PEARSON

C和C++实务精选
品味岁月积淀，读享技术菁华

中文版
累计销量超
80 000册

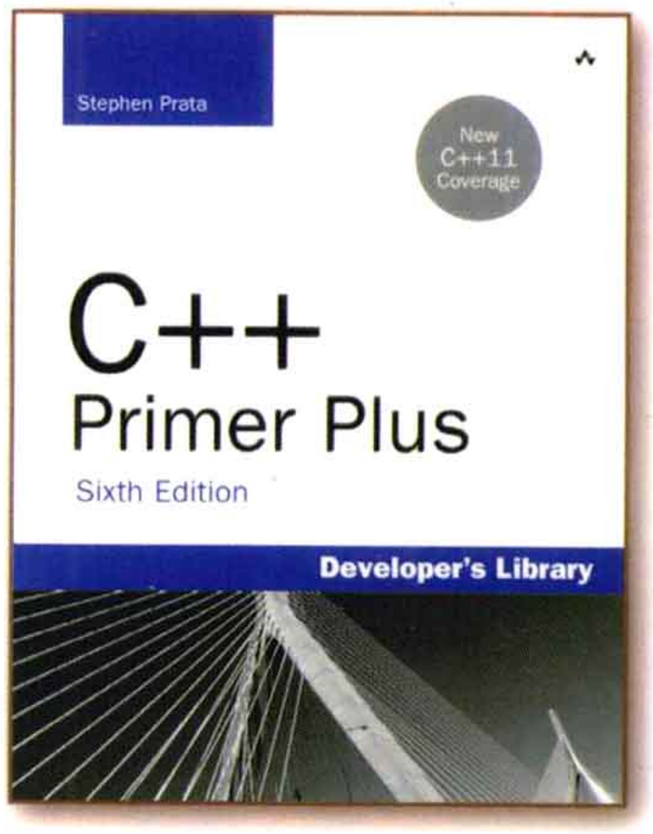
C++ Primer Plus

（第6版）中文版

[美] Stephen Prata 著　　张海龙 袁国忠 译

C++ Primer Plus Sixth Edition

- 经久不衰的C++畅销经典教程
- 涵盖C++11新标准



NLIC2970801560

人民邮电出版社
POSTS & TELECOM PRESS

PEARSON

C++ Primer Plus

（第6版）中文版

[美] Stephen Prata 著　　张海龙 袁国忠 译

C++ Primer Plus Sixth Edition

人民邮电出版社
北京

图书在版编目（CIP）数据

C++ Primer Plus中文版 : 第6版 / (美) 普拉达 (Prata,S.) 著 ; 张海龙, 袁国忠译. -- 北京 : 人民邮电出版社, 2012.7
ISBN 978-7-115-27946-0

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

中国版本图书馆CIP数据核字(2012)第065092号

版 权 声 明

Authorized translation from the English language edition, entitled C++ Primer Plus (sixth edition), 9780321776402 by Stephen Prata, published by Pearson Education, Inc., publishing as Addison-Wesley, Copyright © 2011 Pearson Education, Inc.

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from Pearson Education Inc. CHINESE SIMPLIFIED language edition published by PEARSON EDUCATION ASIA LTD., and POSTS & TELECOMMUNICATIONS PRESS Copyright © 2012.

本书封面贴有 Pearson Education（培生教育出版集团）激光防伪标签。无标签者不得销售。

C++ Primer Plus（第 6 版）中文版

◆ 著　[美] Stephen Prata
译　张海龙　袁国忠
责任编辑　傅道坤

◆ 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　电子邮件　315@ptpress.com.cn
网址　http://www.ptpress.com.cn
北京新华印刷有限公司印刷

◆ 开本：787×1092　1/16
印张：59.5
字数：1 946 千字　2012 年 7 月第 1 版
印数：1 – 10 000 册　2012 年 7 月北京第 1 次印刷
著作权合同登记号　图字：01-2012-0244 号

ISBN 978-7-115-27946-0

定价：99.00 元

读者服务热线：(010)67132692　印装质量热线：(010)67129223
反盗版热线：(010)67171154
广告经营许可证：京崇工商广字第 0021 号

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、泛型编程和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，并专辟一章介绍了 C++11 新增的功能。

全书分 18 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、分支语句和逻辑运算符、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、C++11 新增功能等内容。

本书针对 C++初学者，书中从 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 C 语言方面的背景知识。本书可作为高等院校教授 C++课程的教材，也可供初学者自学 C++时使用。

作 者 简 介

Stephen Prata 在美国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，在美国加州大学伯克利分校获得博士学位。他单独或与他人合作编写的编程图书有十多本，其中《New C Primer Plus》获得了计算机出版联合会 1990 年度最佳“How-to”计算机图书奖，《C++ Primer Plus》获得了计算机出版联合会 1991 年度最佳“How-to”计算机图书奖提名。

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范式，其中包括面向对象编程、泛型编程和传统的过程化编程。本书第 5 版是基于 ISO C++标准编写的，该标准的官方名称为 C++99 和 C++03（C++99/C++03），其中 2003 标准主要是对 1999 标准的技术修正，并没有添加任何新功能。C++在不断发展，编写本书时，新标准获得了 C++国际标准委员会的批准。在制定期间，该标准名为 C++0x，但现已改名为 C++11。大多数编译器都能很好地支持 C++99/03，而本书的大多数示例都遵守该标准。有些实现中已显现了新标准的很多功能，而本书也对这些新功能进行了探索。

本书在介绍 C++特性的同时，讨论了基本 C 语言，使两者成为有机的整体。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。书中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，本书还讲述了 C++在 C 语言的基础上新增的很多特性，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；
- 泛型（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。

初级教程方法

大约 20 年前，《C Primer Plus》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。
- 初级教程不要求您已经熟悉相关的编程概念。
- 初级教程强调的是动手学习，通过简短、容易输入的示例阐述一两个概念。
- 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您对知识的理解，从而适于自学或课堂教学。

基于上述理念，本书帮助您理解这种用途广泛的语言，并学习如何使用它。

- 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，您阅读本书后，能够编写出可靠、高效的程序，并且觉得这是一种享受。

示例代码

本书包含大量的示例代码，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。书中的示例在 Windows 7 系统、Macintosh OS X 系统和 Linux 系统上进行了测试。

使用了 C++11 功能的程序要求编译器支持这些功能，但其他程序可在遵循 C++ 99/03 的任何系统上运行。

书中完整程序的源代码可从配套网站下载，详情请参阅封底的链接信息。

本书内容

本书分为 18 章和 10 个附录。

- 第 1 章　预备知识：本章介绍 Bjarne Stroustrup 如何通过在 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。讨论面向过程语言（如 C 语言）与面向对象语言（如 C++）之间的区别。您将了解 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。您可以学习到 main( )函数扮演的角色以及 C++程序使用的一些语句。您将使用预定义的 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个类型。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++让程序员能够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 章～第 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）。您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 运算符。
- 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++的关系运算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，您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　分支语句和逻辑运算符：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，您将了解到如何使用 if、if else 和 switch 语句及条件运算符来控制程序流程，学习如何使用逻辑运算符来表达决策测试。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，您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。您将学习内联函数，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，您还将学习函数模板，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，您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，如栈。
- 第 11 章　使用类：在本章中，您将深入了解类。首先了解运算符重载，它使程序员能够定义与类

对象一起使用的运算符，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运算符成员函数是如何被用来管理类类型转换的。

- 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，定义显式拷贝构造函数和显式赋值运算符。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。您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，您还将学习类模板，它让程序员能够使用泛型定义类，然后使用模板根据具体类型创建特定的类。例如，栈模板使程序员能够创建整数栈或字符串栈。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。您还将学习 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。它还提供了高效的泛型算法库，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。您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。您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。
- 第 18 章　探讨 C++新标准：本章首先复习之前介绍过的几项 C++11 新功能，包括新类型、统一的初始化语法、自动类型推断、新的智能指针以及作用域内枚举。然后，讨论新增的右值引用类型以及如何使用它来实现移动语义。接下来，介绍了新增的类功能、lambda 表达式和可变参数模板。最后，概述了众多其他的新功能。
- 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　运算符优先级：本附录按优先级从高到低的顺序列出了 C++的运算符。
- 附录 E　其他运算符：本附录总结了正文中没有介绍的其他 C++运算符，如按位运算符等。
- 附录 F　模板类 string：本附录总结了 string 类方法和函数。
- 附录 G　标准模板库方法和函数：本附录总结了 STL 容器方法和通用的 STL 算法函数。
- 附录 H　精选读物和网上资源：本附录列出一些参考书，帮助您深入了解 C++。
- 附录 I　转换为 ISO 标准 C++：本附录提供了从 C 和老式 C++实现到标准 C++的转换指南。
- 附录 J　复习题答案：本附录提供各章结尾的复习题的答案。

对教师的提示

本书宗旨之一是，提供一本既可用于自学又可用于教学的书籍。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++，不依赖于特定实现。
- 本书内容跟踪了 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。
- 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）、栈、队列、简单链表、模拟、泛型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 << "What's up, Doc!\n";
    return 0;
}
```

- 用户需要输入的程序输入用粗体表示：

```
Please enter your name:
Plato
```

- 语法描述中的占位符用斜体表示。您应使用实际的文件名、参数等替换占位符。
- 新术语用斜体表示。

旁注：提供更深入的讨论和额外的背景知识，帮助阐明主题。

提示：提供特定编程情形下很有帮助的简单指南。

警告：指出潜在的陷阱。

注意：提供不属于其他类别的各种说明。

开发本书编程示例时使用的系统

本书的 C++11 示例是使用 Microsoft Visual C++ 2010 和带 Gnu g++ 4.5.0 的 Cygwin 开发的，它们都运行在 64 位的 Windows 7 系统上。其他示例在这些系统上进行了测试，还在 OS X 10.6.8 系统和 Ubuntu Linux 系统上分别使用 g++ 4.21 和 g++ 4.4.1 进行了测试。大多数非 C++11 示例最初都是在 Windows XP Professional 系统上使用 Microsoft Visual C++ 2003 和 Metrowerks CodeWarrior Development Studio 9 开发的，并在该系统上使用 Borland C++ 5.5 命令行编译器和 GNU gpp 3.3.3 进行了测试；其次，在运行 SuSE 9.0 Linux 的系统上使用 Comeau 4.3.3 和 GNU g++3.3.1 进行了测试；最后，在运行 OS 10.3 的 Macintosh G4 上使用 Metrowerks Development Studio 9 进行了测试。

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++和泛型编程……4
1.2.5　C++的起源……4
1.3　可移植性和标准……5
1.3.1　C++的发展……6
1.3.2　本书遵循的 C++标准……6
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……10
第 2 章　开始学习 C++……12
2.1　进入 C++……12
2.1.1　main( )函数……13
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……29
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……32
2.4.5　在多函数程序中使用 using 编译指令……33
2.5　总结……34
2.6　复习题……35
2.7　编程练习……35
第 3 章　处理数据……37
3.1　简单变量……37
3.1.1　变量名……38
3.1.2　整型……39
3.1.3　整型 short、int、long 和 long long……39
3.1.4　无符号类型……43
3.1.5　选择整型类型……45
3.1.6　整型字面值……45
3.1.7　C++如何确定常量的类型……47
3.1.8　char 类型：字符和小整数……47
3.1.9　bool 类型……53
3.2　const 限定符……54
3.3　浮点数……55
3.3.1　书写浮点数……55
3.3.2　浮点类型……56
3.3.3　浮点常量……58
3.3.4　浮点数的优缺点……58
3.4　C++算术运算符……59
3.4.1　运算符优先级和结合性……60
3.4.2　除法分支……60
3.4.3　求模运算符……61
3.4.4　类型转换……62
3.4.5　C++11 中的 auto 声明……66
3.5　总结……67
3.6　复习题……67
3.7　编程练习……68
第 4 章　复合类型……70
4.1　数组……70
4.1.1　程序说明……72
4.1.2　数组的初始化规则……73
4.1.3　C++11 数组初始化方法……73
4.2　字符串……74
4.2.1　拼接字符串常量……75
4.2.2　在数组中使用字符串……76
4.2.3　字符串输入……77
4.2.4　每次读取一行字符串输入……78
4.2.5　混合输入字符串和数字……81
4.3　string 类简介……82
4.3.1　C++11 字符串初始化……83

4.3.2 赋值、拼接和附加 ……83
4.3.3 string 类的其他操作 ……84
4.3.4 string 类 I/O ……86
4.3.5 其他形式的字符串字面值 ……87
4.4 结构简介 ……88
4.4.1 在程序中使用结构 ……89
4.4.2 C++11 结构初始化 ……91
4.4.3 结构可以将 string 类作为成员吗 ……91
4.4.4 其他结构属性 ……91
4.4.5 结构数组 ……93
4.4.6 结构中的位字段 ……94
4.5 共用体 ……94
4.6 枚举 ……95
4.6.1 设置枚举量的值 ……97
4.6.2 枚举的取值范围 ……97
4.7 指针和自由存储空间 ……97
4.7.1 声明和初始化指针 ……100
4.7.2 指针的危险 ……101
4.7.3 指针和数字 ……102
4.7.4 使用 new 来分配内存 ……102
4.7.5 使用 delete 释放内存 ……104
4.7.6 使用 new 来创建动态数组 ……104
4.8 指针、数组和指针算术 ……106
4.8.1 程序说明 ……107
4.8.2 指针小结 ……109
4.8.3 指针和字符串 ……111
4.8.4 使用 new 创建动态结构 ……114
4.8.5 自动存储、静态存储和动态存储 ……117
4.9 类型组合 ……118
4.10 数组的替代品 ……120
4.10.1 模板类 vector ……120
4.10.2 模板类 array（C++11） ……120
4.10.3 比较数组、vector 对象和 array 对象 ……120
4.11 总结 ……122
4.12 复习题 ……123
4.13 编程练习 ……123

第 5 章 循环和关系表达式 ……125

5.1 for 循环 ……125
5.1.1 for 循环的组成部分 ……126
5.1.2 回到 for 循环 ……131
5.1.3 修改步长 ……132
5.1.4 使用 for 循环访问字符串 ……133
5.1.5 递增运算符（++）和递减运算符（--） ……133
5.1.6 副作用和顺序点 ……134
5.1.7 前缀格式和后缀格式 ……135
5.1.8 递增/递减运算符和指针 ……135
5.1.9 组合赋值运算符 ……136
5.1.10 复合语句（语句块） ……136
5.1.11 其他语法技巧——逗号运算符 ……138
5.1.12 关系表达式 ……140
5.1.13 赋值、比较和可能犯的错误 ……141
5.1.14 C-风格字符串的比较 ……142
5.1.15 比较 string 类字符串 ……144
5.2 while 循环 ……145
5.2.1 for 与 while ……147
5.2.2 等待一段时间：编写延时循环 ……148
5.3 do while 循环 ……150
5.4 基于范围的 for 循环（C++11） ……152
5.5 循环和文本输入 ……152
5.5.1 使用原始的 cin 进行输入 ……152
5.5.2 使用 cin.get(char)进行补救 ……153
5.5.3 使用哪一个 cin.get( ) ……154
5.5.4 文件尾条件 ……155
5.5.5 另一个 cin.get( )版本 ……157
5.6 嵌套循环和二维数组 ……159
5.6.1 初始化二维数组 ……160
5.6.2 使用二维数组 ……160
5.7 总结 ……162
5.8 复习题 ……163
5.9 编程练习 ……163

第 6 章 分支语句和逻辑运算符 ……165

6.1 if 语句 ……165
6.1.1 if else 语句 ……167
6.1.2 格式化 if else 语句 ……168
6.1.3 if else if else 结构 ……169
6.2 逻辑表达式 ……170
6.2.1 逻辑 OR 运算符：|| ……171
6.2.2 逻辑 AND 运算符：&& ……172
6.2.3 用&&来设置取值范围 ……174
6.2.4 逻辑 NOT 运算符：! ……175
6.2.5 逻辑运算符细节 ……176
6.2.6 其他表示方式 ……177
6.3 字符函数库 cctype ……177
6.4 ?:运算符 ……179
6.5 switch 语句 ……180
6.5.1 将枚举量用作标签 ……183
6.5.2 switch 和 if else ……184
6.6 break 和 continue 语句 ……185
6.7 读取数字的循环 ……187

6.8 简单文件输入/输出 ……190
6.8.1 文本 I/O 和文本文件 ……190
6.8.2 写入到文本文件中 ……191
6.8.3 读取文本文件 ……194
6.9 总结 ……197
6.10 复习题 ……198
6.11 编程练习 ……199

第 7 章 函数——C++的编程模块 ……202

7.1 复习函数的基本知识 ……202
7.1.1 定义函数 ……203
7.1.2 函数原型和函数调用 ……205
7.2 函数参数和按值传递 ……207
7.2.1 多个参数 ……208
7.2.2 另外一个接受两个参数的函数 ……210
7.3 函数和数组 ……212
7.3.1 函数如何使用指针来处理数组 ……213
7.3.2 将数组作为参数意味着什么 ……213
7.3.3 更多数组函数示例 ……215
7.3.4 使用数组区间的函数 ……220
7.3.5 指针和 const ……221
7.4 函数和二维数组 ……224
7.5 函数和 C-风格字符串 ……225
7.5.1 将 C-风格字符串作为参数的函数 ……225
7.5.2 返回 C-风格字符串的函数 ……227
7.6 函数和结构 ……228
7.6.1 传递和返回结构 ……228
7.6.2 另一个处理结构的函数示例 ……230
7.6.3 传递结构的地址 ……234
7.7 函数和 string 对象 ……235
7.8 函数与 array 对象 ……236
7.9 递归 ……238
7.9.1 包含一个递归调用的递归 ……239
7.9.2 包含多个递归调用的递归 ……240
7.10 函数指针 ……241
7.10.1 函数指针的基础知识 ……241
7.10.2 函数指针示例 ……243
7.10.3 深入探讨函数指针 ……244
7.10.4 使用 typedef 进行简化 ……248
7.11 总结 ……248
7.12 复习题 ……249
7.13 编程练习 ……250

第 8 章 函数探幽 ……253

8.1 C++内联函数 ……253
8.2 引用变量 ……255
8.2.1 创建引用变量 ……256
8.2.2 将引用用作函数参数 ……258
8.2.3 引用的属性和特别之处 ……260
8.2.4 将引用用于结构 ……263
8.2.5 将引用用于类对象 ……268
8.2.6 对象、继承和引用 ……271
8.2.7 何时使用引用参数 ……274
8.3 默认参数 ……274
8.4 函数重载 ……276
8.4.1 重载示例 ……278
8.4.2 何时使用函数重载 ……280
8.5 函数模板 ……281
8.5.1 重载的模板 ……283
8.5.2 模板的局限性 ……285
8.5.3 显式具体化 ……285
8.5.4 实例化和具体化 ……288
8.5.5 编译器选择使用哪个函数版本 ……289
8.5.6 模板函数的发展 ……295
8.6 总结 ……297
8.7 复习题 ……297
8.8 编程练习 ……298

第 9 章 内存模型和名称空间 ……300

9.1 单独编译 ……300
9.2 存储持续性、作用域和链接性 ……304
9.2.1 作用域和链接 ……305
9.2.2 自动存储持续性 ……305
9.2.3 静态持续变量 ……309
9.2.4 静态持续性、外部链接性 ……310
9.2.5 静态持续性、内部链接性 ……313
9.2.6 静态存储持续性、无链接性 ……315
9.2.7 说明符和限定符 ……317
9.2.8 函数和链接性 ……318
9.2.9 语言链接性 ……319
9.2.10 存储方案和动态分配 ……319
9.3 名称空间 ……324
9.3.1 传统的 C++名称空间 ……324
9.3.2 新的名称空间特性 ……325
9.3.3 名称空间示例 ……331
9.3.4 名称空间及其前途 ……334
9.4 总结 ……335
9.5 复习题 ……335
9.6 编程练习 ……338

第 10 章 对象和类 ……340

10.1 过程性编程和面向对象编程 ……340
10.2 抽象和类 ……341
10.2.1 类型是什么 ……341
10.2.2 C++中的类 ……342

10.2.3 实现类成员函数 ……345
10.2.4 使用类 ……349
10.2.5 修改实现 ……350
10.2.6 小结 ……351
10.3 类的构造函数和析构函数……352
10.3.1 声明和定义构造函数……353
10.3.2 使用构造函数 ……354
10.3.3 默认构造函数 ……354
10.3.4 析构函数 ……355
10.3.5 改进 Stock 类 ……356
10.3.6 构造函数和析构函数小结……362
10.4 this 指针 ……363
10.5 对象数组 ……368
10.6 类作用域 ……370
10.6.1 作用域为类的常量……371
10.6.2 作用域内枚举（C++11）……372
10.7 抽象数据类型 ……373
10.8 总结 ……376
10.9 复习题 ……377
10.10 编程练习 ……377

第 11 章 使用类 ……380

11.1 运算符重载 ……380
11.2 计算时间：一个运算符重载示例……381
11.2.1 添加加法运算符……384
11.2.2 重载限制……387
11.2.3 其他重载运算符……388
11.3 友元 ……390
11.3.1 创建友元……391
11.3.2 常用的友元：重载<<运算符……392
11.4 重载运算符：作为成员函数还是非成员函数……397
11.5 再谈重载：一个矢量类……398
11.5.1 使用状态成员……404
11.5.2 为 Vector 类重载算术运算符 …406
11.5.3 对实现的说明……408
11.5.4 使用 Vector 类来模拟随机漫步……408
11.6 类的自动转换和强制类型转换……411
11.6.1 转换函数……415
11.6.2 转换函数和友元函数……419
11.7 总结 ……421
11.8 复习题 ……422
11.9 编程练习 ……422

第 12 章 类和动态内存分配 ……425

12.1 动态内存和类 ……425
12.1.1 复习示例和静态类成员……425
12.1.2 特殊成员函数 ……432
12.1.3 回到 Stringbad：复制构造函数的哪里出了问题 ……434
12.1.4 Stringbad 的其他问题：赋值运算符 ……436
12.2 改进后的新 String 类 ……437
12.2.1 修订后的默认构造函数 ……438
12.2.2 比较成员函数 ……439
12.2.3 使用中括号表示法访问字符 …439
12.2.4 静态类成员函数 ……441
12.2.5 进一步重载赋值运算符 ……441
12.3 在构造函数中使用 new 时应注意的事项……446
12.3.1 应该和不应该 ……447
12.3.2 包含类成员的类的逐成员复制 ……448
12.4 有关返回对象的说明 ……449
12.4.1 返回指向 const 对象的引用 ……449
12.4.2 返回指向非 const 对象的引用……449
12.4.3 返回对象 ……450
12.4.4 返回 const 对象……450
12.5 使用指向对象的指针 ……451
12.5.1 再谈 new 和 delete……453
12.5.2 指针和对象小结 ……454
12.5.3 再谈定位 new 运算符……456
12.6 复习各种技术 ……459
12.6.1 重载<<运算符……459
12.6.2 转换函数 ……460
12.6.3 其构造函数使用 new 的类……460
12.7 队列模拟 ……460
12.7.1 队列类 ……461
12.7.2 Customer 类……468
12.7.3 ATM 模拟……471
12.8 总结 ……475
12.9 复习题 ……476
12.10 编程练习 ……477

第 13 章 类继承 ……480

13.1 一个简单的基类 ……481
13.1.1 派生一个类 ……482
13.1.2 构造函数：访问权限的考虑 ……484
13.1.3 使用派生类 ……486
13.1.4 派生类和基类之间的特殊关系 ……488
13.2 继承：is-a 关系 ……489
13.3 多态公有继承 ……490

13.4 静态联编和动态联编 ……501
13.4.1 指针和引用类型的兼容性……502
13.4.2 虚成员函数和动态联编……503
13.4.3 有关虚函数注意事项……505
13.5 访问控制：protected……507
13.6 抽象基类 ……508
13.6.1 应用 ABC 概念 ……510
13.6.2 ABC 理念 ……516
13.7 继承和动态内存分配 ……516
13.7.1 第一种情况：派生类不使用 new……516
13.7.2 第二种情况：派生类使用 new……517
13.7.3 使用动态内存分配和友元的继承示例……519
13.8 类设计回顾 ……523
13.8.1 编译器生成的成员函数……523
13.8.2 其他的类方法 ……524
13.8.3 公有继承的考虑因素……527
13.8.4 类函数小结 ……530
13.9 总结 ……530
13.10 复习题 ……531
13.11 编程练习 ……531

第 14 章 C++中的代码重用……534

14.1 包含对象成员的类 ……534
14.1.1 valarray 类简介 ……535
14.1.2 Student 类的设计 ……535
14.1.3 Student 类示例 ……537
14.2 私有继承 ……543
14.2.1 Student 类示例（新版本）……543
14.2.2 使用包含还是私有继承……549
14.2.3 保护继承 ……549
14.2.4 使用 using 重新定义访问权限 ……550
14.3 多重继承 ……551
14.3.1 有多少 Worker……555
14.3.2 哪个方法 ……558
14.3.3 MI 小结 ……567
14.4 类模板 ……567
14.4.1 定义类模板 ……568
14.4.2 使用模板类 ……570
14.4.3 深入探讨模板类 ……572
14.4.4 数组模板示例和非类型参数……577
14.4.5 模板多功能性 ……578
14.4.6 模板的具体化 ……582
14.4.7 成员模板 ……584
14.4.8 将模板用作参数 ……586
14.4.9 模板类和友元 ……588
14.4.10 模板别名（C++11）……593
14.5 总结 ……594
14.6 复习题 ……595
14.7 编程练习 ……597

第 15 章 友元、异常和其他……602

15.1 友元 ……602
15.1.1 友元类 ……602
15.1.2 友元成员函数 ……606
15.1.3 其他友元关系 ……609
15.1.4 共同的友元 ……610
15.2 嵌套类 ……611
15.2.1 嵌套类和访问权限 ……612
15.2.2 模板中的嵌套 ……613
15.3 异常 ……616
15.3.1 调用 abort( ) ……616
15.3.2 返回错误码 ……617
15.3.3 异常机制 ……619
15.3.4 将对象用作异常类型 ……621
15.3.5 异常规范和 C++11 ……624
15.3.6 栈解退 ……625
15.3.7 其他异常特性 ……629
15.3.8 exception 类……631
15.3.9 异常、类和继承 ……634
15.3.10 异常何时会迷失方向 ……639
15.3.11 有关异常的注意事项 ……641
15.4 RTTI ……642
15.4.1 RTTI 的用途 ……642
15.4.2 RTTI 的工作原理……642
15.5 类型转换运算符 ……649
15.6 总结 ……652
15.7 复习题 ……653
15.8 编程练习 ……654

第 16 章 string 类和标准模板库 ……655

16.1 string 类 ……655
16.1.1 构造字符串 ……655
16.1.2 string 类输入 ……659
16.1.3 使用字符串 ……661
16.1.4 string 还提供了哪些功能 ……665
16.1.5 字符串种类 ……666
16.2 智能指针模板类 ……667
16.2.1 使用智能指针 ……668
16.2.2 有关智能指针的注意事项 ……670
16.2.3 unique_ptr 为何优于 auto_ptr……672
16.2.4 选择智能指针 ……673
16.3 标准模板库 ……674
16.3.1 模板类 vector ……675

16.3.2 可对矢量执行的操作……676
16.3.3 对矢量可执行的其他操作……680
16.3.4 基于范围的 for 循环（C++11）……684
16.4 泛型编程……684
16.4.1 为何使用迭代器……685
16.4.2 迭代器类型……688
16.4.3 迭代器层次结构……689
16.4.4 概念、改进和模型……690
16.4.5 容器种类……695
16.4.4 关联容器……702
16.4.5 无序关联容器（C++11）……707
16.5 函数对象……707
16.5.1 函数符概念……707
16.5.2 预定义的函数符……710
16.5.3 自适应函数符和函数适配器……711
16.6 算法……713
16.6.1 算法组……713
16.6.2 算法的通用特征……714
16.6.3 STL 和 string 类……715
16.6.4 函数和容器方法……716
16.6.5 使用 STL……717
16.7 其他库……720
16.7.1 vector、valarray 和 array……720
16.7.2 模板 initializer_list（C++11）……724
16.7.3 使用 initializer_list……725
16.8 总结……727
16.9 复习题……728
16.10 编程练习……728

第 17 章 输入、输出和文件……731

17.1 C++输入和输出概述……731
17.1.1 流和缓冲区……732
17.1.2 流、缓冲区和 iostream 文件……733
17.1.3 重定向……735
17.2 使用 cout 进行输出……736
17.2.1 重载的<<运算符……736
17.2.2 其他 ostream 方法……738
17.2.3 刷新输出缓冲区……740
17.2.4 用 cout 进行格式化……741
17.3 使用 cin 进行输入……753
17.3.1 cin>>如何检查输入……754
17.3.2 流状态……756
17.3.3 其他 istream 类方法……759
17.3.4 其他 istream 方法……764
17.4 文件输入和输出……768
17.4.1 简单的文件 I/O……768
17.4.2 流状态检查和 is_open( )……770
17.4.3 打开多个文件……771
17.4.4 命令行处理技术……772
17.4.5 文件模式……773
17.4.6 随机存取……781
17.5 内核格式化……788
17.6 总结……790
17.7 复习题……791
17.8 编程练习……792

第 18 章 探讨 C++新标准……795

18.1 复习前面介绍过的 C++11 功能……795
18.1.1 新类型……795
18.8.2 统一的初始化……795
18.1.3 声明……796
18.1.4 智能指针……798
18.1.5 异常规范方面的修改……798
18.1.6 作用域内枚举……799
18.1.7 对类的修改……799
18.1.8 模板和 STL 方面的修改……800
18.1.9 右值引用……801
18.2 移动语义和右值引用……802
18.2.1 为何需要移动语义……802
18.2.2 一个移动示例……803
18.2.3 移动构造函数解析……808
18.2.4 赋值……809
18.2.5 强制移动……809
18.3 新的类功能……813
18.3.1 特殊的成员函数……813
18.3.2 默认的方法和禁用的方法……814
18.3.3 委托构造函数……815
18.3.4 继承构造函数……815
18.3.5 管理虚方法：override 和 final……817
18.4 Lambda 函数……817
18.4.1 比较函数指针、函数符和 Lambda 函数……818
18.4.2 为何使用 lambda……820
18.5 包装器……822
18.5.1 包装器 function 及模板的低效性……823
18.5.2 修复问题……825
18.5.3 其他方式……826
18.6 可变参数模板……827
18.6.1 模板和函数参数包……827
18.6.2 展开参数包……828
18.6.3 在可变参数模板函数中使用递归……828
18.7 C++11 新增的其他功能……831
18.7.1 并行编程……831

18.7.2 新增的库 ……831
18.7.3 低级编程 ……832
18.7.4 杂项 ……832
18.8 语言变化 ……832
18.8.1 Boost 项目 ……833
18.8.2 TR1 ……833
18.8.3 使用 Boost ……833
18.9 接下来的任务 ……834
18.10 总结 ……834
18.11 复习题 ……835
18.12 编程练习 ……838

附录 A 计数系统 ……839

A.1 十进制数 ……839
A.2 八进制整数 ……839
A.3 十六进制数 ……839
A.4 二进制数 ……840
A.5 二进制和十六进制 ……841

附录 B C++保留字 ……842

B.1 C++关键字 ……842
B.2 替代标记 ……842
B.3 C++库保留名称 ……843
B.4 有特殊含义的标识符 ……843

附录 C ASCII 字符集 ……845

附录 D 运算符优先级 ……849

附录 E 其他运算符 ……852

E.1 按位运算符 ……852
E.1.1 移位运算符 ……852
E.1.2 逻辑按位运算符 ……853
E.1.3 按位运算符的替代表示 ……855
E.1.4 几种常用的按位运算符技术 ……856
E.2 成员解除引用运算符 ……857
E.3 alignof（C++11） ……860
E.4 noexcept（C++11） ……861

附录 F 模板类 string ……862

F.1 13 种类型和一个常量 ……862
F.2 数据信息、构造函数及其他 ……863
F.2.1 默认构造函数 ……865
F.2.2 使用 C-风格字符串的构造函数 ……865
F.2.3 使用部分 C-风格字符串的构造函数 ……865
F.2.4 使用左值引用的构造函数 ……866
F.2.5 使用右值引用的构造函数（C++11） ……866
F.2.6 使用一个字符的 n 个副本的构造函数 ……867
F.2.7 使用区间的构造函数 ……867
F.2.8 使用初始化列表的构造函数（C++11） ……868
F.2.9 内存杂记 ……868
F.3 字符串存取 ……868
F.4 基本赋值 ……869
F.5 字符串搜索 ……869
F.5.1 find( )系列 ……870
F.5.2 rfind( )系列 ……870
F.5.3 find_first_of( )系列 ……870
F.5.4 find_last_of( )系列 ……871
F.5.5 find_first_not_of( )系列 ……871
F.5.6 find_last_not_of( )系列 ……871
F.6 比较方法和函数 ……872
F.7 字符串修改方法 ……873
F.7.1 用于追加和相加的方法 ……873
F.7.2 其他赋值方法 ……874
F.7.3 插入方法 ……874
F.7.4 清除方法 ……875
F.7.5 替换方法 ……875
F.7.6 其他修改方法：copy( )和 swap( ) ……876
F.8 输出和输入 ……876

附录 G 标准模板库方法和函数 ……877

G.1 STL 和 C++11 ……877
G.1.1 新增的容器 ……877
G.1.2 对 C++98 容器所做的修改 ……877
G.2 大部分容器都有的成员 ……878
G.3 序列容器的其他成员 ……881
G.4 set 和 map 的其他操作 ……883
G.4 无序关联容器（C++11） ……884
G.5 STL 函数 ……886
G.5.1 非修改式序列操作 ……886
G.5.2 修改式序列操作 ……890
G.5.3 排序和相关操作 ……897
G.5.4 数值运算 ……907

附录 H 精选读物和网上资源 ……909

H.1 精选读物 ……909
H.2 网上资源 ……910

附录 I 转换为 ISO 标准 C++ ……911

I.1 使用一些预处理器编译指令的替代品 ……911

I.1.1 使用 const 而不是#define 来定义常量……911
I.1.2 使用 inline 而不是# define 来定义小型函数……912
I.2 使用函数原型……913
I.3 使用类型转换……913
I.4 熟悉 C++特性……913
I.5 使用新的头文件……914
I.6 使用名称空间……914
I.7 使用智能指针……915
I.8 使用 string 类……915
I.9 使用 STL……915

附录 J 复习题答案……916

第 2 章复习题答案……916
第 3 章复习题答案……916
第 4 章复习题答案……917
第 5 章复习题答案……919
第 6 章复习题答案……919
第 7 章复习题答案……920
第 8 章复习题答案……922
第 9 章复习题答案……924
第 10 章复习题答案……925
第 11 章复习题答案……927
第 12 章复习题答案……927
第 13 章复习题答案……929
第 14 章复习题答案……930
第 15 章复习题答案……931
第 16 章复习题答案……932
第 17 章复习题答案……933
第 18 章复习题答案……935

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。
- 过程性编程和面向对象编程。
- 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 语言的基础上添加泛型编程概念的。
- 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和泛型编程的支持，在 20 世纪 90 年代便是最重要的编程语言之一，并在 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——泛型编程。这三件法宝既是福也是祸，一方面让 C++语言功能强大，另一方面则意味着有更多的东西需要学习。

本章首先介绍 C++的背景，然后介绍创建 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，然后介绍它对泛型编程的支持（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 种不同的编程方式：C 语言代表的过程性语言、C++在 C 语言基础上添加的类代表的面向对象语言、C++模板支持的泛型编程。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、运算符、控制结构和语法规则。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 知识以及泛型编程知识，但无需摈弃任何编程习惯。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 20 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，比当今智能手机的内存都少得多。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编程模式的成长：面向对象编程（OOP）。SmallTalk 和 C++语言具备这种功能。下面更深入地介绍 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以提供终端式界面、提供管理窗口和鼠标的图形界面以及运行程序）。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。然而，汇编语言是低级（low-level）语言，即直接操作硬件，如直接访问 CPU 寄存器和内存单元。因此汇编语言针对于特定的计算机处理器，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。

然而，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）。

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（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。

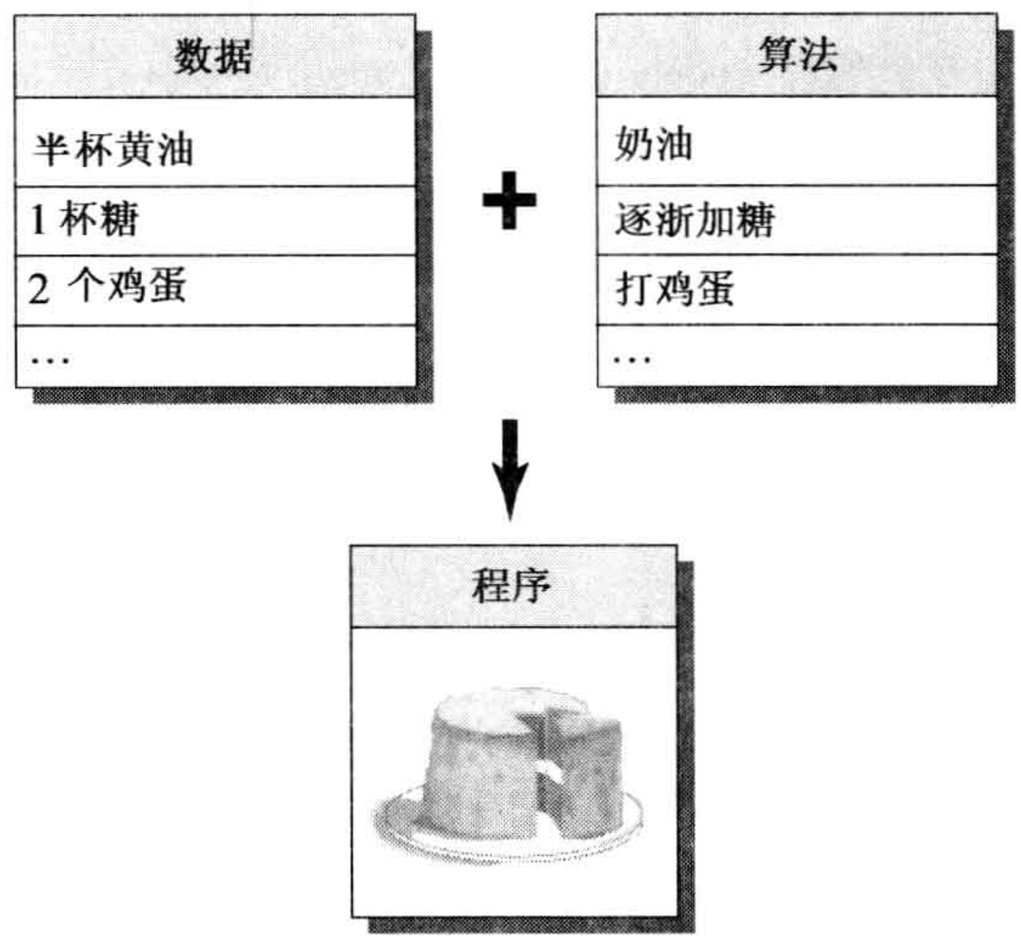


图 1.1　数据+算法=程序

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、$925 000、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、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。多态让您能够为运算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。有时不一定使用自上向下的编程方法，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示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++和泛型编程

泛型编程（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，而泛型编程强调的是独立于特定数据类型。它们的侧重点不同。OOP 是一个管理大型项目的工具，而泛型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。术语泛型（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、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。泛型编程需要对语言进行扩展，以便可以只编写一个泛型（即不是特定类型的）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

与 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。您可以访问 Stroustrup 的网站，阅读 FAQ 部分并下载.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 的泛型编程支持，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 语言中的递增运算符++，该运算符将变量加 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，C++是在 C 语言的基础上添加 OOP 特性，您可以忽略 C++的面向对象特性，但将错过很多有用的东西。

在 C++获得一定程度的成功后，Stroustrup 才添加了模板，这使得进行泛型编程成为可能。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、泛型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。

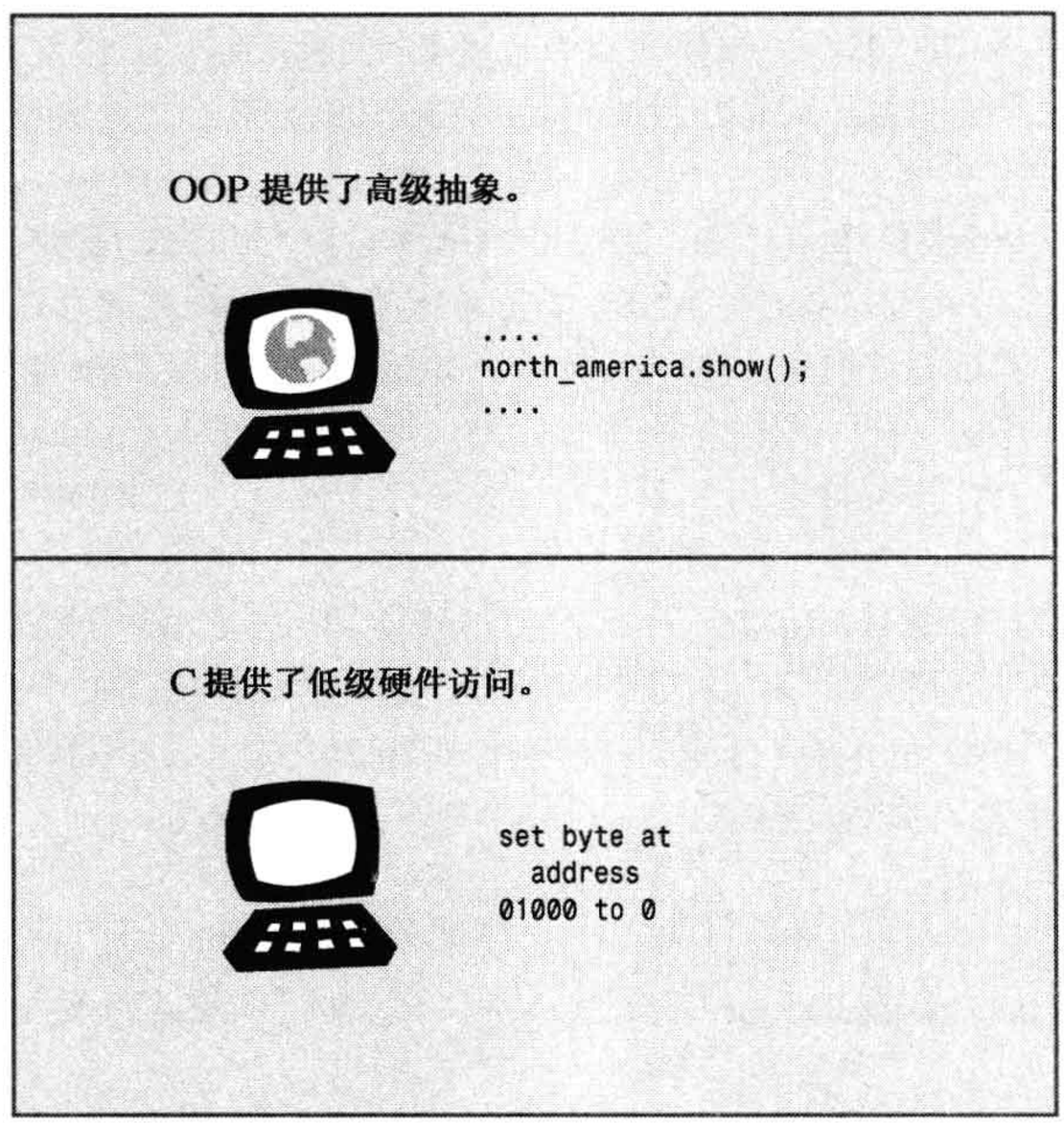


图 1.2　C++的二重性

1.3　可移植性和标准

假设您为运行 Windows 2000 的老式奔腾 PC 编写了一个很好用的 C++程序，而管理人员决定用使用不同操作系统（如 Mac OS X 或 Linux）和处理器（如 SPARC 处理器）的计算机替换它。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 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，虽然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。

经过多年的努力，制定出了一个国际标准 ISO/IEC 14882:1998，并于 1998 年获得了 ISO、IEC（International Electrotechnical Committee，国际电工技术委员会）和 ANSI 的批准。该标准常被称为 C++98，它不仅描述了已有的 C++特性，还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（STL）。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。这个版本常被称为 C++03。由于 C++03 没有改变语言特性，因此我们使用 C++98 表示 C++98/C++2003。

C++在不断发展。ISO 标准委员会于 2001 年 8 月批准了新标准 ISO/IEC 14882:2011，该标准以前称为

C++11。与 C++98 一样，C++11 也新增了众多特性。另外，其目标是消除不一致性，让 C++学习和使用起来更容易。该标准还曾被称为 C++0x，最初预期 x 为 7 或 8，但标准制定工作是一个令人疲惫的缓慢过程。所幸的是，可将 0x 视为十六进制数，这意味着委员会只需在 2015 年前完成这项任务即可。根据这个度量标准，委员会还是提前完成了任务。

ISO C++标准还吸收了 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，如函数原型和类型限定符 const。

在 ANSI C 出现之前，C 语言社区遵循一种事实标准，该标准基于 Kernighan 和 Ritchie 编写的《The C Programming Language》一书，通常被称为 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；本书将其称为标准 C 库或标准库。另外，ANSI/ISO C++标准还提供了一个 C++标准类库。

最新的 C 标准为 C99，ISO 和 ANSI 分别于 1999 年和 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。

1.3.1 C++的发展

Stroustrup 编写的《The Programming Language》包含 65 页的参考手册，它成了最初的 C++事实标准。

下一个事实标准是 Ellis 和 Stroustrup 编写的《The Annotated C++ Reference Manual》。

C++98 标准新增了大量特性，其篇幅将近 800 页，且包含的说明很少。

C++11 标准的篇幅长达 1350 页，对旧标准做了大量的补充。

1.3.2 本书遵循的 C++标准

当代的编译器都对 C++98 提供了很好的支持。编写本书期间，有些编译器还支持一些 C++特性；随着新标准获批，对这些特性的支持将很快得到提高。本书反映了当前的情形，详尽地介绍了 C++98，并涵盖了 C++11 新增的一些特性。在探讨相关的 C++98 主题时顺便介绍了一些 C++新特性，而第 18 章专门介绍新特性，它总结了本书前面提到的一些特性，并介绍了其他特性。

在编写本书期间，对 C++11 的支持还不全面，因此难以全面介绍 C++11 新增的所有特性。考虑到篇幅限制，即使这个新标准获得了全面支持，也无法在一本书中全面介绍它。本书重点介绍大多数编译器都支持的特性，并简要地总结其他特性。

详细介绍 C++之前，先介绍一些有关程序创建的基本知识。

1.4 程序创建的技巧

假设您编写了一个 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，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。

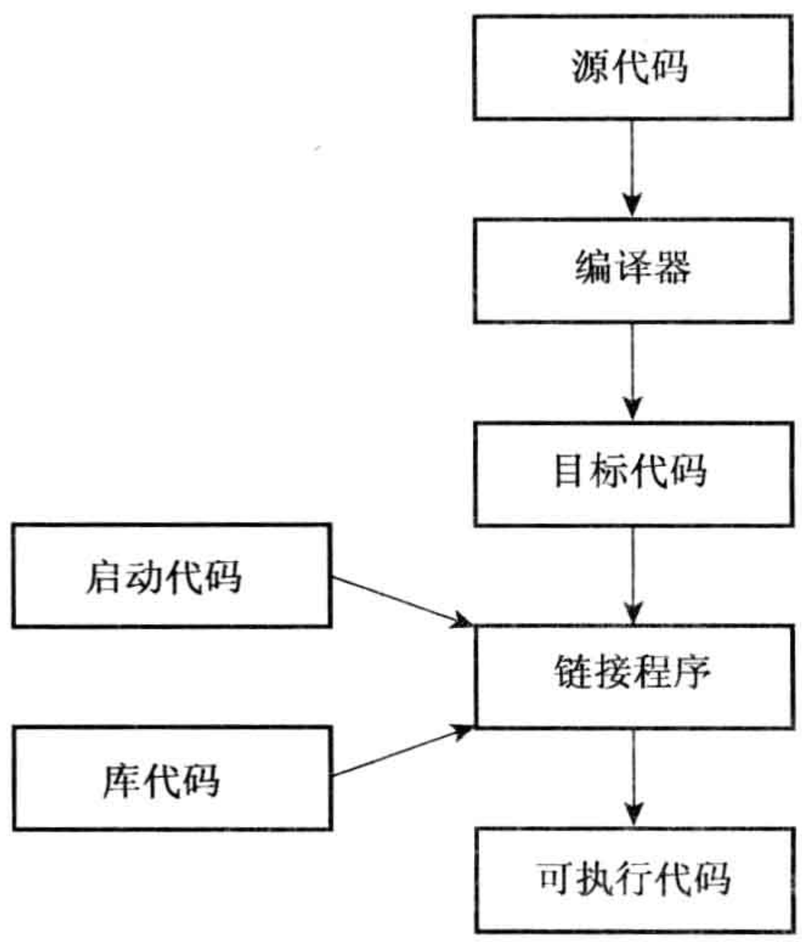


图 1.3　编程步骤

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 C++98 的系统中运行；但第 18 章的程序要求系统支持 C++11。编写本书期间，有些编译器要求您使用特定的标记，让其支持部分 C++11 特性。例如，从 4.3 版起，g++要求您编译源代码文件时使用标记-std=c++0x：

