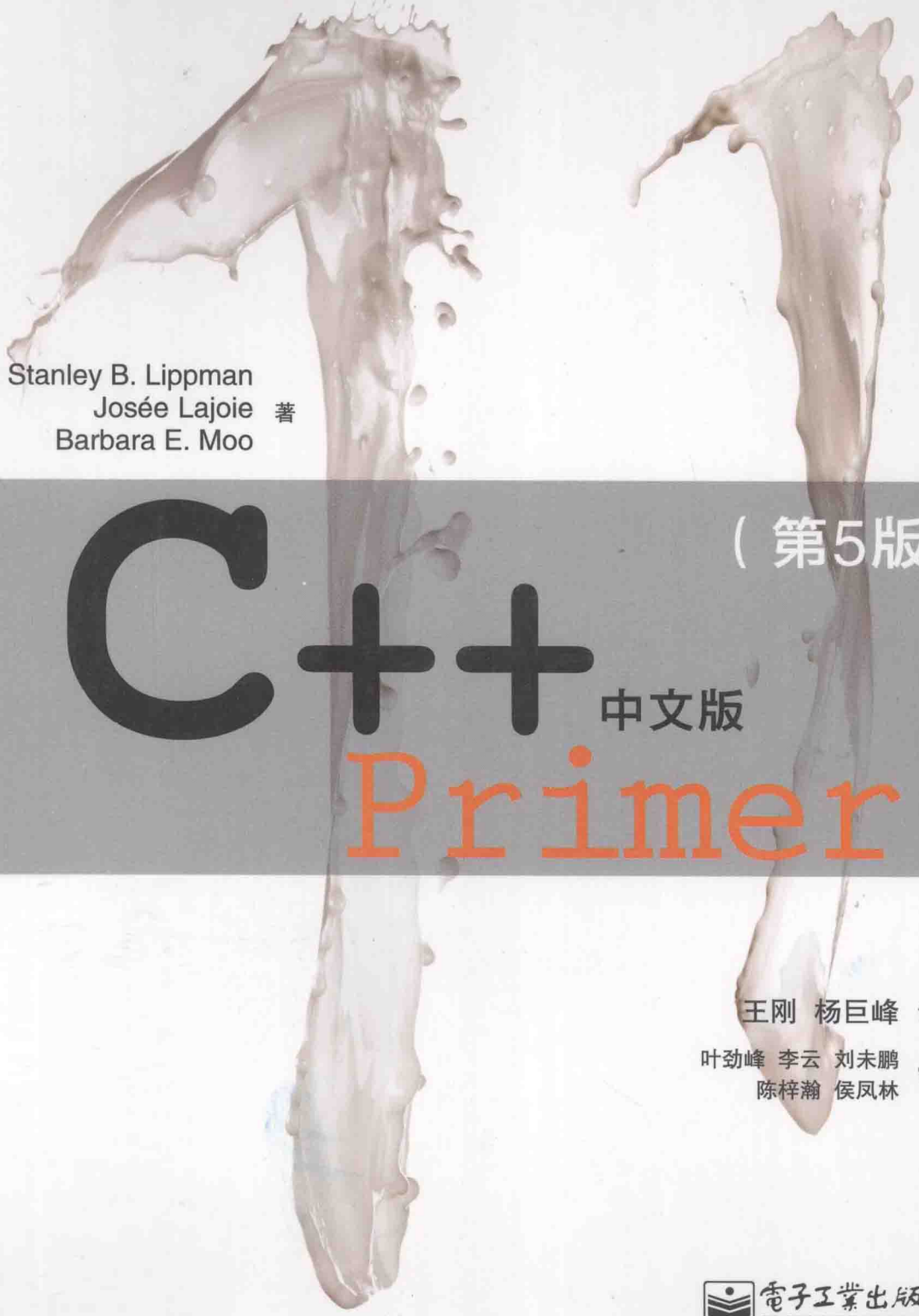
Broadview
www.broadview.com.cn

PEARSON

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
Barbara E. Moo
著

C++ Primer

中文版

（第5版）

王刚 杨巨峰 译

叶劲峰 李云 刘未鹏
陈梓瀚 侯凤林 审校

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

Broadview®
www.broadview.com.cn
博文视点·IT出版旗舰品牌
技术凝聚实力·专业创新出版



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。

◎得益于新式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

PEARSON
程语言/C++
978-7-121-15535-2
9 787121 155352 >
定价：128.00元

Stanley B. Lippman
Josée Lajoie 著
Barbara E. Moo

C++ Primer 中文版（第5版）

王刚 杨巨峰 译

叶劲峰 李云 刘未鹏
陈梓瀚 侯凤林 审校

電子工業出版社
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出版权由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 舫
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zlts@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++成功的根本原因，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，每当你需要走下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，与其说是教你怎么用 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 constexpr 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 Sales_data 类……228
7.1.2 定义改进的 Sales_data 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*this 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 bitset 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 noexcept 异常指示符 ····· 690
18.1.4 noexcept 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 enum ····· 736
19.3 说明类型用于保存 enum 对象 ····· 738
19.3 enum 的前置声明 ····· 738
19.4.3 标准库 mem_fn 类模板 ····· 746
19.6 类类型的 union 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）：

3
```
{
    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
7 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值常量，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。当注释界定符跨越多行时，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。

10 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>                                          15
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;
}
```

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 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。

19 **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。 20

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：

22

```
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。27一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

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，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 IO 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 IO 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。

第 I 部分
C++基础

内容

第 2 章　变量和基本类型 29
第 3 章　字符串、向量和数组 73
第 4 章　表达式 119
第 5 章　语句 153
第 6 章　函数 181
第 7 章　类 227

任何常用的编程语言都具备一组公共的语法特征，不同语言仅在特征的细节上有所区别。要想学习并掌握一种编程语言，理解其语法特征的实现细节是第一步。最基本的特征包括：

- 整型、字符型等内置类型
- 变量，用来为对象命名
- 表达式和语句，用于操纵上述数据类型的具体值
- `if` 或 `while` 等控制结构，这些结构允许我们有选择地执行一些语句或者重复地执行一些语句
- 函数，用于定义可供随时调用的计算单元

大多数编程语言通过两种方式来进一步补充其基本特征：一是赋予程序员自定义数据类型的权利，从而实现对语言的扩展；二是将一些有用的功能封装成库函数提供给程序员。

30 与大多数编程语言一样，C++的对象类型决定了能对该对象进行的操作，一条表达式是否合法依赖于其中参与运算的对象的类型。一些语言，如 Smalltalk 和 Python 等，在程序运行时检查数据类型；与之相反，C++是一种静态数据类型语言，它的类型检查发生在编译时。因此，编译器必须知道程序中每一个变量对应的数据类型。

C++提供了一组内置数据类型、相应的运算符以及为数不多的几种程序流控制语句，这些元素共同构成了 C++语言的基本形态。以这些元素为基础，我们可以编写出规模庞大、结构复杂、用于解决实际问题的软件系统。仅就 C++的基本形态来说，它是一种简单的编程语言，其强大的能力显示于它对程序员自定义数据结构的支持。这种支持作用巨大，显而易见的一个事实是，C++语言的缔造者无须洞悉所有程序员的要求，而程序员恰好可以通过自主定义新的数据结构来使语言满足他们各自的需求。

C++中最重要的语法特征应该就是类了，通过它，程序员可以定义自己的数据类型。为了与 C++的内置类型区别开来，它们通常被称为“类类型（class type）”。在一些编程语言中，程序员自定义的新类型仅能包含数据成员；另外一些语言，比如 C++，则允许新类型中既包含数据成员，也包含函数成员。C++语言主要的一个设计目标就是让程序员自定义的数据类型像内置类型一样好用。基于此，标准 C++库实现了丰富的类和函数。

本书第Ⅰ部分的主题是学习 C++语言的基础知识，这也是掌握 C++语言的第一步。第 2 章详述内置类型，并初步介绍了自定义数据类型的方法。第 3 章介绍了两种最基本的数据类型：字符串和向量。C++和许多编程语言所共有的一种底层数据结构——数组也在本章有所提及。接下来，第 4～6 章依次介绍了表达式、语句和函数。作为第Ⅰ部分的最后一章，第 7 章描述了如何构建我们自己的类，完成这一任务需要综合运用之前各章所介绍的知识。

第 2 章
变量和基本类型

内容

2.1 基本内置类型 30
2.2 变量 38
2.3 复合类型 45
2.4 const 限定符 53
2.5 处理类型 60
2.6 自定义数据结构 64
小结 69
术语表 69

数据类型是程序的基础：它告诉我们数据的意义以及我们能在数据上执行的操作。

C++语言支持广泛的数据类型。它定义了几种基本内置类型（如字符、整型、浮点数等），同时也为程序员提供了自定义数据类型的机制。基于此，C++标准库定义了一些更加复杂的数据类型，比如可变长字符串和向量等。本章将主要讲述内置类型，并带领大家初步了解 C++语言是如何支持更复杂数据类型的。

32 数据类型决定了程序中数据和操作的意义。如下所示的语句是一个简单示例：

```
i = i + j;
```

其含义依赖于 i 和 j 的数据类型。如果 i 和 j 都是整型数，那么这条语句执行的就是最普通的加法运算。然而，如果 i 和 j 是 Sales_item 类型的数据（参见 1.5.1 节，第 17 页），则上述语句把这两个对象的成分相加。

2.1 基本内置类型

C++定义了一套包括**算术类型**（arithmetic type）和**空类型**（void）在内的基本数据类型。其中算术类型包含了字符、整型数、布尔值和浮点数。空类型不对应具体的值，仅用于一些特殊的场合，例如最常见的是，当函数不返回任何值时使用空类型作为返回类型。

2.1.1 算术类型

算术类型分为两类：**整型**（integral type，包括字符和布尔类型在内）和浮点型。

算术类型的尺寸（也就是该类型数据所占的比特数）在不同机器上有所差别。表 2.1 列出了 C++标准规定的尺寸的最小值，同时允许编译器赋予这些类型更大的尺寸。某一类型所占的比特数不同，它所能表示的数据范围也不一样。

表 2.1：C++：算术类型

类型	含义	最小尺寸
bool	布尔类型	未定义
char	字符	8 位
wchar_t	宽字符	16 位
char16_t	Unicode 字符	16 位
char32_t	Unicode 字符	32 位
short	短整型	16 位
int	整型	16 位
long	长整型	32 位
long long	长整型	64 位
float	单精度浮点数	6 位有效数字
double	双精度浮点数	10 位有效数字
long double	扩展精度浮点数	10 位有效数字

布尔类型（bool）的取值是真（true）或者假（false）。

C++提供了几种字符类型，其中多数支持国际化。基本的字符类型是 char，一个 char 的空间应确保可以存放机器基本字符集中任意字符对应的数字值。也就是说，一个 char 的大小和一个机器字节一样。

33 其他字符类型用于扩展字符集，如 wchar_t、char16_t、char32_t。wchar_t 类型用于确保可以存放机器最大扩展字符集中的任意一个字符，类型 char16_t 和 char32_t 则为 Unicode 字符集服务（Unicode 是用于表示所有自然语言中字符的标准）。

除字符和布尔类型之外，其他整型用于表示（可能）不同尺寸的整数。C++语言规定一个 int 至少和一个 short 一样大，一个 long 至少和一个 int 一样大，一个 long long

至少和一个 long 一样大。其中，数据类型 long long 是在 C++11 中新定义的。 C++ 11

内置类型的机器实现

计算机以比特序列存储数据，每个比特非 0 即 1，例如：

```
00011011011100010110010000111011 ...
```

大多数计算机以 2 的整数次幂个比特作为块来处理内存，可寻址的最小内存块称为“字节（byte）”，存储的基本单元称为“字（word）”，它通常由几个字节组成。在 C++语言中，一个字节要至少能容纳机器基本字符集中的字符。大多数机器的字节由 8 比特构成，字则由 32 或 64 比特构成，也就是 4 或 8 字节。

大多数计算机将内存中的每个字节与一个数字（被称为“地址（address）”）关联起来，在一个字节为 8 比特、字为 32 比特的机器上，我们可能看到一个字的内存区域如下所示：

736424	0	0	1	1	1	0	1	1
736425	0	0	0	1	1	0	1	1
736426	0	1	1	1	0	0	0	1
736427	0	1	1	0	0	1	0	0

其中，左侧是字节的地址，右侧是字节中 8 比特的具体内容。

我们能够使用某个地址来表示从这个地址开始的大小不同的比特串，例如，我们可能会说地址 736424 的那个字或者地址 736427 的那个字节。为了赋予内存中某个地址明确的含义，必须首先知道存储在该地址的数据的类型。类型决定了数据所占的比特数以及该如何解释这些比特的内容。

如果位置 736424 处的对象类型是 float，并且该机器中 float 以 32 比特存储，那么我们就能知道这个对象的内容占满了整个字。这个 float 数的实际值依赖于该机器是如何存储浮点数的。或者如果位置 736424 处的对象类型是 unsigned char，并且该机器使用 ISO-Latin-1 字符集，则该位置处的字节表示一个分号。

浮点型可表示单精度、双精度和扩展精度值。C++标准指定了一个浮点数有效位数的最小值，然而大多数编译器都实现了更高的精度。通常，float 以 1 个字（32 比特）来表示，double 以 2 个字（64 比特）来表示，long double 以 3 或 4 个字（96 或 128 比特）来表示。一般来说，类型 float 和 double 分别有 7 和 16 个有效位；类型 long double 则常常被用于有特殊浮点需求的硬件，它的具体实现不同，精度也各不相同。 34

带符号类型和无符号类型

除去布尔型和扩展的字符型之外，其他整型可以划分为**带符号的**（signed）和**无符号的**（unsigned）两种。带符号类型可以表示正数、负数或 0，无符号类型则仅能表示大于等于 0 的值。

类型 int、short、long 和 long long 都是带符号的，通过在这些类型名前添加 unsigned 就可以得到无符号类型，例如 unsigned long。类型 unsigned int 可以缩写为 unsigned。

与其他整型不同，字符型被分为了三种：char、signed char 和 unsigned char。

特别需要注意的是：类型 char 和类型 signed char 并不一样。尽管字符型有三种，但是字符的表现形式却只有两种：带符号的和无符号的。类型 char 实际上会表现为上述两种形式中的一种，具体是哪种由编译器决定。

无符号类型中所有比特都用来存储值，例如，8 比特的 unsigned char 可以表示 0 至 255 区间内的值。

C++标准并没有规定带符号类型应如何表示，但是约定了在表示范围内正值和负值的量应该平衡。因此，8 比特的 signed char 理论上应该可以表示-127 至 127 区间内的值，大多数现代计算机将实际的表示范围定为-128 至 127。

建议：如何选择类型

和 C 语言一样，C++的设计准则之一也是尽可能地接近硬件。C++的算术类型必须满足各种硬件特质，所以它们常常显得繁杂而令人不知所措。事实上，大多数程序员能够（也应该）对数据类型的使用做出限定从而简化选择的过程。以下是选择类型的一些经验准则：

- 当明确知晓数值不可能为负时，选用无符号类型。
- 使用 int 执行整数运算。在实际应用中，short 常常显得太小而 long 一般和 int 有一样的尺寸。如果你的数值超过了 int 的表示范围，选用 long long。
- 在算术表达式中不要使用 char 或 bool，只有在存放字符或布尔值时才使用它们。因为类型 char 在一些机器上是有符号的，而在另一些机器上又是无符号的，所以如果使用 char 进行运算特别容易出问题。如果你需要使用一个不大的整数，那么明确指定它的类型是 signed char 或者 unsigned char。
- 执行浮点数运算选用 double，这是因为 float 通常精度不够而且双精度浮点数和单精度浮点数的计算代价相差无几。事实上，对于某些机器来说，双精度运算甚至比单精度还快。long double 提供的精度在一般情况下是没有必要的，况且它带来的运行时消耗也不容忽视。

35

2.1.1 节练习

练习 2.1：类型 int、long、long long 和 short 的区别是什么？无符号类型和带符号类型的区别是什么？float 和 double 的区别是什么？

练习 2.2：计算按揭贷款时，对于利率、本金和付款分别应选择何种数据类型？说明你的理由。



2.1.2 类型转换

对象的类型定义了对象能包含的数据和能参与的运算，其中一种运算被大多数类型支持，就是将对象从一种给定的类型**转换**（convert）为另一种相关类型。

当在程序的某处我们使用了一种类型而其实对象应该取另一种类型时，程序会自动进行类型转换，在 4.11 节（第 141 页）中我们将对类型转换做更详细的介绍。此处，有必要说明当给某种类型的对象强行赋了另一种类型的值时，到底会发生什么。

当我们像下面这样把一种算术类型的值赋给另外一种类型时：

