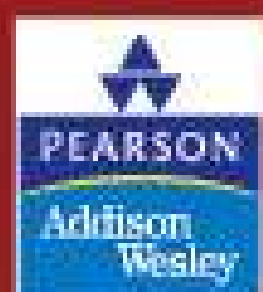




图灵程序设计丛书



第4版

C++ Primer 中文版

Stanley B. Lippman

Josée Lajoie

Barbara E. Moo

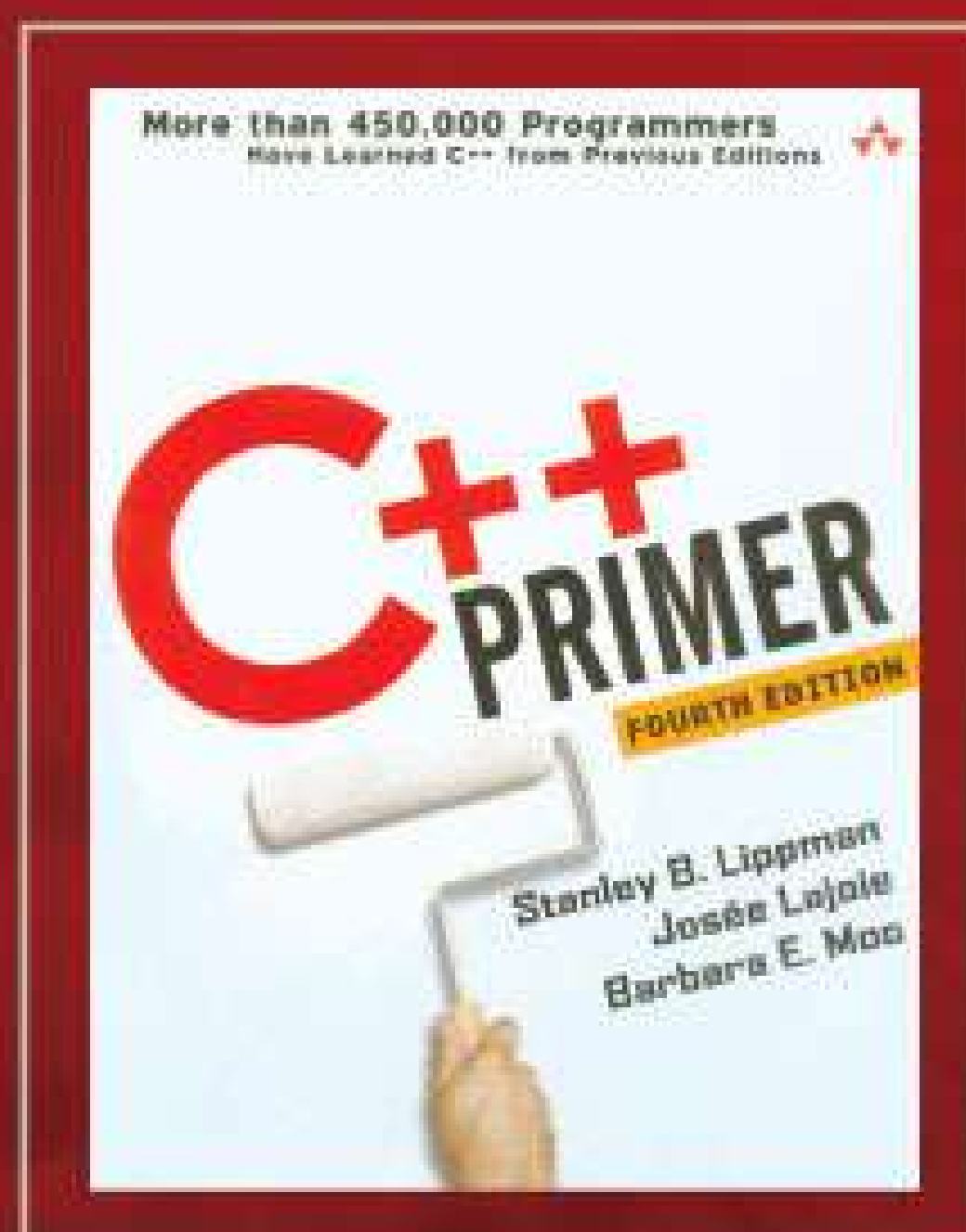
李师贤 蒋爱军

梅晓勇 林 瑛

著

译

- 久负盛名、无可替代的 C++ 经典著作
- 原版销量超过 450 000 册
- 教授最新的编程理念与实践
- 所有 C++ 程序员不可或缺的参考书



人民邮电出版社
POSTS & TELECOM PRESS

TURING 图灵程序设计丛书

C++ Primer 中文版

(第4版)

Stanley B. Lippman
Josée Lajoie 著
Barbara E. Moo

李师贤 蒋爱军 译
梅晓勇 林 瑛

 **人民邮电出版社**
POSTS & TELECOM PRESS

图书在版编目 (CIP) 数据

C++ Primer 中文版: 第4版 / (美) 李普曼, (美) 拉茹瓦, (美) 穆著; 李师贤等译.

—北京: 人民邮电出版社, 2006.3

(图灵程序设计丛书)

ISBN 7-115-14554-7

I. C... II. ①李...②拉...③穆...④李... III. C 语言—程序设计 IV. TP312

中国版本图书馆 CIP 数据核字 (2006) 第 014313 号

内 容 提 要

本书是久负盛名的 C++ 经典教程, 完美结合了 C++ 大师 Stanley B. Lippman 丰富的实践经验和 C++ 标准委员会原负责人 Josée Lajoie 对 C++ 标准的深入理解, 已经帮助全球无数程序员学会了 C++。本版对前一版进行了彻底的修订, 内容经过了重新组织, 更加入了 C++ 先驱 Barbara E. Moo 在 C++ 教学方面的真知灼见, 既显著改善了可读性, 又充分体现了 C++ 语言的最新进展和当前的业界最佳实践。书中不但新增大量教学辅助内容, 用于强调重要的知识点, 提醒常见的错误, 推荐优秀的编程实践, 给出使用提示, 还包含大量来自实战的示例和习题。

对 C++ 基本概念和技术全面而且权威的阐述, 对现代 C++ 编程风格的强调, 使本书成为 C++ 初学者的最佳指南; 对于中高级程序员, 本书也是不可或缺的参考书。

图灵程序设计丛书

C++ Primer 中文版 (第4版)

-
- ◆ 著 Stanley B. Lippman Josée Lajoie Barbara E. Moo
 - 译 李师贤 蒋爱军 梅晓勇 林 瑛
 - 责任编辑 杨海玲
 - ◆ 人民邮电出版社出版发行 北京市崇文区夕照寺街 14 号
 - 邮编 100061 电子函件 315@ptpress.com.cn
 - 网址 <http://www.ptpress.com.cn>
 - 北京顺义振华印刷厂印刷
 - 新华书店总店北京发行所经销
 - ◆ 开本: 800×1000 1/16
 - 印张: 47.75
 - 字数: 1 149 千字 2006 年 3 月第 1 版
 - 印数: 8 001 - 11 000 册 2006 年 6 月北京第 2 次印刷

著作权合同登记号 图字: 01-2005-3578 号

ISBN 7-115-14554-7/TP · 5273

定价: 99.00 元

读者服务热线: (010)88593802 印装质量热线: (010)67129223

译者序

作为目前业界广泛使用的编程语言，C++可谓包罗万象、博大精深。20 年来，讲述 C++的图书早已经汗牛充栋、层出不穷，但其中业界公认的完整涵盖 C++标准的权威著作只有两部，曾经有评论将之喻为“倚天屠龙”。其中一部当然是 C++之父 Bjarne Stroustrup 所著的《C++程序设计语言》，内容精辟深刻，但是要求较高，只适合有一定经验的程序员提升功力之用。而另一部就是本书，自 1989 年初版以来，历经多次修订，始终保持了内容全面准确、循序渐进、明快易读的特色，早已奠定了无可替代的经典地位。原版到第 3 版就累积销售了 45 万册以上，第 3 版的中文版引入国内时，也产生了极大影响，甚至曾经出现过洛阳纸贵的局面。

本书的成功当然离不开强大的作者阵容。本书第一作者 Stanley Lippman 早在 C++还处于萌芽时期就是 Stroustrup 所在的 C++编译器项目团队的成员，目前又在微软领导 Visual C++和 CLI 的开发，对 C++可以说是了如指掌，实践经验极为丰富，加之多年来著书撰文不辍，在开发社区深孚众望，已成为公认的大师级人物。第二作者 Josée Lajoie 从第 3 版开始加入，她曾经在 IBM 从事 C++编译器开发，并担任过 ISO C++标准委员会核心语言工作组主席多年，她的加盟，充分保证了本书与 C++标准的兼容。

应该说，在很大程度上本书的第 3 版已臻完美。但是拿到第 4 版样书之后，我们发现新版完全不是对前版的简单扩充，不仅在布局结构上进行了彻底更新和重新规划，对具体文字和实例也进行了大幅改动，两个版本甚至很难找到相同的段落。在并无新版本的 C++标准定案发布的情况下，作者撰写新版而且做出这么大修订的原因何在？而新版又有何重要改进呢？

众所周知，C++从 C 语言继承而来的历史包袱，C++对多种编程风格的支持，以及各种误解和不良习惯，都大大增加了 C++教学和使用的复杂性，而传统教材和教学方法的各种弊端更加剧了这一情况，使 C++成为不少人望而生畏的难学难用的“专家语言”。

阅读本书后，我们不得不承认，几位大师级作者们很好地回应了上述挑战。这里我们应该特别提到本书新版增加的第三作者 Barbara Moo。她作为项目经理，曾经领导了包括 Stroustrup 和 Lippman 在内的贝尔实验室 C++编译器团队。她在斯坦福大学教授 C++课程的丰富经验和教学改革成果，对本书新版产生了重大影响。

相比之下，本书第 4 版主要有如下特点：

- 反映了现代理念。新版紧扣 C++语言当前的应用趋势——更加关注程序员的开发效率而不是系统的运行效率，摒弃了传统的阐述方式，不再注重低层编程技术，而是从一开始就强调标准库的使用，让人耳目一新。
- 突出了实践性。新版在继承了本书原有特色——全面、详细、准确地介绍 C++知识点的基础上，特别注重介绍那些实际开发中通用的、行之有效的编程技术。在特定场合，C++提供的丰富“武器库”中应该选择哪些设施，应该注意哪些问题，业界已经总结了哪些优秀的编程实践和易犯的错误等，而这些正是目前其他 C++图书所缺乏的。

- 增加了教学环节，改善了可读性。新版版式设计非常适合阅读，而且每一章都精心组织了重要术语、小结、大量示例和习题，文中另有丰富的额外提示和交叉引用，便于读者查找复习，消化核心概念，巩固所学知识。

我们在翻译过程中深深感到，本书新版在经典前版的基础上又有了质的飞跃，体现了世界 C++ 教学方面的最新进展和最高水平。我们衷心希望本书中文版的出版，能够推动国内 C++ 教学和使用的发展。

参加本书翻译工作的有李师贤、蒋爱军、梅晓勇、林瑛，全书由李师贤审校。参与部分校对或录入工作的还有古思山、黎永基、陈晓君、刘海伟等，在此对他们的工作表示衷心的感谢！

感谢人民邮电出版社图灵公司的编辑们，他们为保证本书的质量做了大量的工作。

由于书中概念和术语数目繁多，且有许多概念和术语目前尚无公认的中文译法，加之译者水平所限，译文中不当之处，恳请读者批评指正。

前言

本书全面介绍了 C++ 语言。作为一本入门书 (Primer)，它以教程的形式对 C++ 语言进行清晰的讲解，并辅以丰富的示例和各种学习辅助手段。与大多数入门教程不同，本书对 C++ 语言本身进行了详尽的描述，并特别着重介绍了目前通行的、行之有效的程序设计技巧。

无数程序员曾使用本书的前几个版本学习 C++，在此期间 C++ 也逐渐发展成熟。这些年来，C++ 语言的发展方向以及 C++ 程序员的关注点，已经从以往注重运行时的效率，转到千方百计地提高程序员的编程效率上。随着标准库的广泛可用，我们现在能够比以往任何时候更高效地学习和使用 C++。本书这一版本充分体现了这一点。

第 4 版的改动

为了体现现代 C++ 编程风格，我们重新组织并重写了本书。书中不再强调低层编程技术，而把中心转向标准库的使用。书中很早就开始介绍标准库，示例也已经重新改写，充分利用了标准库设施。我们也对语言主题叙述的先后次序进行了重新编排，使讲解更加流畅。

除重新组织内容外，为了便于读者理解，我们还增加了几个新的环节。每一章都新增了“小结”和“术语”，概括本章要点。读者可以利用这些部分进行自我检查；如果发现还有不理解的概念，可以重新学习该章中的相关部分。

书中还加入了下述几种学习辅助手段：

- 重要术语用**黑体**表示，我们认为读者已经熟悉的重要术语则用**楷体**表示。这些术语都会出现在章后的“术语”部分。
- 书中用特殊版式突出标注的文字，是为了向读者提醒语言的重要特征，警示常见的错误，标明良好的编程实践，列出通用的使用技巧。希望这些标注可以帮助读者更快地消化重要概念，避免犯常见错误。
- 为了更易于理解各种特征或概念间的关系，书中大量使用了前后交叉引用。
- 对于某些重要概念和 C++ 新手最头疼的问题，我们进行了额外的讨论和解释。这部分也以特殊版式标出。
- 学习任何程序设计语言都需要编写程序。因此，本书提供了大量的示例。所有示例的源代码可从下列网址获得：

http://www.awprofessional.com/cpp_primer

万变不离其宗，本书保持了前几版的特色，仍然是一部全面介绍 C++ 的教程。我们的目标是提供一本清晰、全面、准确的指南性读物。我们通过讲解一系列示例来教授 C++ 语言，示例除了解释语言特征外，还展示了如何善用这门语言。虽然读者不需要事先学过 C 语言 (C++ 最初的基

础) 的知识, 但我们假定读者已经掌握了一种现代结构化语言。

本书结构

本书介绍了 C++ 国际标准, 既涵盖语言的特征, 又讲述了也是标准组成部分的丰富标准库。C++ 的强大很大程度上来自它支持抽象程序设计。要学会用 C++ 高效地编程, 只是掌握句法和语义是远远不够的。我们的重点在于, 教会读者怎样利用 C++ 的特性, 快速地写出安全的而且性能可与 C 语言低层程序相媲美的程序。

C++ 是一种大型的编程语言, 这可能会吓倒一些新手。现代 C++ 可以看成由以下三部分组成:

- 低级语言, 多半继承自 C。
- 更高级的语言特征, 用户可以借此定义自己的数据类型, 组织大规模的程序和系统。
- 标准库, 使用上述高级特征提供一整套有用的数据结构和算法。

多数 C++ 教材按照下面的顺序展开: 先讲低级细节, 再介绍更高级的语言特征; 在讲完整个语言后才开始解释标准库。结果往往使读者纠缠于低级的程序设计问题和复杂类型定义的编写等细节, 而不能真正领会抽象编程的强大, 更不用说学到足够的知识去创建自己的抽象了。

本版中我们独辟蹊径。一开始就讲述语言的基础知识和标准库, 这样读者就可以写出比较大的有实际意义的程序来。透彻阐释了使用标准库 (并且用标准库编写了各种抽象程序) 的基础知识之后, 我们才进入下一步, 学习用 C++ 的其他高级特征来编写自己的抽象。

第一和第二部分讨论语言的基础知识和标准库设施。其重点在于学会如何编写 C++ 程序, 如何使用标准库提供的抽象设施。大部分 C++ 程序员需要了解本书这两部分的内容。

除了讲解基础知识以外, 这两部分还有另外一个重要的意图。标准库设施本身是用 C++ 编写的抽象数据类型, 定义标准库所使用的是任何 C++ 程序员都能使用的构造类的语言特征。我们教授 C++ 的经验说明, 一开始就使用设计良好的抽象类型, 读者会更容易理解如何建立自己的类型。

第三到第五部分着重讨论如何编写自己的类型。第三部分介绍 C++ 的核心, 即对类的支持。类机制提供了编写自定义抽象的基础。类也是第四部分中讨论的面向对象编程和泛型编程的基础。全书正文的最后是第五部分, 这一部分讨论了一些高级特征, 它们在构建大型复杂系统时最为常用。

致谢

与前几版一样, 我们要感谢 Bjarne Stroustrup, 他不知疲倦地从事着 C++ 方面的工作, 他与我们的深厚友情由来已久。我们还要感谢 Alex Stepanov, 正是他最初凭借敏锐的洞察力创造了容器和算法的概念, 这些概念最终形成了标准库的核心。此外, 我们要感谢 C++ 标准委员会的所有成员, 他们多年来为 C++ 澄清概念、细化标准和改进功能付出了艰苦的努力。

我们要衷心地感谢本书的审稿人, 他们审阅了我们的多份书稿, 帮助我们对本书进行了无数大大小小的修改。他们是 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 和 Steve Vinoski。

书中所有示例都已通过 GNU 和微软编译器的编译。感谢他们的开发者和所有开发其他 C++ 编译器的人，是他们使 C++ 变成现实。

最后，感谢 Addison-Wesley 的工作人员，他们引领了这一版的整个出版过程：Debbie Lafferty——我们最初的编辑，是他提出出版本书的新版，他从本书最初版本起就一直致力于本书；Peter Gordon——我们的新编辑，他坚持更新和精简本书内容，极大地改进了这一版本；Kim Boedigheimer——他保证了我们所有人能按进度工作；还有 Tyrrell Albaugh、Jim Markham、Elizabeth Ryan 和 John Fuller，他们和我们一起经历了整个设计和制作过程。

{ĐÊ ç Ô's<ú

ì' {2! >•h•Ú •t X+h%t ÿìUì Y©qè•\ ÿÔ;htô!•“l F©•

r• •èË&ì'ÿpÈMäh ruæ,ÿì'pQ©lp » ÿqÓ{h2%81Î •X

+

§^ » öl Ó«•'îîšìÿ&ö÷ì O%u÷Ñr:héÒ[vhβO-î[r:h{òVê
l 2ìp βO-ÿî[!.3+ / éÒ[v K [v=l ¬{ {2 2Cÿ{•lÓ!¢ `ÿÕp
î^k•îÀt!l•» ÿqÓ{hì'FOL @KJAP lB? R>ÿ" \î•À l• Š³Ò Ê ÒNâ...ì
p» WìFOL-@KJAPÀ ïf¯ÿ_• %w>õý•t•æ»

•rî~ÿ áH {2...§^\t r•...! •t b•<§^h†1ü+\$!•t b §^{Òh
ÿwì©ì=OL JAP * ì• Èy •Ü! JAP F AA LDL §^•h Úh4¶N_ ðl••h%r
ÿe•vy Yu•'ìÿOMHOANRAN~ ?OO~ =F=T~ v û[•>~hHtβÿ æìW „Ö€ù |l
HtβW°ã š l^Ëy ™ ïï»+r...Wr§^ ïï ©æ•? €ì' æ•q€•? •Úæ
l••? ì!•©)h...§^Jh l•©) ps ¥ÿxè „ðÓÓ b• •r•kk³à! e
á Úh« JAPìp Æ•háF AAÿs7ÿhá=LE{2 <•héÒ[v % ³[• y4æ ì'
ÿ¢ðŠ ••rèË&ì'ÿMäæ l•» ÿqÓ{!šîÀ

ÂÈ JAP-F=R=ÿN_?t Øð!|{ éÒ[vl• Ê tÕp l•» !fî‡»ÿh „Óh
ýÈt ýš!Èh©lB?Š JAPP¬Ù¶Ö¼§^kæp•hfÿÚhÊ •‡»k• Ü t•h!•©
•ràkàt! l » ...žp t bïÿVê t b €ÿV ©ì JAPÿ á9Àáée•Ü¶
÷t÷ð ¥ü r ð•t b„§^ltâð „Ó©Ê N_ Ót bïÿVê €•Ók•Ü t
•hÊýû E÷ðÿhtÈ¥ N_ĩ< hÊÓ©•ÜïÿVêyWð ÀfÿÚ

Ó{ ¾'ÿ± wJÓ•ûÿ» hβ>÷Ñr:ŠO%ìÿtPhx^ 2Cÿ{•JÆ¾8 xJ
¬SD•WÁuUÿ÷Ñr: β>t•ÜéÒ[vhxJ••Ül•N_'pÿVê ï'k »™ éÒ[v
ÿ t•Ü|{h{òVêhx^ b2CEÆ ÓË„ %ÓË éh t•{ Êòÿ±

β>...!ìp» ÒNâ Wì JAPhF=R=l•#by ÀÚh#s+hì'ÿ[ÿ©,ĐË•h%wtÊ
© ph ™q ~hIR? KNll•A:hÓhÀð•t ý„§^ðh •ÚlpÀh•Ú...ìp %Ó«
¬{ h4¶'2hêβ^Rl•» ìp• W l• k³ÖtbEpb

î tÓîpxñ p©,šP% ,#2 6+*ll©,ªÈû¥hk β>æ,ÿ» t•ûh ì ©
šp!pÆ ÒJhh• ! &ht• ! Kÿªÿ l•pËÿ©,û •èËF ÿVêhÓ< f
ÿì'©,H sšCh••ûûÿ«» ïìph!•l•» hÓÕpe•p©,• f •

Nâ{2E÷òì! ! [v=LE{2h „ï' ØFF hì'¯' •á¥ZVê ìp•? !
{2Š» h Wì'lB? JAPl• Às757h%wf.û äêì'4 {2§Ó € hì'» 2Š
4!ÿ2Cêβ§Ó > e^β>Óòì4 h 4ì'y^h %tÀ hH >h%^C4hβ>òì'
y^h 4ì4 h•èÓh§Ó € ' •ìhª9Ëk

{2 Þsk-hß>ì‘ JAP IB? R>IþÎÿùÞŠ'þ Ê- ½R h » ÿì‘ - hÆq
½R‰ -t•û tk•”„Ê>ÿ fË¥Àà„ ® Â •Åáb„§^k...!ôô hÅb„
ôÿË¬ 4 Åh /û k -l‰hô ¾1‘oŠyNâk‘-Õ=÷Õ âÿ I¢Uÿ
-h}ÀËÇúÅY.rk‘•†H\ÂÓÿ•Ž pŠÊ.-Ñ]>>H©Šyhn™k Æ k¬ÿi
-hñgkÕÅáb„ÂÓs•ô ÿ » ÿh• qÓÿÖt {2•Þs Ê[hL»...žÞ !
ÖïÀHÔÿh‰w h#ÊhïÒ•#8q l•F ĩŠ +t ...!ď•t Ê¢ÿt•ìh‰ •
´m³ÕÞÿ»

l• €¥ÿ;Õhüÿt f¥Zhî k³ l•&ì´ÿpÈh Ý óyÎ ÖøÀ!šÎÀ
ÿ „•ÿÚÎ < h•«l•&ì´ÿMäÿùBøê”•L@B Î ©s¬s6• 2% Î !81 L@B
a- DPPL >>O PDAEPDKIA ?KI NA=@ DPI PE@ DPIH
ü•l^Ë ª <„ØÎ ÓžÞ•ûÿ» htÓh´R{ ÿ©,hžÞ» Wië•©,-
X.ºãh æ/ÿ_ À Þÿ

» l^qÓxJìÞïöI•ÚhìÞï´E÷ Él•» 2šÿ[ÿÿ¬ È Ö•pÿ’s ìO
‰ é {2ç< ± îö þlªo©+é³q+’ÿÐ±p l l•pÿ’s t¨•81•´žÞ» h‰w
•´ÿ2C¥ï•• ĩ³Îÿ_•h Ýà 6 ól•’šyh•´ÿ2CŠ+À= ²ď È‰l
•’š?tÞ !•î• \ÿ h¥ ÿîïďI•’\ô0ÿ•tË l•’Yå{êé æ, !4ÿôÀ
l•^„ ‡ ™ <!MÖ€é³hî úwáéì‘l•’šh Àìÿ .û Ô¤ !% lJ •
„³ÿÿì‘žs•ûÿ»

+é³×§j >>O PDAEPDKIA ?KI
„ßO- ì^–» tÐ•ûÿ"\È ÿ{hÓÆq -
„ì‘^Ú²pÿ’s hÈ E ’•’šhxJk -
„ì‘^Ý×Ö€ž”ÿªÿh ìÞhtÀe^^„
q+’ B áéh vhÀìÿ .û Ô¤
áéÿk pÿ’s hqÖ+kÀâ•´a î• q+’ áé f!Æ‰ÿ

目 录

第 1 章 快速入门	1
1.1 编写简单的 C++ 程序	2
1.2 初窥输入/输出	5
1.2.1 标准输入与输出对象	5
1.2.2 一个使用 IO 库的程序	5
1.3 关于注释	8
1.4 控制结构	10
1.4.1 while 语句	10
1.4.2 for 语句	12
1.4.3 if 语句	14
1.4.4 读入未知数目的输入	15
1.5 类的简介	17
1.5.1 Sales_item 类	17
1.5.2 初窥成员函数	19
1.6 C++ 程序	21
小结	22
术语	22

第一部分 基本语言

第 2 章 变量和基本类型	29
2.1 基本内置类型	30
2.1.1 整型	30
2.1.2 浮点型	32
2.2 字面值常量	34
2.3 变量	38
2.3.1 什么是变量	39
2.3.2 变量名	40
2.3.3 定义对象	42
2.3.4 变量初始化规则	44
2.3.5 声明和定义	45
2.3.6 名字的作用域	46
2.3.7 在变量使用处定义变量	48
2.4 const 限定符	49
2.5 引用	50
2.6 typedef 名字	53
2.7 枚举	53

2.8 类类型	54
2.9 编写自己的头文件	57
2.9.1 设计自己的头文件	58
2.9.2 预处理器的简单介绍	60
小结	62
术语	62

第 3 章 标准库类型	67
3.1 命名空间的 using 声明	68
3.2 标准库 string 类型	70
3.2.1 string 对象的定义和初始化	70
3.2.2 String 对象的读写	71
3.2.3 string 对象的操作	72
3.2.4 string 对象中字符的处理	76
3.3 标准库 vector 类型	78
3.3.1 vector 对象的定义和初始化	79
3.3.2 vector 对象的操作	81
3.4 迭代器简介	83
3.5 标准库 bitset 类型	88
3.5.1 bitset 对象的定义和初始化	88
3.5.2 bitset 对象上的操作	90
小结	92
术语	92

