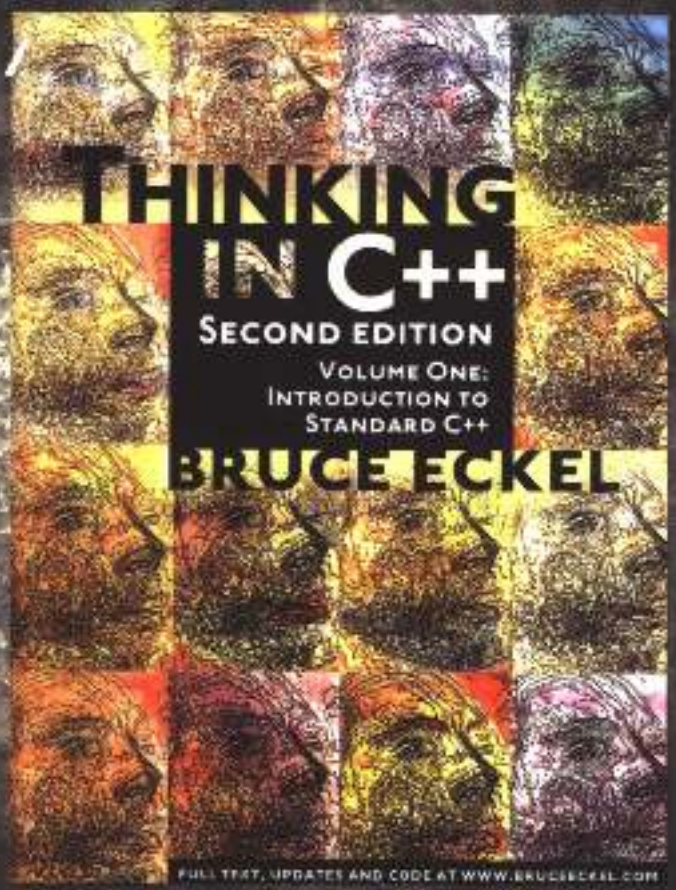
计 算 机 科 学 丛 书

第2版

C++编程思想

第1卷：标准C++导引

（美）Bruce Eckel 著　刘宗田　袁兆山　潘秋菱　等译



Thinking in C++

Second Edition

Volume One: Introduction to Standard C++

机械工业出版社
China Machine Press

Prentice Hall

《C++编程思想》（第1版）荣获“软件开发”杂志评选的1996年图书震撼大奖（Jolt Award）。

本书的内容、讲授方法、选用例子和跟随的练习，别具特色。作者Bruce Eckel不是按传统的方法讲解C++的概念和编程方法，而是根据他自己是怎样学习C++的亲身体会，根据他在多年教学实践中发现的问题，用一些非常简单的例子和简练的叙述，例明了在学习C++中非常容易混淆的概念，特别是，他经常通过例子引导读者从C++编译实现的汇编代码的角度反回审视C++的语法和语义，常常使读者有“心有灵犀一点通”的奇特效果。

本书第1版的中文版自2000年年初推出以来，经久不衰，获得了读者的高度评价和广泛欢迎，充分说明了这本书在中国读者心目中的地位。

从章节上看，第2版相比第1版，主要有以下改变。首先，增加了两章：“对象的创建与使用”和“C++中的C”。前者与“对象导言”实际上是第1版“对象的演化”一章的彻底重写，增加了近几年面向对象方法和编程方法的最新研究与实践的有效成果。后者的添加使不熟悉C的读者直接使用这本书成为可能。其次，删去了四章“输入输出流介绍”、“多重继承”、“异常处理”和“运行时类型识别”。这四章属于C++中较复杂的主题。作者将它们连同C++标准完成后又增加的一些内容放到本书的第2卷中，使得本书的第1卷内容更加集中，供不同程度的读者选择阅读。最后需要补充的是，第2版的改变不仅仅体现在这些章节的调整上，更多的改变是体现在每一章的字里行间中，包括例子的调整和练习的补充，与众不同的精心选材和认真推敲的叙述使得这个广受读者关注的第2版更趋成熟。

本书附带光盘的主要内容是一个“CD ROM课堂讨论”，题目是“Thinking in C: Foundations for Java & C++”，由Chuck Allison编写，包含几小时的讲课录音和幻灯片，其目的是让读者认真思考C语言的基础，其重点放在使读者能够转移到C++或Java高级语言所必需的知识上。读者可以在学习本书之前阅读这个光盘上的预备知识。

本书的英文原文、修订、近代码和补充内容可以在作者的专属网站www.BruceEckel.com上找到。关于本书的第2卷，有兴趣的读者同样可以在www.BruceEckel.com上找到最新的修订版。

作者简介



Bruce Eckel 《Thinking in Java》（Java编程思想）作者，Eckel有20年专业编程经验，并自1986年起教育人们如何编写面向对象程序，足迹遍及全球，成为一位知名的C++教师和顾问，如今兼涉Java。他是C++标准委员会拥有表决权的成员之一，曾经写过另五本面向对象编程书籍，发表过150篇以上的文章，是多本计算机杂志的专栏作家。Eckel开创Software Development Conference的C++、Java、Python等多项研讨活动，拥有应用物理学学士和计算机工程学硕士学位。

ISBN 7-111-10807-6

9 787111 108078

HZ BOOKS

华章图书

网上购书：www.china-pub.com

北京市西城区百万庄南街1号 100037

购书热线：（010）68995259，8006100280（北京地区）

总编信箱：chieleditor@hzbook.com

ISBN 7-111-10807-8/TP·2560

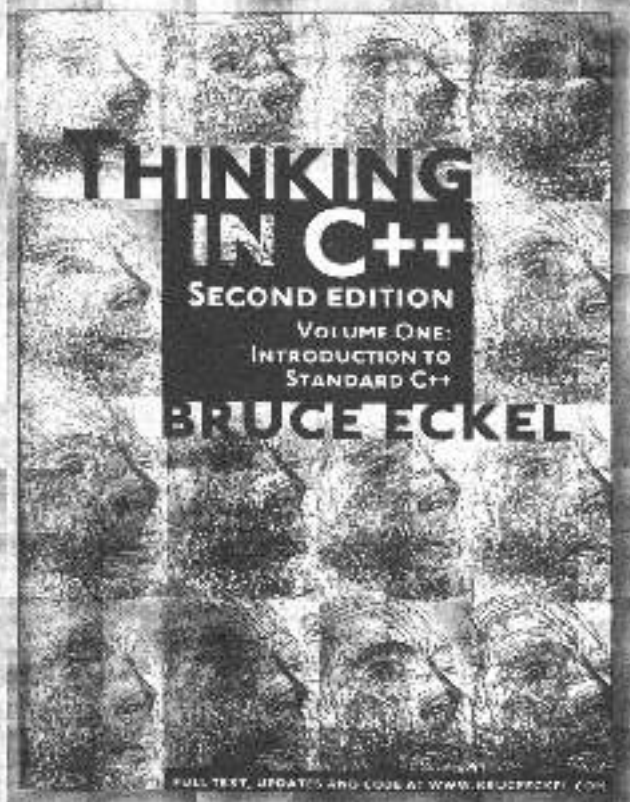
定价：59.00元（附光盘）

计 算 机 科 学 丛 书

第2版

C++编程思想

(美) Bruce Eckel 著　刘宗田　袁兆山　潘秋菱　等译



Thinking in C++

Second Edition

Volume One: Introduction to Standard C++

机械工业出版社
China Machine Press

本书第1版荣获美国“软件开发”杂志评选的1996年图书震撼大奖（Jolt Award），中文版自2000年推出以来，经久不衰，获得了读者的充分肯定和高度评价。

第2版与第1版相比，在章节安排上有以下改变。增加了两章：“对象的创建与使用”和“C++中的C”。前者与“对象导言”实际上是第1版“对象的演化”一章的彻底重写，增加了近几年面向对象方法和编程方法的最新研究与实践的丰硕成果；后者的添加使不熟悉C的读者可以直接使用本书。删去了四章：“输入输出流介绍”、“多重继承”、“异常处理”和“运行时类型识别”，删去的内容均为C++中较复杂的主题，作者将它们连同C++标准完成后增加的一些内容放到本书的第2卷中，使本书的第1卷内容显得更加集中，可以供不同程度的读者选择阅读。需要强调的是，第2版的改变不仅体现在这些章节的调整上，更多的改变体现在每一章的字里行间，包括例子的调整和练习的补充。与众不同的精心选材和认真推敲的叙述使得第2版更趋成熟。

本书是C++领域内一本权威的著作，书中的内容、讲授方法、例子和练习既适合课堂教学，又适合读者自学。无论是高等院校计算机及相关专业的学生，还是业界的从业人员，以及广大的计算机爱好者，都可从阅读本书中获益。

Bruce Eckel: Thinking in C++ (Second Edition), Volume One: Introduction to Standard C++ (ISBN 0-13-979809-9).

Copyright © 2000 by Bruce Eckel, MindView, Inc.
All rights reserved.
Chinese simplified language edition published by China Machine Press.
Copyright © 2002 by China Machine Press.

版权所有，侵权必究。

本书版权登记号：图字：01-2000-4303

图书在版编目(CIP)数据

C++编程思想（第2版）：第1卷：标准C++导引/（美）埃克尔（Eckel, B.）著；刘宗田等译. -北京：机械工业出版社，2002.9
（计算机科学丛书）
书名原文：Thinking in C++ (Second Edition), Volume One: Introduction to Standard C++
ISBN 7-111-10807-8

Ⅰ.C… Ⅱ.①埃… ②刘… Ⅲ.C语言-程序设计 Ⅳ.TP312

中国版本图书馆CIP数据核字（2002）第062698号

机械工业出版社（北京市西城区百万庄大街22号 邮政编码 100037）
责任编辑：姚 蕾
北京牛山世兴印刷厂印刷 · 新华书店北京发行所发行
2002年9月第1版第1次印刷
787mm × 1092mm 1/16 · 32.25印张
印数：0 001-10 000册
定价：59.00元（附光盘）

凡购本书，如有倒页、脱页、缺页，由本社发行部调换

出版者的话

文艺复兴以降，源远流长的科学精神和逐步形成的学术规范，使西方国家在自然科学的各个领域取得了垄断性的优势；也正是这样的传统，使美国在信息技术发展的六十多年间名家辈出、独领风骚。在商业化的进程中，美国的产业界与教育界越来越紧密地结合，计算机学科中的许多泰山北斗同时身处科研和教学的最前线，由此而产生的经典科学著作，不仅擘划了研究的范畴，还揭橥了学术的源变，既遵循学术规范，又自有学者个性，其价值并不会因年月的流逝而减退。

近年，在全球信息化大潮的推动下，我国的计算机产业发展迅猛，对专业人才的需求日益迫切。这对计算机教育界和出版界都既是机遇，也是挑战；而专业教材的建设在教育战略上显得举足轻重。在我国信息技术发展时间较短、从业人员较少的现状下，美国等发达国家在其计算机科学发展的几十年间积淀的经典教材仍有许多值得借鉴之处。因此，引进一批国外优秀计算机教材将对我国计算机教育事业的发展起积极的推动作用，也是与世界接轨、建设真正的世界一流大学的必由之路。

机械工业出版社华章图文信息有限公司较早意识到“出版要为教育服务”。自1998年始，华章公司就将工作重点放在了遴选、移译国外优秀教材上。经过几年的不懈努力，我们与Prentice Hall，Addison-Wesley，McGraw-Hill，Morgan Kaufmann等世界著名出版公司建立了良好的合作关系，从它们现有的数百种教材中甄选出Tanenbaum，Stroustrup，Kernighan，Jim Gray等大师名家的一批经典作品，以“计算机科学丛书”为总称出版，供读者学习、研究及庋藏。大理石纹理的封面，也正体现了这套丛书的品位和格调。

“计算机科学丛书”的出版工作得到了国内外学者的鼎力襄助，国内的专家不仅提供了中肯的选题指导，还不辞劳苦地担任了翻译和审校的工作；而原书的作者也相当关注其作品在中国的传播，有的还专诚为其书的中译本作序。迄今，“计算机科学丛书”已经出版了近百个品种，这些书籍在读者中树立了良好的口碑，并被许多高校采用为正式教材和参考书籍，为进一步推广与发展打下了坚实的基础。

随着学科建设的初步完善和教材改革的逐渐深化，教育界对国外计算机教材的需求和应用都步入一个新的阶段。为此，华章公司将加大引进教材的力度，在“华章教育”的总规划之下出版三个系列的计算机教材：针对本科生的核心课程，剔抉外版菁华而成“国外经典教材”系列；对影印版的教材，则单独开辟出“经典原版书库”；定位在高级教程和专业参考的“计算机科学丛书”还将保持原来的风格，继续出版新的品种。为了保证这三套丛书的权威性，同时也为了更好地为学校和老师们服务，华章公司聘请了中国科学院、北京大学、清华大学、国防科技大学、复旦大学、上海交通大学、南京大学、浙江大学、中国科技大学、哈尔滨工业大学、西安交通大学、中国人民大学、北京航空航天大学、北京邮电大学、中山大学、解放军理工大学、郑州大学、湖北工学院、中国国家信息安全测评认证中心等国内重点大学和科研机构在计算机的各个领域的著名学者组成“专家指导委员会”，为我们提供选题意见和出版监督。

权威的作者、经典的教材、一流的译者、严格的审校、精细的编辑，这些因素使我们的图书有了质量的保证，但我们的目标是尽善尽美，而反馈的意见正是我们达到这一终极目标的重

要帮助。教材的出版只是我们的后续服务的起点。华章公司欢迎老师和读者对我们的工作提出建议或给予指正，我们的联系方法如下：

电子邮件：hzedu@hzbook.com
联系电话：（010）68995265
联系地址：北京市西城区百万庄南街1号
邮政编码：100037

专家指导委员会

（按姓氏笔画顺序）

尤晋元	王　珊	冯博琴	史忠植	史美林
石教英	吕　建	孙玉芳	吴世忠	吴时霖
张立昂	李伟琴	李师贤	李建中	杨冬青
邵维忠	陆丽娜	陆鑫达	陈向群	周伯生
周克定	周傲英	孟小峰	岳丽华	范　明
郑国梁	施伯乐	钟玉琢	唐世渭	袁崇义
高传善	梅　宏	程　旭	程时端	谢希仁
裘宗燕	戴　葵			

本书为1996年软件开发图书震撼大奖（Jolt Award）得主。

“这本书是一项巨大的成就。你的书架上早就该有这本书了。讲述输入输出流的这一章是我至今见过的对这个主题最全面最易懂的叙述。”

——**《*Doctor Dobbs Journal*》杂志的资深编辑Al Stevens**

“Eckel的作品是惟一一本能如此清晰地叙述以面向对象方法构造程序的书籍。这本书也是一本优秀的C++入门指南。”

——**《*Unix Review*》杂志的编辑Andrew Binstock**

“Bruce继续用他对C++的洞察力让我惊叹。《C++编程思想》是他思想的精华荟萃。如果你需要C++中难题的清晰解答，就去买这部杰作吧。”

——**《*The Tao of Objects*》一书的作者Gary Entsminger**

“《C++编程思想》耐心而系统地探讨了C++的一些重要问题，例如内联、引用、运算符重载、继承和动态对象，还有一些更深奥的主题，例如模板的合理使用、异常和多继承等。所有这些内容，连同Eckel本人的关于对象和程序设计的观点，完美地编织为一体，成为每个C++开发者案头必备之书。如果你正在用C++开发软件，《C++编程思想》应当是你必须拥有的一本书。”

——**《*PC Magazine*》的特邀编辑Richard Hale Shaw**

读者评论

好书……了不起的作品!
——**Andrew Schulman, “Doctor Dobbs Journal” 杂志**

一本绝对的必备之书。我的书架上最有用的、最可信赖的书。
——**TUG Lines**

这是一本对程序员真正有用的书籍。
——**IEEE Computer**

令人耳目一新。
——**PJ Plauger, “Embedded Systems Programming” 杂志**

Eckel成功了，…这本书是如此清新可读。
——**Unix World**

绝对应当是你购书的首选。
——**C Gazette**

一本关于C++的绝妙参考书。
——**Michael Brandt, 高级分析员/程序员, 澳大利亚悉尼**

在HRB系统公司的项目中，我们称你的书为“答案书”，对于这个项目，它是我们的C++圣经。
——**Curt Snyder, HRB系统公司**

你的书真是伟大之作，你还使它可以从Web上免费获得，我的感激实在是难以言表。这是我已经见到的最透彻和最有用的C++书籍。
——**Russell Davis**

……这是我所仅见的讲述C++复杂知识（以及面向对象程序设计的基础）还能如此好读的书。
——**Gunther Schulz, KawaiiSoft**

我喜欢你书中的例子。这里有我从未想到过的东西（还有一些内容，我真是不知道你是怎么写出来的）！
——**Rich Herrick, 洛克希德－马丁联邦系统公司(Owego, NY)高级副软件工程师**

这是一本奇书。只要我有问题，我就在线查阅这本书。可以说是屡试不爽。能够读到如此水准的书，真是一种幸福。
——**Wes Kells, 计算机工程学生, SLC Kingston**

你是一个无价的源泉，我真的喜欢你的书、邮件列表……。我所做的几乎每个项目都得益于你的真知灼见。

——**Justin Voshell**

这正是我一直在寻找的C++书籍。

——**Thomas A. Fink, Trepp公司经理**

你的书非常权威，并且易于阅读。我对我的同事说你就是C++的K & R。

——**Mark Orlassino, 高级设计工程师, Harmon工业公司(Hauppauge, NY)**

当我第一次开始学习C++时，你的书《C++编程思想》是黑暗隧道中的指引明灯。我尽力提高我的C++技能，而在此过程中《C++编程思想》是我得以持续提高的坚实基础。

——**Peter Tran, 高级系统分析员（IM），原康柏计算机公司**

在我学习C++过程中，这本书是最好的综合参考书。大多数书透彻地解释了某些主题，而对于另一些主题则挂一漏万。而《C++编程思想》第2版当仁不让，有问必答。

——**Thomas Michel**

我藏书甚丰，但却没有一本令我满意。我渴求一本好的关于模板和STL的书。后来读了你的书，惊叹不已。你告诉我们如何用模板和STL写C++，一点都不拖泥带水。你所做的正是我期望C++社团（也就是C++的下一代作者）能够做到的。作为一名作者，我被你的写作和阐释才能所深深折服。你阐明了前人未能充分恰当说明的问题，你的方法源于你对所述素材的深入细致的研究。你对明智的情况和问题所在提出了质疑。在我的书架上，它是一本重要书籍，与Petzold放在一起。

——**Christian Gross, 顾问／导师, cgross@eusoft.com**

我认为你的书非常、非常、非常好。我将它与书店里的其他书比较，发现当我学习STL时，是你的书真正教会了我C++基础知识。这是非常好的学习经验，无论是一气呵成还是循序渐进。我认为你的书出奇的好，叙述方式非常容易理解。

——**Jeff Meininger, 软件开发员, boxybutgood.com**

你的书是我迄今看到的里面最好的。快点买吧，这样我们大家就有了优秀的和“可靠的”参考书。赶快！赶快！为了这样一本优秀书籍，这样做是值得的。

——**Steve Strickland, Live Minds (a Puzzle business)**

（来自Usenet）与其他大多数C++作者不同，Eckel专职教授C++和Java课程，他能从大量新手的学习反馈中受益，他的书反映了这一点。他的书不只是写如何用C++/Java编程，而且讲述了如何理解这些语言的意图和它们中蕴含的思想。Eckel也是一个我所读过的继Jeff

Duntemann以后的最好的技术作者。他的书非常清楚，容易阅读。不要被它们的大部头吓倒，实际上，每一本都能在21天之内读完。

——**Randy Crawford, MRJ Technology Solutions (Fairfax VA)**

你的作品广受好评，感谢你帮助我很好地理解了C++和Java。

——**Barry Wallin, 数学/计算机教师, Rosemount高中(Rosemount, MN)**

我应当感谢你的书《C++编程思想》，毋庸置疑，这是我迄今读到的这方面最好的书。

——**Riccardo Tarli, 软件工程师, 意大利R&D TXT Ingegneria Informatica**

我已经读了你的《Java编程思想》和《C++编程思想》，两者都是各自领域最好的书。

——**Ratnakarprasad H. Tiwari, Mumbai, 印度**

“调试技巧”一节相当有用，以至于我将它打印出来长期保留。我认为这一节应当是在第一个或第二个程序设计问题之后任何入门课的必须部分。

——**Fred Ballard, Synectics公司**

你的书实在是关于C++知识的珍宝。我觉得你总能先给出好的概括，然后解释具体细节。

——**Raymond Pickles, 美国海军研究实验室雷达部天线分部, 华盛顿特区**

我作为一个内科医生和计算机科技人员，需要花费大量的时间从书本和杂志中提取信息。我的经验是，一个好的作者能使困难的概念变得容易理解。如果要对这方面的图书打分，你是我认为的最高分数的技术作家之一。祝你常出优秀作品。

——**Declan O'Kane博士, 英国莱斯特**

对于我的二年级C++课程，《C++编程思想》是我要经常参考的书，我要求我的学生认真学习它。我经常提到关于运算符重载这一章。书中的例子或代码本身就很有价值，超过这本书定价的许多倍。许多书和开发环境假设所论述的语言只在Windows环境上应用，因此能找到和使用一本以C++为纯粹内容的书是重要的，这样我就能培养我的学生从事专门领域，例如嵌入式系统、网络等，这些都要求真正的深入理解。

——**Robert Chase教授, Sweet Briar学院**

我认为它是C++的绝佳介绍，特别对于像我这样的长期的业余者，我常常“知其然”，但很少能够“知其所以然”，TIC2真是天赐之物。

——**Tony Likhite, 系统管理员/数据库分析员, Together网络公司**

读了这本书的前80页之后，我感到比读成堆的该主题的书籍对OOP有了更深刻的理解。多谢!

