

本书各版全球总销量达1300万册
为新的C++11标准重新撰写

Stanley B. Lippman
Josée Lajoie 著
Barbara E. Moo

C++ Primer (第5版) 中文版

王刚 杨巨峰 译

叶劲峰 李云 刘未鹏
陈梓瀚 侯凤林 审校

C++ Primer 第5版

顶级畅销的程序设计教程和参考书籍

为新的C++11标准重新撰写

本书针对最新发布的C++11标准进行了彻底的更新和修改，其中对C++语言权威而又全面的介绍将帮助你更快地学习这门编程语言，并且能以一种现代、高效的方式使用它。通过强调一些现代C++编程的最佳方法，作者展示了如何使用核心语言特性和标准库来编写高效、易读、强大的代码。

《C++ Primer（第5版）》从一开始就介绍C++标准库相关内容，利用标准库功能和设施来帮助你编写有用的程序，而不要求你首先掌握每个语言细节。书中很多例子都已经修订过，以使用新的语言特性，以及展示如何更好地利用它们。本书是C++新特性的值得信赖的教程，是C++核心概念和技术的权威介绍。对有经验的程序员，特别是那些迫切希望一探C++11增强特性究竟的程序员，本书也是很有价值的。

Stanley B. Lippman曾担任喷气推进实验室特别顾问，微软Visual C++开发组架构师，贝尔实验室技术部门成员，以及迪斯尼、梦工厂、皮克斯和PDI工作室的动画长片首席软件工程师。**Josée Lajoie**现供职于皮克斯，曾为IBM加拿大C/C++编译器开发团队成员，并曾担任最初的ANSI/ISO C++标准委员会核心语言特性工作组的主席。

Barbara E. Moo有近三十年的软件开发经验，她曾在AT&T工作了15年，其间与C++的发明人Bjarne Stroustrup密切合作，并曾数年担任C++开发团队的管理工作。



快速入门，收获更多

- ◎ 学习如何使用新的C++11的语言特性和标准库来快速构建健壮的程序，并感受高级语言程序设计的轻松自如。
- ◎ 通过一些展示当前最佳编码风格和程序设计技术的例子来学习。
- ◎ 理解“规则背后的原理”：为什么C++11是这样的。
- ◎ 使用大量的交叉引用来帮助你将有关联的概念和见解联系起来。
- ◎ 得益于新的学习辅助和强调关键点的课后练习，帮助你避免一些陷阱，促进你在实践中养成好的习惯，以及巩固你所学的知识。

从informit.com/title/0321714113
获取扩展示例的源码

PEARSON

www.pearson.com



策划编辑：张春雨
责任编辑：刘舫
封面设计：李玲



Stanley B. Lippman
Josée Lajoie 著
Barbara E. Moo

(第5版)

C++ 中文版 Primer

王刚 杨巨峰 译

叶劲峰 李云 刘未鹏
陈梓瀚 侯凤林 审校

电子工业出版社
Publishing House of Electronics Industry
北京•BEIJING

内 容 简 介

这本久负盛名的C++经典教程，时隔八年之久，终于迎来史无前例的重大升级。除令全球无数程序员从中受益，甚至为之迷醉的——C++大师Stanley B. Lippman的丰富实践经验，C++标准委员会原负责人Josée Lajoie对C++标准的深入理解，以及C++先驱Barbara E. Moo在C++教学方面的真知灼见外，更是基于全新的C++11标准进行了全面而彻底的内容更新。非常难能可贵的是，书中所有示例均全部采用C++11标准改写，这在经典升级版中极其罕见——充分体现了C++语言的重大进展及其全面实践。书中丰富的教学辅助内容、醒目的知识点提示，以及精心组织的编程示范，让这本书在C++领域的权威地位更加不可动摇。无论是初学者入门，或是中高级程序员提升使用，本书均为不容置疑的首选。

Authorized translation from the English language edition, entitled C++ Primer, 5E, 9780321714114 by STANLEY B. LIPPMAN; JOSEE LAJOIE; BARBARA E. MOO, published by Pearson Education, Inc., publishing as Addison-Wesley Professional, Copyright©2013 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 PUBLISHING HOUSE OF ELECTRONICS INDUSTRY Copyright ©2013

本书简体中文版专有版权由Pearson Education培生教育出版亚洲有限公司授予电子工业出版社。未经出版者预先书面许可，不得以任何方式复制或抄袭本书的任何部分。

本书简体中文版贴有Pearson Education培生教育出版集团激光防伪标签，无标签者不得销售。

版权贸易合同登记号 图字：01-2013-2487

图书在版编目（CIP）数据

C++ Primer中文版：第5版 / (美) 李普曼 (Lippman,S.B.), (美) 拉乔伊 (Lajoie,J.), (美) 默 (Moo,B.E.) 著；王刚，杨巨峰译。—北京：电子工业出版社，2013.9

书名原文：C++ Primer, 5E

ISBN 978-7-121-15535-2

I. ①C… II. ①李… ②拉… ③默… ④王… ⑤杨… III. ①C语言—程序设计 IV. ①TP312

中国版本图书馆CIP数据核字（2013）第169583号

策划编辑：张春雨

责任编辑：刘 脍

印 刷：涿州市京南印刷厂

装 订：涿州市京南印刷厂

出版发行：电子工业出版社

北京市海淀区万寿路173信箱 邮编：100036

开 本：787×1092 1/16 印张：54 字数：1521千字

印 次：2013年11月第2次印刷

印 数：20001~30000册 定价：128.00元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：(010) 88254888。

质量投诉请发邮件至zltsc@phei.com.cn，盗版侵权举报请发邮件至dbqq@phei.com.cn。

服务热线：(010) 88258888。

*To Beth,
who makes this,
and all things,
possible.*

*To Daniel and Anna,
who contain
virtually
all possibilities.
—SBL*

*To Mark and Mom,
for their
unconditional love and support.
—JL*

*To Andy,
who taught me
to program
and so much more.
—BEM*

推荐序 1

C++一直是我最为钟情的程序设计语言，我曾经在有些场合下提到“C++正在成为一门完美的程序设计语言”。从 C++标准 1998 年版本到 2011 年版本的变化，基本上印证了我的这一提法。原来版本中来不及引入的内容，以及语言机制中发现的一些缺陷，都在新版本中得以弥补和发展。比如新版标准中引入了无序容器，以弥补原版标准中对 hash 容器的缺漏；新版标准支持移动构造函数和移动赋值运算符，以减小特定场景下对象拷贝的性能开销。新版标准不仅在语法上增加了大量特性，而且在标准库里也引入大量设施，使得标准库对于 C++ 语言的重要性远超从前。

“完美的程序设计语言”，听起来很好，但代价是语言变得越来越复杂。从一个完善的类型系统或者一门程序设计语言的角度来看，新版本的 C++ 标准是一个里程碑，但是，从 C++ 学习者和使用者角度来看这未必是好事。语言的复杂性导致学习难度增加，学习周期变长；C++ 程序员写出好程序的门槛也相应提高。这差不多正是 C++ 语言这几年的现状。我相信，随着计算机科学技术的发展，这种状况未来还会加剧。即便如此，我仍然乐于看到 C++ 语言走向完美。

我与《C++ Primer》这本书的缘分从第 3 版开始，2001 年有机会将其翻译成中文版本。当时，我使用 C++ 已将近十年，通过这本书才第一次全面地梳理了实践中积累起来的 C++ 知识。本书第 3 版是对 1998 版标准的全面诠释，我相信至今无出其右者。时隔 12 年以后，这本书第 5 版出版，虽然叙述风格跟第 3 版完全不同，但它在内容上全面顾及到 2011 版 C++ 标准。第 5 版之于 2011 版标准，如同第 3 版之于 1998 版标准，必将成为经典的学习读本。

阅读这本书可以全面了解 2011 版本 C++ 标准的内容。以三位作者在 C++ 语言发展历程中的经历，本书的权威性自不容置疑：既有编译器的开发和实践，又参与 C++ 标准的制定，再加上丰富的 C++ 教学经历。如果说本书第 3 版是针对 C++ 语言的特性和设计思想来展开讲述，那么，第 5 版则更加像一本学习教程，由浅入深，并结合大量代码实例来讲述 C++ 语法和标准库。此外，由于本书的全面性，读者也可以将本书当作参考书，以备随时查阅。

本书在讲解的时候，常常会提到“编译器会如何如何”，学习语言的一个境界是把自己想象成编译器，这种要求对于一般的程序设计语言可能太高了，但是对于学习 C 和 C++ 语言是最理想的方法。像编译器一样来思考和理解 C++ 语言，如果暂时做不到，也不要紧，当有了一定的编写程序经验以后，在“揣摩”编译器行为的过程中可逐渐掌握 C++ 语法特性。因此，本书值得阅读多遍，每多读一遍，就会加深理解。可能是考虑到篇幅的原因，本书很多地方没有展开来透彻地讲解。我相信，作者们已经在深度和广度上做了较为理想的折中。

本书的另一个特色是将 C++ 的语法和标准库融为一体来介绍。C++ 标准库本身就是 C++ 语法的最佳样例，其中包含不少 C++ 高级特性的指导性用法。在我的程序设计经历中，有些 C++ 语言特性（比如虚拟继承），我只在标准库中看到过实用做法。本书贯穿始终融合了 C++ 标准库的知识和用法，这符合新版本 C++ 标准的发展和变化，也符合现代软件开发现状。

最后，结合我在工程实践中使用和倡导 C++ 语言的经验，我想提一个关于学习和使用 C++ 语言的“两面性”观点。如前所述，C++ 语言正在走向完美，所以，C++ 语言值得学习（甚至研究），这些知识可以成为一切编程的基础。然而，在实践中，不必全面地使用 C++ 语言的各种特性，而应根据工程项目的实际情况，适当取舍（譬如动态类型信息、虚拟继承、异常等特性的使用很值得商榷）。通常只鼓励使用 C++ 语言的一个子集就够了，一个值得学习和参考的例子是 Google 发布的 *Google C++ Style Guide*。尽管在工程中只使用 C++ 的子集，但全面地学习 C++ 语言仍然是必要的，毕竟 C++ 语言是一个整体，并且 C++ 标准库自身全面地使用了 C++ 语言的各种特性。我自己在过去多年的实践中就一直恪守着这种两面的做法。

很幸运，我有机会在本书正式出版以前读到中文翻译版，译文通顺，术语规范。作为经典权威之作的最新版本，本书值得拥有。

潘爱民

2013 年 8 月于杭州

推荐序 2

C++11 标准公布之后，C++社群出现了久违的热情，有人甚至叫出“C++的复兴”。指望 C++回到 20 世纪 90 年代中期那样的地位显然是昧于大势的奢望，但是 C++经历了这么多年的打磨与起伏，其在工业界的地位已经非常稳固，在很多领域里已经是不可取代也没必要被取代的统治者。新标准的出现能够大大提升 C++开发的效率和质量，因此赢得欢呼也是情理之中。在这种氛围之下，编译器实现的速度也令人惊喜。短短两年时间，从开源的 GCC、LLVM 到专有的 Visual C++ 和 Intel C++，对于新标准的追踪之快，覆盖之全，与当年 C++98 标准颁布之后迟迟不能落地的窘境相比，可谓对比强烈。当年是热情的开发者反复敦促厂商实现完整标准而不得，为此沮丧无奈，那种心情，至今记忆犹新。时过境迁，今天是编译器实现远远冲在前面，开发者倒是大大地落在了后面。

时至今日，能够基本了解 C++11 标准的程序员恐怕不多，而能够以新的 C++风格开发实践的人更是凤毛麟角。因此，今天的 C++开发者面临的一个重要任务就是快速掌握新的 C++风格和工具。

而说到教授“正宗的”C++11 编程风格，《C++ Primer（第 5 版）》如同它之前的版本一样，扮演着法定教科书的角色。

一种优秀的编程语言，一定要对于计算这件事情实现一个完整和自洽的抽象。十几年来编程语言领域的竞争，除却实现质量之外，基本上是在比拼抽象的设计。而 C 语言之所以四十年长盛不衰，根本在于它对于现代计算机提供了一个底层的高级抽象：凡是比它低的抽象都过于简陋，凡是比它高的抽象都可以用 C 语言构造出来。C++成功的原因，恰恰是因为它虽然试图提供一些高级的抽象机制，但是其根基与 C 在同一层面。正因为如此，每当你需要走过去直接与硬件对话时，C++成为 C 之外唯一有效率的选择。我的一个朋友在进行了多年的大型系统软件开发之后，不无感慨地说，C++最大的力量不在于其抽象，恰恰在于其不抽象。

