

本书仅提供部分阅读，如需完整版，请QQ:378286825.或下载完整版。
提供各种IT类书籍pdf下载，如有需要，请QQ.

注：链接至淘宝，不喜者勿入，整理大量资料，着实不易，付出总归要有收货，敬请谅解！

[点击下载完整版](#)

C++ Primer Plus

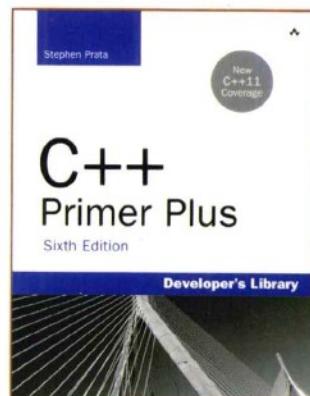


(第6版) 中文版

[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



人民邮电出版社
POSTS & TELECOM PRESS



经典C++教程十年新版再现 众多C++高手和读者好评如潮

C++很有用，但也很难学。学C++之难，一是学习曲线陡峭，二是容易误入歧途。C++11标准颁布之后，这个问题就更严重。*C++ Primer Plus*是在市场的多年检验中脱颖而出的一本名著，它的价值，正在于降低陡峭的学习曲线，并且确保读者学到“正确”的C++。

——孟岩

如果说*C++ Primer*是C++语言的一本百科全书，讲述了C++语言里面“有什么”；那么*C++ Primer Plus*就是这门语言的一本通识课本，它实实在在地教给程序员如何理解和使用这种内容丰富、威力强大的语言。本书最大的特点就是务实，通过类比、举例和习惯多维度的方式，为程序员打下坚实的、使用C++进行日常工作的基础。

——高博

如果你从未接触过C语言，我猜有80%的C++程序员会推荐*C++ Primer*这本书（而不是*C++ Primer Plus*），但实际上这两本书是有所不同的，别看仅仅就多了这么个Plus。*C++ Primer Plus*可以说是一本面向从未学习过C语言甚至是从未学习过编程的人的入门书籍。

——豆瓣读者“梦中惊醒”

这本书对于入门学者来说真的很好，讲的很细致，很透彻，非常人性化！对于初次接触面向对象编程的人来讲，真的很棒！书上的例子和课后题目也很有代表性！强烈建议阅读！

——豆瓣读者“慧质岚心”

最适合初学者的书，我看过好几本C++的图书，包括最出名的*C++ Primer*，但是还是觉得这本书通俗易懂，能让初学者更好地学好C++。

——豆瓣读者“冷月潇风”

不愧为“程序员和开发人员学习C++的完整教程”，称得上是计算机行业的经典著作。

——当当读者“刘圈点绕”



请访问www.ptpress.com或<http://t.cn/zOnA8Jy>下载书中相关源代码

ISBN 978-7-115-27946-0



9 787115 279460 >

ISBN 978-7-115-27946-0

定价：99.00 元

分类建议：计算机 / 程序设计 / C++

人民邮电出版社网址：www.ptpress.com.cn

封面设计：任文杰

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 (C I P) 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

