内联函数。因为它们都不会修改对象数据，所以声明时使用了 const 限定符。第 10 章介绍过，这种句法用于声明那些不会对其显式访问的对象进行修改的函数。

程序清单 11.14 列出了程序清单 11.13 中声明的方法和友元函数的定义，该清单利用了名称空间的开放性，将方法定义添加到 VECTOR 名称空间中。请注意，构造函数和 set() 函数都设置了矢量的直角坐标和极坐标表示，因此需要这些值时，可直接使用而无须进行计算。另外，正如第 4 章和第 7 章指出的，C++

的内置数学函数在使用角度时以弧度为单位，所以函数在度和弧度之间进行转换。该 Vector 类实现对用户隐藏了极坐标和直角坐标之间的转换以及弧度和度之间的转换等内容。用户只需知道：类在使用角度时以度为单位，可以使用两种等价的形式来表示矢量。

程序清单 1.14　vector.cpp

```
// vect.cpp -- methods for Vector class
#include <cmath>
#include "vect.h"      //includes<iostream>
using std::sin;
using std::cos;
using std::atan2;
using std::cout;

namespace VECTOR
{

   const double Rad_to_deg = 57.2957795130823;

   // private methods
   // calculates magnitude from x and y
   void Vector::set_mag()
   {
      mag = sqrt (x * x + y * y);
   }

   void Vector::set_ang()
   {
      if (x == 0.0 && y == 0.0)
         ang = 0.0;
   else
      ang = atan2 (y, x);
}
// set x from polar coordinate
void Vector::set_x()
{
   x = mag * cos (ang);
}

// set y from polar coordinate
void Vector::set_y()
{
   y = mag * sin (ang);
}

// public methods
Vector::Vector()          // default constructor
{
   x = y = mag = ang = 0.0;
   mode = 'r';
}

// construct vector from rectangular coordinates if form is r
// (the default) or else from polar coordinates if form is p
Vector::Vector (double n1, double n2, char form)
{
   mode = form;
   if (form == 'r')
    {
      x = n1;
```

```
        y = n2;
        set_mag();
        set_ang();
    }
    else if (form == 'p')
    {
        mag = n1;
        ang = n2 / Rad_to_deg;
        set_x();
        set_y();
    }
    else
    {
        cout << "Incorrect 3rd argument to Vector()-- ";
        cout << "vector set to 0\n";
        x = y = mag = ang = 0.0;
        mode = 'r';
    }
}

// set vector from rectangular coordinates if form is r (the
// default) or else from polar coordinates if form is p
void Vector::set (double n1, double n2, char form)
{
    mode = form;
    if (form == 'r')
    {
        x = n1;
        y = n2;
        set_mag();
        set_ang();
    }
    else if (form == 'p')
    {
        mag = n1;
        ang = n2 / Rad_to_deg;
        set_x();
        set_y();
    }
    else
    {
        cout << "Incorrect 3rd argument to Vector()-- ";
        cout << "vector set to 0\n";
        x = y = mag = ang = 0.0;
        mode = 'r';
    }
}

Vector::~Vector()  // destructor
{
}

void Vector::polar_mode()  // set to polar mode
{
    mode = 'p';
}

void Vector::rect_mode()  // set to rectangular mode
{
    mode = 'r';
```

```
}

// operator overloading
// add two Vectors
Vector Vector::operator+ (const Vector & b) const
{
    return Vector (x + b.x, y + b.y);
}

// subtract Vector b from a
Vector Vector::operator- (const Vector & b) const
{
    return Vector (x - b.x, y - b.y);
}
    // reverse sign of Vector
    Vector Vector::operator-()const
    {
        return Vector (-x, -y);
    }

    // multiple vector by n
    Vector Vector::operator* (double n) const
    {
        return Vector (n * x, n * y);
    }

    // friend methods
    // multiply n by Vector a
    Vector operator* (double n, const Vector & a)
    {
        return a * n;
    }

    // display rectangular coordinates if mode is r,
    // else display polar coordinates if mode is p
    std::ostream & operator<< (std::ostream & os, const Vector & v)
    {
        if (v.mode == 'r')
            os << " (x, y) = (" << v.x << ", " << v.y << ") ";
        else if (v.mode == 'p')
        {
            os << " (m, a) = (" << v.mag << ", "
                << v.ang * Rad_to_deg << ") ";
        }
        else
            os << "Vector object mode is invalid";
        return os;
    }

} // end namespace VECTOR
```

也可以以另一种方式来设计这个类。例如，在对象中存储直角坐标而不是极坐标，并使用方法 magval() 和 angval() 来计算极坐标。对于很好进行坐标转换的应用来说，这将是一种效率更高的设计。另外，方法 set() 并非必不可少的。假设 shove 是一个 Vector 对象，而您编写了如下代码：

```
shove.set(100,300);
```

可以使用构造函数来得到相同的结果：

```
shove = Vector(100,300);    // create and assign a temporary object
```

然而，方法 set() 直接修改 shove 的内容，而使用构造函数将增加额外的步骤：创建一个临时对象，然后将其赋给 shove。

这些设计决策遵守了 OOP 传统，即将类接口的重点放在其本质上（抽象模型），而隐藏细节。这样，当用户使用 Vector 类时，只需考虑矢量的通用特性，例如，矢量可以表示位移，可以将两个矢量相加等。使用分量还是大小和方向来表示矢量已无关紧要，因为程序员可以设置矢量的值，并选择最方便的格式来显示它们。

下面更详细地介绍 Vector 类的一些特性。

注意：有些系统可能使用的是 math.h，而不是 cmath。另外，有些 C++系统不自动搜索数学库。例如，有些 UNIX 系统要求这样做：

```
$ CC source_file(s) -lm
```

-lm 选项命令编译器搜索数学库。因此，最终编译使用 Vector 类的程序时，如果显示关于未定义的外部函数的消息，请尝试使用-lm 选项，或查看系统是否需要使用类似的选项。

和头文件一样，如果您的系统不支持名称空间，可以删除下面几行：

```
namespace VECTOR
{
}  // end namespace VECTOR
```

11.5.1 使用状态成员

Vector 类储存了矢量的直角坐标和极坐标。它使用名为 mode 的成员来控制使用构造函数、set()方法和重载的 operator<<()函数使用哪种形式，其中“*r*”表示直角坐标模式（默认值）、“*p*”表示极坐标模式。这样的成员被称为状态成员（state member），因为这种成员描述的是对象所处的状态。要知道具体含义，请看构造函数的代码：

```
Vector::Vector(double n1, double n2, char form)
{
    mode = form;
    if(form == 'r')
    {
        x = n1;
        y = n2;
        set_mag();
        set_ang();
    }
    else if(form == 'p')
    {
        mag = n1;
        ang = n2 / Rad_to_deg;
        set_x();
        set_y();
    }
    else
    {
        cout << "Incorrect 3rd argument to Vector()-- ";
        cout << "vector set to 0\n";
        x = y = mag = ang = 0.0;
        mode = 'r';
    }
}
```

如果第三个参数是“*r*”或省略了（原型将默认值设置为“*r*”），则将输入解释为直角坐标；如果为“*p*”，则将输入解释为极坐标：

```
Vector folly(3.0, 4.0);          // set x = 3, y = 4
Vector foolery(20.0, 30.0, 'p');  // set mag = 20, ang = 30
```

注意，如果用户提供的是 *x* 值和 *y* 值，则构造函数将使用私有方法 set_mag()和 set_ang()来设置距离和角度值；如果提供的是距离和角度值，则构造函数将使用 set_x()和 set_y()方法来设置 *x* 值和 *y* 值。另外，如果用户指定的不是“*r*”或“*p*”，则构造函数将显示一条警告消息，并将状态设置为“*r*”。

同样，operator<<()函数也使用模式来确定如何显示值：

```
// display rectangular coordinates if mode is r,
// else display polar coordinates if mode is p
ostream & operator<< (ostream & os, const Vector & v)
{
    if (v.mode == 'r')
        os << " (x, y) = (" << v.x << ", " << v.y << ") ";
    else if (v.mode == 'p')
    {
        os << " (m, a) = (" << v.mag << ", "
          << v.ang * Rad_to_deg << ") ";
    }
    else
        os << "Vector object mode is invalid";
    return os;
}
```

设置模式的各种方法只接受‘*r*’和‘*p*’为合法的值，因此该函数中的 else 永远不会执行。但进行检查还是一个不错的主意，它有助于捕获难以发现的编程错误。

多种表示方式和类

可以用不同但等价的方式表示的量很常见。例如，可以按每加仑汽油消耗汽车能行驶的英里数来计算油耗（美国），也可以按每 100 公里消耗多少公升汽油来计算（欧洲）。可以用字符串表示数字，也可以用数值方式表示，可以使用 IQ 或 kiloturkey 的方法表示智商。类非常适于在一个对象中表示实体的不同方面。首先在一个对象中存储多种表示方式；然后，编写这样的类函数，以便给一种表示方式赋值时，将自动给其他表示方式赋值。例如，Vector 类的 set_by_polar()方法将 mag 和 ang 成员设置为函数参数的值，并同时设置成员 *x* 和 *y*。通过在内部处理转换，类允许从本质（而不是表示方式）上来看待一个量。

11.5.2　为 Vector 类重载算术操作符

在使用 *x*、*y* 坐标时，将两个矢量相加将非常简单，只要将两个 *x* 分量相加，得到最终的 *x* 分量，将两个 *y* 分量相加，得到最终的 *y* 分量即可。根据这种描述，可能使用下面的代码：

```
Vector Vector::operator+ (const Vector & b) const
{
    Vector sum;
    sum.x = x + b.x;
    sum.y = y + b.y;
    return sum;          //incomplete version
}
```

如果对象只存储 *x* 和 *y* 分量，则这很好。遗憾的是，上述代码无法设置极坐标值。可以通过添加另外一些代码来解决这种问题：

```
Vector Vector::operator+ (const Vector & b) const
{
    Vector sum;
    sum.x = x + b.x;
    sum.y = y + b.y;
    sum.set_ang (sum.x, sum.y);
    sum.set_mag (sum.x, sum.y);
    return sum;                    //version duplicate needlessly
}
```

不过，使用构造函数来完成这种工作，将更简单、更可靠：

```
Vector Vector::operator+ (const Vector & b) const
{
    return Vector (x + b.x, y + b.y);      // return the constructed Vector
}
```

上述代码将新的 *x* 分量和 *y* 分量传递给 Vector 构造函数，而后者将使用这些值来创建无名的新对象，

并返回该对象的拷贝。这确保了新的 Vector 对象是根据构造函数制定的标准规则创建的。

提示：如果方法通过计算得到一个新的类对象，则应考虑是否可以使用类构造函数来完成这种工作。这样做不仅可以避免麻烦，而且可以确保新的对象是按照正确的方式创建的。

1. 乘法

将矢量与一个数相乘，将使该矢量加长或缩短（取决于这个数）。因此，将矢量乘以 3 得到的矢量的长度为原来的 3 倍，而方向不变。要在 Vector 类中实现矢量的这种行为很容易。对于极坐标，只要将长度进行伸缩，并保持角度不变即可；对于直角坐标，只需将 x 和 y 分量进行伸缩即可。也就是说，如果矢量的分量为 5 和 12，则将其乘以 3 后，分量将分别是 15 和 36。这正是重载的乘法操作符要完成的工作：

```
Vector Vector::operator*(double n) const
{
   return Vector(n * x, n * y);
}
```

和重载加法一样，上述代码允许构造函数使用新的 x 和 y 分量来创建正确的 Vector 对象。上述函数用于处理 Vector 值和 double 值相乘。可以像 Time 范例那样，使用一个内联友元函数来处理 double 与 Vector 相乘：

```
Vector operator*(double n, const Vector & a) // friend function
{
   return a * n;   // convert double times Vector to Vector times double
}
```

2. 对已经被重载的操作符进行重载

在 C++中，(-)操作符已经有两种含义。首先，使用两个操作数，它是减法操作符。减法操作符是一个二元操作符，因为它有两个操作数。其次，使用一个操作数时（如-x），它是负号操作符。这种形式被称为一元操作符，即只有一个操作数。对于矢量来说，这两种操作（减法和符号反转）都是有意义的，因此 Vector 类有这两种操作。

要从矢量 A 中减去矢量 B，只要将分量相减即可，因此重载减法与重载加法相似：

```
Vector operator-(const Vector & b) const;          // prototype
Vector Vector::operator-(const Vector & b) const   // definition
{
   return Vector(x - b.x, y - b.y);  // return the constructed Vector
}
```

其中操作数的顺序非常重要。下面的语句：

```
diff = v1 - v2;
```

将被转换为下面的成员函数调用：

```
diff = v1.operator-(v2);
```

这意味着将从隐式矢量参数减去以显式参数传递的矢量，所以应使用 x - b.x，而不是 b.x - x。

接下来，来看一元负号操作符，它只使用一个操作数。将这个操作符用于数字（如-x）时，将改变它的符号。因此，将这个操作符用于矢量时，将反转矢量的每个分量的符号。更准确地说，函数应返回一个与原来的矢量相反的矢量（对于极坐标，长度不变，但方向相反。许多没受过数学训练的政治家在这方面是天才）。下面是重载负号的原型和定义：

```
Vector operator-()const;
Vector Vector::operator-()const
{
   return Vector(-x, -y);
}
```

现在，operator-()有两种不同的定义。这是可行的，因为它们的特征标不同。可以定义(-)操作符的一元和二元版本，因为 C++提供了该操作符的一元和二元版本。对于只有二元形式的操作符（如除法操作符），只能将其重载为二元操作符。

记住：因为操作符重载是通过函数来实现的，所以只要操作符函数的特征标不同，使用的操作符数量

与相应的内置 C++操作符相同，就可以多次重载同一个操作符。

11.5.3　对实现的说明

前几节介绍的实现在 Vector 对象中存储了矢量的直角坐标和极坐标，但公有接口并不依赖于这一事实。所有接口都只要求能够显示这两种表示，并可以返回各个值。内部实现方式可以完全不同。正如前面指出的，对象可以只存储 x 和 y 分量，而返回矢量长度的 magval() 方法可以根据 x 和 y 的值来计算出长度，而不是查找对象中存储的这个值。这种方法改变了实现，但用户接口不变。将接口与实现分离是 OOP 的目标之一，这样允许对实现进行调整，而无需修改使用这个类的程序中的代码。

这两种实现各有利弊。存储数据意味着对象将占据更多的内存，每次 Vector 对象被修改时，都需要更新直角坐标和极坐标表示；但查找数据的速度比较快。如果应用程序经常需要访问矢量的这两种表示，则这个例子采用的实现比较合适；如果只是偶尔需要使用极坐标，则另一种实现更好。可以在一个程序中使用一种实现，而在另一个程序中使用另一种实现，但它们的用户接口相同。

11.5.4　使用 Vector 类来模拟随机行走

程序清单 11.15 是一个小程序，它使用了修订后的 Vector 类。该程序模拟了著名的醉鬼走路问题（Drunkard Walk problem）。实际上，醉鬼被认为是一个有许多健康问题的人，而不是大家娱乐消遣的谈资，因此这个问题通常被称为随机行走问题。其意思是，将一个人领到街灯柱下。这个人开始走动，但每一步的方向都是随机的（与前一步不同）。这个问题的一种表述是，这个人走到离灯柱 50 英尺处需要多少步。从矢量的角度看，这相当于不断将方向随机的矢量相加，直到长度超过 50 英尺。

程序清单 11.15 允许用户选择行走距离和步长。该程序用一个变量来表示位置（一个矢量），并报告到达指定距离处（用两种格式表示）所需的步数。可以看到，行走者前进得相当慢。虽然走了 1000 步，每步的距离为 2 英尺，但离起点可能只有 50 英尺。这个程序将行走者所走的净距离（这里为 50 英尺）除以步数，来指出这种行走方式的低效性。随机改变方向使得该平均值远远小于步长。为了随机选择方向，该程序使用了标准库函数 rand()、srand() 和 time()（参见程序说明）。请务必将程序清单 11.14 和程序清单 11.15 一起进行编译。

程序清单 11.15　randwalk.cpp

```
// randwalk.cpp -- using the Vector class
// compile with the vect.cpp file
#include <iostream>
#include <cstdlib>      // rand(), srand()prototypes
#include <ctime>        // time()prototype
#include "vect.h"
int main()
{
   using namespace std;
   using VECTOR::Vector;
   srand (time (0));     // seed random-number generator
   double direction;
   Vector step;
   Vector result (0.0, 0.0);
   unsigned long steps = 0;
   double target;
   double dstep;
   cout << "Enter target distance (q to quit): ";
   while (cin >> target)
   {
      cout << "Enter step length: ";
      if (! (cin >> dstep))
        break;
```

```
        while (result.magval()< target)
        {
          direction = rand()% 360;
          step.set (dstep, direction, 'p');
          result = result + step;
          steps++;
        }
        cout << "After " << steps << " steps, the subject "
          "has the following location: \n";
        cout << result << endl;
        result.polar_mode();
        cout <<" or\n"<< result << endl;
        cout << "Average outward distance per step = "
          << result.magval()/steps << endl;
        steps = 0;
        result.set (0.0, 0.0);
        cout << "Enter target distance (q to quit): ";
    }
    cout << "Bye!\n";

    return 0;
}
```

注意：可能需要使用 stdlib.h 和 time.h，而不是 cstdlib 和 ctime。如果所用的系统不支持名称空间，请省略下面一行：

```
using VECTOR::Vector;
```

下面是程序清单 11.13～11.15 组成的程序的运行情况：

```
Enter target distance (q to quit) : 50
Enter step length: 2
After 253 steps, the subject has the following location:
(x, y) = (46.1512, 20.4902)
  or
(m, a) = (50.495, 23.9402)
Average outward distance per step = 0.199587
Enter target distance (q to quit) : 50
Enter step length: 2
After 951 steps, the subject has the following location:
(x, y) = (-21.9577, 45.3019)
  or
(m, a) = (50.3429, 115.8593)
Average outward distance per step = 0.0529362
Enter target distance (q to quit) : 50
Enter step length: 1
After 1716 steps, the subject has the following location:
(x, y) = (40.0164, 31.1244)
  or
(m, a) = (50.6956, 37.8755)
Average outward distance per step = 0.0295429
Enter target distance (q to quit) : q
Bye!
```

这种处理的随机性使得每次运行结果都不同，即使初始条件相同。不过，平均而言，步长减半，步数将为原来的 4 倍。概率理论表明，平均而言，步数（N）、步长（s），净距离 D 之间的关系如下：

$$N = (D/s)^2$$

这只是平均情况，但每次试验结果可能相差很大。例如，进行 1000 次试验（走 50 英尺，步长为 2 英尺）时，平均步数为 636（与理论值 625 非常接近），但实际步数位于 91～3951 之间。同样，进行 1000 次试验（走 50 英尺，步长为 1 英尺）时，平均步数为 2557（与理论值 2500 非常接近），但实际步数位于 345～10882 之间。因此，如果发现自己在随机行走时，请保持自信，迈大步走。虽然在蜿蜒前进的过程中仍旧

无法控制前进的方向，但至少会走得远一点。

程序说明

首先需要指出的是，在程序清单 11.15 中使用 VECTOR 名称空间非常方便。下面的 using 声明：

```
using VECTOR::Vector;
```

使 Vector 类的名称可用。因为所有的 Vector 类方法的作用域都为整个类，所以导入类名后，无需提供其他 using 声明，就可以使用 Vector 的方法。

接下来谈谈随机数。标准 ANSI C 库（C++也有）中有一个 rand()函数，它返回一个从 0 到某个值（取决于实现）之间的随机整数。该程序使用求模操作数来获得一个 0～359 的角度值。rand()函数将一种算法用于一个初始种子值来获得随机数，该随机值将用作下一次函数调用的种子，依此类推。这些数实际上是伪随机数，因为 10 次连续的调用通常将生成 10 个同样的随机数（具体值取决于实现）。不过，srand()函数允许覆盖默认的种子值，重新启动另一个随机数序列。该程序使用 time（0）的返回值来设置种子。time（0）函数返回当前时间，通常为从某一个日期开始的秒数（更广义地，time()接受 time_t 变量的地址，将时间放到该变量中，并返回它。将 0 用作地址参数，可以省略 time_t 变量声明）。因此，下面的语句：

```
srand (time (0));
```

在每次运行程序时，都将设置不同的种子，使随机输出看上去更为随机。头文件 cstdlib（以前为 stdlib.h）包含了 srand()和 rand()的原型，而 ctime（以前是 time.h）包含了 time()的原型。

该程序使用 result 矢量记录行走者的前进情况。内循环每轮将 step 矢量设置为新的方向，并将它与当前的 result 矢量相加。当 result 的长度超过指定的距离后，该循环结束。

程序通过设置矢量的模式，用直角坐标和极坐标显示最终的位置。

下面这条语句：

```
result = result + step;
```

将 result 设置为‘r’模式，而不管 result 和 step 的初始模式是什么。这样做的原因如下。首先，加法操作符函数创建并返回一个新矢量，该矢量存储了这两个参数的和。该函数使用默认构造函数以‘r’模式创建矢量。因此，被赋给 result 的矢量的模式为‘r’。默认情况下，赋值时将分别给每个成员变量赋值，因此将‘r’赋给了 result.mode。如果偏爱其他方式，例如，result 保留原来的模式，可以通过为类定义赋值操作符来覆盖默认的赋值方式。第 12 章将介绍这样的范例。

顺便说一句，在将一系列位置存储到文件中很容易。首先包含头文件 fstream，声明一个 ofstream 对象，将其同一个文件关联起来：

```
#include <fstream>
...
ofstream fout;
fout.open("thewalk.txt");
```

然后，在计算结果的循环中加入类似于下面的代码：

```
fout << result << endl;
```

这将调用友元函数 operator<<(fout, result)，导致引用参数 os 指向 fout，从而将输出写入到文件中。您还可以使用 fout 将其他信息写入到文件中，如当前由 cout 显示的总结信息。

11.6　类的自动转换和强制类型转换

下面介绍类的另一个主题——类型转换。本节讨论 C++如何处理用户定义类型的转换。在讨论这个问题之前，我们先来复习一下 C++是如何处理内置类型转换的。将一个标准类型变量的值赋给另一种标准类型的变量时，如果这两种类型兼容，则 C++自动将这个值转换为接收变量的类型。例如，下面的语句都将导致数值类型转换：

```
long count = 8;      // int value 8 converted to type long
double time = 11;    // int value 11 converted to type double
int side = 3.33;     // double value 3.33 converted to type int 3
```

上述赋值语句都是可行的，因为在C++看来，各种数值类型都表示同一样基本的东西——一数字，同时C++包含用于进行转换的内置规则。不过，第3章介绍过，这些转换将降低精度。例如，将3.33赋给int变量时，转换后的值为3，丢失了0.33。

C++语言不自动转换不兼容的类型。例如，下面的语句：

```
int * p = 10;  // type clash
```

是非法的，因为左边是指针类型，而右边是数字。虽然计算机内部可能使用整数来表示地址，但从概念上说，整数和指针完全不同。例如，不能计算指针的平方。不过，在无法自动转换时，可以使用强制类型转换：

```
int * p = (int *) 10;  // ok, p and (int *) 10 both pointers
```

上述语句将10强制转换为int指针类型（即int *类型），将指针设置为地址10。

可以将类定义成与基本类型或另一个类相关，使得从一种类型转换为另一种类型是有意义的。在这种情况下，程序员可以指示C++如何自动进行转换，或通过强制类型转换来完成。为了说明这是如何进行的，我们将第3章中的磅转换为英石的程序改写成类的形式。首先，设计一种合适的类型。我们基本上是以两种方式（磅和英石）来表示重量的。对于在一个实体中包含一个概念的两种表示来说，类提供了一种非常好的方式。因此可以将重量的两种表示放在同一个类中，然后提供以这两种方式表达重量的类方法。程序清单11.16提供了这个类的头文件。

程序清单11.16 stonewt.h

```
// stonewt.h -- definition for Stonewt class
#ifndef STONEWT_H_
#define STONEWT_H_
class Stonewt
{
private:
    enum {Lbs_per_stn = 14};          // pounds per stone
    int stone;                        // whole stones
    double pds_left;                  // fractional pounds
    double pounds;                    // entire weight in pounds
public:
    Stonewt (double lbs);             // constructor for double pounds
    Stonewt (int stn, double lbs);    // constructor for stone, lbs
    Stonewt();                        // default constructor
    ~Stonewt();
    void show_lbs()const;         // show weight in pounds format
    void show_stn()const;         // show weight in stone format
};
#endif
```

正如第10章指出的，对于定义类特定的常量来说，如果它们是整数，enum提供了一种方便的途径。新的编译器也允许采用下面这种方法：

```
static const int Lbs_per_stn = 14;
```

Stonewt类有3个构造函数，让您能够将Stonewt对象初始化为一个浮点数（单位为磅）或两个浮点数（分别代表英石和磅）。也可以创建Stonewt对象，而不进行初始化：

```
Stonewt kindy (132);  // weight = 132 pounds
Stonewt kandy (10, 2); // weight = 10 stone, 2 pounds
Stonewt kendy;        // weight = default value
```

另外，Stonewt类还提供了两个显示函数。一个以磅为单位来显示重量，另一个以英石和磅为单位来显示重量。程序清单11.17列出了类方法的实现。每个构造函数都给这3个私有成员全部赋了值。因此创建Stonewt对象时，将自动设置这两种重量表示。

程序清单11.17 stonewt.cpp

```
// stonewt.cpp -- Stonewt methods
#include <iostream>
```

```
using std::cout;
#include "stonewt.h"

// construct Stonewt object from double value
Stonewt::Stonewt (double lbs)
{
    stone = int (lbs) / Lbs_per_stn;    // integer division
    pds_left = int (lbs) % Lbs_per_stn + lbs - int (lbs);
    pounds = lbs;
}

// construct Stonewt object from stone, double values
Stonewt::Stonewt (int stn, double lbs)
{
    stone = stn;
    pds_left = lbs;
    pounds =  stn * Lbs_per_stn +lbs;
}

Stonewt::Stonewt()        // default constructor, wt = 0
{
    stone = pounds = pds_left = 0;
}

Stonewt::~Stonewt()       // destructor
{
}

// show weight in stones
void Stonewt::show_stn()const
{
    cout << stone << " stone, " << pds_left << " pounds\n";
}

// show weight in pounds
void Stonewt::show_lbs()const
{
    cout << pounds << " pounds\n";
}
```

因为 Stonewt 对象表示一个重量，所以可以提供一些将整数或浮点值转换为 Stonewt 对象的方法。我们已经这样做了！在 C++中，接受一个参数的构造函数为将类型与该参数相同的值转换为类提供了蓝图。因此，下面的构造函数：

```
Stonewt (double lbs);  // template for double-to-Stonewt conversion
```

用于将 double 类型的值转换为 Stonewt 类型。也就是说，可以编写这样的代码：

```
Stonewt myCat;        // create a Stonewt object
myCat = 19.6;         // use Stonewt (double) to convert 19.6 to Stonewt
```

程序将使用构造函数 Stonewt（double）来创建一个临时的 Stonewt 对象，并将 19.6 作为初始化值。随后，采用逐成员赋值方式将该临时对象的内容复制到 myCat 中。这一过程称为隐式转换，因为它是自动进行的，而不需要显式强制类型转换。

只有接受一个参数的构造函数才能作为转换函数。下面的构造函数：

```
Stonewt (int stn, double lbs);
```

有两个参数，因此不能用来转换类型。

将构造函数用作自动类型转换函数似乎是一项不错的特性。不过，当程序员拥有更丰富的 C++经验时，将发现这种自动特性并非总是合乎需要的，因为这会导致意外的类型转换。因此，最新的 C++实现新增了一个关键字（explicit），用来关闭这种自动特性。也就是说，可以这样声明构造函数：

```
explicit Stonewt (double lbs);   // no implicit conversions allowed
```

这将关闭上述范例中介绍的隐式转换，但仍然允许显式转换，即显式强制类型转换：

```
Stonewt myCat;              // create a Stonewt object
myCat = 19.6;               // not valid if Stonewt(double) is declared as explicit
mycat = Stonewt(19.6);      // ok, an explicit conversion
mycat = (Stonewt) 19.6;     // ok, old form for explicit typecast
```

记住：只接受一个参数的构造函数定义了从参数类型到类类型的转换。如果使用关键字 explicit 限定了这种构造函数，则它只能用于显示转换，否则也可以用于隐式转换。

编译器在什么时候将使用 Stonewt（double）函数呢？如果在声明中使用了关键字 explicit，则 Stonewt（double）将只用于显式强制类型转换，否则还可以用于下面的隐式转换。

- 将 Stonewt 对象初始化为 double 值时。
- 将 double 值赋给 Stonewt 对象时。
- 将 double 值传递给接受 Stonewt 参数的函数时。
- 返回值被声明为 Stonewt 的函数试图返回一个 double 值时。
- 在上述任意一种情况下，使用可转换为 double 类型的内置类型时。

下面详细介绍最后一点。函数原型化提供的参数匹配过程，允许使用 Stonewt（double）构造函数来转换其他数值类型。也就是说，下面两条语句都首先将 int 转换为 double，然后使用 Stonewt（double）构造函数。

```
Stonewt Jumbo(7000);        // uses Stonewt(double), converting int to double
Jumbo = 7300;               // uses Stonewt(double), converting int to double
```

但是，当且仅当转换不存在二义性时，才会进行这种二步转换。也就是说，如果这个类还定义了构造函数 Stonewt（long），则编译器将拒绝这些语句，可能指出：int 可被转换为 long 或 double，因此调用存在二义性。

程序清单 11.18 使用类的构造函数初始化了一些 Stonewt 对象，并处理类型转换。请务必将程序清单 11.18 和程序清单 11.17 一起编译。

程序清单 11.18 stone.cpp

```
// stone.cpp -- user-defined conversions
// compile with stonewt.cpp
#include <iostream>
using std::cout;
#include "stonewt.h"
void display(const Stonewt st, int n);
int main()
{
    Stonewt pavarotti = 260; // uses constructor to initialize
    Stonewt wolfe(285.7);    // same as Stonewt wolfe = 285.7;
    Stonewt taft(21, 8);

    cout << "The tenor weighed ";
    pavarotti.show_stn();
    cout << "The detective weighed ";
    wolfe.show_stn();
    cout <<"The President weighed ";
    taft.show_lbs();
    pavarotti = 265.8;       // uses constructor for conversion
    taft = 325;              // same as taft = Stonewt(325);
    cout << "After dinner, the tenor weighed ";
    pavarotti.show_stn();
    cout << "After dinner, the President weighed ";
    taft.show_lbs();
    display(taft, 2);
    cout << "The wrestler weighed even more.\n";
    display(422, 2);
```

```
    cout << "No stone left unearned\n";
    return 0;
}

void display(const Stonewt & st, int n)
{
    for (int i = 0; i < n; i++)
    {
        cout << "Wow! ";
        st.show_stn();
    }
}
```

下面是程序清单 11.18 中程序的输出：

```
The tenor weighed 18 stone, 8 pounds
The detective weighed 20 stone, 5.7 pounds
The President weighed 302 pounds
After dinner, the tenor weighed 18 stone, 13.8 pounds
After dinner, the President weighed 325 pounds
Wow! 23 stone, 3 pounds
Wow! 23 stone, 3 pounds
The wrestler weighed even more.
Wow! 30 stone, 2 pounds
Wow! 30 stone, 2 pounds
No stone left unearned
```

11.6.1　程序说明

当构造函数只接受一个参数时，可以使用下面的格式来初始化类对象：

```
// a syntax for initializing a class object when
// using a constructor with one argument
Stonewt pavarotti = 260;
```

这等价于我们使用过的另外两种格式：

```
// standard syntax forms for initializing class objects
Stonewt pavarotti(260);
Stonewt pavarotti = Stonewt(260);
```

不过，后两种格式可用于接受多个参数的构造函数。

接下来，请注意程序清单 11.18 的下面两条赋值语句：

```
pavarotti = 265.8;
taft = 325;
```

第一条赋值语句使用接受 double 参数的构造函数，将 265.8 转换为一个 Stonewt 值，这将把 pavarotti 的 pound 成员设置为 265.8。因为该语句使用了构造函数，所以还将设置 stone 和 pds_left 成员。同样，第二条赋值语句将一个 int 值转换为 double 类型，然后使用 Stonewt（double）来设置全部 3 个成员。

最后，请注意下面的函数调用：

```
display(422, 2);    // convert 422 to double, then to Stonewt
```

display()的原型表明，第一个参数应是 Stonewt 对象（Stonewt 和 Stonewt &形参都与 Stonewt 实参匹配）。遇到 int 参数时，编译器查找 Stonewt（int）构造函数，以便将该 int 转换为 Stonewt 类型。由于没有找到这样的构造函数，因此编译器寻找接受其他内置类型（int 可以转换为这种类型）的构造函数。Stone（double）构造函数满足这种要求，因此编译器将 int 转换为 double，然后使用 Stonewt（double）将其转换为一个 Stonewt 对象。

11.6.2　转换函数

程序清单 11.18 将数字转换为 Stonewt 对象。可以做相反的转换吗？也就是说，是否可以将 Stonewt 对象转换为 double 值，就像如下所示的那样？

```
Stonewt wolfe(285.7);
```

```
double host = wolfe; // ?? possible ??
```

可以这样做，但不是使用构造函数。构造函数只用于从某种类型到类类型的转换。要进行相反的转换，必须使用特殊的 C++操作符函数——转换函数。

转换函数是用户定义的强制类型转换，可以像使用强制类型转换那样使用它们。例如，如果定义了从 Stonewt 到 double 的转换函数，就可以使用下面的转换：

```
Stonewt wolfe(285.7);
double host = double (wolfe);      // syntax #1
double thinker = (double) wolfe;   // syntax #2
```

也可以让编译器来决定如何做：

```
Stonewt wells(20, 3);
double star = wells;   // implicit use of conversion function
```

编译器发现，右侧是 Stonewt 类型，而左侧是 double 类型，因此它将查看程序员是否定义了与此匹配的转换函数（如果没有找到这样的定义，编译器将生成错误消息，指出无法将 Stonewt 赋给 double）。

那么，如何创建转换函数呢？要转换为 typeName 类型，需要使用这种形式的转换函数：

```
operator typeName();
```

请注意以下几点：

- 转换函数必须是类方法。
- 转换函数不能指定返回类型。
- 转换函数不能有参数。

例如，转换为 double 类型的函数的原型如下：

```
operator double();
```

typeName（这里为 double）指出了要转换成的类型，因此不需要指定返回类型。转换函数是类方法意味着：它需要通过类对象来调用，从而告知函数要转换的值。因此，函数不需要参数。

要添加将 stone_wt 对象转换为 int 类型和 double 类型的函数，需要将下面的原型添加到类声明中：

```
operator int();
operator double();
```

程序清单 11.19 列出了修改后的类声明。

程序清单 11.19 stonewt1.h

```
// stonewt1.h -- revised definition for Stonewt class
#ifndef STONEWT1_H_
#define STONEWT1_H_
class Stonewt
{
private:
    enum {Lbs_per_stn = 14};          // pounds per stone
    int stone;                        // whole stones
    double pds_left;                  // fractional pounds
    double pounds;                    // entire weight in pounds
public:
    Stonewt (double lbs);             // construct from double pounds
    Stonewt (int stn, double lbs);    // construct from stone, lbs
    Stonewt();                        // default constructor
    ~Stonewt();
    void show_lbs()const;             // show weight in pounds format
    void show_stn()const;             // show weight in stone format
// conversion functions
    operator int()const;
    operator double()const;
};
#endif
```

程序清单 11.20 是在程序清单 11.18 的基础上修改而成的，包括了这两个转换函数的定义。注意，虽然没有声明返回类型，这两个函数也将返回所需的值。另外，int 转换将待转换的值四舍五入为最接近的整数，

而不是去掉小数部分。例如，如果 pounds 为 114.4，则 pounds +0.5 等于 114.9，int（114.9）等于 114。但是如果 pounds 为 114.6，则 pounds + 0.5 是 115.1，而 int（115.1）为 115。

程序清单 11.20　stonewt1.cpp

```
// stonewt1.cpp -- Stonewt class methods + conversion functions
#include <iostream>
using std::cout;
#include "stonewt1.h"

// construct Stonewt object from double value
Stonewt::Stonewt(double lbs)
{
    stone = int (lbs) / Lbs_per_stn;    // integer division
    pds_left = int (lbs) % Lbs_per_stn + lbs - int(lbs);
    pounds = lbs;
}

// construct Stonewt object from stone, double values
Stonewt::Stonewt(int stn, double lbs)
{
    stone = stn;
    pds_left = lbs;
    pounds =  stn * Lbs_per_stn +lbs;
}

Stonewt::Stonewt()          // default constructor, wt = 0
{
    stone = pounds = pds_left = 0;
}
Stonewt::~Stonewt()         // destructor
{
}

// show weight in stones
void Stonewt::show_stn() const
{
    cout << stone << " stone, " << pds_left << " pounds\n";
}

// show weight in pounds
void Stonewt::show_lbs() const
{
    cout << pounds << " pounds\n";
}

// conversion functions
Stonewt::operator int() const
{

    return int (pounds + 0.5);

}

Stonewt::operator double()const
{
    return pounds;
}
```

程序清单 11.21 对新的转换函数进行测试。该程序中的赋值语句使用隐式转换，而最后的 cout 语句使

用显式强制类型转换。请务必将程序清单 11.20 与程序清单 11.21 一起编译。

程序清单 11.21 stone1.cpp

```
// stone1.cpp -- user-defined conversion functions
// compile with stonewt1.cpp
#include <iostream>
#include "stonewt1.h"

int main()
{
   using std::cout;
   Stonewt poppins(9, 2.8);      // 9 stone, 2.8 pounds
   double p_wt = poppins;        // implicit conversion
   cout << "Convert to double => ";
   cout << "Poppins: "<< p_wt << " pounds.\n";
   cout << "Convert to int => ";
   cout << "Poppins: " << int(poppins) << " pounds.\n";
   return 0;
}
```

下面是程序清单 11.19～11.21 组成的程序的输出；它显示了将 Stonewt 对象转换为 double 类型和 int 类型的结果：

```
Convert to double => Poppins: 128.8 pounds.
Convert to int => Poppins: 129 pounds.
```

自动应用类型转换

程序清单 11.21 将 int（poppins）和 cout 一起使用。假设省略了显式强制类型转换：

```
cout << "Poppins: " << poppins << " pounds.\n";
```

程序会像在下面的语句中那样使用隐式转换吗？

```
double p_wt = poppins;
```

答案是否定的。在 p_wt 范例中，上下文表明，poppins 应被转换为 double 类型。但在 cout 范例中，并没有指出应转换为 int 类型还是 double 类型。在缺少信息时，编译器将指出，程序中使用了二义性转换。该语句没有指出要使用什么类型。

有趣的是，如果类只定义了 double 转换函数，则编译器将接受该语句。这是因为只有一种转换可能，因此不存在二义性。

赋值的情况与此类似。对于当前的类声明来说，编译器将认为下面的语句有二义性而拒绝它。

```
long gone = poppins;   // ambiguous
```

在 C++中，int 和 double 值都可以被赋给 long 变量，所以编译器使用任意一个转换函数都是合法的。编译器不想承担选择转换函数的责任。但是，如果删除了这两个转换函数之一，编译器将接受这条语句。例如，假设省略了 double 定义，则编译器将使用 int 转换，将 poppins 转换为一个 int 类型的值。然后在将它赋给 gone 时，将 int 类型值转换为 long 类型。

当类定义了两种或更多的转换时，仍可以用显式强制类型转换来指出要使用哪个转换函数。可以使用下面任何一种强制类型转换表示法：

```
long gone = (double) poppins;  // use double conversion
long gone = int (poppins);     // use int conversion
```

第一条语句将 poppins 转换为一个 double 值，然后赋值操作将该 double 值转换为 long 类型。同样，第二条语句将 poppins 首先转换为 int 类型，随后转换为 long。

和转换构造函数一样，转换函数也有其优缺点。提供执行自动、隐式转换的函数所存在的问题是：在用户不希望进行转换时，转换函数也可能进行转换。例如，假设您在睡眠不足时编写了下面的代码：

```
int ar[20];
...
Stonewt temp(14, 4);
...
```

```
int Temp = 1;
...
cout << ar[temp] << "!\n";  // used temp instead of Temp
```

您希望编译器能够捕获诸如使用了对象而不是整数作为数组索引等错误，但 Stonewt 类定义了一个 operator int()，因此 Stonewt 对象 temp 将被转换为 int 200，并用作数组索引。原则上说，最好使用显式转换，而避免隐式转换。关键字 explicit 不能用于转换函数，但只需用一个功能相同的非转换函数替换该转换函数即可，但仅在被显式地调用时，该函数才会执行。也就是说，可以将：

```
Stonewt::operator int(){ return int (pounds + 0.5) ; }
```