```
bool b = 42;                    // b 为真
```

```
int i = b;                  // i 的值为 1
i = 3.14;                   // i 的值为 3
double pi = i;              // pi 的值为 3.0
unsigned char c = -1;       // 假设 char 占 8 比特，c 的值为 255
signed char c2 = 256;       // 假设 char 占 8 比特，c2 的值是未定义的
```

类型所能表示的值的范围决定了转换的过程：

- 当我们把一个非布尔类型的算术值赋给布尔类型时，初始值为 0 则结果为 false，否则结果为 true。
- 当我们把一个布尔值赋给非布尔类型时，初始值为 false 则结果为 0，初始值为 true 则结果为 1。
- 当我们把一个浮点数赋给整数类型时，进行了近似处理。结果值将仅保留浮点数中小数点之前的部分。
- 当我们把一个整数值赋给浮点类型时，小数部分记为 0。如果该整数所占的空间超过了浮点类型的容量，精度可能有损失。
- 当我们赋给无符号类型一个超出它表示范围的值时，结果是初始值对无符号类型表示数值总数取模后的余数。例如，8 比特大小的 unsigned char 可以表示 0 至 255 区间内的值，如果我们赋了一个区间以外的值，则实际的结果是该值对 256 取模后所得的余数。因此，把-1 赋给 8 比特大小的 unsigned char 所得的结果是 255。
- 当我们赋给带符号类型一个超出它表示范围的值时，结果是**未定义的**（undefined）。此时，程序可能继续工作、可能崩溃，也可能生成垃圾数据。

建议：避免无法预知和依赖于实现环境的行为 36

无法预知的行为源于编译器无须（有时是不能）检测的错误。即使代码编译通过了，如果程序执行了一条未定义的表达式，仍有可能产生错误。

不幸的是，在某些情况和/或某些编译器下，含有无法预知行为的程序也能正确执行。但是我们却无法保证同样一个程序在别的编译器下能正常工作，甚至已经编译通过的代码再次执行也可能会出错。此外，也不能认为这样的程序对一组输入有效，对另一组输入就一定有效。

程序也应该尽量避免依赖于实现环境的行为。如果我们把 int 的尺寸看成是一个确定不变的已知值，那么这样的程序就称作**不可移植的**（nonportable）。当程序移植到别的机器上后，依赖于实现环境的程序就可能发生错误。要从过去的代码中定位这类错误可不是一件轻松愉快的工作。

当在程序的某处使用了一种算术类型的值而其实所需的是另一种类型的值时，编译器同样会执行上述的类型转换。例如，如果我们使用了一个非布尔值作为条件（参见 1.4.1 节，第 10 页），那么它会被自动地转换成布尔值，这一做法和把非布尔值赋给布尔变量时的操作完全一样：

```
int i = 42;
if (i)                      // if 条件的值将为 true
  i = 0;
```

如果 i 的值为 0，则条件的值为 false；i 的所有其他取值（非 0）都将使条件为 true。

以此类推，如果我们把一个布尔值用在算术表达式里，则它的取值非 0 即 1，所以一般不宜在算术表达式里使用布尔值。

含有无符号类型的表达式

尽管我们不会故意给无符号对象赋一个负值，却可能（特别容易）写出这么做的代码。例如，当一个算术表达式中既有无符号数又有 int 值时，那个 int 值就会转换成无符号数。把 int 转换成无符号数的过程和把 int 直接赋给无符号变量一样：

```
unsigned u = 10;
int i = -42;
std::cout << i + i << std::endl; // 输出-84
std::cout << u + i << std::endl; // 如果 int 占 32 位，输出 4294967264
```

在第一个输出表达式里，两个（负）整数相加并得到了期望的结果。在第二个输出表达式里，相加前首先把整数-42 转换成无符号数。把负数转换成无符号数类似于直接给无符号数赋一个负值，结果等于这个负数加上无符号数的模。

当从无符号数中减去一个值时，不管这个值是不是无符号数，我们都必须确保结果不能是一个负值：

37

```
unsigned u1 = 42, u2 = 10;
std::cout << u1 - u2 << std::endl; // 正确：输出 32
std::cout << u2 - u1 << std::endl; // 正确：不过，结果是取模后的值
```

无符号数不会小于 0 这一事实同样关系到循环的写法。例如，在 1.4.1 节的练习（第 11 页）中需要写一个循环，通过控制变量递减的方式把从 10 到 0 的数字降序输出。这个循环可能类似于下面的形式：

```
for (int i = 10; i >= 0; --i)
    std::cout << i << std::endl;
```

可能你会觉得反正也不打算输出负数，可以用无符号数来重写这个循环。然而，这个不经意的改变却意味着死循环：

```
// 错误：变量 u 永远也不会小于 0，循环条件一直成立
for (unsigned u = 10; u >= 0; --u)
    std::cout << u << std::endl;
```

来看看当 u 等于 0 时发生了什么，这次迭代输出 0，然后继续执行 for 语句里的表达式。表达式--u 从 u 当中减去 1，得到的结果-1 并不满足无符号数的要求，此时像所有表示范围之外的其他数字一样，-1 被自动地转换成一个合法的无符号数。假设 int 类型占 32 位，则当 u 等于 0 时，--u 的结果将会是 4294967295。

一种解决的办法是，用 while 语句来代替 for 语句，因为前者让我们能够在输出变量之前（而非之后）先减去 1：

```
unsigned u = 11; // 确定要输出的最大数，从比它大 1 的数开始
while (u > 0){
    --u;          // 先减 1，这样最后一次迭代时就会输出 0
    std::cout << u << std::endl;
}
```

改写后的循环先执行对循环控制变量减 1 的操作，这样最后一次迭代时，进入循环的 u 值为 1。此时将其减 1，则这次迭代输出的数就是 0；下一次再检验循环条件时，u 的值等

于 0 而无法再进入循环。因为我们要先做减 1 的操作，所以初始化 u 的值应该比要输出的最大值大 1。这里，u 初始化为 11，输出的最大数是 10。

提示：切勿混用带符号类型和无符号类型

如果表达式里既有带符号类型又有无符号类型，当带符号类型取值为负时会出现异常结果，这是因为带符号数会自动地转换成无符号数。例如，在一个形如 a*b 的式子中，如果 a = -1，b = 1，而且 a 和 b 都是 int，则表达式的值显然为-1。然而，如果 a 是 int，而 b 是 unsigned，则结果须视在当前机器上 int 所占位数而定。在我们的环境里，结果是 4294967295。

2.1.2 节练习

38

练习 2.3：读程序写结果。

```
unsigned u = 10, u2 = 42;
std::cout << u2 - u << std::endl;
std::cout << u - u2 << std::endl;

int i = 10, i2 = 42;
std::cout << i2 -i<< std::endl;
std::cout << i - i2<< std::endl;
std::cout << i - u<< std::endl;
std::cout << u - i<< std::endl;
```

练习 2.4：编写程序检查你的估计是否正确，如果不正确，请仔细研读本节直到弄明白问题所在。

2.1.3 字面值常量

一个形如 42 的值被称作**字面值常量**（literal），这样的值一望而知。每个字面值常量都对应一种数据类型，字面值常量的形式和值决定了它的数据类型。

整型和浮点型字面值

我们可以将整型字面值写作十进制数、八进制数或十六进制数的形式。以 0 开头的整数代表八进制数，以 0x 或 0X 开头的代表十六进制数。例如，我们能用下面的任意一种形式来表示数值 20：

```
20 /* 十进制 */      024 /* 八进制 */      0x14 /* 十六进制 */
```

整型字面值具体的数据类型由它的值和符号决定。默认情况下，十进制字面值是带符号数，八进制和十六进制字面值既可能是带符号的也可能是无符号的。十进制字面值的类型是 int、long 和 long long 中尺寸最小的那个（例如，三者当中最小是 int），当然前提是这种类型要能容纳下当前的值。八进制和十六进制字面值的类型是能容纳其数值的 int、unsigned int、long、unsigned long、long long 和 unsigned long long 中的尺寸最小者。如果一个字面值连与之关联的最大的数据类型都放不下，将产生错误。类型 short 没有对应的字面值。在表 2.2（第 37 页）中，我们将以后缀代表相应的字面值类型。

尽管整型字面值可以存储在带符号数据类型中，但严格来说，十进制字面值不会是负

数。如果我们使用了一个形如-42的负十进制字面值，那个负号并不在字面值之内，它的作用仅仅是对字面值取负值而已。

浮点型字面值表现为一个小数或以科学计数法表示的指数，其中指数部分用E或e标识：

```
3.14159      3.14159E0      0.      0e0      .001
```

39 默认的，浮点型字面值是一个double，我们可以使用表2.2（第37页）中的后缀来表示其他浮点型。

字符和字符串字面值

由单引号括起来的一个字符称为char型字面值，双引号括起来的零个或多个字符则构成字符串型字面值。

```
'a'       // 字符字面值
"Hello World!"   // 字符串字面值
```

字符串字面值的类型实际上是由常量字符构成的数组（array），该类型将在3.5.4节（第109页）介绍。编译器在每个字符串的结尾处添加一个空字符（'\0'），因此，字符串字面值的实际长度要比它的内容多1。例如，字面值'A'表示的就是单独的字符A，而字符串"A"则代表了一个字符的数组，该数组包含两个字符：一个是字母A、另一个是空字符。

如果两个字符串字面值位置紧邻且仅由空格、缩进和换行符分隔，则它们实际上是一个整体。当书写的字符串字面值比较长，写在一行里不太合适时，就可以采取分开书写的方式：

```
// 分多行书写的字符串字面值
std::cout << "a really, really long string literal "
             "that spans two lines" << std::endl;
```

转义序列

有两类字符程序员不能直接使用：一类是**不可打印**（nonprintable）的字符，如退格或其他控制字符，因为它们没有可视的图符；另一类是在C++语言中有特殊含义的字符（单引号、双引号、问号、反斜线）。在这些情况下需要用到**转义序列**（escape sequence），转义序列均以反斜线作为开始，C++语言规定的转义序列包括：

换行符	\n	横向制表符	\t	报警（响铃）符	\a
纵向制表符	\v	退格符	\b	双引号	\"
反斜线	\\	问号	\?	单引号	\'
回车符	\r	进纸符	\f		

在程序中，上述转义序列被当作一个字符使用：

```
std::cout << '\n';            // 转到新一行
std::cout << "\tHi!\n";       // 输出一个制表符，输出"Hi!"，转到新一行
```

我们也可以使用泛化的转义序列，其形式是\x后紧跟1个或多个十六进制数字，或者\后紧跟1个、2个或3个八进制数字，其中数字部分表示的是字符对应的数值。假设使用的是Latin-1字符集，以下是一些示例：

```
\7（响铃）    \12（换行符）      \40（空格）
\0（空字符）  \115（字符M）      \x4d（字符M）
```

40 我们可以像使用普通字符那样使用C++语言定义的转义序列：

```
std::cout << "Hi \x4dO\115!\n";   //输出Hi MOM!，转到新一行
```

```
std::cout << '\115' << '\n';        //输出M，转到新一行
```

注意，如果反斜线\后面跟着的八进制数字超过 3 个，只有前 3 个数字与\构成转义序列。例如，"\1234"表示 2 个字符，即八进制数 123 对应的字符以及字符 4。相反，\x 要用到后面跟着的所有数字，例如，"\x1234"表示一个 16 位的字符，该字符由这 4 个十六进制数所对应的比特唯一确定。因为大多数机器的 char 型数据占 8 位，所以上面这个例子可能会报错。一般来说，超过 8 位的十六进制字符都是与表 2.2 中某个前缀作为开头的扩展字符集一起使用的。

指定字面值的类型

通过添加如表 2.2 中所列的前缀和后缀，可以改变整型、浮点型和字符型字面值的默认类型。

```
L'a'          // 宽字符型字面值，类型是 wchar_t
u8"hi!"       // utf-8 字符串字面值（utf-8 用 8 位编码一个 Unicode 字符）
42ULL         // 无符号整型字面值，类型是 unsigned long long
1E-3F         // 单精度浮点型字面值，类型是 float
3.14159L      // 扩展精度浮点型字面值，类型是 long double
```



当使用一个长整型字面值时，请使用大写字母 L 来标记，因为小写字母 l 和数字 1 太容易混淆了。

表 2.2：指定字面值的类型

字符和字符串字面值			
前缀	含义		类型
u	Unicode 16 字符		char16_t
U	Unicode 32 字符		char32_t
L	宽字符		wchar_t
u8	UTF-8（仅用于字符串字面常量）		char
整型字面值		浮点型字面值	
后缀	最小匹配类型	后缀	类型
u or U	unsigned	f 或 F	float
l or L	long	l 或 L	long double
ll or LL	long long		

对于一个整型字面值来说，我们能分别指定它是否带符号以及占用多少空间。如果后缀中有 U，则该字面值属于无符号类型，也就是说，以 U 为后缀的十进制数、八进制数或十六进制数都将从 unsigned int、unsigned long 和 unsigned long long 中选择能匹配的空间最小的一个作为其数据类型。如果后缀中有 L，则字面值的类型至少是 long；如果后缀中有 LL，则字面值的类型将是 long long 和 unsigned long long 中的一种。显然我们可以将 U 与 L 或 LL 合在一起使用。例如，以 UL 为后缀的字面值的数据类型将根据具体数值情况或者取 unsigned long，或者取 unsigned long long。

41

布尔字面值和指针字面值

true 和 false 是布尔类型的字面值：

```
bool test = false;
```

nullptr 是指针字面值，2.3.2 节（第 47 页）将有更多关于指针和指针字面值的介绍。

2.1.3 节练习

练习 2.5：指出下述字面值的数据类型并说明每一组内几种字面值的区别：

