Broadview
www.broadview.com.cn
PEARSON

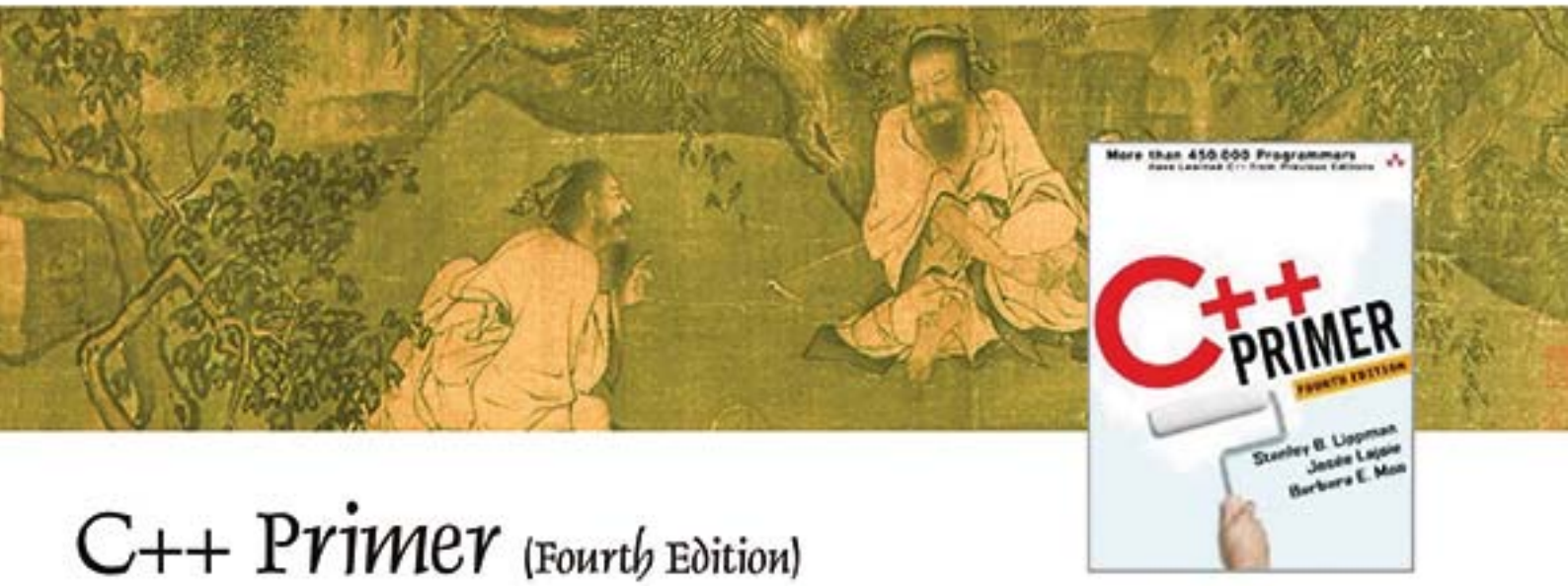
传世经典书丛
Eternal Classics

C++ Primer (第4版)

(评注版)

Stanley B. Lippman
[美] Josée Lajoie 著
Barbara E. Moo

陈硕 评注



C++ Primer (Fourth Edition)

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

内 容 简 介

《C++ Primer》是一本系统而权威的C++教材，它全面而深入地讲解了C++语言及其标准库。本书作者Stanley B. Lippman在20世纪80年代早期即在C++之父Bjarne Stroustrup领导下开发C++编译器，另一作者Josée Lajoie曾多年担任C++标准委员会核心语言组主席，他们对这门编程语言的理解与把握非常人可比。本书对C++语法和语义的阐释兼具准确性与可读性，在坊间无出其右者。第4版更吸收了先进的C++教学经验，在内容组织上对初学者更加友好，详略得当且重点突出，使读者能更快上手编写有用的程序，也更适合自学。全球已有45万人通过该书的各个版本学习了C++编程。

对于国外技术图书，选择翻译版还是影印版，常常让人陷入两难的境地。本评注版力邀国内资深专家执笔，在英文原著基础上增加中文点评与注释，旨在融合二者之长，既保留经典的原创文字与味道，又以先行者的学研心得与实践感悟，对读者阅读与学习加以点拨、指明捷径。

经过评注的版本，更值得反复阅读与体会。希望这本书能够帮助您跨越C++的重重险阻，领略高处才有的壮美风光，做一个成功而快乐的C++程序员。

版权贸易登记号　图字：01-2012-4456

图书在版编目（CIP）数据

C++ Primer：第4版：评注版/（美）李普曼（Lippman,S.B.），（美）拉乔伊（Lajoie,J.），（美）莫（Moo,B.E.）著；陈硕评注. —北京：电子工业出版社，2012.7

（传世经典书丛）

ISBN 978-7-121-17441-4

Ⅰ.①C… Ⅱ.①李… ②拉… ③莫… ④陈… Ⅲ.①C语言－程序设计－教材 Ⅳ.①TP312

中国版本图书馆CIP数据核字（2012）第138153号

策划编辑：张春雨

责任编辑：李云静

印　　刷：北京天宇星印刷厂

装　　订：三河市皇庄路通装订厂

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：850×1168　1/16　印张：43　　字数：1320千字

印　　次：2012年7月第1次印刷

定　　价：108.00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。

出版说明

悦读上品　得乎益友

孔子云：“取乎其上，得乎其中；取乎其中，得乎其下；取乎其下，则无所得矣”。

对于读书求知而言，这句古训教我们去读好书，最好是好书中的上品——经典书。其中，科技人员要读的技术书，因为直接关乎客观是非与生产效率，阅读选材本更应慎重。然而，随着技术图书品种的日益丰富，发现经典书越来越难，尤其对于涉世尚浅的新读者，更为不易，而他们又往往是最需要阅读、提升的重要群体。

所谓经典书，或说上品，是指选材精良、内容精练、讲述生动、外延丰盈、表现手法体贴入微的读品，它们会成为读者的知识和经验库中的重要组成部分，并且拥有从不断重读中汲取养分的空间。因此，选择阅读上品的问题便成了有效阅读的首要问题。当然，这不只是效率问题，上品促成的既是对某一种技术、思想的真正理解和掌握，同时又是一种感悟或享受，是一种愉悦。

与技术本身类似，经典 IT 技术书多来自国外。深厚的积累、良好的写作氛围，使一批大师为全球技术学习者留下了璀璨的智慧瑰宝。就在那个年代即将远去之时，无须回眸，也能感受到这一部部厚重而深邃的经典著作，在造福无数读者后从未蒙尘的熠熠光辉。而这些凝结众多当今国内技术中坚美妙记忆与绝佳体验的技术图书，虽然尚在国外图书市场上大放异彩，却已逐渐淡出国人的视线。最为遗憾的是，迟迟未有可以填补空缺的新书问世。而无可替代，不正是经典书被奉为圭臬的原因？

为了不让国内读者，尤其是即将步入技术生涯的新一代读者，就此错失这些滋养过先行者们的好书，以出版 IT 精品图书，满足技术人群需求为己任的我们，愿意承担这一使命。本次机遇惠顾了我们，让我们有机会携手权威的 Pearson 公司，精心推出“传世经典书丛”。

在我们眼中，“传世经典”的价值首先在于——既适合喜爱科技图书的读者，也符合专家们挑剔的标准。幸运的是，我们的确找到了这些堪称上品的佳作。丛书带给我们的幸运颇多，细数一下吧。

得以引荐大师著作

有恐思虑不周，我们大量参考了国外权威机构和网站的评选结果，并得到了 Pearson 的专业支持，又进

一步对符合标准之图书的国内外口碑与销售情况进行细致分析，也听取了国内技术专家的宝贵建议，才有幸选出对国内读者最富有技术养分的大师上品。

向深邃的技术内涵致敬

中外技术环境存在差异，很多享誉国外的好书未必适用于国内读者；且技术与应用瞬息万变，很容易让人心生迷惘或疲于奔命。本丛书的图书遴选，注重打好思考方法与技术理念的根基，旨在帮助读者修炼内功，提升境界，将技术真正融入个人知识体系，从而可以一通百通，从容面对随时涌现的技术变化。

翻译与评注的双项选择

引进优秀外版著作，将其翻译为中文供国内读者阅读，较为有效与常见。但另有一些外语水平较高、喜好阅读原版的读者，苦于对技术理解不足，不能充分体会原文表述的精妙，需要有人指导与点拨。而一批本土技术精英经过长期经典熏陶及实践锤炼，已足以胜任这一工作。有鉴于此，本丛书在翻译版的同时推出融合英文原著与中文点评、注释的评注版，供不同志趣的读者自由选择。

承蒙国内一流译(注)者的扶持

优秀的英文原著最终转化为真正的上品，尚需跨越翻译鸿沟，外版图书的翻译质量一直屡遭国内读者诟病。评注版的增值与含金量，同样依赖于评注者的高卓才具。好在，本丛书得到了久经考验的权威译(注)者的认可和支持，首肯我们选用其佳作，或亲自参与评注工作。正是他们的参与保证了经典的品质，既再次为我们的选材把关，更提供了一流的中文表述。

期望带给读者良好的阅读体验

一本好书带给人的愉悦不止于知识收获，良好的阅读感受同样不可缺少，且对学业不无助益。为让读者收获与上品相称的体验，我们在图书装帧设计与选材用料上同样不敢轻率，惟愿送到读者手中的除了珠玑章句，还有舒适与熨帖的视觉感受。

所有参与丛书出版的人员，尽管能力有限，却无不心怀严谨之心与完美愿望。如果读者朋友能从潜心阅读这些上品中偶有获益，不啻为对我们工作的最佳褒奖。若有阅读感悟，敬请拨冗告知，以鼓励我们继续在这一道路上贡献绵薄之力。如有不周之处，也请不吝指教。

电子工业出版社博文视点

从《C++ Primer（第 4 版）》入手学习 C++

为什么要学习 C++ ？

2009 年本书作者 Stanley Lippman 先生来华参加上海祝成科技举办的 C++ 技术大会，他表示人们现在还用 C++ 的唯一理由是其性能。相比之下，Java/C#/Python 等语言更加易学易用并且开发工具丰富，它们的开发效率都高于C++。但C++目前仍然是运行最快的语言[1]，如果你的应用领域确实在乎这个性能，那么 C++ 是不二之选。

这里略举几个例子[2]。对于手持设备而言，提高运行效率意味着完成相同的任务需要更少的电能，从而延长设备的操作时间，增强用户体验。对于嵌入式[3]设备而言，提高运行效率意味着：实现相同的功能可以选用较低档的处理器和较少的存储器，降低单个设备的成本；如果设备销量大到一定的规模，可以弥补 C++ 开发的成本。对于分布式系统而言，提高 10% 的性能就意味着节约 10% 的机器和能源。如果系统大到一定的规模（数千台服务器），值得用程序员的时间去换取机器的时间和数量，可以降低总体成本。另外，对于某些延迟敏感的应用（如游戏[4]、金融交易），通常不能容忍垃圾收集（GC）带来的不确定延时，而 C++ 可以自动并精确地控制对象销毁和内存释放时机[5]。我曾经不止一次见到，出于性能原因，用 C++ 重写现有的 Java 或 C# 程序。

C++之父Bjarne Stroustrup把C++定位于偏重系统编程（system programming）[6]的通用程序设计语言，开发信息基础架构（infrastructure）是 C++ 的重要用途之一[7]。Herb Sutter 总结道[8]，C++ 注重运行效率（efficiency）、灵活性（flexibility）[9]和抽象能力（abstraction），并为此付出了生产力（productivity）方面

1 见编程语言性能对比网站（http://shootout.alioth.debian.org/）和 Google 员工写的语言性能对比论文（https://days2011.scala-lang.org/sites/days2011/files/ws3-1-Hundt.pdf）。

2 C++ 之父 Bjarne Stroustrup 维护的 C++ 用户列表：http://www2.research.att.com/~bs/applications.html。

3 初窥 C++ 在嵌入式系统中的应用，参见 http://aristeia.com/TalkNotes/MISRA_Day_2010.pdf。

4 Milo Yip 在《C++ 强大背后》提到大部分游戏引擎（如 Unreal/Source）及中间件（如 Havok/FMOD）是 C++ 实现的。参见 http://www.cnblogs.com/miloyip/archive/2010/09/17/behind_cplusplus.html。

5 参见孟岩的《垃圾收集机制批判》：C++ 利用智能指针达成的效果是，一旦某对象不再被引用，系统刻不容缓，立刻回收内存。这通常发生在关键任务完成后的清理（cleanup）时期，不会影响关键任务的实时性；同时，内存里所有的对象都是有用的，绝对没有垃圾空占内存。参见 http://blog.csdn.net/myan/article/details/1906。

6 有人半开玩笑地说：“所谓系统编程，就是那些 CPU 时间比程序员的时间更重要的工作。”

7 《Software Development for Infrastructure》（http://www2.research.att.com/~bs/Computer-Jan12.pdf）。

8 Herb Sutter 在 C++ and Beyond 2011 会议上的开场演讲：《Why C++?》，参见 http://channel9.msdn.com/posts/C-and-Beyond-2011-Herb-Sutter-Why-C。

9 这里的灵活性指的是编译器不阻止你干你想干的事情，比如为了追求运行效率而实现即时编译（just-in-time compilation）。

的代价[10]。用本书作者的话来说，就是“C++ is about efficient programming with abstractions”（C++ 的核心价值在于能写出“运行效率不打折扣的抽象[11]”）。

要想发挥 C++ 的性能优势，程序员需要对语言本身及各种操作的代价有深入的了解[12]，特别要避免不必要的对象创建[13]。例如下面这个函数如果漏写了 &，功能还是正确的，但性能将会大打折扣。编译器和单元测试都无法帮我们查出此类错误，程序员自己在编码时须得小心在意。

```
inlineintfind_longest(conststd::vector<std::string>& words)
{
  // std::max_element(words.begin(), words.end(), LengthCompare());
}
```

在现代 CPU 体系结构下，C++ 的性能优势很大程度上得益于对内存布局（memory layout）的精确控制，从而优化内存访问的局部性[14]（locality of reference）并充分利用内存阶层（memory hierarchy）提速[15]，这一点优势在近期内不会被基于 GC 的语言赶上[16]。

C++ 的协作性不如 C、Java、Python，开源项目也比这几个语言少得多，因此在 TIOBE 语言流行榜中节节下滑。但是据我所知，很多企业内部使用 C++ 来构建自己的分布式系统基础架构，并且有替换 Java 开源实现的趋势。

学习 C++ 只需要读一本大部头

C++ 不是特性（features）最丰富的语言，却是最复杂的语言，诸多语言特性相互干扰，使其复杂度成倍增加。鉴于其学习难度和知识点之间的关联性，恐怕不能用“粗粗看看语法，就撸起袖子开干，边查 Google 边学习[17]”这种方式来学习 C++，那样很容易掉到陷阱里或养成坏的编程习惯。如果想成为专业 C++ 开发者，全面而深入地了解这门复杂语言及其标准库，你需要一本系统而权威[18]的书，这样的书必定会是一本八九百页的大部头[19]。

兼具系统性和权威性的 C++ 教材有两本，C++ 之父 Bjarne Stroustrup 的代表作《The C++ Programming Language》和 Stanley Lippman 的这本《C++ Primer》。侯捷先生评价道：“泰山北斗已现，

10 我曾向 Stanley Lippman 介绍目前我在 Linux 下的工作环境（编辑器、编译器、调试器），他表示这跟他在 20 世纪 70 年代的工作环境相差无几，可见 C++ 在开发工具方面的落后。另外，C++ 的编译运行调试周期也比现代的语言长，这多少影响了工作效率。

11 可参考 Ulrich Drepper 在《Stop Underutilizing Your Computer》中举的 SIMD 例子。参见 http://www.redhat.com/f/pdf/summit/udrepper_945_stop_underutilizing.pdf。

12 《Technical Report on C++ Performance》（http://www.open-std.org/jtc1/sc22/wg21/docs/18015.html）。

13 可参考 Scott Meyers 的《Effective C++ in an Embedded Environment》（http://www.artima.com/shop/effective_cpp_in_an_embedded_environment）。

14 我们知道 std::list 的任一位置插入的是 O(1) 操作，而 std::vector 的任一位置插入的是 O(*N*) 操作，但由于 std::vector 的元素布局更加紧凑（compact），很多时候 std::vector 的随机插入性能甚至会高于 std::list。参见 http://ecn.channel9.msdn.com/events/GoingNative12/GN12Cpp11Style.pdf，这也佐证了 std::vector 是首选容器。

15 可参考 Scott Meyers 的技术报告《CPU Caches and Why You Care》和任何一本现代的计算机体系结构教材（http://aristeia.com/TalkNotes/ACCU2011_CPUCaches.pdf）。

16 Bjarne Stroustrup 有一篇论文《Abstraction and the C++ machine model》，对比了 C++ 和 Java 的对象内存布局。参见 http://www2.research.att.com/~bs/abstraction-and-machine.pdf。

17 语出孟岩的《快速掌握一个语言最常用的 50%》（http://blog.csdn.net/myan/article/details/3144661）。

18 “权威”的意思是说你不用担心作者讲错了，能达到这个水准的 C++ 图书作者全世界也屈指可数。

19 同样篇幅的 Java/C#/Python 教材可以从语言、标准库一路讲到多线程、网络编程、图形编程。

又何必案牍劳形于墨瀚书海之中！这两本书都从 C++ 盘古开天以来，一路改版，斩将擎旗，追奔逐北，成就一生荣光[20]。”

从实用的角度，这两本书读一本即可，因为它们覆盖的 C++ 知识点相差无几。就我个人的阅读体验而言，Primer 更易读一些，我 10 年前深入学习 C++ 正是用的《C++ Primer（第 3 版）》。这次借评注的机会仔细阅读了《C++ Primer（第 4 版）》，感觉像在读一本完全不同的新书。第 4 版的内容组织及文字表达比第 3 版进步很多[21]，第 3 版可谓“事无巨细、面面俱到”;第 4 版则重点突出、详略得当，甚至篇幅也缩短了，这多半归功于新加盟的作者 Barbara Moo。

《C++ Primer（第 4 版）》讲什么？适合谁读？

这是一本 C++ 语言的教程，不是编程教程。本书不讲八皇后问题、Huffman 编码、汉诺塔、约瑟夫环、大整数运算等经典编程例题，本书的例子和习题往往都跟 C++ 本身直接相关。本书的主要内容是精解 C++ 语法（syntax）与语意（semantics），并介绍 C++ 标准库的大部分内容（含 STL）。“这本书在全世界 C++ 教学领域的突出和重要，已经无须我再赘言[22]。”

本书适合 C++ 语言的初学者，但不适合编程初学者。换言之，这本书可以是你的第一本 C++ 书，但恐怕不能作为第一本编程书。如果你不知道什么是变量、赋值、分支、条件、循环、函数，你需要一本更加初级的书[23]，本书第 1 章可用做自测题。

如果你已经学过一门编程语言，并且打算成为专业 C++ 开发者，从《C++ Primer（第 4 版）》入手不会让你走弯路。值得特别说明的是，学习本书不需要事先具备 C 语言知识。相反，这本书教你编写真正的 C++ 程序，而不是披着 C++ 外衣的 C 程序。

《C++ Primer（第 4 版）》的定位是语言教材，不是语言规格书，它并没有面面俱到地谈到 C++ 的每一个角落，而是重点讲解 C++ 程序员日常工作中真正有用的、必须掌握的语言设施和标准库[24]。本书的作者一点也不炫耀自己的知识和技巧，虽然他们有十足的资本[25]。这本书用语非常严谨（没有那些似是而非的比喻），用词平和，讲解细致，读起来并不枯燥。特别是如果你已经有一定的编程经验，在阅读时不妨思考如何用 C++ 来更好地完成以往的编程任务。

尽管英文原版书篇幅近 900 页，但其内容还是十分紧凑的，很多地方读一个句子就值得写一小段代码去验证。为了节省篇幅，本书经常修改前文代码中的一两行，来说明新的知识点，值得把每一行代码敲到机器中去验证。习题当然也不能轻易放过。

《C++ Primer（第 4 版）》体现了现代 C++ 教学与编程理念：在现成的高质量类库上构建自己的程序，而不是什么都从头自己写。这本书在第 3 章中介绍了 string 和 vector 这两个常用的类，立刻就能写出很

20 侯捷《大道之行也——C++ Primer 3/e 译序》（http://jjhou.boolan.com/cpp-primer-foreword.pdf）。

21 Bjarne Stroustrup 在《Programming: Principles and Practice Using C++》的参考文献中引用了本书，并特别注明“use only the 4th edition”。

22 参见侯捷的《C++ Primer 4/e 译序》。

23 如果没有时间精读注 21 中提到的那本大部头，短小精干的《Accelerated C++》亦是上佳之选。另外，如果想从 C 语言入手，我推荐裘宗燕老师的《从问题到程序：程序设计与 C 语言引论（第 2 版）》（机械工业出版社出版）。

24 本书把 iostream 的格式化输出放到附录，彻底不谈 locale/facet，可谓匠心独运。

25 Stanley Lippman 曾 说：“Virtual base class support wanders off into the Byzantine...The material is simply too esoteric to warrant discussion...”

多有用的程序。但作者不是一次性把 string 的上百个成员函数一一列举，而是有选择地讲解了最常用的那几个函数。

《C++ Primer（第 4 版）》的代码示例质量很高，不是那种随手写的玩具代码。第 10.4.2 节实现了带禁用词的单词计数；第 10.6 利用标准库容器简洁地实现了基于倒排索引思路的文本检索；第 15.9 节又用面向对象方法扩充了文本检索的功能，支持布尔查询。值得一提的是，这本书讲解继承和多态时举的例子符合 Liskov 替换原则，是正宗的面向对象。相反，某些教材以复用基类代码为目的，常以“人、学生、老师、教授”或“雇员、经理、销售、合同工”为例，这是误用了面向对象的“复用”。

《C++ Primer（第 4 版）》出版于 2005 年，遵循 2003 年的 C++ 语言标准[26]。C++ 新标准已于 2011 年定案（称为 C++11），本书不涉及 TR1[27] 和 C++11，这并不意味着这本书过时了[28]。相反，这本书里沉淀的都是当前广泛使用的 C++ 编程实践，学习它可谓正当时。评注版也不会越俎代庖地介绍这些新内容，但是会指出哪些语言设施已在新标准中废弃，避免读者浪费精力。

《C++ Primer（第 4 版）》是平台中立的，并不针对特定的编译器或操作系统。目前最主流的 C++ 编译器有两个，GNUG++ 和微软的 Visual C++。实际上，这两个编译器阵营基本上“模塑[29]”了 C++ 语言的行为。理论上讲，C++ 语言的行为是由 C++ 标准规定的。但是 C++ 不像其他很多语言有“官方参考实现[30]”，因此 C++ 的行为实际上是由语言标准、几大主流编译器、现有不计其数的 C++ 产品代码共同确定的，三者相互制约。C++ 编译器不光要尽可能符合标准，同时也要遵循目标平台的成文或不成文规范和约定，例如高效地利用硬件资源、兼容操作系统提供的 C 语言接口等。在 C++ 标准没有明文规定的地方，C++ 编译器也不能随心所欲地自由发挥。学习 C++ 的要点之一是明白哪些行为是由标准保证的，哪些是由实现（软硬件平台和编译器）保证的[31]，哪些是编译器自由实现、没有保证的；换言之，明白哪些程序行为是可依赖的。从学习的角度，我建议如果有条件，不妨两个编译器都用[32]，相互比照，避免把编译器和平台特定的行为误解为 C++ 语言规定的行为。尽管不是每个人都需要写跨平台的代码，但也大可不必自我限定在编译器的某个特定版本，毕竟编译器是会升级的。

本着“练从难处练，用从易处用”的精神，我建议在命令行下编译运行本书的示例代码，并尽量少用调试器。另外，值得了解 C++ 的编译链接模型[33]，这样才能不被实际开发中遇到的编译错误或链接错误绊住手脚。（C++ 不像现代语言那样有完善的模块（module）和包（package）设施，它从 C 语言继承了头文件、源文件、库文件等古老的模块化机制，这套机制相对较为脆弱，需要花一定时间学习规范的做法，避免误用。）

就学习 C++ 语言本身而言，我认为有几个练习非常值得一做。这不是“重复发明轮子”，而是必

---

26 基本等同于 1998 年的初版 C++ 标准，修正了编译器作者关心的一些问题，与普通程序员基本无关。

27 TR1 是 2005 年 C++ 标准库的一次扩充，增加了智能指针、bind/function、哈希表、正则表达式等。

28 作者正在编写《C++ Primer（第 5 版）》，会包含 C++11 的内容。

29 G++ 统治了 Linux 平台，并且能用在很多 Unix 平台上；Visual C++ 统治了 Windows 平台。其他 C++ 编译器的行为通常要向它们靠拢，例如 Intel C++ 在 Linux 上要兼容 G++，而在 Windows 上要兼容 Visual C++。

30 曾经是 Cfront，本书作者正是其主要开发者。参见 http://www.softwarepreservation.org/projects/c_plus_plus。

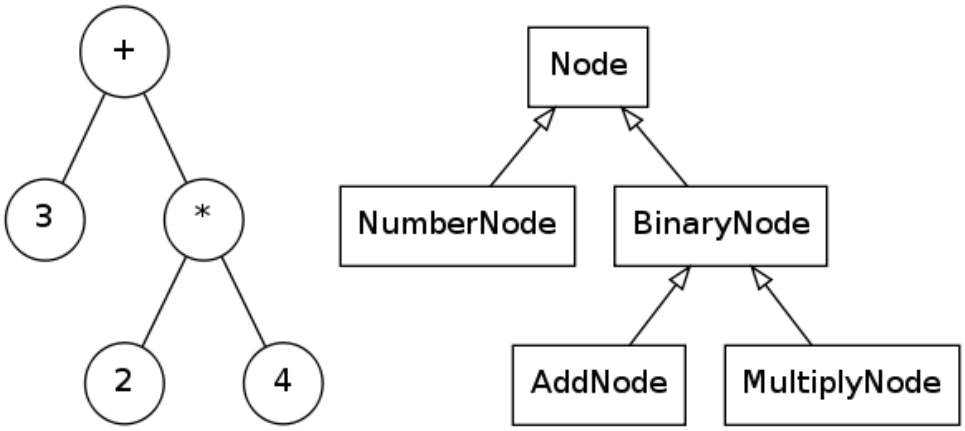
31 包括 C++ 标准有规定，但编译器拒绝遵循的。参见 http://stackoverflow.com/questions/3931312/value-initialization-and-non-pod-types。

32 G++ 是免费的，可使用较新的 4.x 版，最好 32-bit 和 64-bit 一起用，因为服务端已经普及 64 位。微软也有免费的编译器，可考虑 Visual C++ 2010 Express，建议不要用老掉牙的 Visual C++ 6.0 作为学习平台。

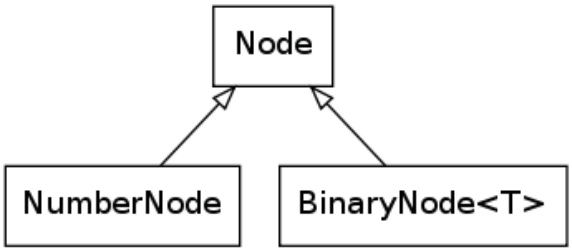
33 可参考陈硕写的《C++ 工程实践经验谈》中的“C++ 编译模型精要”一节。

要的编程练习，帮助你熟悉、掌握这门语言。一是写一个复数类或者大整数类[34]，实现基本的运算，熟悉封装与数据抽象。二是写一个字符串类，熟悉内存管理与拷贝控制。三是写一个简化的 vector<T> 类模板，熟悉基本的模板编程，你的这个 vector 应该能放入 int 和 string 等元素类型。四是写一个表达式计算器，实现一个节点类的继承体系（右图），体会面向对象编程。前三个练习是写独立的值语义的类，第四个练习是对象语义，同时要考虑类与类之间的关系。