替换为：

```
int Stonewt::Stone_to_Int(){ return int (pounds + 0.5) ; }
```

这样，下面的语句将是非法的：

```
int plb = poppins;
```

但如果确实需要这种转换，可以这样做：

```
int plb = poppins.Stone_to_Int();
```

警告：应谨慎地使用隐式转换函数。通常，最好选择仅在被显式地调用时才会执行的函数。

总之，C++为类提供了下面的类型转换：

- 只有一个参数的类构造函数用于将类型与该参数相同的值转换为类类型。例如，将 int 值赋给 Stonewt 对象时，接受 int 参数的 Stonewt 类构造函数将自动被调用。不过，在构造函数声明中使用 explicit 可防止隐式转换，而只允许显式转换。
- 被称为转换函数的特殊类成员操作符函数，用于将类对象转换为其他类型。转换函数是类成员，没有返回类型、没有参数、名为 operator typeName()，其中，typeName 是对象将被转换成的类型。将类对象赋给 typeName 变量或将其强制转换为 typeName 类型时，该转换函数将自动被调用。

11.6.3　转换函数和友元函数

下面为 Stonewt 类重载加法操作符。在讨论 Time 类时指出过，可以使用成员函数或友元函数来重载加法（出于简化的目的，我们假设没有定义 operator double()转换函数）。可以使用下面的成员函数实现加法：

```
Stonewt Stonewt::operator+ (const Stonewt & st) const
{
   double pds = pounds + st.pounds;
   Stonewt sum (pds);
   return sum;
}
```

也可以将加法作为友元函数来实现，如下所示：

```
Stonewt operator+ (const Stonewt & st1, const Stonewt & st2)
{
   double pds = st1.pounds + st2.pounds;
   Stonewt sum (pds);
   return sum;
}
```

别忘了，可以提供方法定义或友元函数定义，但不能都提供。

上面任何一种格式都允许这样做：

```
Stonewt jennySt (9, 12);
Stonewt bennySt (12, 8);
Stonewt total;
total = jennySt + bennySt;
```

另外，如果提供了 Stonewt（double）构造函数，则也可以这样做：

```
Stonewt jennySt (9, 12);
double kennyD = 176.0;
Stonewt total;
total = jennySt + kennyD;
```

但是只有友元函数才允许这样做：

```
Stonewt jennySt (9, 12);
double pennyD = 146.0;
Stonewt total;
total = pennyD + jennySt;
```

为了解其中的原因，将每一种加法都转换为相应的函数调用。首先：

```
total = jennySt + bennySt;
```

被转换为：

```
total = jennySt.operator+ (bennySt);   // member function
```

或：

```
total = operator+ (jennySt, bennySt);  // friend function
```

上述两种转换中，实参的类型都和形参匹配。另外，成员函数是通过 Stonewt 对象调用的。

其次：

```
total = jennySt + kennyD;
```

被转换为：

```
total = jennySt.operator+ (kennyD);   // member function
```

或：

```
total = operator+ (jennySt, kennyD);  // friend function
```

同样，成员函数也是通过 Stonewt 对象调用的。这一次，每个调用中都有一个参数（kennyD）是 double 类型的，因此将调用 Stonewt（double）构造函数，将该参数转换为 Stonewt 对象。

另外，在这种情况下，如果定义了 operator double()成员函数，将造成混乱，因为该函数将提供另一种解释方式。编译器不是将 kennyD 转换为 double 并执行 Stonewt 加法，而是将 jennySt 转换为 double 并执行 double 加法。过多的转换函数将导致二义性。

最后：

```
total = pennyD + jennySt;
```

被转换为：

```
total = operator+ (pennyD, jennySt);  // friend function
```

其中，两个参数都是 double 类型，因此将调用 Stonewt（double）构造函数，将它们转换为 Stonewt 对象。不过，不能调用成员函数：

```
total = pennyD.operator+ (jennySt);   // not meaningful
```

原因是只有类对象才可以调用成员函数。C++不会试图将 pennyD 转换为 Stonewt 对象。将对成员函数参数进行转换，而不是调用成员函数的对象。

这里的经验是，将加法定义为友元可以让程序更容易适应自动类型转换。原因在于，两个操作数都成为函数参数，因此与函数原型匹配。

实现加法时的选择

要将 double 量和 Stonewt 量相加，有两种选择。第一种方法是（我们刚刚介绍过）将 operator +（const Stonewt &，const Stonewt &）定义为友元函数，让 Stonewt（double）构造函数将 double 类型的参数转换为 Stonewt 类型的参数。

第二种方法是，将加法操作符重载为一个显式使用 double 类型参数的函数：

```
Stonewt operator+ (double x);  // member function
friend Stonewt operator+ (double x, Stonewt & s);
```

这样，下面的语句：

```
total = jennySt + kennyD; // Stonewt + double
```

将与 operator +（double x）成员函数完全匹配，而下面的语句：

```
total = pennyD + jennySt; // double + Stonewt
```

将与友元函数 operator +（double x，Stonewt & s）完全匹配。前面我们对 Vector 乘法做了类似的处理。

每一种方法都有其优点。第一种方法（依赖于隐式转换）使程序更简短，因为定义的函数较少。这也意味程序员需要完成的工作较少，出错的机会较小。这种方法的缺点是，每次需要转换时，都将调用转换构造函数，这增加时间和内存开销。第二种方法（增加一个显式地匹配类型的函数）则正好相反。它使程

序较长，程序员需要完成的工作更多，但运行速度较快。

如果程序经常需要将 double 值与 Stonewt 对象相加，则重载加法更合适；如果程序只是偶尔使用这种加法，则依赖于自动转换更合适，或者为了更保险，可以使用显式转换。

在 main() 之前调用 Bootstrap 函数

在所有可执行程序中，调用的第一个函数通常都是其入口 main()，但可以使用一些技巧来修改这种行为。例如，考虑一个协调高尔夫俱乐部运作的时间安排程序。一般来说，程序首先需要获取来自各方面的信息，以精确地安排高尔夫俱乐部每天的运作情况，所以可能需要首先调用“bootstrap”函数，为 main() 做准备

全局对象（即具有文件作用域的对象）能满足这种要求，因为全局对象将在程序的 main() 函数被调用之前创建　程序员可以创建一个类，其默认构造函数将调用所有的 bootstrap 函数。例如，它们初始化对象的不同数据部分。这样便可以创建一个全局对象。下面的代码说明了这种技术。

```
class CompileRequirements
{
   private:
   // essential information
   public:
   CompileRequirements()        // default constructor
   {
     GetDataFromSales();              // various
     GetDataFromManufacturing();      // bootstrap
     GetDataFromFinance();            // functions
   }
};

//Instance of Req class has global scope
CompileRequirements Req;     // uses default constructor

int main(void)
{
   // Read Req and build schedule
   BuildScheduleFromReq();

   //
   // rest of program code
   //
}
```

11.7　总结

本章介绍了定义和使用类的许多重要方面，其中的一些内容可能较难理解，但随着实践经验的不断增加，读者将逐渐掌握它们。

一般来说，访问私有类成员的惟一方法是使用类方法。C++使用友元函数来避开这种限制。要让函数成为友元，需要在类声明中声明该函数，并在声明前加上关键字 **friend**。

C++扩展了对操作符的重载，允许自定义特殊的操作符函数，这种函数描述了特定的操作符与类之间的关系。操作符函数可以是类成员函数，也可以是友元函数（有一些操作符函数只能是类成员函数）。要调用操作符函数，可以直接调用该函数，也可以以通常的句法使用被重载的操作符。对于操作符 *op*，其操作符函数的格式如下：

```
operatorop(argument-list)
```

argument-list 表示该操作符的操作数。如果操作符函数是类成员函数，则第一个操作数是调用对象，它不在 argument-list 中。例如，本章通过为 Vector 类定义 operator+() 成员函数来重载加法。如果 up、right

和 result 都是 Vector 对象，则可以使用下面的任何一条语句来调用矢量加法：

```
result = up.operator+(right);
result = up + right;
```

在第二条语句中，由于操作数 up 和 right 的类型都是 Vector，因此 C++将使用 Vector 的加法定义。

当操作符函数是成员函数时，则第一个操作数将是调用该函数的对象。例如，在前面的语句中，up 对象是调用函数的对象。定义操作符函数时，如果要使其第一个操作数不是类对象，则必须使用友元函数。这样就可以将操作数按所需的顺序传递给函数了。

最常见的操作符重载任务之一是定义<<操作符，使之可与 cout 一起使用，来显示对象的内容。要让 ostream 对象成为第一个操作数，需要将操作符函数定义为友元；要使重新定义的操作符能与其自身拼接，需要将返回类型声明为 ostream &。下面的通用格式能够满足这种要求：

```
ostream & operator<<(ostream & os, const c_name & obj)
{
    os << ...; // display object contents
    return os;
}
```

不过，如果类包含这样的方法，它返回需要显示的数据成员的值，则可以使用这些方法，无需在 operator<<()中直接访问这些成员。在这种情况下，函数不必（也不应当）是友元。

C++允许指定在类和基本类型之间进行转换的方式。首先，任何接受惟一一个参数的构造函数都可被用作转换函数，将类型与该参数相同的值转换为类。如果将类型与该参数相同的值赋给对象，则 C++将自动调用该构造函数。例如，假设有一个 String 类，它包含一个将 char *值作为其惟一参数的构造函数，那么如果 bean 是 String 对象，则可以使用下面的语句：

```
bean = "pinto";   // converts type char * to type String
```

不过，如果在该构造函数的声明前加上了关键字 explicit，则该构造函数将只能用于显式转换：

```
bean = String("pinto");   // converts type char * to type String explicitly
```

要将类对象转换为其他类型，必须定义转换函数，指出如何进行这种转换。转换函数必须是成员函数。将类对象转换为 typeName 类型的转换函数的原型如下：

```
operator typeName();
```

注意，转换函数没有返回类型、没有参数，但必须返回转换后的值（虽然没有声明返回类型）。例如，下面是将 Vector 转换为 double 类型的函数：

```
Vector::operator double()
{
    ...
    return a_double_value;
}
```

经验表明，最好不要依赖于这种隐式转换函数。

读者可能已经注意到了，与简单的 C 风格结构相比，使用类时，必须更谨慎、更小心，但作为补偿，它们为我们完成的工作也更多。

11.8 复习题

1. 使用成员函数为 Stonewt 类重载乘法操作符，该操作符将数据成员与 double 类型的值相乘。注意，用英石和磅表示时，需要进位。也就是说，将 10 英石 8 磅乘以 2 等于 21 英石 2 磅。
2. 友元函数与成员函数之间的区别是什么？
3. 非成员函数必须是友元才能访问类成员吗？
4. 使用友元函数为 Stonewt 类重载乘法操作符，该操作符将 double 值与 Stone 值相乘。
5. 哪些操作符不能重载？
6. 在重载操作符=、()、[]和->时，有什么限制？
7. 为 Vector 类定义一个转换函数，将 Vector 类转换为一个 double 类型的值，后者表示矢量的长度。

11.9　编程练习

1. 修改程序清单 11.5，使之将一系列连续的随机漫步者位置写入到文件中。对于每个位置，用步号进行标示。另外，让该程序将初始条件（目标距离和步长）以及结果小结写入到该文件中。该文件的内容与下面类似：

```
Target Distance: 100, Step Size: 20
0: (x,y) = (0, 0)
1: (x,y) = (-11.4715, 16.383)
2: (x,y) = (-8.68807, -3.42232)
...
26: (x,y) = (42.2919, -78.2594)
27: (x,y) = (58.6749, -89.7309)
After 27 steps, the subject has the following location:
(x,y) = (58.6749, -89.7309)
  or
(m,a) = (107.212, -56.8194)
Average outward distance per step = 3.97081
```

2. 对 Vector 类的头文件（程序清单 11.13）和实现文件（程序清单 11.14）进行修改，使其不再存储矢量的长度和角度，而是在 magval() 和 angval() 被调用时计算它们。

应保留公有接口不变（公有方法及其参数不变），但对私有部分（包括一些私有方法）和方法实现进行修改。然后，使用程序清单 11.15 对修改后的版本进行测试，结果应该与以前相同，因为 Vector 类的公有接口与原来相同。

3. 修改程序清单 11.15，使之报告 *N* 次测试中的最高、最低和平均步数（其中 *N* 是用户输入的整数），而不是报告每次测试的结果。

4. 重新编写最后的 Time 类范例（程序清单 11.10、11.11 和 11.12），使用友元函数来实现所有的重载操作符。

5. 重新编写 Stonewt 类（程序清单 11.16 和 11.17），使它有一个状态成员，由该成员控制对象应转换为英石格式、整数磅格式还是浮点磅格式。重载<<操作符，使用它来替换 show_stn() 和 show_lbs() 方法。重载加法、减法和乘法操作符，以便可以对 Stonewt 值进行加、减、乘运算。编写一个使用所有类方法和友元的小程序，来测试这个类。

6. 重新编写 Stonewt 类（程序清单 11.16 和 11.17），重载全部 6 个关系操作符。操作符对 pounds 成员进行比较，并返回一个 bool 值。编写一个程序，它声明一个包含 6 个 Stonewt 对象的数组，并在数组声明中初始化前 3 个对象。然后使用循环来读取用于设置剩余 3 个数组元素的值。接着报告最小的元素、最大的元素以及大于或等于 11 英石的元素的数量（最简单的方法是创建一个 Stonewt 对象，并将其初始化为 11 英石，然后将其同其他对象进行比较）。

7. 复数由两个部分组成：实数部分和虚数部分。复数的一种书写方式是：(3.0, 4.0)，其中，3.0 是实数部分，4.0 是虚数部分。假设 a = (A, Bi)，c = (C, Di)，则下面是一些复数运算：

- 加法：a + c = (A+C, (B+D) i)。
- 减法：a – c = (A-C, (B-D) i)。
- 乘法：a * c = (A*C-B*D, (A*D + B*C) i)。
- 乘法：x*c = (x * C, x *Di)，其中 x 为实数。
- 共轭：~a = (A, -Bi)。

请定义一个复数类，以便下面的程序可以使用它来获得正确的结果。

```
#include <iostream>
using namespace std;
#include "complex0.h"  // to avoid confusion with complex.h
int main()
```

```
{
    complex a(3.0, 4.0);   // initialize to (3, 4i)
    complex c;
    cout << "Enter a complex number (q to quit):\n";
    while (cin >> c)
    {
        cout << "c is "<< c << '\n';
        cout << "complex conjugate is " << ~c << '\n';
        cout << "a is " << a << '\n';
        cout << "a + c is " << a + c << '\n';
        cout << "a - c is " << a - c << '\n';
        cout << "a * c is " << a * c << '\n';
        cout <<"2 * c is " << 2 * c << '\n';
        cout << "Enter a complex number (q to quit):\n";
    }
    cout << "Done!\n";
    return 0;
}
```

注意，必须重载<<和>>操作符。许多系统都在头文件 complex.h 提供了对复数的支持，因此应将自定义的头文件命名为 complex0.h，以免发生冲突。应尽可能使用 const。

下面是该程序的运行情况。

```
Enter a complex number (q to quit):
real: 10
imaginary: 12
c is (10, 12i)
complex conjugate is (10, -12i)
a is (3, 4i)
a + c is (13, 16i)
a - c is (-7, -8i)
a * c is (-18, 76i)
2 * c is (20, 24i)
Enter a complex number (q to quit):
real: q
Done!
```

请注意，经过重载后，cin >>c 将提示用户输入实数和虚数部分。

第 12 章　类和动态内存分配

本章内容包括：

- 对类成员使用动态内存分配。
- 隐式和显式地复制构造函数。
- 隐式和显式地重载赋值操作符。
- 在构造函数中使用 new 所必须完成的工作。
- 使用静态类成员。
- 将布局 new 操作符用于对象。
- 使用指向对象的指针。
- 实现队列抽象数据类型（ADT）。

本章将介绍如何对类使用 new 和 delete 以及如何处理由于使用动态内存而引起的一些微妙的问题。这里涉及的主题好像不多，但它们将影响构造函数和析构函数的设计以及操作符的重载。

让我们看一个具体的例子——C++如何增加内存负载。假设要创建一个类，其中一个成员表示某人的姓。最简单的方法是使用字符数组成员来保存姓，但这种方法有一些缺陷。开始也许会使用一个 14 个字符的数组，然后发现数组太小，更保险的方法是，使用一个 40 个字符的数组。但是，如果创建包含 2000 个这种对象的数组，就会由于字符数组只有部分被使用而浪费大量的内存（在这种情况下，增加了计算机的内存负载）。不过可以采取另外一种方法。

通常，最好是在程序运行时（而不是编译时）确定诸如使用多少内存等问题。对于在对象中保存姓名来说，通常的 C++方法是，在类构造函数中使用 new 操作符来在程序运行时分配所需的内存。但是，除非同时执行一系列的额外步骤，如扩展类析构函数、使所有的构造函数与 new 析构函数协调一致、编写额外的类方法来帮助正确完成初始化和赋值（当然，本章将介绍这些步骤），否则，在类构造函数中使用 new 将导致新的问题。对于 C++初学者来说，最好先使用这种简单的字符数组方法，然后在设计好类后回到 OOP 方式，使用 new 来改善类声明。简而言之，应循序渐进地掌握 C++。

12.1　动态内存和类

您希望下个月的早餐、午餐和晚餐吃些什么？在第 3 天的晚餐喝多少盎司的牛奶？在第 15 天的早餐中需要在谷类食品添加多少葡萄干？如果您与大多数人的习惯相似，就会等到进餐时再做决定。C++在分配内存时采取的部分策略与此相同，让程序在运行时决定内存分配，而不是在编译时决定。这样，可根据程序的需要，而不是根据一系列严格的存储类型规则来使用内存。C++使用 new 和 delete 操作符来动态控制内存。遗憾的是，在类中使用这些操作符将导致许多新的编程问题。在这种情况下，析构函数将是必不可少的，而不再是可有可无的。有时候，还必须重载赋值操作符，以保证程序正常运行。下面来看一看这些问题。

12.1.1 复习范例和静态类成员

我们已经有一段时间没有使用 new 和 delete 了，所以这里使用一个小程序来复习一下它们。这个程序使用了一个新的存储类型：静态类成员。首先设计一个 StringBad 类，然后设计一个功能稍强的 String 类（本书前面介绍过 C++标准 String 类，第 16 章将更深入地讨论它；而本章的 StringBad 和 String 类将介绍这个类的底层结构，提供这种友好的接口涉及大量的编程技术）。

StringBad 和 String 类对象将包含一个字符串指针和一个表示字符串长度的值。这里使用 StringBad 和 String 类，主要是为了深入了解 new、delete 和静态类成员的工作原理。因此，构造函数和析构函数调用时将显示一些消息，以便读者能够按照提示来完成操作。同时，我们也将省略一些有用的成员和友元函数，例如重载的++和>>操作符以及转换函数，以简化类接口（但是本章的复习题将要求您添加这些函数）。程序清单 12.1 列出了这个类的声明。

程序清单 12.1 strngbad.h

```
// strngbad.h -- flawed string class definition
#include <iostream>
#ifndef STRNGBAD_H_
#define STRNGBAD_H_
class StringBad
{
private:
    char * str;                    // pointer to string
    int len;                       // length of string
    static int num_strings;        // number of objects
public:
    StringBad(const char * s); // constructor
    StringBad();                   // default constructor
    ~StringBad();                  // destructor
// friend function
    friend std::ostream & operator<<(std::ostream & os,
                    const StringBad & st);
};
#endif
```

为何将这个类命名为 StringBad 呢？这旨在告诉读者，这是一个不太完整的类。它是使用动态内存分配来开发类的第一个阶段，正确地完成了一些显而易见的工作，例如，在构造函数和析构函数中正确地使用了 new 和 delete。这个类并没有什么错误，但忽略了一些不明显却必不可少的东西。通过了解这个类存在的问题，将有助于读者理解并记住后面将其转换为功能更强大的 String 类时，所做的不明显的修改。

对这个声明，需要注意的有两点。首先，它使用 char 指针（而不是 char 数组）来表示姓名。这意味着类声明没有为字符串本身分配存储空间，而是在构造函数中使用 new 来为字符串分配空间。这避免了在类声明中预先定义字符串的长度。

其次，将 num_strings 成员声明为静态存储类。静态类成员有一个特点：无论创建了多少对象，程序都只创建一个静态类变量副本。也就是说，类的所有对象共享同一个静态成员，就像家中的电话可供全体家庭成员共享一样。假设创建了 10 个 StringBad 对象，将有 10 个 str 成员和 10 个 len 成员，但只有一个共享的 num_strings 成员（参见图 12.1）。这对于所有类对象都具有相同值的类私有数据是非常方便的。例如，num_strings 成员可以记录所创建的对象数目。

顺便说一句，程序清单 21.1 使用 num_strings 成员，只是为了方便说明静态数据成员，并指出潜在的编程问题，字符串类通常并不需要这样的成员。

来看一看程序清单 12.2 中的类方法实现，它演示了如何使用指针和静态成员。

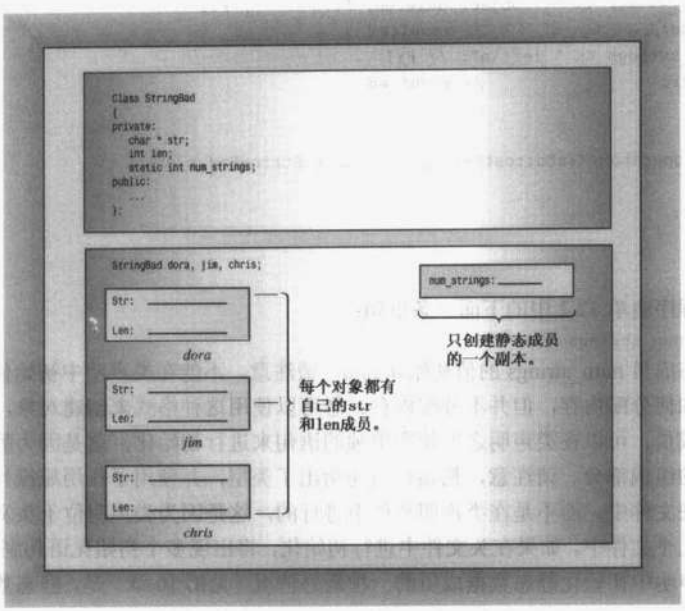


图 12.1　静态数据成员

程序清单 12.2　strngbad.cpp

```
// strngbad.cpp -- StringBad class methods
#include <cstring>                  // string.h for some
#include "strngbad.h"
using std::cout;

// initializing static class member
int StringBad::num_strings = 0;

// class methods

// construct StringBad from C string
StringBad::StringBad(const char * s)
{
    len = std::strlen(s);                   // set size
    str = new char[len + 1];                // allot storage
    std::strcpy(str, s);                    // initialize pointer
    num_strings++;                          // set object count
    cout << num_strings << ": \"" << str
    << "\" object created\n";               // For Your Information
}

StringBad::StringBad()                      // default constructor
{
    len = 4;
    str = new char[4];
    std::strcpy(str, "C++");                // default string
    num_strings++;
    cout << num_strings << ": \"" << str
        << "\" default object created\n";  // FYI
}

StringBad::~StringBad()             // necessary destructor
{
```

```
    cout << "\"" << str << "\" object deleted, ";       // FYI
    --num_strings;                  // required
    cout << num_strings << " left\n"; // FYI
    delete [] str;                  // required
}

std::ostream & operator<<(std::ostream & os, const StringBad & st)
{
    os << st.str;
    return os;
}
```

首先，请注意程序清单 12.2 中的下面一条语句：

```
int StringBad::num_strings = 0;
```

这条语句将静态成员 num_strings 的值初始化为 0。请注意，不能在类声明中初始化静态成员变量，这是因为声明描述了如何分配内存，但并不分配内存。您可以使用这种格式来创建对象，从而分配和初始化内存。对于静态类成员，可以在类声明之外使用单独的语句来进行初始化，这是因为静态类成员是单独存储的，而不是对象的组成部分。请注意，初始化语句指出了类型，并使用了作用域操作符。

初始化是在方法文件中，而不是在类声明文件中进行的，这是因为类声明位于头文件中，程序可能将头文件包括在其他几个文件中。如果在头文件中进行初始化，将出现多个初始化语句副本，从而引发错误。

对于不能在类声明中初始化静态数据成员的一种例外情况（见第 10 章）是，静态数据成员为整型或枚举型 const。

记住：静态数据成员在类声明中声明，在包含类方法的文件中初始化。初始化时使用作用域操作符来指出静态成员所属的类。但如果静态成员是整型或枚举型 const，则可以在类声明中初始化。

每个构造函数都包含表达式 num_strings++，这确保程序每创建一个新对象，共享变量 num_strings 的值都将增加 1，从而记录 String 对象的总数。另外，析构函数包含表达式--num_strings，因此 String 类也将跟踪对象被删除的情况，从而使 num_string 成员的值是最新的。

现在来看程序清单 12.2 中的第一个构造函数，它使用一个常规 C 字符串来初始化 String 对象：

```
StringBad::StringBad(const char * s)
{
    len = std::strlen(s);          // set size
    str = new char[len + 1];       // allot storage
    std::strcpy(str, s);           // initialize pointer
    num_strings++;                 // set object count
    cout << num_strings << ": \"" << str
       << "\" object created\n"; // For Your Information
}
```

类成员 str 是一个指针，因此构造函数必须提供内存来存储字符串。初始化对象时，可以给构造函数传递一个字符串指针：

```
String boston ("Boston");
```

构造函数必须分配足够的内存来存储字符串，然后将字符串复制到内存中。下面介绍其中的每一个步骤。

首先，使用 strlen()函数计算字符串的长度，并对 len 成员进行初始化。接着，使用 new 分配足够的空间来保存字符串，然后将新内存的地址赋给 str 成员（strlen()返回字符串长度，但不包括末尾的空字符，因此构造函数将 len 加 1，使分配的内存能够存储包含空字符的字符串）。

接着，构造函数使用 strcpy()将传递的字符串复制到新的内存中，并更新对象计数。最后，构造函数显示当前的对象数目和当前对象中存储的字符串，以助于掌握程序运行情况。稍后当我们故意使 String 类出错时，该特性将派上用场。

要理解这种方法，必须知道字符串并不保存在对象中。字符串单独保存在堆内存中，对象仅保存了指出到哪里去查找字符串的信息。

不能这样做：

```
str = s;  // not the way to go
```

这里只保存了地址，而没有创建字符串副本。

默认构造函数与此相似，不过它提供了一个默认字符串："C++"。

析构函数中包含了范例中对处理类来说最重要的东西：

```
StringBad::~StringBad()                              // necessary destructor
{
   cout << "\"" << str << "\" object deleted. ";     // FYI
   --num_strings;                                    // required
   cout << num_strings << " left\n";                 // FYI
   delete [] str;                                    // required
}
```

析构函数首先指出自己何时被调用。这部分包含了丰富的信息，但并不是必不可少的。不过，delete 语句却是至关重要的。str 成员指向 new 分配的内存。当 StringBad 对象过期时，str 指针也将过期。但 str 指向的内存仍被分配，除非使用 delete 将其释放。删除对象可以释放对象本身占用的内存，但并不能自动释放属于对象成员的指针指向的内存。因此，必须使用析构函数。在析构函数中使用 delete 语句可确保对象过期时，由构造函数使用 new 分配的内存被释放。

记住：在构造函数中使用 new 来分配内存时，必须在相应析构函数中使用 delete 来释放内存。如果使用 new[ ]（包括中括号）来分配内存，则应使用 delete[ ]（包括中括号）来释放内存。

程序清单 12.3 是从处于开发阶段的 Daily Vegetable 程序中摘录出来的，演示了 StringBad 的构造函数和析构函数何时运行以及如何运行。请务必将程序清单 12.2 和程序清单 12.3 一起编译。

程序清单 12.3　vegnews.cpp

```
// vegnews.cpp -- using new and delete with classes
// compile with strngbad.cpp
#include <iostream>
using std::cout;
#include "strngbad.h"

void callme1 (StringBad &);  // pass by reference
void callme2 (StringBad);    // pass by value

int main()
{
   using std::endl;
   StringBad headline1 ("Celery Stalks at Midnight");
   StringBad headline2 ("Lettuce Prey");
   StringBad sports ("Spinach Leaves Bowl for Dollars");
   cout << "headline1: " << headline1 << endl;
   cout << "headline2: " << headline2 << endl;
   cout << "sports: "<< sports << endl;
   callme1 (headline1);
   cout << "headline1: " << headline1 << endl;
   callme2 (headline2);
   cout << "headline2: " << headline2 << endl;
   cout <<"Initialize one object to another: \n";
   StringBad sailor = sports;
   cout << "sailor: " << sailor << endl;
   cout << "Assign one object to another: \n";
   StringBad knot;
   knot = headline1;
   cout << "knot: " << knot << endl;
   cout <<"End of main()\n";

   return 0;
```

```
}

void callme1 (StringBad & rsb)
{
    cout << "String passed by reference: \n";
    cout << "    \"" << rsb << "\"\n";
}

void callme2 (StringBad sb)
{
    cout << "String passed by value: \n";
    cout << "    \"" << sb << "\"\n";
}
```

注意：StringBad 的第一个版本有许多故意留下的缺陷，这些缺陷使得输出是不确定的。例如，有些编译器无法编译它。虽然输出的具体内容有所差别，但基本问题和解决方法（稍后将介绍）是相同的。

下面是使用 Borland C++5.5 命令行编译器进行编译时，该程序的输出：

```
1: "Celery Stalks at Midnight" object created
2: "Lettuce Prey" object created
3: "Spinach Leaves Bowl for Dollars" object created
headline1: Celery Stalks at Midnight
headline2: Lettuce Prey
sports: Spinach Leaves Bowl for Dollars
String passed by reference:
    "Celery Stalks at Midnight"
headline1: Celery Stalks at Midnight
String passed by value:
    "Lettuce Prey"
"Lettuce Prey" object deleted, 2 left
headline2: D o
Initialize one object to another:
sailor: Spinach Leaves Bowl for Dollars
Assign one object to another:
3: "C++" default object created
knot: Celery Stalks at Midnight
End of main()
"Celery Stalks at Midnight" object deleted, 2 left
"Spinach Leaves Bowl for Dollars" object deleted, 1 left
"Spinach Leaves Bowl for Doll8" object deleted, 0 left
"@g" object deleted, -1 left
"-!" object deleted, -2 left
```

输出中出现的各种非标准字符随系统而异，这些字符表明，StringBad 类名副其实（是一个糟糕的类）。另一种迹象是对象计数为负。在使用较新的编译器和操作系统的机器上运行时，该程序通常会在显示有关还有-1 个对象的信息之前中断，而有些这样的机器将报告通用保护错误（GPF）。GPF 表明程序试图访问禁止它访问的内存单元，这是另一种糟糕的信号。

程序说明

程序清单 12.3 中的程序开始时还是正常的，但逐渐变得异常，最终导致了灾难性结果。首先来看正常的部分。构造函数指出自己创建了 3 个 StringBad 对象，并为这些对象进行了编号，然后程序使用重载操作符 >> 列出了这些对象：

```
1: "Celery Stalks at Midnight" object created
2: "Lettuce Prey" object created
3: "Spinach Leaves Bowl for Dollars" object created
headline1: Celery Stalks at Midnight
headline2: Lettuce Prey
sports: Spinach Leaves Bowl for Dollars
```

然后，程序将 headline1 传递给 cllme1()函数，并在调用后重新显示 headline1。代码如下：

```
callme1 (headline1) ;
cout << "headline1: " << headline1 << endl;
```

下面是运行结果：

```
String passed by reference:
   "Celery Stalks at Midnight"
headline1: Celery Stalks at Midnight
```

这部分代码看起来也是正常的。

但是随后程序执行了如下代码：

```
callme2 (headline2) ;
cout << "headline2: " << headline2 << endl;
```

这里，callme2()按值（而不是按引用）传递 headline2，结果表明这是一个严重的问题！

```
String passed by value:
   "Lettuce Prey"
"Lettuce Prey" object deleted, 2 left
headline2: D o
```

首先，将 headline2 作为函数参数来传递从而导致析构函数被调用。其次，虽然按值传递可以防止原始参数被修改，但实际上函数已使原始字符串无法识别，导致显示一些非标准字符（显示的具体内存取决于内存中包含的内容）。

请看输出结果，在为每一个创建的对象自动调用析构函数时，情况更糟糕：

```
End of main()
"Celery Stalks at Midnight" object deleted, 2 left
"Spinach Leaves Bowl for Dollars" object deleted, 1 left
"Spinach Leaves Bowl for Doll8" object deleted, 0 left
"@g" object deleted, -1 left
"-|" object deleted, -2 left
```

因为自动存储对象被删除的顺序与创建顺序相反，所以最先删除的 3 个对象是 knots、sailor 和 sport。删除 knots 和 sailor 时是正常的，但在删除 sport 时，Dollars 变成了 Doll8。对于 sport，程序只使用它来初始化 sailor，但这种操作修改了 sport。最后被删除的两个对象（headline2 和 headline1）已经无法识别。这些字符串在被删除之前，有些操作将它们搞乱了。另外，计数也很奇怪，如何会余下-2 个对象呢？

实际上，计数异常是一条线索。因为每个对象被构造和析构一次，因此调用构造函数的次数应当与析构函数的调用次数相同。对象计数（num_strings）递减的次数比递增次数多 2，这表明使用了不将 num_string 递增的构造函数创建了两个对象。类定义声明并定义了两个构造函数（这两个构造函数都使 num_string 递增），但结果表明程序使用了 3 个构造函数。例如，请看下面的代码：

```
StringBad sailor = sports;
```

这使用的是哪个构造函数呢？不是默认构造函数，也不是参数为 const char *的构造函数。记住，这种形式的初始化等效于下面的语句：

```
StringBad sailor = StringBad (sports) ; //constructor using sports
```

因为 sports 的类型为 StringBad，因此相应的构造函数原型应该如下：

```
StringBad (const StringBad &) ;
```

当您使用一个对象来初始化另一个对象时，编译器将自动生成上述构造函数（称为复制构造函数，因为它创建对象的一个副本）。自动生成的构造函数不知道需要更新静态变量 num_string，因此会将计数方案搞乱。实际上，这个例子说明的所有问题都是由编译器自动生成的成员函数引起的，下面介绍这一主题。

12.1.2　隐式成员函数

StringBad 类中的问题是由自动定义的隐式成员函数引起的，这种函数的行为与类设计不符。具体来说，C++自动提供了下面这些成员函数：

- 默认构造函数，如果没有定义构造函数。
- 复制构造函数，如果没有定义。

- 赋值操作符，如果没有定义。
- 默认析构函数，如果没有定义。
- 地址操作符，如果没有定义。

更准确地说，编译器将生成上述最后 4 个函数的定义——如果程序使用对象的方式要求这样做。例如，如果您将一个对象赋给另一个对象，编译器将提供赋值操作符的定义。

结果表明，StringBad 类中的问题是由隐式复制构造函数和隐式赋值操作符引起的。

隐式地址操作符返回调用对象的地址（即 this 指针的值）。这与我们的初衷是一致的，在此将不详细讨论该成员函数。默认析构函数不执行任何操作，因此这里也不讨论它，不过需要指出的是，这个类已经提供默认构造函数。至于其他成员函数还需要进一步讨论。

1. 默认构造函数

如果没有提供任何构造函数，C++将创建默认构造函数。例如，假如定义了一个 Klunk 类，但没有提供任何构造函数，则编译器将提供下述默认构造函数：

```
Klunk::Klunk(){ }  // implicit default constructor
```

也就是说，编译器将提供一个不接受任何参数，也不执行任何操作的构造函数，这是因为创建对象时总是会调用构造函数：

```
Klunk lunk;  // invokes default constructor
```

默认构造函数使 Lunk 类似于一个常规的自动变量，也就是说，它的值在初始化时是未知的。

如果定义了构造函数，C++将不会定义默认构造函数。如果希望在创建对象时显式地对它进行初始化，或需要创建对象数组时，则必须显式地定义默认构造函数。这种构造函数没有任何参数，但可以使用它来设置特定的值：

```
Klunk::Klunk() // explicit default constructor
{
   klunk_ct = 0;
   ...
}
```

带参数的构造函数也可以是默认构造函数，只要所有参数都有默认值。例如，Klunk 类可以包含下述内联构造函数：

```
Klunk(int n = 0) { klunk_ct = n; }
```

但只能有一个默认构造函数。也就是说，不能这样做：

```
Klunk(){ klunk_ct = 0 }              // constructor #1
Klunk(int n = 0) { klunk_ct = n; }   // ambiguous constructor #2
```

这为何有二义性呢？请看下面两个声明：

```
Klunk kar(10);    // clearly matches Klunt(int n)
Klunk bus;        // could match either constructor
```

第二个声明既与构造函数 1（没有参数）匹配，也与构造函数 2（使用默认参数 0）匹配。这将导致编译器发出一条错误消息。

2. 复制构造函数

复制构造函数用于将一个对象复制到新创建的对象中。也就是说，它用于初始化过程中，而不是常规的赋值过程中。类的复制构造函数原型通常如下：

```
Class_name(const Class_name &);
```

它接受一个指向类对象的常量引用作为参数。例如，String 类的复制构造函数的原型如下：

```
StringBad(const StringBad &);
```

对于复制构造函数，需要知道两点：何时调用和有何功能。

（1）何时调用复制构造函数

新建一个对象并将其初始化为同类现有对象时，复制构造函数都将被调用。这在很多情况下都可能发生，最常见的情况是将新对象显式地初始化为现有的对象。例如，假设 motto 是一个 StringBad 对象，则下面 4 种声明都将调用复制构造函数：

```
StringBad ditto (motto);  // calls StringBad (const StringBad &)
StringBad metoo = motto; // calls StringBad (const StringBad &)
StringBad also = StringBad (motto);
                  // calls StringBad (const StringBad &)
StringBad * pStringBad = new StringBad (motto);
                  // calls StringBad (const StringBad &)
```

其中中间的 2 种声明可能会使用复制构造函数直接创建 metoo 和 also，也可能使用复制构造函数生成一个临时对象，然后将临时对象的内容赋给 metoo 和 also，这取决于具体的实现。最后一种声明使用 motto 初始化一个匿名对象，并将新对象的地址赋给 pstring 指针。

每当程序生成了对象副本时，编译器都将使用复制构造函数。具体地说，当函数按值传递对象（如程序清单 12.3 中的 callme2()）或函数返回对象时，都将使用复制构造函数。记住，按值传递意味着创建原始变量的一个副本。编译器生成临时对象时，也将使用复制构造函数。例如，将 3 个 Vector 对象相加时，编译器可能生成临时的 Vector 对象来保存中间结果。何时生成临时对象随编译器而异，但无论是哪种编译器，当按值传递和返回对象时，都将调用复制构造函数。具体地说，程序清单 12.3 中的函数调用将调用下面的复制构造函数：

```
callme2 (headline2) ;
```

程序使用复制构造函数初始化 sb——callme2()函数的 StringBad 型形参。

由于按值传递对象将调用复制构造函数，因此应该按引用传递对象。这样可以节省调用构造函数的时间以及存储新对象的空间。

（2）复制构造函数的功能

默认的复制构造函数逐个复制非静态成员（成员复制也称为浅复制），复制的是成员的值。在程序清单 12.3 中，下述语句：

```
StringBad sailor = sports;
```

与下面的代码等效（只是由于私有成员是无法访问的，因此这些代码不能通过编译）：

```
StringBad sailor;
sailor.str = sports.str;
sailor.len = sports.len;
```

如果成员本身就是类对象，则将使用这个类的复制构造函数来复制成员对象。静态函数（如 num_strings）不受影响，因为它们属于整个类，而不是各个对象。图 12.2 说明了隐式复制构造函数执行的操作。

3. 复制构造函数的哪里出了问题

现在介绍程序清单 12.3 的两个异常之处（假设输出为该程序清单后面列出的）。首先，程序的输出表明，析构函数的调用次数比构造函数的调用次数多 2，原因可能是程序确实使用默认的复制构造函数另外创建了两个对象。当 callme2()被调用时，复制构造函数被用来初始化 callme2()的形参，还被用来将对象 sailor 初始化为对象 sports。默认的复制构造函数不说明其行为，因此它不指出创建过程，也不增加计数器 num_strings 的值。但析构函数更新了计数，并且在任何对象过期时都将被调用，而不管对象是如何被创建的。这是一个问题，因为这意味着程序无法准确地记录对象计数。解决办法是提供一个对计数进行更新的显式复制构造函数：

```
String::String (const String & s)
{
   num_strings++;
   ...// important stuff to go here
}
```