```
g++ -std=c++0x use_auto.cpp
```

创建程序的步骤可能各不相同，下面深入介绍这些步骤。

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++、Embarcadero C++ Builder、Apple Xcode、Open Watcom C++、Digital Mars C++和 Freescale 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（如用于 UNIX 和 Linux 的 GNU C++、用于 AIX 的 IBM XL C/C++、Embarcadero 分发的 Borland 5.5 免费版本以及 Digital Mars 编译器）只能处理编译和链接阶段，要求在系统命令行输入命令。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；在以命令提示符模式运行的 Windows 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 ASCII 文本文件（而不是特殊的字处理器格式），甚至可以使用字处理器。另外，还可能有 IDE 选项，让您能够使用这些命令行编译器。

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，还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）。

使用什么扩展名取决于 C++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。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

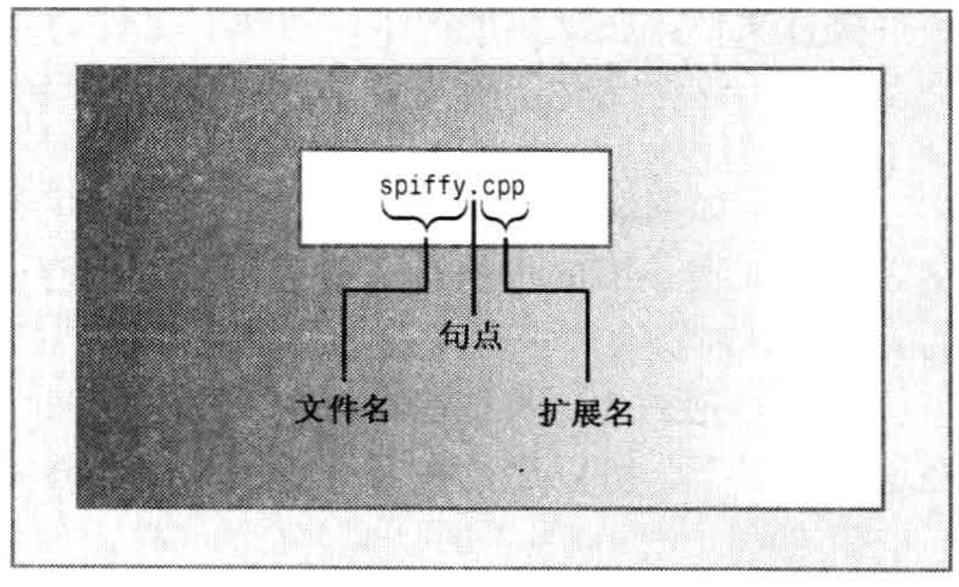


图 1.4　源文件的扩展名

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 源代码文件的扩展名

C++实现	源代码文件的扩展名
UNIX	C、cc、cxx、c
GNU C++	C、cc、cxx、cpp、c++
Digital Mars	cpp、cxx
Borland C++	cpp
Watcom	cpp
Microsoft Visual C++	cpp、cxx、cc
Freestyle CodeWarrior	cp、cpp、cc、cxx、c++

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。

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。

1. UNIX 编译和链接

最初，UNIX 命令 CC 调用 cfront，但 cfront 未能紧跟 C++的发展步伐，其最后一个版本发布于 1993 年。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 提示符下输入如下命令：

```
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）。然而，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～第 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，但并不一定总会安装它。g++编译器的工作方式很像标准 UNIX 编译器。例如，下面的命令将生成可执行文件 a.out

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

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
```

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 命令提示符模式（在这种模式下，将打开一个类似于 MS-DOS 的窗口）下运行的免费命令行编译器。Cygwin 和 MinGW 都包含编译器 GNU C++，且可免费下载；它们使用的编译器名为 g++。

要使用 g++编译器，首先需要打开一个命令提示符窗口。启动程序 Cygwin 和 MinGW 时，它们将自动为您打开一个命令提示符窗口。要编译名为 great.cpp 的源代码文件，请在提示符下输入如下命令：

```
g++ great.cpp
```

如果程序编译成功，则得到的可执行文件名为 a.exe。

4. Windows 编译器

Windows 产品很多且修订频繁，无法对它们分别进行介绍。当前，最流行是 Microsoft Visual C++ 2010，可通过免费的 Microsoft Visual C++ 2010 学习版获得。虽然设计和目标不同，但大多数基于 Windows 的 C++编译器都有一些相同的功能。

通常，必须为程序创建一个项目，并将组成程序的一个或多个文件添加到该项目中。每个厂商提供的 IDE（集成开发环境）都包含用于创建项目的菜单选项（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。其中一些可能既有 32 位版本，又有 64 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，应让程序以字符模式运行。具体选项取决于编译器。一般而言，应选择包含字样“控制台”、“字符模式”或“DOS 可执行文件”等选项。例如，在 Microsoft Visual C++ 2010 中，应选择 Win32 Console Application（控制台应用程序）选项，单击 Application Settings（应用程序设置），并选择 Empty Project（空项目）。在 C++ Builder 中，应从 C++ Builder Projects（C++ Builder 项目）中选择 Console Application（控制台应用程序）。

创建好项目后，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）、Run（运行）和 Debug（调试），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 通常意味着编译项目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，如果项目包含 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。
- Debug 意味着以步进方式执行程序。
- 编译器可能让您选择要生成调试版还是发布版。调试版包含额外的代码，这会增大程序、降低执行速度，但可提供详细的调试信息。

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。与几年前相比，现在的编译器更严格地遵循了 C++标准。另外，编译器通常提供了可用于控制严格程度的选项。

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 不关闭。如果编译器关闭窗口，将难以看到程序输出，除非您眼疾手快、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 键之前，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。

5. Macintosh 上的 C++

当前，Apple 随操作系统 Mac OS X 提供了开发框架 Xcode，该框架是免费的，但通常不会自动安装。要安装它，可使用操作系统安装盘，也可从 Apple 网站免费下载（但需要注意的是，它超过 4GB）。Xcode 不仅提供了支持多种语言的 IDE，还自带了两个命令行编译器（g++和 clang），可在 UNIX 模式下运行它们。而要进入 UNIX 模式，可通过实用程序 Terminal。

提示：为节省时间，可对所有示例程序使用同一个项目。方法是从项目列表中删除前一个示例程序的源代码文件，并添加当前的源代码。这样可节省时间、工作量和磁盘空间。

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泛型编程的支持，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++的流行导致大量用于各种计算平台的 C++实现得以面世；而 ISOC++标准（C++98/03 和 C++11）为确保众多实现的相互兼容提供了基础。这些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，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，如项目管理和调试功能。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：

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

要建造简单的房屋，首先要打地基、搭框架。如果一开始没有牢固的结构，后面就很难建造窗子、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。本章对 C++程序的基本结构做一概述，并预览后面将介绍的主题，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示例中相同。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;  // more output
```

```
    return 0;                                   // terminate main()
}                                               // end of function body
```

程序调整

要在自己的系统上运行本书的示例，可能需要对其进行修改。有些窗口环境在独立的窗口中运行程序，并在程序运行完毕后自动关闭该窗口。正如第 1 章讨论的，要让窗口一直打开，直到您按任何键，可在 return 语句前添加如下语句：

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

对于有些程序，要让窗口一直打开，直到您按任何键，必须添加两条这样的语句。第 4 章将更详细地介绍 cin.get( )。

如果您使用的系统很旧，它可能不支持 C++98 新增的特性。

有些程序要求编译器对 C++11 标准提供一定的支持。对于这样的程序，将明确的指出这一点，并在可能的情况下提供非 C++11 代码。

将该程序复制到您选择的编辑器中（或使用本书配套网站的源代码，详情请参阅封底）后，便可以 C++编译器创建可执行代码了（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 语言输入和输出

如果已经使用过 C 语言进行编程，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。

您使用函数来创建 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 中的示例非常简单，只包含一个名为 main( )的函数。myfirst.cpp 示例包含下述元素。

- 注释，由前缀//标识。
- 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 return 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 中的示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。图 2.1 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。所有的语句都以分号结束，因此在输入示例代码时，请不要省略分号。

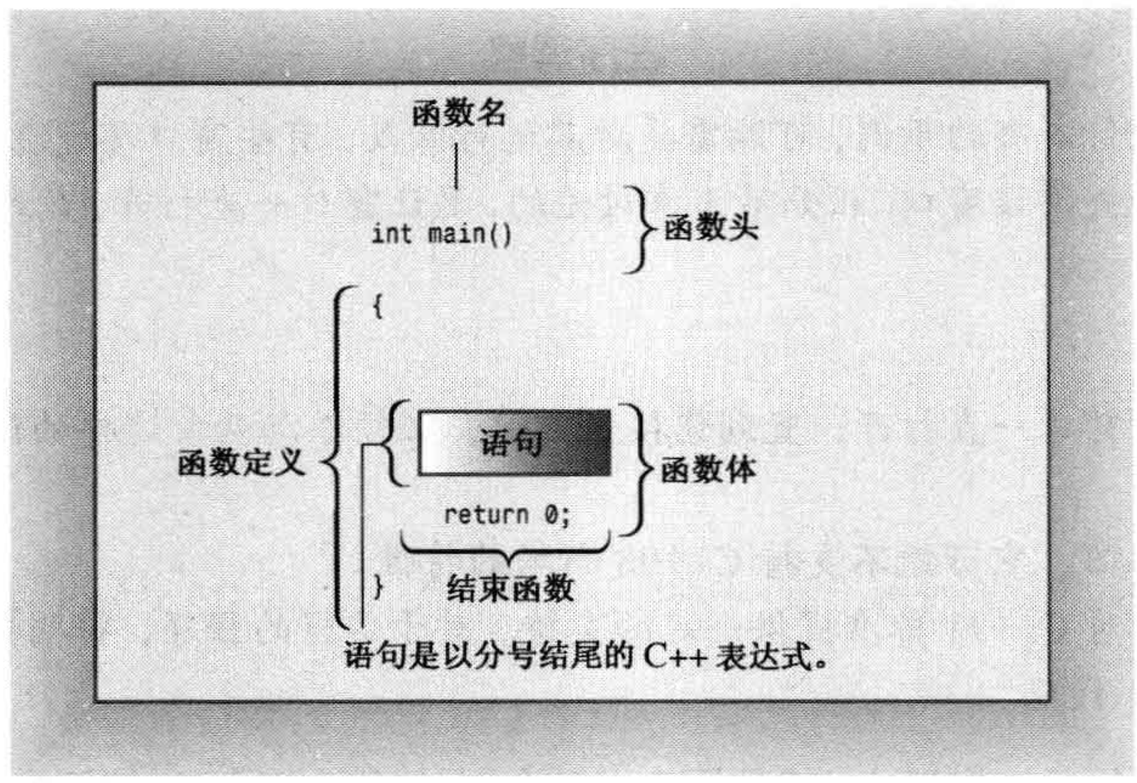


图 2.1 main( )函数

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

语句是要执行的操作。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，这种情况下，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，然而，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，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( )。

然而，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 7 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( )的接口描述，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：

```
int main()
```

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。然而，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( )。有些编程环境提供一个框架程序，该程序调用一些非标准函数，如_tmain( )。在这种情况下，有一个隐藏的 main( )，它调用_tmain( )。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
```

本书所有的程序都以注释开始，这些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该程序的情况下。

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;
```

可使用其他代码替换第 2 行，这里使用这行代码旨在简化该程序（如果编译器不接受这几行代码，则说明它没有遵守标准 C++98，使用它来编译本书的示例时，将出现众多其他的问题）。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，也使用一个预处理器，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，而只讨论这样的预处理器，即它处理名称以#开头的编译指令）。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 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。

注意：使用 cin 和 cout 进行输入和输出的程序必须包含文件 iostream。

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，头文件 math.h 支持各种 C 语言数学函数，但 C++的用法变了。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。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	示　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，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.hpp 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。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）。但这里还是简要地介绍它，以免您一头雾水。

名称空间支持是一项 C++特性，旨在让您编写大型程序以及将多个厂商现有的代码组合起来的程序时更容易，它还有助于组织程序。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，可以省略编译指令 using，以下述方式进行编码：

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

然而，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，在大型项目中一个潜在的问题。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，便可以使用 cin 和 cout，而不必加上 std::前缀：

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

然而，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，即如何使用它。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。然而，现在来看看 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运算符（<<），它可以将其右侧的信息插入到流中。请看下面的语句（注意结尾的分号）：

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

它将字符串“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）。

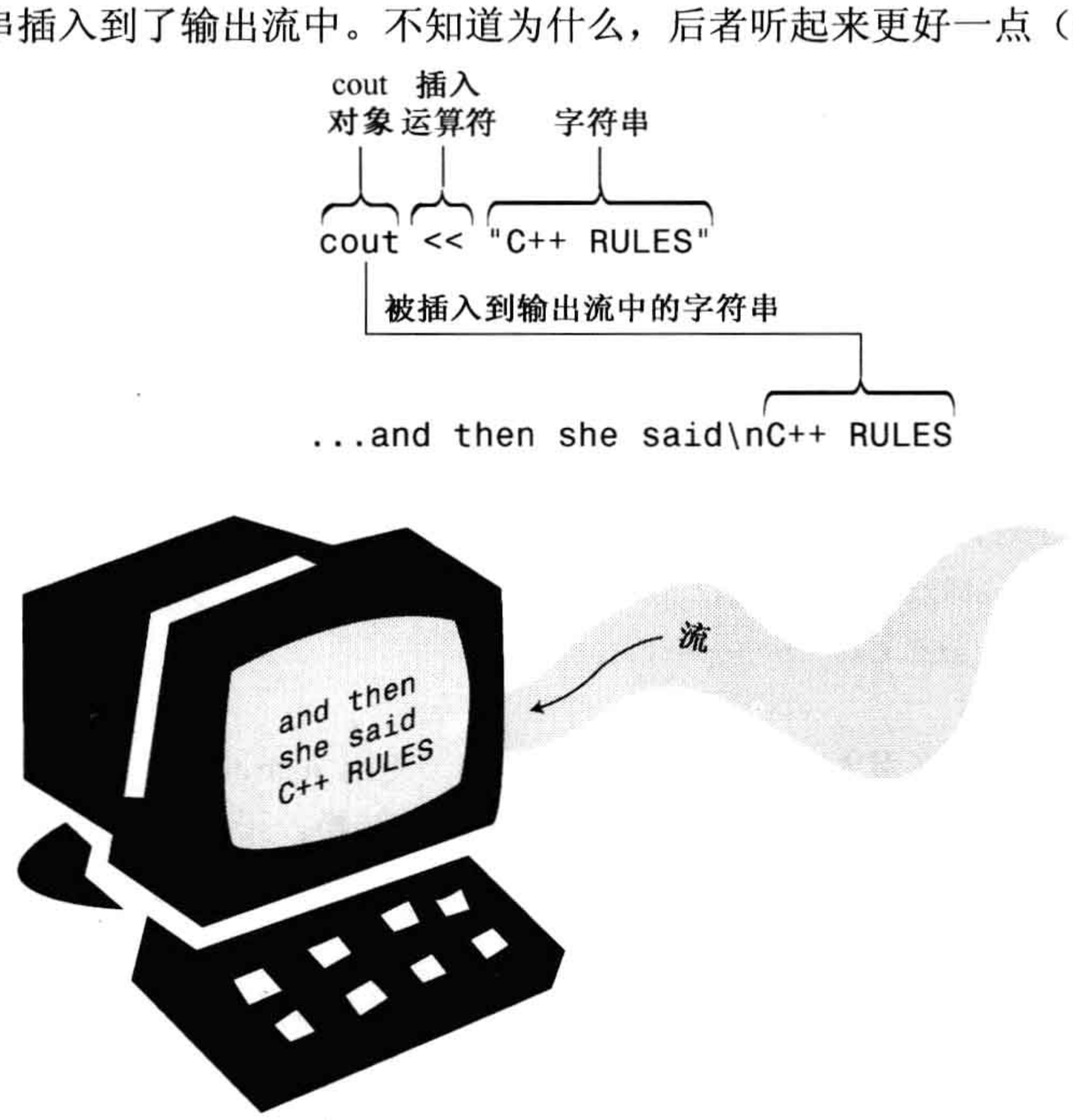


图 2.2 使用 cout 显示字符串

初识运算符重载

如果熟悉 C 后才开始学习 C++，则可能注意到了，插入运算符（<<）看上去就像按位左移运算符（<<），这是一个运算符重载的例子，通过重载，同一个运算符将有不同的含义。编译器通过上下文来确定运算符的含义。C 本身也有一些运算符重载的情况。例如，&符号既表示地址运算符，又表示按位 AND 运算符；

* 既表示乘法，又表示对指针解除引用。这里重要的不是这些运算符的具体功能，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++扩展了运算符重载的概念，允许为用户定义的类型（类）重新定义运算符的含义。

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开头。诸如 endl 等对于 cout 来说有特殊含义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，要尝试上述输出示例，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 << "Pluto is a dwarf planet.\n";          // show text, go to next line
cout << "Pluto is a dwarf planet." << endl;    // show text, go 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。一个差别是，endl 确保程序继续运行前刷新输出（将其立即显示在屏幕上）；而使用“\n”不能提供这样的保证，这意味着在有些系统中，有时可能在您输入信息后才会出现提示。

换行符是一种被称为“转义序列”的按键组合，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，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。然而，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
```

然而，C++11 新增的原始（raw）字符串可包含回车，这将在第 4 章简要地讨论。

1. 源代码中的标记和空白

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。下面的一些示例说明了什么情况下可以使用空白，什么情况下可以省略：

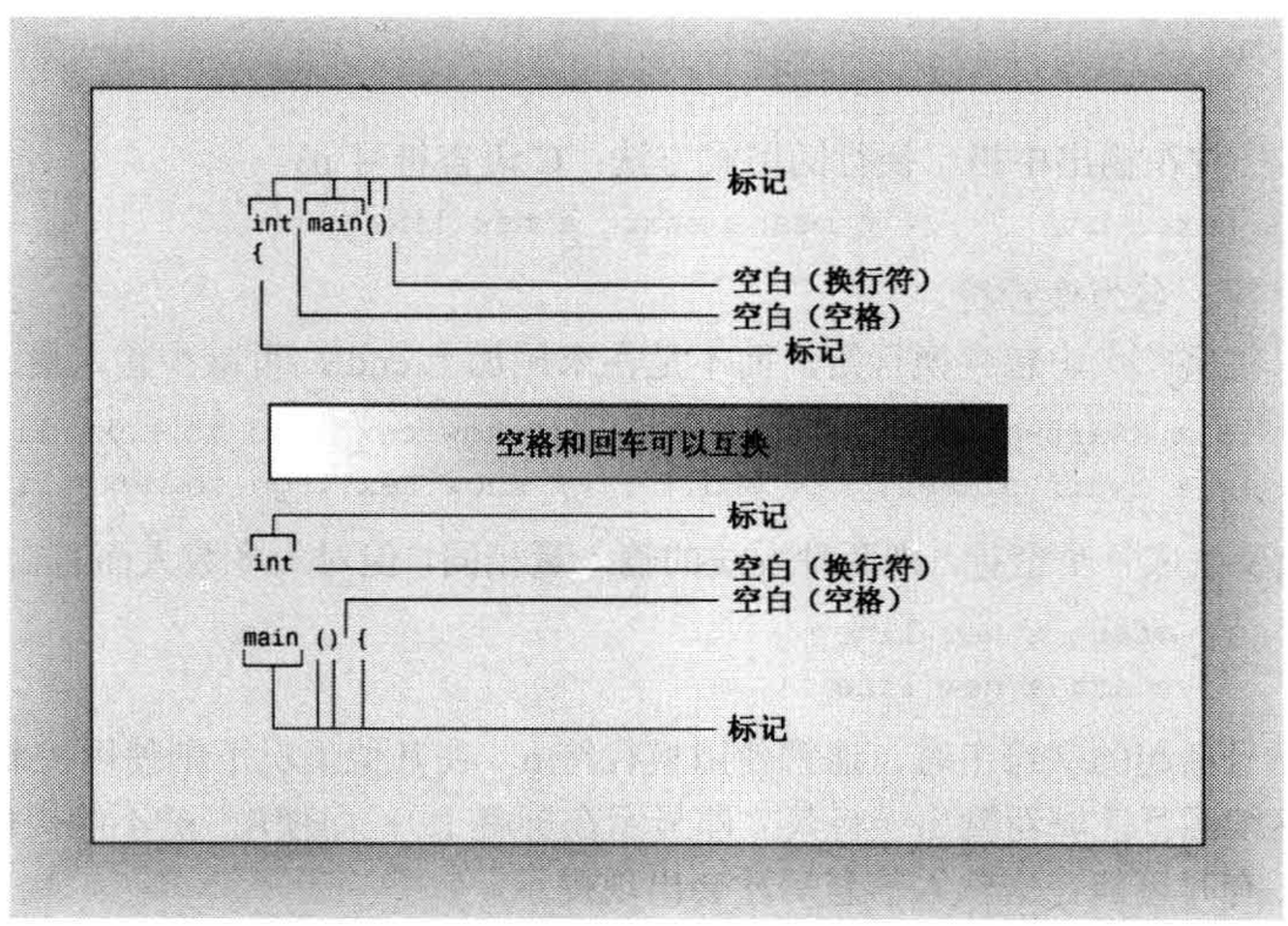


图 2.3 标记和空白

```
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 in ()
int main ( )        // ALSO VALID, white space used in ( )
```

2. C++源代码风格

虽然 C++在格式方面赋予了您很大的自由，但如果遵循合理的风格，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 所示的风格，它遵循了下述规则。

- 每条语句占一行。
- 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。

前三条规则旨在确保代码清晰易读；第四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;
```

这条语句提供了两项信息：需要的内存以及该内存单元的名称。具体地说，这条语句指出程序需要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。

完成的第二项任务是给存储单元指定名称。在这里，该声明语句指出，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，ss 程序员可能编写如下语句：

```
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）。

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。

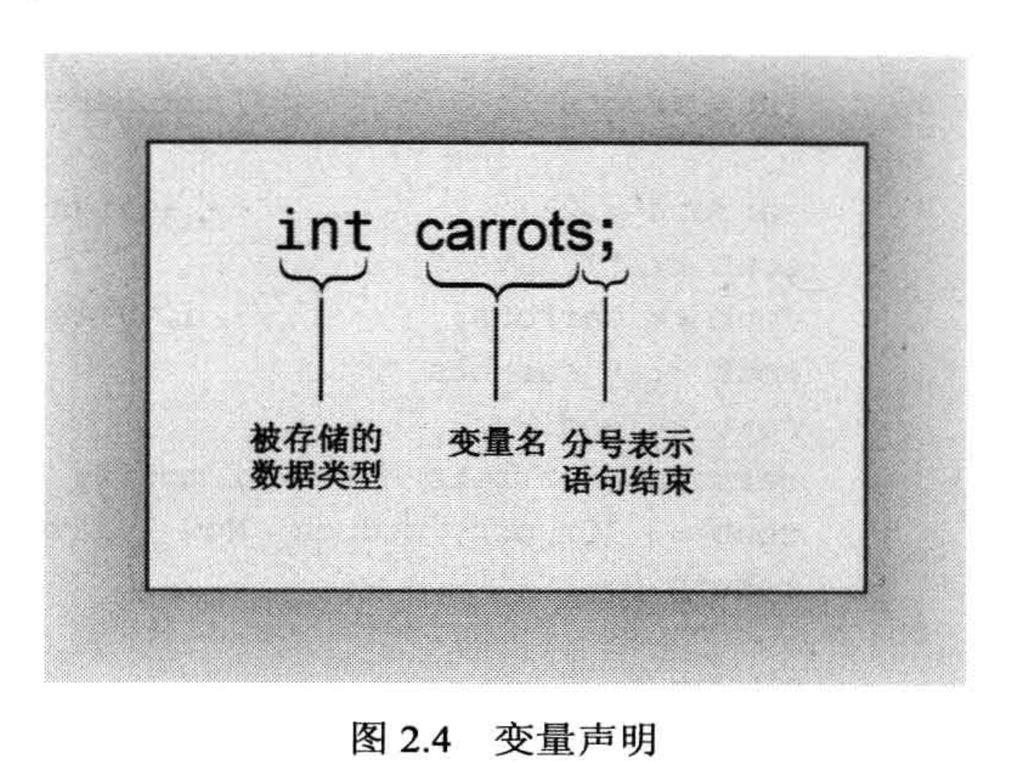


图 2.4 变量声明

如果您熟悉 C 语言或 Pascal，就一定熟悉变量声明。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，下面的语句将整数 25 赋给变量 carrots 表示的内存单元：

```
carrots = 25;
```

符号=叫做赋值运算符。C++（和 C）有一项不寻常的特性——可以连续使用赋值运算符。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
```