——**Rick Schneewind**

译 者 序

作为译者，我有幸组织翻译了《C++编程思想》第1版。在这之前，我仅仅耳闻这是一本别具特色的畅销书，至于如何别具特色，如何得以畅销，并不十分清楚。在第1版的翻译过程中，我逐渐领悟了Eckel编写技巧的真谛。在第1版中文版的译者序中，我曾这样总结他的技巧："其内容、讲授方法、选用例子和跟随的练习，别具特色。原书作者不是按传统的方法讲解C++的概念和编程方法，而是根据他自己过去学习C++的亲身体会，根据他多年教学中从他的学生们的学习中发现的问题，用一些非常简单的例子和简练的叙述，阐明了在学习C++中特别容易混淆的概念。特别是，他经常通过例子引导读者从C++编译实现的汇编代码的角度反向审视C++的语法和语义，常常使读者有'心有灵犀一点通'的奇特效果，这在以往的C++书中并不多见。"

《C++编程思想》第1版的中文版自2000年1月第1次印刷以来，在中国市场上的畅销势头经久不衰。这充分说明了这本书在中国读者心目中的地位。

Eckel致力于计算机程序设计语言教学数十年，而且是全心全意地从事这项工作，这本身就是难能可贵的，是他成功的根本原因。另外，他的成功还有赖于他的精益求精的精神，这不仅表现在第1版的与众不同的精心选材和认真推敲的叙述方面，也体现在第2版与第1版的不同点上。

表面上，第2版与第1版并无太多的变化，但是通过分析，可以看出，其中的任何变化都是经过深思熟虑的。从章节上看，最大的区别是增加了两章和去掉了四章。增加的两章分别是"对象的创建与使用"和"C++中的C"。前者与"对象导言"实际上是第1版的"对象的演化"一章的彻底重写，增加了近几年面向对象方法和编程方法的最新研究与实践的丰硕成果。后者的添加不仅使不熟悉C的读者直接使用这本书成为可能，而且C本身就是C++的组成部分，这是C++得以成功的主要原因之一。删去的四章是"输入输出流介绍"、"多重继承"、"异常处理"和"运行时类型识别"。这四章属于C++中较复杂的主题，作者将它们连同C++标准完成后又增加的一些内容放到这本书的第2卷中。这样就使得这本书的第1卷内容更加集中，一般的读者可以不被这些复杂内容所困扰，而需要这些复杂知识的读者可以阅读这本书的第2卷。

实际上，第2版的改变不仅仅在于这些章节的调整，更多的改变体现在每一章中，包括例子的调整和练习的补充。这本书更成熟了。

受机械工业出版社华章公司计算机编辑部委托，我又承担起《C++编程思想》第2版的翻译组织任务。翻译这样的成功之作，既是机遇，更是压力。有如此众多的读者阅读我们翻译的作品，无论如何这是令人高兴的事情。诚然，吸引读者的魅力来源于原作，而不是我们的翻译技巧，但是能将如此光辉灿烂的作品变成中文版本，奉献给中国的读者，这其中毕竟融入了我们的心血，而且，第1版中文版的畅销，已经充分证明，并未因我们翻译水平的限制而黯淡了原作的光芒，这对我们已经够宽慰的了。然而，百万双眼睛在阅读这本书的同时也在审视我们的翻译水平，这就足以使我们诚惶诚恐的了。翻译在某种意义上是再创作的过程，

读者见仁见智。为求更多的满意，我们只有尽力而为。由于时间和水平限制，翻译错误在所难免，恳请读者指正。

参加第2版翻译和审校工作的人员包括：刘宗田、韩冬、蔡烈斌、袁兆山、潘秋菱、许东、李航、肖苑、刘璐、姜桂华、张卿、邵坤、陈慧琼、何允如、贾亮、童朝柱、邢大红、潘飚、刘莹、姜川、冯鸿等。

感谢为本书第1版和第2版中文版作出贡献的所有朋友。感谢关心和支持本书翻译出版的广大读者。

刘宗田

2002.8.10

前　言

像任何人类语言一样，C++提供了一种表达思想的方法。如果这种表达方法是成功的，那么当问题变得更大和更复杂时，该方法将会明显地表现出比其他方法更容易和更灵活的优点。

1

不能只把C++看做是语言要素的一个集合，因为有些要素单独使用是没有意义的。如果我们不只是用C++语言编写代码，而是用它思考“设计”问题，那么必须综合使用这些要素。而且，为了以这种方法理解C++，我们必须了解使用C的问题和一般的编程问题。本书讨论的是编程问题、为什么这些编程问题会成为要解决的问题以及用C++解决编程问题所采用的方法。因此，在每一章中所解释的一组语言要素，都建立在C++语言解决某一类特殊问题所用方法的基础之上。以这种方式，我希望一点一点地引导读者，从掌握C开始，直到读者使用C++变成自己的母语思维方式。

我将始终坚持一种观点：读者应当在头脑中建立一个模型，以便逐步理解这种语言，直到炉火纯青的程度。如果读者遇到难题，他可以将问题纳入这个模型，推导出答案。我将努力把已经印在我脑海中的见解传授给读者，正是这些见解，使得我能开始“用C++进行思考”。

0.1　第2版中的新内容

本书是第1版的彻底重写，反映了C++标准最终完成所带来的C++的所有改变，也反映了自从第1版写完后我又学习到的内容。我已经检查并重写了第1版中的全部文字，在这个过程中，我删去了一些过时的例子，修改了一些现有的例子，并增加了一些新的例子和新的练习。我对第1版的内容进行了大规模的重新整理和重新编排，以便反映新出现的更好的工具和我对人们如何学习C++的进一步理解。为方便没有C背景知识的读者能阅读本书后面的章节，在第2版增加了一章，简要地介绍C概念和基本的C++特征。本书配套的CD ROM包含了一份课堂讨论的材料，这是为了理解C++（或Java）所必需的C概念的介绍，这是由Chuck Allison为我的公司（MindView, Inc.）创建的，称为“Thinking in C: Foundations for Java and C++”，它介绍了转向C++或Java所必需的C的知识。其中省略了C程序员必须每天面对，而C++和Java语言能让我们避免的繁琐的内容（在Java中，甚至可能消除这些繁琐的工作）。

2

因而，对于“第2版与第1版相比有何不同”这个问题的简要回答是：不同之处不在于版本号是新的，而是进行了重写，有的地方读者甚至无法认出原来的例子和材料。

0.1.1　第2卷的内容是什么

C++标准增加了一些重要的新库，例如**String**、在标准C++库中的容器和算法，以及模板中的新的复杂性。这些新增的内容和其他更高级的主题被放进本书的第2卷，包括多重继承、异常处理、设计模式和建立和调试稳定系统等内容。

0.1.2 如何得到第2卷

就像当前你手上的这本书一样，《C++编程思想》第2卷完全可以从我的网站www.BruceEckel.com上下载。

这个站点还包括这两本书的源代码，以及有关MindView公司提供的CD ROM上其他课堂讨论材料的更新和信息，其中包括：公开课堂讨论、内部培训、辅导课和演示。

0.2 预备知识

在本书第1版中，我假定读者已经学习了C，并至少具有自如阅读的水平。我的重点放在简化我认为比较困难的部分：C++语言。第2版增加了一章，快速地介绍C，并在光盘上提供“Thinking in C”的课堂讨论材料，但是即使如此，我仍然假设读者具有一定的程序设计经验。
3 另外，正如读者可以通过读小说而直接地学会许多新词一样，读者也可以从在本书后面的文字中学习有关于C的大量知识。

0.3 学习C++

我希望本书的读者有和我进入C++时相同的情况：作为一个C程序员，对于编程持有实在而执着的态度。但糟糕的是，我的背景和经验是在硬件层的嵌入式编程方面。在那里，C常常被看做高层语言，它对于位操作是低效率的。后来我发现，自己甚至不是一个好的C程序员，平时总是掩盖了对**malloc( )**和**free( )**、**setjmp( )**和**longjmp( )**结构以及其他“复杂”概念的无知，当开始触及这些主题时就竭力回避，而不是努力去获取新的知识。

在我开始致力于学习C++时，当时惟一像样的书是Stroustrup夫子自道式的“专家指南”[⊖]，因此我只好靠自己弄清基本概念。这引出了我的第一本关于C++的书[⊜]，这本书基本上就是直接把我头脑中的经验倒出来而写成的。它的构思是作为读者的指南，引导程序员同时进入C和C++。这本书的两个版本[⊜]都收到了读者的热情反响。

几乎就在《*Using C++*》出版的同时，我开始讲授这门语言。讲授C++已经变成了我的职业。自1989年以来，在授课时我看到了世界各地听众昏昏欲睡的样子、茫然不知的面容和困惑不解的表情。当我对一些人数不多的人群进行内部培训时，在练习过程中又发现了某些问
4 题。即便那些面带微笑和会心点头的学生，实际上对许多问题也还是糊涂的。通过开创和多年主持“软件开发会议”的C++和Java系列专题，我发现，我和其他讲演者都有一种倾向，即过快地向听众灌输了过多的主题。后来，我做了一些努力，通过区别对待不同层次的听众和提供相关资料的方法，尽量吸引听众。也许这是过分的要求，但是因为我是一个抵触传统教学的人（对于大部分人而言，我相信这种抵触源于厌倦），所以希望我通过努力，使每一个人都能跟得上教学进度。

有一段时间，我编写了大量的教学演示。这样，我结束了通过实验和重复方式进行学习（在设计C++程序的过程中，这也是一项很有用的技术）的阶段。最后，从我多年的教学经验中总结出来的所有内容，形成了一门课程。在课程中，我用一系列分离的、易于理解的步骤

⊖ Bjarne Stroustrup, *The C++ Programming Language*, Addison-Wesley, 1986 (第1版).

⊜ *Using C++*, Osborne/McGraw-Hill 1989.

⊜ *Using C++* and *C++ Inside & Out*, Osborne/McGraw-Hill 1993.

并采用实地课堂讨论的形式解决学习中的问题（理想的学习情况），并在每次课后面跟随着练习。读者能从*www.BruceEckel.com*找到我的公开课堂讨论，还可以学习我已经制作成CD ROM的课堂讨论材料。

本书的第1版是作为两学年制课程编写的，并且书中的内容已经在许多不同的课堂讨论上通过了多种形式的检验。我从每次课堂讨论上收集反馈意见，不断地修改和调整内容，直到我感觉到它已经成为一本很好的教材为止。但这本书不仅仅是课堂讨论的分发教材，而且我在其中放入了尽可能多的信息，在结构上使得它能引导读者顺利地通过当前主题和进入下一个主题。另外，这本书也适合于自学读者，能帮助他们尽快地掌握这门新的编程语言。

0.4 目标

在这本书中，我的目标是：

1) 以适当的进度介绍内容。每次将学习向前推进一小步，因此读者能很容易地在继续下一步学习之前消化每个已学过的概念。

5

2) 尽可能使用简短的例子。当然，这有时会妨碍我解决“现实世界”的问题。但是，我发现，当初学者能够掌握例子的每个细节，而不受问题的领域所影响时，他们通常会更有兴趣进行学习。另外，在课堂情况下能达到的接受能力，对代码的长短也有严格的限制。为此，我有时会受到使用“玩具例子”的批评，但是我甘愿承受这一批评，因为这样更有利于取得某些教学法上的效果。

3) 仔细安排描述内容的顺序，不让读者看到还没有揭示的内容。当然，这不是总能做到的；如果出现了这种情况，我将会给出简明的介绍性的描述。

4) 只把对于理解这门语言比较重要的东西介绍给读者，而不是介绍我知道的所有内容。我相信，不同信息的重要性是不同的。有些内容是95%的程序员不需要知道的，这些东西只会迷惑人们，增加他们对该语言复杂性的感觉。举一个C语言的例子，如果我们记住运算符优先表（我是记不住的），我们就可以写更漂亮的代码。但是，如果一定要这样做，反而会使代码的读者或维护者糊涂。所以可以忘掉优先级，当不清楚时使用括号。我们对于C++中的某些内容也可以采取同样的态度，因为我认为这些内容对于写编译器的人更重要，而对于程序员就不是那么重要。

5) 保持每一节的内容充分集中，使得授课时间以及两个练习之间的间隔时间不长。这不仅能使听众保持活跃的思想和在课堂讨论中精力集中，而且会使他们有更大的成就感。

6) 帮助读者打下坚实的基础，使得他们能充分地理解面对的问题，从而可以顺利地转向学习更困难的课程和书籍(特别是这本书的第2卷)。

6

7) 我尽力不用任何特定厂商的C++版本，因为对于学习编程语言，我不认为特定实现的细节像语言本身一样重要。大部分厂商的文档只适合于他们自己的特定实现。

0.5 各章概要

C++是一个在已有文法上面增加了新的不同特征的语言（因此，它被认为是混合的面向对象的编程语言）。由于很多人走了学习弯路，因此我们已经开始探索C程序员转移到C++语言特征的方法。因为这是过程型训练思想的自然延伸，所以我决定去理解和重复相同的道路，并通过引出和回答一些问题来加速这一进程，这些问题是当我学习该语言时遇到的和听众在

听我的课时提出来的。

设计这门课时，我始终注意一件事：精练C++语言的学习过程。听众的反馈意见帮助我认识到哪些部分是很难学习的和需要额外解释的。在这个领域中，我曾经雄心勃勃，一次讲解包括了太多的内容。通过讲解过程，我知道了，如果包括大量新特征，就必须对它们全部作出解释，而且学生也特别容易混淆这些新特征。因此，我努力一次只介绍尽可能少的特征，理想的情况是每章一次只介绍一个主要概念。

本书的目标是只在每一章中讲授一个概念，或只讲授一小组相关的概念，用这种方法，不会依赖于其他的特征。这样，在进入下一章的学习之前，学生可以对自己的当前知识融会贯通。为了实现这个目标，我把一些C特征留到后面的章节去介绍，甚至放在比我希望的还要往后的地方介绍。这样做的好处是读者不会因为看到了许多未解释的C++特征被使用而困惑，
7 因此，对该语言的介绍将是和缓的，并且将反映出读者自己消化这些特征时将会采用的方式。

下面是本书各章内容的简要说明。

第1章　对象导言。当项目对于维护而言变得太大和太复杂时，就产生了“软件危机”。按程序员们的说法，“我们无法完成那些项目，即便能完成，它们也太昂贵了”。这引出了一些问题，在本章中我将讨论这些问题，并且讨论面向对象程序设计（OOP）的思想和如何运用这一思想解决软件危机问题。这一章引导读者了解OOP的基本概念和特征，介绍分析和设计过程。另外，在这一章中，我还将阐述采用这种语言的好处，提出关于如何转入C++语言领域的建议。

第2章　对象的创建与使用。这一章解释用编译器和库建立程序的过程。它介绍了本书中的第一个C++程序，显示程序如何构造和编译，然后介绍标准C++中的可用对象的基本库。在结束这一章时，我们就对如何用流行的对象库编写C++程序有一个深刻的领会。

第3章　C++中的C。这一章详细综述在C++中使用的C的特征和一些只在C++中使用的特征，还介绍在软件开发领域通用的“制作”工具，并且用它建立了本书中的所有例子（本书的源代码在*www.BruceEckel.com*中可找到，包含了对每章的makefile）。第3章假设读者已经具有某种过程型程序设计语言的坚实基础，例如Pascal和C语言或者甚至某种形式的Basic（只要读者已经用这种语言编写了大量的代码，特别是函数）。如果读者发现这一章有些难，就应当首先看本书附带的CD ROM上的“Thinking in C”的课堂讨论材料(亦可在*www.BruceEckel.com*下载)。

第4章　数据抽象。C++的大部分特征都围绕着创建新数据类型的能力。这不仅可以提供
8 优质代码组织，而且可以为更强大的OOP能力奠定基础。读者将可以看到如何用将函数放入结构内部的简单过程来实现这一思想，并可以看到如何具体地完成这样的过程和创建什么样的代码。读者还能学会组织代码成为头文件和实现文件的最好方法。

第5章　隐藏实现。通过说明结构中的一些数据和函数是**private**（私有的），可以把它们设置为对于这个新结构类型的用户是不可见的。这意味着能够把下层实现和客户程序员看到的接口隔离开来，这样就容易改变具体实现，而不影响客户代码。另外，C++还引入关键字**class**作为描述新数据类型的更具吸引力的方法，而单词“对象”的意思并不神秘，它是一种美妙的变量。

第6章　初始化与清除。C语言的最通常的一类错误是由于变量未初始化而引起的。C++的构造函数使得程序员能保证他的新数据类型（即“他的类的对象”）的变量总是能被恰当地

初始化。如果他的对象还需要某种方式的清除，他可以保证这个清除动作总是由C++的析构函数来完成。

第7章　函数重载与默认参数。C++可以帮助程序员建立大而复杂的项目。这时，可能会引进使用相同函数名的多个库，还可能会在同一个库中选择具有不同含义的相同的名字。C++采用函数重载使这一问题容易解决。重载允许当参数表不同时重用相同的函数名。默认参数通过自动为某些参数提供默认值，使我们能用不同的方式调用同一个函数。

第8章　常量。本章讨论了**const**和**volatile**关键字，它们在C++中有另外的含义，特别是在类的内部。我们将学习对指针定义使用**const**的含义。本章还说明**const**的含义在类的内部和外部有何不同，以及如何在类的内部创建编译时常量。 9

第9章　内联函数。预处理宏省去了函数调用开销，但是也排除了有价值的C++类型检查。内联函数具有预处理宏和实际函数调用的所有好处。这一章深入地研究了内联函数的实现和使用。

第10章　名字控制。在程序设计中，创建名字是基本的活动，而当项目变大时，名字的数目是无法限制的。C++允许在名字创建、可视性、存储代换和连接方面有大量的控制。这一章将说明如何在C++中用两种技术控制名字。第一，用关键字**static**控制可视性和连接，研究它对于类的特殊含义。另一种在全局范围内更有用的控制名字的技术是C++的**namespace**（名字空间）特征，它允许把全局名字空间划分为不同的区域。

第11章　引用和拷贝构造函数。C++指针的作用和C指针一样，而且具有更强的C++类型检查的好处。C++还提供了另外的处理地址的方法：继Algol和Pascal之后，C++使用了“引用”，允许当程序员使用平常的符号时由编译器来处理地址操作。读者还会遇到拷贝构造函数，它通过按值传送控制将对象传送给函数或从函数中返回的方式。最后，本章还将解释C++指向成员的指针。

第12章　运算符重载。这个特征有时被称为“语法糖（syntactic suger）”。由于运算符也可以像函数调用那样使用，这使得程序员在运用类型的语法时更加灵活。在这一章中，读者将了解到，运算符重载只是不同类型的函数调用，并且将学会如何写自己的运算符重载，学 10
会处理参数、类型返回以及确定一个运算符是成员还是友元时的一些易混淆的情况。

第13章　动态对象创建。一个空中交通系统将处理多少架飞机？一个CAD系统将需要多少种造形？在一般的程序设计问题中，我们不可能知道程序运行所需要的对象的数量、生命期和类型。在这一章中，我们将学习C++的**new**和**delete**如何漂亮地通过在堆上安全地创建对象而解决上述问题。我们还将看到，**new**和**delete**如何用不同的方法重载，从而使我们能控制如何分配和释放存储。

第14章　继承和组合。数据抽象允许程序员从零开始创建新的类型。通过组合和继承，程序员可以用已存在的类型创建新的类型。用组合方法，程序员可以以老的类型作为零件组装成新的类型；而用继承方法，程序员可以创建已存在类型的一个更特殊的版本。在这一章中，读者将学习这一文法，学习如何重定义函数，以及理解构造和析构对于继承和组合的重要性。

第15章　多态性和虚函数。单靠读者自己，可能要花九个月的时间才能发现和理解OOP的这一基石。通过一些小而简单的例子，读者可以看到如何通过继承创建一个类型族，看到在这个类型族中如何通过公共基类对这个族中的对象进行操作。关键字**virtual**使程序员能够按类属处理这个族中的所有对象，这意味着大块代码将不依赖于特殊类型的信息，因此，程

序是可扩充的，构造程序和维护代码也变得更容易和更便宜。

第16章　模板介绍。继承和组合允许程序员重用对象代码，但不能解决有关重用需要的所有问题。模板通过为编译器提供了一种在类或函数体中代换类型名的方法，来允许程序员重用源代码。这就支持了容器类库的使用，容器类库是使我们能快速而有效地开发面向对象程序的重要工具(标准C++库包含了一个重要的容器类库)。这一章给出了这个基本主题的详尽阐述。

另一些主题（更高级的课题），可以在本书的第2卷中看到，本书的第2卷可以从网站
11 *www.BruceEckel.com*下载。

0.6 练习

我已经发现，在课堂讨论期间，练习对同学们的完全理解是特别有用的，因此，本书的每章后面都有一组练习。练习题的数量在第1版的基础上有很大的增加。

在这些练习中，很多是比较简单的，可以在课堂内或实验室内用合理的时间完成,老师可以通过观察证实所有的学生正在吸收这些材料。有些练习是为了激发优秀学生的学习兴趣。大量练习被设计成能在短期内求解，目的是为了测试和完善学生所掌握的知识，而不是提出主要的挑战(也许我们将自己找到那些挑战，更可能的是，那些挑战会自动出现在我们面前)。

0.6.1 练习的答案

部分练习题的答案可以在本书的电子文档“*Thinking in C++ Annotated Solution Guide*”中找到，只需支付很少的费用就可以从网站*www.BruceEckel.com*上获得这个电子文档。

0.7 源代码

本书中的源代码是免费软件版权，通过网站*www.BruceEckel.com*分发。该版权防止未经允许用印刷媒体重印这些代码，但是，在许多其他情况下可以使用这些代码（见下）。

这些代码放在一个压缩文件中，可以从任何有zip工具的平台上提取（如果没有安装合适的平台，可以从Internet上找到适合你的平台的某个版本）。在解压缩的首目录上，我们可以找到如下所示的版权声明:

```
//:! :Copyright.txt
Copyright (c) 2000, Bruce Eckel
12 Source code file from the book "Thinking in C++"
All rights reserved EXCEPT as allowed by the
following statements: You can freely use this file
for your own work (personal or commercial),
including modifications and distribution in
executable form only. Permission is granted to use
this file in classroom situations, including its
use in presentation materials, as long as the book
"Thinking in C++" is cited as the source.
Except in classroom situations, you cannot copy
and distribute this code; instead, the sole
distribution point is http://www.BruceEckel.com
(and official mirror sites) where it is
available for free. You cannot remove this
```

```
copyright and notice. You cannot distribute
modified versions of the source code in this
package. You cannot use this file in printed
media without the express permission of the
author. Bruce Eckel makes no representation about
the suitability of this software for any purpose.
It is provided "as is" without express or implied
warranty of any kind, including any implied
warranty of merchantability, fitness for a
particular purpose, or non-infringement. The entire
risk as to the quality and performance of the
software is with you. Bruce Eckel and the
publisher shall not be liable for any damages
suffered by you or any third party as a result of
using or distributing this software. In no event
will Bruce Eckel or the publisher be liable for
any lost revenue, profit, or data, or for direct,
indirect, special, consequential, incidental, or
punitive damages, however caused and regardless of
the theory of liability, arising out of the use of
or inability to use software, even if Bruce Eckel
and the publisher have been advised of the
possibility of such damages. Should the software
prove defective, you assume the cost of all
necessary servicing, repair, or correction. If you
think you've found an error, please submit the
correction using the form you will find at
www.BruceEckel.com. (Please use the same
form for non-code errors found in the book.)
///:~
```

13

读者可以在自己的项目中和在课堂上使用这些代码，只要遵守以上的版权声明。

0.8 语言标准

在本书中，当谈到遵循ISO C标准时，我一般只是说‘C’。只有当有必要区别标准C和老的、以前版本的C时，我才加以区分。

在写这本书时，C++标准委员会完成了语言的标准化工作。这样，我将用术语“标准C++”来指代这个标准化的语言。如果我简单地谈到C++，读者就应该假设这意味着“标准C++”。

在C++标准委员会的实际名字与标准本身的名字之间有些混淆。委员会的主席Steve Clamage就此作了如下澄清：

> 有两个C++标准委员会：NCITS（以前的X3）J16委员会和ISO JTC1/SC22/WG14委员会。ANSI授权NCITS建立制订美国国家标准的技术委员会。
>
> 1989年J16受委托制订C++美国标准。1991年WG14受委托制订国际标准。J16项目转变为“Type I”（国际）项目，并服从于ISO标准化计划。
>
> 这两个委员会在同一时间、同一地点开会，J16的投票作为美国在WG14的票数。WG14委派J16做技术工作，并对J16的技术工作进行表决。
>
> 最初，C++标准是作为ISO标准制订的。ANSI后来投票（在J16的建议下）决定采用ISO C++标准作为C++美国标准。

14

因此，“ISO”是称呼C++标准的正确方式。

0.8.1 语言支持

读者的编译器可能不支持在本书中讨论的所有特征，如果还没有编译器的最新版本就更是如此了。实现像C++这样的语言是艰巨的任务，而且读者可能希望将这些特征划分成一些部分后分别出现，而不是一下子全出现。如果读者试用本书中的某个例子，从编译器中得到了大量的错误，这可能不是代码或编译错误，而可能是他的特定编译器还没有实现例子中的某个特征。

0.9 本书的CD ROM

本书附带的CD ROM的主要内容是一个“CD ROM课堂讨论”。题目是“*Thinking in C: Foundations for Java & C++*”，由Chuck Allison编写（MindView, Inc.出版）。这些内容包含几小时的讲课录音和幻灯片，如果读者有CD ROM播放机和音响系统，就能在大多数计算机上观看。

*Thinking in C*的目标是让读者关注整个C语言的基础。它的重点放在使读者能够转移到C++或Java语言所必需的知识上，而不是试图使读者在C的所有问题上都成为专家。（使用诸如C++或Java高级语言的原因之一，正是我们能因而避开许多犄角旮旯的问题。）它还包括练习和答案。请记住，因为本书的第3章超出了“Thinking in C”的CD ROM的内容，所以这个CD ROM不能代替这一章，但它应当用做本书的预备知识。

15 请注意，这个CD ROM是基于浏览器的，所以读者应当先安装Web浏览器。

0.10 CD ROM、课堂讨论和咨询

CD ROM上有一些课堂讨论，覆盖了本书的第1卷和第2卷。它包括几小时的我的讲课录音，配套幻灯片，包含本书每一章选用的材料。如果读者有CD ROM播放机和音响系统，就能在大多数计算机上观看。这些CD ROM可以在网站*www.BruceEckel.com*购买，同时那里还可以发现更多的信息和示范讲义。

我的公司MindView, Inc提供基于本书的材料以及高级主题的公开课堂讨论。从每一章选取的材料代表一课，后面有一个检测练习的阶段，这样每个学生能得到指导。我们还提供现场培训、咨询、辅导、设计和代码演练的服务。

我有时做设计咨询、项目评价和代码演练。我第一次写计算机程序时，我的最初动机是使我的咨询活动更好，因为我发现咨询是富有挑战性的、具有教育意义的和最愉快的个人经历。这样，我将竭尽全力为满足客户要求而安排日程，或者向你推荐我的一个合作者（这些人是我熟悉的和信任的伙伴，他们常常与我一起工作和举办课堂讨论）。

0.11 错误

无论一个作者具有多少发现错误的技巧，总会有一些错误漏网，它们常常能被新读者发现。如果读者发现了任何认为是错误的地方，请填写网站*www.BruceEckel.com*上关于本书的