表达式计算器能把四则运算式 3+2×4 解析为左图的表达式树[35]，对根节点调用 calculate() 虚函数就能算出表达式的值。做完之后还可以再扩充功能，比如支持三角函数和变量。



在写完面向对象版的表达式树之后，还可以略微尝试泛型编程。比如把类的继承体系简化为下图，然后用 BinaryNode<std::plus<double>> 和 BinaryNode<std:: multiplies<double>> 来具现化 BinaryNode<T> 类模板，通过控制模板参数的类型来实现不同的运算。



在表达式树这个例子中，节点对象是动态创建的，值得思考：如何才能安全地、不重不漏地释放内存。本书第 15.8 节的 Handle 可供参考。（C++ 的面向对象基础设施相对于现代的语言而言显得很简陋，现在 C++ 也不再以“支持面向对象”为卖点了。）

C++ 难学吗？“能够靠读书、看文章、读代码、做练习学会的东西没什么门槛，智力正常的人只要愿意花工夫，都不难达到（不错）的程度。[36]”C++ 好书很多，不过优秀的 C++ 开源代码很少，而且风格迥异[37]。我这里按个人口味和经验列几个供读者参考阅读：Google 的 protobuf、leveldb、PCRE 的 C++ 封装，我自己写的 muduo 网络库。这些代码都不长，功能明确，阅读难度不大。如果有时间，还可以读一读 Chromium 中的基础库源码。在读 Google 开源的 C++ 代码时要连注释一起细读。我不建议

34 大整数类可以以 std::vector<int> 为成员变量，避免手动资源管理。

35 “解析”可以用数据结构课程介绍的逆波兰表达式方法，也可以用编译原理中介绍的递归下降法，还可以用专门的 Packrat 算法。可参考 http://www.relisoft.com/book/lang/poly/3tree.html。

36 参见孟岩的《技术路线的选择重要但不具有决定性》（http://blog.csdn.net/myan/article/details/3247071）。

37 从代码风格上往往能判断项目成型的时代。

一开始就读 STL 或 Boost 的源码，因为编写通用 C++ 模板库和编写 C++ 应用程序的知识体系相差很大。另外可以考虑读一些优秀的 C 或 Java 开源项目，并思考是否可以用 C++ 更好地实现或封装之（特别是资源管理方面能否避免手动清理）。

继续前进

我能够随手列出十几本 C++ 好书，但是从实用角度出发，这里只举两三本必读的书。读过《C++ Primer》和这几本书之后，想必读者已能自行识别 C++ 图书的优劣，可以根据项目需要加以钻研。

第一本是《Effective C++（第 3 版）》[38]。学习语法是一回事，高效地运用这门语言是另一回事。C++ 是一个遍布陷阱的语言，吸取专家的经验尤为重要，既能快速提高眼界，又能避免重蹈覆辙。《C++ Primer》加上这本书包含的 C++ 知识足以应付日常应用程序开发。

我假定读者一定会阅读这本书，因此在评注中不引用《Effective C++（第 3 版）》的任何章节。

《Effective C++（第 3 版）》的内容也反映了 C++ 用法的进步。第 2 版建议“总是让基类拥有虚析构函数”，第 3 版改为“为多态基类声明虚析构函数”。因为在 C++ 中，“继承”不光只有面向对象这一种用途，即 C++ 的继承不一定是为了覆写（override）基类的虚函数。第 2 版花了很多笔墨介绍浅拷贝与深拷贝，以及对指针成员变量的处理 [39]。第 3 版则提议，对于多数 class 而言，要么直接禁用拷贝构造函数和赋值操作符；要么通过选用合适的成员变量类型 [40]，使得编译器默认生成的这两个成员函数就能正常工作。

什么是 C++ 编程中最重要的编程技法（idiom）？我认为是“用对象来管理资源”，即 RAII。资源包括动态分配的内存 [41]，也包括打开的文件、TCP 网络连接、数据库连接、互斥锁等。借助 RAII，我们可以把资源管理和对象生命期管理等同起来，而对象生命期管理在现代 C++ 里根本不困难（见注 3），只需要花几天时间熟悉几个智能指针 [42] 的基本用法即可。学会了这三招两式，现代的 C++ 程序中可以完全不写 delete，也不必为指针或内存错误操心。现代 C++ 程序里出现资源和内存泄漏的唯一可能是循环引用，一旦发现，也很容易修正设计和代码。这方面的详细内容请参考《Effective C++（第 3 版）》的第 3 章（资源管理）。

C++ 是目前唯一能实现自动化资源管理的语言，C 语言完全靠手工释放资源，而其他基于垃圾收集的语言只能自动清理内存，而不能自动清理其他资源 [43]（网络连接、数据库连接等）。

除了智能指针，TR1 中的 bind/function 也十分值得投入精力去学一学 [44]。让你从一个崭新的视角，重新审视类与类之间的关系。Stephan T. Lavavej 有一套 PPT 介绍 TR1 的这几个主要部件 [45]。

---

38 Scott Meyers 著，侯捷译，电子工业出版社出版。

39 Andrew Koenig 的《Teaching C++ Badly: Introduce Constructors and Destructors at the Same Time》（http://drdobbs.com/blogs/cpp/229500116）。

40 能自动管理资源的 string、vector、shared_ptr 等，这样多数 class 连析构函数都不必写。

41 “分配内存”包括在堆（heap）上创建对象。

42 包括 TR1 中的 shared_ptr、weak_ptr，还有更简单的 boost::scoped_ptr。

43 Java 7 有 try-with-resources 语句，Python 有 with 语句，C# 有 using 语句，可以自动清理栈上的资源，但对生命期大于局部作用域的资源无能为力，需要程序员手工管理。

44 参见孟岩的《function/bind 的救赎（上）》（http://blog.csdn.net/myan/article/details/5928531）。

45 参见 http://blogs.msdn.com/b/vcblog/archive/2008/02/22/tr1-slide-decks.aspx。

第二本书，如果读者还是在校学生，已经学过数据结构课程[46]的话，可以考虑读一读《泛型编程与 STL》[47]；如果已经工作，学完《C++ Primer》立刻就要参加 C++ 项目开发，那么我推荐阅读《C++ 编程规范》[48]。

泛型编程有一套自己的术语，如 concept、model、refinement 等，理解这套术语才能阅读泛型程序库的文档。即便不掌握泛型编程作为一种程序设计方法，也要掌握 C++ 中以泛型思维设计出来的标准容器库和算法库（STL）。坊间面向对象的书琳琅满目，学习机会也很多，而泛型编程只有这么一本，读之可以开拓视野，并且可加深对 STL 的理解（特别是迭代器[49]）和应用。

C++ 模板是一种强大的抽象手段，我不赞同每个人都把精力花在钻研艰深的模板语法和技巧上。从实用角度，能在应用程序中写写简单的函数模板和类模板即可（以 type traits 为限），并非每个人都要去写公用的模板库。

由于 C++ 语言过于庞大复杂，我见过的开发团队都对其剪裁使用[50]。往往团队越大，项目成立时间越早，剪裁得越厉害，也越接近 C。制订一份好的编程规范相当不容易。若规范定得太紧（比如定为团队成员知识能力的交集），则程序员束手束脚，限制了生产力，对程序员个人发展也不利[51]。若规范定得太松（定为团队成员知识能力的并集），则项目内代码风格迥异，学习交流协作成本上升，恐怕对生产力也不利。由两位顶级专家合写的《C++ 编程规范》一书可谓是现代 C++ 编程规范的范本。

《C++ 编程规范》同时也是专家经验一类的书，这本书的篇幅比《Effective C++（第 3 版）》短小，条款数目却多了近一倍，可谓言简意赅。有的条款看了就明白，照做即可：

- 第 1 条，以高警告级别编译代码，确保编译器无警告。
- 第 31 条，避免写出依赖于函数实参求值顺序的代码。C++ 操作符的优先级、结合性与表达式的求值顺序是无关的。裘宗燕老师写的《C/C++ 语言中表达式的求值》[52]一文对此有明确的说明。
- 第 35 条，避免继承“并非设计作为基类使用”的 class。
- 第 43 条，明智地使用 pimpl。这是编写 C++ 动态链接库的必备手法，可以最大限度地提高二进制兼容性。
- 第 56 条，尽量提供不会失败的 swap() 函数。有了 swap() 函数，我们在自定义赋值操作符时就不必检查自赋值了。
- 第 59 条，不要在头文件中或 #include 之前写 using。
- 第 73 条，以 by value 方式抛出异常，以 by reference 方式捕捉异常。
- 第 76 条，优先考虑 vector，其次再选择适当的容器。
- 第 79 条，容器内只可存放 value 和 smart pointer。

有的条款则需要相当的设计与编码经验才能解其中三昧：

---

46 最好再学一点基础的离散数学。

47 Matthew Austern 著，侯捷译，中国电力出版社出版。

48 Herb Sutter 等著，刘基诚译，人民邮电出版社出版（这本书的繁体版由侯捷先生和我翻译）。

49 侯捷先生的《芝麻开门：从 Iterator 谈起》（http://jjhou.boolan.com/programmer-3-traits.pdf）。

50 参见孟岩的《编程语言的层次观点——兼谈 C++ 的剪裁方案》（http://blog.csdn.net/myan/article/details/1920）。

51 一个人通常不会在一个团队工作一辈子，其他团队可能有不同的 C++ 剪裁使用方式，程序员要有“一桶水”的本事，才能应付不同形状大小的水碗。

52 参见 http://www.math.pku.edu.cn/teachers/qiuzy/technotes/expression2009.pdf。

- 第 5 条，为每个物体（entity）分配一个内聚任务。
- 第 6 条，正确性、简单性、清晰性居首。
- 第 8、9 条，不要过早优化；不要过早劣化。
- 第 22 条，将依赖关系最小化。避免循环依赖。
- 第 32 条，搞清楚你写的是哪一种 class。明白 value class、base class、trait class、policy class、exception class 各有其作用，写法也不尽相同。
- 第 33 条，尽可能写小型 class，避免写出“大怪兽”。
- 第 37 条，public 继承意味着可替换性。继承非为复用，乃为被复用。
- 第 57 条，将 class 类型及其非成员函数接口放入同一个 namespace。

值得一提的是，《C++ 编程规范》是出发点，但不是一份终极规范。例如 Google 的 C++ 编程规范 [53] 和 LLVM 编程规范 [54] 都明确禁用异常，这跟这本书的推荐做法正好相反。

评注版使用说明

评注版采用大 16 开印刷，在保留原书版式的前提下，对其进行了重新分页，评注的文字与正文左右分栏并列排版。另外，本书已依据原书 2010 年第 11 次印刷的版本进行了全面修订。为了节省篇幅，原书每章末尾的小结、术语表及书末的索引都没有印在评注版中，而是做成 PDF 供读者下载，这也方便读者检索。评注的目的是帮助初次学习 C++ 的读者快速深入掌握这门语言的核心知识，澄清一些概念、比较与其他语言的不同、补充实践中的注意事项等。评注的内容约占全书篇幅的 15%，大致比例是三分评、七分注，并有一些补白的内容 [55]。如果读者拿不定主意是否购买，可以先翻一翻第 5 章。我在评注中不谈 C++11[56]，但会略微涉及 TR1，因为 TR1 已经投入实用。

为了不打断读者阅读的思路，评注中不会给 URL 链接，评注中偶尔会引用《C++ 编程规范》的条款，以 [CCS] 标明，这些条款的标题已在前文列出。另外评注中出现的 soXXXXXX 表示 http://stackoverflow.com/questions/XXXXXX 网址。

网上资源

代码下载：http://www.informit.com/store/product.aspx?isbn=0201721481

豆瓣页面：http://book.douban.com/subject/10944985/

术语表与索引 PDF 下载：http://chenshuo.com/cp4（本序的电子版也发布于此，方便读者访问脚注中的网站）。

我的联系方式：giantchen@gmail.com
http://weibo.com/giantchen

陈硕
2012 年 5 月
中国 • 香港

53 参见 http://google-styleguide.googlecode.com/svn/trunk/cppguide.xml#Exceptions。

54 参见 http://llvm.org/docs/CodingStandards.html#ci_rtti_exceptions。

55 第 10 章绘制了数据结构示意图，第 11 章补充了 lower_bound 和 upper_bound 的示例。

56 通过 Scott Meyers 的讲义可以快速学习 C++11（http://www.artima.com/shop/overview_of_the_new_cpp）。

To Beth,
who makes this,
and all things,
possible.
—SBL

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

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

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

Contents

Preface

C++ Primer, Fourth Edition, provides a comprehensive introduction to the C++ language. As a primer, it provides a clear tutorial approach to the language, enhanced by numerous examples and other learning aids. Unlike most primers, it also provides a detailed description of the language, with particular emphasis on current and effective programming techniques.

Countless programmers have used previous editions of *C++ Primer* to learn C++. In that time C++ has matured greatly. Over the years, the focus of the language—and of C++ programmers—has grown beyond a concentration on run-time efficiency to focus on ways of making *programmers* more efficient. With the widespread availability of the standard library, it is possible to use and learn C++ more effectively than in the past. This revision of the *C++ Primer* reflects these new possiblities.

Changes to the Fourth Edition

In this edition, we have completely reorganized and rewritten the *C++ Primer* to highlight modern styles of C++ programming. This edition gives center stage to using the standard library while deemphasizing techniques for low-level programming. We introduce the standard library much earlier in the text and have reformulated the examples to take advantage of library facilities. We have also streamlined and reordered the presentation of language topics.

In addition to restructuring the text, we have incorporated several new elements to enhance the reader's understanding. Each chapter concludes with a Chapter Summary and glossary of Defined Terms, which recap the chapter's most important points. Readers should use these sections as a personal checklist: If you do not understand a term, restudy the corresponding part of the chapter.

We've also incorporated a number of other learning aids in the body of the text:

- Important terms are indicated in **bold**; important terms that we assume are already familiar to the reader are indicated in ***bold italics***. Each term appears in the chapter's Defined Terms section.
- Throughout the book, we highlight parts of the text to call attention to important aspects of the language, warn about common pitfalls, suggest good programming practices, and provide general usage tips. We hope that these notes will help readers more quickly digest important concepts and avoid common pitfalls.
- To make it easier to follow the relationships among features and concepts, we provide extensive forward and backward cross-references.
- We have provided sidebar discussions that focus on important concepts and supply additional explanations for topics that programmers new to C++ often find most difficult.
- Learning any programming language requires writing programs. To that end, the primer provides extensive examples throughout the text. Source code for the extended examples is available on the Web at the following URL:

  `http://www.awprofessional.com/cpp_primer`

What hasn't changed from earlier versions is that the book remains a comprehensive tutorial introduction to C++. Our intent is to provide a clear, complete and correct guide to the language. We teach the language by presenting a series of examples, which, in addition to explaining language features, show how to make the best use of C++. Although knowledge of C (the language on which C++ was originally based) is not assumed, we do assume the reader has programmed in a modern block-structured language.

Structure of This Book

C++ Primer provides an introduction to the International Standard on C++, covering both the language proper and the extensive library that is part of that standard. Much of the power of C++ comes from its support for programming with abstractions. Learning to program effectively in C++ requires more than learning new syntax and semantics. Our focus is on how to use the features of C++ to write programs that are safe, that can be built quickly, and yet offer performance comparable to the sorts of low-level programs often written in C.

C++ is a large language and can be daunting to new users. Modern C++ can be thought of as comprising three parts:

- The low-level language, largely inherited from C
- More advanced language features that allow us to define our own data types and to organize large-scale programs and systems
- The standard library, which uses these advanced features to provide a set of useful data structures and algorithms

Most texts present C++ in this same order: They start by covering the low-level details and then introduce the the more advanced language features. They explain the standard library only after having covered the entire language. The result, all too often, is that readers get bogged down in issues of low-level programming or the complexities of writing type definitions and never really understand the power of programming in a more abstract way. Needless to say, readers also often do not learn enough to build their own abstractions.

In this edition we take a completely different tack. We start by covering the basics of the language and the library together. Doing so allows you, the reader, to write significant programs. Only after a thorough grounding in using the library—and writing the kinds of abstract programs that the libary allows—do we move on to those features of C++ that will enable you to write your own abstractions.

Parts I and II cover the basic language and library facilities. The focus of these parts is to learn how to write C++ programs and how to use the abstractions from the library. Most C++ programmers need to know essentially everything covered in this portion of the book.

In addition to teaching the basics of C++, the material in Parts I and II serves another important purpose. The library facilities are themselves abstract data types written in C++. The library can be defined using the same class-construction features that are available to any C++ programmer. Our experience in teaching C++ is that by first using well-designed abstract types, readers find it easier to understand how to build their own types.

Parts III through V focus on how we can write our own types. Part III introduces the heart of C++: its support for classes. The class mechanism provides the basis for writing our own abstractions. Classes are also the foundation for object-oriented and generic programming, which we cover in Part IV. The *Primer* concludes with Part V, which covers advanced features that are of most use in structuring large, complex systems.

Acknowledgments

As in previous editions of this *Primer*, we'd like to extend our thanks to Bjarne Stroustrup for his tireless work on C++ and for his friendship to these authors throughout most of that time. We'd also like to thank Alex Stepanov for his original insights that led to the containers and algorithms that form the core of the standard library. Finally, our thanks go to the C++ Standards committee members for their hard work in clarifying, refining, and improving C++ over many years.

We also extend our deep-felt thanks to our reviewers, whose helpful comments on multiple drafts led us to make improvements great and small throughout the book: Paul Abrahams, Michael Ball, Mary Dageforde, Paul DuBois, Matt Greenwood, Matthew P. Johnson, Andrew Koenig, Nevin Liber, Bill Locke, Robert Murray, Phil Romanik, Justin Shaw, Victor Shtern, Clovis Tondo, Daveed Vandevoorde, and Steve Vinoski.

This book was typeset using LaTeX and the many packages that accompany the LaTeX distribution. Our well-justified thanks go to the members of the LaTeX community, who have made available such powerful typesetting tools.

The examples in this book have been compiled on the GNU and Microsoft compilers. Our thanks to their developers, and to those who have developed all the other C++ compilers, thereby making C++ a reality.

Finally, we thank the fine folks at Addison-Wesley who have shepherded this edition through the publishing process: Debbie Lafferty, our original editor, who initiated this edition and who had been with the *Primer* from its very first edition; Peter Gordon, our new editor, whose insistence on updating and streamlining the text have, we hope, greatly improved the presentation; Kim Boedigheimer, who keeps us all on schedule; and Tyrrell Albaugh, Jim Markham, Elizabeth Ryan, and John Fuller, who saw us through the design and production process.

CHAPTER 2

VARIABLES AND BASIC TYPES

CONTENTS

Types are fundamental to any program. They tell us what our data mean and what operations we can perform on our data.

C++ defines several primitive types: characters, integers, floating-point numbers, and so on. The language also provides mechanisms that let us define our own data types. The library uses these mechanisms to define more complex types such as variable-length character `strings`, `vectors`, and so on. Finally, we can modify existing types to form compound types. This chapter covers the built-in types and begins our coverage of how C++ supports more complicated types.

34

Types determine what the data and operations in our programs mean. As we saw in Chapter 1, the same statement

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

can mean different things depending on the types of `i` and `j`. If `i` and `j` are integers, then this statement has the ordinary, arithmetic meaning of `+`. However, if `i` and `j` are `Sales_item` objects, then this statement adds the components of these two objects.

In C++ the support for types is extensive: The language itself defines a set of primitive types and ways in which we can modify existing types. It also provides a set of features that allow us to define our own types. This chapter begins our exploration of types in C++ by covering the built-in types and showing how we associate a type with an object. It also introduces ways we can both modify types and can build our own types.

2.1 Primitive Built-in Types

▶C++ 的基本内置类型包括算术类型（整数、浮点数、字符、布尔值）和 void。

C++ defines a set of **arithmetic types**, which represent integers, floating-point numbers, and individual characters and boolean values. In addition, there is a special type named **`void`**. The `void` type has no associated values and can be used in only a limited set of circumstances. The `void` type is most often used as the return type for a function that has no return value.

The size of the arithmetic types varies across machines. By size, we mean the number of bits used to represent the type. The standard guarantees a minimum size for each of the arithmetic types, but it does not prevent compilers from using larger sizes. Indeed, almost all compilers use a larger size for `int` than is strictly required. Table 2.1 (p. 36) lists the built-in arithmetic types and the associated minimum sizes.

Because the number of bits varies, the maximum (or minimum) values that these types can represent also vary by machine.

2.1.1 Integral Types

▶ 一般把 integral types 翻译为“整值类型”或“有序类型”；而“整型”特指 int。

The arithmetic types that represent integers, characters, and boolean values are collectively referred to as the **integral types**.

There are two character types: `char` and `wchar_t`. The `char` type is guaranteed to be big enough to hold numeric values that correspond to any character in the machine's basic character set. As a result, `char`s are usually a single machine byte. The `wchar_t` type is used for extended character sets, such as those used for Chinese and Japanese, in which some characters cannot be represented within a single `char`.

The types `short`, `int`, and `long` represent integer values of potentially different sizes. Typically, `short`s are represented in half a machine word, `int`s in a

35

▶ C++ 的内存模型相当简单，内存由一个个相邻的字节组成，每个字节有一个地址（整数），即“按字节寻址”；每个字节的含义由使用它的代码决定；字节的大小理论上由实现决定，而实际上通用处理器都是 8-bit。

MACHINE-LEVEL REPRESENTATION OF THE BUILT-IN TYPES

The C++ built-in types are closely tied to their representation in the computer's memory. Computers store data as a sequence of bits, each of which holds either 0 or 1. A segment of memory might hold

```
00011011011100010110010000111011 ...
```

At the bit level, memory has no structure and no meaning.

The most primitive way we impose structure on memory is by processing it in chunks. Most computers deal with memory as chunks of bits of particular sizes, usually powers of 2. They usually make it easy to process 8, 16, or 32 bits at a time, and chunks of 64 and 128 bits are becoming more common. Although the exact sizes can

vary from one machine to another, we usually refer to a chunk of 8 bits as a "byte" and 32 bits, or 4 bytes, as a "word."

Most computers associate a number—called an address—with each byte in memory. Given a machine that has 8-bit bytes and 32-bit words, we might represent a word of memory as follows:

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

In this illustration, each byte's address is shown on the left, with the 8 bits of the byte following the address.

We can use an address to refer to any of several variously sized collections of bits starting at that address. It is possible to speak of the word at address 736424 or the byte at address 736426. We can say, for example, that the byte at address 736425 is not equal to the byte at address 736427.

To give meaning to the byte at address 736425, we must know the type of the value stored there. Once we know the type, we know how many bits are needed to represent a value of that type and how to interpret those bits.

If we know that the byte at location 736425 has type "unsigned 8-bit integer," then we know that the byte represents the number 112. On the other hand, if that byte is a character in the ISO-Latin-1 character set, then it represents the lower-case letter q. The bits are the same in both cases, but by ascribing different types to them, we interpret them differently.

machine word, and `long`s in either one or two machine words (on 32-bit machines, `int`s and `long`s are usually the same size).

▶ 机器字（machine word）的大小由机器字长决定。通常我们用的都是 32 位机，服务端编程普遍使用 64 位机。

The type `bool` represents the truth values, `true` and `false`. We can assign any of the arithmetic types to a `bool`. An arithmetic type with value 0 yields a `bool` that holds `false`. Any nonzero value is treated as `true`.

Signed and Unsigned Types

The integral types, except the boolean type, may be either **signed** or **unsigned**. As its name suggests, a signed type can represent both negative and positive numbers

Table 2.1: C++: Arithmetic Types

Type	Meaning	Minimum Size
`bool`	boolean	NA
`char`	character	8 bits
`wchar_t`	wide character	16 bits
`short`	short integer	16 bits
`int`	integer	16 bits
`long`	long integer	32 bits
`float`	single-precision floating-point	6 significant digits
`double`	double-precision floating-point	10 significant digits
`long double`	extended-precision floating-point	10 significant digits

36

▶ 表 2.1 列出的类型中最常用的有四个：int、bool、double、char。本章后文会介绍基本类型的选用规则。

(including zero), whereas an `unsigned` type represents only values greater than or equal to zero.

The integers, `int`, `short`, and `long`, are all signed by default. To get an unsigned type, the type must be specified as `unsigned`, such as `unsigned long`. The `unsigned int` type may be abbreviated as `unsigned`. That is, `unsigned` with no other type implies `unsigned int`.

Unlike the other integral types, there are three distinct types for `char`: plain `char`, `signed char`, and `unsigned char`. Although there are three distinct types, there are only two ways a `char` can be represented. The `char` type is respresented using either the `signed char` or `unsigned char` version. Which representation is used for `char` varies by compiler.

▶ 这是 C++ 语言的一个奇怪之处，char 是否有符号由编译器决定。通常编译器会提供一个选项，用于指明 char 是否有符号。

How Integral Values Are Represented

In an `unsigned` type, all the bits represent the value. If a type is defined for a particular machine to use 8 bits, then the `unsigned` version of this type could hold the values 0 through 255.

The C++ standard does not define how `signed` types are represented at the bit level. Instead, each compiler is free to decide how it will represent `signed` types. These representations can affect the range of values that a `signed` type can hold. We are guaranteed that an 8-bit `signed` type will hold at least the values from –127 through 127; many implementations allow values from –128 through 127.

Under the most common strategy for representing `signed` integral types, we can view one of the bits as a sign bit. Whenever the sign bit is 1, the value is negative; when it is 0, the value is either 0 or a positive number. An 8-bit integral `signed` type represented using a sign-bit can hold values from –128 through 127.

▶ 整数和浮点数在计算机内的表示方法并不是本书的重点，读者可参考计算机组成原理或计算机体系结构一类的教材。在通常的硬件平台上，二进制整数用补码（two's complement）表示，浮点数的表示方法由 IEEE 754 标准定义。二进制整数的表示与运算遵循相当简单的规律（从硬件实现的角度看），但是这个规律不一定符合直觉。例如，正负整数的表示范围是不对称的（因为有 0），16-bit int 的表示范围通常是 –32 768~32 767。这意味着 –32 768 的绝对值超过了 short int 的表示范围，if (s<0) { s = -s; } 并不总是能取到 s 的绝对值。

37

Assignment to Integral Types

The type of an object determines the values that the object can hold. This fact raises the question of what happens when one tries to assign a value outside the allowable range to an object of a given type. The answer depends on whether the type is `signed` or `unsigned`.

For `unsigned` types, the compiler *must* adjust the out-of-range value so that it will fit. The compiler does so by taking the remainder of the value modulo the number of distinct values the `unsigned` target type can hold. An object that is an 8-bit `unsigned char`, for example, can hold values from 0 through 255 inclusive. If we assign a value outside this range, the compiler actually assigns the remainder of the value modulo 256. For example, we might attempt to assign the value 336 to an 8-bit `signed char`. If we try to store 336 in our 8-bit `unsigned char`, the actual value assigned will be 80, because 80 is equal to 336 modulo 256.

For the `unsigned` types, a negative value is always out of range. An object of `unsigned` type may never hold a negative value. Some languages make it illegal to assign a negative value to an `unsigned` type, but C++ does not.

In C++ it is perfectly legal to assign a negative number to an object with `unsigned` type. The result is the negative value modulo the size of the type. So, if we assign –1 to an 8-bit `unsigned char`, the resulting value will be 255, which is –1 modulo 256.

▶ 写程序时一般应该在“安全范围”内使用整数，避免溢出（整数的加减乘除运算都有溢出的可能）。在极少数情况下，溢出是意料中的，比如算 hash 值时。

When assigning an out-of-range value to a `signed` type, it is up to the compiler to decide what value to assign. In practice, many compilers treat `signed` types similarly to how they are required to treat `unsigned` types. That is, they do the assignment as the remainder modulo the size of the type. However, we are not guaranteed that the compiler will do so for the `signed` types.

2.1.2 Floating-Point Types

The types `float`, `double`, and `long double` represent floating-point single-, double-, and extended-precision values. Typically, `float`s are represented in one word (32 bits), `double`s in two words (64 bits), and `long double` in either three or four words (96 or 128 bits). The size of the type determines the number of significant digits a floating-point value might contain.