提示：如果类中包含这样的静态数据成员，即其值将在新对象被创建时发生变化，则应该提供一个显式复制构造函数来处理计数问题。

第二个异常之处更微妙，也更危险，其表现之一是字符串内容出现乱码：

```
headline2: D0²
```

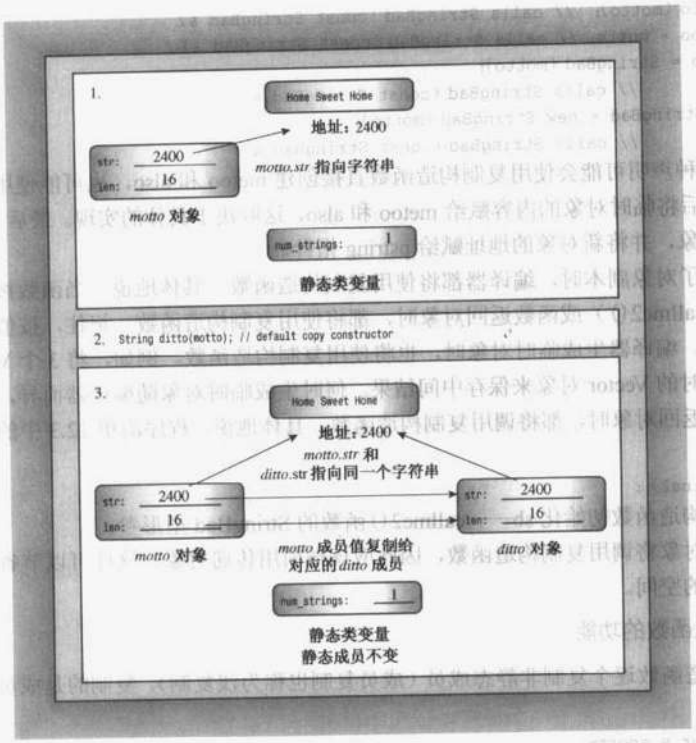


图 12.2 逐个复制成员

另一个表现是，使用许多编译器对其进行编译后，运行时将异常终止。例如 Microsoft Visual C++7.1（调试模式）显示一个错误消息窗口，指出调试断言失败，而 gpp 发出通用保护错误。其他系统可能提供不同的消息，甚至不提供任何消息，但程序中的错误是相同的。

原因在于隐式地复制构造函数是按值进行复制的。例如，对于程序清单 12.3，隐式地复制构造函数的功能相当于：

```
sailor.str = sport.str;
```

这里复制的并不是字符串，而是一个指向字符串的指针。也就是说，将 sailor 初始化为 sports 后，得到的是两个指向同一个字符串的指针。当 operator <<() 函数使用指针来显示字符串时，这并不会出现问题。但当析构函数被调用时，这将引发问题。析构函数 StringBad 释放 str 指针指向的内存，因此释放 sailor 的效果如下：

```
delete [] sailor.str;   // delete the string that ditto.str points to
```

sailor.str 指针指向“Spinach Leaves Bowl for Dollars”，因为它被赋值为 sports.str，而 sports.str 指向的正是上述字符串。所以 delete 语句将释放字符串“Spinach Leaves Bowl for Dollars”占用的内存。

然后，释放 sports 的效果如下：

```
delete [] sports.str;  // effect is undefined
```

sports.str 指向的内存已经被 sailor 的析构函数释放，这将导致不确定的、可能有害的后果。程序清单 12.3 中的程序生成受损的字符串，这通常是内存管理不善的表现。

4. 使用显式复制构造函数来解决问题

解决类设计中这种问题的方法是进行深度复制（deep copy）。也就是说，复制构造函数应当复制字符串并将副本的地址赋给 str 成员，而不仅仅是复制字符串地址。这样每个对象都有自己的字符串，而不是引用另一个对象的字符串。调用析构函数时都将释放不同的字符串，而不会试图去释放已经被释放的字符串。可以这样编写 String 的复制构造函数：

```
StringBad::StringBad (const StringBad & st)
{
    num_strings++;                  // handle static member update
    len = st.len;                   // same length
    str = new char [len + 1];       // allot space
    std::strcpy (str, st.str);      // copy string to new location
    cout << num_strings << ": \"" << str
        << "\" object created\n"; // For Your Information
}
```

必须定义复制构造函数的原因在于，一些类成员是使用 new 初始化的、指向数据的指针，而不是数据本身。图 12.3 说明了深度复制。

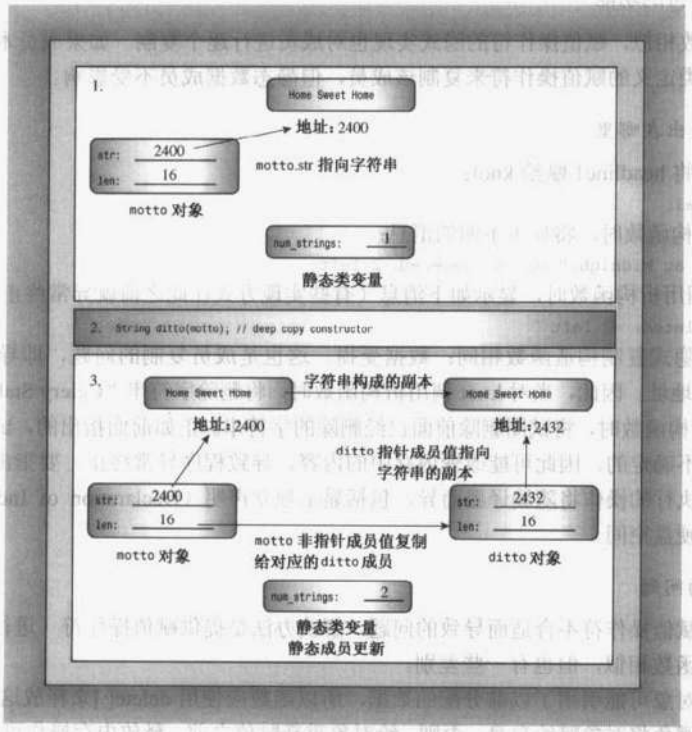


图 12.3　深度复制

警告：如果类中包含了使用 new 初始化的指针成员，应当定义一个复制构造函数，以复制指向的数据，而不是指针，这被称为深度复制。复制的另一种形式（成员复制或浅复制）只是复制指针值。浅复制仅浅浅地复制指针信息，而不会深入“挖掘”来复制指针引用的结构。

5. 赋值操作符

并不是程序清单 12.3 的所有问题都可以归咎于默认的复制构造函数，还需要看一看默认的赋值操作符。ANSI C 允许结构赋值，而 C++允许类对象赋值，这是通过自动为类重载赋值操作符实现的。这种操作符的原型如下：

```
Class_name & Class_name::operator= (const Class_name &);
```

它接受并返回一个指向类对象的引用。例如，StringBad 类的赋值操作符的原型如下：

```
StringBad & StringBad::operator= (const StringBad &);
```

（1）何时使用赋值操作符

将已有的对象赋给另一个对象时，将使用重载的赋值操作符：

```
StringBad headline1 ("Celery Stalks at Midnight");
...
StringBad knot;
knot = headline1;  // assignment operator invoked
```

初始化对象时，并不一定会使用赋值操作符：

```
StringBad metoo = knot; // use copy constructor, possibly assignment, too
```

这里，metoo 是一个新创建的对象，被初始化为 knot 的值，因此使用复制构造函数。不过，正如前面指出的，实现时也可能分两步来处理这条语句：使用复制构造函数创建一个临时对象，然后通过赋值将临时对象的值复制到新对象中。这就是说，初始化总是会调用复制构造函数，而使用=操作符时也可能调用赋值操作符。

（2）赋值操作符的功能

与复制构造函数相似，赋值操作符的隐式实现也对成员进行逐个复制。如果成员本身就是类对象，则程序将使用为这个类定义的赋值操作符来复制该成员，但静态数据成员不受影响。

6. 赋值的问题出在哪里

程序清单 12.3 将 headline1 赋给 knot：

```
knot = headline1;
```

为 knot 调用析构函数时，将显示下面的消息：

```
"Celery Stalks at Midnight" object deleted, 2 left
```

为 Headline1 调用析构函数时，显示如下消息（有些实现方式在此之前就异常终止了）：

```
"~|" object deleted, -2 left
```

出现的问题与隐式复制构造函数相同：数据受损。这也是成员复制的问题，即导致 headline1.str 和 knot.str 指向相同的地址。因此，当对 knot 调用析构函数时，将删除字符串“Celery Stalks at Midnight”；当对 headline1 调用析构函数时，将试图删除前面已经删除的字符串。正如前面指出的，试图删除已经删除的数据导致的结果是不确定的，因此可能改变内存中的内容，导致程序异常终止。要指出的是，如果操作结果是不确定的，则执行的操作将随编译器而异，包括显示独立声明（Declaration of Independence）或者释放隐藏文件占用的硬盘空间。

7. 解决赋值的问题

对于由于默认赋值操作符不合适而导致的问题，解决办法是提供赋值操作符（进行深度复制）定义。其实现与复制构造函数相似，但也有一些差别。

- 由于目标对象可能引用了以前分配的数据，所以函数应使用 delete[ ]来释放这些数据。
- 函数应当避免将对象赋给自身；否则，给对象重新赋值之前，释放内存操作可能删除对象的内容。
- 函数返回一个指向调用对象的引用。

通过返回一个对象，函数可以像常规赋值操作那样，连续进行赋值，即如果 S0、S1 和 S2 都是 StringBad 对象，则可以编写这样的代码：

```
S0 = S1 = S2;
```

使用函数表示法时，上述代码为：

```
S0.operator=(S1.operator=(S2));
```

因此，S1.operator=（S2）的返回值是函数 S0.operator=()的参数。

因为返回值是一个指向 StringBad 对象的引用，因此参数类型是正确的。

下面的代码说明了如何为 StringBad 类编写赋值操作符：

```
StringBad & StringBad::operator=(const StringBad & st)
{
   if (this == &st)              // object assigned to itself
      return *this;              // all done
   delete [] str;                // free old string
   len = st.len;
   str = new char [len + 1];     // get space for new string
```

```
    std::strcpy (str, st.str);    // copy the string
    return *this;                 // return reference to invoking object
}
```

代码首先检查自我复制，这是通过查看赋值操作符右边的地址（&s）是否与接收对象（this）的地址相同来完成的。如果相同，程序将返回*this，然后结束。第 10 章介绍过，赋值操作符是只能由类成员函数重载的操作符之一。

如果地址不同，函数将释放 str 指向的内存，这是因为稍后将把一个新字符串的地址赋给 str。如果不首先使用 delete 操作符，则上述字符串将保留在内存中。由于程序中不再包含指向该字符串的指针，因此这些内存被浪费掉。

接下来的操作与复制构造函数相似，即为新字符串分配足够的内存空间，然后将赋值操作符右边的对象中的字符串复制到新的内存单元中。

上述操作完成后，程序返回*this 并结束。

赋值操作并不创建新的对象，因此不需要调整静态数据成员 num_strings 的值。

将前面介绍的复制构造函数和赋值操作符添加到 StringBad 类中后，所有的问题都解决了。例如，下面是在完成上述修改后，程序输出的最后几行：

```
End of main()
"Celery Stalks at Midnight" object deleted, 4 left
"Spinach Leaves Bowl for Dollars" object deleted, 3 left
"Spinach Leaves Bowl for Dollars" object deleted, 2 left
"Lettuce Prey" object deleted, 1 left
"Celery Stalks at Midnight" object deleted, 0 left
```

现在，对象计数是正确的，字符串也没有被损坏。

12.1.3　改进后的新 String 类

有了更丰富的知识后，可以对 StringBad 类进行修订，将它重命名为 String。首先，添加前面介绍过的复制构造函数和赋值操作符，使类能够正确管理类对象使用的内存。其次，由于我们已经知道对象何时被创建和释放，因此可以让类构造函数和析构函数保持沉默，不再在每次被调用时都显示消息。另外，也不用再监视构造函数的工作情况，因此可以简化默认构造函数，使之创建一个空字符串，而不是“C++”。

接下来，可以在类中添加一些新功能。String 类应该包含标准字符串函数库 cstring 的所有功能，才会比较有用，但这里只添加足以说明其工作原理的功能（注意，String 类只是一个用作说明的范例，而 C++ 标准 string 类的内容丰富得多）。具体地说，将添加以下方法：

```
int length()const { return len; }
friend bool operator<(const String &st, const String &st2);
friend bool operator>(const String &st1, const String &st2);
friend bool operator==(const String &st, const String &st2);
friend operator>>(istream & is, String & st);
char & operator[](int i);
const char & operator[](int i) const;
static int HowMany();
```

第一个新方法返回被存储的字符串的长度。接下来的 3 个友元函数能够对字符串进行比较。Operator>>() 函数提供了简单的输入功能；两个 operator[]() 函数提供了以数组表示法访问字符串中各个字符的功能。静态类方法 Howmany() 将补充静态类数据成员 num_string。下面来看一看具体情况。

1. 修订后的默认构造函数

需要注意的是新的默认构造函数，它与下面类似：

```
String::String()
{
    len = 0;
    str = new char[1];

    str[0] = '\0';              // default string
```

```
}
```

读者可能会问，为什么代码为：

```
str = new char[1];
```

而不是：

```
str = new char;
```

上面两种方式分配的内存量相同，区别在于前者与类析构函数兼容，而后者不兼容。析构函数中包含如下代码：

```
delete [] str;
```

delete[ ]与使用 new[ ]初始化的指针和空指针都兼容。因此对于下述代码：

```
str = new char[1];
str[0] = '\0';              // default string
```

可修改为：

```
str = 0; // sets str to the null pointer
```

对于以其他方式初始化的指针，使用 delete [ ]时，结果将是不确定的：

```
char words[15] = "bad idea";
char * p1= words;
char * p2 = new char;
char * p3;
delete [] p1; // undefined, so don't do it
delete [] p2; // undefined, so don't do it
delete [] p3; // undefined, so don't do it
```

2. 比较成员函数

在 String 类中，执行比较操作的方法有 3 个。如果按字母顺序（更准确地说，按照机器排序序列），第一个字符串在第二个字符串之前，则 Operator<() 函数返回 true。要实现字符串比较函数，最简单的方法是使用标准的 trcmp() 函数，如果依照字母顺序，第一个参数位于第二个参数之前，则该函数返回一个负值；如果两个字符串相同，则返回 0；如果第一个参数位于第二个参数之后，则返回一个正值。因此，可以这样使用 strcmp()：

```
bool operator< (const String &st1, const String &st2)
{
   if (std::strcmp (st1.str, st2.str) > 0)
      return true;
   else
      return false;
}
```

因为内置的 > 操作符返回的是一个布尔值，所以可以将代码进一步简化为：

```
bool operator< (const String &st1, const String &st2)
{
   return (std::strcmp (st1.str, st2.str) < 0);
}
```

同样，可以按照下面的方式来编写另外两个比较函数：

```
bool operator> (const String &st1, const String &st2)
{
   return st2.str < st1.str;
}
bool operator== (const String &st1, const String &st2)
{
   return (std::strcmp (st1.str, st2.str) == 0);
}
```

第一个定义利用了 < 操作符来表示 > 操作符，对于内联函数，这是一种很好的选择。

将比较函数作为友元，有助于将 String 对象与常规的 C 字符串进行比较。例如，假设 answer 是 String 对象，则下面的代码：

```
if ("love" == answer)
```

将被转换为：

```
if (operator== ("love", answer) )
```

然后，编译器将使用某个构造函数将代码转换为：

```
if (operator== (String ("love") , answer) )
```

这与原型是相匹配的。

3. 使用中括号表示法来访问字符

对于标准 C 字符串来说，可以使用中括号来访问其中的字符：

```
char city[40] = "Amsterdam";
cout << city[0] << endl; // display the letter A
```

在 C++中，两个中括号组成一个操作符——中括号操作符，可以使用方法 operator[ ]()来重载该操作符。通常，二元 C++操作符（带两个操作数）位于两个操作数之间，例如 2 +5。但对于中括号操作符，一个操作数位于第一个中括号的前面，另一个操作数位于两个中括号之间。因此，在表达式 city[0]中，city 是第一个操作数，[ ]是操作符，0 是第二个操作数。

假设 opera 是一个 String 对象：

```
String opera ("The Magic Flute");
```

则对于表达式 opera[4]，C++将查找名称和特征标与此相同的方法：

```
operator[] (int i)
```

如果找到匹配的原型，编译器将使用下面的函数调用来替代表达式 opera[4]：

```
opera.operator[] (4)
```

opera 对象调用该方法，数组下标 4 成为该函数的参数。

下面是该方法的简单实现：

```
char & String::operator[] (int i)
{
   return str[i];
}
```

有了上述定义后，语句：

```
cout << opera[4];
```

将被转换为：

```
cout << opera.operator[](4);
```

返回值是 opera.str[4]（字符 M）。由此，公有方法可以访问私有数据。

将返回类型声明为 char &，便可以给特定元素赋值。例如，可以编写这样的代码：

```
String means ("might");
means[0] = 'r';
```

第二条语句将被转换为一个重载操作符函数调用：

```
means.operator[](0) = 'r';
```

这里将 r 赋给方法的返回值，而函数返回的是指向 means.str[0]的引用，因此上述代码等同于下面的代码：

```
means.str[0] = 'r';
```

代码的最后一行访问的是私有数据，但由于 operator [ ]() 是类的一个方法，因此能够修改数组的内容。最终的结果是“might”被修改为“right”。

假设有下面的常量对象：

```
const String answer ("futile");
```

如果只有上述 operator[ ]() 定义，则下面的代码将出错：

```
cout << answer[1]; // compile-time error
```

原因是 answer 是常量，而上述方法无法确保不修改数据（实际上，有时方法的工作就是修改数据，因此无法确保不修改数据）。

但在重载时，C++将区分常量和非常量函数的特征标，因此可以提供另一个仅供 const String 对象使用的 operator[ ]() 版本：

```
// for use with const String objects
const char & String::operator[] (int i) const
{
```

```
    return str[i];
}
```

有了上述定义后，就可以读/写常规的 String 对象了；而对于 const String 对象，则只能读取其数据：

```
String text ("Once upon a time");
const String answer ("futile");
cout << text[1];      // ok, uses non-const version of operator[]()
cout << answer[1];    // ok, uses const version of operator[]()
cout >> text[1];      // ok, uses non-const version of operator[]()
cin >> answer[1];     // compile-time error
```

4. 静态类成员函数

可以将成员函数声明为静态的（函数声明中必须包含关键字 static，如果函数定义是独立的，则其中不能包含关键字 static），这样做有两个重要的后果。

首先，不能通过对象调用静态成员函数；实际上，静态成员函数甚至不能使用 this 指针。如果静态成员函数是在公有部分声明的，则可以使用类名和作用域解析操作符来调用它。可以给 String 类添加一个名为 HowMany()的静态成员函数，方法是在类声明中添加如下原型/定义：

```
static int HowMany(){ return num_strings; }
```

调用它的方式如下：

```
int count = String::HowMany();  // invoking a static member function
```

其次，由于静态成员函数不与特定的对象相关联，因此只能使用静态数据成员。例如，静态方法 HowMany()可以访问静态成员 num_string，但不能访问 str 和 len。

同样，也可以使用静态成员函数设置类级（classwide）标记，以控制某些类接口的行为。例如，类级标记可以控制显示类内容的方法所使用的格式。

5. 进一步重载赋值操作符

介绍针对 String 类的程序清单之前，先来考虑另外一个问题。假设要将常规字符串复制到 String 对象中。例如，假设使用 getline()读取了一个字符串，并要将这个字符串放置到 String 对象中，由前面定义的类方法能够这样编写代码：

```
String name;
char temp[40];
cin.getline (temp, 40);
name = temp;  // use constructor to convert type
```

但如果经常需要这样做，这将不是一种理想的解决方案。为解释其原因，先来回顾一下最后一条语句是怎样工作的：

（1）程序使用构造函数 String（const char *）来创建一个临时 String 对象，其中包含 temp 中的字符串拷贝。第 11 章介绍过，只有一个参数的构造函数被用作转换函数。

（2）本章后面的程序清单 12.6 中的程序使用 String & String::operator =（const String &）函数将临时对象中的信息复制到 name 对象中。

（3）程序调用析构函数~String()来删除临时对象。

为提高处理效率，最简单的方法是重载赋值操作符，使之能够直接使用常规字符串，这样就不用创建和删除临时对象了。下面是一种可能的实现：

```
String & String::operator= (const char * s)
{
    delete [] str;
    len = std::strlen (s);
    str = new char[len + 1];
    std::strcpy (str, s);
    return *this;
}
```

一般说来，必须释放 str 指向的内存，并为新字符串分配足够的内存。

程序清单 12.4 列出了修订后的类声明。除了前面提到过的修改之外，这里还定义了一个 CINLIM 常量，

用于实现 operator>>()。

程序清单 12.4　string1.h

```
// string1.h -- fixed and augmented string class definition
#include <iostream>
using std::ostream;
using std::istream;

#ifndef STRING1_H_
#define STRING1_H_
class String
{
private:
    char * str;             // pointer to string
    int len;                // length of string
    static int num_strings; // number of objects
    static const int CINLIM = 80;  // cin input limit
public:
// constructors and other methods
    String(const char * s);  // constructor
    String();                // default constructor
    String(const String &); // copy constructor
    ~String();              // destructor
    int length()const { return len; }
// overloaded operator methods
    String & operator=(const String &);
    String & operator=(const char *);
    char & operator[](int i);
    const char & operator[](int i) const;
// overloaded operator friends
    friend bool operator<(const String &st, const String &st2);
    friend bool operator>(const String &st1, const String &st2);
    friend bool operator==(const String &st, const String &st2);
    friend ostream & operator<<(ostream & os, const String & st);
    friend istream & operator>>(istream & is, String & st);
// static function
    static int HowMany();
};
#endif
```

注意：如果编译器没有实现布尔类型的话，可以使用 int 代替 bool，0 代替 false，1 代替 true。如果编译器不支持静态类常量，可以用枚举来定义 CINLIM。

```
enum {CINLIM = 90};
```

程序清单 12.5 给出了修订后的方法定义。

程序清单 12.5　string1.cpp

```
// string1.cpp -- String class methods
#include <cstring>              // string.h for some
#include "string1.h"            //includes <iostream>
using std::cin;
using std::cout;

// initializing static class member

int String::num_strings = 0;

// static method
int String::HowMany()
```

```
{
    return num_strings;
}

// class methods
String::String(const char * s)  // construct String from C string
{

    len = std::strlen(s);       // set size
    str = new char[len + 1];    // allot storage
    std::strcpy(str, s);        // initialize pointer
    num_strings++;              // set object count
}

String::String()                // default constructor
{
    len = 4;
    str = new char[1];
    str[0] = '\0';              // default string
    num_strings++;
}

String::String(const String & st)
{
    num_strings++;              // handle static member update
    len = st.len;               // same length
    str = new char [len + 1];   // allot space
    std::strcpy(str, st.str);   // copy string to new location
}

String::~String()               // necessary destructor
{
    --num_strings;              // required
    delete [] str;              // required
}

// overloaded operator methods

    // assign a String to a String
String & String::operator=(const String & st)
{
    if (this == &st)
        return *this;
    delete [] str;
    len = st.len;
    str = new char[len + 1];
    std::strcpy(str, st.str);
    return *this;
}

    // assign a C string to a String
String & String::operator=(const char * s)
{
    delete [] str;
    len = std::strlen(s);
    str = new char[len + 1];
    std::strcpy(str, s);
    return *this;
}
```

```
    // read-write char access for non-const String
char & String::operator[] (int i)
{
    return str[i];
}

    // read-only char access for const String
const char & String::operator[] (int i) const
{
    return str[i];
}

// overloaded operator friends

bool operator< (const String &st1, const String &st2)
{
    return (std::strcmp (st1.str, st2.str) < 0);
}

bool operator> (const String &st1, const String &st2)
{
    return st2.str < st1.str;
}

bool operator== (const String &st1, const String &st2)
{
    return (std::strcmp (st1.str, st2.str) == 0);
}

    // simple String output
ostream & operator<< (ostream & os, const String & st)
{
    os << st.str;
    return os;
}

    // quick and dirty String input
istream & operator>> (istream & is, String & st)
{
    char temp[String::CINLIM];
    is.get (temp, String::CINLIM);
    if (is)
        st = temp;
    while (is && is.get() != '\n')
        continue;
    return is;
}
```

重载>>操作符提供了一种将键盘输入行读入到 String 对象中的简单方法。它假定输入的字符数不多于 String::CINLIM 的字符数，并丢弃多余的字符。在 if 条件下，如果由于某种原因（如到达文件尾，或 get（char *，int）读取的是一个空行）导致输入失败，istream 对象的值将置为 false。

程序清单 12.6 通过一个小程序来使用这个类，该程序允许输入几个字符串。程序首先提示用户输入，然后将用户输入的字符串存储到 String 对象中，并显示它们，最后指出哪个字符串最短、哪个字符串按字母顺序排在最前面。

程序清单 12.6　sayings1.cpp

```
// sayings1.cpp -- using expanded String class
// compile with string1.cpp
#include <iostream>
```

```
#include "string1.h"
const int ArSize = 10;
const int MaxLen =81;
int main()
{
   using std::cout;
   using std::cin;
   using std::endl;
   String name;
   cout <<"Hi, what's your name?\n>> ";
   cin >> name;

   cout << name << ", please enter up to " << ArSize
      << " short sayings <empty line to quit>:\n";
   String sayings[ArSize];    // array of objects
   char temp[MaxLen];         // temporary string storage
   int i;
   for (i = 0; i < ArSize; i++)
   {
      cout << i+1 << ": ";
      cin.get(temp, MaxLen);
      while (cin && cin.get() != '\n')
         continue;
      if (!cin || temp[0] == '\0')   // empty line
         break;            // i not incremented
      else
         sayings[i] = temp;  // overloaded assignment
   }
   int total = i;            // total # of lines read

   cout << "Here are your sayings:\n";
   for (i = 0; i < total; i++)
      cout << sayings[i][0] << ": " << sayings[i] << endl;

   int shortest = 0;
   int first = 0;
   for (i = 1; i < total; i++)
   {
      if (sayings[i].length() < sayings[shortest].length())
         shortest = i;
      if (sayings[i] < sayings[first])
         first = i;
   }
   cout << "Shortest saying:\n" << sayings[shortest] << endl;;
   cout << "First alphabetically:\n" << sayings[first] << endl;
   cout << "This program used "<< String::HowMany()
      << " String objects. Bye.\n";

   return 0;
}
```

注意：较早的 get（char *、int）版本在读取空行后，返回的值不为 false。不过，对于这些版本来说，如果读取了一个空行，则字符串中第一个字符将是一个空字符。这个范例使用了下述代码：

```
if (!cin || temp[0] == '\0')   // empty line?
     break;                    // i not incremented
```

如果实现遵循了最新的 C++标准，则 if 语句中的第一个条件将检测到空行，第二个条件用于旧版本实现中检测空行。

程序清单 12.6 中程序要求用户输入至多 10 条谚语。每条谚语都被读到一个临时字符数组，然后被复

制到 String 对象中。如果用户输入空行，break 语句将终止输入循环。显示用户的输入后，程序使用成员函数 length()和 operator<()来确定最短的字符串以及按字母顺序排列在最前面的字符串。程序还使用下标操作符（[]）提取每条谚语的第一个字符，并将其放在该谚语的最前面。下面就是运行情况：

```
Hi, what's your name?
>> Misty Gutz
Misty Gutz, please enter up to 10 short sayings <empty line to quit>:
1: a fool and his money are soon parted
2: penny wise, pound foolish
3: the love of money is the root of much evil
4: out of sight, out of mind
5: absence makes the heart grow fonder
6: absinthe makes the hart grow fonder
7:
a: a fool and his money are soon parted
p: penny wise, pound foolish
t: the love of money is the root of much evil
o: out of sight, out of mind
a: absence makes the heart grow fonder
a: absinthe makes the hart grow fonder
Shortest saying:
penny wise, pound foolish
First alphabetically:
a fool and his money are soon parted
This program used 11 String objects. Bye.
```

12.1.4　在构造函数中使用 new 时应注意的事项

至此，可以知道使用 new 来初始化对象的指针成员时必须特别小心。具体地说，应当这样做：

- 如果在构造函数中使用 new 来初始化指针成员，则应在析构函数中使用 delete。
- new 和 delete 必须相互兼容。new 对应于 delete，new[ ]对应于 delete[ ]。
- 如果有多个构造函数，则必须以相同的方式使用 new，要么都带中括号，要么都不带。因为只有一个析构函数，因此所有的构造函数都必须与它兼容。不过，可以在一个构造函数中使用 new 来初始化指针，而在另外一个构造函数中将指针初始化为空（NULL 或 0），这是因为 delete（无论是带中括号还是不带中括号）可以用于空指针。

NULL 还是 0

空指针可以用 0 或 NULL（在许多头文件中，NULL 是一个被定义为 0 的符号常量）来表示。C 程序员通常使用 NULL 而不是 0，因为这样易知它是一指针值（就像使用“\0”而不是 0 来表示空字符串一样，前者指出这是一个字符值）。不过，C++传统上更喜欢用简单的 0，而不是等价的 NULL。

- 应定义一个复制构造函数，通过深度复制将一个对象初始化为另一个对象。通常，这种构造函数与下面类似：

```
String::String (const String & st)
{
    num_strings++;                  // handle static member update if necessary
    len = st.len;                   // same length
    str = new char [len + 1];       // allot space
    std::strcpy (str, st.str);          // copy string to new location
}
```

- 具体地说，复制构造函数应分配足够的空间来存储复制的数据，并复制数据，而不仅仅是数据的地址。另外，还应该更新所有受影响的静态类成员。
- 应当定义一个赋值操作符，通过深度复制将一个对象复制给另一个对象。通常，该类方法与下面类似：

```
String & String::operator= (const String & st)
```

```
{
    if (this == &st)          // object assigned to itself
        return *this;         // all done
    delete [] str;            // free old string
    len = st.len;
    str = new char [len + 1];  // get space for new string
    std::strcpy (str, st.str);       // copy the string
    return *this;             // return reference to invoking object
}
```

具体地说，该方法应完成这些操作：检查自我赋值的情况，释放成员指针以前指向的内存，复制数据而不仅仅是数据的地址，并返回一个指向调用对象的引用。

下面的摘要包含了两个不正确的范例（指出什么是不应当做的）以及一个良好的构造函数范例：

```
String::String()
{
    str = "default string";    // oops, no new []
    len = std::strlen (str);
}

String::String (const char * s)
{
    len = std::strlen (s);
    str = new char;           // oops, no []
    std::strcpy (str, s);           // oops, no room
}

String::String (const String & st)
{
    len = st.len;
    str = new char[len + 1];     // good, allocate space
    std::strcpy (str, st.str);        // good, copy value
}
```

第一个构造函数没有使用 new 来初始化 str。对默认对象调用析构函数时，析构函数使用 delete 来释放 str。对不是使用 new 初始化的指针使用 delete 时，结果将是不确定的，并可能是有害的。可将该构造函数修改为下面的任何一种形式：

```
String::String()
{
   len = 0;
   str = new char[1];  // uses new with []
   str[0] = '\0';
}

String::String()
{
   len = 0;
   str = NULL;  // or the equivalent str = 0;
}

String::String()
{
   static const char * s = "C++";    // initialized just once
   len = std::strlen (s);
   str = new char[len + 1];          // uses new with []
   std::strcpy (str, s);
}
```

摘录中的第二个构造函数使用了 new，但分配的内存量不正确。因此，new 返回的内存块只能保存一个字符。

试图将过长的字符串复制到内存单元中，将导致内存问题。另外，这里使用的 new 不带中括号，这与

另一个构造函数的正确格式不一致。

第三个构造函数是正确的。

最后，下面的析构函数无法与前面的构造函数正常地协同工作：

```
String::~String()
{
    delete str;     // oops, should be delete [] str;
}
```

该析构函数未能正确地使用 delete。由于构造函数创建的是一个字符数组，因此析构函数应该删除一个数组。

12.1.5　有关返回对象的说明

当成员函数或独立的函数返回对象时，有几种返回方式可供选择。可以返回指向对象的引用、指向对象的 const 引用或 const 对象。到目前为止，介绍了前两种方式，但没有介绍最后一种方式，现在是复习这些方式的好时机。

1. 返回指向 const 对象的引用

使用 const 引用的常见原因是旨在提高效率，但对于何时可以采用这种方式存在一些限制。如果函数返回（通过调用对象的方法或将对象作为参数）传递给它的对象，可以通过传递引用来提高方法的效率。例如，假设要编写函数 Max()，它返回两个 Vector 对象中较大的一个，其中 Vector 是第 11 章开发的一个类。该函数将以下面的方式被使用：

```
Vector force1(50,60);
Vector force2(10,70);
Vector max;
max = Max(force1, force2);
```

下面两种实现都是可行的：

```
// version 1
Vector Max(const Vector & v1, const Vector & v2)
{
    if (v1.magval() > v2.magval())
        return v1;
    else
        return v2;
}

// version 2
const Vector & Max(const Vector & v1, const Vector & v2)
{
    if (v1.magval() > v2.magval())
        return v1;
    else
        return v2;
}
```

这里有三点需要说明。首先，返回对象将调用复制构造函数，而返回引用不会。因此，第二个版本所做的工作更少，效率更高。其次，引用指向的对象应该在调用函数执行时存在。在这个例子中，引用指向 force1 或 force2，它们都是在调用函数中定义的，因此满足这种条件。第三，v1 和 v2 都被声明为 const 引用，因此返回类型必须为 const，这样才匹配。

2. 返回指向非 const 对象的引用

两种常见的返回非 const 对象情形是，重载赋值操作符以及重载与 cout 一起使用的<<操作符。前者这样做旨在提高效率，而后者必须这样做。

operator=()的返回值用于连续赋值：

```
String s1("Good stuff");
```

```
String s2, s3;
s3 = s2 = s1;
```

在上述代码中，s2.operator=(s2)的返回值被赋给 s3。为此，返回 String 对象或 String 对象的引用都是可行的，但与 Vector 范例中一样，通过使用引用，可避免该函数调用 String 的复制构造函数来创建一个新的 String 对象。在这个例子中，返回类型不是 const，因为方法 operator=()返回一个指向 s2 的引用，可以对其进行修改。

Operator<<()的返回值用于串接输出：

```
String s1("Good stuff");
cout << s1 << "is coming!";
```

在上述代码中，operator<<(cout, s1)的返回值成为一个用于显示字符串“is coming!”的对象。返回类型必须是 ostream &，而不能仅仅是 ostream。如果使用返回类型 ostream，将要求调用 ostream 类的复制构造函数，而 ostream 没有公有的复制构造函数。幸运的是，返回一个指向 cout 的引用不会带来任何问题，因为 cout 已经在调用函数的作用域内。

3. 返回对象

如果被返回的对象是被调用函数中的局部变量，则不应按引用方式返回它，因为在被调用函数执行完毕时，局部对象将调用其析构函数。因此，当控制权回到调用函数时，引用指向的对象将不再存在。在这种情况下，应返回对象而不是引用。通常，被重载的算术操作符属于这一类。请看下述范例，它再次使用了 Vector 类：

```
Vector force1(50,60);
Vector force2(10,70);
Vector net;
net = force1 + force2;
```

被返回的不是 force1，也不是 force2，force1 和 force2 在这个过程中应该保持不变。因此，返回值不能是指向在调用函数中已经存在的对象的引用。相反，在 Vector::operator+()中计算得到的两个矢量的和被存储在一个新的临时对象中，该函数也不应返回一个指向该临时对象的引用，而应该返回实际的 Vector 对象，而不是引用：

```
Vector Vector::operator+(const Vector & b) const
{
   return Vector(x + b.x, y + b.y);
}
```

在这种情况下，存在调用复制构造函数来创建被返回的对象的开销，然而这是无法避免的。

在上述范例中，构造函数调用 Vector(x + b.x, y + b.y)创建一个方法 operator+()能够访问的对象；而返回语句引发的对复制构造函数的隐式调用创建一个调用程序能够访问的对象。

4. 返回 const 对象

前面的 Vector::operator+()定义有一个奇异的属性，它旨在让您以下面这样的方式使用它：

```
net = force1 + force2;                                  // 1: three Vector objects
```

然而，这种定义也允许您这样使用它：

```
force1 + force2 = net;                                  // 2: dyslectic programming
cout << (force1 + force2 = net).magval() << endl; // 3: demented programming
```

这提出了三个问题。为何编写这样的语句？这些语句为何可行？这些语句有何功能？

首先，没有要编写这种语句的合理理由，但并非所有代码都是合理的。即使是程序员也会犯错。例如，为 Vector 类定义 operator==()时，您可能错误地输入这样的代码：

```
if (force1 + force2 = net)
```

而不是：

```
if (force1 + force2 == net)
```

另外，程序员通常很有创意，这可能导致错误。

其次，这种代码之所以可行，是因为复制构造函数将创建一个临时对象来表示返回值。因此，在前面的代码中，表达式 force1 + force2 的结果为一个临时对象。在语句 1 中，该临时对象被赋给 net；在语句 2

和 3 中，net 被赋给该临时对象。

第三，使用完临时对象后，将把它丢弃。例如，对于语句 2，程序计算 force1 和 force2 之和，将结果复制到临时返回对象中，再用 net 的内容覆盖临时对象的内容，然后将该临时对象丢弃。原来的矢量全都保持不变。语句 3 显示临时对象的长度，然后将其删除。

如果您担心这种行为可能引发的误用和滥用，有一种简单的解决方案：将返回类型声明为 const Vector。例如，如果 Vector::operator+()的返回类型被声明为 const Vector，则语句 1 仍然合法，但语句 2 和 3 将是非法的。

总之，如果方法或函数要返回局部对象，则应返回对象，而不是指向对象的引用。在这种情况下，将使用复制构造函数来生成返回的对象。如果方法或函数要返回一个没有公有复制构造函数的类（如 ostream 类）的对象，它必须返回一个指向这种对象的引用。最后，有些方法和函数（如重载的赋值操作符）可以返回对象，也可以返回指向对象的引用，在这种情况下，应首选引用，因为其效率更高。

12.1.6　使用指向对象的指针

C++程序经常使用指向对象的指针，因此，这里来练习一下。程序清单 12.6 使用数组索引值来跟踪最短的字符串和按字母顺序排在最前面的字符串。另一种方法是使用指针指向这些类别的开始位置，程序清单 12.7 使用两个指向 String 的指针实现了这种方法。最初，shortest 指针指向数组中的第一个对象。每当程序找到比指向的字符串更短的对象时，就把 shortest 重新设置为指向该对象。同样，first 指针跟踪按字母顺序排在最前面的字符串。这两个指针并不创建新的对象，而只是指向已有的对象。因此，这些指针并不要求使用 new 来分配内存。

除此之外，程序清单 12.7 中的程序还使用一个指针来跟踪新对象：

```
String * favorite = new String (sayings[choice]) ;
```

这里指针 favorite 指向 new 创建的未被命名对象。这种特殊的句法意味着使用对象 saying [choice]来初始化新的 String 对象，这将调用复制构造函数，因为复制构造函数（const String &）的参数类型与初始化值（saying [choice]）匹配。程序使用 srand()、rand()和 time()随机选择一个值。

程序清单 12.7　sayings2.cpp

```
// sayings2.cpp -- uses pointers to objects
// compile with string1.cpp
#include <iostream>
#include <cstdlib>          // (or stdlib.h) for rand(), srand()
#include <ctime>            // (or time.h) for time()
#include "string1.h"
const int ArSize = 10;
const int MaxLen = 81;
int main()
{
    using namespace std;
    String name;
    cout <<"Hi, what's your name?\n>> ";
    cin >> name;

    cout << name << ", please enter up to " << ArSize
        << " short sayings <empty line to quit>: \n";
    String sayings[ArSize];
    char temp[MaxLen];              // temporary string storage
    int i;
    for (i = 0; i < ArSize; i++)
    {
        cout << i+1 << ": ";
        cin.get (temp, MaxLen);
        while (cin && cin.get()!= '\n')
            continue;
```

```
        if (!cin || temp[0] == '\0')   // empty line?
            break;                     // i not incremented
        else
            sayings[i] = temp;         // overloaded assignment
    }
    int total = i;                     // total # of lines read

    if (total > 0)
    {
        cout << "Here are your sayings:\n";
        for (i = 0; i < total; i++)
         cout << sayings[i] << "\n";

    // use pointers to keep track of shortest, first strings
        String * shortest = &sayings[0]; // initialize to first object
        String * first = &sayings[0];
        for (i = 1; i < total; i++)
        {
            if (sayings[i].length()< shortest->length())
                shortest = &sayings[i];
            if (sayings[i] < *first)     // compare values
                first = &sayings[i];     // assign address
        }
        cout << "Shortest saying:\n" << * shortest << endl;
        cout << "First alphabetically:\n" << * first << endl;

        srand (time (0));
        int choice = rand()% total;      // pick index at random
    // use new to create, initialize new String object
        String * favorite = new String (sayings[choice]);
        cout << "My favorite saying:\n" << *favorite << endl;
        delete favorite;
    }
    else
        cout << "No much to say, eh?\n";
    cout << "Bye.\n";
    return 0;
}
```

使用 new 初始化对象

通常，如果 Class_name 是类，value 的类型为 Type_name，则下面的语句：

```
Class_name * pclass = new Class_name (value);
```