第 4 章 数组和指针	95
4.1 数组	96
4.1.1 数组的定义和初始化	96
4.1.2 数组操作	99
4.2 指针的引入	100
4.2.1 什么是指针	100
4.2.2 指针的定义和初始化	101
4.2.3 指针操作	104
4.2.4 使用指针访问数组元素	106
4.2.5 指针和 const 限定符	110
4.3 C 风格字符串	113
4.3.1 创建动态数组	117
4.3.2 新旧代码的兼容	120

4.4 多维数组	122	6.6 switch 语句	172
小结	124	6.6.1 使用 switch	173
术语	125	6.6.2 switch 中的控制流	173
第 5 章 表达式	127	6.6.3 default 标号	175
5.1 算术操作符	129	6.6.4 switch 表达式与 case 标号	176
5.2 关系操作符和逻辑操作符	131	6.6.5 switch 内部的变量定义	176
5.3 位操作符	134	6.7 while 语句	177
5.3.1 bitset 对象或整型值的使用	135	6.8 for 循环语句	179
5.3.2 将移位操作符用于 IO	137	6.8.1 省略 for 语句头的某些部分	180
5.4 赋值操作符	137	6.8.2 for 语句头中的多个定义	181
5.4.1 赋值操作的右结合性	138	6.9 do while 语句	182
5.4.2 赋值操作具有低优先级	138	6.10 break 语句	183
5.4.3 复合赋值操作符	139	6.11 continue 语句	184
5.5 自增和自减操作符	140	6.12 goto 语句	185
5.6 箭头操作符	142	6.13 try 块和异常处理	186
5.7 条件操作符	143	6.13.1 throw 表达式	186
5.8 sizeof 操作符	144	6.13.2 try 块	187
5.9 逗号操作符	145	6.13.3 标准异常	189
5.10 复合表达式的求值	145	6.14 使用预处理器进行调试	190
5.10.1 优先级	145	小结	192
5.10.2 结合性	146	术语	192
5.10.3 求值顺序	148	第 7 章 函数	195
5.11 new 和 delete 表达式	150	7.1 函数的定义	196
5.12 类型转换	154	7.1.1 函数返回类型	197
5.12.1 何时发生隐式类型转换	154	7.1.2 函数形参表	198
5.12.2 算术转换	155	7.2 参数传递	199
5.12.3 其他隐式转换	156	7.2.1 非引用形参	199
5.12.4 显式转换	158	7.2.2 引用形参	201
5.12.5 何时需要强制类型转换	158	7.2.3 vector 和其他容器类型的 形参	206
5.12.6 命名的强制类型转换	158	7.2.4 数组形参	206
5.12.7 旧式强制类型转换	160	7.2.5 传递给函数的数组的处理	209
小结	161	7.2.6 main: 处理命令行选项	210
术语	162	7.2.7 含有可变形参的函数	211
第 6 章 语句	165	7.3 return 语句	211
6.1 简单语句	166	7.3.1 没有返回值的函数	212
6.2 声明语句	167	7.3.2 具有返回值的函数	212
6.3 复合语句 (块)	167	7.3.3 递归	216
6.4 语句作用域	168	7.4 函数声明	217
6.5 if 语句	169	7.5 局部对象	220

7.5.1 自动对象	220	9.3.2 begin 和 end 成员	273
7.5.2 静态局部对象	220	9.3.3 在顺序容器中添加元素	273
7.6 内联函数	221	9.3.4 关系操作符	277
7.7 类的成员函数	222	9.3.5 容器大小的操作	278
7.7.1 定义成员函数的函数体	223	9.3.6 访问元素	279
7.7.2 在类外定义成员函数	225	9.3.7 删除元素	280
7.7.3 编写 Sales_item 类的构造 函数	225	9.3.8 赋值与 swap	282
7.7.4 类代码文件的组织	227	9.4 vector 容器的自增长	284
7.8 重载函数	228	9.5 容器的选用	287
7.8.1 重载与作用域	230	9.6 再谈 string 类型	289
7.8.2 函数匹配与实参转换	231	9.6.1 构造 string 对象的其他方法	290
7.8.3 重载确定的三个步骤	232	9.6.2 修改 string 对象的其他方法	292
7.8.4 实参类型转换	234	9.6.3 只适用于 string 类型的操作	293
7.9 指向函数的指针	237	9.6.4 string 类型的查找操作	295
小结	239	9.6.5 string 对象的比较	298
术语	240	9.7 容器适配器	300
第 8 章 标准 IO 库	243	9.7.1 栈适配器	301
8.1 面向对象的标准库	244	9.7.2 队列和优先级队列	302
8.2 条件状态	247	小结	303
8.3 输出缓冲区的管理	249	术语	303
8.4 文件的输入和输出	251	第 10 章 关联容器	305
8.4.1 文件流对象的使用	251	10.1 引言: pair 类型	306
8.4.2 文件模式	254	10.2 关联容器	308
8.4.3 一个打开并检查输入文件的 程序	256	10.3 map 类型	309
8.5 字符串流	257	10.3.1 map 对象的定义	309
小结	259	10.3.2 map 定义的类型	310
术语	259	10.3.3 给 map 添加元素	311
第 2 部分 容器和算法		10.3.4 使用下标访问 map 对象	311
第 9 章 顺序容器	263	10.3.5 map::insert 的使用	313
9.1 顺序容器的定义	264	10.3.6 查找并读取 map 中的元素	315
9.1.1 容器元素的初始化	265	10.3.7 从 map 对象中删除元素	316
9.1.2 容器内元素的类型约束	267	10.3.8 map 对象的迭代遍历	316
9.2 迭代器和迭代器范围	268	10.3.9 “单词转换” map 对象	317
9.2.1 迭代器范围	270	10.4 set 类型	319
9.2.2 使迭代器失效的容器操作	271	10.4.1 set 容器的定义和使用	319
9.3 顺序容器的操作	272	10.4.2 创建“单词排除”集	321
9.3.1 容器定义的类型别名	272	10.5 multimap 和 multiset 类型	322
		10.5.1 元素的添加和删除	322
		10.5.2 在 multimap 和 multiset 中查找元素	323

第三部分 类和数据抽象

14.9.2	转换操作符	455
14.9.3	实参匹配和转换	458
14.9.4	重载确定和类的实参	461
14.9.5	重载、转换和操作符	464
小结		466
术语		467

第四部分 面向对象编程与泛型编程

第 15 章	面向对象编程	471
15.1	面向对象编程：概述	472
15.2	定义基类和派生类	473
15.2.1	定义基类	474
15.2.2	protected 成员	475
15.2.3	派生类	476
15.2.4	virtual 与其他成员函数	479
15.2.5	公用、私有和受保护的继承	482
15.2.6	友元关系与继承	486
15.2.7	继承与静态成员	486
15.3	转换与继承	487
15.3.1	派生类到基类的转换	487
15.3.2	基类到派生类的转换	489
15.4	构造函数和复制控制	490
15.4.1	基类构造函数和复制控制	490
15.4.2	派生类构造函数	490
15.4.3	复制控制和继承	494
15.4.4	虚析构函数	495
15.4.5	构造函数和析构函数中的 虚函数	497
15.5	继承情况下的类作用域	497
15.5.1	名字查找在编译时发生	498
15.5.2	名字冲突与继承	498
15.5.3	作用域与成员函数	499
15.5.4	虚函数与作用域	500
15.6	纯虚函数	502
15.7	容器与继承	503
15.8	句柄类与继承	504
15.8.1	指针型句柄	505
15.8.2	复制未知类型	507
15.8.3	句柄的使用	508
15.9	再谈文本查询示例	511

15.9.1	面向对象的解决方案	513
15.9.2	值型句柄	514
15.9.3	Query_base 类	515
15.9.4	Query 句柄类	516
15.9.5	派生类	518
15.9.6	eval 函数	520
小结		522
术语		523

第 16 章	模板与泛型编程	525
16.1	模板定义	526
16.1.1	定义函数模板	526
16.1.2	定义类模板	528
16.1.3	模板形参	529
16.1.4	模板类型形参	531
16.1.5	非类型模板形参	533
16.1.6	编写泛型程序	534
16.2	实例化	535
16.2.1	模板实参推断	537
16.2.2	函数模板的显式实参	540
16.3	模板编译模型	542
16.4	类模板成员	545
16.4.1	类模板成员函数	548
16.4.2	非类型形参的模板实参	551
16.4.3	类模板中的友元声明	552
16.4.4	Queue 和 QueueItem 的 友元声明	554
16.4.5	成员模板	556
16.4.6	完整的 Queue 类	558
16.4.7	类模板的 static 成员	559
16.5	一个泛型句柄类	560
16.5.1	定义句柄类	561
16.5.2	使用句柄	562
16.6	模板特化	564
16.6.1	函数模板的特化	565
16.6.2	类模板的特化	567
16.6.3	特化成员而不特化类	569
16.6.4	类模板的部分特化	570
16.7	重载与函数模板	570
小结		573
术语		574

第五部分 高级主题

第 17 章 用于大型程序的工具.....	579	第 18 章 特殊工具与技术.....	631
17.1 异常处理.....	580	18.1 优化内存分配.....	632
17.1.1 抛出类类型的异常.....	581	18.1.1 C++中的内存分配.....	632
17.1.2 栈展开.....	582	18.1.2 allocator 类.....	633
17.1.3 捕获异常.....	583	18.1.3 operator new 函数和	
17.1.4 重新抛出.....	585	operator delete 函数.....	636
17.1.5 捕获所有异常的处理代码.....	586	18.1.4 定位 new 表达式.....	638
17.1.6 函数测试块与构造函数.....	586	18.1.5 显式析构函数的调用.....	639
17.1.7 异常类层次.....	587	18.1.6 类特定的 new 和 delete.....	639
17.1.8 自动资源释放.....	589	18.1.7 一个内存分配器基类.....	641
17.1.9 auto_ptr 类.....	591	18.2 运行时类型识别.....	646
17.1.10 异常说明.....	595	18.2.1 dynamic_cast 操作符.....	647
17.1.11 函数指针的异常说明.....	598	18.2.2 typeid 操作符.....	649
17.2 命名空间.....	599	18.2.3 RTTI 的使用.....	650
17.2.1 命名空间的定义.....	599	18.2.4 type_info 类.....	652
17.2.2 嵌套命名空间.....	603	18.3 类成员的指针.....	653
17.2.3 未命名的命名空间.....	604	18.3.1 声明成员指针.....	653
17.2.4 命名空间成员的使用.....	606	18.3.2 使用类成员的指针.....	655
17.2.5 类、命名空间和作用域.....	609	18.4 嵌套类.....	658
17.2.6 重载与命名空间.....	612	18.4.1 嵌套类的实现.....	658
17.2.7 命名空间与模板.....	614	18.4.2 嵌套类作用域中的名字查找.....	661
17.3 多重继承与虚继承.....	614	18.5 联合：节省空间的类.....	662
17.3.1 多重继承.....	615	18.6 局部类.....	665
17.3.2 转换与多个基类.....	617	18.7 固有的不可移植的特征.....	666
17.3.3 多重继承派生类的复制控制.....	619	18.7.1 位域.....	666
17.3.4 多重继承下的类作用域.....	620	18.7.2 volatile 限定符.....	668
17.3.5 虚继承.....	622	18.7.3 链接指示：extern "C".....	669
17.3.6 虚基类的声明.....	624	小结.....	672
17.3.7 特殊的初始化语义.....	625	术语.....	673
小结.....	628	附录 标准库.....	675
术语.....	628	索引.....	703

第 1 章

快速入门

目录

1.1 编写简单的 C++ 程序.....	2
1.2 初窥输入/输出.....	5
1.3 关于注释.....	8
1.4 控制结构.....	10
1.5 类的简介.....	17
1.6 C++ 程序.....	21
小结.....	22
术语.....	22

本章介绍 C++ 的大部分基本要素：内置类型、库类型、类类型、变量、表达式、语句和函数。在这一过程中还会简要说明如何编译和运行程序。

读者读完本章内容并做完练习，就应该可以编写、编译和执行简单的程序。后面的章节会进一步阐明本章所介绍的主题。

要学会一门新的程序设计语言，必须实际动手编写程序。在这一章，我们将编写程序解决一个简单的数据处理问题：某书店以文件形式保存其每一笔交易。每一笔交易记录某本书的销售情况，含有 ISBN（国际标准书号，世界上每种图书的唯一标识符）、销售册数和销售单价。每一笔交易形如：

0-201-70353-X 4 24.99

第一个元素是 ISBN，第二个元素是销售的册数，最后是销售单价。店主定期地查看这个文件，统计每本书的销售册数、总销售收入以及平均售价。我们要编写程序来进行这些计算。

在编写这个程序之前，必须知道 C++ 的一些基本特征。至少我们要知道怎么样去编写、编译和执行简单的程序。这个程序要做什么呢？虽然还没有设计出解决方案，但是我们知道程序必须：

- 定义变量。
- 实现输入和输出。
- 定义数据结构来保存要处理的数据。
- 测试是否两条记录具有相同的 ISBN。
- 编写循环，处理交易文件中的每一条记录。

我们将首先考察 C++ 的这些部分，然后编写书店问题的解决方案。

1.1 编写简单的 C++ 程序

每个 C++ 程序都包含一个或多个函数，而且必须有一个命名为 **main**。函数由执行函数功能的语句序列组成。操作系统通过调用 main 函数来执行程序，main 函数则执行组成自己的语句并返回一个值给操作系统。

下面是一个简单的 main 函数，它不执行任何功能，只是返回一个值：

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

操作系统通过 main 函数返回的值来确定程序是否成功执行完毕。返回 0 值表明程序成功执行完毕。

main 函数在很多方面都比较特别，其中最重要的是每个 C++ 程序必须含有 main 函数，且 main 函数是（唯一）被操作系统显式调用的函数。

定义 main 函数和定义其他函数一样。定义函数必须指定 4 个元素：返回类型、函数名、圆括号内的形参表（可能为空）和函数体。main 函数的形参个数是有限的。本例中定义的 main 函数形参表为空。7.2.6 节将介绍 main 函数中可以定义的其他形参。

main 函数的返回值必须是 int 型，该类型表示整数。int 类型是**内置类型**，即该类型是由 C++ 语言定义的。

函数体是函数定义的最后部分，是以花括号开始并以花括号结束的语句块：

```
{
    return 0;
}
```

例中唯一的语句就是 `return`，该语句终止函数。



注意 `return` 语句后面的分号。在 C++ 中多数语句以分号作为结束标记。分号很容易被忽略，而漏写分号将会导致莫名其妙的编译错误信息。

当 `return` 带上一个值（如 0）时，这个值就是函数的返回值。返回值类型必须和函数的返回类型相同，或者可以转换成函数的返回类型。对于 `main` 函数，返回类型必须是 `int` 型，0 是 `int` 型的。

在大多数系统中，`main` 函数的返回值是一个状态指示器。返回值 0 往往表示 `main` 函数成功执行完毕。任何其他非零的返回值都有操作系统定义的含义。通常非零返回值表明有错误出现。每一种操作系统都有自己的方式告诉用户 `main` 函数返回什么内容。

编译与执行程序

程序编写完后需要进行编译。如何进行编译，与具体操作系统和编译器有关。你需要查看有关参考手册或者询问有经验的同事，以了解所用的编译器的工作细节。

许多基于 PC 的编译器都在集成开发环境（IDE）中运行，IDE 将编译器与相关的构建和分析工具绑定在一起。这些环境在开发复杂程序时非常有用，但掌握起来需要花费一点时间。通常这些环境包含点击式界面，程序员在此界面下可以编写程序，并使用各种菜单来编译与执行程序。本书不介绍怎样使用这些环境。

大多数编译器，包括那些来自 IDE 的，都提供了命令行界面。除非你已经很熟悉你的 IDE，否则从使用简单的命令行界面开始可能更容易些。这样可以避免在学习语言之前得先去学习 IDE。

3

调用 GNU 或微软编译器

调用 C++ 编译器的命令因编译器和操作系统的不同而不同，常用的编译器是 GNU 编译器和微软 Visual Studio 编译器。调用 GNU 编译器的默认命令是 `g++`：

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

这里的 `$` 是系统提示符。这个命令产生一个名为 `prog1` 或 `prog1.exe` 的可执行文件。在 UNIX 系统下，可执行文件没有后缀；而在 Windows 下，后缀为 `.exe`。 `-o prog1` 是编译器参数以及用来存放可执行文件的文件名。如果省略 `-o prog1`，那么编译器在 UNIX 系统下产生名为 `a.out` 而在 Windows 下产生名为 `a.exe` 的可执行文件。

微软编译器采用命令 `cl` 来调用：

```
C:\directory> cl -GX prog1.cpp
```

这里的 `C:\directory>` 是系统提示符，`directory` 是当前目录名。`cl` 是调用编译器的命令，`-GX` 是一个选项，该选项在使用命令行界面编译程序时是必需的。微软编译器自动产生与源文件同名的可执行文件，这个可执行文件具有 `.exe` 后缀且与源文件同名。本例中，可执行文件命名为 `prog1.exe`。

更多的信息请参考你的编译器用户指南。

1. 程序源文件命名规范

不管我们使用命令行界面还是 IDE，大多数编译器希望待编译的程序保存在文件中。程序文件称作源文件。大多数系统中，源文件的名称由文件名（如 prog1）和文件后缀两部分组成。依据惯例，文件后缀表明该文件是程序。文件后缀通常也表明程序是用什么语言编写的，以及选择哪一种编译器运行。我们用来编译本书实例的系统将带有后缀 .cc 的文件视为 C++ 程序，因此我们将该程序保存为：

```
prog1.cc
```

C++ 程序文件的后缀与所运行的具体编译器有关。其他的形式还包括：

```
prog1.cxx
```

```
prog1.cpp
```

```
prog1.cp
```

```
prog1.C
```

4

2. 从命令行运行编译器

如果使用命令行界面，一般在控制台窗口（例如 UNIX 的 shell 窗口或 Windows 的命令提示窗口）编译程序。假设 main 程序在名为 prog1.cc 的文件中，可以使用如下命令来编译：

```
$ CC prog1.cc
```

这里 CC 是编译器命令名，\$ 表示系统提示符。编译器输出一个可执行文件，我们可以按名调用这个可执行文件。在我们的系统中，编译器产生一个名为 a.exe 的可执行文件。UNIX 编译器则会将可执行文件放到一个名为 a.out 的文件中。要运行可执行文件，可在命令行提示符处给出该文件名：

```
$ a .exe
```

执行编译过的程序。在 UNIX 系统中，即使当前目录，有时还必须指定文件所在的目录。这种情况下，键入：

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

“.” 后面的斜杠表明文件处于当前目录下。

访问 main 函数的返回值的方式和系统有关。不论 UNIX 还是 Windows 系统，执行程序后，必须发出一个适当的 echo 命令。UNIX 系统中，通过键入如下命令获取状态：

```
$ echo $?
```

要在 Windows 系统下查看状态，键入

```
C:\directory> echo %ERRORLEVEL%
```

习题

习题 1.1 查看所用的编译器文档，了解它所用的文件命名规范。编译并运行本节的 main 程序。

习题 1.2 修改程序使其返回 -1。返回值 -1 通常作为程序运行失败的指示器。然而，系统不同，如何（甚至是否）报告 main 函数运行失败也不同。重新编译并再次运行程序，看看你的系统如何处理 main 函数的运行失败指示器。

1.2 初窥输入/输出

C++并没有直接定义进行输入或输出（IO）的任何语句，这种功能是由**标准库**提供的。IO库提供了大量的设施。然而，对许多应用，包括本书的例子而言，编程者只需要了解一些基本概念和操作。

5

本书的大多数例子都使用了处理格式化输入和输出的 **iostream** 库。iostream 库的基础是两种命名为 **istream** 和 **ostream** 的类型，分别表示输入流和输出流。流是指要从某种 IO 设备上读入或写出的字符序列。术语“流”试图说明字符是随着时间顺序生成或消耗的。

1.2.1 标准输入与输出对象

标准库定义了 4 个 IO 对象。处理输入时使用命名为 **cin**（读作 see-in）的 **istream** 类型对象。这个对象也称为**标准输入**。处理输出时使用命名为 **cout**（读作 see-out）的 **ostream** 类型对象，这个对象也称为**标准输出**。标准库还定义了另外两个 **ostream** 对象，分别命名为 **cerr** 和 **clog**（分别读作“see-err”和“see-log”）。**cerr** 对象又叫作**标准错误**，通常用来输出警告和错误信息给程序的使用者。而 **clog** 对象用于产生程序执行的一般信息。

一般情况下，系统将这些对象与执行程序的窗口联系起来。这样，当我们从 **cin** 读入时，数据从执行程序的窗口读入，当写到 **cout**、**cerr** 或 **clog** 时，输出写至同一窗口。运行程序时，大部分操作系统都提供了重定向输入或输出流的方法。利用重定向可以将这些流与所选择的文件联系起来。

1.2.2 一个使用 IO 库的程序

到目前为止，我们已经明白如何编译与执行简单的程序，虽然那个程序什么也不做。在开篇的书店问题中，有一些记录含有相同的 ISBN，需要将这些记录进行汇总，也就是说需要弄清楚如何累加已售出书籍的数量。

为了弄清楚如何解决这个问题，我们先来看应如何把两数相加。我们可以使用 IO 库来扩充 main 程序，要求用户给出两个数，然后输出它们的和：

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

6

程序首先在用户屏幕上显示提示语：

Enter two numbers:

然后程序等待用户输入。如果用户输入

跟着一个换行符，则程序产生下面的输出：

```
The sum of 3 and 7 is 10
```

程序的第一行是一个预处理指示：

```
#include <iostream>
```

告诉编译器要使用 `iostream` 库。尖括号里的名字是一个**头文件**。程序使用库工具时必须包含相关的头文件。`#include` 指示必须单独写成一行——头文件名和 `#include` 必须在同一行。通常，`#include` 指示应出现在任何函数的外部。而且习惯上，程序的所有 `#include` 指示都在文件开头部分出现。

1. 写入到流

`main` 函数体中第一条语句执行了一个**表达式**。C++中，一个表达式由一个或几个操作数和通常是一个操作符组成。该语句的表达式使用**输出操作符**（`<<`操作符），在标准输出上输出提示语：

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

这个语句用了两次输出操作符。每个输出操作符实例都接受两个操作数：左操作数必须是 `ostream` 对象；右操作数是要输出的值。操作符将其右操作数写到作为其左操作数的 `ostream` 对象。

C++中，每个表达式都会产生一个结果，通常是将操作符作用到其操作数所产生的值。当操作符是输出操作符时，结果是左操作数的值。也就是说，输出操作返回的值是输出流本身。

既然输出操作符返回的是其左操作数，那么我们就可以将输出请求链接在一起。输出提示语的那条语句等价于

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

因为 `(std::cout << "Enter two numbers:")` 返回其左操作数 `std::cout`，这条语句等价于

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

7

`endl` 是一个特殊值，称为**操纵符**（`manipulator`），将它写入输出流时，具有输出换行的效果，并刷新与设备相关联的缓冲区（`buffer`）。通过刷新缓冲区，用户可立即看到写入到流中的输出。



程序员经常在调试过程中插入输出语句，这些语句都应该刷新输出流。忘记刷新输出流可能会造成输出停留在缓冲区中，如果程序崩溃，将会导致对程序崩溃位置的错误推断。

2. 使用标准库中的名字

细心的读者会注意到这个程序中使用的是 `std::cout` 和 `std::endl`，而不是 `cout` 和 `endl`。前缀 `std::` 表明 `cout` 和 `endl` 是定义在**命名空间**（`namespace`）`std` 中的。使用命名空间，程序员可以避免由于无意中使用了与库中所定义名字相同的名字而引致冲突。因为标准库定义的名字

是定义在命名空间中，所以我们可以按自己的意图使用相同的名字。

标准库使用命名空间的副作用是，当我们使用标准库中的名字时，必须显式地表达出使用的是命名空间 `std` 下的名字。`std::cout` 的写法使用了作用域操作符（scope operator，`::`操作符），表示使用的是定义在命名空间 `std` 中的 `cout`。我们将在 3.1 节学习到程序中经常使用的避免这种冗长句法的方法。

3. 读入流

在输出提示语后，将读入用户输入的数据。先定义两个名为 `v1` 和 `v2` 的变量（variable）来保存输入：

```
int v1, v2;
```

将这些变量定义为 `int` 类型，`int` 类型是一种代表整数值的内置类型。这些变量未初始化（uninitialized），表示没有赋给它们初始值。这些变量在首次使用时会读入一个值，因此可以没有初始值。

下一条语句读取输入：

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

输入操作符（`>>`操作符）行为与输出操作符相似。它接受一个 `istream` 对象作为其左操作数，接受一个对象作为其右操作数，它从 `istream` 操作数读取数据并保存到右操作数中。像输出操作符一样，输入操作符返回其左操作数作为结果。由于输入操作符返回其左操作数，我们可以将输入请求序列合并成单个语句。换句话说，这个输入操作等价于：

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

8

输入操作的效果是从标准输入读取两个值，将第一个存放在 `v1` 中，第二个存放在 `v2` 中。

4. 完成程序

剩下的就是要输出结果：

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

这条语句虽然比输出提示语的语句长，但概念上没什么区别。它将每个操作数输出到标准输出。有趣的是操作数并不都是同一类型的值，有些操作数是字符串字面值。例如

```
"The sum of "
```

其他是各种 `int` 值，如 `v1`、`v2` 以及对算术表达式 `v1 + v2` 求值的结果。`iostream` 库定义了接受全部内置类型的输入输出操作符版本。



在写 C++ 程序时，大部分出现空格符的地方可用换行符代替。这条规则的一个例外是字符串字面值中的空格符不能用换行符代替。另一个例外是空格符不允许出现在预处理指示中。

关键概念：已初始化变量和未初始化变量

在 C++ 中，初始化是一个非常重要的概念，对它的讨论将贯穿本书始终。

已初始化变量是指变量在定义时就给定一个值。未初始化变量则未给定初始值：

```
int val1 = 0;           // initialized
int val2;              // uninitialized
```

给变量一个初始值几乎总是正确的，但不要求必须这样做。当我们确定变量在第一次使用时会赋一个新值，那就不需要创建初始值。例如，在本节开始我们的第一个有意义的程序中，定义了未初始化变量，并立即读取值给它们。