赋值运算符右边的表达式 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，赋值运算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，本章的示例都使用 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 3 和 8。程序在内部存储的是字符 3 和字符 8 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 语言的多功能输出函数 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++插入运算符（<<）将根据其后的数据类型相应地调整其行为，这是一个运算符重载的例子。在后面的章节中学习函数重载和运算符重载时，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<<运算符，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;
}
```

程序调整

如果您发现在以前的程序清单中需要添加 cin.get( )，则在这个程序清单中，需要添加两条 cin.get( )语句，这样才能在屏幕上看到输出。第一条 cin.get( )语句在您输入数字并按 Enter 键时读取输入，而第二条 cin.get( )语句让程序暂停，直到您按 Enter 键。

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四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，<<运算符将字符串插入到输出流中；输入时，cin 使用>>运算符从输入流中抽取字符。通常，需要在运算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<<运算符，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 语句之前有其他语句，情况也将如此。

2.3.3　类简介

看了足够多的 cin 和 cout 示例后，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。

注意：类描述了一种数据类型的全部属性（包括可使用它执行的操作），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，而是语言标准指定的类。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唯一两组类定义。然而，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运算符，cin 和 cout 采用的就是这种方式。因此，下面的语句使用重新定义的<<运算符将“显示消息”发送给 cout：

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

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）。

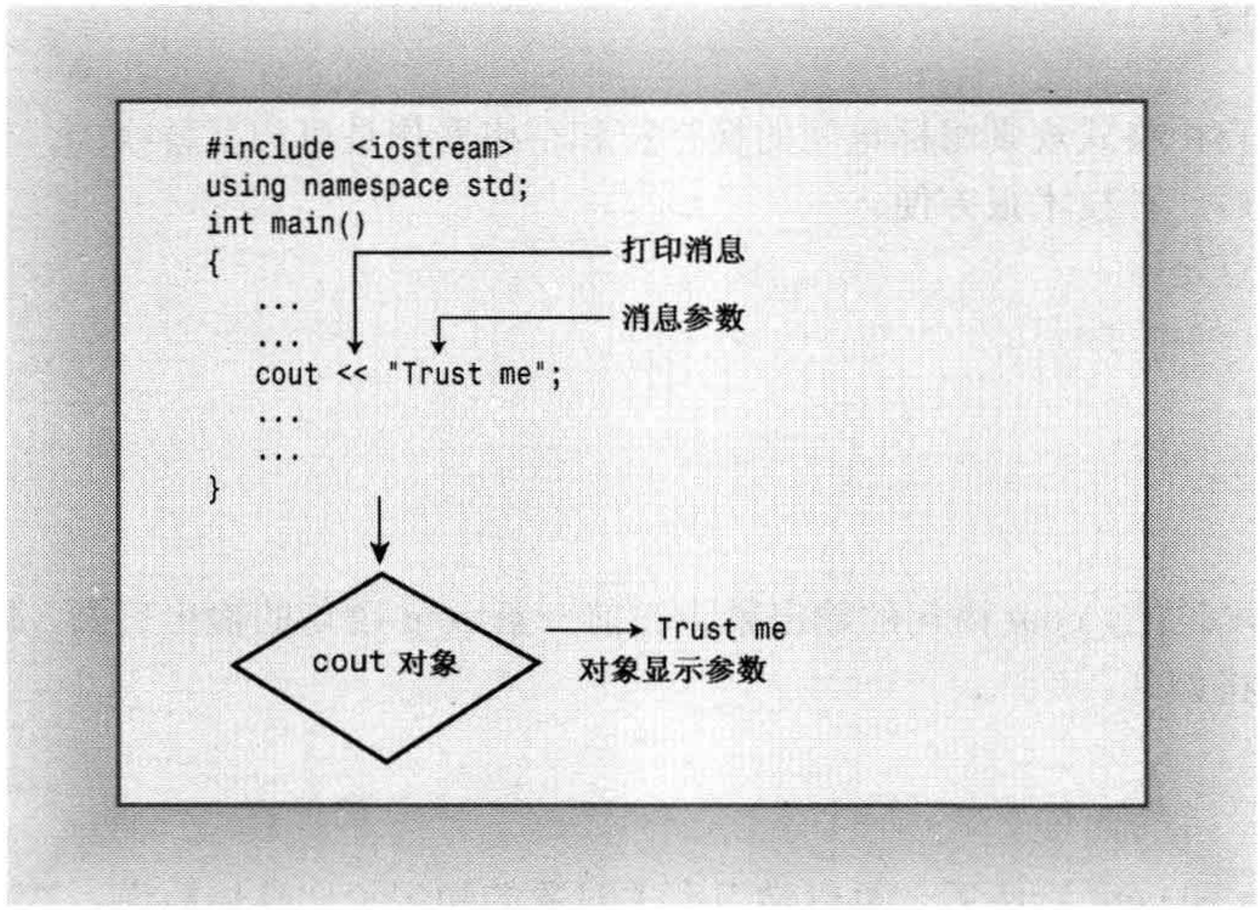


图 2.5 向对象发送消息

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。然而，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值，而这个值可赋给变量或在其他表达式中使用。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）。

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。

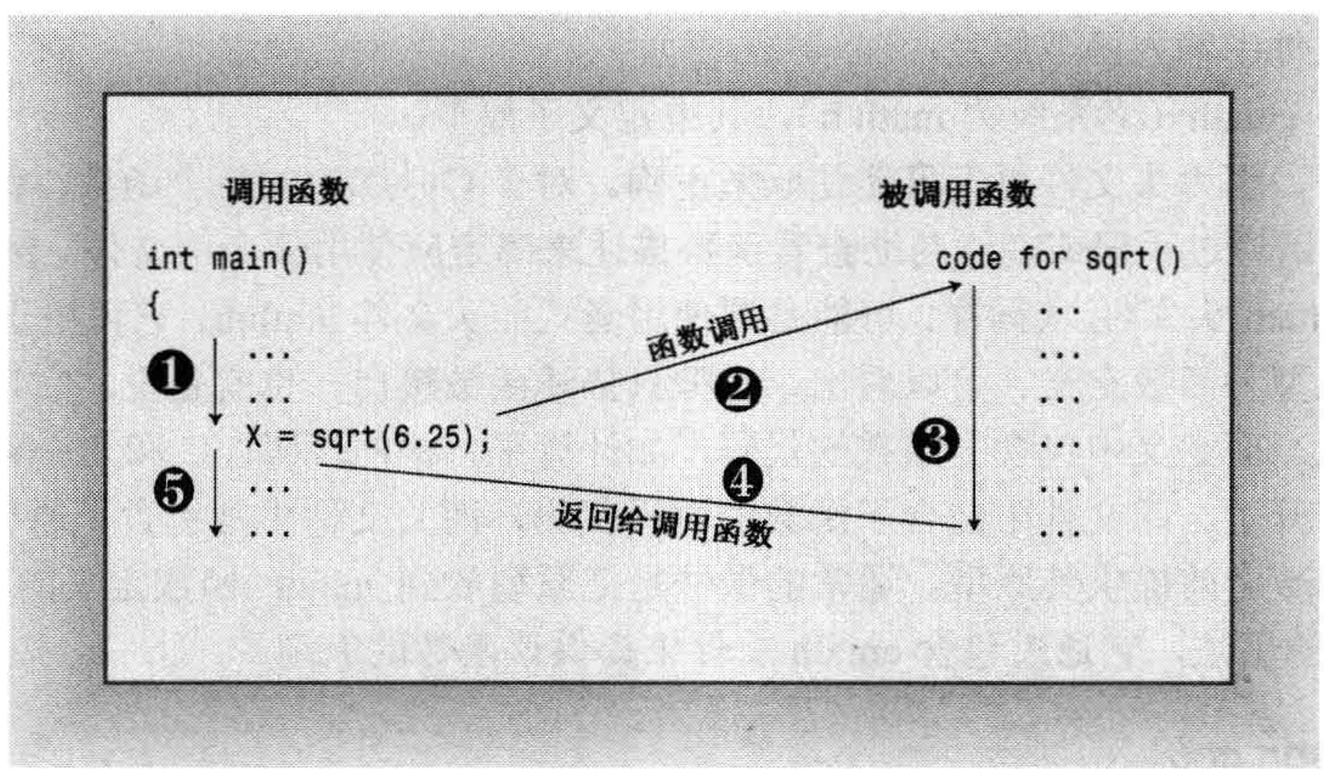


图 2.6　调用函数

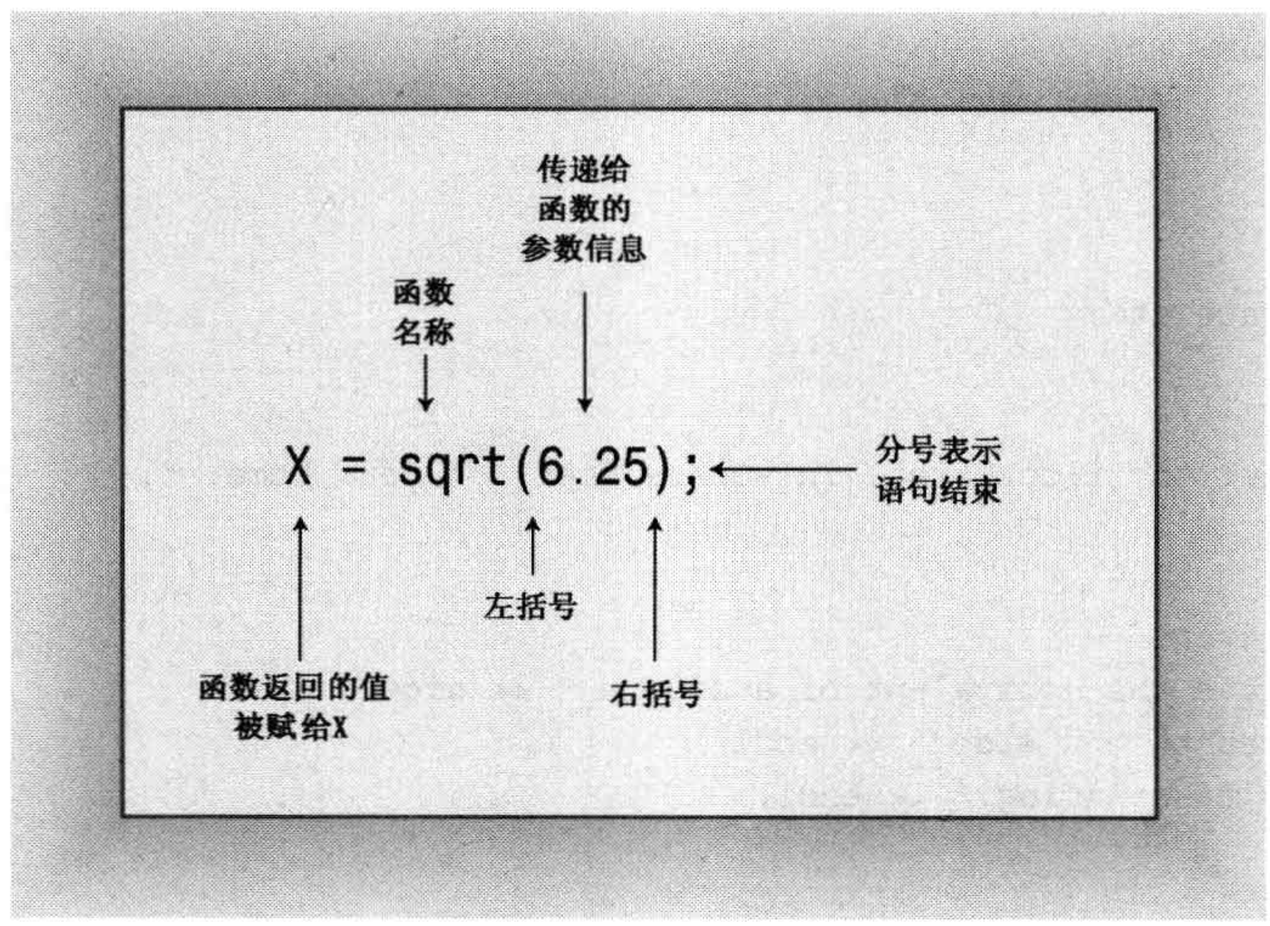


图 2.7　函数调用的句法

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。

注意：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);
```

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 " << side
         << " 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，将得到一条消息，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）。如果在 UNIX 实现中遇到这样的消息，可能需要在命令行结尾使用-lm 选项：

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

在 Linux 系统中，有些版本的 Gnu 编译器与此类似：

```
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 1 536.0 和 39.191 8。将看起来是整数（如 1536）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 章更详细地介绍。

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，因此不需要返回值。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。但现在您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 statements
```

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）。

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
```

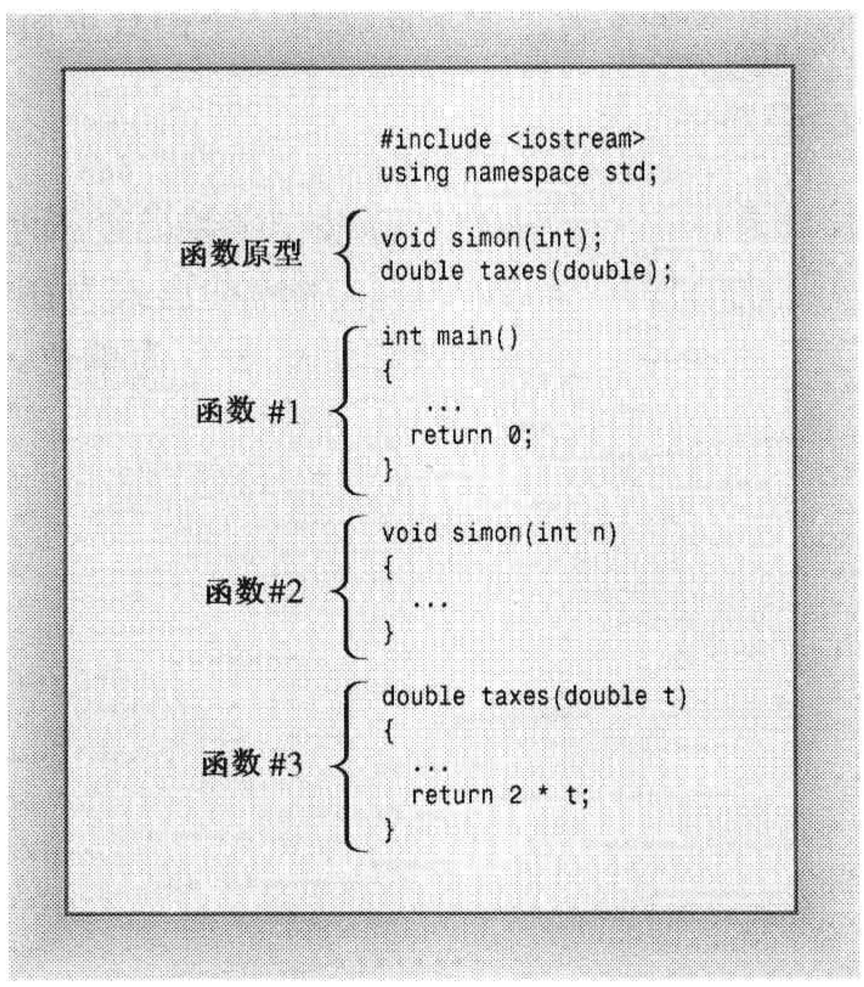


图 2.8　函数定义在文件中依次出现

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，下面的函数调用将 3 赋给 simon( )函数头中定义的变量 n：

```
simon(3);
```

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。在示例运行中，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。

下面复习一下 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，可以将计算机操作系统（如 UNIX 或 Windows）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 Windows 命令行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。然而，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，其他函数名和对象名也都不能是关键字。然而，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: 15
15 stone = 210 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 210），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;
}
```

这两个版本返回的结果相同，但第二个版本更容易理解和修改，因为它将计算和返回分开了。

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）。

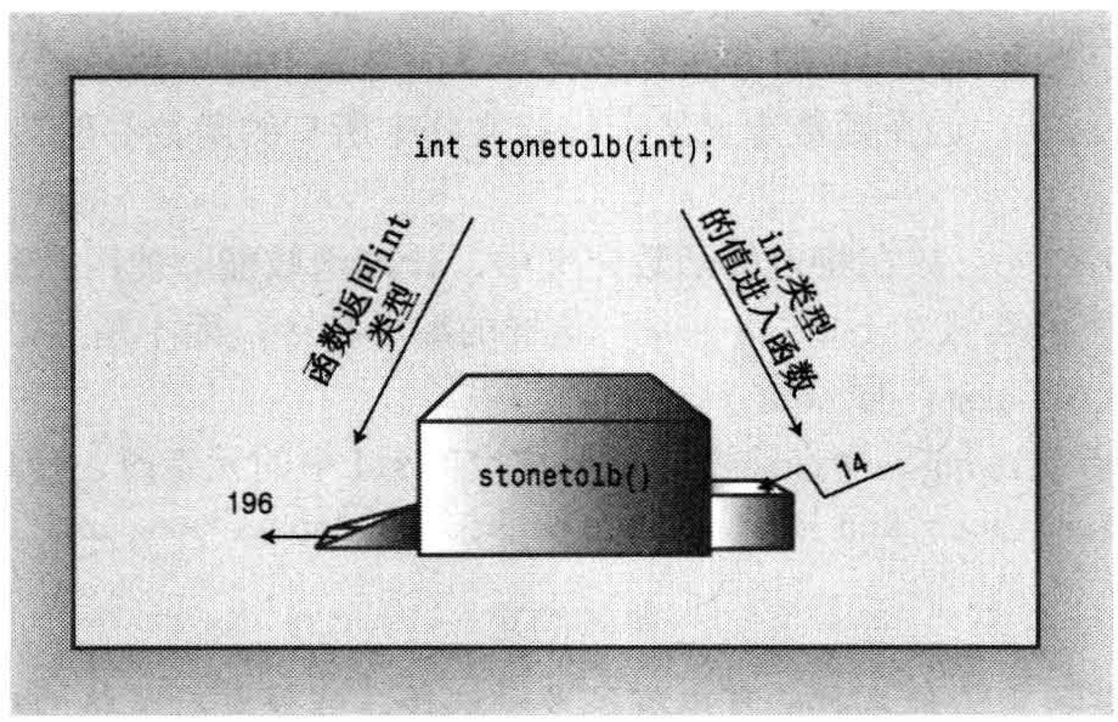


图 2.9　函数原型和作为黑盒的函数

函数 stonetolb( )短小、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 4 种。

- 将 `using namespace std;` 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 namespace std;` 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 namespace std;`，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，而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。因此凡是符合第 3 章将介绍的 C++规则的风格都是正确的，都可以根据个人的判断而使用。

撇开语言是否允许不谈，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，包括下述 6 种。

- 声明语句：定义函数中使用的变量的名称和类型。
- 赋值语句：使用赋值运算符（=）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 类定义的插入运算符（<<）使得将数据插入到输出流成为可能；为 istream 类定义的抽取运算符（>>）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,”，其中 X 为变量 cheeses 的当前值。

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？

11．假设您编写的 main( )函数包含如下代码：

```
cout << "Please enter your PIN: ";
```

而编译器指出 cout 是一个未知标识符。导致这种问题的原因很可能是什么？指出 3 种修复这种问题的方法。

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 long 为单位的距离，然后将它转换为码（一 long 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，让用户输入其年龄，然后显示该年龄包含多少个月，如下所示：

```
Enter your age: 29
```

5．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

6. 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 150000000 公里或 93000000 英里），光年是光一年走的距离（约 10 万亿公里或 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=63240 天文单位

7. 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。
- C++11 新增的整型：unsigned long long 和 long long。
- 表示各种整型的系统限制的 climits 文件。
- 各种整型的数字字面值（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字面值。
- C++的算术运算符。
- 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。然而，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，即整数和浮点数。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的数据的方法，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；最后，介绍 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 Google 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，本书的示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；然后，您可在程序中使用 braincount 来访问该内存单元。这些语句没有告诉您，这个值将存储在内存的什么位置，但程序确实记录了这种信息。实际上，可以使用&运算符来检索 braincount 的内存地址。下一章介绍另一种标识数据的方法（使用指针）时，将介绍这个运算符。

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，但有些平台有长度限制。

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。

命名方案

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示例（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、long 和 C++11 新增的 long long，其中每种类型都有符号版本和无符号版本，因此总共有 10 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　整型 short、int、long 和 long long

计算机内存由一些叫做位（bit）的单元组成（参见本章后面的旁注“位与字节”）。C++的 short、int、long 和 long long 类型通过使用不同数目的位来存储值，最多能够表示 4 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，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；
- long long 至少 64 位，且至少与 long 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，把 64 位单元设置为 18 446 744 073 709 551 616 个不同的值。作为比较，unsigned long 存储不了地球上当前的人数和银河系的星星数，而 long long 能够。

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 1024 字节，1MB 等于 1024KB。然而，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。然而，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。有些人使用术语八位组（octet）表示 8 位字节。

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；甚至可以是 64 位，因为 long 和 long long 至少长 64 位。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 XP、Windows Vista、Windows 7、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（包括在同一个系统中使用不同编译器）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 4 种类型（int、short、long 和 long 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-32768 到+32767。

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 运算符返回类型或变量的长度，单位为字节（运算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，加号运算符+将两个值相加）。前面说过，“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 climits file
    long n_long = LONG_MAX;
    long long n_llong = LLONG_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;
    cout << "long long is " << sizeof n_llong << " bytes." << endl;
    cout << endl;

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

```
    cout << "long: " << n_long << endl;
    cout << "long long: " << n_llong << endl << endl;

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

注意：如果您的系统不支持类型 long long，应删除使用该类型的代码行。

下面是程序清单 3.1 中程序的输出：

```
int is 4 bytes.
short is 2 bytes.
long is 4 bytes.
long long is 8 bytes.

Maximum values:
int: 2147483647
short: 32767
long: 2147483647
long long: 9223372036854775807

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

这些输出来自运行 64 位 Windows 7 的系统。

我们来看一下该程序的主要编程特性。

1. 运算符 sizeof 和头文件 limits

sizeof 运算符指出，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 运算符。对类型名（如 int）使用 sizeof 运算符时，应将名称放在括号中；但对变量名（如 n_short）使用该运算符，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（参见本章后面的旁注“符号常量——预处理器方式”）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 Windows 7 系统，为 2 147 483 647。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 16 位 int 的老系统中，climits 文件将 INT_MAX 定义为 32 767。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

续表

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

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并替换命令相似。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）。然而，#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。然而，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。

2. *初始化*

初始化将赋值与声明合并在一起。例如，下面的语句声明了变量 n_int，并将 int 的最大取值赋给它：

```
int n_int = INT_MAX;
```

也可以使用字面值常量来初始化。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。

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

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

警告：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，相应内存单元保存的值。

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。

3. *C++11 初始化方式*

还有另一种初始化方式，这种方式用于数组和结构，但在 C++98 中，也可用于单值变量：

```
int hamburgers = {24};  // set hamburgers to 24
```

将大括号初始化器用于单值变量的情形还不多，但 C++11 标准使得这种情形更多了。首先，采用这种方式时，可以使用等号（=），也可以不使用：

```
int emus{7};        // set emus to 5
int rheas = {12};   // set rheas to 12
```

其次，大括号内可以不包含任何东西。在这种情况下，变量将被初始化为零：

```
int rocs = {};      // set rocs to 0
int psychics{};     // set psychics to 0
```

第三，这有助于更好地防范类型转换错误，这个主题将在本章末尾讨论。

为何需要更多的初始化方法？有充分的理由吗？原因是让新手更容易学习 C++，这可能有些奇怪。以前，C++使用不同的方式来初始化不同的类型：初始化类变量的方式不同于初始化常规结构的方式，而初始化常规结构的方式又不同于初始化简单变量的方式；通过使用 C++新增的大括号初始化器，初始化常规变量的方式与初始化类变量的方式更像。C++11 使得可将大括号初始化器用于任何类型（可以使用等号，也可以不使用），这是一种通用的初始化语法。以后，教材可能介绍使用大括号进行初始化的方式，并出于向后兼容的考虑，顺便提及其他初始化方式。

3.1.4　无符号类型

前面介绍的 4 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−32768 到+32767，则无符号版本的表示范围为 0-65535。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
unsigned long long lang_lang;   // unsigned long 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;
}
```

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 32767。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 32767 变成了−32768！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 65535。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。

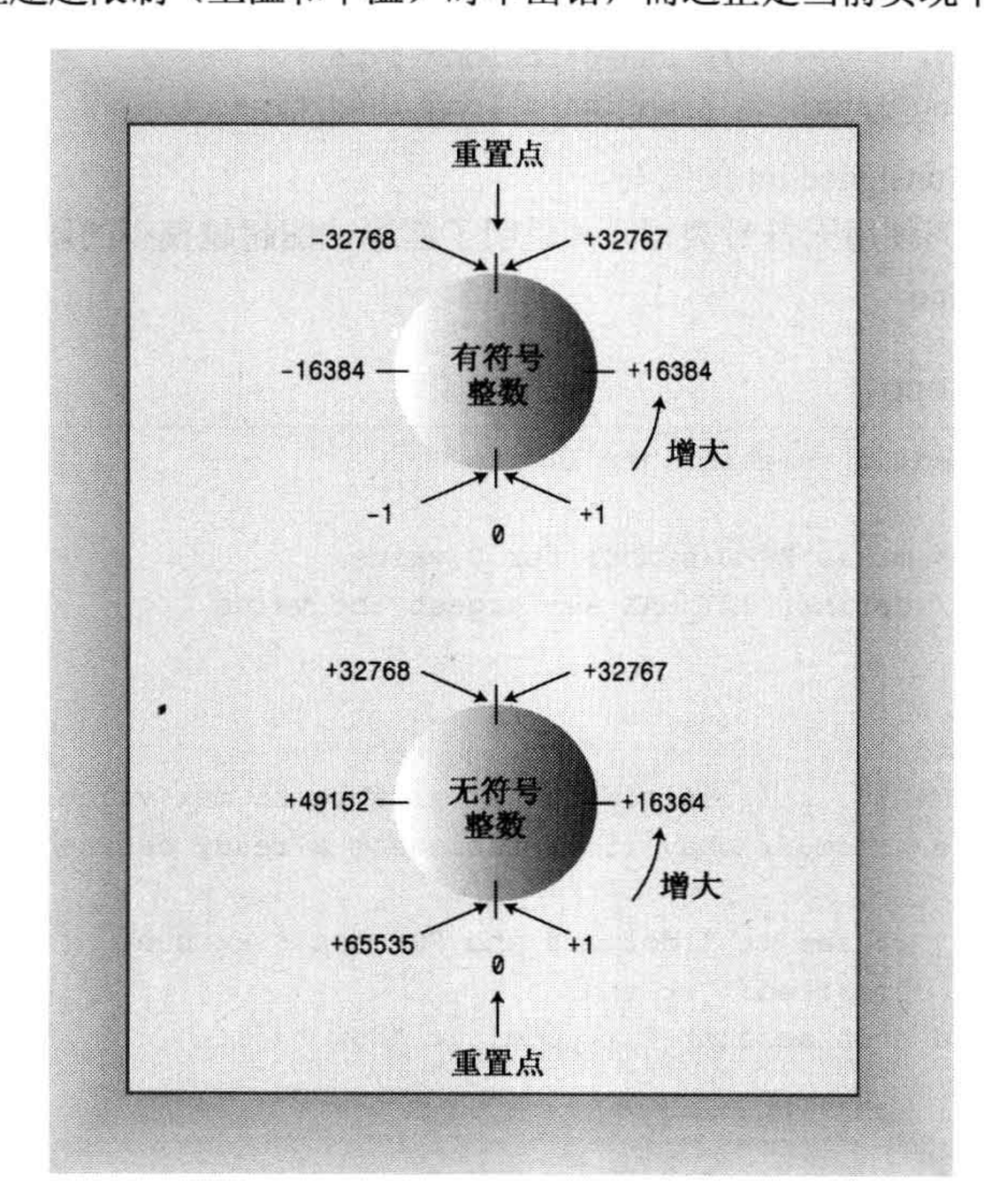


图 3.1　典型的整型溢出行为

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，将程序移植到 16 位系统时，就不会突然无法正常工作（参见图 3.2）。如果要存储的值超过 20 亿，可使用 long long。

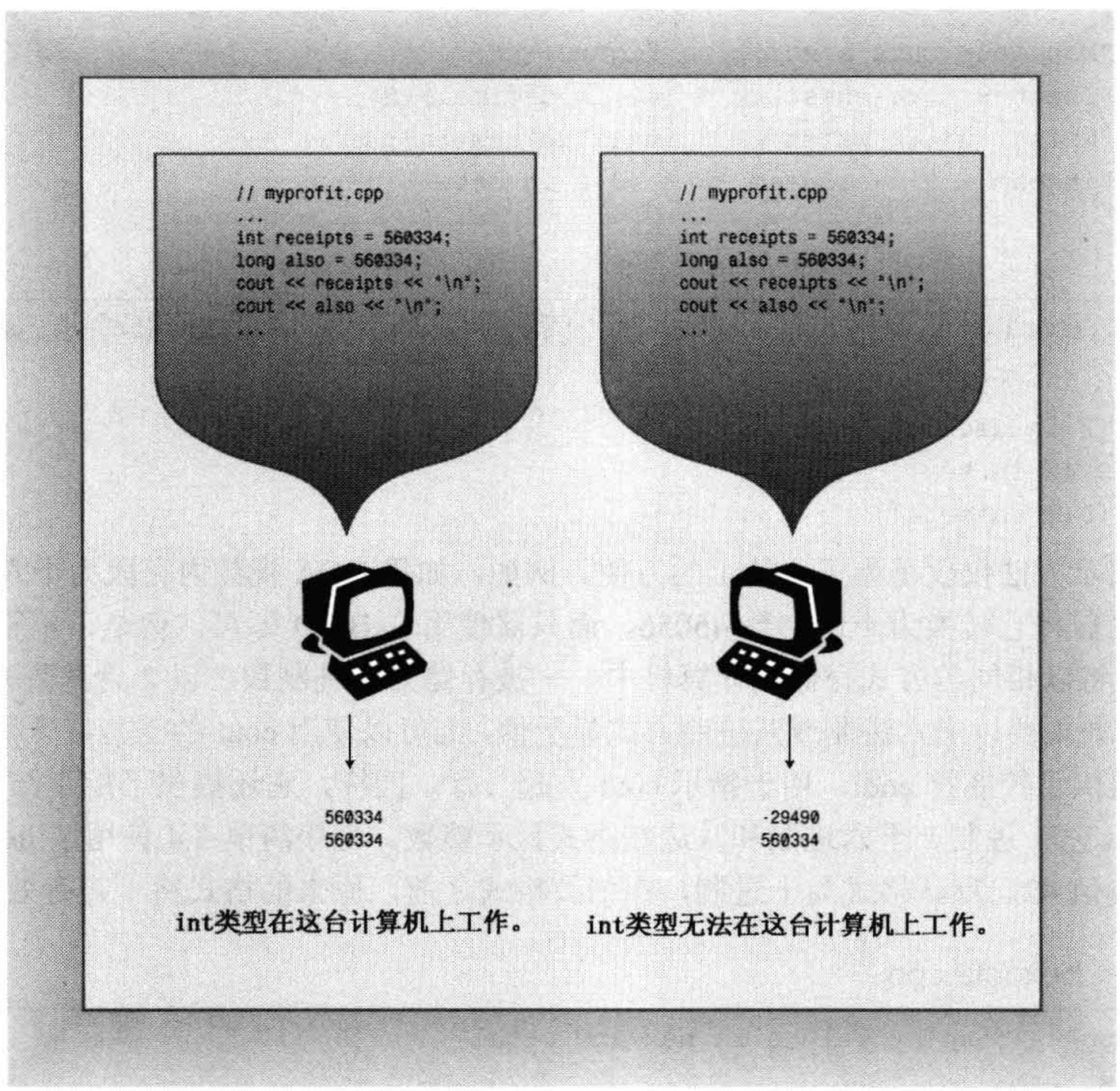


图 3.2　为提高可移植性，请使用 long

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 位的系统移到 int 为 32 位的系统，则用于存储 int 数组的内存量将加倍，但 short 数组不受影响。请记住，节省一点就是赢得一点。

如果只需要一个字节，可使用 char，这将稍后介绍。

3.1.6　整型字面值

整型字面值（常量）是显式地书写的常量，如 212 或 1776。与 C 相同，C++能够以三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（硬件黑客的最爱）。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 15，0xA5 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

```
// hexoct1.cpp -- shows hex and octal literals
#include <iostream>
int main()
{
    using namespace std;
    int chest = 42;     // decimal integer literal
    int waist = 0x42;   // hexadecimal integer literal
    int inseam = 042;   // octal integer literal

    cout << "Monsieur cuts a striking figure!\n";
    cout << "chest = " << chest << " (42 in decimal)\n";
    cout << "waist = " << waist << " (0x42 in hex)\n";
    cout << "inseam = " << inseam << " (042 in octal)\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，则不必在程序中使用之前将它转换为十进制数 45056，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 以上述三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 for 42)" << endl;
    cout << hex;      // manipulator for changing number base
    cout << "waist = " << waist << " (hexadecimal for 42)" << endl;
    cout << oct;      // manipulator for changing number base
    cout << "inseam = " << inseam << " (octal for 42)" << 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 for 42)
waist = 2a (hexadecimal for 42)
inseam = 52 (octal for 42)
```

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。C++11 提供了用于表示类型 long long 的后缀 ll 和 LL，还提供了用于表示类型 unsigned long long 的后缀 ull、Ull、uLL 和 ULL。

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 long 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 long 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、unsigned long、long long 或 unsigned long 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，很多系统支持的字符都不超过 128 个，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，字母 M 的编码为 77。为方便起见，本书在示例中使用的是 ASCII 码。然而，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 << "Hola! ";
    cout << "Thank you for the " << ch << " character." << endl;
    return 0;
}
```

同样，\n 在 C++中表示换行符。下面是该程序的输出：

```
Enter a character:
M
Hola! 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++中书写字符字面值：将字符用单引号括起，如'M'（注意，示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 code for M to ch
    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 ch
    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 ch 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 ch 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 ch 和 i 的值都是 77。接下来，cout 把 ch 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 ch 实际上是一个整数，因此可以对它使用整数操作，如加 1，这将把 ch 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在字符和 ASCII 码之间来回转换。

即使通过键盘输入的数字也被视为字符。请看下面的代码：

```
char ch;
cin >> ch;
```

如果您输入 5 并按回车键，上述代码将读取字符“5”，并将其对应的字符编码（ASCII 编码 53）存储到变量 ch 中。请看下面的代码：

```
int n;
cin >> n;
```

如果您也输入 5 并按回车键，上述代码将读取字符“5”，将其转换为相应的数字值 5，并存储到变量 n 中。

最后，该程序使用函数 cout.put( )显示变量 ch 和一个字符常量。

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。句点被称为成员运算符。cout.put( )的意思是，通过类对象 cout 来使用函数 put( )。第 10 章介绍类时将更详细地介绍这一点。现在，您接触的类只有 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<<运算符。现在读者可能会问，为何需要 cout.put( )。答案与历史有关。在 C++的 Release 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。下面的语句从常量‘M’中复制 8 位（左边的 8 位）到变量 ch 中：

```
char ch = 'M';
```

遗憾的是，这意味着对 cout 来说，‘M’和 ch 看上去有天壤之别，虽然它们存储的值相同。因此，下面的语句将打印$字符的 ASCII 码，而不是字符$：

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

但下面的语句将打印字符$：

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

在 Release 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 章介绍了循环后再来讨论这个主题。

3. char 字面值

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'为 97，即字符 a 的 ASCII 码；
- '5'为 53，即数字 5 的 ASCII 码；
- ' '为 32，即空格字符的 ASCII 码；
- '!'为 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，按回车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字面值，因此不能把双引号放在字符串字面值中。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!
```

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

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。

转义序列的概念可追溯到使用电传打字机与计算机通信的时代，现代系统并非都支持所有的转义序列。例如，输入振铃字符时，有些系统保持沉默。

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开头：

```
cout << endl;     // using the endl manipulator
cout << '\n';     // using a character 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 << "The 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，绘制了一幅哈得逊河图画；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\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，它包括在程序执行期间可处理的字符（如可从文件中读取或显示到屏幕上的字符）。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 码点（ISO 10646 是一种正在制定的国际标准，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 码点为 00F6，而 â 的码点为 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。

实际上，从易读性的角度看，在变量名中使用\u00F6 没有多大意义，但如果实现的扩展源字符集包含 ö，它可能允许您从键盘输入该字符。

请注意，C++使用术语“通用编码名”，而不是“通用编码”，这是因为应将\u00F6 解释为“Unicode 码点为 U-00F6 的字符”。支持 Unicode 的编译器知道，这表示字符 ö，但无需使用内部编码 00F6。无论计算机使用是 ASCII 还是其他编码系统，都可在内部表示字符 T；同样，在不同的系统中，将使用不同的编码来表示字符 ö。在源代码中，可使用适用于所有系统的通用编码名，而编译器将根据当前系统使用合适的内部编码来表示它。

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。然而，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、切罗基语、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 109000 多种符号和 90 多个手写符号（script），它还在不断发展中。

Unicode 给每个字符指定一个编号——码点。Unicode 码点通常类似于下面这样：U-222B。其中 U 表示这是一个 Unicode 字符，而 222B 是该字符（积分正弦符号）的十六进制编号。

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。

7. C++11 新增的类型：char16_t 和 char32_t

随着编程人员日益熟悉 Unicode，类型 wchar_t 显然不再能够满足需求。事实上，在计算机系统上进行字符和字符串编码时，仅使用 Unicode 码点并不够。具体地说，进行字符串编码时，如果有特定长度和符号特征的类型，将很有帮助，而类型 wchar_t 的长度和符号特征随实现而已。因此，C++11 新增了类型 char16_t 和 char32_t，其中前者是无符号的，长 16 位，而后者也是无符号的，但长 32 位。C++11 使用前缀 u 表示 char16_t 字符常量和字符串常量，如 u‘C’和 u“be good”；并使用前缀 U 表示 char32_t 常量，如 U‘R’和 U“dirty rat”。类型 char16_t 与/u00F6 形式的通用字符名匹配，而类型 char32_t 与/U0000222B 形式的通用字符名匹配。前缀 u 和 U 分别指出字符字面值的类型为 char16_t 和 char32_t：

```
char16_t ch1 = u'q';            // basic character in 16-bit form
char32_t ch2 = U'\U0000222B';   // universal character name in 32-bit form
```

与 wchar_t 一样，char16_t 和 char32_t 也都有底层类型——一种内置的整型，但底层类型可能随系统而已。

3.1.9 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。然而，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 is_ready = 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，示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。符号名称指出了常量表示的内容。另外，如果程序在多个地方使用同一个常量，则需要修改该常量时，只需修改一个符号定义即可。本章前面关于#define 语句的说明（旁注“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。如果您这样做，g++将指出程序试图给一个只读变量赋值。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的首字母大写，以提醒您 Months 是个常量。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。另一种约定是将整个名称大写，使用#define 创建常量时通常使用这种约定。还有一种约定是以字母 k 打头，如 kmonths。当然，还有其他约定。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，且无法修改。

如果以前使用过 C 语言，您可能觉得前面讨论的#define 语句已经足够完成这样的工作了。但 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，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。如果熟悉 ANSI C 版本，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，来看看浮点类型，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，如人体的细菌数（估计超过 100 兆），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。然而，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 1000000 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 3450000，6 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
1.69e12            // 2010 Brazilian public debt in reais
5.98E24            // mass of earth in kilograms
9.11e-31           // mass of an electron in kilograms
```

读者可能注意到了，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。）然而，数字中不能有空格，因此 7.2 E6 是非法的。

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示数字（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。之所以称为“浮点”，就是因为小数点可移动。

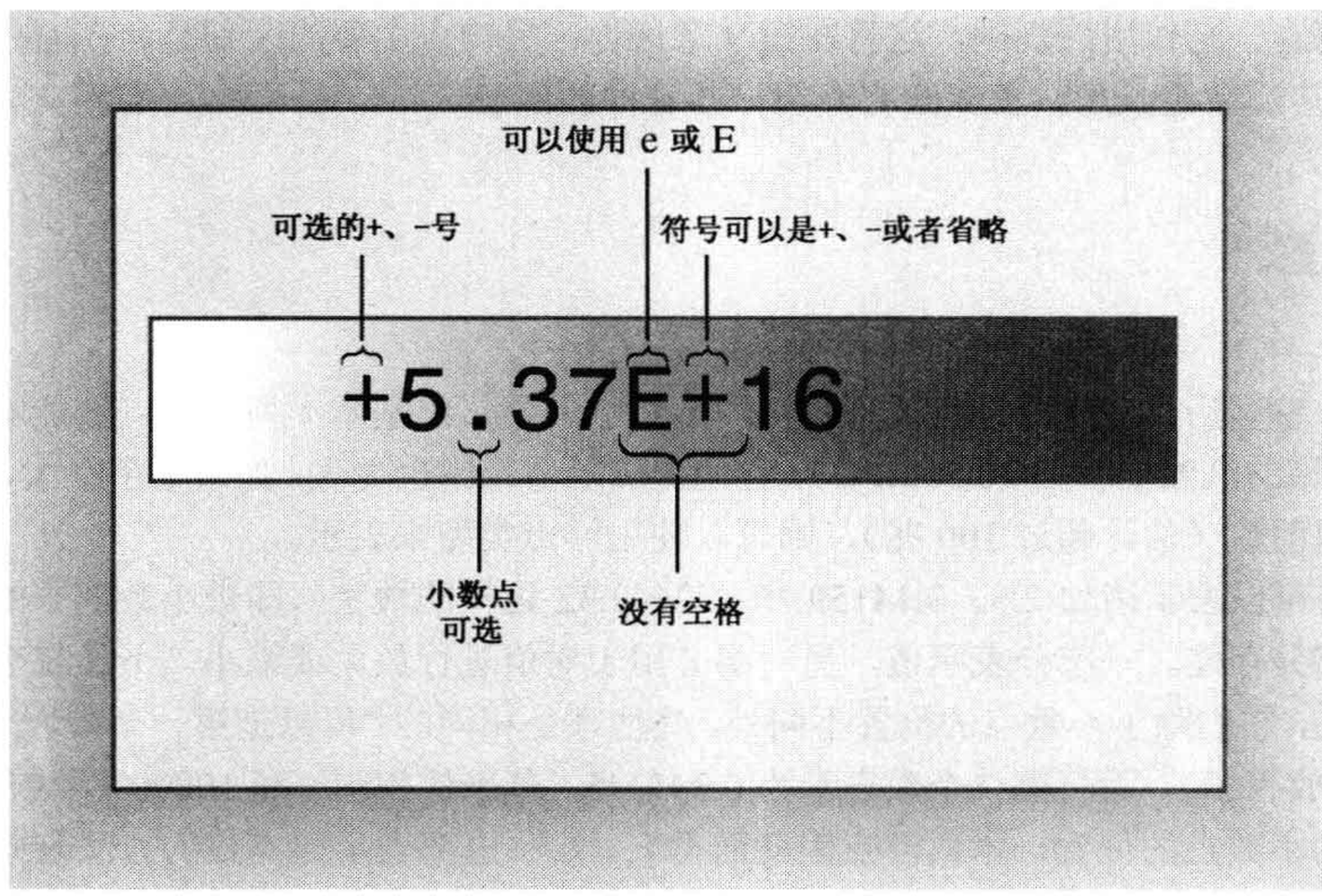


图 3.3　E 表示法

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 Shasta 山脉的高度为 14179 英尺，该数字使用了 5 个有效位，指出了最接近的英尺数。然而，将 Shasta 山脉的高度写成约 14000 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。然而，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。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
```

1. 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——3.333333333333333333……由于 cout 打印 6 位小数，因此 tub 和 mint 都是精确的。但当程序将每个数乘以一百万后，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，但这是最糟糕的情况）。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 乘以一百万，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 类有一个类成员函数，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。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<<运算符。有时候，这种方法显示的位数比需要的位数多，但这只会影响美观。如果您介意这种问题，可以浏览第 17 章，了解如何使用格式化方法。然而，在这里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、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——在有些情况下，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）。下面是一些示例：