```
(a) 'a', L'a', "a", L"a"
(b) 10, 10u, 10L, 10uL, 012, 0xC
(c) 3.14, 3.14f, 3.14L
(d) 10, 10u, 10., 10e-2
```

练习 2.6：下面两组定义是否有区别，如果有，请叙述之：

```
int month = 9, day = 7;
int month = 09, day = 07;
```

练习 2.7：下述字面值表示何种含义？它们各自的数据类型是什么？

```
(a) "Who goes with F\145rgus?\012"
(b) 3.14e1L      (c) 1024f      (d) 3.14L
```

练习 2.8：请利用转义序列编写一段程序，要求先输出 2M，然后转到新一行。修改程序使其先输出 2，然后输出制表符，再输出 M，最后转到新一行。

2.2 变量

变量提供一个具名的、可供程序操作的存储空间。C++中的每个变量都有其数据类型，数据类型决定着变量所占内存空间的大小和布局方式、该空间能存储的值的范围，以及变量能参与的运算。对 C++程序员来说，“变量（variable）”和“对象（object）”一般可以互换使用。

2.2.1 变量定义

变量定义的基本形式是：首先是**类型说明符**（type specifier），随后紧跟由一个或多个
42 变量名组成的列表，其中变量名以逗号分隔，最后以分号结束。列表中每个变量名的类型都由类型说明符指定，定义时还可以为一个或多个变量赋初值：

```
int sum = 0, value,  // sum、value 和 units_sold 都是 int
    units_sold = 0;  // sum 和 units_sold 初值为 0
Sales_item item;     // item 的类型是 Sales_item（参见 1.5.1 节，第 17 页）
// string 是一种库类型，表示一个可变长的字符序列
std::string book("0-201-78345-X");//book 通过一个 string 字面值初始化
```

book 的定义用到了库类型 std::string，像 iostream（参见 1.2 节，第 6 页）一样，string 也是在命名空间 std 中定义的，我们将在第 3 章中对 string 类型做更详细的介绍。眼下，只需了解 string 是一种表示可变长字符序列的数据类型就可以了。C++库提供了几种初始化 string 对象的方法，其中一种是把字面值拷贝给 string 对象（参见 2.1.3 节，第 36 页），因此在上例中，book 被初始化为 0-201-78345-X。

术语：何为对象？

C++程序员们在很多场合都会使用**对象**（object）这个名词。通常情况下，对象是指一块能存储数据并具有某种类型的内存空间。

一些人仅在与类有关的场景下才使用“对象”这个词。另一些人则已把命名的对象和未命名的对象区分开来，他们把命名了的对象叫做变量。还有一些人把对象和值区分开来，其中对象指能被程序修改的数据，而值（value）指只读的数据。

本书遵循大多数人的习惯用法，即认为对象是具有某种数据类型的内存空间。我们在使用对象这个词时，并不严格区分是类还是内置类型，也不区分是否命名或是否只读。

初始值

当对象在创建时获得了一个特定的值，我们说这个对象被**初始化**（initialized）了。用于初始化变量的值可以是任意复杂的表达式。当一次定义了两个或多个变量时，对象的名字随着定义也就马上可以使用了。因此在同一条定义语句中，可以用先定义的变量值去初始化后定义的其他变量。

```
//正确：price 先被定义并赋值，随后被用于初始化 discount
double price = 109.99, discount = price * 0.16;
//正确：调用函数 applyDiscount，然后用函数的返回值初始化 salePrice
double salePrice = applyDiscount(price, discount);
```

在C++语言中，初始化是一个异常复杂的问题，我们也将反复讨论这个问题。很多程序员对于用等号=来初始化变量的方式倍感困惑，这种方式容易让人认为初始化是赋值的一种。事实上在C++语言中，初始化和赋值是两个完全不同的操作。然而在很多编程语言中二者的区别几乎可以忽略不计，即使在C++语言中有时这种区别也无关紧要，所以人们特别容易把二者混为一谈。需要强调的是，这个概念至关重要，我们也将在后面不止一次 43 提及这一点。



初始化不是赋值，初始化的含义是创建变量时赋予其一个初始值，而赋值的含义是把对象的当前值擦除，而以一个新值来替代。

列表初始化

C++语言定义了初始化的好几种不同形式，这也是初始化问题复杂性的一个体现。例如，要想定义一个名为 `units_sold` 的 `int` 变量并初始化为 0，以下的4条语句都可以做到这一点：

```
int units_sold = 0;
int units_sold = {0};
int units_sold{0};
int units_sold(0);
```

作为C++11新标准的一部分，用花括号来初始化变量得到了全面应用，而在此之前， C++11 这种初始化的形式仅在某些受限的场合下才能使用。出于3.3.1节（第88页）将要介绍的原因，这种初始化的形式被称为**列表初始化**（list initialization）。现在，无论是初始化对象还是某些时候为对象赋新值，都可以使用这样一组由花括号括起来的初始值了。

当用于内置类型的变量时，这种初始化形式有一个重要特点：如果我们使用列表初始化且初始值存在丢失信息的风险，则编译器将报错：

```
long double ld = 3.1415926536;
int a{ld}, b = {ld};      // 错误：转换未执行，因为存在丢失信息的危险
int c(ld), d = ld;        // 正确：转换执行，且确实丢失了部分值
```

使用 long double 的值初始化 int 变量时可能丢失数据，所以编译器拒绝了 a 和 b 的初始化请求。其中，至少 ld 的小数部分会丢失掉，而且 int 也可能存不下 ld 的整数部分。

刚刚所介绍的看起来无关紧要，毕竟我们不会故意用 long double 的值去初始化 int 变量。然而，像第 16 章介绍的一样，这种初始化有可能在不经意间发生。我们将在 3.2.1 节（第 76 页）和 3.3.1 节（第 88 页）对列表初始化做更多介绍。

默认初始化

如果定义变量时没有指定初值，则变量被**默认初始化**（default initialized），此时变量被赋予了“默认值”。默认值到底是什么由变量类型决定，同时定义变量的位置也会对此有影响。

44

如果是内置类型的变量未被显式初始化，它的值由定义的位置决定。定义于任何函数体之外的变量被初始化为 0。然而如 6.1.1 节（第 185 页）所示，一种例外情况是，定义在函数体内部的内置类型变量将**不被初始化**（uninitialized）。一个未被初始化的内置类型变量的值是未定义的（参见 2.1.2 节，第 33 页），如果试图拷贝或以其他形式访问此类值将引发错误。

每个类各自决定其初始化对象的方式。而且，是否允许不经初始化就定义对象也由类自己决定。如果类允许这种行为，它将决定对象的初始值到底是什么。

绝大多数类都支持无须显式初始化而定义对象，这样的类提供了一个合适的默认值。例如，以刚刚所见为例，string 类规定如果没有指定初值则生成一个空串：

```
std::string empty;    // empty 非显式地初始化为一个空串
Sales_item item;      // 被默认初始化的 Sales_item 对象
```

一些类要求每个对象都显式初始化，此时如果创建了一个该类的对象而未对其做明确的初始化操作，将引发错误。



定义于函数体内的内置类型的对象如果没有初始化，则其值未定义。类的对象如果没有显式地初始化，则其值由类确定。

2.2.1 节练习

练习 2.9：解释下列定义的含义。对于非法的定义，请说明错在何处并将其改正。

```
(a) std::cin >> int input_value;     (b) int i = { 3.14 };
(c) double salary = wage = 9999.99; (d) int i = 3.14;
```

练习 2.10：下列变量的初值分别是什么？

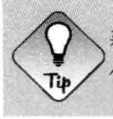
```
std::string global_str;
int global_int;
int main()
{
    int local_int;
    std::string local_str;
}
```

45

提示：未初始化变量引发运行时故障

未初始化的变量含有一个不确定的值，使用未初始化变量的值是一种错误的编程行为并且很难调试。尽管大多数编译器都能对一部分使用未初始化变量的行为提出警告，但严格来说，编译器并未被要求检查此类错误。

使用未初始化的变量将带来无法预计的后果。有时我们足够幸运，一访问此类对象程序就崩溃并报错，此时只要找到崩溃的位置就很容易发现变量没被初始化的问题。另外一些时候，程序会一直执行完并产生错误的结果。更糟糕的情况是，程序结果时对时错、无法把握。而且，往无关的位置添加代码还会导致我们误以为程序对了，其实结果仍旧有错。



建议初始化每一个内置类型的变量。虽然并非必须这么做，但如果我们不能确保初始化后程序安全，那么这么做不失为一种简单可靠的方法。

2.2.2 变量声明和定义的关系



为了允许把程序拆分成多个逻辑部分来编写，C++语言支持**分离式编译**（separate compilation）机制，该机制允许将程序分割为若干个文件，每个文件可被独立编译。

如果将程序分为多个文件，则需要有在文件间共享代码的方法。例如，一个文件的代码可能需要使用另一个文件中定义的变量。一个实际的例子是 `std::cout` 和 `std::cin`，它们定义于标准库，却能被我们写的程序使用。

为了支持分离式编译，C++语言将声明和定义区分开来。**声明**（declaration）使得名字为程序所知，一个文件如果想使用别处定义的名字则必须包含对那个名字的声明。而**定义**（definition）负责创建与名字关联的实体。

变量声明规定了变量的类型和名字，在这一点上定义与之相同。但是除此之外，定义还申请存储空间，也可能会为变量赋一个初始值。

如果想声明一个变量而非定义它，就在变量名前添加关键字 `extern`，而且不要显式地初始化变量：

```
extern int i;     // 声明 i 而非定义 i
int j;            // 声明并定义 j
```

任何包含了显式初始化的声明即成为定义。我们能给由 `extern` 关键字标记的变量赋一个初始值，但是这么做也就抵消了 `extern` 的作用。`extern` 语句如果包含初始值就不再是声明，而变成定义了：

```
extern double pi = 3.1416; // 定义
```

在函数体内部，如果试图初始化一个由 `extern` 关键字标记的变量，将引发错误。



变量能且只能被定义一次，但是可以被多次声明。

声明和定义的区别看起来也许微不足道，但实际上却非常重要。如果要在多个文件中使用同一个变量，就必须将声明和定义分离。此时，变量的定义必须出现在且只能出现在一个文件中，而其他用到该变量的文件必须对其进行声明，却绝对不能重复定义。

关于 C++语言对分离式编译的支持我们将在 2.6.3 节（第 67 页）和 6.1.3 节（第 186

46

页）中做更详细的介绍。

2.2.2 节练习

练习 2.11：指出下面的语句是声明还是定义：

```
(a) extern int ix = 1024;
(b) int iy;
(c) extern int iz;
```

关键概念：静态类型

C++是一种静态类型（statically typed）语言，其含义是在编译阶段检查类型。其中，检查类型的过程称为类型检查（type checking）。

我们已经知道，对象的类型决定了对象所能参与的运算。在C++语言中，编译器负责检查数据类型是否支持要执行的运算，如果试图执行类型不支持的运算，编译器将报错并且不会生成可执行文件。

程序越复杂，静态类型检查越有助于发现问题。然而，前提是编译器必须知道每一个实体对象的类型，这就要求我们在使用某个变量之前必须声明其类型。

2.2.3 标识符

C++的标识符（identifier）由字母、数字和下画线组成，其中必须以字母或下画线开头。标识符的长度没有限制，但是对大小写字母敏感：

```
// 定义 4 个不同的 int 变量
int somename, someName, SomeName, SOMENAME;
```

如表 2.3 和表 2.4 所示，C++语言保留了一些名字供语言本身使用，这些名字不能被用作标识符。

同时，C++也为标准库保留了一些名字。用户自定义的标识符中不能连续出现两个下画线，也不能以下画线紧连大写字母开头。此外，定义在函数体外的标识符不能以下画线开头。

变量命名规范

变量命名有许多约定俗成的规范，下面的这些规范能有效提高程序的可读性：

47

- 标识符要能体现实际含义。
- 变量名一般用小写字母，如 index，不要使用 Index 或 INDEX。
- 用户自定义的类名一般以大写字母开头，如 Sales_item。
- 如果标识符由多个单词组成，则单词间应有明显区分，如 student_loan 或 studentLoan，不要使用 studentloan。

Best Practices 对于命名规范来说，若能坚持，必将有效。

表 2.3：C++关键字

alignas	continue	friend	register	true
alignof	decltype	goto	reinterpret_cast	try
asm	default	if	return	typedef
auto	delete	inline	short	typeid
bool	do	int	signed	typename
break	double	long	sizeof	union
case	dynamic_cast	mutable	static	unsigned
catch	else	namespace	static_assert	using
char	enum	new	static_cast	virtual
char16_t	explicit	noexcept	struct	void
char32_t	export	nullptr	switch	volatile
class	extern	operator	template	wchar_t
const	false	private	this	while
constexpr	float	protected	thread_local	
const_cast	for	public	throw	

表 2.4：C++操作符替代名

and	bitand	compl	not_eq	or_eq	xor_eq
and_eq	bitor	not	or	xor	

2.2.3 节练习

练习 2.12：请指出下面的名字中哪些是非法的？

(a) `int double = 3.14;`　　(b) `int _;`

(c) `int catch-22;`　　(d) `int 1_or_2 = 1;`

(e) `double Double = 3.14;`

2.2.4 名字的作用域



不论是在程序的什么位置，使用到的每个名字都会指向一个特定的实体：变量、函数、类型等。然而，同一个名字如果出现在程序的不同位置，也可能指向的是不同实体。

48

作用域（scope）是程序的一部分，在其中名字有其特定的含义。C++语言中大多数作用域都以花括号分隔。

同一个名字在不同的作用域中可能指向不同的实体。名字的有效区域始于名字的声明语句，以声明语句所在的作用域末端为结束。

一个典型的示例来自于 1.4.2 节（第 11 页）的程序：

```
#include <iostream>
int main()
{
    int sum = 0;
    // sum 用于存放从 1 到 10 所有数的和
    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;
}
```

这段程序定义了3个名字：main、sum和val，同时使用了命名空间名字std，该空间提供了2个名字cout和cin供程序使用。

名字main定义于所有花括号之外，它和其他大多数定义在函数体之外的名字一样拥有**全局作用域**（global scope）。一旦声明之后，全局作用域内的名字在整个程序的范围内都可使用。名字sum定义于main函数所限定的作用域之内，从声明sum开始直到main函数结束为止都可以访问它，但是出了main函数所在的块就无法访问了，因此说变量sum拥有**块作用域**（block scope）。名字val定义于for语句内，在for语句之内可以访问val，但是在main函数的其他部分就不能访问它了。

建议：当你第一次使用变量时再定义它

一般来说，在对象第一次被使用的地方附近定义它是一种好的选择，因为这样做有助于更容易地找到变量的定义。更重要的是，当变量的定义与它第一次被使用的地方很近时，我们也会赋给它一个比较合理的初始值。

嵌套的作用域

作用域能彼此包含，被包含（或者说被嵌套）的作用域称为**内层作用域**（inner scope），包含着别的作用域的作用域称为**外层作用域**（outer scope）。

作用域中一旦声明了某个名字，它所嵌套着的所有作用域中都能访问该名字。同时，允许在内层作用域中重新定义外层作用域已有的名字：

49