定义变量时，应该给变量赋初始值，除非确定将变量用于其他意图之前会覆盖这个初值。如果不能保证读取变量之前重置变量，就应该初始化变量。

9

习题

习题 1.3 编一个程序，在标准输出上打印 “Hello, World”。

习题 1.4 我们的程序利用内置的加法操作符 “+” 来产生两个数的和。编写程序，使用乘法操作符 “*” 产生两个数的积。

习题 1.5 我们的程序使用了一条较长的输出语句。重写程序，使用单独的语句打印每一个操作数。

习题 1.6 解释下面的程序段：

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

这段代码合法吗？如果合法，为什么？如果不合法，又为什么？

1.3 关于注释

在程序变得更复杂之前，我们应该明白 C++ 如何处理注释 (comment)。注释可以帮助其他人阅读程序，通常用于概括算法、确认变量的用途或者阐明难以理解的代码段。注释并不会增加可执行程序的大小，编译器会忽略所有注释。



本书中，注释排成斜体以区别于一般程序文本。实际程序中，注释文本是否区别于程序代码文本取决于编程环境是否完善。

C++ 中有单行注释和成对注释两种类型的注释。单行注释以双斜线 (//) 开头，行中处于双斜线右边的内容是注释，被编译器忽略。

另一种定界符，注释对 (/**/)，是从 C 语言继承过来的。这种注释以 “/*” 开头，以 “*/” 结尾。编译器把落入注释对 “/**/” 之间的内容作为注释：

```
#include <iostream>
/* Simple main function: Read two numbers and write their sum */
```

```

int main()
{
    // prompt user to enter two numbers
    std::cout << "Enter two numbers:" << std::endl;
    int v1, v2;           // uninitialized
    std::cin >> v1 >> v2; // read input
    return 0;
}

```

10

任何允许有制表符、空格或换行符的地方都允许放注释对。注释对可跨越程序的多行，但不是一定要如此。当注释跨越多行时，最好能直观地指明每一行都是注释。我们的风格是在注释的每一行以星号开始，指明整个范围是多行注释的一部分。

程序通常混用两种注释形式。注释对一般用于多行解释，而双斜线注释则常用于半行或单行的标记。

太多的注释混入程序代码可能会使代码难以理解，通常最好是将一个注释块放在所解释代码的上方。

代码改变时，注释应与代码保持一致。程序员即使知道系统其他形式的文档已经过期，还是会信任注释，认为它会是正确的。错误的注释比没有注释更糟，因为它会误导后来者。

注释对不可嵌套

注释总是以`/*`开始并以`*/`结束。这意味着，一个注释对不能出现在另一个注释对中。由注释对嵌套导致的编译器错误信息容易使人迷惑。例如，在你的系统上编译下面的程序：

```

#include <iostream>
/*
 * comment pairs /* */ cannot nest.
 * "cannot nest" is considered source code,
 * as is the rest of the program
 */
int main()
{
    return 0;
}

```

当注释掉程序的一大部分时，似乎最简单的办法就是在要临时忽略的区域前后放一个注释对。问题是如果那段代码已经有了注释对，那么新插入的注释对将提前终止。临时忽略一段代码更好的方法，是用编辑器在要忽略的每一行代码前面插入单行注释。这样，你就无需担心要注释的代码是否已包含注释对。

习题

习题 1.7 编译有不正确嵌套注释的程序。

习题 1.8 指出下列输出语句哪些（如果有）是合法的。

```

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

```

预测结果，然后编译包含上述三条语句的程序，检查你的答案。纠正所遇到的错误。

1.4 控制结构

语句总是顺序执行的：函数的第一条语句首先执行，接着是第二条，依次类推。当然，少数程序——包括我们将要编写的解决书店问题的程序——可以仅用顺序执行语句编写。事实上，程序设计语言提供了多种控制结构支持更为复杂的执行路径。本节将简要地介绍 C++ 提供的控制结构，第 6 章再详细介绍各种语句。

11

1.4.1 while 语句

while 语句提供了迭代执行功能。可以用 **while** 语句编写一个如下所示的从 1 到 10（包括 10）的求和程序：

```
#include <iostream>
int main()
{
    int sum = 0, val = 1;
    // keep executing the while until val is greater than 10
    while (val <= 10) {
        sum += val;    // assigns sum + val to sum
        ++val;        // add 1 to val
    }
    std::cout << "Sum of 1 to 10 inclusive is "
              << sum << std::endl;
    return 0;
}
```

编译并执行后，将输出：

Sum of 1 to 10 inclusive is 55

与前面一样，程序首先包含 `iostream` 头文件并定义 `main` 函数。在 `main` 函数中定义两个 `int` 型变量：`sum` 保存总和，`val` 表示从 1 到 10 之间的每一个值。我们给 `sum` 赋初值 0，而 `val` 则从 1 开始。

重要的部分是 **while** 语句。**while** 结构有这样的形式：

```
while (condition) while_body_statement;
```

while 通过测试 *condition*（条件）和执行相关的 *while_body_statement* 来重复执行，直到 *condition* 为假。

12

条件是一个可求值的表达式，所以可以测试其结果。如果结果值非零，那么条件为真；如果值为零，则条件为假。

如果 *condition* 为真（表达式求值不为零），则执行 *while_body_statement*。执行完后，再次测试 *condition*。如果 *condition* 仍为真，则再次执行 *while_body_statement*。**while** 语句一直交替测试 *condition* 和执行 *while_body_statement*，直到 *condition* 为假为止。

在这个程序中，**while** 语句是

```
// keep executing the while until val is greater than 10
while (val <= 10) {
```

```

    sum += val;    // assigns sum + val to sum
    ++val;        // add 1 to val
}

```

while 语句的条件用了小于或等于操作符 (`<=`操作符), 将 `val` 的当前值和 10 比较, 只要 `val` 小于或等于 10, 就执行 while 循环体。这种情况下, while 循环体是一个包含两个语句的块:

```

{
    sum += val;    // assigns sum + val to sum
    ++val;        // add 1 to val
}

```

块是被花括号括起来的语句序列。C++中, 块可用于任何可以用一条语句的地方。块中第一条语句使用了**复合赋值操作符** (`+=`操作符), 这个操作符把它的右操作数加至左操作数, 这等效于编写含一个加法和**一个赋值的语句**:

```
sum = sum + val; // assign sum + val to sum
```

因此第一条语句是把 `val` 的值加到 `sum` 的当前值, 并把结果存入 `sum`。

第二条语句

```
++val;    // add 1 to val
```

使用了**前自增操作符** (`++`操作符), 自增操作符就是在它的操作数上加 1, `++val` 和 `val=val+1` 是一样的。

执行 while 的循环体后, 再次执行 while 的条件。如果 `val` 的值 (自增后) 仍小于或等于 10, 那么再次执行 while 的循环体。循环继续, 测试条件并执行循环体, 直到 `val` 的值不再小于或等于 10 为止。

一旦 `val` 的值大于 10, 程序就跳出 while 循环并执行 while 后面的语句, 此例中该语句打印输出, 其后的 `return` 语句结束 main 程序。

13

关键概念: C++程序的缩排和格式

C++程序的格式非常自由, 花括号、缩排、注释和换行的位置通常对程序的语义没有影响。例如, 表示 main 函数体开始的花括号可以放在与 main 同一行, 也可以像我们那样, 放在下一行的开始, 或放在你喜欢的任何地方。唯一的要求是, 它是编译器所看到在 main 的参数列表的右括号之后的第一个非空白、非注释字符。

虽然说我们可以很自由地编排程序的格式, 但如果编排不当, 会影响程序的可读性。例如, 我们可以将 main 写成单独的一长行。这样的定义尽管合法, 但很难阅读。

关于什么是 C 或 C++程序的正确格式存在无休止的争论, 我们相信没有唯一正确的风格, 但一致性是有价值的。我们倾向于把确定函数边界的花括号自成一成一行, 且缩进复合的输入或输出表达式从而使操作符排列整齐, 正如 1.2.2 节的 main 函数中的输出语句那样。随着程序的复杂化, 其他缩排规范也会变得清晰。

可能存在其他格式化程序的方式, 记住这一点很重要。在选择格式化风格时, 要考虑提高程序的可读性, 使其更易于理解。一旦选择了某种风格, 就要始终如一地使用。

1.4.2 for 语句

在 while 循环中，我们使用变量 `val` 来控制循环执行次数。每次执行 while 语句，都要测试 `val` 的值，然后在循环体中增加 `val` 的值。

由于需要频频使用像 `val` 这样的变量控制循环，因而 C++ 语言定义了第二种控制结构，称为 **for 语句**，简化管理循环变量的代码。使用 for 循环重新编写求 1 到 10 的和的程序，如下：

```
#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;
}
```

在 for 循环之前，我们定义 `sum` 并赋 0 值。用于迭代的变量 `val` 被定义为 for 语句自身的一部分。for 语句

14

```
for (int val = 1; val <= 10; ++val)
    sum += val; // equivalent to sum = sum + val
```

包含 for 语句头和 for 语句体两部分。for 语句头控制 for 语句体的执行次数。for 语句头由三部分组成：一个初始化语句，一个条件，一个表达式。在这个例子中，初始化语句

```
int val = 1;
```

定义一个名为 `val` 的 `int` 对象并给定初始值 1。初始化语句仅在进入 for 语句时执行一次。条件

```
val <= 10
```

将 `val` 的当前值和 10 比较，每次经过循环都要测试。只要 `val` 小于或等于 10，就执行 for 语句体。仅当 for 语句体执行后才执行表达式。在这个 for 循环中，表达式使用前自增操作符，`val` 的值加 1，执行完表达式后，for 语句重新测试条件，如果 `val` 的新值仍小于或等于 10，则执行 for 语句体，`val` 再次自增，继续执行直到条件不成立。

在这个循环中，for 语句体执行求和

```
sum += val; // equivalent to sum = sum + val
```

for 语句体使用复合赋值操作符，把 `val` 的当前值加到 `sum`，并将结果保存到 `sum` 中。

扼要重述一下，for 循环总的执行流程为：

- (1) 创建 `val` 并初始化为 1。
- (2) 测试 `val` 是否小于或等于 10。
- (3) 如果 `val` 小于或等于 10，则执行 for 循环体，把 `val` 加到 `sum` 中。如果 `val` 大于 10，就退出循环，接着执行 for 语句体后的第一条语句。
- (4) `val` 递增。

(5) 重复第2步的测试，只要条件为真，就继续执行其余步骤。

退出 for 循环后，变量 val 不再可访问，循环终止后使用 val 是不可能的。然而，不是所有的编译器都有这一要求。



在标准化之前的 C++ 中，定义在 for 语句头的名字在 for 循环外是可访问的。语言定义中的这一改变，可能会使习惯于使用老式编译器的人，在使用遵循标准的新编译器时感到惊讶。

15

再谈编译

编译器的部分是寻找程序代码中的错误。编译器不能查出程序的意义是否正确，但它可以查出程序形式上的错误。下面是编译器能查出的最普遍的一些错误。

(1) 语法错误。程序员犯了 C++ 语言中的语法错误。下面代码段说明常见的语法错误；每个注释描述下一行的错误。

```
// error: missing ')' in parameter list for main
int main ( {
    // error: used colon, not a semicolon after endl
    std::cout << "Read each file." << std::endl:
    // error: missing quotes around string literal
    std::cout << Update master. << std::endl;
    // ok: no errors on this line
    std::cout << "Write new master." <<std::endl;
    // error: missing ';' on return statement
    return 0
}
```

(2) 类型错误。C++ 中每个数据项都有其相关联的类型。例如，值 10 是一个整数。用双引号标注起来的单词 “hello” 是字符串面值。类型错误的一个实例是传递了字符串面值给应该得到整型参数的函数。

(3) 声明错误。C++ 程序中使用的每个名字必须在使用之前声明。没有声明名字通常会导致错误信息。最常见的两种声明错误，是从标准库中访问名字时忘记使用 “std::”，以及由于疏忽而拼错标识符名：

```
#include <iostream>
int main()
{
    int v1, v2;
    std::cin >> v >> v2; // error: uses "v" not "v1"
    // cout not defined, should be std::cout
    cout << v1 + v2 << std::endl;
    return 0;
}
```

错误信息包含行号和编译器对我们所犯错误的简要描述。按错误报告的顺序改正错误是个好习惯，通常一个错误可能会产生一连串的影响，并导致编译器报告比实际多得多的错误。最好在每次修改后或最多改正了一些显而易见的错误后，就重新编译代码。这个循环就是众所周知的编辑-编译-调试。

16

习题

习题 1.9 下列循环做什么? sum 的最终值是多少?

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

习题 1.10 用 for 循环编程, 求从 50 到 100 的所有自然数的和。然后用 while 循环重写该程序。

习题 1.11 用 while 循环编程, 输出 10 到 0 递减的自然数。然后用 for 循环重写该程序。

习题 1.12 对比前面两个习题中所写的循环。两种形式各有何优缺点?

习题 1.13 编译器不同, 理解其诊断内容的难易程度也不同。编写一些程序, 包含本节“再谈编译”部分讨论的那些常见错误。研究编译器产生的信息, 这样你在编译更复杂的程序遇到这些信息时就不会陌生。

1.4.3 if 语句

求 1 到 10 之间数的和, 其逻辑延伸是求用户提供的两个数之间的数的和。可以直接在 for 循环中使用这两个数, 使用第一个输入值作为下界而第二个输入值作为上界。然而, 如果用户首先给定的数较大, 这种策略将会失败: 程序会立即退出 for 循环。因此, 我们应该调整范围以便较大的数作上界而较小的数作下界。这样做, 我们需要一种方式来判定哪个数更大一些。

像大多数语言一样, C++ 提供支持条件执行的 if 语句。使用 if 语句来编写修订的求和程序如下:

```
#include <iostream>
int main()
{
    std::cout << "Enter two numbers:" << std::endl;
    int v1, v2;
    std::cin >> v1 >> v2; // read input
    // use smaller number as lower bound for summation
    // and larger number as upper bound
    int lower, upper;
    if (v1 <= v2) {
        lower = v1;
        upper = v2;
    } else {
        lower = v2;
        upper = v1;
    }
    int sum = 0;
    // sum values from lower up to and including upper
    for (int val = lower; val <= upper; ++val)
        sum += val; // sum = sum + val

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

如果我们编译并执行这个程序给定输入数为 7 和 3，程序的输出结果将为：

Sum of 3 to 7 inclusive is 25

这个程序中大部分代码我们在之前的举例中已经熟知了。程序首先向用户输出提示并定义 4 个 int 变量，然后从标准输入读入值到 v1 和 v2 中。仅有 if 条件语句是新增加的代码：

```
// use smaller number as lower bound for summation
// and larger number as upper bound
int lower, upper;
if (v1 <= v2) {
    lower = v1;
    upper = v2;
} else {
    lower = v2;
    upper = v1;
}
```

这段代码的效果是恰当地设置 upper 和 lower。if 的条件测试 v1 是否小于或等于 v2。如果是，则执行条件后面紧接着的语句块。这个语句块包含两条语句，每条语句都完成一次赋值，第一条语句将 v1 赋值给 lower，而第二条语句将 v2 赋值给 upper。

如果这个条件为假（也就是说，如果 v1 大于 v2）那么执行 else 后面的语句。这个语句同样是一个由两个赋值语句组成的块，把 v2 赋值给 lower 而把 v1 赋值给 upper。

习题

习题 1.14 如果输入值相等，本节展示的程序将产生什么问题？

习题 1.15 用两个相等的值作为输入编译并运行本节中的程序。将实际输出与你在上一习题中所做的预测相比较，解释实际结果和你预计的结果间的不相符之处。

习题 1.16 编写程序，输出用户输入的两个数中的较大者。

习题 1.17 编写程序，要求用户输入一组数。输出信息说明其中有多少个负数。

1.4.4 读入未知数目的输入

对 1.4.1 节的求和程序稍作改变，还可以允许用户指定一组数求和。这种情况下，我们不知道要对多少个数求和，而是要一直读数直到程序输入结束。输入结束时，程序将总和写到标准输出：

```
#include <iostream>
int main()
{
    int sum = 0, value;
    // read till end-of-file, calculating a running total of all values read
    while (std::cin >> value)
        sum += value; // equivalent to sum = sum + value
    std::cout << "Sum is: " << sum << std::endl;
    return 0;
}
```

如果我们给出本程序的输入：

```
3 4 5 6
```

那么输出是：

```
sum is: 18
```

与平常一样，程序首先包含必要的头文件。main 中第一行定义了两个 int 变量，命名为 sum 和 value。在 while 条件中，用 value 保存读入的每一个数：

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

这里所产生的是，为判断条件，先执行输入操作

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

它具有从标准输入读取下一个数并且将读入的值保存在 value 中的效果。输入操作符 (1.2.2 节) 返回其左操作数。while 条件测试输入操作符的返回结果，意味着测试 std::cin。

当我们使用 istream 对象作为条件，结果是测试流的状态。如果流是有效的（也就是说，如果读入下一个输入是可能的）那么测试成功。遇到文件结束符（end-of-file）或遇到无效输入时，如读取了一个不是整数的值，则 istream 对象是无效的。处于无效状态的 istream 对象将导致条件失败。

19

在遇到文件结束符（或一些其他输入错误）之前，测试会成功并且执行 while 循环体。循环体是一条使用复合赋值操作符的语句，这个操作符将它的右操作数加到左操作数上。

从键盘输入文件结束符

操作系统使用不同的值作为文件结束符。Windows 系统下我们通过键入 control-z——同时键入“ctrl”键和“z”键，来输入文件结束符。Unix 系统中，包括 Mac OS-X 机器，通常用 control-d。

一旦测试失败，while 终止并退出循环体，执行 while 之后的语句。该语句在输出 sum 后输出 endl，endl 输出换行并刷新与 cout 相关联的缓冲区。最后，执行 return，通常返回零表示程序成功运行完毕。

习题

习题 1.18 编写程序，提示用户输入两个数并将这两个数范围内的每个数写到标准输出。

习题 1.19 如果上题给定数 1000 和 2000，程序将产生什么结果？修改程序，使每一行输出不超过 10 个数。

习题 1.20 编写程序，求用户指定范围内的数的和，省略设置上界和下界的 if 测试。假定输入数是 7 和 3，按照这个顺序，预测程序运行结果。然后按照给定的数是 7 和 3 运行程序，看结果是否与你预测的相符。如果不相符，反复研究关于 for 和 while 循环的讨论直到弄清楚其中的原因。

1.5 类的简介

解决书店问题之前，还需要弄明白如何编写数据结构来表示交易数据。C++中我们通过定义类来定义自己的数据结构。类机制是 C++ 中最重要的特征之一。事实上，C++ 设计的主要焦点就是使所定义的类型（class type）的行为可以像内置类型一样自然。我们前面已看到的像 `istream` 和 `ostream` 这样的库类型，都是定义为类的，也就是说，它们严格说来不是语言的一部分。

完全理解类机制需要掌握很多内容。所幸我们可以使用他人写的类而无需掌握如何定义自己的类。在这一节，我们将描述一个用于解决书店问题的简单类。当我们学习了更多关于类型、表达式、语句和函数的知识（所有这些在类定义中都将用到）后，将会在后面的章节实现这个类。

20

使用类时我们需要回答三个问题：

- (1) 类的名字是什么？
- (2) 它在哪里定义？
- (3) 它支持什么操作？

对于书店问题，我们假定类命名为 `Sales_item` 且类定义在命名为 `Sales_item.h` 的头文件中。

1.5.1 `Sales_item` 类

`Sales_item` 类的目的是存储 ISBN 并保存该书的销售册数、销售收入和平均售价。我们不关心如何存储或计算这些数据。使用类时我们不需要知道这个类是怎样实现的，相反，我们需要知道的是该类提供什么操作。

正如我们所看到的，使用像 IO 一样的库工具，必须包含相关的头文件。类似地，对于自定义的类，必须使得编译器可以访问和类相关的定义。这几乎可以采用同样的方式。一般来说，我们将类定义放入一个文件中，要使用该类的任何程序都必须包含这个文件。

依据惯例，类类型存储在一个文件中，其文件名如同程序的源文件名一样，由文件名和文件后缀两部分组成。通常文件名和定义在头文件中的类名是一样的。通常后缀是 `.h`，但也有一些程序员用 `.H`、`.hpp` 或 `.hxx`。编译器通常并不挑剔头文件名，但 IDE 有时会。假设我们的类定义在名为 `Sale_item.h` 的文件中。

1. `Sales_item` 对象上的操作

每个类定义一种类型，类型名与类名相同。因此，我们的 `Sales_item` 类定义了一种命名为 `Sales_item` 的类型。像使用内置类型一样，可以定义类类型的变量。当写下

```
Sales_item item;
```

就表示 `item` 是类型 `Sales_item` 的一个对象。通常将“类型 `Sales_item` 的一个对象”简称为“一个 `Sales_item` 对象”，或者更简单地简称为“一个 `Sales_item`”。

除了可以定义 `Sales_item` 类型的变量，我们还可以执行 `Sales_item` 对象的以下操作：

- 使用加法操作符，`+`，将两个 `Sales_item` 相加。
- 使用输入操作符，`>>`，来读取一个 `Sales_item` 对象。
- 使用输出操作符，`<<`，来输出一个 `Sales_item` 对象。

21

- 使用赋值操作符, =, 将一个 Sales_item 对象赋值给另一个 Sales_item 对象。
- 调用 same_isbn 函数确定两个 Sales_item 是否指同一本书。

2. 读入和写出 Sales_item 对象

知道了类提供的操作, 就可以编写一些简单的程序使用这个类。例如, 下面的程序从标准输入读取数据, 使用该数据建立一个 Sales_item 对象, 并将该 Sales_item 对象写到标准输出:

```
#include <iostream>
#include "Sales_item.h"
int main()
{
    Sales_item book;
    // read ISBN, number of copies sold, and sales price
    std::cin >> book;
    // write ISBN, number of copies sold, total revenue, and average price
    std::cout << book << std::endl;
    return 0;
}
```

如果输入到程序的是

0-201-70353-X 4 24.99

则输出将是

0-201-70353-X 4 99.96 24.99

输入表明销售了 4 本书, 每本价格是 24.99 美元。输出表明卖出书的总数是 4 本, 总收入是 99.96 美元, 每本书的平均价格是 24.99 美元。

这个程序以两个 #include 指示开始, 其中之一使用了一种新格式。iostream 头文件由标准库定义, 而 Sales_item 头文件则不是。Sales_item 是一种自定义类型。当使用自定义头文件时, 我们采用双引号 (" ") 把头文件名括起来。



标准库的头文件用尖括号 < > 括起来, 非标准库的头文件用双引号 " " 括起来。

在 main 函数中, 首先定义一个对象, 命名为 book, 用它保存从标准输入读取的数据。下一条语句读入数据到此对象, 第三条语句将它打印到标准输出, 像平常一样紧接着打印 endl 来刷新缓冲区。

22

关键概念: 类定义行为

在编写使用 Sales_item 的程序时, 重要的是记住类 Sales_item 的创建者定义该类对象可以执行的所有操作。也就是说, Sales_item 数据结构的创建者定义创建 Sales_item 对象时会发生什么, 以及加操作符或输入输出操作符应用到 Sales_item 对象时又会发生什么, 等等。

通常, 只有由类定义的操作可被用于该类类型的对象。此时, 我们知道的可以在 Sales_item 对象上执行的操作只是前面列出的那些。

我们将在 7.7.3 节和 14.2 节看到如何定义这些操作。

3. 将 Sales_item 对象相加

更有趣的例子是将两个 Sales_item 对象相加：

```
#include <iostream>
#include "Sales_item.h"
int main()
{
    Sales_item item1, item2;
    std::cin >> item1 >> item2;    // read a pair of transactions
    std::cout << item1 + item2 << std::endl; // print their sum
    return 0;
}
```

如果我们给这个程序下面的输入：

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

则输出为

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

程序首先包含两个头文件 Sales_item 和 iostream。接下来定义两个 Sales_item 对象来存放要求和的两笔交易。输出表达式做加法运算并输出结果。从前面列出的操作，可以得知将两个 Sales_item 相加将创建一个新对象，新对象的 ISBN 是其操作数的 ISBN，销售的数量和收入反映其操作数中相应值的和。我们也知道相加的项必须具有同样的 ISBN。

值得注意的是这个程序是如何类似于 1.2.2 节中的程序：读入两个输入并输出它们的和。令人感兴趣的是，本例并不是读入两个整数并输出两个整数的和，而是读入两个 Sales_item 对象并输出两个 Sales_item 对象的和。此外，“和”的意义也不同。在整数的实例中我们产生的是传统求和——两个数值相加后的结果。在 Sales_item 对象的实例上我们使用了在概念上有新意义的求和——两个 Sales_item 对象的成分相加后的结果。

23

习题

- 习题 1.21 本书配套网站 (http://www.awprofessional.com/cpp_primer) 的第 1 章的代码目录下有 Sales_item.h 源文件。复制该文件到你的工作目录。编写程序，循环遍历一组书的销售交易，读入每笔交易并将交易写至标准输出。
- 习题 1.22 编写程序，读入两个具有相同 ISBN 的 Sales_item 对象并产生它们的和。
- 习题 1.23 编写程序，读入几个具有相同 ISBN 的交易，输出所有读入交易的和。

1.5.2 初窥成员函数

不幸的是，将 Sales_item 相加的程序有一个问题。如果输入指向了两个不同的 ISBN 将发生什么？将两个不同 ISBN 的数据相加没有意义。为解决这个问题，首先检查 Sales_item 操作数是否都具有相同的 ISBN。

```
#include <iostream>
```

```

#include "Sales_item.h"
int main()
{
    Sales_item item1, item2;
    std::cin >> item1 >> item2;
    // first check that item1 and item2 represent the same book
    if (item1.same_isbn(item2)) {
        std::cout << item1 + item2 << std::endl;
        return 0;    // indicate success
    } else {
        std::cerr << "Data must refer to same ISBN"
                  << std::endl;
        return -1;    // indicate failure
    }
}

```

这个程序和前一个程序不同之处在于 `if` 测试语句以及与它相关联的 `else` 分支。在解释 `if` 语句的条件之前，我们明白程序的行为取决于 `if` 语句中的条件。如果测试成功，那么产生与前一程序相同的输出，并返回 0 表示程序成功运行完毕。如果测试失败，执行 `else` 后面的语句块，输出信息并返回错误提示。

什么是成员函数

上述 `if` 语句的条件