话虽然如此，但是 C++之所以脱离 C 而存在，毕竟还是因为其强大的抽象能力。Bjarne Stroustrup 曾经总结说，C++同时支持 4 种不同的编程风格：C 风格、基于对象、面向对象和泛型。事实上，把微软的 COM 也算进来的话，还可以加上一种“基于组件”的风格。这么多的风格共存于一种语言，就是其强大抽象机制的证明。但是，在 C++11 以前，C++ 的抽象可以说存在若干缺陷，其中最严重的是缺少自动内存管理和对象级别的消息发送机制。今天看来，C++98 只能说是特定历史条件造成的半成品，无论是从语言机制，还是标准库完备程度来说，可以说都存在明显的、不容忽略的缺陷。其直接后果，就是优雅性的缺失和效率的降低。我本人在十年前曾经与当时中国 C++社群中不少杰出的人物交流探讨，试图从 C++98 中剪裁出一个小巧、优雅的、自成一体的子集，希望至少在日常编程中，能够在这个子集之内可以写出与当时的 Java 和 C# 同样干净明晰的代码。为此我们尝试了各种古怪的模板技巧，并且到处寻找有启发的代码和经验来构造这个语言子集，结果并不理想，甚至可以说是令人非常失望。后来我在我的博客中发表过好几篇文章，探讨所谓的 C++风格问题，其实也就是说，C++不支持简洁明快的面向对象风格，大家还不如回到基于对象甚至 C 语言的风格，最多加点模板，省一点代码量。非要面向对象的话，就必须依赖像 Qt 或者 MFC 那样的基础设施才可以。

C++11 出来之后，增强的语言机制和大为完善的标准库，为 C++ 语言的编程风格带来了革命性的变化。如果能够纯熟地运用 C++11 的新特征、新机制，那么就能够形成一种简洁优雅的 C++ 编程风格，以比以前更高的效率、更好的质量进行软件开发。对于这种新的风格，我认为“直觉、自然”是最佳的描述。也就是说，解决任何问题不必拘泥于什么笼盖一切的编程思想，也不再沉溺于各种古怪的模板技巧中无法自拔，而是能够根据那个问题本身采用最自然、最符合直觉的方式。C++ 有自己的一套思维方式，比如容器、算法、作为概念抽象的对象等，很大程度上这套思维方式确实是合乎直觉的。只有到了 C++11 这一代，C++ 语言的高级抽象才基本完备，这样一种风格才可能真正落实。因此可以说 C++11 对于 C++98 而言，不是一次简单的升级，而是一次本质的跃升。

学习新的 C++ 风格，并不是轻而易举的事情。即便对于以前已经精通 C++ 的人来说，熟练掌握 rvalue reference、move 语义，了解 unique_ptr、shared_ptr 和 weak_ptr 的完整用法，明智地使用 function/bind 和 lambda 机制，学习 C++ Concurrency 的新技术，都绝非一朝一夕之功。对于那些初学者来说，这件事情更不简单。

本书无论对于初学者还是提高者，都是最经典的教科全书。一直以来，它的特点就是完整而详细，基本上关于语言本身的问题，都可以在这本书里得到解决。而本书的另一个重要优点，就是其完全基于新的编程风格编写，所有的例子和讲解都遵循 C++11 标准所体现出来的思路和风格进行，如果能够踏下心来认真学习和练习，那么就能“一次到位”地掌握 C++11，尽管可能会比较慢。有经验的 C++ 开发者阅读这本书当然不用从头到尾，选择自己关心的内容学习 C++11 的新特性就可以，是快速提升自身能力的捷径。

差不多十年前，我提出一个观点，每一个具体的技术领域，只需要读四五本书就够了。以前的 C++ 是个例外，因为语言设计有缺陷，所以要读很多书才知道如何绕过缺陷。现在的 C++11 完全可以了，大家读四五本书就可以达到合格的水平，这恰恰是语言进步的体现。

本书是这四五本中的一本，而且是“教程+参考书”，扛梁之作，初学者的不二法门。另一本是《C++ 标准程序库》，对于 C++ 熟手来说更为快捷。Scott Meyers 的 *Effective C++* 永远是学习 C++ 者必读的，只不过这本书的第 4 版不知道什么时候出来。Anthony Williams 的 *C++ Concurrency in Action* 是学习用标准 C++ 开发并发程序的最佳选择。国内的作品，我则高度推荐陈硕的《Linux 多线程服务端编程》。这本书的名字赶跑了不少潜在的读者，所以我要特别说明一下。这本书是 C++ 开发的高水平作品，与其说是教你 how 用 C++ 写服务端开发，不如说是教你如何以服务端开发为例子提升 C++ 开发水平。前面几本书都是谈标准 C++ 自己的事情，碰到像 iostream 这样失败的标准组件也不得不硬着头皮介绍。而这本书是接地气的实践结晶，告诉你面对具体问题时应怎样权衡，C++ 里什么好用，什么不好用，为什么，等等。

今天的 C++ 学习者是非常幸运的，可以在 C++11 这个基础上大步向前，不必再因为那些语言的缺陷和过度的技巧而烦恼。大家静下心来认真读几本书，可以打下很好的基础。

孟岩

2013 年 8 月于北京

前言

难以计数的程序员已经通过旧版的《C++ Primer》学会了 C++ 语言。而在这段时间中，C++ 本身又已成熟了许多：语言本身的关注点和程序设计社区的关注点都已大大开阔，已经从主要关注机器效率转变为更多地关注编程效率。

2011 年，C++ 标准委员会发布了 ISO C++ 标准的一个重要修订版。此修订版是 C++ 进化过程中的最新一步，延续了前几个版本对编程效率的强调。新标准的主要目标是：

- 使语言更为统一，更易于教学。
- 使标准库更简单、安全、使用更高效。
- 使编写高效率的抽象和库变得更简单。

因此，在这个版本的《C++ Primer》中，我们进行了彻底的修改，使用了最新的 C++ 标准，即 C++11。为了了解新标准是如何全面影响 C++ 语言的，你可以看一下 XXIII 页至 XXV 页的新特性列表，其中列出了哪些章节涉及了 C++ 的新特性。

新标准增加的一些特性是具有普适性的，例如用于类型推断的 `auto`。这些新特性使本书中的代码更易于阅读和理解。程序（以及程序员！）可以忽略类型的细节，从而更容易集中精力于程序逻辑上来。其他一些新特性，例如智能指针和允许移动的容器，允许我们编写更为复杂的类，而又不必与错综复杂的资源管理做斗争。因此，在本书中开始讲授如何编写自己的类，会比第 4 版简单得多。旧标准中阻挡在我们前进路上的很多细节，你我都不必再担心了。

对于本书中涉及新标准定义的新特性的那些部分，我们都已用一个特殊的图标标记出来了。我们希望这些提示标记对那些已经熟悉 C++ 语言核心内容的读者是有帮助的，可以帮助他们决定将注意力投向哪里。对于那些可能尚不支持所有新特性的编译器，我们还希望这些图标能有助于解释这类编译器所给出的编译错误信息。这是因为虽然本书中几乎所有例子都已经用最新版本的 GNU 编译器编译通过，但我们知道一些读者可能尚未将编译器更新到最新版本。虽然新标准增加了大量新功能，但核心 C++ 语言并未变化，这构成了本书的大部分内容。读者可以借助这些图标来判断哪些功能可能还没有被自己的编译器所支持。

C++
11

为什么选择这本书？

现代 C++ 语言可以看作是三部分组成的：

- 低级语言，大部分继承自 C 语言。
- 现代高级语言特性，允许我们定义自己的类型以及组织大规模程序和系统。
- 标准库，它利用高级特性来提供有用的数据结构和算法。

大多数 C++ 教材按照语言进化的顺序来组织其内容。首先讲授 C++ 的 C 子集，然后将 C++ 中更为抽象的一些特性作为高级话题在书的最后进行介绍。这种方式存在两个问题：读者会陷入那些继承自低级程序设计的细节，从而由于挫折感而放弃；读者被强加学习一些坏习惯，随后又需要忘记这些内容。

我们采用一种相反的方法：从一开始就介绍一些语言特性，能让程序员忽略那些继承自低级程序设计的细节。例如，在介绍和使用内置的算术和数组类型时，我们还连同介绍和使用标准库中的类型 `string` 和 `vector`。使用这些类型的程序更易写、易理解且更少出错。

太多时候，标准库被当作一种“高级”话题来讲授。很多教材不使用标准库，而是使用基于字符数组指针和动态内存管理的低级程序设计技术。让使用这种低级技术的程序正确运行，要比编写相应的使用标准库的 C++ 代码困难得多。

贯穿全书，我们都在强调好的风格：我们想帮助读者直接养成好的习惯，而不是在获得很多很复杂的知识后再去忘掉那些坏习惯。我们特别强调那些棘手的问题，并对常见的错误想法和陷阱提出警告。

我们还注意解释规则背后的基本原理——使读者不仅知其然，还能知其所以然。我们相信，通过体会程序的工作原理，读者会更快地巩固对语言的理解。

虽然你不必为了学习本书而掌握 C 语言，但我们还是假定你了解足够多的程序设计知识，了解至少一门现代分程序结构语言，知道如何用这门语言编写、编译以及运行程序。特别是，我们假定你已经使用过变量，编写、调用过函数，也使用过编译器。

第 5 版变化的内容

这一版《C++ Primer》的新特点是用边栏图标来帮助引导读者。C++ 是一种庞大的编程语言，它提供了一些为特定程序设计问题定制的功能。其中一些功能对大型项目团队有很重要的意义，但对于小型项目开发可能并无必要。因此，并非每个程序员都需要了解每种语言特性的所有细节。我们加入这些边栏图标来帮助读者弄清哪些内容可以随后再学习，而哪些主题是更为重要的。



对于包含 C++ 语言基础内容的章节，我们用一个小人正在读书的图标加以标记。用这个图标标记的那些章节，涵盖了构成语言核心部分的主题。每个人都应该阅读并理解这些章节的内容。



对于那些涉及高级主题或特殊目的主题的章节，我们也进行了标记。在首次阅读时，这些章节可以跳过或快速浏览。我们用一叠书的图标标记这些章节，指出在这些地方，你可以放心地放下书本。快速浏览这些章节可能是一个好主意，这样你就可以知道有这些特性存在。但在真正需要在自己的程序中使用这些特性之前，没有必要花费时间仔细学习这些主题。



为了进一步引导读者的注意力，我们还用放大镜图标标记了特别复杂的概念。我们希望读者对有这种标记的章节能多花费一些时间彻底理解其中的内容。在这些章节中，至少有一些，其主题的重要性可能不是那么明显；但我们认为，你会发现这些章节涉及的主题对理解 C++ 语言原来至关重要。

交叉引用的广泛使用，是本书采用的另外一种阅读帮助。我们希望这些引用能帮助读者容易地翻阅书中的内容，同时还能在后面的例子涉及到前面的内容时容易地跳回到前面。

没有改变的是，《C++ Primer》仍是一本清晰、正确、全面的 C++ 入门教材。我们通过给出一系列复杂度逐步增加的例子来讲授这门语言，这些例子说明了语言特性，展示了如何充分用好 C++ 语言。

本书的结构

我们首先在第 I 部分和第 II 部分中介绍了 C++ 语言和标准库的基础内容。这两部分包含的内容足够你编写出有意义的程序，而不是只能写一些玩具程序。大部分程序员基本上都需要掌握本书这两部分所包含的所有内容。

除了讲授 C++ 的基础内容，第 I 部分和第 II 部分还有另外一个重要目的：通过使用标准库中定义的抽象设施，使你更加适应高级程序设计技术。标准库设施本身是一组抽象数据类型，通常用 C++ 编写。用来设计标准库的，就是任何 C++ 程序员都可以使用的用来构造类的那些语言特性。我们讲授 C++ 语言的一个经验是，在先学习了使用设计良好的抽象类型后，读者会发现理解如何构造自己的类型更容易了。

只有在经过全面的标准库使用训练，并编写了各种标准库所支持的抽象程序后，我们才真正进入到那些允许你编写自己的抽象类型的 C++ 特性中去。本书的第 III 部分和第 IV 部分介绍了如何编写类的形式的抽象类型。第 III 部分包含基础内容，第 IV 部分介绍更专门的语言特性。

在第 III 部分中，我们将介绍拷贝控制问题，以及其他一些使类能像内置类型一样容易使用的技术。类是面向对象编程和泛型编程的基础，第 III 部分也会介绍这些内容。第 IV 部分是《C++ Primer》的结束部分，它介绍了一些在组织大型复杂系统时非常有用的语言特性。此外，我们将在附录 A 中总结标准库算法。

读者帮助

本书的每一章均以一个总结和一个术语表结束，两者一起扼要回顾了这一章的大部分学习重点。读者应该将这些部分作为个人备忘录：如果你不理解某个术语，可以重新学习这一章的相应部分。

在本书中我们还使用了其他一些学习辅助：

- 重要的术语用**黑体**显示，我们假定读者已经熟悉的重要术语用**楷体**显示。每个术语都会列在章末尾的术语表中。
- 贯穿全书，我们用灰底衬托的方式来提醒读者需要注意的重要部分，对常见的陷阱提出警告，建议好的程序设计习惯，以及提供一般性的使用提示。
- 为了更好地理解语言特性间和概念间的联系，我们提供大量向前的和向后的交叉引用。
- 对重要的概念和 C++ 新程序员常常觉得困难的主题，我们提供边栏讨论。
- 学习任何程序设计语言都需要编写程序。为此，贯穿全书我们提供大量程序示例。扩展示例的源码可从下面的网址获得：

<http://www.informit.com/title/0321714113>

- 正文中切口处以“□”形式标注的页码为英文原版书中的页码，便于读者与英文原版书进行对照阅读。

关于编译器的注意事项

在撰写本书时（2012 年 7 月），编译器提供商正在努力工作，升级编译器以匹配最新的 ISO 标准。我们使用最多的编译器是 GNU 编译器 4.7.0。本书中只有一小部分特性在此编译器中尚未实现：继承构造函数、成员函数的引用限定符以及正则表达式库。