```
#include <iostream>
// 该程序仅用于说明：函数内部不宜定义与全局变量同名的新变量
int reused = 42; // reused 拥有全局作用域
int main()
{
    int unique = 0; // unique 拥有块作用域
    // 输出#1：使用全局变量 reused;输出 42 0
    std::cout << reused << " " << unique << std::endl;
    int reused = 0; // 新建局部变量 reused，覆盖了全局变量 reused
    // 输出#2：使用局部变量 reused；输出 0 0
    std::cout << reused << " " << unique << std::endl;
    // 输出#3：显式地访问全局变量 reused；输出 42 0
    std::cout << ::reused << " " << unique << std::endl;
    return 0;
}
```

输出#1出现在局部变量reused定义之前，因此这条语句使用全局作用域中定义的名字reused，输出42 0。输出#2发生在局部变量reused定义之后，此时局部变量reused**正在作用域内**（in scope），因此第二条输出语句使用的是局部变量reused而非全局变量，输出0 0。输出#3使用作用域操作符（参见1.2节，第7页）来覆盖默认的作用域规则，因为全局作用域本身并没有名字，所以当作用域操作符的左侧为空时，向全局作用域发出请求获取作用域操作符右侧名字对应的变量。结果是，第三条输出语句使用全局变量reused，输出42 0。



如果函数有可能用到某全局变量，则不宜再定义一个同名的局部变量。

2.2.4 节练习

练习 2.13：下面程序中 j 的值是多少？

```
int i = 42;
int main()
{
    int i = 100;
    int j = i;
}
```

练习 2.14：下面的程序合法吗？如果合法，它将输出什么？

```
int i = 100, sum = 0;
for (int i = 0; i != 10; ++i)
    sum += i;
std::cout << i << " " << sum << std::endl;
```

2.3 复合类型



复合类型（compound type）是指基于其他类型定义的类型。C++语言有几种复合类型，50本章将介绍其中的两种：引用和指针。

与我们已经掌握的变量声明相比，定义复合类型的变量要复杂很多。2.2 节（第 38 页）提到，一条简单的声明语句由一个数据类型和紧随其后的一个变量名列表组成。其实更通用的描述是，一条声明语句由一个**基本数据类型**（base type）和紧随其后的一个**声明符**（declarator）列表组成。每个声明符命名了一个变量并指定该变量为与基本数据类型有关的某种类型。

目前为止，我们所接触的声明语句中，声明符其实就是变量名，此时变量的类型也就是声明的基本数据类型。其实还可能有更复杂的声明符，它基于基本数据类型得到更复杂的类型，并把它指定给变量。

2.3.1 引用





C++11 中新增了一种引用：所谓的“右值引用（rvalue reference）”，我们将在 13.6.1 节（第 471 页）做更详细的介绍。这种引用主要用于内置类。严格来说，当我们使用术语“引用（reference）”时，指的其实是“左值引用（lvalue reference）”。

引用（reference）为对象起了另外一个名字，引用类型引用（refers to）另外一种类型。通过将声明符写成&d 的形式来定义引用类型，其中 d 是声明的变量名：

```
int ival = 1024;
int &refVal = ival;        // refVal 指向 ival（是 ival 的另一个名字）
int &refVal2;              // 报错：引用必须被初始化
```

一般在初始化变量时，初始值会被拷贝到新建的对象中。然而定义引用时，程序把引用和它的初始值**绑定**（bind）在一起，而不是将初始值拷贝给引用。一旦初始化完成，引用将和它的初始值对象一直绑定在一起。因为无法令引用重新绑定到另外一个对象，因此引用必须初始化。

引用即别名



引用并非对象，相反的，它只是为一个已经存在的对象所起的另外一个名字。

定义了一个引用之后，对其进行的所有操作都是在与之绑定的对象上进行的：

```
refVal = 2;        // 把 2 赋给 refVal 指向的对象，此处即是赋给了 ival
int ii = refVal; // 与 ii = ival 执行结果一样
```

51 为引用赋值，实际上是把值赋给了与引用绑定的对象。获取引用的值，实际上是获取了与引用绑定的对象的值。同理，以引用作为初始值，实际上是以与引用绑定的对象作为初始值：

```
// 正确：refVal3 绑定到了那个与 refVal 绑定的对象上，这里就是绑定到 ival 上
int &refVal3 = refVal;
// 利用与 refVal 绑定的对象的值初始化变量 i
int i = refVal; // 正确：i 被初始化为 ival 的值
```

因为引用本身不是一个对象，所以不能定义引用的引用。

引用的定义

允许在一条语句中定义多个引用，其中每个引用标识符都必须以符号&开头：

```
int i = 1024, i2 = 2048; // i 和 i2 都是 int
int &r = i, r2 = i2;     // r 是一个引用，与 i 绑定在一起，r2 是 int
int i3 = 1024, &ri = i3; // i3 是 int，ri 是一个引用，与 i3 绑定在一起
int &r3 = i3, &r4 = i2;  // r3 和 r4 都是引用
```

除了 2.4.1 节（第 55 页）和 15.2.3 节（第 534 页）将要介绍的两种例外情况，其他所有引用的类型都要和与之绑定的对象严格匹配。而且，引用只能绑定在对象上，而不能与字面值或某个表达式的计算结果绑定在一起，相关原因将在 2.4.1 节详述：

```
int &refVal4 = 10;        // 错误：引用类型的初始值必须是一个对象
double dval = 3.14;
int &refVal5 = dval;      // 错误：此处引用类型的初始值必须是 int 型对象
```

2.3.1 节练习

练习 2.15：下面的哪个定义是不合法的？为什么？

(a) `int ival = 1.01;`　　(b) `int &rval1 = 1.01;`
(c) `int &rval2 = ival;`　　(d) `int &rval3;`

练习 2.16：考查下面的所有赋值然后回答：哪些赋值是不合法的？为什么？哪些赋值是合法的？它们执行了什么样的操作？

```
int i = 0, &r1 = i;        double d = 0, &r2 = d;
```
(a) `r2 = 3.14159;`　　(b) `r2 = r1;`
(c) `i = r2;`　　(d) `r1 = d;`

练习 2.17：执行下面的代码段将输出什么结果？

```
int i, &ri = i;
i = 5; ri = 10;
std::cout << i << " " << ri << std::endl;
```

2.3.2 指针



指针（pointer）是“指向（point to）”另外一种类型的复合类型。与引用类似，指针 52 也实现了对其他对象的间接访问。然而指针与引用相比又有很多不同点。其一，指针本身就是一个对象，允许对指针赋值和拷贝，而且在指针的生命周期内它可以先后指向几个不同的对象。其二，指针无须在定义时赋初值。和其他内置类型一样，在块作用域内定义的指针如果没有被初始化，也将拥有一个不确定的值。



指针通常难以理解，即使是有经验的程序员也常常因为调试指针引发的错误而被备受折磨。

定义指针类型的方法将声明符写成*d 的形式，其中 d 是变量名。如果在一条语句中定义了几个指针变量，每个变量前面都必须有符号*：

```
int *ip1, *ip2;   // ip1 和 ip2 都是指向 int 型对象的指针
double dp, *dp2;  // dp2 是指向 double 型对象的指针，dp 是 double 型对象
```

获取对象的地址

指针存放某个对象的地址，要想获取该地址，需要使用**取地址符**（操作符&）：

```
int ival = 42;
int *p = &ival;  // p 存放变量 ival 的地址，或者说 p 是指向变量 ival 的指针
```

第二条语句把 p 定义为一个指向 int 的指针，随后初始化 p 令其指向名为 ival 的 int 对象。因为引用不是对象，没有实际地址，所以不能定义指向引用的指针。

除了 2.4.2 节（第 56 页）和 15.2.3 节（第 534 页）将要介绍的两种例外情况，其他所有指针的类型都要和它所指向的对象严格匹配：

```
double dval;
double *pd = &dval;   // 正确：初始值是 double 型对象的地址
double *pd2 = pd;     // 正确：初始值是指向 double 对象的指针

int *pi = pd;         // 错误：指针 pi 的类型和 pd 的类型不匹配
pi = &dval;           // 错误：试图把 double 型对象的地址赋给 int 型指针
```

因为在声明语句中指针的类型实际上被用于指定它所指向对象的类型，所以二者必须匹配。如果指针指向了一个其他类型的对象，对该对象的操作将发生错误。

指针值

指针的值（即地址）应属下列 4 种状态之一：

1. 指向一个对象。
2. 指向紧邻对象所占空间的下一个位置。
3. 空指针，意味着指针没有指向任何对象。
4. 无效指针，也就是上述情况之外的其他值。

试图拷贝或以其他方式访问无效指针的值都将引发错误。编译器并不负责检查此类错误，
53 这一点和试图使用未经初始化的变量是一样的。访问无效指针的后果无法预计，因此程序员必须清楚任意给定的指针是否有效。

尽管第2种和第3种形式的指针是有效的，但其使用同样受到限制。显然这些指针没有指向任何具体对象，所以试图访问此类指针（假定的）对象的行为不被允许。如果这样做了，后果也无法预计。

利用指针访问对象

如果指针指向了一个对象，则允许使用**解引用符**（操作符*）来访问该对象：

```
int ival = 42;
int *p = &ival;  // p存放着变量ival的地址，或者说p是指向变量ival的指针
cout << *p;      // 由符号*得到指针p所指的对象，输出42
```

对指针解引用会得出所指的对象，因此如果给解引用的结果赋值，实际上也就是给指针所指的对象赋值：

```
*p = 0;       // 由符号*得到指针p所指的对象，即可经由p为变量ival赋值
cout << *p;  // 输出0
```

如上述程序所示，为*p赋值实际上是为p所指的对象赋值。



解引用操作仅适用于那些确实指向了某个对象的有效指针。

关键概念：某些符号有多重含义

像&和*这样的符号，既能用作表达式里的运算符，也能作为声明的一部分出现，符号的上下文决定了符号的意义：

```
int i = 42;
int &r = i;       // &紧随类型名出现，因此是声明的一部分，r是一个引用
int *p;           // *紧随类型名出现，因此是声明的一部分，p是一个指针
p = &i;           // &出现在表达式中，是一个取地址符
*p = i;           // *出现在表达式中，是一个解引用符
int &r2 = *p;     // &是声明的一部分，*是一个解引用符
```

在声明语句中，&和*用于组成复合类型；在表达式中，它们的角色又转变成运算符。在不同场景下出现的虽然是同一个符号，但是由于含义截然不同，所以我们完全可以把它当作不同的符号来看待。

空指针

空指针（null pointer）不指向任何对象，在试图使用一个指针之前代码可以首先检查它是否为空。以下列出几个生成空指针的方法：

54
```
int *p1 = nullptr;        // 等价于int *p1 = 0;
int *p2 = 0;              // 直接将p2初始化为字面常量0
// 需要首先#include cstdlib
int *p3 = NULL;           // 等价于int *p3 = 0;
```

C++11 得到空指针最直接的办法就是用字面值 **nullptr** 来初始化指针，这也是C++11新标准刚刚引入的一种方法。nullptr是一种特殊类型的字面值，它可以被转换成（参见2.1.2节，

第 32 页）任意其他的指针类型。另一种办法就如对 p2 的定义一样，也可以通过将指针初始化为字面值 0 来生成空指针。

过去的程序还会用到一个名为 NULL 的**预处理变量**（preprocessor variable）来给指针赋值，这个变量在头文件 cstdlib 中定义，它的值就是 0。

2.6.3 节（第 68 页）将稍微介绍一点关于预处理器的知识，现在只要知道预处理器是运行于编译过程之前的一段程序就可以了。预处理变量不属于命名空间 std，它由预处理器负责管理，因此我们可以直接使用预处理变量而无须在前面加上 std::。

当用到一个预处理变量时，预处理器会自动地将它替换为实际值，因此用 NULL 初始化指针和用 0 初始化指针是一样的。在新标准下，现在的 C++程序最好使用 nullptr，同时尽量避免使用 NULL。

把 int 变量直接赋给指针是错误的操作，即使 int 变量的值恰好等于 0 也不行。

```
int zero = 0;
pi = zero;          // 错误：不能把 int 变量直接赋给指针
```

建议：初始化所有指针

使用未经初始化的指针是引发运行时错误的一大原因。

和其他变量一样，访问未经初始化的指针所引发的后果也是无法预计的。通常这一行为将造成程序崩溃，而且一旦崩溃，要想定位到出错位置将是特别棘手的问题。

在大多数编译器环境下，如果使用了未经初始化的指针，则该指针所占内存空间的当前内容将被看作一个地址值。访问该指针，相当于去访问一个本不存在的位置上的本不存在的对象。糟糕的是，如果指针所占内存空间中恰好有内容，而这些内容又被当作了某个地址，我们就很难分清它到底是合法的还是非法的了。

因此建议初始化所有的指针，并且在可能的情况下，尽量等定义了对象之后再定义指向它的指针。如果实在不清楚指针应该指向何处，就把它初始化为 nullptr 或者 0，这样程序就能检测并知道它没有指向任何具体的对象了。

赋值和指针

指针和引用都能提供对其他对象的间接访问，然而在具体实现细节上二者有很大不同，其中最重要的一点就是引用本身并非一个对象。一旦定义了引用，就无法令其再绑定到另外的对象，之后每次使用这个引用都是访问它最初绑定的那个对象。 55

指针和它存放的地址之间就没有这种限制了。和其他任何变量（只要不是引用）一样，给指针赋值就是令它存放一个新的地址，从而指向一个新的对象：

```
int i = 42;
int *pi = 0;          // pi 被初始化，但没有指向任何对象
int *pi2 = &i;        // pi2 被初始化，存有 i 的地址
int *pi3;             // 如果 pi3 定义于块内，则 pi3 的值是无法确定的

pi3 = pi2;            // pi3 和 pi2 指向同一个对象 i
pi2 = 0;              // 现在 pi2 不指向任何对象了
```

有时候要想搞清楚一条赋值语句到底是改变了指针的值还是改变了指针所指对象的值不太容易，最好的办法就是记住赋值永远改变的是等号左侧的对象。当写出如下语句时，

```
pi = &ival;           // pi 的值被改变，现在 pi 指向了 ival
```

意思是为 pi 赋一个新的值，也就是改变了那个存放在 pi 内的地址值。相反的，如果写出如下语句，

```
*pi = 0;              // ival 的值被改变，指针 pi 并没有改变
```

则*pi（也就是指针 pi 指向的那个对象）发生改变。

其他指针操作

只要指针拥有一个合法值，就能将它用在条件表达式中。和采用算术值作为条件（参见 2.1.2 节，第 32 页）遵循的规则类似，如果指针的值是 0，条件取 false：