```

// first check that item1 and item2 represent the same book
if (item1.same_isbn(item2)) {

```

24

调用命名为 `item1` 的 `Sales_item` 对象的**成员函数**（member function）。成员函数是由类定义的函数，有时称为**类方法**（method）。

成员函数只定义一次，但被视为每个对象的成员。我们将这些操作称为成员函数，是因为它们（通常）在特定对象上操作。在这个意义上，它们是对象的成员，即使同一类型的所有对象共享同一个定义也是如此。

当调用成员函数时，（通常）指定函数要操作的对象。语法是使用**点操作符**（.）：

```
item1.same_isbn
```

意思是“命名为 `item1` 的对象的 `same_isbn` 成员”。点操作符通过它的左操作数取得右操作数。点操作符仅应用于类类型的对象：左操作数必须是类类型的对象，右操作数必须指定该类型的成员。



与大多数其他操作符不同，点操作符（“.”）的右操作数不是对象或值，而是成员的名字。

通常使用成员函数作为点操作符的右操作数来调用成员函数。执行成员函数和执行其他函数相似：要调用函数，可将**调用操作符**（`()`）放在函数名之后。调用操作符是一对圆括号，括住传递给函数的实参列表（可能为空）。

`same_isbn` 函数接受单个参数，且该参数是另一个 `Sales_item` 对象。函数调用

```
item1.same_isbn(item2)
```

将 item2 作为参数传递给名为 same_isbn 的函数，该函数是名为 item1 的对象的成员。它将比较参数 item2 的 ISBN 与函数 same_isbn 要操作的对象 item1 的 ISBN。效果是测试两个对象是否具有相同的 ISBN。

如果对象具有相同的 ISBN，执行 if 后面的语句，输出两个 Sales_item 对象的和；否则，如果对象具有不同的 ISBN，则执行 else 分支的语句块。该块输出适当的错误信息并退出程序，返回-1。回想 main 函数的返回值被视为状态指示器；本例中，返回一个非零值表示程序未能产生期望的结果。

习题

习题 1.24 编写程序，读入几笔不同的交易。对于每笔新读入的交易，要确定它的 ISBN 是否和以前的交易的 ISBN 一样，并且记下每一个 ISBN 的交易的总数。通过给定多笔不同的交易来测试程序。这些交易必须代表多个不同的 ISBN，但是每个 ISBN 的记录应分在同一组。

1.6 C++程序

现在我们已经做好准备，可以着手解决最初的书店问题了：我们需要读入销售交易文件，并产生报告显示每本书的总销售收入、平均销售价格和销售册数。

25

假定给定 ISBN 的所有交易出现在一起。程序将把每个 ISBN 的数据组合至命名为 total 的 Sales_item 对象中。从标准输入中读取的每一笔交易将被存储到命名为 trans 的第二个 Sales_item 对象中。每读取一笔新的交易，就将它与 total 中的 Sales_item 对象相比较，如果对象含有相同的 ISBN，就更新 total；否则就输出 total 的值，并使用刚读入的交易重置 total。

```
#include <iostream>
#include "Sales_item.h"
int main()
{
    // declare variables to hold running sum and data for the next record
    Sales_item total, trans;
    // is there data to process?
    if (std::cin >> total) {
        // if so, read the transaction records
        while (std::cin >> trans)
            if (total.same_isbn(trans))
                // match: update the running total
                total = total + trans;
            else {
                // no match: print & assign to total
                std::cout << total << std::endl;
                total = trans;
            }
        // remember to print last record
        std::cout << total << std::endl;
    } else {
        // no input!, warn the user
```

```

        std::cout << "No data?!" << std::endl;
        return -1;    // indicate failure
    }
    return 0;
}

```

这个程序是到目前我们见到的程序中最为复杂的一个，但它仅使用了我们已遇到过的工具。和平常一样，我们从包含所使用的头文件开始：标准库中的 `iostream` 和自定义的头文件 `Sales_item.h`。

26

在 `main` 中我们定义了所需要的对象 `total` 用来计算给定的 ISBN 的交易的总数，`trans` 用来存储读取的交易。我们首先将交易读入 `total` 并测试是否读取成功；如果读取失败，表示没有记录，程序进入最外层的 `else` 分支，输出信息警告用户没有输入。

假如我们成功读取了一个记录，则执行 `if` 分支里的代码。首先执行 `while` 语句，循环遍历剩余的所有记录。就像 1.4.3 节的程序一样，`while` 循环的条件从标准输入中读取值并测试实际读取的是否是合法数据。本例中，我们将一个 `Sales_item` 对象读至 `trans`。只要读取成功，就执行 `while` 循环体。

`while` 循环体只是一条 `if` 语句。我们测试 ISBN 是否相等。如果相等，我们将这两个对象相加并将结果存储到 `total` 中。否则，我们就输出存储在 `total` 中的值，并将 `trans` 赋值给 `total` 来重置 `total`。执行完 `if` 语句之后，将返回到 `while` 语句中的条件，读入下一个交易，直到执行完所有记录。

一旦 `while` 完成，我们仍须写出与最后一个 ISBN 相关联的数据。当 `while` 语句结束时，`total` 包含文件中最后一条 ISBN 数据，但是我们没有机会输出这条数据。我们在结束最外层 `if` 语句的语句块的最后一条语句中进行输出。

习题

习题 1.25 使用源自本书配套网站的 `Sales_item.h` 头文件，编译并执行本节给出的书店程序。

习题 1.26 在书店程序中，我们使用了加法操作符而不是复合赋值操作符将 `trans` 加到 `total` 中，为什么我们不使用复合赋值操作符？

27

小结

本章介绍了足够多的 C++ 知识，让读者能够编译和执行简单 C++ 程序。我们看到了如何定义 `main` 函数，这是任何 C++ 程序首先执行的函数。我们也看到了如何定义变量，如何进行输入和输出，以及如何编写 `if`、`for` 和 `while` 语句。本章最后介绍 C++ 最基本的工具：类。在这一章中，我们看到了如何创建和使用给定类的对象。后面的章节中将介绍如何自定义类。

术语

argument (实参) 传递给被调用函数的值。 **block (块)** 花括号括起来的语句序列。

buffer (缓冲区) 一段用来存放数据的存储区域。IO 设备常存储输入（或输出）到缓冲区，并独立于程序动作对缓冲区进行读写。输出缓冲区通常必须显式刷新以强制输出缓冲区内内容。默认情况下，读 `cin` 会刷新 `cout`；当程序正常结束时，`cout` 也被刷新。

built-in type (内置类型) C++语言本身定义的类型，如 `int`。

cerr 绑定到标准错误的 `ostream` 对象，这通常是与标准输出相同的流。默认情况下，输出 `cerr` 不缓冲，通常用于不是程序正常逻辑部分的错误信息或其他输出。

cin 用于从标准输入中读入的 `istream` 对象。

class (类) 用于自定义数据结构的 C++ 机制。类是 C++ 中最基本的特征。标准库类型，如 `istream` 和 `ostream`，都是类。

class type (类类型) 由类所定义的类型，类型名就是类名。

clog 绑定到标准错误的 `ostream` 对象。默认情况下，写到 `clog` 时是带缓冲的。通常用于将程序执行信息写入到日志文件中。

comment (注释) 编译器会忽略的程序文本。C++ 有单行注释和成对注释两种类型的注释。单行注释以 `//` 开头，从 `//` 到行的结尾是一条注释。成对注释以 `/*` 开始包括到下一个 `*/` 为止的所有文本。

condition (条件) 求值为真或假的表达式。值为 0 的算术表达式是假，其他所有非 0 值都是真。

cout 用于写入到标准输出的 `ostream` 对象，一般情况下用于程序的输出。

curly brace (花括号) 花括号对语句块定界。左花括号 “{” 开始一个块，右花括号 “}” 结束块。

data structure (数据结构) 数据及数据上操作的逻辑组合。

edit-compile-debug (编辑-编译-调试) 使得程序正确执行的过程。

end-of-file (文件结束符) 文件中与特定系统有关的标记，表示这个文件中不再有其他输入。

expression (表达式) 最小的计算单元。表达式包含一个或多个操作数并经常含有一个操作符。表达式被求值并产生一个结果。例如，假定 `i` 和 `j` 都为 `int` 型，则 `i + j` 是一个算术加法表达式并求这两个 `int` 值的和。表达式将在第 5 章详细介绍。

for statement (for 语句) 提供迭代执行的控制语句，通常用于步进遍历数据结构或对一个计算重复固定次数。

function (函数) 有名字的计算单元。

function body (函数体) 定义函数所执行的动作的语句块。

function name (函数名) 函数的名字标识，函数通过函数名调用。

header (头文件) 使得类或其他名字的定义在多个程序中可用的一种机制。程序中通过 `#include` 指示包含头文件。

if statement (if 语句) 根据指定条件的值执行的语句。如果条件为真，则执行 `if` 语句体；否则控制流执行 `else` 后面的语句，如果没有 `else` 将执行 `if` 后面的语句。

iostream (输入输出流) 提供面向流的输入和输出的标准库类型。

istream (输入流) 提供面向流的输入的标准库类型。

library type (标准库类型) 标准库所定义的

类型，如 `istream`。

main function (主函数) 执行 C++ 程序时，操作系统调用的函数。每一个程序有且仅有一个主函数 `main`。

manipulator (操纵符) 在读或写时“操纵”流本身的对象，如 `std::endl`。A.3.1 节详细讲述操纵符。

member function (成员函数) 类定义的操作。成员函数通常在特定的对象上进行操作。

method (方法) 成员函数的同义词。

namespace (命名空间) 将库所定义的名字放至单独一个地方的机制。命名空间有助于避免无意的命名冲突。C++ 标准库所定义的名字在命名空间 `std` 中。

ostream (输出流) 提供面向流的输出的库类型。

parameter list (形参表) 函数定义的组成部分。指明可以用什么参数来调用函数，可能为空。

preprocessor directive (预处理指示) C++ 预处理器的指示。`#include` 是一个预处理器指示。预处理器指示必须出现在单独的行中。2.9.2 节将对预处理器作详细的介绍。

return type (返回类型) 函数返回值的类型。

source file (源文件) 用来描述包含在 C++ 程序中的文件的术语。

standard error (标准错误) 用于错误报告的输出流。通常，在视窗操作系统中，将标准输出和标准错误绑定到程序的执行窗口。

standard input (标准输入) 和程序执行窗口相关联的输入流，通常这种关联由操作系统设定。

standard library (标准库) 每个 C++ 编译器

必须支持的类型和函数的集合。标准库提供了强大的功能，包括支持 IO 的类型。C++ 程序员谈到的“标准库”，是指整个标准库，当提到某个标准库类型时也指标准库中某个特定的部分。例如，程序员提到的“`iostream` 库”，专指标准库中由 `iostream` 类定义的那部分。

standard output (标准输出) 和程序执行窗口相关联的输出流，通常这种关联由操作系统设定。

statement (语句) C++ 程序中最小的独立单元，类似于自然语言中的句子。C++ 中的语句一般以分号结束。

std 标准库命名空间的名字，`std::cout` 表明正在使用定义在 `std` 命名空间中的名字 `cout`。

string literal (字符串面值) 以双引号括起来的字符序列。

uninitialized variable (未初始化变量) 没有指定初始值的变量。类类型没有未初始化变量。没有指定初始值的类类型变量由类定义初始化。在使用变量值之前必须给未初始化的变量赋值。未初始化变量是造成 *bug* 的主要原因之一。

variable (变量) 有名字的对象。

while statement (while 语句) 一种迭代控制语句，只要指定的条件为真就执行 `while` 循环体。`while` 循环体执行 0 次还是多次，依赖于条件的真值。

() operator [() 操作符] 调用操作符。跟在函数名后且成对出现的圆括号。该操作符导致函数被调用，给函数的实参可在括号里传递。

++ operator (++ 操作符) 自增操作符。将操作数加 1，`++i` 等价于 `i=i+1`。

+= operator (+= 操作符) 复合赋值操作符，

将右操作数和左操作数相加，并将结果存储到左操作数中；`a+=b` 等价于 `a=a+b`。

`.operator (.`操作符) 点操作符。接受两个操作数：左操作数是一个对象，而右边是该对象的一个成员的名字。这个操作符从指定对象中取得成员。

`:: operator (::`操作符) 作用域操作符。在第2章中，我们将看到更多关于作用域的介绍。在其他的使用过程中，`::`操作符用于在命名空间中访问名字。例如，`std::cout` 表示使用命名空间 `std` 中的名字 `cout`。

`= operator (=`操作符) 表示把右操作数的值赋给左操作数表示的对象。

`<< operator (<<`操作符) 输出操作符。把右操作数写到左操作数指定的输出流：`cout<<"hi"`把 `hi` 写入到标准输出流。输出操作可以链接在一起使用：`cout << "hi" << "bye"` 输出 `hibye`。

`>> operator (>>`操作符) 输入操作符。从左操作数指定的输入流读入数据到右操作数：

`cin >> i` 把标准输入流中的下一个值读入到 `i` 中。输入操作能够链接在一起使用：`cin >> i >> j` 先读入 `i` 然后再读入 `j`。

`== operator (==`操作符) 等于操作符，测试左右两边的操作数是否相等。

`!= operator (!=`操作符) 不等于操作符。测试左右两边的操作数是否不等。

`<= operator (<=`操作符) 小于或等于操作符。测试左操作数是否小于或等于右操作数。

`< operator (<`操作符) 小于操作符。测试左操作数是否小于右操作数。

`>= operator (>=`操作符) 大于或等于操作符。测试左操作数是否大于或等于右操作数。

`> operator (>`操作符) 大于操作符。测试左操作数是否大于右操作数。

基本语言

目录

第 2 章 变量和基本类型..... 29

第 3 章 标准库类型..... 67

第 4 章 数组和指针..... 95

第 5 章 表达式.....127

第 6 章 语句.....165

第 7 章 函数.....195

第 8 章 标准 IO 库 243

各种程序设计语言都具有许多独具特色的特征，这些特征决定了用每种语言适合开发哪些类型的应用程序。程序设计语言也有一些共同的特征。基本上所有的语言都要提供下列特征：

- 内置数据类型，如整型、字符型等。
- 表达式和语句：表达式和语句用于操纵上述类型的值。
- 变量：程序员可以使用变量对所用的对象命名。
- 控制结构：如 if 或 while，程序员可以使用控制结构有条件地执行或重复执行一组动作。
- 函数：程序员可以使用函数把行为抽象成可调用的计算单元。

大多数现代程序设计语言都采用两种方式扩充上述基本特征集：允许程序员通过自

定义数据类型扩展该语言；提供一组库例程，这些例程定义了一些并非内置在语言中的实用函数和数据类型。

[31]

和大多数程序设计语言一样，C++中对象的类型决定了该对象可以执行的操作。语句正确与否取决于该语句中对象的类型。一些程序设计语言，特别是 Smalltalk 和 Python，在运行时才检查语句中对象的类型。相反，C++是静态类型（statically typed）语言，在编译时执行类型检查。结果是程序中使用某个名字之前，必须先告知编译器该名字的类型。

C++提供了一组内置数据类型、操纵这些类型的操作符和一组少量的程序流控制语句。这些元素形成了一个“词汇表”，使用这个词汇表可以而且已经编写出许多大型、复杂的实际系统。从这个基本层面来看，C++是一门简单的语言。C++的表达能力是通过支持一些允许程序员定义新数据结构的机制来提升的。

可能 C++中最重要的特征是类（class），程序员可以使用类自定义数据类型。C++中这些类型有时也称为“类类型（class type）”，以区别于语言的内置类型。有一些语言允许程序员定义的数据类型只能指定组成该类型的数据。包括 C++在内的其他语言允许程序员定义的类型不仅有数据还包括操作。C++主要设计目标之一就是允许程序员自定义类型，而且这些类型和内置类型一样易于使用。C++标准库利用这些特征，实现了一个具有丰富类类型和相关函数的标准库。

掌握 C++的第一步是学习语言的基本知识和标准库，这正是第一部分介绍的内容。第 2 章介绍了内置数据类型，并简单探讨了自定义新类型的机制。第 3 章引入了两种最基本的标准库类型：string 和 vector。第 4 章介绍了数组，数组是一种低级的数据结构，内置于 C++和许多其他语言。数组类似于 vector 对象，但较难使用。第 5 章到第 7 章介绍了表达式、语句和函数。第 8 章是第一部分的最后一章，介绍了 IO 标准库中最重要的设施。

[32]

第 2 章

变量和基本类型

目录

2.1 基本内置类型.....	30
2.2 字面值常量.....	34
2.3 变量.....	38
2.4 const 限定符.....	49
2.5 引用.....	50
2.6 typedef 名字.....	53
2.7 枚举.....	53
2.8 类类型.....	54
2.9 编写自己的头文件.....	57
小结.....	62
术语.....	62

类型是所有程序的基础。类型告诉我们数据代表什么意思以及可以对数据执行哪些操作。

C++语言定义了几种基本类型：字符型、整型、浮点型等。C++还提供了可用于自定义数据类型的机制，标准库正是利用这些机制定义了许多更复杂的类型，比如可变长字符串 `string`、`vector` 等。此外，我们还能修改已有的类型以形成复合类型。本章介绍内置类型，并开始介绍 C++如何支持更复杂的类型。

类型确定了数据和操作在程序中的意义。我们在第 1 章已经看到，如下的语句

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

有不同的含义，具体含义取决于 i 和 j 的类型。如果 i 和 j 都是整型，则这条语句表示一般的算术“+”运算；如果 i 和 j 都是 Sales_item 对象，则这条语句是将这两个对象的组成成分分别加起来。

C++中对类型的支持是非常广泛的：语言本身定义了一组基本类型和修改已有类型的方法，还提供了一组特征用于自定义类型。本章通过介绍内置类型和如何关联类型与对象来探讨 C++ 中的类型。本章还将介绍更改类型和建立自定义类型的方法。

2.1 基本内置类型

C++定义了一组表示整数、浮点数、单个字符和布尔值的算术类型 (arithmetic type)，另外还定义了一种称为 void 的特殊类型。void 类型没有对应的值，仅用在有限的一些情况下，通常用作无返回值函数的返回类型。

算术类型的存储空间依机器而定。这里的存储空间是指用来表示该类型的二进制位 (bit) 数。C++标准规定了每个算术类型的最小存储空间，但它并不阻止编译器使用更大的存储空间。事实上，对于 int 类型，几乎所有的编译器使用的存储空间都比所要求的大。表 2-1 列出了内置算术类型及其对应的最小存储空间。



因为位数的不同，这些类型所能表示的最大（最小）值也因机器的不同而有所不同。

表 2-1 C++: 算术类型		
类 型	含 义	最小存储空间
bool	布尔型	—
char	字符型	8 位
wchar_t	宽字符型	16 位
short	短整型	16 位
int	整型	16 位
long	长整型	32 位
float	单精度浮点型	6 位有效数字
double	双精度浮点型	10 位有效数字
long double	扩展精度浮点型	10 位有效数字

2.1.1 整型

表示整数、字符和布尔值的算术类型合称为整型 (integral type)。

字符类型有两种：char 和 wchar_t。char 类型保证了有足够的空间，能够存储机器基本字符集中任何字符相应的数值，因此，char 类型通常是单个机器字节 (byte)。wchar_t 类型用于扩展字符集，比如汉字和日语，这些字符集中的一些字符不能用单个 char 表示。

内置类型的机器级表示

C++的内置类型与其在计算机的存储器中的表示方式紧密相关。计算机以位序列存储数据，每一位存储0或1。一段内存可能存储着

```
00011011011100010110010000111011...
```

在位这一级上，存储器是没有结构和意义的。

让存储具有结构的最基本方法是用块（chunk）处理存储。大部分计算机都使用特定位数的块来处理存储，块的位数一般是2的幂，因为这样可以一次处理8、16或32位。64和128位的块如今也变得更为普遍。虽然确切的大小因机器不同而不同，但是通常将8位的块作为一个字节，32位或4个字节作为一个“字（word）”。

大多数计算机将存储器中的每一个字节和一个称为地址的数关联起来。对于一个8位字节和32位字的机器，我们可以将存储器的字表示如下：

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

在这个图中，左边是字节的地址，地址后面为字节的8位。

可以用地址表示从该地址开始的任何几个不同大小的位集合。可以说地址为736424的字，也可以说地址为736426的字节。例如，可以说地址为736425的字节和地址为736427的字节不相等。

要让地址为736425的字节具有意义，必须要知道存储在该地址的值的类型。一旦知道了该地址的值的类型，就知道了表示该类型的值需要多少位和如何解释这些位。

如果知道地址为736425的字节的类型是8位无符号整数，那么就可以知道该字节表示整数112。另外，如果这个字节是ISO-Latin-1字符集中的一个字符，那它就表示小写字母q。虽然两种情况的位相同，但归属于不同类型，解释也就不同。

short、int和long类型都表示整型值，存储空间的大小不同。一般，short类型为半个机器字（word）长，int类型为一个机器字长，而long类型为一个或两个机器字长（在32位机器中int类型和long类型通常字长是相同的）。

bool类型表示真值true和false。可以将算术类型的任何值赋给bool对象。0值算术类型代表false，任何非0的值都代表true。

1. 带符号和无符号类型

除bool类型外，整型可以是带符号的（signed）也可以是无符号的（unsigned）。顾名思义，带符号类型可以表示正数也可以表示负数（包括0），而无符号型只能表示大于或等于0的数。

整型int、short和long都默认为带符号型。要获得无符号型则必须指定该类型为unsigned，比如unsigned long。unsigned int类型可以简写为unsigned，也就是说，unsigned后不加其他类型说明符意味着是unsigned int。

和其他整型不同, char 有三种不同的类型: 普通 char、unsigned char 和 signed char。虽然 char 有三种不同的类型, 但只有两种表示方式。可以使用 unsigned char 或 signed char 表示 char 类型。使用哪种 char 表示方式由编译器而定。

2. 整型值的表示

无符号型中, 所有的位都表示数值。如果在某种机器中, 定义一种类型使用 8 位表示, 那么这种类型的 unsigned 型可以取值 0 到 255。

C++ 标准并未定义 signed 类型如何用位来表示, 而是由每个编译器自由决定如何表示 signed 类型。这些表示方式会影响 signed 类型的取值范围。8 位 signed 类型的取值肯定至少是从 -127 到 127, 但也有许多实现允许取值从 -128 到 127。

表示 signed 整型类型最常见的策略是用其中一个位作为符号位。符号位为 1, 值就为负数; 符号位为 0, 值就为 0 或正数。一个使用一位符号位的表示方式的 8 位 signed 整型取值是从 -128 到 127。

3. 整型的赋值

36

对象的类型决定对象的取值。这会引起一个疑问: 当我们试着把一个超出其取值范围的值赋给一个指定类型的对象时, 结果会怎样呢? 答案取决于这种类型是 signed 还是 unsigned 的。

对于 unsigned 类型来说, 编译器必须调整越界值使其满足要求。编译器会将该值对 unsigned 类型的可能取值数目求模, 然后取所得值。比如 8 位的 unsigned char, 其取值范围从 0 到 255 (包括 255)。如果赋给超出这个范围的值, 那么编译器将会取该值对 256 求模后的值。例如, 如果试图将 336 存储到 8 位的 unsigned char 中, 则实际赋值为 80, 因为 80 是 336 对 256 求模后的值。

对于 unsigned 类型来说, 负数总是超出其取值范围。unsigned 类型的对象可能永远不会保存负数。有些语言中将负数赋给 unsigned 类型是非法的, 但在 C++ 中这是合法的。



C++ 中, 把负值赋给 unsigned 对象是完全合法的, 其结果是该负数对该类型的取值个数求模后的值。所以, 如果把 -1 赋给 8 位的 unsigned char, 那么结果是 255, 因为 255 是 -1 对 256 求模后的值。

当将超过取值范围的值赋给 signed 类型时, 由编译器决定实际赋的值。在实际操作中, 很多的编译器处理 signed 类型的方式和 unsigned 类型类似。也就是说, 赋值时是取该值对该类型取值数目求模后的值。然而我们不能保证编译器都会这样处理 signed 类型。

2.1.2 浮点型

类型 float、double 和 long double 分别表示单精度浮点数、双精度浮点数和扩展精度浮点数。一般 float 类型用一个字 (32 位) 来表示, double 类型用两个字 (64 位) 来表示, long double 类型用三个或四个字 (96 或 128 位) 来表示。类型的取值范围决定了浮点数所含的有效数字位数。



对于实际的程序来说, float 类型精度通常是不够的——float 型只能保证 6 位有效数字, 而 double 型至少可以保证 10 位有效数字, 能满足大多数计算的需要。

建议：使用内置算术类型

C++中整型数有点令人迷惑不解。就像C语言一样，C++被设计成允许程序在必要时直接处理硬件，因此整型被定义成满足各种各样硬件的特性。大多数程序员可以（应该）通过限制实际使用的类型来忽略这些复杂性。

实际上，许多人用整型进行计数。例如：程序经常计算像vector或数组这种数据结构的元素个数。在第3章和第4章中，我们将看到标准库定义了一组类型用于统计对象的大小。因此，当计数这些元素时使用标准库定义的类型总是正确的。其他情况下，使用unsigned类型比较明智，可以避免值越界导致结果为负数的可能性。

当执行整型算术运算时，很少使用short类型。大多数程序中，使用short类型可能会隐含赋值越界的错误。这个错误会产生什么后果将取决于所使用的机器。比较典型的情况是值“截断（wrap around）”以至于因越界而变成很大的负数。同样的道理，虽然char类型是整型，但是char类型通常用来存储字符而不用于计算。事实上，在某些实现中char类型被当作signed类型，在另外一些实现中则被当作unsigned类型，因此把char类型作为计算类型使用时容易出问题。

在大多数机器上，使用int类型进行整型计算不易出错。就技术上而言，int类型用16位表示——这对大多数应用来说太小了。实际应用中，大多数通用机器都是使用与long类型一样长的32位来表示int类型。整型运算时，用32位表示int类型和用64位表示long类型的机器会出现应该选择int类型还是long类型的难题。在这些机器上，用long类型进行计算所付出的运行时代价远远高于用int类型进行同样计算的代价，所以选择类型前要先了解程序的细节并且比较long类型与int类型的实际运行时性能代价。

决定使用哪种浮点型就容易多了：使用double类型基本上不会有错。在float类型中隐式的精度损失是不能忽视的，而双精度计算的代价相对于单精度可以忽略。事实上，有些机器上，double类型比float类型的计算要快得多。long double类型提供的精度通常没有必要，而且还需要承担额外的运行代价。

习题

- 习题 2.1 int、long 和 short 类型之间有什么差别？
- 习题 2.2 unsigned 和 signed 类型有什么差别？
- 习题 2.3 如果在某机器上 short 类型占 16 位，那么可以赋给 short 类型的最大数是什么？unsigned short 类型的最大数又是什么？
- 习题 2.4 当给 16 位的 unsigned short 对象赋值 100000 时，赋的值是什么？
- 习题 2.5 float 类型和 double 类型有什么差别？
- 习题 2.6 要计算抵押贷款的偿还金额，利率、本金和付款额应分别选用哪种类型？解释你选择的理由。

2.2 字面值常量

像 42 这样的值，在程序中被当作**字面值常量**（literal constant）。称之为字面值是因为只能用它的值称呼它，称之为常量是因为它的值不能修改。每个字面值都有相应的类型，例如：0 是 int 型，3.14159 是 double 型。只有内置类型存在字面值，没有类类型的字面值。因此，没有任何标准库类型的字面值。

37

1. 整型字面值规则

定义字面值整数常量可以使用以下三种进制中的任一种：十进制、八进制和十六进制。当然这些进制不会改变其二进制位的表示形式。例如，我们能将值 20 定义成下列三种形式中的任意一种：

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

以 0（零）开头的字面值整数常量表示八进制，以 0x 或 0X 开头的表示十六进制。

38

字面值整数常量的类型默认为 int 或 long 类型。其精度类型决定于字面值——其值适合 int 就是 int 类型，比 int 大的值就是 long 类型。通过增加后缀，能够强制将字面值整数常量转换为 long、unsigned 或 unsigned long 类型。通过在数值后面加 L 或者 l（字母“l”大写或小写）指定常量为 long 类型。



定义长整型时，应该使用大写字母 L。小写字母 l 很容易和数值 1 混淆。

类似地，可通过在数值后面加 U 或 u 定义 unsigned 类型。同时加 L 和 U 就能够得到 unsigned long 类型的字面值常量。但其后缀不能有空格：

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

没有 short 类型的字面值常量。

2. 浮点字面值规则

通常可以用十进制或者科学计数法来表示浮点字面值常量。使用科学计数法时，指数用 E 或者 e 表示。默认的浮点字面值常量为 double 类型。在数值的后面加上 F 或 f 表示单精度。同样加上 L 或者 l 表示扩展精度（再次提醒，不提倡使用小写字母 l）。下面每一组字面值表示相同的值：

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

3. 布尔字面值和字符字面值

单词 true 和 false 是布尔型的字面值：

39