The `float` type is usually not precise enough for real programs—`float` is guaranteed to offer only 6 significant digits. The `double` type guarantees at least 10 significant digits, which is sufficient for most calculations.

▶ 程序里如果用到浮点数，应该默认用 double 类型，少用 float 类型。

2.2 Literal Constants

A value, such as `42`, in a program is known as a **literal constant**: literal because we can speak of it only in terms of its value; constant because its value cannot be changed. Every literal has an associated type. For example, `0` is an `int` and `3.14159` is a `double`. Literals exist only for the built-in types. There are no literals of class types. Hence, there are no literals of any of the library types.

▶ 字面常量的关键知识点是：不同的写法有不同的类型。

ADVICE: USING THE BUILT-IN ARITHMETIC TYPES

38

The number of integral types in C++ can be bewildering. C++, like C, is designed to let programs get close to the hardware when necessary, and the integral types are defined to cater to the peculiarities of various kinds of hardware. Most programmers can (and should) ignore these complexities by restricting the types they actually use.

In practice, many uses of integers involve counting. For example, programs often count the number of elements in a data structure such as a `vector` or an array. We'll see in Chapters 3 and 4 that the library defines a set of types to use when dealing with the size of an object. When counting such elements it is always right to use the library-defined type intended for this purpose. When counting in other circumstances, it is usually right to use an `unsigned` value. Doing so avoids the possibility that a value that is too large to fit results in a (seemingly) negative result.

When performing integer arithmetic, it is rarely right to use `shorts`. In most programs, using `shorts` leads to mysterious bugs when a value is assigned to a `short` that is bigger than the largest number it can hold. What happens depends on the machine, but typically the value "wraps around" so that a number too large to fit turns into a large negative number. For the same reason, even though `char` is an integral type, the `char` type should be used to hold characters and not for computation. The fact that `char` is `signed` on some implementations and `unsigned` on others makes it problematic to use it as a computational type.

On most machines, integer calculations can safely use `int`. Technically speaking, an `int` can be as small as 16 bits—too small for most purposes. In practice, almost all general-purpose machines use 32-bits for `int`s, which is often the same size used for `long`. The difficulty in deciding whether to use `int` or `long` occurs on machines that have 32-bit `int`s and 64-bit `long`s. On such machines, the *run-time* cost of doing arithmetic with `long`s can be considerably greater than doing the same calculation using a 32-bit `int`. Deciding whether to use `int` or `long` requires detailed understanding of the program and the actual run-time performance cost of using `long` versus `int`.

Determining which floating-point type to use is easier: It is almost always right to use `double`. The loss of precision implicit in `float` is significant, whereas the cost of double precision calculations versus single precision is negligible. In fact, on some machines, double precision is faster than single. The precision offered by `long double` usually is unnecessary and often entails considerable extra run-time cost.

Rules for Integer Literals

We can write a literal integer constant using one of three notations: decimal, octal, or hexadecimal. These notations, of course, do not change the bit representation of the value, which is always binary. For example, we can write the value `20` in any of the following three ways:

```
20     // decimal
024    // octal
0x14   // hexadecimal
```

▶ 根据语法来看，0 其实是个八进制数。

Literal integer constants that begin with a leading `0` (zero) are interpreted as octal; those that begin with either `0x` or `0X` are interpreted as hexadecimal.

By default, the type of a literal integer constant is either `int` or `long`. The precise type depends on the value of the literal—values that fit in an `int` are type

39

EXERCISES SECTION 2.1.2

Exercise 2.1: What is the difference between an `int`, a `long`, and a `short` value?

Exercise 2.2: What is the difference between an `unsigned` and a `signed` type?

Exercise 2.3: If a `short` on a given machine has 16 bits then what is the largest number that can be assigned to a `short`? To an `unsigned short`?

Exercise 2.4: What value is assigned if we assign 100,000 to a 16-bit `unsigned short`? What value is assigned if we assign 100,000 to a plain 16-bit `short`?

Exercise 2.5: What is the difference between a `float` and a `double`?

Exercise 2.6: To calculate a mortgage payment, what types would you use for the rate, principal, and payment? Explain why you selected each type.

▶ 根据语法，八进制或十六进制的整数字面常量有可能是 unsigned 类型（unsigned int 或 unsigned long），而十进制的整数字面常量只能是 int 或 long 类型。不过代码中还是直接以后缀标明类型为佳。

`int` and larger values are type `long`. By adding a suffix, we can force the type of a literal integer constant to be type `long` or `unsigned` or `unsigned long`. We specify that a constant is a `long` by immediately following the value with either `L` or `l` (the letter "ell" in either uppercase or lowercase).

When specifying a long, use the uppercase `L`: the lowercase letter `l` is too easily mistaken for the digit 1.

In a similar manner, we can specify `unsigned` by following the literal with either `U` or `u`. We can obtain an `unsigned long` literal constant by following the value by both `L` and `U`. The suffix must appear with no intervening space:

```
128u     /* unsigned   */        1024UL    /* unsigned long   */
1L       /* long   */             8Lu      /* unsigned long   */
```

There are no literals of type `short`.

▶ 注意：整数字面常量没有负数，代码中出现的形如“-1”的数是个表达式，表示 1 这个字面常量的相反数（详见第 5.1 节：一元取反操作符）。

Rules for Floating-Point Literals

We can use either common decimal notation or scientific notation to write floating-point literal constants. Using scientific notation, the exponent is indicated either by `E` or `e`. By default, floating-point literals are type `double`. We indicate single precision by following the value with either `F` or `f`. Similarly, we specify extended precision by following the value with either `L` or `l` (again, use of the lowercase `l` is discouraged). Each pair of literals below denote the same underlying value:

```
3.14159F          .001f          12.345L          0.
3.14159E0f        1E-3F          1.2345E1L        0e0
```

Boolean and Character Literals

The words `true` and `false` are literals of type `bool`:

```
bool test = false;
```

40

Printable character literals are written by enclosing the character within single quotation marks:

```
'a'       '2'       ','       ' ' // blank
```

▶ 注意：一对单引号之间只能有一个字符，否则编译器会给出警告。

Such literals are of type `char`. We can obtain a wide-character literal of type `wchar_t` by immediately preceding the character literal with an `L`, as in

```
L'a'
```

Escape Sequences for Nonprintable Characters

Some characters are **nonprintable**. A nonprintable character is a character for which there is no visible image, such as backspace or a control character. Other characters have special meaning in the language, such as the single and double quotation marks, and the backslash. Nonprintable characters and special characters are written using an **escape sequence**. An escape sequence begins with a backslash. The language defines the following escape sequences:

newline	\n	horizontal tab	\t
vertical tab	\v	backspace	\b
carriage return	\r	formfeed	\f
alert (bell)	\a	backslash	\\
question mark	\?	single quote	\'
double quote	\"		

We can write any character as a generalized escape sequence of the form

```
\ooo
```

where ooo represents a sequence of as many as three octal digits. The value of the octal digits represents the numerical value of the character. The following examples are representations of literal constants using the ASCII character set:

```
\7 (bell)     \12 (newline)     \40 (blank)
\0 (null)     \062 ('2')        \115 ('M')
```

▶ 如果用 \ooo 方式书写转义字符，则建议写满三位八进制整数。

The character represented by '\0' is often called a "null character," and has special significance, as we shall soon see.

We can also write a character using a hexadecimal escape sequence

```
\xddd
```

consisting of a backslash, an x, and one or more hexadecimal digits.

▶ 如果用 \xdd 方式书写转义字符，则建议写满两位十六进制整数。

Character String Literals

All of the literals we've seen so far have primitive built-in types. There is one additional literal—string literal—that is more complicated. String literals are ***arrays*** of constant characters, a type that we'll discuss in more detail in Section 4.3 (p. 130).

▶ 字符串字面常量，简称字符串字面量。在 C++ 中，字符串字面量的类型是个以常量字符（const char）为元素的数组，它可以退化为指向常量字符的指针。

String literal constants are written as zero or more characters enclosed in double quotation marks. Nonprintable characters are represented by their underlying escape sequence.

```
"Hello World!"                  // simple string literal
""                              // empty string literal
"\nCC\toptions\tfile.[cC]\n"    // string literal using newlines and tabs
```

41

For compatibility with C, string literals in C++ have one character in addition to those typed in by the programmer. Every string literal ends with a null character added by the compiler. A character literal

```
'A'    // single quote: character literal
```

represents the single character A, whereas

```
"A"    // double quote: character string literal
```

represents an array of two characters: the letter A and the null character.

Just as there is a wide character literal, such as

```
L'a'
```

there is a wide string literal, again preceded by L, such as

```
L"a wide string literal"
```

The type of a wide string literal is an array of constant wide characters. It is also terminated by a wide null character.

Concatenated String Literals

Two string literals (or two wide string literals) that appear adjacent to one another and separated only by spaces, tabs, or newlines are concatenated into a single new string literal. This usage makes it easy to write long literals across separate lines:

```
// concatenated long string literal
std::cout << "a multi-line "
             "string literal "
             "using concatenation"
          << std::endl;
```

When executed this statement would print:

```
a multi-line string literal using concatenation
```

What happens if you attempt to concatenate a string literal and a wide string literal? For example:

```
// Concatenating plain and wide character strings is undefined
std::cout << "multi-line " L"literal " << std::endl;
```

The result is **undefined**—that is, there is no standard behavior defined for concatenating the two different types. The program might appear to work, but it also might crash or produce garbage values. Moreover, the program might behave differently under one compiler than under another.

42

▶C++ 程序一定要避免“未定义的行为”，所以 C++ 程序员一定要知道哪些是“未定义的行为”。

ADVICE: DON'T RELY ON UNDEFINED BEHAVIOR

Programs that use undefined behavior are in error. If they work, it is only by coincidence. Undefined behavior results from a program error that the compiler cannot detect or from an error that would be too much trouble to detect.

Unfortunately, programs that contain undefined behavior can appear to execute correctly in some circumstances and/or on one compiler. There is no guarantee that the same program, compiled under a different compiler or even a subsequent release of the current compiler, will continue to run correctly. Nor is there any guarantee that what works with one set of inputs will work with another.

Programs should not (knowingly) rely on undefined behavior.

Similarly, programs usually should not rely on machine-dependent behavior, such as assuming that the size of an `int` is a fixed and known value. Such programs are said to be *nonportable*. When the program is moved to another machine, any code that relies on machine-dependent behavior may have to be found and corrected. Tracking down these sorts of problems in previously working programs is, mildly put, a profoundly unpleasant task.

Multi-Line Literals

There is a more primitive (and less useful) way to handle long strings that depends on an infrequently used program formatting feature: Putting a backslash as the last character on a line causes that line and the next to be treated as a single line.

As noted on page 14, C++ programs are largely free-format. In particular, there are only a few places that we may not insert whitespace. One of these is in the middle of a word. In particular, we may not break a line in the middle of a word. We can circumvent this rule by using a backslash:

```
// ok: A \ before a newline ignores the line break
std::cou\
t << "Hi" << st\
d::endl;
```

is equivalent to

```
std::cout << "Hi" << std::endl;
```

We could use this feature to write a long string literal:

```
    // multiline string literal
    std::cout << "a multi-line \
string literal \
using a backslash"
              << std::endl;
    return 0;
}
```

Note that the backslash must be the last thing on the line—no comments or trailing blanks are allowed. Also, any leading spaces or tabs on the subsequent lines are part of the literal. For this reason, the continuation lines of the long literal do not have the normal indentation.

EXERCISES SECTION 2.2

Exercise 2.7: Explain the difference between the following sets of literal constants:

(a) `'a', L'a', "a", L"a"`
(b) `10, 10u, 10L, 10uL, 012, 0xC`
(c) `3.14, 3.14f, 3.14L`

Exercise 2.8: Determine the type of each of these literal constants:

```
(a) -10   (b) -10u   (c) -10.   (d) -10e-2
```

Exercise 2.9: Which, if any, of the following are illegal?

```
(a) "Who goes with F\145rgus?\012"
(b) 3.14e1L          (c) "two" L"some"
(d) 1024f            (e) 3.14UL
(f) "multiple line
     comment"
```

Exercise 2.10: Using escape sequences, write a program to print `2M` followed by a newline. Modify the program to print `2`, then a tab, then an `M`, followed by a newline.

43

▶ 根据勘误，这里四个选项中 10 前面的负号都应该去掉，因为“–10”等是表达式，不是字面量。

2.3 Variables

Imagine that we are given the problem of computing 2 to the power of 10. Our first attempt might be something like

```
#include <iostream>
int main()
{
    // a first, not very good, solution
    std::cout << "2 raised to the power of 10: ";
    std::cout << 2*2*2*2*2*2*2*2*2*2;
    std::cout << std::endl;
    return 0;
}
```

This program solves the problem, although we might double- or triple-check to make sure that exactly 10 literal instances of 2 are being multiplied. Otherwise, we're satisfied. Our program correctly generates the answer 1,024.

We're next asked to compute 2 raised to the power of 17 and then to the power of 23. Changing our program each time is a nuisance. Worse, it proves to be remarkably error-prone. Too often, the modified program produces an answer with one too few or too many instances of 2.

An alternative to the explicit brute force power-of-2 computation is twofold:

1. Use named objects to perform and print each computation.
2. Use flow-of-control constructs to provide for the repeated execution of a sequence of program statements while a condition is true.

44 Here, then, is an alternative way to compute 2 raised to the power of 10:

```
#include <iostream>
int main()
{
    // local objects of type int
    int value = 2;
    int pow = 10;
    int result = 1;
    // repeat calculation of result until cnt is equal to pow
    for (int cnt = 0; cnt != pow; ++cnt)
        result *= value;   // result = result * value;
    std::cout << value
              << " raised to the power of "
              << pow << ": \t"
              << result << std::endl;
    return 0;
}
```

`value`, `pow`, `result`, and `cnt` are variables that allow for the storage, modification, and retrieval of values. The `for` loop allows for the repeated execution of our calculation until it's been executed `pow` times.

EXERCISES SECTION 2.3

Exercise 2.11: Write a program that prompts the user to input two numbers, the base and exponent. Print the result of raising the base to the power of the exponent.

KEY CONCEPT: STRONG STATIC TYPING

C++ is a statically typed language, which means that types are checked at compile time. The process by which types are checked is referred to as type-checking.

In most languages, the type of an object constrains the operations that the object can perform. If the type does not support a given operation, then an object of that type cannot perform that operation.

In C++, whether an operation is legal or not is checked at compile time. When we write an expression, the compiler checks that the objects used in the expression are used in ways that are defined by the type of the objects. If not, the compiler generates an error message; an executable file is not produced.

As our programs, and the types we use, get more complicated, we'll see that static type checking helps find bugs in our programs earlier. A consequence of static checking is that the type of every entity used in our programs must be known to the compiler. Hence, we must define the type of a variable before we can use that variable in our programs.

45

2.3.1 What Is a Variable?

▶ C++ 中的“变量”就是 object with a name，即带名字的对象。C++ 的“对象”在下一页有定义。

A variable provides us with named storage that our programs can manipulate. Each variable in C++ has a specific type, which determines the size and layout of the variable's memory; the range of values that can be stored within that memory; and the set of operations that can be applied to the variable. C++ programmers tend to refer to variables as “variables” or as “objects” interchangeably.

Lvalues and Rvalues

We'll have more to say about expressions in Chapter 5, but for now it is useful to know that there are two kinds of expressions in C++:

1. **lvalue** (pronounced "ell-value"): An expression that is an lvalue may appear as either the left-hand or right-hand side of an assignment.
2. **rvalue** (pronounced "are-value"): An expression that is an rvalue may appear on the right- but not left-hand side of an assignment.

▶ 左值和右值的主要区别在于左值可以被赋值。当然，我们将会看到（见 2.4 节），也有一些左值不能被赋值，称为 unmodifible lvalue 。左值（lvalue）中的字母 l 的另外一个意思是"位置（location）"，即左值一定在内存里有位置，可以取其地址（用一元 & 操作符），而右值不能取其地址，详见英文原版书第 115 页。

Variables are lvalues and so may appear on the left-hand side of an assignment. Numeric literals are rvalues and so may not be assigned. Given the variables:

```
int units_sold = 0;
double sales_price = 0, total_revenue = 0;
```

it is a compile-time error to write either of the following:

```
// error: arithmetic expression is not an lvalue
units_sold * sales_price = total_revenue;
// error: literal constant is not an lvalue
0 = 1;
```

Some operators, such as assignment, require that one of their operands be an lvalue. As a result, lvalues can be used in more contexts than can rvalues. The context in which an lvalue appears determines how it is used. For example, in the expression

```
units_sold = units_sold + 1;
```

the variable `units_sold` is used as the operand to two different operators. The + operator cares only about the values of its operands. The value of a variable is the value currently stored in the memory associated with that variable. The effect of the addition is to fetch that value and add one to it.

The variable `units_sold` is also used as the left-hand side of the = operator. The = operator reads its right-hand side and writes to its left-hand side. In this expression, the result of the addition is stored in the storage associated with `units_sold`; the previous value in `units_sold` is overwritten.

In the course of the text, we'll see a number of situations in which the use of an rvalue or lvalue impacts the behavior and/or the performance of our programs—in particular when passing and returning values from a function.

EXERCISES SECTION 2.3.1

46

Exercise 2.12: Distinguish between an lvalue and an rvalue; show examples of each.

Exercise 2.13: Name one case where an lvalue is required.

TERMINOLOGY: WHAT IS AN OBJECT?

C++ programmers tend to be cavalier in their use of the term *object*. Most generally, an object is a region of memory that has a type. More specifically, evaluating an expression that is an lvalue yields an object.

Strictly speaking, some might reserve the term *object* to describe only variables or values of class types. Others might distinguish between named and unnamed objects, always referring to variables when discussing named objects. Still others distinguish between objects and values, using the term *object* for data that can be changed by the program and using the term *value* for those that are read-only.

In this book, we'll follow the more colloquial usage that an object is a region of memory that has a type. We will freely use *object* to refer to most of the data manipu-

▶C++ 中的对象不一定是指"面向对象程序设计"中的对象，int、double、char* 类型的变量都是对象；不过，函数不是对象（函数对象 /function object 是另外一个意思）。

lated by our programs regardless of whether those data have built-in or class type, are named or unnamed, or are data that can be read or written.

2.3.2 The Name of a Variable

▶ 这是标识符的语法规则。变量名、函数名、类名等都是标识符。注意，有些标识符是系统保留的，我们应该避免在程序中使用，包括以下画线开头，或者包含两个下画线的标识符等。见 so228783。

在代码中给变量和函数命名是一项关键的任务，可参考《代码大全》有关章节。C++ 程序应该遵循统一的命名格式，例如对函数和类型命名 LikeThis，对局部变量命名 likeThis，对成员变量命名 likeThis_，对预处理宏命名 LIKE_THIS。

The name of a variable, its **identifier**, can be composed of letters, digits, and the underscore character. It must begin with either a letter or an underscore. Upper- and lowercase letters are distinct: Identifiers in C++ are case-sensitive. The following defines four distinct identifiers:

```
// declares four different int variables
int somename, someName, SomeName, SOMENAME;
```

There is no language-imposed limit on the permissible length of a name, but out of consideration for others that will read and/or modify our code, it should not be too long.

For example,

```
gosh_this_is_an_impossibly_long_name_to_type
```

is a really bad identifier name.

C++ Keywords

C++ reserves a set of words for use within the language as keywords. Keywords may not be used as program identifiers. Table 2.2 on the next page lists the complete set of C++ keywords.

C++ also reserves a number of words that can be used as alternative names for various operators. These alternative names are provided to support character sets

47

Table 2.2: C++ Keywords

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

that do not support the standard set of C++ operator symbols. These names, listed in Table 2.3, also may not be used as identifiers:

Table 2.3: C++ Operator Alternative Names

and	bitand	compl	not_eq	or_eq	xor_eq
and_eq	bitor	not	or	xor	

In addition to the keywords, the standard also reserves a set of identifiers for use in the library. Identifiers cannot contain two consecutive underscores, nor can an identifier begin with an underscore followed immediately by an upper-case letter. Certain identifiers—those that are defined outside a function—may not begin with an underscore.

Conventions for Variable Names

There are a number of generally accepted conventions for naming variables. Following these conventions can improve the readability of a program.

- A variable name is normally written in lowercase letters. For example, one writes `index`, not `Index` or `INDEX`.
- An identifier is given a mnemonic name—that is, a name that gives some indication of its use in a program, such as `on_loan` or `salary`.
- An identifier containing multiple words is written either with an underscore between each word or by capitalizing the first letter of each embedded word. For example, one generally writes `student_loan` or `studentLoan`, not `studentloan`.

The most important aspect of a naming convention is that it be applied consistently.

48

EXERCISES SECTION 2.3.2

Exercise 2.14: Which, if any, of the following names are invalid? Correct each identified invalid name.

```
(a) int double = 3.14159;      (b) char _;
(c) bool catch-22;             (d) char 1_or_2 = '1';
(e) float Float = 3.14f;
```

2.3.3 Defining Objects

The following statements define five variables:

```
int units_sold;
double sales_price, avg_price;
std::string title;
Sales_item  curr_book;
```

Each definition starts with a **type specifier**, followed by a comma-separated list of one or more names. A semicolon terminates the definition. The type specifier names the type associated with the object: `int`, `double`, `std::string`, and `Sales_item` are all names of types. The types `int` and `double` are built-in types, `std::string` is a type defined by the library, and `Sales_item` is a type that we used in Section 1.5 (p. 20) and will define in subsequent chapters. The type determines the amount of storage that is allocated for the variable and the set of operations that can be performed on it.

Multiple variables may be defined in a single statement:

```
double salary, wage;    // defines two variables of type double
int month,
    day, year;          // defines three variables of type int
std::string address;    // defines one variable of type std::string
```

▶ 本书会反复提到拷贝初始化和直接初始化两个术语，值得印在脑中。另外，拷贝初始化不是赋值。

Initialization

A definition specifies a variable's type and identifier. A definition may also provide an initial value for the object. An object defined with a specified first value is spoken of as **initialized**. C++ supports two forms of variable initialization: **copy-initialization** and **direct-initialization**. The copy-initialization syntax uses the equal (=) symbol; direct-initialization places the initializer in parentheses:

```
int ival(1024);    // direct-initialization
int ival = 1024;   // copy-initialization
```

49 In both cases, `ival` is initialized to `1024`.

Although, at this point in the book, it may seem obscure to the reader, in C++ it is essential to understand that initialization is not assignment. Initialization happens when a variable is created and gives that variable its initial value. Assignment involves obliterating an object's current value and replacing that value with a new one.

Many new C++ programmers are confused by the use of the = symbol to initialize a variable. It is tempting to think of initialization as a form of assignment. But initialization and assignment are different operations in C++. This concept is particularly confusing because in many other languages the distinction is irrelevant and can be ignored. Moreover, even in C++ the distinction rarely matters until one attempts to write fairly complex classes. Nonetheless, it is a crucial concept and one that we will reiterate throughout the text.

There are subtle differences between copy- and direct-initialization when initializing objects of a class type. We won't completely explain these differences until Chapter 13. For now, it's worth knowing that the direct syntax is more flexible and can be slightly more efficient.

Using Multiple Initializers

When we initialize an object of a built-in type, there is only one way to do so: We supply a value, and that value is copied into the newly defined object. For built-in types, there is little difference between the direct and the copy forms of initialization.

For objects of a class type, there are initializations that can be done only using direct-initialization. To understand why, we need to know a bit about how classes control initialization.

Each class may define one or more special member functions (Section 1.5.2, p. 24) that say how we can initialize variables of the class type. The member functions that define how initialization works are known as **constructors**. Like any function, a constructor can take multiple arguments. A class may define several constructors, each of which must take a different number or type of arguments.

As an example, we'll look a bit at the `string` class, which we'll cover in more detail in Chapter 3. The `string` type is defined by the library and holds character strings of varying sizes. To use `string`s, we must include the `string` header. Like the IO types, `string` is defined in the `std` namespace.

The `string` class defines several constructors, giving us various ways to initialize a `string`. One way we can initialize a `string` is as a copy of a character string literal:

```
#include <string>
// alternative ways to initialize string from a character string literal
std::string titleA = "C++ Primer, 4th Ed.";
std::string titleB("C++ Primer, 4th Ed.");
```

In this case, either initialization form can be used. Both definitions create a `string` object whose initial value is a copy of the specified string literal. 50

However, we can also initialize a `string` from a count and a character. Doing so creates a `string` containing the specified character repeated as many times as indicated by the count:

```
std::string all_nines(10, '9');  // all_nines = "9999999999"
```

In this case, the only way to initialize `all_nines` is by using the direct form of initialization. It is not possible to use copy-initialization with multiple initializers.

▶ 如果构造函数有不止一个实参，那么只能直接初始化。

Initializing Multiple Variables

▶ 建议一行代码、一条语句只定义一个变量。

When a definition defines two or more variables, each variable may have its own initializer. The name of an object becomes visible immediately, and so it is possible to initialize a subsequent variable to the value of one defined earlier in the same definition. Initialized and uninitialized variables may be defined in the same definition. Both forms of initialization syntax may be intermixed:

```
#include <string>
// ok: salary defined and initialized before it is used to initialize wage
double salary = 9999.99,
       wage(salary + 0.01);
// ok: mix of initialized and uninitialized
int interval,
    month = 8, day = 7, year = 1955;
// ok: both forms of initialization syntax used
std::string title("C++ Primer, 4th Ed."),
            publisher = "A-W";
```

An object can be initialized with an arbitrarily complex expression, including the return value of a function:

```
double price = 109.99, discount = 0.16;
double sale_price = apply_discount(price, discount);
```

In this example, `apply_discount` is a function that takes two values of type `double` and returns a value of type `double`. We pass the variables `price` and `discount` to that function and use its return value to initialize `sale_price`.

2.3.4 Variable Initialization Rules

When we define a variable without an initializer, the system sometimes initializes the variable for us. What value, if any, is supplied depends on the type of the variable and may depend on where it is defined.

Initialization of Variables of Built-in Type

Whether a variable of built-in type is automatically initialized depends on where it is defined. Variables defined outside any function body are initialized to zero.

EXERCISES SECTION 2.3.3

51

Exercise 2.15: What, if any, are the differences between the following definitions:

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

If either definition contains an error, how might you correct the problem?

Exercise 2.16: Assuming `calc` is a function that returns a `double`, which, if any, of the following are illegal definitions? Correct any that are identified as illegal.

```
(a) int car = 1024, auto = 2048;
(b) int ival = ival;
(c) std::cin >> int input_value;
(d) double salary = wage = 9999.99;
(e) double calc = calc();
```

Variables of built-in type defined inside the body of a function are **uninitialized**. Using an uninitialized variable for anything other than as the left-hand operand of an assignment is undefined. Bugs due to uninitialized variables can be hard to find. As we cautioned on page 42, you should never rely on undefined behavior.

We recommend that every object of built-in type be initialized. It is not always necessary to initialize such variables, but it is easier and safer to do so until you can be certain it is safe to omit an initializer.

CAUTION: UNINITIALIZED VARIABLES CAUSE RUN-TIME PROBLEMS

Using an uninitialized object is a common program error, and one that is often difficult to uncover. The compiler is not required to detect a use of an uninitialized variable, although many will warn about at least some uses of uninitialized variables. However, no compiler can detect all uses of uninitialized variables.

Sometimes, we're lucky and using an uninitialized variable results in an immediate crash at run time. Once we track down the location of the crash, it is usually pretty easy to see that the variable was not properly initialized.

Other times, the program completes but produces erroneous results. Even worse, the results can appear correct when we run our program on one machine but fail on another. Adding code to the program in an unrelated location can cause what we thought was a correct program to suddenly start to produce incorrect results.