16 修改表格并在线提交，我将非常感激。

0.12 关于封面

这本书的第1版的封面上有我的头像，但是我最初希望第2版的封面能像《Java编程思想》

一样更加像个艺术品。由于某些原因，在我看来C++似乎隐含着装饰艺术，具有简单的线条并刷过铬黄的颜料。在我的脑海中会浮现出像轮船和飞机这样的招贴画，上面是长长的流线体。

我的朋友Daniel Will-Harris (*www.Will-Harris.com*)与我初识于初中的唱诗班，现在他已经成了世界级的设计师和作者。他已经实际上为我做了所有的设计，包括这本书第1版的封面。在这一版封面的设计过程中，Daniel不满意我们正在制作的进程，不停地问："怎样才能体现人与计算机的关系呢？"我们卡壳了。

他突发奇想，让我把自己的头像放在屏幕上。Daniel有一个图像程序（Corel Xara，他的至爱），能自动勾勒我的头像的扫描图。正如他所描述的，"自动勾勒是一种计算机方法，能将图片变为一些逼真线条。"然后他开始工作，直到使我的头像看上去像一张地形图，这个图像可能就是计算机眼中人的形象吧。

我看了这个图像，将它影印在水彩画纸上，然后通过在图像上增加水彩开始创造大量实验。我挑选了最喜欢的几幅图像，然后Daniel再把它们扫描回来，安排在封面上，又增加了文字和其他内容。整个过程花了几个月，主要是涂水彩的时间。但是我特别高兴，因为我参与了封面艺术设计，激励我做了大量的水彩画。 17

0.13 感谢

首先，感谢在Internet上提交修改和建议的每一位读者，这对于改进本书的质量起了重要的帮助作用。特别要感谢John Cook。

本书中的思想和观点来源于许多方面，包括：我的朋友，例如Chuck Allison、Andrea Provaglio、Dan Saks、Scott Meyers、Charles Petzold和Michael Wilk；C++语言的倡导者，例如Bjarne Stroustrup、Andrew Koenig和Rob Murray；C++标准委员会的成员，例如Nathan Myers（他给出特别的帮助和毫无保留的见解）、Bill Plauger、Reg Charney、Tom Penello、Tom Plum、Sam Druker和Uwe Steinmueller；在软件开发会议上对我的C++教学发表过见解的学者；在我的课堂上参与讨论的学生，我认真地听取他们提出的问题，以使这本教材更加清晰易懂。

我对我的朋友Gen Kiyooka深表感谢，他的公司Digigami为我提供了Web服务器。

我的朋友Richard Hale Shaw和我一起教授C++；Richard（还有Kim）的见解和支持对我很有帮助。我还要感谢KoAnn Vikoren、Eric Faurot、Jennifer Jessup、Tara Arrowood、Marco Pardi、Nicole Freeman、Barbara Hanscome、Regina Ridley、Alex Dunne以及MFI的全体人员。

我特别感谢我的所有老师和所有学生（他们也是我的老师）。

还要感谢为这本书作出贡献的John Irving、Neal Stephenson、Robertson Davies (我们会思念你)、Tom Robbins、William Gibson、Richard Bach、Carlos Castaneda和 Gene Wolfe。

Python语言的发明者Guido van Rossum将他的发明无私地贡献给了全世界。他的贡献使我受益。 19

感谢Prentice Hall的工作人员：Alan Apt、Ana Terry、Scott Disanno、Toni Holm以及我的电子版编辑Stephanie English。在市场营销方面，还要感谢Bryan Gambrel和Jennie Burger。

Sonda Donovan帮助制作CD ROM产品。Daniel Will-Harris在盘片上创造了丝网装璜设计。

感谢Crested Butte中友好的人们，他们构筑了这么一个奇妙的地方，特别是Al Smith（很棒的咖啡乐园的创造者）、我的邻居们Dave & Erika、Heg地方书店的Marsha、Teocalli Tamale

的Pat & John 、面包咖啡馆的Sam，以及帮助我进行音频研究的Tiller。还要感谢Camp4的所有充满激情的人们，使我的每一个早晨都是如此美妙。

为我提供帮助的朋友包括：Zack Urlocker、Andrew Binstock、Neil Rubenking、 Kraig Brockschmidt、Steve Sinofsky、JD Hildebrandt、Brian McElhinney、Brinkley Barr、Larry O'Brien、“子夜工程杂志”的Bill Gates、Larry Constantine、Lucy Lockwood、 Tom Keffer、Dan Putterman、Gene Wang、Dave Mayer、David Intersimone、Claire Sawyers、意大利的朋友们(Andrea Provaglio、Rossella Gioia、Laura Fallai、Marco & Lella Cantu、Corrado、Ilsa和Christina Giustozzi)、Chris和Laura Strand (和Parker)、the Almquists、Brad Jerbic、Marilyn Cvitanic、the Mabrys、the Haflingers、the Pollocks、Peter Vinci、the Robbins、the Moelters、Dave Stoner、Laurie Adams、the Cranstons、Larry Fogg、Mike和Karen Sequeira、Gary Entsminger和Allison Brody、Kevin、Sonda, & Ella Donovan、Chester和Shannon Andersen、Joe Lordi、Dave和Brenda Bartlett、the Rentschlers、Lynn和Todd, 以及他们的家庭等等，当然还包括我的父母。

20

目　录

译者序
前言
第1章　对象导言 ……1
1.1　抽象的过程 ……1
1.2　对象有一个接口 ……2
1.3　实现的隐藏 ……4
1.4　实现的重用 ……5
1.5　继承:重用接口 ……5
1.5.1　is-a 关系和is-like-a 关系 ……8
1.6　具有多态性的可互换对象 ……8
1.7　创建和销毁对象 ……11
1.8　异常处理：应对错误 ……12
1.9　分析和设计 ……12
1.9.1　第0阶段：制定计划 ……14
1.9.1.1　任务陈述 ……14
1.9.2　第1阶段：我们在做什么 ……14
1.9.3　第2阶段：我们将如何建立对象 ……16
1.9.3.1　对象设计的五个阶段 ……17
1.9.3.2　对象开发准则 ……18
1.9.4　第3阶段：创建核心 ……18
1.9.5　第4阶段：迭代用例 ……19
1.9.6　第5阶段：进化 ……19
1.9.7　计划的回报 ……20
1.10　极限编程 ……20
1.10.1　先写测试 ……21
1.10.2　结对编程 ……22
1.11　为什么C++会成功 ……22
1.11.1　一个较好的C ……22
1.11.2　延续式的学习过程 ……23
1.11.3　效率 ……23
1.11.4　系统更容易表达和理解 ……23
1.11.5　尽量使用库 ……23
1.11.6　利用模板的源代码重用 ……24
1.11.7　错误处理 ……24
1.11.8　大型程序设计 ……24
1.12　为向OOP转变而采取的策略 ……24
1.12.1　指导方针 ……25
1.12.1.1　训练 ……25
1.12.1.2　低风险项目 ……25
1.12.1.3　来自成功的模型 ……25
1.12.1.4　使用已有的类库 ……25
1.12.1.5　不要用C++重写已有的代码 ……25
1.12.2　管理的障碍 ……25
1.12.2.1　启动的代价 ……26
1.12.2.2　性能问题 ……26
1.12.2.3　常见的设计错误 ……26
1.13　小结 ……27
第2章　对象的创建与使用 ……28
2.1　语言的翻译过程 ……28
2.1.1　解释器 ……28
2.1.2　编译器 ……29
2.1.3　编译过程 ……29
2.1.3.1　静态类型检查 ……30
2.2　分段编译工具 ……30
2.2.1　声明与定义 ……30
2.2.1.1　函数声明的语法 ……31
2.2.1.2　一点说明 ……31
2.2.1.3　函数的定义 ……31
2.2.1.4　变量声明的语法 ……32
2.2.1.5　包含头文件 ……33
2.2.1.6　标准C++ include 语句格式 ……33
2.2.2　连接 ……34
2.2.3　使用库文件 ……34
2.2.3.1　连接器如何查找库 ……34
2.2.3.2　秘密的附加模块 ……35
2.2.3.3　使用简单的C语言库 ……35
2.3　编写第一个C++程序 ……35
2.3.1　使用iostream类 ……35

2.3.2 名字空间……36
2.3.3 程序的基本结构……37
2.3.4 “Hello, World!”……37
2.3.5 运行编译器……38
2.4 关于输入输出流……38
2.4.1 字符数组的拼接……39
2.4.2 读取输入数据……39
2.4.3 调用其他程序……40
2.5 字符串简介……40
2.6 文件的读写……41
2.7 **vector**简介……42
2.8 小结……45
2.9 练习……46
第3章 C++中的C……47
3.1 创建函数……47
3.1.1 函数的返回值……48
3.1.2 使用C的函数库……49
3.1.3 通过库管理器创建自己的库……49
3.2 执行控制语句……50
3.2.1 真和假……50
3.2.2 **if-else**语句……50
3.2.3 **while**语句……51
3.2.4 **do-while**语句……51
3.2.5 **for**语句……52
3.2.6 关键字**break** 和 **continue**……53
3.2.7 **switch**语句……54
3.2.8 使用和滥用**goto**……55
3.2.9 递归……55
3.3 运算符简介……56
3.3.1 优先级……56
3.3.2 自增和自减……57
3.4 数据类型简介……57
3.4.1 基本内部类型……57
3.4.2 **bool**类型与**true**和**false**……58
3.4.3 说明符……59
3.4.4 指针简介……60
3.4.5 修改外部对象……62
3.4.6 C++引用简介……64
3.4.7 用指针和引用作为修饰符……65
3.5 作用域……66
3.5.1 实时定义变量……67
3.6 指定存储空间分配……68
3.6.1 全局变量……68
3.6.2 局部变量……69
3.6.2.1 寄存器变量……69
3.6.3 静态变量……70
3.6.4 外部变量……71
3.6.4.1 连接……71
3.6.5 常量……72
3.6.5.1 常量值……72
3.6.6 **volatile**变量……73
3.7 运算符及其使用……73
3.7.1 赋值……73
3.7.2 数学运算符……73
3.7.2.1 预处理宏介绍……74
3.7.3 关系运算符……75
3.7.4 逻辑运算符……75
3.7.5 位运算符……75
3.7.6 移位运算符……76
3.7.7 一元运算符……78
3.7.8 三元运算符……78
3.7.9 逗号运算符……79
3.7.10 使用运算符时的常见问题……79
3.7.11 转换运算符……80
3.7.12 C++的显式转换……80
3.7.12.1 静态转换（**static_cast**）……81
3.7.12.2 常量转换（**const_cast**）……82
3.7.12.3 重解释转换（**reinterpret_cast**）82
3.7.13 **sizeof**——独立运算符……83
3.7.14 **asm** 关键字……84
3.7.15 显式运算符……84
3.8 创建复合类型……84
3.8.1 用**typedef**命名别名……85
3.8.2 用**struct**把变量结合在一起……85
3.8.2.1 指针和**struct**……87
3.8.3 用**enum**提高程度清晰度……87
3.8.3.1 枚举类型检查……88
3.8.4 用**union**节省内存……88

3.8.5 数组……89
3.8.5.1 指针和数组……91
3.8.5.2 探究浮点格式……93
3.8.5.3 指针算术……94
3.9 调试技巧……96
3.9.1 调试标记……96
3.9.1.1 预处理器调试标记……97
3.9.1.2 运行期调试标记……97
3.9.2 把变量和表达式转换成字符串……98
3.9.3 C语言**assert()**宏……98
3.10 函数地址……99
3.10.1 定义函数指针……99
3.10.2 复杂的声明和定义……99
3.10.3 使用函数指针……100
3.10.4 指向函数的指针数组……101
3.11 **make**：管理分段编译……101
3.11.1 **make**的行为……102
3.11.1.1 宏……102
3.11.1.2 后缀规则……103
3.11.1.3 默认目标……103
3.11.2 本书中的**makefile**……104
3.11.3 **makefile**的一个例子……104
3.12 小结……106
3.13 练习……106
第4章 数据抽象……109
4.1 一个袖珍C库……109
4.1.1 动态存储分配……112
4.1.2 有害的猜测……114
4.2 哪儿出问题……115
4.3 基本对象……116
4.4 什么是对象……120
4.5 抽象数据类型……121
4.6 对象细节……121
4.7 头文件形式……122
4.7.1 头文件的重要性……122
4.7.2 多次声明问题……123
4.7.3 预处理器指示**#define**、**#ifdef**和**#endif**……124
4.7.4 头文件的标准……124
4.7.5 头文件中的名字空间……125
4.7.6 在项目中使用头文件……125
4.8 嵌套结构……126
4.8.1 全局作用域解析……128
4.9 小结……129
4.10 练习……129
第5章 隐藏实现……132
5.1 设置限制……132
5.2 C++的访问控制……132
5.2.1 **protected**说明符……134
5.3 友元……134
5.3.1 嵌套友元……136
5.3.2 它是纯面向对象的吗……138
5.4 对象布局……138
5.5 类……139
5.5.1 用访问控制来修改**Stash**……141
5.5.2 用访问控制来修改**Stack**……141
5.6 句柄类……142
5.6.1 隐藏实现……142
5.6.2 减少重复编译……142
5.7 小结……144
5.8 练习……144
第6章 初始化与清除……146
6.1 用构造函数确保初始化……146
6.2 用析构函数确保清除……147
6.3 清除定义块……149
6.3.1 **for**循环……150
6.3.2 内存分配……151
6.4 带有构造函数和析构函数的**Stash**……152
6.5 带有构造函数和析构函数的**Stack**……154
6.6 集合初始化……156
6.7 默认构造函数……158
6.8 小结……159
6.9 练习……159
第7章 函数重载与默认参数……161
7.1 名字修饰……162
7.1.1 用返回值重载……162
7.1.2 类型安全连接……162
7.2 重载的例子……163

7.3 联合……166
7.4 默认参数……168
7.4.1 占位符参数……169
7.5 选择重载还是默认参数……170
7.6 小结……173
7.7 练习……173
第8章 常量……175
8.1 值替代……175
8.1.1 头文件里的const……176
8.1.2 const的安全性……176
8.1.3 集合……177
8.1.4 与C语言的区别……177
8.2 指针……178
8.2.1 指向const的指针……179
8.2.2 const指针……179
8.2.2.1 格式……180
8.2.3 赋值和类型检查……180
8.2.3.1 字符数组的字面值……180
8.3 函数参数和返回值……181
8.3.1 传递const值……181
8.3.2 返回const值……181
8.3.2.1 临时量……183
8.3.3 传递和返回地址……183
8.3.3.1 标准参数传递……185
8.4 类……185
8.4.1 类里的const……186
8.4.1.1 构造函数初始化列表……186
8.4.1.2 内部类型的“构造函数”……187
8.4.2 编译期间类里的常量……188
8.4.2.1 旧代码中的“enum hack”……189
8.4.3 const对象和成员函数……190
8.4.3.1 可变的：按位const和按逻辑const……192
8.4.3.2 只读存储能力……193
8.5 volatile……194
8.6 小结……195
8.7 练习……195
第9章 内联函数……197
9.1 预处理器的缺陷……197
9.1.1 宏和访问……199
9.2 内联函数……200
9.2.1 类内部的内联函数……200
9.2.2 访问函数……201
9.2.2.1 访问器和修改器……202
9.3 带内联函数的Stash和Stack……205
9.4 内联函数和编译器……208
9.4.1 限制……209
9.4.2 向前引用……209
9.4.3 在构造函数和析构函数里隐藏行为…210
9.5 减少混乱……210
9.6 预处理器的更多特征……211
9.6.1 标志粘贴……212
9.7 改进的错误检查……212
9.8 小结……215
9.9 练习……215
第10章 名字控制……217
10.1 来自C语言中的静态元素……217
10.1.1 函数内部的静态变量……217
10.1.1.1 函数内部的静态对象……218
10.1.1.2 静态对象的析构函数……219
10.1.2 控制连接……220
10.1.2.1 冲突问题……221
10.1.3 其他存储类型说明符……222
10.2 名字空间……222
10.2.1 创建一个名字空间……222
10.2.1.1 未命名的名字空间……223
10.2.1.2 友元……224
10.2.2 使用名字空间……224
10.2.2.1 作用域解析……224
10.2.2.2 使用指令……225
10.2.2.3 使用声明……226
10.2.3 名字空间的使用……227
10.3 C++中的静态成员……228
10.3.1 定义静态数据成员的存储……228
10.3.1.1 静态数组的初始化……229
10.3.2 嵌套类和局部类……231
10.3.3 静态成员函数……232
10.4 静态初始化的相依性……234

10.4.1 怎么办……235
10.4.1.1 技术一……235
10.4.1.2 技术二……237
10.5 替代连接说明……240
10.6 小结……240
10.7 练习……241
第11章 引用和拷贝构造函数……244
11.1 C++中的指针……244
11.2 C++中的引用……244
11.2.1 函数中的引用……245
11.2.1.1 常量引用……246
11.2.1.2 指针引用……246
11.2.2 参数传递准则……247
11.3 拷贝构造函数……247
11.3.1 按值传递和返回……247
11.3.1.1 传递和返回大对象……248
11.3.1.2 函数调用栈框架……248
11.3.1.3 重入……249
11.3.1.4 位拷贝与初始化……249
11.3.2 拷贝构造函数……251
11.3.2.1 临时对象……254
11.3.3 默认拷贝构造函数……255
11.3.4 替代拷贝构造函数的方法……256
11.3.4.1 防止按值传递……257
11.3.4.2 改变外部对象的函数……257
11.4 指向成员的指针……257
11.4.1 函数……259
11.4.1.1 一个例子……259
11.5 小结……261
11.6 练习……261
第12章 运算符重载……264
12.1 两个极端……264
12.2 语法……264
12.3 可重载的运算符……265
12.3.1 一元运算符……266
12.3.1.1 自增和自减……269
12.3.2 二元运算符……269
12.3.3 参数和返回值……278
12.3.3.1 作为常量通过传值方式返回……279
12.3.3.2 返回值优化……279
12.3.4 不常用的运算符……280
12.3.4.1 **operator,**……280
12.3.4.2 **operator->**……280
12.3.4.3 嵌入的迭代器……282
12.3.4.4 **operator->***……284
12.3.5 不能重载的运算符……285
12.4 非成员运算符……286
12.4.1 基本方针……287
12.5 重载赋值符……287
12.5.1 **operator=**的行为……288
12.5.1.1 类中指针……289
12.5.1.2 引用计数……291
12.5.1.3 自动创建**operator=**……295
12.6 自动类型转换……296
12.6.1 构造函数转换……296
12.6.1.1 阻止构造函数转换……297
12.6.2 运算符转换……297
12.6.2.1 反身性……298
12.6.3 类型转换例子……299
12.6.4 自动类型转换的缺陷……300
12.6.4.1 隐藏的行为……301
12.7 小结……302
12.8 练习……302
第13章 动态对象创建……305
13.1 对象创建……305
13.1.1 C从堆中获取存储单元的方法……306
13.1.2 **operator new**……307
13.1.3 **operator delete**……307
13.1.4 一个简单的例子……308
13.1.5 内存管理的开销……308
13.2 重新设计前面的例子……309
13.2.1 使用**delete void***可能会出错……309
13.2.2 对指针的清除责任……310
13.2.3 指针的**Stash**……310
13.2.3.1 一个测试程序……312
13.3 用于数组的**new**和**delete**……314
13.3.1 使指针更像数组……315
13.4 耗尽内存……315

13.5 重载new和delete ……316
13.5.1 重载全局new和delete ……317
13.5.2 对于一个类重载new和delete ……318
13.5.3 为数组重载new和delete ……320
13.5.4 构造函数调用……322
13.5.5 定位new和delete ……323
13.6 小结 ……324
13.7 练习 ……324
第14章 继承和组合 ……326
14.1 组合语法 ……326
14.2 继承语法 ……327
14.3 构造函数的初始化表达式表 ……329
14.3.1 成员对象初始化……329
14.3.2 在初始化表达式表中的内部类型…329
14.4 组合和继承的联合 ……330
14.4.1 构造函数和析构函数调用的次序…331
14.5 名字隐藏 ……333
14.6 非自动继承的函数 ……336
14.6.1 继承和静态成员函数……339
14.7 组合与继承的选择 ……339
14.7.1 子类型设置 ……340
14.7.2 私有继承……342
14.7.2.1 对私有继承成员公有化 ……342
14.8 protected ……343
14.8.1 protected继承……343
14.9 运算符的重载与继承 ……343
14.10 多重继承……345
14.11 渐增式开发……345
14.12 向上类型转换……346
14.12.1 为什么要“向上类型转换”……347
14.12.2 向上类型转换和拷贝构造函数 …347
14.12.3 组合与继承（再论）……349
14.12.4 指针和引用的向上类型转换 ……350
14.12.5 危机 ……350
14.13 小结……351
14.14 练习……351
第15章 多态性和虚函数 ……354
15.1 C++程序员的演变 ……354
15.2 向上类型转换 ……355
15.3 问题 ……356
15.3.1 函数调用捆绑……356
15.4 虚函数 ……356
15.4.1 扩展性……357
15.5 C++如何实现晚捆绑 ……359
15.5.1 存放类型信息……360
15.5.2 虚函数功能图示……361
15.5.3 撩开面纱……362
15.5.4 安装vpointer ……363
15.5.5 对象是不同的……363
15.6 为什么需要虚函数 ……364
15.7 抽象基类和纯虚函数 ……365
15.7.1 纯虚定义……368
15.8 继承和VTABLE ……368
15.8.1 对象切片……370
15.9 重载和重新定义 ……372
15.9.1 变量返回类型……373
15.10 虚函数和构造函数……375
15.10.1 构造函数调用次序 ……375
15.10.2 虚函数在构造函数中的行为 ……376
15.11 析构函数和虚拟析构函数……376
15.11.1 纯虚析构函数 ……378
15.11.2 析构函数中的虚机制 ……379
15.11.3 创建基于对象的继承 ……380
15.12 运算符重载……382
15.13 向下类型转换……384
15.14 小结……386
15.15 练习……387
第16章 模板介绍 ……390
16.1 容器 ……390
16.1.1 容器的需求……391
16.2 模板综述 ……392
16.2.1 模板方法……393
16.3 模板语法 ……394
16.3.1 非内联函数定义……395
16.3.1.1 头文件 ……396
16.3.2 作为模板的IntStack ……396
16.3.3 模板中的常量……398
16.4 作为模板的Stash和Stack ……399

16.4.1 模板化的指针**Stash** ……401
16.5 打开和关闭所有权 ……405
16.6 以值存放对象 ……407
16.7 迭代器简介 ……408
16.7.1 带有迭代器的栈……415
16.7.2 带有迭代器的**PStash**……417
16.8 为什么使用迭代器 ……422
16.8.1 函数模板……424
16.9 小结 ……425
16.10 练习……425
附录A 编码风格 ……428
A.1 常规 ……428
A.2 文件名 ……428
A.3 开始和结束注释标记 ……429
A.4 圆括号、大括号和缩排 ……429
A.5 标识符名 ……432
A.6 头文件的包含顺序 ……432
A.7 在头文件上包含警卫 ……432
A.8 使用名字空间 ……433
A.9 **require( )**和**assure( )**的使用……433
附录B 编程准则 ……434
附录C 推荐读物 ……441
C.1 C……441
C.2 基本C++……441
C.2.1 我自己的书 ……441
C.3 深入研究和死角分析 ……442
C.4 分析和设计 ……443
索引……445

第1章 对象导言

计算机革命起源于一台机器。因此，程序设计语言的起源看上去也起源于那台机器。 21

然而，计算机并不仅仅是一台机器，因为它们就像是智力放大工具（Steve Jobs[⊖]喜欢称其为“智力的自行车”）和另一种富于表现力的媒体。所以，它们渐渐地不太像机器，而更像我们大脑的一部分了，它们还挺像其他的一些富于表现力的媒体（例如写作、绘画、雕刻、动画或电影制作）。面向对象程序设计是“使计算机成为一种富于表现力的媒体”这一华彩乐章的一部分。

本章将介绍面向对象程序设计（OOP）的基本概念，包括OOP开发方法的概述。在读者阅读本书之前，我们假设读者已经有了使用过程型程序设计语言的经验，当然不一定是C语言。如果读者认为在学习本书之前需要补习程序设计方面的预备知识和C的语法知识，可以参考本书附带的光盘“Thinking in C: Foundations for C++ and Java”，这些内容也可以在*www.BruceEckel.com*中找到。

本章是一些背景和辅助材料。如果在未了解这些大背景之前就进入面向对象程序设计，许多人都会感到有困难。因此，这里为读者预先介绍有关OOP的一些基本概念。但是，还有另一些读者，在不见到某些具体结构之前，就不能理解语言的整体概念；这些人不接触某些代码就会停止不前和不知所措。如果读者属于后者，急于学习这门语言的具体内容，则可以跳过本章，这样不会妨碍写程序和学习这门语言。但是，读者以后终将回过头来补充本章的知识，这样才能理解为什么对象如此重要，以及如何用对象设计程序。