```
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，浮点运算的速度通常比整数运算慢，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；C++11 新增了 long long。bool、char、wchar_t、符号整数和无符号整型统称为整型；C++11 新增了 char16_t 和 char32_t。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++算术运算符

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++使用运算符来运算。它提供了几种运算符来完成 5 种基本的算术计算：加法、减法、乘法、除法以及求模。每种运算符都使用两个值（操作数）来计算结果。运算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，+是加法运算符，4+2 则是一个表达式，其值为 6。

下面是 5 种基本的 C++算术运算符。

- +运算符对操作数执行加法运算。例如，4+20 等于 24。
- –运算符从第一个数中减去第二个数。例如，12–3 等于 9。
- *运算符将操作数相乘。例如，28*4 等于 112。
- /运算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。
- %运算符求模。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，将该运算符用于浮点数将导致编译错误。如果其中一个是负数，则结果的符号满足如下规则：(a/b)*b + a%b ＝ a。

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;
}
```

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 运算符优先级和结合性

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运算符。这样将产生一个问题：究竟哪个运算符最先被使用呢？例如，请看下面的语句：

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

操作数 4 旁边有两个运算符：+和*。当多个运算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运算符。算术运算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++运算符的优先级。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 也位于两个运算符中间，但运算符/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。当两个运算符的优先级相同时，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运算符被同时用于同一个操作数，则首先应用左侧的运算符。从右到左的结合性则首先应用右侧的运算符。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 使用左侧的运算符。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，仅当两个运算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;
```

运算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*运算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 章讨论递增运算符时，将介绍一个这样的例子。

3.4.2 除法分支

除法运算符（/）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运算符时，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。

运算符重载简介

在程序清单 3.11 中，除法运算符表示了 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）来确定运算符的含义。使用相同的符号进行多种操作叫做运算符重载（operator overloading）。C++有一些内置的重载示例。C++还允许扩展运算符重载，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）。

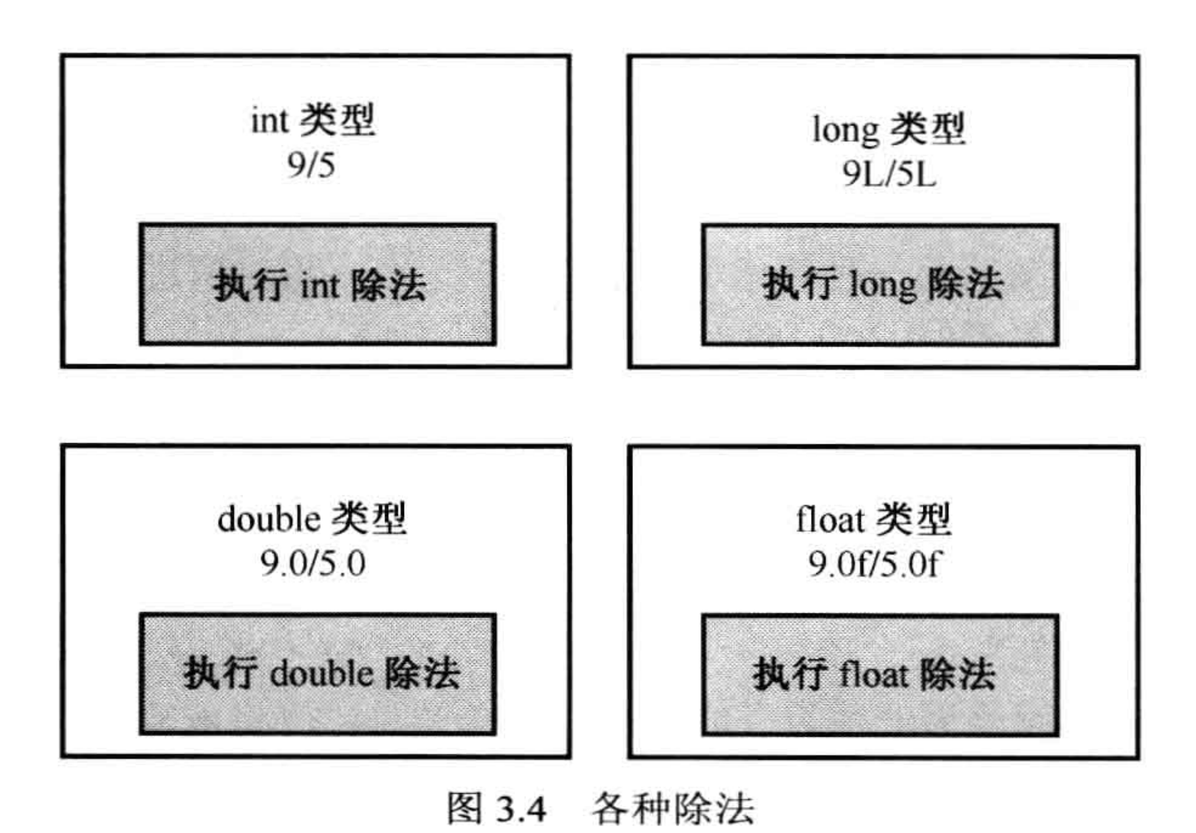


图 3.4　各种除法

3.4.3　求模运算符

比起求模运算符来说，多数人更熟悉加、减、乘、除，因此这里花些时间介绍这种运算符。求模运算

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，再用求模运算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: 181
181 pounds are 12 stone, 13 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 181，所以表达式的值为 12。12 和 14 的乘积是 168，所以 181 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. *初始化和赋值进行的转换*

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 long 值（通常为 32 位）。扩展后将得到一个新值，这个值被存储在 so_long 中，而 thirty 的内容不变。

将一个值赋给值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。然而，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，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。

传统初始化的行为与赋值相同，程序清单 3.13 演示了一些初始化进行的转换。

程序清单 3.13　assign.cpp

```
// init.cpp -- type changes on initialization
#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);  // double 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. *以{ }方式初始化时进行的转换（C++11）*

C++11 将使用大括号的初始化称为列表初始化（list-initialization），因为这种初始化常用于给复杂的数据类型提供值列表。与程序清单 13.3 所示的初始化方式相比，它对类型转换的要求更严格。具体地说，列

表初始化不允许缩窄（narrowing），即变量的类型可能无法表示赋给它的值。例如，不允许将浮点型转换为整型。在不同的整型之间转换或将整型转换为浮点型可能被允许，条件是编译器知道目标变量能够正确地存储赋给它的值。例如，可将 long 变量初始化为 int 值，因为 long 总是至少与 int 一样长；相反方向的转换也可能被允许，只要 int 变量能够存储赋给它的 long 常量：

```
const int code = 66;
int x = 66;
char c1 {31325};  // narrowing, not allowed
char c2 = {66};   // allowed because char can hold 66
char c3 {code};   // ditto
char c4 = {x};    // not allowed, x is not constant
x = 31325;
char c5 = x;      // allowed by this form of initialization
```

在上述代码中，初始化 c4 时，您知道 x 的值为 66，但在编译器看来，x 是一个变量，其值可能很大。编译器不会跟踪下述阶段可能发生的情况：从 x 被初始化到它被用来初始化 c4。

3. 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 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。C++11 对这个校验表稍做了修改，下面是 C++11 版本的校验表，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，如果两个操作数都是有符号或无符号的，且其中一个操作数的级别比另一个低，则转换为级别高的类型。

（6）如果一个操作数为有符号的，另一个操作数为无符号的，且无符号操作数的级别比有符号操作数

高，则将有符号操作数转换为无符号操作数所属的类型。

（7）否则，如果有符号类型可表示无符号类型的所有可能取值，则将无符号操作数转换为有符号操作数所属的类型。

（8）否则，将两个操作数都转换为有符号类型的无符号版本。

ANSI C 遵循的规则与 ISO 2003 C++相同，这与前述规则稍有不同；而传统 K&R C 的规则又与 ANSI C 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。

前面的列表谈到了整型级别的概念。简单地说，有符号整型按级别从高到低依次为 long long、long、int、short 和 signed char。无符号整型的排列顺序与有符号整型相同。类型 char、signed char 和 unsigned char 的级别相同。类型 bool 的级别最低。wchar_t、char16_t 和 char32_t 的级别与其底层类型相同。

4. 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。然而，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。

5. 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 个强制类型转换运算符，对它们的使用要求更为严格，这将在第 15 章介绍。在这四个运算符中，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。运算符 static_cast<>比传统强制类型转换更严格。

程序清单 3.14 演示了这两种基本的强制类型转换和 static_cast<>。可以将该程序第一部分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
    cout << "Yes, the code is ";
    cout << static_cast<int>(ch) << endl;       // using static_cast
    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
Yes, the code 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。接下来，两条 cout 语句使用强制类型转换将 char 类型的值转换为 int，再显示它。这些转换导致 cout 将值打印为整数，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。

3.4.5 C++11 中的 auto 声明

C++11 新增了一个工具，让编译器能够根据初始值的类型推断变量的类型。为此，它重新定义了 auto 的含义。auto 是一个 C 语言关键字，但很少使用，有关其以前的含义，请参阅第 9 章。在初始化声明中，如果使用关键字 auto，而不指定变量的类型，编译器将把变量的类型设置成与初始值相同：

```
auto n = 100;      // n is int
auto x = 1.5;      // x is double
auto y = 1.3e12L;  // y is long double
```

然而，自动推断类型并非为这种简单情况而设计的；事实上，如果将其用于这种简单情形，甚至可能让您误入歧途。例如，假设您要将 x、y 和 z 都指定为 double 类型，并编写了如下代码：

```
auto x = 0.0;    // ok, x is double because 0.0 is double
double y = 0;    // ok, 0 automatically converted to 0.0
auto z = 0;      // oops, z is int because 0 is int
```

显式地声明类型时，将变量初始化 0（而不是 0.0）不会导致任何问题，但采用自动类型推断时，这却会导致问题。

处理复杂类型，如标准模块库（STL）中的类型时，自动类型推断的有时才能显现出来。例如，对于下述 C++98 代码：

```
std::vector<double> scores;
std::vector<double>::iterator pv = scores.begin();
```

C++11 允许您将其重写为下面这样：

```
std::vector<double> scores;
auto pv = scores.begin();
```

本书后面讨论相关的主题时，将再次提到 auto 的这种新含义。

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、unsigned long 以及 C++11 新增的 long long 和 unsigned long 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++11 新增了类型 char16_t 和 char32_t，它们的宽度足以分别存储 16 和 32 位的字符编码。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++使用运算符来提供对数字类型的算术运算：加、减、乘、除和求模。当两个运算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，此时您将感谢 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 变量会导致舍入误差，将 long 值赋给 double 变量呢？将 long long 值赋给 double 变量呢？

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 double 变量，您要将它们作为整数相加，再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。如果要将它们作为 double 值相加并转换为 int 呢？

10. 下面每条语句声明的变量都是什么类型？

```
a. auto cars = 15
b. auto iou = 150.37f
c. auto level = 'B'
d. auto crat = U'/U00002155'
e. auto fract = 8.25f/2.5
```

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 或 long 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, 17 hours, 46 minutes, 40 seconds
```

5．编写一个程序，要求用户输入全球当前的人口和美国当前的人口（或其他国家的人口）。将这些信息存储在 long long 变量中，并让程序显示美国（或其他国家）的人口占全球人口的百分比。该程序的输出应与下面类似：

```
Enter the world's population: 6898758899
Enter the population of the US: 310783781
The population of the US is 4.50492% of the world population.
```

6．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）。

7．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）与欧洲方法（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：

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

假设您开发了一个名叫 User-Hostile 的计算机游戏，玩家需要用智慧来应对一个神秘、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，它是将学习的 OOP 的堡垒。然而，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：

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

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。然而，本章稍后将介绍如何使用 new 运算符来避开这种限制。

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。

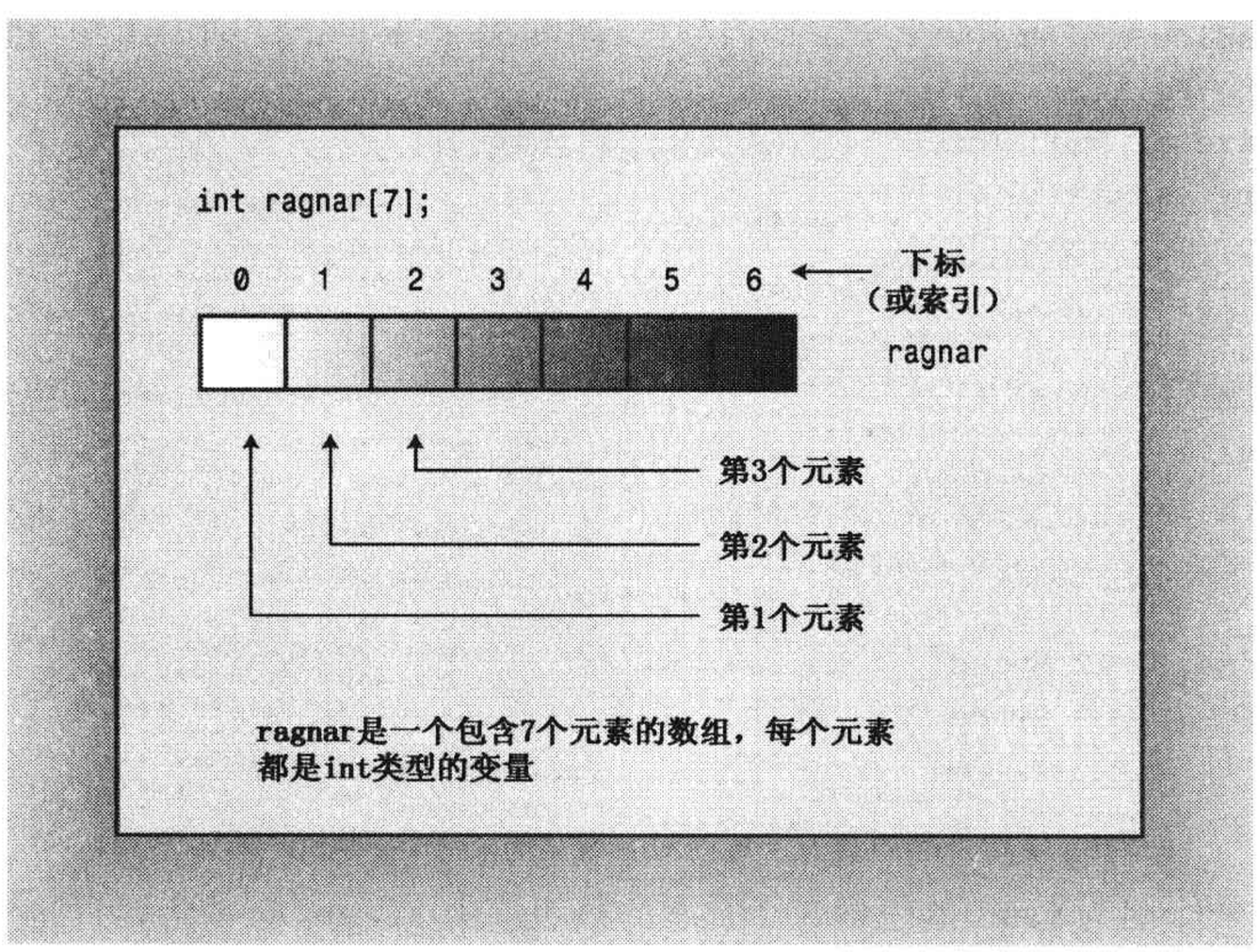


图 4.1　创建数组

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。

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;
}
```

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.
```

4.1.1　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。

您可能还记得，sizeof 运算符返回类型或数据对象的长度（单位为字节）。注意，如果将 sizeof 运算符用于数组名，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。

4.1.2　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
```

然而，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种很糟的做法，因为其计数可能与您想象的不一样。例如，您可能不小心在列表中遗漏了一个值。然而，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，很快您将明白这一点。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。

4.1.3　C++11 数组初始化方法

第 3 章说过，C++11 将使用大括号的初始化（列表初始化）作为一种通用初始化方式，可用于所有类

型。数组以前就可使用列表初始化，但 C++11 中的列表初始化新增了一些功能。

首先，初始化数组时，可省略等号（=）：

```
double earnings[4] {1.2e4, 1.6e4, 1.1e4, 1.7e4};  // okay with C++11
```

其次，可不在大括号内包含任何东西，这将把所有元素都设置为零：

```
unsigned int counts[10] = {};  // all elements set to 0
float balances[100] {};        // all elements set to 0
```

第三，列表初始化禁止缩窄转换，这在第 3 章介绍过：

```
long plifs[] = {25, 92, 3.0};            // not allowed
char slifs[4] {'h', 'i', 1122011, '\0'}; // not allowed
char tlifs[4] {'h', 'i', 112, '\0'};     // allowed
```

在上述代码中，第一条语句不能通过编译，因为将浮点数转换为整型是缩窄操作，即使浮点数的小数点后面为零。第二条语句也不能通过编译，因为 1122011 超出了 char 变量的取值范围（这里假设 char 变量的长度为 8 位）。第三条语句可通过编译，因为虽然 112 是一个 int 值，但它在 char 变量的取值范围内。

C++标准模板库（STL）提供了一种数组替代品——模板类 vector，而 C++11 新增了模板类 array。这些替代品比内置复合类型数组更复杂、更灵活，本章将简要地讨论它们，而第 16 章将更详细地讨论它们。

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[8] = { 'b', 'e', 'a', 'u', 'x', ' ', 'I', 'I'};      // not a string!
char cat[8] = {'f', 'a', 't', 'e', 's', 's', 'a', '\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 7 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 8 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 数组示例中，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[11] = "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。

警告：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。

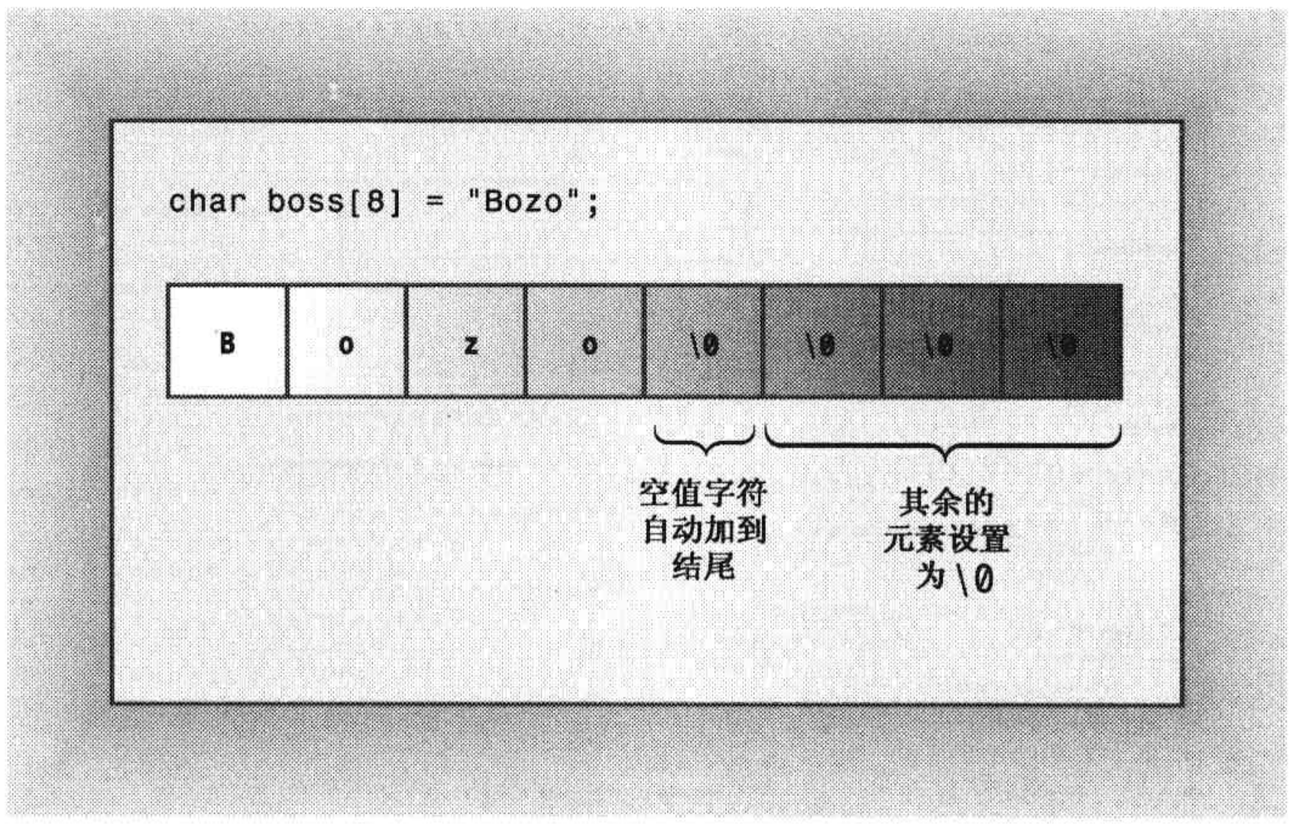


图 4.2　将数组初始化为字符串

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，下面的语句将 83 赋给 shirt_size：

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

但"S"不是字符常量，它表示的是两个字符（字符 S 和\0）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。因此下面的语句试图将一个内存地址赋给 shirt_size：

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

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++允许拼接字符串字面值，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 C 语言库函数 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';                // set to null character
    cout << "Here are the first 3 characters of my name: ";
    cout << name2 << endl;
    return 0;
}
```

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 运算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）。

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。

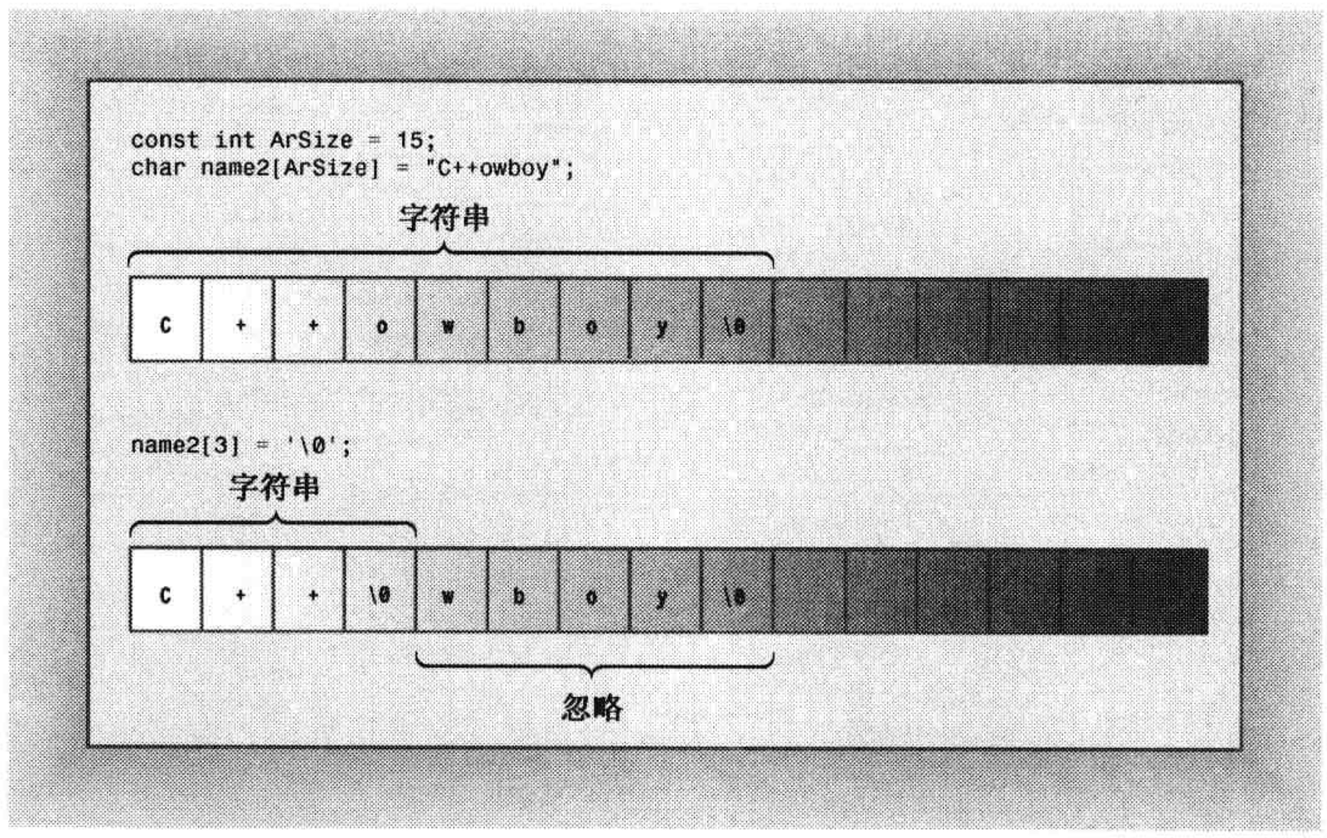


图 4.3　使用\0 截短字符串

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”作出反应，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）来确定字符串的结束位置，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）。

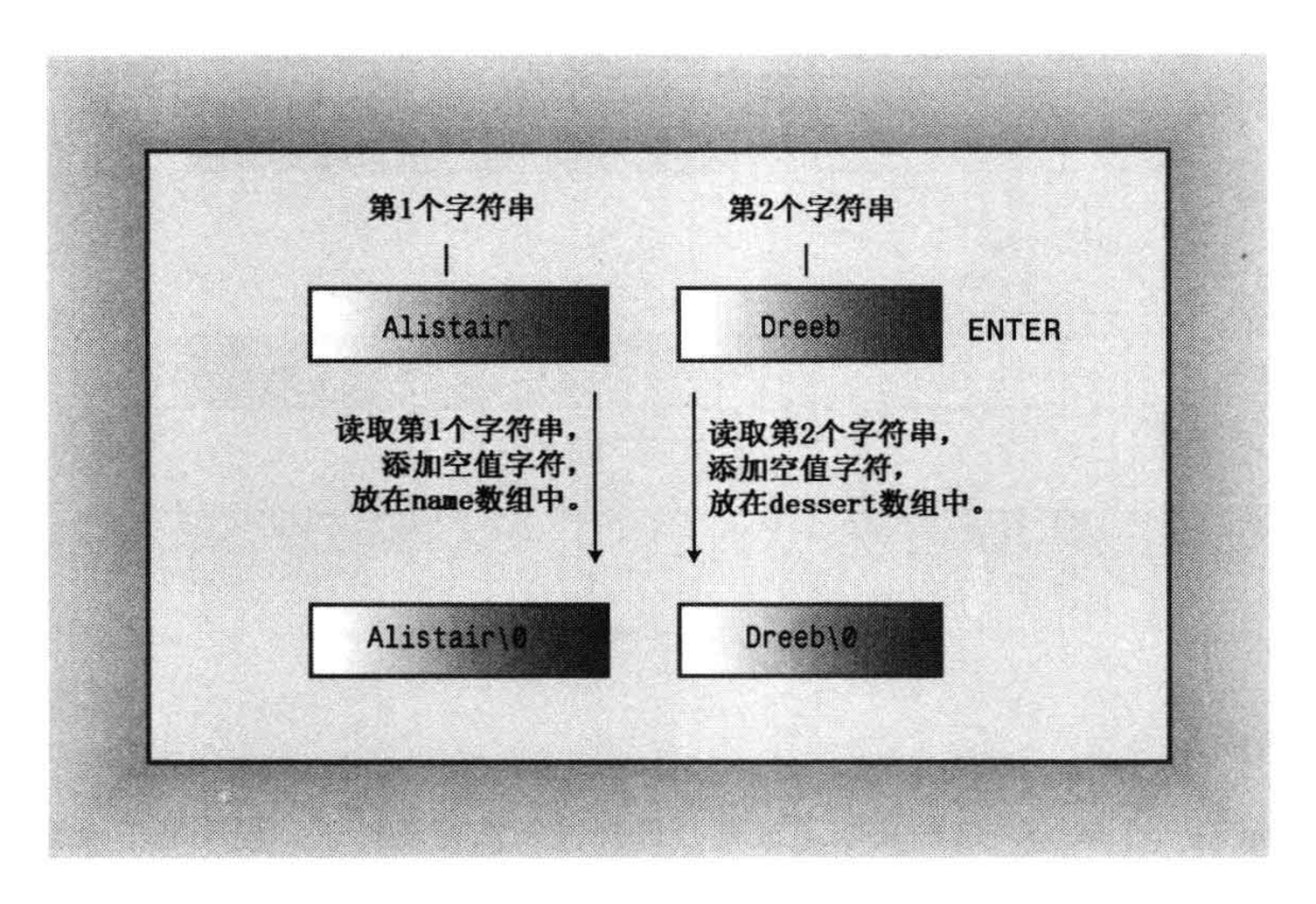


图 4.4　使用 cin 读取字符串输入时的情况

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，因此有必要对该主题做进一步探讨。我们必须使用 cin 的较高级特性，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;
}
```

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）。

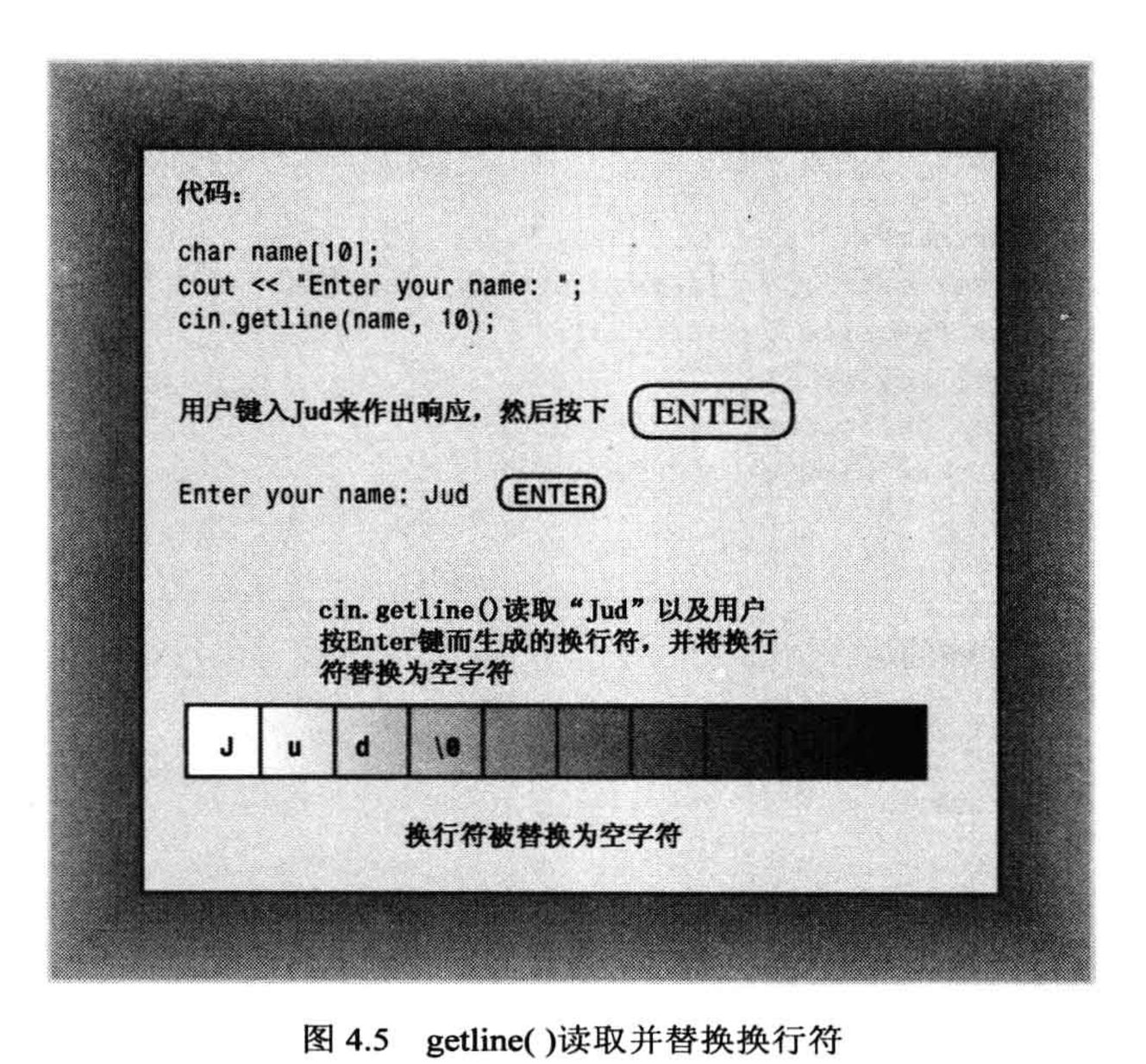


图 4.5　getline( )读取并替换换行符

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
```

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：

```
cin.getline(name1, ArSize).getline(name2, ArSize);
```

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;
}
```

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 章和第 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，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，再介绍字符串这个方面的特性。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++98 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。您将看到，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
```

从这个示例可知，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　C++11 字符串初始化

正如您预期的，C++11 也允许将列表初始化用于 C-风格字符串和 string 对象：

```
char first_date[] = {"Le Chapon Dodu"};
char second_date[] {"The Elegant Plate"};
string third_date = {"The Bread Bowl"};
string fourth_date {"Hank's Fine Eats"};
```

4.3.2　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 类简化了字符串合并操作。可以使用运算符+将两个 string 对象合并起来，还可以使用运算符+=将字符串附加到 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-风格字符串或 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
```

4.3.3 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-风格字符串，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 str1
    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 对象的语法通常比使用 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-风格字符串时，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，如下面的示例所示：

```
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运算符来指出要使用哪个字符串。

4.3.4 string 类 I/O

正如您知道的，可以使用 cin 和运算符<<来将输入存储到 string 对象中，使用 cout 和运算符<<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示例：

```
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。这里要两点需要说明。首先，为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，因此您在运行该程序时，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，因此您可能会问，下述代码为何可行呢？

```
cin >> str;  // read a word into the str string object
```

像下面这样的代码使用 istream 类的一个成员函数：

```
cin >> x;  // read a value into a basic C++ type
```

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，您可以将 cin 和 cout 用于 string 对象，而不用考虑其内部工作原理。

4.3.5　其他形式的字符串字面值

本书前面说过，除 char 类型外，C++还有类型 wchar_t；而 C++11 新增了类型 char16_t 和 char32_t。可创建这些类型的数组和这些类型的字符串字面值。对于这些类型的字符串字面值，C++分别使用前缀 L、u 和 U 表示，下面是一个如何使用这些前缀的例子：

```
wchar_t title[] = L"Chief Astrogator";  // w_char string
char16_t name[] = u"Felonia Ripova";    // char_16 string
char32_t car[] = U"Humber Super Snipe"; // char_32 string
```

C++11 还支持 Unicode 字符编码方案 UTF-8。在这种方案中，根据编码的数字值，字符可能存储为 1～4 个八位组。C++使用前缀 u8 来表示这种类型的字符串字面值。

C++11 新增的另一种类型是原始（raw）字符串。在原始字符串中，字符表示的就是自己，例如，序列\n 不表示换行符，而表示两个常规字符——斜杠和 n，因此在屏幕上显示时，将显示这两个字符。另一个例子是，可在字符串中使用"，而无需像程序清单 4.8 中那样使用繁琐的\"。当然，既然可在字符串字面量包含"，就不能再使用它来表示字符串的开头和末尾。因此，原始字符串将"(和)"用作定界符，并使用前缀 R 来标识原始字符串：

```
cout << R"(Jim "King" Tutt uses "\n" instead of endl.)" << '\n';
```

上述代码将显示如下内容：

```
Jim "King" Tutt uses \n instead of endl.
```

如果使用标准字符串字面值，将需编写如下代码：

```
cout << "Jim \"King\" Tutt uses \" \\n\" instead of endl." << '\n';
```

在上述代码中，使用了\\来显示\，因为单个\表示转义序列的第一个字符。

输入原始字符串时，按回车键不仅会移到下一行，还将在原始字符串中添加回车字符。

如果要在原始字符串中包含)"，该如何办呢？编译器见到第一个)"时，会不会认为字符串到此结束？会的。但原始字符串语法允许您在表示字符串开头的"和(之间添加其他字符，这意味着表示字符串结尾的"和)之间也必须包含这些字符。因此，使用 R"+*(标识原始字符串的开头时，必须使用)+*"标识原始字符串的结尾。因此，下面的语句：