The problem is that uninitialized variables actually do have a value. The compiler puts the variable somewhere in memory and treats whatever bit pattern was in that memory as the variable's initial state. When interpreted as an integral value, any bit pattern is a legitimate value—although the value is unlikely to be one that the programmer intended. Because the value is legal, using it is unlikely to lead to a crash. What it is likely to do is lead to incorrect execution and/or incorrect calculation.

52

Initialization of Variables of Class Type

Each class defines how objects of its type can be initialized. Classes control object initialization by defining one or more constructors (Section 2.3.3, p. 49). As an example, we know that the `string` class provides at least two constructors. One of these constructors lets us initialize a `string` from a character string literal and another lets us initialize a `string` from a character and a count.

Each class may also define what happens if a variable of the type is defined but an initializer is not provided. A class does so by defining a special constructor, known as the **default constructor**. This constructor is called the default constructor because it is run "by default;" if there is no initializer, then this constructor is used. The default constructor is used regardless of where a variable is defined.

Most classes provide a default constructor. If the class has a default constructor, then we can define variables of that class without explicitly initializing them. For example, the `string` type defines its default constructor to initialize the `string` as an empty string—that is, a string with no characters:

```
std::string empty;  // empty is the empty string; empty = ""
```

Some class types do not have a default constructor. For these types, every definition must provide explicit initializer(s). It is not possible to define variables of such types without giving an initial value.

EXERCISES SECTION 2.3.4

Exercise 2.17: What are the initial values, if any, of each of the following variables?

```
std::string global_str;
int global_int;
int main()
{
    int local_int;
    std::string local_str;
    // ...
    return 0;
}
```

2.3.5 Declarations and Definitions

▶C++ 里“声明”和“定义”是两码事。本节谈的是变量的定义与声明，我们还将了解函数以及 class 的定义与声明。

As we'll see in Section 2.9 (p. 67), C++ programs typically are composed of many files. In order for multiple files to access the same variable, C++ distinguishes between declarations and definitions.

A **definition** of a variable allocates storage for the variable and may also specify an initial value for the variable. There must be one and only one definition of a variable in a program.

A **declaration** makes known the type and name of the variable to the program. A definition is also a declaration: When we define a variable, we declare its name and type. We can declare a name without defining it by using the `extern` keyword. A declaration that is not also a definition consists of the object's name and its type preceded by the keyword `extern`:

53

```
extern int i;    // declares but does not define i
int i;           // declares and defines i
```

An `extern` declaration is *not* a definition and does not allocate storage. In effect, it claims that a definition of the variable exists elsewhere in the program. A variable can be declared multiple times in a program, but it must be defined only once.

A declaration may have an initializer only if it is also a definition because only a definition allocates storage. The initializer must have storage to initialize. If an initializer is present, the declaration is treated as a definition even if the declaration is labeled `extern`:

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

▶这种用法较少见，有的编译器甚至会给出警告。这个语法是为了兼容 extern const 定义，见第 2.4 节。

Despite the use of `extern`, this statement defines `pi`. Storage is allocated and initialized. An `extern` declaration may include an initializer only if it appears outside a function.

Because an `extern` that is initialized is treated as a definition, any subseqent definition of that variable is an error:

```
extern double pi = 3.1416; // definition
double pi;                 // error: redefinition of pi
```

Similarly, a subsequent `extern` declaration that has an initializer is also an error:

```
extern double pi = 3.1416;  // definition
extern double pi;           // ok: declaration not definition
extern double pi = 3.1416;  // error: redefinition of pi
```

The distinction between a declaration and a definition may seem pedantic but in fact is quite important.

In C++ a variable must be defined exactly once and must be defined or declared before it is used.

Any variable that is used in more than one file requires declarations that are separate from the variable's definition. In such cases, one file will contain the definition for the variable. Other files that use that same variable will contain declarations for—but not a definition of—that same variable.

EXERCISES SECTION 2.3.5

Exercise 2.18: Explain the meaning of each of these instances of `name`:

```
extern std::string name;
std::string name("exercise 3.5a");
extern std::string name("exercise 3.5a");
```

54

2.3.6 Scope of a Name

Every name in a C++ program must refer to a unique entity (such as a variable, function, type, etc.). Despite this requirement, names can be used more than once in a program: A name can be reused as long as it is used in different contexts, from which the different meanings of the name can be distinguished. The context used to distinguish the meanings of names is a **scope**. A scope is a region of the program. A name can refer to different entities in different scopes.

Most scopes in C++ are delimited by curly braces. Generally, names are visible from their point of declaration until the end the scope in which the declaration appears. As an example, consider this program, which we first encountered in Section 1.4.2 (p. 14):

```
#include <iostream>
int main()
{
    int sum = 0;
    // sum values from 1 up to 10 inclusive
    for (int val = 1; val <= 10; ++val)
        sum += val;  // equivalent to sum = sum + val

    std::cout << "Sum of 1 to 10 inclusive is "
              << sum << std::endl;
    return 0;
}
```

►C++ 中的作用域除了能决定名字的能见度，还能直接控制对象的生命期（lifetime），不可不察。

This program defines three names and uses two names from the standard library. It defines a function named `main` and two variables named `sum` and `val`. The name `main` is defined outside any curly braces and is visible throughout the program. Names defined outside any function have **global scope**; they are accessible from anywhere in the program. The name `sum` is defined within the scope of the `main` function. It is accessible throughout the `main` function but not outside of it. The variable `sum` has **local scope**. The name `val` is more interesting. It is defined in the scope of the `for` statement (Section 1.4.2, p. 14). It can be used in that statement but not elsewhere in `main`. It has **statement scope**.

Scopes in C++ Nest

Names defined in the global scope can be used in a local scope; global names and those defined local to a function can be used inside a statement scope, and so on. Names can also be redefined in an inner scope. Understanding what entity a name refers to requires unwinding the scopes in which the names are defined:

```
#include <iostream>
```

```
#include <string>
/*  Program for illustration purposes only:
 *  It is bad style for a function to use a global variable and then
 *  define a local variable with the same name
 */
 std::string s1 = "hello";  //  s1 has global scope
 int main()
 {
     std::string s2 = "world"; //  s2 has local scope
     //  uses global s1; prints ''hello world''
     std::cout << s1 << " " << s2 << std::endl;
     int s1 = 42; //  s1 is local and hides global s1
     //  uses local s1; prints ''42 world''
     std::cout << s1 << " " <<  s2 << std::endl;
     return 0;
 }
```

55

This program defines three variables: a global `string` named `s1`, a local `string` named `s2`, and a local `int` named `s1`. The definition of the local `s1` *hides* the global `s1`.

Variables are visible from their point of declaration. Thus, the local definition of `s1` is not visible when the first output is performed. The name `s1` in that output expression refers to the global `s1`. The output printed is `hello world`. The second statement that does output follows the local definition of `s1`. The local `s1` is now in scope. The second output uses the local rather than the global `s1`. It writes `42 world`.

Programs such as the preceeding are likely to be confusing. It is almost always a bad idea to define a local variable with the same name as a global variable that the function uses or might use. It is much better to use a distinct name for the local.

▶ 应该避免内层变量名遮盖外围作用域的变量名（函数名、类型名同理），对此有的编译器提供了警告选项，如 g++ 的 -Wshadow。

We'll have more to say about local and global scope in Chapter 7 and about statement scope in Chapter 6. C++ has two other levels of scope: **class scope**, which we'll cover in Chapter 12 and **namespace scope**, which we'll see in Section 17.2.

2.3.7 Define Variables Where They Are Used

In general, variable definitions or declarations can be placed anywhere within the program that a statement is allowed. A variable must be declared or defined before it is used.

It is usually a good idea to define an object near the point at which the object is first used.

Defining an object where the object is first used improves readability. The reader does not have to go back to the beginning of a section of code to find the definition of a particular variable. Moreover, it is often easier to give the variable a useful initial value when the variable is defined close to where it is first used.

One constraint on placing declarations is that variables are accessible from the point of their definition until the end of the enclosing block. A variable must be defined in or before the outermost scope in which the variable will be used.

EXERCISES SECTION 2.3.6

56

Exercise 2.19: What is the value of `j` in the following program?

```
int i = 42;
int main()
{
```

```
    int i = 100;
    int j = i;
    // ...
}
```

Exercise 2.20: Given the following program fragment, what values are printed?

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

Exercise 2.21: Is the following program legal?

```
int sum = 0;
for (int i = 0; i != 10; ++i)
    sum += i;
std::cout << "Sum from 0 to " << i
          << " is " << sum << std::endl;
```

▶C++ 中的 const 有多种用途，既可以修饰变量，也可以修饰函数；并且可以和指针、引用相结合，表达更为复杂的意图。

2.4 `const` Qualifier

There are two problems with the following `for` loop, both concerning the use of 512 as an upper bound.

```
for (int index = 0; index != 512; ++index) {
    // ...
}
```

The first problem is readability. What does it mean to compare `index` with 512? What is the loop doing—that is, what makes 512 matter? (In this example, 512 is known as a **magic number**, one whose significance is not evident within the context of its use. It is as if the number had been plucked by magic from thin air.)

The second problem is maintainability. Imagine that we have a large program in which the number 512 occurs 100 times. Let's further assume that 80 of these references use 512 to indicate the size of a particular buffer but the other 20 use 512 for different purposes. Now we discover that we need to increase the buffer size to 1024. To make this change, we must examine every one of the places that the number 512 appears. We must determine—correctly in every case—which of those uses of 512 refer to the buffer size and which do not. Getting even one instance wrong breaks the program and requires us to go back and reexamine each use.

The solution to both problems is to use an object initialized to 512:

```
int bufSize = 512;     // input buffer size
for (int index = 0; index != bufSize; ++index) {
    // ...
}
```

57

By choosing a mnemonic name, such as `bufSize`, we make the program more readable. The test is now against the object rather than the literal constant:

```
index != bufSize
```

If we need to change this size, the 80 occurrences no longer need to be found and corrected. Rather, only the one line that initializes `bufSize` requires change. Not only does this approach require significantly less work, but also the likelihood of making a mistake is greatly reduced.

Defining a `const` Object

There is still a serious problem with defining a variable to represent a constant value. The problem is that `bufSize` is modifiable. It is possible for `bufSize` to be changed—accidentally or otherwise. The `const` type qualifier provides a solution: It transforms an object into a constant.

```
const int bufSize = 512;     // input buffer size
```

defines `bufSize` to be a constant initialized with the value 512. The variable `bufSize` is still an lvalue (Section 2.3.1, p. 45), but now the lvalue is unmodifiable. Any attempt to write to `bufSize` results in a compile-time error.

```
bufSize = 0; // error: attempt to write to const object
```

Because we cannot subsequently change the value of an object declared to be `const`, we must initialize it when it is defined:

```
const std::string hi = "hello!"; // ok: initialized
const int i, j = 0;  // error: i is uninitialized const
```

`const` Objects Are Local to a File By Default

When we define a nonconst variable at global scope (Section 2.3.6, p. 54), it is accessible throughout the program. We can define a nonconst variable in one file and—assuming an appropriate declaration has been made—can use that variable in another file:

```
// file_1.cc
int counter;  // definition
// file_2.cc
extern int counter; // uses counter from file_1
++counter;          // increments counter defined in file_1
```

Unlike other variables, unless otherwise specified, `const` variables declared at global scope are local to the file in which the object is defined. The variable exists in that file only and cannot be accessed by other files.

We can make a `const` object accessible throughout the program by specifying that it is `extern`:

```
// file_1.cc
// defines and initializes a const that is accessible to other files
extern const int bufSize = fcn();
// file_2.cc
extern const int bufSize; // uses bufSize from file_1
// uses bufSize defined in file_1
for (int index = 0; index != bufSize; ++index)
      // ...
```

In this program, `file_1.cc` defines and initializes `bufSize` to the result returned from calling the function named `fcn`. The definition of `bufSize` is `extern`, meaning that `bufSize` can be used in other files. The declaration in `file_2.cc` is also made `extern`. In this case, the `extern` signifies that `bufSize` is a declaration and hence no initializer is provided.

We'll see in Section 2.9.1 (p. 69) why `const` objects are made local to a file.

Nonconst variables are `extern` by default. To make a `const` variable accessible to other files we must explicitly specify that it is `extern`.

58

▶ 注意：这里把 extern 和 const 连用，定义了一个外部可见的 const 对象。这个用法与英文原版书第 53 页的讲法相呼应。一般我们把 const 对象定义在头文件中，有了 extern const 这种用法，我们也可以把 const 对象放在 .cc 文件中，这样修改其值时就不必重新编译全部源文件了。

EXERCISES SECTION 2.4

Exercise 2.22: The following program fragment, while legal, is an example of poor style. What problem(s) does it contain? How would you improve it?

```
for (int i = 0; i < 100; ++i)
     // process i
```

Exercise 2.23: Which of the following are legal? For those usages that are illegal, explain why.

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

2.5 References

A **reference** serves as an alternative name for an object. In real-world programs, references are primarily used as formal parameters to functions. We'll have more to say about reference parameters in Section 7.2.2 (p. 232). In this section we introduce and illustrate the use of references as independent objects.

A reference is a **compound type** that is defined by preceding a variable name by the & symbol. A compound type is a type that is defined in terms of another type. In the case of references, each reference type "refers to" some other type. We cannot define a reference to a reference type, but can make a reference to any other data type.

▶"引用"在C++里是一个特殊的东西，其本意是"别名（alias）"。所有对"引用"的操作都会实施在"本尊"身上，"本尊"即被引用的对象——对"引用"取地址，实际取得的是"本尊"的地址；对"引用"取sizeof，实际取得的是"本尊"的大小。"引用"的特殊性还体现在很多地方，如只有数组的引用（reference to array），但没有引用的数组（array of references）；只有指针的引用（reference to pointer），没有指向引用的指针（pointer to reference），等等。引用的主要用途之一是用于函数（含操作符）的参数和返回类型，所以前面这几章尚看不出其作用。

59

A reference *must* be initialized using an object of the same type as the reference:

```
int ival = 1024;
int &refVal = ival; // ok: refVal refers to ival
int &refVal2;       // error: a reference must be initialized
int &refVal3 = 10;  // error: initializer must be an object
```

A Reference Is an Alias

Because a reference is just another name for the object to which it is bound, *all* operations on a reference are actually operations on the underlying object to which the reference is bound:

```
refVal += 2;
```

adds 2 to `ival`, the object referred to by `refVal`. Similarly,

```
int ii = refVal;
```

assigns to `ii` the value currently associated with `ival`.

When a reference is initialized, it remains bound to that object as long as the reference exists. There is no way to rebind a reference to a different object.

▶如果A是一个类型，那么"复合类型"包括A*（指针）、A&（引用）、A[]（数组）等。"引用"只有一层，不存在"二级引用"或"引用的引用"；相反，C++里有"指向指针的指针"。

The important concept to understand is that a reference is *just another name for an object*. Effectively, we can access `ival` either through its actual name or through its alias, `refVal`. Assignment is just another operation, so that when we write

```
refVal = 5;
```

the effect is to change the value of `ival` to 5. A consequence of this rule is that you must initialize a reference when you define it; initialization is the only way to say to which object a reference refers.

Defining Multiple References

We can define multiple references in a single type definition. Each identifier that is a reference must be preceded by the & symbol:

```
int i = 1024, i2 = 2048;
int &r = i, r2 = i2;       // r is a reference, r2 is an int
int i3 = 1024, &ri = i3;   // defines one object, and one reference
int &r3 = i3, &r4 = i2;    // defines two references
```

const References

A **const reference** is a reference that may refer to a const object:

```
const int ival = 1024;
const int &refVal = ival; // ok: both reference and object are const
int &ref2 = ival;         // error: nonconst reference to a const object
```

We can read from but not write to refVal. Thus, any assignment to refVal is illegal. This restriction should make sense: We cannot assign directly to ival and so it should not be possible to use refVal to change ival.

60

▶const reference 不能写入，但并不代表其值一定不会改变，因为它引用的对象有可能是 non-const，会被其他代码修改。

For the same reason, the initialization of ref2 by ival is an error: ref2 is a plain, **nonconst reference** and so could be used to change the value of the object to which ref2 refers. Assigning to ival through ref2 would result in changing the value of a const object. To prevent such changes, it is illegal to bind a plain reference to a const object.

TERMINOLOGY: CONST REFERENCE IS A REFERENCE TO CONST

C++ programmers tend to be cavalier in their use of the term const reference. Strictly speaking, what is meant by "const reference" is "reference to const." Similarly, programmers use the term "nonconst reference" when speaking of reference to a nonconst type. This usage is so common that we will follow it in this book as well.

▶ 注意：const reference 和 reference to const 是一回事，提法不同而已。因为 C++ 里没有真正的 non-const reference（指可以 rebind 的 reference，见前页），所以 non-const reference 就借指 reference to non-const 了。这和指针不同，const pointer 和 pointer to const 是两码事，C++ 里甚至还有 const pointer to const。

A const reference can be initialized to an object of a different type or to an rvalue (Section 2.3.1, p. 45), such as a literal constant:

```
int i = 42;
// legal for const references only
const int &r = 42;
const int &r2 = r + i;
```

The same initializations are not legal for nonconst references. Rather, they result in compile-time errors. The reason is subtle and warrants an explanation.

This behavior is easiest to understand when we look at what happens when we bind a reference to an object of a different type. If we write

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

the compiler transforms this code into something like this:

```
int temp = dval;          // create temporary int from the double
const int &ri = temp;     // bind ri to that temporary
```

If ri were not const, then we could assign a new value to ri. Doing so would not change dval but would instead change temp. To the programmer expecting that assignments to ri would change dval, it would appear that the change did not work. Allowing only const references to be bound to values requiring temporaries avoids the problem entirely because a const reference is read-only.

A nonconst reference may be attached only to an object of the same type as the reference itself.

A const reference may be bound to an object of a different but related type or to an rvalue.

EXERCISES SECTION 2.5

61

Exercise 2.24: Which of the following definitions, if any, are invalid? Why? How would you correct them?

```
(a) int ival = 1.01;     (b) int &rval1 = 1.01;
```

```
(c) int &rval2 = ival;   (d) const int &rval3 = 1;
```

Exercise 2.25: Given the preceeding definitions, which, if any, of the following assignments are invalid? If they are valid, explain what they do.

```
(a) rval2 = 3.14159;     (b) rval2 = rval3;
(c) ival = rval3;        (d) rval3 = ival;
```

Exercise 2.26: What are the differences among the definitions in (a) and the assignments in (b)? Which, if any, are illegal?

```
(a) int ival = 0;            (b) ival = ri;
    const int &ri = 0;           ri = ival;
```

Exercise 2.27: What does the following code print?

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

2.6 Typedef Names

▶ 复杂 typedef 的认法：把 typedef 遮住，那么这个语句定义了一个变量，如 wages；再把 typedef 放回去，wages 就成了一个类型，该类型与遮住 typedef 时定义的 wages 变量的类型相同。

A **typedef** lets us define a synonym for a type:

```
typedef double wages;          // wages is a synonym for double
typedef int exam_score;        // exam_score is a synonym for int
typedef wages salary;          // indirect synonym for double
```

A typedef name can be used as a type specifier:

```
wages hourly, weekly;     // double hourly, weekly;
exam_score test_result;   // int test_result;
```

▶ 注意：typedef 不定义新的类型，而是定义一个“同义词”。

A typedef definition begins with the keyword `typedef`, followed by the data type and identifier. The identifier, or typedef name, does not introduce a new type but rather a synonym for the existing data type. A typedef name can appear anywhere in a program that a type name can appear.

Typedefs are commonly used for one of three purposes:

- To hide the implementation of a given type and emphasize instead the purpose for which the type is used
- To streamline complex type definitions, making them easier to understand
- To allow a single type to be used for more than one purpose while making the purpose clear each time the type is used

62

2.7 Enumerations

Often we need to define a set of alternative values for some attribute. A file, for example, might be open in one of three states: input, output, and append. One way to keep track of these state values might be to associate a unique constant number with each. We might write the following:

```
const int input = 0;
const int output = 1;
const int append = 2;
```

Although this approach works, it has a significant weakness: There is no indication that these values are related in any way. **Enumerations** provide an alternative method of not only defining but also grouping sets of integral constants.

Defining and Initializing Enumerations

An enumeration is defined using the `enum` keyword, followed by an optional enu-

meration name, and a comma-separated list of **enumerators** enclosed in braces.

```
// input is 0, output is 1, and append is 2
enum open_modes {input, output, append};
```

By default, the first enumerator is assigned the value zero. Each subsequent enumerator is assigned a value one greater than the value of the enumerator that immediately precedes it.

Enumerators Are `const` Values

We may supply an initial value for one or more enumerators. The value used to initialize an enumerator must be a **constant expression**. A constant expression is an expression of integral type that the compiler can evaluate at compile time. An integral literal constant is a constant expression, as is a `const` object (Section 2.4, p. 56) that is itself initialized from a constant expression.

For example, we might define the following enumeration:

```
// shape is 1, sphere is 2, cylinder is 3, polygon is 4
enum Forms {shape = 1, sphere, cylinder, polygon};
```

In the `enum Forms` we explicitly assigned `shape` the value `1`. The other enumerators are implicitly initialized: `sphere` is initialized to `2`, `cylinder` to `3`, and `polygon` to `4`.

An enumerator value need not be unique.

```
// point2d is 2, point2w is 3, point3d is 3, point3w is 4
enum Points { point2d = 2, point2w,
              point3d = 3, point3w };
```

In this example, the enumerator `point2d` is explicitly initialized to `2`. The next enumerator, `point2w`, is initialized by default, meaning that its value is one more than the value of the previous enumerator. Thus, `point2w` is initialized to `3`. The enumerator `point3d` is explicitly initialized to `3`, and `point3w`, again is initialized by default, in this case to `4`.

63

It is not possible to change the value of an enumerator. As a consequence an enumerator is itself a constant expression and so can be used where a constant expression is required.

Each `enum` Defines a Unique Type

Each `enum` defines a new type. As with any type, we can define and initialize objects of type `Points` and can use those objects in various ways. An object of enumeration type may be initialized or assigned only by one of its enumerators or by another object of the same enumeration type:

```
Points pt3d = point3d; // ok: point3d is a Points enumerator
Points pt2w = 3;       // error: pt2w initialized with int
pt2w = polygon;        // error: polygon is not a Points enumerator
pt2w = pt3d;           // ok: both are objects of Points enum type
```

Note that it is illegal to assign the value `3` to a `Points` object even though `3` is a value associated with one of the `Points` enumerators.

2.8 Class Types

In C++ we define our own data types by defining a **class**. A class defines the data that an object of its type contains and the operations that can be executed by objects of that type. The library types `string`, `istream`, and `ostream` are all defined as classes.

▶ 类（class）是 C++ 的重要元素。本节只简单介绍类的数据成员的写法，完整的介绍见第 12 章。

C++ support for classes is extensive—in fact, defining classes is so important that we shall devote Parts III through V to describing C++ support for classes and operations using class types.

In Chapter 1 we used the `Sales_item` type to solve our bookstore problem. We used objects of type `Sales_item` to keep track of sales data associated with a particular ISBN. In this section, we'll take a first look at how a simple class, such as `Sales_item`, might be defined.

Class Design Starts with the Operations

Each class defines an **interface** and **implementation**. The interface consists of the operations that we expect code that uses the class to execute. The implementation typically includes the data needed by the class. The implementation also includes any functions needed to define the class but that are not intended for general use.

When we define a class, we usually begin by defining its interface—the operations that the class will provide. From those operations we can then determine what data the class will require to accomplish its tasks and whether it will need to define any functions to support the implementation.

The operations our type will support are the operations we used in Chapter 1. These operations were outlined in Section 1.5.1 (p. 21):

64

- The addition operator to add two `Sales_item`s
- The input and output operators to read and write `Sales_item` objects
- The assignment operator to assign one `Sales_item` object to another
- The `same_isbn` function to determine if two objects refer to the same book

We'll see how to define these operations in Chapters 7 and 14 after we learn how to define functions and operators. Even though we can't yet implement these functions, we can figure out what data they'll need by thinking a bit about what these operations must do. Our `Sales_item` class must

1. Keep track of how many copies of a particular book were sold
2. Report the total revenue for that book
3. Calculate the average sales price for that book

Looking at this list of tasks, we can see that we'll need an `unsigned` to keep track of how many books are sold and a `double` to keep track of the total revenue. From these data we can calculate the average sales price as total revenue divided by number sold. Because we also want to know which book we're reporting on, we'll also need a `string` to keep track of the ISBN.

Defining the `Sales_item` Class

Evidently what we need is the ability to define a data type that will have these three data elements and the operations we used in Chapter 1. In C++, the way we define such a data type is to define a class:

```
class Sales_item {
public:
    // operations on Sales_item objects will go here
private:
    std::string isbn;
    unsigned units_sold;
    double revenue;
};
```

A class definition starts with the keyword `class` followed by an identifier that names the class. The body of the class appears inside curly braces. The close curly must be followed by a semicolon.

It is a common mistake among new programmers to forget the semicolon at the end of a class definition.

▶ 漏写分号是一个常见错误，而且错误信息不一定准确。特别是如果在头文件中漏写了 class 定义末尾的分号，那么出错的位置有可能是在包含这个头文件的 .cc 文件里，这时候要记得去检查头文件。本节练习 2.8 一定要上机实验。

The class body, which can be empty, defines the data and operations that make up the type. The operations and data that are part of a class are referred to as its **members**. The operations are referred to as the member functions (Section 1.5.2, p. 24) and the data as **data members**.

The class also may contain zero or more `public` or `private` **access labels**. An access label controls whether a member is accessible outside the class. Code that uses the class may access only the `public` members.

When we define a class, we define a new type. The class name is the name of that type. By naming our class `Sales_item` we are saying that `Sales_item` is a new type and that programs may define variables of this type.

Each class defines its own scope (Section 2.3.6, p. 54). That is, the names given to the data and operations inside the class body must be unique within the class but can reuse names defined outside the class.

Class Data Members

The data members of a class are defined in somewhat the same way that normal variables are defined. We specify a type and give the member a name just as we do when defining a simple variable:

```
std::string isbn;
unsigned units_sold;
double revenue;
```

Our class has three data members: a member of type `string` named `isbn`, an `unsigned` member named `units_sold`, and a member of type `double` named `revenue`. The data members of a class define the contents of the objects of that class type. When we define objects of type `Sales_item`, those objects will contain a `string`, an `unsigned`, and a `double`.

There is one crucially important difference between how we define variables and class data members: We ordinarily cannot initialize the members of a class as part of their definition. When we define the data members, we can only name them and say what types they have. Rather than initializing data members when they are defined inside the class definition, classes control initialization through special member functions called constructors (Section 2.3.3, p. 49). We will define the `Sales_item` constructors in Section 7.7.3 (p. 262).

Access Labels

Access labels control whether code that uses the class may use a given member. Member functions of the class may use any member of their own class, regardless of the access level. The access labels, `public` and `private`, may appear multiple times in a class definition. A given label applies until the next access label is seen.

The **public** section of a class defines members that can be accessed by any part of the program. Ordinarily we put the operations in the `public` section so that any code in the program may execute these operations.

Code that is not part of the class does not have access to the **private** members. By making the `Sales_item` data members `private`, we ensure that code that operates on `Sales_item` objects cannot directly manipulate the data members. Programs, such as the one we wrote in Chapter 1, may not access the `private` members of the class. Objects of type `Sales_item` may execute the operations but not change the data directly.

▶ 注意：虽然 private/public 通常被翻译为“私有 / 公有”，但其实它们的意思更接近“私用 / 公用”，因为 private/public 限制的不是所有权，而是使用权。无论是 private 还是 public 的成员，都是 class 私自拥有的。

66

Using the `struct` Keyword

C++ supports a second keyword, **`struct`**, that can be used to define class types. The `struct` keyword is inherited from C.

If we define a class using the `class` keyword, then any members defined before the first access label are implicitly `private`; if we use the `struct` keyword, then those members are `public`. Whether we define a class using the `class` keyword or the `struct` keyword affects only the default initial access level.