1.1 抽象的过程

所有的程序设计语言都提供抽象。可以说，人们能解决的问题的复杂性直接与抽象的类型和质量有关。这里“类型”指的是“要抽象的东西”。汇编语言是对底层机器的小幅度抽象。其 22
后的许多所谓“命令式”语言（例如Fortran、BASIC和C）都是对汇编语言的抽象。这些语言较之汇编语言有了很大的改进，但是它们的主要抽象仍然要求程序员按计算机的结构去思考，而不是按要解决的问题的结构去思考。程序员必须在机器模型（在“解空间”，即建模该问题的空间中，例如在计算机中）和实际上要解决的问题的模型（在“问题空间”，即问题存在的空间中）之间建立联系。程序员必须努力进行这之间的对应，这实际上是程序设计语言之外的任务，它使得程序难以编写且维护费用昂贵，并且作为一种副效应，造就了整个“程序设计方法”产业。

取代对机器建模的另一种方式是对要解决的问题进行建模。像LISP和APL这样的早期语言选取了特殊的“世界观”（“所有的问题最终都是表”或“所有的问题都是算法”），PROLOG则把所有的问题都看做决策链。还有一些语言，创造它们的目的是为了基于约束的编程和专门用于通过绘图符号来编程（后者已被证明局限太大）。这些方法中的每一种对于它所针对的特定问题都是很好的解决方案，但对于这个领域之外的问题，就笨拙难用了。

⊖ 史蒂夫·乔布斯（Steve Jobs）是苹果公司的创始人和精神领袖。——编辑注

面向对象的方法为程序员提供了在问题空间中表示各种事物元素的工具，从而向前迈进了一大步。这种表示方法是通用的，并不限定程序员只处理特定类型的问题。我们把问题空间中的事物和它们在解空间中的表示称为“对象”（当然，还需要另外一些对象，它们在问题空间中没有对应物）。其思想是允许程序通过添加新的对象类型，而使程序本身能够根据问题的术语进行调整，这样当我们读描述解决方案的代码时，也就是在读表达该问题的文字。这
23 是比以前更灵活和更强大的语言抽象。因此，OOP允许程序员用问题本身的术语来描述问题，而不是用要运行解决方案的计算机的术语来描述问题。当然这些问题的术语仍然与计算机有联系。每个对象看上去像一台小计算机，它有状态，有可以执行的运算。这似乎是现实世界中对象的很好类比，它们都有特性和行为。

一些语言的设计者认为，面向对象程序设计本身并不足以轻易解决所有程序设计问题，他们提倡结合各种方法，形成多范型（*multiparadigm*）的程序设计语言[⊖]。

Alan Kay总结了Smalltalk的五个基本特性，这些特性代表了纯面向对象程序设计的方法。Smalltalk是第一个成功的面向对象语言，是C++的基础语言之一。

(1) **万物皆对象**。对象可以被认为是一个奇特的变量，它能存放数据，而且可以对它“提出请求”，要求它执行对它自身的运算。理论上，我们可以在需要解决的问题中取出任意概念性的成分（狗、建筑物、服务等），把它表示为程序中的对象。

(2) **程序就是一组对象，对象之间通过发送消息互相通知做什么**。更具体地讲，可以将消息看做是对于调用某个特定对象所属函数的请求。

(3) **每一个对象都有它自己的由其他对象构成的存储区**。这样，就可以通过包含已经存在的对象创造新对象。因此，程序员可以构造出复杂的程序，而且能将程序的复杂性隐藏在对
24 象的简明性背后。

(4) **每个对象都有一个类型**。采用OOP术语，每个对象都是某个类的实例（instance），其中“类”（class）与“类型”（type）是同义词。类的最重要的突出特征是“能向它发送什么消息”。

(5) **一个特定类型的所有对象都能接收相同的消息**。正如后面将看到的，这实际上是一条装载语句。因为一个“circle”类型的对象也是一个“shape”类型的对象，所以保证circle能接收shape消息。这意味着，我们可以编写与shape通信的代码，该代码能自动地对符合shape描述的任何东西进行处理。这种替换能力（*substitutability*）是OOP的最强大的思想之一。

1.2 对象有一个接口

亚里士多德可能是第一个认真研究类型（*type*）概念的人，他提到了“鱼类和鸟类”。所有对象（虽然都具有惟一性）都是一类对象中的一员，它们有共同的特征和行为。这一思想在第一个面向对象语言Simula-67中得到了直接的应用，该语言用基本关键字**class**在程序中引入新类型。

顾名思义，创造Simula的目的是为了解决模拟问题，例如著名的“银行出纳员问题”[⊖]。其中包括出纳员、顾客、账户、交易、货币的单位等大量的“对象”。把那些在程序执行期间的状态之外其他方面都一样的对象归为“几类对象”，这就是关键字**class**（类）的来源。创建抽象数据类型是面向对象程序设计的基本思想。抽象数据类型几乎能完全像内部类型一样工

⊖ 参见Timothy Budd所写的专著《*Multiparadigm Programming in Leda*》(Addison-Wesley, 1995)。

⊖ 可以在本书的第2卷中找到这一问题的有趣实现，本书的第2卷可从www.BruceEckel.com上找到。

作。程序员可以创建类型的变量［面向对象的说法称为对象（*object*）或实例（*instance*）］ 25
和操纵这些变量（称为发送消息或请求，对象根据发来的消息推断需要做什么事情）。每个类的成员（元素）都有共性：每个账户有余额，每个出纳员都能接收存款，等等。同时，每个成员都有自己的状态，每个账户有不同的余额，每个出纳员都有名字。这样，在计算机程序中，出纳员、客户、账户、交易等，每一个都被描述为惟一的实体。这个实体就是对象，每个对象都属于一个定义了它的特性和行为的特定类。

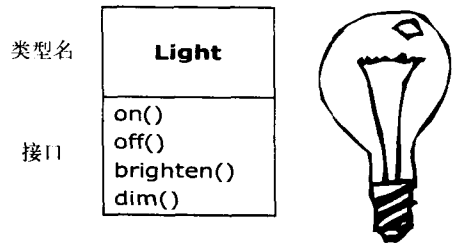
所以，虽然在面向对象的程序设计中，我们所做的工作实际上是创造新数据类型，但事实上所有的面向对象的程序设计语言都使用关键字“class”。当碰到“类型（type）”时可以看做“类（class）”，反之亦然[⊖]。

由于类描述了一组有相同特性（数据元素）和相同行为（功能）的对象，因此类实际上就是数据类型，例如浮点数也有一组特性和行为。区别在于，程序员定义类是为了与具体问题相适应，而不是被迫使用已存在的数据类型，而设计这些已存在的数据类型的动机是为了表示机器中的存储单元。程序员可以通过增添专门针对自己需要的新数据类型来扩展程序设计语言。这种程序设计系统欢迎新的类，关注新的类，对它们进行与内置类型一样的类型检查。

面向对象方法并不限于模拟创建。无论我们是否同意，都不能否认任何程序都是我们正在设计的系统的一种模拟，OOP技术的使用确实可以容易地将大量问题缩减为一个简单的解决方案。

一旦建立了一个类，程序员想制造这个类的多少个对象就可以制造多少个，然后操作这些对象，就如同它们是所解决的问题中的元素。实际上，面向对象程序设计的难题之一，是在问题空间中的元素和解空间的对象之间建立一对一的映射。 26

但是，我们如何得到一个对象去为我们做有用工作呢？必须有一种方法能向对象作出请求，使得它能做某件事情，例如完成交易、在屏幕上画图或打开开关。每个对象只能满足特定的请求。可以向对象发出的请求是由它的接口（*interface*）定义的，而接口由类型确定。一个简单的例子可能该是电灯泡的表示了。




```
Light lt;
lt.on();
```

接口规定我们能向特定的对象发出什么请求。然而，必须有代码满足这种请求，再加上隐藏的数据，就组成了实现（*implementation*）。从过程型程序设计的观点看，这并不复杂。类型对每个可能的请求都有一个相关的函数，当向对象发请求时，就调用这个函数。这个过程通常概括为向对象“发送消息”（提出请求），对象根据这个消息确定做什么（执行代码）。

如上例，这个类型或称类的名字是**Light**，这个特定的**Light**对象的名字是**lt**，可以对**Light**对象提出一些请求：打开它、关闭它、使它变亮或变暗。通过声明一个名字（**lt**），可以

⊖ 一些人对此做了区分，他们认为类型（type）确定接口，而类（class）是对这个接口的特定实现。

创建一个**Light**对象。为了向这个对象发送消息，可以说出这个对象的名字，并用句点连接对消息的请求。从使用预定义类的用户的角度看，用对象编程只需要这些工作就行了。

上图符合统一建模语言（Unified Modeling Language, UML）的格式，每个类由一个方框
27 表示，这个方框的顶部标有类型名，中间部分列出所关注的数据成员，底部是成员函数（member function属于这个对象的函数，它们能接收发送给这个对象的任何消息）。通常，只有类的名字和公共成员函数会在UML设计图中表示出来，所以中间部分不显示。如果只对类的名字感兴趣，则底部也不需要显示。

1.3 实现的隐藏

把程序员划分为类创建者（创建新数据类型的人）和客户程序员[⊖]（在应用程序中使用数据类型的类的用户）是有益的。客户程序员的目标是去收集一个装满类的工具箱，用于快速应用开发。类创建者的目标是去建造类，这个类只暴露对于客户程序员是必需的东西，其他的都隐藏起来。为什么呢？因为如果是隐藏的东西，客户程序员就不能使用它，这意味着，这个类的创建者可以改变隐藏的部分，而不用担心会影响其他人。被隐藏的部分通常是对象内部的管理功能，容易受到粗心或无知的客户程序的损害，所以隐藏实现会减少程序错误。隐藏的概念怎么强调都不过分。

在任何关系中，存在一个所有的参与者都遵从的边界是重要的。当我们创建一个库时，也就与客户程序员建立了关系。他们也是程序员，但是他们是通过使用我们的库来组装应用程序，也可能建立更大的库。

如果类的所有成员对于任何人都能用，那么客户程序员就可以用这个类做其中的任何事
28 情，不存在强制规则。虽然我们可能实际上不希望客户程序员直接操纵这个类的某些成员，但是因为没有访问控制，所以就没有办法保护它，所有的东西都暴露无遗。

访问控制的第一个理由是为了防止客户程序员插手他们不应当接触的部分，也就是对于数据类型的内部实施方案是必需的部分，而不是用户为了解决他们的特定问题所需要的接口部分。这实际上是对用户的很好服务，因为这使得他们容易看到哪些部分对于他们是重要的，哪些部分是可以忽略的。

访问控制的第二个理由是允许库设计者去改变这个类的内部工作方式，而不必担心这样做会影响客户程序员。例如，库设计者可能为了容易开发而用简单的方法实现了一个特殊的类，但后来发现需要重写这个类以使得它运行得更快。如果接口和实现是严格分离和被保护的，那么库设计者就可以很容易完成重写任务，用户只需要重新连接就可以了。

C++语言使用了三个明确的关键字来设置类中的边界：**public**、**private**和**protected**。它们的使用和含义相当简明。这些访问说明符（*access specifier*）确定谁能用随后的定义。**public**意味着随后的定义对所有人都可用。相反，**private**关键字则意味着，除了该类型的创建者和该类型的内部成员函数之外，任何人都不能访问这些定义。**private**在我们与客户程序员之间筑起了一道墙。如果有人试图访问一个私有成员，就会产生一个编译时错误。**protected**与**private**基本相似，只有一点不同，即继承的类可以访问**protected**成员，但不能访问**private**成员。我们将在稍后部分介绍继承。

⊖ 对于这个术语，我要感谢我的朋友Scott Meyers（名著《Effective C++》的作者——编辑注）。

1.4 实现的重用

创建了一个类并进行了测试后，这个类（理论上）就成了有用的代码单元。重用性并不像许多人所希望的那样容易达到，产生一个好设计需要经验和洞察力。但是只要有了这样的设计，就应该提供重用。代码重用是面向对象程序设计语言的最大优点之一。 29

重用一个类最简单的方法就是直接使用这个类的对象，并且还可以将这个类的对象放到一个新类的里面。我们称之为“创建一个成员对象”。可以用任何数量和类型的其他对象组成新类，通过组合得到新类所希望的功能。因为这是由已经存在的类组成新类，所以称为组合（*composition*）［或者更通常地称为聚合（*aggregation*）］。组合常常被称为“has-a（有）”关系，比如“在小汽车中有发动机”一样。



（上面的UML图用实心菱形表示组合，说明这里有一个小汽车。通常我将采用一种更简单的形式，即仅用一条不带菱形的线来表示一个关联。⊖）

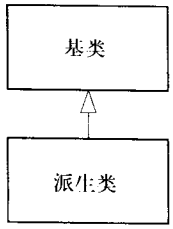
组合具有很大的灵活性，新类的成员对象通常是私有的，使用这个类的客户程序员不能接触它们。这种特点允许我们改变这些成员而不会干扰已存在的客户代码。我们还可以在运行时改变这些成员对象，动态地改变程序的行为。下面将介绍的继承没有这种灵活性，因为编译器必须在用继承方法创造的类上加入编译时限制。

因为继承在面向对象的程序设计中很重要，所以它常常得到高度重视，并且新程序员可能会产生在任何地方都使用继承的想法。这会形成拙笨和极度复杂的设计。实际上，当创建新类时，程序员应当首先考虑组合，因为它更简单和更灵活。如果采用组合的方法，设计将变得清晰。一旦我们具备一些经验之后，就能很明显地知道什么时候需要采用继承方法。 30

1.5 继承：重用接口

对象的思想本身是一种很方便的工具。我们可以将数据和功能通过概念封装在一起，使得我们能描述合适的问题空间思想，而不是被强制使用底层机器的用语。通过使用class关键字，这些概念被表示为程序设计语言中的基本单元。

然而，克服许多困难去创造一个类，并随后强制性地创造一个有类似功能的全新的类，似乎并不是一种很好的方法。如果能选取已存在的类，克隆它，然后对这个克隆增加和修改，则是再好不过的事。这是继承（*inheritance*）带来的好处，缺点是，如果原来的类（称为基类、超类或父类）被修改，则这个修改过的“克隆”（称为派生类、继承类或子类）也会表现出这些改变。



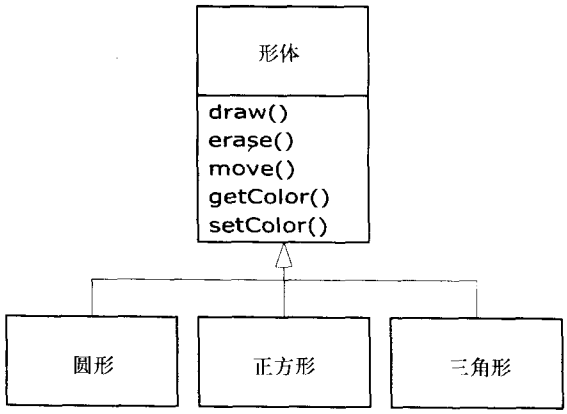
⊖ 这种形式对于大部分图通常是足够详细的，不需要特别指出何处使用聚合或称组合。

（在上面的UML图中，箭头从派生类指向基类。正如你将会看到的，可以有多个派生类。）

类型不仅仅描述一组对象上的约束，而且还描述与其他类型之间的关系。两个类型可以有共同的特性和行为，但是一个类型可以包括比另一个类型更多的特性，也可以处理更多的
31 消息（或对消息进行不同的处理）。继承表示了在基类型和派生类型之间的这种相似性。一个基类型具有所有由它派生出来的类型所共有的特性和行为。程序员创建一个基类型以描述关于系统中的一些对象的思想核心。由这个基类型，我们可以派生出其他类型来表述实现该核心的不同途径。

例如，垃圾再生机器要对垃圾进行分类。这里基类型是“垃圾”，每件垃圾有重量、价值等，并且可以被破碎、被融化或被分解。这样，可以派生出更特殊的垃圾类型，它们可以有另外的特性（瓶子有颜色）或行为（铝可以被压碎，钢可以带有磁性）。另外，有些行为可以不同（纸的价值取决于它的种类和情况）。使用继承，我们可以建立类型的层次结构，在该层次结构中用其类型术语来表述我们需要解决的问题。

第二个例子是经典的“形体”范例，可以用于计算机辅助设计系统或游戏模拟。在这里，基类型是“形体”，每个形体有大小、颜色、位置等。每个形体能被绘制、擦除、移动、着色等。由此，可以派生出特殊类型的形体：圆形、正方形、三角形等，它们中的每一个都有另外的特性和行为。例如，某些形体可以翻转。有些行为可以不同，形体面积的计算。类型层
32 次结构既体现了形体之间的相似性，又体现了它们之间的区别。

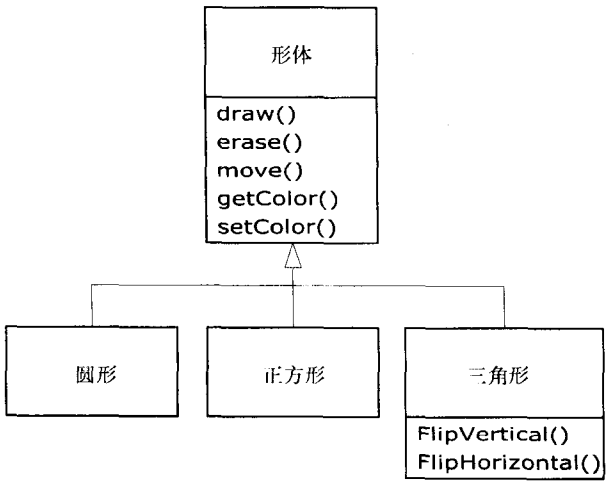


用与问题相同的术语描述问题的解是非常有益的，因为这样我们就无需在问题的描述和解的描述之间使用许多的中间模型。使用对象，类的层次结构就是最初的模型，所以能直接从实际世界中的系统描述进入代码中的系统描述。实际上，使用面向对象设计，人们的困难之一是从开始到结束过于简单。一个已经习惯于寻找复杂解的训练有素的头脑，往往会被问题本来的简单性所难住。

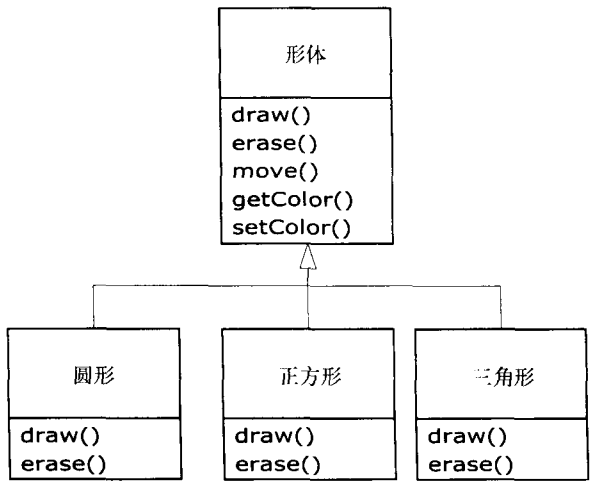
当我们从已经存在的类型来继承时，我们就创造了一个新类型。这个新类型不仅包含那个已经存在的类型的所有成员（虽然私有成员已被隐藏且不可访问），但更重要的是，它复制了这个基类的接口。也就是说，所有能够发送给这个基类对象的消息，也能够发送给这个派生类的对象。因为我们能够根据发送给一个类的消息知道这个类的类型，所以这意味着这个派生类与这个基类是相同类型的。在前面的例子中，“圆形是一个形体”。这种通过继承实现
33 类型等价性，是理解面向对象程序设计含义的基本途径之一。

由于基类和派生类有相同接口，因此伴随着接口必然有一些实现。也就是说，当对象接收到一个特定的消息后必定执行一些代码。如果只是简单地继承一个类，而不做其他任何事情，来自基类接口的方法也就进入了派生类。这就意味着，派生类的对象不仅有相同的类型，而且有相同的行为，这一点并不是特别有意义的。

有两种方法能使新派生类区别于原始基类。第一种相当直接，简单地向派生类添加全新的函数。这些新函数不是基类接口的一部分。这意味着，这个基类不能做我们希望它做的事情，所以必须添加函数。继承的这种简单和原始的运用有时就是问题的完美解。但是，我们会进一步看到，基类可能也需要这些添加的函数。在面向对象程序设计中，这种设计的发现和迭代过程会经常发生。



虽然继承有时意味着向接口添加新函数，但这未必真的需要。使新类有别于基类的第二个和更重要的方法是，改变已经存在的基类函数的行为，这称为重载（*overriding*）这个函数。 34

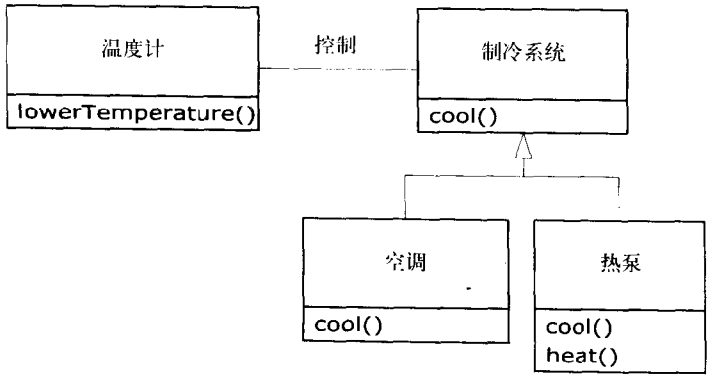


为了重载函数，可以简单地在派生类中创建新定义。相当于说："我正在使用同一个接口函数，但是我希望它为我的新类型做不同的事情。"

1.5.1 is-a关系和is-like-a关系

对于继承有一些争论。继承应当只重载基类（并且不添加基类中没有的新成员函数）吗？这就意味着派生类与基类是完全相同的类型，因为它们有相同的接口。结果是，我们可以用派生类的对象代替基类的对象。这被认为是纯代替（*pure substitution*），常常被称做代替原则（*substitution principle*）。在某种意义上，这是对待继承的理想方法。我们常把基类和派生类之间的关系看做是一个“*is-a*（是）”关系，因为我们可以说“圆形是一个形体”。对继承
35 的一种测试方法就是看我们是否可以说这些类有“*is-a*”关系，而且还有意义。

有时需要向一个派生类型添加新的接口元素，这样就扩展了接口并创建了新类型。这个新类型仍然可以代替这个基类，但这个代替不是完美的，因为这些新函数不能从基类访问。这可以描述为“is-like-a（像）”关系；新类型有老类型的接口，但还包含其他函数，所以不能说它们完全相同。以一台空调为例。假设你的房子与制冷的全部控制连线；也就是说，它有一个允许你控制冷却的接口。设想这台空调坏了，用一台热泵代替它，这台热泵既可以制冷又可以制热，这台热泵就像一台空调，但它能做更多的事情。因为你的房子的控制系统仅仅是针对制冷功能设计的，所以它仅限于与新对象的制冷部分通信。新对象的接口已经被扩展，而这个已经存在的系统只知道原来的接口，并不知道扩展的部分。



很显然，基类“制冷系统”是不充分的，应当改为“温度控制系统”，使它也能包含加热功能。在这一点上，代替原则可用。上图是一个例子，它既可以发生在设计过程中，也可以发生在现实世界中。

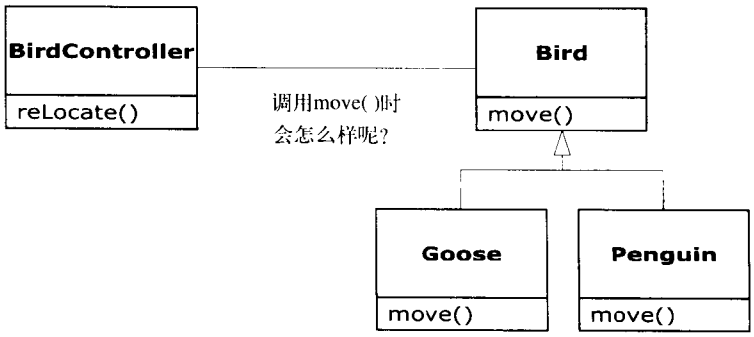
36 当我们考虑代替原则，很容易将这种方法（纯代替）看做是做事情的惟一方法。实际上，如果我们的设计能够采用这种方法，效果也很好。但是，我们还发现有时必须向派生类的接口添加新函数。通过考察，我们发现两种情况都很常见。

1.6 具有多态性的可互换对象

当处理类型层次结构时，程序员常常希望不把对象看做是某一特殊类型的成员，而是想把它看做是其基本类型的成员，这样就允许程序员编写不依赖于特殊类型的程序代码。在形体的例子中，函数可以对一般形体进行操作，而不关心它们是圆形、正方形还是三角形。所有的形体都能被绘制、擦除和移动，所以这些函数能简单地发送消息给一个形体对象，而不考虑这个对象如何处理这个消息。