PDG

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- 风格字符串数组和 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 ctype。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++ 4.2.1 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 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	
第 3 章 处理数据	37	
2.7 编程练习	35	
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	5.1.6 副作用和顺序点	134
4.3.3 string 类的其他操作	84	5.1.7 前缀格式和后缀格式	135
4.3.4 string 类 I/O	86	5.1.8 递增/递减运算符和指针	135
4.3.5 其他形式的字符串字面值	87	5.1.9 组合赋值运算符	136
4.4 结构简介	88	5.1.10 复合语句(语句块)	136
4.4.1 在程序中使用结构	89	5.1.11 其他语法技巧——	
4.4.2 C++11 结构初始化	91	逗号运算符	138
4.4.3 结构可以将 string 类 作为成员吗	91	5.1.12 关系表达式	140
4.4.4 其他结构属性	91	5.1.13 赋值、比较和可能犯的错误	141
4.4.5 结构数组	93	5.1.14 C-风格字符串的比较	142
4.4.6 结构中的位字段	94	5.1.15 比较 string 类字符串	144
4.5 共用体	94	5.2 while 循环	145
4.6 枚举	95	5.2.1 for 与 while	147
4.6.1 设置枚举量的值	97	5.2.2 等待一段时间: 编写延时循环	148
4.6.2 枚举的取值范围	97	5.3 do while 循环	150
4.7 指针和自由存储空间	97	5.4 基于范围的 for 循环 (C++11)	152
4.7.1 声明和初始化指针	100	5.5 循环和文本输入	152
4.7.2 指针的危险	101	5.5.1 使用原始的 cin 进行输入	152
4.7.3 指针和数字	102	5.5.2 使用 cin.get(char) 进行补救	153
4.7.4 使用 new 来分配内存	102	5.5.3 使用哪一个 cin.get()	154
4.7.5 使用 delete 释放内存	104	5.5.4 文件尾条件	155
4.7.6 使用 new 来创建动态数组	104	5.5.5 另一个 cin.get() 版本	157
4.8 指针、数组和指针算术	106	5.6 嵌套循环和二维数组	159
4.8.1 程序说明	107	5.6.1 初始化三维数组	160
4.8.2 指针小结	109	5.6.2 使用二维数组	160
4.8.3 指针和字符串	111	5.7 总结	162
4.8.4 使用 new 创建动态结构	114	5.8 复习题	163
4.8.5 自动存储、静态存储和 动态存储	117	5.9 编程练习	163
4.9 类型组合	118	第 6 章 分支语句和逻辑运算符	165
4.10 数组的替代品	120	6.1 if 语句	165
4.10.1 模板类 vector	120	6.1.1 if else 语句	167
4.10.2 模板类 array (C++11)	120	6.1.2 格式化 if else 语句	168
4.10.3 比较数组、vector 对象和 array 对象	120	6.1.3 if else if else 结构	169
4.11 总结	122	6.2 逻辑表达式	170
4.12 复习题	123	6.2.1 逻辑 OR 运算符:	171
4.13 编程练习	123	6.2.2 逻辑 AND 运算符: &&	172
第 5 章 循环和关系表达式	125	6.2.3 用 && 来设置取值范围	174
5.1 for 循环	125	6.2.4 逻辑 NOT 运算符: !	175
5.1.1 for 循环的组成部分	126	6.2.5 逻辑运算符细节	176
5.1.2 回到 for 循环	131	6.2.6 其他表示方式	177
5.1.3 修改步长	132	6.3 字符函数库 ctype	177
5.1.4 使用 for 循环访问字符串	133	6.4 ?: 运算符	179
5.1.5 递增运算符 (++) 和递减运算符 (--)	133	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	8.2.2 将引用用作函数参数.....	258
6.8.1 文本 I/O 和文本文件.....	190	8.2.3 引用的属性和特别之处.....	260
6.8.2 写入到文本文件中.....	191	8.2.4 将引用用于结构.....	263
6.8.3 读取文本文件.....	194	8.2.5 将引用用于类对象.....	268
6.9 总结.....	197	8.2.6 对象、继承和引用.....	271
6.10 复习题.....	198	8.2.7 何时使用引用参数.....	274
6.11 编程练习.....	199	8.3 默认参数.....	274
第 7 章 函数——C++ 的编程模块.....	202	8.4 函数重载.....	276
7.1 复习函数的基本知识.....	202	8.4.1 重载示例.....	278
7.1.1 定义函数.....	203	8.4.2 何时使用函数重载.....	280
7.1.2 函数原型和函数调用.....	205	8.5 函数模板.....	281
7.2 函数参数和按值传递.....	207	8.5.1 重载的模板.....	283
7.2.1 多个参数.....	208	8.5.2 模板的局限性.....	285
7.2.2 另外一个接受两个参数的函数.....	210	8.5.3 显式具体化.....	285
7.3 函数和数组.....	212	8.5.4 实例化和具体化.....	288
7.3.1 函数如何使用指针来处理数组.....	213	8.5.5 编译器选择使用哪个函数版本.....	289
7.3.2 将数组作为参数意味着什么.....	213	8.5.6 模板函数的发展.....	295