```
int ival = 1024;
int *pi = 0;                  // pi 合法，是一个空指针
int *pi2 = &ival;             // pi2 是一个合法的指针，存放着 ival 的地址
if (pi)                       // pi 的值是 0，因此条件的值是 false
    // ...
if (pi2)                      // pi2 指向 ival，因此它的值不是 0，条件的值是 true
    // ...
```

任何非 0 指针对应的条件值都是 true。

对于两个类型相同的合法指针，可以用相等操作符（==）或不相等操作符（!=）来比较它们，比较的结果是布尔类型。如果两个指针存放的地址值相同，则它们相等；反之它们不相等。这里两个指针存放的地址值相同（两个指针相等）有三种可能：它们都为空、都指向同一个对象，或者都指向了同一个对象的下一地址。需要注意的是，一个指针指向某对象，同时另一个指针指向另外对象的下一地址，此时也有可能出现这两个指针值相同的情况，即指针相等。

因为上述操作要用到指针的值，所以不论是作为条件出现还是参与比较运算，都必须使用合法指针，使用非法指针作为条件或进行比较都会引发不可预计的后果。

3.5.3 节（第 105 页）将介绍更多关于指针的操作。

56 void* 指针

void*是一种特殊的指针类型，可用于存放任意对象的地址。一个 void*指针存放着一个地址，这一点和其他指针类似。不同的是，我们对该地址中到底是个什么类型的对象并不了解：

```
double obj = 3.14, *pd = &obj;
                          // 正确：void*能存放任意类型对象的地址
void *pv = &obj;          // obj 可以是任意类型的对象
pv = pd;                  // pv 可以存放任意类型的指针
```

利用 void*指针能做的事儿比较有限：拿它和别的指针比较、作为函数的输入或输出，或者赋给另外一个 void*指针。不能直接操作 void*指针所指的对象，因为我们并不知道这个对象到底是什么类型，也就无法确定能在这个对象上做哪些操作。

概括说来，以 void*的视角来看内存空间也就仅仅是内存空间，没办法访问内存空间中所存的对象，关于这点将在 19.1.1 节（第 726 页）有更详细的介绍，4.11.3 节（第 144 页）将讲述获取 void*指针所存地址的方法。

2.3.2 节练习

练习 2.18：编写代码分别更改指针的值以及指针所指对象的值。

练习 2.19：说明指针和引用的主要区别。

练习 2.20：请叙述下面这段代码的作用。

```
int i = 42;
int *p1 = &i;
*p1 = *p1 * *p1;
```

练习 2.21：请解释下述定义。在这些定义中有非法的吗？如果有，为什么？

```
int i = 0;
(a) double* dp = &i; (b) int *ip = i; (c) int *p = &i;
```

练习 2.22：假设 p 是一个 int 型指针，请说明下述代码的含义。

```
if (p) // ...
if (*p) // ...
```

练习 2.23：给定指针 p，你能知道它是否指向了一个合法的对象吗？如果能，叙述判断的思路；如果不能，也请说明原因。

练习 2.24：在下面这段代码中为什么 p 合法而 lp 非法？

```
int i = 42;      void *p = &i;        long *lp = &i;
```



2.3.3 理解复合类型的声明

如前所述，变量的定义包括一个基本数据类型（base type）和一组声明符。在同一条 57 定义语句中，虽然基本数据类型只有一个，但是声明符的形式却可以不同。也就是说，一条定义语句可能定义出不同类型的变量：

```
// i 是一个 int 型的数，p 是一个 int 型指针，r 是一个 int 型引用
int i = 1024, *p = &i, &r = i;
```



很多程序员容易迷惑于基本数据类型和类型修饰符的关系，其实后者不过是声明符的一部分罢了。



定义多个变量

经常有一种观点会误以为，在定义语句中，类型修饰符（*或&）作用于本次定义的全部变量。造成这种错误看法的原因有很多，其中之一是我们可以把空格写在类型修饰符和变量名中间：

```
int* p;          // 合法但是容易产生误导
```

我们说这种写法可能产生误导是因为 int*放在一起好像是这条语句中所有变量共同的类型一样。其实恰恰相反，基本数据类型是 int 而非 int*。*仅仅是修饰了 p 而已，对该声明语句中的其他变量，它并不产生任何作用：

```
int* p1, p2;     // p1 是指向 int 的指针，p2 是 int
```

涉及指针或引用的声明，一般有两种写法。第一种把修饰符和变量标识符写在一起：

```
int *p1, *p2; // p1 和 p2 都是指向 int 的指针
```

这种形式着重强调变量具有的复合类型。第二种把修饰符和类型名写在一起，并且每条语句只定义一个变量：

```
int* p1;          // p1 是指向 int 的指针
int* p2;          // p2 是指向 int 的指针
```

这种形式着重强调本次声明定义了一种复合类型。



上述两种定义指针或引用的不同方法没有孰对孰错之分，关键是选择并坚持其中的一种写法，不要总是变来变去。

本书采用第一种写法，将*（或是&）与变量名连在一起。

指向指针的指针

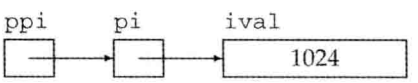
一般来说，声明符中修饰符的个数并没有限制。当有多个修饰符连写在一起时，按照其逻辑关系详加解释即可。以指针为例，指针是内存中的对象，像其他对象一样也有自己的地址，因此允许把指针的地址再存放到另一个指针当中。

58

通过*的个数可以区分指针的级别。也就是说，**表示指向指针的指针，***表示指向指针的指针的指针，以此类推：

```
int ival = 1024;
int *pi = &ival; // pi 指向一个 int 型的数
int **ppi = &pi; // ppi 指向一个 int 型的指针
```

此处 pi 是指向 int 型数的指针，而 ppi 是指向 int 型指针的指针，下图描述了它们之间的关系。



解引用 int 型指针会得到一个 int 型的数，同样，解引用指向指针的指针会得到一个指针。此时为了访问最原始的那个对象，需要对指针的指针做两次解引用：

```
cout << "The value of ival\n"
     << "direct value: " << ival << "\n"
     << "indirect value: " << *pi << "\n"
     << "doubly indirect value: " << **ppi
     << endl;
```

该程序使用三种不同的方式输出了变量 ival 的值：第一种直接输出；第二种通过 int 型指针 pi 输出；第三种两次解引用 ppi，取得 ival 的值。

指向指针的引用

引用本身不是一个对象，因此不能定义指向引用的指针。但指针是对象，所以存在对指针的引用：

```
int i = 42;
int *p;           // p 是一个 int 型指针
int *&r = p;      // r 是一个对指针 p 的引用

r = &i;           // r 引用了一个指针，因此给 r 赋值&i 就是令 p 指向 i
*r = 0;           // 解引用 r 得到 i，也就是 p 指向的对象，将 i 的值改为 0
```

要理解 r 的类型到底是什么，最简单的办法是从右向左阅读 r 的定义。离变量名最近的符号（此例中是&r 的符号&）对变量的类型有最直接的影响，因此 r 是一个引用。声明符的其余部分用以确定 r 引用的类型是什么，此例中的符号*说明 r 引用的是一个指针。最后，声明的基本数据类型部分指出 r 引用的是一个 int 指针。



面对一条比较复杂的指针或引用的声明语句时，从右向左阅读有助于弄清楚它的真实含义。

2.3.3 节练习

59

练习 2.25：说明下列变量的类型和值。

```
(a) int* ip,i, &r = i; (b) int i, *ip = 0; (c) int* ip, ip2;
```

2.4 const 限定符



有时我们希望定义这样一种变量，它的值不能被改变。例如，用一个变量来表示缓冲区的大小。使用变量的好处是当我们觉得缓冲区大小不再合适时，很容易对其进行调整。另一方面，也应随时警惕防止程序一不小心改变了这个值。为了满足这一要求，可以用关键字 **const** 对变量的类型加以限定：

```
const int bufSize = 512;        // 输入缓冲区大小
```

这样就把 bufSize 定义成了一个常量。任何试图为 bufSize 赋值的行为都将引发错误：

```
bufSize = 512;                  // 错误：试图向 const 对象写值
```

因为 const 对象一旦创建后其值就不能再改变，所以 const 对象必须初始化。一如既往，初始值可以是任意复杂的表达式：

```
const int i = get_size();       // 正确：运行时初始化
const int j = 42;               // 正确：编译时初始化
const int k;                    // 错误：k 是一个未经初始化的常量
```

初始化和 const

正如之前反复提到的，对象的类型决定了其上的操作。与非 const 类型所能参与的操作相比，const 类型的对象能完成其中大部分，但也不是所有的操作都适合。主要的限制就是只能在 const 类型的对象上执行不改变其内容的操作。例如，const int 和普通的 int 一样都能参与算术运算，也都能转换成一个布尔值，等等。

在不改变 const 对象的操作中还有一种是初始化，如果利用一个对象去初始化另外一个对象，则它们是不是 const 都无关紧要：

```
int i = 42;
const int ci = i;               // 正确：i 的值被拷贝给了 ci
int j = ci;                     // 正确：ci 的值被拷贝给了 j
```

尽管 ci 是整型常量，但无论如何 ci 中的值还是一个整型数。ci 的常量特征仅仅在执行改变 ci 的操作时才会发挥作用。当用 ci 去初始化 j 时，根本无须在意 ci 是不是一个常量。拷贝一个对象的值并不会改变它，一旦拷贝完成，新的对象就和原来的对象没什么关系了。

60 默认状态下，const 对象仅在文件内有效

当以编译时初始化的方式定义一个 const 对象时，就如对 bufSize 的定义一样：

```
const int bufSize = 512; // 输入缓冲区大小
```

编译器将在编译过程中把用到该变量的地方都替换成对应的值。也就是说，编译器会找到代码中所有用到 bufSize 的地方，然后用 512 替换。

为了执行上述替换，编译器必须知道变量的初始值。如果程序包含多个文件，则每个用了 const 对象的文件都必须得能访问到它的初始值才行。要做到这一点，就必须在每一个用到变量的文件中都有对它的定义（参见 2.2.2 节，第 41 页）。为了支持这一用法，同时避免对同一变量的重复定义，默认情况下，const 对象被设定为仅在文件内有效。当多个文件中出现了同名的 const 变量时，其实等同于在不同文件中分别定义了独立的变量。

某些时候有这样一种 const 变量，它的初始值不是一个常量表达式，但又确实有必要在文件间共享。这种情况下，我们不希望编译器为每个文件分别生成独立的变量。相反，我们想让这类 const 对象像其他（非常量）对象一样工作，也就是说，只在一个文件中定义 const，而在其他多个文件中声明并使用它。

解决的办法是，对于 const 变量不管是声明还是定义都添加 extern 关键字，这样只需定义一次就可以了：

```
// file_1.cc 定义并初始化了一个常量，该常量能被其他文件访问
extern const int bufSize = fcn();
// file_1.h 头文件
extern const int bufSize; // 与 file_1.cc 中定义的 bufSize 是同一个
```

如上述程序所示，file_1.cc 定义并初始化了 bufSize。因为这条语句包含了初始值，所以它（显然）是一次定义。然而，因为 bufSize 是一个常量，必须用 extern 加以限定使其被其他文件使用。

file_1.h 头文件中的声明也由 extern 做了限定，其作用是指明 bufSize 并非本文件所独有，它的定义将在别处出现。



如果想在多个文件之间共享 const 对象，必须在变量的定义之前添加 extern 关键字。

2.4 节练习

练习 2.26：下面哪些句子是合法的？如果有不合法的句子，请说明为什么？

(a) `const int buf;`　　(b) `int cnt = 0;`
(c) `const int sz = cnt;`　　(d) `++cnt; ++sz;`

2.4.1 const 的引用

61 可以把引用绑定到 const 对象上，就像绑定到其他对象上一样，我们称之为**对常量的引用**（reference to const）。与普通引用不同的是，对常量的引用不能被用作修改它所绑定的对象：

```
const int ci = 1024;
const int &r1 = ci;  // 正确：引用及其对应的对象都是常量
```

```
r1 = 42;                    // 错误：r1 是对常量的引用
int &r2 = ci;               // 错误：试图让一个非常量引用指向一个常量对象
```

因为不允许直接为 ci 赋值，当然也就不能通过引用去改变 ci。因此，对 r2 的初始化是错误的。假设该初始化合法，则可以通过 r2 来改变它引用对象的值，这显然是不正确的。

术语：常量引用是对 const 的引用

C++程序员们经常把词组“对 const 的引用”简称为“常量引用”，这一简称还是挺靠谱的，不过前提是你得时刻记得这就是个简称而已。

严格来说，并不存在常量引用。因为引用不是一个对象，所以我们没法让引用本身恒定不变。事实上，由于 C++语言并不允许随意改变引用所绑定的对象，所以从这层意义上理解所有的引用又都算是常量。引用的对象是常量还是非常量可以决定其所能参与的操作，却无论如何都不会影响到引用和对象的绑定关系本身。

初始化和对 const 的引用

2.3.1 节（第 46 页）提到，引用的类型必须与其所引用对象的类型一致，但是有两个例外。第一种例外情况就是在初始化常量引用时允许用任意表达式作为初始值，只要该表达式的结果能转换成（参见 2.1.2 节，第 32 页）引用的类型即可。尤其，允许为一个常量引用绑定非常量的对象、字面值，甚至是个一般表达式：

```
int i = 42;
const int &r1 = i;          // 允许将 const int&绑定到一个普通 int 对象上
const int &r2 = 42;         // 正确：r1 是一个常量引用
const int &r3 = r1 * 2;     // 正确：r3 是一个常量引用
int &r4 = r1 * 2;           // 错误：r4 是一个普通的非常量引用
```

要想理解这种例外情况的原因，最简单的办法是弄清楚当一个常量引用被绑定到另外一种类型上时到底发生了什么：

```
double dval = 3.14;
const int &ri = dval;
```

此处 ri 引用了一个 int 型的数。对 ri 的操作应该是整数运算，但 dval 却是一个双精度浮点数而非整数。因此为了确保让 ri 绑定一个整数，编译器把上述代码变成了如下形式：

```
const int temp = dval;      // 由双精度浮点数生成一个临时的整型常量
const int &ri = temp;       // 让 ri 绑定这个临时量
```

62

在这种情况下，ri 绑定了一个**临时量**（temporary）对象。所谓临时量对象就是当编译器需要一个空间来暂存表达式的求值结果时临时创建的一个未命名的对象。C++程序员们常常把临时量对象简称为临时量。

接下来探讨当 ri 不是常量时，如果执行了类似于上面的初始化过程将带来什么样的后果。如果 ri 不是常量，就允许对 ri 赋值，这样就会改变 ri 所引用对象的值。注意，此时绑定的对象是一个临时量而非 dval。程序员既然让 ri 引用 dval，就肯定想通过 ri 改变 dval 的值，否则干什么要给 ri 赋值呢？如此看来，既然大家基本上不会想着把引用绑定到临时量上，C++语言也就把这种行为归为非法。

对 const 的引用可能引用一个并非 const 的对象

必须认识到，常量引用仅对引用可参与的操作做出了限定，对于引用的对象本身是不是一个常量未作限定。因为对象也可能是个非常量，所以允许通过其他途径改变它的值：