这样，程序代码不受增添新类型的影响，而且增添新类型是扩展面向对象程序来处理新情况最普通的方法。例如，可以派生出形体的一个新的子类型，称为五边形，而不必修改那些处理一般形体的函数。通过派生新的子类型，可以很容易扩展程序，这个能力很重要，因为这会在降低软件维护费用的同时，极大地改善软件设计。

然而，如果试图把派生类型的对象看做是比它们自身更一般的基本类型（圆形看做形体，自行车看做车辆，鸬鹚看做鸟），这里就有一个问题：如果一个函数告诉一个一般的形体去绘制它自己，或者告诉一个一般的车辆去行驶，或者告诉一只一般的鸟去飞翔，则编译器在编译时就不能确切地知道应当执行哪段代码。同样的问题是，消息发送时，程序员并不想知道将执行哪段代码。绘图函数能等同地应用于圆形、正方形或三角形，对象根据它的特殊类型来执行合适的代码。如果增加一个新的子类型，不用修改函数调用，它就可以执行不同的代码。编译器不能确切地知道执行哪段代码，那么它应该怎么办呢？例如，在下图中，**BirdController**对象只是与一般的**Bird**对象交互，并不知道它们到底是什么类型。这对于**BirdController**是方便的，因为不需要编写专门的代码来确定它正在对哪种**Bird**工作以及它有什么样的行为。但当忽略专门的**Bird**类型而调用**move()**时，将发生什么事情呢？会出现正确的行为吗？（**Goose**是跑、是飞、还是游泳？**Penguin**是跑、还是游泳？） 37



在面向对象的程序设计中，答案是非常新奇的：编译器并不做传统意义上的函数调用。由非OOP编译器产生的函数调用会导致与被调用代码的*早捆绑*（*early binding*），对于这一术语，读者可能还没有听说过，因为从来没有想到过它。早捆绑的意思是，编译器会对特定的函数名产生调用，而连接器将这个调用解析为要执行代码的绝对地址。在OOP中，直到程序运行时，编译器才能确定执行代码的地址，所以，当消息被发送给一般对象时，需要采用其他的方案。

为了解决这一问题，面向对象语言采用*晚捆绑*（*late binding*）的思想。当给对象发送消息时，在程序运行时才去确定被调用的代码。编译器保证这个被调用的函数存在，并执行参数和返回值的类型检查［其中不采用这种处理方式的语言称为*弱类型*（*weakly typed*）语言］，但是它并不知道将执行的确切代码。 38

为了执行晚捆绑，C++编译器在真正调用的地方插入一段特殊的二进制代码。通过使用存放在对象自身中的信息，这段代码在运行时计算被调用函数函数体的地址（这一过程将在第15章中详细介绍）。这样，每个对象就能根据这段二进制代码的内容有不同的行为。当一个对象接收到消息时，它根据这个消息判断应当做什么。

我们可以用关键字**virtual**声明他希望某个函数有晚捆绑的灵活性。我们并不需要懂得**virtual**使用的机制，但是没有它，我们就不能用C++进行面向对象的程序设计。在C++中，

必须记住添加**virtual**关键字，因为根据规定，默认情况下成员函数不能动态捆绑。virtual函数（虚函数）可用来表示出在相同家族中的类具有不同的行为。这些不同是产生多态行为的原因。

考虑形体的例子，在本章的前面画出过类的家族（所有基于统一接口的类）。为了论述多态性，我们希望编写一段简单的代码，只涉及基类，不涉及类型的具体细节。这段代码是从特定类型信息中分离出来的，编写起来比较简单，理解起来也比较容易。如果有一个新的类型（例如**Hexagon**）通过继承添加进来，那么所写的这段代码将会适用于“**Shape**”的这个新类型，就像在已经存在的类型上使用一样。这样，程序就是*可扩展的*（*extensible*）。

如果用C++编写一个函数（很快，读者将会学习如何去写）：

```
void doStuff(Shape& s) {
  s.erase();
  // ...
  s.draw();
}
```

这个函数与任何**Shape**对话，所以它独立于它正在绘制或擦除的对象的特定类型（‘**&**’表示“取这个对象的地址，传给**doStuff( )**”，但是现在理解这些细节并不重要）。如果在程序
39 的其他部分使用**doStuff( )**函数：

```
Circle c;
Triangle t;
Line l;
doStuff(c);
doStuff(t);
doStuff(l);
```

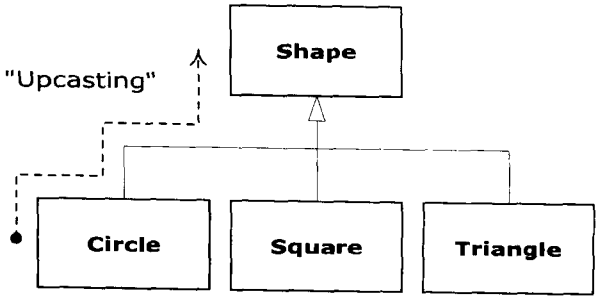
对**doStuff( )**的调用会自动正确工作，而不管调用对象的确切类型。

这真是一个令人惊讶的技术。想一想这行代码：

```
doStuff(c);
```

这里发生的事情是将**Circle**传递给对**Shape**有效的函数。因为**Circle**是一个**Shape**，所以可以由**doStuff( )**处理。这就是说，任何能由**doStuff( )**发送给**Shape**的消息，**Circle**都能接收。所以它是完全安全的和符合逻辑的。

我们把处理派生类型就如同处理其基类型的过程称为*向上类型转换*（*upcasting*）。“cast”一词来自铸造领域，“up”一词来自于继承图的典型排列方式，基类型置于顶层，派生类向下层展开。这样，类型向基类型的转换是沿继承图向上移动，即“向上类型转换”。



面向对象程序在一些地方会包含一些向上类型转换，因为这正是我们从必须了解所处理的是什么具体类型这一桎梏中解脱出来的方式。请看在**doStuff( )**中的代码：

```
s.erase();
// ...
s.draw();
```

40

注意，这可不是在说："如果是**Circle**，做这件事，如果是**Square**，做这件事，等等"。如果写这种代码，要检查**Shape**所有可能的类型，那将是很糟糕的，因为每次添加一种新的**Shape**，都必须改变代码。这里，我们只是在说："它是一种形体，我知道它能**erase( )**和**draw( )**自己，如法操作，注意细节的正确性。"

doStuff( )代码最引人注意的地方是它能做正确的事情。对**Circle**调用**draw( )**将执行的代码不同于对**Square**或**Line**调用**draw( )**所执行的代码。但是当**draw( )**的消息发送给一个匿名**Shape**时，正确行为的发生取决于这个**Shape**的实际类型。这是令人惊奇的，因为，如前所述，当C++编译器编译**doStuff( )**代码时，它并不知道它所处理对象的准确类型。所以以此类推，似乎最终调用的是**Shape**的**erase( )**和**draw( )**的版本，而不是特殊的**Circle**、**Square**或**Line**的版本。然而，因为多态性，一切操作都完全正确。编译器和运行系统可以处理这些细节，我们只需要知道它会这样做和知道如何用它设计程序就行了。如果一个成员函数是**virtual**的，则当我们给一个对象发送消息时，这个对象将做正确的事情，即便是在有向上类型转换的情况下。

1.7 创建和销毁对象

从技术角度，OOP的论域就是抽象数据类型、继承和多态性。但是，其他一些问题也是重要的。本节对这些问题给出综述。

特别重要的是对象创建和销毁的方法。对象的数据存放在何处？如何控制对象的生命期？不同的程序设计语言有不同的行事之道。C++采取的方法是把效率控制作为最重要的问题，所以它为程序员提供了一个选择。为了最大化运行速度，通过将对象存放在栈中或静态存储区域中，存储和生命期可以在编写程序时确定。栈是内存中的一个区域，可以直接由微处理器在程序执行期间存放数据。在栈中的变量有时称为*自动变量*（*automatic variable*）或*局部变量*（*scoped variable*）。静态存储区域简单说是内存的一个固定块，在程序开始执行以前分配。使 41
用栈或静态存储区，可以快速分配和释放，有时这是有价值的。然而，我们牺牲了灵活性，因为程序员必须在写程序时知道对象的准确数量、生命期和类型。如果程序员正在解决一个更一般的问题，例如计算机辅助设计、仓库管理或者空中交通控制，这就太受限制了。

第二种方法是在称为*堆*（*heap*）的区域动态创建对象。用这种方法，可以直到运行时还不知道需要多少个对象，它们的生命期是什么和它们的准确的数据类型是什么。这些决定是在程序运行之中作出的。如果需要新的对象，直接使用**new**关键字让它在堆上生成。当使用结束时，用关键字**delete**释放。

因为这种存储是在运行时动态管理的，所以在堆上分配存储所需要的时间比在栈上创建存储的时间长得多（在栈上创建存储常常只是一条向下移动栈指针的微处理器指令，另外一条是移回指令）。动态方法做出了一般性的逻辑假设，即对象趋向于更加复杂，所以，为找出存储和释放这个存储的额外开销对于对象的创建没有重要的影响。另外，对于解决一般性的程序设计问题，最大的灵活性是主要的。

另一个问题是对象的生命期。如果在栈上或在静态存储上创建一个对象，编译器决定这个对象持续多长时间并能自动销毁它。然而，如果在堆上创建它，编译器则不知道它的生命期。在C++中，程序员必须编程决定何时销毁此对象。然后使用**delete**关键字执行这个销毁任务。作
42 为一个替换，运行环境可以提供一个称为*垃圾收集器*（*garbage collector*）的功能，当一个对象不再被使用时此功能可以自动发现并销毁这个对象。当然，使用垃圾收集器编写程序是非常方便的，但是它需要所有应用软件能忍受垃圾收集器的存在及垃圾收集的系统开销。这并不符合C++语言的设计需要，因此C++没有包括它，尽管存在用于C++的第三方垃圾收集器。

1.8 异常处理：应对错误

从程序设计语言出现开始，错误处理就是最重要的问题之一。因为设计一个好的错误处理方案非常困难，许多语言忽略这个问题，将这个问题转交给库的设计者，而库的设计者往往采取不彻底的措施，即可以在许多情况下起作用，但很容易被绕开，通常是被忽略。大多数错误处理方案的一个主要问题，在于一厢情愿地认为程序员会小心地遵循一些语言本身并不强制要求而是商定好的规范。如果程序员不够小心（这种情况在他们非常匆忙的时候经常发生），这些方案就会很容易被忘记。

异常处理（*exception handling*）将错误处理直接与程序设计语言甚至有时是操作系统联系起来。异常是一个对象，它在出错的地方被抛出，并且被一段用以处理特定类型错误的异常处理代码（exception handler）所接收。异常处理似乎是另一个并行的执行路径，在出错的时候被调用。由于它使用一个单独的执行路径，它不需要干涉正常的执行代码。因为不需经常检查错误，代码可以很简洁。另外，一个抛出的异常并不同于一个由函数返回的错误值或为了指出错误条件而由函数设置的标记，后两者可以被忽略，而异常不能被忽略，必须保证它们在某些点上进行处理。最后，异常提供了一个从错误状态中进行可靠恢复的方法。除了从这个程序中退出以外，我们常常还可以作出正确的设置，并且恢复程序执行，这有助于产
43 生更健壮的系统。

异常处理并不是面向对象的一个特性，尽管在面向对象语言中异常通常用一个对象表示。异常处理的出现早于面向对象语言。

异常处理在本卷中介绍得很少且很少使用；第2卷详尽讨论了异常处理。

1.9 分析和设计

面向对象范例是一种新的异于前辈的编程思考方式，许多人一开始在学习如何处理一个OOP项目时都会感到非常困难。但是了解到任何事物都被认为是对象，并且学会用面向对象的风格去进一步思考之后，我们就可以开始利用OOP所提供的所有优点创造出“好的”设计。

方法（*method*）［通常称为*方法论*（*methodology*）］是一系列的过程和探索，用以降低程序设计问题的复杂性。自从面向对象程序设计出现，已有许多OOP方法被提出来。本节将让读者感受使用一种方法时应完成什么任务？

尤其在OOP中，方法论是一个充满实验的领域，因此在我们考虑采用一个方法之前，理解它试图要解决什么问题是重要的。这在C++中尤其正确，这种编程语言在表达一个程序时试图减少复杂性（同C相比）。这在实际上可能减缓对更复杂方法论的需求。相反，简单的方法可以满足在C++中处理更大类的问题，在过程型语言中用简单方法处理的问题相比起来则小很多。

认识到术语“方法论”通常太大且承诺太多也是很重要的。设计和编写一个程序时，我们所做的一切就是一个方法。它可能是我们自创的方法，我们可能没有意识到正在创造一种方法，但它确实是我们创造时经历的一个过程。如果它是一个有效的过程，只需要略加调整 44
以和C++配合。如果我们对自己的效率和程序生产方式不满意，就可以考虑采纳一个正式的方法或在许多正式方法中选择某些部分。

经历开发过程时，最重要的问题是：不要迷路。这很容易做到。大部分分析和设计方法都是为了解决最大的一些问题。记住，大多数项目并不适合这一点，因此我们通常可以用一个相对小的子集成功地进行分析和设计[⊖]。但是采用某种过程，不论它怎么有局限，总比一上来就直接编码好得多。

在开发过程中很容易受阻，陷入“分析瘫痪状态”，这种状态中往往由于没有弄清当前阶段的所有小细节而感到不能继续了。记住，不论做了多少分析，总有系统的一些问题直到设计时才暴露出来，并且更多的问题是到编程或直到程序完成运行时才出现。因此，迅速进行分析和设计并对提出的系统执行测试是相当重要的。

这个问题值得强调。因为我们在过程型语言上的历史经验，一个项目组希望在进入设计和实现之前认真处理和理解每个细节，这是值得赞扬的。的确，在构造DBMS时，需要彻底理解用户的需要。但是DBMS属于能很好表述和充分理解的一类问题。在许多这种程序中，数据库结构就主要是问题之所在。本章讨论的编程问题属于所谓“不定（wild card）”（本人的术语）类型，这种问题的解决方法不是将众所周知的解决方案简单地重组，而是包含一个或多个“不定要素”——先前没有较了解的解决方案的要素，为此，需要研究[⊖]。由于在分析 45
阶段没有充分的信息去解决这类问题，因此在设计和执行之前试图彻底地分析“不定型”问题会造成分析瘫痪。解决“不定型”问题需要在整个循环中反复，且需要冒风险（这是很有意义的，由于是在试图完成一些新颖的且潜在回报很高的事情）。看起来似乎有风险是由于“匆忙”进入初步实现而引起的，但这样反而能降低风险，因为我们正在较早地确定一个特定的方法对这个问题是不是可行的。产品开发也是一种风险管理。

经常有人提到“建立一个然后丢掉”。在OOP中，我们仍可以将一部分丢掉，然而由于代码被封装成类，在第一次迭代中我们将必然生成一些有用的类设计，并且产生一些不必抛弃的关于系统设计的有价值的思想。因此，在问题的第一次快速遍历中不仅要为下一遍分析、设计及实现产生关键的信息，而且为下一遍建立代码基础。

也就是说，如果我们正在考虑的是一个包含丰富细节而且需要许多步骤和文档的方法学，将很难判断什么时候停止。应当牢记我们正在努力寻找的是什么：

(1) 什么是对象？（如何将项目分成多个组成部分？）

(2) 它们的接口是什么？（需要向每个对象发送什么信息？）

只要我们知道了对象和接口，就可以编写程序了。由于各种原因我们可能需要比这些更 46
多的描述和文档，但是我们需要的信息不能比这些更少。

整个过程可以分5个阶段完成，阶段0只是使用一些结构的初始约定。

⊖ 一个极好的例子是Martin Fowler所写的专著《*UML Distilled*》(Addison-Wesley, 2000)，该书将复杂的UML过程简化为可管理的子集。

⊖ 我估计这样的项目有一条经验规则：如果不定因素不止一个，在没有创建一个能工作的原型之前，不要计划它将用多长时间和将花费多少。这里的自由度太大了。

1.9.1 第0阶段：制定计划

我们必须首先决定在此过程中应当有哪些步骤。这听起来简单（事实上，所有听起来都挺简单的），但是人们常常在开始编码之前没有考虑这一问题。如果计划是“让我们一开始就编码”，那很好（有时，当我们对问题充分理解时，这是合适的）。至少，我们承认这是一个计划。

在这个阶段，我们可能还要决定一些另外的过程结构，但不是全部。可以理解，有些程序员喜欢用“休假方式”工作，也就是在开展他们的工作过程中，没有强制性的结构。“想做的时候就做”。这可能在短时间内是吸引人的，但是我发现，在进程中设立一些里程碑可以帮助集中我们的注意力，激发我们的热情，而不是只注意“完成项目”这个单一的目标。另外，里程碑将项目分成更细的阶段使得风险减小（此外里程碑还提供更多庆祝的机会）。

当我开始研究小说结构时（有一天我也要写小说），我最初是反对结构思想的，我觉得自己在写作时，直接下笔千言就行了。但是，稍后我认识到，在写涉及计算机的文字时，本身结构足够清晰，所以不需要多想。但是我仍然要组织文字结构，虽然在我头脑中是半意识的。即便我们认为自己的计划只是一上来就开始编码，在后续阶段仍然需要不断询问和回答一些问题。

1.9.1.1 任务陈述

47 无论建造什么系统，不管如何复杂，都有其基本的目的，有其要处理的业务，有其所满足的基本需要。通过依次审视用户界面、硬件或系统的特殊细节、算法编码和效率问题，我们将最终找出它的核心，通常简单而又直接。就像来自好莱坞电影的所谓高层概念（*high concept*），我们能用一句或两句话表述。这种纯粹的表述是起点。

高层概念相当重要，因为它设定了项目的基调，这是一种任务陈述。我们不必一开始就让它正确（我们也许正处于在项目变得完全清晰之前的最后阶段），但是要不停地努力直至它越来越正确。例如：在一个空中交通指挥系统中，我们可以从关于正在建立的系统的一个高层概念入手：“塔楼程序跟踪飞机”。但是当我们将这一系统收缩以适用于一个非常小的机场时，考虑将发生什么情况；可能只有一个控制人员甚至什么都没有。一个更有用的模型不应当像它描述问题那样多地关注正在创建的解决方案，例如“飞机到达、卸货、维修、重新装货和离开等”。

1.9.2 第1阶段：我们在做什么

在上一代程序设计［称为过程型设计（*procedural design*）］中，这一阶段称为“建立需求分析（*requirements analysis*）和系统规范说明（*system specification*）”。这些当然是容易迷路的地方。它们是一些名字很吓人的文档，本身可能就是大项目。当然，它们的目的是好的。需求分析说的是“制定一系列指导方针，我们将利用它了解任务什么时候完成且用户什么时候满足”。系统规范说明指出，“这是程序将做什么（不是现在）以满足需求的一个描述”。需求分析实际上是我们和用户之间的一个合同（即使用户在我们的公司工作或是另一些对象或系统）。系统规范说明是对问题的一个顶层探测，且在一定程度上要说明项目是否能做和将需多少时间。由于这两个问题需要人们的共识（并且因为他们通常会随时间的推移而改变意见），我认为最好将它们尽可能地保持最小限度（理想情况下，是列表和基本的图表）以节省时间。

48 我们可能有其他的限制，如需要将它们扩展为大一些的文档，但是小而简洁的最初文档，可以在由一个动态创建描述的领导者的带领下，通过很少几次头脑风暴(brainstorming)讨论而得

到。这不仅需要每个人的投入，而且能激励小组中每个成员参与，使他们意见一致。也许，最重要的是，它可以使项目以极大的热情开始。

这一阶段中我们有必要把注意力始终放在核心问题上：确定这个系统要做什么。为此，最有价值的工具是一组所谓的“用例（use case）”。用例指明了系统中的关键特性，它们将展现我们使用的一些基本的类。它们实际上是对类似下述这些问题的描述性回答[⊖]：

1）“谁将使用这个系统？”

2）“执行者用这个系统做什么？”

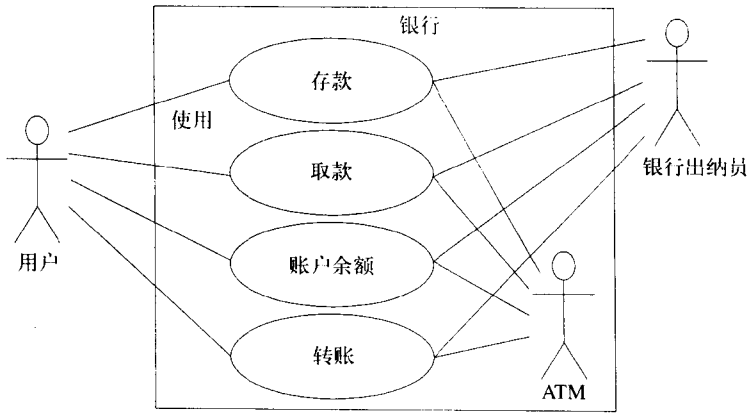
3）“执行者如何用这个系统工作？”

4）“如果其他人也做这件事，或者同一个执行者有不同的目标，该怎么办？（揭示变化）”

5）“当使用这个系统时，会发生什么问题？（揭示异常）”

例如，如果设计一个自动取款机，此系统的一个特定功能方面的用例能够描述这台自动取款机在任何可能情况下的行为。这些“情况”每一个称为情节（*scenario*），而用例可以认为是情节的集合。我们可以把情节认为是以“如果...系统将怎样？”开头的问题。例如，“如果一个用户在24小时内刚刚存了一张支票，且在此支票之外该账户中没有足够的钱能满足提款要求，这时自动取款机怎么办？”。

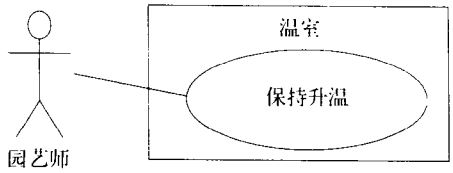
下面的用例图特意进行了简化，以防止我们过早地陷入到系统的实现细节问题中去。 49



每个小人代表一个“执行者（actor）”，它通常是一个人或其他类型的自由代理（甚至可以是其他计算机系统，如“ATM”中的情况）。方框代表系统的边界。椭圆代表用例，是对此系统能完成的有价值工作的描述。在执行者和用例之间的直线代表交互。

只要符合用户的使用感受，系统实际上如何实现并不重要。

用例不必十分复杂，即便底层系统非常复杂。这只是为了表示用户眼中的系统形象。例如：



⊖ 感谢James H Jarrett的帮助。

50 通过确定用户在系统中可能有的所有交互行为，用例就生成了需求规范说明。我们试图找到系统的完整用例，完成之后，我们就得到了系统任务的核心内容。注意力集中在用例上的好处是，它们总是将我们带回到要点部分而不至于留心那些对完成任务无关紧要的问题。也就是说，如果得到了全部用例就可以描述系统并进入到下一个阶段。在最初的尝试中我们很难完全得到特征，但这已经很好了。任何事物随着时间的推移都会自己暴露出来，如果在这一点上就要求系统规范说明完美将会永远止步不前。

如果遇到了困难，我们可以通过使用一个近似工具启动这个阶段：用很少的段落描述此系统，然后寻找名词和动词。名词往往意味着执行者、用例的上下文（如“休息室”）或在用例中使用的物品。动词往往意味着执行者和用例之间的交互行为，表示用例中的特定步骤。我们还将发现名词和动词在设计阶段将对应产生对象和消息（注意用例描述子系统间的交互，因此“名词和动词”技术只能作为集体讨论工具，因为它不生成用例）[⊖]。

用例和执行者之间的边界能指出用户界面的存在，但不能定义这样的用户界面。为了定义和创建用户界面，可参看Larry Constantine和Lucy Lockwood所著的《Software for Use》(Addison Wesley Longman, 1999) 或访问*www.ForUse.com*。

虽然这有点像魔术，但此时进行某种基本的进度安排是重要的。我们现在有了创建目标的总体概念，所以我们可能产生它需要多长时间的概念。这里涉及大量因素。如果估算了一个长时间表，则公司可能决定不创建它（并把资源用在更合理的项目上去，这是好事）。或者管理人员可能已经决定这个项目应当花多少时间，并且试图改变我们做出的估计。但是最好
51 一开始有一个准确的时间表，解决早期决心的问题。已经有大量的努力以产生精确建立时间表的技术（就像预测股票市场的技术），然而，最好的方法或许是依靠我们的经验和直觉。得到实际上将花多少时间的估计，加倍，再加上百分之十。我们的直觉也许是对的，我们能及时得到能用的产品。“加倍”将使产品更好，加百分之十用于最后的润色和细化[⊖]。然而，我们需要解释它，克服抱怨和当我们拿出这个时间表时发生种种事情，最终完成这一问题。