```
bool test = false;
```

可打印的字符型字面值通常用一对单引号来定义：

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

这些字面值都是 char 类型的。在字符面值前加 L 就能够得到 wchar_t 类型的宽字符面值。如：

```
L'a'
```

4. 非打印字符的转义序列

有些字符是不可打印的。不可打印字符实际上是不可显示的字符，比如退格或者控制符。还有一些在语言中有特殊意义的字符，例如单引号、双引号和反斜线符号。不可打印字符和特殊字符都用转义字符书写。转义字符都以反斜线符号开始，C++语言中定义了如下转义字符：

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

我们可以将任何字符表示为以下形式的通用转义字符：

```
\ooo
```

这里 ooo 表示三个八进制数字，这三个数字表示字符的数字值。下面的例子是用 ASCII 码字符集表示字面值常量：

```
\7   (响铃符)      \12  (换行符)      \40 (空格符)
\0   (空字符)      \062 ('2')      \115 ('M')
```

字符 '\0' 通常表示“空字符（null character）”，我们将会看到它有着非常特殊的意义。

同样也可以用十六进制转义字符来定义字符：

```
\xdd
```

它由一个反斜线符、一个 x 和一个或者多个十六进制数字组成。

5. 字符串字面值

之前见过的所有字面值都有基本内置类型。还有一种字面值（字符串字面值）更加复杂。字符串字面值是一串常量字符，这种类型将在 4.3 节详细说明。

字符串字面值常量用双引号括起来的零个或者多个字符表示。不可打印字符表示成相应的转义字符。

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

为了兼容 C 语言，C++中所有的字符串字面值都由编译器自动在末尾添加一个空字符。字符串字面值

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

表示单个字符 A，然而

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

表示包含字母 A 和空字符两个字符的字符串。

正如存在宽字符字面值，如

```
L'a'
```

也存在宽字符串字面值，一样在前面加“L”，如

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

宽字符串字面值是一串常量宽字符，同样以一个宽空字符结束。

6. 字符串字面值的连接

两个相邻的仅由空格、制表符或换行符分开的字符串字面值（或宽字符串字面值），可连接成一个新的字符串字面值。这使得多行书写长字符串字面值变得简单：

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

执行这条语句将会输出：

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

如果连接字符串字面值和宽字符串字面值，将会出现什么结果呢？例如：

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

其结果是未定义的（undefined），也就是说，连接不同类型的行为标准没有定义。这个程序可能会执行，也可能会崩溃或者产生没有用的值，而且在不同的编译器下程序的动作可能不同。

41

7. 多行字面值

处理长字符串有一个更基本的（但不常使用）方法，这个方法依赖于很少使用的程序格式化特性：在一行的末尾加一反斜线符号可将此行和下一行当作同一行处理。

正如 1.4.1 节提到的，C++ 的格式非常自由。特别是有一些地方不能插入空格，其中之一是在单词中间。特别是不能在单词中间断开一行。但可以通过使用反斜线符号巧妙实现：

```
// ok: A \ before a newline ignores the line break
std::cout <<
    "Hi" << std::endl;
```

等价于

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

可以使用这个特性来编写长字符串字面值：

```
// multiline string literal
std::cout << "a multi-line \
```

```

string literal \
using a backslash"
        << std::endl;
    return 0;
}

```

注意反斜线符号必须是该行的尾字符——不允许其后面有注释或空格。同样，后继行行首的任何空格和制表符都是字符串字面值的一部分。正因如此，长字符串字面值的后继行才不会有正常的缩进。

建议：不要依赖未定义行为

使用了未定义行为的程序都是错误的，即使程序能够运行，也只是巧合。未定义行为源于编译器不能检测到的程序错误或太麻烦以至无法检测的错误。

不幸的是，含有未定义行为的程序在有些环境或编译器中可以正确执行，但不能保证同一程序在不同编译器中甚至在当前编译器的后继版本中会继续正确运行，也不能保证程序在一组输入上可以正确运行且在另一组输入上也能够正确运行。

程序不应该依赖未定义行为。

同样地，通常程序不应该依赖机器相关的行为，比如假定 `int` 的位数是个固定且已知的值。我们称这样的程序是不可移植的（nonportable）。当程序移植到另一台机器上时，要寻找并更改任何依赖机器相关操作的代码。在本来可以运行的程序中寻找这类问题是一项非常不愉快的任务。

42

习题

习题 2.7 解释下列字面值常量的不同之处。

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

习题 2.8 确定下列字面值常量的类型：

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

习题 2.9 下列哪些（如果有）是非法的？

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

习题 2.10 使用转义字符编写一段程序，输出 `2M`，然后换行。修改程序，输出 `2`，跟着一个制表符，然后是 `M`，最后是换行符。

2.3 变量

如果要计算 2 的 10 次方，我们首先想到的可能是：

```
#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;
}
```

这个程序确实解决了问题，尽管我们可能要一而再、再而三地检查确保恰好有 10 个字面值常量 2 相乘。这个程序产生正确的答案 1024。

接下来要计算 2 的 17 次方，然后是 23 次方。而每次都要改变程序是很麻烦的事。更糟的是，这样做还容易引起错误。修改后的程序常常会产生多乘或少乘 2 的结果。

替代这种蛮力型计算的方法包括两部分内容：

(1) 使用已命名对象执行运算并输出每次计算。

(2) 使用控制流结构，当某个条件为真时重复执行一系列程序语句。

以下是计算 2 的 10 次方的替代方法：

```
#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 和 cnt 都是变量，可以对数值进行存储、修改和查询。for 循环使得计算过程重复执行 pow 次。

习题

习题 2.11 编写程序，要求用户输入两个数——底数（base）和指数（exponent），输出底数的指数次方的结果。

关键概念：强静态类型

C++是一门静态类型语言，在编译时会作类型检查。

在大多数语言中，对象的类型限制了对象可以执行的操作。如果某种类型不支持某种操作，那么这种类型的对象也就不能执行该操作。

在 C++ 中，操作是否合法是在编译时检查的。当编写表达式时，编译器检查表达式中的对象是否按该对象的类型定义的使用方式使用。如果不是的话，那么编译器会提示错误，而不产生可执行文件。

随着程序和使用的类型变得越来越复杂，我们将看到静态类型检查能帮助我们更早地发现错误。静态类型检查使得编译器必须能识别程序中的每个实体的类型。因此，程序中使用变量前必须先定义变量的类型。

44

2.3.1 什么是变量

变量提供了程序可以操作的有名字的存储区。C++中的每一个变量都有特定的类型，该类型决定了变量的内存大小和布局、能够存储于该内存中的值的取值范围以及可应用在该变量上的操作集。C++程序员常常把变量称为“变量”或“对象(object)”。

左值和右值

我们在第5章再详细探讨表达式，现在先介绍 C++ 的两种表达式：

- (1) **左值** (lvalue, 发音为 ell-value)：左值可以出现在赋值语句的左边或右边。
- (2) **右值** (rvalue, 发音为 are-value)：右值只能出现在赋值的右边，不能出现在赋值语句的左边。

变量是左值，因此可以出现在赋值语句的左边。数字字面值是右值，因此不能被赋值。给定以下变量：

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

下列两条语句都会产生编译错误：

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

有些操作符，比如赋值，要求其中的一个操作数必须是左值。结果，可以使用左值的上下文比右值更广。左值出现的上下文决定了左值是如何使用的。例如，表达式

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

中，units_sold 变量被用作两种不同操作符的操作数。+操作符仅关心其操作数的值。变量的值是当前存储在和该变量相关联的内存中的值。加法操作符的作用是取得变量的值并加 1。

变量 units_sold 也被用作=操作符的左操作数。=操作符读取右操作数并写到左操作数。在这个表达式中，加法运算的结果被保存到与 units_sold 相关联的存储单元中，而 units_sold

45 之前的值则被覆盖。



在本书中，我们将看到在许多情形中左值或右值的使用影响程序的操作和/或性能——特别是在向函数传递值或从函数中返回值的时候。

习题

习题 2.12 区分左值和右值，并举例说明。

习题 2.13 举出一个需要左值的例子。

术语：什么是对象？

C++程序员经常随意地使用术语对象。一般而言，对象就是内存中具有类型的区域。说得更具体一些，计算左值表达式就会产生对象。

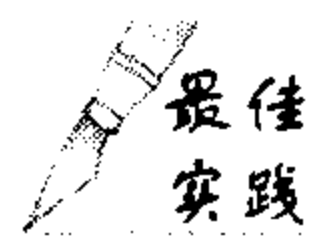
严格地说，有些人只把术语对象用于描述变量或类类型的值。有些人还区别有名字的对象和没名字的对象，当谈到有名字的对象时一般指变量。还有一些人区分对象和值，用术语对象描述可被程序改变的数据，用术语值描述只读数据。

在本书中，我们遵循更为通用的用法，即对象是内存中具有类型的区域。我们可以自由地使用对象描述程序中可操作的大部分数据，而不管这些数据是内置类型还是类类型，是有名字的还是没名字的，是可读的还是可写的。

2.3.2 变量名

变量名，即变量的标识符（identifier），可以由字母、数字和下划线组成。变量名必须以字母或下划线开头，并且区分大小写字母：C++中的标识符都是大小写敏感的。下面定义了4个不同的标识符：

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



语言本身并没有限制变量名的长度，但考虑到将会阅读和/或修改我们的代码的其他人，变量名不应太长。

例如：

```
gosh_this_is_an_impossibly_long_name_to_type
```

就是一个糟糕的标识符。

1. C++关键字

C++保留了一组词用作该语言的关键字。关键字不能用作程序的标识符。表 2-2 列出了 C++ 所有的关键字。

表 2-2 C++关键字

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		

C++还保留了一些词用作各种操作符的替代名。这些替代名用于支持某些不支持标准 C++操作符号集的字符集。它们也不能用作标识符。表 2-3 列出了这些替代名。

46

表 2-3 C++操作符替代名

and	bitand	compl	not_eq	or_eq	xor_eq
and_eq	bitor	not	or	xor	

除了关键字, C++标准还保留了一组标识符用于标准库。标识符不能包含两个连续的下划线, 也不能以下划线开头后面紧跟一个大写字母。有些标识符(在函数外定义的标识符)不能以下划线开头。

2. 变量命名习惯

变量命名有许多被普遍接受的习惯, 遵循这些习惯可以提高程序的可读性。

- 变量名一般用小写字母。例如, 通常会写成 `index`, 而不写成 `Index` 或 `INDEX`。
- 标识符应使用能帮助记忆的名字, 也就是说, 能够提示其在程序中的用法, 如 `on_loan` 或 `salary`。
- 包含多个词的标识符书写为在每个词之间添加一个下划线, 或者每个内嵌的词的第一个字母都大写。例如通常会写成 `student_loan` 或 `studentLoan`, 而不写成 `studentloan`。

47



命名习惯最重要的是保持一致。

习题

习题 2.14 下面哪些(如果有)名字是非法的? 更正每个非法的标识符名字。

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

2.3.3 定义对象

下列语句定义了 5 个变量：

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

每个定义都是以类型说明符 (type specifier) 开始，后面紧跟着以逗号分开的含有一个或多个说明符的列表。分号结束定义。类型说明符指定与对象相关联的类型：int、double、std::string 和 Sales_item 都是类型名。其中 int 和 double 是内置类型，std::string 是标准库定义的类型，Sales_item 是我们在 1.5 节使用的类型，将会在后面章节定义。类型决定了分配给变量的存储空间的大小和可以在其上执行的操作。

多个变量可以定义在同一条语句中：

```
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
```

1. 初始化

变量定义指定了变量的类型和标识符，也可以为对象提供初始值。定义时指定了初始值的对象被称为是已初始化的 (initialized)。C++ 支持两种初始化变量的形式：复制初始化 (copy-initialization) 和直接初始化 (direct-initialization)。复制初始化语法用等号 (=)，直接初始化则是把初始化式放在括号中：

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

48

这两种情形中，ival 都被初始化为 1024。



虽然在本书到目前为止还没有清楚说明，但是在 C++ 中理解“初始化不是赋值”是必要的。初始化指创建变量并给它赋初始值，而赋值则是擦除对象的当前值并用新值代替。

使用 = 来初始化变量使得许多 C++ 编程新手感到迷惑，他们很容易把初始化当成是赋值的一种形式。但是在 C++ 中初始化和赋值是两种不同的操作。这个概念特别容易误导人，因为在许多其他的语言中这两者的差别不过是枝节问题因而可以被忽略。即使在 C++ 中也只有在编写非常复杂的类时才会凸显这两者之间的区别。无论如何，这是一个关键的概念，也是我们将会在整个本书中反复强调的概念。



当初始化类类型对象时，复制初始化和直接初始化之间的差别是很微妙的。我们在第 13 章再详细解释它们之间的差别。现在我们只需知道，直接初始化语法更灵活且效率更高。

2. 使用多个初始化式

初始化内置类型的对象只有一种方法：提供一个值，并且把这个值复制到新定义的对象中。对内置类型来说，复制初始化和直接初始化几乎没有差别。

对类类型的对象来说，有些初始化仅能用直接初始化完成。要想理解其中缘由，需要初步了解类是如何控制初始化的。

每个类都可能会定义一个或几个特殊的成员函数（1.5.2 节）来告诉我们如何初始化类类型的变量。定义如何进行初始化的成员函数称为**构造函数**（constructor）。和其他函数一样，构造函数能接受多个参数。一个类可以定义几个构造函数，每个构造函数必须接受不同数目或者不同类型的参数。

我们以 `string` 类为例（`string` 类将在第 3 章详细讨论）。`string` 类型在标准库中定义，用于存储不同长度的字符串。使用 `string` 时必须包含 `string` 头文件。和 `IO` 类型一样，`string` 定义在 `std` 命名空间中。

`string` 类定义了几个构造函数，使得我们可以用不同的方式初始化 `string` 对象。其中一种初始化 `string` 对象的方式是作为字符串字面值的副本：

```
#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.");
```

49

本例中，两种初始化方式都可以使用。两种定义都创建了一个 `string` 对象，其初始值都是指定的字符串字面值的副本。

也可以通过一个计数器和一个字符初始化 `string` 对象。这样创建的对象包含重复多次的指定字符，重复次数由计数器指定：

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

本例中，初始化 `all_nines` 的唯一方法是直接初始化。有多个初始化式时不能使用复制初始化。

3. 初始化多个变量

当一个定义中定义了两个以上变量的时候，每个变量都可能有自己的初始化式。对象的名字立即变成可见，所以可以用同一个定义中前面已定义变量的值初始化后面的变量。已初始化变量和未初始化变量可以在同一个定义中定义。两种形式的初始化文法可以相互混合。

```
#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";
```

对象可以用任意复杂的表达式（包括函数的返回值）来初始化：

```
double price = 109.99, discount = 0.16;
```

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

本例中，函数 `apply_discount` 接受两个 `double` 类型的值并返回一个 `double` 类型的值。将变量 `price` 和 `discount` 传递给函数，并且用它的返回值来初始化 `sale_price`。

习题

习题 2.15 下面两个定义是否不同？有何不同？

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

如果上述定义有错的话，那么应该怎样改正呢？

习题 2.16 假设 `calc` 是一个返回 `double` 对象的函数。下面哪些是非法定义？改正所有的非法定义。

- (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();`

2.3.4 变量初始化规则

当定义没有初始化式的变量时，系统有时候会帮我们初始化变量。这时，系统提供什么样的值取决于变量的类型，也取决于变量定义的位置。

1. 内置类型变量的初始化

50

内置类型变量是否自动初始化取决于变量定义的位置。在函数体外定义的变量都初始化成 0，在函数体里定义的内置类型变量不进行自动初始化。除了用作赋值操作符的左操作数，未初始化变量用作任何其他用途都是没有定义的。未初始化变量引起的错误难以发现。正如我们在 2.2 节劝告的，永远不要依赖未定义行为。

警告：未初始化的变量引起运行问题

使用未初始化的变量是常见的程序错误，通常也是难以发现的错误。虽然许多编译器都会至少会提醒不要使用未初始化变量，但是编译器并未被要求去检测未初始化变量的使用。而且，没有一个编译器能检测出所有未初始化变量的使用。

有时我们很幸运，使用未初始化的变量导致程序在运行时突然崩溃。一旦跟踪到程序崩溃的位置，就可以轻易地发现没有正确地初始化变量。

但有时，程序运行完毕却产生错误的结果。更糟糕的是，程序运行在一部机器上时能产生正确的结果，但在另外一部机器上却不能得到正确的结果。添加代码到程序的一些不相关的位置，会导致我们认为是正确的程序产生错误的结果。

问题出在未初始化的变量事实上都有一个值。编译器把该变量放到内存中的某个位置，而把这个位置的无论哪种位模式都当成是变量初始的状态。当被解释成整型值时，任何位模式都是合法的值——虽然这个值不可能是程序员想要的。因为这个值合法，所以使用它也不可能会导致程序崩溃。可能的结果是导致程序错误执行和/或错误计算。



最佳
实践

建议每个内置类型的对象都要初始化。虽然这样做并不总是必需的，但是会更加容易和安全，除非你确定忽略初始化式不会带来风险。

51

2. 类类型变量的初始化

每个类都定义了该类型的对象可以怎样初始化。类通过定义一个或多个构造函数来控制类对象的初始化（2.3.3 节）。例如：我们知道 `string` 类至少提供了两个构造函数，其中一个允许我们通过字符串字面值初始化 `string` 对象，另外一个允许我们通过字符和计数器初始化 `string` 对象。

如果定义某个类的变量时没有提供初始化式，这个类也可以定义初始化时的操作。它是通过定义一个特殊的构造函数即**默认构造函数**（default constructor）来实现的。这个构造函数之所以被称作默认构造函数，是因为它是“默认”运行的。如果没有提供初始化式，那么就会使用默认构造函数。不管变量在哪里定义，默认构造函数都会被使用。

大多数类都提供了默认构造函数。如果类具有默认构造函数，那么就可以在定义该类的变量时不用显式地初始化变量。例如，`string` 类定义了默认构造函数来初始化 `string` 变量为空字符串，即没有字符的字符串：

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

有些类类型没有默认构造函数。对于这些类型来说，每个定义都必须提供显式的初始化式。没有初始值是根本不可能定义这种类型的变量的。

习题

习题 2.17 下列变量的初始值（如果有）是什么？

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

2.3.5 声明和定义

正如将在 2.9 节所看到的那样，C++ 程序通常由许多文件组成。为了让多个文件访问相同的变量，C++ 区分了声明和定义。

变量的定义（definition）用于为变量分配存储空间，还可以为变量指定初始值。在一个程序中，变量有且仅有一个定义。

声明（declaration）用于向程序表明变量的类型和名字。定义也是声明：当定义变量时我们声明了它的类型和名字。可以通过使用 `extern` 关键字声明变量名而不定义它。不定义变量的声

52

明包括对象名、对象类型和对象类型前的关键字 `extern`:

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

`extern` 声明不是定义，也不分配存储空间。事实上，它只是说明变量定义在程序的其他地方。程序中变量可以声明多次，但只能定义一次。

只有当声明也是定义时，声明才可以有初始化式，因为只有定义才分配存储空间。初始化式必须要有存储空间来进行初始化。如果声明有初始化式，那么它可被当作是定义，即使声明标记为 `extern`:

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

虽然使用了 `extern`，但是这条语句还是定义了 `pi`，分配并初始化了存储空间。只有当 `extern` 声明位于函数外部时，才可以含有初始化式。

因为已初始化的 `extern` 声明被当作是定义，所以该变量任何随后的定义都是错误的:

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

同样，随后的含有初始化式的 `extern` 声明也是错误的:

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

声明和定义之间的区别可能看起来微不足道，但事实上却是举足轻重的。



在 C++ 语言中，变量必须且仅能定义一次，而且在使用变量之前必须定义或声明变量。

任何在多个文件中使用的变量都需要有与定义分离的声明。在这种情况下，一个文件含有变量的定义，使用该变量的其他文件则包含该变量的声明（而不是定义）。

习题

习题 2.18 解释下列例子中 `name` 的意义:

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

53

2.3.6 名字的作用域

C++ 程序中，每个名字都与唯一的实体（比如变量、函数和类型等）相关联。尽管有这样的要求，还是可以在程序中多次使用同一个名字，只要它用在不同的上下文中，且通过这些上下文可以区分该名字的不同意义。用来区分名字的不同意义的上下文称为作用域（scope）。作用域是程序的一段区域。一个名称可以和不同作用域中的不同实体相关联。

C++语言中，大多数作用域是用花括号来界定的。一般来说，名字从其声明点开始直到其声明所在的作用域结束处都是可见的。例如，思考 1.4.2 节中的程序：

```
#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;
}
```

这个程序定义了三个名字，使用了两个标准库的名字。程序定义了一个名为 main 的函数，以及两个名为 sum 和 val 的变量。名字 main 定义在所有花括号之外，在整个程序都可见。定义在所有函数外部的名字具有**全局作用域**（global scope），可以在程序中的任何地方访问。名字 sum 定义在 main 函数的作用域中，在整个 main 函数中都可以访问，但在 main 函数外则不能。变量 sum 有**局部作用域**（local scope）。名字 val 更有意思，它定义在 for 语句的作用域中，只能在 for 语句中使用，而不能用在 main 函数的其他地方。它具有**语句作用域**（statement scope）。

C++中作用域可嵌套

定义在全局作用域中的名字可以在局部作用域中使用，定义在全局作用域中的名字和定义在函数的局部作用域中的名字可以在语句作用域中使用，等等。名字还可以在内部作用域中重新定义。理解和名字相关联的实体需要明白定义名字的作用域：

```
#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;
}
```

54

这个程序中定义三个变量：string 类型的全局变量 s1、string 类型的局部变量 s2 和 int 类型的局部变量 s1。局部变量 s1 的定义屏蔽（hide）了全局变量 s1。

变量从声明开始才可见，因此执行第一次输出时局部变量 s1 不可见，输出表达式中的 s1 是全局变量 s1，输出“hello world”。第二条输出语句跟在 s1 的局部定义后，现在局部变量

s1 在作用域中。第二条输出语句使用的是局部变量 s1 而不是全局变量 s1，输出 “42 world”。



像上面这样的程序很可能让人大惑不解。在函数内定义一个与函数可能会用到的全局变量同名的局部变量总是不好的。局部变量最好使用不同的名字。

第 7 章将详细讨论局部作用域和全局作用域，第 6 章将讨论语句作用域。C++ 还有另外两种不同级别的作用域：类作用域（class scope，第 12 章将介绍）和命名空间作用域（namespace scope，17.2 节将介绍）。

2.3.7 在变量使用处定义变量

一般来说，变量的定义或声明可以放在程序中能摆放语句的任何位置。变量在使用前必须先声明或定义。



通常把一个对象定义在它首次使用的地方是一个很好的办法。

在对象第一次被使用的地方定义对象可以提高程序的可读性。读者不需要返回到代码段的开始位置去寻找某一特殊变量的定义，而且，在此处定义变量，更容易给它赋以有意义的初始值。

放置声明的一个约束是，变量只在从其定义处开始到该声明所在的作用域的结束处才可以访问。必须在使用该变量的最外层作用域里面或之前定义变量。

55

习题

习题 2.19 下列程序中 j 的值是多少？

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

习题 2.20 下列程序段将会输出什么？

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

习题 2.21 下列程序合法吗？

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

2.4 const 限定符

下列 for 循环语句有两个问题，两个都和使用 512 作为循环上界有关。

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

第一个问题是程序的可读性。比较 index 与 512 有什么意思呢？循环在做什么呢？也就是说 512 作用何在？[本例中，512 被称为魔数 (magic number)，它的意义在上下文中没有体现出来。好像这个数是魔术般地从空中出现的。]