7.3.3 更多数组函数示例.....	215	8.6 总结.....	297
7.3.4 使用数组区间的函数.....	220	8.7 复习题.....	297
7.3.5 指针和 const.....	221	8.8 编程练习.....	298
7.4 函数和二维数组.....	224	第 9 章 内存模型和名称空间.....	300
7.5 函数和 C- 风格字符串.....	225	9.1 单独编译.....	300
7.5.1 将 C- 风格字符串作为参数的函数.....	225	9.2 存储持续性、作用域和链接性.....	304
7.5.2 返回 C- 风格字符串的函数.....	227	9.2.1 作用域和链接.....	305
7.6 函数和结构.....	228	9.2.2 自动存储持续性.....	305
7.6.1 传递和返回结构.....	228	9.2.3 静态持续变量.....	309
7.6.2 另一个处理结构的函数示例.....	230	9.2.4 静态持续性、外部链接性.....	310
7.6.3 传递结构的地址.....	234	9.2.5 静态持续性、内部链接性.....	313
7.7 函数和 string 对象.....	235	9.2.6 静态存储持续性、无链接性.....	315
7.8 函数与 array 对象.....	236	9.2.7 说明符和限定符.....	317
7.9 递归.....	238	9.2.8 函数和链接性.....	318
7.9.1 包含一个递归调用的递归.....	239	9.2.9 语言链接性.....	319
7.9.2 包含多个递归调用的递归.....	240	9.2.10 存储方案和动态分配.....	319
7.10 函数指针.....	241	9.3 名称空间.....	324
7.10.1 函数指针的基础知识.....	241	9.3.1 传统的 C++ 名称空间.....	324
7.10.2 函数指针示例.....	243	9.3.2 新的名称空间特性.....	325
7.10.3 深入探讨函数指针.....	244	9.3.3 名称空间示例.....	331
7.10.4 使用 typedef 进行简化.....	248	9.3.4 名称空间及其前途.....	334
7.11 总结.....	248	9.4 总结.....	335
7.12 复习题.....	249	9.5 复习题.....	335
7.13 编程练习.....	250	9.6 编程练习.....	338
第 8 章 函数探幽.....	253	第 10 章 对象和类.....	340
8.1 C++ 内联函数.....	253	10.1 过程性编程和面向对象编程.....	340
8.2 引用变量.....	255	10.2 抽象和类.....	341
8.2.1 创建引用变量.....	256	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	12.1.2 特殊成员函数	432
10.2.4 使用类	349	12.1.3 回到 Stringbad: 复制构造 函数的哪里出了问题	434
10.2.5 修改实现	350	12.1.4 Stringbad 的其他问题: 赋值运算符	436
10.2.6 小结	351		
10.3 类的构造函数和析构函数	352	12.2 改进后的新 String 类	437
10.3.1 声明和定义构造函数	353	12.2.1 修订后的默认构造函数	438
10.3.2 使用构造函数	354	12.2.2 比较成员函数	439
10.3.3 默认构造函数	354	12.2.3 使用中括号表示法访问字符	439
10.3.4 析构函数	355	12.2.4 静态类成员函数	441
10.3.5 改进 Stock 类	356	12.2.5 进一步重载赋值运算符	441
10.3.6 构造函数和析构函数小结	362		
10.4 this 指针	363	12.3 在构造函数中使用 new 时应注意的 事项	446
10.5 对象数组	368	12.3.1 应该和不应该	447
10.6 类作用域	370	12.3.2 包含类成员的类的逐成员 复制	448
10.6.1 作用域为类的常量	371		
10.6.2 作用域内枚举 (C++11)	372	12.4 有关返回对象的说明	449
10.7 抽象数据类型	373	12.4.1 返回指向 const 对象的引用	449
10.8 总结	376	12.4.2 返回指向非 const 对象的 引用	449
10.9 复习题	377	12.4.3 返回对象	450
10.10 编程练习	377	12.4.4 返回 const 对象	450
第 11 章 使用类	380	12.5 使用指向对象的指针	451
11.1 运算符重载	380	12.5.1 再谈 new 和 delete	453
11.2 计算时间: 一个运算符重载示例	381	12.5.2 指针和对象小结	454
11.2.1 添加加法运算符	384	12.5.3 再谈定位 new 运算符	456
11.2.2 重载限制	387		
11.2.3 其他重载运算符	388	12.6 复习各种技术	459
11.3 友元	390	12.6.1 重载<<运算符	459
11.3.1 创建友元	391	12.6.2 转换函数	460
11.3.2 常用的友元: 重载<<运算符	392	12.6.3 其构造函数使用 new 的类	460
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	12.7 队列模拟	460
11.5.1 使用状态成员	404	12.7.1 队列类	461
11.5.2 为 Vector 类重载算术运算符	406	12.7.2 Customer 类	468
11.5.3 对实现的说明	408	12.7.3 ATM 模拟	471
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	12.8 总结	475
11.6.1 转换函数	415		
11.6.2 转换函数和友元函数	419	12.9 复习题	476
11.7 总结	421		
11.8 复习题	422	12.10 编程练习	477
11.9 编程练习	422		
第 12 章 类和动态内存分配	425	第 13 章 类继承	480
12.1 动态内存和类	425	13.1 一个简单的基类	481
12.1.1 复习示例和静态类成员	425	13.1.1 派生一个类	482