1.9.3 第2阶段：我们将如何建立对象

在这一阶段，我们必须做出设计，描述这些类和它们如何交互。确定类和交互的出色技术是*类职责协同*（*Class-Responsibility-Collaboration, CRC*）卡片。此技术的部分价值是它非常简单：只要有一组3到5英寸的空白卡片，在上面书写。每张卡片描述一个类，在卡片上写的内容是：

(1) 类的名字。这很重要，因为名字体现了类行为的本质，所以有一目了然的作用。

(2) 类的职责：它应当做什么。通常，它可以仅由成员函数的名字陈述（因为在好的设计中，这些名字应当是描述性的），但并不产生其他的注记。如果需要开始这个过程，请从一个
52 懒程序员的立场看这个问题：你希望有什么样的对象魔术般地出现，把你的问题全部解决？

(3) 类的协同：它与其他类有哪些交互？“交互”是非常宽泛的术语。它可以是一些已经

⊖ 更多有关用例的内容可以在Schneider & Winters所写的专著《*Applying Use Cases*》(Addison-Wesley, 1998)和Rosenberg所写的专著《*Use Case Driven Object Modeling with UML*》(Addison-Wesley, 1999)中找到。

⊖ 我个人观点后来已经变了。加倍和增加百分之十将给出相当准确的估计（假设这里没有太多的不定要素），但是我们仍然需要勤奋工作，以及时完成。如果我们希望时间真的花这么长，并且在这个过程中得到乐趣，我认为，正确的增加是3到4倍。

存在的其他对象对这个类的对象提供的服务。协同还应当考虑这个类的观众。例如如果创建了**Firecracker**（鞭炮），那么谁将观察它，是**Chemist**（药剂师）还是**Spectator**（观众）？前者希望知道鞭炮由什么化学成分组成，后者对鞭炮爆炸后的颜色和形状有反应。

我们可能想让卡片更大一些，因为我们希望从中得到全部信息，但是它们是非常小的，这不仅能保持我们的类小，而且能防止过早地陷入过多的细节。如果一张小卡片上放不下类所需要的信息，那么这个类就太复杂了（或者是考虑过细了，或者应当创建多个类）。理想的类应当一目了然。CRC卡片的思想是帮助我们找到设计的第一印象，使得我们能得到总体概念，然后精炼我们的设计。

CRC卡片的最大的好处之一是在交流中。在一个组中，最好实时进行交流，而不是用计算机。每个人负责几个类（起初它们没有名字或其他信息）。每次只解决一个情节，决定发送什么消息给不同的对象以满足每个情节，这样就能作出一个比较形象的对问题的模拟。当我们经历了这个过程后，就会找出我们所需要的类以及它们的职责和协同，这样，我们同时也填写好这些卡片。当我们完成所有用例后，就有了一个相当完整的设计的第一印象。

在我开始用CRC卡片之前，当提出最初的设计时，我最成功的咨询经验就是站在一个没有OOP经验的项目组前，在白板上描述对象。我们讨论对象应当如何互相通信，擦除其中的一些，用其他的对象替换它们。实际上我是在白板上管理所有的“CRC卡片”。项目组（他们知道项目的目标）真正地在做这个设计，他们“拥有”这个设计，而不是获得既成的设计。我所做的 53
所有事情就是通过提问正确的问题，提炼这些假设，并且从项目组得到反馈，修改这些假设来指导这个过程。这个过程的真正好处是项目组学习了如何做面向对象的设计，不是通过复审抽象的例子，而是通过在一个设计上工作，这对于他们是最有兴趣的。

制作了一组CRC卡片之后，我们可能希望用UML[⊖]创建这个设计的更形式化的描述。我们并不非要用UML，但它可能有帮助，特别是如果我们要将一个图表挂在墙上，让大家一起思考时，这是一个很好的想法。除了UML之外的另一选择是对象及其接口的文字描述，这或许依赖于我们的程序设计语言，也就是代码本身[⊖]。

UML还提供了另外一种图形符号来描述系统的动态模型。在一个系统或子系统的状态转换占主导地位，以至于它们需要自己的图表的情况下，这是有帮助的（例如在控制系统中）。我们可能还需要描述数据结构，因为系统或子系统中数据结构是重要因素（例如数据库）。

当已经描述了对象及其接口后，第2阶段就要完成了。这时已经知道了对象中的大多数，通常会有对象漏掉，直到第3阶段才被发现。这没问题。我们关心的是最终能找到所有的对象。在这个阶段较早地发现它们是好的。因为OOP提供了充分的结构，所以如果我们稍迟发现它们也可以。事实上，对象设计可能在程序设计全过程的五个阶段中都会发生。

1.9.3.1 对象设计的五个阶段

对象的设计生命期不仅仅限于写程序的时间。实际上，它出现在一系列阶段上。接受这种观点很有好处，因为我们不再期望设计立刻尽善尽美，而是认识到，对对象做什么和它应 54
当像什么的理解，会随着时间的推移而呈现。这个观点也适用于不同类型程序的设计。特殊类型程序的模式是通过一次又一次地求解问题而形成的（设计模式在第2卷介绍）。同样，对象有自己的模式，通过理解、使用和重用而形成。

⊖ 对于初学者，我推荐上面提到过的专著《*UML Distilled*》。

⊖ Python (www.Python.org)常常被用做“可执行伪代码”。

(1) **对象发现** 这个阶段出现在程序的最初分析期间。对象可以通过寻找外部因素及边界、系统中重复的元素和最小概念单元而发现。如果已经有了一组类库，某些对象是很明显的。类之间的共同性（暗示着基类和继承关系），可以立刻出现或在设计过程的后期出现。

(2) **对象装配** 当我们正在建立对象时会发现需要一些新成员，这些新成员在对象发现时期未出现过。对象的这种内部需要可能要用新类去支持它。

(3) **系统构造** 再次指出，对对象的更多要求可能出现在以后阶段。随着不断学习，我们会改进我们的对象。与系统中其他对象通信和互相连接的需要，可以改变已有的类或要求新类。例如，我们可以发现需要辅助类，这些类如像一个链表，它们包含很少的状态信息或没有状态信息，只有帮助其他类的功能。

(4) **系统扩充** 当我们向系统增添新的性能时，可能发现我们先前的设计不容易支持系统扩充。这时，我们可以重新构造部分系统，并很可能要增加新类或类层次。

(5) **对象重用** 这是对类真正的强度测试。如果某些人试图在全新的情况下重用它，他们也许会发现一些缺点。当我们修改一个类以适应更新的程序时，类的一般原则将变得更清楚，
55 直到我们有了一个真正可重用的对象。然而，不要期望从一个系统设计而来的大多数对象是可重用的，大量对象是对于特定系统的。可重用类一般共性较少，为了重用，它们必须解决更一般的问题。

1.9.3.2 对象开发准则

下述步骤提出了考虑开发类时要用到的一些准则：

1) 让特定问题生成一个类，然后在解决其他问题期间让这个类生长和成熟。
2) 记住，发现所需要的类（和它们的接口），是设计系统的主要内容。如果已经有了那些类，这个项目就不困难了。
3) 不要强迫自己在一开始就知道每一件事情，应当不断地学习。
4) 开始编程，让一些部分能够运行，这样就可以证明或否定已生成的设计。不要害怕过程型大杂烩式的代码——类的隔离性可以控制它们。坏的类不会破坏好的类。
5) 尽量保持简单。具有明显用途的不太清楚的对象比很复杂的接口好。当需要下决心时，用Occam的Razor方法：选择简单的类，因为简单的类总是好一些。从小的和简单的类开始，当我们对它有了较好的理解时再扩展这个类接口，但是很难从一个类中删去元素。

1.9.4 第3阶段：创建核心

这是从粗线条设计向编译和执行可执行代码体的最初转换阶段，特别是，它将证明或否定我们的体系结构。这不是一遍的过程，而是反复地建立系统的一系列步骤的开始，我们将
56 在第4阶段中看到这一点。

我们的目标是寻找实现系统体系结构的核心，尽管这个系统在第一遍不太完整。我们正在创建一个框架，在将来的反复中可以完善它。我们正在完成第一遍多系统集成和测试，向风险承担者（stakeholder）提出反馈意见，关于他们的系统看上去如何以及如何发展等等。理想情况下，我们还可以暴露一些严重的问题。我们大概还可以发现对最初的体系结构能做哪些改变和改进——本来在没有实现这个系统之前，可能是无法了解这些内容的。

建立这个系统的一部分工作是实际检查，就是对照需求分析和系统规范说明与测试结果

(无论需求分析和规范说明以何种形式存在)。确保我们的测试结果与需求和用例符合。当系统核心稳定后，我们就可以向下进行和增加更多的功能了。

1.9.5 第4阶段：迭代用例

一旦代码框架运行起来，我们增加的每一组特征本身就是一个小项目。在一次迭代(*iteration*)期间，我们增加一组特征，一次迭代是一个相当短的开发时期。

一次迭代有多长时间？理想情况下，每次迭代为一到三个星期(具体随实现语言而异)。在这个期间的最后，我们得到一个集成的、测试过的、比前一周期有更多功能的系统。特别有趣的是迭代的基础：一个用例。每个用例是一组相关功能，在一次迭代中加入系统。这不仅为我们更好地提供了“用例应当处于什么范围内”的概念，而且还对用例概念进行了巩固，在分析和设计之后这个概念并未丢弃，它是整个软件建造过程中开发的基本单元。

当我们达到目标功能或外部最终期限到了，并且客户对当前版本满意时，我们就停止迭代。(记住，软件行业是建立在双方约定的基础之上。)因为这个过程是迭代的，所以我们有许多机会交货，而不是只有一个终点；开放源代码项目是在一次迭代的和高反馈的环境中开 57
发、而这正是它成功的原因。

有许多理由说明迭代开发过程是有价值的。我们可以更早地揭露和解决严重问题，客户有足够的机会改变它们的意见，程序员会更满意，能更精确地掌握项目。而另一个重要的好处是对风险承担者(stakeholder)意见的反馈，他们能从项目当前状态准确地看到各方面因素。这可以减少或消除令人头脑昏昏然的会议，增强风险承担者的信心和支持。

1.9.6 第5阶段：进化

这是开发周期中，传统上称为“维护”的一个阶段，是一个含义广泛的术语，包含了从“让软件真正按最初提出的方式运行”到“添加用户忘记说明的性能”，到更传统的“排除暴露的错误”和“在出现新的需求时添加性能”。所以，对术语“维护”有许多误解，它已经有点虚假的成分，部分因为它假设我们已经实际上建立了原始的程序，且所有的需要就是改变其中一些部分，加加油，防止生锈。也许，有更好的术语来描述所进行的工作。

此处将使用术语“进化”(*evolution*)[⊖]。这就是说，“我们不可能第一次就使软件正确，所以应该为学习、返工和修改留有余地”。当我们对问题有了深入的学习和领会之后，可能需要做大量的修改。如果我们使软件不断进化，直到使软件正确，无论在短期内还是在长期内，将产生极为优雅的程序。进化是使程序从好到优秀，是使第一遍不理解的问题变清楚的过程。它也是我们的类能从只为一个项目使用进化为可重用资源的过程。 58

“使软件正确”的意思不只是使程序按照要求和用例工作，还意味着我们理解代码的内部结构，并且认识到它能很好地协同工作，没有拙笨的语法和过大的对象，也没有难看的暴露的代码。另外，必须认识到，程序结构将经历各种修改而保全下来，这些修改贯穿整个生命期，也要认识到，这些修改可以是很容易进行和很简洁的。这可不是小成就。我们不仅必须懂得我们正在建造的程序，而且必须懂得这个程序将进化[我称之为改变矢量(*vector of change*)[⊖]]。

⊖ 进化的一个方面在Martin Fowler的专著《*Refactoring: improving the design of existing code*》(Addison-Wesley, 1999)中做了介绍。需预先警告，这本书的例子全部使用JAVA编写。

⊖ 这一术语在第2卷的“设计模式”一章中研究。

幸运的是，面向对象程序设计语言特别适合支持这样连续的修改，由对象创建的边界是防止结构被破坏的保障。面向对象程序设计语言还允许我们做大幅度改变而不引起代码的全面动荡，这在过程型程序中似乎太剧烈了。事实上，支持进化可能是OOP的最重要的好处。

借助于进化，我们创建了近似于我们所要创建的程序，然后进行检查，与需求比较，看哪些地方有缺点。随后回头看，重新设计和重新实现程序不能正确工作的部分，改进它[⊖]。在得到正确解决方案之前，可能实际上需要几次解决这个问题或问题的一个方面。（对设计模式的研究在第2卷中描述，对此阶段非常有用。）

进化还发生在我们建造系统时，开始它好像符合需求，以后又发现它实际上不是想要的系
59 统。当看到这个系统运行时，我们发现实际上想要解决的是另一个问题。如果我们认为这种进化将会发生，则应该尽快地建造第一个版本，这样可以发现它实际上是不是想要的系统。

也许，需要记住的最重要的事情是，按默认情况（实际上是按定义），如果修改了一个类，则它的超类和子类都仍然正常工作。不要害怕修改（特别是，如果我们已经有内部的一组单元测试，验证了修改的正确性时）。修改不一定会破坏程序，结果的任何改变都将限定于子类和/或被改变的类的协同者。

1.9.7 计划的回报

当然了，谁也不会在没有仔细计划之前，就建造房子。然而，如果建造鸡圈或狗圈，计划就不需要那么精细了，但是可能仍然以某种草图开始，指导建造过程。软件开发已经走向了极端。在很长的时间里，人们在开发中没有多少结构概念，很多大项目都失败了。其结果最终使各种方法学应运而生，它们包含大量的结构和细节，首先主要针对大项目。这些方法学大得可怕，并不好用，看上去好像我们要花所有的时间在写文档，没有时间编写程序。（真的常常如此。）我希望在这里提出的是走中间道路因地制宜。用一种能满足需要（和个性化）的方法。无论选择的方法学多么小，在项目中进行一些计划还是会有较大改进的，比完全没有计划强得多。请记住，根据各种估计，超过50%的项目都失败了（有些估计说70%以上）。

遵循计划（简单和短小的更加适宜），在编码之前就提出设计结构，我们会发现，事情总的说来比一上来就编码的方法容易得多，并且会认识到大多数情况是令人满意的。就我的经验，提出一个漂亮的方案实际上是在一种完全不同水平上的满足，感觉上更接近于艺术，而
60 不是技术。精致总是有回报的，这不是一种虚浮的追求。它不仅给出了一个容易建造和调试的程序，而且容易理解和维护，这就是其经济价值的体现。

1.10 极限编程

我从研究生时开始就断断续续研究过分析和设计技术。极限编程（*eXtreme Programming*, XP）的思想在我见过的方法学中最为激进也最令人愉快。在由Kent Beck编写的《*Extreme Programming Explained*》(Addison-Wesley, 2000)[⊖] 一书和在Web站点*www.xprogramming.com*上可以找到它的历史。

XP既是程序设计工作的哲学，又是做程序设计的一组原则。其中一些原则反映在新近出

⊖ 这是有些像“快速生成原型”，先建造一个快而稍差（qwick-and-dirty）的版本，然后研究这个系统，再丢弃这个原型并正确地建造它。快速生成原型的麻烦是人们不丢弃这个原型，而是在它上面建造。与过程型程序设计中缺乏结构相结合，这常常会产生混乱的系统，使得维护费用提高。

⊖ 本书的中文版已由人民邮电出版社出版。——编辑注

现的其他方法学中，但我认为，有两个原则最重要、贡献最大，即“先写测试”和“结对编程”。虽然Beck强烈坚持全过程，但他也指出，如果只采用这两项实践，就能极大地改进生产效率和可靠性。

1.10.1 先写测试

测试传统上被归于项目的最后阶段，在“万事俱备，只欠肯定”之后。这意味着它的优先级很低，专门做此工作的人没有足够的地位，甚至可怜兮兮地只能在地下室里干活，不算“真正的程序员”。测试组对此的反应是，穿着黑色的衣服，当攻破了某个程序时就兴高采烈（老实说，当我攻破了C++编译器时我也有这种感觉）。

XP革命性地改变了测试的这个概念，将它置于与编码相等（甚至更高）的优先地位。事实上，我们需要在编写被测试代码之前写测试，而且这些测试与代码永远在一起。这些测试必须在每次项目集成时都能成功地执行（这是经常地，有时一天几次）。

先写测试有两个极其重要的作用。 61

第一，它强制类的接口有清楚的定义。我经常建议，人们设计系统时会把解决特定的问题的理想的类，作为工具。XP的测试策略比这走得更远，它准确地指明这个类看上去必须像什么，对于这个类的用户，这个类准确地如何动作。用确切的术语，可以写所有的文档描述，或者创建所有图表，描述一个类应当如何行动和看上去像什么，但是什么都比不上一组测试真实。前者列出的是期望，而测试是由编译器和运行程序强制的合约。很难想像有比测试更具体的类描述。

当创建测试时，我们被迫要充分思考这个类，常常会发现所需要的功能，而这些功能可能会在思考UML图、CRC、用例等过程期间漏掉。

写测试的第二个重要的作用，是能在每次编连软件时运行这些测试。这实际上让编译器又执行了测试。如果从这个角度观察程序设计语言的发展，就会发现在技术上的改进实际上是围绕着测试开展的。汇编语言只检查语法，而C增加了一些语义约束，能防止程序员犯某些类型的错误。OOP语言强加了更多的语义约束，可以把它们看成是某种实际形式的测试。“这个数据类型的用法合适吗？这个函数的调用合适吗？”这些都是由编译器或运行时系统执行的测试任务。我们已经看到了在程序设计语言中加入这些测试的成效：人们能用更少的时间和劳动写更复杂的程序，并使它们运行。开始我不理解为什么能有如此成效，后来认识到，这正是测试的作用。如果程序员写错了，内部测试的安全网就告诉他有问题，并指出在什么地方。

但是，由语言设计提供的内部测试只能做部分的工作。有时，我们必须自己动手，增加其余的测试，形成配套（与编译器和运行时系统协作），验证程序的一切。正如编译器能陪伴 62
帮助我们一样，我们也希望其他的测试也能从一开始就帮助我们。这就是为什么我们要书写测试、并在每次系统创建时自动地运行它们的原因。我们的测试就变成了这个语言提供的安全网的扩充。

在功能越来越强大的程序设计语言中，我发现可以更大胆地尝试更多易错的实验，因为我知道高级语言功能会帮助排除错误，避免浪费时间。XP测试机制对于整个项目的作用也是如此。因为我们知道测试始终能捕捉我们引进的任何问题（当需要时我们有规律地增加新的测试），所以当需要时可以做大的改变，不用担心会使整个系统混乱。这真是太强大了。

1.10.2 结对编程

结对编程（pair programming）反对深植于我们心中的个人主义，我们从小就通过学校（在那里，成功与失败全在自己，与邻座一起工作这样的事情会被认为是“欺骗”）和媒体（特别是好莱坞电影，其中的英雄总是在与盲目服从作斗争）在灌输这种思想[⊖]。程序员也被认为是个人主义的典范，正如Larry Constantine喜欢说的“牛仔编码者”。而XP，这一打破传统的方法学，主张代码应当在每个工作站上由两个人编写。而且这应当在有一堆工作站的工作场合中进行，拆掉人们喜欢的隔板。实际上，Beck说，转向XP的第一个任务是用螺丝刀和
63 螺钉完成的，拆除挡道的一切东西[⊖]（这需要经理能说服后勤部门）。

结对编程的好处是，一个人编写代码时另一个人在思考。思考者的头脑中保持总体概念，不仅是手头问题的这一段，而且还有XP指导方针。例如，如果两个人都在工作，就不太可能会有其中的一个说“我不想首先写测试”而愤然离去。如果编码者遇到障碍，他们就交换位置。如果两个人都遇到障碍，他们的讨论可能被在这个区域工作的其他人听到，可能给出帮助。这种结对方式，使事情顺畅、有章可循。也许更重要的是，它能使程序设计更具有社交性和娱乐性。

我在一些讲习班的练习期间用过结对编程，似乎明显地改进了每个人的练习过程。

1.11 为什么C++会成功

C++能够如此成功，部分原因是它的目标不只是为了将C语言转变成为OOP语言（虽然这是最初的目的），而且还为了解决当今程序员，特别是那些在C语言中已经大量投入的程序员所面临的许多问题。传统上，人们已经对OOP语言有了这样的看法：程序员应当抛弃所知道的每件事情并且从一组新概念和新文法重新开始，程序员应当相信，从长远观点来看，最好是丢掉所有来自过程型语言的老行装。从长远角度看，这是对的，但从短期角度看，这些行装还是有价值的。最有价值的可能不是那些原有的代码库（用合适的工具，可以转变它），而是原有的头脑库。作为一个职业C程序员，如果让他丢掉他知道的关于C的每一件事情，以适应新的语言，那么，在几个月内，他将毫无成果，直到他的头脑适应了这一新范例时为止。但如果他能调整已有的C知识，并在这个基础上扩展，那么他就可以继续保持高的生产效率，
64 带着已有的知识，进入面向对象程序设计的世界。因为每一个人都有自己的程序设计思维模型，所以这个转变是很混乱的。因此，简而言之，C++成功的原因是很经济的：转变到OOP上需要代价，而转变到C++上所花的代价可能比较小[⊜]。

C++的目的是提高生产效率。生产效率与多方面因素有关，而语言是为了尽可能地帮助使用者，尽可能少地因为使用武断的规则或特殊性能的需求而妨碍使用者。C++成功的原因是它立足于实际：尽可能地为程序员提供最大利益（至少从C的观点上看是这样）。

1.11.1 一个较好的C

即便程序员在C++环境下继续写C代码，也能直接得到好处，因为C++堵塞了C语言中的

⊖ 虽然这个观点可能太美国化了，但是好莱坞的故事遍布世界。

⊖ （尤其是）包括PA系统。我一度在某公司工作，他们坚持广播每个管理人员的电话，这种做法经常会打断我们的工作（但是管理者都不能想到把它去掉）。最后，当没人注意时，我就剪断了电线。

⊜ 我说“可能”，因为C++太复杂了，实际上转变到Java上可能更便宜。决定选择哪种语言有许多因素，本书中我假定已经选择了C++。

许多漏洞，并提供更好的类型检查和编译时的分析。程序员必须先声明函数，使编译器能检查它们的使用。预处理器也限制了值替换和宏，这就减少了查找错误的困难。C++有一个特征，称为引用（*reference*），它允许对函数参数和返回值的地址进行更方便的处理。通过函数重载（*function overloading*），改进了对名字的处理，使程序员能对不同的函数使用相同的名字。另外，一个称为名字空间（*namespaces*）的特征也改进了对名字的控制。除此之外，还有许多较小的特征改善了C的安全性。

1.11.2 延续式的学习过程

与学习新语言有关的问题是生产效率问题。所有公司都很难承受因为软件工程师正在学习新语言而突然降低了生产效率。C++是对C的扩充，而不是新的文法和新的程序设计模型。当程序员学习和理解这些性能时，他可以逐渐应用它们，这就允许他继续创建有用的代码。 65
这是C++成功的最重要的原因之一。

另外，已有的C代码在C++中仍然是有用的，但因为C++编译器是更严格的，所以，重新编译这些代码时，常常会发现隐藏的错误。

1.11.3 效率

有时，以牺牲程序执行速度换取程序员的生产效率是值得的。假如，一个金融模型仅在短期内有用，那么，快速创建这个模型比所写程序能更快速执行更重要。然而，很多应用程序都要求一定程度的运行效率，所以C++在更高运行效率方面总是有些偏差。但因为C程序员通常具有很强的效率意识，所以这也保证他们并不认为这个语言太庞大、太慢。C++的一些性能允许程序员在产生的代码不够有效时做一些改善。

C++不仅有与C相同的低层控制能力（和在C++程序中直接写汇编语言的能力），而且非正式的证据表明，面向对象的C++程序的速度与用C写的程序的速度相差在±10%之内，而且常常更接近[⊖]。用OOP方法设计的程序实际上可能比C的对应版本更有效。

1.11.4 系统更容易表达和理解

为适合于某问题而设计的类当然能更好地表达这个问题。这意味着编写代码时，程序员是在用问题空间的术语描述问题的解（例如“把垫圈放进材料箱”），而不是用计算机的术语，也就是解空间的术语，来描述问题的解（例如“设置芯片的一位，即合上继电器”）。程序员 66
所涉及的是较高层的概念，单行代码能做更多的事情。

易于表达所带来的另一个优点是易于维护。据报道，在程序的整个生命周期中，维护占了花费的很大一部分。如果程序容易理解，那么它就更容易维护，这还能减少创建和维护文档的花费。