致谢

我们要特别感谢标准委员会几位现任和前任委员：Dave Abrahams、Andy Koenig、Stephan T. Lavavej、Jason Merrill、John Spicer 和 Herb Sutter 在准备本书的过程中提供的帮助。在理解新标准的一些更微妙之处，他们为我们提供了宝贵的帮助。我们还要感谢很多致力于升级 GNU 编译器以实现新标准的人们。

与旧版《C++ Primer》中一样，我们要感谢 Bjarne Stroustrup 不知疲倦地为 C++ 工作以及他和作者长时间的友谊。我们还要感谢 Alex Stepanov 的非凡洞察力，催生了标准库核心的容器和算法。最后，我们要感谢 C++ 标准委员会的所有委员，感谢他们这么多年来在净化、精炼和改进 C++ 语言方面的辛苦工作。

我们衷心感谢审稿人：Marshall Clow、Jon Kalb、Nevin Liber、Dr. C. L. Tondo、Daveed Vandevoorde 和 Steve Vinoski，他们建设性的意见帮助我们对全书做出了大大小小的改进。

最后，我们要感谢 Addison-Wesley 公司的优秀员工，他们指导了本书的整个出版过程：Peter Gordon，我们的编辑，他给了我们动力再次修改 *C++ Primer*；Kim Boedigheimer，保证了一切按计划进行；Barbara Wood，她在编辑过程中找到了大量编辑错误；还有 Elizabeth Ryan，很高兴再次和她共同工作，她指导我们完成了整个设计和生产流程。

目录

第 1 章 开始	1
1.1 编写一个简单的 C++ 程序.....	2
1.1.1 编译、运行程序	3
1.2 初识输入输出	5
1.3 注释简介	8
1.4 控制流	10
1.4.1 while 语句	10
1.4.2 for 语句	11
1.4.3 读取数量不定的输入数据	13
1.4.4 if 语句.....	15
1.5 类简介	17
1.5.1 Sales_item 类.....	17
1.5.2 初识成员函数	20
1.6 书店程序.....	21
小结	23
术语表	23
第 I 部分 C++ 基础	27
第 2 章 变量和基本类型	29
2.1 基本内置类型.....	30
2.1.1 算术类型	30
2.1.2 类型转换	32
2.1.3 字面值常量	35
2.2 变量	38
2.2.1 变量定义	38
2.2.2 变量声明和定义的关系	41
2.2.3 标识符	42
2.2.4 名字的作用域	43
2.3 复合类型	45
2.3.1 引用	45
2.3.2 指针	47
2.3.3 理解复合类型的声明	51
2.4 const 限定符	53
2.4.1 const 的引用	54
2.4.2 指针和 const	56
2.4.3 顶层 const	57
2.4.4 constexpr 和常量表达式.....	58

2.5 处理类型	60
2.5.1 类型别名	60
2.5.2 auto 类型说明符	61
2.5.3 decltype 类型指示符	62
2.6 自定义数据结构	64
2.6.1 定义 Sales_data 类型	64
2.6.2 使用 Sales_data 类	66
2.6.3 编写自己的头文件	67
小结	69
术语表	69
第 3 章 字符串、向量和数组	73
3.1 命名空间的 using 声明	74
3.2 标准库类型 string	75
3.2.1 定义和初始化 string 对象	76
3.2.2 string 对象上的操作	77
3.2.3 处理 string 对象中的字符	81
3.3 标准库类型 vector	86
3.3.1 定义和初始化 vector 对象	87
3.3.2 向 vector 对象中添加元素	90
3.3.3 其他 vector 操作	91
3.4 迭代器介绍	95
3.4.1 使用迭代器	95
3.4.2 迭代器运算	99
3.5 数组	101
3.5.1 定义和初始化内置数组	101
3.5.2 访问数组元素	103
3.5.3 指针和数组	105
3.5.4 C 风格字符串	109
3.5.5 与旧代码的接口	111
3.6 多维数组	112
小结	117
术语表	117
第 4 章 表达式	119
4.1 基础	120
4.1.1 基本概念	120
4.1.2 优先级与结合律	121
4.1.3 求值顺序	123
4.2 算术运算符	124
4.3 逻辑和关系运算符	126
4.4 赋值运算符	129
4.5 递增和递减运算符	131
4.6 成员访问运算符	133
4.7 条件运算符	134
4.8 位运算符	135

4.9 sizeof 运算符	139
4.10 逗号运算符	140
4.11 类型转换	141
4.11.1 算术转换	142
4.11.2 其他隐式类型转换	143
4.11.3 显式转换	144
4.12 运算符优先级表	147
小结	149
术语表	149
第 5 章 语句	153
5.1 简单语句	154
5.2 语句作用域	155
5.3 条件语句	156
5.3.1 if 语句	156
5.3.2 switch 语句	159
5.4 迭代语句	165
5.4.1 while 语句	165
5.4.2 传统的 for 语句	166
5.4.3 范围 for 语句	168
5.4.4 do while 语句	169
5.5 跳转语句	170
5.5.1 break 语句	170
5.5.2 continue 语句	171
5.5.3 goto 语句	172
5.6 try 语句块和异常处理	172
5.6.1 throw 表达式	173
5.6.2 try 语句块	174
5.6.3 标准异常	176
小结	178
术语表	178
第 6 章 函数	181
6.1 函数基础	182
6.1.1 局部对象	184
6.1.2 函数声明	186
6.1.3 分离式编译	186
6.2 参数传递	187
6.2.1 传值参数	187
6.2.2 传引用参数	188
6.2.3 const 形参和实参	190
6.2.4 数组形参	193
6.2.5 main: 处理命令行选项	196
6.2.6 含有可变形参的函数	197
6.3 返回类型和 return 语句	199
6.3.1 无返回值函数	200

6.3.2 有返回值函数	200
6.3.3 返回数组指针	205
6.4 函数重载	206
6.4.1 重载与作用域	210
6.5 特殊用途语言特性	211
6.5.1 默认实参	211
6.5.2 内联函数和 <code>constexpr</code> 函数	213
6.5.3 调试帮助	215
6.6 函数匹配	217
6.6.1 实参类型转换	219
6.7 函数指针	221
小结	225
术语表	225
第 7 章 类	227
7.1 定义抽象数据类型	228
7.1.1 设计 <code>Sales_data</code> 类	228
7.1.2 定义改进的 <code>Sales_data</code> 类	230
7.1.3 定义类相关的非成员函数	234
7.1.4 构造函数	235
7.1.5 拷贝、赋值和析构	239
7.2 访问控制与封装	240
7.2.1 友元	241
7.3 类的其他特性	243
7.3.1 类成员再探	243
7.3.2 返回 <code>*this</code> 的成员函数	246
7.3.3 类类型	249
7.3.4 友元再探	250
7.4 类的作用域	253
7.4.1 名字查找与类的作用域	254
7.5 构造函数再探	257
7.5.1 构造函数初始值列表	258
7.5.2 委托构造函数	261
7.5.3 默认构造函数的作用	262
7.5.4 隐式的类类型转换	263
7.5.5 聚合类	266
7.5.6 字面值常量类	267
7.6 类的静态成员	268
小结	273
术语表	273
第 II 部分 C++ 标准库	275
第 8 章 IO 库	277
8.1 IO 类	278
8.1.1 IO 对象无拷贝或赋值	279
8.1.2 条件状态	279

8.1.3 管理输出缓冲	281
8.2 文件输入输出	283
8.2.1 使用文件流对象	284
8.2.2 文件模式	286
8.3 string 流	287
8.3.1 使用 istringstream	287
8.3.2 使用 ostringstream	289
小结	290
术语表	290
第 9 章 顺序容器	291
9.1 顺序容器概述	292
9.2 容器库概览	294
9.2.1 迭代器	296
9.2.2 容器类型成员	297
9.2.3 begin 和 end 成员	298
9.2.4 容器定义和初始化	299
9.2.5 赋值和 swap	302
9.2.6 容器大小操作	304
9.2.7 关系运算符	304
9.3 顺序容器操作	305
9.3.1 向顺序容器添加元素	305
9.3.2 访问元素	309
9.3.3 删除元素	311
9.3.4 特殊的 forward_list 操作	312
9.3.5 改变容器大小	314
9.3.6 容器操作可能使迭代器失效	315
9.4 vector 对象是如何增长的	317
9.5 额外的 string 操作	320
9.5.1 构造 string 的其他方法	321
9.5.2 改变 string 的其他方法	322
9.5.3 string 搜索操作	325
9.5.4 compare 函数	327
9.5.5 数值转换	327
9.6 容器适配器	329
小结	332
术语表	332
第 10 章 泛型算法	335
10.1 概述	336
10.2 初识泛型算法	338
10.2.1 只读算法	338
10.2.2 写容器元素的算法	339
10.2.3 重排容器元素的算法	342
10.3 定制操作	344
10.3.1 向算法传递函数	344

10.3.2 lambda 表达式	345
10.3.3 lambda 捕获和返回	349
10.3.4 参数绑定	354
10.4 再探迭代器	357
10.4.1 插入迭代器	358
10.4.2 iostream 迭代器	359
10.4.3 反向迭代器	363
10.5 泛型算法结构	365
10.5.1 5 类迭代器	365
10.5.2 算法形参模式	367
10.5.3 算法命名规范	368
10.6 特定容器算法	369
小结	371
术语表	371
第 11 章 关联容器	373
11.1 使用关联容器	374
11.2 关联容器概述	376
11.2.1 定义关联容器	376
11.2.2 关键字类型的要求	378
11.2.3 pair 类型	379
11.3 关联容器操作	381
11.3.1 关联容器迭代器	382
11.3.2 添加元素	383
11.3.3 删除元素	386
11.3.4 map 的下标操作	387
11.3.5 访问元素	388
11.3.6 一个单词转换的 map	391
11.4 无序容器	394
小结	397
术语表	397
第 12 章 动态内存	399
12.1 动态内存与智能指针	400
12.1.1 shared_ptr 类	400
12.1.2 直接管理内存	407
12.1.3 shared_ptr 和 new 结合使用	412
12.1.4 智能指针和异常	415
12.1.5 unique_ptr	417
12.1.6 weak_ptr	420
12.2 动态数组	423
12.2.1 new 和数组	423
12.2.2 allocator 类	427
12.3 使用标准库：文本查询程序	430
12.3.1 文本查询程序设计	430
12.3.2 文本查询程序类的定义	432

小结	436
术语表	436
第III部分 类设计者的工具	437
第 13 章 拷贝控制	439
13.1 拷贝、赋值与销毁	440
13.1.1 拷贝构造函数	440
13.1.2 拷贝赋值运算符	443
13.1.3 析构函数	444
13.1.4 三/五法则	447
13.1.5 使用=default	449
13.1.6 阻止拷贝	449
13.2 拷贝控制和资源管理	452
13.2.1 行为像值的类	453
13.2.2 定义行为像指针的类	455
13.3 交换操作	457
13.4 拷贝控制示例	460
13.5 动态内存管理类	464
13.6 对象移动	470
13.6.1 右值引用	471
13.6.2 移动构造函数和移动赋值运算符	473
13.6.3 右值引用和成员函数	481
小结	486
术语表	486
第 14 章 重载运算与类型转换	489
14.1 基本概念	490
14.2 输入和输出运算符	494
14.2.1 重载输出运算符<<	494
14.2.2 重载输入运算符>>	495
14.3 算术和关系运算符	497
14.3.1 相等运算符	497
14.3.2 关系运算符	498
14.4 赋值运算符	499
14.5 下标运算符	501
14.6 递增和递减运算符	502
14.7 成员访问运算符	504
14.8 函数调用运算符	506
14.8.1 lambda 是函数对象	507
14.8.2 标准库定义的函数对象	509
14.8.3 可调用对象与 function	511
14.9 重载、类型转换与运算符	514
14.9.1 类型转换运算符	514
14.9.2 避免有二义性的类型转换	517
14.9.3 函数匹配与重载运算符	521
小结	523
术语表	523

第 15 章 面向对象程序设计	525
15.1 OOP：概述	526
15.2 定义基类和派生类	527
15.2.1 定义基类	528
15.2.2 定义派生类	529
15.2.3 类型转换与继承	534
15.3 虚函数	536
15.4 抽象基类	540
15.5 访问控制与继承	542
15.6 继承中的类作用域	547
15.7 构造函数与拷贝控制	551
15.7.1 虚析构函数	552
15.7.2 合成拷贝控制与继承	552
15.7.3 派生类的拷贝控制成员	554
15.7.4 继承的构造函数	557
15.8 容器与继承	558
15.8.1 编写 Basket 类	559
15.9 文本查询程序再探	562
15.9.1 面向对象的解决方案	563
15.9.2 Query_base 类和 Query 类	567
15.9.3 派生类	568
15.9.4 eval 函数	571
小结	575
术语表	575
第 16 章 模板与泛型编程	577
16.1 定义模板	578
16.1.1 函数模板	578
16.1.2 类模板	583
16.1.3 模板参数	592
16.1.4 成员模板	595
16.1.5 控制实例化	597
16.1.6 效率与灵活性	599
16.2 模板实参推断	600
16.2.1 类型转换与模板类型参数	601
16.2.2 函数模板显式实参	603
16.2.3 尾置返回类型与类型转换	604
16.2.4 函数指针和实参推断	607
16.2.5 模板实参推断和引用	608
16.2.6 理解 std::move	610
16.2.7 转发	612
16.3 重载与模板	614
16.4 可变参数模板	618
16.4.1 编写可变参数函数模板	620
16.4.2 包扩展	621
16.4.3 转发参数包	622