We could have defined our `Sales_item` equivalently by writing

```
struct Sales_item {
    // no need for public label, members are public by default
    // operations on Sales_item objects
private:
    std::string isbn;
    unsigned units_sold;
    double revenue;
};
```

There are only two differences between this class definition and our initial class definition: Here we use the `struct` keyword, and we eliminate the use of `public` keyword immediately following the opening curly brace. Members of a `struct` are public, unless otherwise specified, so there is no need for the `public` label.

The only difference between a class defined with the `class` keyword or the `struct` keyword is the default access level: By default, members in a `struct` are `public`; those in a `class` are `private`.

EXERCISES SECTION 2.8

Exercise 2.28: Compile the following program to determine whether your compiler warns about a missing semicolon after a class definition:

```
class Foo {
      // empty
} // Note: no semicolon
int main()
{
    return 0;
}
```

If the diagnostic is confusing, remember the message for future reference.

Exercise 2.29: Distinguish between the `public` and `private` sections of a class.

Exercise 2.30: Define the data members of classes to represent the following types:
(a) a phone number (b) an address
(c) an employee or a company (d) a student at a university

67

2.9 Writing Our Own Header Files

We know from Section 1.5 (p. 20) that ordinarily class definitions go into a **header file**. In this section we'll see how to define a header file for the `Sales_item` class.

In fact, C++ programs use headers to contain more than class definitions. Recall that every name must be declared or defined before it is used. The programs we've written so far handle this requirement by putting all their code into a single file. As long as each entity precedes the code that uses it, this strategy works. However, few programs are so simple that they can be written in a single file. Programs made up of multiple files need a way to link the use of a name and its declaration. In C++ that is done through header files.

To allow programs to be broken up into logical parts, C++ supports what is commonly known as ***separate compilation***. Separate compilation lets us compose a program from several files. To support separate compilation, we'll put the definition of `Sales_item` in a header file. The member functions for `Sales_item`, which we'll define in Section 7.7 (p. 258), will go in a separate source file. Functions such as `main` that use `Sales_item` objects are in other source files. Each of the source files that use `Sales_item` must include our `Sales_item.h` header file.

2.9.1 Designing Our Own Headers

A header provides a centralized location for related declarations. Headers normally contain class definitions, `extern` variable declarations, and function declarations, about which we'll learn in Section 7.4 (p. 251). Files that use or define these entities include the appropriate header(s).

Proper use of header files can provide two benefits: All files are guaranteed to use the same declaration for a given entity; and should a declaration require change, only the header needs to be updated.

Some care should be taken in designing headers. The declarations in a header should logically belong together. A header takes time to compile. If it is too large programmers may be reluctant to incur the compile-time cost of including it.

To reduce the compile time needed to process headers, some C++ implementations support precompiled header files. For more details, consult the reference manual of your C++ implementation.

Headers Are for Declarations, Not Definitions

When designing a header it is essential to remember the difference between definitions, which may only occur once, and declarations, which may occur multiple times (Section 2.3.5, p. 52). The following statements are definitions and therefore should not appear in a header:

```
extern int ival = 10;      // initializer, so it's a definition
double fica_rate;          // no extern, so it's a definition
```

Although `ival` is declared `extern`, it has an initializer, which means this statement is a definition. Similarly, the declaration of `fica_rate`, although it does not

COMPILING AND LINKING MULTIPLE SOURCE FILES 68

To produce an executable file, we must tell the compiler not only where to find our `main` function but also where to find the definition of the member functions defined by the `Sales_item` class. Let's assume that we have two files: `main.cc`, which contains the definition of `main`, and `Sales_item.cc`, which contains the `Sales_item` member functions. We might compile these files as follows:

```
$ CC -c main.cc Sales_item.cc # by default generates a.exe
                              # some compilers generate a.out


# puts the executable in main.exe
$ CC -c main.cc Sales_item.cc -o main
```

where `$` is our system prompt and `#` begins a command-line comment. We can now run the executable file, which will run our `main` program.

If we have only changed one of our `.cc` source files, it is more efficient to recompile only the file that actually changed. Most compilers provide a way to separately compile each file. This process usually yields a `.o` file, where the `.o` extension implies that the file contains object code.

The compiler lets us *link* object files together to form an executable. On the system we use, in which the compiler is invoked by a command named `CC`, we would compile our program as follows:

```
$ CC -c main.cc           # generates main.o
$ CC -c Sales_item.cc     # generates Sales_item.o
$ CC main.o Sales_item.o  # by default generates a.exe;
                          # some compilers generate a.out

# puts the executable in main.exe
$ CC main.o Sales_item.o -o main
```

You'll need to check with your compiler's user's guide to understand how to compile and execute programs made up of multiple source files.

Many compilers offer an option to enhance the error detection of the compiler. Check your compiler's user's guide to see what additional checks are available.

have an initializer, is a definition because the `extern` keyword is absent. Including either of these definitions in two or more files of the same program will result in a linker error complaining about multiple definitions.

Because headers are included in multiple source files, they should not contain definitions of variables or functions.

There are three exceptions to the rule that headers should not contain definitions: classes, `const` objects whose value is known at compile time, and `inline` functions (Section 7.6 (p. 256) covers `inline` functions) are all defined in headers. These entities may be defined in more than one source file as long as the definitions in each file are exactly the same.

69

These entities are defined in headers because the compiler needs their definitions (not just declarations) to generate code. For example, to generate code that defines or uses objects of a class type, the compiler needs to know what data members make up that type. It also needs to know what operations can be performed on these objects. The class definition provides the needed information. That `const` objects are defined in a header may require a bit more explanation.

Some `const` Objects Are Defined in Headers

Recall that by default a `const` variable (Section 2.4, p. 57) is local to the file in which it is defined. As we shall now see, the reason for this default is to allow `const` variables to be defined in header files.

In C++ there are places where constant expression (Section 2.7, p. 62) is required. For example, the initializer of an enumerator must be a constant expression. We'll see other cases that require constant expressions in later chapters.

Generally speaking, a constant expression is an expression that the compiler can evaluate at compile-time. A `const` variable of integral type may be a constant expression when it is itself initialized from a constant expression. However, for the `const` to be a constant expression, the initializer must be visible to the compiler. To allow multiple files to use the same constant value, the `const` and its initializer must be visible in each file. To make the initializer visible, we normally define such `const`s inside a header file. That way the compiler can see the initializer whenever the `const` is used.

However, there can be only one definition (Section 2.3.5, p. 52) for any variable in a C++ program. A definition allocates storage; all uses of the variable must refer to the same storage. Because, by default, `const` objects are local to the file in which they are defined, it is legal to put their definition in a header file.

There is one important implication of this behavior. When we define a `const` in a header file, every source file that includes that header has its own `const` variable with the same name and value.

When the `const` is initialized by a constant expression, then we are guaranteed that all the variables will have the same value. Moreover, in practice, most compilers will replace any use of such `const` variables by their corresponding constant expression at compile time. So, in practice, there won't be any storage used to hold `const` variables that are initialized by constant expressions.

When a `const` is initialized by a value that is not a constant expression, then it should not be defined in header file. Instead, as with any other variable, the `const` should be defined and initialized in a source file. An `extern` declaration for that `const` should be made in the header, enabling multiple files to share that variable.

▶ 例如英文原版书第 58 页的例子，以 fcn() 函数调用的返回值来初始化 const int bufSize，那么应该把初始化放到 .cc 文件中，以避免 fcn() 被反复调用。

2.9.2 A Brief Introduction to the Preprocessor

Now that we know what we want to put in our headers, our next problem is to actually write a header. We know that to use a header we have to `#include` it in our source file. In order to write our own headers, we need to understand a bit more about how a `#include` directive works. The `#include` facility is a part of the C++ **preprocessor**. The preprocessor manipulates the source text of our programs and runs before the compiler. C++ inherits a fairly elaborate preprocessor from C. Modern C++ programs use the preprocessor in a very restricted fashion.

A `#include` directive takes a single argument: the name of a header. The preprocessor replaces each `#include` by the contents of the specified header. Our own headers are stored in files. System headers may be stored in a compiler-specific format that is more efficient. Regardless of the form in which a header is stored, it ordinarily contains class definitions and declarations of the variables and functions needed to support separate compilation.

EXERCISES SECTION 2.9.1

70

Exercise 2.31: Identify which of the following statements are declarations and which ones are definitions. Explain why they are declarations or definitions.

```
(a) extern int ix = 1024;
(b) int iy;
(c) extern int iz;
(d) extern const int &ri;
```

Exercise 2.32: Which of the following declarations and definitions would you put in a header? In a source file? Explain why.

```
(a) int var;
(b) const double pi = 3.1416;
(c) extern int total = 255;
(d) const double sq2 = sqrt(2.0);
```

Exercise 2.33: Determine what options your compiler offers for increasing the warning level. Recompile selected earlier programs using this option to see whether additional problems are reported.

Headers Often Need Other Headers

Headers often `#include` other headers. The entities that a header defines often use facilities from other headers. For example, the header that defines our `Sales_item` class must include the `string` library. The `Sales_item` class has a `string` data member and so must have access to the `string` header.

Including other headers is so common that it is not unusual for a header to be included more than once in the same source file. For example, a program that used the `Sales_item` header might also use the `string` library. That program wouldn't—indeed shouldn't—know that our `Sales_item` header uses the

`string` library. In this case, the `string` header would be included twice: once by the program itself and once as a side-effect of including our `Sales_item` header.

Accordingly, it is important to design header files so that they can be included more than once in a single source file. We must ensure that including a header file
71 more than once does not cause multiple definitions of the classes and objects that the header file defines. A common way to make headers safe uses the preprocessor to define a **header guard**. The guard is used to avoid reprocessing the contents of a header file if the header has already been seen.

Avoiding Multiple Inclusions

Before we write our own header, we need to introduce some additional preprocessor facilities. The preprocessor lets us define our own variables.

Names used for preprocessor variables must be unique within the program. Any uses of a name that matches a preprocessor variable is assumed to refer to the preprocessor variable.

To help avoid name clashes, preprocessor variables usually are written in all uppercase letters.

A preprocessor variable has two states: defined or not yet defined. Various preprocessor directives define and test the state of preprocessor variables. The `#define` directive takes a name and defines that name as a preprocessor variable. The `#ifndef` directive tests whether the specified preprocessor variable has not yet been defined. If it hasn't, then everything following the `#ifndef` is processed up to the next `#endif`.

We can use these facilities to guard against including a header more than once:

```
#ifndef SALESITEM_H
#define SALESITEM_H
// Definition of Sales_item class and related functions goes here
#endif
```

The conditional directive

```
#ifndef SALESITEM_H
```

tests whether the `SALESITEM_H` preprocessor variable has *not* been defined. If `SALESITEM_H` has not been defined, the `#ifndef` succeeds and all the lines following `#ifndef` until the `#endif` is found are processed. Conversely, if the variable `SALESITEM_H` has been defined, then the `#ifndef` directive is false. The lines between it and the `#endif` directive are ignored.

To guarantee that the header is processed only once in a given source file, we start by testing the `#ifndef`. The first time the header is processed, this test will succeed, because `SALESITEM_H` will not yet have been defined. The next statement defines `SALESITEM_H`. That way, if the file we are compiling happens to include this header a second time, the `#ifndef` directive will discover that `SALESITEM_H` is defined and skip the rest of the header file.

Headers should have guards, even if they aren't included by another header. Header guards are trivial to write and can avoid mysterious compiler errors if the header subsequently is included more than once.

72 This strategy works well provided that no two headers define and use a preprocessor constant with the same name. We can avoid problems with duplicate preprocessor variables by naming them for an entity, such as a class, that is defined inside the header. A program can have only one class named `Sales_item`. By using the class name to compose the name of the header file and the preprocessor variable, we make it pretty likely that only one file will use this preprocessor variable.

Using Our Own Headers

The `#include` directive takes one of two forms:

```
#include <standard_header>
#include "my_file.h"
```

If the header name is enclosed by angle brackets (< >), it is presumed to be a standard header. The compiler will look for it in a predefined set of locations, which can be modified by setting a search path environment variable or through a command line option. The search methods used vary significantly across compilers. We recommend you ask a colleague or consult your compiler's user's guide for further information. If the header name is enclosed by a pair of quotation marks, the header is presumed to be a nonsystem header. The search for nonsystem headers usually begins in the directory in which the source file is located.

CHAPTER 5

EXPRESSIONS

CONTENTS

C++ provides a rich set of operators and defines what these operators do when applied to operands of built-in type. It also allows us to define meanings for the operators when applied to class types. This facility, known as operator overloading, is used by the library to define the operators that apply to the library types.

In this chapter our focus is on the operators as defined in the language and applied to operands of built-in type. We will also look at some of the operators defined by the library. Chapter 14 shows how we can define our own overloaded operators.

▶ C++ 语言里的算术表达式和其他一般编程语言没有多大差别，值得注意的是 C++ 独有的一些表达式。另外，表达式的求值顺序（order of evaluation）也值得注意，应该避免写出依赖于特定求值顺序的代码。

148

▶ 表达式由操作数（operands）和连接操作数的操作符（operators）组成；表达式可以没有操作符，但是不能没有操作数。

An expression is composed of one or more **operands** that are combined by **operators**. The simplest form of an **expression** consists of a single literal constant or variable. More complicated expressions are formed from an operator and one or more operands.

Every expression yields a **result**. In the case of an expression with no operator, the result is the operand itself, e.g., a literal constant or a variable. When an object is used in a context that requires a value, then the object is evaluated by fetching the object's value. For example, assuming `ival` is an `int` object,

```
if (ival)                    // evaluate ival as a condition
    // ....
```

we could use `ival` as an expression in the condition of an `if`. The condition succeeds if the value of `ival` is not zero and fails otherwise.

▶ 一个表达式除了有运算结果（result，即表达式的值），还可能有副作用（side effect）。我们在程序里写一个表达式，有可能是为了要它的结果，也有可能是为了要它的副作用，或者二者皆有。

The result of expressions that involve operators is determined by applying each operator to its operand(s). Except when noted otherwise, the result of an expression is an rvalue (Section 2.3.1, p. 45). We can read the result but cannot assign to it.

The meaning of an operator—what operation is performed and the type of the result—depends on the types of its operands.

▶C++ 里一个表达式的意义（类型、值、副作用）往往要由它的操作数的类型来决定，若脱离上下文单独地去看一个表达式，我们较难明白它的用处。

Until one knows the type of the operand(s), it is not possible to know what a particular expression means. The expression

```
i + j
```

might mean integer addition, concatenation of `strings`, floating-point addition, or something else entirely. How the expression is evaluated depends on the types of `i` and `j`.

▶C++ 既有一元操作符，也有二元操作符，还有一个特殊的三元操作符。

There are both ***unary*** operators and ***binary*** operators. Unary operators, such as address-of (`&`) and dereference (`*`), act on one operand. Binary operators, such as addition (`+`) and subtraction (`-`), act on two operands. There is also one ternary operator that takes three operands. We'll look at this operator in Section 5.7 (p. 165).

Some *symbols*, such as `*`, are used to represent both a unary and a binary operator. The `*` symbol is used as the (unary) dereference operator and as the (binary) multiplication operator. The uses of the symbol are independent; it can be helpful to think of them as two different symbols. The context in which an operator symbol is used always determines whether the symbol represents a unary or binary operator.

Operators impose requirements on the type(s) of their operand(s). The language defines the type requirements for the operators when applied to built-in or compound types. For example, the dereference operator, when applied to an object of built-in type, requires that its operand be a pointer type. Attempting to dereference an object of any other built-in or compound type is an error.

The binary operators, when applied to operands of built-in or compound type, usually require that the operands be the same type, or types that can be converted to a common type. We'll look at conversions in Section 5.12 (p. 178). Although the rules can be complex, for the most part conversions happen in expected ways. For example, we can convert an integer to floating-point, and vice versa, but we cannot convert a pointer type to floating-point.

149

▶ 如果一个表达式里有不止一个操作符，那么我们要先弄明白操作符的优先级、结合性，以及操作数的求值顺序。

Understanding expressions with multiple operators requires understanding operator ***precedence***, ***associativity***, and the ***order of evaluation*** of the operands. For example, the expression

```
5 + 10 * 20/2;
```

uses addition, multiplication, and division. The result of this expression depends on how the operands are grouped to the operators. For example, the operands to the `*` operator could be `10` and `20`, or `10` and `20/2`, or `15` and `20` or `15` and `20/2`. Associativity and precedence rules specify the grouping of operators and their operands. In C++ this expression evaluates to `105`, which is the result of multiplying `10` and `20`, dividing that result by `2`, and then adding `5`.

Knowing how operands and operators are grouped is not always sufficient to determine the result. It may also be necessary to know in what order the operands to each operator are evaluated. Each operator controls what assumptions, if any, can be made as to the order in which the operands will be evaluated—that is, whether we can assume that the left-hand operand is always evaluated before the right or not. Most operators do not guarantee a particular order of evaluation. We will cover these topics in Section 5.10 (p. 168).

5.1 Arithmetic Operators

Table 5.1: Arithmetic Operators

Operator	Function	Use
+	unary plus	`+ expr`
-	unary minus	`- expr`
*	multiplication	`expr * expr`
/	division	`expr / expr`
%	remainder	`expr % expr`
+	addition	`expr + expr`
-	subtraction	`expr - expr`

Unless noted otherwise, these operators may be applied to any of the arithmetic types (Section 2.1, p. 34), or any type that can be converted to an arithmetic type.

The table groups the operators by their precedence—the unary operators have the highest precedence, then the multiplication and division operators, and then the binary addition and subtraction operators. Operators of higher precedence group more tightly than do operators with lower precedence. These operators are all left associative, meaning that they group left to right when the precedence levels are the same.

▶ 算术表达式的优先级和结合性符合直觉，先乘除后加减，从左往右计算。

Applying precedence and associativity to the previous expression:

150

```
5 + 10 * 20/2;
```

we can see that the operands to the multiplication operator (`*`) are `10` and `20`. The result of that expression and `2` are the operands to the division operator (`/`). The result of that division and `5` are the operands to the addition operator (`+`).

The unary minus operator has the obvious meaning. It negates its operand:

```
int i = 1024;
int k = -i; // negates the value of its operand
```

Unary plus returns the operand itself. It makes no change to its operand.

CAUTION: OVERFLOW AND OTHER ARITHMETIC EXCEPTIONS

The result of evaluating some arithmetic expressions is undefined. Some expressions are undefined due to the nature of mathematics—for example, division by zero. Others are undefined due to the nature of computers—such as overflow, in which a value is computed that is too large for its type.

Consider a machine on which `short`s are 16 bits. In that case, the maximum `short` is 32767. Given only 16 bits, the following compound assignment overflows:

```
// max value if shorts are 8 bits
short short_value = 32767;
short ival = 1;
// this calculation overflows
```

▶ 这里有误，应该是：// max value if shorts are 16 bits。

```
short_value += ival;
cout << "short_value: " << short_value << endl;
```

Representing a signed value of 32768 requires 17 bits, but only 16 are available. On many systems, there is *no* compile-time or run-time warning when an overflow might occur. The actual value put into `short_value` varies across different machines. On our system the program completes and writes

```
short_value: -32768
```

The value "wrapped around:" The sign bit, which had been 0, was set to 1, resulting in a negative value. Because the arithmetic types have limited size, it is always possible for some calculations to overflow. Adhering to the recommendations from the "Advice" box on page 38 can help avoid such problems.

▶ 注意：整数乘除法的运算顺序可能改变运算结果。比如有两个 int 型变量 a 和 b，已知它们的最大公约数是 c，那么它们的最小公倍数是 a*b/c。但是这个表达式的中间结果 a*b 有溢出（即超过 int 的表示范围）的可能，所以我们一般把最小公倍数公式写成 a*(b/c) 或者 a/c*b，以避免中间结果溢出。

The binary + and - operators may also be applied to pointer values. The use of these operators with pointers was described in Section 4.2.4 (p. 123).

The arithmetic operators, `+`, `-`, `*`, and `/` have their obvious meanings: addition, subtraction, multiplication, and division. Division between integers results in an integer. If the quotient contains a fractional part, it is truncated:

```
int ival1 = 21/6;   // integral result obtained by truncating the remainder
int ival2 = 21/7;   // no remainder, result is an integral value
```

Both `ival1` and `ival2` are initialized with a value of 3.

151

The `%` operator is known as the "remainder" or the "modulus" operator. It computes the remainder of dividing the left-hand operand by the right-hand operand. This operator can be applied only to operands of the integral types: `bool`, `char`, `short`, `int`, `long`, and their associated `unsigned` types:

```
int ival = 42;
double dval = 3.14;
ival % 12;    // ok: returns 6
ival % dval; // error: floating point operand
```

▶ 根据定义，C++ 的整数除法与取模运算满足：被除数 = 商 * 除数 + 余数。

C++ 语言的整数除法是“截断”的，即两个整数相除仍然得到整数。对于被除数和除数都是正数的情况一般不会有什么疑问；如果被除数或除数有一个是负数，现版的 C++ 标准留给实现去决定余数的正负号和商的截断方向。如果余数的符号与被除数相同，那么商是向零截断；如果余数的符号与除数相同，那么商是向负无穷截断。

注意：新版的 C++ 标准要求余数的正负号和被除数相同，并且商向零取整，即 –21/5=–4 且 –21%5=–1。实际的硬件也都是按这个规则来实现整数除法的。

For both division (`/`) and modulus (`%`), when both operands are positive, the result is positive (or zero). If both operands are negative, the result of division is positive (or zero) and the result of modulus is negative (or zero). If only one operand is negative, then the value of the result is machine-dependent for both operators. The sign is also machine-dependent for modulus; the sign is negative (or zero) for division:

```
21 % 6;    // ok: result is 3
21 % 7;    // ok: result is 0
-21 % -8; // ok: result is -5
21 % -5;   // machine-dependent: result is 1 or -4
21 / 6;    // ok: result is 3
21 / 7;    // ok: result is 3
-21 / -8; // ok: result is 2
21 / -5;   // machine-dependent: result -4 or -5
```

When only one operand is negative, the sign and value of the result for the modulus operator can follow either the sign of the numerator or of the denominator. On a machine where modulus follows the sign of the numerator then the value of division truncates toward zero. If modulus matches the sign of the denominator, then the result of division truncates toward minus infinity.

EXERCISES SECTION 5.1

Exercise 5.1: Parenthesize the following expression to indicate how it is evaluated. Test your answer by compiling the expression and printing its result.

```
12 / 3 * 4 + 5 * 15 + 24 % 4 / 2
```

Exercise 5.2: Determine the result of the following expressions and indicate which

results, if any, are machine-dependent.

```
-30 * 3 + 21 / 5
-30 + 3 * 21 / 5
30 / 3 * 21 % 5
-30 / 3 * 21 % 4
```

Exercise 5.3: Write an expression to determine whether an `int` value is even or odd.

Exercise 5.4: Define the term overflow. Show three expressions that will overflow.

5.2 Relational and Logical Operators

152

Table 5.2: Relational and Logical Operators

Each of these operators yields `bool`

Operator	Function	Use
`!`	logical NOT	`!expr`
`<`	less than	`expr < expr`
`<=`	less than or equal	`expr <= expr`
`>`	greater than	`expr > expr`
`>=`	greater than or equal	`expr >= expr`
`==`	equality	`expr == expr`
`!=`	inequality	`expr != expr`
`&&`	logical AND	`expr && expr`
`||`	logical OR	`expr || expr`

The relational and logical operators take operands of arithmetic or pointer type and return values of type `bool`.

▶ 表 5.2 列出了关系与逻辑操作符的五种优先级，值得牢记。

Logical AND and OR Operators

The logical operators treat their operands as conditions (Section 1.4.1, p. 12). The operand is evaluated; if the result is zero the condition is `false`, otherwise it is `true`. The overall result of the AND operator is `true` if and only if both its operands evaluate to `true`. The logical OR (`||`) operator evaluates to `true` if either of its operands evaluates to `true`. Given the forms

```
expr1 && expr2   // logical AND
expr1 || expr2   // logical OR
```

expr2 is evaluated if and only if *expr1* does not by itself determine the result. In other words, we're guaranteed that *expr2* will be evaluated if and only if

- In a logical AND expression, *expr1* evaluates to `true`. If *expr1* is `false`, then the expression will be `false` regardless of the value of *expr2*. When *expr1* is `true`, it is possible for the expression to be `true` if *expr2* is also `true`.
- In a logical OR expression, *expr1* evaluates to `false`; if *expr1* is `false`, then the expression depends on whether *expr2* is `true`.

The logical AND and OR operators always evaluate their left operand before the right. The right operand is evaluated only if the left operand does not determine the result. This evaluation strategy is often referred to as "short-circuit evaluation."

153 A valuable use of the logical AND operator is to have *expr1* evaluate to false in the presence of a boundary condition that would make the evaluation of *expr2* dangerous. As an example, we might have a `string` that contains the characters in a sentence and we might want to make the first word in the sentence all uppercase. We could do so as follows:

```
string s("Expressions in C++ are composed...");
string::iterator it = s.begin();
//  convert first word in s to uppercase
while (it != s.end() && !isspace(*it)) {
    *it = toupper(*it);  //  toupper covered in section 3.2.4 (p. 88)
    ++it;
}
```

In this case, we combine our two tests in the condition in the `while`. First we test whether `it` has reached the end of the `string`. If not, `it` refers to a character in `s`. Only if that test succeeds is the right-hand operand evaluated. We're guaranteed that `it` refers to an actual character before we test to see whether the character is a space or not. The loop ends either when a space is encountered or, if there are no spaces in `s`, when we reach the end of `s`.

Logical NOT Operator

The logical NOT operator (`!`) treats its operand as a condition. It yields a result that has the opposite truth value from its operand. If the operand evaluates as nonzero, then `!` returns `false`. For example, we might determine that a `vector` has elements by applying the logical NOT operator to the value returned by `empty`:

```
//  assign value of first element in vec to x if there is one
int x = 0;
if (!vec.empty())
    x = *vec.begin();
```

▶ 一般把“!”读做 not，if (!vec.empty()) 读做 if not vec.empty()，表示 if vec is not empty。

The subexpression

```
!vec.empty()
```

evaluates to `true` if the call to `empty` returns `false`.

The Relational Operators Do Not Chain Together

The relational operators (`<`, `<=`, `>`, `<=`) are left associative. The fact that they are left associative is rarely of any use because the relational operators return `bool` results. If we do chain these operators together, the result is likely to be surprising:

```
//  oops! this condition does not determine if the 3 values are unequal
if (i < j < k) { /* ... */ }
```

As written, this expression will evaluate as `true` if `k` is greater than one! The reason is that the left operand of the second less-than operator is the `true/false` result of the first—that is, the condition compares `k` to the integer values of 0 or 1. To accomplish the test we intended, we must rewrite the expression as follows:

```
if (i < j && j < k) { /* ... */ }
```

154 ### Equality Tests and the `bool` Literals

As we'll see in Section 5.12.2 (p. 180) a `bool` can be converted to any arithmetic type—the `bool` value `false` converts to zero and `true` converts to one.

Because `bool` converts to one, is almost never right to write an equality test that tests against the `bool` literal `true`:

```
if (val == true) { /* ... */ }
```

Either `val` is itself a `bool` or it is a type to which a `bool` can be converted. If `val` is a `bool`, then this test is equivalent to writing

```
if (val) { /* ... */ }
```

which is shorter and more direct (although admittedly when first learning the language this kind of abbreviation can be perplexing).

More importantly, if `val` is not a `bool`, then comparing `val` with `true` is equivalent to writing

```
if (val == 1) { /* ... */ }
```

which is very different from

```
// condition succeeds if val is any nonzero value
if (val) { /* ... */ }
```

in which any nonzero value in `val` is true. If we write the comparison explicitly, then we are saying that the condition will succeed only for the specific value 1.

EXERCISES SECTION 5.2

Exercise 5.5: Explain when operands are evaluated in the logical AND operator, logical OR operator, and equality operator.

Exercise 5.6: Explain the behavior of the following `while` condition:

```
char *cp = "Hello World";
while (cp && *cp)
```

Exercise 5.7: Write the condition for a `while` loop that would read `int`s from the standard input and stop when the value read is equal to `42`.