1.11.5 尽量使用库

创建程序最快的方法是使用已经写好的代码：库。C++的主要目标是让程序员能更容易地使用库，这是通过将库转换为新数据类型（类）来完成的。引入一个库，就是向该语言增

⊖ 参看Dan Saks在“C/C++ User's Journal”杂志上的栏目，对C++库性能的重要调查。

加一个新类型。因为是由编译器负责这个库如何使用，也就是保证适当的初始化和清除，保证函数正确调用，所以程序员的精力可以集中在他想要这个库做什么，而不是如何做这件事。

因为名字能够在程序的各部分之间隔离，所以程序员想使用多少库就使用多少库，不会有像C语言那样的名字冲突。

1.11.6 利用模板的源代码重用

有一些重要的类型的类，它们要求修改源代码以有效地重用它们。C++的模板（*template*）功能可以自动完成对代码的修改，因而它是重用库代码的特别有用的工具。用模板设计的类型很容易与其他类型一起工作。因为模板对客户程序员隐藏了这类代码重用的复杂性，所以它很有好处。

1.11.7 错误处理

在C语言中，错误处理是一个很糟糕的问题。程序员常常会忽视它们，而且常常对它们束手无策。如果正在建立庞大而复杂的程序，没有什么比让错误隐藏在某处，且没有引导告诉
67 我们它来自何处更糟的了。C++的异常处理（*exception handling*）（在本卷介绍，在第2卷中将有完整的讨论，其材料可以从*www.BruceEckel.com*上下载）可以保证能检查到错误并使特定的某件事情发生。

1.11.8 大型程序设计

许多传统语言对程序的规模和复杂性有内在的限制。例如，BASIC对于某些类型的问题能很快解决，但是如果这个程序有几页纸长，或者超出该语言的正常解题范围，那么它可能永远也算不出结果。C语言同样有这样的限制，例如当程序超过50 000行时，名字冲突就开始成为问题。实际上，程序员会用光函数和变量名。另一个特别糟糕的问题是如果C语言中存在一些小漏洞——有错误隐藏在大程序中，要找出它们是极其困难的。

并没有清楚的文字告诉程序员，什么时候他的语言会失效，即便有，他也会忽视它们。他不说“我的BASIC程序太大，我必须用C重写”，而经常是试图硬塞进另外几行，以增加额外的功能。所以额外的花费就悄悄加上来了。

设计C++的目的是为了辅助大型程序设计，这就是说，去掉这些在小程序和大程序之间的复杂性的分界。当程序员写hello-world之类实用程序时，他确实不需要用OOP、模板、名字空间和异常处理，但是当他需要的时候，这些性能就有用了。而且，编译器在排除错误方面，对于小程序和大程序一样有效。

1.12 为向OOP转变而采取的策略

如果决定采用OOP，我们的下一个问题可能是“如何才能使得经理/同事/部门/伙伴开始使用OOP？”想想看，作为独立的程序员，应当如何学习使用新语言和新的程序设计形式。
68 和前面一样，首先训练和做例子，再通过一个试验项目得到一个基本的感觉，不要做太混乱的任何事情，然后尝试做一个“真实世界”的实际有用的项目。在第一个项目中，通过读、向专家问问题、与朋友切磋等方式，继续我们的训练。基本上，这就是许多有经验的程序员建议的从C转到C++的方法。转变整个公司当然应当采用某些动态的方法，但回忆个人是如何

做这件事的，能在转变的每一步中起帮助作用。

1.12.1 指导方针

当向OOP和C++转变时，有一些方针要考虑：

1.12.1.1 训练

第一步是某种形式的培训。记住公司在原始C代码上的投资，并且当每个人都在为这些遗留的东西而为难时，应努力在6到9个月内不使公司完全陷入混乱。挑选一个小组进行培训，更适宜的情况是，这个小组成员是一些勤奋好学、能很好地在一起工作的人们，当他们正在学习C++时，能形成他们自己的支持网。

有时建议采用另一种方法，即对公司各级人员同时进行培训，包括为策略经理而开设的概论课程，以及为项目开发者而开设的设计课程和编程课程。对于较小的公司或较大公司的下层，对他们做事情的方法做一些基本的改变是非常好的。因为代价较高，所以一些公司可能选择以项目层训练而开始，做导航式的项目（可能请一个外面的导师），然后让这个项目组变成公司其他人的老师。

1.12.1.2 低风险项目

首先尝试一个低风险项目，并允许出错。一旦得到了一些经验，就将这第一个小组的成员安插进其他项目组中，或者用这个组的成员作为OOP的技术顶梁柱。这第一个项目可能不能正确工作，所以该项目不应是公司的关键任务。它应当是简单的、自成一体的和有指导意义的。这意味着它应当包括创建对于公司的其他程序员学习C++有意义的类。 69

1.12.1.3 来自成功的模型

在动手之前，挑一些好的面向对象设计的例子。很可能有些人已经解决过我们的问题，如果他们还没有正确地解决它，我们可以应用已经学到的关于抽象的知识，来修改存在的设计，以适合我们自己的需要。这是设计模式的一般概念，将在第2卷中详细讲解。

1.12.1.4 使用已有的类库

转变为C++的主要经济动机是容易使用以类库形式存在的代码（特别是标准C++库，将在本书的第2卷中深入探讨），最短的应用开发周期是利用现有库创建和使用对象，除了**main()**以外不必自己写任何东西。然而，一些新程序员并不理解这一点，不知道已有的类库，或出于对语言的迷恋希望写可能已经存在的类。如果在转变过程的早期努力查找和重用其他人的代码，那么我们在OOP和C++方面将得到最好的成功。

1.12.1.5 不要用C++重写已有的代码

虽然用C++编译C代码通常会有（有时是很大的）好处，它能发现老代码中的问题，但是把时间花在对已有的功能代码用C++重写上，通常不是时间的最佳利用方法（如果必须翻译成对象，我们可以用C++类“包装”C代码）。特别是，如果代码是为重用而编写的，会有很大的好处。但是，有可能出现这种情况：在最初的几个项目中，并不能看到生产效率如您梦想的一样增长，除非这是新项目。如果是从概念到实现的项目，C++和OOP表现最为出色。 70

1.12.2 管理的障碍

如果我们是经理，我们的工作是为项目组争取资源，搬除通往胜利道路上的障碍，并且

通常要努力提供更高的生产效率和和谐的环境，使项目组更有可能产生奇迹。转向C++的三类方式，如果不花费任何代价都是不可能的。与C程序员（也可能是其他过程型语言的程序员）项目组的OOP替代品相比，虽然转向C++可能更便宜一些（这取决于约束条件）[⊖]，但并不是免费的，在试图说服公司转向C++并对转移投资之前，我们应当知道会有障碍。

1.12.2.1 启动的代价

转移到C++的代价比获得C++编译器（最好的编辑器之一，GNU C++编译器，是免费的）大得多。进行培训（也可能是第一个项目的指导），并且确定和购买解决问题的类库而不是自己开发，那么中长期的代价就将减小。这种直接的经费开支是实际建议所必须考虑的因素。另外还有隐藏的花费，在于学习新语言和新程序设计环境期间的生产效率下降。培训和指导确实能减少花费，但是项目组成员必须自己克服了解新技术的各种困难。在这个过程中，将犯更多错误（这是特点，因为认识错误是学习的最快途径），生产效率下降。尽管如此，对于一些类型的程序设计问题，有了正确的类和正确的开发环境，C++学习时可能会比继续用C语
71 言有更高的生产效率（即便考虑到程序员正在犯更多的错误和每天写更少的代码行）。

1.12.2.2 性能问题

一个普遍的问题是，“OOP不会自动使得我们的程序变大和变慢吗？”回答是“不一定”。大多数传统的OOP语言是以实验和快速原型方法设计的，这样实际上就决定了其在规模上的扩大和在速度上的下降。然而，C++是以生产性程序的方式设计的。当用快速原型方式时，我们能尽可能快地将构件组合在一起，而忽视效率问题。如果使用了第三方库，通常已经由它们的厂商优化过了，在这种情况下，用快速开发方法，效率也不是问题。如果我们有一个喜欢的系统，它足够小和快，就继续使用，如果不是，就调整，用描述工具(*profiling tool*)，首先改进速度。这可用简单的C++内部功能完成。如果无效，就寻找对底层实现的修改，但要做到不改变所需要的特殊类。只有当全都不能解决问题时，才需要改变设计。性能在设计中的地位很重要，是主要的设计标准之一。运用快速原型法，可以尽早地了解系统性能。

如前所述，在C和C++之间的规模和速度之比常常不同，但一般是10%之内，而且通常更接近。当使用C++代替C时，可能在规模和速度上得到大的改进，因为为C++所做的设计很大程度上不同于为C所做的。

在C和C++之间比较规模和速度的证据至今还只是传说性的估计，也许还会继续如此。尽管有一些人建议对相同的项目用C和C++同时做，但也许不会有公司把钱浪费在这里，除非它非常大并且对这个研究项目感兴趣。即便如此，它也希望钱花得更好。已经从C（或其他过程
72 型语言）转到C++（或一些其他OOP语言）的程序员几乎一致地都有在程序设计效率上得到很大提高的个人经验，这是能找到的最引人注目的证据。

1.12.2.3 常见的设计错误

当项目组开始使用OOP和C++时，程序员们将会出现一系列常见的设计错误。这经常会发生，因为在早期项目的设计和实现过程中从专家们那里得到的反馈太少，在公司中没有专家，而聘请顾问可能有阻力。我们可能会觉得，在这个周期中，我们懂得OOP太早了并开始了一条不好的道路。有时，对于在这个语言上有经验的一些人而言，显而易见的问题可能是新手们在内部的激烈争论。大量的这类问题都能通过聘用外部富有经验的专家培训和指导来避免。

⊖ 因为生产效率的改进，所以Java语言也应当被考虑。

另一方面，容易出现设计错误的事实也反映出C++的主要缺点：对C向后兼容（当然，这也是它的主要优势）。为了完成能编译C代码的任务，C++不得不做一些妥协，这形成了一些“死角”。这些都是事实，并且包含了学习这个语言的大量弯路。在本书和后续的卷（以及其他书，参看附录C）中，试图揭示当使用C++时会遇到的大量陷阱。应当知道，在这个安全网中有一些漏洞。

1.13 小结

本章希望使读者对面向对象程序设计和C++的大量问题有一定的感性认识，包括为什么OOP是不同的，为什么C++特别不同，什么是OOP方法的思想，和最终当公司转到OOP和C++时会遇到的各种问题。

OOP和C++可能不一定对每个人都适合。对自己的需要作出估计，并决定是否C++能很好地满足自己的需要，或者是否用别的程序设计系统（包括当前正在用的这种系统）会更好，这是很重要的。如果读者知道，在可预见的未来自己的需要非常专门化，如果有特殊的约束， 73
不能由C++满足，那么可以自己研究替代物[⊖]。即便最终选择了C++，也至少应当懂得这些选择是什么，并应当对为什么取这个方向有清晰的看法。

我们已经知道过程型程序的概貌：数据定义和函数调用。为了理解这样程序的含义，必浏览函数调用和底层概念，在头脑中创建一个模型。这就是我们设计过程型程序时需要中间描述的原因，这些程序往往是混乱的，因为表达的术语更偏向于计算机而不是所解决的问题。

因为C++对C语言增加了许多新概念，所以我们自然会认为在C++中的**main()**会比等价的C程序复杂得多。在这里，我们将很高兴地看到：一个写得好的C++程序一般比等价的C程序简单得多和更容易理解。我们将看到的是描述问题空间概念的对象的定义（而不是计算机描述的问题）和发送给这些对象的消息，这些消息描述了问题空间的活动。面向对象程序设计的乐趣之一是，对于设计良好的程序，通过读程序可以很容易理解它。通常，这里代码更少，因为我们的许多问题都能通过重用库代码解决。 74

⊖ 我特别推荐Java(http://java.sun.com)和Python (http://www.Python.org)。

第2章 对象的创建与使用

本章介绍一些C++语法和程序构造概念，使读者能编写和运行一些简单的面向对
75 象的程序。下一章，我们再详细介绍C和C++的基本语法。

首先通过学习本章，我们会对C++面向对象编程的风格有个基本的了解，进而明白人们热衷于这种语言的一些原因。这些足以引导读者读完第3章的内容。第3章中包含了大量的C语言细节，几乎是对C语言的详尽无遗的介绍。

用户定义的数据类型或类（*class*），是C++区别于传统过程型语言的地方。类是一种新的数据类型，用来解决特定问题。一旦创建了一个类，任何人都可以使用它而不需知道它的构造方式和工作原理细节。本章仅把类作为C++内置的另一种数据类型来使用。

通常将创建好的类存放在库里。本章会使用几个C++的类库。一个很重要的标准库是输入输出流库，可以用它从文件或键盘读取数据，并且将数据写入文件和显示出来。我们还将看到来自标准C++类库中的非常方便的**string**类和**vector**容器。到本章结束时，我们会发现使用预先定义好的类库是很容易的事情。

为了创建我们的第一个程序，我们必须先了解用于创建应用程序的工具。

2.1 语言的翻译过程

任何一种计算机语言都要从某种人们容易理解的形式（源代码）转化成计算机能执行的
76 形式（机器指令）。通常，翻译器分为两类：解释器（*interpreter*）和编译器（*compiler*）。

2.1.1 解释器

解释器将源代码转化成一些动作（它可由多组机器指令组成）并立即执行这些动作。例如，BASIC就是一个流行的解释性语言。传统的BASIC解释器一次翻译和执行一行，然后将这一行的解释丢掉。因为解释器必须重新翻译任何重复的代码，程序执行就变慢了。为了提高速度，也可对BASIC进行编译。现在许多的解释器，诸如Python语言的解释器，先把整个程序转化成某种中间语言，然后由执行速度更快的解释器来执行[⊖]。

使用解释器有许多好处。从写代码到执行代码的转换几乎能立即完成，并且源代码总是现存的，所以一旦出现错误，解释器能很容易地指出。对于解释器，较好的交互性和适于快速程序开发（不必要求可执行程序）也是常被提到的两个优点。

当做大项目时解释性语言有某些局限性（而Python似乎是一个例外）。解释器（或其简化版）必须驻留内存以执行程序，而这样一来，即使是最快的解释器其速度也会变得让人难以接受。大部分的解释器要求一次输入整个源代码。这不仅造成内存空间的限制，而且如果语言不提供设施来隔离不同代码段之间的影响，一旦出现错误，就很难调试。

⊖ 解释器和编译器之间的界限非常模糊，尤其对Python来说，它具有编译语言的许多特点和功能，但它只是一种解释语言的快速转换。

2.1.2 编译器

编译器直接把源代码转化成汇编语言或机器指令。最终的结果是一个或多个机器代码的文件。这是一个复杂的过程，通常分几步完成。使用编译器，从写源代码到执行代码的转换，是一个较长的过程。 77

仰仗编译器设计者的聪明才智，编译器生成的程序往往只需较少的运行空间，并且执行速度更快。虽然编译后的程序较小、运行速度快是人们认为应当使用编译器的理由，但在许多时候这却不是最重要的。某些语言（如C语言）可以分别编译各段程序。最后使用*连接器*（*linker*）把各段程序连接成一个完整的*可执行程序*。这个过程称为*分段编译*（*separate compilation*）。

分段编译有许多好处。由于编译器或编译环境的限制，不能一次完成编译的整个程序，可以分段编译。每次创建和测试程序的一部分，当这部分程序能正常运行后，就把它作为程序组块保存起来。人们把测试通过并能正常运行的程序块收集起来加入*库*（*library*）中，供其他程序员使用。由于独立创建每一段程序，其他各段程序的复杂性便被隐藏起来。所有这些特点支持大型程序的创建[⊖]。

编译器的调试功能不断地得以改进。早期的编译器只能产生机器代码，要知道程序的运行状态，程序员要插入打印语句。但这样做并不总是有效的。现代编译器能在可执行程序中插入与源代码有关的信息。这个信息由一些强大的*源代码层的调试器*（*source-level debugger*）使用，以便通过跟踪程序经过源代码的进展来显示程序的执行情况。

为了提高编译速度，一些编译器采用了*内存中编译*（*in-memory compilation*）。大多数编译器，编译时每一步都要读写文件。内存中编译器就是将编译器程序存放在RAM中。对于小程序来说，内存中编译器几乎能和解释器一样响应。 78

2.1.3 编译过程

为了用C/C++编程，应该了解编译过程的步骤和所需工具。某些语言（特别是C/C++）编译时，首先要对源代码执行预处理。*预处理器*（*preprocessor*）是一个简单的程序，它用程序员（利用*预处理器指令*）定义好的模式代替源代码中的模式。预处理器指令用来节省输入，增加代码的可读性。（C++程序设计并不鼓励多使用预处理指令，因为它可能会引起一些不易发现错误，这些将在本书的后面分析）。预处理过的代码通常存放在一个中间文件中。

编译一般分两遍进行。首先，对预处理过的代码进行语法分析。编译器把源代码分解成小的单元并把它们按树形结构组织起来。表达式“**A+B**”中的“**A**”、“**+**”和“**B**”就是语法分析树的叶子节点。

有时候会在编译的第一遍和第二遍之间使用*全局优化器*（*global optimizer*）来生成更短、更快的代码。

编译的第二遍，由代码生成器（*code generator*）遍历语法分析树，把树的每个节点转化成汇编语言或机器代码。如果代码生成器生成的是汇编语言，那么还必须用汇编器对其汇编。两种情况的最后结果都是生成目标模块（通常是，一个以**.o**或**.obj**为扩展名的文件）。有时也会在第二遍中使用*窥孔优化器*（*peephole optimizer*）从相邻一段代码中查找冗余汇编语句。

用“object”（目标）一词表示一段机器代码是一种不合适的选择，在面向对象程序设计

⊖ Python 又是一个例外，因为它也支持分段编译。

之前这一名词就普遍使用了。在讨论编译时“object”与“goal”（目标）含义相同，而在面向对象程序设计中，它的意思是“一个有边界的事物”。

连接器（*linker*）把一组目标模块连接成为一个可执行程序，操作系统可以装载和运行它。
79 当某个目标模块中的函数要引用另一目标模块中的函数或变量时，由连接器来处理这些引用；这就保证了所有需要的、在编译时存在的外部函数和变量仍然存在。连接器还要添加一个特殊的目标模块来完成程序启动任务。

连接器能搜索称为“库”的特殊文件来处理它的所有引用。库将一组目标模块包含在一个文件中。库由一个被称为库管理器（*librarian*）的程序来创建和维护。

2.1.3.1 静态类型检查

类型检查（*type checking*）是编译器在第一遍中完成的。类型检查是检查函数参数是否正确使用，以防止许多程序设计错误。由于类型检查是在编译阶段而不是程序运行阶段进行的，所以称之为静态类型检查（*static type checking*）。

某些面向对象的语言（如Java）也可在程序运行时作部分类型检查[动态类型检查（*dynamic type checking*）]。动态类型检查和静态类型检查结合使用，比仅仅使用静态类型检查更有效。但它也增加了程序执行的开销。

C++使用静态类型检查，因为C++语言不采用任何特殊的运行时支持来处理错误操作。静态类型检查在编译时就告知程序员类型被误用，从而加快了执行时的速度。通过对C++的学习，我们会看到C++语言的主要设计目标也是追求运行速度快，这与面向生产的编程语言C语言一样。

在C++里可以不使用静态类型检查。我们可以自己做动态类型检查——这只需要写一些代码。

2.2 分段编译工具

当创建大的项目时，分段编译尤其重要。在C/C++中，可以将一个大程序构造成为许多小程
80 序块，而这些小程序块容易管理，可独立调试。程序分割的最基本的方法是创建命名子程序。在C和C++里，子程序称为函数（*function*），函数是一段代码段，可以将这些函数放在不同的文件中，并能分别编译。另一种解释，函数是程序的基本单位，因为不能把一个函数分开，让其不同的部分放在不同的文件中；整个函数必须完整地放在一个文件里（尽管文件可拥有不止一个函数）。

当调用函数时，通常要传给它一些参数（*argument*）。这些参数是一些值，我们希望用这些值来执行函数。当函数执行完后，可得到一个返回值（*return value*），返回值是函数作为执行结果返回的一个值。但也可以编写不带参数没有返回值的函数。

程序可由多个文件构成，一个文件中的函数很可能要访问另一些文件中的函数和数据。编译一个文件时，C或C++编译器必须知道在另一些文件中的函数和数据，特别是它的名字和基本用法。编译器就是要确保函数和数据被正确地使用。“告知编译器”外部函数和数据的名称及它们的模样，这一过程就是声明（*declaration*）。一旦声明了一个函数或变量，编译器知道怎样检查对它们的引用，以确保引用正确。

2.2.1 声明与定义

声明（*declaration*）和定义（*definition*）这两个术语在整本书中都会准确地区分使用，因此必须弄清它们之间的区别。事实上，所有的C/C++程序都要求声明。编写第一个程序之前，需要了解声明的基本方法。

声明是向编译器介绍名字——标识符。它告诉编译器“这个函数或这个变量在某处可找到，它的模样像什么”。而定义是说：“在这里建立变量”或“在这里建立函数”。它为名字分配存储空间。无论定义的是函数还是变量，编译器都要为它们在定义点分配存储空间。对于变量，编译器确定变量的大小，然后在内存中开辟空间来保存变量的数据。对于函数，编译器会生成代码，这些代码最终也要占用一定的内存。 81

在C和C++中，可以在不同的地方声明相同的变量和函数，但只能有一个定义［有时这称为ODR（one-definition rule，单一定义规则）］。当连接器连接所有的目标模块时，如果发现一个函数或变量有多个定义，连接器将报告出错。

定义也可以是声明。如果定义**int x**；之前，编译器没有发现标识符**x**，编译器则把这一标识符看成是声明并立即为它分配存储空间。

2.2.1.1 函数声明的语法

C/C++的函数声明就是给函数取名、指定函数的参数类型和返回值。例如，下面是一个叫**func1( )**的函数声明，它带了两个整数类型的参数（整数类型在C/C++中以关键字**int**表示）并返回一个整数：

```
int func1(int,int);
```

第一个关键字是函数返回值类型：**int**。参数按其使用的顺序依次排在函数后面的括号内。分号说明声明结束，在这种情况下，它告诉编译器“就这些，这里没有函数定义。”

C/C++尽量使声明形式和使用形式一致。例如，假设**a**是另一个整数变量，上面的函数可以如下方式使用：

```
a = func1(2,3);
```

因为**func1( )**返回的是一个整数，C/C++编译器要检查**func1( )**的使用情况，以确保**a**能接受返回值，并且还要检查函数参数的类型匹配情况。 82

在函数声明时，可以给参数命名。编译器会忽略这些参数名，但对程序员来说它们可以帮助记忆。例如，我们有下面的形式声明**func1( )**，它与前面的声明意义相同：

```
int func1(int length, int width);
```

2.2.1.2 一点说明

对于带空参数表的函数，C和C++有很大的不同。在C语言中，声明

```
int func2();
```

表示“一个可带任意参数（任意数目，任意类型）的函数”。这就妨碍了类型检查。而在C++语言中它就意味着“不带参数的函数”。

2.2.1.3 函数的定义

函数定义看起来像函数声明，但它还有函数体。函数体是一个用大括号括起来的语句集。大括号表示这段代码的开始和结束。为了定义函数体为空的（函数体不含代码）函数**func1( )**，应当写为：

```
int func1(int length, int width) { }
```

注意，在函数定义中，大括号代替了分号的作用，因为大括号括起了一条或一组语句，

所以就不需要分号了。另外也要注意，如果要在函数体中使用参数的话，函数定义中的参数必须有名称（上面的函数没有用到定义的参数，因此在这里是可选的）。

2.2.1.4 变量声明的语法

对“变量声明”的解释向来很模糊且自相矛盾，而理解它准确的含义对于正确的理解定义和阅读程序十分重要。变量声明告知编译器变量的外表特征。这好像是对编译器说：“我知
83 道你以前没有看到过这名字，但我保证它一定在某个地方，它是X类型的变量。”

函数声明包括函数类型（即返回值类型）、函数名、参数列表和一个分号。这些信息使得编译器足以认出它是一个函数声明并可识别出这个函数的外部特征。由此推断，变量声明应该是类型标识后面跟一个标识符。例如:

```
int a;
```

可以声明变量**a**是一个整数，这符合上面的逻辑。但这就产生了一个矛盾：这段代码有足够的信息让编译器为整数**a**分配空间，而且编译器也确实给整数**a**分配了空间。要解决这个矛盾，对于C/C++需要一个关键字来说明“这只是一个声明，它的定义在别的地方”。这个关键字就是**extern**，它表示变量是在文件以外定义的，或在文件后面部分才定义。