```
int i = 42;
int &r1 = i;                    // 引用 ri 绑定对象 i
const int &r2 = i;              // r2 也绑定对象 i，但是不允许通过 r2 修改 i 的值
r1 = 0;                         // r1 并非常量，i 的值修改为 0
r2 = 0;                         // 错误：r2 是一个常量引用
```

r2 绑定（非常量）整数 i 是合法的行为。然而，不允许通过 r2 修改 i 的值。尽管如此，i 的值仍然允许通过其他途径修改，既可以直接给 i 赋值，也可以通过像 r1 一样绑定到 i 的其他引用来修改。

2.4.2 指针和 const

与引用一样，也可以令指针指向常量或非常量。类似于常量引用（参见 2.4.1 节，第 54 页），**指向常量的指针**（pointer to const）不能用于改变其所指对象的值。要想存放常量对象的地址，只能使用指向常量的指针：

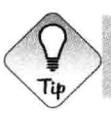
```
const double pi = 3.14;         // pi 是个常量，它的值不能改变
double *ptr = &pi;              // 错误：ptr 是一个普通指针
const double *cptr = &pi;       // 正确：cptr 可以指向一个双精度常量
*cptr = 42;                     // 错误：不能给*cptr 赋值
```

2.3.2 节（第 47 页）提到，指针的类型必须与其所指对象的类型一致，但是有两个例外。第一种例外情况是允许令一个指向常量的指针指向一个非常量对象：

```
double dval = 3.14;             // dval 是一个双精度浮点数，它的值可以改变
cptr = &dval;                   // 正确：但是不能通过 cptr 改变 dval 的值
```

63

和常量引用一样，指向常量的指针也没有规定其所指的对象必须是一个常量。所谓指向常量的指针仅仅要求不能通过该指针改变对象的值，而没有规定那个对象的值不能通过其他途径改变。



试试这样想吧：所谓指向常量的指针或引用，不过是指针或引用“自以为是”罢了，它们觉得自己指向了常量，所以自觉地不去改变所指对象的值。

const 指针

指针是对象而引用不是，因此就像其他对象类型一样，允许把指针本身定为常量。**常量指针**（const pointer）必须初始化，而且一旦初始化完成，则它的值（也就是存放在指针中的那个地址）就不能再改变了。把*放在 const 关键字之前用以说明指针是一个常量，这样的书写形式隐含着一层意味，即不变的是指针本身的值而非指向的那个值：

```
int errNumb = 0;
int *const curErr = &errNumb;     // curErr 将一直指向 errNumb
const double pi = 3.14159;
const double *const pip = &pi;    // pip 是一个指向常量对象的常量指针
```

如同 2.3.3 节（第 52 页）所讲的，要想弄清楚这些声明的含义最行之有效的办法是从右向左阅读。此例中，离 curErr 最近的符号是 const，意味着 curErr 本身是一个常量对象，对象的类型由声明符的其余部分确定。声明符中的下一个符号是*，意思是 curErr

是一个常量指针。最后，该声明语句的基本数据类型部分确定了常量指针指向的是一个 int 对象。与之相似，我们也能推断出，pip 是一个常量指针，它指向的对象是一个双精度浮点型常量。

指针本身是一个常量并不意味着不能通过指针修改其所指对象的值，能否这样做完全依赖于所指对象的类型。例如，pip 是一个指向常量的常量指针，则不论是 pip 所指的对象值还是 pip 自己存储的那个地址都不能改变。相反的，curErr 指向的是一个一般的非常量整数，那么就完全可以用 curErr 去修改 errNumb 的值：

```
*pip = 2.72;          // 错误：pip 是一个指向常量的指针
                      // 如果 curErr 所指的对象（也就是 errNumb）的值不为 0
if (*curErr) {
    errorHandler();
    *curErr = 0;      // 正确：把 curErr 所指的对象的值重置
}
```

2.4.2 节练习

练习 2.27：下面的哪些初始化是合法的？请说明原因。

```
(a) int i = -1, &r = 0;          (b) int *const p2 = &i2;
(c) const int i = -1, &r = 0;    (d) const int *const p3 = &i2;
(e) const int *p1 = &i2;         (f) const int &const r2;
(g) const int i2 = i, &r = i;
```

练习 2.28：说明下面的这些定义是什么意思，挑出其中不合法的。

```
(a) int i, *const cp;            (b) int *p1, *const p2;
(c) const int ic, &r = ic;       (d) const int *const p3;
(e) const int *p;
```

练习 2.29：假设已有上一个练习中定义的那些变量，则下面的哪些语句是合法的？请说明原因。

```
(a) i = ic;                      (b) p1 = p3;
(c) p1 = &ic;                    (d) p3 = &ic;
(e) p2 = p1;                     (f) ic = *p3;
```

2.4.3 顶层 const



如前所述，指针本身是一个对象，它又可以指向另外一个对象。因此，指针本身是不是常量以及指针所指的是不是一个常量就是两个相互独立的问题。用名词**顶层 const**（top-level const）表示指针本身是个常量，而用名词**底层 const**（low-level const）表示指针所指的对象是一个常量。

64

更一般的，顶层 const 可以表示任意的对象是常量，这一点对任何数据类型都适用，如算术类型、类、指针等。底层 const 则与指针和引用等复合类型的基本类型部分有关。比较特殊的是，指针类型既可以是顶层 const 也可以是底层 const，这一点和其他类型相比区别明显：

```
int i = 0;
int *const p1 = &i;          // 不能改变 p1 的值，这是一个顶层 const
const int ci = 42;           // 不能改变 ci 的值，这是一个顶层 const
const int *p2 = &ci;         // 允许改变 p2 的值，这是一个底层 const
```

```
const int *const p3 = p2; // 靠右的 const 是顶层 const，靠左的是底层 const
const int &r = ci;        // 用于声明引用的 const 都是底层 const
```



当执行对象的拷贝操作时，常量是顶层 const 还是底层 const 区别明显。其中，顶层 const 不受什么影响：

```
i = ci;      // 正确：拷贝 ci 的值，ci 是一个顶层 const，对此操作无影响
p2 = p3;     // 正确：p2 和 p3 指向的对象类型相同，p3 顶层 const 的部分不影响
```

执行拷贝操作并不会改变被拷贝对象的值，因此，拷入和拷出的对象是否是常量都没什么影响。

另一方面，底层 const 的限制却不能忽视。当执行对象的拷贝操作时，拷入和拷出的对象必须具有相同的底层 const 资格，或者两个对象的数据类型必须能够转换。一般来说，非常量可以转换成常量，反之则不行：

65

```
int *p = p3;              // 错误：p3 包含底层 const 的定义，而 p 没有
p2 = p3;                  // 正确：p2 和 p3 都是底层 const
p2 = &i;                  // 正确：int*能转换成 const int*
int &r = ci;              // 错误：普通的 int&不能绑定到 int 常量上
const int &r2 = i;        // 正确：const int&可以绑定到一个普通 int 上
```

p3 既是顶层 const 也是底层 const，拷贝 p3 时可以不在乎它是一个顶层 const，但是必须清楚它指向的对象得是一个常量。因此，不能用 p3 去初始化 p，因为 p 指向的是一个普通的（非常量）整数。另一方面，p3 的值可以赋给 p2，是因为这两个指针都是底层 const，尽管 p3 同时也是一个常量指针（顶层 const），仅就这次赋值而言不会有什么影响。

2.4.3 节练习

练习 2.30：对于下面的这些语句，请说明对象被声明成了顶层 const 还是底层 const？

```
const int v2 = 0;      int v1 = v2;
int *p1 = &v1, &r1 = v1;
const int *p2 = &v2, *const p3 = &i, &r2 = v2;
```

练习 2.31：假设已有上一个练习中所做的那些声明，则下面的哪些语句是合法的？请说明顶层 const 和底层 const 在每个例子中有何体现。

```
r1 = v2;
p1 = p2; p2 = p1;
p1 = p3; p2 = p3;
```



2.4.4 constexpr 和常量表达式

常量表达式（const expression）是指值不会改变并且在编译过程就能得到计算结果的表达式。显然，字面值属于常量表达式，用常量表达式初始化的 const 对象也是常量表达式。后面将会提到，C++语言中有几种情况下是要用到常量表达式的。

一个对象（或表达式）是不是常量表达式由它的数据类型和初始值共同决定，例如：

```
const int max_files = 20;         // max_files 是常量表达式
const int limit = max_files + 1;  // limit 是常量表达式
int staff_size = 27;              // staff_size 不是常量表达式
```

```
const int sz = get_size();          // sz 不是常量表达式
```

尽管 staff_size 的初始值是个字面值常量，但由于它的数据类型只是一个普通 int 而非 const int，所以它不属于常量表达式。另一方面，尽管 sz 本身是一个常量，但它的具体值直到运行时才能获取到，所以也不是常量表达式。

constexpr 变量

在一个复杂系统中，很难（几乎肯定不能）分辨一个初始值到底是不是常量表达式。 66
当然可以定义一个 const 变量并把它的初始值设为我们认为的某个常量表达式，但在实际使用时，尽管要求如此却常常发现初始值并非常量表达式的情况。可以这么说，在此种情况下，对象的定义和使用根本就是两回事儿。

C++11 新标准规定，允许将变量声明为 **constexpr** 类型以便由编译器来验证变量的值是否是一个常量表达式。声明为 constexpr 的变量一定是一个常量，而且必须用常量表达式初始化： C++11

```
constexpr int mf = 20;              // 20 是常量表达式
constexpr int limit = mf + 1;       // mf + 1 是常量表达式
constexpr int sz = size();          // 只有当 size 是一个 constexpr 函数时
                                    // 才是一条正确的声明语句
```

尽管不能使用普通函数作为 constexpr 变量的初始值，但是正如 6.5.2 节（第 214 页）将要介绍的，新标准允许定义一种特殊的 constexpr 函数。这种函数应该足够简单以使得编译时就可以计算其结果，这样就能用 constexpr 函数去初始化 constexpr 变量了。



一般来说，如果你认定变量是一个常量表达式，那就把它声明成 constexpr 类型。

字面值类型

常量表达式的值需要在编译时就得到计算，因此对声明 constexpr 时用到的类型必须有所限制。因为这些类型一般比较简单，值也显而易见、容易得到，就把它们称为“字面值类型”（literal type）。

到目前为止接触过的数据类型中，算术类型、引用和指针都属于字面值类型。自定义类 Sales_item、IO 库、string 类型则不属于字面值类型，也就不能被定义成 constexpr。其他一些字面值类型将在 7.5.6 节（第 267 页）和 19.3 节（第 736 页）介绍。

尽管指针和引用都能定义成 constexpr，但它们的初始值却受到严格限制。一个 constexpr 指针的初始值必须是 nullptr 或者 0，或者是存储于某个固定地址中的对象。

6.1.1 节（第 184 页）将要提到，函数体内定义的变量一般来说并非存放在固定地址中，因此 constexpr 指针不能指向这样的变量。相反的，定义于所有函数体之外的对象其地址固定不变，能用来初始化 constexpr 指针。同样是在 6.1.1 节（第 185 页）中还将提到，允许函数定义一类有效范围超出函数本身的变量，这类变量和定义在函数体之外的变量一样也有固定地址。因此，constexpr 引用能绑定到这样的变量上，constexpr 指针也能指向这样的变量。

指针和 constexpr

67

必须明确一点，在 constexpr 声明中如果定义了一个指针，限定符 constexpr 仅

对指针有效，与指针所指的对象无关：

```
const int *p = nullptr;        // p是一个指向整型常量的指针
constexpr int *q = nullptr;    // q是一个指向整数的常量指针
```

p和q的类型相差甚远，p是一个指向常量的指针，而q是一个常量指针，其中的关键在于constexpr把它所定义的对象置为了顶层const（参见2.4.3节，第57页）。

与其他常量指针类似，constexpr指针既可以指向常量也可以指向一个非常量：

```
constexpr int *np = nullptr; // np是一个指向整数的常量指针，其值为空
int j = 0;
constexpr int i = 42;        // i的类型是整型常量
// i和j都必须定义在函数体之外
constexpr const int *p = &i; // p是常量指针，指向整型常量i
constexpr int *p1 = &j;      // p1是常量指针，指向整数j
```

2.4.4 节练习

练习2.32：下面的代码是否合法？如果非法，请设法将其修改正确。

```
int null = 0, *p = null;
```

2.5 处理类型

随着程序越来越复杂，程序中用到的类型也越来越复杂，这种复杂性体现在两个方面。一是一些类型难于“拼写”，它们的名字既难记又容易写错，还无法明确体现其真实目的和含义。二是有时候根本搞不清到底需要的类型是什么，程序员不得不回过头去从程序的上下文中寻求帮助。

2.5.1 类型别名

类型别名（type alias）是一个名字，它是某种类型的同义词。使用类型别名有很多好处，它让复杂的类型名字变得简单明了、易于理解和使用，还有助于程序员清楚地知道使用该类型的真实目的。

有两种方法可用于定义类型别名。传统的方法是使用关键字**typedef**：

```
typedef double wages;     //wages是double的同义词
typedef wages base, *p;   //base是double的同义词，p是double*的同义词
```

68 其中，关键字typedef作为声明语句中的基本数据类型（参见2.3节，第45页）的一部分出现。含有typedef的声明语句定义的不再是变量而是类型别名。和以前的声明语句一样，这里的声明符也可以包含类型修饰，从而也能由基本数据类型构造出复合类型来。

C++ 11

新标准规定了一种新的方法，使用**别名声明**（alias declaration）来定义类型的别名：

```
using SI = Sales_item;    // SI是Sales_item的同义词
```

这种方法用关键字using作为别名声明的开始，其后紧跟别名和等号，其作用是把等号左侧的名字规定成等号右侧类型的别名。

类型别名和类型的名字等价，只要是类型的名字能出现的地方，就能使用类型别名：

```
wages hourly, weekly;          // 等价于double hourly、weekly;
```

```
SI item;                          // 等价于 Sales_item item
```

指针、常量和类型别名



如果某个类型别名指代的是复合类型或常量，那么把它用到声明语句里就会产生意想不到的后果。例如下面的声明语句用到了类型 pstring，它实际上是类型 char*的别名：

```
typedef char *pstring;
const pstring cstr = 0;   // cstr 是指向 char 的常量指针
const pstring *ps;        // ps 是一个指针，它的对象是指向 char 的常量指针
```

上述两条声明语句的基本数据类型都是 const pstring，和过去一样，const 是对给定类型的修饰。pstring 实际上是指向 char 的指针，因此，const pstring 就是指向 char 的常量指针，而非指向常量字符的指针。

遇到一条使用了类型别名的声明语句时，人们往往会错误地尝试把类型别名替换成它本来的样子，以理解该语句的含义：

```
const char *cstr = 0;     // 是对 const pstring cstr 的错误理解
```

再强调一遍：这种理解是错误的。声明语句中用到 pstring 时，其基本数据类型是指针。可是用 char*重写了声明语句后，数据类型就变成了 char，*成为了声明符的一部分。这样改写的结果是，const char 成了基本数据类型。前后两种声明含义截然不同，前者声明了一个指向 char 的常量指针，改写后的形式则声明了一个指向 const char 的指针。

2.5.2 auto 类型说明符



编程时常常需要把表达式的值赋给变量，这就要求在声明变量的时候清楚地知道表达式的类型。然而要做到这一点并非那么容易，有时甚至根本做不到。为了解决这个问题，C++11 新标准引入了 **auto** 类型说明符，用它就能让编译器替我们去分析表达式所属的类型。和原来那些只对应一种特定类型的说明符（比如 double）不同，auto 让编译器通过初始值来推算变量的类型。显然，auto 定义的变量必须有初始值：

C++ 11 69