13.4 静态联编和动态联编	501	14.4.10 模板别名 (C++11)	593	
13.4.1 指针和引用类型的兼容性	502	14.5 总结	594	
13.4.2 虚成员函数和动态联编	503	14.6 复习题	595	
13.4.3 有关虚函数注意事项	505	14.7 编程练习	597	
13.5 访问控制: <code>protected</code>	507	第 15 章 友元、异常和其他 602		
13.6 抽象基类	508	15.1 友元	602	
13.6.1 应用 ABC 概念	510	15.1.1 友元类	602	
13.6.2 ABC 理念	516	15.1.2 友元成员函数	606	
13.7 继承和动态内存分配	516	15.1.3 其他友元关系	609	
13.7.1 第一种情况: 派生类 不使用 <code>new</code>	516	15.1.4 共同的友元	610	
13.7.2 第二种情况: 派生类 使用 <code>new</code>	517	15.2 嵌套类	611	
13.7.3 使用动态内存分配和友元的 继承示例	519	15.2.1 嵌套类和访问权限	612	
13.8 类设计回顾	523	15.2.2 模板中的嵌套	613	
13.8.1 编译器生成的成员函数	523	15.3 异常	616	
13.8.2 其他的类方法	524	15.3.1 调用 <code>abort()</code>	616	
13.8.3 公有继承的考虑因素	527	15.3.2 返回错误码	617	
13.8.4 类函数小结	530	15.3.3 异常机制	619	
13.9 总结	530	15.3.4 将对象用作异常类型	621	
13.10 复习题	531	15.3.5 异常规范和 C++11	624	
13.11 编程练习	531	15.3.6 栈解退	625	
第 14 章 C++ 中的代码重用	534	15.3.7 其他异常特性	629	
14.1 包含对象成员的类	534	15.3.8 <code>exception</code> 类	631	
14.1.1 <code>valarray</code> 类简介	535	15.3.9 异常类和继承	634	
14.1.2 <code>Student</code> 类的设计	535	15.3.10 异常何时会迷失方向	639	
14.1.3 <code>Student</code> 类示例	537	15.3.11 有关异常的注意事项	641	
14.2 私有继承	543	15.4 RTTI	642	
14.2.1 <code>Student</code> 类示例 (新版本)	543	15.4.1 RTTI 的用途	642	
14.2.2 使用包含还是私有继承	549	15.4.2 RTTI 的工作原理	642	
14.2.3 保护继承	549	15.5 类型转换运算符	649	
14.2.4 使用 <code>using</code> 重新定义访问权限	550	15.6 总结	652	
14.3 多重继承	551	15.7 复习题	653	
14.3.1 有多少 <code>Worker</code>	555	15.8 编程练习	654	
14.3.2 哪个方法	558	第 16 章 <code>string</code> 类和标准模板库 655		
14.3.3 MI 小结	567	16.1 <code>string</code> 类	655	
14.4 类模板	567	16.1.1 构造字符串	655	
14.4.1 定义类模板	568	16.1.2 <code>string</code> 类输入	659	
14.4.2 使用模板类	570	16.1.3 使用字符串	661	
14.4.3 深入探讨模板类	572	16.1.4 <code>string</code> 还提供了哪些功能	665	
14.4.4 数组模板示例和非类型参数	577	16.1.5 字符串种类	666	
14.4.5 模板多功能性	578	16.2 智能指针模板类	667	
14.4.6 模板的具体化	582	16.2.1 使用智能指针	668	
14.4.7 成员模板	584	16.2.2 有关智能指针的注意事项	670	
14.4.8 将模板用作参数	586	16.2.3 <code>unique_ptr</code> 为何优于 <code>auto_ptr</code>	672	
14.4.9 模板类和友元	588	16.2.4 选择智能指针	673	