16.5 模板特例化	624
小结	630
术语表	630
第IV部分 高级主题	633
第 17 章 标准库特殊设施	635
17.1 tuple 类型	636
17.1.1 定义和初始化 tuple	637
17.1.2 使用 tuple 返回多个值	638
17.2 bitset 类型	640
17.2.1 定义和初始化 bitset	641
17.2.2 bitset 操作	643
17.3 正则表达式	645
17.3.1 使用正则表达式库	646
17.3.2 匹配与 Regex 迭代器类型	650
17.3.3 使用子表达式	653
17.3.4 使用 regex_replace	657
17.4 随机数	659
17.4.1 随机数引擎和分布	660
17.4.2 其他随机数分布	663
17.5 IO 库再探	666
17.5.1 格式化输入与输出	666
17.5.2 未格式化的输入/输出操作	673
17.5.3 流随机访问	676
小结	680
术语表	680
第 18 章 用于大型程序的工具	683
18.1 异常处理	684
18.1.1 抛出异常	684
18.1.2 捕获异常	687
18.1.3 函数 try 语句块与构造函数	689
18.1.4 noexcept 异常说明	690
18.1.5 异常类层次	693
18.2 命名空间	695
18.2.1 命名空间定义	695
18.2.2 使用命名空间成员	701
18.2.3 类、命名空间与作用域	705
18.2.4 重载与命名空间	708
18.3 多重继承与虚继承	710
18.3.1 多重继承	711
18.3.2 类型转换与多个基类	713
18.3.3 多重继承下的类作用域	715
18.3.4 虚继承	717
18.3.5 构造函数与虚继承	720
小结	722
术语表	722

第 19 章 特殊工具与技术	725
19.1 控制内存分配	726
19.1.1 重载 new 和 delete	726
19.1.2 定位 new 表达式	729
19.2 运行时类型识别	730
19.2.1 dynamic_cast 运算符	730
19.2.2 typeid 运算符	732
19.2.3 使用 RTTI	733
19.2.4 type_info 类	735
19.3 枚举类型	736
19.4 类成员指针	739
19.4.1 数据成员指针	740
19.4.2 成员函数指针	741
19.4.3 将成员函数用作可调用对象	744
19.5 嵌套类	746
19.6 union：一种节省空间的类	749
19.7 局部类	754
19.8 固有的不可移植的特性	755
19.8.1 位域	756
19.8.2 volatile 限定符	757
19.8.3 链接指示：extern "C"	758
小结	762
术语表	762
附录 A 标准库	765
A.1 标准库名字和头文件	766
A.2 算法概览	770
A.2.1 查找对象的算法	771
A.2.2 其他只读算法	772
A.2.3 二分搜索算法	772
A.2.4 写容器元素的算法	773
A.2.5 划分与排序算法	775
A.2.6 通用重排操作	776
A.2.7 排列算法	778
A.2.8 有序序列的集合算法	778
A.2.9 最小值和最大值	779
A.2.10 数值算法	780
A.3 随机数	781
A.3.1 随机数分布	781
A.3.2 随机数引擎	783
索引	785

C++11 的新特性

2.1.1 long long 类型.....	31
2.2.1 列表初始化.....	39
2.3.2 nullptr 常量.....	48
2.4.4 constexpr 变量.....	59
2.5.1 类型别名声明.....	60
2.5.2 auto 类型指示符.....	61
2.5.3 decltype 类型指示符.....	62
2.6.1 类内初始化.....	65
3.2.2 使用 auto 或 decltype 缩写类型.....	79
3.2.3 范围 for 语句.....	82
3.3 定义 vector 对象的 vector（向量的向量）.....	87
3.3.1 vector 对象的列表初始化.....	88
3.4.1 容器的 cbegin 和 cend 函数.....	98
3.5.3 标准库 begin 和 end 函数.....	106
3.6 使用 auto 和 decltype 简化声明.....	115
4.2 除法的舍入规则.....	125
4.4 用大括号包围的值列表赋值.....	129
4.9 将 sizeof 用于类成员.....	139
5.4.3 范围 for 语句.....	168
6.2.6 标准库 initializer_list 类.....	197
6.3.2 列表初始化返回值.....	203
6.3.3 定义尾置返回类型.....	206
6.3.3 使用 decltype 简化返回类型定义.....	206
6.5.2 constexpr 函数.....	214
7.1.4 使用=default 生成默认构造函数.....	237
7.3.1 类对象成员的类内初始化.....	246
7.5.2 委托构造函数.....	261
7.5.6 constexpr 构造函数.....	268
8.2.1 用 string 对象处理文件名.....	284
9.1 array 和 forward_list 容器.....	293
9.2.3 容器的 cbegin 和 cend 函数.....	298
9.2.4 容器的列表初始化.....	300
9.2.5 容器的非成员函数 swap.....	303
9.3.1 容器 insert 成员的返回类型.....	308
9.3.1 容器的 emplace 成员.....	308
9.4 shrink_to_fit.....	318
9.5.5 string 的数值转换函数.....	327
10.3.2 lambda 表达式.....	346

10.3.3 lambda 表达式中的尾置返回类型	353
10.3.4 标准库 bind 函数	354
11.2.1 关联容器的列表初始化	377
11.2.3 列表初始化 pair 的返回类型	380
11.3.2 pair 的列表初始化	384
11.4 无序容器	394
12.1 智能指针	400
12.1.1 shared_ptr 类	400
12.1.2 动态分配对象的列表初始化	407
12.1.2 auto 和动态分配	408
12.1.5 unique_ptr 类	417
12.1.6 weak_ptr 类	420
12.2.1 范围 for 语句不能应用于动态分配数组	424
12.2.1 动态分配数组的列表初始化	424
12.2.1 auto 不能用于分配数组	424
12.2.2 allocator::construct 可使用任意构造函数	428
13.1.5 将=default 用于拷贝控制成员	449
13.1.6 使用=delete 阻止拷贝类对象	449
13.5 用移动类对象代替拷贝类对象	469
13.6.1 右值引用	471
13.6.1 标准库 move 函数	472
13.6.2 移动构造函数和移动赋值	473
13.6.2 移动构造函数通常应该是 noexcept	473
13.6.2 移动迭代器	480
13.6.3 引用限定成员函数	483
14.8.3 function 类模板	512
14.9.1 explicit 类型转换运算符	516
15.2.2 虚函数的 override 指示符	530
15.2.2 通过定义类为 final 来阻止继承	533
15.3 虚函数的 override 和 final 指示符	538
15.7.2 删除的拷贝控制和继承	553
15.7.4 继承的构造函数	557
16.1.2 声明模板类型形参为友元	590
16.1.2 模板类型别名	590
16.1.3 模板函数的默认模板参数	594
16.1.5 实例化的显式控制	597
16.2.3 模板函数与尾置返回类型	605
16.2.5 引用折叠规则	609
16.2.6 用 static_cast 将左值转换为右值	612
16.2.7 标准库 forward 函数	614
16.4 可变参数模板	618
16.4 sizeof... 运算符	619
16.4.3 可变参数模板与转发	622
17.1 标准库 tuple 类模板	636

17.2.2	新的 <code>bitset</code> 运算	643
17.3	正则表达式库	645
17.4	随机数库	659
17.5.1	浮点数格式控制	670
18.1.4	<code>noexcept</code> 异常指示符	690
18.1.4	<code>noexcept</code> 运算符	691
18.2.1	内联命名空间	699
18.3.1	继承的构造函数与多重继承	712
19.3	有作用域的 <code>enum</code>	736
19.3	说明类型用于保存 <code>enum</code> 对象	738
19.3	<code>enum</code> 的前置声明	738
19.4.3	标准库 <code>mem_fn</code> 类模板	746
19.6	类类型的 <code>union</code> 成员	751

第1章 开始

内容

1.1 编写一个简单的 C++ 程序	2
1.2 初识输入输出	5
1.3 注释简介	8
1.4 控制流	10
1.5 类简介	17
1.6 书店程序	21
小结	23
术语表	23

本章介绍 C++ 的大部分基础内容：类型、变量、表达式、语句及函数。在这个过程中，我们会简要介绍如何编译及运行程序。

在学习完本章并认真完成练习之后，你将具备编写、编译及运行简单程序的能力。后续章节将假定你已掌握本章中介绍的语言特性，并将更详细地解释这些特性。

2 学习一门新的程序设计语言的最好方法就是练习编写程序。在本章中，我们将编写一个程序来解决简单的书店问题。

我们的书店保存所有销售记录的档案，每条记录保存了某本书的一次销售的信息（一册或多册）。每条记录包含三个数据项：

0-201-70353-X 4 24.99

第一项是书的 ISBN 号（国际标准书号，一本书的唯一标识），第二项是售出的册数，最后一项是书的单价。有时，书店老板需要查询此档案，计算每本书的销售量、销售额及平均售价。

为了编写这个程序，我们需要使用若干 C++的基本特性。而且，我们需要了解如何编译及运行程序。

虽然我们还没有编写这个程序，但显然它必须

- 定义变量
- 进行输入和输出
- 使用数据结构保存数据
- 检测两条记录是否有相同的 ISBN
- 包含一个循环来处理销售档案中的每条记录

我们首先介绍如何用 C++来解决这些子问题，然后编写书店程序。

1.1 编写一个简单的 C++程序

每个 C++程序都包含一个或多个函数（function），其中一个必须命名为 **main**。操作系统通过调用 **main** 来运行 C++程序。下面是一个非常简单的 **main** 函数，它什么也不干，只是返回给操作系统一个值：

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

一个函数的定义包含四部分：返回类型（return type）、函数名（function name）、一个括号包围的形参列表（parameter list，允许为空）以及函数体（function body）。虽然 **main** 函数在某种程度上比较特殊，但其定义与其他函数是一样的。

在本例中，**main** 的形参列表是空的（() 中什么也没有）。6.2.5 节（第 196 页）将会讨论 **main** 的其他形参类型。

main 函数的返回类型必须为 **int**，即整数类型。**int** 类型是一种内置类型（built-in type），即语言自身定义的类型。

函数定义的最后一部分是函数体，它是一个以左花括号（curly brace）开始，以右花括号结束的语句块（block of statements）：

```
{  
    return 0;  
}
```

这个语句块中唯一的一条语句是 **return**，它结束函数的执行。在本例中，**return**

还会向调用者返回一个值。当 `return` 语句包括一个值时，此返回值的类型必须与函数的返回类型相容。在本例中，`main` 的返回类型是 `int`，而返回值 0 的确是一个 `int` 类型的值。



请注意，`return` 语句末尾的分号。在 C++ 中，大多数 C++ 语句以分号表示结束。它们很容易被忽略，但如果忘记了写分号，就会导致莫名其妙的编译错误。

在大多数系统中，`main` 的返回值被用来指示状态。返回值 0 表明成功，非 0 的返回值的含义由系统定义，通常用来指出错误类型。

重要概念：类型

类型是程序设计最基本的概念之一，在本书中我们会反复遇到它。一种类型不仅定义了数据元素的内容，还定义了这类数据上可以进行的运算。

程序所处理的数据都保存在变量中，而每个变量都有自己的类型。如果一个名为 `v` 的变量的类型为 `T`，我们通常说“`v` 具有类型 `T`”，或等价的，“`v` 是一个 `T` 类型变量”。

1.1.1 编译、运行程序

编写好程序后，我们就需要编译它。如何编译程序依赖于你使用的操作系统和编译器。你所使用的特定编译器的相关使用细节，请查阅参考手册或询问经验丰富的同事。

很多 PC 机上的编译器都具备集成开发环境（Integrated Developed Environment, IDE），将编译器与其他程序创建和分析工具包装在一起。在开发大型程序时，这类集成环境可能是非常有用的工具，但需要一些时间来学习如何高效地使用它们。学习如何使用这类开发环境已经超出了本书的范围。

大部分编译器，包括集成 IDE 的编译器，都会提供一个命令行界面。除非你已经了解 IDE，否则你会觉得借助命令行界面开始学习 C++ 还是很容易的。这种学习方式的好处是，可以先将精力集中于 C++ 语言本身（而不是一些开发工具），而且，一旦你掌握了语言，IDE 通常是很容易学习的。

程序源文件命名约定

无论你使用命令行界面或者 IDE，大多数编译器都要求程序源码存储在一个或多个文件中。程序文件通常被称为源文件（source file）。在大多数系统中，源文件的名字以一个后缀为结尾，后缀是由一个句点后接一个或多个字符组成的。后缀告诉系统这个文件是一个 C++ 程序。不同编译器使用不同的后缀命名约定，最常见的包括 `.cc`、`.cxx`、`.cpp`、`.cp` 及 `.C`。4

从命令行运行编译器

如果我们正在使用命令行界面，那么通常是在一个控制台窗口内（例如 UNIX 系统中的外壳程序窗口或者 Windows 系统中的命令提示符窗口）编译程序。假定我们的 `main` 程序保存在文件 `prog1.cc` 中，可以用如下命令来编译它

```
$ CC prog1.cc
```