将调用如下构造函数：

```
Class_name (Type_name);
```

这里可能还有一些琐碎的转换，例如：

```
Class_name (const Type_name &);
```

另外，如果不存在二义性，则将发生由原型匹配导致的转换（如从 int 到 double）。下面的初始化方式将调用默认构造函数：

```
Class_name * ptr = new Class_name;
```

注意：早期的 C++实现可能要求包含 stdlib.h 和 time.h，而不是 cstdlib 和 ctime。

下面是程序清单 12.7 中程序的运行情况：

```
Hi, what's your name?
>> Kirt Rood
Kirt Rood, please enter up to 10 short sayings <empty line to quit>:
1: a friend in need is a friend indeed
2: neither a borrower nor a lender be
```

```
3: a stitch in time saves nine
4: a niche in time saves stine
5: it takes a crook to catch a crook
6: cold hands, warm heart
7:
Here are your sayings:
a friend in need is a friend indeed
neither a borrower nor a lender be
a stitch in time saves nine
a niche in time saves stine
it takes a crook to catch a crook
cold hands, warm heart
Shortest saying:
cold hands, warm heart
First alphabetically:
a friend in need is a friend indeed
My favorite saying:
a stitch in time saves nine
Bye
```

由于该程序随机选择用户输入的格言，因此即使输入相同，显示的结果也可能不同。

1. 再谈 new 和 delete

程序清单 12.4、12.5 和 12.7 组成的程序在两个层次上使用了 new 和 delete。首先，它使用 new 为创建的每一个对象的名称字符串分配存储空间，这是在构造函数中进行的，因此析构函数使用 delete 来释放这些内存。因为字符串是一个字符数组，所以析构函数使用的是带中括号的 delete。这样，当对象被释放时，用于存储字符串内容的内存将被自动释放。其次，程序清单 12.7 中的代码使用 new 来为整个对象分配内存：

```
String * favorite = new String (sayings[choice]) ;
```

这不是为要存储的字符串分配内存，而是为对象分配内存：也就是说，为保存字符串地址的 str 指针和 len 成员分配内存（程序并没有给 num_string 成员分配内存，这是因为 num_string 成员是静态成员，它独立于对象被保存）。创建对象将调用构造函数，后者分配用于保存字符串的内存，并将字符串的地址赋给 str。然后，当程序不再需要该对象时，使用 delete 删除它。对象是单个的，因此，程序使用不带中括号的 delete。与前面介绍的相同，这将只释放用于保存 str 指针和 len 成员的空间，并不释放 str 指向的内存，而该任务将由析构函数来完成（参见图 12.4）。

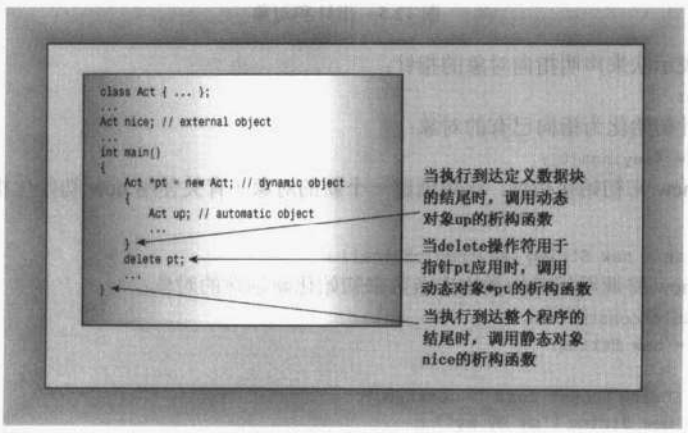


图 12.4　调用析构函数

在下述情况下析构函数将被调用（参见图 12.4）。

- 如果对象是动态变量，则当执行完定义该对象的程序块时，将调用该对象的析构函数。因此，在

程序清单 12.3 中，执行完 main()时，将调用 headline[0]和 headline[1]的析构函数；执行完 callme1()时，将调用 grub 的析构函数。

- 如果对象是静态变量（外部、静态、静态外部或来自名称空间），则在程序结束时将调用对象的析构函数。这就是程序清单 12.3 中 sports 对象所发生的情况。
- 如果对象是用 new 创建的，则仅当您显式使用 delete 删除对象时，其析构函数才会被调用。

2. 指针和对象小结

使用对象指针时，需要注意几点（参见图 12.5）：

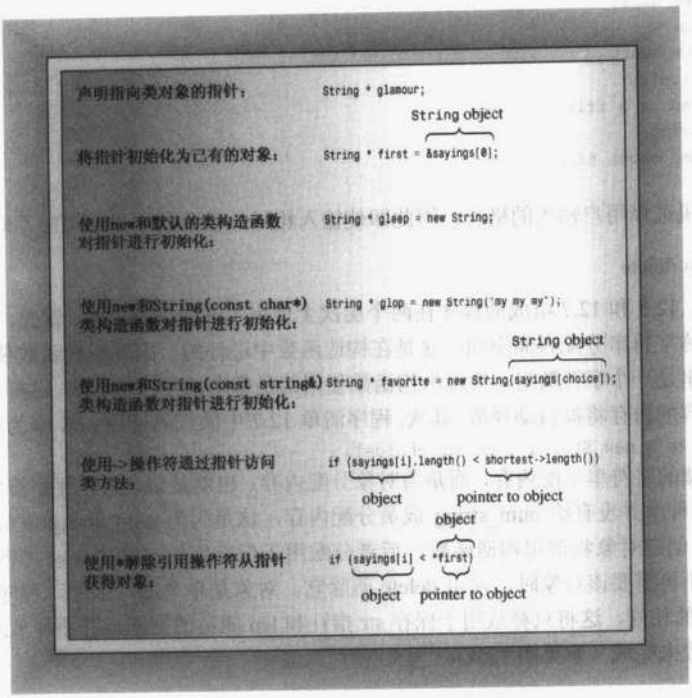


图 12.5 指针和对象

- 使用常规表示法来声明指向对象的指针：

```
String * glamour;
```

- 可以将指针初始化为指向已有的对象：

```
String * first = &sayings[0];
```

- 可以使用 new 来初始化指针，这将创建一个新的对象（有关使用 new 初始化指针的细节，请参见图 12.6）：

```
String * favorite = new String (sayings[choice]) ;
```

- 对类使用 new 将调用相应的类构造函数来初始化新创建的对象：

```
// invokes default constructor
String * gleep = new String;

// invokes the String (const char *) constructor
String * glop = new String ("my my my") ;

// invokes the String (const String &) constructor
String * favorite = new String (sayings[choice]) ;
```

- 可以使用-> 操作符通过指针访问类方法：

```
if (sayings[i].length()< shortest->length())
```

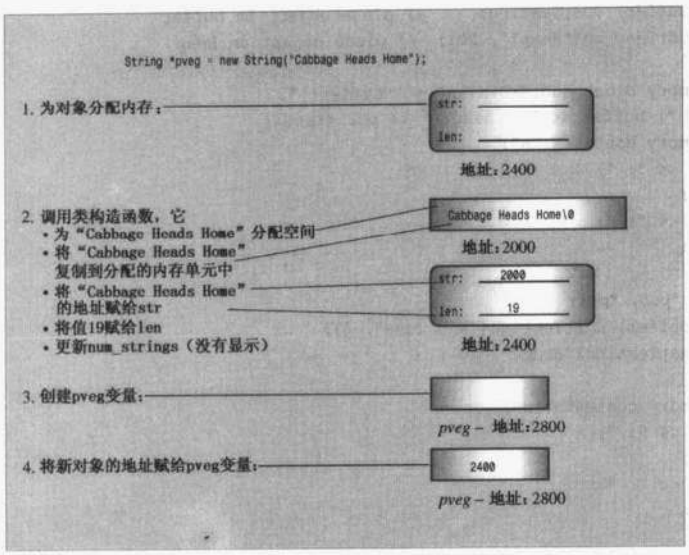


图 12.6　使用 new 创建对象

- 可以对对象指针应用解除引用操作符（*）来获得对象：

```
if (sayings[i] < *first)     // compare object values
    first = &sayings[i];     // assign object address
```

3. 再谈布局 new 操作符

本书前面介绍过，布局 new 操作符让您能够在分配内存时指定内存位置。第 9 章从内置类型的角度讨论了布局 new 操作符，将这种操作符用于对象时情况有些不同，程序清单 12.8 使用了布局 new 操作符和常规 new 操作符给对象分配内存，其中定义的类的构造函数和析构函数都会显示一些信息，让用户能够了解对象的历史。

程序清单 12.8　placenew1.cpp

```
// placenew1.cpp  -- new, placement new, no delete
#include <iostream>
#include <string>
#include <new>
using namespace std;
const int BUF = 512;

class JustTesting
{
private:
    string words;
    int number;
public:
    JustTesting(const string & s = "Just Testing", int n = 0)
    {words = s; number = n; cout << words << " constructed\n"; }
    ~JustTesting() { cout << words << " destroyed\n";}
    void Show() const { cout << words << ", " << number << endl;}
};
int main()
{
    char * buffer = new char[BUF];      // get a block of memory

    JustTesting *pc1, *pc2;
```

```
    pc1 = new (buffer) JustTesting;     // place object in buffer
    pc2 = new JustTesting("Heap1", 20);  // place object on heap

    cout << "Memory block addresses:\n" << "buffer: "
       << (void *) buffer << "    heap: " << pc2 <<endl;
    cout << "Memory contents:\n";
    cout << pc1 << ": ";
    pc1->Show();
    cout << pc2 << ": ";
    pc2->Show();

    JustTesting *pc3, *pc4;
    pc3 = new (buffer) JustTesting("Bad Idea", 6);
    pc4 = new JustTesting("Heap2", 10);

    cout << "Memory contents:\n";
    cout << pc3 << ": ";
    pc3->Show();
    cout << pc4 << ": ";
    pc4->Show();

    delete pc2;                     // free Heap1
    delete pc4;                     // free Heap2
    delete [] buffer;                // free buffer
    cout << "Done\n";
    return 0;
}
```

该程序使用 new 操作符创建了一个 512 字节的内存缓冲区，然后使用 new 操作符在堆中创建两个 JustTesting 对象，并试图使用布局 new 操作符在内存缓冲区中创建两个 JustTesting 对象。最后，它使用 delete 来释放使用 new 分配的内存。下面是该程序的输出：

```
Just Testing constructed
Heap1 constructed
Memory block addresses:
buffer: 00320AB0    heap: 00320CE0
Memory contents:
00320AB0: Just Testing, 0
00320CE0: Heap1, 20
Bad Idea constructed
Heap2 constructed
Memory contents:
00320AB0: Bad Idea, 6
00320EC8: Heap2, 10
Heap1 destroyed
Heap2 destroyed
Done
```

和往常一样，内存地址的格式和值将随系统而异。

程序清单 12.8 在使用布局 new 操作符时存在两个问题。首先，在创建第二个对象时，布局 new 操作符使用一个新对象来覆盖用于第一个对象的内存单元。显然，如果类动态地为其成员分配内存，这将引发问题。

其次，将 delete 用于 pc2 和 pc4 时，将自动调用为 pc2 和 pc4 指向的对象调用析构函数；然而，将 delete[] 用于 buffer 时，不会为使用布局 new 操作符创建的对象调用析构函数。

这里的经验教训与第 9 章介绍的相同：程序员必须负责管理布局 new 操作符从中使用的缓冲区内存单元。要使用不同的内存单元，程序员需要提供两个位于缓冲区的不同地址，并确保这两个内存单元不重叠。例如，可以这样做：

```
pc1 = new (buffer) JustTesting;
pc3 = new (buffer + sizeof (JustTesting)) JustTesting("Better Idea", 6);
```

其中指针 pc3 相对于 pc1 的偏移量为 JustTesting 对象的大小。

第三个教训是，如果使用布局 new 操作符来为对象分配内存，必须确保其析构函数被调用。但如何确保呢？对于在堆中创建的对象，可以这样做：

```
delete pc2;  // delete object pointed to by pc2
```

然而，对于使用布局 new 操作符创建的对象，不能像下面这样做：

```
delete pc1;      // delete object pointed to by pc1? NO! Reason #1
delete pc3;      // delete object pointed to by pc2? NO! Reason #2
```

原因在于 delete 可与常规 new 操作符配合使用，但不能与布局 new 操作符配合使用。例如，指针 pc3 没有收到 new 操作符返回的地址，因此 delete pc3 将导致运行阶段错误。在另一方面，指针 pc1 指向的地址与 buffer 相同，但 buffer 是使用 new [ ]初始化的，因此必须使用 delete [ ]而不是 delete 来释放。即使 buffer 是使用 new 而不是 new [ ]初始化的，delete pc1 也将释放 buffer，而不是 pc1。这是因为 new/delete 系统知道已分配的 512 字节块 buffer，但对布局 new 操作符对该内存块做了何种处理一无所知。

该程序确实释放了 buffer：

```
delete [] buffer;                    // free buffer
```

正如上述注释指出的，delete [ ] buffer;释放使用常规 new 操作符分配的整个内存块，但它没有为布局 new 操作符在该内存块中创建的对象调用析构函数。读者之所以知道这一点，是因为该程序使用了一个显示信息的析构函数，该析构函数宣布了“Heap1”和“Heap2”的死亡，但却没有宣布“Just Testing”和“Bad Idea”的死亡。

这种问题的解决方案是，显式地为使用布局 new 操作符创建的对象调用析构函数。正常情况下将自动调用析构函数，这是需要显式调用析构函数的少数几种情形之一。显式地调用析构函数时，必须指定要销毁的对象。由于有指向对象的指针，因此可以使用这些指针：

```
pc3->~JustTesting();  // destroy object pointed to by pc3
pc1->~JustTesting();  // destroy object pointed to by pc1
```

程序清单 12.9 对布局 new 操作符使用的内存单元进行管理，加入到合适的 delete 和显式析构函数调用，从而修复了程序清单 12.8 中的问题。需要注意的一点是正确的删除顺序。对于使用布局 new 操作符创建的对象，应以与创建顺序相反的顺序进行删除。原因在于，晚创建的对象可能依赖于早创建的对象。另外，仅当所有对象都被销毁后，才能释放用于存储这些对象的缓冲区。

程序清单 12.9　placenew2.cpp

```
// placenew2.cpp  -- new, placement new, no delete
#include <iostream>
#include <string>
#include <new>
using namespace std;
const int BUF = 512;

class JustTesting
{
private:
    string words;
    int number;
public:
    JustTesting(const string & s = "Just Testing", int n = 0)
    {words = s; number = n; cout << words << " constructed\n"; }
    ~JustTesting() { cout << words << " destroyed\n";}
    void Show() const { cout << words << ", " << number << endl;}
};
int main()
{
    char * buffer = new char[BUF];      // get a block of memory

    JustTesting *pc1, *pc2;
```

```
    pc1 = new (buffer) JustTesting;     // place object in buffer
    pc2 = new JustTesting("Heap1", 20);  // place object on heap

    cout << "Memory block addresses:\n" << "buffer: "
       << (void *) buffer << "    heap: " << pc2 <<endl;
    cout << "Memory contents:\n";
    cout << pc1 << ": ";
    pc1->Show();
    cout << pc2 << ": ";
    pc2->Show();

    JustTesting *pc3, *pc4;
// fix placement new location
    pc3 = new (buffer + sizeof (JustTesting))
       JustTesting("Better Idea", 6);
    pc4 = new JustTesting("Heap2", 10);

    cout << "Memory contents:\n";
    cout << pc3 << ": ";
    pc3->Show();
    cout << pc4 << ": ";
    pc4->Show();

    delete pc2;          // free Heap1
    delete pc4;          // free Heap2
// explicitly destroy placement new objects
    pc3->~JustTesting();  // destroy object pointed to by pc3
    pc1->~JustTesting();  // destroy object pointed to by pc1
    delete [] buffer;     // free buffer
    cout << "Done\n";
    return 0;
}
```

下面是该程序的输出：

```
Just Testing constructed
Heap1 constructed
Memory block addresses:
buffer: 00320AB0    heap: 00320CE0
Memory contents:
00320AB0: Just Testing, 0
00320CE0: Heap1, 20
Better Idea constructed
Heap2 constructed
Memory contents:
00320AD0: Better Idea, 6
00320EC8: Heap2, 10
Heap1 destroyed
Heap2 destroyed
Better Idea destroyed
Just Testing destroyed
Done
```

该程序使用布局 new 操作符在相邻的内存单元中创建两个对象，并调用了合适的析构函数。

12.1.7 复习各种技术

至此，介绍了多种用于处理各种与类相关的问题的编程技术。可能难以掌握这些技术，下面对它们进行总结，并介绍何时使用它们。

1. 重载<<操作符

要重新定义 << 操作符，以便将它和 cout 一起用来显示对象的内容，请定义下面的友元操作符函数：

```
ostream & operator<< (ostream & os, const c_name  & obj)
{
   os << ...;  // display object contents
   return os;
}
```

其中 c_name 是类名。如果该类提供了能够返回所需内容的公有方法，则可在操作符函数中使用这些方法，这样便不用将它们设置为友元函数了。

2. 转换函数

要将单个值转换为类类型，需要创建原型如下所示的类构造函数：

```
c_name (type_name value);
```

其中 c_name 为类名，type_name 是要转换的类型的名称。

要将类转换为其他类型，需要创建原型如下所示的类成员函数：

```
operator type_name();
```

虽然该函数没有声明返回类型，但应返回所需类型的值。

使用转换函数时要小心。可以在声明构造函数时使用关键字 explicit，以防止它被用于隐式转换。

3. 其构造函数使用 new 的类

如果类使用 new 操作符来分配类成员指向的内存，在设计时应采取一些预防措施（前面总结了这些预防措施，应牢记这些规则，这是因为编译器并不知道这些规则，因此无法发现错误）。

- 对于指向的内存是由 new 分配的所有类成员，都应在类的析构函数中对其使用 delete，该操作符将释放分配的内存。
- 如果析构函数通过对指针类成员使用 delete 来释放内存，则每个构造函数都应当使用 new 来初始化指针，或将它设置为空指针。
- 构造函数中要么使用 new [ ]，要么使用 new，而不能混用。如果构造函数使用的是 new[ ]，则析构函数应使用 delete [ ]；如果构造函数使用的是 new，则析构函数应使用 delete。
- 应定义一个分配内存（而不是将指针指向已有内存）的复制构造函数。这样程序能够将类对象初始化为另一个类对象。这种构造函数的原型通常如下：

  ```
  className (const className  &)
  ```

- 应定义一个重载赋值操作符的类成员函数，其函数定义如下（其中 c_pointer 是 c_name 的类成员，类型为指向 type_name 的指针）。下面的范例假设使用 new [ ]来初始化变量 c_pointer：

```
c_name  & c_name::operator= (const c_name  & cn)
{
   if (this == & cn_)
      return *this;    // done if self-assignment
   delete[ ] c_pointer;
   // set size number of type_name units to be copied
   c_pointer = new type_name[size];
   // then copy data pointed to by cn.c_pointer to
   // location pointed to by c_pointer
   ...
   return *this;
}
```

12.2　队列模拟

进一步了解类后，可将这方面的知识用于解决编程问题。Heather 银行打算在 Food Heap 超市开设一个自动柜员机（ATM）。Food Heap 超市的管理者担心排队等待使用 ATM 的人流会干扰超市的交通，希望限制排队等待的人数。Heather 银行希望对顾客排队等待的时间进行估测。要编写一个程序来模拟这种情况，让超市的管理者可以了解 ATM 可能造成的影响。

对于这种问题，最自然的方法是使用顾客队列。队列是一种抽象的数据类型（Abstract Data Type，ADT），可以存储有序的项目序列。新项目被添加在队尾，并可以删除队首的项目。队列有点像堆栈，不过堆栈在同一端进行添加和删除。这使得栈是一种后进先出（LIFO，last-in，first-out）的结构，而队列是先进先出（FIFO，first-in，first-out）的。从概念上说，队列就好比是收款台或者 ATM 前面排的队，所以对于上述问题，队列非常合适。因此，工程的任务之一是定义一个 Queue 类（第 16 章将介绍标准模板库 queue 类，也将介绍如何开发自己的类）。

队列中的项目是顾客。Heather 银行的代表介绍：通常，三分之一的顾客只需要一分钟便可获得服务，三分之一的顾客需要两分钟，另外三分之一的顾客需要三分钟。另外，顾客到达的时间是随机的，但每个小时使用自动柜员机的顾客数量相当稳定。工程的另外两项任务是：设计一个表示顾客的类；编写一个程序来模拟顾客和队列之间的交互（参见图 12.7）。

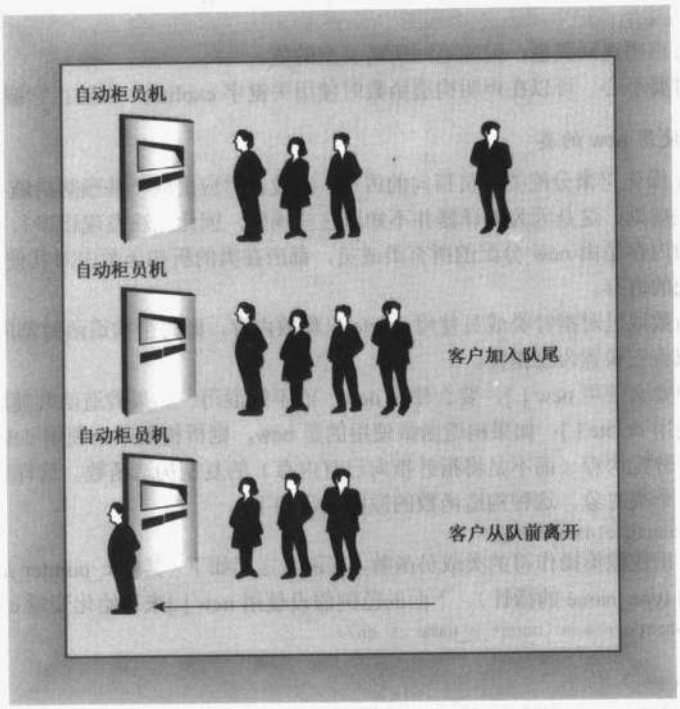


图 12.7 队列

12.2.1 队列类

首先需要设计一个 Queue 类。这里先列出队列的特征：

- 队列存储有序的项目序列。
- 队列所能容纳的项目数有一定的限制。
- 应当能够创建空队列。
- 应当能够检查队列是否为空。
- 应当能够检查队列是否是满的。
- 应当能够在队尾添加项目。
- 应当能够从队首删除项目。
- 应当能够确定队列中项目数。

设计类时，需要开发公有接口和私有实现。

1. Queue 类的接口

从队列的特征可知，Queue 类的公有接口应该如下：

```
class Queue
{
   enum {Q_SIZE = 10};
private:
// private representation to be developed later
public:
   Queue (int qs = Q_SIZE); // create queue with a qs limit
   ~Queue();
   bool isempty()const;
   bool isfull()const;
   int queuecount()const;
   bool enqueue (const Item &item);   // add item to end
   bool dequeue (Item &item);         // remove item from front
};
```

构造函数创建一个空队列。默认情况下，队列最多可存储 10 个项目，但是可以用显式初始化参数覆盖该默认值：

```
Queue line1;              // queue with 10-item limit
Queue line2 (20);         // queue with 20-item limit
```

使用队列时，可以使用 typedef 来定义 Item（第 14 章将介绍如何使用类模板）。

2. Queue 类的实现

确定接口后，便可以实现它。首先，需要确定如何表示队列数据。一种方法是使用 new 动态分配一个数组，它包含所需的元素数。不过，对于队列操作而言，数组并不太合适。例如，删除数组的第一个元素后，需要将余下的所有元素向前移动一位；否则需要作一些更费力的工作，如将数组视为是循环的。不过，链表能够很好地满足队列的要求。链表由节点序列构成。每一个节点中都包含要保存到链表中的信息以及一个指向下一个节点的指针。对于这里的队列来说，数据部分都是一个 Item 类型的值，因此可以使用下面的结构来表示节点：

```
struct Node
{
    Item item;              // data stored in the node
    struct Node * next;     // pointer to next node
};
```

图 12.8 说明了链表。

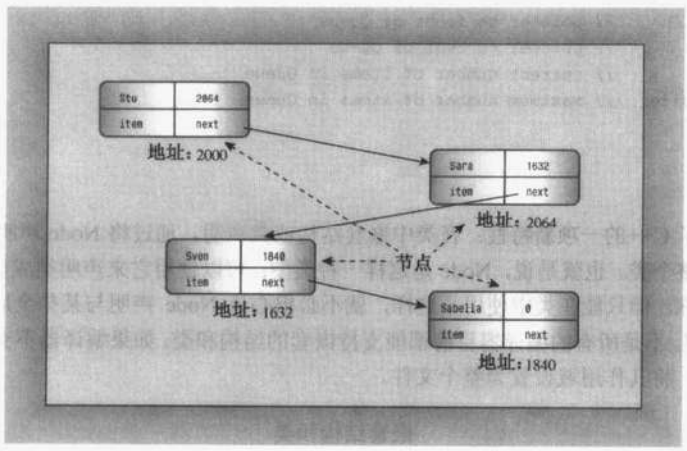


图 12.8　链表

如图 12.8 所示是一个单向链表，因为每个节点都只包含一个指向其他节点的指针。知道第一个节点的地址后，就可以沿指针找到后面的每一个节点。通常，链表的最后一个节点中的指针被设置为 NULL（或 0），以指出后面没有节点了。要跟踪链表，必须知道第一个节点的地址。可以让 Queue 类的一个数据成员指向链表的起始位置。具体地说，这是所需要的全部信息，有了这种信息后，就可以沿节点链找到任何节点。不过，由于队列总是将新项目添加到队尾，因此包含一个指向最后一个节点的数据成员将非常方便（参见图 12.9）。此外，还可以使用数据成员来跟踪队列可存储的最大项目数以及当前的项目数。所以，类声明的私有部分与下面类似：

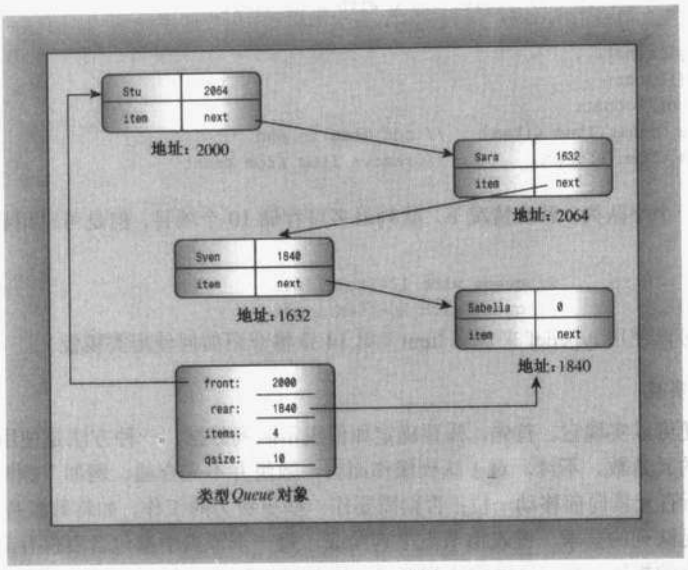


图 12.9 Queue 对象

```
class Queue
{
private:
// class scope definitions
   // Node is a nested structure definition local to this class
   struct Node { Item item; struct Node * next; };
   enum {Q_SIZE = 10};
//private class member:
   Node * front;      // pointer to front of Queue
   Node * rear;       // pointer to rear of Queue
   int items;         // current number of items in Queue
   const int qsize;   // maximum number of items in Queue
   ...
public:
//...
};
```

上述声明使用了 C++的一项新特性：在类中嵌套结构或类声明。通过将 Node 声明放在 Queue 类中，可以使其作用域为整个类。也就是说，Node 是这样一种类型：可以使用它来声明类成员，也可以将它作为类方法中的类型名称，但只能在类中使用。这样，就不必担心该 Node 声明与某些全局声明或其他类中声明的 Node 发生冲突。不是所有的编译器目前都能支持嵌套的结构和类。如果编译器不支持，则必须将 Node 结构定义为全局的，将其作用域设置为整个文件。

嵌套结构和类

在类声明中声明的结构、类或枚举被称为是被嵌套在类中，其作用域为整个类。这种声明不会创建数

据对象，而只是指定了可以在类中使用的类型。如果声明是在类的私有部分进行的，则只能在这个类使用被声明的类型；如果声明是在公有部分进行的，则可以从类的外部通过作用域解析操作符使用被声明的类型。例如，如果 Node 是在 Queue 类的公有部分声明的，则可以在类的外面声明 Queue::Node 类型的变量。

设计好数据的表示方式后，接下来需要编写类方法。

3. 类方法

类构造函数应提供类成员的值。由于在这个例子中，队列最初是空的，因此队首和队尾指针都设置为 NULL（或 0），并将 items 设置为 0。另外，还应将队列的最大长度 qsize 设置为构造函数参数 qs 的值。下面的实现方法无法正常运行：

```
Queue::Queue (int qs)
{
    front = rear = NULL;
    items = 0;
    qsize = qs;    // not acceptable!
}
```

问题在于 *qsize* 是常量，所以可以对它进行初始化，但不能给它赋值。从概念上说，调用构造函数时，对象将在括号中的代码执行之前被创建。因此，调用 Queue（int qs）构造函数将导致程序首先给 4 个成员变量分配内存。然后，程序流程进入到括号中，使用常规的赋值方式将值存储到内存中。因此，对于 const 数据成员，必须在执行到构造函数体之前，即创建对象时进行初始化。C++提供了一种特殊的句法来完成上述工作，它叫做成员初始化列表（member initializer list）。成员初始化列表由逗号分隔的初始化列表组成（前面带冒号）。它位于参数列表的右括号之后、函数体左括号之前。如果数据成员的名称为 mdata，并需要将它初始化为 val，则初始化器为 mdata（val）。使用这种表示法，可以这样编写 Queue 的构造函数：

```
Queue::Queue (int qs) : qsize (qs)  // initialize qsize to qs
{
    front = rear = NULL;
    items = 0;
}
```

通常，初值可以是常量或构造函数的参数列表中的参数。这种方法并不限于初始化常量，可以将 Queue 构造函数写成如下所示：

```
Queue::Queue (int qs) : qsize (qs), front (NULL), rear (NULL), items (0)
{
}
```

只有构造函数可以使用这种初始化列表句法。如上所示，对于 const 类成员，必须使用这种句法。另外，对于被声明为引用的类成员，也必须使用这种句法：

```
class Agency {...};
class Agent
{
private:
    Agency & belong;    // must use initializer list to initialize
    ...
};
Agent::Agent (Agency & a) : belong (a) {...}
```

这是因为引用与 const 数据类似，只能在被创建时进行初始化。对于简单数据成员（例如 front 和 items），使用成员初始化列表和在函数体中使用赋值没有什么区别。不过，正如第 14 章将要介绍的，对于本身就是类对象的成员来说，使用成员初始化列表的效率更高。

成员初始化列表的句法

如果 Classy 是一个类，而 mem1、mem2 和 mem3 都是这个类的数据成员，则类构造函数可以使用如下的句法来初始化数据成员：

```
Classy::Classy (int n, int m) : mem1 (n), mem2 (0), mem3 (n*m + 2)
{
```

```
//...
}
```

上述代码将 mem1 初始化为 *n*，将 mem2 初始化为 0，将 mem3 初始化为 *n***m* + 2。从概念上说，这些初始化工作是在对象创建时完成的，此时还未执行括号中的任何代码。请注意以下几点：

- 这种格式只能用于构造函数。
- 必须用这种格式来初始化非静态 const 数据成员。
- 必须用这种格式来初始化引用数据成员。

数据成员被初始化的顺序与它们出现在类声明中的顺序相同，与初始化器中的排列顺序无关。

警告： 不能将成员初始化列表句法用于构造函数之外的其他类方法。

成员初始化列表中使用的括号方式也可用于常规的初始化中。也就是说，如果愿意，可以将下述代码：

```
int games = 162;
double talk = 2.71828;
```

替换为：

```
int games (162);
double talk (2.71828);
```

这使得初始化内置类型就像初始化类对象一样。

isempty()、isfull()和 queuecount()的代码都非常简单。如果 items 为 0，则队列是空的；如果 items 等于 qsize，则队列是满的。要知道队列中的项目数，只需返回 items 的值。后面的程序清单 12.11 列出了这些代码。

将项目添加到队尾（入队）比较麻烦。下面是一种方法：

```
bool Queue::enqueue (const Item & item)
{
    if (isfull())
        return false;
    Node * add = new Node;  // create node
    if (add == NULL)
        return false;       // quit if none available
    add->item = item;       // set node pointers
    add->next = NULL;
    items++;
    if (front == NULL)      // if queue is empty,
        front = add;        // place item at front
    else
        rear->next = add;   // else place at rear
    rear = add;             // have rear point to new node
    return true;
}
```

简而言之，方法需要经过下面几个阶段（见图 12.10）：

（1）如果队列已满，则结束（在这里的实现中，队列的最大长度由用户通过构造函数指定）。

（2）创建一个新节点，如果无法创建新节点（例如，如果请求内存失败）则结束。

（3）在节点中放入正确的值。在这个例子中，代码将 Item 值复制到节点的数据部分，并将节点的 next 指针设置为 NULL。这样就为将节点作为队列中的最后一个项目做好了准备。

（4）将项目计数（items）加 1。

（5）将节点附加到队尾。这包括两个部分：首先，将节点与列表中的另一个节点连接起来。这是通过将当前队尾节点的 next 指针指向新的队尾节点来完成的。第二部分是将 Queue 的成员指针 rear 设置为指向新节点，使队列可以直接访问最后一个节点。如果队列为空，则还必须将 front 指针设置成指向新节点（如果只有一个节点，则它既是队首节点，也是队尾节点）。

删除队首项目（出队）也需要多个步骤才能完成。下面是一种方式：

```
bool Queue::dequeue (Item & item)
{
    if (front == NULL)
```

```
    return false;
  item = front->item;     // set item to first item in queue
  items--;
  Node * temp = front;    // save location of first item
  front = front->next;    // reset front to next item
  delete temp;            // delete former first item
  if (items == 0)
    rear = NULL;
  return true;
}
```

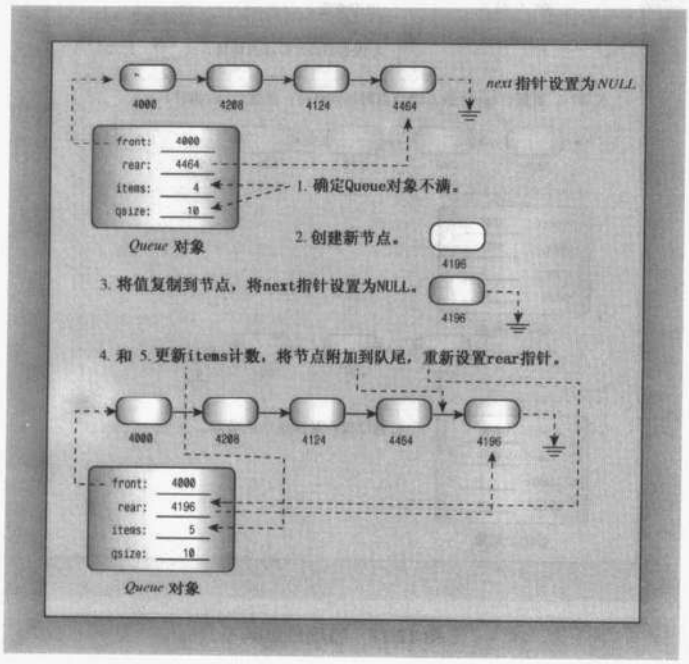


图 12.10　将项目入队

简而言之，需要经过下面几个阶段（参见图 12.11）：

（1）如果队列为空，则结束。

（2）将队列的第一个项目提供给调用函数，这是通过将当前 front 节点中的数据部分复制到传递给方法的引用变量中来实现。

（3）将项目计数（items）减 1。

（4）保存 front 节点的位置，供以后删除。

（5）让节点出队。这是通过将 Queue 成员指针 front 设置成指向下一个节点来完成的，该节点的位置由 front->next 提供。

（6）为节省内存，删除以前的第一个节点。

（7）如果链表为空，则将 rear 设置为 NULL（在这个例子中，将 front 指针设置成 front->next 后，它已经是 NULL 了）。

步骤 4 是必不可少的，这是因为步骤 5 将删除关于先前第一个节点位置的信息。

4. 是否需要其他类方法

是否需要其他方法呢？类构造函数没有使用 new，所以乍一看，好像不用理会由于在构造函数中使用 new 给类带来的特殊要求。当然，这种印象是错误的，因为向队列中添加对象将调用 new 来创建新的节点。通过删除节点的方式，dequeue() 方法确实可以清除节点，但这并不能保证队列在到期时为空。因此，类需

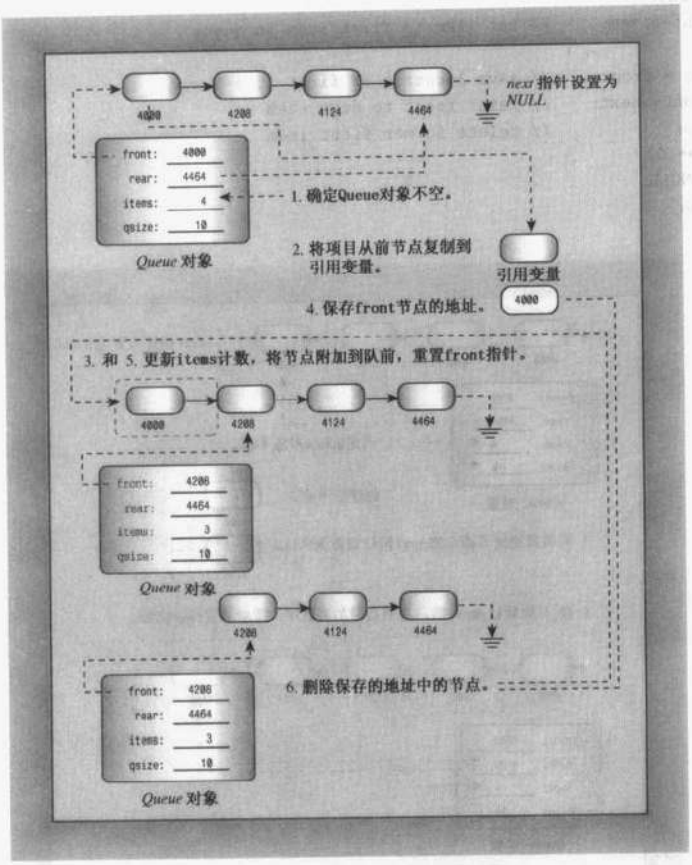


图 12.11 将项目出队

要一个显式析构函数——该函数删除剩余的所有节点。下面是一种实现，它从链表头开始，依次删除其中的每个节点：

```
Queue::~Queue()
{
    Node * temp;
    while (front != NULL)          // while queue is not yet empty
    {
        temp = front;              // save address of front item
        front = front->next;       // reset pointer to next item
        delete temp;               // delete former front
    }
}
```

您知道，使用 new 的类通常需要包含显式复制构造函数和执行深度复制的赋值操作符，这个例子也是如此吗？首先要回答的问题是，默认的成员复制是否合适？答案是否定的。复制 Queue 对象的成员将生成一个新的对象，该对象指向链表原来的头和尾。因此，将项目添加到复制的 Queue 对象中，将修改共享的链表。这样做将造成非常严重的后果。更糟的是，只有副本的尾指针得到了更新，从原始对象的角度看，这将损坏链表。显然，要克隆或复制队列，必须提供复制构造函数和执行深度复制的赋值构造函数。

当然，这提出了这样一个问题：为什么要复制队列呢？也许是希望在模拟的不同阶段保存队列的瞬像，也可能是希望为两个不同的策略提供相同的输入。实际上，拥有拆分队列的操作是非常有用的，超市在开设额外的收款台时经常这样做。同样，也可能希望将两个队列结合成一个或者截短一个队列。

但是在这里的模拟中，将不实现上述功能。难道不能忽略这些问题，而使用已有的方法吗？当然可以。

不过，在将来的某个时候，可能需要再次使用队列且需要复制。另外，您可能会忘记没有为复制提供适当的代码。在这种情况下，程序将能编译和运行，但结果却是混乱的，甚至会崩溃。因此，最好还是提供复制构造函数和赋值操作符，尽管目前并不需要它们。

幸运的是，有一种小小的技巧可以避免这些额外的工作，并确保程序不会崩溃。这就是将所需的方法定义为伪私有方法：

```
class Queue
{
private:
   Queue (const Queue & q): qsize (0) { }   // preemptive definition
   Queue & operator= (const Queue & q) { return *this; }
//...
};
```

这样做有两个作用：第一，它避免了本来将自动生成的默认方法定义。第二，因为这些方法是私有的，所以不能被广泛使用。也就是说，如果 nip 和 tuck 是 Queue 对象，则编译器就不允许这样做：

```
Queue snick (nip);    // not allowed
tuck = nip;           // not allowed
```