16.3.2 可对矢量执行的操作	676	17.4.3 打开多个文件	771
16.3.3 对矢量可执行的其他操作	680	17.4.4 命令行处理技术	772
16.3.4 基于范围的 for 循环 (C++11)	684	17.4.5 文件模式	773
16.4 泛型编程	684	17.4.6 随机存取	781
16.4.1 为何使用迭代器	685	17.5 内核格式化	788
16.4.2 迭代器类型	688	17.6 总结	790
16.4.3 迭代器层次结构	689	17.7 复习题	791
16.4.4 概念、改进和模型	690	17.8 编程练习	792
16.4.5 容器种类	695	第 18 章 探讨 C++ 新标准	795
16.4.6 关联容器	702	18.1 复习前面介绍过的 C++11 功能	795
16.4.5 无序关联容器 (C++11)	707	18.1.1 新类型	795
16.5 函数对象	707	18.1.2 统一的初始化	795
16.5.1 函数符概念	707	18.1.3 声明	796
16.5.2 预定义的函数符	710	18.1.4 智能指针	798
16.5.3 自适应函数符和函数适配器	711	18.1.5 异常规范方面的修改	798
16.6 算法	713	18.1.6 作用域内枚举	799
16.6.1 算法组	713	18.1.7 对类的修改	799
16.6.2 算法的通用特征	714	18.1.8 模板和 STL 方面的修改	800
16.6.3 STL 和 string 类	715	18.1.9 右值引用	801
16.6.4 函数和容器方法	716	18.2 移动语义和右值引用	802
16.6.5 使用 STL	717	18.2.1 为何需要移动语义	802
16.7 其他库	720	18.2.2 一个移动示例	803
16.7.1 vector、valarray 和 array	720	18.2.3 移动构造函数解析	808
16.7.2 模板 initializer_list (C++11)	724	18.2.4 赋值	809
16.7.3 使用 initializer_list	725	18.2.5 强制移动	809
16.8 总结	727	18.3 新的类功能	813
16.9 复习题	728	18.3.1 特殊的成员函数	813
16.10 编程练习	728	18.3.2 默认的方法和禁用的方法	814
第 17 章 输入、输出和文件	731	18.3.3 委托构造函数	815
17.1 C++ 输入和输出概述	731	18.3.4 继承构造函数	815
17.1.1 流和缓冲区	732	18.3.5 管理虚方法：override 和 final	817
17.1.2 流、缓冲区和 iostream 文件	733	18.4 Lambda 函数	817
17.1.3 重定向	735	18.4.1 比较函数指针、函数符和 Lambda 函数	818
17.2 使用 cout 进行输出	736	18.4.2 为何使用 lambda	820
17.2.1 重载的 <> 运算符	736	18.5 包装器	822
17.2.2 其他 ostream 方法	738	18.5.1 包装器 function 及模板的 低效性	823
17.2.3 刷新输出缓冲区	740	18.5.2 修复问题	825
17.2.4 用 cout 进行格式化	741	18.5.3 其他方式	826
17.3 使用 cin 进行输入	753	18.6 可变参数模板	827
17.3.1 cin>> 如何检查输入	754	18.6.1 模板和函数参数包	827
17.3.2 流状态	756	18.6.2 展开参数包	828
17.3.3 其他 istream 类方法	759	18.6.3 在可变参数模板函数中使用 递归	828
17.3.4 其他 istream 方法	764	18.7 C++11 新增的其他功能	831
17.4 文件输入和输出	768	18.7.1 并行编程	831
17.4.1 简单的文件 I/O	768		
17.4.2 流状态检查和 is_open()	770		