其中，`CC` 是编译器程序的名字，`$` 是系统提示符。编译器生成一个可执行文件。Windows 系统会将这个可执行文件命名为 `prog1.exe`。UNIX 系统中的编译器通常将可执行文件命名为 `a.out`。

为了在 Windows 系统中运行一个可执行文件，我们需要提供可执行文件的文件名，可

以忽略其扩展名.exe:

```
$ prog1
```

在一些系统中，即使文件就在当前目录或文件夹中，你也必须显式指出文件的位置。在此情况下，我们可以键入

```
$ .\prog1
```

“.”后跟一个反斜线指出该文件在当前目录中。

为了在 UNIX 系统中运行一个可执行文件，我们需要使用全文件名，包括文件扩展名：

```
$ a.out
```

如果需要指定文件位置，需要用一个“.”后跟一个斜线来指出可执行文件位于当前目录中。

```
$ ./a.out
```

访问 main 的返回值的方法依赖于系统。在 UNIX 和 Windows 系统中，执行完一个程序后，都可以通过 echo 命令获得其返回值。

在 UNIX 系统中，通过如下命令获得状态：

```
$ echo $?
```

在 Windows 系统中查看状态可键入：

```
$ echo %ERRORLEVEL%
```

5 >

运行 GNU 或微软编译器

在不同操作和编译器系统中，运行 C++ 编译器的命令也各不相同。最常用的编译器是 GNU 编译器和微软 Visual Studio 编译器。默认情况下，运行 GNU 编译器的命令是 g++:

```
$ g++ -o prog1 prog1.cc
```

此处，\$ 是系统提示符。-o prog1 是编译器参数，指定了可执行文件的文件名。在不同的操作系统中，此命令生成一个名为 prog1 或 prog1.exe 的可执行文件。在 UNIX 系统中，可执行文件没有后缀；在 Windows 系统中，后缀为.exe。如果省略了 -o prog1 参数，在 UNIX 系统中编译器会生成一个名为 a.out 的可执行文件，在 Windows 系统中则会生成一个名为 a.exe 的可执行文件（注意：根据使用的 GNU 编译器的版本，你可能需要指定 -std=c++0x 参数来打开对 C++11 的支持）。

运行微软 Visual Studio 2010 编译器的命令为 cl:

```
C:\Users\me\Programs> cl /EHsc prog1.cpp
```

此处，C:\Users\me\Programs> 是系统提示符，\Users\me\Programs 是当前目录名（即当前文件夹）。命令 cl 调用编译器，/EHsc 是编译器选项，用来打开标准异常处理。微软编译器会自动生成一个可执行文件，其名字与第一个源文件名对应。可执行文件的文件名与源文件名相同，后缀为.exe。在此例中，可执行文件的文件名为 prog1.exe。

编译器通常都包含一些选项，能对有问题的程序结构发出警告。打开这些选项通常是一个好习惯。我们习惯在 GNU 编译器中使用 -Wall 选项，在微软编译器中则使用 /W4。

更详细的信息请查阅你使用的编译器的参考手册。

1.1 节练习

练习 1.1: 查阅你使用的编译器的文档，确定它所使用的文件命名约定。编译并运行第 2 页的 main 程序。

练习 1.2: 改写程序，让它返回 -1。返回值 -1 通常被当作程序错误的标识。重新编译并运行你的程序，观察你的系统如何处理 main 返回的错误标识。

1.2 初识输入输出

C++ 语言并未定义任何输入输出 (IO) 语句，取而代之，包含了一个全面的 **标准库** (standard library) 来提供 IO 机制（以及很多其他设施）。对于很多用途，包括本书中的示例来说，我们只需了解 IO 库中一部分基本概念和操作。

本书中的很多示例都使用了 **iostream** 库。**iostream** 库包含两个基础类型 **istream** 和 **ostream**，分别表示输入流和输出流。一个流就是一个字符序列，是从 IO 设备读出或写入 IO 设备的。术语“流”(stream) 想要表达的是，随着时间的推移，字符是顺序生成或消耗的。

标准输入输出对象

6

标准库定义了 4 个 IO 对象。为了处理输入，我们使用一个名为 **cin** (发音为 see-in) 的 **istream** 类型的对象。这个对象也被称为 **标准输入** (standard input)。对于输出，我们使用一个名为 **cout** (发音为 see-out) 的 **ostream** 类型的对象。此对象也被称为 **标准输出** (standard output)。标准库还定义了其他两个 **ostream** 对象，名为 **cerr** 和 **clog** (发音分别为 see-err 和 see-log)。我们通常用 **cerr** 来输出警告和错误消息，因此它也被称为 **标准错误** (standard error)。而 **clog** 用来输出程序运行时的一般性信息。

系统通常将程序所运行的窗口与这些对象关联起来。因此，当我们读取 **cin**，数据将从程序正在运行的窗口读入，当我们向 **cout**、**cerr** 和 **clog** 写入数据时，将会写到同一个窗口。

一个使用 IO 库的程序

在书店程序中，我们需要将多条记录合并成单一的汇总记录。作为一个相关的，但更简单的问题，我们先来看一下如何将两个数相加。通过使用 IO 库，我们可以扩展 main 程序，使之能提示用户输入两个数，然后输出它们的和：

```
#include <iostream>
int main()
{
    std::cout << "Enter two numbers:" << std::endl;
    int v1 = 0, v2 = 0;
    std::cin >> v1 >> v2;
    std::cout << "The sum of " << v1 << " and " << v2
        << " is " << v1 + v2 << std::endl;
    return 0;
}
```

这个程序开始时在用户屏幕打印

Enter two numbers:

然后等待用户输入。如果用户键入

3 7

然后键入一个回车，则程序产生如下输出：

The sum of 3 and 7 is 10

程序的第一行

```
#include <iostream>
```

告诉编译器我们想要使用 `iostream` 库。尖括号中的名字（本例中是 `iostream`）指出了一个头文件（header）。每个使用标准库设施的程序都必须包含相关的头文件。`#include` 指令和头文件的名字必须写在同一行中。通常情况下，`#include` 指令必须出现在所有函数之外。我们一般将一个程序的所有`#include` 指令都放在源文件的开始位置。

向流写入数据

`main` 的函数体的第一条语句执行了一个表达式（expression）。在 C++ 中，一个表达式产生一个计算结果，它由一个或多个运算对象和（通常是）一个运算符组成。这条语句中的表达式使用了输出运算符（`<<`）在标准输出上打印消息：

```
std::cout << "Enter two numbers:" << std::endl;
```

`<<` 运算符接受两个运算对象：左侧的运算对象必须是一个 `ostream` 对象，右侧的运算对象是要打印的值。此运算符将给定的值写到给定的 `ostream` 对象中。输出运算符的计算结果就是其左侧运算对象。即，计算结果就是我们写入给定值的那个 `ostream` 对象。

我们的输出语句使用了两次`<<` 运算符。因为此运算符返回其左侧的运算对象，因此第一个运算符的结果成为了第二个运算符的左侧运算对象。这样，我们就可以将输出请求连接起来。因此，我们的表达式等价于

```
(std::cout << "Enter two numbers:") << std::endl;
```

链中每个运算符的左侧运算对象都是相同的，在本例中是 `std::cout`。我们也可以用两条语句生成相同的输出：

```
std::cout << "Enter two numbers:";  
std::cout << std::endl;
```

第一个输出运算符给用户打印一条消息。这个消息是一个字符串字面值常量（string literal），是用一对双引号包围的字符序列。在双引号之间的文本被打印到标准输出。

第二个运算符打印 `endl`，这是一个被称为操纵符（manipulator）的特殊值。写入 `endl` 的效果是结束当前行，并将与设备关联的缓冲区（buffer）中的内容刷到设备中。缓冲刷新操作可以保证到目前为止程序所产生的所有输出都真正写入输出流中，而不是仅停留在内存中等待写入流。



程序员常常在调试时添加打印语句。这类语句应该保证“一直”刷新流。否则，如果程序崩溃，输出可能还留在缓冲区中，从而导致关于程序崩溃位置的错误推断。

使用标准库中的名字

细心的读者可能会注意到这个程序使用了 `std::cout` 和 `std::endl`, 而不是直接的 `cout` 和 `endl`。前缀 `std::` 指出名字 `cout` 和 `endl` 是定义在名为 **std** 的命名空间 (namespace) 中的。命名空间可以帮助我们避免不经意的名字定义冲突, 以及使用库中相同名字导致的冲突。标准库定义的所有名字都在命名空间 `std` 中。8

通过命名空间使用标准库有一个副作用: 当使用标准库中的一个名字时, 必须显式说明我们想使用来自命名空间 `std` 中的名字。例如, 需要写出 `std::cout`, 通过使用作用域运算符 (`::`) 来指出我们想使用定义在命名空间 `std` 中的名字 `cout`。3.1 节 (第 74 页) 将给出一个更简单的访问标准库中名字的方法。

从流读取数据

在提示用户输入数据之后, 接下来我们希望读入用户的输入。首先定义两个名为 `v1` 和 `v2` 的变量 (variable) 来保存输入:

```
int v1 = 0, v2 = 0;
```

我们将这两个变量定义为 `int` 类型, `int` 是一种内置类型, 用来表示整数。还将它们初始化 (initialize) 为 0。初始化一个变量, 就是在变量创建的同时为它赋予一个值。

下一条语句是

```
std::cin >> v1 >> v2;
```

它读入输入数据。输入运算符 (`>>`) 与输出运算符类似, 它接受一个 `istream` 作为其左侧运算对象, 接受一个对象作为其右侧运算对象。它从给定的 `istream` 读入数据, 并存入给定对象中。与输出运算符类似, 输入运算符返回其左侧运算对象作为其计算结果。因此, 此表达式等价于

```
(std::cin >> v1) >> v2;
```

由于此运算符返回其左侧运算对象, 因此我们可以将一系列输入请求合并到单一语句中。本例中的输入操作从 `std::cin` 读入两个值, 并将第一个值存入 `v1`, 将第二个值存入 `v2`。换句话说, 它与下面两条语句的执行结果是一样的

```
std::cin >> v1;
std::cin >> v2;
```

完成程序

剩下的就是打印计算结果了:

```
std::cout << "The sum of " << v1 << " and " << v2
<< " is " << v1 + v2 << std::endl;
```

这条语句虽然比提示用户输入的打印语句更长, 但原理上是一样的, 它将每个运算对象打印在标准输出上。本例一个有意思的地方在于, 运算对象并不都是相同类型的值。某些运算对象是字符串字面常量, 例如 "The sum of "。其他运算对象则是 `int` 值, 如 `v1`、`v2` 以及算术表达式 `v1+v2` 的计算结果。标准库定义了不同版本的输入输出运算符, 来处理这些不同类型的运算对象。

9

1.2 节练习

练习 1.3: 编写程序，在标准输出上打印 Hello, World。

练习 1.4: 我们的程序使用加法运算符+来将两个数相加。编写程序使用乘法运算符*，来打印两个数的积。

练习 1.5: 我们将所有输出操作放在一条很长的语句中。重写程序，将每个运算对象的打印操作放在一条独立的语句中。

练习 1.6: 解释下面程序片段是否合法。

```
std::cout << "The sum of " << v1;
           << " and " << v2;
           << " is " << v1 + v2 << std::endl;
```

如果程序是合法的，它输出什么？如果程序不合法，原因何在？应该如何修正？

1.3 注释简介

在程序变得更复杂之前，我们应该了解一下 C++ 是如何处理注释（comments）的。注释可以帮助人类读者理解程序。注释通常用于概述算法，确定变量的用途，或者解释晦涩难懂的代码段。编译器会忽略注释，因此注释对程序的行为或性能不会有任何影响。

虽然编译器会忽略注释，但读者并不会。即使系统文档的其他部分已经过时，程序员也倾向于相信注释的内容是正确可信的。因此，错误的注释比完全没有注释更糟糕，因为它会误导读者。因此，当你修改代码时，不要忘记同时更新注释！

C++ 中注释的种类

C++ 中有两种注释：单行注释和界定符对注释。单行注释以双斜线（//）开始，以换行符结束。当前行双斜线右侧的所有内容都会被编译器忽略，这种注释可以包含任何文本，包括额外的双斜线。

另一种注释使用继承自 C 语言的两个界定符（/* 和 */）。这种注释以/* 开始，以*/ 结束，可以包含除*/外的任意内容，包括换行符。编译器将落在/* 和 */之间的所有内容都当作注释。

注释界定符可以放置于任何允许放置制表符、空格符或换行符的地方。注释界定符可以跨越程序中的多行，但这并不是必须的。当注释界定符跨过多行时，最好能显式指出其内部的程序行都属于多行注释的一部分。我们所采用的风格是，注释内的每行都以一个星号开头，从而指出整个范围都是多行注释的一部分。

程序中通常同时包含两种形式的注释。注释界定符对通常用于多行解释，而双斜线注释常用于半行和单行附注。

```
#include <iostream>
/*
 * 简单主函数：
 * 读取两个数，求它们的和
 */
int main()
{
```

```
// 提示用户输入两个数
std::cout << "Enter two numbers:" << std::endl;
int v1 = 0, v2 = 0; // 保存我们读入的输入数据的变量
std::cin >> v1 >> v2; // 读取输入数据
std::cout << "The sum of " << v1 << " and " << v2
    << " is " << v1 + v2 << std::endl;
return 0;
}
```