第二个问题是程序的可维护性。假设这个程序非常庞大，512 出现了 100 次。进一步假设在这 100 次中，有 80 次是表示某一特殊缓冲区的大小，剩余 20 次用于其他目的。现在我们需要把缓冲区的大小增大到 1024。要实现这一改变，必须检查每个 512 出现的位置。我们必须确定（在每种情况下都准确地确定）哪些 512 表示缓冲区大小，而哪些不是。改错一个都会使程序崩溃，又得回过头来重新检查。

解决这两个问题的方法是使用一个初始化为 512 的对象：

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

56

通过使用好记的名字如 bufSize，增强了程序的可读性。现在是对对象 bufSize 测试而不是对字面值常量 512 测试：

```
index != bufSize
```

现在如果想要改变缓冲区大小，就不再需要查找和改正 80 次出现的地方。而只有初始化 bufSize 那行需要修改。这种方法不但明显减少了工作量，而且还大大减少了出错的可能性。

1. 定义 const 对象

定义一个变量代表某一常数的方法仍然有一个严重的问题。即 bufSize 是可以被修改的。bufSize 可能被有意或无意地修改。const 限定符提供了一个解决办法，它把一个对象转换成一个常量。

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

定义 bufSize 为常量并初始化为 512。变量 bufSize 仍然是一个左值 (2.3.1 节)，但是现在这个左值是不可修改的。任何修改 bufSize 的尝试都会导致编译错误：

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



因为常量在定义后就不能被修改，所以定义时必须初始化：

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

2. const 对象默认为文件的局部变量

在全局作用域（2.3.6 节）里定义非 const 变量时，它在整个程序中都可以访问。我们可以把一个非 const 变量定义在一个文件中，假设已经做了合适的声明，就可在另外的文件中使用这个变量：

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

与其他变量不同，除非特别说明，在全局作用域声明的 const 变量是定义该对象的文件的局部变量。此变量只存在于那个文件中，不能被其他文件访问。

57 通过指定 const 变量为 extern，就可以在整个程序中访问 const 对象：

```
// 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)
    // ...
```

本程序中，file_1.cc 通过函数 fcn() 的返回值来定义和初始化 bufSize。而 bufSize 定义为 extern，也就意味着 bufSize 可以在其他的文件中使用。file_2.cc 中 bufSize 的声明同样是 extern；这种情况下，extern 标志着 bufSize 是一个声明，所以没有初始化式。



非 const 变量默认为 extern。要使 const 变量能够在其他的文件中访问，必须显式地指定它为 extern。

我们将会在第 2.9.1 节看到为何 const 对象局部于文件创建。

习题

习题 2.22 下列程序段虽然合法，但是风格很糟糕。有什么问题呢？怎样改善？

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

习题 2.23 下列哪些语句合法？对于那些不合法的，请解释为什么不合法。

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

2.5 引用

引用（reference）就是对象的另一个名字。在实际程序中，引用主要用作函数的形式参数。

我们将在 7.2.2 节再详细介绍引用参数。在这一节，我们用独立的对象来介绍并举例说明引用的用法。

引用是一种**复合类型** (compound type)，通过在变量名前添加“&”符号来定义。复合类型是指用其他类型定义的类型。在引用的情况下，每一种引用类型都“关联到”某一其他类型。不能定义引用类型的引用，但可以定义任何其他类型的引用。

58

引用必须用与该引用同类型的对象初始化：

```
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
```

1. 引用是别名

因为引用只是它绑定的对象的另一名字，作用在引用上的所有操作事实上都是作用在该引用绑定的对象上：

```
refVal += 2;
```

将 refVal 指向的对象 ival 加 2。类似地，

```
int ii = refVal;
```

把和 ival 相关联的值赋给 ii。



当引用初始化后，只要该引用存在，它就保持绑定到初始化时指向的对象。不可能将引用绑定到另一个对象。

要理解的重要概念是引用只是对象的另一名字。事实上，我们可以通过 ival 的原名访问 ival，也可以通过它的别名 refVal 访问。赋值只是另外一种操作，因此我们编写

```
refVal = 5;
```

的效果是把 ival 的值修改为 5。这一规则的结果是必须在定义引用时进行初始化。初始化是指明引用指向哪个对象的唯一方法。

2. 定义多个引用

可以在一个类型定义行中定义多个引用。必须在每个引用标识符前添加“&”符号：

```
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
```

3. const 引用

const 引用是指向 const 对象的引用：

```
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
```

59

可以读取但不能修改 refVal，因此，任何对 refVal 的赋值都是不合法的。这个限制有其意义：不能直接对 ival 赋值，因此不能通过使用 refVal 来修改 ival。

同理，用 ival 初始化 ref2 也是不合法的：ref2 是普通的非 **const** 引用 (nonconst reference)，

因此可以用来修改 `ref2` 指向的对象的值。通过 `ref2` 对 `ival` 赋值会导致修改 `const` 对象的值。为阻止这样的修改，需要规定将普通的引用绑定到 `const` 对象是不合法的。

术语：`const` 引用是指向 `const` 的引用

C++程序员常常随意地使用术语 `const` 引用。严格来说，“`const` 引用”的意思是“指向 `const` 对象的引用”。类似地，程序员使用术语“非 `const` 引用”表示指向非 `const` 类型的引用。这种用法非常普遍，我们在本书中也遵循这种用法。

`const` 引用可以初始化为不同类型的对象或者初始化为右值（2.3.1 节），如字面值常量：

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

同样的初始化对于非 `const` 引用却是不合法的，而且会导致编译时错误。其原因非常微妙，值得解释一下。

观察将引用绑定到不同的类型时所发生的事情，最容易理解上述行为。假如我们编写

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

编译器会把这些代码转换成如下形式的编码：

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

如果 `ri` 不是 `const`，那么可以给 `ri` 赋一新值。这样做不会修改 `dval`，而是修改了 `temp`。期望对 `ri` 的赋值会修改 `dval` 的程序员会发现 `dval` 并没有被修改。仅允许 `const` 引用绑定到需要临时使用的值完全避免了这个问题，因为 `const` 引用是只读的。



非 `const` 引用只能绑定到与该引用同类型的对象。
`const` 引用则可以绑定到不同但相关的类型的对象或绑定到右值。

习题

习题 2.24 下列哪些定义是非法的？为什么？如何改正？

- (a) `int ival = 1.01;` (b) `int &rval1 = 1.01;`
 (c) `int &rval2 = ival;` (d) `const int &rval3 = 1;`

习题 2.25 在上题给出的定义下，下列哪些赋值是非法的？如果赋值合法，解释赋值的作用。

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

习题 2.26 (a)中的定义和(b)中的赋值存在哪些不同？哪些是非法的？

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

习题 2.27 下列代码输出什么？

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

2.6 typedef 名字

typedef 可以用来定义类型的同义词：

```
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
```

typedef 名字可以用作类型说明符：

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

typedef 定义以关键字 **typedef** 开始，后面是数据类型和标识符。标识符或类型名并没有引入新的类型，而只是现有数据类型的同义词。**typedef** 名字可出现在程序中类型名可出现的任何位置。

typedef 通常被用于以下三种目的：

- 为了隐藏特定类型的实现，强调使用类型的目的。
- 简化复杂的类型定义，使其更易理解。
- 允许一种类型用于多个目的，同时使得每次使用该类型的目的明确。

61

2.7 枚举

我们经常需要为某些属性定义一组可选择的值。例如，文件打开的状态可能会有三种：输入、输出和追加。记录这些状态值的一种方法是使每种状态都与一个唯一的常数值相关联。我们可能会这样编写代码：

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

虽然这种方法也能奏效，但是它有个明显的缺点：没有指出这些值是相关联的。**枚举**(enumeration) 提供了一种替代的方法，不但定义了整数常量集，而且还把它们聚集成组。

1. 定义和初始化枚举

枚举的定义包括关键字 **enum**，其后是一个可选的枚举类型名，和一个用花括号括起来、用逗号分开的**枚举成员**(enumerator)列表。

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

默认地，第一个枚举成员赋值为 0，后面的每个枚举成员赋的值比前面的大 1。

2. 枚举成员是常量

可以为一个或多个枚举成员提供初始值，用来初始化枚举成员的值必须是一个常量表达式 (constant expression)。常量表达式是编译器在编译时就能够计算出结果的整型表达式。整型字面值常量是常量表达式，正如一个通过常量表达式自我初始化的 const 对象 (2.4 节) 也是常量表达式一样。

例如，可以定义下列枚举类型：

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

在枚举类型 Forms 中，显式将 shape 赋值为 1。其他枚举成员隐式初始化：sphere 初始化为 2，cylinder 初始化为 3，polygon 初始化为 4。

枚举成员值可以是不唯一的。

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

本例中，枚举成员 point2d 显式初始化为 2。下一个枚举成员 point2w 默认初始化，即它的值比前一枚举成员的值大 1，因此 point2w 初始化为 3。枚举成员 point3d 显式初始化为 3。一样，point3w 默认初始化，结果为 4。

62

不能改变枚举成员的值。枚举成员本身就是一个常量表达式，所以也可用于需要常量表达式的任何地方。

3. 每个 enum 都定义一种唯一的类型

每个 enum 都定义了一种新的类型。和其他类型一样，可以定义和初始化 Points 类型的对象，也可以以不同的方式使用这些对象。枚举类型的对象的初始化或赋值，只能通过其枚举成员或同一枚举类型的其他对象来进行：

```
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
```

注意把 3 赋给 Points 对象是非法的，即使 3 与一个 Points 枚举成员相关联。

2.8 类类型

C++ 中，通过定义类 (class) 来自定义数据类型。类定义了该类型的对象包含的数据和该类型的对象可以执行的操作。标准库类型 string、istream 和 ostream 都定义成类。

C++ 对类的支持非常丰富——事实上，定义类是如此重要，我们把第三到第五部分全部用来描述 C++ 对类及类操作的支持。

在第 1 章中，我们使用 Sales_item 类型来解决书店问题。使用 Sales_item 类型的对象来记录对应于特定 ISBN 的销售数据。在这节中，我们先了解如何定义简单的类，如 Sales_item 类。

1. 从操作开始设计类

每个类都定义了一个接口 (interface) 和一个实现 (implementation)。接口由使用该类的代码需要执行的操作组成。实现一般包括该类所需要的数据。实现还包括定义该类需要的但又不供一般性使用的函数。

定义类时, 通常先定义该类的接口, 即该类所提供的操作。通过这些操作, 可以决定该类完成其功能所需要的数据, 以及是否需要定义一些函数来支持该类的实现。

我们将要定义的类型所支持的操作, 就是我们在第1章中所用到的操作。这些操作如下 (参见 1.5.1 节):

63

- 加法操作符, 将两个 Sales_item 相加。
- 输入和输出操作符, 读和写 Sales_item 对象。
- 赋值操作符, 把 Sales_item 对象赋给另一个 Sales_item 对象。
- same_isbn 函数, 检测两个对象是否指同一本书。

在学完怎样定义函数和操作符后, 我们将会在第7章和第14章看到该怎样来定义这些操作。虽然现在不能实现这些函数, 但通过思考这些操作必须要实现的功能, 我们可以看出该类需要什么样的数据。Sales_item 类必须

- (1) 记录特定书的销售册数。
- (2) 记录该书的总销售收入。
- (3) 计算该书的平均售价。

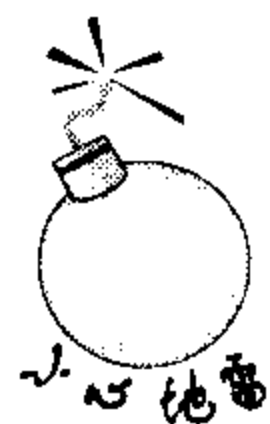
查看以上所列出的任务, 可以知道需要一个 unsigned 类型的对象来记录书的销售册数, 一个 double 类型的对象来记录总销售收入, 然后可以用总收入除以销售册数计算出平均售价。因为我们还想知道是在记录哪本书, 所以还需要定义一个 string 类型的对象来记录书的 ISBN。

2. 定义 Sales_item 类

很明显, 我们需要能够定义一种包含这三个数据元素和在第1章所用到的操作的数据类型。在 C++ 语言中, 定义这种数据类型的方法就是定义类:

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

类定义以关键字 class 开始, 其后是该类的名字标识符。类体位于花括号里面。花括号后面必须要跟一个分号。



编程新手经常会忘记类定义后面的分号, 这是个很普遍的错误!

类体可以为空。类体定义了组成该类型的数据和操作。这些操作和数据是类的一部分, 也称为类的成员 (member)。操作称为成员函数 (1.5.2 节), 而数据则称为数据成员 (data member)。

64

类也可以包含 0 个到多个 `private` 或 `public` 访问标号 (access label)。访问标号控制类的成员在类外部是否可访问。使用该类的代码可能只能访问 `public` 成员。

定义了类，也就定义了一种新的类型。类名就是该类型的名字。通过命名 `Sales_item` 类，表示 `Sales_item` 是一种新的类型，而且程序也可以定义该类型的变量。

每一个类都定义了它自己的作用域 (2.3.6 节)。也就是说，数据和操作的名字在类的内部必须唯一，但可以重用定义在类外的名字。

3. 类的数据成员

定义类的数据成员和定义普通变量有些相似。我们同样是指定一种类型并给该成员一个名字：

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

这个类含有三个数据成员：一个名为 `isbn` 的 `string` 类型成员，一个名为 `units_sold` 的 `unsigned` 类型成员，一个名为 `revenue` 的 `double` 类型成员。类的数据成员定义了该类类型对象的内容。当定义 `Sales_item` 类型的对象时，这些对象将包含一个 `string` 型变量，一个 `unsigned` 型变量和一个 `double` 型变量。

定义变量和定义数据成员存在非常重要的区别：一般不能把类成员的初始化作为其定义的一部分。当定义数据成员时，只能指定该数据成员的名字和类型。类不是在类定义里定义数据成员时初始化数据成员，而是通过称为构造函数 (2.3.3 节) 的特殊成员函数控制初始化。我们将在 7.7.3 节定义 `Sales_item` 的构造函数。

4. 访问标号

访问标号负责控制使用该类的代码是否可以使用给定的成员。类的成员函数可以使用类的任何成员，而不管其访问级别。访问标号 `public`、`private` 可以多次出现在类定义中。给定的访问标号应用到下一个访问标号出现时为止。

类中 `public` 部分定义的成员在程序的任何部分都可以访问。一般把操作放在 `public` 部分，这样程序的任何代码都可以执行这些操作。

不是类的组成部分的代码不能访问 `private` 成员。通过设定 `Sales_item` 的数据成员为 `private`，可以保证对 `Sales_item` 对象进行操作的代码不能直接操纵其数据成员。就像我们在第 1 章编写的程序那样，程序不能访问类中的 `private` 成员。`Sales_item` 类型的对象可以执行那些操作，但是不能直接修改这些数据。

65

5. 使用 `struct` 关键字

C++ 支持另一个关键字 `struct`，它也可以定义类类型。`struct` 关键字是从 C 语言中继承过来的。

如果使用 `class` 关键字来定义类，那么定义在第一个访问标号前的任何成员都隐式指定为 `private`；如果使用 `struct` 关键字，那么这些成员都是 `public`。使用 `class` 还是 `struct` 关键字来定义类，仅仅影响默认的初始访问级别。

可以等效地定义 `Sales_item` 类为：

```
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;
};

```

本例的类定义和前面的类定义只有两个区别：这里使用了关键字 `struct`，并且没有在花括号后使用关键字 `public`。`struct` 的成员都是 `public`，除非有其他特殊的声明，所以就没有必要添加 `public` 标号。



用 `class` 和 `struct` 关键字定义类的唯一差别在于默认访问级别：默认情况下，`struct` 的成员为 `public`，而 `class` 的成员为 `private`。

习题

习题 2.28 编译以下程序，确定你的编译器是否会警告遗漏了类定义后面的分号。

```

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

```

如果编译器的诊断结果难以理解，记住这些信息以备后用。

习题 2.29 区分类中的 `public` 部分和 `private` 部分。

习题 2.30 定义表示下列类型的类的数据成员：

- | | |
|-----------|------------|
| (a) 电话号码 | (b) 地址 |
| (c) 员工或公司 | (d) 某大学的学生 |

2.9 编写自己的头文件

我们已经从 1.5 节了解到，一般类定义都会放入头文件 (header file)。在本节中我们将看到怎样为 `Sales_item` 类定义头文件。

事实上，C++ 程序使用头文件包含的不仅仅是类定义。回想一下，名字在使用前必须先声明或定义。到目前为止，我们编写的程序是把代码放到一个文件里来处理这个要求。只要每个实体位于使用它的代码之前，这个策略就有效。但是，很少有程序简单到可以放置在一个文件中。由多个文件组成的程序需要一种方法连接名字的使用和声明，在 C++ 中这是通过头文件实现的。

为了允许把程序分成独立的逻辑块，C++ 支持所谓的**分别编译** (separate compilation)。这样程序可以由多个文件组成。为了支持分别编译，我们把 `Sales_item` 的定义放在一个头文件里面。我们后面在 7.7 节中定义的 `Sales_item` 成员函数将放在单独的源文件中。像 `main` 这样使用 `Sales_item` 对象的函数放在其他的源文件中，任何使用 `Sales_item` 的源文件都必须包含 `Sales_item.h` 头文件。

2.9.1 设计自己的头文件

头文件为相关声明提供了一个集中存放的位置。头文件一般包含类的定义、extern 变量的声明和函数的声明。函数的声明将在 7.4 节介绍。使用或定义这些实体的文件要包含适当的头文件。

头文件的正确使用能够带来两个好处：保证所有文件使用给定实体的同一声明；当声明需要修改时，只有头文件需要更新。

设计头文件还需要注意以下几点：头文件中所做的声明在逻辑上应该是适于放在一起的。编译头文件需要一定的时间。如果头文件太大，程序员可能不愿意承受包含该头文件所带来的编译时代价。

 提示

为了减少处理头文件的编译时间，有些 C++ 的实现支持预编译头文件。欲进一步了解详细情况，请参考你的 C++ 实现的手册。

编译和链接多个源文件

要产生可执行文件，我们不但要告诉编译器到哪里去查找 **main** 函数，而且还要告诉编译器到哪里去查找 **Sales_item** 类所定义的成员函数的定义。假设我们有两个文件：main.cc 含有 main 函数的定义，Sales_item.cc 含有 Sales_item 的成员函数。我们可以按以下方式编译这两个文件：

```
$ 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
```

其中 \$ 是我们的系统提示符，# 开始命令行注释。现在我们可以运行可执行文件，它将运行我们的 main 程序。

如果我们只是修改了一个 .cc 源文件，较有效的方法是只重新编译修改过的文件。大多数编译器都提供了分别编译每一个文件的方法。通常这个过程产生 .o 文件，.o 扩展名暗示该文件含有目标代码。

编译器允许我们把目标文件链接在一起以形成可执行文件。我们所使用的系统可以通过命令名 CC 调用编译。因此可以按以下方式编译程序：

```
$ 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
```

你需要检查所用编译器的用户手册，了解如何编译和执行由多个源文件组成的程序。

 提示

许多编译器提供了增强其错误检测能力的选项。查看所用编译器的用户指南，了解有哪些额外的检测方法。

1. 头文件用于声明而不是用于定义

当设计头文件时，记住定义和声明的区别是很重要的。定义只可以出现一次，而声明则可以出现多次（2.3.5 节）。下列语句是一些定义，所以不应该放在头文件里：

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

虽然 `ival` 声明为 `extern`，但是它有初始化式，代表这条语句是一个定义。类似地，`fica_rate` 的声明虽然没有初始化式，但也是一个定义，因为没有关键字 `extern`。同一个程序中有两个以上文件含有上述任一个定义都会导致多重定义链接错误。

67



因为头文件包含在多个源文件中，所以不应该含有变量或函数的定义。

对于头文件不应该含有定义这一规则，有三个例外。头文件可以定义类、值在编译时就已知道的 `const` 对象和 `inline` 函数（7.6 节介绍 `inline` 函数）。这些实体可在多个源文件中定义，只要每个源文件中的定义是相同的。

68

在头文件中定义这些实体，是因为编译器需要它们的定义（不只是声明）来产生代码。例如：为了产生能定义或使用类的对象的代码，编译器需要知道组成该类型的数据成员。同样还需要知道能够在这些对象上执行的操作。类定义提供所需要的信息。在头文件中定义 `const` 对象则需要更多的解释。

2. 一些 `const` 对象定义在头文件中

回想一下，`const` 变量（2.4 节）默认时是定义该变量的文件的局部变量。正如我们现在所看到的，这样设置默认情况的原因在于允许 `const` 变量定义在头文件中。

在 C++ 中，有些地方需要放置常量表达式（2.7 节）。例如，枚举成员的初始化式必须是常量表达式。在以后的章节中将会看到其他需要常量表达式的例子。

一般来说，常量表达式是编译器在编译时就能够计算出结果的表达式。当 `const` 整型变量通过常量表达式自我初始化时，这个 `const` 整型变量就可能是常量表达式。而 `const` 变量要成为常量表达式，初始化式必须为编译器可见。为了能够让多个文件使用相同的常量值，`const` 变量和它的初始化式必须是每个文件都可见的。而要使初始化式可见，一般都把这样的 `const` 变量定义在头文件中。那样的话，无论该 `const` 变量何时使用，编译器都能够看见其初始化式。

但是，C++ 中的任何变量都只能定义一次（2.3.5 节）。定义会分配存储空间，而所有对该变量的使用都关联到同一存储空间。因为 `const` 对象默认为定义它的文件的局部变量，所以把它们的定义放在头文件中是合法的。

这种行为有一个很重要的含义：当我们在头文件中定义了 `const` 变量后，每个包含该头文件的源文件都有了自己的 `const` 变量，其名称和值都一样。

当该 `const` 变量是用常量表达式初始化时，可以保证所有的变量都有相同的值。但是在实践中，大部分的编译器在编译时都会用相应的常量表达式来替换对这些 `const` 变量的使用。所以，在实践中不会有任何存储空间用于存储用常量表达式初始化的 `const` 变量。

如果 `const` 变量不是用常量表达式初始化，那么它就不应该在头文件中定义。相反，和其他的变量一样，该 `const` 变量应该在一个源文件中定义并初始化。应在头文件中为它添加 `extern` 声明，以使其能被多个文件共享。

习题

习题 2.31 判别下列语句哪些是声明，哪些是定义，请解释原因。

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

习题 2.32 下列声明和定义哪些应该放在头文件中？哪些应该放在源文件中？请解释原因。

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

习题 2.33 确定你的编译器提供了哪些提高警告级别的选项。使用这些选项重新编译以前选择的程序，查看是否会报告新的问题。

2.9.2 预处理器的简单介绍

既然已经知道了什么应该放在头文件中，那么我们下一个问题就是真正地编写头文件。我们
 69 知道要使用头文件，必须在源文件中 `#include` 该头文件。为了编写头文件，我们需要进一步理解 `#include` 指示是怎样工作的。`#include` 设施是 C++ 预处理器 (preprocessor) 的一部分。预处理器处理程序的源代码，在编译器之前运行。C++ 继承了 C 的非常精细的预处理器。现在的 C++ 程序以高度受限的方式使用预处理器。

`#include` 指示只接受一个参数：头文件名。预处理器用指定的头文件的内容替代每个 `#include`。我们自己的头文件存储在文件中。系统的头文件可能用特定于编译器的更高效的格式保存。无论头文件以何种格式保存，一般都含有支持分别编译所需的类定义及变量和函数的声明。

1. 头文件经常需要其他头文件

头文件经常 `#include` 其他头文件。头文件定义的实体经常使用其他头文件的设施。例如，定义 `Sales_item` 类的头文件必须包含 `string` 库。`Sales_item` 类含有一个 `string` 类型的数据成员，因此必须可以访问 `string` 头文件。

包含其他头文件是如此司空见惯，甚至一个头文件被多次包含进同一源文件也不稀奇。例如，使用 `Sales_item` 头文件的程序也可能使用 `string` 库。该程序不会(也不应该)知道 `Sales_item` 头文件使用了 `string` 库。在这种情况下，`string` 头文件被包含了两次：一次是通过程序本身直接包含，另一次是通过包含 `Sales_item` 头文件而间接包含。

70 因此，设计头文件时，应使其可以多次包含在同一源文件中，这一点很重要。我们必须保证多次包含同一头文件不会引起该头文件定义的类和对象被多次定义。使得头文件安全的通用做法，是使用预处理器定义头文件保护符 (header guard)。头文件保护符用于避免在已经见到头文件的情况下重新处理该头文件的内容。

2. 避免多重包含

在编写头文件之前,我们需要引入一些额外的预处理器设施。预处理器允许我们自定义变量。



预处理器变量¹的名字在程序中必须是唯一的。任何与预处理器变量相匹配的名字的使用都关联到该预处理器变量。

为了避免名字冲突,预处理器变量经常用全大写字母表示。

预处理器变量有两种状态:已定义或未定义。定义预处理器变量和检测其状态所用的预处理器指示不同。`#define` 指示接受一个名字并定义该名字为预处理器变量。`#ifndef` 指示检测指定的预处理器变量是否未定义。如果预处理器变量未定义,那么跟在其后的所有指示都被处理,直到出现`#endif`。

可以使用这些设施来预防多次包含同一头文件:

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

条件指示

```
#ifndef SALESITEM_H
```

测试 `SALESITEM_H` 预处理器变量是否未定义。如果 `SALESITEM_H` 未定义,那么`#ifndef` 测试成功,跟在`#ifndef` 后面的所有行都被执行,直到发现`#endif`。相反,如果 `SALESITEM_H` 已定义,那么`#ifndef` 指示测试为假,该指示和`#endif` 指示间的代码都被忽略。

为了保证头文件在给定的源文件中只处理过一次,我们首先检测`#ifndef`。第一次处理头文件时,测试会成功,因为 `SALESITEM_H` 还未定义。下一条语句定义了 `SALESITEM_H`。那样的话,如果我们编译的文件恰好又一次包含了该头文件。`#ifndef` 指示会发现 `SALESITEM_H` 已经定义,并且忽略该头文件的剩余部分。



头文件应该含有保护符,即使这些头文件不会被其他头文件包含。编写头文件保护符并不困难,而且如果头文件被包含多次,它可以避免难以理解的编译错误。

71

当没有两个头文件定义和使用同名的预处理器变量时,这个策略相当有效。我们可以用定义在头文件里的实体(如类)来命名预处理器变量来避免预处理器变量重名的问题。一个程序只能含有一个名为 `Sales_item` 的类。通过使用类名来组成头文件和预处理器变量的名字,可以使得很可能只有一个文件将会使用该预处理器变量。