所以，与其将来面对无法预料的运行故障，不如得到一个易于跟踪的编译错误，指出这些方法是不可访问的。另外，在定义其对象不允许被复制的类时，这种方法也很有用。

还有没有其他影响需要注意呢？当然有。当对象被按值传递（或返回）时，复制构造函数将被调用。不过，如果遵循优先采用按引用传递对象的惯例，将不会有任何问题。另外，复制构造函数还被用于创建其他的临时对象，但 Queue 定义中并没有导致创建临时对象的操作，例如重载加法操作符。

12.2.2　Customer 类

接下来需要设计客户类。通常，ATM 客户有很多属性，例如姓名、账户和账户结余。不过，这里的模拟需要使用的惟一一个属性是客户何时进入队列以及客户交易所需的时间。当模拟生成新客户时，程序将创建一个新的客户对象，并在其中存储客户的到达时间以及一个随机生成的交易时间。当客户到达队首时，程序将记录此时的时间，并将其与进入队列的时间相减，得到客户的等候时间。下面的代码演示了如何定义和实现 Customer 类：

```
class Customer
{
private:
   long arrive;       // arrival time for customer
   int processtime;   // processing time for customer
public:
   Customer() { arrive = processtime = 0; }
   void set (long when);
   long when()const { return arrive; }
   int ptime()const { return processtime; }
};
void Customer::set (long when)
{
   processtime = std::rand()% 3 + 1;
   arrive = when;
}
```

默认构造函数创建一个空客户。set() 成员函数将到达时间设置为参数，并将处理时间设置为 1～3 中的一个随机值。

程序清单 12.10 将 Queue 和 Customer 类声明放到一起，而程序清单 12.11 列出了方法。

程序清单 12.10　queue.h

```
// queue.h -- interface for a queue
#ifndef QUEUE_H_
#define QUEUE_H_
```

```
// This queue will contain Customer items
class Customer
{
private:
    long arrive;       // arrival time for customer
    int processtime;    // processing time for customer
public:
    Customer() { arrive = processtime = 0; }
    void set(long when);
    long when() const { return arrive; }
    int ptime() const { return processtime; }
};

typedef Customer Item;

class Queue
{
private:
// class scope definitions
    // Node is a nested structure definition local to this class
    struct Node { Item item; struct Node * next; };
    enum {Q_SIZE = 10};
// private class members
    Node * front;       // pointer to front of Queue
    Node * rear;        // pointer to rear of Queue
    int items;          // current number of items in Queue
    const int qsize;    // maximum number of items in Queue
    // preemptive definitions to prevent public copying
    Queue(const Queue & q): qsize(0) { }
    Queue & operator=(const Queue & q) { return *this; }
public:
    Queue(int qs = Q_SIZE); // create queue with a qs limit
    ~Queue();
    bool isempty() const;
    bool isfull() const;
    int queuecount() const;
    bool enqueue(const Item &item);   // add item to end
    bool dequeue(Item &item);         // remove item from front
};
#endif
```

程序清单 12.11 queue.cpp

```
// queue.cpp -- Queue and Customer methods
#include "queue.h"
#include <cstdlib>        // (or stdlib.h) for rand()

// Queue methods
Queue::Queue(int qs): qsize(qs)
{
    front = rear = NULL;
    items = 0;
}

Queue::~Queue()
{
    Node * temp;
    while (front != NULL)         // while queue is not yet empty
    {
        temp = front;             // save address of front item
        front = front->next;      // reset pointer to next item
```

```
        delete temp;                // delete former front
    }
}
bool Queue::isempty()const
{
    return items == 0;
}
bool Queue::isfull()const
{
    return items == qsize;
}

int Queue::queuecount()const
{
    return items;
}

// Add item to queue
bool Queue::enqueue (const Item & item)
{
    if (isfull())
        return false;
    Node * add = new Node;  // create node
    if (add == NULL)
        return false;       // quit if none available
    add->item = item;       // set node pointers
    add->next = NULL;
    items++;
    if (front == NULL)      // if queue is empty,
        front = add;        // place item at front
    else
        rear->next = add;   // else place at rear
    rear = add;             // have rear point to new node
    return true;
}

// Place front item into item variable and remove from queue
bool Queue::dequeue (Item & item)
{
    if (front == NULL)
        return false;
    item = front->item;     // set item to first item in queue
    items--;
    Node * temp = front;    // save location of first item
    front = front->next;    // reset front to next item
    delete temp;            // delete former first item
    if (items == 0)
        rear = NULL;
    return true;
}

// customer method

// when is the time at which the customer arrives
// the arrival time is set to when and the processing
// time set to a random value in the range 1 - 3

void Customer::set (long when)
{
    processtime = std::rand()% 3 + 1;
```

```
    arrive = when;
}
```

注意：编译器如果没有实现 bool，可以用 int 代替 bool，0 代替 false，1 代替 true。另外，可能还必须使用 stdlib.h 代替较新的 cstdlib。

12.2.3 模拟

现在已经拥有模拟 ATM 所需的工具。程序允许用户输入 3 个数：队列的最大长度、程序模拟的持续时间（单位为小时）以及平均每小时的客户数。程序将使用循环——每次循环代表一分钟。在每分钟的循环中，程序将完成下面的工作：

1. 判断是否来了新的客户。如果来了，并且此时队列未满，则将它添加到队列中，否则拒绝客户入队。
2. 如果没有客户在进行交易，则选取队列的第一个客户。确定该客户的已等候时间，并将 wait_time 计数器设置为新客户所需的处理时间。
3. 如果客户正在处理中，则将 wait_time 计数器减 1。
4. 记录各种数据，如获得服务的客户数目、被拒绝的客户数目、排队等候的累积时间以及累积的队列长度等。

当模拟循环结束时，程序将报告各种统计结果。

一个有趣的问题是，程序如何确定是否有新的客户到来。假设平均每小时有 10 名客户到达，则相当于每 6 分钟有一名客户。程序将计算这个值，并将它保存在 min_per_cust 变量中。不过，刚好每 6 分钟来一名客户不太现实，我们真正（至少在大部分时间内）希望的是一个更随机的过程——但平均每 6 分钟来一名客户。程序将使用下面的函数来确定是否在循环期间有客户到来：

```
bool newcustomer (double x)
{
    return (std::rand()* x / RAND_MAX < 1);
}
```

其工作原理如下：值 RAND_MAX 是在 cstdlib 文件（以前是 stdlib.h）中定义的，是 rand() 函数可能返回的最大值（0 是最小值）。假设客户到达的平均间隔时间 x 为 6，则 rand()* x /RAND_MAX 的值将位于 0 到 6 之间。具体地说，平均每隔 6 次，这个值会有 1 次小于 1。不过，这个函数可能会导致客户到达的时间间隔有时为 1 分钟，有时为 20 分钟。这种方法虽然很笨拙，但可使实际情况不同于有规则地每 6 分钟到来一个客户。如果客户到达的平均时间间隔少于 1 分钟，则上述方法将无效，但模拟并不是针对这种情况设计的。如果确实需要处理这种情况，最好提高时间分辨率，比如每次循环代表 10 秒钟。

注意：有些编译器没有定义 RAND_MAX。在这种情况下，可以使用#define 或 const int 给 RAND_MAX 定义一个值。如果在文档中不能找到正确的值，请尝试使用 climits 中的 INT_MAX 或 limits.h 头文件指定的最大 int 值。

程序清单 12.12 给出了模拟的细节。长时间运行该模拟程序，可以知道长期的平均值；短时间运行该模拟程序，将只能知道短期的变化。

程序清单 12.12 bank.cpp

```
// bank.cpp -- using the Queue interface
// compile with queue.cpp
#include <iostream>
#include <cstdlib> // for rand()and srand()
#include <ctime>   // for time()
#include "queue.h"
const int MIN_PER_HR = 60;

bool newcustomer (double x); // is there a new customer?

int main()
{
```

```
    using std::cin;
    using std::cout;
    using std::endl;
    using std::ios_base;
// setting things up
    std::srand (std::time (0));       // random initializing of rand()

    cout << "Case Study: Bank of Heather Automatic Teller\n";
    cout <<"Enter maximum size of queue: ";
    int qs;
    cin >> qs;
    Queue line (qs);          // line queue holds up to qs people

    cout <<"Enter the number of simulation hours: ";
    int hours;              // hours of simulation
    cin >> hours;
    // simulation will run 1 cycle per minute
    long cyclelimit = MIN_PER_HR * hours; // # of cycles

    cout << "Enter the average number of customers per hour: ";
    double perhour;           // average # of arrival per hour
    cin >> perhour;
    double min_per_cust;      // average time between arrivals
    min_per_cust = MIN_PER_HR / perhour;

    Item temp;                // new customer data
    long turnaways = 0;       // turned away by full queue
    long customers = 0;       // joined the queue
    long served = 0;          // served during the simulation
    long sum_line = 0;        // cumulative line length
    int wait_time = 0;        // time until autoteller is free
    long line_wait = 0;       // cumulative time in line

// running the simulation
    for (int cycle = 0; cycle < cyclelimit; cycle++)
    {
        if (newcustomer (min_per_cust)) // have newcomer
        {
            if (line.isfull())
                turnaways++;
            else
            {
                customers++;
                temp.set (cycle);    // cycle = time of arrival
                line.enqueue (temp); // add newcomer to line
            }
        }
        if (wait_time <= 0 && !line.isempty())
        {
            line.dequeue (temp);      // attend next customer
            wait_time = temp.ptime(); // for wait_time minutes
            line_wait += cycle - temp.when();
            served++;
        }
        if (wait_time > 0)
            wait_time--;
        sum_line += line.queuecount();
    }

// reporting results
```

```
    if (customers > 0)
    {
        cout << "customers accepted: "<< customers << endl;
        cout << "  customers served: " << served << endl;
        cout << "         turnaways: " << turnaways << endl;
        cout << "average queue size: ";
        cout.precision(2);
        cout.setf(ios_base::fixed, ios_base::floatfield);
        cout.setf(ios_base::showpoint);
        cout << (double) sum_line / cyclelimit << endl;
        cout << " average wait time: "
             << (double) line_wait / served << " minutes\n";
    }
    else
        cout << "No customers!\n";
    cout << "Done!\n";
    return 0;
}

// x = average time, in minutes, between customers
// return value is true if customer shows up this minute
bool newcustomer(double x)
{
    return (std::rand() * x / RAND_MAX < 1);
}
```

注意：编译器如果没有实现 bool，可以用 int 代替 bool，用 0 代替 false，用 1 代替 true；还可能必须使用 stdlib.h 和 time.h 代替较新的 cstdlib 和 ctime；另外可能必须自己来定义 RAND_MAX。

下面是程序清单 12.10～12.12 组成的程序长时间运行的几个例子：

```
Case Study: Bank of Heather Automatic Teller
Enter maximum size of queue: 10
Enter the number of simulation hours: 100
Enter the average number of customers per hour: 15
customers accepted: 1485
   customers served: 1485
       turnaways: 0
average queue size: 0.15
 average wait time: 0.63 minutes

Case Study: Bank of Heather Automatic Teller
Enter maximum size of queue: 10
Enter the number of simulation hours: 100
Enter the average number of customers per hour: 30
customers accepted: 2896
   customers served: 2888
       turnaways: 101
average queue size: 4.64
 average wait time: 9.63 minutes

Case Study: Bank of Heather Automatic Teller
Enter maximum size of queue: 20
Enter the number of simulation hours: 100
Enter the average number of customers per hour: 30
customers accepted: 2943
   customers served: 2943
       turnaways: 93

average queue size: 13.06
 average wait time: 26.63 minutes
```

注意，每小时到达的客户从 15 名增加到 30 名时，等候时间并不是加倍，而是增加了 15 倍。如果允许队列更长，情况将更糟。不过，模拟没有考虑到这个事实——许多客户由于不愿意排很长的队而离开了。

下面也是该程序的另外几个运行范例。从中可以知道，即使平均每小时到达的客户数不变，也会出现短期变化。

```
Case Study: Bank of Heather Automatic Teller
Enter maximum size of queue: 10
Enter the number of simulation hours: 4
Enter the average number of customers per hour: 30
customers accepted: 114
   customers served: 110
       turnaways: 0
average queue size: 2.15
  average wait time: 4.52 minutes

Case Study: Bank of Heather Automatic Teller
Enter maximum size of queue: 10
Enter the number of simulation hours: 4
Enter the average number of customers per hour: 30
customers accepted: 121
   customers served: 116
       turnaways: 5
average queue size: 5.28
  average wait time: 10.72 minutes

Case Study: Bank of Heather Automatic Teller
Enter maximum size of queue: 10
Enter the number of simulation hours: 4
Enter the average number of customers per hour: 30
customers accepted: 112
   customers served: 109
       turnaways: 0
average queue size: 2.41
  average wait time: 5.16 minutes
```

12.3　总结

本章介绍了定义和使用类的许多重要方面。其中的一些方面是非常微妙、甚至很难理解的概念。如果其中的某些概念对于您来说过于复杂，也不用害怕——这些问题对于大多数 C++的初学者来说都是很难的。通常，对于诸如复制构造函数等概念，都是在由于忽略它们而遇到了麻烦后逐步理解的。本章介绍的一些内容乍看起来非常难以理解，但是随着经验越来越丰富，对其理解也将越透彻。

在类构造函数中，可以使用 new 为数据分配内存，然后将内存地址赋给类成员。这样，类便可以处理长度不同的字符串，而不用在类设计时提前固定数组的长度。在类构造函数中使用 new，也可能在对象过期时引发问题。如果对象包含成员指针，同时它指向的内存是由 new 分配的，则释放用于保存对象的内存并不会自动释放对象成员指针指向的内存。因此在类构造函数中使用 new 类来分配内存时，应在类析构函数中使用 delete 来释放分配的内存。这样，当对象过期时，将自动释放其指针成员指向的内存。

如果对象包含指向 new 分配的内存的指针成员，则将一个对象初始化为另一个对象，或将一个对象赋给另一个对象时，也会出现问题。在默认情况下，C++逐个对成员进行初始化和赋值，这意味着被初始化或被赋值的对象的成员将与原始对象完全相同。如果原始对象的成员指向一个数据块，则副本成员将指向同一个数据块。当程序最终删除这两个对象时，类的析构函数将试图删除同一个内存数据块两次，这将出错。解决方法是：定义一个特殊的复制构造函数来重新定义初始化，并重载赋值操作符。在上述任何一种情况下，新的定义都将创建指向数据的副本，并使新对象指向这些副本。这样，旧对象和新对象都将引用

独立的、相同的数据，而不会重叠。由于同样的原因，必须定义赋值操作符。对于每一种情况，最终目的都是执行深度复制，也就是说，复制实际的数据，而不仅仅是复制指向数据的指针。

对象的存储持续性为自动或外部时，在它不再存在时将自动调用其析构函数。如果使用 new 操作符为对象分配内存，并将其地址赋给一个指针，则当您将 delete 用于该指针时将自动为对象调用析构函数。然而，如果使用布局 new 操作符（而不是常规 new 操作符）为类对象分配内存，则必须负责显式地为该对象调用析构函数，方法是使用指向该对象的指针调用析构函数方法。C++允许在类中包含结构、类和枚举定义。这些嵌套类型的作用域为整个类，这意味着它们被局限于类中，不会与其他地方定义的同名结构、类和枚举发生冲突。

C++为类构造函数提供了一种可用来初始化数据成员的特殊句法。这种句法包括冒号和由逗号分隔的初始化列表，被放在构造函数参数的右括号后，函数体的左括号之前。每一个初始化器都由被初始化的成员的名称和包含初始值的括号组成。从概念上来说，这些初始化操作是在对象创建时进行的，此时函数体中的语句还没有执行。句法如下：

```
queue(int qs) : qsize(qs), items(0), front(NULL), rear(NULL) { }
```

如果数据成员是非静态 const 成员或引用，则必须采用这种格式。

您可能已经注意到，与简单的 C 结构相比，需要注意的类细节要多得多。作为回报，它们的功能也更强。

12.4 复习题

1. 假设 String 类有如下私有成员：

```
class String{
{
private:
    char * str;    // points to string allocated by new
    int len;       // holds length of string
//...
};
```

a. 下述默认构造函数有什么问题？

```
String::String(){}
```

b. 下述构造函数有什么问题？

```
String::String(const char * s)
{
    str = s;
    len = strlen(s);
}
```

c. 下述构造函数有什么问题？

```
String::String(const char * s)
{
    strcpy(str, s);
    len = strlen(s);
}
```

2. 如果您定义了一个类，其指针成员是使用 new 初始化的，请指出可能出现的 3 个问题以及如何纠正这些问题。

3. 如果没有显式地提供类方法，编译器将自动生成哪些类方法？请描述这些隐式生成的函数的行为。

4. 找出并改正下述类声明中的错误：

```
class nifty
{
// data
    char personality[];
    int talents;
// methods
```

```
    nifty();
    nifty(char * s);
    ostream & operator<< (ostream & os, nifty & n);
};

nifty::nifty()
{
    personality = NULL;
    talents = 0;
}

nifty::nifty(char * s)
{
    personality = new char [strlen(s)];
    personality = s;
    talents = 0;
}

ostream & nifty::operator<< (ostream & os, nifty & n)
{
    os << n;
}
```

5. 对于下面的类声明：

```
class Golfer
{
private:
    char * fullname;        // points to string containing golfer's name
    int games;              // holds number of golf games played
    int * scores;           // points to first element of array of golf scores
public:
    Golfer();
    Golfer(const char * name, int g= 0);
     // creates empty dynamic array of g elements if g > 0
    Golfer(const Golfer & g);
    ~Golfer();
};
```

a. 下列各条语句将调用哪些类方法？

```
Golfer nancy;                       // #1
Golfer lulu("Little Lulu");         // #2
Golfer roy("Roy Hobbs", 12);        // #3
Golfer * par = new Golfer;          // #4
Golfer next = lulu;                 // #5
Golfer hazzard = "Weed Thwacker";   // #6
*par = nancy;                       // #7
nancy = "Nancy Putter";             // #8
```

b. 很明显，类需要有另外几种方法才能更有用，但是类需要那些方法才能防止数据被损坏呢？

12.5　编程练习

1. 对于下面的类声明：

```
class Cow {
    char name[20];
    char * hobby;
    double weight;
public:
    Cow();
    Cow(const char * nm, const char * ho, double wt);
```

```
    Cow(const Cow c&);
    ~Cow();
    Cow & operator=(const Cow & c);
    void ShowCow()const;  // display all cow data
};
```

给这个类提供实现，并编写一个使用所有成员函数的小程序。

2. 通过完成下面的工作来改进 String 类声明（即将 String1.h 升级为 String2.h）：

a. 对+操作符进行重载，使之可将两个字符串合并成 1 个。

b. 提供一个 Stringlow()成员函数，将字符串中所有的字母字符转换为小写（别忘了 cctype 系列字符函数）。

c. 提供 String()成员函数，将字符串中所有字母字符转换成大写。

d. 提供一个这样的成员函数，它接受一个 char 参数，返回该字符在字符串中出现的次数。

使用下面的程序来测试您的工作：

```
// pe12_2.cpp
#include <iostream>
using namespace std;
#include "string2.h"
int main()
{
    String s1(" and I am a C++ student. ");
    String s2 = "Please enter your name: ";
    String s3;
    cout << s2;                    // overloaded << operator
    cin >> s3;                     // overloaded >> operator
    s2 = "My name is " + s3;       // overloaded =, + operators
    cout << s2 << ".\n";
    s2 = s2 + s1;
    s2.stringup();              // converts string to uppercase
    cout << "The string\n" << s2 << "\ncontains " << s2.has('A')
         << "'A' characters in it.\n";
    s1 = "red";         // String(const char *),
                        // then String & operator=(const String&)
    String rgb[3] = { String(s1), String(" green"), String("blue") };
    cout << "Enter the name of a primary color for mixing light: ";
    String ans;
    bool success = false;
    while(cin >> ans)
    {
        ans.stringlow();        // converts string to lowercase
        for(int i = 0; i < 3; i++)
        {
            if(ans == rgb[i])  // overloaded == operator
            {
                cout << "That's right!\n";
                success = true;
                break;
            }
        }
        if(success)
            break;
        else
            cout << "Try again!\n";
    }
    cout << "Bye\n";
    return 0;
}
```

输出应与下面相似：

```
Please enter your name: Fretta Farbo
My name is Fretta Farbo.
The string
MY NAME IS FRETTA FARBO AND I AM A C++ STUDENT.
contains 6 'A' characters in it.
Enter the name of a primary color for mixing light: yellow
Try again!
BLUE
That's right!
Bye
```

3. 重新编写程序清单 10.7 和 10.8 描述的 Stock 类，使之使用动态分配的内存，而不是 string 类对象来存储股票名称。另外，使用重载的 operator < <() 定义代替 show() 成员函数。然后使用程序清单 10.9 测试新的定义程序。

4. 请看下面程序清单 10.10 定义的 Stack 类的变量：

```
// stack.h -- class declaration for the stack ADT
typedef unsigned long Item;

class Stack
{
private:
    enum {MAX = 10};        // constant specific to class
    Item * pitems;          // holds stack items
    int size;               // number of elements in stack
    int top;                // index for top stack item
public:
    Stack (int n = 10);     // creates stack with n elements
    Stack (const Stack & st);
    ~Stack();
    bool isempty()const;
    bool isfull()const;
    // push()returns false if stack already is full, true otherwise
    bool push (const Item & item); // add item to stack
    // pop()returns false if stack already is empty, true otherwise
    bool pop (Item & item);  // pop top into item
    Stack & operator= (const Stack & st);
};
```

正如私有成员表明的，这个类使用动态分配的数组来保存堆栈项。请重新编写方法，以适应这种新的表示法，并编写一个程序来演示所有的方法，包括复制构造函数和赋值操作符。

5. Heather 银行进行的研究表明，ATM 客户不希望排队时间不超过 1 分钟。使用程序清单 12.10 中的模拟，找出要使平均等候时间为 1 分钟，每小时到达的客户数应为多少（试验时间不短于 100 小时）？

6. Heather 银行想知道，如果再开设一台 ATM，情况将如何。请对模拟进行修改，以包含两个队列。假设当第一台 ATM 前的排队人数少于第二台 ATM 时，客户将排在第一队，否则将排在第二队。然后再找出要使平均等候时间为 1 分钟，每小时到达的客户数应该为多少（注意，这是一个非线性问题，即将 ATM 数量加倍，并不能保证每小时处理的客户数量也翻倍，同时客户等候的时间少于 1 分钟）？

第 13 章　类继承

本章内容包括：

- is-a 关系的继承。
- 如何以公有的方式从一个类派生出另一个类。
- 保护访问。
- 构造函数成员初始化列表
- 向上和向下强制转换。
- 虚拟成员函数。
- 早期（静态）联编与晚期（动态）联编。
- 抽象基类。
- 纯虚函数。
- 何时以及如何使用公有继承。

面向对象编程的主要目的之一是提供可重用的代码。开发新工程，尤其是当工程十分庞大时，重用经过测试的代码比重新编写代码要好得多。使用已有的代码可以节省时间，由于已有的代码已被使用和测试过，因此有助于避免在程序中引入错误。另外，必须考虑的细节越少，便越能专注于程序的整体策略。

传统的 C 函数库通过预定义、预编译的函数（如 strlen()和 rand()，可以在程序中使用这些函数）提供了可重用性。许多厂商都提供了专用的 C 库，这些专用库提供标准 C 库没有的函数。例如，可以购买数据库管理函数库和屏幕控制函数库。不过，函数库也有局限性。除非厂商提供了库函数的源代码（通常是不提供的），否则您将无法根据自己特定的需求，对函数进行扩展或修改，而必须根据库的情况修改自己的程序。即使厂商提供了源代码，在修改时也有一定的风险，如不经意地修改了函数的工作方式或改变了库函数之间的关系。

C++类提供了更高层次的重用性。目前，许多厂商提供了类库，类库由类声明和实现构成。因为类组合了数据表示和类方法，因此提供了比函数库更加完整的程序包。例如，单个类就可以提供用于管理对话框的全部资源。通常，类库是以源代码的方式提供的，这意味着可以对其进行修改，以满足需求。但是，C++提供了比修改代码更好的方法来扩展和修改类。这种方法叫作类继承（class inheritance），它能够从已有的类派生出新的类，而派生类继承了原有类（称为基类）的特征，包括方法。正如继承一笔财产要比自己白手起家容易一样，通过继承派生出的类通常比设计新类要容易得多。下面是可以通过继承完成的一些工作。

- 可以在已有类的基础上添加功能。例如，对于数组类，可以添加数学运算。
- 可以给类添加数据。例如，对于字符串类，可以派生出一个类，并添加指定字符串显示颜色的数据成员。
- 可以修改类方法的行为。例如，对于代表提供给飞机乘客的服务的 Passenger 类，可以派生出提供更高级别服务的 FirstClassPassenger 类。

当然，可以通过复制原始类代码，并对其进行修改来完成上述工作，但继承机制只需提供新特性，甚至不需要访问源代码就可以派生出类。因此，如果购买的类库只提供了类方法的头文件和编译后代码，仍可以使用库中的类派生出新的类。而且可以在不公开实现的情况下将自己的类分发给其他人，同时允许他

们在类中添加新特性。

继承是一种非常好的概念，其基本实现非常简单。但要对继承进行管理，使之在所有情况下都能正常工作，则需要做一些调整。本章将介绍继承简单的一面和复杂的一面。

13.1　一个简单的基类

从一个类派生出另一个类时，原始类称为基类，继承类称为派生类。为说明继承，首先需要一个基类。Webtown 俱乐部决定跟踪乒乓球会会员。作为俱乐部的首席程序员，需要设计一个简单的 TableTennisPlayer 类，如程序清单 13.1 和 13.2 所示。

程序清单 13.1　tabtenn0.h

```
// tabtenn0.h-- a table-tennis base class
#ifndef TABTENN0_H_
#define TABTENN0_H_
// simple base class
class TableTennisPlayer
{
private:
    enum {LIM = 20};
    char firstname[LIM];
    char lastname[LIM];
    bool hasTable;
public:
    TableTennisPlayer (const char * fn = "none",
                const char * ln = "none", bool ht = false);
    void Name()const;
    bool HasTable()const { return hasTable; };
    void ResetTable (bool v) { hasTable = v; };
};
#endif
```

程序清单 13.2　tabtenn0.cpp

```
//tabtenn0.cpp -- simple base class methods
#include "tabtenn0.h"
#include <iostream>
#include <cstring>

TableTennisPlayer::TableTennisPlayer (const char * fn, const char * ln,
            bool ht)
{
    std::strncpy (firstname, fn, LIM - 1);
    firstname[LIM - 1] = '\0';
    std::strncpy (lastname, ln, LIM - 1);
    lastname[LIM - 1] = '\0';
    hasTable = ht;
}

void TableTennisPlayer::Name()const
{
    std::cout << lastname << ", " << firstname;
}
```

TableTennisPlayer 类只是记录会员的姓名以及是否有球桌。程序清单 13.3 使用了这个类。

程序清单 13.3　usett0.cpp

```
// usett0.cpp -- using a base class
```

```
#include <iostream>
#include "tabtenn0.h"

int main(void)
{
    using std::cout;
    TableTennisPlayer player1("Chuck", "Blizzard", true);
    TableTennisPlayer player2("Tara", "Boomdea", false);
    player1.Name();
    if (player1.HasTable())
        cout << ": has a table.\n";
    else
        cout << ": hasn't a table.\n";
    player2.Name();
    if (player2.HasTable())
        cout << ": has a table";
    else
        cout << ": hasn't a table.\n";

    return 0;
}
```

下面是程序清单 13.1～13.3 组成的程序的输出：

```
Blizzard, Chuck: has a table.
Boomdea, Tara: hasn't a table.
```

13.1.1 派生一个类

Webtown 俱乐部的一些成员曾经参加过当地的乒乓球锦标赛，需要这样一个类，它能包括成员在比赛中的比分。与其从零开始，不如从 TableTennisClass 类派生出一个类。首先将 RatedPlayer 类声明为从 TableTennisClass 类派生而来：

```
TableTennisClass class;
// RatedPlayer derives from the TableTennisPlayer base class
class RatedPlayer: public TableTennisPlayer
{
...
};
```

冒号指出 RatedPlayer 类的基类是 TableTennisplayer 类。上述特殊的声明头表明 TableTennisPlayer 是一个公有基类，这被称为公有派生。派生类对象包含基类对象。使用公有派生，基类的公有成员将成为派生类的公有成员；基类的私有部分也将成为派生类的一部分，但只能通过基类的公有和保护方法访问（稍后将介绍保护成员）。

上述代码完成了哪些工作呢？Ratedplayer 对象将具有以下特征：

- 派生类对象存储了基类的数据成员（派生类继承了基类的实现）。
- 派生类对象可以使用基类的方法（派生类继承了基类的接口）。

因此，RatedPlayer 对象可以存储运动员的姓名及其是否有球桌。另外，RatedPlayer 对象还可以使用 TableTennisPlayer 类的 Name()、hasTable()和 ResetTable()方法（参见图 13.1）。

需要在继承特性中添加什么呢？

- 派生类需要自己的构造函数。
- 派生类可以根据需要添加额外的数据成员和成员函数。

在这个例子中，派生类需要另一个数据成员来存储比分，还应包含检索比分的方法和重置比分的方法。因此，类声明与下面类似：

```
// simple derived class
class RatedPlayer: public TableTennisPlayer
{
private:
```

```
    unsigned int rating;     // add a data member
public:
    RatedPlayer (unsigned int r = 0, const char * fn = "none",
                 const char * ln = "none", bool ht = false);
    RatedPlayer (unsigned int r, const TableTennisPlayer & tp);
    unsigned int Rating(){ return rating; }       // add a method
    void ResetRating (unsigned int r) {rating = r; } // add a method
};
```

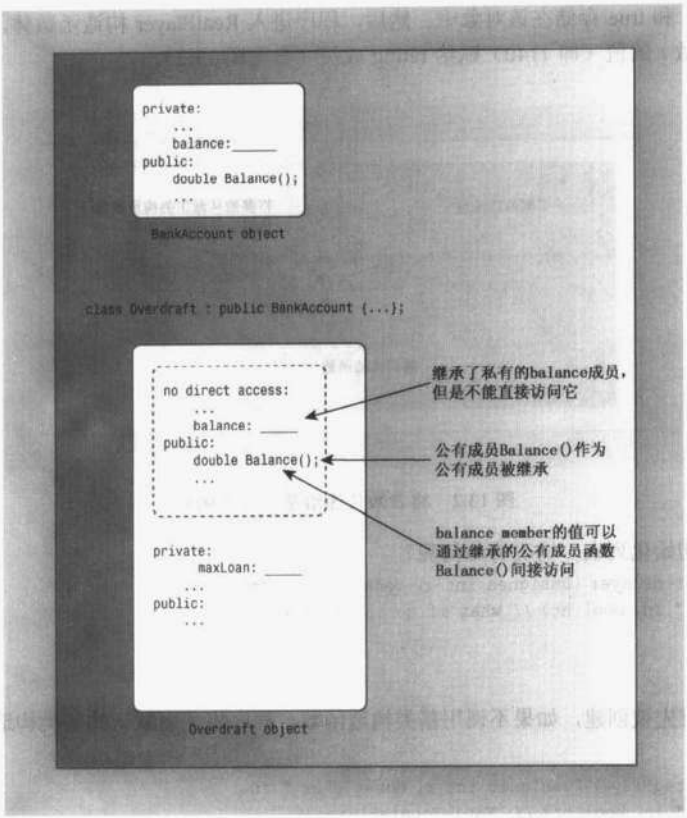


图 13.1　基类对象和派生类对象

构造函数必须给新成员（如果有的话）和继承的成员提供数据。在第一个 RatedPlayer 构造函数中，每个成员对应一个形参；而第二个 Ratedplayer 构造函数使用一个 TableTennisPlayer 参数，该参数包括 firstname、lastname 和 hasTable。

13.1.2　构造函数：访问权限的考虑

派生类不能直接访问基类的私有成员，而必须通过基类方法进行访问。例如，RatedPlayer 构造函数不能直接设置继承的成员（firstname、lastname 和 hasTable），而必须使用基类的公有方法来访问私有的基类成员。具体地说，派生类构造函数必须使用基类构造函数。

创建派生类对象时，程序首先创建基类对象。从概念上说，这意味着基类对象应当在程序进入派生类构造函数之前被创建。C++使用成员初始化列表句法来完成这种工作。例如，下面是第一个 RatedPlayer 构造函数的代码：

```
RatedPlayer::RatedPlayer (unsigned int r, const char * fn,
    const char * ln, bool ht): TableTennisPlayer (fn, ln, ht)
```

```
{
    rating = r;
}
```

其中 TableTennisPlayer（fn，ln，ht）是成员初始化列表。它是可执行的代码，它调用 TableTennisPlayer 构造函数。例如，假设程序包含以下声明：

```
RatedPlayer rplayer1 (1140, "Mallory", "Duck", true);
```

则 RealPlayer 构造函数将把实参“Mallory”、“Duck”和 true 赋给形参 fn、In 和 ht，然后将这些参数作为实参传递给 TableTennisPlayer 构造函数，后者将创建一个嵌套 TableTennisPlayer 对象，并将数据“Mallory”、“Duck”和 true 存储在该对象中。然后，程序进入 RealPlayer 构造函数体，完成 RealPlayer 对象的创建，并将参数 *r* 的值（即 1140）赋给 rating 成员（参见图 13.2）。

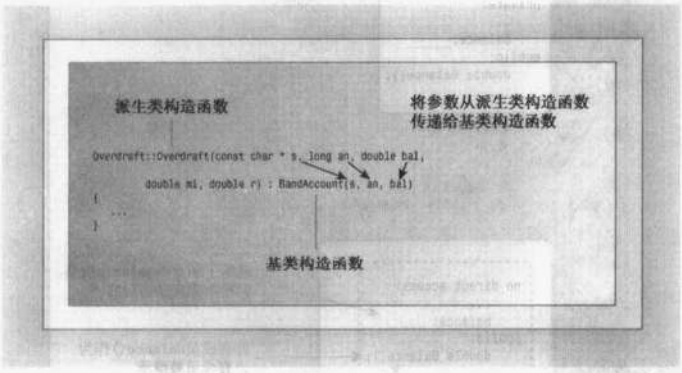


图 13.2 将参数传递给基类构造函数

如果省略成员初始化列表，情况将如何呢？

```
RatedPlayer::RatedPlayer (unsigned int r, const char * fn,
    const char * ln, bool ht) // what if no initializer list?
{
    rating = r;
}
```

基类对象必须首先被创建，如果不调用基类构造函数，程序将使用默认的基类构造函数，因此上述代码与下面等效：

```
RatedPlayer::RatedPlayer (unsigned int r, const char * fn,
    const char * ln, bool ht) //: TableTennisPlayer()
{
    rating = r;
}
```

除非要使用默认构造函数，否则应显式调用正确的基类构造函数。

下面来看第二个构造函数的代码：

```
RatedPlayer::RatedPlayer (unsigned int r, const TableTennisPlayer & tp)
  : TableTennisPlayer (tp)
{
    rating = r;
}
```

这里也将 TableTennisPlayer 的信息传递给了 TableTennisPlayer 构造函数：

```
TableTennisPlayer (tp)
```

因为 tp 的类型为 const TableTennisPlayer &，因此将调用基类的复制构造函数。基类没有定义复制构造函数，但第 12 章介绍过，如果需要使用复制构造函数，而又没有定义，编译器将自动生成一个。在这种情况下，执行成员复制的隐式复制构造函数是合适的，因为这个类没有使用动态内存分配。

如果愿意，也可以对派生类成员使用成员初始化列表句法。在这种情况下，应在列表中使用成员名，而不是类名。所以，第二个构造函数可以按照下述方式编写：

```
// alternative version
RatedPlayer::RatedPlayer (unsigned int r, const TableTennisPlayer & tp)
    : TableTennisPlayer (tp), rating (r)
{
}
```

有关派生类构造函数的要点如下：

- 基类对象首先被创建。
- 派生类构造函数应通过成员初始化列表将基类信息传递给基类构造函数。
- 派生类构造函数应初始化派生类新增的数据成员。

这个例子没有提供显式构造函数，因此将使用隐式构造函数。释放对象的顺序与创建对象的顺序相反，即首先执行派生类的析构函数，然后自动调用基类的析构函数。

记住： 创建派生类对象时，程序首先调用基类构造函数，然后再调用派生类构造函数。基类构造函数负责初始化继承的数据成员；派生类构造函数主要用于初始化新增的数据成员。派生类的构造函数总是调用一个基类构造函数。可以使用初始化器列表句法指明要使用的基类构造函数，否则将使用默认的基类构造函数。

派生类对象过期时，程序将首先调用派生类析构函数，然后再调用基类析构函数。

成员初始化列表

派生类构造函数可以使用初始化器列表机制将值传递给基类构造函数。请看下面的例子：

```
derived::derived (type1 x, type2 y): base (x, y) // initializer list
{
    ...
}
```

其中 derived 是派生类，base 是基类，*x* 和 *y* 是基类构造函数使用的变量。例如，如果派生类构造函数接收到参数 10 和 12，则这种机制将把 10 和 12 传递给被定义为接受这些类型的参数的基类构造函数。除虚拟基类外（参见第 14 章），类只能将值传递回相邻的基类，但后者可以使用相同的机制将信息传递给相邻的基类，依此类推。如果没有在成员初始化列表中提供基类构造函数，程序将使用默认的基类构造函数。成员初始化列表只能用于构造函数。

13.1.3　使用派生类

要使用派生类，程序必须要能够访问基类声明。程序清单 13.4 将这两种类的声明置于同一个头文件中。也可以将每个类放在独立的头文件中，但由于这两个类是相关的，所以把其类声明放在一起更合适。

程序清单 13.4　tabtenn1.h

```
// tabtenn1.h -- simple inheritance
#ifndef TABTENN1_H_
#define TABTENN1_H_
// simple base class
class TableTennisPlayer
{
private:
    enum {LIM = 20};
    char firstname[LIM];
    char lastname[LIM];
    bool hasTable;
public:
    TableTennisPlayer (const char * fn = "none",
                       const char * ln ="none", bool ht = false);
    void Name()const;
    bool HasTable()const { return hasTable; };
    void ResetTable (bool v) { hasTable = v; };
};
```

```
// simple derived class
class RatedPlayer: public TableTennisPlayer
{
private:
    unsigned int rating;
public:
    RatedPlayer (unsigned int r = 0, const char * fn = "none",
                 const char * ln = "none", bool ht = false);
    RatedPlayer (unsigned int r, const TableTennisPlayer & tp);
    unsigned int Rating() { return rating; }
    void ResetRating (unsigned int r) {rating = r; }
};

#endif
```

程序清单 13.5 是这两个类的方法定义。同样，也可以使用不同的文件，但将定义放在一起更为简便。

程序清单 13.5 tabtenn1.cpp

```
// tabtenn1.cpp -- base class methods and derived - class methods
#include "tabtenn1.h"
#include <iostream>
#include <cstring>

// TableTennisPlayer methods
TableTennisPlayer::TableTennisPlayer (const char * fn, const char * ln,
            bool ht)
{
    std::strncpy (firstname, fn, LIM - 1);
    firstname[LIM - 1] = '\0';
    std::strncpy (lastname, ln, LIM - 1);
    lastname[LIM - 1] = '\0';
    hasTable = ht;
}

void TableTennisPlayer::Name()const
{
    std::cout << lastname << ", " << firstname;
}

// RatedPlayer methods
RatedPlayer::RatedPlayer (unsigned int r, const char * fn,
    const char * ln, bool ht): TableTennisPlayer (fn, ln, ht)
{
   rating = r;
}

RatedPlayer::RatedPlayer (unsigned int r, const TableTennisPlayer & tp)
  : TableTennisPlayer (tp), rating (r)
{
}
```

程序清单 13.6 创建了 TableTennisPlayer 类和 RatedPlayer 类的对象。请注意这两个类对象是如何使用 TableTennisPlayer 类的 Name() 和 HasTable() 方法的。

程序清单 13.6 usett1.cpp

```
// usett1.cpp -- using base class and derived class
#include <iostream>
#include "tabtenn1.h"
```

```
int main (void)
{
    using std::cout;
    using std::endl;
    TableTennisPlayer player1 ("Tara", "Boomdea", false);
    RatedPlayer rplayer1 (1140, "Mallory", "Duck", true);
    rplayer1.Name();          // derived object uses base method
    if (rplayer1.HasTable())
        cout << ": has a table.\n";
    else
        cout << ": hasn't a table.\n";
    player1.Name();           // base object uses base method
    if (player1.HasTable())
        cout << ": has a table";
    else
        cout << ": hasn't a table.\n";
    cout << "Name: ";
    rplayer1.Name();
    cout << ": Rating: " << rplayer1.Rating()<< endl;
    RatedPlayer rplayer2 (1212, player1);
    cout << "Name: ";
    rplayer2.Name();
    cout << ": Rating: " << rplayer2.Rating()<< endl;

    return 0;
}
```