```
// 由 val1 和 val2 相加的结果可以推断出 item 的类型
auto item = val1 + val2; // item 初始化为 val1 和 val2 相加的结果
```

此处编译器将根据 val1 和 val2 相加的结果来推断 item 的类型。如果 val1 和 val2 是类 Sales_item（参见 1.5 节，第 17 页）的对象，则 item 的类型就是 Sales_item；如果这两个变量的类型是 double，则 item 的类型就是 double，以此类推。

使用 auto 也能在一条语句中声明多个变量。因为一条声明语句只能有一个基本数据类型，所以该语句中所有变量的初始基本数据类型都必须一样：

```
auto i = 0, *p = &i;      // 正确：i 是整数、p 是整型指针
auto sz = 0, pi = 3.14;   // 错误：sz 和 pi 的类型不一致
```

复合类型、常量和 auto

编译器推断出来的 auto 类型有时候和初始值的类型并不完全一样，编译器会适当地改变结果类型使其更符合初始化规则。

首先，正如我们所熟知的，使用引用其实是使用引用的对象，特别是当引用被用作初始值时，真正参与初始化的其实是引用对象的值。此时编译器以引用对象的类型作为 auto 的类型：

```
int i = 0, &r = i;
auto a = r;              // a是一个整数(r是i的别名,而i是一个整数)
```

其次，auto一般会忽略掉顶层const（参见2.4.3节，第57页），同时底层const则会保留下来，比如当初始值是一个指向常量的指针时：

```
const int ci = i, &cr = ci;
auto b = ci;    // b是一个整数(ci的顶层const特性被忽略掉了)
auto c = cr;    // c是一个整数(cr是ci的别名,ci本身是一个顶层const)
auto d = &i;    // d是一个整型指针(整数的地址就是指向整数的指针)
auto e = &ci;   // e是一个指向整数常量的指针(对常量对象取地址是一种底层const)
```

如果希望推断出的auto类型是一个顶层const，需要明确指出：

```
const auto f = ci;          // ci的推演类型是int,f是const int
```

还可以将引用的类型设为auto，此时原来的初始化规则仍然适用：

```
auto &g = ci;               // g是一个整型常量引用,绑定到ci
auto &h = 42;               // 错误:不能为非常量引用绑定字面值
const auto &j = 42;         // 正确:可以为常量引用绑定字面值
```

70 设置一个类型为auto的引用时，初始值中的顶层常量属性仍然保留。和往常一样，如果我们给初始值绑定一个引用，则此时的常量就不是顶层常量了。

要在一条语句中定义多个变量，切记，符号&和*只从属于某个声明符，而非基本数据类型的一部分，因此初始值必须是同一种类型：

```
auto k = ci, &l = i;        // k是整数,l是整型引用
auto &m = ci, *p = &ci;     // m是对整型常量的引用,p是指向整型常量的指针
// 错误:i的类型是int而&ci的类型是const int
auto &n = i, *p2 = &ci;
```

2.5.2 节练习

练习2.33：利用本节定义的变量，判断下列语句的运行结果。

```
a = 42; b = 42; c = 42;
d = 42; e = 42; g = 42;
```

练习2.34：基于上一个练习中的变量和语句编写一段程序，输出赋值前后变量的内容，你刚才的推断正确吗？如果不对，请反复研读本节的示例直到你明白错在何处为止。

练习2.35：判断下列定义推断出的类型是什么，然后编写程序进行验证。

```
const int i = 42;
auto j = i; const auto &k = i; auto *p = &i;
const auto j2 = i, &k2 = i;
```



2.5.3 decltype类型指示符

C++11 有时会遇到这种情况：希望从表达式的类型推断出要定义的变量的类型，但是不想用该表达式的值初始化变量。为了满足这一要求，C++11新标准引入了第二种类型说明符**decltype**，它的作用是选择并返回操作数的数据类型。在此过程中，编译器分析表达式并得到它的类型，却不实际计算表达式的值：

```
decltype(f()) sum = x;    // sum的类型就是函数f的返回类型
```

编译器并不实际调用函数 f，而是使用当调用发生时 f 的返回值类型作为 sum 的类型。换句话说，编译器为 sum 指定的类型是什么呢？就是假如 f 被调用的话将会返回的那个类型。

decltype 处理顶层 const 和引用的方式与 auto 有些许不同。如果 decltype 使用的表达式是一个变量，则 decltype 返回该变量的类型（包括顶层 const 和引用在内）：

71

```
const int ci = 0, &cj = ci;
decltype(ci) x = 0;       // x 的类型是 const int
decltype(cj) y = x;       // y 的类型是 const int&，y 绑定到变量 x
decltype(cj) z;           // 错误：z 是一个引用，必须初始化
```

因为 cj 是一个引用，decltype(cj)的结果就是引用类型，因此作为引用的 z 必须被初始化。

需要指出的是，引用从来都作为其所指对象的同义词出现，只有用在 decltype 处是一个例外。

decltype 和引用

如果 decltype 使用的表达式不是一个变量，则 decltype 返回表达式结果对应的类型。如 4.1.1 节（第 120 页）将要介绍的，有些表达式将向 decltype 返回一个引用类型。一般来说当这种情况发生时，意味着该表达式的结果对象能作为一条赋值语句的左值：

```
// decltype 的结果可以是引用类型
int i = 42, *p = &i, &r = i;
decltype(r + 0) b;    // 正确：加法的结果是 int，因此 b 是一个（未初始化的）int
decltype(*p) c;       // 错误：c 是 int&，必须初始化
```

因为 r 是一个引用，因此 decltype(r)的结果是引用类型。如果想让结果类型是 r 所指的类型，可以把 r 作为表达式的一部分，如 r+0，显然这个表达式的结果将是一个具体值而非一个引用。

另一方面，如果表达式的内容是解引用操作，则 decltype 将得到引用类型。正如我们所熟悉的那样，解引用指针可以得到指针所指的对象，而且还能给这个对象赋值。因此，decltype(*p)的结果类型就是 int&，而非 int。



decltype 和 auto 的另一处重要区别是，decltype 的结果类型与表达式形式密切相关。有一种情况需要特别注意：对于 decltype 所用的表达式来说，如果变量名加上了一对括号，则得到的类型与不加括号时会有不同。如果 decltype 使用的是一个不加括号的变量，则得到的结果就是该变量的类型；如果给变量加上了一层或多层括号，编译器就会把它当成是一个表达式。变量是一种可以作为赋值语句左值的特殊表达式，所以这样的 decltype 就会得到引用类型：

```
// decltype 的表达式如果是加上了括号的变量，结果将是引用
decltype((i)) d;      // 错误：d 是 int&，必须初始化
decltype(i) e;        // 正确：e 是一个（未初始化的）int
```



切记：decltype((*variable*))（注意是双层括号）的结果永远是引用，而 decltype(*variable*)结果只有当 *variable* 本身就是一个引用时才是引用。

72

2.5.3 节练习

练习 2.36：关于下面的代码，请指出每一个变量的类型以及程序结束时它们各自的值。

```
int a = 3, b = 4;
decltype(a) c = a;
decltype((b)) d = a;
++c;
++d;
```

练习 2.37：赋值是会产生引用的一类典型表达式，引用的类型就是左值的类型。也就是说，如果 i 是 int，则表达式 i=x 的类型是 int&。根据这一特点，请指出下面的代码中每一个变量的类型和值。

```
int a = 3, b = 4;
decltype(a) c = a;
decltype(a = b) d = a;
```

练习 2.38：说明由 decltype 指定类型和由 auto 指定类型有何区别。请举出一个例子，decltype 指定的类型与 auto 指定的类型一样；再举一个例子，decltype 指定的类型与 auto 指定的类型不一样。



2.6 自定义数据结构

从最基本的层面理解，数据结构是把一组相关的数据元素组织起来然后使用它们的策略和方法。举一个例子，我们的 Sales_item 类把书本的 ISBN 编号、售出量及销售收入等数据组织在了一起，并且提供诸如 isbn 函数、>>、<<、+、+=等运算在内的一系列操作，Sales_item 类就是一个数据结构。

C++语言允许用户以类的形式自定义数据类型，而库类型 string、istream、ostream 等也都是以类的形式定义的，就像第 1 章的 Sales_item 类型一样。C++语言对类的支持甚多，事实上本书的第III部分和第IV部分都将大篇幅地介绍与类有关的知识。尽管 Sales_item 类非常简单，但是要想给出它的完整定义可在第 14 章介绍自定义运算符之后。



2.6.1 定义 Sales_data 类型

尽管我们还写不出完整的 Sales_item 类，但是可以尝试着把那些数据元素组织到一起形成一个简单点儿的类。初步的想法是用户能直接访问其中的数据元素，也能实现一些基本的操作。

既然我们筹划的这个数据结构不带有任何运算功能，不妨把它命名为 Sales_data 以示与 Sales_item 的区别。Sales_data 初步定义如下：

73

```
struct Sales_data {
    std::string bookNo;
    unsigned units_sold = 0;
    double revenue = 0.0;
};
```

我们的类以关键字 **struct** 开始，紧跟着类名和类体（其中类体部分可以为空）。类体由

花括号包围形成了一个新的作用域（参见 2.2.4 节，第 43 页）。类内部定义的名字必须唯一，但是可以与类外部定义的名字重复。

类体右侧的表示结束的花括号后必须写一个分号，这是因为类体后面可以紧跟变量名以示对该类型对象的定义，所以分号必不可少：

```
struct Sales_data { /* ... */ } accum, trans, *salesptr;
// 与上一条语句等价，但可能更好一些
struct Sales_data { /* ... */ };
Sales_data accum, trans, *salesptr;
```

分号表示声明符（通常为空）的结束。一般来说，最好不要把对象的定义和类的定义放在一起。这么做无异于把两种不同实体的定义混在了一条语句里，一会儿定义类，一会儿又定义变量，显然这是一种不被建议的行为。



很多新手程序员经常忘了在类定义的最后加上分号。

类数据成员

类体定义类的**成员**，我们的类只有**数据成员**（data member）。类的数据成员定义了类的对象的具体内容，每个对象有自己的一份数据成员拷贝。修改一个对象的数据成员，不会影响其他 Sales_data 的对象。

定义数据成员的方法和定义普通变量一样：首先说明一个基本类型，随后紧跟一个或多个声明符。我们的类有 3 个数据成员：一个名为 bookNo 的 string 成员、一个名为 units_sold 的 unsigned 成员和一个名为 revenue 的 double 成员。每个 Sales_data 的对象都将包括这 3 个数据成员。

C++11 新标准规定，可以为数据成员提供一个**类内初始值**（in-class initializer）。创建对象时，类内初始值将用于初始化数据成员。没有初始值的成员将被默认初始化（参见 2.2.1 节，第 40 页）。因此当定义 Sales_data 的对象时，units_sold 和 revenue 都将初始化为 0，bookNo 将初始化为空字符串。



对类内初始值的限制与之前（参见 2.2.1 节，第 39 页）介绍的类似：或者放在花括号里，或者放在等号右边，记住不能使用圆括号。

7.2 节（第 240 页）将要介绍，用户可以使用 C++语言提供的另外一个关键字 class 来定义自己的数据结构，到时也将说明现在我们使用 struct 的原因。在第 7 章学习与 class 有关的知识之前，建议读者继续使用 struct 定义自己的数据类型。

2.6.1 节练习

74

练习 2.39：编译下面的程序观察其运行结果，注意，如果忘记写类定义体后面的分号会发生什么情况？记录下相关信息，以后可能会有用。

```
struct Foo { /* 此处为空 */ } // 注意：没有分号
int main()
{
    return 0;
}
```

练习 2.40：根据自己的理解写出 Sales_data 类，最好与书中的例子有所区别。



2.6.2 使用 Sales_data 类

和 Sales_item 类不同的是，我们自定义的 Sales_data 类没有提供任何操作，Sales_data 类的使用者如果想执行什么操作就必须自己动手实现。例如，我们将参照 1.5.2 节（第 20 页）的例子写一段程序实现求两次交易相加结果的功能。程序的输入是下面这两条交易记录：

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

每笔交易记录着图书的 ISBN 编号、售出数量和售出单价。

添加两个 Sales_data 对象

因为 Sales_data 类没有提供任何操作，所以我们必须自己编码实现输入、输出和相加的功能。假设已知 Sales_data 类定义于 Sales_data.h 文件内，2.6.3 节（第 67 页）将详细介绍定义头文件的方法。

因为程序比较长，所以接下来分成几部分介绍。总的来说，程序的结构如下：

```
#include <iostream>
#include <string>
#include "Sales_data.h"
int main()
{
    Sales_data data1, data2;
    // 读入 data1 和 data2 的代码
    // 检查 data1 和 data2 的 ISBN 是否相同的代码
    // 如果相同，求 data1 和 data2 的总和
}
```

和原来的程序一样，先把所需的头文件包含进来并且定义变量用于接受输入。和 Sales_item 类不同的是，新程序还包含了 string 头文件，因为我们的代码中将用到 string 类型的成员变量 bookNo。

75 Sales_data 对象读入数据

第 3 章和第 10 章将详细介绍 string 类型的细节，在此之前，我们先了解一点儿关于 string 的知识以便定义和使用我们的 ISBN 成员。string 类型其实就是字符的序列，它的操作有>>、<<和==等，功能分别是读入字符串、写出字符串和比较字符串。这样我们就能书写代码读入第一笔交易了：

```
double price = 0; // 书的单价，用于计算销售收入
// 读入第 1 笔交易： ISBN、销售数量、单价
std::cin >> data1.bookNo >> data1.units_sold >> price;
// 计算销售收入
data1.revenue = data1.units_sold * price;
```

交易信息记录的是书售出的单价，而数据结构存储的是一次交易的销售收入，因此需要将单价读入到 double 变量 price，然后再计算销售收入 revenue。输入语句

```
std::cin >> data1.bookNo >> data1.units_sold >> price;
```

使用点操作符（参见 1.5.2 节，第 20 页）读入对象 data1 的 bookNo 成员和 units_sold 成员。

最后一条语句把 data1.units_sold 和 price 的乘积赋值给 data1 的 revenue 成员。

接下来程序重复上述过程读入对象 data2 的数据：

```
// 读入第 2 笔交易
std::cin >> data2.bookNo >> data2.units_sold >> price;
data2.revenue = data2.units_sold * price;
```

输出两个 Sales_data 对象的和

剩下的工作就是检查两笔交易涉及的 ISBN 编号是否相同了。如果相同输出它们的和，否则输出一条报错信息：