```
cout << R"+*("(Who wouldn't?)", she whispered.)+*" << endl;
```

将显示如下内容：

```
"(Who wouldn't?)", she whispered.
```

总之，这使用"+*(和)+*"替代了默认定界符"(和)"。自定义定界符时，在默认定界符之间添加任意数量的基本字符，但空格、左括号、右括号、斜杠和控制字符（如制表符和换行符）除外。

可将前缀 R 与其他字符串前缀结合使用，以标识 wchar_t 等类型的原始字符串。可将 R 放在前面，也可将其放在后面，如 Ru、UR 等。

下面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++中的结构的可以满足要求（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 declaration
{
    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）。总之，结构定义指出了新类型（这里是 inflatable）的特征。

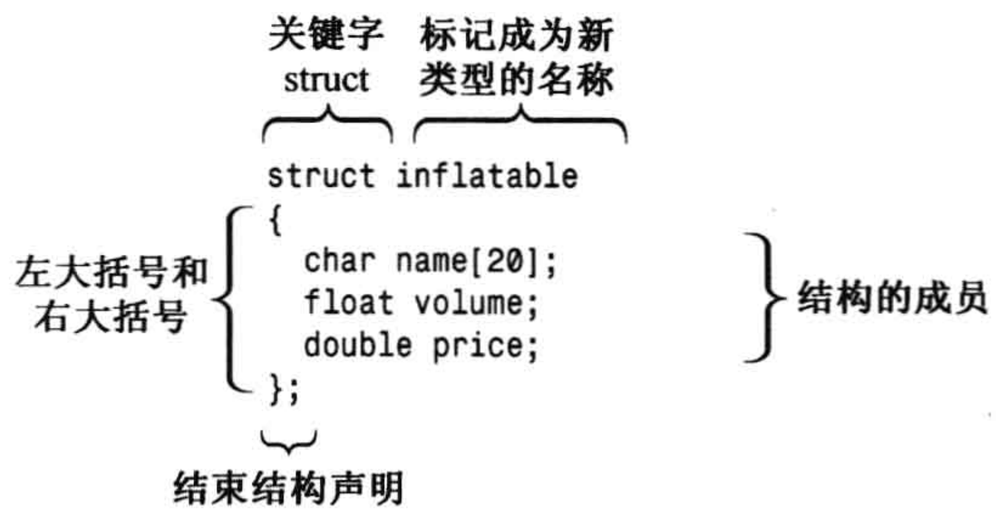


图 4.6 结构描述的组成部分

定义结构后，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
```

如果您熟悉 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，因此可以使用成员运算符（.）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;
}
```

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）。

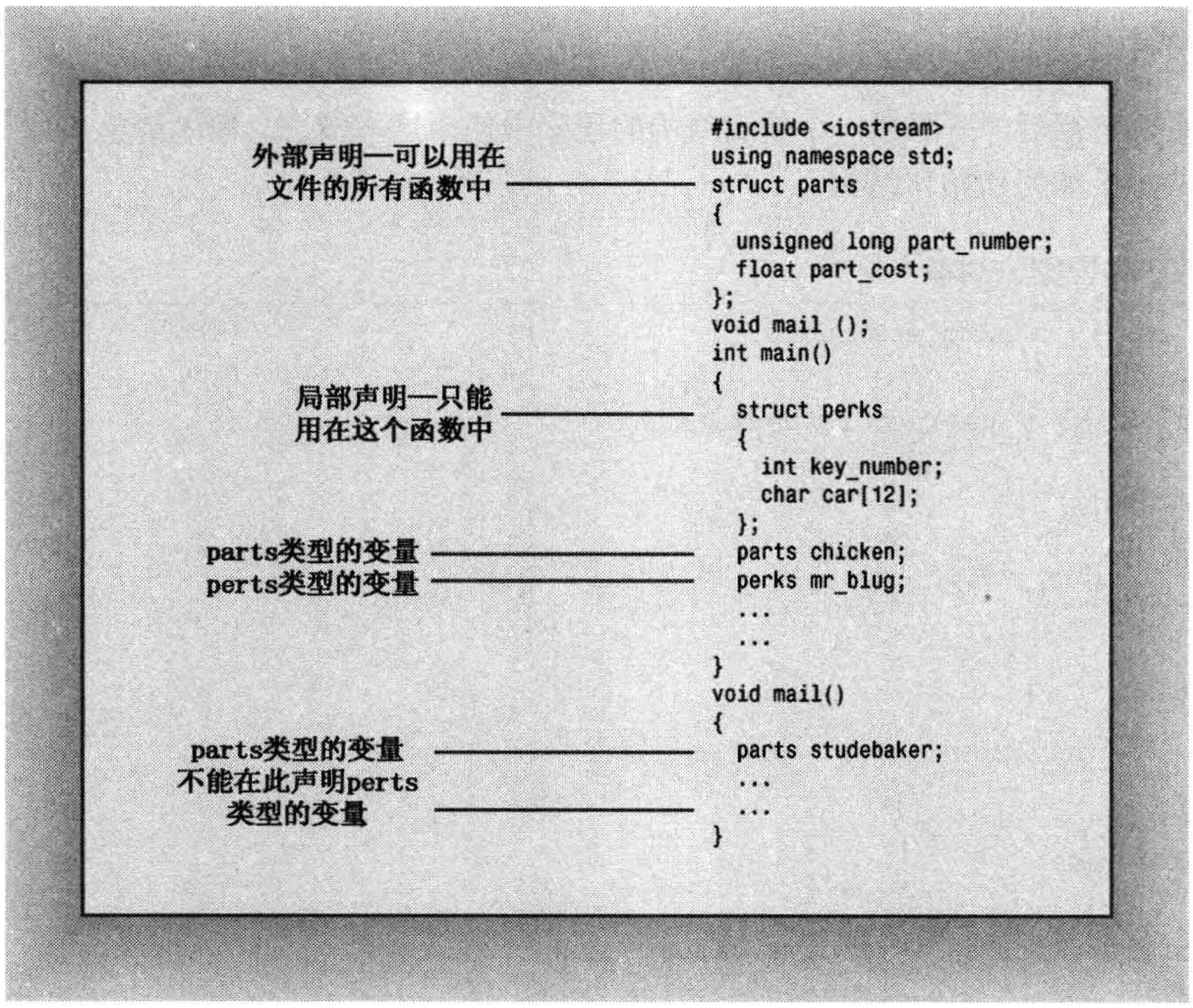


图 4.7　局部结构声明和外部结构声明

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　C++11 结构初始化

与数组一样，C++11 也支持将列表初始化用于结构，且等号（=）是可选的：

```
inflatable duck {"Daphne", 0.12, 9.98};  // can omit the = in C++11
```

其次，如果大括号内未包含任何东西，各个成员都将被设置为零。例如，下面的声明导致 mayor.volume 和 mayor.price 被设置为零，且 mayor.name 的每个字节都被设置为零：

```
inflatable mayor {};
```

最后，不允许缩窄转换。

4.4.3　结构可以将 string 类作为成员吗

可以将成员 name 指定为 string 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 definition
{
    std::string name;
    float volume;
    double price;
};
```

答案是肯定的，只要您使用的编译器支持对以 string 对象作为成员的结构进行初始化。

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。

4.4.4　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，还可以使用赋值运算符（=）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），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 是一个这样的示例。

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
};
```

然而，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。可以使用成员运算符来访问它的成员（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讨论的 C++结构的所有特性（C++11 特性除外），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。

4.4.5　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，可以与成员运算符一起使用：

```
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示例。由于 guests 是一个 inflatable 数组，因此 guests[0]的类型为 inflatable，可以使用它和句点运算符来访问相应 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.
```

4.4.6 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 介绍的按位运算符来代替这种方式。

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。

共用体常用于（但并非只能用于）节省内存。当前，系统的内存多达数 GB 甚至数 TB，好像没有必要节省内存，但并非所有的 C++程序都是为这样的系统编写的。C++还用于嵌入式系统编程，如控制烤箱、MP3 播放器或火星漫步者的处理器。对这些应用程序来说，内存可能非常宝贵。另外，共用体常用于操作系统数据结构或硬件数据结构。

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，只定义了赋值运算符。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
...
```

然而，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。

前面示例：

```
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运算符+，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，以了解一下哪些值合适，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（有关示例见第 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

可以使用赋值运算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，但这种限制取消了，因此可以使用 long 甚至 long 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。然而，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。

C++11 扩展了枚举，增加了作用域内枚举（scoped enumeration），第 10 章的“类作用域”一节将简要地介绍这种枚举。

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。

您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

变量应用地址运算符（&），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 演示了这个运算符的用法。

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;
}
```

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（即 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（参见旁注“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。您可能认为，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。

在运行阶段做决策并非 OOP 独有的，但使用 C++编写这样的代码比使用 C 语言简单。

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。*运算符被称为间接值（indirect velue）或解除引用（dereferencing）运算符，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
```

从中可知，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&运算符来获得地址；而变量 p_updates 表示地址，并使用*运算符来获得值（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。

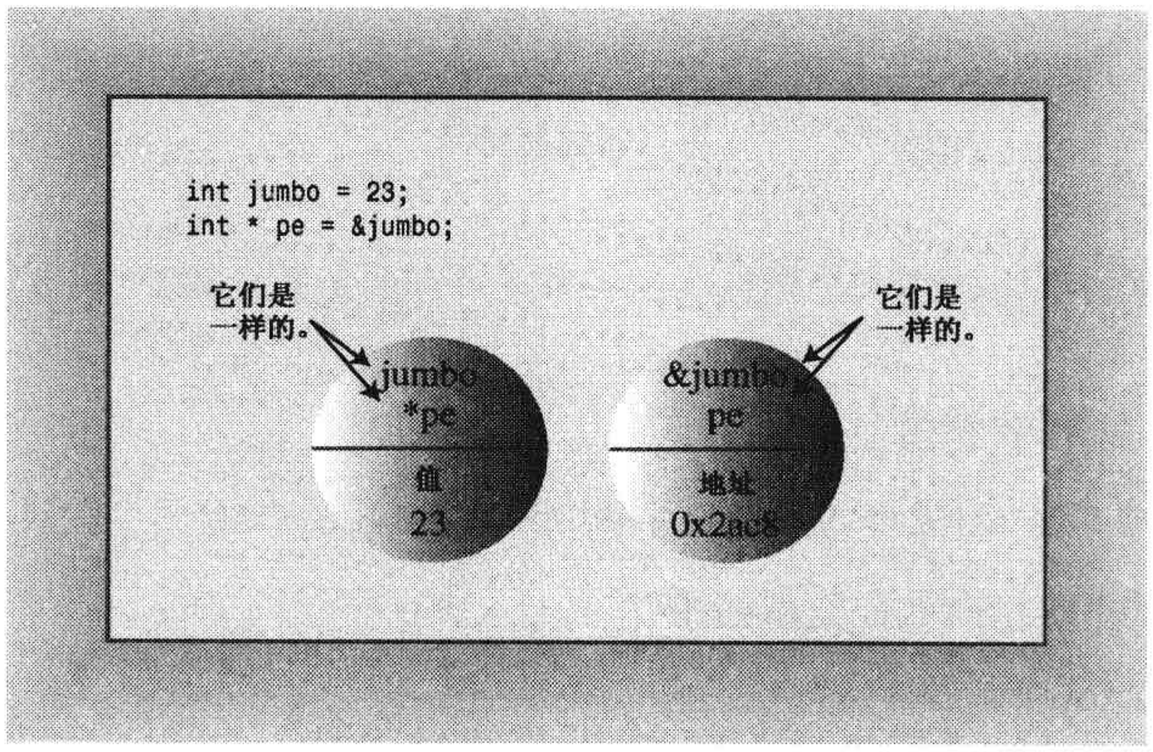


图 4.8　硬币的两面

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，前一个示例包含这样的声明：

```
int * p_updates;
```

这表明，* p_updates 的类型为 int。由于*运算符被用于指针，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）。

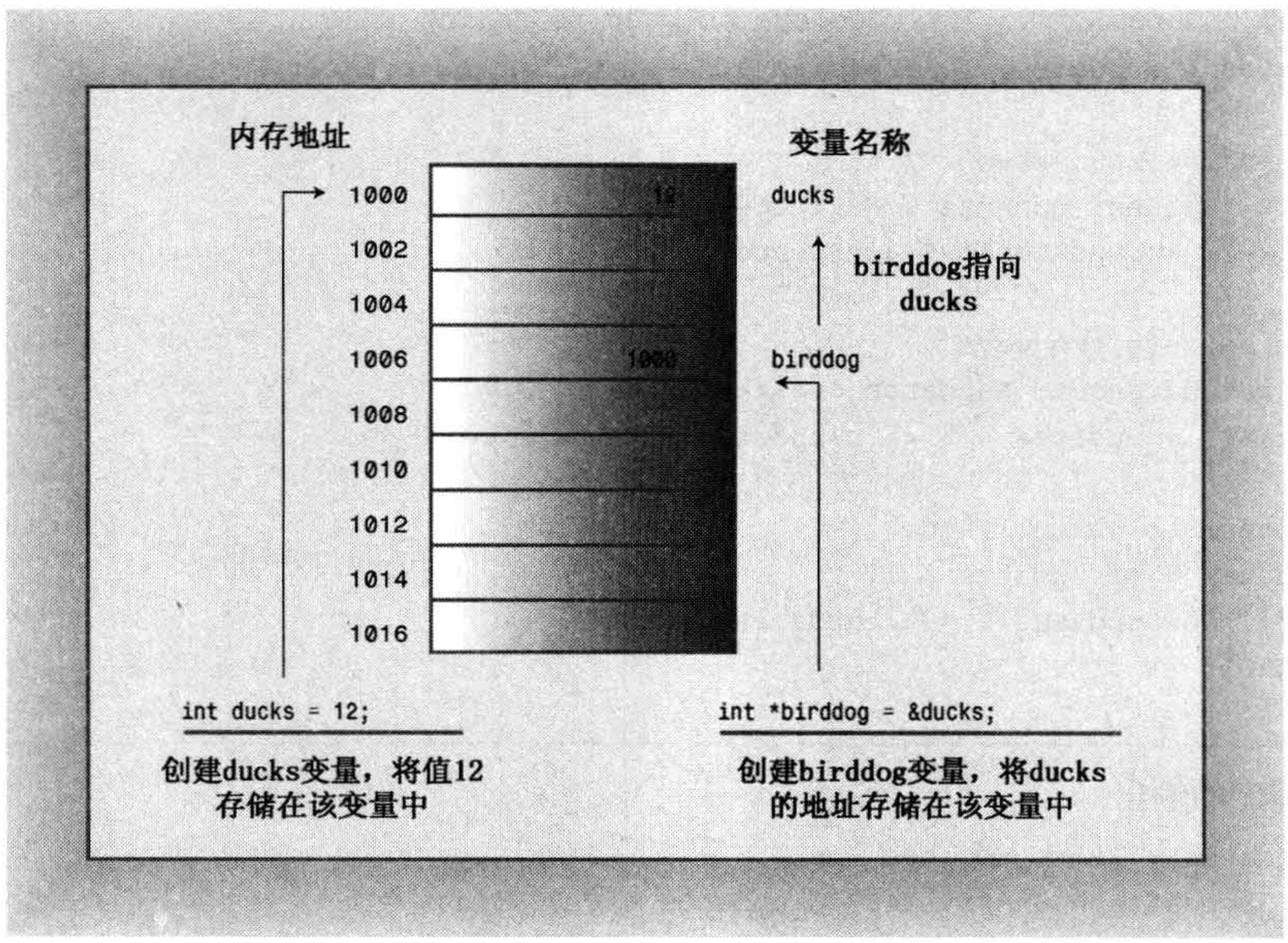


图 4.9 指针存储地址

顺便说一句，*运算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;
```

这强调的是：int*是一种类型——指向 int 的指针。在哪里添加空格对于编译器来说没有任何区别，您甚至可以这样做：

```
int*ptr;
```

但要知道的是，下面的声明创建一个指针（p1）和一个 int 变量（p2）：

```
int* p1, p2;
```

对每个指针变量名，都需要使用一个*。

注意：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（在大多数系统上）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，下面的语句将 pt（而不是*pt）的值设置为&higgens：

```
int higgens = 5;
int * pt = &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;
}
```

下面是该程序的示例输出：

```
Value of higgens = 5; Address of higgens = 0012FED4
Value of *pt = 5; Value of pt = 0012FED4
```

从中可知，程序将 pi（而不是*pi）初始化为 higgens 的地址。在您的系统上，显示的地址可能不同，显示格式也可能不同。

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运算符（*）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 是老式计算机系统中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，在有些平台中，int 类型是个 2 字节值，而地址是个 4 字节值。

指针还有其他一些有趣的特性，这将在合适的时候讨论。下面看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了解后，来看看它如何实现在程序运行时分配内存。前面我们都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 运算符。

下面来试试这种新技术，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 运算符。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。下面是一个这样的示例：

```
int * pn = new int;
```

new int 告诉程序，需要适合存储 int 的内存。new 运算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 nights = 1001;
    int * pt = new int;          // allocate space for an int
    *pt = 1001;                  // store a value there

    cout << "nights value = ";
    cout << nights << ": location " << &nights << endl;
    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 << "location of pointer pd: " << &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：

```
nights value = 1001: location 0028F7F8
int value = 1001: location = 00033A98
double value = 1e+007: location = 000339B8
location of pointer pd: 0028F7FC
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。

对于指针，需要指出的另一点是，new 分配的内存块通常与常规变量声明分配的内存块不同。变量 nights 和 pd 的值都存储在被称为栈（stack）的内存区域中，而 new 从被称为堆（heap）或自由存储区（free store）的内存区域分配内存。第 9 章将更详细地讨论这一点。

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 通常会引发异常——一种将在第 15 章讨论的错误处理技术；而在较老的实现中，new 将返回 0。在 C++中，值为 0 的指针被称为空指针（null pointer）。C++确保空指针不会指向有效的数据，因此它常被用来表示运算符或函数失败（如果成功，它们将返回一个有用的指针）。将在第 6 章讨论的 if 语句可帮助您处理这种问题；就目前而言，您只需如下要点：C++提供了检测并处理内存分配失败的工具。

4.7.5 使用 delete 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 运算符，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。然而，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。但稍后您会看到，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。

下面来看一下关于动态数组的两个基本问题：如何使用 C++的 new 运算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 运算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。

当程序使用完 new 分配的内存块时，应使用 delete 释放它们。然而，对于使用 new 创建的数组，应使用另一种格式的 delete 来释放：

```
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。然而，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 [ ]运算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 运算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 运算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：

```
int * psome = new int [10]; // get a block of 10 ints
```

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您大吃一惊：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（这在有时候也是个问题，但这是另一码事）。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.
```

从中可知，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 = 0x28ccf0, *pw = 10000
add 1 to the pw pointer:
pw = 0x28ccf8, *pw = 20000

ps = 0x28ccea, *ps = 3
add 1 to the ps pointer:
ps = 0x28ccec, *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：

```
double * pw = wages;
```

和所有数组一样，wages 也存在下面的等式：

```
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]中显式地使用地址运算符来将 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，后者是存储在该地址中的值。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，这样数字地址值将增加 8，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，为使改图更为清晰，对其中的地址值做了调整）。

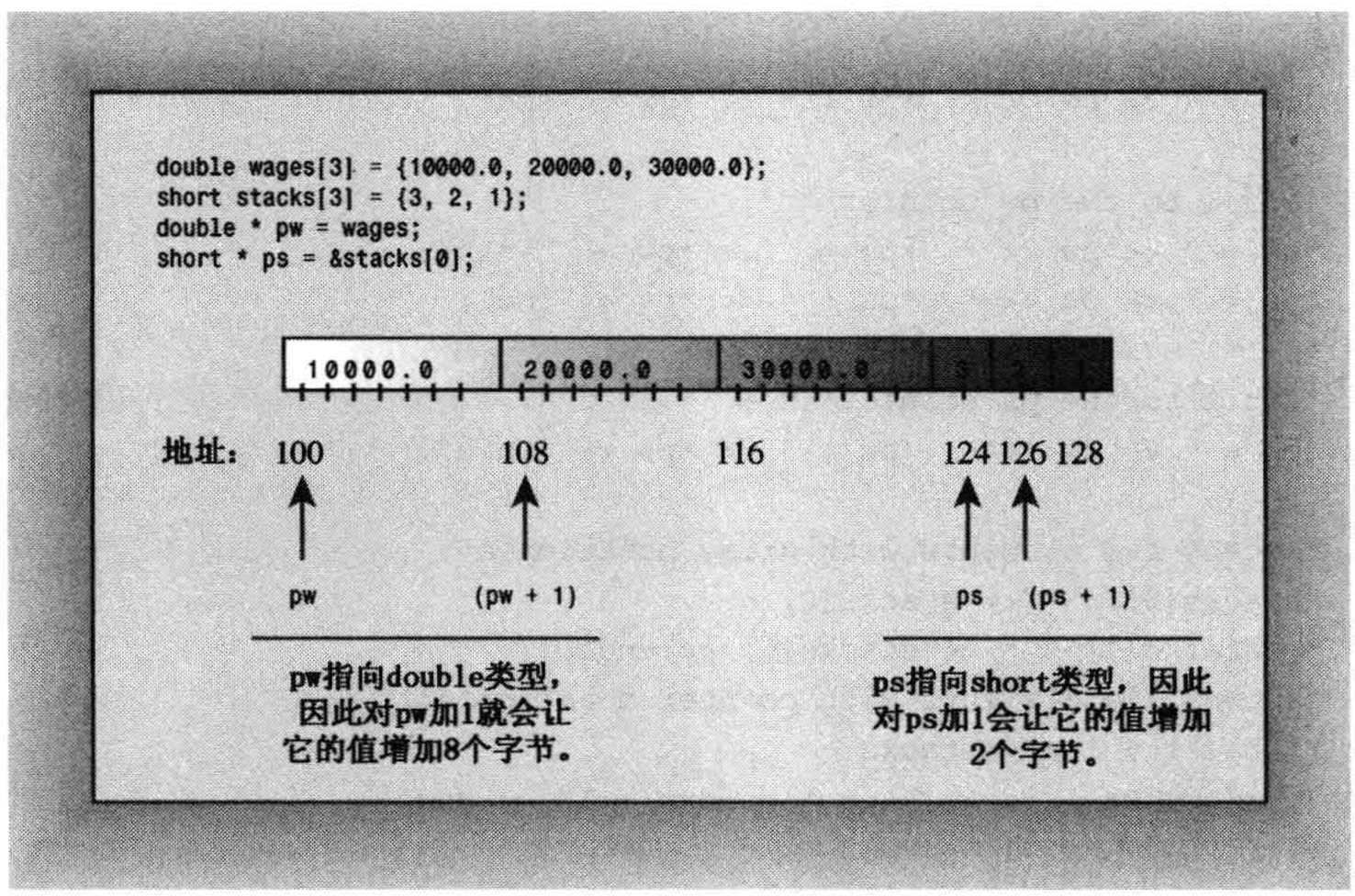


图 4.10 指针加法

此后，程序对 ps 执行相同的操作。这一次由于 ps 指向的是 shor t 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。

注意：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（运算符优先级要求使用括号，如果不使用括号，将给*stacks 加 1，而不是给 stacks 加 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运算符（*）。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 运算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 运算符得到的结果如下：

```
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。

数组的地址

对数组取地址时，数组名也不会被解释为其地址。等等，数组名难道不被解释为数组的地址吗？不完全如此：数组名被解释为其第一个元素的地址，而对数组名应用地址运算符时，得到的是整个数组的地址：

```
short tell[10];          // tell an array of 20 bytes
cout << tell << endl;    // displays &tell[0]
cout << &tell << endl;   // displays address of whole array
```

从数字上说，这两个地址相同；但从概念上说，&tell[0]（即 tell）是一个 2 字节内存块的地址，而&tell 是一个 20 字节内存块的地址。因此，表达式 tell + 1 将地址值加 2，而表达式&tell + 2 将地址加 20。换句话说，tell 是一个 short 指针（* short），而&tell 是一个这样的指针，即指向包含 20 个元素的 short 数组（short (*) [20]）。

您可能会问，前面有关&tell 的类型描述是如何来的呢？首先，您可以这样声明和初始化这种指针：

```
short (*pas)[20] = &tell;  // pas points to array of 20 shorts
```

如果省略括号，优先级规则将使得 pas 先与[20]结合，导致 pas 是一个 short 指针数组，它包含 20 个元素，因此括号是必不可少的。其次，如果要描述变量的类型，可将声明中的变量名删除。因此，pas 的类型为 short (*) [20]。另外，由于 pas 被设置为&tell，因此*pas 与 tell 等价，所以(*pas) [0]为 tell 数组的第一个元素。

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。然而，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。

4.8.2　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 * pointerName;*

下面是一些示例：

```
double * pn;       // pn can point to a double value
char * pc;         // pc can point 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&运算符，来获得被命名的内存的地址，new 运算符返回未命名的内存的地址。

下面是一些示例：

```
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 1st element of array of 30 double to pa
```

3. 对指针解除引用

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运算符（*）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。

下面是一些示例：

```
cout << *pn; // print the value of bubble
*pc = 'S';   // place 'S' into the memory location whose address is pc
```

另一种对指针解除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。

下面是一些示例：

```
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。

下面是一个示例：

```
int tacos[10];           // now tacos is the same as &tacos[0]
```

一种例外情况是，将 sizeof 运算符用于数组名用时，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。

下面是一些示例：

```
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[ ]运算符创建数组时，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。

下面是一些示例：

```
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
```

4.8.3　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，该指针指向字符串的开头，稍后将核实这一点。

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，这个用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。

注意：在 cout 和多数 C++表达式中，char 数组名、char 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的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 << "!\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;
}
```

下面是该程序的运行情况：

```
bear and wren
Enter a kind of animal: fox
fox!
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 数组初始化为字符串“bear”，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，如下面的示例所示：

```
cout << "A concerned " << 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（正如您知道的，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。然而，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唯一地存储。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唯一一个副本。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，请注意下述代码完成的工作：

```
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 显示为字符串“fox”，而（int *）ps 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唯一途径。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=运算符；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可能导致问题，因为 food 数组比字符串小：

```
strcpy(food, "a picnic basket filled with many goodies");
```

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。然而，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。

警告：应使用 strcpy( )或 strncpy( )，而不是赋值运算符来将字符串赋给数组。

您对使用 C-风格字符串和 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，并可以使用赋值运算符而不是函数 strcpy( )和 strncpy( )。

4.8.4 使用 new 创建动态结构

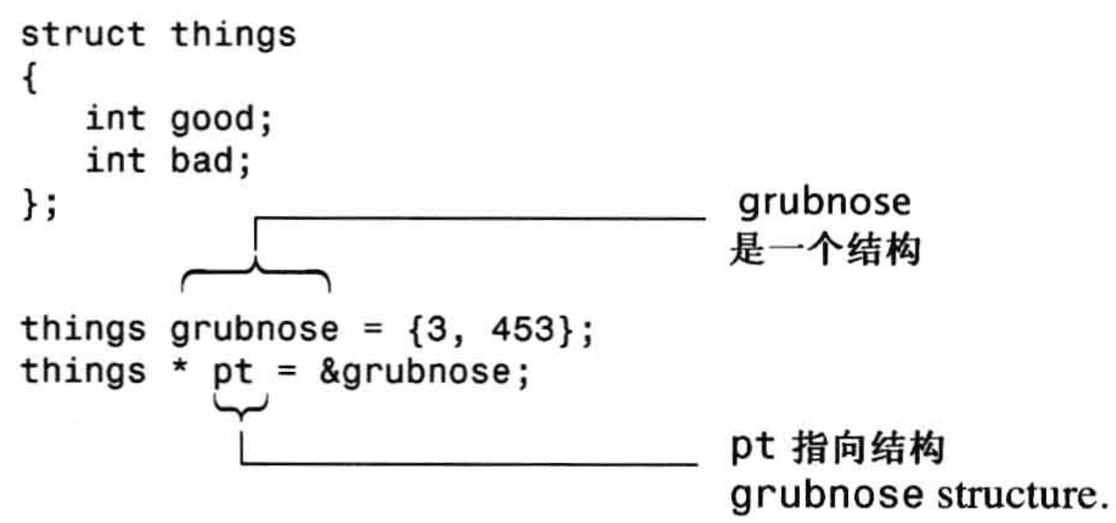
在运行时创建数组优于在编译时创建数组，对于结构也是如此。需要在程序运行时为结构分配所需的空间，这也可以使用 new 运算符来完成。通过使用 new，可以创建动态结构。同样，“动态”意味着内存是在运行时，而不是编译时分配的。由于类与结构非常相似，因此本节介绍的有关结构的技术也适用于类。

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，不能将成员运算符句点用于结构名，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运算符：箭头成员运算符（->）。该运算符由连字符和大于号组成，可用于指向结构的指针，就像点运算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）。



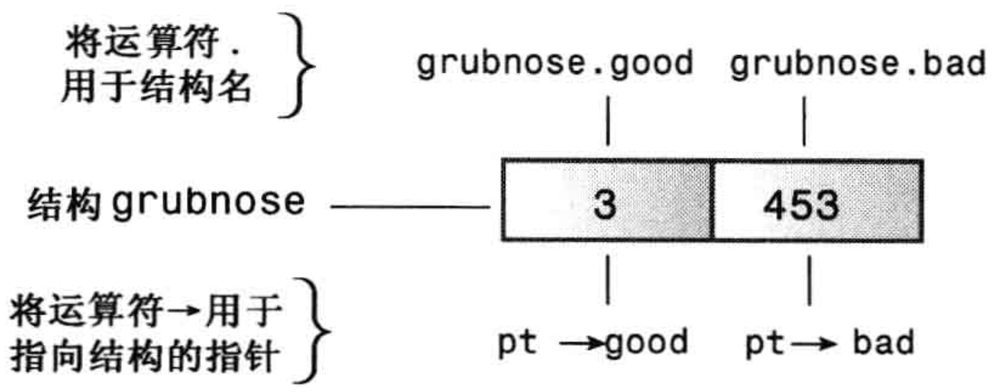


图 4.11 标识结构成员

提示：有时，C++新手在指定结构成员时，搞不清楚何时应使用句点运算符，何时应使用箭头运算符。规则非常简单。如果结构标识符是结构名，则使用句点运算符；如果标识符是指向结构的指针，则使用箭头运算符。

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++的运算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 definition
{
    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 的示例

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示例。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内存（实际编写程序时，使用 string 类将更容易，因为这样可以使用内置的 new 和 delete）。

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 80000 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 strle（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 章做全面介绍。

4.8.5　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。下面简要地介绍每种类型（C++11 新增了第四种类型——线程存储，这将在第 9 章简要地讨论）。

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。然而，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。

自动变量通常存储在栈中。这意味着执行代码块时，其中的变量将依次加入到栈中，而在离开代码块时，将按相反的顺序释放这些变量，这被称为后进先出（LIFO）。因此，在程序执行过程中，栈将不断地增大和缩小。

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++ Release 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。然而，一些您可能已经发现，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 运算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）或堆（heap）。该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 让您能够在一个函数中分配内存，而在另一个函数中释放它。因此，数据的生命周期不完全受程序或函数的生存时间控制。与使用常规变量相比，使用 new 和 delete 让程序员对程序如何使用内存有更大的控制权。然而，内存管理也更复杂了。在栈中，自动添加和删除机制使得占用的内存总是连续的，但 new 和 delete 的相互影响可能导致占用的自由存储区不连续，这使得跟踪新分配内存的位置更困难。

栈、堆和内存泄漏

如果使用 new 运算符在自由存储空间（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 运算符，在自由存储空间上动态分配内存，随后便释放它。C++智能指针有助于自动完成这种任务，这将在第 16 章介绍。

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，就是希望您能够越来越熟悉它。

4.9 类型组合

本章介绍了数组、结构和指针。可以各种方式组合它们，下面介绍其中的一些，从结构开始：

```
struct antarctica_years_end
{
    int year;
 /* some really interesting data, etc. */
};
```

可以创建这种类型的变量：

```
antarctica_years_end s01, s02, s03;  // s01, s02, s03 are structures
```

然后使用成员运算符访问其成员：

```
s01.year = 1998;
```

可创建指向这种结构的指针：

```
antarctica_years_end * pa = &s02;
```

将该指针设置为有效地址后，就可使用间接成员运算符来访问成员：

```
pa->year = 1999;
```

可创建结构数组：

```
antarctica_years_end trio[3]; // array of 3 structures
```

然后，可以使用成员运算符访问元素的成员：

```
trio[0].year = 2003;  // trio[0] is a structure
```

其中 trio 是一个数组，trio[0]是一个结构，而 trio[0].year 是该结构的一个成员。由于数组名是一个指针，因此也可使用间接成员运算符：

```
(trio+1)->year = 2004; // same as trio[1].year = 2004;
```

可创建指针数组：

```
const antarctica_years_end * arp[3] = {&s01, &s02, &s03};
```

咋一看，这有点复杂。如何使用该数组来访问数据呢？既然 arp 是一个指针数组，arp[1]就是一个指针，可将间接成员运算符应用于它，以访问成员：

```
std::cout << arp[1]->year << std::endl;
```

可创建指向上述数组的指针：

```
const antarctica_years_end ** ppa = arp;
```

其中 arp 是一个数组的名称，因此它是第一个元素的地址。但其第一个元素为指针，因此 ppa 是一个指针，指向一个指向 const antarctica_years_end 的指针。这种声明很容易容错。例如，您可能遗漏 const，忘记*，搞错顺序或结构类型。下面的示例演示了 C++11 版本的 auto 提供的方便。编译器知道 arp 的类型，能够正确地推断出 ppb 的类型：

```
auto ppb = arp; // C++11 automatic type deduction
```

在以前，编译器利用它推断的类型来指出声明错误，而现在，您可利用它的这种推断能力。

如何使用 ppa 来访问数据呢？由于 ppa 是一个指向结构指针的指针，因此*ppa 是一个结构指针，可将间接成员运算符应用于它：

```
std::cout << (*ppa)->year << std::endl;
std::cout << (*(ppb+1))->year << std::endl;
```

由于 ppa 指向 arp 的第一个元素，因此*ppa 为第一个元素，即&s01。所以，(*ppa)->year 为 s01 的 year 成员。在第二条语句中，ppb+1 指向下一个元素 arp[1]，即&s02。其中的括号必不可少，这样才能正确地结合。例如，*ppa->year 试图将运算符*应用于 ppa->year，这将导致错误，因为成员 year 不是指针。

上面所有的说法都对吗？程序清单 4.23 将这些语句放到了一个简短的程序中。

程序清单 4.23　mixtypes.cpp

```
// mixtypes.cpp -- some type combinations
#include <iostream>

struct antarctica_years_end
{
    int year;
 /* some really interesting data, etc. */
};

int main()
{
    antarctica_years_end s01, s02, s03;
    s01.year = 1998;
    antarctica_years_end * pa = &s02;
    pa->year = 1999;
    antarctica_years_end trio[3]; // array of 3 structures
    trio[0].year = 2003;
    std::cout << trio->year << std::endl;
    const antarctica_years_end * arp[3] = {&s01, &s02, &s03};
    std::cout << arp[1]->year << std::endl;
    const antarctica_years_end ** ppa = arp;
    auto ppb = arp; // C++11 automatic type deduction
// or else  use const antarctica_years_end ** ppb = arp;
    std::cout << (*ppa)->year << std::endl;
    std::cout << (*(ppb+1))->year << std::endl;
    return 0;
}
```

该程序的输出如下：

```
2003
1999
1998
1999
```

该程序通过了编译，并向前面介绍的那样运行。

4.10 数组的替代品

本章前面说过，模板类 vector 和 array 是数组的替代品。下面简要地介绍它们的用法以及使用它们带来的一些好处。

4.10.1 模板类 vector

模板类 vector 类似于 string 类，也是一种动态数组。您可以在运行阶段设置 vector 对象的长度，可在末尾附加新数据，还可在中间插入新数据。基本上，它是使用 new 创建动态数组的替代品。实际上，vector 类确实使用 new 和 delete 来管理内存，但这种工作是自动完成的。

这里不深入探讨模板类意味着什么，而只介绍一些基本的实用知识。首先，要使用 vector 对象，必须包含头文件 vector。其次，vector 包含在名称空间 std 中，因此您可使用 using 编译指令、using 声明或 std::vector。第三，模板使用不同的语法来指出它存储的数据类型。第四，vector 类使用不同的语法来指定元素数。下面是一些示例：

```
#include <vector>
...
using namespace std;
vector<int> vi;         // create a zero-size array of int
int n;
cin >> n;
vector<double> vd(n);   // create an array of n doubles
```

其中，vi 是一个 vector<int>对象，vd 是一个 vector<double>对象。由于 vector 对象在您插入或添加值时自动调整长度，因此可以将 vi 的初始长度设置为零。但要调整长度，需要使用 vector 包中的各种方法。

一般而言，下面的声明创建一个名为 vt 的 vector 对象，它可存储 n_elem 个类型为 typeName 的元素：

```
vector<typeName> vt(n_elem);
```

其中参数 n_elem 可以是整型常量，也可以是整型变量。

4.10.2 模板类 array（C++11）

vector 类的功能比数组强大，但付出的代价是效率稍低。如果您需要的是长度固定的数组，使用数组是更佳的选择，但代价是不那么方便和安全。有鉴于此，C++11 新增了模板类 array，它也位于名称空间 std 中。与数组一样，array 对象的长度也是固定的，也使用栈（静态内存分配），而不是自由存储区，因此其效率与数组相同，但更方便，更安全。要创建 array 对象，需要包含头文件 array。array 对象的创建语法与 vector 稍有不同：

```
#include <array>
...
using namespace std;
array<int, 5> ai;   // create array object of 5 ints
array<double, 4> ad = {1.2, 2.1, 3.43. 4.3};
```

推而广之，下面的声明创建一个名为 arr 的 array 对象，它包含 n_elem 个类型为 typename 的元素：

```
array<typeName, n_elem> arr;
```

与创建 vector 对象不同的是，n_elem 不能是变量。

在 C++11 中，可将列表初始化用于 vector 和 array 对象，但在 C++98 中，不能对 vector 对象这样做。

4.10.3 比较数组、vector 对象和 array 对象

要了解数组、vector 对象和 array 对象的相似和不同之处，最简单的方式可能是看一个使用它们的简单

示例，如程序清单 4.24 所示。

程序清单 4.24　choices.cpp

```
// choices.cpp -- array variations
#include <iostream>
#include <vector>   // STL C++98
#include <array>    // C++11
int main()
{
    using namespace std;
// C, original C++
    double a1[4] = {1.2, 2.4, 3.6, 4.8};
// C++98 STL
    vector<double> a2(4);   // create vector with 4 elements
// no simple way to initialize in C98
    a2[0] = 1.0/3.0;
    a2[1] = 1.0/5.0;
    a2[2] = 1.0/7.0;
    a2[3] = 1.0/9.0;
// C++11 -- create and initialize array object
    array<double, 4> a3 = {3.14, 2.72, 1.62, 1.41};
    array<double, 4> a4;
    a4 = a3;     // valid for array objects of same size
// use array notation
    cout << "a1[2]: " << a1[2] << " at " << &a1[2] << endl;
    cout << "a2[2]: " << a2[2] << " at " << &a2[2] << endl;
    cout << "a3[2]: " << a3[2] << " at " << &a3[2] << endl;
    cout << "a4[2]: " << a4[2] << " at " << &a4[2] << endl;
// misdeed
    a1[-2] = 20.2;
    cout << "a1[-2]: " << a1[-2] <<" at " << &a1[-2] << endl;
    cout << "a3[2]: " << a3[2] << " at " << &a3[2] << endl;
    cout << "a4[2]: " << a4[2] << " at " << &a4[2] << endl;
    return 0;
}
```