18.7.2 新增的库	831	F.2.5 使用右值引用的构造函数 (C++11)	866
18.7.3 低级编程	832	F.2.6 使用一个字符的 n 个副本的构造 函数	867
18.7.4 杂项	832	F.2.7 使用区间的构造函数	867
18.8 语言变化	832	F.2.8 使用初始化列表的构造函数 (C++11)	868
18.8.1 Boost 项目	833	F.2.9 内存杂记	868
18.8.2 TR1	833	F.3 字符串存取	868
18.8.3 使用 Boost	833	F.4 基本赋值	869
18.9 接下来的任务	834	F.5 字符串搜索	869
18.10 总结	834	F.5.1 find() 系列	870
18.11 复习题	835	F.5.2 rfind() 系列	870
18.12 编程练习	838	F.5.3 find_first_of() 系列	870
附录 A 计数系统	839	F.5.4 find_last_of() 系列	871
A.1 十进制数	839	F.5.5 find_first_not_of() 系列	871
A.2 八进制整数	839	F.5.6 find_last_not_of() 系列	871
A.3 十六进制数	839	F.6 比较方法和函数	872
A.4 二进制数	840	F.7 字符串修改方法	873
A.5 二进制和十六进制	841	F.7.1 用于追加和相加的方法	873
附录 B C++保留字	842	F.7.2 其他赋值方法	874
B.1 C++关键字	842	F.7.3 插入方法	874
B.2 替代标记	842	F.7.4 清除方法	875
B.3 C++库保留名称	843	F.7.5 替换方法	875
B.4 有特殊含义的标识符	843	F.7.6 其他修改方法: copy() 和 swap()	876
附录 C ASCII 字符集	845	F.8 输出和输入	876
附录 D 运算符优先级	849	附录 G 标准模板库方法和函数	877
附录 E 其他运算符	852	G.1 STL 和 C++11	877
E.1 按位运算符	852	G.1.1 新增的容器	877
E.1.1 移位运算符	852	G.1.2 对 C++98 容器所做的修改	877
E.1.2 逻辑按位运算符	853	G.2 大部分容器都有的成员	878
E.1.3 按位运算符的替代表示	855	G.3 序列容器的其他成员	881
E.1.4 几种常用的按位运算符技术	856	G.4 set 和 map 的其他操作	883
E.2 成员解除引用运算符	857	G.4 无序关联容器 (C++11)	884
E.3 alignof (C++11)	860	G.5 STL 函数	886
E.4 noexcept (C++11)	861	G.5.1 非修改式序列操作	886
附录 F 模板类 string	862	G.5.2 修改式序列操作	890
F.1 13 种类型和一个常量	862	G.5.3 排序和相关操作	897
F.2 数据信息、构造函数及其他	863	G.5.4 数值运算	907
F.2.1 默认构造函数	865	附录 H 精选读物和网上资源	909
F.2.2 使用 C-风格字符串的构造函数	865	H.1 精选读物	909
F.2.3 使用部分 C-风格字符串的构造 函数	865	H.2 网上资源	910
F.2.4 使用左值引用的构造函数	866	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，则在过渡期到 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罝和功能，客户不得不重新学习驾驶。