Exercise 5.8: Write an expression that tests four values, `a`, `b`, `c`, and `d`, and ensures that `a` is greater than `b`, which is greater than `c`, which is greater than `d`.

5.3 The Bitwise Operators

The bitwise operators take operands of integral type. These operators treat their integral operands as a collection of bits, providing operations to test and set individual bits. In addition, these operators may be applied to `bitset` (Section 3.5, p. 101) operands with the behavior as described here for integral operands.

Table 5.3: Bitwise Operators

Operator	Function	Use
~	bitwise NOT	`~expr`
<<	left shift	`expr1 << expr2`
>>	right shift	`expr1 >> expr2`
&	bitwise AND	`expr1 & expr2`
^	bitwise XOR	`expr1 ^ expr2`
|	bitwise OR	`expr1 | expr2`

The type of an integer manipulated by the bitwise operators can be either signed or unsigned. If the value is negative, then the way that the "sign bit" is handled in a number of the bitwise operations is machine-dependent. It is, therefore, likely to differ across implementations; programs that work under one implementation may fail under another.

▶ 一般只有在非常底层的编程中才会用到按位操作符。

我注意到有些初学者过分关注微观（语句级）性能，比方说关心 while(true) 和 for(;;) 哪个更快，++i 与 i=i+1 哪个更快，i/=16 和 i >>= 4 哪个快，等等。而且受一些过时的教科书的影响，他们往往以为使用位运算代表“高效”，并为此牺牲代码的清晰性。我认为这是很不值得的。

155

▶ 注意：按位操作符的优先级很特殊且违反直觉，在与其他操作符混用时或许会有意想不到的结果。比如 y = x << 4 + 1，本意是将 x 左移 4 位后再加一，但实际却是将 x 左移了 5 位。因此，建议多使用小括号。

Because there are no guarantees for how the sign bit is handled, we strongly recommend using an `unsigned` type when using an integral value with the bitwise operators.

In the following examples we assume that an `unsigned char` has 8 bits.

The bitwise NOT operator (`~`) is similar in behavior to the `bitset flip` (Section 3.5.2, p. 105) operation: It generates a new value with the bits of its operand inverted. Each 1 bit is set to 0; each 0 bit is set to 1:

```
unsigned char bits = 0227;
bits = ~bits;
```

1	0	0	1	0	1	1	1
0	1	1	0	1	0	0	0

The `<<`, `>>` operators are the bitwise shift operators. These operators use their right-hand operand to indicate by how many bits to shift. They yield a value that is a copy of the left-hand operand with the bits shifted as directed by the right-hand operand. The bits are shifted left (`<<`) or right (`>>`), discarding the bits that are shifted off the end.

▶ 据勘误，第一句应该是 unsigned char bits = 0233;

```
unsigned char bits = 1;
bits << 1; // left shift
bits << 2; // left shift
bits >> 3; // right shift
```

1	0	0	1	1	0	1	1
0	0	1	1	0	1	1	0
0	1	1	0	1	1	0	0
0	0	0	1	0	0	1	1

The left shift operator (`<<`) inserts 0-valued bits in from the right. The right shift operator (`>>`) inserts 0-valued bits in from the left if the operand is unsigned. 156 If the operand is signed, it can either insert copies of the sign bit or insert 0-valued bits; which one it uses is implementation defined. The right-hand operand must not be negative and must be a value that is strictly less than the number of bits in the left-hand operand. Otherwise, the effect of the operation is undefined.

The bitwise AND operator (`&`) takes two integral operands. For each bit position, the result is 1 if both operands contain 1; otherwise, the result is 0.

It is a common error to confuse the bitwise AND operator (`&`) with the logical AND operator (`&&`) (Section 5.2, p. 152). Similarly, it is common to confuse the bitwise OR operator (`|`) and the logical OR operator(`||`).

Here we illustrate the result of bitwise AND of two `unsigned char` values, each of which is initialized by an octal literal:

```
unsigned char b1 = 0145;
unsigned char b2 = 0257;
unsigned char result = b1 & b2;
```

0	1	1	0	0	1	0	1
1	0	1	0	1	1	1	1
0	0	1	0	0	1	0	1

The bitwise XOR (exclusive or) operator (`^`) also takes two integral operands. For each bit position, the result is 1 if either but not both operands contain 1; otherwise, the result is 0.

```
result = b1 ^ b2;
```

1	1	0	0	1	0	1	0

The bitwise OR (inclusive or) operator (`|`) takes two integral operands. For each bit position, the result is 1 if either or both operands contain 1; otherwise, the result is 0.

```
result = b1 | b2;
```

1	1	1	0	1	1	1	1

5.3.1 Using `bitset` Objects or Integral Values

We said that the `bitset` class was easier to use than the lower-level bitwise operations on integral values. Let's look at a simple example and show how we might solve a problem using either a `bitset` or the bitwise operators. Assume that a teacher has 30 students in a class. Each week the class is given a pass/fail quiz. We'll track the results of each quiz using one bit per student to represent the pass or fail grade on a given test. We might represent each quiz in either a `bitset` or as an integral value:

```
bitset<30> bitset_quiz1;       // bitset solution
unsigned long int_quiz1 = 0;   // simulated collection of bits
```

In the `bitset` case we can define `bitset_quiz1` to be exactly the size we need. By default each of the bits is set to zero. In the case where we use a built-in type to hold our quiz results, we define `int_quiz1` as an `unsigned long`, meaning that it will have at least 32 bits on any machine. Finally, we explicitly initialize 157
`int_quiz1` to ensure that the bits start out with well-defined values.

The teacher must be able to set and test individual bits. For example, assuming that the student represented by position 27 passed, we'd like to be able to set that bit appropriately:

```
bitset_quiz1.set(27);    // indicate student number 27 passed
int_quiz1 |= 1UL<<27;    // indicate student number 27 passed
```

In the `bitset` case we do so directly by passing the bit we want turned on to `set`. The `unsigned long` case will take a bit more explanation. The way we'll set a specific bit is to OR our quiz data with another integer that has only one bit—the one we want—turned on. That is, we need an `unsigned long` where bit 27 is a one and all the other bits are zero. We can obtain such a value by using the left shift operator and the integer constant 1:

```
1UL << 27;   // generate a value with only bit number 27 set
```

Now when we bitwise OR this value with `int_quiz1`, all the bits except bit 27 will remain unchanged. That bit will be turned on. We use a compound assignment (Section 1.4.1, p. 13) to OR this value into `int_quiz1`. This operator, `|=`, executes in the same way that `+=` does. It is equivalent to the more verbose:

```
// following assignment is equivalent to int_quiz1 |= 1UL << 27;
int_quiz1 = int_quiz1 | 1UL << 27;
```

Imagine that the teacher reexamined the quiz and discovered that student 27 actually had failed the test. The teacher must now turn off bit 27:

```
bitset_quiz1.reset(27);    // student number 27 failed
int_quiz1 &= ~(1UL<<27);   // student number 27 failed
```

Again, the `bitset` version is direct. We `reset` the indicated bit. For the simulated case, we need to do the inverse of what we did to set the bit: This time we'll need an integer that has bit 27 turned off and all the other bits turned on. We'll bitwise AND this value with our quiz data to turn off just that bit. We can obtain a value with all but bit 27 turned on by inverting our previous value. Applying the bitwise NOT to the previous integer will turn on every bit except the 27th. When we bitwise AND this value with `int_quiz1`, all except bit 27 will remain unchanged.

Finally, we might want to know how the student at position 27 fared. To do so, we could write

```
bool status;
status = bitset_quiz1[27];       // how did student number 27 do?
status = int_quiz1 & (1UL<<27); // how did student number 27 do?
```

In the `bitset` case we can fetch the value directly to determine how that student did. In the `unsigned long` case, the first step is to set the 27th bit of an integer to 1. The bitwise AND of this value with `int_quiz1` evaluates to nonzero if bit 27 of `int_quiz1` is also on; otherwise, it evaluates to zero.

158

In general, the library `bitset` operations are more direct, easier to read, easier to write, and more likely to be used correctly. Moreover, the size of a `bitset` is not limited by the number of bits in an `unsigned`. Ordinarily `bitset` should be used in preference to lower-level direct bit manipulation of integral values.

EXERCISES SECTION 5.3.1

Exercise 5.9: Assume the following two definitions:

```
unsigned long ul1 = 3, ul2 = 7;
```

What is the result of each of the following expressions?

```
(a)  ul1 & ul2          (c)  ul1 | ul2
(b)  ul1 && ul2         (d)  ul1 || ul2
```

Exercise 5.10: Rewrite the `bitset` expressions that set and reset the quiz results using a subscript operator.

5.3.2 Using the Shift Operators for IO

The IO library redefines the bitwise `>>` and `<<` operators to do input and output. Even though many programmers never need to use the bitwise operators directly, most programs do make extensive use of the overloaded versions of these operators for IO. When we use an overloaded operator, it has the same precedence and associativity as is defined for the built-in version of the operator. Therefore, programmers need to understand the precedence and associativity of these operators even if they never use them with their built-in meaning as the shift operators.

The IO Operators Are Left Associative

Like the other binary operators, the shift operators are left associative. These operators group from left to right, which accounts for the fact that we can concatenate input and output operations into a single statement:

```
cout << "hi" << " there" << endl;
```

executes as:

```
( (cout << "hi") << " there" ) << endl;
```

In this statement, the operand `"hi"` is grouped with the first `<<` symbol. Its result is grouped with the second, and then that result is grouped to the third.

The shift operators have midlevel precedence: lower precedence than the arithmetic operators but higher than the relational, assignment, or conditional operators. These relative precedence levels affect how we write IO expressions involving
159 operands that use operators with lower precedence. We often need to use parentheses to force the right grouping:

```
cout << 42 + 10;    // ok, + has higher precedence, so the sum is printed
cout << (10 < 42); // ok: parentheses force intended grouping; prints 1
cout << 10 < 42;    // error: attempt to compare cout to 42!
```

The second `cout` is interpreted as

```
(cout << 10) < 42;
```

this expression says to "write 10 onto `cout` and then compare the result of that operation (e.g., `cout`) to 42."

5.4 Assignment Operators

The left-hand operand of an assignment operator must be a nonconst lvalue. Each of these assignments is illegal:

```
int i, j, ival;
const int ci = i; // ok: initialization not assignment
1024 = ival;      // error: literals are rvalues
i + j = ival;     // error: arithmetic expressions are rvalues
ci = ival;        // error: can't write to ci
```

Array names are nonmodifiable lvalues: An array cannot be the target of an assignment. Both the subscript and dereference operators return lvalues. The result of dereference or subscript, when applied to a nonconst array, can be the left-hand operand of an assignment:

```
int ia[10];
ia[0] = 0;    // ok: subscript is an lvalue
*ia = 0;      // ok: dereference also is an lvalue
```

The result of an assignment is the left-hand operand; the type of the result is the type of the left-hand operand.

The value assigned to the left-hand operand ordinarily is the value that is in the right-hand operand. However, assignments where the types of the left and right operands differ may require conversions that might change the value being assigned. In such cases, the value stored in the left-hand operand might differ from the value of the right-hand operand:

```
ival = 0;         // result: type int value 0
ival = 3.14159;   // result: type int value 3
```

Both these assignments yield values of type `int`. In the first case the value stored in `ival` is the same value as in its right-hand operand. In the second case the value stored in `ival` is different from the right-hand operand.

▶ 在C++中，赋值是一种表达式，其值是（赋完值之后的）左侧操作数，其类型与左侧操作数（也就是被赋值的对象）相同，其副作用是改变左侧操作数的值。这是本章出现的第一个有副作用的表达式，而前面讲的表达式都不会改变操作数本身（重载之后用于输出的左移表达式除外）。

160

5.4.1 Assignment Is Right Associative

Like the subscript and dereference operators, assignment returns an lvalue. As such, we can perform multiple assignments in a single expression, provided that each of the operands being assigned is of the same general type:

```
int ival, jval;
ival = jval = 0; // ok: each assigned 0
```

Unlike the other binary operators, the assignment operators are right associative. We group an expression with multiple assignment operators from right to left. In this expression, the result of the rightmost assignment (i.e., `jval`) is assigned to `ival`. The types of the objects in a multiple assignment either must be the same type or of types that can be converted (Section 5.12, p. 178) to one another:

```
int ival; int *pval;
ival = pval = 0; // error: cannot assign the value of a pointer to an int
string s1, s2;
s1 = s2 = "OK";  // ok: "OK" converted to string
```

The first assignment is illegal because `ival` and `pval` are objects of different types. It is illegal even though zero happens to be a value that could be assigned to either object. The problem is that the result of the assignment to `pval` is a value of type `int*`, which cannot be assigned to an object of type `int`. On the other hand, the second assignment is fine. The string literal is converted to `string`, and that `string` is assigned to `s2`. The result of that assignment is `s2`, which is then assigned to `s1`.

5.4.2 Assignment Has Low Precedence

Inside a condition is another common place where assignment is used as a part of a larger expression. Writing an assignment in a condition can shorten programs and clarify the programmer's intent. For example, the following loop uses a function named `get_value`, which we assume returns `int` values. We can test those values until we obtain some desired value—say, 42:

```
int i = get_value();  // get_value returns an int
while (i != 42) {
    // do something ...
    i = get_value();
}
```

The program begins by getting the first value and storing it in `i`. Then it establishes the loop, which tests whether `i` is 42, and if not, does some processing. The last statement in the loop gets a value from `get_value()`, and the loop repeats. We can write this loop more succinctly as

```
int i;
while ((i = get_value()) != 42) {
    // do something ...
}
```

161 The condition now more clearly expresses our intent: We want to continue until `get_value` returns 42. The condition executes by assigning the result returned by `get_value` to `i` and then comparing the result of that assignment with 42.

The additional parentheses around the assignment are necessary because assignment has lower precedence than inequality.

Without the parentheses, the operands to `!=` would be the value returned from calling `get_value` and `42`. The `true` or `false` result of that test would be assigned to `i`—clearly not what we intended!

Beware of Confusing Equality and Assignment Operators

The fact that we can use assignment in a condition can have surprising effects:

```
if (i = 42)
```

This code is legal: What happens is that 42 is assigned to `i` and then the result of the assignment is tested. In this case, 42 is nonzero, which is interpreted as a `true` value. The author of this code almost surely intended to test whether `i` was 42:

```
if (i == 42)
```

▶ 出现这种错误的根本原因是 C++ 让“赋值”成为表达式。好在现在的编译器大多能帮我们发现这种低级错误。[CCS, item 1]：在最高警告级别下干净地编译。

Bugs of this sort are notoriously difficult to find. Some, but not all, compilers are kind enough to warn about code such as this example.

EXERCISES SECTION 5.4.2

Exercise 5.11: What are the values of `i` and `d` after the each assignment:

```
int i;   double d;
```

```
d = i = 3.5;
i = d = 3.5;
```

Exercise 5.12: Explain what happens in each of the `if` tests:

```
if (42 = i)    // . . .
if (i = 42)    // . . .
```

5.4.3 Compound Assignment Operators

We often apply an operator to an object and then reassign the result to that same object. As an example, consider the sum program from page 14:

```
int sum = 0;
// sum values from 1 up to 10 inclusive
for (int val = 1; val <= 10; ++val)
    sum += val;  // equivalent to sum = sum + val
```

This kind of operation is common not just for addition but for the other arithmetic operators and the bitwise operators. There are compound assignments for each of these operators. The general syntactic form of a compound assignment operator is

162

```
a op= b;
```

where *op*= may be one of the following ten operators:

```
 +=    -=    *=    /=    %=        // arithmetic operators
<<=   >>=   &=    ^=    |=        // bitwise operators
```

▶ 注意：复合赋值操作符不包括“逻辑与（&&）”和“逻辑或（||）”，即 a = a && b 不能写成 a &&= b。另外，等号之前不能有空格。

Each compound operator is essentially equivalent to

```
a = a op b;
```

There is one important difference: When we use the compound assignment, the left-hand operand is evaluated only once. If we write the similar longer version, that operand is evaluated twice: once as the right-hand operand and again as the left. In many, perhaps most, contexts this difference is immaterial aside from possible performance consequences.

EXERCISES SECTION 5.4.3

Exercise 5.13: The following assignment is illegal. Why? How would you correct it?

```
double dval; int ival; int *pi;
dval = ival = pi = 0;
```

Exercise 5.14: Although the following are legal, they probably do not behave as the programmer expects. Why? Rewrite the expressions as you think they should be.

```
(a) if (ptr = retrieve_pointer() != 0)
(b) if (ival = 1024)
(c) ival += ival + 1;
```

5.5 Increment and Decrement Operators

The increment (++) and decrement (--) operators provide a convenient notational shorthand for adding or subtracting 1 from an object. There are two forms of these operators: prefix and postfix. So far, we have used only the prefix increment, which increments its operand and yields the *changed* value as its result. The prefix decrement operates similarly, except that it decrements its operand. The postfix versions of these operators increment (or decrement) the operand but yield a copy of the original, *unchanged* value as its result:

▶ 注意：递增与递减操作符除了有值，还有副作用。这两个操作符会改变操作数本身的值。

从语法上说，++ 和 -- 是一个字元（token），不能分开写。分开写的话相当于两个一元 + 或 - 操作符。

```
int i = 0, j;
j = ++i; //  j = 1, i = 1: prefix yields incremented value
j = i++; //  j = 1, i = 2: postfix yields unincremented value
```

▶ 注意：C++ 规定前缀递增操作符返回的是左值，这与 C 语言的规定不同。

Because the prefix version returns the incremented value, it returns the object itself as an lvalue. The postfix versions return an rvalue.

163

ADVICE: USE POSTFIX OPERATORS ONLY WHEN NECESSARY

Readers from a C background might be surprised that we use the prefix increment in the programs we've written. The reason is simple: The prefix version does less work. It increments the value and returns the incremented version. The postfix operator must store the original value so that it can return the unincremented value as its result. For `int`s and pointers, the compiler can optimize away this extra work. For more complex iterator types, this extra work potentially could be more costly. By habitually favoring the use of the prefix versions, we do not have to worry if the performance difference matters.

Postfix Operators Return the Unincremented Value

The postfix version of `++` and `--` is used most often when we want to use the current value of a variable and increment it in a single compound expression:

```
vector<int> ivec;                  //  empty vector
int cnt = 10;
//  add elements 10...1 to ivec
while (cnt > 0)
    ivec.push_back(cnt--);          //  int postfix decrement
```

This program uses the postfix version of `--` to decrement `cnt`. We want to assign the value of `cnt` to the next element in the `vector` and then decrement `cnt` before the next iteration. Had the loop used the prefix version, then the decremented value of `cnt` would be used when creating the elements in `ivec` and the effect would be to add elements from `9` down to `0`.

▶ 注意：最后这句是虚拟语气：倘若把代码中的后缀递减改为前缀递减，其效果是把［9..0］这 10 个元素压入 ivec。

Combining Dereference and Increment in a Single Expression

The following program, which prints the contents of `ivec`, represents a very common C++ programming pattern:

```
vector<int>::iterator iter = ivec.begin();
//  prints 10 9 8 ... 1
while (iter != ivec.end())
    cout << *iter++ << endl; //  iterator postfix increment
```

The expression `*iter++` is usually very confusing to programmers new to both C++ and C.

The precedence of postfix increment is higher than that of the dereference operator, so `*iter++` is equivalent to `*(iter++)`. The subexpression `iter++` increments `iter` and yields a copy of the previous value of `iter` as its result. Accordingly, the operand of `*` is a copy of the unincremented value of `iter`.

This usage relies on the fact that postfix increment returns a copy of its original, unincremented operand. If it returned the incremented value, we'd dereference the incremented value, with disastrous results: The first element of `ivec` would not get written. Worse, we'd attempt to dereference one too many elements!

164

ADVICE: BREVITY CAN BE A VIRTUE

Programmers new to C++ who have not previously programmed in a C-based language often have trouble with the terseness of some expressions. In particular, expressions such as `*iter++` can be bewildering—at first. Experienced C++ programmers value being concise. They are much more likely to write

```
cout << *iter++ << endl;
```

than the more verbose equivalent

```
cout << *iter << endl;
++iter;
```

For programmers new to C++, the second form is clearer because the action of incrementing the iterator and fetching the value to print are kept separate. However, the first version is much more natural to most C++ programmers.

It is worthwhile to study examples of such code until their meanings are immediately clear. Most C++ programs use succinct expressions rather than more verbose equivalents. Therefore, C++ programmers must be comfortable with such usages. Moreover, once these expressions are familiar, you will find them less error-prone.

▶C++ 程序员应熟悉这种常见写法。值得花点时间锻炼，使自己一眼就能看懂这种复杂表达式的意思。

EXERCISES SECTION 5.5

Exercise 5.15: Explain the difference between prefix and postfix increment.

Exercise 5.16: Why do you think C++ wasn't named ++C?

Exercise 5.17: What would happen if the `while` loop that prints the contents of a `vector` used the prefix increment operator?

5.6 The Arrow Operator

▶箭头操作符要求操作数是个指针或类似指针的东西（迭代器或智能指针）。

The arrow operator (`->`) provides a synonym for expressions involving the dot and dereference operators. The dot operator (Section 1.5.2, p. 25) fetches an element from an object of class type:

```
item1.same_isbn(item2);  // run the same_isbn member of item1
```

If we had a pointer (or iterator) to a `Sales_item`, we would have to dereference the pointer (or iterator) before applying the dot operator:

```
Sales_item *sp = &item1;
(*sp).same_isbn(item2);  // run same_isbn on object to which sp points
```

Here we dereference `sp` to get the underlying `Sales_item`. Then we use the dot operator to run `same_isbn` on that object. We must parenthesize the dereference because dereference has a lower precedence than dot. If we omit the parentheses, this code means something quite different:

```
// run the same_isbn member of sp then dereference the result!
*sp.same_isbn(item2);     // error: sp has no member named same_isbn
```

This expression attempts to fetch the `same_isbn` member of the object `sp`. It is equivalent to 165

```
*(sp.same_isbn(item2));  // equivalent to *sp.same_isbn(item2);
```

However, `sp` is a pointer, which has no members; this code will not compile.

Because it is easy to forget the parentheses and because this kind of code is a common usage, the language defines the arrow operator as a synonym for a dereference followed by the dot operator. Given a pointer (or iterator) to an object of class type, the following expressions are equivalent:

```
(*p).foo;  // dereference p to get an object and fetch its member named foo
p->foo;    // equivalent way to fetch the foo from the object to which p points
```

More concretely, we can rewrite the call to `same_isbn` as

```
sp->same_isbn(item2);     // equivalent to (*sp).same_isbn(item2)
```

EXERCISES SECTION 5.6

Exercise 5.18: Write a program that defines a `vector` of pointers to `string`s. Read the `vector`, printing each `string` and its corresponding size.

Exercise 5.19: Assuming that `iter` is a `vector<string>::iterator`, indicate which, if any, of the following expressions is legal. Explain the behavior of the legal expressions.

```
(a) *iter++;              (b) (*iter)++;
(c) *iter.empty()         (d) iter->empty();
(e) ++*iter;              (f) iter++->empty();
```

5.7 The Conditional Operator

The **conditional operator** is the only ternary operator in C++. It allows us to embed simple if-else tests inside an expression. The conditional operator has the following syntactic form

```
cond ? expr1 : expr2;
```

where *cond* is an expression that is used as a condition (Section 1.4.1, p. 12). The operator executes by evaluating *cond*. If *cond* evaluates to 0, then the condition is `false`; any other value is `true`. *cond* is always evaluated. If it is `true`, then *expr1* is evaluated; otherwise, *expr2* is evaluated. Like the logical AND and OR (`&&` and `||`) operators, the conditional operator guarantees this order of evaluation for its operands. Only one of *expr1* or *expr2* is evaluated. The following program illustrates use of the conditional operator:

▶ 注意：不要写出 (a > b) ? true : false 这种多余的条件操作符，直接写 (a > b) 即可。

```
int i = 10, j = 20, k = 30;
// if i > j then maxVal = i else maxVal = j
int maxVal = i > j ? i : j;
```

166

Avoid Deep Nesting of the Conditional Operator

We could use a set of nested conditional expressions to set `max` to the largest of three variables:

```
int max = i > j
              ? i > k ? i : k
              : j > k ? j : k;
```

We could do the equivalent comparison in the following longer but simpler way:

```
int max = i;
if (j > max)
    max = j;
if (k > max)
    max = k;
```

Using a Conditional Operator in an Output Expression

The conditional operator has fairly low precedence. When we embed a conditional expression in a larger expression, we usually must parenthesize the conditional subexpression. For example, the conditional operator is often used to print one or another value, depending on the result of a condition. Incompletely parenthesized uses of the conditional operator in an output expression can have surprising results:

```
cout << (i < j ? i : j);   // ok: prints larger of i and j
cout << (i < j) ? i : j;   // prints 1 or 0!
cout << i < j ? i : j;     // error: compares cout to int
```

The second expression is the most interesting: It treats the comparison between `i` and `j` as the operand to the `<<` operator. The value 1 or 0 is printed, depending on whether `i < j` is true or false. The `<<` operator returns `cout`, which is tested as the condition for the conditional operator. That is, the second expression is equivalent to

```
cout << (i < j); // prints 1 or 0
cout ? i : j;    // test cout and then evaluate i or j
                 // depending on whether cout evaluates to true or false
```

EXERCISES SECTION 5.7

Exercise 5.20: Write a program to prompt the user for a pair of numbers and report which is smaller.

Exercise 5.21: Write a program to process the elements of a `vector<int>`. Replace each element with an odd value by twice that value.

5.8 The `sizeof` Operator

167

The `sizeof` operator returns a value of type `size_t` (Section 3.5.2, p. 104) that is the size, in bytes (Section 2.1, p. 35), of an object or type name. The result of `sizeof` expression is a compile-time constant. The `sizeof` operator takes one of the following forms:

```
sizeof (type name);
sizeof (expr);
sizeof expr;
```

Applying `sizeof` to an *expr* returns the size of the result type of that expression:

```
Sales_item item, *p;
// three ways to obtain size required to hold an object of type Sales_item
sizeof(Sales_item); // size required to hold an object of type Sales_item
sizeof item; // size of item's type, e.g., sizeof(Sales_item)
sizeof *p;   // size of type to which p points, e.g., sizeof(Sales_item)
```

Evaluating `sizeof` *expr* does not evaluate the expression. In particular, in `sizeof *p`, the pointer `p` may hold an invalid address, because `p` is not dereferenced.

The result of applying `sizeof` depends in part on the type involved:

- `sizeof char` or an expression of type `char` is guaranteed to be 1
- `sizeof` a reference type returns the size of the memory necessary to contain an object of the referenced type
- `sizeof` a pointer returns the size needed hold a pointer; to obtain the size of the object to which the pointer points, the pointer must be dereferenced
- `sizeof` an array is equivalent to taking the `sizeof` the element type times the number of elements in the array

Because `sizeof` returns the size of the entire array, we can determine the number of elements by dividing the `sizeof` the array by the `sizeof` an element:

```
// sizeof(ia)/sizeof(*ia) returns the number of elements in ia
int sz = sizeof(ia)/sizeof(*ia);
```

▶ 注意：sizeof 是操作符，不是函数。sizeof 表达式的值是在编译时确定的无符号整数，它只关心操作数的类型，而不会对操作数求值，sizeof 当中的表达式的副作用也不会发生。

▶ 根据本节的描述，在 C++ 语言里一个 char 就是一个 byte，而一个 byte 是 8 bits。这是符合普遍的实际情况的。

EXERCISES SECTION 5.8

Exercise 5.22: Write a program to print the size of each of the built-in types.