下面是程序清单 13.4～13.6 组成的程序的输出：

```
Duck, Mallory: has a table.
Boomdea, Tara: hasn't a table.
Name: Duck, Mallory: Rating: 1140
Name: Boomdea, Tara: Rating: 1212
```

13.2　派生类和基类之间的特殊关系

派生类与基类之间有一些特殊关系。其中之一是派生类对象可以使用基类的方法，条件是方法不是私有的：

```
RatedPlayer rplayer1 (1140, "Mallory", "Duck", true);
rplayer1.Name();  // derived object uses base method
```

另外两个重要的关系是：基类指针可以在不进行显式类型转换的情况下指向派生类对象；基类引用可以在不进行显式类型转换的情况下引用派生类对象：

```
RatedPlayer rplayer1 (1140, "Mallory", "Duck", true);
TableTennisPlayer & rt = rplayer;
TableTennisPlayer * pt = &rplayer;
rt.Name();  // invoke Name()with reference
pt->Name();  // invoke Name()with pointer
```

不过，基类指针或引用只能用于调用基类方法，因此，不能使用 rt 或 pt 来调用派生类的 ResetRanking 方法。

通常，C++要求引用和指针类型与赋给的类型匹配，但这一规则对继承来说是例外。不过，这种例外只是单向的，不可以将基类对象和地址赋给派生类引用和指针：

```
TableTennisPlary player ("Betsy", "Bloop", true);
RatedPlayer & rr = player;       // NOT ALLOWED
RatedPlayer * pr = player;       // NOT ALLOWED
```

上述规则是有道理的。例如，如果允许基类引用隐式地引用派生类对象，则可以使用基类引用为派生类对象调用基类的方法。因为派生类继承了基类的方法，所以这样做不会出现问题。如果可以将基类对象

赋给派生类引用，将发生什么情况呢？派生类引用能够为基对象调用派生类方法，这样做将出现问题。例如，将 RatedPlayer::Rating() 方法用于 TableTennisPlayer 对象是没有意义的，因为 TableTennisPlayer 对象没有 rating 成员。

如果基类引用和指针可以指向派生类对象，将出现一些很有趣的结果。其中之一是基类引用定义的函数或指针参数可用于基类对象或派生类对象。例如，在下面的函数中：

```
void Show(const TableTennisPlayer & rt)
{
    cout << "Name: ";
    rt.Name();
    cout << "\nTable: ";
    if (rt.HasTable())
        cout << "yes\n";
    else
        cout <<"no\n";
}
```

形参 rt 是一个基类引用，它可以指向基类对象或派生类对象，所以可以在 Show() 中使用 TableTennis 参数或 Ratedplayer 参数：

```
TableTennisPlayer player1("Tara", "Boomdea", false);
RatedPlayer rplayer1(1140, "Mallory", "Duck", true);
Show(player1);   // works with TableTennisPlayer argument
Show(rplayer1);  // works with RatedPlayer argument
```

对于形参为指向基类的指针的函数，也存在相似的关系。它可以使用基类对象的地址或派生类对象的地址作为实参：

```
void Wohs(const TableTennisPlayer * pt);  // function with pointer parameter
...
TableTennisPlayer player1("Tara", "Boomdea", false);
RatedPlayer rplayer1(1140, "Mallory", "Duck", true);
Wohs(&player1);   // works with TableTennisPlayer * argument
Wohs(&rplayer1);  // works with RatedPlayer * argument
```

引用兼容性属性也能够将基类对象初始化为派生类对象，尽管不那么直接。假设有这样的代码：

```
RatedPlayer olaf1(1840, "Olaf", "Loaf", true);
TableTennisPlayer olaf2(olaf1);
```

要初始化 olaf2，匹配的构造函数的原型如下：

```
TableTennisPlayer(const RatedPlayer &);      // doesn't exist
```

类定义中没有这样的构造函数，但存在隐式复制构造函数：

```
// implicit copy constructor
TableTennisPlayer(const TableTennisPlayer &);
```

形参是基类引用，因此它可以引用派生类。这样，将 olaf2 初始化为 olaf1 时，将要使用该构造函数，它复制 firstname、lastname 和 hasTable 成员。换句话来说，它将 olaf2 初始化为嵌套在 RatedPlayer 对象 olaf1 中的 TableTennisPlayer 对象。

同样，也可以将派生对象赋给基类对象：

```
RatedPlayer olaf1(1840, "Olaf", "Loaf", true);
TableTennisPlayer winner;
winner = olaf1; // assign derived to base object
```

在这种情况下，程序将使用隐式重载赋值操作符：

```
TableTennisPlayer & operator=(const TableTennisPlayer &) const;
```

基类引用指向的也是派生类对象，因此 olaf1 的基类部分被复制给 winner。

13.3 继承——is-a 关系

派生类和基类之间的特殊关系是基于 C++继承的底层模型的。实际上，C++有 3 种继承方式：公有继

承、保护继承和私有继承。公有继承是最常用的方式，它建立一种 *is-a* 关系，即派生类对象也是一个基类对象，可以对基类对象执行的任何操作，也可以对派生类对象执行。例如，假设有一个 Fruit 类，可以保存水果的重量和热量。因为香蕉是一种特殊的水果，所以可以从 Fruit 类派生出 Banana 类。新类将继承原始类的所有数据成员，因此，Banana 对象将包含表示香蕉重量和热量的成员。新的 Banana 类还添加了专门用于香蕉的成员，这些成员通常不用于水果，例如 Banana Institute Peel Index（香蕉机构果皮索引）。因为派生类可以添加特性，所以，将这种关系称为 *is-a-kind-of*（是一种）关系可能更准确，但是通常使用术语 *is-a*。

为阐明 *is-a* 关系，来看一些与该模型不符的例子。公有继承不建立 *has-a* 关系。例如，午餐可能包括了水果，但是通常，午餐并不是水果。所以，不能通过从 Fruit 类中派生出 Lunch 类来在午餐中添加水果。在午餐中加入水果的正确方法是将其作为一种 *has-a* 关系：午餐有水果。正如将在第 14 章介绍的，最容易的建模方式是，将 Fruit 对象作为 Lunch 类的数据成员（参见图 13.3）。

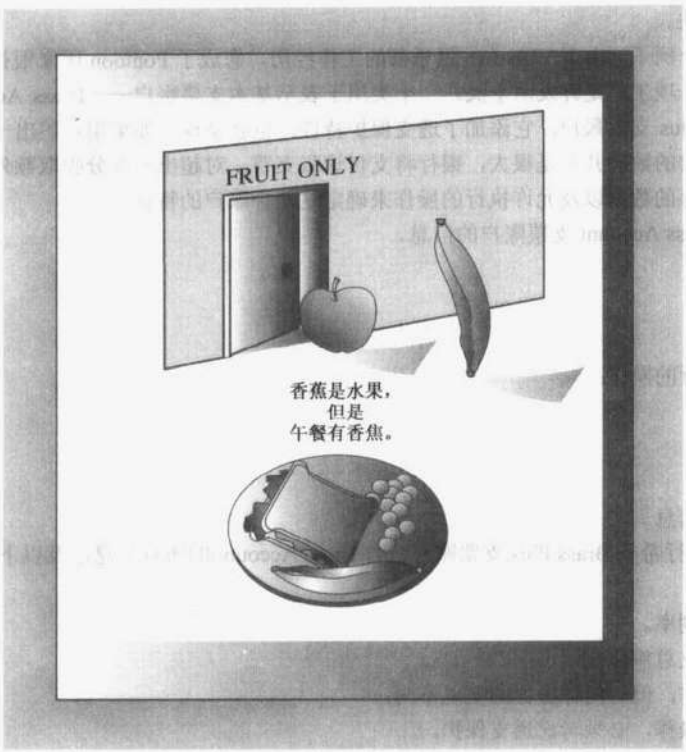


图 13.3　*is-a* 和 *has-a* 关系

公有继承不能建立 *is-like-a* 关系，也就是说，它不采用明喻。人们通常说律师就像鲨鱼，但律师并不是鲨鱼。例如，鲨鱼可以在水下生活。所以，不应从 Shark 类派生出 Lawyer 类。继承可以在基类的基础上添加属性，但不能删除基类的属性。在一些情况下，可以设计一个包含共有特征的类，然后以 *is-a* 或 *has-a* 关系，使用这个类来定义相关的类。

公有继承不建立 *is-implemented-as-a*（作为……来实现）关系。例如，可以使用数组来实现堆栈，但从 Array 类派生出 Stack 类是不合适的，因为堆栈不是数组。例如，数组索引不是堆栈的属性。另外，可以以其他方式实现堆栈，如使用链表。正确的方法是：通过让堆栈包含一个私有 Array 对象成员，来隐藏数组实现。

公有继承不建立 *uses-a* 关系。例如，计算机可以使用激光打印机，但是从 Computer 类派生出 Printer 类（或反过来）是没有意义的。不过，可以使用友元函数或类来处理 Printer 对象和 Computer 对象之间的通信。

在C++语言中，完全可以使用公有继承来建立 *has-a*、*is-implemented-as-a* 或 *uses-a* 关系；不过，这样做通常会导致编程方面的问题。因此，还是坚持使用 *is-a* 关系吧。

13.4 多态公有继承

RatedPlayer 继承范例很简单。派生类对象使用基类的方法，而未做任何修改。不过，可能会遇到这样的情况，即希望同一个方法在派生类和基类中的行为是不同的。换句话来说，方法的行为应取决于调用该方法的对象。这种较复杂的行为称为多态——具有多种形态，就是指同一个方法的行为将随上下文而异。有两种重要的机制可用于实现多态公有继承：

- 在派生类中重新定义基类的方法。
- 使用虚方法。

现在来看另一个例子。由于 Webtown 俱乐部的工作经历，您成了 Pontoon 国家银行的首席程序员。银行要求您完成的第一项工作是开发两个类。一个类用于表示基本支票账户——Brass Account，另一个类用于表示代表 Brass Plus 支票账户，它添加了透支保护特性。也就是说，如果用户签出一张超出其存款余额的支票——但是超出的数额并不是很大，银行将支付这张支票，对超出的部分收取额外的费用，并追加罚款。可以根据要保存的数据以及允许执行的操作来确定这两种账户的特征。

下面是用于 Brass Account 支票账户的信息。

- 客户姓名。
- 账号。
- 当前结余。

下面是可以执行的操作：

- 创建账户。
- 存款。
- 取款。
- 显示账户信息。

Pontoon 国家银行希望 Brass Plus 支票账户包含 Brass Account 的所有信息以及以下信息项：

- 透支上限。
- 透支贷款利率。
- 当前的透支总额。

不需要新增操作，但有两种操作的实现不同：

- 对于取款操作，必须考虑透支保护。
- 显示操作必须显示 Brass Plus 账户的其他信息。

我们将第一个类命名为 Brass，第二个类为 BrassPlus。应从 Brass 公有派生出 BrassPlus 吗？要回答这个问题，必须先回答另一个问题：BrassPlus 类是否满足 *is-a* 条件？当然满足。对于 Brass 对象是正确的事情，对于 BrassPlus 对象也是正确的。它们都将保存客户姓名、账号以及结余。使用这两个类都可以存款、取款和显示账户信息。请注意，*is-a* 关系通常是不可逆的。也就是说，水果不是香蕉；同样，Brass 对象不具备 BrassPlus 对象的所有功能。

13.4.1 开发 Brass 类和 BrassPlus 类

Brass Account 类的信息很简单，但是银行没有告诉您有关透支系统的细节。当您向友好的 Pontoon 国家银行代表询问时，他提供了如下信息：

- Brass Plus 账户限制了客户的透支款额。默认为 500 元，但有些客户的限额可能不同。
- 银行可以修改客户的透支限额。
- Brass Plus 账户对贷款收取利息。默认值是 10%，但是有些客户的利率可能不同。

- 银行可以修改客户的利率。
- 账户记录客户所欠银行的金额（透支数额加利息）。用户不能通过常规存款或从其他账户转账的方式偿付，而必须以现金的方式交给特定的银行工作人员。如果有必要，工作人员可以找到该客户。欠款偿还后，欠款金额将归零。

最后一种特性是银行出于做生意的考虑而采用的，这种方法有它有利的一面——使编程更简单。

上述列表表明，新的类需要构造函数，而且构造函数应提供账户信息，设置透支上限（默认为 500 元）和利率（默认为 10%）。另外，还应有重新设置透支限额、利率和当前欠款的方法。要添加到 Brass 类中的就是这些，这将在 BrassPlus 类声明中声明。

有关这两个类的信息声明，类声明应类似于程序清单 13.7。

程序清单 13.7　brass.h

```
// brass.h  -- bank account classes
#ifndef BRASS_H_
#define BRASS_H_

// Brass Account Class
class Brass
{
private:
    enum {MAX = 35};
    char fullName[MAX];
    long acctNum;
    double balance;
public:
    Brass (const char *s = "Nullbody", long an = -1,
                double bal = 0.0);
    void Deposit (double amt);
    virtual void Withdraw (double amt);
    double Balance()const;
    virtual void ViewAcct()const;
    virtual ~Brass(){}
};

//Brass Plus Account Class
class BrassPlus :  public Brass
{
private:
    double maxLoan;
    double rate;
    double owesBank;
public:
    BrassPlus (const char *s = "Nullbody", long an = -1,
          double bal = 0.0, double ml = 500,
          double r = 0.10);
    BrassPlus (const Brass & ba, double ml = 500, double r = 0.1);
    virtual void ViewAcct()const;
    virtual void Withdraw (double amt);
    void ResetMax (double m) { maxLoan = m; }
    void ResetRate (double r) { rate = r; }
    void ResetOwes(){ owesBank = 0; }
};

#endif
```

对于程序清单 13.7，需要说明的有下面几点：

- BrassPlus 类在 Brass 类的基础上添加了 3 个私有数据成员和 3 个公有成员函数。
- Brass 类和 BrassPlus 类都声明了 ViewAcct() 和 Withdraw() 方法，但 BrassPlus 对象和 Brass 对象

的这些方法的行为是不同的。

- Brass类在声明ViewAcct()和Withdraw()时使用了新关键字virtual。这些方法被称为虚方法(virtual method)。
- Brass类还声明了一个虚拟析构函数，虽然该析构函数不执行任何操作。

第一点没有什么新鲜的。RatedPlayer类在TableTennisPlayer类的基础上添加新数据成员和2个新方法的方式与此类似。

第二点介绍了声明如何指出方法在派生类的行为的不同。两个ViewAcct()原型表明将有2个独立的方法定义。基类版本的限定名为Brass::ViewAcct()，派生类版本的限定名为BrassPlus::ViewAcct()。程序将使用对象类型来确定使用哪个版本：

```
Brass dom("Dominic Banker", 11224, 4183.45);
BrassPlus dot("Dorothy Banker", 12118, 2592.00);
dom.ViewAcct();     // use Brass::ViewAcct()
dot.ViewAcct();     // use BrassPlus::ViewAcct()
```

同样，Withdraw()也有2个版本，一个供Brass对象使用，另一个供BrassPlus对象使用。对于在两个类中行为相同的方法（如Deposit()和Balance()），则只在基类中声明。

第三点（使用virtual）比前两点要复杂。如果方法是通过引用或指针而不是对象调用的，它将确定使用哪一种方法。如果没有使用关键字virtual，程序将根据引用类型或指针类型选择方法；如果使用了virtual，程序将根据引用或指针指向的对象的类型来选择方法。如果ViewAcct()不是虚拟的，则程序的行为如下：

```
// behavior with non-virtual ViewAcct()
// method chosen according to reference type
Brass dom("Dominic Banker", 11224, 4183.45);
BrassPlus dot("Dorothy Banker", 12118, 2592.00);
Brass & b1_ref = dom;
Brass & b2_ref = dot;
b1_ref.ViewAcct();     // use Brass::ViewAcct()
b2_ref.ViewAcct();     // use Brass::ViewAcct()
```

引用变量的类型为Brass，所以选择了Brass::ViewAccount()。使用Brass指针代替引用时，行为将与此类似。

如果ViewAcct()是虚拟的，则行为如下：

```
// behavior with virtual ViewAcct()
// method chosen according to object type
Brass dom("Dominic Banker", 11224, 4183.45);
BrassPlus dot("Dorothy Banker", 12118, 2592.00);
Brass & b1_ref = dom;
Brass & b2_ref = dot;
b1_ref.ViewAcct();     // use Brass::ViewAcct()
b2_ref.ViewAcct();     // use BrassPlus::ViewAcct()
```

这里两个引用的类型都是Brass，但b2_ref引用的是一个BrassPlus对象，所以使用的是BrassPlus::ViewAcct()。使用Brass指针代替引用时，行为将类似。

稍后您将看到，虚函数的这种行为非常方便。因此，经常在基类中将派生类会重新定义的方法声明为虚方法。方法在基类中被声明为虚拟的后，它在派生类中将自动成为虚方法。不过，在派生类声明中使用关键字virtual来指出哪些函数是虚函数也不失为一个好办法。

第四点是，基类声明了一个虚拟析构函数。这样做是为了确保释放派生对象时，按正确的顺序调用析构函数。本章后面将详细介绍这个问题。

记住：如果要在派生类中重新定义基类的方法，通常应将基类方法声明为虚拟的。这样，程序将根据对象类型而不是引用或指针的类型来选择方法版本。为基类声明一个虚拟析构函数也是一种惯例。

1. 类实现

接下来需要实现类，其中的部分工作已由头文件中的内联函数定义完成了。程序清单13.8列出了其他方法的定义。注意，关键字virtual只用于类声明的方法原型中，而没有用于程序清单13.8的方法定义中。

程序清单 13.8　brass.cpp

```
// brass.cpp -- bank account class methods
#include <iostream>
#include <cstring>
#include "brass.h"
using std::cout;
using std::ios_base;
using std::endl;
// Brass methods

Brass::Brass(const char *s, long an, double bal)
{
    std::strncpy(fullName, s, MAX - 1);
    fullName[MAX - 1] = '\0';
    acctNum = an;
    balance = bal;
}

void Brass::Deposit(double amt)
{
    if (amt < 0)
        cout << "Negative deposit not allowed; "
             << "deposit is cancelled.\n";
    else
        balance += amt;
}

void Brass::Withdraw(double amt)
{
    if (amt < 0)
        cout << "Withdrawal amount must be positive; "
             << "withdrawal canceled.\n";
    else if (amt <= balance)
        balance -= amt;
    else
        cout << "Withdrawal amount of $" << amt
             << " exceeds your balance.\n"
             << "Withdrawal canceled.\n";
}
double Brass::Balance()const
{
    return balance;
}

void Brass::ViewAcct()const
{
    // set up ###.## format
    ios_base::fmtflags initialState =
        cout.setf(ios_base::fixed, ios_base::floatfield);
    cout.setf(ios_base::showpoint);
    cout.precision(2);
    cout << "Client: " << fullName << endl;
    cout << "Account Number: " << acctNum << endl;
    cout << "Balance: $" << balance << endl;
    cout.setf(initialState); // restore original format
}

// BrassPlus Methods
BrassPlus::BrassPlus(const char *s, long an, double bal,
        double ml, double r) : Brass(s, an, bal)
```

```
{
    maxLoan = ml;
    owesBank = 0.0;
    rate = r;
}

BrassPlus::BrassPlus (const Brass & ba, double ml, double r)
            : Brass (ba)  // uses implicit copy constructor
{
    maxLoan = ml;
    owesBank = 0.0;
    rate = r;
}

// redefine how ViewAcct()works
void BrassPlus::ViewAcct()const
{
    // set up ###.## format
    ios_base::fmtflags initialState =
        cout.setf (ios_base::fixed, ios_base::floatfield);
    cout.setf (ios_base::showpoint);
    cout.precision (2);

    Brass::ViewAcct();   // display base portion
    cout << "Maximum loan: $" << maxLoan << endl;
    cout << "Owed to bank: $" << owesBank << endl;
    cout << "Loan Rate: " << 100 * rate << "%\n";
    cout.setf (initialState);
}

// redefine how Withdraw()works
void BrassPlus::Withdraw (double amt)
{
    // set up ###.## format
    ios_base::fmtflags initialState =
        cout.setf (ios_base::fixed, ios_base::floatfield);
    cout.setf (ios_base::showpoint);
    cout.precision (2);

    double bal = Balance();
    if (amt <= bal)
        Brass::Withdraw (amt);
    else if (amt <= bal + maxLoan - owesBank)
    {
        double advance = amt - bal;
        owesBank += advance * (1.0 + rate);
        cout << "Bank advance: $"<< advance << endl;
        cout << "Finance charge: $"<< advance * rate << endl;
        Deposit (advance);
        Brass::Withdraw (amt);
    }
    else
        cout << "Credit limit exceeded. Transaction cancelled.\n";
    cout.setf (initialState);
}
```

介绍程序清单 13.8 的具体细节（如一些方法的格式化处理）之前，先来看一下与继承直接相关的方面。记住，派生类并不能直接访问基类的私有数据，而必须使用基类的公有方法才能访问这些数据。访问的方式取决于方法。构造函数使用一种技术，而其他成员函数使用另一种技术。

派生类构造函数在初始化基类私有数据时，采用的是成员初始化列表句法。RatedPlayer 类构造函数和

BrassPlus 构造函数都使用这种技术。

```
BrassPlus::BrassPlus (const char *s, long an, double bal,
        double ml, double r)      : Brass (s, an, bal)
{
    maxLoan = ml;
    owesBank = 0.0;
    rate = r;
}

BrassPlus::BrassPlus (const Brass & ba, double ml, double r)
       : Brass (ba)  // uses implicit copy constructor
{
    maxLoan = ml;
    owesBank = 0.0;
    rate = r;
}
```

这几个构造函数都使用成员初始化列表句法，将基类信息传递给基类构造函数，然后使用构造函数体初始化 BrassPlus 类新增的数据项。

非构造函数不能使用成员初始化列表句法，但派生类方法可以调用公有的基类方法。例如，BrassPlus 版本的 ViewAcct() 核心内容如下（忽略了格式方面）：

```
// redefine how ViewAcct()works
void BrassPlus::ViewAcct()const
{
...
    Brass::ViewAcct();   // display base portion
    cout << "Maximum loan: $" << maxLoan << endl;
    cout << "Owed to bank: $" << owesBank << endl;
    cout << "Loan Rate: " << 100 * rate << "%\n";
...
}
```

换句话说，BrassPlus::ViewAcct() 显示新增的 BrassPlus 数据成员，并调用基类方法 Brass::ViewAcct() 来显示基类数据成员。在派生类方法中，标准的技术是使用作用域解析操作符来调用基类方法。

代码必须使用作用域解析操作符。假如这样编写代码：

```
// redefine how ViewAcct()works
void BrassPlus::ViewAcct()const
{
...
    ViewAcct();   // oops! recursive call
...
}
```

如果代码没有使用作用域解析操作符，编译器将认为 ViewAcct() 是 BrassPlus::ViewAcct()，这将创建一个不会终止的递归函数——这种情况可不好。

接下来看 BrassPlus::Withdraw() 方法。如果客户提取的金额超过了结余，该方法将安排贷款。它可以使用 Brass::Withdraw() 来访问 balance 成员，但如果取款金额超过了结余，Brass::Withdraw() 将发出一个错误消息。这种实现使用 Deposit() 方法进行放贷，然后在得到了足够的结余后调用 Brass::Withdraw，从而避免了错误消息：

```
// redefine how Withdraw()works
void BrassPlus::Withdraw (double amt)
{
...
    double bal = Balance();
    if (amt <= bal)
        Brass::Withdraw (amt);
    else if (amt <= bal + maxLoan - owesBank)
    {
```

```
        double advance = amt - bal;
        owesBank += advance * (1.0 + rate);
        cout << "Bank advance: $"<< advance << endl;
        cout << "Finance charge: $" << advance * rate << endl;
        Deposit (advance);
        Brass::Withdraw (amt);
    }
    else
        cout << "Credit limit exceeded. Transaction cancelled.\n";
...
}
```

该方法使用基类的 Balance() 函数来确定结余。因为派生类没有重新定义该方法，代码不必对 Balance() 使用作用域解析操作符。

ViewAcct() 方法使用格式化命令将浮点值的输出模式设置为定点，即包含两位小数。设置模式后，输出的模式将固定不变，因此该方法将格式模式重置为调用前的状态。因此，该方法使用下面的代码捕获最初的格式状态：

```
ios_base::fmtflags initialState =
    cout.setf (ios_base::fixed, ios_base::floatfield);
```

setf() 方法返回一个值，这个值表示调用函数之前的格式状态。新的 C++实现将这个值的类型定义为 ios_base::fmtflags，这条语句将状态保存在一个该类型的变量（initialState）中（早期的 C++版本使用的类型可能是 unsinged int）。ViewAcct() 结束时，将 initialState 作为参数传递给 setf()，以恢复原来的格式设置：

```
cout.setf (initialState);
```

2. 使用 Brass 和 BrassPlus 类

清单 13.9 使用了一个 Brass 对象和一个 BrassPlus 对象来测试类定义。

程序清单 13.9 usebrass1.cpp

```
// usebrass1.cpp -- testing bank account classes
// compile with brass.cpp
#include <iostream>
#include "brass.h"

int main()
{
    using std::cout;
    using std::endl;

    Brass Piggy ("Porcelot Pigg", 381299, 4000.00);
    BrassPlus Hoggy ("Horatio Hogg", 382288, 3000.00);
    Piggy.ViewAcct();
    cout << endl;
    Hoggy.ViewAcct();
    cout << endl;
    cout << "Depositing $1000 into the Hogg Account: \n";
    Hoggy.Deposit (1000.00);
    cout << "New balance: $" << Hoggy.Balance()<< endl;
    cout << "Withdrawing $4200 from the Pigg Account: \n";
    Piggy.Withdraw (4200.00);
    cout << "Pigg account balance: $" << Piggy.Balance()<< endl;
    cout << "Withdrawing $4200 from the Hogg Account: \n";
    Hoggy.Withdraw (4200.00);
    Hoggy.ViewAcct();

    return 0;
}
```

下面是程序清单 13.9 中程序的输出，请注意为何 Hogg 受透支限制，而 Pigg 没有：

```
Client: Porcelot Pigg
Account Number: 381299
Balance: $4000.00

Client: Horatio Hogg
Account Number: 382288
Balance: $3000.00
Maximum loan: $500.00
Owed to bank: $0.00
Loan Rate: 10.00%

Depositing $1000 into the Hogg Account:
New balance: $4000.00
Withdrawing $4200 from the Pigg Account:
Withdrawal amount of $4200.00 exceeds your balance.
Withdrawal canceled.
Pigg account balance: $4000.00
Withdrawing $4200 from the Hogg Account:
Bank advance: $200.00
Finance charge: $20.00
Client: Horatio Hogg
Account Number: 382288
Balance: $0.00
Maximum loan: $500.00
Owed to bank: $220.00
Loan Rate: 10.00%
```

3. 虚方法的行为

在程序清单 13.9 中，由于方法是通过对象（而不是指针或引用）调用的，所以没有使用虚方法特性。下面来看一个使用了虚方法的例子。假设要同时管理 Brass 和 BrassPlus 账户，如果能使用同一个数组来保存 Brsss 和 BrassPlus 对象，将很有帮助，但这是不可能的。数组中所有元素的类型必须相同，而 Brass 和 BrassPlus 是不同的类型。不过，可以创建指向 Brass 的指针数组。这样，每个元素的类型都相同，但由于使用的是公有继承模型，因此 Brass 指针既可以指向 Brass 对象，也可以指向 BrassPlus 对象。因此，可以使用一个数组来表示多种类型的对象。这就是多态性，程序清单 13.10 是一个简单的例子。

程序清单 13.10　usebrass2.cpp

```
// usebrass2.cpp -- polymorphic example
// compile with brass.cpp
#include <iostream>
#include "brass.h"
const int CLIENTS = 4;
const int LEN = 40;
int main()
{
    using std::cin;
    using std::cout;
    using std::endl;
    Brass * p_clients[CLIENTS];

    int i;
    for (i = 0; i < CLIENTS; i++)
    {
     char temp[LEN];
     long tempnum;
     double tempbal;
     char kind;
     cout << "Enter client's name: ";
     cin.getline(temp, LEN);
```

```
        cout << "Enter client's account number: ";
        cin >> tempnum;
        cout << "Enter opening balance: $";
        cin >> tempbal;
        cout << "Enter 1 for Brass Account or "
             << "2 for BrassPlus Account: ";
        while (cin >> kind && (kind != '1' && kind != '2'))
          cout <<"Enter either 1 or 2: ";
        if (kind == '1')
          p_clients[i] = new Brass (temp, tempnum, tempbal);
        else
        {
          double tmax, trate;
          cout << "Enter the overdraft limit: $";
          cin >> tmax;
          cout << "Enter the interest rate "
               << "as a decimal fraction: ";
          cin >> trate;
          p_clients[i] = new BrassPlus (temp, tempnum, tempbal,
                                        tmax, trate);
        }
        while (cin.get() != '\n')
          continue;
      }
      cout << endl;
      for (i = 0; i < CLIENTS; i++)
      {
        p_clients[i]->ViewAcct();
        cout << endl;
      }

      for (i = 0; i < CLIENTS; i++)
      {
        delete p_clients[i];  // free memory
      }
      cout << "Done.\n";

      return 0;
}
```

程序 13.10 中程序根据用户的输入来确定要添加的账户类型，然后使用 new 创建并初始化相应类型的对象。下面是该程序的运行情况：

```
Enter client's name: Harry Fishsong
Enter client's account number: 112233
Enter opening balance: $ 1500
Enter 1 for Brass Account or 2 for BrassPlus Account: 1
Enter client's name: Dinah Otternose
Enter client's account number: 121213
Enter opening balance: $ 1800
Enter 1 for Brass Account or 2 for BrassPlus Account: 2
Enter the overdraft limit: $350
Enter the interest rate as a decimal fraction: 0.12
Enter client's name: Brenda Birdherd
Enter client's account number: 212118
Enter opening balance: $ 5200
Enter 1 for Brass Account or 2 for BrassPlus Account: 2
Enter the overdraft limit: $ 800
Enter the interest rate as a decimal fraction: 0.10
Enter client's name: Tim Turtletop
Enter client's account number: 233255
```

```
Enter opening balance: $688
Enter 1 for Brass Account or 2 for BrassPlus Account: 1

Client: Harry Fishsong
Account Number: 112233
Balance: $1500.00

Client: Dinah Otternoe
Account Number: 121213
Balance: $1800.00
Maximum loan: $350.00
Owed to bank: $0.00
Loan Rate: 12.00%

Client: Brenda Birdherd
Account Number: 212118
Balance: $5200.00
Maximum loan: $800.00
Owed to bank: $0.00
Loan Rate: 10.00%

Client: Tim Turtletop
Account Number: 233255
Balance: $688.00

Done.
```

多态特性是由下述代码提供的：

```
for (i = 0; i < CLIENTS; i++)
{
   p_clients[i]->ViewAcct();
   cout << endl;
}
```

如果数组成员指向的是 Brass 对象，则调用 Brass::ViewAcct()；如果指向的是 BrassPlus 对象，则调用 BrassPlus::ViewAcct()。如果 Brass::ViewAcct() 被声明为虚拟的，则在任何情况下都将调用 Brass::ViewAcct()。

4. 为何需要虚拟析构函数

在程序清单 13.10 中，使用 delete 释放由 new 分配的对象的代码说明了为何基类应包含一个虚拟析构函数，虽然有时好像并不需要析构函数。如果析构函数不是虚拟的，则将只调用对应于指针类型的析构函数。对于程序清单 13.10，这意味着只有 Brass 的析构函数被调用，即使指针指向的是一个 BrassPlus 对象。如果析构函数是虚拟的，将调用相应对象类型的析构函数。因此，如果指针指向的是 BrassPlus 对象，将调用 BrassPlus 的析构函数，然后自动调用基类的析构函数。因此，使用虚拟析构函数可以确保正确的析构函数序列被调用。对于程序清单 13.10，这种正确的行为并不是很重要，因为析构函数没有执行任何操作。但是，如果 BrassPlus 包含一个执行某些操作的析构函数，则 Brass 必须有一个虚拟析构函数，即使该析构函数不执行任何操作。

13.4.2　静态联编和动态联编

程序调用函数时，将使用哪个可执行代码块呢？编译器负责回答这个问题。将源代码中的函数调用解释为执行特定的函数代码块被称为函数名联编（binding）。在 C 语言中，这非常简单，因为每个函数名都对应一个不同的函数。在 C++中，由于函数重载的缘故，这项任务更复杂。编译器必须查看函数参数以及函数名才能确定使用哪个函数。然而，C/C++编译器可以在编译过程完成这种联编。在编译过程中进行联编被称为静态联编（static binding），又称为早期联编（early binding）。不过，虚函数使这项工作变得更困难。正如在程序清单 13.10 所示的那样，使用哪一个函数是不能在编译时确定的，因为编译器不知道用户

将选择哪种类型的对象。所以，编译器必须生成能够在程序运行时选择正确的虚方法的代码，这被称为动态联编（dynamic binding），又称为晚期联编（late binding）。

知道虚方法的运行情况后，下面深入地探讨这一过程，首先介绍C++如何处理指针和引用类型的兼容性。

13.4.3 指针和引用类型兼容性

在C++中，动态联编与指针和引用调用的方法相关，从某种程度上说，这是由继承控制的。公有继承建立 *is-a* 关系的一种方法是如何处理指向对象的指针和引用。通常，C++不允许将一种类型的地址赋给另一种类型的指针，也不允许一种类型的引用指向另一种类型：

```
double x = 2.5;
int * pi = &x;   // invalid assignment, mismatched pointer types
long & rl = x;   // invalid assignment, mismatched reference type
```

不过，正如您看到的，指向基类的引用或指针可以引用派生类对象，而不必进行显式类型转换。例如，下面的初始化是允许的：

```
BrassPlus dilly ("Annie Dill", 493222, 2000);
Brass * pb = &dilly;   // ok
Brass & rb = dilly;    // ok
```

将派生类引用或指针转换为基类引用或指针被称为向上强制转换（upcasting），这使公有继承不需要进行显式类型转换。该规则是 *is-a* 关系的一部分。BrassPlus 对象都是 Brass 对象，因为它继承了 Brass 对象所有的数据成员和成员函数。所以，可以对 Brass 对象执行的任何操作，都适用于 BrassPlus 对象。因此，为处理 Brass 引用而设计的函数可以对 BrassPlus 对象执行同样的操作，而不必担心会导致任何问题。将指向对象的指针作为函数参数时，也是如此。向上强制转换是可传递的，也就是说，如果从 BrassPlus 派生出 BrassPlusPlus 类，则 Brass 指针或引用可以引用 Brass 对象、BrassPlus 对象或 BrassPlusPlus 对象。

相反的过程——将基类指针或引用转换为派生类指针或引用——称为向下强制转换（downcasting）。如果不使用显式类型转换，则向下强制转换是不允许的。原因是 *is-a* 关系通常是不可逆的。派生类可以新增数据成员，因此使用这些数据成员的类成员函数不能应用于基类。例如，假设从 Employee 类派生出 Singer 类，并添加了表示歌手音域的数据成员和用于报告音域的值的成员函数 range()，则将 range() 方法应用于 Employee 对象是没有意义的。但如果允许隐式向下强制转换，则可能无意间将指向 Singer 的指针设置为一个 Employee 对象的地址，并使用该指针来调用 range() 方法（参见图 13.4）。

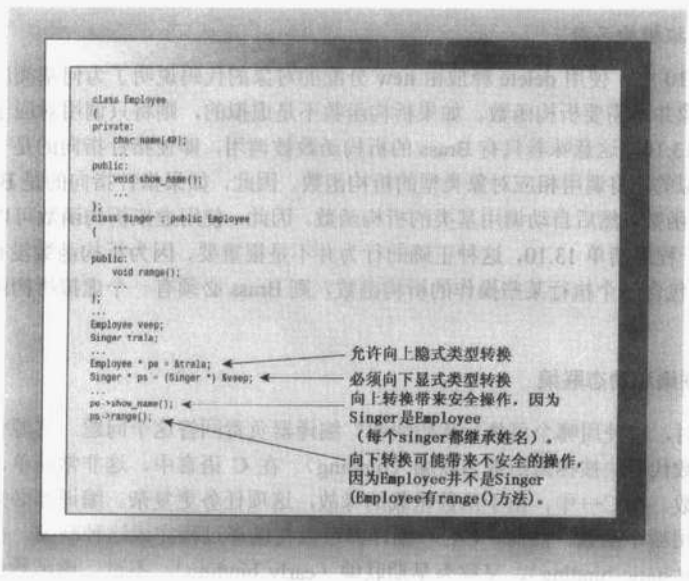


图 13.4 向上强制转换和向下强制转换

隐式向上强制转换使基类指针或引用可以指向基类对象或派生类对象，因此需要动态联编。C++使用虚成员函数来满足这种需求。

13.4.4　虚拟成员函数和动态联编

我们来回顾一下使用引用或指针调用方法的过程。请看下面的代码：

```
BrassPlus ophelia;     // derived-class object
Brass * bp;            // base-class pointer
bp = &ophelia;         // Brass pointer to BrassPlus object
bp->ViewAcct();        // which version?
```

正如前面介绍的，如果在基类中没有将 ViewAcct() 声明为虚拟的，则 bp->ViewAcct() 将根据指针类型（Brass *），而调用 Brass::ViewAcct()。指针类型在进行编译时是已知的，因此编译器在进行编译时，可以将 ViewAcct() 关联到 Brass::ViewAcct()。简而言之，编译器对非虚方法使用静态联编。

但是，如果在基类中将 ViewAcct() 声明为虚拟的，则 bp->ViewAcct() 根据对象类型（BrassPlus）调用 BrassPlus::ViewAcct()。在这个例子中，对象类型为 BrassPlus，但通常（如程序清单 13.10 所示）只有在运行程序时才能确定对象的类型。所以编译器生成的代码将在程序执行时，根据对象类型将 ViewAcct() 关联到 Brass::ViewAcct() 或 BrassPlus::ViewAcct()。简而言之，编译器对虚方法使用动态联编。

在大多数情况下，动态联编很好，因为它让程序能够选择为特定类型设计的方法。因此，您可能会问：

- 为什么有两种类型的联编？
- 既然动态联编如此之好，为什么不将它设置成默认的？
- 动态联编是如何工作的？

下面来看一看这些问题的答案。

1. 为什么有两种类型的联编以及为什么默认为静态联编

如果动态联编让您能够重新定义类方法，而静态联编在这方面很差，为何不摒弃静态联编呢？原因有两个——效率和概念模型。

首先来看效率。为使程序能够在运行阶段进行决策，必须采取一些方法来跟踪基类指针或引用指向的对象类型，这增加了额外的处理开销（稍后将介绍一种动态联编方法）。例如，如果类不会用作基类，则不需要动态联编。同样，如果派生类（如 RatedPlayer）不重新定义基类的任何方法，也不需要使用动态联编。在这些情况下，使用静态联编更合理，效率也更高。由于静态联编的效率更高，因此被设置为 C++的默认选择。Strousstrup 说，C++的指导原则之一是，不要为不使用的特性付出代价（内存或者处理时间）。仅当程序设计确实需要虚函数时，才使用它们。

接下来看概念模型。在设计类时，可能包含一些不在派生类重新定义的成员函数。例如，Brass::Balance() 函数返回账户结余，不应该重新定义。不将该函数设置为虚函数，有两方面的好处：首先效率更高；其次，指出不要重新定义该函数。这表明，仅将那些预期将被重新定义的方法声明为虚拟的。

提示：如果要在派生类中重新定义基类的方法，则将它设置为虚方法；否则，设置为非虚方法。

当然，设计类时，方法属于哪种情况有时并不那么明显。与现实世界中的许多方面一样，类设计并不是一个线性过程。

2. 虚函数的工作原理

C++规定了虚函数的行为，但将实现方法留给了编译器作者。不需要知道实现方法就可以使用虚函数，但了解虚函数的工作原理有助于更好地理解概念，因此，这里对其进行介绍。

通常，编译器处理虚函数的方法是：给每个对象添加一个隐藏成员。隐藏成员中保存了一个指向函数地址数组的指针。这种数组称为虚函数表（virtual function table，vtbl）。虚函数表中存储了为类对象进行声明的虚函数的地址。例如，基类对象包含一个指针，该指针指向基类中所有虚函数的地址表。派生类对象将包含一个指向独立地址表的指针。如果派生类提供了虚函数的新定义，该虚函数表将保存新函数的地址；如果派生类没有重新定义虚函数，该 vtbl 将保存函数原始版本的地址。如果派生类定义了新的虚函数，则

该函数的地址也将被添加到 vtbl 中（参见图 13.5）。注意，无论类中包含的虚函数是 1 个还是 10 个，都只需要在对象中添加 1 个地址成员，只是表的大小不同而已。