然而，UNIX是在不同的计算机(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是在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。

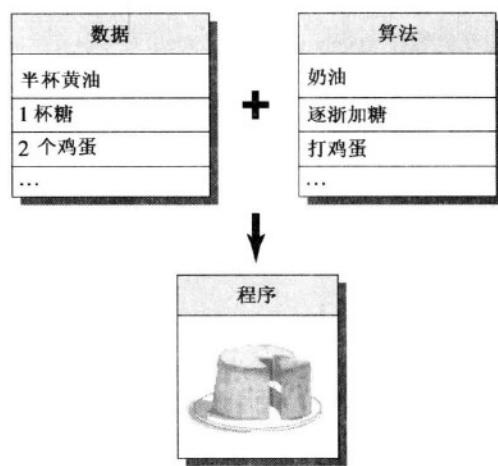


图 1.1 数据+算法=程序

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。Stroustrup 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，但如果缺少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（ISO/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。

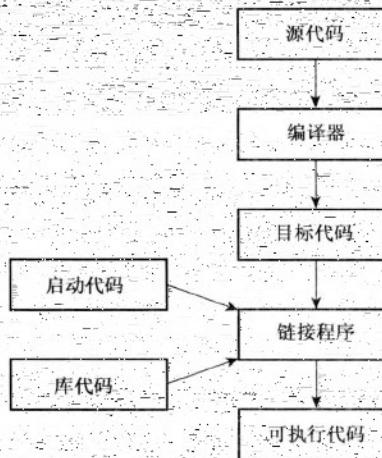


图 1.3 编程步骤

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

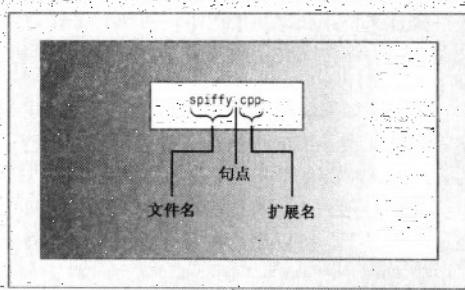


图 1.4 源文件的扩展名

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 ISO C++ 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
{
    using namespace std;                           // start of function body
    cout << "Come up and C++ me some time.";        // make definitions visible
    cout << endl;                                    // 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;  
}
```

这几句表明有一个名为`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。

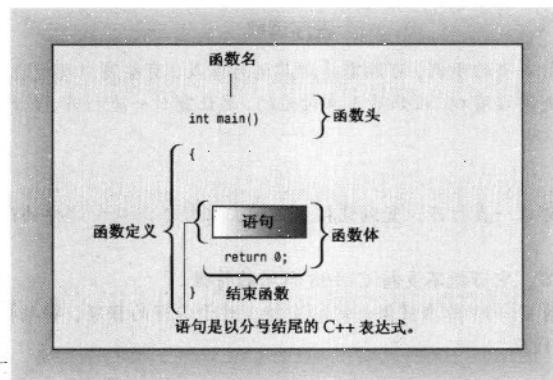


图 2.1 main() 函数

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。

2.1 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 组合成为一个复合文件, 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。这说明既可以将一条语句放

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：

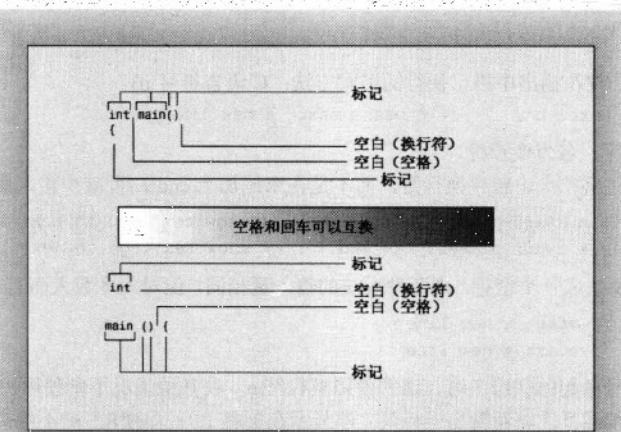


图 2.3 标记和空白