Exercise 5.23: Predict the output of the following program and explain your reasoning. Now run the program. Is the output what you expected? If not, figure out why.

```
int x[10];   int *p = x;
cout << sizeof(x)/sizeof(*x) << endl;
cout << sizeof(p)/sizeof(*p) << endl;
```

168

5.9 Comma Operator

A **comma expression** is a series of expressions separated by commas. The expressions are evaluated from left to right. The result of a comma expression is the value of the rightmost expression. The result is an lvalue if the rightmost operand is an lvalue. One common use for the comma operator is in a `for` loop.

```
int cnt = ivec.size();
//  add elements from size...1 to ivec
for(vector<int>::size_type ix = 0;
                  ix != ivec.size(); ++ix, --cnt)
    ivec[ix] = cnt;
```

▶ 注意：用逗号表达式来缩短代码行数通常是个坏主意。

This loop increments `ix` and decrements `cnt` in the expression in the `for` header. Both `ix` and `cnt` are changed on each trip through the loop. As long as the test of `ix` succeeds, we reset the next element to the current value of `cnt`.

EXERCISES SECTION 5.9

Exercise 5.24: The program in this section is similar to the program on page 163 that added elements to a `vector`. Both programs decremented a counter to generate the element values. In this program we used the prefix decrement and the earlier one used postfix. Explain why we used prefix in one and postfix in the other.

5.10 Evaluating Compound Expressions

An expression with two or more operators is a **compound expression**. In a compound expression, the way in which the operands are grouped to the operators may determine the result of the overall expression. If the operands group in one way, the result differs from what it would be if they grouped another way.

Precedence and associativity determine how the operands are grouped. That is, precedence and associativity determine which part of the expression is the operand for each of the operators in the expression. Programmers can override these rules by parenthesizing compound expressions to force a particular grouping.

▶ 本节内容非常重要，但重要的不是记住各种操作符的优先级与结合性，而是要明白，子表达式求值顺序（order of evaluation）是由编译器自由决定的（除了第 5.10.3 节列出的四个例外）。如果表达式有副作用，不同的求值顺序会造成不同的运行结果。

Precedence specifies how the operands are grouped. It says nothing about the order in which the operands are evaluated. In most cases, operands may be evaluated in whatever order is convenient.

5.10.1 Precedence

The value of an expression depends on how the subexpressions are grouped. For example, in the following expression, a purely left-to-right evaluation yields 20:

```
6 + 3 * 4 / 2 + 2;
```

Other imaginable results include 9, 14, and 36. In C++, the result is 14. 169

Multiplication and division have higher precedence than addition. Their operands are bound to the operator in preference to the operands to addition. Multiplication and division have the same precedence as each other. Operators also have associativity, which determines how operators at the same precedence level are grouped. The arithmetic operators are left associative, which means they group left to right. We now can see that our expression is equivalent to

```
int temp = 3 * 4;          // 12
int temp2 = temp / 2;      // 6
int temp3 = temp2 + 6;     // 12
int result = temp3 + 2;    // 14
```

Parentheses Override Precedence

We can override precedence with parentheses. Parenthesized expressions are evaluated by treating each parenthesized subexpression as a unit and otherwise applying the normal precedence rules. For example, we can use parentheses on our initial expression to force the evaluation to result in any of the four possible values:

```
// parentheses on this expression match default precedence/associativity
cout << ((6 + ((3 * 4) / 2)) + 2) << endl; // prints 14
// parentheses result in alternative groupings
cout << (6 + 3) * (4 / 2 + 2) << endl;      // prints 36
cout << ((6 + 3) * 4) / 2 + 2 << endl;      // prints 20
cout << 6 + 3 * 4 / (2 + 2) << endl;        // prints 9
```

We have already seen examples where precedence rules affect the correctness of our programs. For example, consider the expression described in the "Advice" box on page 164:

```
*iter++;
```

Precedence says that `++` has higher precedence than `*`. That means that `iter++` is grouped first. The operand of `*`, therefore, is the result of applying the increment operator to `iter`. If we wanted to increment the value that `iter` denotes, we'd have to use parentheses to force our intention:

```
(*iter)++;   // increment value to which iter refers and yield unincremented value
```

The parentheses specify that the operand of `*` is `iter`. The expression now uses `*iter` as the operand to `++`.

As another example, recall the condition in the `while` on page 161:

```
while ((i = get_value()) != 42) {
```

The parentheses around the assignment were necessary to implement the desired operation, which was to assign to `i` the value returned from `get_value` and then test that value to see whether it was 42. Had we failed to parenthesize the assignment, the effect would be to test the return value to see whether it was 42. The true or false value of that test would then be assigned to `i`, meaning that `i` would either be 1 or 0.

5.10.2 Associativity 170

Associativity specifies how to group operators at the same precedence level. We have also seen cases where associativity matters. As one example, the assignment operator is right associative. This fact allows concatenated assignments:

```
ival = jval = kval = lval        // right associative
(ival = (jval = (kval = lval))) // equivalent, parenthesized version
```

This expression first assigns `lval` to `kval`, then the result of that to `jval`, and finally the result of that to `ival`.

The arithmetic operators, on the other hand, are left associative. The expression

```
ival * jval / kval * lval          // left associative
(((ival * jval) / kval) * lval)    // equivalent, parenthesized version
```

multiplies `ival` and `jval`, then divides that result by `kval`, and finally multiplies the result of the division by `lval`.

Table 5.4 presents the full set of operators ordered by precedence. The table is organized into segments separated by double lines. Operators in each segment have the same precedence, and have higher precedence than operators in subsequent segments. For example, the prefix increment and dereference operators share the same precedence and have higher precedence than the arithmetic or relational operators. We have seen most of these operators, although a few will not be defined until later chapters.

Table 5.4: Operator Precedence

Associativity and Operator		Function	Use	See Page
L	`::`	global scope	`::`name	p. 450
L	`::`	class scope	class`::`name	p. 85
L	`::`	namespace scope	namespace`::`name	p. 78
L	`.`	member selectors	object`.`member	p. 25
L	`->`	member selectors	pointer`->`member	p. 164
L	`[]`	subscript	variable`[` expr `]`	p. 113
L	`()`	function call	name`(`expr_list`)`	p. 25
L	`()`	type construction	type`(`expr_list`)`	p. 460
R	`++`	postfix increment	lvalue`++`	p. 162
R	`--`	postfix decrement	lvalue`--`	p. 162
R	`typeid`	type ID	`typeid(`type`)`	p. 775
R	`typeid`	run-time type ID	`typeid(`expr`)`	p. 775
R	explicit cast	type conversion	*cast_name*<type>(expr)	p. 183
R	`sizeof`	size of object	`sizeof` expr	p. 167
R	`sizeof`	size of type	`sizeof(` type `)`	p. 167
R	`++`	prefix increment	`++`lvalue	p. 162
R	`--`	prefix decrement	`--`lvalue	p. 162
R	`~`	bitwise NOT	`~`expr	p. 154
R	`!`	logical NOT	`!`expr	p. 152
R	`-`	unary minus	`-`expr	p. 150
R	`+`	unary plus	`+`expr	p. 150
R	`*`	dereference	`*`expr	p. 119
R	`&`	address-of	`&`expr	p. 115
R	`()`	type conversion	(type) expr	p. 186
R	`new`	allocate object	`new` type	p. 174
R	`delete`	deallocate object	`delete` expr	p. 176
R	`delete[]`	deallocate array	`delete[]` expr	p. 137
L	`->*`	ptr to member select	ptr`->*`ptr_to_member	p. 783
L	`.*`	ptr to member select	obj`.*`ptr_to_member	p. 783
L	`*`	multiply	expr `*` expr	p. 149
L	`/`	divide	expr `/` expr	p. 149
L	`%`	modulo (remainder)	expr `%` expr	p. 149
L	`+`	add	expr `+` expr	p. 149
L	`-`	subtract	expr `-` expr	p. 149
L	`<<`	bitwise shift left	expr `<<` expr	p. 154
L	`>>`	bitwise shift right	expr `>>` expr	p. 154

171

(continued)

Associativity and Operator		Function	Use	See Page
L	<	less than	expr < expr	p. 152
L	<=	less than or equal	expr <= expr	p. 152
L	>	greater than	expr > expr	p. 152
L	>=	greater than or equal	expr >= expr	p. 152
L	==	equality	expr == expr	p. 152
L	!=	inequality	expr != expr	p. 152
L	&	bitwise AND	expr & expr	p. 154
L	^	bitwise XOR	expr ^ expr	p. 154
L	|	bitwise OR	expr | expr	p. 154
L	&&	logical AND	expr && expr	p. 152
L	||	logical OR	expr || expr	p. 152
R	?:	conditional	expr ? expr : expr	p. 165
R	=	assignment	lvalue = expr	p. 159
R	*=, /=, %=,	compound assign	lvalue += expr, etc.	p. 159
R	+=, -=,			p. 159
R	<<=, >>=,			p. 159
R	&=, |=, ^=			p. 159
R	throw	throw exception	throw expr	p. 216
L	,	comma	expr , expr	p. 168

EXERCISES SECTION 5.10.2

172

Exercise 5.25: Using Table 5.4 (p. 170), parenthesize the following expressions to indicate the order in which the operands are grouped:

```
(a)  ! ptr == ptr->next
(b)  ch = buf[ bp++ ] != '\n'
```

Exercise 5.26: The expressions in the previous exercise evaluate in an order that is likely to be surprising. Parenthesize these expressions to evaluate in an order you imagine is intended.

Exercise 5.27: The following expression fails to compile due to operator precedence. Using Table 5.4 (p. 170), explain why it fails. How would you fix it?

```
string s = "word";
// add an 's' to the end, if the word doesn't already end in 's'
string pl = s + s[s.size() - 1] == 's' ? "" : "s" ;
```

5.10.3 Order of Evaluation

In Section 5.2 (p. 152) we saw that the && and || operators specify the order in which their operands are evaluated: In both cases the right-hand operand is evaluated if and only if doing so might affect the truth value of the overall expression. Because we can rely on this property, we can write code such as

```
// iter only dereferenced if it isn't at end
while (iter != vec.end() && *iter != some_val)
```

The only other operators that guarantee the order in which operands are evaluated are the conditional (? :) and comma operators. In all other cases, the order is unspecified.

For example, in the expression

```
f1() * f2();
```

we know that both `f1` and `f2` must be called before the multiplication can be done. After all, their results are what is multiplied. However, we have no way to know whether `f1` will be called before `f2` or vice versa.

▶ 注意：括号只能改变运算的优先级（参见英文原版书第 169 页），不能改变求值的顺序。例如，f()+g()-h() 和 f()+(g()-h()) 两种写法的运算优先级不同，但求值顺序（即先调用哪个函数）有可能相同。

The order of operand evaluation often, perhaps even usually, doesn't matter. It can matter greatly, though, if the operands refer to *and change* the same objects.

The order of operand evaluation matters if one subexpression changes the value of an operand used in another subexpression:

```
// oops! language does not define order of evaluation
if (ia[index++] < ia[index])
```

173

The behavior of this expression is undefined. The problem is that the left- and right-hand operands to the < both use the variable `index`. However, the left-hand operand involves changing the value of that variable. Assuming `index` is zero, the compiler might evaluate this expression in one of the following two ways:

```
if (ia[0] < ia[0])    // execution if rhs is evaluated first
if (ia[0] < ia[1])    // execution if lhs is evaluated first
```

We can guess that the programmer intended that the left operand be evaluated, thereby incrementing `index`. If so, the comparison would be between `ia[0]` and `ia[1]`. The language, however, does not guarantee a left-to-right evaluation order. In fact, an expression like this is undefined. An implementation might evaluate the right-hand operand first, in which case `ia[0]` is compared to itself. Or the implementation might do something else entirely.

ADVICE: MANAGING COMPOUND EXPRESSIONS

Beginning C and C++ programmers often have difficulties understanding order of evaluation and the rules of precedence and associativity. Misunderstanding how expressions and operands are evaluated is a rich source of bugs. Moreover, the resulting bugs are difficult to find because reading the program does not reveal the error unless the programmer already understands the rules.

Two rules of thumb can be helpful:

1. **When in doubt, parenthesize expressions to force the grouping that the logic of your program requires.**
2. **If you change the value of an operand, don't use that operand elsewhere in the same statement. If you need to use the changed value, then break the expression up into separate statements in which the operand is changed in one statement and then used in a subsequent statement.**

An important exception to the second rule is that subexpressions that use the result of the subexpression that changes the operand are safe. For example, in `*++iter` the increment changes the value of `iter`, and the (changed) value of `iter` is then used as the operand to `*`. In this, and similar, expressions, order of evaluation of the operand isn't an issue. To evaluate the larger expression, the subexpression that changes the operand must first be evaluated. Such usage poses no problems and is quite common.

▶ 按照 C++ 标准，一个对象的值在一个表达式里只能改变一次，因此 i= i++ 是未定义行为（undefined）。此外，如果一个表达式修改了某个对象的值，那么这个对象在该表达式里只能出现一次（即上面的第 2 点建议），因为副作用发生的时机是未指明的（unspecified）。

Do not use an increment or decrement operator on the same object in more than two subexpressions of the same expression.

One safe and machine-independent way to rewrite the previous comparison of two array elements is

```
if (ia[index] < ia[index + 1]) {
    // do whatever
}
++index;
```

Now neither operand can affect the value of the other.

EXERCISES SECTION 5.10.3

174

Exercise 5.28: With the exception of the logical AND and OR, the order of evaluation of the binary operators is left undefined to permit the compiler freedom to provide an optimal implementation. The trade-off is between an efficient implementation and a potential pitfall in the use of the language by the programmer. Do you consider that an acceptable trade-off? Why or why not?

Exercise 5.29: Given that `ptr` points to a class with an `int` member named `ival`, `vec` is a `vector` holding `int`s, and that `ival`, `jval`, and `kval` are also `int`s, explain the behavior of each of these expressions. Which, if any, are likely to be incorrect? Why? How might each be corrected?

```
(a) ptr->ival != 0          (b) ival != jval < kval
(c) ptr != 0 && *ptr++      (d) ival++ && ival
(e) vec[ival++] <= vec[ival]
```

5.11 The `new` and `delete` Expressions

In Section 4.3.1 (p. 134) we saw how to use `new` and `delete` expressions to dynamically allocate and free arrays. We can also use `new` and `delete` to dynamically allocate and free single objects.

When we define a variable, we specify a type and a name. When we dynamically allocate an object, we specify a type but do not name the object. Instead, the `new` expression returns a pointer to the newly allocated object; we use that pointer to access the object:

```
int i;              // named, uninitialized int variable
int *pi = new int; // pi points to dynamically allocated,
                    // unnamed, uninitialized int
```

This `new` expression allocates one object of type `int` from the free store and returns the address of that object. We use that address to initialize the pointer `pi`.

Initializing Dynamically Allocated Objects

Dynamically allocated objects may be initialized, in much the same way as we initialize variables:

```
int i(1024);              // value of i is 1024
int *pi = new int(1024); // object to which pi points is 1024
string s(10, '9');                    // value of s is "9999999999"
string *ps = new string(10, '9');    // *ps is "9999999999"
```

We must use the direct-initialization syntax (Section 2.3.3, p. 48) to initialize dynamically allocated objects. When an initializer is present, the `new` expression allocates the required memory and initializes that memory using the given initializer(s). In the case of `pi`, the newly allocated object is initialized to 1024. The object pointed to by `ps` is initialized to a string of 10 nines.

▶ free store 是 C++ 术语，指的是借助 new/delete 动态管理的内存。在其他编程语言里，一般叫做堆（heap），例如，我们常说“在堆上分配内存 / 对象”。但要注意，这与数据结构里讲的用于实现优先队列的堆（heap）是不同的。另外，C++ 程序里也可以有堆（heap），一般指的是 malloc/free 动态管理的内存，所以严格来说 free store 和 heap 在 C++ 里的含义不同，不过在一般交流时可以统称为堆（heap），与栈（stack）相对。

▶ 这里容易犯的一个错误是把方括号 [] 写成了圆括号 ()。它们的意思完全不一样。比如原本打算分配 1024 个元素的整数数组，应写为 int*pi = new int[1024];，再使用其中一些元素，例如 pi[82] = 43; ，但是由于一时疏忽，误写为 int* pi = new int(1024);，程序编译没有问题，但运行时会出错（如果“运气好”的话会直接崩溃）。应该尽量以 std::vector 或 tr1::array 代替原生数组。

175 ## Default Initialization of Dynamically Allocated Objects

If we do not explicitly state an initializer, then a dynamically allocated object is initialized in the same way as is a variable that is defined inside a function. (Section 2.3.4, p. 50) If the object is of class type, it is initialized using the default constructor for the type; if it is of built-in type, it is uninitialized.

```
string *ps = new string;    // initialized to empty string
int *pi = new int;          // pi points to an uninitialized int
```

As usual, it is undefined to use the value associated with an uninitialized object in any way other than to assign a good value to it.

Just as we (almost) always initialize the objects we define as variables, it is (almost) always a good idea to initialize dynamically allocated objects.

We can also value-initialize (Section 3.3.1, p. 92) a dynamically allocated object:

```
string *ps = new string(); // initialized to empty string
int *pi = new int();  // pi points to an int value-initialized to 0
cls *pc = new cls();  // pc points to a value-initialized object of type cls
```

▶ 这段话是说，如果 new 的对象定义了构造函数，那么加不加后面那对圆括号 () 并不影响程序的行为；如果 new 的对象没有构造函数，或者是内置类型，那么就大不一样了。

We indicate that we want to value-initialize the newly allocated object by following the type name by a pair of empty parentheses. The empty parentheses signal that we want initialization but are not supplying a specific initial value. In the case of class types (such as `string`) that define their own constructors, requesting value-initialization is of no consequence: The object is initialized by running the default constructor whether we leave it apparently uninitialized or ask for value-initialization. In the case of built-in types or types that do not define any constructors, the difference is significant:

```
int *pi = new int;          // pi points to an uninitialized int
int *pi = new int();        // pi points to an int value-initialized to 0
```

In the first case, the `int` is uninitialized; in the second case, the `int` is initialized to zero.

The `()` syntax for value initialization must follow a type name, not a variable. As we'll see in Section 7.4 (p. 251)

```
int x(); // does not value initialize x
```

declares a function named `x` with no arguments that returns an `int`.

Memory Exhaustion

Although modern machines tend to have huge memory capacity, it is always possible that the free store will be exhausted. If the program uses all of available memory, then it is possible for a `new` expression to fail. If the `new` expression cannot acquire the requested memory, it throws an exception named `bad_alloc`. We'll look at how exceptions are thrown in Section 6.13 (p. 215).

176 ## Destroying Dynamically Allocated Objects

When our use of the object is complete, we must *explicitly* return the object's memory to the free store. We do so by applying the `delete` expression to a pointer that addresses the object we want to release.

```
delete pi;
```

frees the memory associated with the `int` object addressed by `pi`.

It is illegal to apply `delete` to a pointer that addresses memory that was not allocated by `new`.

▶ 注意：delete 的实参必须是 new 返回的指针。

The effect of deleting a pointer that addresses memory that was not allocated by new is undefined. The following are examples of safe and unsafe delete expressions:

```
int i;
int *pi = &i;
string str = "dwarves";
double *pd = new double(33);
delete str; //  error: str is not a dynamic object
delete pi;  //  error: pi refers to a local
delete pd;  //  ok
```

It is worth noting that the compiler might refuse to compile the delete of `str`. The compiler knows that `str` is not a pointer and so can detect this error at compile-time. The second error is more insidious: In general, compilers cannot tell what kind of object a pointer addresses. Most compilers will accept this code, even though it is in error.

`delete` of a Zero-Valued Pointer

It is legal to `delete` a pointer whose value is zero; doing so has no effect:

```
int *ip = 0;
delete ip;  //  ok: always ok to delete a pointer that is equal to 0
```

The language guarantees that deleting a pointer that is equal to zero is safe.

Resetting the Value of a Pointer after a `delete`

When we write

```
delete p;
```

`p` becomes undefined. Although `p` is undefined, on many machines, `p` still contains the address of the object to which it pointed. However, the memory to which `p` points was freed, so `p` is no longer valid.

After deleting a pointer, the pointer becomes what is referred to as a **dangling pointer**. A dangling pointer is one that refers to memory that once held an object but does so no longer. A dangling pointer can be the source of program errors that are difficult to detect.

Setting the pointer to 0 after the object it refers to has been deleted makes it clear that the pointer points to no object.

177

▶ 这其实用处不大，因为指针是可以拷贝的，delete p; 之后再令 p = NULL；并不能保证没有其他指针指向这块已经被释放的内存。

Dynamic Allocation and Deallocation of `const` Objects

It is legal to dynamically create `const` objects:

```
//  allocate and initialize a const object
const int *pci = new const int(1024);
```

Like any `const`, a dynamically created `const` must be initialized when it is created and once initialized cannot be changed. The value returned from this new expression is a pointer to `const int`. Like the address of any other `const` object, the return from a `new` that allocates a `const` object may only be assigned to a pointer to `const`.

A `const` dynamic object of a class type that defines a default constructor may be initialized implicitly:

```
//  allocate default initialized const empty string
const string *pcs = new const string;
```

This new expression does not explicitly initialize the object pointed to by pcs. Instead, the object to which pcs points is implicitly initialized to the empty string. Objects of built-in type or of a class type that does not provide a default constructor must be explicitly initialized.

▶ 这一段值得反复阅读，它指出了 C++ 中使用 new/delete 手动管理内存的常见错误：内存泄漏、空悬指针、双重释放。对此，现代 C++ 语言有一套完整的解决办法，即使用 STL 容器及智能指针来自动管理内存。

CAUTION: MANAGING DYNAMIC MEMORY IS ERROR-PRONE

The following three common program errors are associated with dynamic memory allocation:

1. **Failing to `delete` a pointer to dynamically allocated memory, thus preventing the memory from being returned to the free store. Failure to delete dynamically allocated memory is spoken of as a "memory leak." Testing for memory leaks is difficult because they often do not appear until the application is run for a test period long enough to actually exhaust memory.**
2. **Reading or writing to the object after it has been deleted. This error can sometimes be detected by setting the pointer to 0 after deleting the object to which the pointer had pointed.**
3. **Applying a delete expression to the same memory location twice. This error can happen when two pointers address the same dynamically allocated object. If `delete` is applied to one of the pointers, then the object's memory is returned to the free store. If we subsequently `delete` the second pointer, then the free store may be corrupted.**

These kinds of errors in manipulating dynamically allocated memory are considerably easier to make than they are to track down and fix.

178

Deleting a const Object

Although the value of a const object cannot be modified, the object itself can be destroyed. As with any other dynamic object, a const dynamic object is freed by deleting a pointer that points to it:

```
delete pci;  // ok: deletes a const object
```

Even though the operand of the delete expression is a pointer to const int, the delete expression is valid and causes the memory to which pci refers to be deallocated.

EXERCISES SECTION 5.11

Exercise 5.30: Which of the following, if any, are illegal or in error?

```
(a) vector<string> svec(10);
(b) vector<string> *pvec1 = new vector<string>(10);
(c) vector<string> **pvec2 = new vector<string>[10];
(d) vector<string> *pv1 = &svec;
(e) vector<string> *pv2 = pvec1;

(f) delete svec;
(g) delete pvec1;
(h) delete [] pvec2;
(i) delete pv1;
(j) delete pv2;
```

5.12 Type Conversions

The type of the operand(s) determine whether an expression is legal and, if the expression is legal, determines the meaning of the expression. However, in C++

some types are related to one another. When two types are related, we can use an object or value of one type where an operand of the related type is expected. Two types are related if there is a **conversion** between them.

As an example, consider

```
int ival = 0;
ival = 3.541 + 3; // typically compiles with a warning
```

which assigns 6 to `ival`.

The operands to the addition operator are values of two different types: `3.541` is a literal of type `double`, and `3` is a literal of type `int`. Rather than attempt to add values of the two different types, C++ defines a set of conversions to transform the operands to a common type before performing the arithmetic. These conversions are carried out automatically by the compiler without programmer intervention—and sometimes without programmer knowledge. For that reason, they are referred to as **implicit type conversions**.

▶ 隐式类型转换是C++的难点之一。C++从C语言继承了复杂的隐式类型转换规则，这些规则不是每一条都符合直觉。如果不熟悉这些规则，有可能造成意想不到的后果。

179

The built-in conversions among the arithmetic types are defined to preserve precision, if possible. Most often, if an expression has both integral and floating-point values, the integer is converted to floating-point. In this addition, the integer value `3` is converted to `double`. Floating-point addition is performed and the result, `6.541`, is of type `double`.

The next step is to assign that `double` value to `ival`, which is an `int`. In the case of assignment, the type of the left-hand operand dominates, because it is not possible to change the type of the object on the left-hand side. When the left- and right-hand types of an assignment differ, the right-hand side is converted to the type of the left-hand side. Here the `double` is converted to `int`. Converting a `double` to an `int` truncates the value; the decimal portion is discarded. 6.541 becomes 6, which is the value assigned to `ival`. Because the conversion of a `double` to `int` may result in a loss of precision, most compilers issue a warning. For example, the compiler we used to check the examples in this book warns us:

```
warning: assignment to 'int' from 'double'
```

To understand implicit conversions, we need to know when they occur and what conversions are possible.

5.12.1 When Implicit Type Conversions Occur

▶ 尽管我们在写代码时应该避免隐式或显式的类型转换，但是有些转换是在不经意间发生的。读者应该熟悉发生隐式类型转换的五种场景。

The compiler applies conversions for both built-in and class type objects as necessary. Implicit type conversions take place in the following situations:

- In expressions with operands of mixed types, the types are converted to a common type:

```
int ival;
double dval;
ival >= dval   // ival converted to double
```

- An expression used as a condition is converted to `bool`:

```
int ival;
if (ival)      // ival converted to bool
while (cin)    // cin converted to bool
```

  Conditions occur as the first operand of the conditional (`? :`) operator and as the operand(s) to the logical NOT (`!`), logical AND (`&&`), and logical OR (`||`) operators. Conditions also appear in the `if`, `while`, `for`, and `do while` statements. (We cover the `do while` in Chapter 6)

- An expression used to initialize or assign to a variable is converted to the type of the variable:

```
int ival = 3.14; // 3.14 converted to int
int *ip;
ip = 0;    // the int 0 converted to a null pointer of type int *
```

In addition, as we'll see in Chapter 7, implicit conversions also occur during function calls.

180

5.12.2 The Arithmetic Conversions

▶ 即使是这种最简单的整型提升也可能造成 bug。例如，在大多数实现中 char 是有符号的，它的表示范围是 –128~127，如果我们想把它转换成一个非负数，用于数组下标，那么 static_cast<unsigned int>(aChar) 并不能达到效果。对于负值，转换的结果将会是一个非常大的整数，因为 char 会被先提升为 int，再转换为 unsigned int。正确的做法是 static_cast<unsigned char>(aChar)，这样转换后的取值范围是 0~255。

The language defines a set of conversions among the built-in types. Among these, the most common are the **arithmetic conversions**, which ensure that the two operands of a binary operator, such as an arithmetic or logical operator, are converted to a common type before the operator is evaluated. That common type is also the result type of the expression.

The rules define a hierarchy of type conversions in which operands are converted to the widest type in the expression. The conversion rules are defined so as to preserve the precision of the values involved in a multitype expression. For example, if one operand is of type `long double`, then the other is converted to type `long double` regardless of what the second type is.

The simplest kinds of conversion are **integral promotions**. Each of the integral types that are smaller than `int`—`char`, `signed char`, `unsigned char`, `short`, and `unsigned short`—is promoted to `int` if all possible values of that type fit in an `int`. Otherwise, the value is promoted to `unsigned int`. When `bool` values are promoted to `int`, a `false` value promotes to zero and `true` to one.

Conversions between Signed and Unsigned Types

When an `unsigned` value is involved in an expression, the conversion rules are defined to preserve the value of the operands. Conversions involving `unsigned` operands depend on the relative sizes of the integral types on the machine. Hence, such conversions are inherently machine dependent.

▶ 建议读者写代码验证这段话的内容，比如 short sv = 30000; int iv = s*s;，看看 iv 的值是多少。

In expressions involving `shorts` and `ints`, values of type `short` are converted to `int`. Expressions involving `unsigned short` are converted to `int` if the `int` type is large enough to represent all the values of an `unsigned short`. Otherwise, both operands are converted to `unsigned int`. For example, if `shorts` are a half word and `ints` a word, then any `unsigned` value will fit inside an `int`. On such a machine, `unsigned shorts` are converted to `int`.