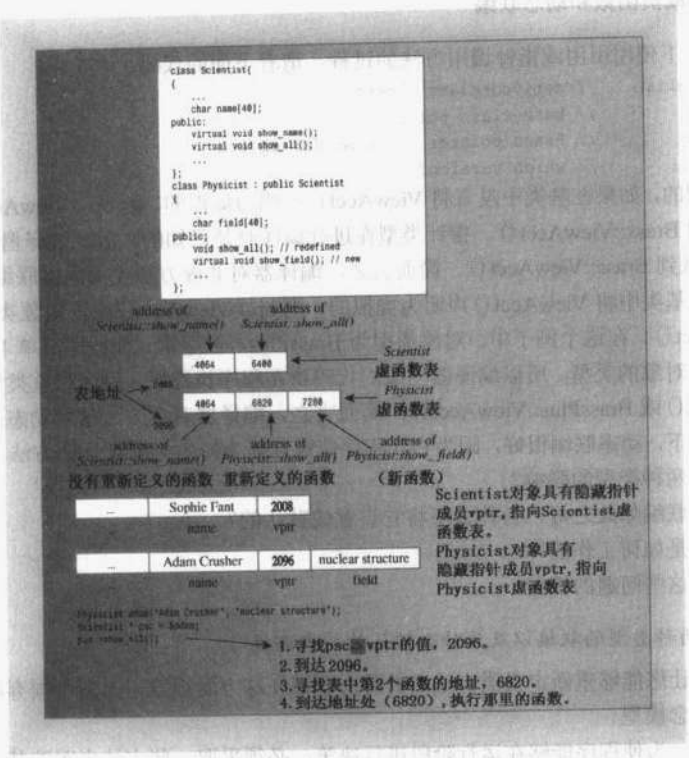


图 13.5 虚函数机制

调用虚函数时，程序将查看存储在对象中的 vtbl 地址，然后转向相应的函数地址表。如果使用类声明中定义的第一个虚函数，则程序将使用数组中的第一个函数地址，并执行具有该地址的函数。如果使用类声明中的第三个虚函数，程序将使用地址为数组中第三个元素的函数。

简而言之，使用虚函数时，在内存和执行速度方面有一定的成本，包括：

- 每个对象都将增大，增大量为存储地址的空间。
- 对每个类，编译器都创建一个虚函数地址表（数组）。
- 每个函数调用都需要执行一步额外的操作，即到表中查找地址。

虽然非虚函数的效率比虚函数稍高，但不具备动态联编功能。

13.4.5 有关虚函数注意事项

我们已经讨论了虚函数的一些要点：

- 在基类方法的声明中使用关键字 virtual 可使该方法在基类以及所有的派生类（包括从派生类派生出来的类）中是虚拟的。
- 如果使用指向对象的引用或指针来调用虚方法，程序将使用为对象类型定义的方法，而不使用为引用或指针类型定义的方法。这称为动态联编或晚期联编。这种行为非常重要，因为这样基类指针或引用可以指向派生类对象。
- 如果定义的类将被用作基类，则应将那些要在派生类中重新定义的类方法声明为虚拟的。

对于虚方法，还需要了解其他一些知识，其中有的已经介绍过。下面我们来看一下这些内容。

1. 构造函数

构造函数不能是虚函数。创建派生类对象时，将调用派生类的构造函数，而不是基类的构造函数，然后，派生类的构造函数将使用基类的一个构造函数，这种顺序不同于继承机制。因此，派生类不继承基类的构造函数，所以将类构造函数声明为虚拟的没有什么意义。

2. 析构函数

析构函数应当是虚函数，除非类不用做基类。例如，假设 Employee 是基类，Singer 是派生类，并添加一个 char *成员，该成员指向由 new 分配的内存。当 Singer 对象过期时，必须调用～Singer() 析构函数来释放内存。

请看下面的代码：

```
Employee * pe = new Singer; // legal because Employee is base for Singer
...
delete pe;                  // ~Employee()or ~Singer()?
```

如果使用默认的静态联编，delete 语句将调用～Employee() 析构函数。这将释放由 Singer 对象中的 Employee 部分指向的内存，但不会释放新的类成员指向的内存。但如果析构函数是虚拟的，则上述代码将先调用～Singer 析构函数释放由 Singer 组件指向的内存，然后，调用～Employee() 析构函数来释放由 Employee 组件指向的内存。

这意味着，即使基类不需要显式析构函数提供服务，也不应依赖于默认构造函数，而应提供虚拟析构函数，即使它不执行任何操作：

```
virtual ~BaseClass(){ }
```

顺便说一句，给类定义一个虚拟析构函数并非错误，即使这个类不用做基类；这只是一个效率方面的问题。

提示：通常应给基类提供一个虚拟析构函数，即使它并不需要析构函数。

3. 友元

友元不能是虚函数，因为友元不是类成员，而只有成员才能是虚函数。如果由于这个原因引起了设计问题，可以通过让友元函数使用虚拟成员函数来解决。

4. 没有重新定义

如果派生类没有重新定义函数，将使用该函数的基类版本。如果派生类位于派生链中，则将使用最新的虚函数版本，例外的情况是基类版本是隐藏的（稍后将介绍）。

5. 重新定义隐藏方法

假设创建了如下所示的代码：

```
class Dwelling
{
public:
     virtual void showperks (int a) const;
...
};
class Hovel: public Dwelling
{
{
public:
     virtual void showperks()const;
...
};
```

这将导致问题，可能会出现类似于下面这样的编译器警告：

```
Warning: Hovel::showperks (void) hides Dwelling::showperks (int)
```

也可能不会出现警告。但不管结果怎样，代码将具有如下含义：

```
Hovel trump;
trump.showperks();      // valid
trump.showperks(5);     // invalid
```

新定义将 showperks()定义为一个不接受任何参数的函数。重新定义不会生成函数的两个重载版本，而是隐藏了接受一个 int 参数的基类版本。简而言之，重新定义继承的方法并不是重载。如果在派生类中重新定义函数，将不是使用相同的函数特征标覆盖基类声明，而是隐藏同名的基类方法，不管参数特征标如何。

这引出了两条经验规则：第一，如果重新定义继承的方法，应确保与原来的原型完全相同，但如果返回类型是基类引用或指针，则可以修改为指向派生类的引用或指针（这种例外是新出现的）。这种特性被称为返回类型协变（covariance of return type），因为允许返回类型随类类型的变化而变化：

```
class Dwelling
{
public:
// a base method
   virtual Dwelling & build(int n);
   ...
};
class Hovel : public Dwelling
{
public:
// a derived method with a covariant return type
   virtual Hovel & build(int n);  // same function signature
   ...
};
```

注意，这种例外只适用于返回值，而不适用于参数。

第二，如果基类声明被重载了，则应在派生类中重新定义所有的基类版本。

```
class Dwelling
{
public:
// three overloaded showperks()
     virtual void showperks(int a) const;
     virtual void showperks(double x) const;
     virtual void showperks()const;
     ...
};
class Hovel: public Dwelling
{
public:
// three redefined showperks()
     virtual void showperks(int a) const;
     virtual void showperks(double x) const;
     virtual void showperks()const;
     ...
};
```

如果只重新定义一个版本，则另外两个版本将被隐藏，派生类对象将无法使用它们。注意，如果不需要修改，则新定义可只调用基类版本。

13.5 访问控制：protected

到目前为止，本书的类范例已经使用了关键字 public 和 private 来控制对类成员的访问。还存在另一个访问类别，这种类别用关键字 protected 表示。关键字 protected 与 private 相似，在类外只能用公有类成员来访问 protected 部分中的类成员。private 和 protected 之间的区别只有在基类派生的类中才会表现出来。派

生类的成员可以直接访问基类的保护成员，但不能直接访问基类的私有成员。因此，对于外部世界来说，保护成员的行为与私有成员相似；但对于派生类来说，保护成员的行为与公有成员相似。

例如，假如 Brass 类将 balance 成员声明为保护的：

```
class Brass
{
protected:
   double balance;
...
};
```

在这种情况下，BrassPlus 类可以直接访问 balance，而不需要使用 Brass 方法。例如，可以这样编写 BrassPlus::Withdraw() 的核心：

```
void BrassPlus::Withdraw (double amt)
{
   if (amt < 0)
      cout << "Withdrawal amout must be positive;"

         << "withdrawal canceled.\n";
   else if (amt <= balance)      // access balance directly
      balance -= amt;
   else if (amt <= balance + maxLoan - owesBank)
   {
      double advance = amt - balance;
      owesBank += advance * (1.0 + rate);
      cout << "Bank advance: $" << advance << endl;
      cout << "Finance charge: $" << advance * rate << endl;
      Deposit (advance);
      balance -= amt;
   }
   else
      cout << "Credit limit exceeded. Transaction cancelled.\n";
}
```

使用保护数据成员可以简化代码的编写工作，但存在设计缺陷。例如，继续以 BrassPlus 为例，如果 balance 是受保护的，则可以按下面的方式编写代码：

```
void BrassPlus::Reset (double amt)
{
   balance = amt;
}
```

Brass 类被设计成只能通过 Deposit() 和 Withdraw() 才能修改 balance。但对于 BrassPlus 对象，Reset() 方法将忽略 Withdraw() 中的保护措施，实际上使 balance 成为公有变量。

警告：最好对类数据成员采用私有访问控制，不要使用保护访问控制；同时通过基类方法使派生类能够访问基类数据

然而，对于成员函数来说，保护访问控制很有用，它让派生类能够访问公众不能使用的内部函数。

单设计模式

经常会遇到这样的情况，即一个通用模式可以解决许多问题。在人类交往中也是这样，有这样一个处方“深吸一口气，数到 10 后再呼气” 同样，在编程过程中，研究软件设计问题时，也会出现通用模式。在软件中，设计模式就像烹饪方式，不论成分组合是否相同，都可以使用同一种烹饪方法。可以使用设计模式创造一个巧妙而一致的方法，来解决众多的问题。例如，希望有且只有一个类的实例返回给调用程序时，就可以使用单元素模式（Singleton pattern）。这种类的声明如下：

```
class TheOnlyInstance
{
   public:
   static TheOnlyInstance* GetTheOnlyInstance ();
```

```
    // other methods
    protected:
    TheOnlyInstance () {}
    private:
    // private data
};
```

通过将 TheOnlyInstance 构造函数声明为 protected，并省略公有构造函数，可以防止局部实例被创建：

```
int main ()
{
    TheOnlyInstance noCanDo;  // not allowed
```

只能通过公有静态方法 GetTheOnlyInstance 来访问类。该方法被调用时，将返回类 TheOnlyInstance 的实例。

```
TheOnlyInstance* TheOnlyInstance:: GetTheOnlyInstance ()
{
    static TheOnlyInstance objTheOnlyInstance;
    return &objTheOnlyInstance;
}
```

GetTheOnlyInstance 方法仅在第一次被调用时，创建 TheOnlyInstance 类的一个实例。以这种方式构造的静态对象一直有效，直到程序终止，此时这种静态对象将自动被释放。要检索指向这个类的惟一一个实例的指针，只需要调用静态方法 GetTheOlyInstance，该方法返回单对象的地址：

```
TheOnlyInstance* pTheOnlyInstance = TheOnlyInstance: : GetTheOnlyInstance () ;
```

因为静态变量在函数调用结束后仍保存在内存中，所以以后再调用 GetTheOnlyInstance 时，将返回同一个静态对象的地址。

13.6 抽象基类

至此，介绍了简单继承和较复杂的多态继承。接下来更为复杂的是抽象基类（abstract base class，ABC）。我们来看一些可使用 ABC 的编程情况。

有时候，使用 *is-a* 规则并不像看上去的那样简单。例如，假设您正在开发一个图形程序，该程序会显示圆和椭圆等。圆是椭圆的一个特殊情况——长轴和短轴等长的椭圆。因此，所有的圆都是椭圆，可以从 Ellipse 类派生出 Circle 类。但涉及到细节时，将发现许多问题。

首先考虑 Ellipse 类包含的内容。数据成员可以包括椭圆中心的坐标、长半轴（长轴的一半）、短半轴（短轴的一半）以及方向角（水平坐标轴与长轴之间的角度）。另外，还可以包括一些移动椭圆、返回椭圆面积、旋转椭圆以及缩放长半轴和短半轴的方法：

```
class Ellipse
{
private:
    double x;      // x-coordinate of the ellipse's center
    double y;      // y-coordinate of the ellipse's center
    double a;      // semimajor axis
    double b;      // semiminor axis
    double angle;  // orientation angle in degrees
    ...
public:
    ...
    void Move (int nx, ny) { x = nx; y = ny; }
    virtual double Area()const { return 3.14159 * a * b; }
    virtual void Rotate (double nang) { angle += nang; }
    virtual void Scale (double sa, double sb) { a *= sa; b *= sb; }
    ...
};
```

现在假设从 Ellipse 类派生出一个 Circle 类：

```
class Circle :  public Ellipse
{
   ...
};
```

虽然圆是一种椭圆，但是这种派生是笨拙的。例如，圆只需要一个值（半径）就可以描述大小和形状，并不需要有长半轴（a）和短半轴（b）。Circle 构造函数可以通过将同一个值赋给成员 a 和 b 来照顾这种情况，但将导致信息冗余。angle 参数和 Rotate()方法对圆来说没有实际意义；而 Scale()方法（顾名思义）会将两个轴作不同的缩放，将圆变成椭圆。可以使用一些技巧来修正这些问题，例如在 Circle 类中的私有部分包含重新定义的 Rotate()方法，使 Rotate()不能以公有的方式被用于圆。但总的来说，不使用继承，直接定义 Circle 类更为简单：

```
class Circle     // no inheritance
{
private:
   double x;     // x-coordinate of the circle's center
   double y;     // y-coordinate of the circle's center
   double r;     // radius
   ...
public:
   ...
   void Move (int nx, ny) { x = nx; y = ny; }
   double Area()const { return 3.14159 * r * r; }
   void Scale (double sr)  { r *= sr; }
   ...
};
```

现在，类只包含所需的成员。但这种解决方法的效率也不高。Circle 和 Ellipse 类有很多共同点，将它们分别定义则忽略了这一事实。

还有一种解决方法，即从 Ellipse 和 Circle 类中抽象出它们的共性，将这些特性放到一个 ABC 中。然后从该 ABC 派生出 Circle 和 Ellipse 类。这样，便可以使用基类指针数组同时管理 Circle 和 Ellipse 对象，即可以使用多态方法。在这个例子中，这两个类的共同点是中心坐标、Move()方法（对于这两个类是相同的）和 Area()方法（对于这两个类来说，是不同的）。确实，甚至不能在 ABC 中实现 Area()方法，因为它没有包含必要的数据成员。C++通过使用纯虚函数（pure virtual function）提供未实现的函数。纯虚函数声明的结尾处为=0，参见 Area()方法：

```
class BaseEllipse  // abstract base class
{
private:
   double x;       // x-coordinate of center
   double y;       // y-coordinate of center
   ...
public:
   BaseEllipse (double x0 = 0, double y0 = 0): x (x0), y (y0) {}
   virtual ~BaseEllipse(){}
   void Move (int nx, ny) { x = nx; y = ny; }
   virtual double Area()const = 0; // a pure virtual function
    ...
}
```

当类声明中包含纯虚函数时，则不能创建该类的对象。这里的理念是，包含纯虚函数的类只用作基类。要成为真正的 ABC，必须至少包含一个纯虚函数。原型中的=0 使虚函数成为纯虚函数。这里的方法 Area()没有定义，但 C++甚至允许纯虚函数有定义。例如，也许所有的基类方法都与 Move()一样，可以在基类中进行定义，但您仍需要将这个类声明为抽象的。在这种情况下，可以将原型声明为虚拟的：

```
void Move(int nx, ny) = 0;
```

这将使基类成为抽象的，但您仍可以在实现文件中提供方法的定义：

```
void BaseEllipse::Move(int nx, ny) { x = nx; y = ny; }
```

总之，在原型中使用=0 指出类是一个抽象基类，在类中可以不定义该函数。

现在，可以从 BaseEllipse 类派生出 Ellips 类和 Circle 类，添加所需的成员来完成每个类。需要注意的一点是，Circle 类总是表示圆，而 Ellipse 类总是表示椭圆——也可以是圆。不过，Ellipse 类圆可被重新缩放为非圆，而 Ciecle 类圆必须始终为圆。

使用这些类的程序将能够创建 Ellipse 对象和 Circle 对象，但是不能创建 BaseEllipse 对象。由于 Circle 和 Ellipse 对象的基类相同，因此可以用 BaseEllipse 指针数组同时管理这两种对象。像 Circle 和 Ellipse 这样的类有时被称为具体（concrete）类，这表示可以创建这些类型的对象。

简而言之，ABC 描述的是至少使用一个纯虚函数的接口，从 ABC 派生出的类将根据派生类的具体特征，使用常规虚函数来实现这种接口。

13.6.1 应用 ABC 概念

您可能希望看到一个完整的 ABC 范例，因此这里将这一概念用于 Brass 和 BrassPlus 账户，首先定义一个名为 AcctABC 的 ABC。这个类包含 Brass 和 BrassPlus 类共有的所有方法和数据成员，而那些在 BrassPlus 类和 Brass 类中的行为不同的方法应被声明为虚函数。至少应有一个虚函数是纯虚函数，这样才能使 AcctABC 成为抽象类。

程序清单 13.11 的头文件声明了 AcctABC 类（ABC）、Brass 类和 BrassPlus 类（两者都是具体类）。为帮助派生类访问基类数据，AcctABC 提供了一些保护方法；派生类方法可以调用这些方法，但它们并不是派生类对象的公有接口的组成部分。AcctABC 还提供一个保护成员函数，用于处理前面几种方法中执行的格式化。另外，AcctABC 类还有两个纯虚函数，所以它确实是抽象类。

程序清单 13.11 acctabc.h

```
// acctabc.h -- bank account classes
#ifndef ACCTABC_H_
#define ACCTABC_H_

// Abstract Base Class
class AcctABC
{
private:
    enum {MAX = 35};
    char fullName[MAX];
    long acctNum;
    double balance;
protected:
    const char * FullName()const {return fullName; }
    long AcctNum()const {return acctNum; }
    std::ios_base::fmtflags SetFormat()const;
public:
    AcctABC(const char *s = "Nullbody", long an = -1,
            double bal = 0.0);
    void Deposit(double amt);
    virtual void Withdraw(double amt) = 0; // pure virtual function
    double Balance()const {return balance; };
    virtual void ViewAcct()const = 0;     // pure virtual function
    virtual ~AcctABC(){}
};

// Brass Account Class
class Brass: public AcctABC
{
public:
    Brass(const char *s = "Nullbody", long an = -1,
         double bal = 0.0): AcctABC(s, an, bal) { }
    virtual void Withdraw(double amt);
```

```
    virtual void ViewAcct()const;
    virtual ~Brass(){}
};

//Brass Plus Account Class
class BrassPlus: public AcctABC
{
private:
    double maxLoan;
    double rate;
    double owesBank;
public:
    BrassPlus (const char *s = "Nullbody", long an = -1,
          double bal = 0.0, double ml = 500,
          double r = 0.10);
    BrassPlus (const Brass & ba, double ml = 500, double r = 0.1);
    virtual void ViewAcct()const;
    virtual void Withdraw (double amt);
    void ResetMax (double m) { maxLoan = m; }
    void ResetRate (double r) { rate = r; };
    void ResetOwes() { owesBank = 0; }
};

#endif
```

接下来需要实现那些不是内联函数的方法，如程序清单 13.12 所示。

程序清单 13.12　acctABC.cpp

```
// acctabc.cpp -- bank account class methods
#include <iostream>
#include <cstring>
using std::cout;
using std::ios_base;
using std::endl;

#include "acctabc.h"

// Abstract Base Class
AcctABC ::AcctABC (const char *s, long an, double bal)
{
    std::strncpy (fullName, s, MAX - 1);
    fullName[MAX - 1] = '\0';
    acctNum = an;
    balance = bal;
}

void AcctABC ::Deposit (double amt)
{
    if (amt < 0)
        cout << "Negative deposit not allowed; "
             << "deposit is cancelled.\n";
    else
        balance += amt;
}

void AcctABC ::Withdraw (double amt)
{
    balance -= amt;
}
```

```
// protected method
ios_base ::fmtflags AcctABC::SetFormat()const
{
  // set up ###.## format
    ios_base ::fmtflags initialState =
        cout.setf (ios_base ::fixed, ios_base ::floatfield);
    cout.setf (ios_base ::showpoint);
    cout.precision (2);
    return initialState;
}

// Brass methods
void Brass ::Withdraw (double amt)
{
    if (amt < 0)
        cout << "Withdrawal amount must be positive;"
            << "withdrawal canceled.\n";
    else if (amt <= Balance())
      AcctABC ::Withdraw (amt);
    else
        cout << "Withdrawal amount of $" << amt
            << " exceeds your balance.\n"
            << "Withdrawal canceled.\n";
}

void Brass ::ViewAcct()const
{

    ios_base ::fmtflags initialState = SetFormat();
    cout << "Brass Client: " << FullName()<< endl;
    cout << "Account Number: " << AcctNum()<< endl;
    cout << "Balance: $" << Balance()<< endl;
    cout.setf (initialState);
}

// BrassPlus Methods
BrassPlus ::BrassPlus (const char *s, long an, double bal,
        double ml, double r): AcctABC (s, an, bal)
{
    maxLoan = ml;
    owesBank = 0.0;
    rate = r;
}

BrassPlus ::BrassPlus (const Brass & ba, double ml, double r)
       : AcctABC (ba)   // uses implicit copy constructor
{
    maxLoan = ml;
    owesBank = 0.0;
    rate = r;
}

void BrassPlus::ViewAcct()const
{
    ios_base ::fmtflags initialState = SetFormat();

    cout << "BrassPlus Client: " << FullName()<< endl;
    cout << "Account Number: " << AcctNum()<< endl;
    cout << "Balance: $" << Balance()<< endl;
    cout << "Maximum loan: $" << maxLoan << endl;
```

```
    cout << "Owed to bank: $" << owesBank << endl;
    cout << "Loan Rate: " << 100 * rate << "%\n";
    cout.setf (initialState);
}

void BrassPlus::Withdraw (double amt)
{
    ios_base ::fmtflags initialState = SetFormat();

    double bal = Balance();
    if (amt <= bal)
        AcctABC ::Withdraw (amt);
    else if (amt <= bal + maxLoan - owesBank)
    {
        double advance = amt - bal;
        owesBank += advance * (1.0 + rate);
        cout << "Bank advance: $" << advance << endl;
        cout << "Finance charge: $" << advance * rate << endl;
        Deposit (advance);
        AcctABC ::Withdraw (amt);
    }
    else
        cout << "Credit limit exceeded. Transaction cancelled.\n";
    cout.setf (initialState);
}
```

保护方法 FullName() 和 AcctNum() 提供了对数据成员 fullName 和 acctNum 的只读访问，使得可以进一步定制每个派生类的 ViewAcct()。

对 Brass 和 BrassPlus 账户的这种新实现，使用方式与旧实现相同，因为类方法的名称和接口都与以前一样。例如，为使程序清单 13.10 能够使用新的实现，需要采取下面的步骤将 usebrass2.cpp 转换为 usebrass3.cpp：

- 使用 acctabc.cpp 而不是 brass.cpp 来链接 usebrass2.cpp。
- 包含文件 acctabc.h，而不是 brass.h。
- 将“Brass * p_clients[CLIENTS]; ”替换为“AcctABC * p_clients[CLIENTS]; ”。

13.6.2　ABC 理念

在处理继承的问题上，RatedPlayer 范例使用的方法比较随意，而 ABC 方法比它更具系统性、更规范。设计 ABC 之前，首先应开发一个模型——指出编程问题所需的类以及它们之间相互关系。一种学院派思想认为，如果要设计类继承层次，则只能将那些不会被用作基类的类设计为具体的类。这种方法的设计更清晰，复杂程度更低。

实际应用注意事项：使用 ABC 实施接口规则

可以将 ABC 看作是一种必须实施的接口。ABC 要求具体派生类覆盖其纯虚函数——迫使派生类遵循 ABC 所设置的接口规则。这种模型在基于组件的编程模式中很常见，在这种情况下，使用 ABC 使得组件设计人员能够制定“接口约定”，这样确保了从 ABC 派生的所有组件都至少支持 ABC 指定的功能。

13.7　继承和动态内存分配

继承是怎样与动态内存分配（使用 new 和 delete）进行互动的呢？例如，如果基类使用动态内存分配，并重新定义赋值和复制构造函数，这将怎样影响派生类的实现呢？这个问题的答案取决于派生类的属性。如果派生类也使用动态内存分配，那么就需要学习几个新的小技巧。下面来看看这两种情况。

13.7.1 第一种情况：派生类不使用 new

假设基类为下述使用了动态内存分配的类：

```
// Base Class Using DMA
class baseDMA
{
private:
    char * label;
    int rating;

public:
    baseDMA(const char * l = "null", int r = 0);
    baseDMA(const baseDMA & rs);
    virtual ~baseDMA();
    baseDMA & operator=(const baseDMA & rs);
...
};
```

声明中包含了构造函数使用 new 时需要的特殊方法：析构函数、复制构造函数和重载赋值操作符。

现在，从 baseDMA 派生出 lackDMA 类，而后者不使用 new 或者包含其他一些不常用的、需要特殊处理的设计特性：

```
// derived class without DMA
class lacksDMA: public baseDMA
{
private:
    char color[40];
public:
...
};
```

是否需要为 lackDMA 类定义显式析构函数、复制构造函数和赋值操作符呢？不需要。

首先，来看是否需要析构函数。如果没有定义析构函数，编译器将定义一个不执行任何操作的默认构造函数。实际上，派生类的默认构造函数总是要进行一些操作：执行自身的代码后调用基类析构函数。因为我们假设 lackDMA 成员不需执行任何特殊操作，所以默认析构函数是合适的。

接着来看复制构造函数。第 12 章介绍过，默认复制构造函数执行成员复制，这对于动态内存分配来说是不合适的，但对于新的 lacksDMA 成员来说是合适的。因此只需考虑继承的 baseDMA 对象。要知道，成员复制将根据数据类型采用相应的复制方式，因此，将 long 复制到 long 中是通过使用常规赋值完成的；但复制类成员或继承的类组件时，则是使用该类的复制构造函数完成的。所以，lacksDMA 类的默认复制构造函数使用显式 baseDMA 复制构造函数来复制 lacksDMA 对象的 baseDMA 部分。因此，默认复制构造函数对于新的 lacksDMA 成员来说是合适的，同时对于继承的 baseDMA 对象来说也是合适的。

对于赋值来说，也是如此。类的默认赋值操作符将自动使用基类的赋值操作符来对基类组件进行赋值。因此，默认赋值操作符也是合适的。

派生类对象的这些属性也适用于本身是对象的类成员。例如，第 10 章介绍过，实现 Stock 类时，可以使用 string 对象而不是 char 数组来存储公司名称。标准 string 类和本书前面创建的 String 类一样，也采用动态内存分配。现在，读者知道了为何这不会引发问题。Stock 的默认复制构造函数将使用 string 的复制构造函数来复制对象的 company 成员；Stock 的默认赋值操作符将使用 string 的赋值操作符给对象的 company 成员赋值；而 Stock 的析构函数（默认或其他析构函数）将自动调用 string 的析构函数。

13.7.2 第二种情况：派生类使用 new

假设派生类使用了 new：

```
// derived class with DMA
class hasDMA: public baseDMA
{
```

```
private:
    char * style;  // use new in constructors
public:
...
};
```

在这种情况下，必须为派生类定义显式析构函数、复制构造函数和赋值操作符。下面依次考虑这些方法。

派生类析构函数自动调用基类的析构函数，故其自身的职责是对派生类构造函数执行工作的进行清理。因此，hasDMA 析构函数必须释放指针 style 管理的内存，并依赖于 baseDMA 的析构函数来释放指针 label 管理的内存。

```
baseDMA ::~baseDMA() // takes care of baseDMA stuff
{
    delete [] label;
}

hasDMA ::~hasDMA()    // takes care of hasDMA stuff
{
    delete [] style;
}
```

接下来看复制构造函数。BaseDMA 的复制构造函数遵循下面的常规模式：

```
baseDMA ::baseDMA(const baseDMA & rs)
{
    label = new char[std::strlen(rs.label) + 1];
    std::strcpy(label, rs.label);
    rating = rs.rating;
}
```

hasDMA 复制构造函数只能访问 hasDMA 的数据，因此它必须调用 baseDMA 复制构造函数来处理共享的 baseDMA 数据：

```
hasDMA ::hasDMA(const hasDMA & hs)
        : baseDMA(hs)
{
    style = new char[std::strlen(hs.style) + 1];
    std::strcpy(style, hs.style);
}
```

需要注意的一点是，成员初始化列表将一个 hasDMA 引用传递给 baseDMA 构造函数。没有参数类型为 hasDMA 引用的 baseDMA 构造函数，也不需要这样的构造函数。因为复制构造函数 baseDMA 有一个 baseDMA 引用参数，而基类引用可以指向派生类型。因此，baseDMA 复制构造函数将使用 hasDMA 参数的 baseDMA 部分来构造新对象的 baseDMA 部分。

接下来看赋值操作符。BaseDMA 赋值操作符遵循下述常规模式：

```
baseDMA & baseDMA ::operator=(const baseDMA & rs)
{
    if (this == &rs)
        return *this;
    delete [] label;
    label = new char[std::strlen(rs.label) + 1];
    std::strcpy(label, rs.label);
    rating = rs.rating;
    return *this;
}
```

由于 hasDMA 也使用动态内存分配，所以它也需要一个显式赋值操作符。作为 hasDMA 的方法，它只能直接访问 hasDMA 的数据。然而，派生类的显式赋值操作符必须负责所有继承的 baseDMA 基类对象的赋值，可以通过显式调用基类赋值操作符来完成这项工作，如下所示：

```
hasDMA & hasDMA ::operator=(const hasDMA & hs)
{
    if (this == &hs)
```

```
    return *this;
baseDMA ::operator=(hs);  // copy base portion
style = new char[std::strlen(hs.style) + 1];
std::strcpy(style, hs.style);
return *this;
}
```

下述语句：

```
baseDMA ::operator=(hs);  // copy base portion
```

看起来有点奇怪。但通过使用函数表示法，而不是操作符表示法，可以使用作用域解析操作符。实际上，该语句的含义如下：

```
*this = hs;  // use baseDMA ::operator=()
```

当然编译器将忽略注释，所以使用后面的代码时，编译器将使用 hasDMA ::operator=()，从而形成递归调用。使用函数表示法使得赋值操作符被正确调用。

总之，当基类和派生类都采用动态内存分配时，派生类的析构函数、复制构造函数以及赋值操作符都必须使用相应的基类方法来处理基类元素。这种要求是通过三种不同的方式来满足的。对于析构函数，这是自动完成的；对于构造函数，这是通过在初始化成员列表中调用基类的复制构造函数来完成的；如果不这样做，将自动调用基类的默认构造函数。对于赋值操作符，这是通过使用作用域解析操作符显式地调用基类的赋值操作符来完成的。

13.7.3 使用动态内存分配和友元的继承范例

为演示这些有关继承和动态内存分配的概念，我们将刚才介绍过的 baseDMA、lacksDMA 和 hasDMA 类集成到一个范例中。程序清单 13.13 是这些类的头文件。除了前面介绍的内容外，这个头文件还包含一个友元函数，以说明派生类如何访问基类的友元。

程序清单 13.13 dma.h

```
// dma.h  -- inheritance and dynamic memory allocation
#ifndef DMA_H_
#define DMA_H_
#include <iostream>

// Base Class Using DMA
class baseDMA
{
private:
    char * label;
    int rating;

public:
    baseDMA(const char * l = "null", int r = 0);
    baseDMA(const baseDMA & rs);
    virtual ~baseDMA();
    baseDMA & operator=(const baseDMA & rs);
    friend std::ostream & operator<<(std::ostream & os, const baseDMA & rs);
};

// derived class without DMA
// no destructor needed
// uses implicit copy constructor
// uses implicit assignment operator
class lacksDMA: public baseDMA
{
private:
   enum {COL_LEN = 40};
   char color[COL_LEN];
public:
```

```
    lacksDMA (const char * c = "blank", const char * l = "null",
            int r = 0);
    lacksDMA (const char * c, const baseDMA & rs);
    friend std::ostream & operator<< (std::ostream & os,
                        const lacksDMA & rs);
};

// derived class with DMA
class hasDMA: public baseDMA
{
private:
    char * style;
public:
    hasDMA (const char * s = "none", const char * l = "null",
            int r = 0);
    hasDMA (const char * s, const baseDMA & rs);
    hasDMA (const hasDMA & hs);
    ~hasDMA();
    hasDMA & operator= (const hasDMA & rs);
    friend std::ostream & operator<< (std::ostream & os, const hasDMA & rs);
};

#endif
```

程序清单 13.14 列出了类 baseDMA、lackDMA 和 hasDMA 的方法定义。

程序清单 13.14　dma.cpp

```
// dma.cpp --dma class methods

#include "dma.h"
#include <cstring>

// baseDMA methods
baseDMA::baseDMA(const char * l, int r)
{
    label = new char[std::strlen(l) + 1];
    std::strcpy(label, l);
    rating = r;
}

baseDMA::baseDMA(const baseDMA & rs)
{
    label = new char[std::strlen(rs.label) + 1];
    std::strcpy(label, rs.label);
    rating = rs.rating;
}

baseDMA::~baseDMA()
{
    delete [] label;
}

baseDMA & baseDMA::operator=(const baseDMA & rs)
{
    if (this == &rs)
        return *this;
    delete [] label;
    label = new char[std::strlen(rs.label) + 1];
    std::strcpy(label, rs.label);
    rating = rs.rating;
```

```
    return *this;
}

std::ostream & operator<<(std::ostream & os, const baseDMA & rs)
{
    os << "Label: " << rs.label << std::endl;
    os << "Rating: " << rs.rating << std::endl;
    return os;
}

// lacksDMA methods
lacksDMA::lacksDMA(const char * c, const char * l, int r)
    : baseDMA(l, r)
{
    std::strncpy(color, c, 39);
    color[39] = '\0';
}

lacksDMA::lacksDMA(const char * c, const baseDMA & rs)
    : baseDMA(rs)
{
    std::strncpy(color, c, COL_LEN - 1);
    color[COL_LEN - 1] = '\0';
}

std::ostream & operator<<(std::ostream & os, const lacksDMA & ls)
{
    os << (const baseDMA &) ls;
    os << "Color: " << ls.color << std::endl;
    return os;
}

// hasDMA methods
hasDMA::hasDMA(const char * s, const char * l, int r)
    : baseDMA(l, r)
{
    style = new char[std::strlen(s) + 1];
    std::strcpy(style, s);
}

hasDMA::hasDMA(const char * s, const baseDMA & rs)
    : baseDMA(rs)
{
    style = new char[std::strlen(s) + 1];
    std::strcpy(style, s);
}

hasDMA::hasDMA(const hasDMA & hs)
    : baseDMA(hs)  // invoke base class copy constructor
{
    style = new char[std::strlen(hs.style) + 1];
    std::strcpy(style, hs.style);
}

hasDMA::~hasDMA()
{
    delete [] style;
}

hasDMA & hasDMA::operator=(const hasDMA & hs)
```

```
{
    if (this == &hs)
        return *this;
    baseDMA::operator=(hs);  // copy base portion
    style = new char[std::strlen(hs.style) + 1];
    std::strcpy(style, hs.style);
    return *this;
}

std::ostream & operator<<(std::ostream & os, const hasDMA & hs)
{
    os << (const baseDMA &) hs;
    os << "Style: " << hs.style << std::endl;
    return os;
}
```

在程序清单 13.13 和 13.14 中，需要注意的新特性是，派生类如何使用基类的友元。例如，请考虑下面这个 hasDMa 类的友元：

```
friend std::ostream & operator<< (std::ostream & os, const hasDMA & rs);
```

作为 hasDMA 类的友元，该函数能够访问 style 成员。但是，还存在一个问题：该函数如不是 baseDMA 类的友元，那它如何访问成员 lable 和 rating 呢？答案是使用 baseDMA 类的友元函数 operator<<()。下一个问题是，因为友元不是成员函数，所以不能使用作用域解析操作符来指出要使用哪个函数。这个问题的解决方法是使用强制类型转换，以便匹配原型时能够选择正确的函数。因此，代码将参数 const hasDMA & 转换成类型为 const baseDMA &的参数：

```
std::ostream & operator<< (std::ostream & os, const hasDMA & hs)
{
//  typecast to match operator<< (ostream &, const baseDMA &)
    os << (const baseDMA &) hs;
    os << "Style: " << hs.style << endl;
    return os;
}
```

程序清单 13.15 是一个测试类 baseDMA、lackDMA 和 hasDMA 的小程序。

程序清单 13.15　usedma.cpp

```
// usedma.cpp -- inheritance, friends, and DMA
// compile with dma.cpp
#include <iostream>
#include "dma.h"
int main()
{
    using std::cout;
    using std::endl;

    baseDMA shirt ("Portabelly", 8);
    lacksDMA balloon ("red", "Blimpo", 4);
    hasDMA map ("Mercator", "Buffalo Keys", 5);
    cout << shirt << endl;
    cout << balloon << endl;
    cout << map << endl;
    lacksDMA balloon2 (balloon);
    hasDMA map2;
    map2 = map;
    cout << balloon2 << endl;
    cout << map2 << endl;
     return 0;
}
```

程序清单 13.13～13.15 组成的程序的输出如下：

```
Label: Portabelly
```

```
Rating: 8

Label: Blimpo
Rating: 4
Color: red

Label: Buffalo Keys
Rating: 5
Style: Mercator
Label: Blimpo
Rating: 4
Color: red

Label: Buffalo Keys
Rating: 5
Style: Mercator
```

13.8 类设计回顾

C++可用于解决各种类型的编程问题，但不能将类设计简化成带编号的例程。不过，有一些常用的指导原则，经常复习介绍过的内容，将会对您有所帮助的。

13.8.1 编译器生成的成员函数

第 12 章介绍过，编译器会自动生成一些公有成员函数。这表明这些成员函数特别重要，下面回顾一下其中的一些函数。

1. 默认构造函数

默认构造函数要么没有参数，要么所有的参数都有默认值。如果没有定义任何构造函数，编译器将定义默认构造函数，让您能够创建对象。例如，假设 Star 是一个类，则下述代码需要使用默认构造函数：

```
Star rigel;        // create an object without explicit initialization
Star pleiades[6];  // create an array of objects
```

自动生成的默认构造函数的另一项功能是，调用基类的默认构造函数以及调用本身是对象的成员所属类的默认构造函数。

另外，如果派生类构造函数的成员初始化列表中没有显式地调用基类构造函数，则编译器将使用基类的默认构造函数来构造派生类对象的基类部分。在这种情况下，如果基类没有构造函数，将导致编译阶段错误。

如果定义了某种构造函数，编译器将不会定义默认构造函数。在这种情况下，如果需要默认构造函数，则必须自己提供。

提供构造函数的动机之一是确保对象总能被正确地初始化。另外，如果类包含指针成员，则必须初始化这些成员。因此，最好提供一个显式默认构造函数，将所有的类数据成员都初始化为合理的值。

2. 复制构造函数

复制构造函数接受其所属类的对象作为参数。例如，Star 类的复制构造函数的原型如下：

```
Star (const Star &);
```

在下述情况下，将使用复制构造函数：

- 将新的对象初始化为一个同类对象。
- 按值将对象传递给函数。
- 函数按值返回对象。
- 编译器生成临时对象。

如果程序没有使用（显式或隐式）复制构造函数，编译器将提供原型，但不提供函数定义；否则，程序将定义一个执行成员初始化的复制构造函数。也就是说，新对象的每个成员都被初始化为原始对象相应成员的值。

在某些情况下，成员初始化是不合适的。例如，使用 new 初始化的成员指针通常要求执行深复制（参见 baseDMA 类范例），或者类可能包含需要修改的静态变量。在上述情况下，需要定义自己的复制构造函数。

3. 赋值操作符

默认的赋值操作符用于处理同类对象之间的赋值。不要将赋值与初始化混淆了。如果语句创建新的对象，则使用初始化；如果语句修改已有对象的值，则是赋值：

```
Star sirius;
Star alpha = sirius;     // initialization (one notation)
Star dogstar;
dogstar = sirius;        // assignment
```

如果需要显式定义复制构造函数，则基于相同的原因，也需要显式定义赋值操作符。Star 类的赋值操作符的原型如下：

```
Star & Star::operator=(const Star &);
```

赋值操作符函数返回一个 Star 对象引用。baseDMA 类演示了一个典型的显式赋值操作符函数范例。

编译器不会生成将一种类型赋给另一种类型的赋值操作符。如果希望能够将字符串赋给 Star 对象，则方法之一是显式定义下面的操作符：

```
Star & Star::operator=(const char *) {...}
```