在本书中，我们用楷体来突出显示注释。在实际程序中，注释文本的显示形式是否区别于程序代码文本的显示，依赖于你所使用的程序设计环境是否提供这一特性。

注释界定符不能嵌套

界定符对形式的注释是以`/*`开始，以`*/`结束的。因此，一个注释不能嵌套在另一个注释之内。编译器对这类问题所给出的错误信息可能是难以理解、令人迷惑的。例如，在你的系统中编译下面的程序，就会产生错误：

```
/*
 * 注释对/* */不能嵌套。
 * “不能嵌套”几个字会被认为是源码,
 * 像剩余程序一样处理
 */
int main()
{
    return 0;
}
```

我们通常需要在调试期间注释掉一些代码。由于这些代码可能包含界定符对形式的注释，因此可能导致注释嵌套错误，因此最好的方式是用单行注释方式注释掉代码段的每一行。

```
// /*
// * 单行注释中的任何内容都会被忽略
// * 包括嵌套的注释对也一样会被忽略
// */
```

1.3 节练习

11

练习 1.7：编译一个包含不正确的嵌套注释的程序，观察编译器返回的错误信息。

练习 1.8：指出下列哪些输出语句是合法的（如果有的话）：

```
std::cout << "/*";
std::cout << "*/";
std::cout << /* "*/" */;
std::cout << /* "*/" /* "/*" */;
```

预测编译这些语句会产生什么样的结果，实际编译这些语句来验证你的答案（编写一个小程序，每次将上述一条语句作为其主体），改正每个编译错误。

1.4 控制流

语句一般是顺序执行的：语句块的第一条语句首先执行，然后是第二条语句，依此类推。当然，少数组程序，包括我们解决书店问题的程序，都可以写成只有顺序执行的形式。但程序设计语言提供了多种不同的控制流语句，允许我们写出更为复杂的执行路径。

1.4.1 while 语句

while 语句反复执行一段代码，直至给定条件为假为止。我们可以用 while 语句编写一段程序，求 1 到 10 这 10 个数之和：

```
#include <iostream>
int main()
{
    int sum = 0, val = 1;
    // 只要 val 的值小于等于 10，while 循环就会持续执行
    while (val <= 10) {
        sum += val; // 将 sum + val 赋予 sum
        ++val;       // 将 val 加 1
    }
    std::cout << "Sum of 1 to 10 inclusive is "
              << sum << std::endl;
    return 0;
}
```

我们编译并执行这个程序，它会打印出

Sum of 1 to 10 inclusive is 55

与之前的例子一样，我们首先包含头文件 `iostream`，然后定义 `main`。在 `main` 中我们定义两个 `int` 变量：`sum` 用来保存和；`val` 用来表示从 1 到 10 的每个数。我们将 `sum` 的初值设置为 0，`val` 从 1 开始。

12> 这个程序的新内容是 while 语句。while 语句的形式为

```
while (condition)
    statement
```

while 语句的执行过程是交替地检测 `condition` 条件和执行关联的语句 `statement`，直至 `condition` 为假时停止。所谓条件（`condition`）就是一个产生真或假的结果的表达式。只要 `condition` 为真，`statement` 就会被执行。当执行完 `statement`，会再次检测 `condition`。如果 `condition` 仍为真，`statement` 再次被执行。while 语句持续地交替检测 `condition` 和执行 `statement`，直至 `condition` 为假为止。

在本程序中，while 语句是这样的

```
// 只要 val 的值小于等于 10，while 循环就会持续执行
while (val <= 10) {
    sum += val; // 将 sum + val 赋予 sum
    ++val;       // 将 val 加 1
}
```

条件中使用了小于等于运算符 (`<=`) 来比较 `val` 的当前值和 10。只要 `val` 小于等于 10，条件即为真。如果条件为真，就执行 while 循环体。在本例中，循环体是由两条语句组

成的语句块：

```
{  
    sum += val;      // 将 sum + val 赋予 sum  
    ++val;          // 将 val 加 1  
}
```

所谓语句块（block），就是用花括号包围的零条或多条语句的序列。语句块也是语句的一种，在任何要求使用语句的地方都可以使用语句块。在本例中，语句块的第一条语句使用了复合赋值运算符（`+=`）。此运算符将其右侧的运算对象加到左侧运算对象上，将结果保存到左侧运算对象中。它本质上与一个加法结合一个赋值（assignment）是相同的：

```
sum = sum + val; // 将 sum + val 赋予 sum
```

因此，语句块中第一条语句将 `val` 的值加到当前和 `sum` 上，并将结果保存在 `sum` 中。

下一条语句

```
++val; // 将 val 加 1
```

使用前缀递增运算符（`++`）。递增运算符将运算对象的值增加 1。`++val` 等价于 `val=val+1`。

执行完 `while` 循环体后，循环会再次对条件进行求值。如果 `val` 的值（现在已经增加了）仍然小于等于 10，则 `while` 的循环体会再次执行。循环连续检测条件、执行循环体，直至 `val` 不再小于等于 10 为止。

一旦 `val` 大于 10，程序跳出 `while` 循环，继续执行 `while` 之后的语句。在本例中，继续执行打印输出语句，然后执行 `return` 语句完成 `main` 程序。

1.4.1 节练习

13

练习 1.9：编写程序，使用 `while` 循环将 50 到 100 的整数相加。

练习 1.10：除了`++`运算符将运算对象的值增加 1 之外，还有一个递减运算符（`-`）实现将值减少 1。编写程序，使用递减运算符在循环中按递减顺序打印出 10 到 0 之间的整数。

练习 1.11：编写程序，提示用户输入两个整数，打印出这两个整数所指定的范围内的所有整数。

1.4.2 for 语句

在我们的 `while` 循环例子中，使用了变量 `val` 来控制循环执行次数。我们在循环条件中检测 `val` 的值，在 `while` 循环体中将 `val` 递增。

这种在循环条件中检测变量、在循环体中递增变量的模式使用非常频繁，以至于 C++ 语言专门定义了第二种循环语句——**for 语句**，来简化符合这种模式的语句。可以用 `for` 语句来重写从 1 加到 10 的程序：

```
#include <iostream>  
int main()  
{  
    int sum = 0;  
    // 从 1 加到 10  
    for (int val = 1; val <= 10; ++val)
```

```

        sum += val; // 等价于 sum = sum + val
    std::cout << "Sum of 1 to 10 inclusive is "
        << sum << std::endl;
    return 0;
}

```

与之前一样，我们定义了变量 `sum`，并将其初始化为 0。在此版本中，`val` 的定义是 `for` 语句的一部分：

```

for (int val = 1; val <= 10; ++val)
    sum += val;

```

每个 `for` 语句都包含两部分：循环头和循环体。循环头控制循环体的执行次数，它由三部分组成：一个初始化语句（*init-statement*）、一个循环条件（*condition*）以及一个表达式（*expression*）。在本例中，初始化语句为

```
int val = 1
```

它定义了一个名为 `val` 的 `int` 型对象，并为其赋初值 1。变量 `val` 仅在 `for` 循环内部存在，在循环结束之后是不能使用的。初始化语句只在 `for` 循环入口处执行一次。循环条件

```
val <= 10
```

14 比较 `val` 的值和 10。循环体每次执行前都会先检查循环条件。只要 `val` 小于等于 10，就会执行 `for` 循环体。表达式在 `for` 循环体之后执行。在本例中，表达式

```
++val
```

使用前缀递增运算符将 `val` 的值增加 1。执行完表达式后，`for` 语句重新检测循环条件。如果 `val` 的新值仍然小于等于 10，就再次执行 `for` 循环体。执行完循环体后，再次将 `val` 的值增加 1。循环持续这一过程直至循环条件为假。

在此循环中，`for` 循环体执行加法

```
sum += val; // 等价于 sum = sum + val
```

简要重述一下 `for` 循环的总体执行流程：

1. 创建变量 `val`，将其初始化为 1。
2. 检测 `val` 是否小于等于 10。若检测成功，执行 `for` 循环体。若失败，退出循环，继续执行 `for` 循环体之后的第一条语句。
3. 将 `val` 的值增加 1。
4. 重复第 2 步中的条件检测，只要条件为真就继续执行剩余步骤。

1.4.2 节练习

练习 1.12：下面的 `for` 循环完成了什么功能？`sum` 的终值是多少？

```

int sum = 0;
for (int i = -100; i <= 100; ++i)
    sum += i;

```

练习 1.13：使用 `for` 循环重做 1.4.1 节中的所有练习（第 11 页）。

练习 1.14：对比 `for` 循环和 `while` 循环，两种形式的优缺点各是什么？

练习 1.15：编写程序，包含第 14 页“再探编译”中讨论的常见错误。熟悉编译器生成的错误信息。

1.4.3 读取数量不定的输入数据

在前一节中，我们编写程序实现了 1 到 10 这 10 个整数求和。扩展此程序一个很自然的方向是实现对用户输入的一组数求和。在这种情况下，我们预先不知道要对多少个数求和，这就需要不断读取数据直至没有新的输入为止：

```
#include <iostream>
int main()
{
    int sum = 0, value = 0;
    // 读取数据直到遇到文件尾，计算所有读入的值的和
    while (std::cin >> value)
        sum += value; // 等价于 sum = sum + value
    std::cout << "Sum is: " << sum << std::endl;
    return 0;
}
```

15

如果我们输入

3 4 5 6

则程序会输出

Sum is: 18

`main` 的首行定义了两个名为 `sum` 和 `value` 的 `int` 变量，均初始化为 0。我们使用 `value` 保存用户输入的每个数，数据读取操作是在 `while` 的循环条件中完成的：

```
while (std::cin >> value)
```

`while` 循环条件的求值就是执行表达式

```
std::cin >> value
```

此表达式从标准输入读取下一个数，保存在 `value` 中。输入运算符（参见 1.2 节，第 7 页）返回其左侧运算对象，在本例中是 `std::cin`。因此，此循环条件实际上检测的是 `std::cin`。

当我们使用一个 `istream` 对象作为条件时，其效果是检测流的状态。如果流是有效的，即流未遇到错误，那么检测成功。当遇到文件结束符（end-of-file），或遇到一个无效输入时（例如读入的值不是一个整数），`istream` 对象的状态会变为无效。处于无效状态的 `istream` 对象会使条件变为假。

因此，我们的 `while` 循环会一直执行直至遇到文件结束符（或输入错误）。`while` 循环体使用复合赋值运算符将当前值加到 `sum` 上。一旦条件失败，`while` 循环将会结束。我们将执行下一条语句，打印 `sum` 的值和一个 `endl`。

从键盘输入文件结束符

当从键盘向程序输入数据时，对于如何指出文件结束，不同操作系统有不同的约定。在 Windows 系统中，输入文件结束符的方法是敲 Ctrl+Z（按住 Ctrl 键的同时按 Z 键），然后按 Enter 或 Return 键。在 UNIX 系统中，包括 Mac OS X 系统中，文件结束符输入是用 Ctrl+D。

16 >

再探编译

编译器的一部分工作是寻找程序文本中的错误。编译器没有能力检查一个程序是否按照其作者的意图工作，但可以检查形式（form）上的错误。下面列出了一些最常见的编译器可以检查出的错误。

语法错误 (syntax error): 程序员犯了 C++ 语言文法上的错误。下面程序展示了一些常见的语法错误；每条注释描述了下一行中语句存在的错误：

```
// 错误：main 的参数列表漏掉了
int main (
    // 错误：endl 后使用了冒号而非分号
    std::cout << "Read each file." << std::endl;
    // 错误：字符串字面常量的两侧漏掉了引号
    std::cout << Update master. << std::endl;
    // 错误：漏掉了第二个输出运算符
    std::cout << "Write new master." std::endl;
    // 错误：return 语句漏掉了分号
    return 0
}
```

类型错误 (type error): C++ 中每个数据项都有其类型。例如，10 的类型是 int（或者更通俗地说，“10 是一个 int 型数据”）。单词“hello”，包括两侧的双引号标记，则是一个字符串字面值常量。一个类型错误的例子是，向一个期望参数为 int 的函数传递了一个字符串字面值常量。

声明错误 (declaration error): C++ 程序中的每个名字都要先声明后使用。名字声明失败通常会导致一条错误信息。两种常见的声明错误是：对来自标准库的名字忘记使用 std::、标识符名字拼写错误：

```
#include <iostream>
int main()
{
    int v1 = 0, v2 = 0;
    std::cin >> v >> v2; // 错误：使用了"v"而非"v1"
    // 错误：cout 未定义；应该是 std::cout
    cout << v1 + v2 << std::endl;
    return 0;
}
```

错误信息通常包含一个行号和一条简短描述，描述了编译器认为的我们所犯的错误。按照报告的顺序来逐个修正错误，是一种好习惯。因为一个单个错误常常会具有传递效应，导致编译器在其后报告比实际数量多得多的错误信息。另一个好习惯是在每修