The same conversion happens among operands of type `long` and `unsigned int`. The `unsigned int` operand is converted to `long` if type `long` on the machine is large enough to represent all the values of the `unsigned int`. Otherwise, both operands are converted to `unsigned long`.

On a 32-bit machine, `long` and `int` are typically represented in a word. On such machines, expressions involving `unsigned ints` and `longs` are converted to `unsigned long`.

▶ 无符号整数也是隐式类型转换 bug 的常客，比方说（0U > -1）这个表达式的值竟然是 false。有的语言干脆取消无符号整数，从而避免将一个负数强行转换为正数的尴尬。

Conversions for expressions involving signed and `unsigned int` can be surprising. In these expressions the signed value is converted to `unsigned`. For example, if we compare a plain `int` and an `unsigned int`, the `int` is first converted to `unsigned`. If the `int` happens to hold a negative value, the result will be converted as described in Section 2.1.1 (p. 36), with all the attendant problems discussed there.

Understanding the Arithmetic Conversions

The best way to understand the arithmetic conversions is to study lots of examples. In most of the following examples, either the operands are converted to the largest type involved in the expression or, in the case of assignment expressions, the right-hand operand is converted to the type of the left-hand operand:

181

```
bool      flag;        char           cval;
short     sval;        unsigned short usval;
int       ival;        unsigned int   uival;
long      lval;        unsigned long  ulval;
float     fval;        double         dval;
3.14159L + 'a'; // promote 'a' to int, then convert to long double
```

```
dval + ival;     // ival converted to double
dval + fval;     // fval converted to double
ival = dval;     // dval converted (by truncation) to int
flag = dval;     // if dval is 0, then flag is false, otherwise true
cval + fval;     // cval promoted to int, that int converted to float
sval + cval;     // sval and cval promoted to int
cval + lval;     // cval converted to long
ival + ulval;    // ival converted to unsigned long
usval + ival;    // promotion depends on size of unsigned short and int
uival + lval;    // conversion depends on size of unsigned int and long
```

In the first addition, the character constant lowercase `'a'` has type `char`, which as we know from Section 2.1.1 (p. 34) is a numeric value. The numeric value that `'a'` represents depends on the machine's character set. On our ASCII machine, `'a'` represents the number 97. When we add `'a'` to a `long double`, the `char` value is promoted to `int` and then that `int` value is converted to a `long double`. That converted value is added to the `long double` literal. The other interesting cases are the last two expressions involving `unsigned` values.

5.12.3 Other Implicit Conversions

Pointer Conversions

In most cases when we use an array, the array is automatically converted to a pointer to the first element:

```
int ia[10];      // array of 10 ints
int* ip = ia;    // convert ia to pointer to first element
```

The exceptions when an array is not converted to a pointer are: as the operand of the address-of (`&`) operator or of `sizeof`, or when using the array to initialize a reference to the array. We'll see how to define a reference (or pointer) to an array in Section 7.2.4 (p. 240).

There are two other pointer conversions: A pointer to any data type can be converted to a `void*`, and a constant integral value of 0 can be converted to any pointer type.

▶ 因此 char* p = false; 是合法的语句，而 char* p = true; 是非法的，因为 false 提升为 int 之后的值是 0。

182

Conversions to `bool`

Arithmetic and pointer values can be converted to `bool`. If the pointer or arithmetic value is zero, then the `bool` is `false`; any other value converts to `true`:

```
if (cp) /* ... */      // true if cp is not zero
while (*cp) /* ... */ // dereference cp and convert resulting char to bool
```

Here, the `if` converts any nonzero value of `cp` to `true`. The `while` dereferences `cp`, which yields a `char`. The null character has value zero and converts to `false`. All other `char` values convert to `true`.

Arithmetic Type and `bool` Conversions

Arithmetic objects can be converted to `bool` and `bool` objects can be converted to `int`. When an arithmetic type is converted to `bool`, zero converts as `false` and any other value converts as `true`. When a `bool` is converted to an arithmetic type, `true` becomes one and `false` becomes zero:

```
bool b = true;
int ival = b;   // ival == 1
double pi = 3.14;
bool b2 = pi;   // b2 is true
pi = false;     // pi == 0
```

Conversions and Enumeration Types

Objects of an enumeration type (Section 2.7, p. 62) or an enumerator can be automatically converted to an integral type. As a result, they can be used where an integral value is required—for example, in an arithmetic expression:

```
// point2d is 2, point2w is 3, point3d is 3, point3w is 4
enum Points { point2d = 2, point2w,
              point3d = 3, point3w };
const size_t array_size = 1024;
// ok: point2w promoted to int
int chunk_size = array_size * point2w;
int array_3d = array_size * point3d;
```

The type to which an `enum` object or enumerator is promoted is machine-defined and depends on the value of the largest enumerator. Regardless of that value, an `enum` or enumerator is always promoted at least to `int`. If the largest enumerator does not fit in an `int`, then the promotion is to the smallest type larger than `int` (`unsigned int`, `long` or `unsigned long`) that can hold the enumerator value.

Conversion to `const`

A nonconst object can be converted to a `const` object, which happens when we use a nonconst object to initialize a reference to `const` object. We can also convert the address of a nonconst object (or convert a nonconst pointer) to a pointer to the related `const` type:

183

```
int i;
const int ci = 0;
const int &j = i;   // ok: convert non-const to reference to const int
const int *p = &ci; // ok: convert address of non-const to address of a const
```

Conversions Defined by the Library Types

Class types can define conversions that the compiler will apply automatically. Of the library types we've used so far, there is one important conversion that we have used. When we read from an `istream` as a condition

```
string s;
while (cin >> s)
```

we are implicitly using a conversion defined by the IO library. In a condition such as this one, the expression `cin >> s` is evaluated, meaning `cin` is read. Whether the read succeeds or fails, the result of the expression is `cin`.

The condition in the `while` expects a value of type `bool`, but it is given a value of type `istream`. That `istream` value is converted to `bool`. The effect of converting an `istream` to `bool` is to test the state of the stream. If the last attempt to read from `cin` succeeded, then the state of the stream will cause the conversion to `bool` to be `true`—the `while` test will succeed. If the last attempt failed—say because we hit end-of-file—then the conversion to `bool` will yield `false` and the `while` condition will fail.

▶ 标准库 std:istream 重载了 14.9.2 节介绍的自定义转换操作符（细节：其转换的类型不是 bool，而是 void*，为的是避免误用，因为直接提供 istream 到 bool 的转换会让 cin << 3 编译通过）。让一个 class 类型的对象能隐式转换为 bool 类型，可以用 safe bool idiom。库的作者有必要了解这一技巧，应用程序的作者大可不必在意。

EXERCISES SECTION 5.12.3

Exercise 5.31: Given the variable definitions on page 180, explain what conversions take place when evaluating the following expressions:

```
(a) if (fval)
(b) dval = fval + ival;
(c) dval + ival + cval;
```

Remember that you may need to consider associativity of the operators in order to determine the answer in the case of expressions involving more than one operator.

5.12.4 Explicit Conversions

▶ 显式转换，又叫强制类型转换，有的书直接称之为“强制”（cast）。

An explicit conversion is spoken of as a **cast** and is supported by the following set of named cast operators: **`static_cast`**, **`dynamic_cast`**, **`const_cast`**, and **`reinterpret_cast`**.

Although necessary at times, casts are inherently dangerous constructs.

5.12.5 When Casts Might Be Useful

184

One reason to perform an explicit cast is to override the usual standard conversions. The following compound assignment

```
double dval;
int ival;
ival *= dval;   // ival = ival * dval
```

converts `ival` to `double` in order to multiply it by `dval`. That double result is then truncated to `int` in order to assign it to `ival`. We can eliminate the unnecessary conversion of `ival` to `double` by explicitly casting `dval` to `int`:

```
ival *= static_cast<int>(dval);  // converts dval to int
```

▶ 注意：这实际上改变了程序的行为。若初值 ival == 2 且 dval == 2.75，第一种写法的结果是 ival == 5，而第二种写法先将 dval 转换为整数 2，则 ival == 4。

Another reason for an explicit cast is to select a specific conversion when more than one conversion is possible. We will look at this case more closely in Chapter 14.

5.12.6 Named Casts

The general form for the named cast notation is the following:

```
cast-name<type>(expression);
```

`cast-name` may be one of `static_cast`, `const_cast`, `dynamic_cast`, or `reinterpret_cast`. *type* is the target type of the conversion, and *expression* is the value to be cast. The type of cast determines the specific kind of conversion that is performed on the *expression*.

▶ 如果程序里确实需要显式转换，那么应该用这四个具名操作符。

`dynamic_cast`

A `dynamic_cast` supports the run-time identification of objects addressed either by a pointer or reference. We cover `dynamic_cast` in Section 18.2 (p. 772).

`const_cast`

A `const cast`, as its name implies, casts away the `const`ness of its expression.

For example, we might have a function named `string_copy` that we are certain reads, but does not write, its single parameter of type `char*`. If we have access to the code, the best alternative would be to correct it to take a `const char*`. If that is not possible, we could call `string_copy` on a `const` value using a `const_cast`:

```
const  char *pc_str;
char *pc = string_copy(const_cast<char*>(pc_str));
```

Only a `const_cast` can be used to cast away `const`ness. Using any of the other three forms of cast in this case would result in a compile-time error. Similarly, it is a compile-time error to use the `const_cast` notation to perform any type conversion other than adding or removing `const`.

185

static_cast

Any type conversion that the compiler performs implicitly can be explicitly requested by using a `static_cast`:

```
double d = 97.0;
// cast specified to indicate that the conversion is intentional
char  ch = static_cast<char>(d);
```

Such casts are useful when assigning a larger arithmetic type to a smaller type. The cast informs both the reader of the program and the compiler that we are aware of and are not concerned about the potential loss of precision. Compilers often generate a warning for assignments of a larger arithmetic type to a smaller type. When we provide the explicit cast, the warning message is turned off.

A `static_cast` is also useful to perform a conversion that the compiler will not generate automatically. For example, we can use a `static_cast` to retrieve a pointer value that was stored in a `void*` pointer (Section 4.2.2, p. 119):

```
void* p = &d;    // ok: address of any data object can be stored in a void*
// ok: converts void* back to the original pointer type
double *dp = static_cast<double*>(p);
```

When we store a pointer in a `void*` and then use a `static_cast` to cast the pointer back to its original type, we are guaranteed that the pointer value is preserved. That is, the result of the cast will be equal to the original address value.

reinterpret_cast

A `reinterpret_cast` generally performs a low-level reinterpretation of the bit pattern of its operands.

▶ 注意：reinterpret_cast 是最“野蛮”的强制类型转换，一般应尽量避免使用。它在 Sockets 网络编程中有一个常见的用途：将 struct sockaddr_in* 强制转换为 struct sockaddr*。当然，这个转换可以分为两步来做，先将 struct sockaddr_in* 隐式转换到 void*，再 static_cast 到 struct sockaddr*。

A `reinterpret_cast` is inherently machine-dependent. Safely using `reinterpret_cast` requires completely understanding the types involved as well as the details of how the compiler implements the cast.

As an example, in the following cast

```
int *ip;
char *pc = reinterpret_cast<char*>(ip);
```

the programmer must never forget that the actual object addressed by `pc` is an `int`, not a character array. Any use of `pc` that assumes it's an ordinary character pointer is likely to fail *at run time* in interesting ways. For example, using it to initialize a `string` object such as

```
string str(pc);
```

is likely to result in bizarre run-time behavior.

The use of `pc` to initialize `str` is a good example of why explicit casts are dangerous. The problem is that types are changed, yet there are no warnings or errors from the compiler. When we initialized `pc` with the address of an `int`, there is no error or warning from the compiler because we explicitly said the conversion was

okay. Any subsequent use of pc will assume that the value it holds is a char*. The compiler has no way of knowing that it actually holds a pointer to an int. Thus, the initialization of str with pc is absolutely correct—albeit in this case meaningless or worse! Tracking down the cause of this sort of problem can prove extremely difficult, especially if the cast of ip to pc occurs in a file separate from the one in which pc is used to initialize a string.

186

ADVICE: AVOID CASTS

By using a cast, the programmer turns off or dampens normal type-checking (Section 2.3, p. 44). We strongly recommend that programmers avoid casts and believe that most well-formed C++ programs can be written without relying on casts.

This advice is particularly important regarding use of reinterpret_casts. Such casts are always hazardous. Similarly, use of const_cast almost always indicates a design flaw. Properly designed systems should not need to cast away const. The other casts, static_cast and dynamic_cast, have their uses but should be needed infrequently. Every time you write a cast, you should think hard about whether you can achieve the same result in a different way. If the cast is unavoidable, errors can be mitigated by limiting the scope in which the cast value is used and by documenting all assumptions about the types involved.

5.12.7 Old-Style Casts

Prior to the introduction of named cast operators, an explicit cast was performed by enclosing a type in parentheses:

```
char *pc = (char*) ip;
```

The effect of this cast is the same as using the reinterpret_cast notation. However, the visibility of this cast is considerably less, making it even more difficult to track down the rogue cast.

Standard C++ introduced the named cast operators to make casts more visible and to give the programmer a more finely tuned tool to use when casts are necessary. For example, nonpointer static_casts and const_casts tend to be safer than reinterpret_casts. As a result, the programmer (as well as readers and tools operating on the program) can clearly identify the potential risk level of each explicit cast in code.

Although the old-style cast notation is supported by Standard C++, we recommend it be used only when writing code to be compiled either under the C language or pre-Standard C++.

The old-style cast notation takes one of the following two forms:

```
type (expr); // Function-style cast notation

(type) expr; // C-language-style cast notation
```

Depending on the types involved, an old-style cast has the same behavior as a const_cast, a static_cast, or a reinterpret_cast. When used where a static_cast or a const_cast would be legal, an old-style cast does the same conversion as the respective named cast. If neither is legal, then an old-style cast performs a reinterpret_cast. For example, we might rewrite the casts from the previous section less clearly using old-style notation:

```
int ival; double dval;
ival += int (dval);  // static_cast: converts double to int
const char* pc_str;
string_copy((char*)pc_str);  // const_cast: casts away const
int *ip;
char *pc = (char*)ip;  // reinterpret_cast: treats int* as char*
```

187

▶ 应该在新写的C++代码中杜绝旧式强制类型转换，因为用文本查找工具在代码中无法完全找出这种转换。某些编译器可以对旧式转换给出警告，例如g++的-Wold-style-cast选项。

▶ 旧式转型有两种写法，第一种看起来像函数调用，但“函数名”是个类型名；第二种是从C语言里继承的写法。第一种写法其实是对象构造的语法，Type(val)构造一个Type类型的匿名对象，以val为其直接初始化（direct-initialization）的实参。C++支持第一种写法，是为了在语法上兼容内置类型与class类型，这在编写模板代码的时候有用。

The old-style cast notation remains supported for backward compatibility with programs written under pre-Standard C++ and to maintain compatibility with the C language.

EXERCISES SECTION 5.12.7

Exercise 5.32: Given the following set of definitions,

```
char cval;     int ival;     unsigned int ui;
float fval;                  double dval;
```

identify the implicit type conversions, if any, taking place:

```
(a) cval = 'a' + 3;       (b) fval = ui - ival * 1.0;
(c) dval = ui * fval;     (d) cval = ival + fval + dval;
```

Exercise 5.33: Given the following set of definitions,

```
int ival;                              double dval;
const string *ps;       char *pc;      void *pv;
```

rewrite each of the following using a named cast notation:

```
(a) pv = (void*)ps;    (b) ival = int(*pc);
(c) pv = &dval;        (d) pc = (char*) pv;
```

CHAPTER 7

FUNCTIONS

CONTENTS

This chapter describes how to define and declare ***functions***. We'll cover how arguments are passed to and values are returned from a function. We'll then look at three special kinds of functions: `inline` functions, class member functions, and overloaded functions. The chapter closes with a more advanced topic: function pointers.

▶ 函数是程序组织的基本单元。从本章开始，C++的一些特性开始体现出来。本章用了约 20 页的篇幅来讲解函数调用时的实参传递（俗称参数传递、传参）和 return 语句。这部分内容非常重要，特别是当参数类型或函数返回类型是 class 类型时，C++ 的行为非常独特，值得认真学习。

A function can be thought of as a programmer-defined operation. Like the built-in operators, each function performs some computation and (usually) yields a result. Unlike the operators, functions have names and may take an unlimited number of operands. Like operators, functions can be overloaded, meaning that the same name may refer to multiple different functions.

7.1 Defining a Function

A function is uniquely represented by a name and a set of operand types. Its operands, referred to as ***parameters***, are specified in a comma-separated list enclosed in parentheses. The actions that the function performs are specified in a block, referred to as the ***function body***. Every function has an associated ***return type***.

As an example, we could write the following function to find the greatest common divisor of two `int`s:

```
// return the greatest common divisor
int gcd(int v1, int v2)
{
    while (v2) {
        int temp = v2;
        v2 = v1 % v2;
        v1 = temp;
    }
    return v1;
}
```

Here we define a function named `gcd` that returns an `int` and has two `int` parameters. To call `gcd`, we must supply two `int` values and we get an `int` in return.

Calling a Function

To invoke a function we use the **call operator**, which is a pair of parentheses. As with any operator, the call operator takes operands and yields a result. The operands to the call operator are the name of the function and a (possibly empty) comma-separated list of ***arguments***. The result type of a call is the return type of the called function, and the result itself is the value returned by the function:

```
// get values from standard input
cout << "Enter two values: \n";
int i, j;
cin >> i >> j;
// call gcd on arguments i and j
// and print their greatest common divisor
cout << "gcd: " << gcd(i, j) << endl;
```

If we gave this program 15 and 123 as input, the output would be 3.

Calling a function does two things: It initializes the function parameters from the corresponding arguments and transfers control to the function being invoked. Execution of the *calling* function is suspended and execution of the *called* function begins. Execution of a function begins with the (implicit) definition and initialization of its parameters. That is, when we invoke `gcd`, the first thing that happens is that variables of type `int` named `v1` and `v2` are created. These variables are initialized with the values passed in the call to `gcd`. In this case, `v1` is initialized by the value of `i` and `v2` by the value of `j`.

Function Body Is a Scope

The body of a function is a statement block, which defines the function's operation. As usual, the block is enclosed by a pair of curly braces and hence forms a new scope. As with any block, the body of a function can define variables. Names defined inside a function body are accessible only within the function itself. Such

226

▶ 本书严格区分形参（parameter）和实参（argument）这两个术语，前者就像函数的局部变量，后者是调用函数时传入的数据；或者说，形参是变量声明，而实参是表达式。

▶ 调用与被调用的函数也称为 caller 和 callee。

▶ 这段话准确地表述了函数调用的过程：先初始化函数参数，再进入被调用的函数体执行其中的语句。“初始化参数”这一步所涉及的副作用会在开始执行被调用函数之前完成。例如，在 foo(bar()) 这个语句中，会先执行 bar()，用它的返回值来初始化 foo() 的参数，再执行 foo()。如果函数参数不止一个，那么参数的初始化顺序是未指明的（unspecified）。一般应该避免写出 foo(g1(), g2()) 这种调用，因为不能确定先调用 g1 () 还是 g2()。同理，应该避免函数实参的表达式具有副作用，例如 foo(i++, i+1) 中的两个表达式的求值顺序是未指明的。当然，这里有一个例外，如果 g1() 和 g2() 都是不改变程序状态的纯函数，那么程序还是安全的。

227

variables are referred to as **local variables**. They are "local" to that function; their names are visible only in the scope of the function. They exist only while the function is executing. Section 7.5 (p. 254) covers local variables in more detail.

Execution completes when a `return` statement is encountered. When the called function finishes, it yields as its result the value specified in the `return` statement. After the return is executed, the suspended, calling function resumes execution at the point of the call. It uses the return value as the result of evaluating the call operator and continues processing whatever remains of the statement in which the call was performed.

▶call operator 就是函数调用操作符，一对小括号。C++ 中的函数调用是个表达式，return 的东西就是这个表达式的值。

Parameters and Arguments

Like local variables, the parameters of a function provide named, local storage for use by the function. The difference is that parameters are defined inside the function's parameter list and are initialized by arguments passed to the function when the function is called.

An argument is an expression. It might be a variable, a literal constant or an expression involving one or more operators. We must pass exactly the same number of arguments as the function has parameters. The type of each argument must match the corresponding parameter in the same way that the type of an initializer must match the type of the object it initializes: The argument must have the same type or have a type that can be implicitly converted (Section 5.12, p. 178) to the parameter type. We'll cover how arguments match a parameter in detail in Section 7.8.2 (p. 269).

7.1.1 Function Return Type

The return type of a function can be a built-in type, such as `int` or `double`, a class type, or a compound type, such as `int&` or `string*`. A return type also can be `void`, which means that the function does not return a value. The following are example definitions of possible function return types:

```
bool is_present(int *, int);      // returns bool
int count(const string &, char); // returns int
Date &calendar(const char*);      // returns reference to Date
void process();                   // process does not return a value
```

A function may not return another function or a built-in array type. Instead, the function may return a pointer to the function or to a pointer to an element in the array:

228

▶C++ 中的函数和数组不是一等公民类型。

```
// ok: pointer to first element of the array
int *foo_bar() { /* ... */ }
```

This function returns a pointer to `int` and that pointer could point to an element in an array.

We'll learn about function pointers in Section 7.9 (p. 276).

Functions Must Specify a Return Type

It is illegal to define or declare a function without an explicit return type:

```
// error: missing return type
test(double v1, double v2) { /* ... */ }
```

Eariler versions of C++ would accept this program and implicitly define the return type of `test` as an `int`. Under Standard C++, this program is an error.

In pre-Standard C++, a function without an explicit return type was assumed to return an `int`. C++ programs compiled under earlier, non-standard compilers may still contain functions that implicitly return `int`.

7.1.2 Function Parameter List

The parameter list of a function can be empty but cannot be omitted. A function with no parameters can be written either with an empty parameter list or a parameter list containing the single keyword `void`. For example, the following declarations of `process` are equivalent:

```
void process() { /* ... */ }      // implicit void parameter list

void process(void){ /* ... */ } // equivalent declaration
```

A parameter list consists of a comma-separated list of parameter types and (optional) parameter names. Even when the types of two parameters are the same, the type must be repeated:

```
int manip(int v1, v2) { /* ... */ }      // error
int manip(int v1, int v2) { /* ... */ } // ok
```

No two parameters can have the same name. Similarly, a variable local to a function may not use the same name as the name of any of the function's parameters.

Names are optional, but in a function definition, normally all parameters are named. A parameter must be named to be used.

229

Parameter Type-Checking

C++ is a statically typed language (Section 2.3, p. 44). The arguments of every call are checked during compilation.

►C++ 程序员的一项基本功是检查函数参数类型，并且在出现编译错误的时候修正形参或实参，使程序正常编译并运行。

When we call a function, the type of each argument must be either the same type as the corresponding parameter or a type that can be converted (Section 5.12, p. 178) to that type. The function's parameter list provides the compiler with the type information needed to check the arguments. For example, the function `gcd`, which we defined on page 226, takes two parameters of type `int`:

```
gcd("hello", "world");   // error: wrong argument types
gcd(24312);              // error: too few arguments
gcd(42, 10, 0);          // error: too many arguments
```

Each of these calls is a compile-time error. In the first call, the arguments are of type `const char*`. There is no conversion from `const char*` to `int`, so the call is illegal. In the second and third calls, `gcd` is passed the wrong number of arguments. The function must be called with two arguments; it is an error to call it with any other number.

But what happens if the call supplies two arguments of type `double`? Is this call legal?

```
gcd(3.14, 6.29);         // ok: arguments are converted to int
```

In C++, the answer is yes; the call is legal. In Section 5.12.1 (p. 179) we saw that a value of type `double` can be converted to a value of type `int`. This call involves such a conversion—we want to use `double` values to initialize `int` objects. Therefore, flagging the call as an error would be too severe. Rather, the arguments are implicitly converted to `int` (through truncation). Because this conversion might lose precision, most compilers will issue a warning. In this case, the call becomes

```
gcd(3, 6);
```

and returns a value of 3.

A call that passes too many arguments, omits an argument, or passes an argument of the wrong type almost certainly would result in serious run-time errors. Catching these so-called interface errors at compile time greatly reduces the compile-debug-test cycle for large programs.

Broadview 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 (Fourth Edition)

“本书在全世界C++教学领域的突出和重要，已经无须我再赘言。”

——侯捷，计算机技术书籍作家、译者、书评人

“《C++ Primer》是公认的学习C++的绝佳书籍之一，对各种层次的C++程序员都能带来帮助。第4版巩固并增强了这种地位。”

——Steve Vinoski，IONA Technologies首席工程师

“本书真的能让你入手学习C++这门庞大而复杂的语言。”

——Justin Shaw，Aerospace集团，编程技术指导委员会资深委员

“本书不仅让初学者尽早上手，还传授良好的编程实践。”

——Nevin":-)" Liber，总工程师（1988年起其就是C++开发者）

这本流行的深入讲解标准C++语言的教材如今有了全面的更新。全书内容重新组织并重新写过，帮助程序员更快地学习这门语言，并以现代的、高效的方式来运用这门语言。

这些年来C++有所进化，本书作者的教学方法也随之改变。新版一开始就介绍C++标准库，让读者立刻就能编写有用的程序，而不需要事先掌握每一个语言细节。新版强调今日之最佳编程实践，展示如何编写安全的、能快速构建的、而且性能卓越的程序。书中范例充分利用了标准库，阐释C++语言特性，并示范如何用好这门语言。与前面三版一样，本书是对C++基础概念和技术的精准论述，对于有经验的程序员也是一份宝贵的参考资料。

这本重生的经典书籍教你快速且高效地编写现代C++程序。

- 重新组织结构，适合快速学习，并从一开始就引入C++标准库
- 更新内容，传授现代的编程风格与程序设计技术
- 新的学习辅导，强调重要知识点、提醒常见易犯的错误、推荐良好的编程实践，并提供实用技巧
- 练习题巩固所学知识

本书示例代码可从 http://www.informit.com/store/product.aspx?isbn=0201721481 下载。

作者简介：

本书的三位作者都是享誉世界的C++专家。Stanley B. Lippman曾任微软Visual C++团队架构师，他从1984年起在贝尔实验室与C++之父Bjarne Stroustrup一同工作，开发C++编译器。他曾在迪斯尼和梦工厂从事电影动画制作，曾担任美国喷气推进实验室的顾问。他还撰写了《深度探索C++对象模型》等书。Josée Lajoie，曾就职于IBM加拿大C/C++编译器开发组，为ISO C++标准委员会服务了7年，担任核心语言工作组主席，并是《C++ Report》的专栏作者。Barbara E. Moo现在是一名独立顾问，具有25年软件研发经验。她在AT&T与Stroustrup和Lippman紧密合作，管理复杂的C++开发项目。Moo和Andrew Koenig一同撰写了《Accelerated C++》和《Ruminations on C++》。

评注者简介：

陈硕，北京师范大学硕士，擅长C++多线程网络编程和实时分布式系统架构。现任职于香港某跨国金融公司IT部门，从事实时外汇交易系统开发。编写了开源C++网络库Muduo；参与翻译了《代码大全（第2版）》（电子工业出版社出版）和《C++编程规范（繁体版）》（台湾碁峰出版）；2009年在上海C++技术大会做技术演讲《当析构函数遇到多线程》，同时担任Stanley B. Lippman先生的口译员；2010年在珠三角技术沙龙做技术演讲《分布式系统的工程化开发方法》；2012年在“我们的开源项目”深圳沽做《Muduo网络库：现代非阻塞C++网络编程》演讲。

PEARSON
www.pearson.com

策划编辑：张春雨
责任编辑：李云静
装帧设计：李 玲

上架建议：程序设计

ISBN 978-7-121-17441-4

9 787121 174414 >

定价：108.00元