3. 使用自定义的头文件

`#include` 指示接受以下两种形式:

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

1. 此处的预处理器变量在许多图书和文献中称为预处理器常量。——译者注

如果头文件名括在尖括号 (< >) 里, 那么认为该头文件是标准头文件。编译器将会在预定义的位置集查找该头文件, 这些预定义的位置可以通过设置查找路径环境变量或者通过命令行选项来修改。使用的查找方法因编译器的不同而差别迥异。建议你咨询同事或者查阅编译器用户指南来获得更多的信息。如果头文件名括在一对引号里, 那么认为它是非系统头文件, 非系统头文件的查找通常开始于源文件所在的路径。

72

小结

类型是 C++ 程序设计的基础。

每种类型都定义了其存储空间要求和可以在该类型的所有对象上执行的操作。C++ 提供了一组基本内置类型, 如 `int`、`char` 等。这些类型与它们在机器硬件上的表示方式紧密相关。

类型可以为 `const` 或非 `const`; `const` 对象必须要初始化, 且其值不能被修改。另外, 我们还可以定义复合类型, 如引用。引用为对象提供了另一个名字。复合类型是用其他类型定义的类型。

C++ 语言支持通过定义类来自定义类型。标准库使用类设施来提供一组高级的抽象概念, 如 `IO` 和 `string` 类型。

C++ 是一种静态类型语言: 变量和函数在使用前必须先声明。变量可以声明多次但是只能定义一次。定义变量时就进行初始化几乎总是个好主意。

术语

access label (访问标号) 类的成员可以定义为 `private`, 这能够防止使用该类型的代码访问该成员。成员还可以定义为 `public`, 这将使该整个程序中都可访问成员。

address (地址) 一个数字, 通过该数字可在存储器上找到一个字节。

arithmetic type (算术类型) 表示数值即整数和浮点数的类型。浮点型值有三种类型: `long double`、`double` 和 `float`, 分别表示扩展精度值、双精度值和单精度值。一般总是使用 `double` 型。特别地, `float` 只能保证六位有效数字, 这对于大多数的计算来说都不够。整型包括 `bool`、`char`、`wchar_t`、`short`、`int` 和 `long`。整型可以是带符号或无符号的。一

般在算术计算中总是避免使用 `short` 和 `char`。`unsigned` 可用于计数。`bool` 类型只有 `true` 和 `false` 两个值。`wchar_t` 类型用于扩展字符集的字符; `char` 类型用于适合 8 个位的字符, 比如 Latin-1 或者 ASCII。

array (数组) 存储一组可通过下标访问的未命名对象的数据结构。本章介绍了存储字符串字面值的字符数组。第 4 章将会更加详细地介绍数组。

byte (字节) 最小的可寻址存储单元。大多数的机器上一个字节有 8 个位 (bit)。

class (类) C++ 中定义数据类型的机制。类可以用 `class` 或 `struct` 关键字定义。类可以有数据和函数成员。成员可以是 `public` 或

private。一般来说，定义该类型的操作的函数成员设为 **public**；用于实现该类的数据成员和函数设为 **private**。默认情况下，用 **class** 关键字定义的类其成员为 **private**，而用 **struct** 关键字定义的类其成员为 **public**。

class member (类成员) 类的一部分，可以是数据或操作。

compound type (复合类型) 用其他类型定义的类型，如引用。第4章将介绍另外两种复合类型：指针和数组。

const reference (const 引用) 可以绑定到 **const** 对象、非 **const** 对象或右值的引用。**const** 引用不能改变与其相关联的对象。

constant expression (常量表达式) 值可以在编译时计算出来的整型表达式。

constructor (构造函数) 用来初始化新建对象的特殊成员函数。构造函数的任务是保证对象的数据成员拥有可靠且合理的初始值。

copy-initialization (复制初始化) 一种初始化形式，用“=”表明变量应初始化为初始化式的副本。

data member (数据成员) 组成对象的数据元素。数据成员一般应设为私有的。

declaration (声明) 表明在程序中其他地方定义的变量、函数或类型的存在性。有些声明也是定义。只有定义才为变量分配存储空间。可以通过在类型前添加关键字 **extern** 来声明变量。名字直到定义或声明后才能使用。

default constructor (默认构造函数) 在没有为类类型对象的初始化式提供显式值时所使用的构造函数。例如，**string** 类的默认构造函数将新建的 **string** 对象初始化为空 **string**，而其他构造函数都是在创建 **string** 对象时用指定的字符去初始化 **string** 对象。

definition (定义) 为指定类型的变量分配存储空间，也可能可选地初始化该变量。名字直到定义或声明后才能使用。

direct-initialization (直接初始化) 一种初始化形式，将逗号分隔的初始化式列表放在圆括号内。

enumeration (枚举) 将一些命名整型常量聚成组的一种类型。

enumerator (枚举成员) 枚举类型的有名字的成员。每个枚举成员都初始化为整型值且值为 **const**。枚举成员可用在需要整型常量表达式的地方，比如数组定义的维度。

escape sequence (转义字符) 一种表示字符的可选机制。通常用于表示不可打印字符如换行符或制表符。转义字符是反斜线后面跟着一个字符、一个3位八进制数或一个十六进制的数。C++语言定义的转义字符列在2.2节。转义字符还可用作字符面值（括在单引号里）或用作字符串面值的一部分（括在双引号里）。

global scope (全局作用域) 位于任何其他作用域外的作用域。

header (头文件) 使得类的定义和其他声明在多个源文件中可见的一种机制。用户定义的头文件以文件方式保存。系统头文件可能以文件方式保存，也可能以系统特有的其他格式保存。

header guard (头文件保护符) 为防止头文件被同一源文件多次包含而定义的预处理器变量。

identifier (标识符) 名字。每个标识符都是字母、数字和下划线的非空序列，且序列不能以数字开头。标识符是大小写敏感的：大写字母和小写字母含义不同。标识符不能使用C++中的关键字，不能包含相邻的下划线，也不能

以下划线后跟一个大写字母开始。

implementation (实现) 定义数据和操作的类成员 (通常为 `private`)，这些数据和操作并非为使用该类型的代码所用。例如，`istream` 和 `ostream` 类管理的 IO 缓冲区是它们的实现的一部分，但并不允许这些类的使用者直接访问。

initialized (已初始化的) 含有初始值的变量。当定义变量时，可指定初始值。变量通常要初始化。

integral type (整型) 见 `arithmetic type`。

interface (接口) 由某种类型支持的操作。设计良好的类分离了接口和实现，在类的 `public` 部分定义接口，`private` 部分定义实现。数据成员一般是实现的一部分。当函数成员是期望该类型的使用者使用的操作时，函数成员就是接口的一部分 (因此为 `public`)；当函数成员执行类所需要的、非一般性使用的操作时，函数成员就是实现的一部分。

link (链接) 一个编译步骤，此时多个目标文件放置在一起以形成可执行程序。链接步骤解决了文件间的依赖，如将一个文件中的函数调用链接到另一个文件中的函数定义。

literal constant (字面值常量) 诸如数、字符或字符串的值，该值不能修改。字面值字符用单引号括住，而字面值字符串则用双引号括住。

local scope (局部作用域) 用于描述函数作用域和函数内嵌套的作用域的术语。

lvalue (左值) 可以出现在赋值操作左边的值。非 `const` 左值可以读也可以写。

magic number (魔数) 程序中意义重大但又不明显的字面值数字。它的出现好像变魔术一般。

nonconst reference (非 `const` 引用) 只能

绑定到与该引用同类型的非 `const` 左值的引用。非 `const` 引用可以修改与其相关联的对象的值。

nonprintable character (非打印字符) 不可见字符。如控制符、回退删除符、换行符等。

object (对象) 具有类型的一段内存区域。变量就是一个有名字的对象。

preprocessor (预处理器) 预处理器是作为 C++ 程序编译的一部分运行的程序。预处理器继承于 C 语言，C++ 的特征大量减少了它的使用，但仍保存了一个很重要的用法：`#include` 设施，用来把头文件并入程序。

private member (私有成员) 使用该类的代码不可访问的成员。

public member (公用成员) 可被程序的任何部分使用的类成员。

reference (引用) 对象的别名。定义如下：

```
type &id = object ;
```

定义 `id` 为 `object` 的另一名字。任何对 `id` 的操作都会转变为对 `object` 的操作。

run time (运行时) 指程序正执行的那段时间。

rvalue (右值) 可用于赋值操作的右边但不能用于左边的值。右值只能读而不能写。

scope (作用域) 程序的一部分，在其中名字有意义。C++ 含有下列几种作用域：

全局——名字定义在任何其他作用域外。

类——名字由类定义。

命名空间——名字在命名空间中定义。

局部——名字在函数内定义。

块——名字定义在语句块中，也就是说，定义在一对花括号里。

语句——名字在语句 (如 `if`、`while` 和 `for` 语句) 的条件内定义。

作用域可嵌套。例如，在全局作用域中声明的名字在函数作用域和语句作用域中都可以访问。

separate compilation (分别编译) 将程序分成多个分离的源文件进行编译。

signed (带符号型) 保存负数、正数或零的整型。

statically typed (静态类型的) 描述进行编译时类型检查的语言(如 C++) 的术语。C++ 在编译时验证表达式使用的类型可以执行该表达式需要的操作。

struct 用来定义类的关键字。除非有特殊的声明，默认情况下 struct 的成员都为公用的。

type-checking (类型检查) 编译器验证给定类型的对象的使用方式是否与该类型的定义一致，描述这一过程的术语。

type specifier (类型说明符) 定义或声明中命名其后变量的类型的部分。

typedef 为某种类型引入同义词。格式：

```
typedef type synonym ;
```

定义 *synonym* 为名为 *type* 的类型的另一名字。

undefined behavior (未定义行为) 语言没有规定其意义的用法。编译器可以自由地做它想

做的事。有意或无意地依赖未定义行为将产生大量难以跟踪的运行时错误和可移植性问题。

uninitialized (未初始化的) 没有指定初始值的变量。未初始化变量不是 0 也不是“空”，相反，它会保存碰巧遗留在分配给它的内存里的任何位。未初始化变量会产生很多错误。

unsigned (无符号型) 保存大于等于零的值的整型。

variable initialization (变量初始化) 描述当没有给出显式初始化式时初始化变量或数组元素的规则的术语。对类类型来说，通过运行类的默认构造函数来初始化对象。如果没有默认构造函数，那么将会出现编译时错误：必须要给对象指定显式的初始化式。对于内置类型来说，初始化取决于作用域。定义在全局作用域的对象初始化为 0，而定义在局部作用域的对象则未初始化，拥有未定义值。

void type (空类型) 用于特殊目的的操作也没有值的类型。不可能定义一个 void 类型的变量。最经常用作不返回结果的函数的返回类型。

word (字) 机器上的自然的整型计算单元。通常一个字足以容纳一个地址。一般在 32 位的机器上，机器字长为 4 个字节。

第 3 章

标准库类型

目录

3.1 命名空间的 using 声明	68
3.2 标准库 string 类型	70
3.3 标准库 vector 类型	78
3.4 迭代器简介	83
3.5 标准库 bitset 类型	88
小结	92
术语	92

除第 2 章介绍的基本数据类型外，C++ 还定义了一个内容丰富的抽象数据类型标准库。其中最重要的标准库类型是 `string` 和 `vector`，它们分别定义了大小可变的字符串和集合。`string` 和 `vector` 往往将迭代器用作配套类型（companion type），用于访问 `string` 中的字符，或者 `vector` 中的元素。这些标准库类型是语言组成部分中更基本的那些数据类型（如数组和指针）的抽象。

另一种标准库类型 `bitset`，提供了一种抽象方法来操作位的集合。与整型值上的内置位操作符相比，`bitset` 类类型提供了一种更方便的处理位的方式。

本章将介绍标准库中的 `vector`、`string` 和 `bitset` 类型。第 4 章将讨论数组和指针，第 5 章将讲述内置位操作符。

第2章所涉及的类型都是低级数据类型：这些类型表示数值或字符的抽象，并根据其具体机器表示来定义。

除了这些在语言中定义的类型外，C++标准库还定义了许多更高级的抽象数据类型（abstract data type）。之所以说这些标准库类型是更高级的，是因为其中反映了更复杂的概念；之所以说它们是抽象的，是因为我们在使用时不需要关心它们是如何表示的，只需知道这些抽象数据类型支持哪些操作就可以了。

两种最重要的标准库类型是string和vector。string类型支持长度可变的字符串，vector可用于保存一组指定类型的对象。说它们重要，是因为它们在C++定义的基本类型基础上作了一些改进。第4章还将学习类似于标准库中string和vector类型的语言级构造，但标准库的string和vector类型可能更灵活，且不易出错。

另一种标准库类型提供了更方便和合理有效的语言级的抽象设施，它就是bitset类。通过这个类可以把某个值当作位的集合来处理。与5.3节介绍的位操作符相比，bitset类提供操作位更直接的方法。

在继续探究标准库类型之前，我们先看一种机制，这种机制能够简化对标准库中所定义名字的访问。

3.1 命名空间的using声明

在本章之前看到的程序，都是通过直接说明名字来自std命名空间，来引用标准库中的名字。例如，需要从标准输入读取数据时，就用std::cin。这些名字都用了::操作符，该操作符是作用域操作符（1.2.2节）。它的含义是右操作数的名字可以在左操作数的作用域中找到。因此，std::cin的意思是说所需名字cin是在命名空间std中定义的。显然，通过这种符号引用标准库名字的方式是非常麻烦的。

幸运的是，C++提供了更简洁的方式来使用命名空间成员。本节将介绍一种最安全的机制：**using声明**。关于其他简化使用命名空间中名字的方法将在17.2节中介绍。

使用using声明可以在不需要加前缀namespace_name::的情况下访问命名空间中的名字。using声明的形式如下：

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

一旦使用了using声明，我们就可以直接引用名字，而不需要再引用该名字的命名空间：

```
#include <string>
#include <iostream>
// using declarations states our intent to use these names from the namespace std
using std::cin;
using std::string;
int main()
{
    string s;           // ok: string is now a synonym for std::string
    cin >> s;           // ok: cin is now a synonym for std::cin
    cout << s;          // error: no using declaration; we must use full name
    std::cout << s;     // ok: explicitly use cout from namespace std
}
```

没有 using 声明，而直接使用命名空间中名字的未限定版本是错误的，尽管有些编译器也许无法检测出这种错误。

1. 每个名字都需要一个 using 声明

一个 using 声明一次只能作用于一个命名空间成员。using 声明可用来明确指定在程序中用到的命名空间中的名字，如果希望使用 std（或其他的命名空间）中的几个名字，则必须为要用到的每个名字都提供一个 using 声明。例如，利用 using 声明可以这样重新编写 1.2.2 节中的加法程序：

```
#include <iostream>
// using declarations for names from the standard library
using std::cin;
using std::cout;
using std::endl;
int main()
{
    cout << "Enter two numbers:" << endl;
    int v1, v2;
    cin >> v1 >> v2;
    cout << "The sum of " << v1
        << " and " << v2
        << " is " << v1 + v2 << endl;
    return 0;
}
```

对 cin, cout 和 endl 进行 using 声明，就意味着以后可以省去前缀 std::，直接使用命名空间中的名字，这样代码可以更易读。

从这里开始，假定本书所有例子中所用到的标准库中的名字都已提供了 using 声明。这样，无论是在文档还是在代码实例中引用 cin，我们都不再写为前缀形式 std::cin。为了使代码实例简短，我们还省略了编译时所必需的 using 声明。同样的，程序实例也会省略必需的#include 指示。本书附录 A 中的表 A-1 列出了本书中用到的标准库名字的库名和相应的头文件。



在编译我们提供的实例程序前，读者一定要注意在程序中添加适当的#include 和 using 声明。

79

2. 使用标准库类型的类定义

有一种情况下，必须总是使用完全限定的标准库名字：在头文件中。理由是头文件的内容会被预处理器复制到程序中。用#include 包含文件时，相当于头文件中的文本将成为我们编写的文件的一部分。如果在头文件中放置 using 声明，就相当于在包含该头文件的每个程序中都放置了同一 using 声明，不论该程序是否需要 using 声明。



通常，头文件中应该只定义确实必要的东西。请养成这个好习惯。

习题

习题 3.1 用适当的 using 声明，而不用 std:: 前缀，访问标准库中名字的方法，重新编写 2.3 节的程序，计算一给定数的给定次幂的结果。

3.2 标准库 string 类型

string 类型支持长度可变的字符串，C++ 标准库将负责管理与存储字符相关的内存，以及提供各种有用的操作。标准库 string 类型的目的就是满足对字符串的一般应用。

与其他的标准库类型一样，用户程序要使用 string 类型对象，必须包含相关头文件。如果提供了合适的 using 声明，那么编写出来的程序将会变得简短些：

```
#include <string>
using std::string;
```

3.2.1 string 对象的定义和初始化

string 标准库支持几个构造函数（2.2.3 节）。构造函数是一个特殊成员函数，定义如何初始化该类型的对象。表 3-1 列出了几个 string 类型常用的构造函数。当没有明确指定对象初始化式时，系统将使用默认构造函数（2.3.4 节）。

表 3-1 几种初始化 String 对象的方式

string s1;	默认构造函数，s1 为空串
string s2(s1);	将 s2 初始化为 s1 的一个副本
string s3("value");	将 s3 初始化为一个字符串字面值副本
string s4(n, 'c');	将 s4 初始化为字符 'c' 的 n 个副本

警告：标准库 string 类型和字符串字面值

因为历史原因以及为了与 C 语言兼容，字符串字面值与标准库 string 类型不是同一种类型。这一点很容易引起混乱，编程时一定要注意区分字符串字面值和 string 数据类型的使用，这很重要。

习题

习题 3.2 什么是默认构造函数？

习题 3.3 列举出三种初始化 String 对象的方法。

习题 3.4 s 和 s2 的值分别是什么？

```
string s;
int main() {
    string s2;
}
```

3.2.2 String 对象的读写

我们已在第1章学习了用 `iostream` 标准库来读写内置类型的值，如 `int`，`double` 等。同样地，也可以用 `iostream` 和 `string` 标准库，使用标准输入输出操作符来读写 `string` 对象：

```
// Note: #include and using declarations must be added to compile this code
int main()
{
    string s;           // empty string
    cin >> s;           // read whitespace-separated string into s
    cout << s << endl; // write s to the output
    return 0;
}
```

以上程序首先定义命名为 `s` 的 `string`，第二行代码：

```
cin >> s;           // read whitespace-separated string into s
```

从标准输入读取 `string`，并将读入的串存储在 `s` 中。`string` 类型的输入操作符：

- 读取并忽略开头所有的空白字符（如空格，换行符，制表符）。
- 读取字符直至再次遇到空白字符，读取终止。

81

因此，如果输入到程序的是 “ **Hello World!** ”（注意到开头和结尾的空格），则屏幕上将输出 “**Hello**”，而不含任何空格。

输入和输出操作的行为与内置类型操作符基本类似。尤其是，这些操作符返回左操作数作为运算结果。因此，我们可以把多个读操作或多个写操作放在一起：

```
string s1, s2;
cin >> s1 >> s2; // read first input into s1, second into s2
cout << s1 << s2 << endl; // write both strings
```

如果给定和上一个程序同样的输入，则输出的结果将是：

HelloWorld!



对于上例，编译时必须加上 `#include` 来标示 `iostream` 和 `string` 标准库，以及给出用到的所有标准库中的名字（如 `string`, `cin`, `cout`, `endl`）的 `using` 声明。

从本例开始的程序均假设程序中所有必须的 `#include` 和 `using` 声明已给出。

1. 读入未知数目的 `string` 对象

和内置类型的输入操作符一样，`string` 的输入操作符也会返回所读的数据流。因此，可以把输入操作作为判断条件，这与我们在 1.4.4 节读取整型数据的程序做法是一样的。下面的程序将从标准输入读取一组 `string` 对象，然后在标准输出上逐行输出：

```
int main()
{
    string word;
    // read until end-of-file, writing each word to a new line
    while (cin >> word)
        cout << word << endl;
    return 0;
}
```

上例中，用输入操作符来读取 `string` 对象。该操作符返回所读的 `istream` 对象，并在读取结束后，作为 `while` 的判断条件。如果输入流是有效的，即还未到达文件尾且未遇到无效输入，则执行 `while` 循环体，并将读取到的字符串输出到标准输出。如果到达了文件尾，则跳出 `while` 循环。

2. 用 `getline` 读取整行文本

82

另外还有一个有用的 `string` IO 操作：**`getline`**。这个函数接受两个参数：一个输入流对象和一个 `string` 对象。`getline` 函数从输入流的下一行读取，并保存读取的内容到 `string` 中，但不包括换行符。和输入操作符不一样的是，`getline` 并不忽略行开头的换行符。只要 `getline` 遇到换行符，即便它是输入的第一个字符，`getline` 也将停止读入并返回。如果第一个字符就是换行符，则 `string` 参数将被置为空 `string`。

`getline` 函数将 `istream` 参数作为返回值，和输入操作符一样也把它用作判断条件。例如，重写前面那段程序，把每行输出一个单词改为每次输出一行文本：

```
int main()
{
    string line;
    // read line at time until end-of-file
    while (getline(cin, line))
        cout << line << endl;
    return 0;
}
```

由于 `line` 不含换行符，若要逐行输出需要自行添加。照常，我们用 `endl` 来输出一个换行符并刷新输出缓冲区。



由于 `getline` 函数返回时丢弃换行符，换行符将不会存储在 `string` 对象中。

习题

习题 3.5 编写程序实现从标准输入每次读入一行文本。然后改写程序，每次读入一个单词。

习题 3.6 解释 `string` 类型的输入操作符和 `getline` 函数分别如何处理空白字符。

3.2.3 `string` 对象的操作

表 3-2 列出了常用的 `string` 操作。

表 3-2 <code>string</code> 操作	
<code>s.empty()</code>	如果 <code>s</code> 为空串，则返回 <code>true</code> ，否则返回 <code>false</code>
<code>s.size()</code>	返回 <code>s</code> 中字符的个数
<code>s[n]</code>	返回 <code>s</code> 中位置为 <code>n</code> 的字符，位置从 0 开始计数
<code>s1 + s2</code>	把 <code>s1</code> 和 <code>s2</code> 连接成 一个新字符串，返回新生成的字符串
<code>s1 = s2</code>	把 <code>s1</code> 内容替换为 <code>s2</code> 的副本
<code>v1 == v2</code>	比较 <code>v1</code> 与 <code>v2</code> 的内容，相等则返回 <code>true</code> ，否则返回 <code>false</code>
<code>!=</code> , <code><</code> , <code><=</code> , <code>></code> 和 <code>>=</code>	保持这些操作符惯有的含义

1. string 的 size 和 empty 操作

string 对象的长度指的是 string 对象中字符的个数，可以通过 size 操作获取：

```
int main()
{
    string st("The expense of spirit\n");
    cout << "The size of " << st << "is " << st.size()
        << " characters, including the newline" << endl;
    return 0;
}
```

83

编译并运行这个程序，得到的结果为：

```
The size of The expense of spirit
is 22 characters, including the newline
```

了解 string 对象是否为空是有用的。一种方法是将 size 与 0 进行比较：

```
if (st.size() == 0)
    // ok: empty
```

本例中，程序员并不需要知道 string 对象中有多少个字符，只想知道 size 是否为 0。用 string 的成员函数 empty() 可以更直接地回答这个问题：

```
if (st.empty())
    // ok: empty
```

empty() 成员函数将返回 bool 值(2.1 节)，如果 string 对象为空则返回 true，否则返回 false。

2. string::size_type 类型

从逻辑上来讲，size() 成员函数似乎应该返回整型数值，或如 2.2 节“建议”中所述的无符号整数。但事实上，size 操作返回的是 string::size_type 类型的值。我们需要对这种类型做一些解释。

string 类类型和许多其他库类型都定义了一些配套类型 (companion type)。通过这些配套类型，库类型的使用就能与机器无关 (machine-independent)。size_type 就是这些配套类型中的一种。它定义为与 unsigned 型 (unsigned int 或 unsigned long) 具有相同的含义，而且可以保证足够大能够存储任意 string 对象的长度。为了使用由 string 类型定义的 size_type 类型，程序员必须加上作用域操作符来说明所使用的 size_type 类型是由 string 类定义的。

84

 **最佳实践** 任何存储 string 的 size 操作结果的变量必须为 string::size_type 类型。特别重要的是，不要把 size 的返回值赋给一个 int 变量。

虽然我们不知道 string::size_type 的确切类型，但可以知道它是 unsigned 型(2.1.1 节)。对于任意一种给定的数据类型，它的 unsigned 型所能表示的最大正数值比对应的 signed 型要大一倍。这个事实表明 size_type 存储的 string 长度是 int 所能存储的两倍。

使用 int 变量的另一个问题是，有些机器上 int 变量的表示范围太小，甚至无法存储实际并不长的 string 对象。如有 16 位 int 型的机器上，int 类型变量最大只能表示 32767 个字符的 string 对象。而能容纳一个文件内容的 string 对象轻易就会超过这个数字。因此，

为了避免溢出，保存一个 string 对象 size 的最安全的方法就是使用标准库类型 `string::size_type`。

3. string 关系操作符

string 类定义了几种关系操作符用来比较两个 string 值的大小。这些操作符实际上是比较每个 string 对象的字符。



string 对象比较操作是区分大小写的，即同一个字符的大小写形式被认为是两个不同的字符。在多数计算机上，大写的字母位于小写字母之前：任何一个大写字母都小于任意的小写字母。

`==`操作符比较两个 string 对象，如果它们相等，则返回 `true`。两个 string 对象相等是指它们的长度相同，且含有相同的字符。标准库还定义了 `!=`操作符来测试两个 string 对象是否不等。

关系操作符 `<`, `<=`, `>`, `>=` 分别用于测试一个 string 对象是否小于、小于或等于、大于、大于或等于另一个 string 对象：

```
string big = "big", small = "small";
string s1 = big;      // s1 is a copy of big
if (big == small)     // false
    // ...
if (big <= s1)         // true, they're equal, so big is less than or equal to s1
    // ...
```

关系操作符比较两个 string 对象时采用了和（大小写敏感的）字典排序相同的策略：

- 如果两个 string 对象长度不同，且短的 string 对象与长的 string 对象的前面部分相匹配，则短的 string 对象小于长的 string 对象。
- 如果两个 string 对象的字符不同，则比较第一个不匹配的字符。