下面是该程序的输出示例：

```
a1[2]: 3.6 at 0x28cce8
a2[2]: 0.142857 at 0xca0328
a3[2]: 1.62 at 0x28ccc8
a4[2]: 1.62 at 0x28cca8
a1[-2]: 20.2 at 0x28ccc8
a3[2]: 20.2 at 0x28ccc8
a4[2]: 1.62 at 0x28cca8
```

程序说明

首先，注意到无论是数组、vector 对象还是 array 对象，都可使用标准数组表示法来访问各个元素。其次，从地址可知，array 对象和数组存储在相同的内存区域（即栈）中，而 vector 对象存储在另一个区域（自由存储区或堆）中。第三，注意到可以将一个 array 对象赋给另一个 array 对象；而对于数组，必须逐元素复制数据。

接下来，下面一行代码需要特别注意：

```
a1[-2] = 20.2;
```

索引-2 是什么意思呢？本章前面说过，这将被转换为如下代码：

```
*(a1-2) = 20.2;
```

其含义如下：找到 a1 指向的地方，向前移两个 double 元素，并将 20.2 存储到目的地。也就是说，将信息存储到数组的外面。与 C 语言一样，C++也不检查这种超界错误。在这个示例中，这个位置位于 array 对象 a3 中。其他编译器可能将 20.2 放在 a4 中，甚至做出更糟糕的选择。这表明数组的行为是不安全的。

vector 和 array 对象能够禁止这种行为吗？如果您让它们禁止，它们就能禁止。也就是说，您仍可编写不安全的代码，如下所示：

```
a2[-2] = .5;    // still allowed
a3[200] = 1.4;
```

然而，您还有其他选择。一种选择是使用成员函数 at()。就像可以使用 cin 对象的成员函数 getline()一样，您也可以使用 vector 和 array 对象的成员函数 at()：

```
a2.at(1) = 2.3;  // assign 2.3 to a2[1]
```

中括号表示法和成员函数 at()的差别在于，使用 at()时，将在运行期间捕获非法索引，而程序默认将中断。这种额外检查的代价是运行时间更长，这就是 C++让允许您使用任何一种表示法的原因所在。另外，这些类还让您能够降低意外超界错误的概率。例如，它们包含成员函数 begin()和 end()，让您能够确定边界，以免无意间超界，这将在第 16 章讨论。

4.11 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，可以使用成员关系运算符（.）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。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运算符，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，用户可以使用赋值运算符来复制字符串。

new 运算符允许在程序运行时为数据对象请求内存。该运算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运算符（*）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运算符（->）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。

运算符 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。

C++98 新增的标准模板库（STL）提供了模板类 vector，它是动态数组的替代品。C++11 提供了模板类 array，它是定长数组的替代品。

4.12　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．使用模板类 array 而不是数组来完成问题 1。

3．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。

4．编写一条语句，将问题 3 中数组第一个元素和最后一个元素的和赋给变量 even。

5．编写一条语句，显示 float 数组 ideas 中的第 2 个元素的值。

6．声明一个 char 的数组，并将其初始化为字符串“cheeseburger”。

7．声明一个 string 对象，并将其初始化为字符串“Waldorf Salad”。

8．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）。

9．声明一个问题 8 中定义的结构的变量，并对它进行初始化。

10．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。

11．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。

12．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。

13．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。首先使用 new 来完成这项任务，再使用 vector 对象来完成这项任务。

14．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";
```

15．编写一段代码，给问题 8 中描述的结构动态分配内存，再读取该结构的成员的值。

16．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？

17．声明一个 vector 对象和一个 array 对象，它们都包含 10 个 string 对象。指出所需的头文件，但不要使用 using。使用 const 来指定要包含的 string 对象数。

4.13　编程练习

1．编写一个 C++程序，如下述输出示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? Yewe
What letter grade do you deserve? B
What is your age? 22
Name: Yewe, 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，再输入其姓；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。对于每个披萨饼，他都需要记录下列信息：

- 披萨饼公司的名称，可以有多个单词组成。
- 披萨饼的直径。
- 披萨饼的重量。

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。

10．编写一个程序，让用户输入三次 40 码跑的成绩（如果您愿意，也可让用户输入 40 米跑的成绩），并显示次数和平均成绩。请使用一个 array 对象来存储数据（如果编译器不支持 array 类，请使用数组）。

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。
- 递增运算符和递减运算符：++和—。
- 组合赋值运算符。
- 复合语句（语句块）。
- 逗号运算符。
- 关系运算符：>、>=、==、<=、<和!=。
- while 循环。
- typedef 工具。
- do while 循环。
- 字符输入方法 get( )。
- 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，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++运算符——递增运算符（increment operator），它将操作数的值加 1。递增运算符并不仅限于用于 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，很容易识别。初始化、测试和更新操作构成了控制部分，这些操作由括号括起。其中每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。然而，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避免程序遇到麻烦。

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。然而，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（这是第一个循环示例所做的工作），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。然而，for 是一个 C++关键字，因此编译器不会将 for 视为一个函数，这还将防止将函数命名为 for。

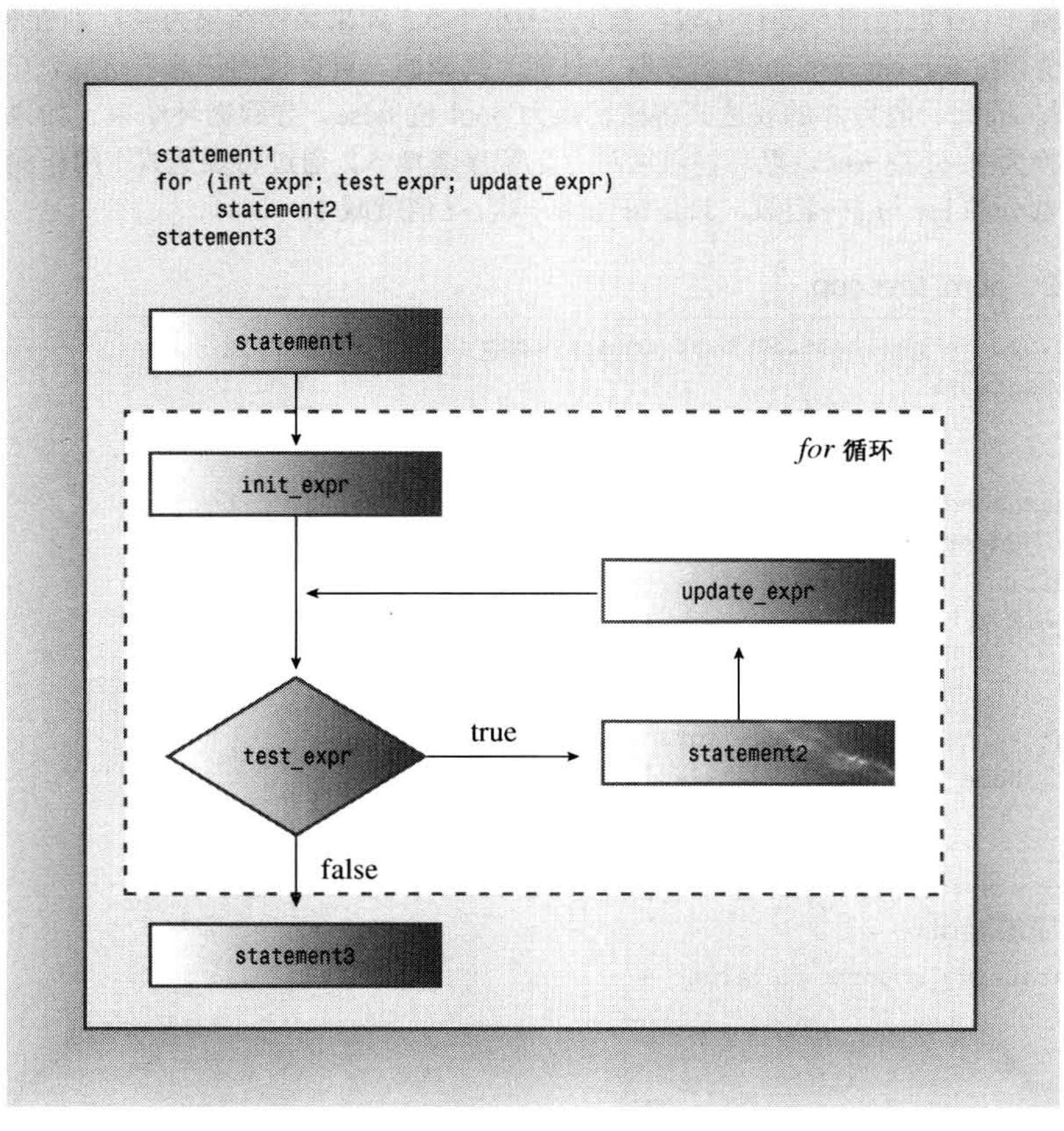


图 5.1 for 循环

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 (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。

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运算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，下面的表达式由两个值和一个加号组成，它的值为 49：

```
22 + 27
```

有时值不这么明显，例如，下面是一个表达式，因为它由两个值和一个赋值运算符组成：

```
x = 20
```

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。然而，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，赋值运算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。<<运算符的优先级比表达式中使用的运算符高，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>
int main()
{
    using namespace std;
    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。

注意：C++表达式是值或值与运算符的组合，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。然而，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很容易，只要加分号即可。因此下面是一个表达式：

```
age = 100
```

而下面是一条语句：

```
age = 100;
```

更准确地说，这是一条表达式语句。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 是个变量，则下面就是一条有效的 C++语句：

```
rodents + 6;      // valid, but useless, statement
```

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。在下面的示例中，for 循环不是表达式，因此没有值，也不能给它赋值：

```
int fx = for (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。然而，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。然而，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
```

您还应知道的一点是，有些较老的 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。

5.1.2　回到 for 循环

下面使用 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 phfffft pptz，带有濡湿的口音。然而，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>
const int ArSize = 16;      // example of external declaration
int main()
{
    long long factorials[ArSize];
    factorials[1] = factorials[0] = 1LL;
    for (int i = 2; i < ArSize; i++)
        factorials[i] = i * factorials[i-1];
    for (int i = 0; i < ArSize; i++)
        std::cout << i << "! = " << factorials[i] << std::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! = 3628800
11! = 39916800
12! = 479001600
13! = 6227020800
14! = 87178291200
15! = 1307674368000
```

阶乘增加得很快！

注意：这个程序清单使用了类型 long long。如果您的系统不支持这种类型，可使用 double。然而，整型使得阶乘的增大方式看起来更明显。

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 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。

另外，这个示例还提醒您，可使用 std::而不是编译指令 using 来让选定的标准名称可用。

5.1.3 修改步长

到现在为止，循环示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 std::cout;   // a using declaration
    using std::cin;
    using std::endl;
    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。

需要指出的另一点是，检测不等通常比检测相等好。例如，在这里使用条件 i == 100 不可行，因为 i 的取值不会为 100。

最后，这个示例使用了 using 声明，而不是 using 编译指令。

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，程序使用递减运算符（--），在每轮循环后将数组下标减 1。另外，程序清单 5.6 使用关系运算符大于或等于（>=）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运算符做一总结。

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;
}
```

注意：如果所用的实现没有添加新的头文件，则必须使用 string.h，而不是 cstring。

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 rotator、redder 或 stats 相比，animal 能更清晰地说明这个程序的作用。

5.1.5　递增运算符（++）和递减运算符（--）

C++中有多个常被用在循环中的运算符，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运算符：递增运算符（++）（名称 C++由此得到）和递减运算符（--）。这两个运算符执行两种极其常见的循环操作：将循环计数加 1 或减 1。然而，它们还有很多特点不为读者所知。这两个运算符都有两种变体。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 演示递增运算符的这种差别。

程序清单 5.7 plus_one.cpp

```
// plus_one.cpp -- the increment operator
#include <iostream>
int main()
{

    using std::cout;
    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
```

递增和递减运算符是处理将值加减 1 这种常见任务的一种简约、方便的方法。

递增运算符和递减运算符都是漂亮的小型运算符，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，下面这条语句在不同的系统上将生成不同的结果：

```
x = 2 * x++ * (3 - ++x);    // don't do it except as an experiment
```

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++就递增运算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，赋值运算符、递增运算符和递减运算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)
    cout << guests << endl;
```

while 循环将在本章后面讨论，它类似于只有测试表达式的 for 循环。在这里，C++新手可能认为“使用值，然后递增”意味着先在 cout 语句中使用 guests 的值，再将其值加 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 cout 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。

在 C++11 文档中，不再使用术语“顺序点”了，因为这个概念难以用于讨论多线程执行。相反，使用了术语“顺序”，它表示有些事件在其他事件前发生。这种描述方法并非要改变规则，而旨在更清晰地描述多线程编程。

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，使用这些运算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运算符，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副本，将其加 1，然后将复制的副本返回。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；但对于用户定义的类型，如果有用户定义的递增和递减运算符，则前缀格式的效率更高。

5.1.8　递增/递减运算符和指针

可以将递增运算符用于指针和基本变量。本书前面介绍过，将递增运算符用于指针时，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运算符和*运算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。这取决于运算符的位置和优先级。前缀递增、前缀递减和解除引用运算符的优先级相同，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，但比前缀运算符的优先级高，这两个运算符以从左到右的方式进行结合。

前缀运算符的从右到到结合规则意味着*++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：

```
double x = *++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，运算符++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：

```
x = *pt++;        // dereference original location, then increment pointer
```

后缀运算符++的优先级更高，这意味着将运算符用于 pt，而不是*pt，因此对指针递增。然而后缀运算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。

注意：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 组合赋值运算符

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运算符，能够更简洁地完成这种任务：

```
i += by
```

+=运算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
```

每个算术运算符都有其对应的组合赋值运算符，表 5.1 对它们进行了总结。其中每个运算符的工作方式都和+=相似。因此，下面的语句将 k 与 10 相乘，再将结果赋给 k：

```
k *= 10;
```

表 5.1　组合赋值运算符

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+=运算符。

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，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>
int main()
{
    using namespace std;
    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；然后就变量再次可见，如下例所示：

```
#include <iostream>
int main()
{
    using std::cout;
    using std::endl;
    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 其他语法技巧——逗号运算符

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。逗号运算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，可以使用逗号运算符将两个表达式合并为一个：

```
++j, --i   // two expressions count as one for syntax purposes
```

逗号并不总是逗号运算符。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运算符，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 <string>
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: stressed
desserts
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，它使用逗号运算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，再次使用逗号运算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，则便开始将交换后的元素回到原来的位置（参见图 5.2）。

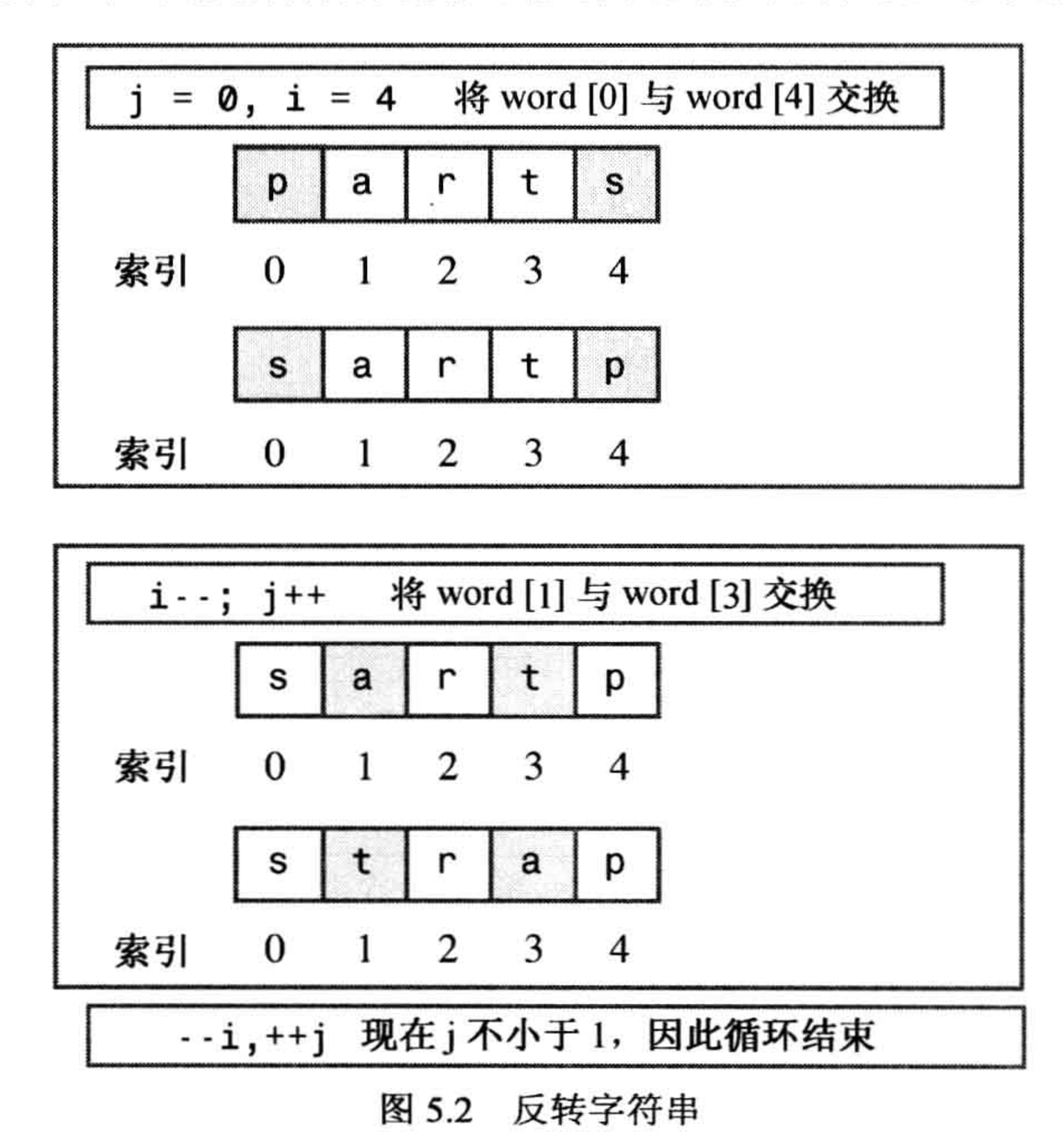


图 5.2　反转字符串

需要注意的另一点是，声明变量 temp、i、j 的位置。代码在循环之前声明 i 和 j，因为不能用逗号运算

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，而不是逗号运算符，因此该表达式对 j 和 i 进行声明和初始化。然而，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. 逗号运算符花絮

到目前为止，逗号运算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++还为这个运算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，逗号运算符是一个顺序点）。如下所示的表达式是安全的：

```
i = 20, j = 2 * i       // i set to 20, then 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。

在所有运算符中，逗号运算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。然而，由于括号的优先级最高，下面的表达式将把 cats 设置为 240——逗号右侧的表达式值：

```
cats = (17,240);
```

5.1.12 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，关系运算符是这种能力的体现。C++提供了 6 种关系运算符来对数字进行比较。由于字符用其 ASCII 码表示，因此也可以将这些运算符用于字符。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 对这些运算符进行了总结。

表 5.2 关系运算符

操 作 符	含 义
<	小于
<=	小于或等于
==	等于
>	大于
>=	大于或等于
!=	不等于

这 6 种关系运算符可以在 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，则这里的运算符就派不上用场了。

下面是一些测试示例：

```
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
```

关系运算符的优先级比算术运算符低。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。

5.1.13　赋值、比较和可能犯的错误

不要混淆等于运算符（= =）与赋值运算符（=）。下面的表达式问了一个音乐问题——musicians 是否等于 4？

```
musicians == 4     // comparison
```

该表达式的值为 true 或 false。下面的表达式将 4 赋给 musicians：

```
musicians = 4      // assignment
```

在这里，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= =运算符中的一个等号，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运算符的循环。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";
// Warning: you may prefer reading about this program
// to actually running it.
    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。最糟糕的是，该程序可能导致其他应用程序无法运行，您必须重新启动计算机。

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（然而，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。然而，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运算符将两个这样的测试合并为一个条件。

5.1.14 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，因此如果使用关系运算符来比较它们，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。

在有些语言（如 BASIC 和标准 Pascal）中，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
```

顺便说一句，虽然不能用关系运算符来比较字符串，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 字符集和 Unicode 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　compstr1.cpp

```
// compstr1.cpp -- comparing strings using arrays
#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;
}
```

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。

检测相等或排列顺序：

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>运算符的角色。

接下来，compstr1.cpp 使用递增运算符使变量 ch 遍历字母表：

```
ch++
```

可以对字符变量使用递增运算符和递减运算符，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;
```

5.1.15 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运算符进行比较。这之所以可行，是因为类函数重载（重新定义）了这些运算符。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运算符用于 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 compstr2.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 != "mate"; ch++)
    {
        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，下面的测试条件使用了一个关系运算符，该运算符的左边是一个 string 对象，右边是一个 C-风格字符串：

```
word != "mate"
```

string 类重载运算符!=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。

正如您看到的，使用 C-风格字符串和 string 对象可获得相同的结果，但使用 string 对象更简单、更直观。

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。

5.2　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，对它进行重新评估。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。

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 C-风格字符串的标准方法。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;
}
```

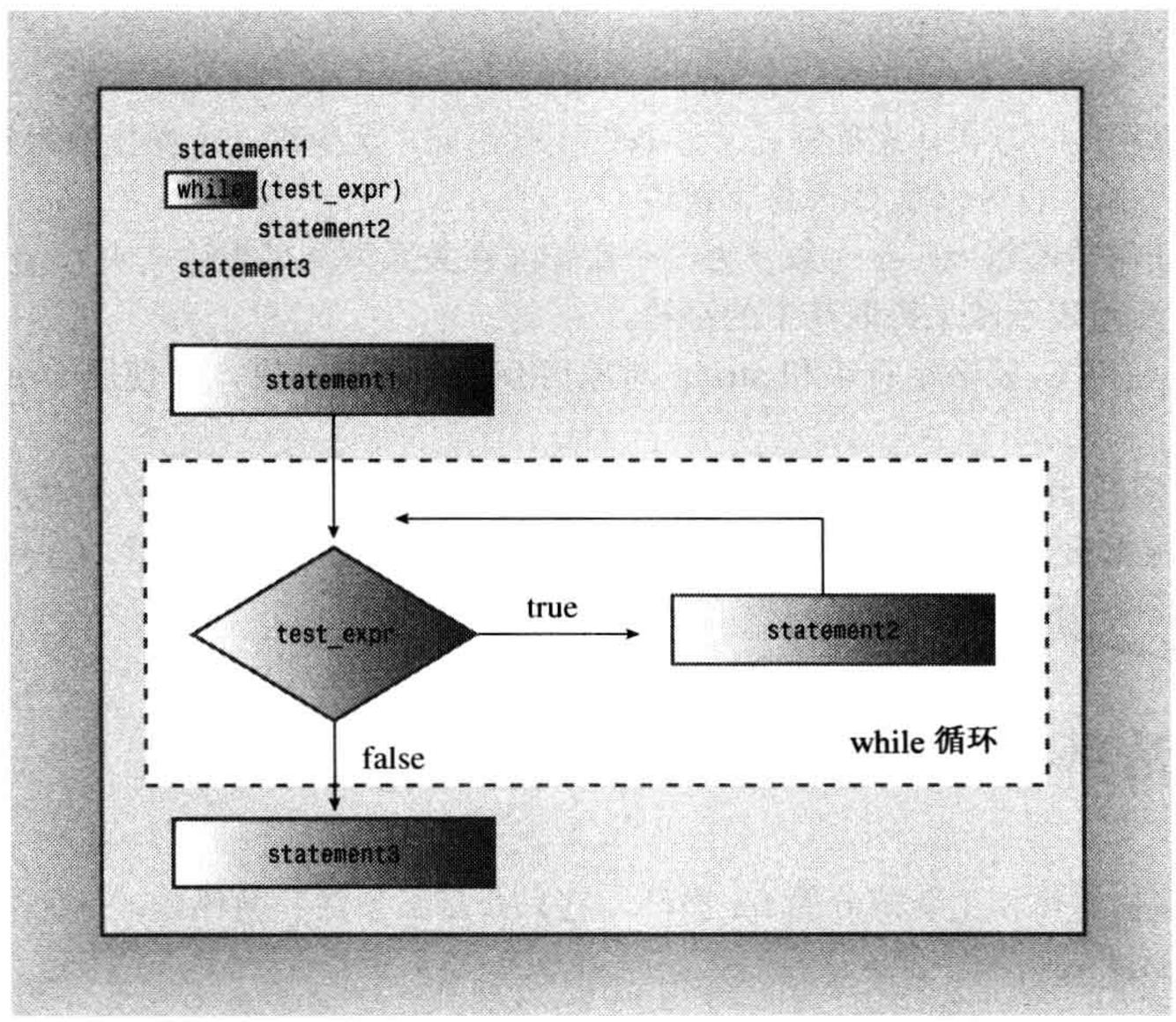


图 5.3 while 循环的结构

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。然而，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 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。

5.2.1　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
```

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。它们之间存在三个差别。首先，在 for 循环中省略了测试条件时，将认为条件为 true；其次，在 for 循环中，可使用初始化语句声明一个局部变量，但在 while 循环中不能这样做；最后，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。

- 指定循环终止的条件。

- 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。但这些指导原则也适用于 while 循环。

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。然而，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。

5.2.2 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；另外，有些编译器可能修改上述代码，将 wait 设置为 10000，从而跳过该循环。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。

但头文件 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。

程序清单 5.14 演示了如何使用 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，这也是唯一的选择。

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。

5.3 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。

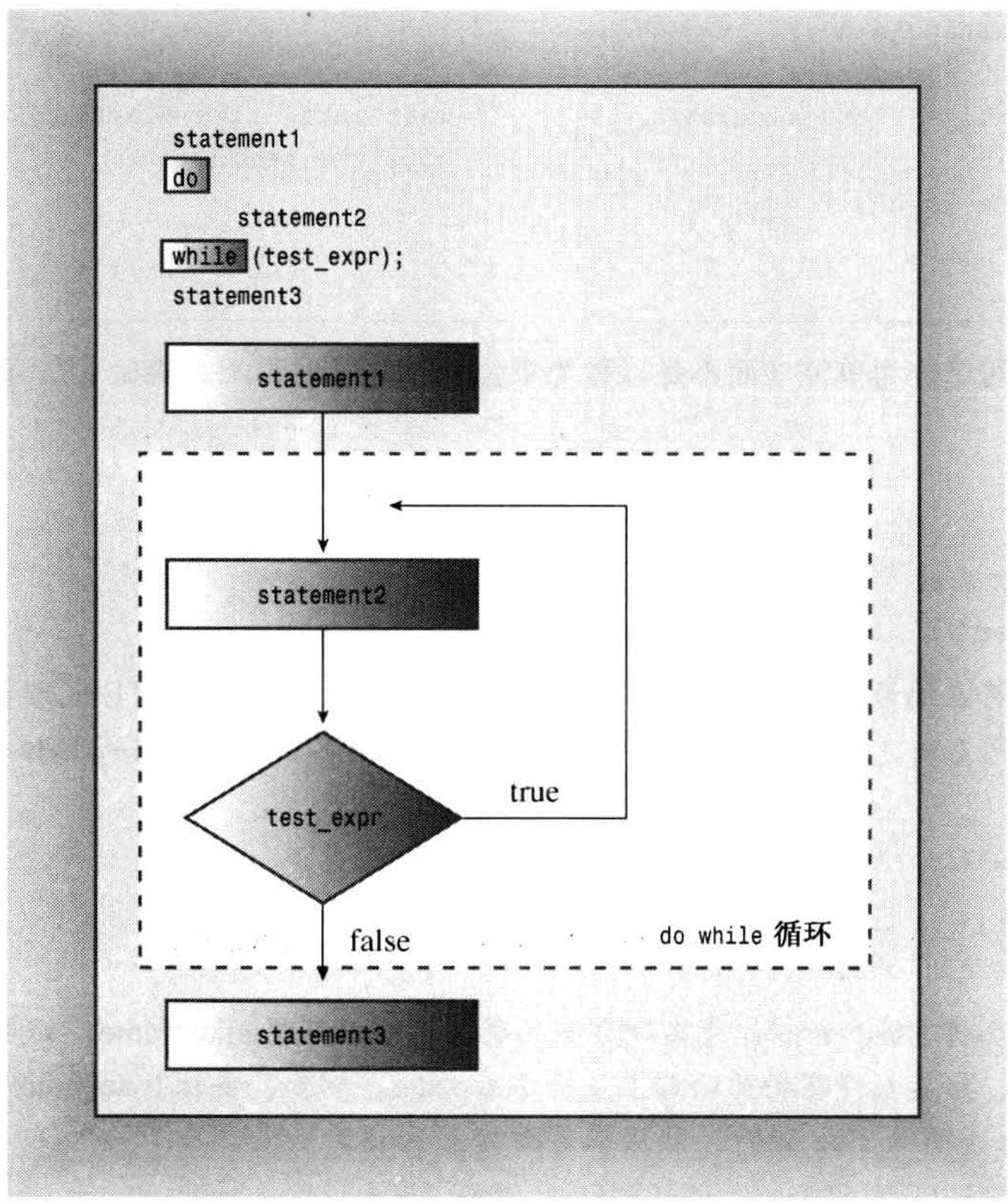


图 5.4 do while 循环的结构

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.
```

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，：

```
int I = 0;
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：

```
int I = 0;
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：

```
int I = 0;
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。

5.4 基于范围的 for 循环（C++11）

C++11 新增了一种循环：基于范围（range-based）的 for 循环。这简化了一种常见的循环任务：对数组（或容器类，如 vector 和 array）的每个元素执行相同的操作，如下例所示：

```
double prices[5] = {4.99, 10.99, 6.87, 7.99, 8.49};
for (double x : prices)
    cout << x << std::endl;
```

其中，x 最初表示数组 prices 的第一个元素。显示第一个元素后，不断执行循环，而 x 依次表示数组的其他元素。因此，上述代码显示全部 5 个元素，每个元素占据一行。总之，该循环显示数组中的每个值。

要修改数组的元素，需要使用不同的循环变量语法：

```
for (double &x : prices)
    x = x * 0.80;        //20% off sale
```

符号&表明 x 是一个引用变量，这个主题将在第 8 章讨论。就这里而言，这种声明让接下来的代码能够修改数组的内容，而第一种语法不能。

还可结合使用基于范围的 for 循环和初始化列表：

```
for (int x : {3, 5, 2, 8, 6})
    cout << x << " ";
cout << '\n';
```

然而，这种循环主要用于第 16 章将讨论的各种模板容器类。

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 循环中使用这三种模式。

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。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：

```
Enter characters; enter # to quit:
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++在 C 语言的基础上新增的一种类型。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。然而，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( )示例，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、Linux 和 Windows 命令提示符模式）都支持重定向，允许用文件替换键盘输入。例如，假设在 Windows 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，则可以在命令提示符模式下输入下面的命令：

```
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 Windows 命令提示符模式的重定向运算符。

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 Windows 命令提示符模式下，可以在任意位置按 Ctrl+Z 和 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。然而，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++、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。然而，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。也可使用 cin.clear( )来重置输入流，这将在第 6 章和第 17 章介绍。

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><ENTER>
73 characters read
```

这里在 Windows 7 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。请注意，在 Unix 和类 Unix（包括 Linux 和 Cygwin）系统中，用户应按 Ctrl+Z 组合键将程序挂起，而命令 fg 恢复执行程序。

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.
48 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。然而，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!运算符可以将 true 切换为 false 或将 false 切换为 true。可以使用此运算符将上述 while 测试改写成这样：

```
while (!cin.fail())     // while input has not failed
```

方法 cin.get(char)的返回值是一个 cin 对象。然而，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。三条指导原则（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。

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。然而，有些 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，上述示例将无法正常运行。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。然而，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。还可使用第 17 章将介绍的 cin.clear( )来重置输入流。

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.
<CTRL>+<Z><ENTER>
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!=运算符的优先级高于=，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，并本质赋给 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 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 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。

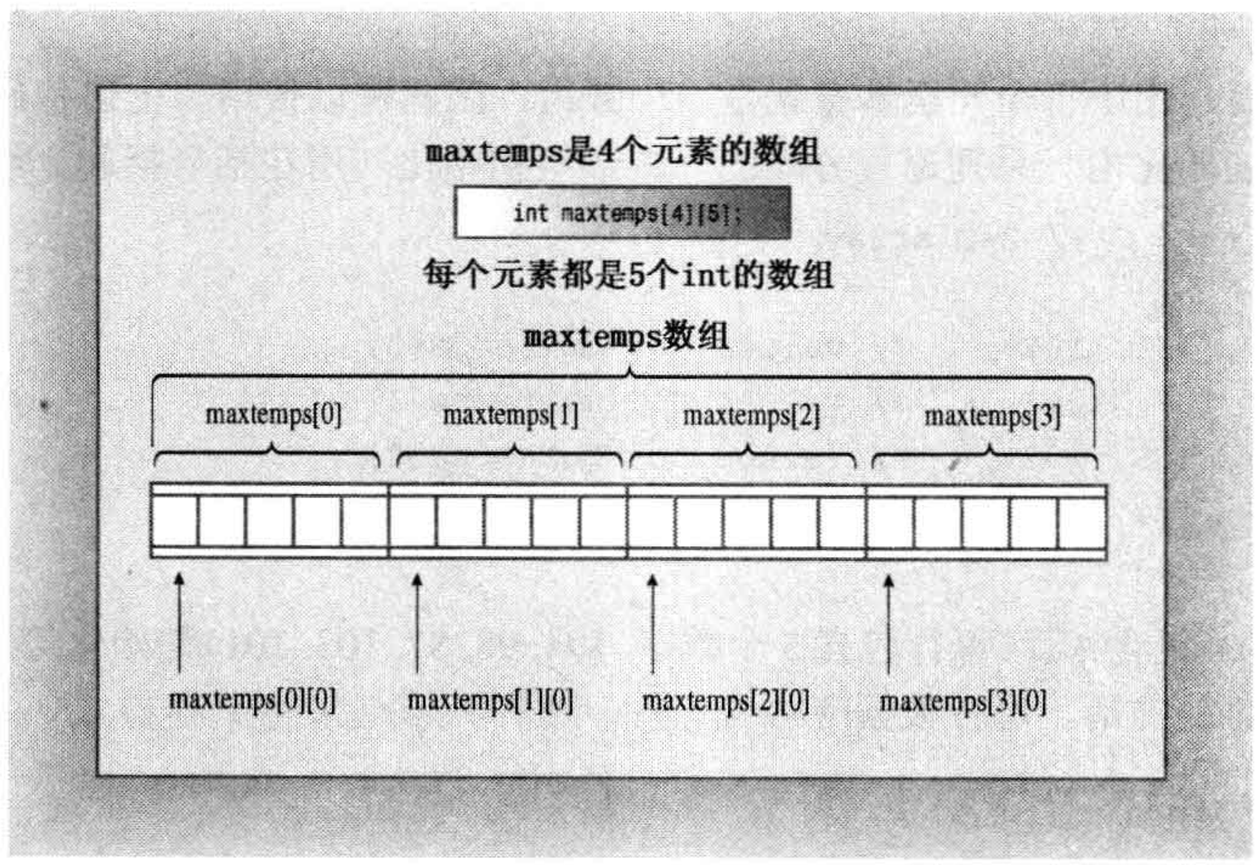


图 5.5　由数组组成的数组

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）。

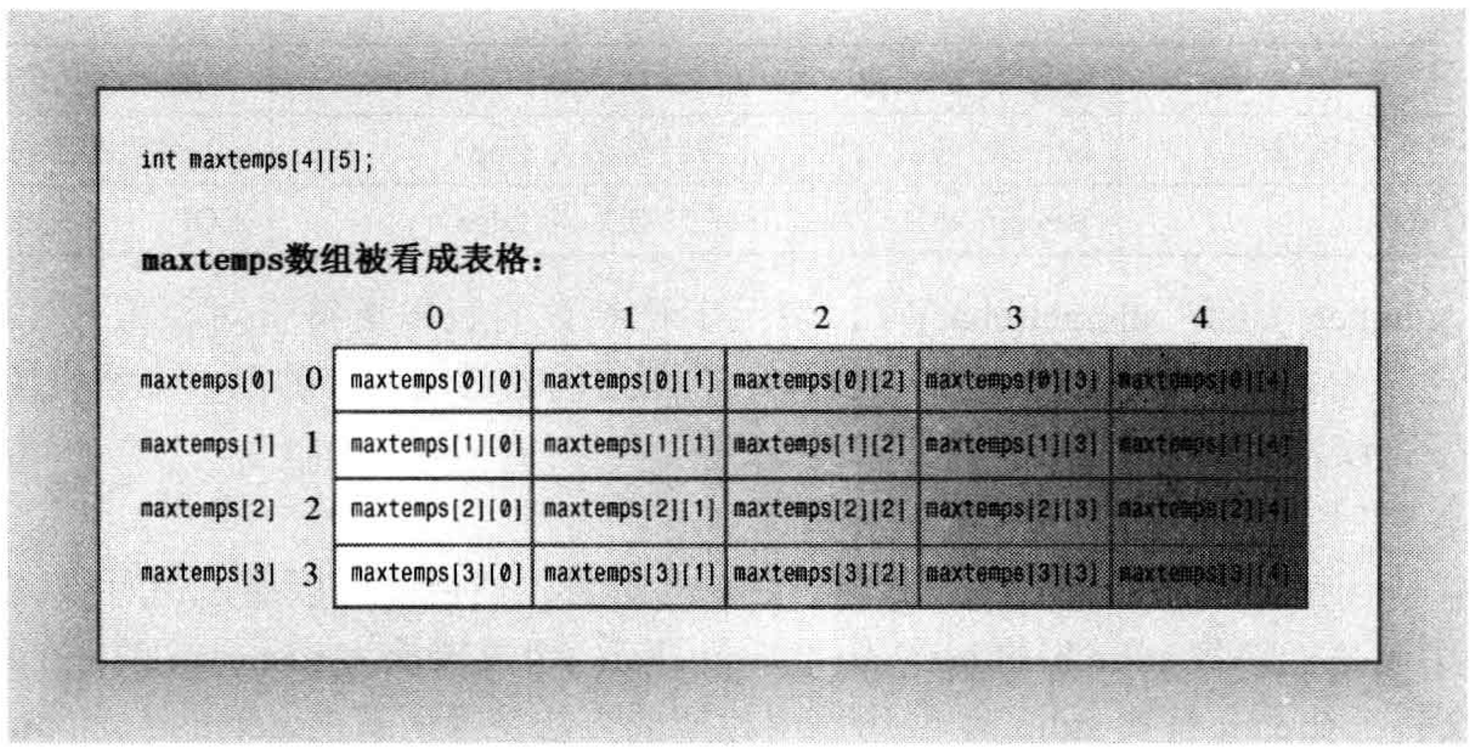


图 5.6 使用下标访问数组元素

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;
}
```

对于每个 row 值，内部的 for 循环将遍历所有的 col 值。这个示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。