正一个错误后就立即重新编译代码，或者最多是修正了一小部分明显的错误后就重新编译。这就是所谓的“编辑-编译-调试”(edit-compile-debug)周期。

1.4.3 节练习

17

练习 1.16：编写程序，从 cin 读取一组数，输出其和。

1.4.4 if 语句

与大多数语言一样，C++也提供了 **if** 语句来支持条件执行。我们可以用 **if** 语句写一个程序，来统计在输入中每个值连续出现了多少次：

```
#include <iostream>
int main()
{
    // currVal 是我们正在统计的数；我们将读入的新值存入 val
    int currVal = 0, val = 0;
    // 读取第一个数，并确保确实有数据可以处理
    if (std::cin >> currVal) {
        int cnt = 1; // 保存我们正在处理的当前值的个数
        while (std::cin >> val) { // 读取剩余的数
            if (val == currVal) // 如果值相同
                ++cnt; // 将 cnt 加 1
            else { // 否则，打印前一个值的个数
                std::cout << currVal << " occurs "
                << cnt << " times" << std::endl;
                currVal = val; // 记住新值
                cnt = 1; // 重置计数器
            }
        } // while 循环在这里结束
        // 记住打印文件中最后一个值的个数
        std::cout << currVal << " occurs "
        << cnt << " times" << std::endl;
    } // 最外层的 if 语句在这里结束
    return 0;
}
```

如果我们输入如下内容：

42 42 42 42 55 55 62 100 100 100

则输出应该是：

```
42 occurs 5 times
55 occurs 2 times
62 occurs 1 times
100 occurs 3 times
```

有了之前多个程序的基础，你对这个程序中的大部分代码应该比较熟悉了。程序以两个变量 **val** 和 **currVal** 的定义开始：**currVal** 记录我们正在统计出现次数的那个数；**val** 则保存从输入读取的每个数。与之前的程序相比，新的内容就是两个 **if** 语句。第一条 **if** 语句

18 > if (std::cin >> currVal) {
 // ...
} //最外层的 if 语句在这里结束

保证输入不为空。与 while 语句类似，if 也对一个条件进行求值。第一条 if 语句的条件是读取一个数值存入 currVal 中。如果读取成功，则条件为真，我们继续执行条件之后的语句块。该语句块以左花括号开始，以 return 语句之前的右花括号结束。

如果需要统计出现次数的值，我们就定义 cnt，用来统计每个数值连续出现的次数。与上一小节的程序类似，我们用一个 while 循环反复从标准输入读取整数。

while 的循环体是一个语句块，它包含了第二条 if 语句：

```
if (val == currVal)          // 如果值相同
    ++cnt;                  // 将 cnt 加 1
else {                      // 否则，打印前一个值的个数
    std::cout << currVal << " occurs "
        << cnt << " times" << std::endl;
    currVal = val;           // 记住新值
    cnt = 1;                 // 重置计数器
}
```

这条 if 语句中的条件使用了相等运算符（==）来检测 val 是否等于 currVal。如果是，我们执行紧跟在条件之后的语句。这条语句将 cnt 增加 1，表明我们再次看到了 currVal。

如果条件为假，即 val 不等于 currVal，则执行 else 之后的语句。这条语句是一个由一条输出语句和两条赋值语句组成的语句块。输出语句打印我们刚刚统计完的值的出现次数。赋值语句将 cnt 重置为 1，将 currVal 重置为刚刚读入的值 val。



C++用=进行赋值，用==作为相等运算符。两个运算符都可以出现在条件中。
一个常见的错误是想在条件中使用==（相等判断），却误用了=。

1.4.4 节练习

练习 1.17：如果输入的所有值都是相等的，本节的程序会输出什么？如果没有重复值，输出又会是怎样的？

练习 1.18：编译并运行本节的程序，给它输入全都相等的值。再次运行程序，输入没有重复的值。

练习 1.19：修改你为 1.4.1 节练习 1.10（第 11 页）所编写的程序（打印一个范围内的数），使其能处理用户输入的第一个数比第二个数小的情况。

关键概念：C++程序的缩进和格式

C++程序很大程度上是格式自由的，也就是说，我们在哪里放置花括号、缩进、注释以及换行符通常不会影响程序的语义。例如，花括号表示 main 函数体的开始，它可以放在 main 的同一行中；也可以像我们所做的那样，放在下一行的起始位置；还可以放在我们喜欢的其他任何位置。唯一的要求是左花括号必须是 main 的形参列表后第一个非空、非注释的字符。

虽然很大程度上可以按照自己的意愿自由地设定程序的格式，但我们所做的选择会影响程序的可读性。例如，我们可以将整个 main 函数写在很长的单行内，虽然这样是合乎语法的，但会非常难读。

关于 C/C++ 的正确格式的辩论是无休止的。我们的信条是，不存在唯一正确的风格，但保持一致性是非常重要的。例如，大多数程序员都对程序的组成部分设置恰当的缩进，就像我们在之前的例子中对 main 函数中的语句和循环体所做的那样。对于作为函数界定符的花括号，我们习惯将其放在单独一行中。我们还习惯对复合 IO 表达式设置缩进，以使输入输出运算符排列整齐。其他一些缩进约定也都会令越来越复杂的程序更加清晰易读。

我们要牢记一件重要的事情：其他可能的程序格式总是存在的。当你要选择一种格式风格时，思考一下它会对程序的可读性和易理解性有什么影响，而一旦选择了一种风格，就要坚持使用。

1.5 类简介

在解决书店程序之前，我们还需要了解的唯一一个 C++ 特性，就是如何定义一个数据结构（data structure）来表示销售数据。在 C++ 中，我们通过定义一个类（class）来定义自己的数据结构。一个类定义了一个类型，以及与其关联的一组操作。类机制是 C++ 最重要的特性之一。实际上，C++ 最初的一个设计焦点就是能定义使用上像内置类型一样自然的类类型（class type）。

在本节中，我们将介绍一个在编写书店程序中会用到的简单的类。当我们在后续章节中学习了更多关于类型、表达式、语句和函数的知识后，会真正实现这个类。

为了使用类，我们需要了解三件事情：

- 类名是什么？
- 它是在哪里定义的？
- 它支持什么操作？

对于书店程序来说，我们假定类名为 Sales_item，头文件 Sales_item.h 中已经定义了这个类。

如前所见，为了使用标准库设施，我们必须包含相关的头文件。类似的，我们也需要使用头文件来访问为自己的应用程序所定义的类。习惯上，头文件根据其中定义的类的名字来命名。我们通常使用.h 作为头文件的后缀，但也有一些程序员习惯.h、.hpp 或.hxx。标准库头文件通常不带后缀。编译器一般不关心头文件名的形式，但有的 IDE 对此有特定要求。

1.5.1 Sales_item 类

Sales_item 类的作用是表示一本书的总销售额、售出册数和平均售价。我们现在不关心这些数据如何存储、如何计算。为了使用一个类，我们不必关心它是如何实现的，只需知道类对象可以执行什么操作。

每个类实际上都定义了一个新的类型，其类型名就是类名。因此，我们的 Sales_item 类定义了一个名为 Sales_item 的类型。与内置类型一样，我们可以定义类类型的变量。当我们写下如下语句

```
Sales_item item;
```

是想表达 `item` 是一个 `Sales_item` 类型的对象。我们通常将“一个 `Sales_item` 类型的对象”简单说成“一个 `Sales_item` 对象”，或更简单的“一个 `Sales_item`”。

除了可以定义 `Sales_item` 类型的变量之外，我们还可以：

- 调用一个名为 `isbn` 的函数从一个 `Sales_item` 对象中提取 ISBN 书号。
- 用输入运算符 (`>>`) 和输出运算符 (`<<`) 读、写 `Sales_item` 类型的对象。
- 用赋值运算符 (`=`) 将一个 `Sales_item` 对象的值赋予另一个 `Sales_item` 对象。
- 用加法运算符 (`+`) 将两个 `Sales_item` 对象相加。两个对象必须表示同一本书（相同的 ISBN）。加法结果是一个新的 `Sales_item` 对象，其 ISBN 与两个运算对象相同，而其总销售额和售出册数则是两个运算对象的对应值之和。
- 使用复合赋值运算符 (`+=`) 将一个 `Sales_item` 对象加到另一个对象上。

关键概念：类定义了行为

当你读这些程序时，一件要牢记的重要事情是，类 `Sales_item` 的作者定义了类对象可以执行的所有动作。即，`Sales_item` 类定义了创建一个 `Sales_item` 对象时会发生什么事情，以及对 `Sales_item` 对象进行赋值、加法或输入输出运算时会发生什么事情。

一般而言，类的作者决定了类型对象上可以使用的所有操作。当前，我们所知道的可以在 `Sales_item` 对象上执行的全部操作就是本节所列出的那些操作。

21 读写 `Sales_item`

既然已经知道可以对 `Sales_item` 对象执行哪些操作，我们现在就可以编写使用类的程序了。例如，下面的程序从标准输入读入数据，存入一个 `Sales_item` 对象中，然后将 `Sales_item` 的内容写回到标准输出：

```
#include <iostream>
#include "Sales_item.h"
int main()
{
    Sales_item book;
    // 读入 ISBN 号、售出的册数以及销售价格
    std::cin >> book;
    // 写入 ISBN、售出的册数、总销售额和平均价格
    std::cout << book << std::endl;
    return 0;
}
```

如果输入：

0-201-70353-X 4 24.99

则输出为：

0-201-70353-X 4 99.96 24.99

输入表示我们以每本 24.99 美元的价格售出了 4 册书，而输出告诉我们总售出册数为 4，总销售额为 99.96 美元，而每册书的平均销售价格为 24.99 美元。

此程序以两个`#include` 指令开始，其中一个使用了新的形式。包含来自标准库的头

文件时，也应该用尖括号（< >）包围头文件名。对于不属于标准库的头文件，则用双引号（" "）包围。

在 main 中我们定义了一个名为 book 的对象，用来保存从标准输入读取出的数据。下一条语句读取数据存入对象中，第三条语句将对象打印到标准输出上并打印一个 endl。

Sales_item 对象的加法

下面是一个更有意思的例子，将两个 Sales_item 对象相加：

```
#include <iostream>
#include "Sales_item.h"
int main()
{
    Sales_item item1, item2;
    std::cin >> item1 >> item2;           // 读取一对交易记录
    std::cout << item1 + item2 << std::endl; // 打印它们的和
    return 0;
}
```

如果输入如下内容：

```
0-201-78345-X 3 20.00
0-201-78345-X 2 25.00
```

则输出为：

```
0-201-78345-X 5 110 22
```

此程序开始包含了 Sales_item 和 iostream 两个头文件。然后定义了两个 Sales_item 对象来保存销售记录。我们从标准输入读取数据，存入两个对象之中。输出表达式完成加法运算并打印结果。

值得注意的是，此程序看起来与第 5 页的程序非常相似：读取两个输入数据并输出它们的和。造成如此相似的原因是，我们只不过将运算对象从两个整数变为两个 Sales_item 而已，但读取与打印和的运算方式没有发生任何变化。两个程序的另一个不同之处是，“和”的概念是完全不一样的。对于 int，我们计算传统意义上的和——两个数值的算术加法结果。对于 Sales_item 对象，我们用了一个全新的“和”的概念——两个 Sales_item 对象的成员对应相加的结果。

使用文件重定向

当你测试程序时，反复从键盘敲入这些销售记录作为程序的输入，是非常乏味的。大多数操作系统支持文件重定向，这种机制允许我们将标准输入和标准输出与命名文件关联起来：

```
$ addItems <infile >outfile
```

假定 \$ 是操作系统提示符，我们的加法程序已经编译为名为 addItems.exe 的可执行文件（在 UNIX 中是 addItems），则上述命令会从一个名为 infile 的文件读取销售记录，并将输出结果写入到一个名为 outfile 的文件中，两个文件都位于当前目录中。

1.5.1 节练习

练习 1.20: 在网站 <http://www.informit.com/title/0321714113> 上, 第 1 章的代码目录中包含了头文件 Sales_item.h。将它拷贝到你自己的工作目录中。用它编写一个程序, 读取一组书籍销售记录, 将每条记录打印到标准输出上。

练习 1.21: 编写程序, 读取两个 ISBN 相同的 Sales_item 对象, 输出它们的和。

练习 1.22: 编写程序, 读取多个具有相同 ISBN 的销售记录, 输出所有记录的和。

23 1.5.2 初识成员函数

将两个 Sales_item 对象相加的程序首先应该检查两个对象是否具有相同的 ISBN。方法如下:

```
#include <iostream>
#include "Sales_item.h"
int main()
{
    Sales_item item1, item2;
    std::cin >> item1 >> item2;
    // 首先检查 item1 和 item2 是否表示相同的书
    if (item1.isbn() == item2.isbn()) {
        std::cout << item1 + item2 << std::endl;
        return 0;      // 表示成功
    } else {
        std::cerr << "Data must refer to same ISBN"
            << std::endl;
        return -1;      // 表示失败
    }
}
```

此程序与上一版本的差别是 if 语句及其 else 分支。即使不了解这个 if 语句的检测条件, 我们也很容易理解这个程序在干什么。如果条件成立, 如上一版本一样, 程序打印计算结果, 并返回 0, 表明成功。如果条件失败, 我们执行跟在 else 之后的语句块, 打印一条错误信息, 并返回一个错误标识。

什么是成员函数?