85

举例来说，给定 string 对象：

```
string substr = "Hello";
string phrase = "Hello World";
string slang = "Hiya";
```

则 `substr` 小于 `phrase`，而 `slang` 则大于 `substr` 或 `phrase`。

4. string 对象的赋值

总体上说，标准库类型尽量设计得和基本数据类型一样方便易用。因此，大多数库类型支持赋值操作。对 string 对象来说，可以把一个 string 对象赋值给另一个 string 对象：

```
// st1 is an empty string, st2 is a copy of the literal
string st1, st2 = "The expense of spirit";
st1 = st2; // replace st1 by a copy of st2
```

赋值操作后，`st1` 就包含了 `st2` 串所有字符的一个副本。

大多数 string 库类型的赋值等操作的实现都会遇到一些效率上的问题，但值得注意的是，从概念上讲，赋值操作确实需要做一些工作。它必须先把 `st1` 占用的相关内存释放掉，然后再分配给 `st1` 足够存放 `st2` 副本的内存空间，最后把 `st2` 中的所有字符复制到新分配的内存空间。

5. 两个 **string** 对象相加

string 对象的加法被定义为连接 (concatenation)。也就是说，两个 (或多个) **string** 对象可以通过使用加操作符+或者复合赋值操作符+= (1.4.1 节) 连接起来。给定两个 **string** 对象：

```
string s1("hello, ");
string s2("world\n");
```

下面把两个 **string** 对象连接起来产生第三个 **string** 对象：

```
string s3 = s1 + s2;    // s3 is hello, world\n
```

如果要把 **s2** 直接追加到 **s1** 的末尾，可以使用+=操作符：

```
s1 += s2;    // equivalent to s1 = s1 + s2
```

6. 和字符串字面值的连接

上面的字符串对象 **s1** 和 **s2** 直接包含了标点符号。也可以通过将 **string** 对象和字符串字面值混合连接得到同样的结果：

```
string s1("hello");
string s2("world");
string s3 = s1 + ", " + s2 + "\n";
```

86

当进行 **string** 对象和字符串字面值混合连接操作时，+操作符的左右操作数必须至少有一个是 **string** 类型的：

```
string s1 = "hello";    // no punctuation
string s2 = "world";
string s3 = s1 + ", ";    // ok: adding a string and a literal
string s4 = "hello" + ", ";    // error: no string operand
string s5 = s1 + ", " + "world";    // ok: each + has string operand
string s6 = "hello" + ", " + s2;    // error: can't add string literals
```

s3 和 **s4** 的初始化只用了一个单独的操作。在这些例子中，很容易判断 **s3** 的初始化是合法的：把一个 **string** 对象和一个字符串字面值连接起来。而 **s4** 的初始化试图将两个字符串字面值相加，因此是非法的。

s5 的初始化方法显得有点不可思议，但这种用法和标准输入输出的串联效果是一样的 (1.2 节)。本例中，**string** 标准库定义加操作返回一个 **string** 对象。这样，在对 **s5** 进行初始化时，子表达式 **s1 + ", "** 将返回一个新 **string** 对象，后者再和字面值 “world\n” 连接。整个初始化过程可以改写为：

```
string tmp = s1 + ", ";    // ok: + has a string operand
s5 = tmp + "world";    // ok: + has a string operand
```

而 **s6** 的初始化是非法的。依次来看每个子表达式，则第一个子表达式试图把两个字符串字面值连接起来。这是不允许的，因此这个语句是错误的。

7. 从 **string** 对象获取字符

string 类型通过下标操作符([])来访问 **string** 对象中的单个字符。下标操作符需要取一个 **size_type** 类型的值，来标明要访问字符的位置。这个下标中的值通常被称为“下标”或“索引 (index)”。



string 对象的下标从 0 开始。如果 s 是一个 string 对象且 s 不空，则 s[0] 就是字符串的第一个字符，s[1] 就表示第二个字符（如果有的话），而 s[s.size()-1] 则表示 s 的最后一个字符。

引用下标时如果超出下标作用范围就会引起溢出错误。

可用下标操作符分别取出 string 对象的每个字符，分行输出：

```
string str("some string");
for (string::size_type ix = 0; ix != str.size(); ++ix)
    cout << str[ix] << endl;
```

87 每次通过循环，就从 str 对象中读取下一个字符，输出该字符并换行。

8. 下标操作可用作左值

前面说过，变量是左值（2.3.1 节），且赋值操作的左操作数必须是左值。和变量一样，string 对象的下标操作返回值也是左值。因此，下标操作可以放于赋值操作符的左边或右边。通过下面循环把 str 对象的每一个字符置为 '*'：

```
for (string::size_type ix = 0; ix != str.size(); ++ix)
    str[ix] = '*';
```

9. 计算下标值

任何可产生整型值的表达式都可用作下标操作符的索引。例如，假设 someval 和 someotherval 是两个整型对象，可以这样写：

```
str[someotherval * someval] = someval;
```

虽然任何整型数值都可作为索引，但索引的实际数据类型却是 unsigned 类型 string::size_type。



前面讲过，应该用 string::size_type 类型的变量接受 size 函数的返回值。在定义用作索引的变量时，出于同样的道理，string 对象的索引变量最好也用 string::size_type 类型。

在使用下标索引 string 对象时，必须保证索引值“在上下界范围内”。“在上下界范围内”就是指索引值是一个赋值为 size_type 类型的值，其取值范围在 0 到 string 对象长度减 1 之间。使用 string::size_type 类型或其他 unsigned 类型作为索引，就可以保证索引值不小于 0。只要索引值是 unsigned 类型，就只需要检测它是否小于 string 对象的长度。



标准库不要求检查索引值，所用索引的下标越界是没有定义的，这样往往会导致严重的运行时错误。

3.2.4 string 对象中字符的处理

我们经常要对 string 对象中的单个字符进行处理，例如，通常要知道某个特殊字符是否为空白字符、字母或数字。表 3-3 列出了各种字符操作函数，适用于 string 对象的字符（或其他任何 char 值）。这些函数都在 ctype 头文件中定义。

表 3-3 ctype 定义的函数

isalnum(c)	如果 c 是字母或数字, 则为 true。
isalpha(c)	如果 c 是字母, 则为 true。
iscntrl(c)	如果 c 是控制字符, 则为 true。
isdigit(c)	如果 c 是数字, 则为 true。
isgraph(c)	如果 c 不是空格, 但可打印, 则为 true。
islower(c)	如果 c 是小写字母, 则为 true。
isprint(c)	如果 c 是可打印的字符, 则为 true。
ispunct(c)	如果 c 是标点符号, 则为 true。
isspace(c)	如果 c 是空白字符, 则为 true。
isupper(c)	如果 c 是大写字母, 则为 true。
isxdigit(c)	如果 c 是十六进制数, 则为 true。
tolower(c)	如果 c 是大写字母, 则返回其小写字母形式, 否则直接返回 c。
toupper(c)	如果 c 是小写字母, 则返回其大写字母形式, 否则直接返回 c。

表 3-3 中的大部分函数是测试一个给定的字符是否符合条件, 并返回一个 int 值作为真值。如果测试失败, 则该函数返回 0, 否则返回一个 (无意义的) 非 0 值, 表示被测字符符合条件。

表中的这些函数, 可打印的字符是指那些可以显式表示的字符。空白字符则是空格、制表符、垂直制表符、回车符、换行符和进纸符中的任意一种; 标点符号则是除了数字、字母或 (可打印的) 空白字符 (如空格) 以外的其他可打印字符。

88

这里给出一个例子, 运用这些函数输出一给定 string 对象中标点符号的个数:

```
string s("Hello World!!!");
string::size_type punct_cnt = 0;
// count number of punctuation characters in s
for (string::size_type index = 0; index != s.size(); ++index)
    if (ispunct(s[index]))
        ++punct_cnt;
cout << punct_cnt
    << " punctuation characters in " << s << endl;
```

这个程序的输出结果是:

3 punctuation characters in Hello World!!!

和返回真值的函数不同的是, tolower 和 toupper 函数返回的是字符, 返回实参字符本身或返回该字符相应的大小写字符。我们可以用 tolower 函数把 string 对象 s 中的字母改为小写字母, 程序如下:

```
// convert s to lowercase
for (string::size_type index = 0; index != s.size(); ++index)
    s[index] = tolower(s[index]);
cout << s << endl;
```

得到的结果为:

hello world!!!

89

建议：采用 C 标准库头文件的 C++ 版本

C++ 标准库除了定义了一些特定于 C++ 的设施外，还包括 C 标准库。C++ 中的头文件 `cctype` 其实就是利用了 C 标准库函数，这些库函数就定义在 C 标准库的 `ctype.h` 头文件中。

C 标准库头文件命名形式为 `name.h`，而 C++ 版本则命名为 `cname`，少了后缀 `.h` 而在头文件名前加了 `c`。`c` 表示这个头文件源自 C 标准库。因此，`cctype` 与 `ctype.h` 文件的内容是一样的，只是采用了更适合 C++ 程序的形式。特别地，`cname` 头文件中定义的名字都定义在命名空间 `std` 内，而 `.h` 版本中的名字却不是这样。

通常，C++ 程序中应采用 `cname` 这种头文件的版本，而不采用 `name.h` 版本，这样，标准库中的名字在命名空间 `std` 中保持一致。使用 `.h` 版本会给程序员带来负担，因为他们必须记得哪些标准库名字是从 C 继承来的，而哪些是 C++ 所特有的。

习题

习题 3.7 编一个程序读入两个 `string` 对象，测试它们是否相等。若不相等，则指出两个中哪个较大。接着，改写程序测试它们的长度是否相等，若不相等指出哪个较长。

习题 3.8 编一个程序，从标准输入读取多个 `string` 对象，把它们连接起来存放到一个更大的 `string` 对象中。并输出连接后的 `string` 对象。接着，改写程序，将连接后相邻 `string` 对象以空格隔开。

习题 3.9 下列程序实现什么功能？实现合法吗？如果不合法，说明理由。

```
string s;
cout << s[0] << endl;
```

习题 3.10 编一个程序，从 `string` 对象中去掉标点符号。要求输入到程序的字符串必须含有标点符号，输出结果则是去掉标点符号后的 `string` 对象。

3.3 标准库 `vector` 类型

`vector` 是同一种类型的对象的集合，每个对象都有一个对应的整数索引值。和 `string` 对象一样，标准库将负责管理与存储元素相关的内存。我们把 `vector` 称为容器，是因为它可以包含其他对象。一个容器中的所有对象都必须是同一种类型的。我们将在第 9 章更详细地介绍容器。

使用 `vector` 之前，必须包含相应的头文件。本书给出的例子，都是假设已作了相应的 `using` 声明：

```
#include <vector>
using std::vector;
```

90

`vector` 是一个类模板 (class template)。使用模板可以编写一个类定义或函数定义，而用于多个不同的数据类型。因此，我们可以定义保存 `string` 对象的 `vector`，或保存 `int` 值的 `vector`，又或是保存自定义的类类型对象（如 `Sales_item` 对象）的 `vector`。将在第 16 章介绍如何定义程序员自己的类模板。幸运的是，使用类模板时只需要简单了解类模板是如何定义的就

可以了。

声明从类模板产生的某种类型的对象，需要提供附加信息，信息的种类取决于模板。以 `vector` 为例，必须说明 `vector` 保存何种对象的类型，通过将类型放在类模板名称后面的尖括号中来指定类型：

```
vector<int> ivec;           // ivec holds objects of type int
vector<Sales_item> Sales_vec; // holds Sales_items
```

和其他变量定义一样，定义 `vector` 对象要指定类型和一个变量的列表。上面的第一个定义，类型是 `vector<int>`，该类型即是含有若干 `int` 类型对象的 `vector`，变量名为 `ivec`。第二个定义的变量名是 `Sales_vec`，它所保存的元素是 `Sales_item` 类型的对象。



`vector` 不是一种数据类型，而只是一个类模板，用来定义任意多种数据类型。`vector` 类型的每一种都指定了其保存元素的类型。因此，`vector<int>` 和 `vector<string>` 都是数据类型。

3.3.1 `vector` 对象的定义和初始化

`vector` 类定义了好几种构造函数（2.3.3 节），用来定义和初始化 `vector` 对象。表 3-4 列出了这些构造函数。

表 3-4 几种初始化 `vector` 对象的方式

<code>vector<T> v1;</code>	<code>vector</code> 保存类型为 <code>T</code> 的对象。默认构造函数 <code>v1</code> 为空。
<code>vector<T> v2(v1);</code>	<code>v2</code> 是 <code>v1</code> 的一个副本。
<code>vector<T> v3(n, i);</code>	<code>v3</code> 包含 <code>n</code> 个值为 <code>i</code> 的元素。
<code>vector<T> v4(n);</code>	<code>v4</code> 含有值初始化的元素的 <code>n</code> 个副本。

1. 创建确定个数的元素

若要创建非空的 `vector` 对象，必须给出初始化元素的值。当把一个 `vector` 对象复制到另一个 `vector` 对象时，新复制的 `vector` 中每一个元素都初始化为原 `vector` 中相应元素的副本。但这两个 `vector` 对象必须保存同一种元素类型：

```
vector<int> ivec1;           // ivec1 holds objects of type int
vector<int> ivec2(ivec1);    // ok: copy elements of ivec1 into ivec2
vector<string> svec(ivec1);  // error: svec holds strings, not ints
```

91

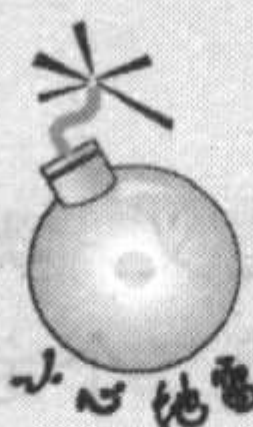
可以用元素个数和元素值对 `vector` 对象进行初始化。构造函数用元素个数来决定 `vector` 对象保存元素的个数，元素值指定每个元素的初始值：

```
vector<int> ivec4(10, -1);    // 10 elements, each initialized to -1
vector<string> svec(10, "hi!"); // 10 strings, each initialized to "hi!"
```

关键概念：`vector` 对象动态增长

`vector` 对象（以及其他标准库容器对象）的重要属性就在于可以在运行时高效地添加元素。因为 `vector` 增长的效率高，在元素值已知的情况下，最好是动态地添加元素。

正如第 4 章将介绍的, 这种增长方式不同于 C 语言中的内置数据类型, 也不同于大多数其他编程语言的数据类型。具体而言, 如果读者习惯了 C 或 Java 的风格, 由于 vector 元素连续存储, 可能希望最好是预先分配合适的空间。但事实上, 为了达到连续性, C++ 的做法恰好相反, 具体原因将在第 9 章探讨。



虽然可以对给定元素个数的 vector 对象预先分配内存, 但更有效的方法是先初始化一个空 vector 对象, 然后再动态地增加元素 (我们随后将学习如何进行这样的操作)。

2. 值初始化

如果没有指定元素的初始化式, 那么标准库将自行提供一个元素初始值进行值初始化 (value initialization)。这个由库生成的初始值将用来初始化容器中的每个元素, 具体值为何, 取决于存储在 vector 中元素的数据类型。

如果 vector 保存内置类型 (如 int 类型) 的元素, 那么标准库将用 0 值创建元素初始化式:

```
vector<int> fvec(10); // 10 elements, each initialized to 0
```

如果 vector 保存的是含有构造函数的类类型 (如 string) 的元素, 标准库将用该类型的默认构造函数创建元素初始化式:

```
vector<string> svec(10); // 10 elements, each an empty string
```



第 12 章将介绍一些有自定义构造函数但没有默认构造函数的类, 在初始化这种类型的 vector 对象时, 程序员就不能仅提供元素个数, 还需要提供元素初始值。

还有第三种可能性: 元素类型可能是没有定义任何构造函数的类类型。这种情况下, 标准库仍产生一个带初始值的对象, 这个对象的每个成员进行了值初始化。

92

习题

习题 3.11 下面哪些 vector 定义不正确?

- (a) `vector< vector<int> > ivec;`
- (b) `vector<string> svec = ivec;`
- (c) `vector<string> svec(10, "null");`

习题 3.12 下列每个 vector 对象中元素个数是多少? 各元素的值是什么?

- (a) `vector<int> ivec1;`
- (b) `vector<int> ivec2(10);`
- (c) `vector<int> ivec3(10, 42);`
- (d) `vector<string> svec1;`
- (e) `vector<string> svec2(10);`
- (f) `vector<string> svec3(10, "hello");`

3.3.2 vector 对象的操作

vector 标准库提供了许多类似于 string 对象的操作，表 3-5 列出了几种最重要的 vector 操作。

表 3-5 vector 操作	
v.empty()	如果 v 为空，则返回 true，否则返回 false。
v.size()	返回 v 中元素的个数。
v.push_back(t)	在 v 的末尾增加一个值为 t 的元素。
v[n]	返回 v 中位置为 n 的元素。
v1 = v2	把 v1 的元素替换为 v2 中元素的副本。
v1 == v2	如果 v1 与 v2 相等，则返回 true。
!=, <, <=, >, >=	保持这些操作符惯有的含义。

1. vector 对象的 size

empty 和 size 操作类似于 string 类型的相关操作 (3.2.3 节)。成员函数 size 返回相应 vector 类定义的 size_type 的值。



使用 size_type 类型时，必须指出该类型是在哪里定义的。vector 类型总是包括 vector 的元素类型：

```
vector<int>::size_type    // ok
vector::size_type         // error
```

93

2. 向 vector 添加元素

push_back() 操作接受一个元素值，并将它作为一个新的元素添加到 vector 对象的后面，也就是“插入 (push)”到 vector 对象的“后面 (back)”：

```
// read words from the standard input and store them as elements in a vector
string word;
vector<string> text;           // empty vector
while (cin >> word) {
    text.push_back(word);     // append word to text
}
```

该循环从标准输入读取一系列 string 对象，逐一追加到 vector 对象的后面。首先定义一个空的 vector 对象 text。每循环一次就添加一个新元素到 vector 对象，并将从输入读取的 word 值赋予该元素。当循环结束时，text 就包含了所有读入的元素。

3. vector 的下标操作

vector 中的对象是没有命名的，可以按 vector 中对象的位置来访问它们。通常使用下标操作符来获取元素。vector 的下标操作类似于 string 类型的下标操作 (3.2.3 节)。

vector 的下标操作符接受一个值，并返回 vector 中该对应位置的元素。vector 元素的位置从 0 开始。下例使用 for 循环把 vector 中的每个元素值都重置为 0：

```
// reset the elements in the vector to zero
for (vector<int>::size_type ix = 0; ix != ivec.size(); ++ix)
```

```
ivec[ix] = 0;
```

和 `string` 类型的下标操作符一样, `vector` 下标操作的结果为左值, 因此可以像循环体中所做的那样实现写入。另外, 和 `string` 对象的下标操作类似, 这里用 `size_type` 类型作为 `vector` 下标的类型。



在上例中, 即使 `ivec` 为空, `for` 循环也会正确执行。`ivec` 为空则调用 `size` 返回 0, 并且 `for` 中的测试比较 `ix` 和 0。第一次循环时, 由于 `ix` 本身就是 0, 则条件测试失败, `for` 循环体一次也不执行。

关键概念: 安全的泛型编程

习惯于 C 或 Java 编程的 C++ 程序员可能会觉得难以理解, `for` 循环的判断条件用 `!=` 而不是用 `<` 来测试 `vector` 下标值是否越界。C 程序员难以理解的还有, 上例中没有在 `for` 循环之前就调用 `size` 成员函数并保存其返回的值, 而是在 `for` 语句头中调用 `size` 成员函数。

C++ 程序员习惯于优先选用 `!=` 而不是 `<` 来编写循环判断条件。在上例中, 选用或不用某种操作符并没有特别的取舍理由。学习完本书第二部分的泛型编程后, 你将会明白这种习惯的合理性。

调用 `size` 成员函数而不保存它返回的值, 在这个例子中同样不是必需的, 但这反映了一种良好的编程习惯。在 C++ 中, 有些数据结构 (如 `vector`) 可以动态增长。上例中循环仅需要读取元素, 而不需要增加新的元素。但是, 循环可以容易地增加新元素, 如果确实增加了新元素的话, 那么测试已保存的 `size` 值作为循环的结束条件就会有问题, 因为没有将新加入的元素计算在内。所以我们倾向于在每次循环中测试 `size` 的当前值, 而不是在进入循环前, 存储 `size` 值的副本。

我们将在第 7 章学习到, C++ 中有些函数可以声明为内联 (`inline`) 函数。编译器遇到内联函数时就会直接扩展相应代码, 而不是进行实际的函数调用。像 `size` 这样的小库函数几乎都定义为内联函数, 所以每次循环过程中调用它的运行时代价是比较小的。

4. 下标操作不添加元素

初学 C++ 的程序员可能会认为 `vector` 的下标操作可以添加元素, 其实不然:

```
vector<int> ivec;    // empty vector
for (vector<int>::size_type ix = 0; ix != 10; ++ix)
    ivec[ix] = ix;    // disaster: ivec has no elements
```

94

上述程序试图在 `ivec` 中插入 10 个新元素, 元素值依次为 0 到 9 的整数。但是, 这里 `ivec` 是空的 `vector` 对象, 而且下标只能用于获取已存在的元素。

这个循环的正确写法应该是:

```
for (vector<int>::size_type ix = 0; ix != 10; ++ix)
    ivec.push_back(ix);    // ok: adds new element with value ix
```



必须是已存在的元素才能用下标操作符进行索引。通过下标操作进行赋值时，不会添加任何元素。

警告：仅能对确知已存在的元素进行下标操作

对于下标操作符（[]操作符）的使用有一点非常重要，就是仅能提取确实已存在的元素，例如：

```
vector<int> ivec;           // empty vector
cout << ivec[0];           // Error: ivec has no elements!

vector<int> ivec2(10);      // vector with 10 elements.
cout << ivec2[10];         // Error: ivec2 has elements 0...9
```

试图获取不存在的元素必然产生运行时错误。和大多数同类错误一样，不能确保执行过程可以捕捉到这类错误，运行程序的结果是不确定的。由于取不存在的元素的结果标准没有定义，因而不同的编译器实现会导致不同的结果，但程序运行时几乎肯定会以某种有趣的方式失败。

本警告适用于任何使用下标操作的时候，如 string 类型的下标操作，以及将要简要介绍的内置数组的下标操作。

不幸的是，试图对不存在的元素进行下标操作是程序设计过程中经常会犯的严重错误。所谓的“缓冲区溢出”错误就是对不存在的元素进行下标操作的结果。这样的缺陷往往导致 PC 机和其他应用中最常见的安全问题。

习题

习题 3.13 读一组整数到 vector 对象，计算并输出每对相邻元素的和。如果读入元素个数为奇数，则提示用户最后一个元素没有求和，并输出其值。然后修改程序：头尾元素两两配对（第一个和最后一个，第二个和倒数第二个，以此类推），计算每对元素的和，并输出。

习题 3.14 读入一段文本到 vector 对象，每个单词存储为 vector 中的一个元素。把 vector 对象中每个单词转化为大写字母。输出 vector 对象中转化后的元素，每八个单词为一行输出。

习题 3.15 下面程序合法吗？如果不合法，如何更正？

```
vector<int> ivec;
ivec[0] = 42;
```

习题 3.16 列出三种定义 vector 对象的方法，给定 10 个元素，每个元素值为 42。指出是否还有更好的实现方法，并说明为什么。

3.4 迭代器简介

除了使用下标来访问 vector 对象的元素外，标准库还提供了另一种访问元素的方法：使用

迭代器 (iterator)。迭代器是一种检查容器内元素并遍历元素的数据类型。

标准库为每一种标准容器 (包括 `vector`) 定义了一种迭代器类型。迭代器类型提供了比下标操作更通用化的方法: 所有的标准库容器都定义了相应的迭代器类型, 而只有少数的容器支持下标操作。因为迭代器对所有的容器都适用, 现代 C++ 程序更倾向于使用迭代器而不是下标操作访问容器元素, 即使对支持下标操作的 `vector` 类型也是这样。

第 11 章将详细讨论迭代器的工作原理, 但使用迭代器并不需要完全了解它复杂的实现细节。

1. 容器的 `iterator` 类型

每种容器类型都定义了自己的迭代器类型, 如 `vector`:

```
vector<int>::iterator iter;
```

这条语句定义了一个名为 `iter` 的变量, 它的数据类型是由 `vector<int>` 定义的 `iterator` 类型。每个标准库容器类型都定义了一个名为 `iterator` 的成员, 这里的 `iterator` 与迭代器实际类型的含义相同。

术语: 迭代器和迭代器类型

程序员首次遇到有关迭代器的术语时可能会困惑不解, 原因之一是由于同一个术语 `iterator` 往往表示两个不同的事物。一般意义上指的是迭代器的概念; 而具体而言时指的则是由容器定义的具体的 `iterator` 类型, 如 `vector<int>`。

重点要理解的是, 有许多用作迭代器的类型, 这些类型在概念上是相关的。若一种类型支持一组确定的操作 (这些操作可用来遍历容器内的元素, 并访问这些元素的值), 我们就称这种类型为迭代器。

各容器类都定义了自己的 `iterator` 类型, 用于访问容器内的元素。换句话说, 每个容器都定义了一个名为 `iterator` 的类型, 而这种类型支持 (概念上的) 迭代器的各种操作。

2. `begin` 和 `end` 操作

每种容器都定义了一对命名为 `begin` 和 `end` 的函数, 用于返回迭代器。如果容器中有元素的话, 由 `begin` 返回的迭代器指向第一个元素:

```
vector<int>::iterator iter = ivec.begin();
```

上述语句把 `iter` 初始化为由名为 `begin` 的 `vector` 操作返回的值。假设 `vector` 不空, 初始化后, `iter` 即指该元素为 `ivec[0]`。

由 `end` 操作返回的迭代器指向 `vector` 的“末端元素的下一个”。通常称为超出末端迭代器 (off-the-end iterator), 表明它指向了一个不存在的元素。如果 `vector` 为空, `begin` 返回的迭代器与 `end` 返回的迭代器相同。



由 `end` 操作返回的迭代器并不指向 `vector` 中任何实际的元素, 相反, 它只是起一个哨兵 (sentinel) 的作用, 表示我们已处理完 `vector` 中所有元素。