5.6.1 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
{
    {96, 100, 87, 101, 105},   // values for maxtemps[0]
    {96, 98, 91, 107, 104},    // values for maxtemps[1]
    {97, 101, 93, 108, 107},   // values for maxtemps[2]
    {98, 103, 95, 109, 108}    // values for maxtemps[3]
};
```

可将数组 maxtemps 包含 4 行，每行包含 5 个数字。{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。

5.6.2 使用二维数组

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
    {
        {96, 100, 87, 101, 105},   // values for maxtemps[0]
        {96, 98, 91, 107, 104},    // values for maxtemps[1]
        {97, 101, 93, 108, 107},   // values for maxtemps[2]
        {98, 103, 95, 109, 108}    // values for maxtemps[3]
    };

    cout << "Maximum temperatures for 2008 - 2011\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;
    }
        // cin.get();
    return 0;
}
```

下面是该程序的输出：

```
Maximum temperatures for 2008 - 2011

Gribble City:   96      96      97      98
Gribbletown:    100     98      101     103
New Gribble:    87      91      93      95
San Gribble:    101     107     108     109
Gribble Vista:  105     104     107     108
```

在输出中使用制表符比使用空格可使数据排列更有规则。然而，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[Cities][25] =   // array of 5 arrays of 25 char
{
    "Gribble City",
    "Gribbletown",
    "New Gribble",
    "San Gribble",
    "Gribble Vista"
};
```

上述方法将全部 5 个字符串的最大长度限制为 24 个字符。指针数组存储 5 个字符串的地址，而使用 char 数组的数组时，将 5 个字符串分别复制到 5 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。然而，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 种关系运算符之一构成的：<、<=、==、>=、>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，则下面的语句将输入中的下一个字符读入到 ch 中：

```
cin >> ch;
```

然而，它将忽略空格、换行符和制表符。下面的成员函数调用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：

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

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. 使用 array 对象（而不是数组）和 long double（而不是 long long）重新编写程序清单 5.4，并计算 100!的值。

3. 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。

4. 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。

5．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。

6．完成编程练习 5，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三年的总销售量。

7．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
```

8．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。

9．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，并使用关系运算符来进行比较测试。

10．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　分支语句和逻辑运算符

本章内容包括：

- if 语句。
- if else 语句。
- 逻辑运算符：&&、||和!。
- cctype 字符函数库。
- 条件运算符：?:。
- 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，还将介绍条件运算符和逻辑运算符，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，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 语句的语法与 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。

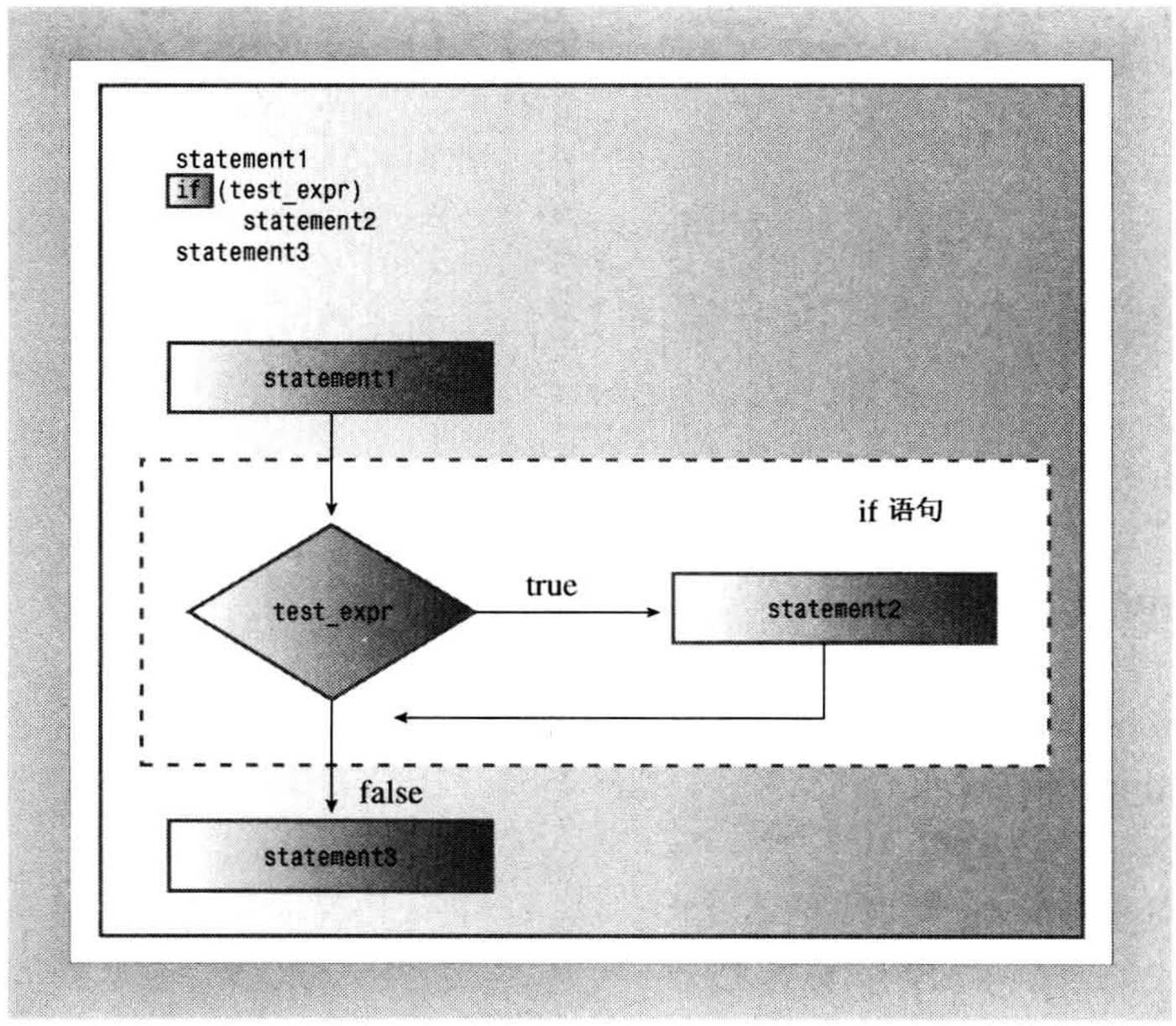


图 6.1 if 语句的结构

程序清单 6.1 if.cpp

```
// if.cpp -- using the if statement
#include <iostream>
int main()
{
    using std::cin;      // using declarations
    using std::cout;
    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，则下面的代码片段将打印第一条信息，否则打印第二条信息：

```
if (answer == 1492)
    cout << "That's right!\n";
else
    cout << "You'd better review Chapter 1 again.\n";
```

每条语句都既可以是一条语句，也可以是用大括号括起的语句块（参见图 6.2）。从语法上看，整个 if else 结构被视为一条语句。

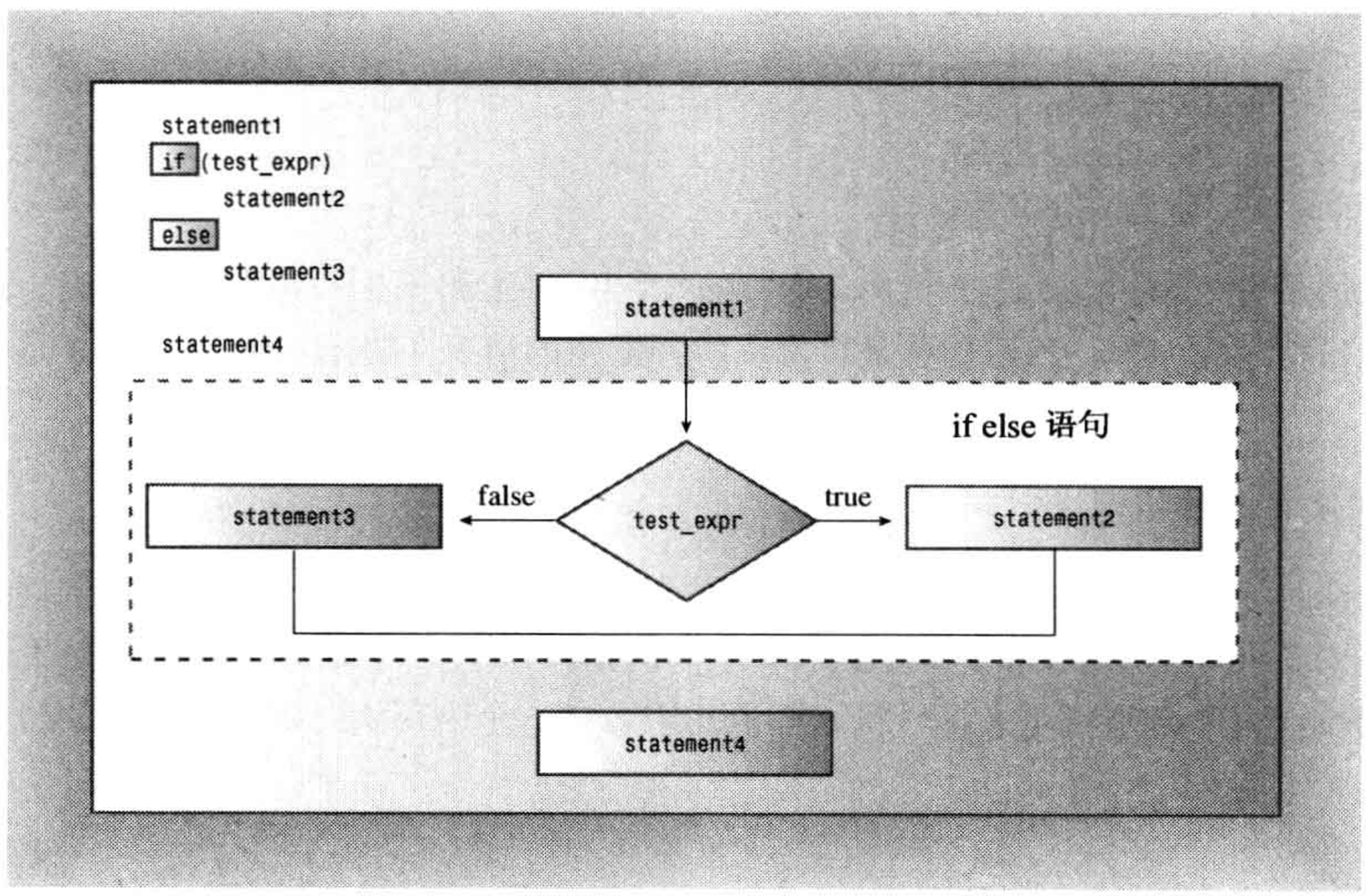


图 6.2　if else 语句的结构

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。该程序清单还演示了限定符 std::，这是编译指令 using 的替代品之一。

程序清单 6.2 ifelse.cpp

```
// ifelse.cpp -- using the if else statement
#include <iostream>
int main()
{
    char ch;

    std::cout << "Type, and I shall repeat.\n";
    std::cin.get(ch);
    while (ch != '.')
    {
        if (ch == '\n')
            std::cout << ch;     // done if newline
        else
            std::cout << ++ch;   // done otherwise
        std::cin.get(ch);
    }
// try ch + 1 instead of ++ch for interesting effect
    std::cout << "\nPlease excuse the slight confusion.\n
        //  std::cin.get();
        //  std::cin.get();
    return 0;
}
```

下面是该程序的运行情况：

```
Type, and I shall repeat.
An ineffable joy suffused me as I beheld
Bo!jofggbcmf!kpz!tvggvtfe!nf!bt!J!cfifme
the wonders of modern computing.
uif!xpoefst!pg!npefso!dpnqvujoh
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，需要用大括号将它们括起来，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，因此必须使用大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，这被视为语法错误。

请添加大括号，将语句组合成一个语句块：

```
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，因此只要使用大括号将语句括起，对大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；然而，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!
```

条件运算符和错误防范

许多程序员将更直观的表达式 variable = =value 反转为 value = =variable，以此来捕获将相等运算符误写为赋值运算符的错误。例如，下述条件有效，可以正常工作：

```
if (3 == myNumber)
```

但如果错误地使用下面的条件，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）：

```
if (3 = myNumber)
```

假设犯了类似的错误，但使用的是前一种表示方法：

```
if (myNumber = 3)
```

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（然而，很多编译器会发出警告，因此注意警告是明智的）。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，其值就必须大于或等于'a'，且小于或等于'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 种逻辑运算符，来组合或修改已有的表达式。这些运算符分别是逻辑 OR（||）、逻辑 AND（&&）

和逻辑 NOT（!）。下面介绍这些运算符。

6.2.1　逻辑 OR 运算符：||

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 运算符（||），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||的优先级比关系运算符低，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。

C++规定，||运算符是个顺序点（sequence point）。也是说，先修改左侧的值，再对右侧的值进行判定（C++11 的说法是，运算符左边的子表达式先于右边的子表达式）。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，冒号和逗号运算符也是顺序点）。

程序清单 6.4 在一条 if 语句中使用||运算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。

表 6.1　　||运算符

	expr1 || expr2 的值	
	expr1 = = true	expr1 = = false
expr2 = = true	true	true
expr2 = = false	true	false

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 n! Apparently you "
            "can't follow\ninstructions, so "
            "I'll trash your disk anyway.\a\a\a\n";
    return 0;
}
```

该程序不会带来任何威胁，下面是其运行情况：

```
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。然而，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 运算符：&&

逻辑 AND 运算符（&&），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&&的优先级低于关系运算符，因此不必在这些表达式中使用括号。和||运算符一样，&&运算符也是顺序点，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&&运算符的工作方式。

表 6.2 &&运算符

	expr1 && expr2 的值	
	expr1 = = true	expr1 = = false
expr2 = = true	true	false
expr2 = = false	false	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&&运算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: 28
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.
```

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。

6.2.3 用&&来设置取值范围

&&运算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 the 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;
}
```

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。

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 岁到 34 岁之间。表达式 age >= 35 && age < 50 使用<=运算符将 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（参见本节后面的旁注“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 运算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++语法。<运算符从左向右结合，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!

6.2.4　逻辑 NOT 运算符：!

!运算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，不使用这个运算符可以更清楚地表示关系：

```
if (!(x > 5))                    // if (x <= 5) is clearer
```

然而，!运算符对于返回 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!运算符用于函数返回值）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( )（稍后将详细介绍）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;
}
```

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，且取值范围更大。另一种选择是，使用 long long 来存储输入的值，因为其取值范围比 int 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。

6.2.5 逻辑运算符细节

正如本章前面指出的，C++逻辑 OR 和逻辑 AND 运算符的优先级都低于关系运算符。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，!运算符的优先级高于所有的关系运算符和算术运算符。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 运算符的优先级高于逻辑 OR 运算符。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 31～44，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++运算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。

6.2.6　其他表示方式

并不是所有的键盘都提供了用作逻辑运算符的符号，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运算符，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　逻辑运算符：另一种表示方式

运算符	另一种表示方式
&&	and
||	or
!	not

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 运算符更方便。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　cctypes.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 sentinel
    {
        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.
AdrenalVision International producer Adrienne Vismonger
announced production of their new 3-D film, a remake of
"My Dinner with Andre," scheduled for 2013. "Wait until
you see the the new scene with an enraged Collossipede!"@
177 letters, 33 whitespace, 5 digits, 9 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
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　?:运算符

C++有一个常被用来代替 if else 语句的运算符，这个运算符被称为条件运算符（?:），它是 C++中唯一一个需要 3 个操作数的运算符。该运算符的通用格式如下：

```
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。下面的两个示例演示了该运算符是如何工作的：

```
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
```

可以这样解释第一个示例：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 使用条件运算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 integers: 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，条件运算符更简洁，但第一次遇到时不那么容易理解。这两种方法之间的区别是，条件运算符生成一个表达式，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。条件运算符格式简洁、语法奇特、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，条件运算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 个选项中选择一个，例如，便宜、适中、昂贵、奢侈、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。

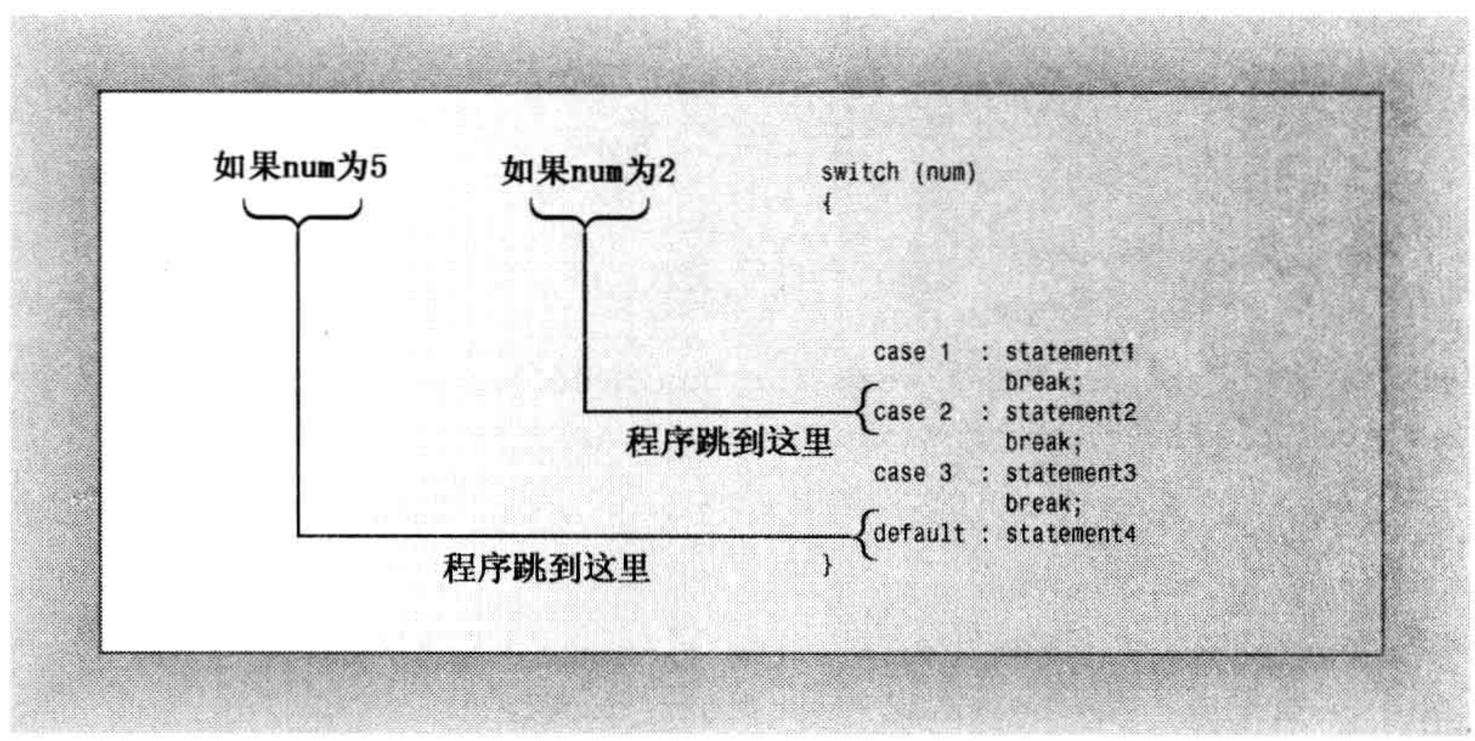


图 6.3　switch 语句的结构

注意：有些硬件/操作系统组合不会将（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。

为让这个程序正确运行，输入必须是整数。例如，如果输入一个字母，输入语句将失效，导致循环不断运行，直到您终止程序。为应对不按指示办事的用户，最好使用字符输入。

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。如果选项涉及取值范围、浮点测试或两个变量的比较，则应使用 if else 语句。

然而，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）。

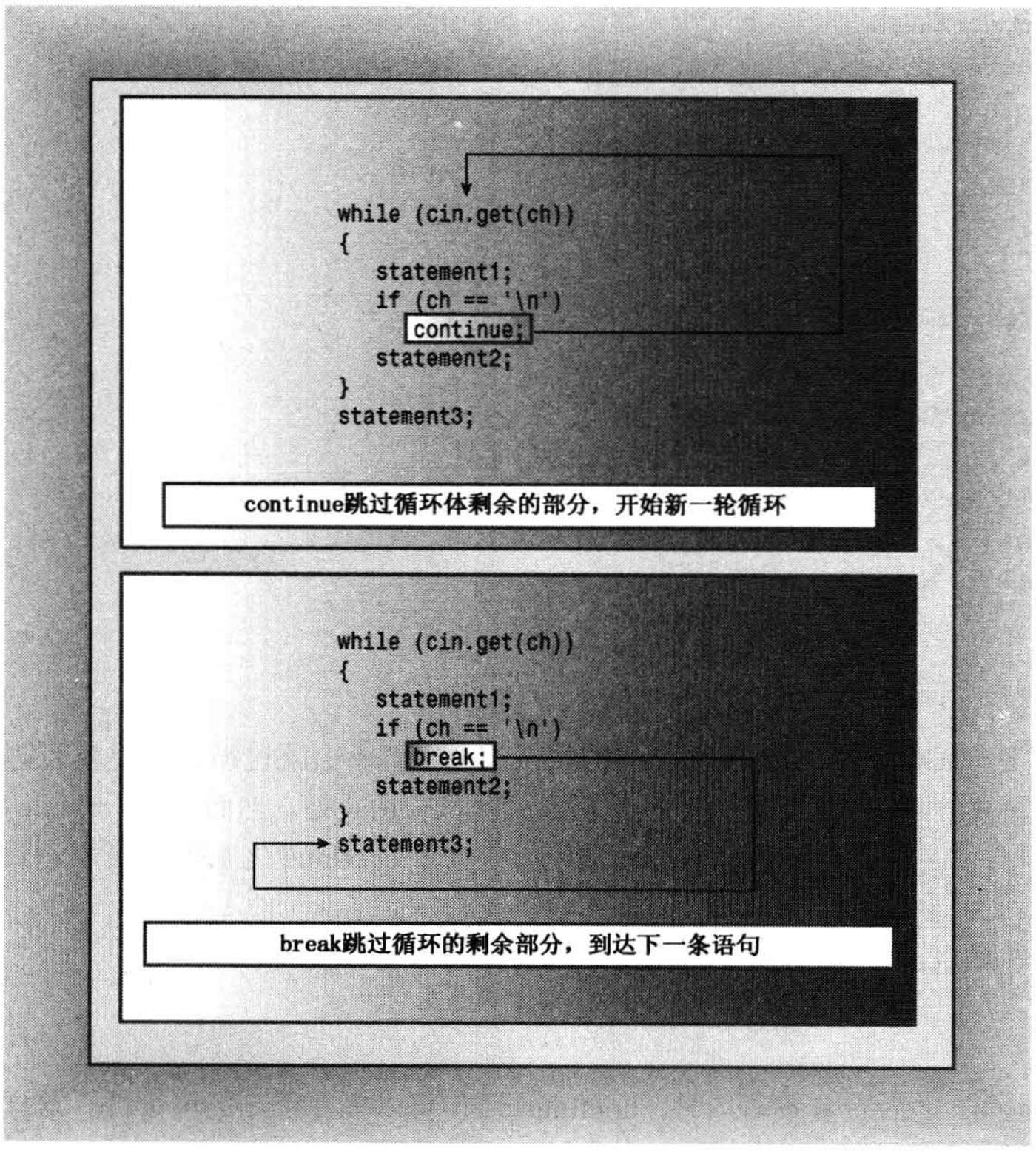


图 6.4　break 和 continue 语句的结构

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.
```

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。然而，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++;
```

然而，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示例。

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：本书前面说过，在有些执行环境中，为让窗口打开以便能够看到输出，需要添加额外的代码。在这个示例中，由于输入‘q’结束输入，处理起来更复杂些：

```
if (!cin)  // input terminated by non-numeric response
{
    cin.clear();  // reset input
    cin.get();    // read q
}
cin.get();        // read end of line after last input
cin.get();        // wait for user to press <Enter>
```

在程序清单 6.13 中，如果要让程序在结束循环后接收输入，也可使用类似的代码。

程序清单 6.14 更进了一步，它使用 cin 来返回值并重置 cin。

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++将不会判断右侧的表达式。在这里，对右侧的表达式进行判定意味着用 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;
}
```

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，因此将一个值放到数组中；而表达式!(cin >> golf [i])为 false，

因此结束内部循环。然而，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，并下载了一个文件，其中包含 1000 种股票的价格。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

这里再介绍一下文本 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。

假设有如下示例输入行：

```
38.5 19.2
```

来看一下使用不同数据类型的变量来存储时，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，这四个字符对应于数值 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（示例输入行少于 50 个字符）。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-1.58E+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，为文件输出做准备。

- 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 和运算符<<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，应使用方法 close( )将其关闭。
- 可结合使用 ofstream 对象和运算符<<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.";
fout << 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（如前述示例中的 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 Perky
Enter the model year: 2009
Enter the original asking price: 13500
Make and model: Flitz Perky
Year: 2009
Was asking $13500.00
Now asking $12325.50
```

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（根据编译器的配置，该文件也可能位于其他文件夹），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 Perky
Year: 2009
Was asking $13500.00
Now asking $12325.50
```

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( )将该对象特定文件关联起来：

```
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。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。它不但能够使用运算符<<，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。控制台输入涉及多个方面，下面首先总结这些方面。

- 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 和运算符>>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 对象和运算符>>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（如前述示例中的 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，详细请参阅后面的“程序说明”。注意，通过使用了 if 语句，该程序受益匪浅。

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;
}
```

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
```

程序还必须能够找到这个文件。通常，除非在输入的文件名中包含路径，否则程序将在可执行文件所属的文件夹中查找。

警告： Windows 文本文件的每行都以回车字符和换行符结尾；通常情况下，C++在读取文件时将这两个字符转换为换行符，并在写入文件时执行相反的转换。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将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，可能出现意外的问题，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 good( )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.good( )，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运算符。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。通过使用逻辑运算符（&&、||和!），可以组合或修改关系表达式，创建更细致的测试。条件运算符（?:）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，下一章将介绍函数。

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（请在每行末尾按回车键）：

```
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 为 1000-2000 或 guest 为 1。

f．ch 是小写字母或大写字母（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（如果您愿意，也可使用模板类 array）。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，收入为 38000 tvarps 时，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”的资金。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：用来储存姓名的字符数组（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，它打开一个文件文件，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。即该文件类似于下面：

```
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。
- 按值传递函数参数。
- 设计处理数组的函数。
- 使用 const 指针参数。
- 设计处理文本字符串的函数。
- 设计处理结构的函数。
- 设计处理 string 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 库加上多个 C++类），但真正的编程乐趣在于编写自己的函数；另一方面，要提高编程效率，可更深入地学习 STL 和 BOOST C++提供的功能。本章和第 8 章介绍如何定义函数、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。然而，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，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。本书前面的示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。

然而，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 用一个简短的示例演示了这 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
        cout << "main() is finished with the simple() function.\n";
    // cin.get();
    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.
main() is finished with the simple() function.
```

执行函数 simple()时，将暂停执行 main( )中的代码；等函数 simple()执行完毕后，继续执行 main()中的代码。在每个函数定义中，都使用了一条 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 编译指令或在函数中使用 std::cout。

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
{

    for (int i = 0; i < n; i++)
        std::cout << "Cheers! ";
    std::cout << std::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。

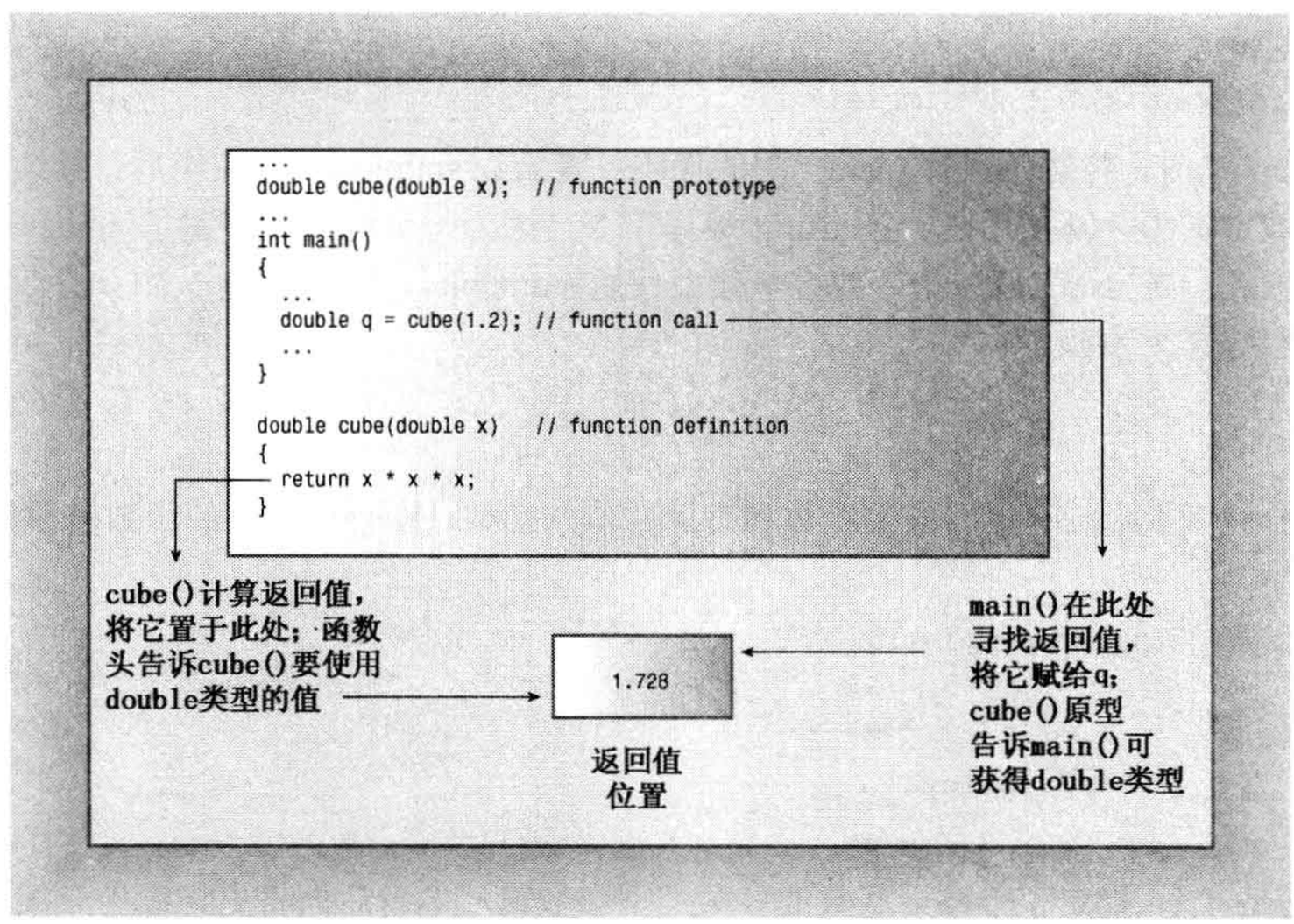


图 7.1 典型的返回值机制

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，但可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
}
```

如果函数包含多条返回语句，通常认为它会令人迷惑，有些编译器将针对这一点发出警告。然而，这里的代码很简单，很容易理解。

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()
{
    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，只需要使用名称空间 std 中成员的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);
```

但正如前面指出的，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语法。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唯一方法是，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. 原型的语法

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
```

然而，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 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 responsibility
```

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

正如您看到的，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 ANSIC 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，该函数将创建一个新的名为 x 的 double 变量，并将其初始化为 5。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 的副本，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）。

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。

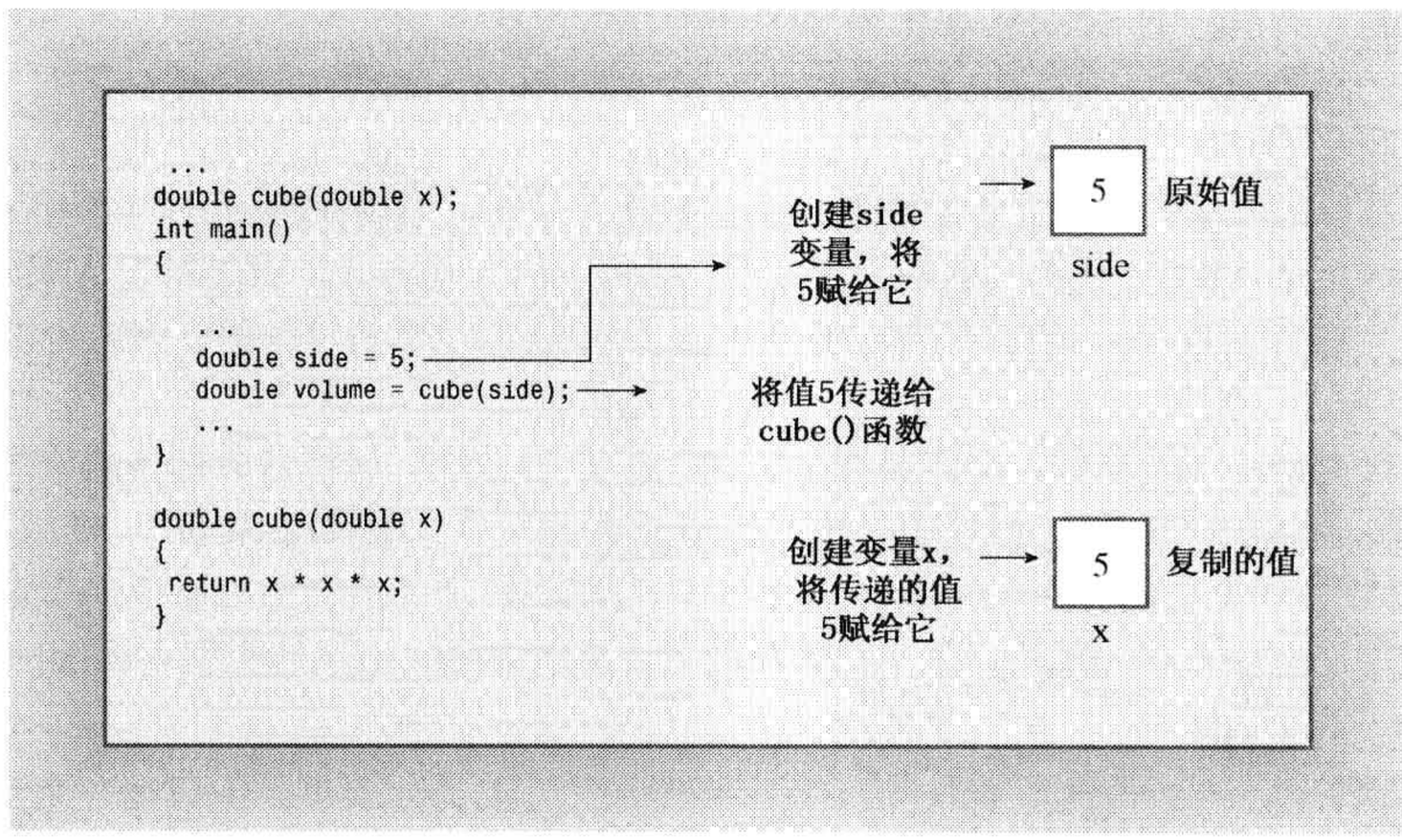


图 7.2　按值传递

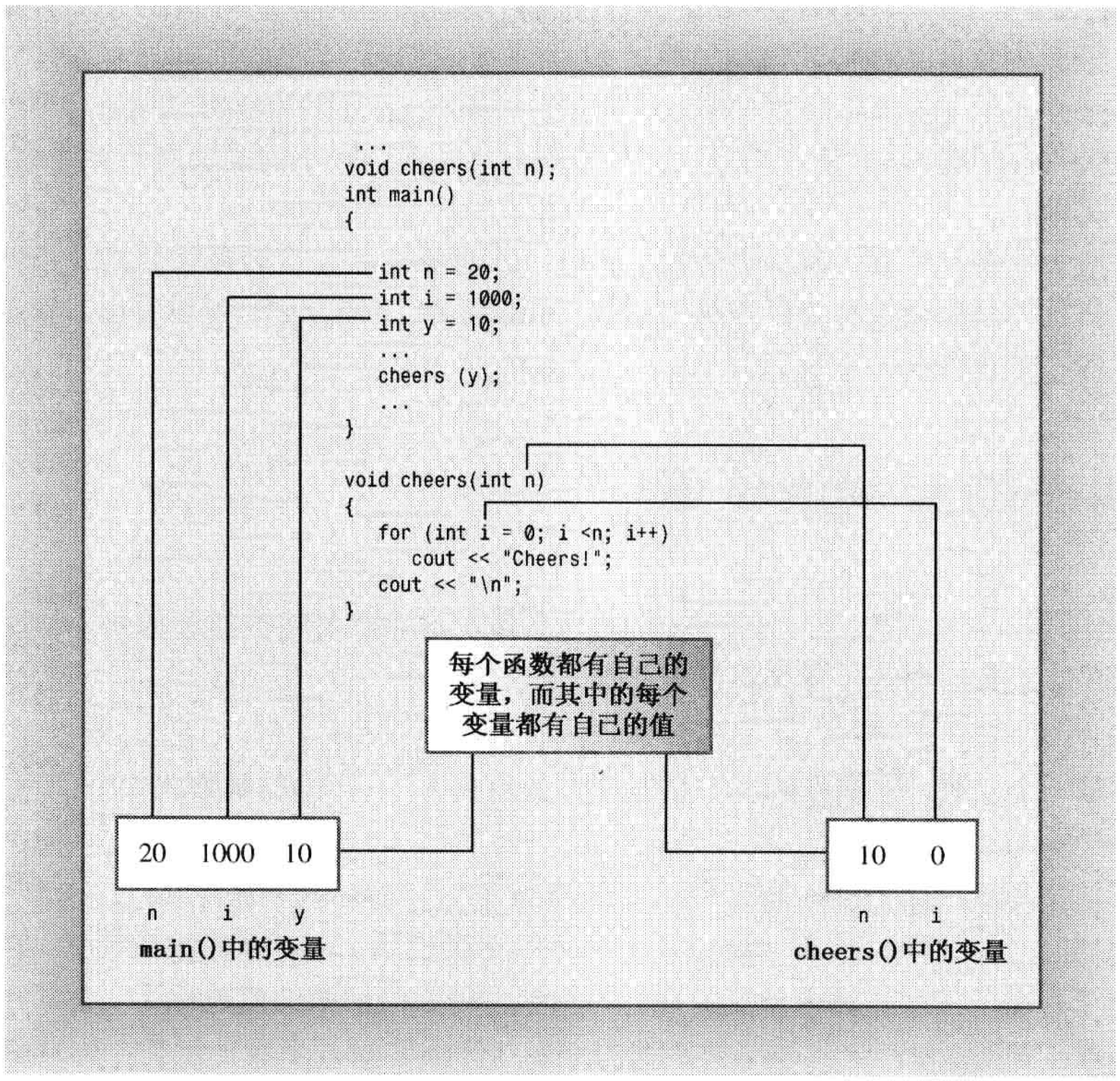


图 7.3　局部变量

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_char( )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
```