这个 if 语句的检测条件

```
item1.isbn() == item2.isbn()
```

调用名为 isbn 的成员函数 (member function)。成员函数是定义为类的一部分的函数, 有时也被称为方法 (method)。

我们通常以一个类对象的名义来调用成员函数。例如, 上面相等表达式左侧运算对象的第一部分

```
item1.isbn()
```

使用点运算符 (.) 来表达我们需要“名为 item1 的对象的 isbn 成员”。点运算符只能用于类类型的对象。其左侧运算对象必须是一个类类型的对象, 右侧运算对象必须是该类型的一个成员名, 运算结果为右侧运算对象指定的成员。

当用点运算符访问一个成员函数时，通常我们是想（效果也确实是）调用该函数。我们使用调用运算符（`()`）来调用一个函数。调用运算符是一对圆括号，里面放置实参（argument）列表（可能为空）。成员函数 `isbn` 并不接受参数。因此

```
item1.isbn()
```

调用名为 `item1` 的对象的成员函数 `isbn`，此函数返回 `item1` 中保存的 ISBN 书号。

< 24

在这个 `if` 条件中，相等运算符的右侧运算对象也是这样执行的——它返回保存在 `item2` 中的 ISBN 书号。如果 ISBN 相同，条件为真，否则为假。

1.5.2 节练习

练习 1.23：编写程序，读取多条销售记录，并统计每个 ISBN（每本书）有几条销售记录。

练习 1.24：输入表示多个 ISBN 的多条销售记录来测试上一个程序，每个 ISBN 的记录应该聚在一起。

1.6 书店程序

现在我们已经准备好完成书店程序了。我们需要从一个文件中读取销售记录，生成每本书的销售报告，显示售出册数、总销售额和平均售价。我们假定每个 ISBN 书号的所有销售记录在文件中是聚在一起保存的。

我们的程序会将每个 ISBN 的所有数据合并起来，存入名为 `total` 的变量中。我们使用另一个名为 `trans` 的变量保存读取的每条销售记录。如果 `trans` 和 `total` 指向相同的 ISBN，我们会更新 `total` 的值。否则，我们会打印 `total` 的值，并将其重置为刚刚读取的数据 (`trans`)：

```
#include <iostream>
#include "Sales_item.h"
int main()
{
    Sales_item total; // 保存下一条交易记录的变量
    // 读入第一条交易记录，并确保有数据可以处理
    if (std::cin >> total) {
        Sales_item trans; // 保存和的变量
        // 读入并处理剩余交易记录
        while (std::cin >> trans) {
            // 如果我们仍在处理相同的书
            if (total.isbn() == trans.isbn())
                total += trans; // 更新总销售额
            else {
                // 打印前一本书的结果
                std::cout << total << std::endl;
                total = trans; // total 现在表示下一本书的销售额
            }
        }
        std::cout << total << std::endl; // 打印最后一本书的结果
    } else {
}
```

```
25 //没有输入！警告读者
    std::cerr << "No data?!" << std::endl;
    return -1; // 表示失败
}
return 0;
}
```

这是到目前为止我们看到的最复杂的程序了，但它所使用的都是我们已经见过的语言特性。

与往常一样，首先包含要使用的头文件：来自标准库的 `iostream` 和自己定义的 `Sales_item.h`。在 `main` 中，我们定义了一个名为 `total` 的变量，用来保存一个给定的 ISBN 的数据之和。我们首先读取第一条销售记录，存入 `total` 中，并检测这次读取操作是否成功。如果读取失败，则意味着没有任何销售记录，于是直接跳到最外层的 `else` 分支，打印一条警告信息，告诉用户没有输入。

假定已经成功读取了一条销售记录，我们继续执行最外层 `if` 之后的语句块。这个语句块首先定义一个名为 `trans` 的对象，它保存读取的销售记录。接下来的 `while` 语句将读取剩下的所有销售记录。与我们之前的程序一样，`while` 条件是一个从标准输入读取值的操作。在本例中，我们读取一个 `Sales_item` 对象，存入 `trans` 中。只要读取成功，就执行 `while` 循环体。

`while` 的循环体是一个单个的 `if` 语句，它检查 ISBN 是否相等。如果相等，使用复合赋值运算符将 `trans` 加到 `total` 中。如果 ISBN 不等，我们打印保存在 `total` 中的值，并将其重置为 `trans` 的值。在执行完 `if` 语句后，返回到 `while` 的循环条件，读取下一条销售记录，如此反复，直至所有销售记录都处理完。

当 `while` 语句终止时，`total` 保存着文件中最后一个 ISBN 的数据。我们在语句块的最后一条语句中打印这最后一个 ISBN 的 `total` 值，至此最外层 `if` 语句就结束了。

1.6 节练习

练习 1.25：借助网站上的 `Sales_item.h` 头文件，编译并运行本节给出的书店程序。

小结

< 26

本章介绍了足够多的 C++ 语言的知识，以使你能够编译、运行简单的 C++ 程序。我们看到了如何定义一个 `main` 函数，它是操作系统执行你的程序的调用入口。我们还看到了如何定义变量，如何进行输入输出，以及如何编写 `if`、`for` 和 `while` 语句。本章最后介绍了 C++ 中最基本的特性——类。在本章中，我们看到了，对于其他人定义的一个类，我们应该如何创建、使用其对象。在后续章节中，我们将介绍如何定义自己的类。

术语表

参数 (实参, argument) 向函数传递的值。

赋值 (assignment) 抹去一个对象的当前值，用一个新值取代之。

程序块 (block) 零条或多条语句的序列，用花括号包围。

缓冲区 (buffer) 一个存储区域，用于保存数据。`IO` 设施通常将输入（或输出）数据保存在一个缓冲区中，读写缓冲区的动作与程序中的动作是无关的。我们可以显式地刷新输出缓冲，以便强制将缓冲区中的数据写入输出设备。默认情况下，读 `cin` 会刷新 `cout`；程序非正常终止时也会刷新 `cout`。

内置类型 (built-in type) 由语言定义的类型，如 `int`。

`cerr` 一个 `ostream` 对象，关联到标准错误，通常写入到与标准输出相同的设备。默认情况下，写到 `cerr` 的数据是不缓冲的。`cerr` 通常用于输出错误信息或其他不属于程序正常逻辑的输出内容。

字符串字面值常量 (character string literal) 术语 `string literal` 的另一种叫法。

`cin` 一个 `istream` 对象，用来从标准输入读取数据。

类 (class) 一种用于定义自己的数据结构及其相关操作的机制。类是 C++ 中最基本的特性之一。标准库类型中，如 `istream` 和 `ostream` 都是类。

类类型 (class type) 类定义的类型。类名即为类型名。

`clog` 一个 `ostream` 对象，关联到标准错误。默认情况下，写到 `clog` 的数据是被缓冲的。`clog` 通常用于报告程序的执行信息，存入一个日志文件中。

注释 (comment) 被编译器忽略的程序文本。C++ 有两种类型的注释：单行注释和界定符对注释。单行注释以 // 开始，从 // 到行尾的所有内容都是注释。界定符对注释以 /* 开始，其后的所有内容都是注释，直至遇到 */ 为止。

条件 (condition) 求值结果为真或假的表达式。通常用值 0 表示假，用非零值表示真。

`cout` 一个 `ostream` 对象，用于将数据写入标准输出。通常用于程序的正常输出内容。

花括号 (curly brace) 花括号用于划定程序块边界。左花括号 {} 为程序块开始，右花括号 } 为结束。

数据结构 (data structure) 数据及其上所允许的操作的一种逻辑组合。

编辑-编译-调试 (edit-compile-debug) 使程序能正确执行的开发过程。

文件结束符 (end-of-file) 系统特定的标识，指出文件中无更多数据了。

表达式 (expression) 最小的计算单元。一个表达式包含一个或多个运算对象，通常还包含一个或多个运算符。表达式求值会产生一个结果。例如，假设 `i` 和 `j` 是 `int` 对象，则 `i+j` 是一个表达式，它产生两个

< 27

`int` 值的和。

for 语句 (for statement) 迭代语句，提供重复执行能力。通常用来将一个计算反复执行指定次数。

函数 (function) 具名的计算单元。

函数体 (function body) 语句块，定义了函数所执行的动作。

函数名 (function name) 函数为人所知的名字，也用来进行函数调用。

头文件 (header) 使类或其他名字的定义可被多个程序使用的一种机制。程序通过 `#include` 指令使用头文件。

if 语句 (if statement) 根据一个特定条件的值进行条件执行的语句。如果条件为真，执行 `if` 语句体。否则，执行 `else` 语句体（如果存在的话）。

初始化 (initialize) 在一个对象创建的时候赋予它一个值。

iostream 头文件，提供了面向流的输入输出的标准库类型。

istream 提供了面向流的输入的库类型。

库类型 (library type) 标准库定义的类型，28如 `istream`。

main 操作系统执行一个 C++ 程序时所调用的函数。每个程序必须有且只有一个命名为 `main` 的函数。

操纵符 (manipulator) 对象，如 `std::endl`，在读写流的时候用来“操纵”流本身。

成员函数 (member function) 类定义的操作。通常通过调用成员函数来操作特定对象。

方法 (method) 成员函数的同义术语。

命名空间 (namespace) 将库定义的名字放在一个单一位置的机制。命名空间可以帮助避免不经意的名字冲突。C++ 标准库定义的名字在命名空间 `std` 中。

ostream 标准库类型，提供面向流的输出。

形参列表 (parameter list) 函数定义的一部分，指出调用函数时可以使用什么样的实参，可能为空列表。

返回类型 (return type) 函数返回值的类型。

源文件 (source file) 包含 C++ 程序的文件。

标准错误 (standard error) 输出流，用于报告错误。标准输出和标准错误通常关联到程序执行所在的窗口。

标准输入 (standard input) 输入流，通常与程序执行所在窗口相关联。

标准库 (standard library) 一个类型和函数的集合，每个 C++ 编译器都必须支持。

标准库提供了支持 I/O 操作的类型。C++ 程序员倾向于用“库”指代整个标准库，还倾向于用库类型表示标准库的特定部分，例如用“`iostream` 库”表示标准库中定义 I/O 类的部分。

标准输出 (standard output) 输出流，通常与程序执行所在窗口相关联。

语句 (statement) 程序的一部分，指定了当程序执行时进行什么动作。一个表达式接一个分号就是一条语句；其他类型的语句包括语句块、`if` 语句、`for` 语句和 `while` 语句，所有这些语句内都包含其他语句。

std 标准库所使用的命名空间。`std::cout` 表示我们要使用定义在命名空间 `std` 中的名字 `cout`。

字符串常量 (string literal) 零或多个字符组成的序列，用双引号包围 ("a string literal")。

未初始化的变量 (uninitialized variable) 未赋予初值的变量。类类型的变量如果未指定初值，则按类定义指定的方式进行初始化。定义在函数内部的内置类型变量默认是不初始化的，除非有显式的初始化语句。试图使用一个未初始化变量的值是错误的。未初始化变量是 bug 的常见成因。

变量 (variable) 具名对象。

while 语句 (while statement) 迭代语句，提供重复执行直至一个特定条件为假的机制。循环体会执行零次或多次，依赖于循环条件求值结果。

()运算符 (() operator) 调用运算符。跟随着在函数名之后的一对括号 “()”，起到调用函数的效果。传递给函数的实参放置在括号内。

++运算符 (++ operator) 递增运算符。将运算对象的值加 1， $++i$ 等价于 $i=i+1$ 。

+=运算符 (+= operator) 复合赋值运算符，将右侧运算对象加到左侧运算对象上； $a+=b$ 等价于 $a=a+b$ 。

.运算符 (. operator) 点运算符。左侧运算对象必须是一个类类型对象，右侧运算对象必须是此对象的一个成员的名字。运算结果即为该对象的这个成员。

::运算符 (:: operator) 作用域运算符。其用处之一是访问命名空间中的名字。例如，`std::cout` 表示命名空间 `std` 中的名字 `cout`。

=运算符 (= operator) 将右侧运算对象的值赋予左侧运算对象所表示的对象。

--运算符 (-- operator) 递减运算符。将运算对象的值减 1， $--i$ 等价于 $i=i-1$ 。

<<运算符 (<< operator) 输出运算符。将

右侧运算对象的值写到左侧运算对象表示的输出流：`cout << "hi"` 表示将 `hi` 写到标准输出。输出运算符可以连接：`cout << "hi" << "bye"` 表示将输出 `hibye`。

>>运算符 (>> operator) 输入运算符。从左侧运算对象所指定的输入流读取数据，存入右侧运算对象中：`cin >> i` 表示从标准输入读取下一个值，存入 `i` 中。输入运算符可以连接：`cin >> i >> j` 表示先读取一个值存入 `i`，再读取一个值存入 `j`。

#include 头文件包含指令，使头文件中代码可被程序使用。

==运算符 (== operator) 相等运算符。检测左侧运算对象是否等于右侧运算对象。

!=运算符 (!= operator) 不等运算符。检测左侧运算对象是否不等于右侧运算对象。

<=运算符 (<= operator) 小于等于运算符。检测左侧运算对象是否小于等于右侧运算对象。

<运算符 (< operator) 小于运算符。检测左侧运算对象是否小于右侧运算对象。

>=运算符 (>= operator) 大于等于运算符。检测左侧运算对象是否大于等于右侧运算对象。

>运算符 (> operator) 大于运算符。检测左侧运算对象是否大于右侧运算对象。