```
if (data1.bookNo == data2.bookNo) {
    unsigned totalCnt = data1.units_sold + data2.units_sold;
    double totalRevenue = data1.revenue + data2.revenue;
    // 输出：ISBN、总销售量、总销售额、平均价格
    std::cout << data1.bookNo << " " << totalCnt
              << " " << totalRevenue << " ";
    if (totalCnt != 0)
        std::cout << totalRevenue/totalCnt << std::endl;
    else
        std::cout << "(no sales)" << std::endl;
    return 0;          // 标示成功
} else {               // 两笔交易的 ISBN 不一样
    std::cerr << "Data must refer to the same ISBN"
              << std::endl;
    return -1;         // 标示失败
}
```

在第一个 if 语句中比较了 data1 和 data2 的 bookNo 成员是否相同。如果相同则执行第一个 if 语句花括号内的操作，首先计算 units_sold 的和并赋给变量 totalCnt，然后计算 revenue 的和并赋给变量 totalRevenue，输出这些值。接下来检查是否确实售出了书籍，如果是，计算并输出每本书的平均价格；如果售量为零，输出一条相应的信息。 76

2.6.2 节练习

练习 2.41：使用你自己的 Sales_data 类重写 1.5.1 节（第 20 页）、1.5.2 节（第 21 页）和 1.6 节（第 22 页）的练习。眼下先把 Sales_data 类的定义和 main 函数放在同一个文件里。

2.6.3 编写自己的头文件



尽管如 19.7 节（第 754 页）所讲可以在函数体内定义类，但是这样的类毕竟受到了一些限制。所以，类一般都不定义在函数体内。当在函数体外部定义类时，在各个指定的源文件中可能只有一处该类的定义。而且，如果要在不同文件中使用同一个类，类的定义就必须保持一致。

为了确保各个文件中类的定义一致，类通常被定义在头文件中，而且类所在头文件的名字应与类的名字一样。例如，库类型 string 在名为 string 的头文件中定义。又如，我们应该把 Sales_data 类定义在名为 Sales_data.h 的头文件中。

头文件通常包含那些只能被定义一次的实体，如类、const 和 constexpr 变量（参见 2.4 节，第 54 页）等。头文件也经常用到其他头文件的功能。例如，我们的 Sales_data 类包含有一个 string 成员，所以 Sales_data.h 必须包含 string.h 头文件。同时，使用 Sales_data 类的程序为了能操作 bookNo 成员需要再一次包含 string.h 头文件。

这样，事实上使用 Sales_data 类的程序就先后两次包含了 string.h 头文件：一次是直接包含的，另有一次是随着包含 Sales_data.h 被隐式地包含进来的。有必要在书写头文件时做适当处理，使其遇到多次包含的情况也能安全和正常地工作。



头文件一旦改变，相关的源文件必须重新编译以获取更新过的声明。

预处理器概述

确保头文件多次包含仍能安全工作的常用技术是**预处理器**（preprocessor），它由 C++
77 语言从 C 语言继承而来。预处理器是在编译之前执行的一段程序，可以部分地改变我们所写的程序。之前已经用到了一项预处理功能#include，当预处理器看到#include 标记时就会用指定的头文件的内容代替#include。

C++程序还会用到的一项预处理功能是**头文件保护符**（header guard），头文件保护符依赖于预处理变量（参见 2.3.2 节，第 48 页）。预处理变量有两种状态：已定义和未定义。**#define** 指令把一个名字设定为预处理变量，另外两个指令则分别检查某个指定的预处理变量是否已经定义：**#ifdef** 当且仅当变量已定义时为真，**#ifndef** 当且仅当变量未定义时为真。一旦检查结果为真，则执行后续操作直至遇到**#endif** 指令为止。

使用这些功能就能有效地防止重复包含的发生：

```
#ifndef SALES_DATA_H
#define SALES_DATA_H
#include <string>
struct Sales_data {
    std::string bookNo;
    unsigned units_sold = 0;
    double revenue = 0.0;
};
#endif
```

第一次包含 Sales_data.h 时，#ifndef 的检查结果为真，预处理器将顺序执行后面的操作直至遇到#endif 为止。此时，预处理变量 SALES_DATA_H 的值将变为已定义，而且 Sales_data.h 也会被拷贝到我们的程序中来。后面如果再一次包含 Sales_data.h，则#ifndef 的检查结果将为假，编译器将忽略#ifndef 到#endif 之间的部分。



预处理变量无视 C++语言中关于作用域的规则。

整个程序中的预处理变量包括头文件保护符必须唯一，通常的做法是基于头文件中类的名字来构建保护符的名字，以确保其唯一性。为了避免与程序中的其他实体发生名字冲突，一般把预处理变量的名字全部大写。



头文件即使（目前还）没有被包含在任何其他头文件中，也应该设置保护符。头文件保护符很简单，程序员只要习惯性地加上就可以了，没必要太在乎你的程序到底需不需要。

2.6.3 节练习

练习 2.42：根据你自己的理解重写一个 Sales_data.h 头文件，并以此为基础重做 2.6.2 节（第 67 页）的练习。

小结

78

类型是C++编程的基础。

类型规定了其对象的存储要求和所能执行的操作。C++语言提供了一套基础内置类型，如 int 和 char 等，这些类型与实现它们的机器硬件密切相关。类型分为非常量和常量，一个常量对象必须初始化，而且一旦初始化其值就不能再改变。此外，还可以定义复合类型，如指针和引用等。复合类型的定义以其他类型为基础。

C++语言允许用户以类的形式自定义类型。C++库通过类提供了一套高级抽象类型，如输入输出和 string 等。

术语表

地址（address）是一个数字，根据它可以找到内存中的一个字节。

别名声明（alias declaration） 为另外一种类型定义一个同义词：使用“名字=类型”的格式将名字作为该类型的同义词。

算术类型（arithmetic type） 布尔值、字符、整数、浮点数等内置类型。

数组（array）是一种数据结构，存放着一组未命名的对象，可以通过索引来访问这些对象。3.5 节将详细介绍数组的知识。

auto 是一个类型说明符，通过变量的初始值来推断变量的类型。

基本类型（base type）是类型说明符，可用 const 修饰，在声明语句中位于声明符之前。基本类型提供了最常见的数据类型，以此为基础构建声明符。

绑定（bind）令某个名字与给定的实体关联在一起，使用该名字也就是使用该实体。例如，引用就是将某个名字与某个对象绑定在一起。

字节（byte）内存中可寻址的最小单元，大多数机器的字节占 8 位。

类成员（class member）类的组成部分。

复合类型（compound type）是一种类型，它的定义以其他类型为基础。

const 是一种类型修饰符，用于说明永不改变的对象。const 对象一旦定义就无法再赋新值，所以必须初始化。

常量指针（const pointer）是一种指针，它的值永不改变。

常量引用（const reference）是一种习惯叫法，含义是指向常量的引用。

常量表达式（const expression）能在编译时计算并获取结果的表达式。

constexpr 是一种函数，用于代表一条常量表达式。6.5.2 节（第 214 页）将介绍 constexpr 函数。

转换（conversion）一种类型的值转变成另外一种类型值的过程。C++语言支持内置类型之间的转换。

数据成员（data member）组成对象的数据元素，类的每个对象都有类的数据成员的一份拷贝。数据成员可以在类内部声明的同时初始化。

声明（declaration）声称存在一个变量、函数或是别处定义的类型。名字必须在定义或声明之后才能使用。 79

声明符（declarator）是声明的一部分，包括被定义的名字和类型修饰符，其中类型修饰符可以有也可以没有。

decltype 是一个类型说明符，从变量或表达式推断得到类型。

默认初始化（default initialization）当对象未被显式地赋予初始值时执行的初始化行

为。由类本身负责执行的类对象的初始化行为。全局作用域的内置类型对象初始化为 0；局部作用域的对象未被初始化即拥有未定义的值。

定义（definition）为某一特定类型的变量申请存储空间，可以选择初始化该变量。名字必须在定义或声明之后才能使用。

转义序列（escape sequence）字符特别是那些不可打印字符的替代形式。转义以反斜线开头，后面紧跟一个字符，或者不多于 3 个八进制数字，或者字母 x 加上 1 个十六进制数。

全局作用域（global scope）位于其他所有作用域之外的作用域。

头文件保护符（header guard）使用预处理变量以防止头文件被某个文件重复包含。

标识符（identifier）组成名字的字符序列，标识符对大小写敏感。

类内初始值（in-class initializer）在声明类的数据成员时同时提供的初始值，必须置于等号右侧或花括号内。

在作用域内（in scope）名字在当前作用域内可见。

被初始化（initialized）变量在定义的同时被赋予初始值，变量一般都应该被初始化。

内层作用域（inner scope）嵌套在其他作用域之内的作用域。

整型（integral type）参见算术类型。

列表初始化（list initialization）利用花括号把一个或多个初始值放在一起的初始化形式。

字面值（literal）是一个不能改变的值，如数字、字符、字符串等。单引号内的是字符字面值，双引号内的是字符串字面值。

局部作用域（local scope）是块作用域的习惯叫法。

底层 const（low-level const）一个不属于顶层的 `const`，类型如果由底层常量定义，则不能被忽略。

成员（member）类的组成部分。

不可打印字符（nonprintable character）不具有可见形式的字符，如控制符、退格、换行符等。

空指针（null pointer）值为 0 的指针，空指针合法但是不指向任何对象。

nullptr 是表示空指针的字面值常量。

对象（object）是内存的一块区域，具有某种类型，变量是命名了的对象。

外层作用域（outer scope）嵌套着别的作用域的作用域。

指针（pointer）是一个对象，存放着某个对象的地址，或者某个对象存储区域之后的下一地址，或者 0。

指向常量的指针（pointer to const）是一个指针，存放着某个常量对象的地址。指向常量的指针不能用来改变它所指对象的值。

预处理器（preprocessor）在 C++编译过程中执行的一段程序。

预处理变量（preprocessor variable）由预处理器管理的变量。在程序编译之前，预处理器负责将程序中的预处理变量替换成它的真实值。

引用（reference）是某个对象的别名。

80 **对常量的引用**（reference to const）是一个引用，不能用来改变它所绑定对象的值。对常量的引用可以绑定常量对象，或者非常量对象，或者表达式的结果。

作用域（scope）是程序的一部分，在其中某些名字有意义。C++有几级作用域：

全局（global）——名字定义在所有其他作用域之外。

类（class）——名字定义在类内部。

命名空间（namespace）——名字定义在命名空间内部。

块（block）——名字定义在块内部。

名字从声明位置开始直至声明语句所在的作用域末端为止都是可用的。

分离式编译（separate compilation）把程序分割为多个单独文件的能力。

带符号类型（signed）保存正数、负数或 0 的整型。

字符串（string）是一种库类型，表示可变长字符序列。

struct 是一个关键字，用于定义类。

临时值（temporary）编译器在计算表达式结果时创建的无名对象。为某表达式创建了一个临时值，则此临时值将一直存在直到包含有该表达式的最大的表达式计算完成为止。

顶层 const（top-level const）是一个 `const`，规定某对象的值不能改变。

类型别名（type alias）是一个名字，是另外一个类型的同义词，通过关键字 `typedef` 或别名声明语句来定义。

类型检查（type checking）是一个过程，编译器检查程序使用某给定类型对象的方式与该类型的定义是否一致。

类型说明符（type specifier）类型的名字。

typedef 为某类型定义一个别名。当关键字 `typedef` 作为声明的基本类型出现时，声明中定义的名字就是类型名。

未定义（undefined）即 C++语言没有明确规定的情况。不论是否有意为之，未定义行为都可能引发难以追踪的运行时错误、安全问题和可移植性问题。

未初始化（uninitialized）变量已定义但未被赋予初始值。一般来说，试图访问未初始化变量的值将引发未定义行为

无符号类型（unsigned）保存大于等于 0 的整型。

变量（variable）命名的对象或引用。C++语言要求变量要先声明后使用。

void* 可以指向任意非常量的指针类型，不能执行解引用操作。

void 类型 是一种有特殊用处的类型，既无操作也无值。不能定义一个 `void` 类型的变量。

字（word）在指定机器上进行整数运算的自然单位。一般来说，字的空间足够存放地址。32 位机器上的字通常占据 4 个字节。

&运算符（& operator）取地址运算符。

***运算符**（* operator）解引用运算符。解引用一个指针将返回该指针所指的对象，为解引用的结果赋值也就是为指针所指的对象赋值。

#define 是一条预处理指令，用于定义一个预处理变量。

#endif 是一条预处理指令，用于结束一个 `#ifdef` 或`#ifndef` 区域。

#ifdef 是一条预处理指令，用于判断给定的变量是否已经定义。

#ifndef 是一条预处理指令，用于判断给定的变量是否尚未定义。

第 3 章
字符串、向量和数组

内容

3.1 命名空间的 using 声明 74
3.2 标准库类型 string 75
3.3 标准库类型 vector 86
3.4 迭代器介绍 95
3.5 数组 101
3.6 多维数组 112
小结 117
术语表 117

除了第 2 章介绍的内置类型之外，C++语言还定义了一个内容丰富的抽象数据类型库。其中，`string` 和 `vector` 是两种最重要的标准库类型，前者支持可变长字符串，后者则表示可变长的集合。还有一种标准库类型是迭代器，它是 `string` 和 `vector` 的配套类型，常被用于访问 `string` 中的字符或 `vector` 中的元素。

内置数组是一种更基础的类型，`string` 和 `vector` 都是对它的某种抽象。本章将分别介绍数组以及标准库类型 `string` 和 `vector`。

82 第2章介绍的内置类型是由C++语言直接定义的。这些类型，比如数字和字符，体现了大多数计算机硬件本身具备的能力。标准库定义了另外一组具有更高级性质的类型，它们尚未直接实现到计算机硬件中。

本章将介绍两种最重要的标准库类型：string和vector。string表示可变长的字符序列，vector存放的是某种给定类型对象的可变长序列。本章还将介绍内置数组类型，和其他内置类型一样，数组的实现与硬件密切相关。因此相较于标准库类型string和vector，数组在灵活性上稍显不足。

在开始介绍标准库类型之前，先来学习一种访问库中名字的简单方法。



3.1 命名空间的using声明

目前为止，我们用到的库函数基本上都属于命名空间std，而程序也显式地将这一点标示了出来。例如，std::cin表示从标准输入中读取内容。此处使用作用域操作符（::）（参见1.2节，第7页）的含义是：编译器应从操作符左侧名字所示的作用域中寻找右侧那个名字。因此，std::cin的意思就是要使用命名空间std中的名字cin。

上面的方法显得比较烦琐，然而幸运的是，通过更简单的途径也能使用到命名空间中的成员。本节将学习其中一种最安全的方法，也就是使用**using声明**（using declaration），18.2.2节（第702页）会介绍另一种方法。

有了using声明就无须专门的前缀（形如命名空间::）也能使用所需的名字了。using声明具有如下的形式：

```
using namespace::name;
```

一旦声明了上述语句，就可以直接访问命名空间中的名字：

```
#include <iostream>
// using声明，当我们使用名字cin时，从命名空间std中获取它
using std::cin;

int main()
{
    int i;
    cin >> i;          // 正确：cin和std::cin含义相同
    cout << i;         // 错误：没有对应的using声明，必须使用完整的名字
    std::cout << i;    // 正确：显式地从std中使用cout
    return 0;
}
```

每个名字都需要独立的using声明

按照规定，每个using声明引入命名空间中的一个成员。例如，可以把要用到的标准库中的名字都以using声明的形式表示出来，重写1.2节（第5页）的程序如下：

83

```
#include <iostream>
// 通过下列using声明，我们可以使用标准库中的名字
using std::cin;
using std::cout; using std::endl;
int main()
{
    cout << "Enter two numbers:" << endl;
```