然而，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
WWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWW
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。当用户对程序提示作出响应时，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。即使您在函数 main( )中使用名称 n 而不是 times，在函数 n_chars()中修改 n 的值时，也不会影响函数 main( )中 n 的值。

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，该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。推而广之，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++实现是这样的，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 the 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，本书的函数示例都很简单，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。这里唯一的新内容是，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，通过一个示例验证这种方法可行，然后看看它为什么可行。

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，该程序管用。下面讨论为何该程序管用。

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
```

该规则有一些例外。首先，数组声明使用数组名来标记存储位置；其次，对数组名使用 sizeof 将得到整个数组的长度（以字节为单位）；第三，正如第 4 章指出的，将地址运算符&用于数组名时，将返回整个数组的地址，例如&cookies 将返回一个 32 字节内存块的地址（如果 int 长 4 字节）。

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。然而，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时等效的。

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。

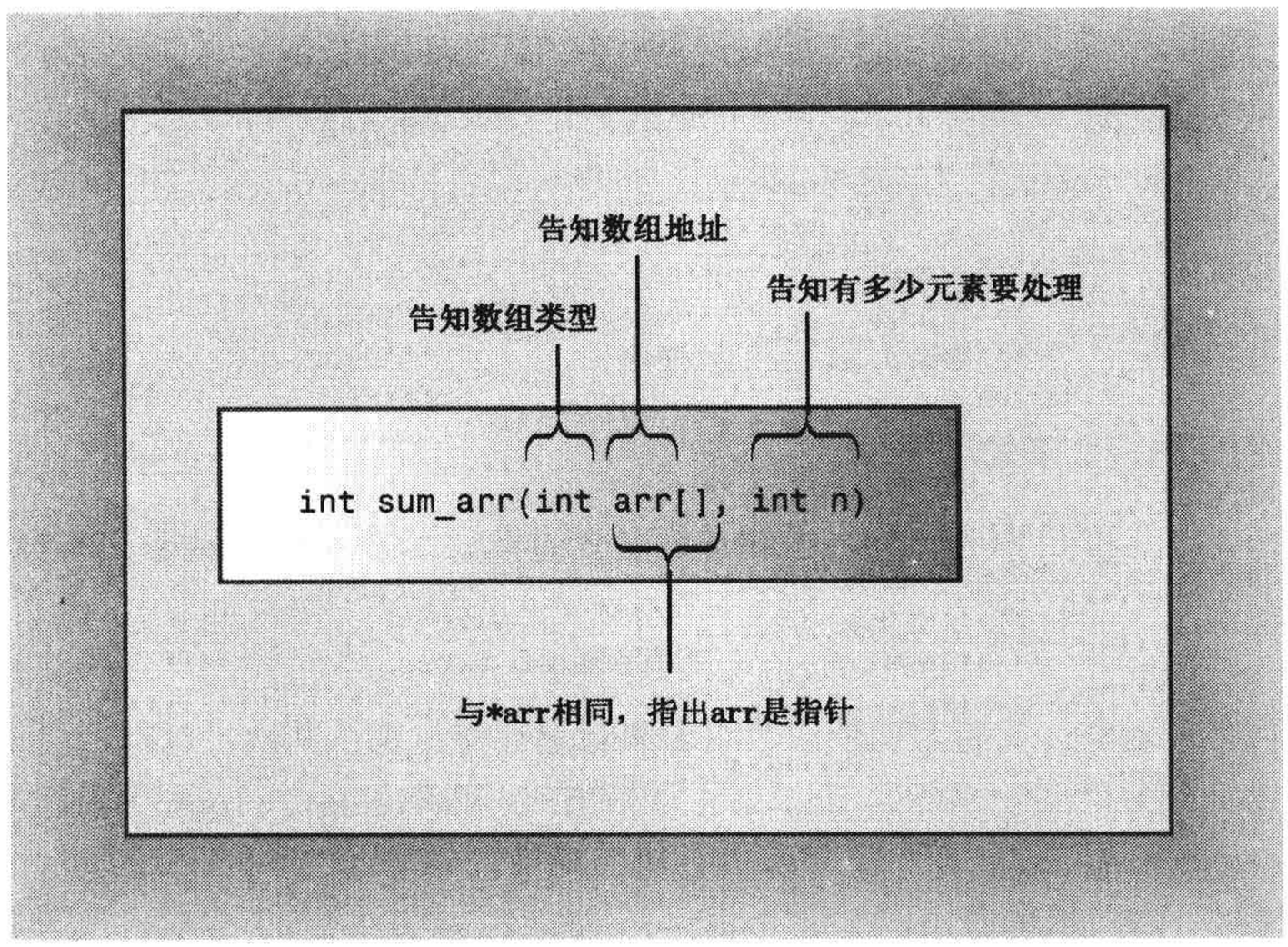


图 7.4　告知函数有关数组的信息

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示例，但先来修改程序清单 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（地址值和数组的长度将随系统而异）：

```
003EF9FC = array address, 32 = sizeof cookies
003EF9FC = arr, 4 = sizeof arr
Total cookies eaten: 255
003EF9FC = arr, 4 = sizeof arr
First three eaters ate 7 cookies.
003EFA0C = arr, 4 = sizeof arr
Last four eaters ate 240 cookies.
```

注意，地址值和数组的长度随系统而异。另外，有些 C++实现以十进制而不是十六进制格式显示地址，还有些编译器以十六进制显示地址时，会加上前缀 0x。

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 32，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）。顺便说一句，这也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；指针本身并没有指出数组的长度。

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。

注意： *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　更多数组函数示例

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。

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示例中。

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，而不是其副本，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，就像前一个示例那样。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。然而，还有另一个问题——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副本。然而，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（稍作了编辑）：

```
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，您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。该程序检查用户输入的是否是数字，如果不是，则要求用户这样做。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);
    if (size > 0)
    {
        cout << "Enter revaluation factor: ";
        double factor;
        while (!(cin >> factor))    // bad input
        {
            cin.clear();
            while (cin.get() != '\n')
                continue;
            cout << "Bad input; Please enter a number: ";
        }
        revalue(factor, properties, size);
        show_array(properties, size);
    }
    cout << "Done.\n";
    cin.get();
    cin.get();
    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 revaluation factor: 0.8
Property #1: $80000
Property #2: $64000
Property #3: $177600
Property #4: $192000
Property #5: $94400
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 reevaluation factor: 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示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从上而下的程序设计（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。

6. 数组处理函数的常用编写方式

假设要编写一个处理 double 数组的函数。如果该函数要修改数组，其原型可能类似于下面这样：

```
void f_modify(double ar[], int n);
```

如果函数不修改数组，其原型可能类似于下面这样：

```
void _f_no_change(const double ar[], int n);
```

当然，在函数原型中可以省略变量名，也可将返回类型指定为类型。这里的要点是，ar 实际上是一个指针，指向传入的数组的第一个元素；另外，由于通过参数传递了元素数，这两个函数都可使用任何长度的数组，只要数组的类型为 double：

```
double rewards[1000];
double faults[50];
...
f_modify(rewards, 1000);
f_modify(faults, 50);
```

这种做法是通过传递两个数字（数组地址和元素数）实现的。正如您看到的，函数缺少一些有关原始数组的知识；例如，它不能使用 sizeof 来获悉原始数组的长度，而必须依赖于程序员传入正确的元素数。

7.3.4 使用数组区间的函数

正如您看到的，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。另外，必须按正确的顺序传递指针，因为这里的代码假定 begin 在前面，end 在后面。

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，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
```

现在来看一个微妙的问题。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常规指针，而这里将常规变量的地址赋给指向 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 章中对运算符 const_cast 的讨论。

如果将指针指向指针，则情况将更复杂。前面讲过，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
```

然而，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。

注意： *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
```

但仍然不能使用 pt 来修改它指向的值（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
```

在最后一个声明中，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）。

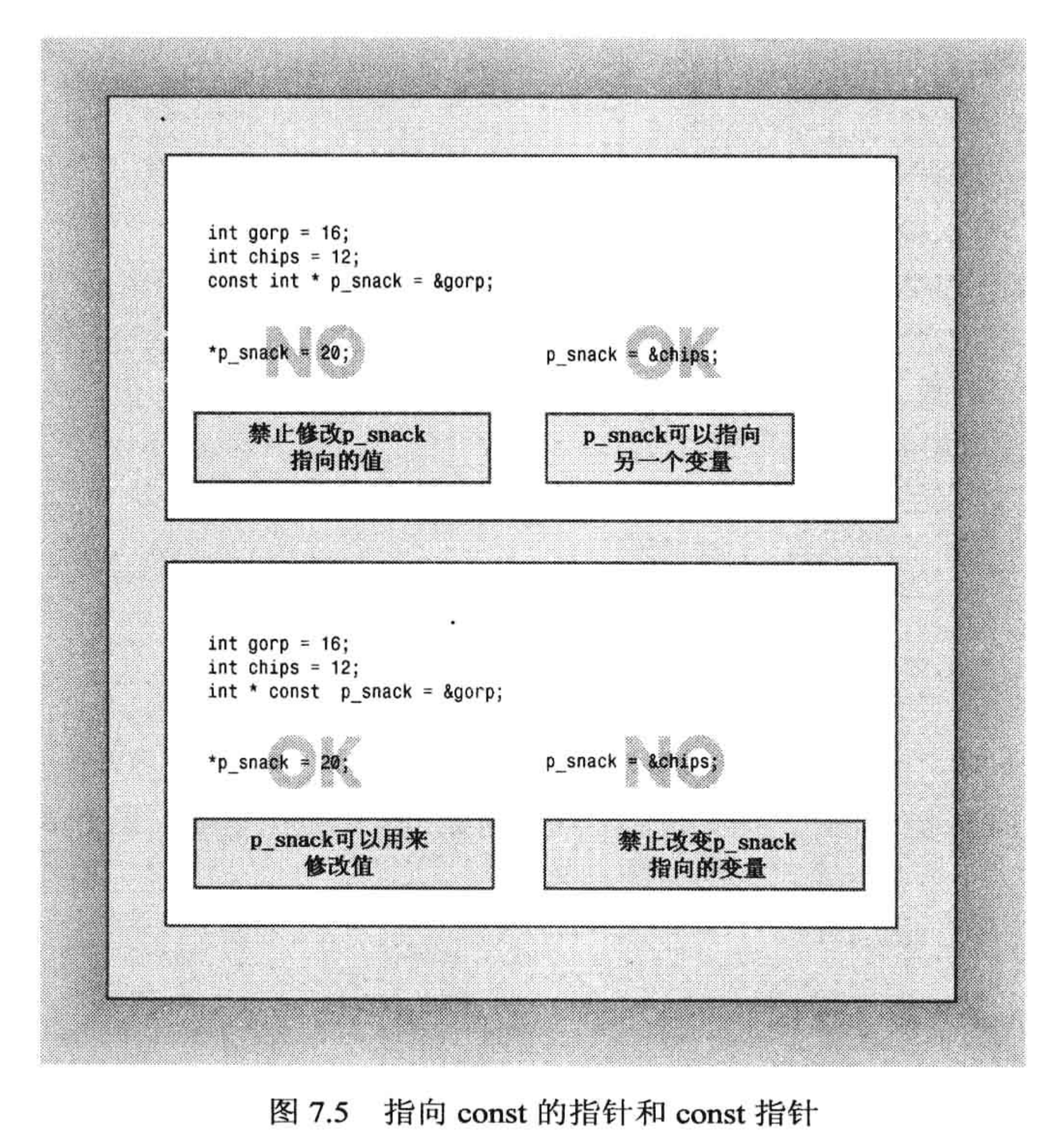


图 7.5　指向 const 的指针和 const 指针

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。只要只有一层间接关系，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，因为下面的声明将声明一个由 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：

```
int *ar2[4]
```

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 (int 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，也可以使用运算符*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。然而，下面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三种：

- 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。由于该程序不需要处

理负数，因此它将计数变量的类型声明为 unsigned int。

程序清单 7.9 strgfun.cpp

```
// strgfun.cpp -- functions with a string argument
#include <iostream>
unsigned 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

    unsigned int ms = c_in_str(mmm, 'm');
    unsigned 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
unsigned int c_in_str(const char * str, char ch)
{
    unsigned 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：

```
unsigned int c_in_str(const char str[], char ch) // also okay
```

然而，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++;
}
```

str 最初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
VVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVV
++++++++++++++++++++-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，下面的循环将循环 n 次，直到 n 减少到 0，这将填充 n 个元素：

```
while (n-- > 0)
    pstr[n] = c;
```

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单个实体或数据对象，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副本。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，必须使用地址运算符&。在 C 语言和 C++中，都使用符号&来表示地址运算符；另外，C++还使用该运算符来表示引用变量，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。然而，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示例。第一个例子处理行程时间。有些地图指出，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，通过求模运算符（%）得到剩余的分钟数。程序清单 7.11 在 sum( )函数中使用了这种计算方式，并使用 show_time( )函数显示 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运算符。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示例

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示例。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：

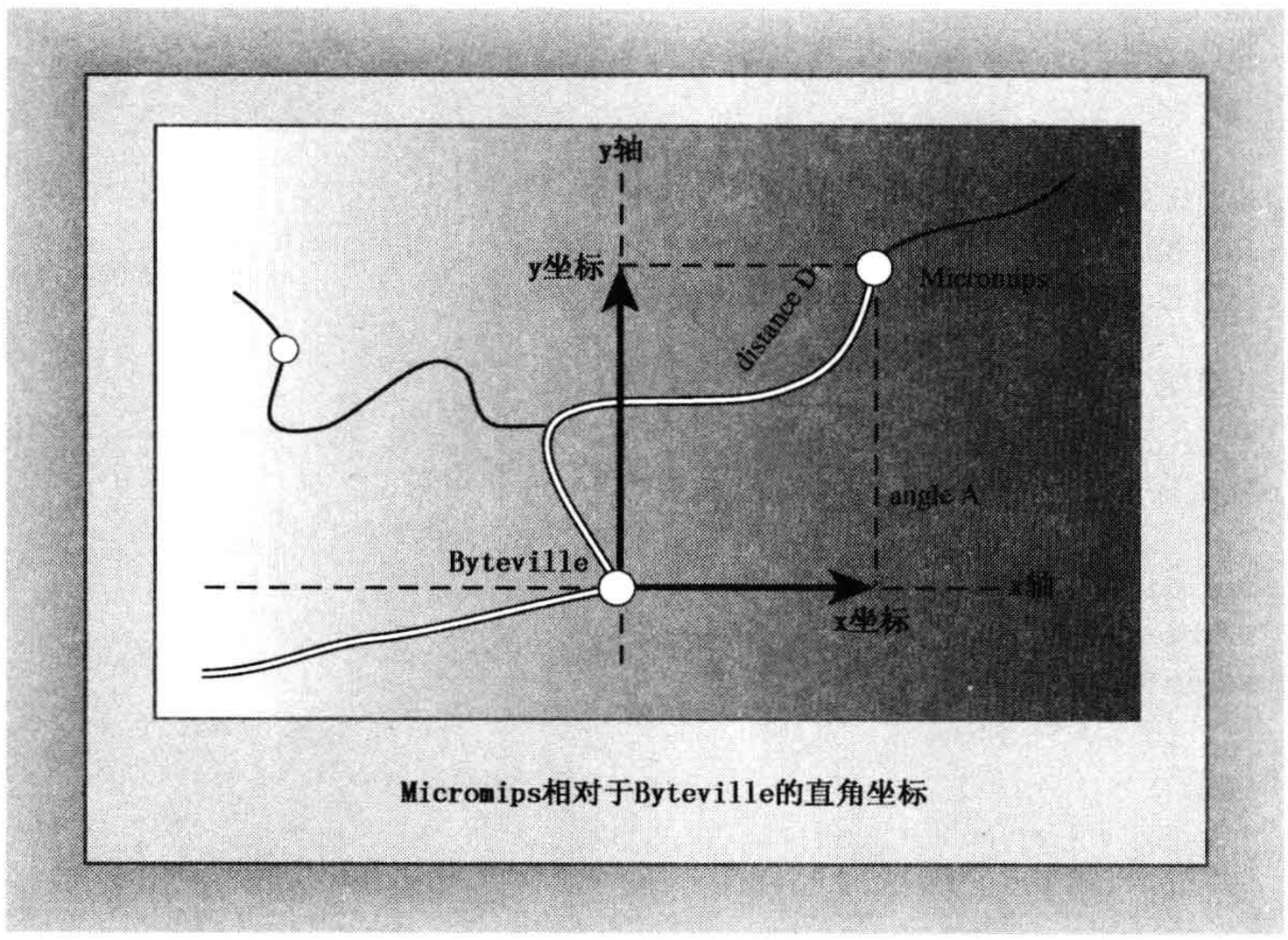


图 7.6 直角坐标

```
struct rect
{
      double x;            // horizontal distance from origin
      double y;            // vertical distance from origin
};
```

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：

```
struct polar
{
      double distance;   // distance from origin
      double angle;      // direction from origin
};
```

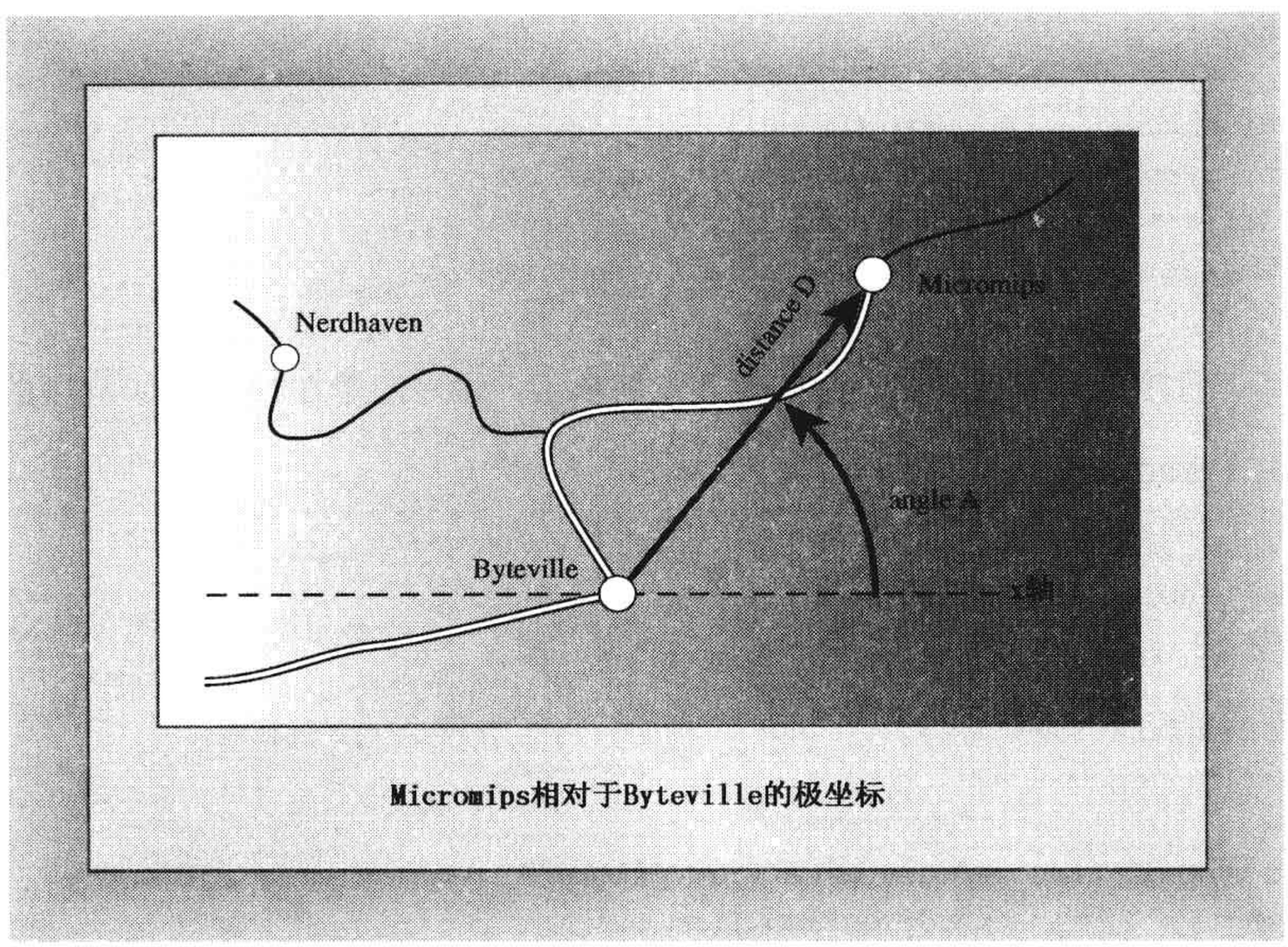


图 7.7　极坐标

下面来创建一个显示 polar 结构的内容的函数。C++库（从 C 语言借鉴而来）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运算符句点（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 cmath（在较旧的系统中为 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。抽取运算符（>>）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，类运算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。将抽取运算符用于 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（前面的输出示例就是这样做的），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。需要修改三个地方：

- 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，因此使用了 const 修饰符；
- 由于形参是指针而不是结构，因此应间接成员运算符（->），而不是成员运算符（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：有些编译器需要明确指示，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
```

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 使用的是结构副本，而程序清单 7.13 使用的是指针，让函数能够对原始结构进行操作。

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 提供了一个小型示例，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;
}
```

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
```

对于该示例，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 函数与 array 对象

在 C++中，类对象是基于结构的，因此结构编程方面的有些考虑因素也适用于类。例如，可按值将对象传递给函数，在这种情况下，函数处理的是原始对象的副本。另外，也可传递指向对象的指针，这让函数能够操作原始对象。下面来看一个使用 C++11 模板类 array 的例子。

假设您要使用一个 array 对象来存储一年四个季度的开支：

```
std::array<double, 4> expenses;
```

本书前面说过，要使用 array 类，需要包含头文件 array，而名称 array 位于名称空间 std 中。如果函数来显示 expenses 的内容，可按值传递 expenses：

```
show(expenses);
```

但如果函数要修改对象 expenses，则需将该对象的地址传递给函数（下一章将讨论另一种方法——使用引用）：

```
fill(&expenses);
```

这与程序清单 7.13 处理结构时使用的方法相同。

如何声明这两个函数呢？expenses 的类型为 array<double, 4>，因此必须在函数原型中指定这种类型：

```
void show(std::array<double, 4> da);   // da an object
void fill(std::array<double, 4> * pa); // pa a pointer to an object
```

这些考虑因素是这个示例程序的核心。该程序还包含其他一些功能。首先，它用符号常量替换了 4：

```
const int Seasons = 4;
```

其次，它使用了一个 const array 对象，该对象包含 4 个 string 对象，用于表示几个季度：

```
const std::array<std::string, Seasons> Snames =
    {"Spring", "Summer", "Fall", "Winter"};
```

请注意，模板 array 并非只能存储基本数据类型，它还可存储类对象。程序清单 7.15 列出了该程序的完整代码。

程序清单 7.15　arrobj.cpp

```
//arrobj.cpp -- functions with array objects (C++11)
#include <iostream>
#include <array>
#include <string>
// constant data
const int Seasons = 4;
const std::array<std::string, Seasons> Snames =
    {"Spring", "Summer", "Fall", "Winter"};

// function to modify array object
void fill(std::array<double, Seasons> * pa);
// function that uses array object without modifying it
void show(std::array<double, Seasons> da);

int main()
{
    std::array<double, Seasons> expenses;
    fill(&expenses);
    show(expenses);
    return 0;
}

void fill(std::array<double, Seasons> * pa)
{
    using namespace std;
    for (int i = 0; i < Seasons; i++)
    {
        cout << "Enter " << Snames[i] << " expenses: ";
        cin >> (*pa)[i];
```

```
    }
}

void show(std::array<double, Seasons> da)
{
    using namespace std;
    double total = 0.0;
    cout << "\nEXPENSES\n";
    for (int i = 0; i < Seasons; i++)
    {
        cout << Snames[i] << ": $" << da[i] << endl;
        total += da[i];
    }
    cout << "Total Expenses: $" << total << endl;
}
```

下面是该程序的运行情况：

```
Enter Spring expenses: 212
Enter Summer expenses: 256
Enter Fall expenses: 208
Enter Winter expenses: 244
EXPENSES
Spring: $212
Summer: $256
Fall: $208
Winter: $244
Total: $920
```

程序说明

由于 const array 对象 Snames 是在所有函数之前声明的，因此可后面的任何函数定义中使用它。与 const Seasons 一样，Snames 也有整个源代码文件共享。这个程序没有使用编译指令 using，因此必须使用 std::限定 array 和 string。为简化程序，并将重点放在函数可如何使用对象上，函数 fill()没有检查输入是否有效。

函数 fill()和 show()都有缺点。函数 show()存在的问题是，expenses 存储了四个 double 值，而创建一个新对象并将 expenses 的值复制到其中的效率太低。如果修改该程序，使其处理每月甚至每日的开支，这种问题将更严重。

函数 fill()使用指针来直接处理原始对象，这避免了上述效率低下的问题，但代价是代码看起来更复杂：

```
fill(&expenses);    // don't forget the &
...
cin >> (*pa)[i];
```

在最后一条语句中，pa 是一个指向 array<double, 4>对象的指针，因此*pa 为这种对象，而(*pa) [i]是该对象的一个元素。由于运算符优先级的影响，其中的括号必不可少。这里的逻辑很简单，但增加了犯错的机会。

使用第 8 章将讨论的引用可解决效率和表示法两方面的问题。

7.9 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（然而，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。

7.9.1　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)
{
      statements1
      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 7.16 演示了这种行为。

程序清单 7.16　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 7.16，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。另外，注意到在 Counting down 阶段和 Kaboom 阶段的相同层级，n 的地址相同。

7.9.2 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之策略（divide-and-conquer strategy）。程序清单 7.17 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 函数使用循环调用 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 7.17　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 7.17 中程序的输出：

```
|                                                               |
|                               |                               |
|               |               |               |               |
|       |       |       |       |       |       |       |       |
|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
|||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
```

程序说明

在程序清单 7.17 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。

7.10　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。

7.10.1　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。

为提供正确的运算符优先级，必须在声明中使用括号将*pf 括起。括号的优先级比*运算符高，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

真是非常棒的语法！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。

7.10.2　函数指针示例

程序清单 7.18 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，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 7.18　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)
```

下面是再次运行该程序的情况：

```
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)
```

7.10.3 深入探讨函数指针

函数指针的表示可能非常恐怖。下面通过一个示例演示使用函数指针时面临的一些挑战。首先，下面是一些函数的原型，它们的特征标和返回类型相同：

```
const double * f1(const double ar[], int n);
const double * f2(const double [], int);
const double * f3(const double *, int);
```

这些函数的特征标看似不同，但实际上相同。首先，前面说过，在函数原型中，参数列表 const double ar [ ]与 const double * ar 的含义完全相同。其次，在函数原型中，可以省略标识符。因此，const double ar [ ]可简化为 const double [ ]，而 const double * ar 可简化为 const double *。因此，上述所有函数特征标的含义都相同。另一方面，函数定义必须提供标识符，因此需要使用 const double ar [ ]或 const double * ar。

接下来，假设要声明一个指针，它可指向这三个函数之一。假定该指针名为 pa，则只需将目标函数原型中的函数名替换为(*pa)：

```
const double * (*p1)(const double *, int);
```

可在声明的同时进行初始化：

```
const double * (*p1)(const double *, int) = f1;
```

使用 C++11 的自动类型推断功能时，代码要简单得多：

```
auto p2 = f2;  // C++11 automatic type deduction
```

现在来看下面的语句：

```
cout <<  (*p1)(av,3) << ": " << *(*p1)(av,3) << endl;
cout << p2(av,3) << ": " << *p2(av,3) << endl;
```

根据前面介绍的知识可知，(*p1) (av, 3)和 p2(av, 3)都调用指向的函数（这里为 f1()和 f2()），并将 av 和 3 作为参数。因此，显示的是这两个函数的返回值。返回值的类型为 const double *（即 double 值的地址），因此在每条 cout 语句中，前半部分显示的都是一个 double 值的地址。为查看存储在这些地址处的实际值，需要将运算符*应用于这些地址，如表达式*(*p1)(av,3)和*p2(av,3)所示。

鉴于需要使用三个函数，如果有一个函数指针数组将很方便。这样，将可使用 for 循环通过指针依次调用每个函数。如何声明这样的数组呢？显然，这种声明应类似于单个函数指针的声明，但必须在某个地方加上[3]，以指出这是一个包含三个函数指针的数组。问题是在什么地方加上[3]，答案如下（包含初始化）：

```
const double * (*pa[3])(const double *, int) = {f1,f2,f3};
```

为何将[3]放在这个地方呢？pa 是一个包含三个元素的数组，而要声明这样的数组，首先需要使用 pa[3]。该声明的其他部分指出了数组包含的元素是什么样的。运算符[]的优先级高于*，因此*pa[3]表明 pa 是一个包含三个指针的数组。上述声明的其他部分指出了每个指针指向的是什么：特征标为 const double *, int，且返回类型为 const double *的函数。因此，pa 是一个包含三个指针的数组，其中每个指针都指向这样的函数，即将 const double *和 int 作为参数，并返回一个 const double *。

这里能否使用 auto 呢？不能。自动类型推断只能用于单值初始化，而不能用于初始化列表。但声明数组 pa 后，声明同样类型的数组就很简单了：

```
auto pb = pa;
```

本书前面说过，数组名是指向第一个元素的指针，因此 pa 和 pb 都是指向函数指针的指针。

如何使用它们来调用函数呢？pa[i]和 pb[i]都表示数组中的指针，因此可将任何一种函数调用表示法用于它们：

```
const double * px = pa[0](av,3);
const double * py = (*pb[1])(av,3);
```

要获得指向的 double 值，可使用运算符*：

```
double x = *pa[0](av,3);
double y = *(*pb[1])(av,3);
```

可做的另一件事是创建指向整个数组的指针。由于数组名 pa 是指向函数指针的指针，因此指向数组的指针将是这样的指针，即它指向指针的指针。这听起来令人恐怖，但由于可使用单个值对其进行初始化，因此可使用 auto：

```
auto pc = &pa;  // C++11 automatic type deduction
```

如果您喜欢自己声明，该如何办呢？显然，这种声明应类似于 pa 的声明，但由于增加了一层间接，因此需要在某个地方添加一个*。具体地说，如果这个指针名为 pd，则需要指出它是一个指针，而不是数组。这意味着声明的核心部分应为(*pd)[3]，其中的括号让标识符 pd 与*先结合：

```
*pd[3]     // an array of 3 pointers
(*pd)[3]   // a pointer to an array of 3 elements
```

换句话说，pd 是一个指针，它指向一个包含三个元素的数组。这些元素是什么呢？由 pa 的声明的其他部分描述，结果如下：

```
const double *(*(*pd)[3])(const double *, int) = &pa;
```

要调用函数，需认识到这样一点：既然 pd 指向数组，那么*pd 就是数组，而(*pd)[i]是数组中的元素，即函数指针。因此，较简单的函数调用是(*pd)[i](av,3)，而*(*pd)[i](av,3)是返回的指针指向的值。也可以使用第二种使用指针调用函数的语法：使用(*(*pd)[i])(av,3)来调用函数，而*(*(*pd)[i])(av,3)是指向的 double 值。

请注意 pa（它是数组名，表示地址）和&pa 之间的差别。正如您在本书前面看到的，在大多数情况下，pa 都是数组第一个元素的地址，即&pa[0]。因此，它是单个指针的地址。但&pa 是整个数组（即三个指针块）的地址。从数字上说，pa 和&pa 的值相同，但它们的类型不同。一种差别是，pa+1 为数组中下一个元素的地址，而&pa+1 为数组 pa 后面一个 12 字节内存块的地址（这里假定地址为 4 字节）。另一个差别是，要得到第一个元素的值，只需对 pa 解除一次引用，但需要对&pa 解除两次引用：

```
**&pa == *pa == pa[0]
```

程序清单 7.19 使用了这里讨论的知识。出于演示的目的，函数 f1()等都非常简单。正如注释指出的，这个程序演示了 auto 的 C++98 替代品。

程序清单 7.19　arfupt.cpp

```
// arfupt.cpp -- an array of function pointers
#include <iostream>
// various notations, same signatures
const double * f1(const double ar[], int n);
const double * f2(const double [], int);
const double * f3(const double *, int);

int main()
{
    using namespace std;
    double av[3] = {1112.3, 1542.6, 2227.9};

    // pointer to a function
    const double *(*p1)(const double *, int) = f1;
    auto p2 = f2;  // C++11 automatic type deduction
    // pre-C++11 can use the following code instead
    // const double *(*p2)(const double *, int) = f2;
    cout << "Using pointers to functions:\n";
    cout << " Address  Value\n";
    cout <<  (*p1)(av,3) << ": " << *(*p1)(av,3) << endl;
    cout << p2(av,3) << ": " << *p2(av,3) << endl;

    // pa an array of pointers
    // auto doesn't work with list initialization
    const double *(*pa[3])(const double *, int) = {f1,f2,f3};
    // but it does work for initializing to a single value
    // pb a pointer to first element of pa
    auto pb = pa;
    // pre-C++11 can use the following code instead
    // const double *(**pb)(const double *, int) = pa;
    cout << "\nUsing an array of pointers to functions:\n";
    cout << " Address  Value\n";
    for (int i = 0; i < 3; i++)
        cout << pa[i](av,3) << ": " << *pa[i](av,3) << endl;
    cout << "\nUsing a pointer to a pointer to a function:\n";
    cout << " Address  Value\n";
    for (int i = 0; i < 3; i++)
```

```
    cout << pb[i](av,3) << ": " << *pb[i](av,3) << endl;

// what about a pointer to an array of function pointers
cout << "\nUsing pointers to an array of pointers:\n";
cout << " Address  Value\n";
// easy way to declare pc
auto pc = &pa;
// pre-C++11 can use the following code instead
// const double *(*(*pc)[3])(const double *, int) = &pa;
cout << (*pc)[0](av,3) << ": " << *(*pc)[0](av,3) << endl;
    // hard way to declare pd
    const double *(*(*pd)[3])(const double *, int) = &pa;
    // store return value in pdb
    const double * pdb = (*pd)[1](av,3);
    cout << pdb << ": " << *pdb << endl;
    // alternative notation
    cout << (*(*pd)[2])(av,3) << ": " << *(*(*pd)[2])(av,3) << endl;
    // cin.get();
    return 0;
}

// some rather dull functions

const double * f1(const double * ar, int n)
{
    return ar;
}
const double * f2(const double ar[], int n)
{
    return ar+1;
}
const double * f3(const double ar[], int n)
{
    return ar+2;
}
```

该程序的输出如下：

```
Using pointers to functions:
 Address  Value
002AF9E0: 1112.3
002AF9E8: 1542.6

Using an array of pointers to functions:
 Address  Value
002AF9E0: 1112.3
002AF9E8: 1542.6
002AF9F0: 2227.9

Using a pointer to a pointer to a function:
 Address  Value
002AF9E0: 1112.3
002AF9E8: 1542.6
002AF9F0: 2227.9

Using pointers to an array of pointers:
```

```
 Address  Value
002AF9E0: 1112.3
002AF9E8: 1542.6
002AF9F0: 2227.9
```

显示的地址为数组 av 中 double 值的存储位置。

这个示例可能看起来比较深奥，但指向函数指针数组的指针并不少见。实际上，类的虚方法实现通常都采用了这种技术（参见第 13 章）。所幸的是，这些细节由编译器处理。

感谢 auto

C++11 的目标之一是让 C++更容易使用，从而让程序员将主要精力放在设计而不是细节上。程序清单 7.19 演示了这一点：

```
auto pc = &pa;                                  // C++11 automatic type deduction
const double *(*(*pd)[3])(const double *, int) = &pa; // C++98, do it yourself
```

自动类型推断功能表明，编译器的角色发生了改变。在 C++98 中，编译器利用其知识帮助您发现错误，而在 C++11 中，编译器利用其知识帮助您进行正确的声明。

存在一个潜在的缺点。自动类型推断确保变量的类型与赋给它的初值的类型一致，但您提供的初值的类型可能不对：

```
auto pc = *pa;    // oops! used *pa instead of &pa
```

上述声明导致 pc 的类型与*pa 一致，在程序清单 7.19 中，后面使用它时假定其类型与&pa 相同，这将导致编译错误。

7.10.4 使用 typedef 进行简化

除 auto 外，C++还提供了其他简化声明的工具。您可能还记得，第 5 章说过，关键字 typedef 让您能够创建类型别名：

```
typedef double real; // makes real another name for double
```

这里采用的方法是，将别名当做标识符进行声明，并在开头使用关键字 typedef。因此，可将 p_fun 声明为程序清单 7.19 使用的函数指针类型的别名：

```
typedef const double *(*p_fun)(const double *, int);  // p_fun now a type name
p_fun p1 = f1;  // p1 points to the f1() function
```

然后使用这个别名来简化代码：

```
p_fun pa[3] = {f1,f2,f3}; // pa an array of 3 function pointers
p_fun (*pd)[3] = &pa;     // pd points to an array of 3 function pointers
```

使用 typedef 可减少输入量，让您编写代码时不容易犯错，并让程序更容易理解。

7.11 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。另外，也可传递两个指针（其中一个指向数组开头，另一个指向数组末尾的下一个元素），以指定一个范围，就像 STL 使用的算法一样。

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。然而，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。这些考虑因素也适用于类对象。

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。

7.12　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。

12．假设有如下结构声明：

```
struct applicant {
    char name[30];
    int credit_ratings[3];
};
```

a．编写一个函数，它将 application 结构作为参数，并显示该结构的内容。

b．编写一个函数，它将 application 结构的地址作为参数，并显示该参数指向的结构的内容。

13．假设函数 f1()和 f2()的原型如下：

```
void f1(applicant * a);
const char * f2(const applicant * a1, const applicant * a2);
```

请将 p1 和 p2 分别声明为指向 f1 和 f2 的指针；将 ap 声明为一个数组，它包含 5 个类型与 p1 相同的指针；将 pa 声明为一个指针，它指向的数组包含 10 个类型与 p2 相同的指针。使用 typedef 来帮助完成这项工作。

7.13 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）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．在不使用 array 类的情况下完成程序清单 7.15 所做的工作。编写两个这样的版本：

a．使用 const char *数组存储表示季度名称的字符串，并使用 double 数组存储开支。

b．使用 const char *数组存储表示季度名称的字符串，并使用一个结构，该结构只有一个成员——一个用于存储开支的 double 数组。这种设计与使用 array 类的基本设计类似。

9．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;
}
```

10．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）：

```
double q = calculate(2.5, 10.4, add);
```

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，其中包含三个指针：

```
double (*pf[3])(double, double);
```

可以采用数组初始化语法，并将函数名作为地址来初始化这样的数组。