```
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。

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

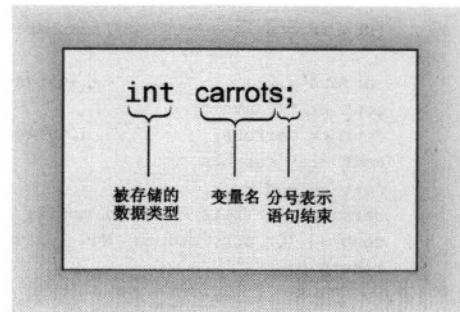


图 2.4 变量声明

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 中的第三条赋值语句表明，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）。

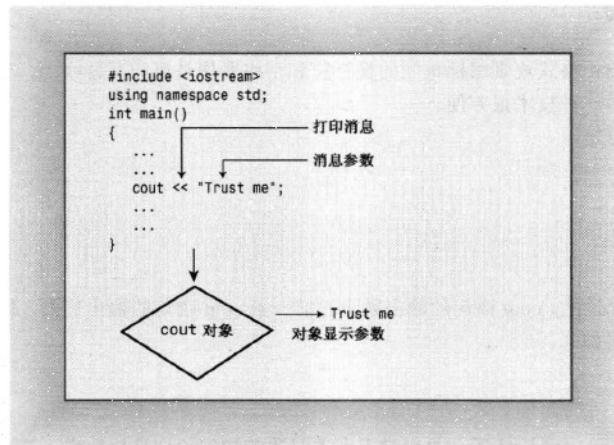


图 2.5 向对象发送消息

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。

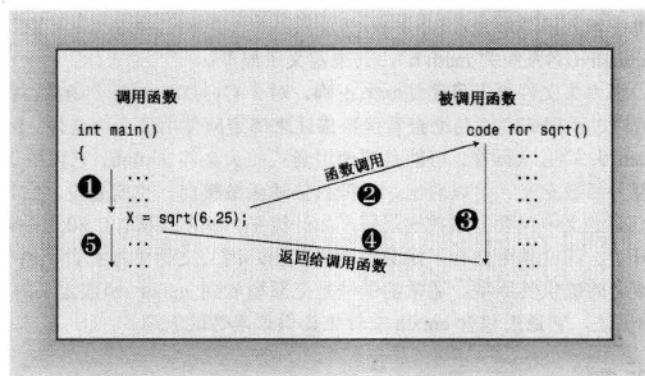


图 2.6 调用函数

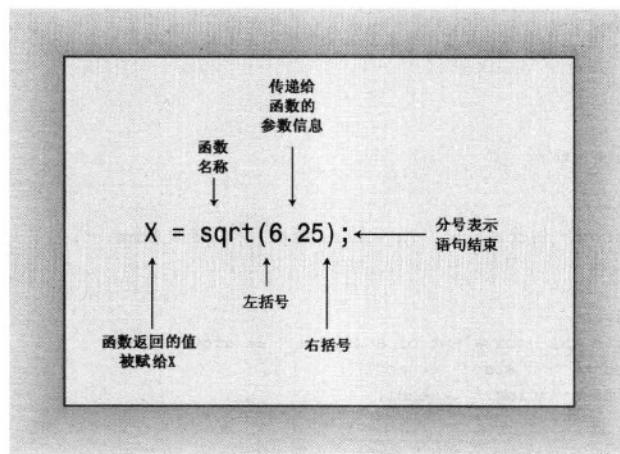


图 2.7 函数调用的句法

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。

第三种方法更好，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 1536.0 和 39.1918。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
```