另一种方法是使用转换函数（参见下一节中的“转换”小节）将字符串转换成 Star 对象，然后使用 Star-to-Star 赋值函数。第一种方法的运行速度较快，但需要的代码较多，而使用转换函数可能导致编译器出现混乱。

13.8.2　其他的类方法

定义类时，还需要注意其他几点。下面的几小节将分别介绍。

1. 构造函数

构造函数不同于其他类方法，因为它创建新的对象，而其他类方法只是被现有的对象调用。这是构造函数不被继承的原因之一。继承意味着派生类对象可以使用基类的方法，然而，构造函数在完成其工作之前，对象并不存在。

2. 析构函数

一定要定义显式析构函数来释放类构造函数使用 new 分配的所有内存，并完成类对象所需的任何特殊的清理工作。对于基类，即使它不需要构造函数，也应提供一个虚拟析构函数。

3. 转换

使用一个参数就可以调用的构造函数定义了从参数类型到类类型的转换。例如，下述 Star 类的构造函数原型：

```
Star(const char *);                      // converts char * to Star
Star(const Spectral &, int members = 1); // converts Spectral to Star
```

将可转换的类型传递给以类为参数的函数时，将调用转换构造函数。例如，在如下代码中：

```
Star north;
north = "polaris";
```

第二条语句将调用 Star::operator=（const Star *）函数，使用 Star::star（const char *）生成一个 Star 对象，该对象将被用作上述赋值操作符函数的参数。这里假设没有定义将 char *赋给 Star 的赋值操作符。

在带一个参数的构造函数原型中使用 explicit 将禁止进行隐式转换，但仍允许显式转换：

```
class Star
```

```
{
...
public:
   explicit Star (const char *);
...
};
Star north;
north = "polaris";          // not allowed
north = Star ("polaris");  // allowed
```

要将类对象转换为其他类型，应定义转换函数（参见第 11 章）。转换函数可以是没有参数的类成员函数，也可以是返回类型被声明为目标类型的类成员函数。即使没有声明返回类型，函数也应返回所需的转换值。下面是一些范例：

```
Star ::Star double(){...}          // converts star to double
Star ::Star const char *(){...}   // converts to const char
```

应理智地使用这样的函数，仅当它们有帮助时才使用。另外，对于某些类，包含转换函数将增加代码的二义性。例如，假设已经为第 11 章的 vector 类型定义了 double 转换，并编写了下面的代码：

```
vector ius (6.0, 0.0);
vector lux = ius + 20.2;        // ambiguous
```

编译器可以将 ius 转换成 double 并使用 double 加法，或将 20.2 转换成 veotor（使用构造函数之一）并使用 vector 加法。但除了指出二义性外，它什么也不做。

4. 按值传递对象与传递引用

通常，编写使用对象作为参数的函数时，应按引用而不是按值来传递对象。这样做的原因之一是为了提高效率。按值传递对象涉及到生成临时拷贝，即调用复制构造函数，然后调用析构函数。调用这些函数需要时间，复制大型对象比传递引用花费的时间要多得多。如果函数不修改对象，应将参数声明为 const 引用。

按引用传递对象的另外一个原因是，在继承使用虚函数时，被定义为接受基类引用参数的函数可以接受派生类。这在本章前面介绍过（同时请参见本章后面的“虚方法”一节）。

5. 返回对象和返回引用

一些类方法返回对象。您可能注意到了，有些成员函数直接返回对象，而另一些则返回引用。有时方法必须返回对象，但如果可以不返回对象，则应返回引用，而不是返回对象。让我们具体看一下。

首先，在编码方面，直接返回对象与返回引用之间惟一的区别在于函数原型和函数头：

```
Star nova1 (const Star &);    // returns a Star object
Star & nova2 (const Star &);  // returns a reference to a Star
```

其次，应返回引用而不是返回对象的的原因在于，返回对象涉及到生成返回对象的临时拷贝，这是调用函数的程序可以使用的拷贝。因此，返回对象的时间成本包括调用复制构造函数来生成拷贝所需的时间和调用析构函数删除拷贝所需的时间。返回引用可节省时间和内存。直接返回对象与按值传递对象相似：它们都生成临时拷贝。同样，返回引用与按引用传递对象相似：调用和被调用的函数对同一个对象进行操作。

不过，并不总是可以返回引用。函数不能返回在函数中创建的临时对象的引用，因为当函数结束时，临时对象将消失，因此这种引用将是非法的。在这种情况下，应返回对象，以生成一个调用程序可以使用的拷贝。

通用的规则是，如果函数返回在函数中创建的临时对象，则不要使用引用。例如，下面的方法使用构造函数来创建一个新对象，然后返回该对象的拷贝：

```
Vector Vector::operator+ (const Vector & b) const
{
   return Vector (x + b.x, y + b.y);
}
```

如果函数返回的是通过引用或指针传递给它的对象，则应按引用返回对象。例如，下面的代码按引用返回调用函数的对象或作为参数传递给函数的对象：

```
const Stock & Stock::topval (const Stock & s) const
{
```

```
    if (s.total_val > total_val)
      return s;          // argument object
    else
      return *this;      // invoking object
}
```

6. 使用 const

使用 const 时应特别注意。可以用它来确保方法不修改参数：

```
Star ::Star (const char * s) {...} // won't change the string to which s points
```

可以使用 const 来确保方法不修改调用它的对象：

```
void Star ::show()const {...} // won't change invoking object
```

这里 const 表示 const Star * this，而 this 指向调用的对象。

通常，可以将返回引用的函数放在赋值语句的左侧，这实际上意味着可以将值赋给引用的对象。但可以使用 const 来确保引用或指针返回的值不能用于修改对象中的数据：

```
const Stock & Stock ::topval (const Stock & s) const
{
    if (s.total_val > total_val)
      return s;          // argument object
    else
      return *this;      // invoking object
}
```

该方法返回对 this 或 s 的引用。因为 this 和 s 都被声明为 const，所以函数不能对它们进行修改，这意味着返回的引用也必须被声明为 const。

注意，如果函数将参数声明为指向 const 的引用或指针，则不能将该参数传递给另一个函数，除非后者也确保了参数不会被修改。

13.8.3　公有继承的考虑因素

通常，在程序中使用继承时，有许多问题需要注意。下面来看其中的一些。

1. *is-a* 关系

要遵循 *is-a* 关系。如果派生类不是一种特殊的基类，则不要使用公有派生。例如，不应从 Brain 类派生出 Programmer 类。如果要指出程序员有大脑，应将 Brain 类对象作为 Programmer 类的成员。

在某些情况下，最好的方法可能是创建包含纯虚函数的抽象数据类，并从它派生出其他的类。

请记住，表示 *is-a* 关系的方式之一是，无须进行显式类型转换，基类指针就可以指向派生类对象，基类引用可以引用派生类对象。另外，反过来是行不通的，即不能在不进行显式类型转换的情况下，将派生类指针或引用指向基类对象。这种显式类型转换（向下强制转换）可能有意义，也可能没有，这取决于类声明（参见图 13.4）。

2. 什么不能被继承

构造函数是不能继承的，也就是说，创建派生类对象时，必须调用派生类的构造函数。不过，派生类构造函数通常使用成员初始化列表句法来调用基类构造函数，以创建派生对象的基类部分。如果派生类构造函数没有使用成员初始化列表句法显式调用基类构造函数，将使用基类的默认构造函数。在继承链中，每个类都可以使用成员初始化列表将信息传递回相邻的基类。

析构函数也是不能继承的。不过，在释放对象时，程序将首先调用派生类的析构函数，然后调用基类的析构函数。如果基类有默认析构函数，编译器将为派生类生成默认析构函数。通常，对于基类，其析构函数应设置为虚拟的。

3. 赋值操作符

赋值操作符是不能继承的，原因很简单。派生类继承的方法的特征标与基类完全相同，但赋值操作符的特征标随类而异，这是因为它包含一个类型为其所属类的形参。赋值操作符确实有一些有趣的特征，下

面介绍它们。

如果编译器发现程序将一个对象赋给同一个类的另一个对象，它将自动为这个类提供一个赋值操作符。这个操作符的默认或隐式版本将采用成员赋值，即将原对象的相应成员赋给目标对象的每个成员。不过，如果对象属于派生类，编译器将使用基类赋值操作符来处理派生对象中基类部分的赋值。如果显式地为基类提供了赋值操作符，将使用该操作符。与此相似，如果成员是另一个类的对象，则对于该成员，将使用其所属类的赋值操作符。

正如多次提到的，如果类构造函数使用 new 来初始化指针，则需要提供一个显式赋值操作符。因为对于派生对象的基类部分，C++将使用基类的赋值操作符，所以不需要为派生类重新定义赋值操作符，除非它添加了需要特别留意的数据成员。例如，baseDMA 类显式地定义了赋值，但派生类 lackDMA 使用为它生成的隐式赋值操作符。

不过，如果派生类使用了 new，则必须提供显式赋值操作符。必须给类的每个成员提供赋值操作符，而不仅仅是新成员。HasDMA 类演示了如何完成这项工作：

```
hasDMA & hasDMA::operator=(const hasDMA & hs)
{
    if (this == &hs)
        return *this;
    baseDMA ::operator=(hs);  // copy base portion
    style = new char[std::strlen(hs.style) + 1];
    std::strcpy(style, hs.style);
    return *this;
}
```

将派生类对象赋给基类对象将会如何呢（注意，这不同于将基类引用初始化为派生类对象）？请看下面的例子：

```
Brass blips;                                                    // base class
BrassPlus snips("Rafe Plosh", 91191, 3993.19, 600.0, 0.12);     // derived class
blips = snips;                    // assign derived object to base object
```

这将使用哪个赋值操作符呢？赋值语句将被转换成左边的对象调用的一个方法：

```
blips.operator=(snips);
```

其中左边的对象是 Brass 对象，因此它将调用 Brass ::operator =（const Brass &）函数。*is-a* 关系允许 Brass 引用指向派生类对象，如 Snips。赋值操作符只处理基类成员，所以上述赋值操作将忽略 Snips 的 maxLoan 成员和其他 BrassPlus 成员。简而言之，可以将派生对象赋给基类对象，但这只涉及到基类的成员。

相反的操作将如何呢？即可以将基类对象赋给派生类对象吗？请看下面的例子：

```
Brass gp("Griff Hexbait", 21234, 1200);   // base class
BrassPlus temp;                           // derived class
temp = gp;   // possible?
```

上述赋值语句将被转换为如下所示：

```
temp.operator=(gp);
```

左边的对象是 BrassPlus 对象，所以它调用 BrassPlus ::operator=（const BrassPlus &）函数。不过，派生类引用不能自动引用基类对象，因此上述代码不能运行，除非有下面的转换构造函数：

```
BrassPlus(const Brass &);
```

与 BrassPlus 类的情况相似，转换构造函数可以接受一个类型为基类的参数和其他参数，条件是其他参数有默认值：

```
BrassPlus(const Brass & ba, double ml = 500, double r = 0.1);
```

如果有转换构造函数，程序将通过它根据 gp 来创建一个临时 BrassPlus 对象，然后将它用作赋值操作符的参数。

另一种方法是，定义一个用于将基类赋给派生类的赋值操作符：

```
BrassPlus & BrassPlus ::operator=(const Brass &){...}
```

该赋值操作符的类型与赋值语句完全匹配，因此无须进行类型转换。

简而言之，问题“是否可以将基类对象赋给派生对象？”的答案是“也许”。如果派生类包含了这样的构造函数，即对将基类对象转换为派生类对象进行了定义，则可以将基类对象赋给派生对象。如果派生类

定义了用于将基类对象赋给派生对象的赋值操作符，则也可以这样做。如果上述两个条件都不满足，则不能这样做，除非使用显式强制类型转换。

4. 私有成员与保护成员

对派生类而言，保护成员类似于公有成员；但对于外部而言，保护成员与私有成员类似。派生类可以直接访问基类的保护成员，但只能通过基类的成员函数来访问私有成员。因此，将基类成员设置为私有的可以提高安全性，而将它们设置为保护成员则可简化代码的编写工作，并提高访问速度。Stroustrup 在其 *The Design and Evolution of C++* 一书中指出，使用私用数据成员比使用保护数据成员更好，但保护方法很有用。

5. 虚方法

设计基类时，必须确定是否将类方法声明为虚拟的。如果希望派生类能够重新定义方法，则应在基类中将方法定义为虚拟的，这样可以启用晚期联编（动态联编）；如果不希望重新定义方法，则不必将其声明为虚拟的，这样虽然无法禁止他人重新定义方法，但表达了这样的意思：您不希望它被重新定义。

请注意，不适当的代码将阻止动态联编。例如，请看下面的两个函数：

```
void show (const Brass & rba)
{
    rba.ViewAcct();
    cout << endl;
}

void inadequate (Brass ba)
{
    ba.ViewAcct();
    cout << endl;
}
```

第一个函数按引用传递对象，第二个按值传递对象。

现在，假设将派生类参数传递给上述两个函数：

```
BrassPlus buzz ("Buzz Parsec", 00001111, 4300);
show (buzz);
inadequate (buzz);
```

show()函数调用使 rba 参数成为 BrassPlus 对象 buzz 的引用，因此，rba.ViewAcct()被解释为 BrassPlus 版本，正如应该的那样。但在 inadequate()函数中（它是按值传递对象的），ba 是 Brass（const Brass &）构造函数创建的一个 Brass 对象（自动向上强制转换使得构造函数参数可以引用一个 BrassPlus 对象）。因此，在 inadequate()中，ba.ViewAcct()是 Brass 版本，所以只有 buzz 的 Brass 部分被显示。

6. 析构函数

正如前面介绍的，基类的析构函数应当是虚拟的。这样，当通过指向对象的基类指针或引用来删除派生对象时，程序将首先调用派生类的析构函数，然后调用基类的析构函数，而不仅仅是调用基类的析构函数。

7. 友元函数

由于友元函数并非类成员，因此不能继承。然而，您可能希望派生类的友元函数能够使用基类的友元函数。为此，可以通过强制类型转换将派生类引用或指针转换为基类引用或指针，然后使用转换后的指针或引用来调用基类的友元函数：

```
ostream & operator<<(ostream & os, const hasDMA & hs)
{
// type cast to match operator<<(ostream & , const baseDMA &)
    os << (const baseDMA &) hs;
    os << "Style: " << hs.style << endl;
    return os;
```

}

也可以使用第 15 章将讨论的操作符 dynamic_cast<>来进行强制类型转换：

```
os << dynamic_cast<const baseDMA &> (hs);
```

鉴于第 15 章将讨论的原因，这是更佳的强制类型转换方式。

8. 有关使用基类方法的说明

以公有方式派生的类的对象可以通过多种方式来使用基类的方法：

- 派生类对象自动使用继承而来的基类方法，如果派生类没有重新定义该方法。
- 派生类的构造函数自动调用基类的构造函数。
- 派生类的构造函数自动调用基类的默认构造函数，如果没有在成员初始化列表中指定其他构造函数。
- 派生类构造函数显式地调用成员初始化列表中指定的基类构造函数。
- 派生类方法可以使用作用域解析操作符来调用公有的和受保护的基类方法。
- 派生类的有元函数可以通过强制类型转换，将派生类引用或指针转换为基类引用或指针，然后使用该引用或指针来调用基类的友元函数。

13.8.4 类函数小结

C++类函数有很多不同的变体，其中有些可以继承，有些不可以。有些操作符函数既可以是成员函数，也可以是友元，而有些操作符函数只能是成员函数。表 13.1（摘自 *The Annotated C++ Reference Manual*）总结了这些特征，其中 *op*=表示诸如+=、*=等格式的赋值操作符。注意，*op*=操作符的特征与“其他操作符”类别并没有区别。单独列出 *op*=旨在指出这些操作符与=操作符的行为是不同的。

表 13.1 成员函数属性

函 数	能否继承	成员还是友元	默认能否生成	能否为虚函数	是否可以有返回类型
构造函数	否	成员	能	否	否
析构函数	否	成员	能	能	否
=	否	成员	能	能	能
&	能	任意	能	能	能
转换函数	能	成员	否	能	否
()	能	成员	否	能	能
[]	能	成员	否	能	能
->	能	成员	否	能	能
op=	能	任意	否	能	能
new	能	静态成员	否	否	void*
delete	能	静态成员	否	否	void
其他操作符	能	任意	否	能	能
其他成员	能	成员	否	能	能
友元	否	友元	否	否	能

13.9 总结

继承通过使用已有的类（基类）定义新的类（派生类），使得能够根据需要修改编程代码。公有继承建立 *is-a* 关系，这意味着派生类对象也应该是某种基类对象。作为 *is-a* 模型的一部分，派生类继承基类的数据成员和大部分方法，但不继承基类的构造函数、析构函数和赋值操作符。派生类可以直接访问基类的公

有成员和保护成员，并能够通过基类的公有方法和保护方法访问基类的私有成员。可以在派生类中新增数据成员和方法，还可以将派生类用作基类，来做进一步的开发。每个派生类都必须有自己的构造函数。程序创建派生类对象时，将首先调用基类的构造函数，然后调用派生类的构造函数；程序删除对象时，将首先调用派生类的析构函数，然后调用基类的析构函数。

如果要将类用作基类，则可以将成员声明为保护的，而不是私有的，这样，派生类将可以直接访问这些成员。不过，使用私有成员通常可以减少出现编程问题的可能性。如果希望派生类可以重新定义基类的方法，则可以使用关键字 virtual 将它声明为虚拟的。这样对于通过指针或引用访问的对象，能够根据对象类型来处理，而不是根据引用或指针的类型来处理。具体地说，基类的析构函数通常应当是虚拟的。

可以考虑定义一个 ABC：只定义接口，而不涉及实现。例如，可以定义抽象类 Shape，然后使用它派生出具体的形状类，如 Circle 和 Square。ABC 必须至少包含一个纯虚方法，可以在声明中的分号前面加上 =0 来声明纯虚方法。

```
virtual double area()const = 0;
```

不一定非得定义纯虚方法。对于包含纯虚成员的类，不能使用它来创建对象。纯虚方法用于定义派生类的通用接口。

13.10　复习题

1. 派生类从基类那里继承了什么？

2. 派生类不能从基类那里继承什么？

3. 假设 baseDMA ::operator=() 函数的返回类型为 void，而不是 baseDMA &，这将有什么后果？如果返回类型为 baseDMA，而不是 baseDMA &，又将有什么后果？

4. 创建和删除派生类对象时，构造函数和析构函数调用的顺序是怎样的？

5. 如果派生类没有添加任何数据成员，它是否需要构造函数？

6. 如果基类和派生类定义了同名的方法，当派生类对象调用该方法时，被调用的将是哪个方法？

7. 在什么情况下，派生类应定义赋值操作符？

8. 可以将派生类对象的地址赋给基类指针吗？可以将基类对象的地址赋给派生类指针吗？

9. 可以将派生类对象赋给基类对象吗？可以将基类对象赋给派生类对象吗？

10. 假设定义了一个函数，它将基类对象的引用作为参数。为什么该函数也可以将派生类对象作为参数？

11. 假设定义了一个函数，它将基类对象作为参数（即函数按值传递基类对象）。为什么该函数也可以将派生类对象作为参数？

12. 为什么通常按引用传递对象比按值传递对象的效率更高？

13. 假设 Corporation 是基类，PublicCorporation 是派生类。再假设这两个类都定义了 head() 成员函数，ph 是指向 Corporation 类型的指针，且被赋给了一个 PublicCorporation 对象的地址。如果基类将 head() 定义为：

a. 常规非虚方法。

b. 虚方法。

则 ph->head() 将被如何解释？

14. 下述代码有什么问题？

```
class Kitchen
{
private:
   double kit_sq_ft;
public:
```

```
    Kitchen(){kit_sq_ft = 0.0; }
    virtual double area() const { return kit_sq_ft * kit_sq_ft; }
};
class House: public Kitchen
{
private:
    double all_sq_ft;
public:
    House(){all_sq_ft += kit_sq_ft; }
    double area(const char *s) const { cout << s; return all_sq_ft; }
};
```

13.11 编程练习

1. 以下面的类声明为基础：

```
// base class
class Cd {  // represents a CD disk
private:
    char performers[50];
    char label[20];
    int selections;   // number of selections
    double playtime;  // playing time in minutes
public:
    Cd(char * s1, char * s2, int n, double x);
    Cd(const Cd & d);
    Cd();
    ~Cd();
    void Report()const;  // reports all CD data
    Cd & operator=(const Cd & d);
};
```

派生出一个 Classic 类，并添加一组 char 成员，用于存储指出 CD 中主要作品的字符串。修改上述声明，使基类的所有函数都是虚拟的。如果上述定义声明的某个方法并不需要，则请删除它。使用下面的程序测试您的产品：

```
#include <iostream>
using namespace std;
#include "classic.h"    // which will contain #include cd.h
void Bravo(const Cd & disk);
int main()
{
    Cd c1("Beatles", "Capitol", 14, 35.5);
    Classic c2 = Classic("Piano Sonata in B flat, Fantasia in C",
                "Alfred Brendel", "Philips", 2, 57.17);
    Cd *pcd = &c1;

    cout << "Using object directly: \n";
    c1.Report();    // use Cd method
    c2.Report();    // use Classic method

    cout << "Using type cd * pointer to objects: \n";
    pcd->Report();  // use Cd method for cd object
    pcd = &c2;
    pcd->Report();  // use Classic method for classic object

    cout << "Calling a function with a Cd reference argument: \n";
    Bravo(c1);
    Bravo(c2);
```

```
    cout << "Testing assignment: ";
    Classic copy;
    copy = c2;
    copy.Report()

    return 0;
}

void Bravo (const Cd & disk)
{
    disk.Report();
}
```

2. 完成练习 1，但让两个类使用动态内存分配而不是长度固定的数组来记录字符串。

3. 修改 baseDMA-lacksDMA-hasDMA 类层次，让 3 个类都从一个 ABC 派生而来，然后使用与程序清单 13.10 相似的程序对结果进行测试。也就是说，它应使用 ABC 指针数组，并让用户决定要创建的对象类型。在类定义中添加 virtual View() 方法以处理数据显示。

4. Benevolent Order of Programmers 用来维护瓶装葡萄酒箱。为描述它，BOP Portmaster 设置了一个 Port 类，其声明如下：

```
#include <iostream>
using namespace std;
class Port
{
private:
    char * brand;
    char style[20]; // i.e., tawny, ruby, vintage
    int bottles;
public:
    Port (const char * br = "none", const char * st = "none", int b = 0);
    Port (const Port & p);              // copy constructor
    virtual ~Port() { delete [] brand; }
    Port & operator= (const Port & p);
    Port & operator+= (int b);          // adds b to bottles
    Port & operator-= (int b);          // subtracts b from bottles, if available
    int BottleCount()const { return bottles; }
    virtual void Show()const;
    friend ostream & operator<< (ostream & os, const Port & p);
};
```

show() 方法按下面的格式显示信息：

```
Brand: Gallo
Kind: tawny
Bottles: 20
```

operator<<() 函数按下面的格式显示信息（末尾没有换行符）：

```
Gallo, tawny, 20
```

PortMaster 完成了 Port 类的方法定义后派生了 VintagePort 类，然后被解职——因为不小心将一瓶 45 度 Cockburn 泼到了正在准备烤肉调料的人身上，VintagePort 类如下所示：

```
class VintagePort: public Port  // style necessarily = "vintage"
{
private:
    char * nickname;          // i.e., "The Noble" or "Old Velvet", etc.
    int year;                 // vintage year
public:
    VintagePort();
    VintagePort (const char * br, int b, const char * nn, int y);
    VintagePort (const VintagePort & vp);
    ~VintagePort() { delete [] nickname; }
    VintagePort & operator= (const VintagePort & vp);
    void Show()const;
```

```
    friend ostream & operator<< (ostream & os, const VintagePort & vp);
};
```

您被指定负责完成 VintagePort。

a. 第一个任务是重新创建 Port 方法定义，因为前任被开除时销毁了方法定义。

b. 第二个任务是解释为什么有的方法重新定义了，而有些没有重新定义。

c. 第三个任务是解释为何没有将 operator=() 和 operator<<() 声明为虚拟的。

d. 第四个任务是提供 VintagePort 中各个方法的定义。

第 14 章　C++中的代码重用

本章内容包括：

- *has-a* 关系。
- 包含成员对象的类。
- 模板类 valarray。
- 私有和保护继承。
- 多重继承。
- 虚基类。
- 创建类模板。
- 使用类模板。
- 模板的具体化。

C++的一个主要目标是促进代码重用。公有继承是实现这种目标的机制之一，但并不是惟一的机制。本章将介绍其他方法，其中之一是使用这样的类成员：本身是另一个类的对象。这种方法称为包含（containment）、组合（composition）或层次化（layering）。另一种方法是使用私有或保护继承。通常，包含、私有继承和保护继承用于实现 *has-a* 关系，即新的类将包含另一个类的对象。例如，HomeTheater 类可能包含一个 DvdPlayer 对象。多重继承使得能够使用两个或更多的基类派生出新的类，将基类的功能组合在一起。

第 10 章介绍了函数模板，本章将介绍类模板——另一种重用代码的方法。类模板使我们能够使用通用术语定义类，然后使用模板来创建针对特定类型定义的特殊类。例如，可以定义一个通用的堆栈模板，然后使用该模板创建一个用于表示 int 值堆栈的类和一个用于表示 double 值堆栈的类，甚至可以创建一个这样的类，即用于表示堆栈的堆栈。

14.1　包含对象成员的类

首先介绍包含对象成员的类。有一些类（如 string 类和第 16 章将介绍的标准 C++类模板）为表示类中的组件提供了方便的途径。下面来看一个具体的例子。

学生是什么？入学者？参加研究的人？残酷现实社会的避难者？有姓名和一系列考试分数的人？显然，最后一个定义完全没有表示出人的特征，但非常适合于简单的计算机表示。因此，让我们根据该定义来开发 Student 类。

将学生简化成姓名和一组考试分数后，可以使用一个包含两个成员的类来表示它：一个成员用于表示姓名，另一个成员用于表示分数。对于姓名，可以使用字符数组来表示，但这将限制姓名的长度。当然，也可以使用 char 指针和动态内存分配，然而正如第 12 章指出的，这将要求提供大量的支持代码。一种更好的方法是，使用一个由他人开发好的类的对象来表示。例如，可以使用一个 String 类（参见第 12 章）或标准 C++的 string 类的对象来表示姓名。较简单的选择是使用 string 类，因为 C++库提供了这个类的所有

实现代码。要使用 String 类，您必须在项目中包含实现文件 string1.cpp。

对于考试分数，存在类似的选择。可以使用一个定长数组，这限制了数组的长度；可以使用动态内存分配并提供大量的支持代码；也可以设计一个使用动态内存分配的类来表示该数组；还可以在标准 C++库中查找一个能够表示这种数据的类。

当然，如果 C++库提供了合适的类，实现起来将更简单。C++库确实提供了一个这样的类，它就是 valarray。

14.1.1 valarray 类简介

valarray 类是由头文件 valarray 支持的。顾名思义，这个类用于处理数值（或具有类似特性的类），它支持诸如将数组中所有元素的值相加以及在数组中找出最大和最小的值等操作。valarray 被定义为一个模板类，以便能够处理不同的数据类型。本章后面将介绍如何定义模板类，但就现在而言，读者只需知道如何使用模板类即可。

模板特性意味着声明对象时，必须指定具体的数据类型。因此，使用 valarray 类来声明一个对象时，需要在标识符 valarray 后面加上一对尖括号，并在其中包含所需的数据类型：

```
valarray<int> q_values;    // an array of int
valarray<double> weights;  // an array of double
```

这是读者需要学习的惟一新句法，它非常简单。

类特性意味着要使用 valarray 对象，需要了解这个类的构造函数和其他类方法。下面是几个使用其构造函数的例子：

```
double gpa[5] = {3.1, 3.5, 3.8, 2.9, 3.3};
valarray<double> v1;        // an array of int, size 0
valarray<int> v2(8);        // an array of 8 int elements
valarray<int> v3(10,8);     // an array of 8 int elements, each set to 10
valarray<double> v4(gpa, 4); // an array of 4 elements//initialized to the first 4 elements of gpa
```

从中可知，可以创建长度为零的空数组、指定长度的空数组、所有元素都被初始化为指定值的数组以及用常规数组中的值进行初始化的数组。

下面是这个类的一些方法：

- operator[ ] ：让您能够访问数组中的元素。
- size ：返回数组中包含的元素数。
- sum ：返回所有元素的总和。
- max ：返回最大的元素。
- min ：返回最小的元素。

还有很多其他的方法，其中的一些将在第 16 章介绍；但就这个例子而言，上述方法足够了。

14.1.2 Student 类的设计

至此，已经确定了 Student 类的设计计划：使用一个 string 对象来表示姓名，使用一个 valarray<double> 来表示考试分数。那么如何设计呢？读者可能想以公有的方式从这两个类派生出 Student 类，这将是多重公有继承，C++允许这样做，但在这里并不合适，因为学生与这些类之间的关系不是 *is-a* 模型。学生不是姓名，也不是一组考试成绩。这里的关系是 *has-a*，学生有姓名，也有一组考试分数。通常，用于建立 *has-a* 关系的 C++技术是组合（包含），即创建一个包含其他类对象的类。例如，可以将 Student 类声明为如下所示：

```
class Student
{
private:
   string name;              // use a string object for name
   valarray<double> scores;  // use a valarray<double> object for scores
   ...
};
```

同样，上述类将数据成员声明为私有的。这意味着 Student 类的成员函数可以使用 string 和 valarray<double>类的公有接口来访问和修改 name 和 scores 对象，但在类的外面不能这样做，而只能通过 Student 类的公有接口访问 name 和 scores（请参见图 14）。对于这种情况，通常被描述为 Student 类获得了其成员对象的实现，但没有继承接口。例如，Student 对象使用 string 的实现，而不是 char * name 或 char name [26]实现来保存姓名。但 Student 对象并不是天生就有使用函数 string operator+=()的能力。

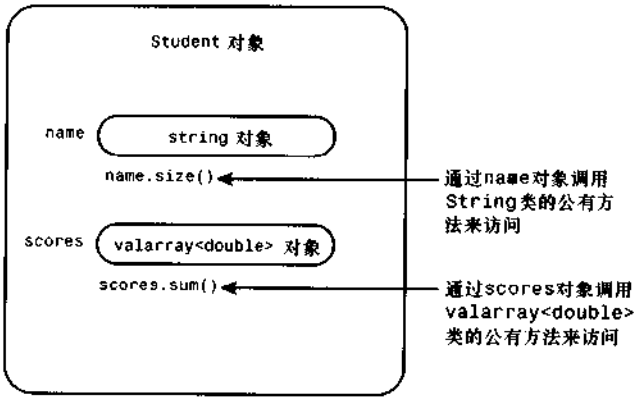


图 14.1　对象中的对象：包含

接口和实现

使用公有继承时，类可以继承接口，可能还有实现（基类的纯虚函数提供接口，但不提供实现）。获得接口是 *is-a* 关系的组成部分，而使用组合，类可以获得实现，但不能获得接口。不继承接口是 *has-a* 关系的组成部分。

对于 *has-a* 关系来说，类对象不能自动获得被包含对象的接口是一件好事。例如，string 类将+操作符重载为将两个字符串连接起来；但从概念上说，将两个 Student 对象串接起来是没有意义的。这也是这里不使用公有继承的原因之一。另一方面，被包含的类的接口部分对新类来说可能是有意义的。例如，可能希望使用 string 接口中的 operator<()方法将 Student 对象按姓名进行排序，为此可以定义 Student::operator<()成员函数，它在内部使用 string::operator<()函数。下面介绍一些细节。

14.1.3　Student 类范例

现在需要提供 Student 类的定义，当然它应包含构造函数以及一些用作 Student 类接口的方法。程序清单 14.1 是 Student 类的定义，其中所有构造函数都被定义为内联的；它还提供了一些用于输入和输出的友元函数。

程序清单 14.1　studentc.h

```
// studentc.h -- defining a Student class using containment
#ifndef STUDENTC_H_
#define STUDENTC_H_

#include <iostream>
#include <string>
#include <valarray>
class Student
{
private:
   typedef std::valarray<double> ArrayDb;
   std::string name;          // contained object
```

```
    ArrayDb scores;          // contained object
    // private method for scores output
    std::ostream & arr_out(std::ostream & os) const;
public:
    Student() : name("Null Student"), scores() {}
    Student(const std::string & s)
        : name(s), scores() {}
    explicit Student(int n) : name("Nully"), scores(n) {}
    Student(const std::string & s, int n)
        : name(s), scores(n) {}
    Student(const std::string & s, const ArrayDb & a)
        : name(s), scores(a) {}
    Student(const char * str, const double * pd, int n)
        : name(str), scores(pd, n) {}
    ~Student() {}
    double Average() const;
    const std::string & Name() const;
    double & operator[](int i);
    double operator[](int i) const;
// friends
    // input
    friend std::istream & operator>>(std::istream & is,
        Student & stu);       // 1 word
    friend std::istream & getline(std::istream & is,
        Student & stu);       // 1 line
    // output
    friend std::ostream & operator<<(std::ostream & os,
    const Student & stu);
};

#endif
```

为简化表示，Student 类的定义中包含下述 typedef：

```
typedef std::valarray<double> ArrayDb;
```

这样，在以后的代码中便可以使用表示 ArrayDb，而不是 std::valarray<double>，因此类方法和友元函数可以使用 ArrayDb 类型。将该 typedef 放在类定义的私有部分意味着可以在 Student 类的实现中使用它，但在 Student 类外面不能使用。

注意，其中的一个构造函数使用了关键字 explicit：

```
explicit Student(int n) : name("Nully"), scores(n) {}
```

本书前面说过，可以用一个参数调用的构造函数将用作从参数类型到类类型的隐式转换函数。在上述构造函数中，第一个参数表示数组的元素个数，而不是数组中的值，因此将一个构造函数用作 int 到 Student 的转换函数是没有意义的，所以使用 explicit 关闭隐式转换。如果省略该关键字，则可以编写如下所示的代码：

```
Student doh("Homer", 10);  // store "Homer", create array of 10 elements
doh = 5;   // reset name to "Nully", reset to empty array of 5 elments
```

在这里，马虎的程序员键入了 doh 而不是 doh[0]。如果构造函数省略了 explicit，则将使用构造函数调用 Student（5）将 5 转换为一个临时 Student 对象，并使用“Nully”来设置成员 name 的值。因此赋值操作将使用临时对象替换原来的 doh 值。使用了 explicit 后，编译器将认为上述赋值操作符是错误的。

C++和约束

C++包含让程序员能够限制程序结构的特性——使用 explicit 防止单参数构造函数的隐式转换，使用 const 限制方法修改数据，等等。这样做的根本原因是：在编译阶段出现错误优于在运行阶段出现错误。

1. 初始化被包含的对象

构造函数全都使用读者熟悉的成员初始化列表句法来初始化 name 和 score 成员对象。在前面的一些例

子中，构造函数用这种句法来初始化内置类型的成员：

```
Queue::Queue (int qs) : qsize (qs) {...} // initialize qsize to qs
```

上述代码在成员初始化列表中使用的是数据成员的名称（qsize）。另外，前面介绍的范例中的构造函数还使用成员初始化列表初始化派生对象的基类部分：

```
hasDMA ::hasDMA (const hasDMA & hs) : baseDMA (hs) {...}
```

对于继承的对象，构造函数在成员初始化列表中使用类名来调用特定的基类构造函数。对于成员对象，构造函数则使用成员名。例如，请看程序清单 14.1 的最后一个构造函数：

```
Student (const char * str, const double * pd, int n)
    : name (str), scores (pd, n) {}
```

因为该构造函数初始化的是成员对象，而不是继承的对象，所以在初始化列表中使用的是成员名，而不是类名。初始化列表中的每一项都调用与之匹配的构造函数，即 name（str）调用 String（const char *）构造函数，scores（pd，n）调用 ArrayDb（const double *，int）构造函数。

如果不使用初始化列表句法，情况将如何呢？C++要求在构建对象的其他部分之前，先构建继承对象的所有成员对象。因此，如果省略初始化列表，C++将使用成员对象类的默认构造函数。

初始化顺序

当初始化列表包含多个项目时，这些项目被初始化的顺序为它们被声明的顺序，而不是它们在初始化列表中的顺序。例如，假设 Student 构造函数如下：

```
Student (const char * str, const double * pd, int n)
    : scores (pd, n), name (str) {}
```

则 name 成员仍将首先被初始化，因为在类定义中它首先被声明。对于这个例子来说，初始化顺序并不重要，但如果代码使用一个成员的值作为另一个成员的初始化表达式的一部分时，初始化顺序就非常重要了。

2. 使用被包含对象的接口

被包含对象的接口不是公有的，但可以在类方法中使用它。例如，下面的代码说明了如何定义一个返回学生平均分数的函数：

```
double Student::Average() const
{
   if (scores.size() > 0)
      return scores.sum()/scores.size();
   else
      return 0;
}
```

上述代码定义了可由 Student 对象调用的方法，该方法内部使用了 valarray 的方法 size() 和 sum()。这是因为 scores 是一个 valarray 对象，所以它可以调用 valarray 类的成员函数。简而言之，Student 对象调用 Student 的方法，而后者使用被包含的 valarray 对象来调用 valarray 类的方法。

同样，可以定义一个使用 string 版本的<<操作符的友元函数：

```
// use string version of operator<<()
ostream & operator<<(ostream & os, const Student & stu)
{
   os << "Scores for " << stu.name << ":\n";
   ...
}
```

因为 stu.name 是一个 string 对象，所以它将调用 operatot<<（ostream &，const string &）函数，该函数位于 string 类中。注意，operator<<(ostream & os, const Student & stu)必须是 Student 类的友元函数，这样才能访问 name 成员。另一种方法是，在该函数中使用公有方法 Name()，而不是私有数据成员 name。

同样，该函数也可以使用 valarray 的<<实现来进行输出，不幸的是没有这样的实现；因此，Student 类定义了一个私有辅助方法来处理这种任务：

```
// private method
ostream & Student::arr_out(ostream & os) const
{
    int i;
    int lim = scores.size();
    if (lim > 0)
    {
        for (i = 0; i < lim; i++)
        {
            os << scores[i] << " ";
            if (i % 5 == 4)
                os << endl;
        }
        if (i % 5 != 0)
            os << endl;
    }
    else
        os << " empty array ";
    return os;
}
```

通过使用这样的辅助方法，可以将零乱的细节放在一个地方，使得友元函数的编码更为整洁：

```
// use string version of operator<<()
ostream & operator<<(ostream & os, const Student & stu)
{
    os << "Scores for " << stu.name << ":\n";
    stu.arr_out(os);  // use private method for scores
    return os;
}
```

辅助函数也可用作其他用户级输出函数的构建块——如果您选择提供这样的函数的话。

程序清单 14.2 是 Student 类的类方法文件，其中包含了让您能够使用[ ]操作符来访问 Student 对象中各项成绩的方法。

程序清单 14.2 student.cpp

```
// studentc.cpp -- Student class using containment
#include "studentc.h"
using std::ostream;
using std::endl;
using std::istream;
using std::string;

//public methods
double Student::Average() const
{
    if (scores.size() > 0)
        return scores.sum()/scores.size();
    else
        return 0;
}

const string & Student::Name() const
{
    return name;
}

double & Student::operator[](int i)
{
    return scores[i];        // use valarray<double>::operator[]()
}
```

```
double Student::operator[](int i) const
{
    return scores[i];
}

// private method
ostream & Student::arr_out(ostream & os) const
{
    int i;
    int lim = scores.size();
    if (lim > 0)
    {
        for (i = 0; i < lim; i++)
        {
            os << scores[i] << " ";
            if (i % 5 == 4)
                os << endl;
        }
        if (i % 5 != 0)
            os << endl;
    }
    else
        os << " empty array ";
    return os;
}

// friends

// use string version of operator>>()
istream & operator>>(istream & is, Student & stu)
{
    is >> stu.name;
    return is;
}

// use string friend getline(ostream &, const string &)
istream & getline(istream & is, Student & stu)
{
    getline(is, stu.name);
    return is;
}

// use string version of operator<<()
ostream & operator<<(ostream & os, const Student & stu)
{
    os << "Scores for " << stu.name << ":\n";
    stu.arr_out(os);  // use private method for scores
    return os;
}
```

除私有辅助方法外，程序清单 14.2 并没有新增多少代码。使用包含让您能够充分利用已有的代码。

3. 使用新的 Student 类

下面编写一个小程序来测试这个新的 Student 类。出于简化的目的，该程序将使用一个只包含 3 个 Student 对象的数组，其中每个对象保存 5 个考试成绩。另外还将使用一个不复杂的输入循环，该循环不验证输入，也不让用户中途退出。程序清单 14.3 列出了该测试程序，请务必将该程序与 Student.cpp 一起进行编译。

程序清单 14.3　use_stuc.cpp

```
// use_stuc.cpp -- using a composite class
```