在变量定义前加**extern**关键字表示声明一个变量但不定义它，例如:

```
extern int a;
```

extern也可用于函数声明。例如:

```
extern int func1(int length, int width);
```

这种声明方式和先前的**func1( )**声明方式一样。因为没有函数体，编译器必定把它作为声明而不是函数定义。**extern**关键字对函数来说是多余的、可选的。C语言的设计者并不要求函数声明使用**extern**，这可能有些令人遗憾；如果函数声明也要求使用**extern**，那么在形式上与变量声明更加一致，从而减少了混乱（但这就需要更多的输入，这也许能解释为什么不要求函数使用**extern**的原因）。

下面是一些声明的例子:

```
//: C02:Declare.cpp
// Declaration & definition examples
extern int i; // Declaration without definition
84 extern float f(float); // Function declaration
float b;  // Declaration & definition
float f(float a) {  // Definition
  return a + 1.0;
}

int i; // Definition
int h(int x) { // Declaration & definition
  return x + 1;
}

int main() {
  b = 1.0;
  i = 2;
  f(b);
  h(i);
} ///:~
```

函数声明时参数标识符是可选的。函数定义时则要求要有标识符（这里指C语言，而C++不要求）。

2.2.1.5 包含头文件

大部分的库包含众多的函数和变量。为了减少工作量，确保一致性，当对这些函数和变量做外部声明时，C/C++使用“头文件”（*header file*）。头文件是一个含有某个库的外部声明函数和变量的文件。它通常是扩展名为“**.h**”的文件，如**headerfile.h**（可能还会看到一些较老的程序使用其他扩展名，如“**.hxx**”或“**.hpp**”，但现在已经很少了）。

头文件由创建库的程序员提供。为了声明在库中已有的函数和变量，用户只需包含头文件即可。包含头文件，要使用**#include**预处理器命令。它告诉预处理器打开指定的头文件并在**#include**语句所在的地方插入头文件。**#include**有两种方式来指定文件：尖括号（< >）或双引号。

以尖括号指定头文件，如下所示：

```
#include <header>
```

85

用尖括号来指定文件时，预处理器是以特定的方式来寻找文件，一般是环境中或编译器命令行指定的某种寻找路径。这种设置寻找路径的机制随机器、操作系统、C++实现的不同而不同，要视具体情况而定。

以双引号指定文件，如下所示：

```
#include "local.h"
```

用双引号时，预处理器以“定义实现的途径”来寻找文件。它通常是从当前目录开始寻找，如果文件没有找到，那么include命令就按与尖括号同样的方式重新开始寻找。

包含iostream 头文件，要用如下语句

```
#include <iostream>
```

预处理器会找到 iostream 头文件（通常在“include”子目录下）并把它插入头文件所在位置。

2.2.1.6 标准C++ include 语句格式

随着C++的不断演化，不同的编译器厂商选用了不同的文件扩展名。而且，不同的操作系统对文件名有不同的限制，特别是对文件名长度限制。结果引起了对源代码的可移植性的限制。为了消除这些差别，标准使用的格式允许文件名长度可以大于众所周知的8个字符，去除了扩展名。例如，代替老式的包含**iostream.h**的语句

```
#include <iostream.h>
```

现在可以写成：

```
#include <iostream>
```

如果需要截短文件名和加上扩展名，翻译器会按照一定的方式来实现包含语句，以适应特定的编译器和操作系统。当然，如果想使用这种没有扩展名的风格，但编译器厂商没有提供这种支持，也可以将厂商提供的头文件拷贝成没有扩展名的文件。

86

从C继承下来的带有传统“**.h**”扩展名的库仍然可用。然而，也可以用更现代的C++风格

使用它们，即在文件名前加一个字母“**c**”。这样

```
#include <stdio.h>
#include <stdlib.h>
```

就变为:

```
#include <cstdio>
#include <cstdlib>
```

对所有的标准C头文件都一样。这就为读者提供了一个区分标志，说明所使用的是C还是C++库。

新的包含格式和老的效果是不一样的：使用**.h**的文件是老的、非模板化的版本，而没有**.h**的文件是新的模板化版本。如果在同一程序中混用这两种形式，会遇到某些问题。

2.2.2 连接

连接器把由编译器生成的目标模块（一般是带“**.o**”或“**.obj**”扩展名的文件）连接成为操作系统可以加载和执行的程序。它是编译过程的最后阶段。

连接器的特性随系统不同而不同。通常，只需告诉连接器目标模块和要连接的库的名称，及可执行程序的名称，连接器就可以开始执行连接任务了。一些系统要求用户自己调用连接
87 器。很多C++软件包可以让用户通过C++编译器来调用连接器。多数情况下，连接器的调用是不可见的。

某些早期的连接器对目标文件和库文件只查找一次，这些连接器从左到右查找一遍所给的目标文件和库文件列表。因此目标文件和库文件的顺序就特别重要。如果连接的时候遇到一些莫名其妙的问题，就有可能与给定连接器的文件顺序有关。

2.2.3 使用库文件

到此，了解了一些基本的术语，现在可以学习如何来使用库了。

使用库必须:

1) 包含库的头文件。

2) 使用库中的函数和变量。

3) 把库连接进可执行程序。

目标模块没有加入库时，也可执行上述步骤。对于C/C++的分段编译，包含头文件和连接目标模块是基本步骤 。

2.2.3.1 连接器如何查找库

当C或C++要对函数和变量进行外部引用时，根据引用情况，连接器会选择两种处理方法中的一种。如果还未遇到过这个函数或变量的定义，连接器会把它的标识符加到“未解析的引用”列表中。如果连接器遇到过函数或变量定义，那么这就是已解决的引用。

如果连接器在目标模块列表中不能找到函数或变量的定义，它将去查找库。库有多种索引方式，连接器不必到库里查找所有目标模块——而只需浏览索引。当连接器在库中找到定义后，就将整个目标模块而不仅仅是函数定义连接到可执行程序。注意，仅仅是库中包含所需定义的目标模块加入连接，而不是整个库参加连接（否则程序会变得毫无意义的庞大）。如
88 果想尽量减小程序的长度，当构造自己的库时，可以考虑一个源代码文件只放一个函数。这

要求更多的编辑工作[⊖]，但它对使用者来说是有益的。

因为连接器按指定的顺序查找文件，所以，用户使用与库函数同名的函数，把带有这种函数的文件插到库文件名列表之前，就能用他自己的函数取代库函数。由于在找到库文件之前，连接器已先用用户所给定的函数来解释引用，因此被使用的是用户的函数而不是库函数。注意，这可能是一个bug，并且C++名字空间禁止这样做。

2.2.3.2 秘密的附加模块

当创建一个C/C++可执行程序时，连接器会秘密连接某些模块。其中之一是启动模块，它包含了对程序的初始化例程。初始化例程是开始执行C/C++程序时必须首先执行一段程序。初始化例程建立堆栈，并初始化程序中的某些变量。

连接器总是从标准库中查找程序中调用的经过编译的“标准”函数。由于标准库总可以被找到，所以只要在程序中包含所需的头文件，就可以使用库中的任何模块，并且不必告诉连接器去找标准库。例如，标准的C++库中有iostream函数。要用这些函数，只需包含**<iostream>**头文件即可。

如果使用附加的库，必须把该库文件名添加到由连接器处理的列表文件中。

2.2.3.3 使用简单的C语言库

用C++来编写代码，并不禁止用C的库函数。事实上，整个C的库以默认方式包含在标准的C++库中。这些函数代替用户作了大量的工作，因此，使用它们，可以节约许多时间。 89

本书将尽可能地使用标准的C++库函数（也包含标准C库函数），但是只有用标准库函数才能保证程序的可移植性。在某些情况下必须使用非标准C++库函数的地方，我们也将尽量使用符合POSIX标准的函数。POSIX是基于UNIX上的一个标准，它包括的函数是C++库中没有的。通常能在UNIX（特别是Linux）平台上找到POSIX函数，也可能在DOS/Windows下找到。例如，如果要用到多线程编程，最好使用POSIX线程库，这样的代码就容易理解、端口通信和维护（POSIX线程库通常只用到操作系统提供的基本的线程设施）。

2.3 编写第一个C++程序

现在，已经了解了几乎足够的基础知识，可以创建和编译一个程序了，它将用到标准的C++ iostream类。这些iostream类可从文件和标准的输入输出设备（通常指控制台，但也可重定向到文件和设备）中读写数据。这个简单的程序将利用流对象在屏幕上显示消息。

2.3.1 使用iostream类

为了声明iostream类中的函数和外部数据，要用如下语句包含头文件：

```
#include <iostream>
```

第一个程序用到了标准输出的概念，标准输出的含义就是“发送输出的通用场所”。在其他例子中会看到使用标准输出的不同方式，但这里指输出到控制台。iostream包自动定义一个名为**cout**的变量（对象），它接受所有与标准输出绑定的数据。

⊖ 我推荐使用Perl或Python自动完成这项任务作为程序员库打包过程的一部分（参见www.Perl.org或www.Python.org）。

90 将数据发送到标准输出，要用操作符“<<”。C程序员知道这个操作符表示“向左移位”，下一章我们将会讨论。应当说向左移位与输出毫无关系。然而，C++允许操作符“重载”。操作符重载后与某种特殊类型的对象一起使用，它就有了新的含义。和iostream对象在一起，操作符“<<”意思就是“发送到”。例如

```
cout << "howdy!";
```

意思就是把字符串“howdy!”发送到**cout**对象（**cout**是“控制台输出（console output）”的简写）。

这是操作符重载的初步知识。第12章将详细讨论操作符重载。

2.3.2 名字空间

正如第1章所提到的那样，在C语言中，当程序达到一定规模之后，会遇到的一个问题是我们“用完了”函数名和标识符。当然，并非我们真正用完了所有函数名和标识符，而是简单地想出一个新名称就不太容易了。更重要的是，当程序达到一定的规模之后，通常分成许多块，每一块由不同的人或小组来构造和连接。由于所有的函数名和标识符都在同一程序里，这就要求所有的开发人员都必须非常小心，不要碰巧使用了相同的函数名和标识符，导致冲突。这种情况很快变得令人烦躁，而且浪费时间，最终导致高昂的代价。

标准的C++有预防这种冲突的机制：**namespace**关键字。库或程序中的每一个C++定义集被封装在一个名字空间中，如果其他的定义中有相同的名字，但它们在不同的名字空间，就不会产生冲突。

名字空间是十分方便和有用的工具，但名字空间的出现意味着在写程序之前，必须知道它
91 们。如果只是简单地包含头文件，使用头文件中的一些函数或对象，编译时，可能会遇到一些奇怪的错误，确切地说，如果仅仅只包含头文件，编译器无法找到任何有关函数和对象的声明。在多次看到编译器的这种提示后，我们会熟悉它所代表的含义（即“虽然包含了头文件，但所有的声明都在一个名字空间中，而没有告诉编译器我们想要用这个名字空间中的声明）。

可以用一个关键字来声明：“我要使用这个名字空间中的声明和（或）定义”。这个关键字是“**using**”。所有的标准C++库都封装在一个名字空间中，即“**std**”（代表“standard”）。由于本书常使用到这些标准的库文件，几乎在每个程序中都有如下的使用指令（**using**指令）：

```
using namespace std;
```

这意味着打开**std**名字空间，使它的所有名字都可用。有了这条语句，就不用担心特殊的库组件是在一个名字空间中，因为在使用**using**指令地方，它使名字空间在整个文件中都是可用的。

在人们费尽心机把名字空间的名字隐藏起来之后，再暴露名字空间的所有名字，这看起来是矛盾的，而事实上，应该对这样的轻率作法倍加小心（这将在本书后面解释）。但是，**using**指令仅暴露当前文件的名字，所以，它并不像起初听起来那样严重（但是，如果再想一想在头文件中这样做，这就是鲁莽的举动）。

名字空间和包含头文件的方法之间存在着相互关系。现代头文件的包含命令已标准化了（如**<iostream>**，不带扩展名“**.h**”），过去典型包含头文件的方式是带上“**.h**”，如**<iostream.h>**。那时，名字空间不是语言的一部分。所以，对已经存在的代码要提供向后兼

容，如果给出

```
#include <icstream.h>
```

92

它相当于

```
#include <iostream>
using namespace std;
```

但本书使用标准的包含格式（即不带“**.h**”），因此就必须显式地使用**using**指令。

到此，介绍了对名字空间必须了解的内容，在第10章将更全面地讨论这个问题。

2.3.3 程序的基本结构

C/C++程序是变量、函数定义、函数调用的集合。程序开始运行时，它执行初始化代码并调用一个特殊的函数“**main( )**”。程序的主要代码放在这里。

正如前面提到的，函数定义包含返回类型（在C++中必须指明）、函数名、括号内的参数列表、大括号内的函数代码。下面是一个简单的定义：

```
int function() {
  // Function code here (this is a comment)
}
```

上面这个函数，参数列表为空，函数体内只含有一条注释。

一个函数定义可有多对花括号，但必须有一对把整个函数体括起来。**main( )**也是一个函数，也必须遵守这些规则。在C++语言中，**main( )**总是返回**int**类型。

C/C++语言书写格式比较自由。除了个别情况，编译器忽略换行符和空格符，所以，必须有某种方式来判定一条语句的结束。C++语句是以分号结束。

93

C的注释行以“/*”开始，以“*/”结束，其中可以包含换行符。C++可使用C风格的注释，也有自己的注释符：“//”。注释从“//”开始，到换行结束。对于只有一行的注释，比起“/* */”来，“//”要方便得多，本书将较多地使用。

2.3.4 “Hello, World!”

现在，终于写出第一个程序：

```
//: C02:Hello.cpp
// Saying Hello with C++
#include <iostream> // Stream declarations
using namespace std;

int main() {
  cout << "Hello, World! I am "
       << 8 << " Today!" << endl;
} ///:~
```

通过“<<”操作符把一系列的参数传递给**cout**对象。然后**cout**对象按从左向右的顺序将参数打印出来。输入输出流函数**endl**表示一行结束并在行末加上一个换行符。使用输入输出流，可将一系列的参数按顺序排起来，使类易于使用。

在C语言中，用双引号括起来的正文称为“字符串”（*string*）。标准的C++类库有一个专门用于正文处理的功能强大的**string**类，所以我们将使用更精确的术语“字符数组”

(*character array*) 来描述双引号之间的正文。

编译器为字符数组分配存储空间，把每个字符相应的ASCII码存放到这个空间中。编译器在字符数组后自动加上含“0”值的额外存储片，标志数组结束。

在字符数组内，通过使用“转义序列”可以插入一些特殊的字符。转义序列是由反斜杠
94 (\) 跟上一个特殊的代码组成。例如，“**\n**” 意思是换行。编译器手册或是C语言指南给出了一组完整的转义序列，其他包括“**\t**”(跳格)，“****”(反斜杠)，“**\b**”(空格)。

注意，一条语句可占多行，整条语句以分号结束。

字符数组变量和常数混合出现在上述**cout**语句中。使用**cout**语句时，操作符“**<<**”根据所带的参数以不同的含义重载，所以当向**cout**发送不同的参数时，它能“识别应该对这个参数作何处理”。

本书中，在每个文件的第一行都有一条注释，以注释符（一般是“//”）跟一个冒号开始，而最后一行是以“//:~”开始的注释，表示文件结束。这是一点技巧，这样标记，可以很容易地从代码文件中提取信息（本书第2卷中可找到这个提取信息的程序，该卷在*www.BruceEckel.com*上）。第一行的注释中有文件名和位置信息，因此文件能被正文和其他文件引用，所以很容易从本书的源代码中找到它（源代码可从*www.BruceEckel.com*下载）。

2.3.5 运行编译器

下载并解压缩本书的源代码后，在子目录**CO2**下找到这个程序。用**Hello.cpp**作为参数调用编译器。对于这样一个简单的、单文件程序，一般的编译器都能很容易地完成它的编译。例如，用GNU C++ 编译器（它在Internet上可免费获得），可以输入

```
g++ Hello.cpp
```

95 其他编译器也有类似的语法，有关细节可参阅编译器文档。

2.4 关于输入输出流

前面所看到的仅仅是输入输出流类最基本的用法。它的输出还有另外的一些格式，比如，对于数值的输出格式有十进制、八进制、十六进制。下面是另一个使用输入输出流的例子:

```
//: C02:Stream2.cpp
// More streams features
#include <iostream>
using namespace std;

int main() {
  // Specifying formats with manipulators:
  cout << "a number in decimal: "
       << dec << 15 << endl;
  cout << "in octal: " << oct << 15 << endl;
  cout << "in hex: " << hex << 15 << endl;
  cout << "a floating-point number: "
       << 3.14159 << endl;
  cout << "non-printing char (escape): "
       << char(27) << endl;
} ///:~
```

在这个例子中，输入输出流利用iostream操作符，将数字分别以十进制、八进制和十六进

制打印出来（操作符不进行打印操作，但它改变输出流的状态）。浮点数的格式由编译器自动确定。此外，通过（显式）类型转换（*cast*），任何字符都能转换成**char**类型(**char**是保存单字符的数据类型)，发送到流对象。显式类型转换看起来像函数调用：**char( )**带上字符的ASCII码值。在上述程序中，**char(27)**是把“escape”发送到**cout**。

2.4.1 字符数组的拼接

C预处理器的一个重要功能就是可以进行字符数组的拼接（*character array concatenation*）。书中的一些例子要用到这项重要功能。如果两个加引号的字符数组邻接，并且它们之间没有标点，编译器就会把这些字符数组连接成单个字符数组。当代码列表宽度有限制时，字符数组的拼接就特别有用。 96

```
//: C02:Concat.cpp
// Character array Concatenation
#include <iostream>
using namespace std;

int main() {
  cout << "This is far too long to put on a "
    "single line but it can be broken up with "
    "no ill effects\nas long as there is no "
    "punctuation separating adjacent character "
    "arrays.\n";
} ///:~
```

初看，上述程序好像是错的，因为在每行结束没有分号。请记住C/C++是自由格式语言，虽然一般情况下看到在每行的末尾带有一个分号，但实际要求是在每个语句结束时才加分号，而一个语句很可能要写好几行。

2.4.2 读取输入数据

输入输出流类提供了读取输入的功能。用来完成标准输入功能的对象是**cin**（代表“**console input**”，控制台输入）。**cin**通常是指从控制台输入，但这种输入可以重定向来自其他输入源。后面将用例子说明。

和**cin**一起使用的输入输出流操作符是“**>>**”。这个操作符接受与参数类型相同的输入。例如，如果设定了一个整型参数，它将等待从控制台传来的一个整数。下面是一个例子：

```
//: C02:Numconv.cpp
// Converts decimal to octal and hex
#include <iostream>
using namespace std;

int main() {
  int number;
  cout << "Enter a decimal number: ";
  cin >> number;
  cout << "value in octal = 0"
       << oct << number << endl;
  cout << "value in hex = 0x"
       << hex << number << endl;
} ///:~
```

97

这个程序是将用户输入的数字转换为八进制和十六进制表示。

2.4.3 调用其他程序

典型的例子是在Unix shell脚本或DOS批处理文件中，使用从标准输入输出读写的程序。用标准的C语言**system( )**函数，C/C++程序可调用任何程序。**system( )**函数在头文件**<cstdlib>**中已声明：

```
//: C02:CallHello.cpp
// Call another program
#include <cstdlib> // Declare "system()"
using namespace std;

int main() {
  system("Hello");
} ///:~
```

为了使用**system( )**，通常需要在操作系统命令提示下输入字符数组。输入的字符数组可以包含命令行参数，字符数组也可以是运行时产生的（不只是如上面所示的使用静态字符数组）。执行命令字符数组，把控制返回给程序。

从这个程序可以看出，在C++中使用普通的C库函数是很容易的事，只要包含头文件和调用所需的库函数就行了。如果已经学过C语言，那么C与C++向上兼容的特性，会为学习C++带来很大的帮助。

2.5 字符串简介

98

虽然字符数组很有用，但它有一定的限制。简单地说它就是存放在内存中的一组字符，如果要用它做什么事情，必须处理所有细节。例如，引号内字符数组的大小在编译时就确定了。如果想在这样的字符数组中添增字符，需要了解很多有关的知识（包括动态内存管理，字符数组的拷贝、连接等），才能完成添加任务。这正是我们所希望的有一种对象能替我们完成的事。

标准的**C++string**类就是设计用来处理（并隐藏）对字符数组的低级操作，而这些操作早期是由C程序员来完成的。从有C语言以来这些操作就一直是一个编程费时、产生错误的原因。虽然本书第二卷中专门有一章介绍**string**类，但由于**string**能简化编程，对程序编写十分重要，所以，在此对它作一些介绍并加以使用。

为使用**string**类，需要包含C++头文件**<string>**。**string**类在名字空间**std**中，因此要用**using**指令。由于操作符重载，**string**类的使用是很直观的：

```
//: C02:HelloStrings.cpp
// The basics of the Standard C++ string class
#include <string>
#include <iostream>
using namespace std;

int main() {
  string s1, s2; // Empty strings
  string s3 = "Hello, World."; // Initialized
  string s4("I am"); // Also initialized
  s2 = "Today"; // Assigning to a string
```

```
  s1 = s3 + " " + s4; // Combining strings
  s1 += " 8."; // Appending to a string
  cout << s1 + s2 + "!" << endl;
} ///:~
```

前两个字符串**s1**和**s2**开始时是空的。**s3**和**s4**的两种不同初始化方法效果是相同的（也可简单地用一个**string**对象来初始化另一个**string**对象）。 99

可以用“=”来给**string**对象赋值。“=”用其右边的内容代替**string**对象先前的内容。不必为先前的内容费心，它将做自动处理。连接**string**对象，只需用“+”操作符。“+”也可将**string**连接到字符数组中。如果想将**string**加到一个**string**或字符数组之后，可以用“+=”操作符完成这一操作。最后说明一点，输入输出流知道如何来处理**string**，所以可直接向**cout**发送**string**（或能产生**string**的表达式，如上面的例子中的**s1+s2+"!"**）来打印它。

2.6 文件的读写

在C语言中，完成打开和处理文件这样复杂的操作，需要对C语言有较深的了解。然而C++语言的iostream库提供了一种简单的方法来处理文件，因此，介绍这个功能可以比在C语言中介绍这一功能更早。

为了打开文件进行读写操作，必须包含**<fstream>**。虽然**<fstream>**会自动包含**<iostream>**，但如果打算使用**cin**，**cout**，最好还是显式地包含**<iostream>**。

为了读而打开文件，要创建一个**ifstream**对象，它的用法与**cin**相同，为了写而打开文件，要创建一个**ofstream**对象，用法与**cout**相同。一旦打开一个文件，就可以像处理其他**iostream**对象那样对它进行读写，非常简单。

在iostream库中，一个十分有用的函数是**getline( )**，用它可以把一行读入到**string**对象中（以换行符结束）[⊖]。**getline( )**的第一个参数是**ifstream**对象，从中读取内容，第二个参数是**stream**对象。函数调用完成之后，**string**对象就装载了一行内容。 100

下面是一个简单的例子，将一个文件的内容拷贝到另一个文件：

```
//: C02:Scopy.cpp
// Copy one file to another, a line at a time
#include <string>
#include <fstream>
using namespace std;

int main() {
  ifstream in("Scopy.cpp"); // Open for reading
  ofstream out("Scopy2.cpp"); // Open for writing
  string s;
  while(getline(in, s)) // Discards newline char
    out << s << "\n"; // ... must add it back
} ///:~
```

从上面的程序可以看出，为了打开一个文件，只要将欲建立的文件名交给**ifstream**和**ofstream**对象即可。

这里引入了一个新概念——**while**循环。我们将在下一章对它进行详细的介绍。**while**循环

⊖ **getline( )**实际有很多参数，我们将在第2卷的“iostreams”一章中详尽讨论。

的基本思想是用**while**后面带括号中的表达式来控制下一条句（也可以是用大括号括起来的多条语句）的执行。只要括号中的表达式（在这个例子中是**getline(in,s)**）产生“true”结果，则继续执行由**while**控制的语句。就是说，如果**getline( )**成功地读入一行，它就返回“true”值。如果到达输入结束，则返回“false”。上面程序中**while**循环逐行读取输入文件，然后将它们写入到输出文件。

getline( )逐行读取字符，遇到换行符终止（终止字符是可以改变的，我们在第二卷输入
101 输出流一章再讨论）。**getline( )**将丢弃换行符而不把它存入**string**对象。因此，想使拷贝的文件看上去和源文件一样，必须加上换行符，如上所示。

另一个有趣味的例子是把整个文件拷贝成单独的一个**string**对象：

```
//: C02:FillString.cpp
// Read an entire file into a single string
#include <string>
#include <iostream>
#include <fstream>
using namespace std;

int main() {
  ifstream in("FillString.cpp");
  string s, line;
  while(getline(in, line))
    s += line + "\n";
  cout << s;
} ///:~
```

string具有动态特性，不必担心**string**的内存分配；只管添加新内容进去就行了，**string**会自动扩展以保存新的输入。

把整个文件都输入到一个字符串中，好处之一就是，**string**类有许多函数可用来对字符串进行查找和操作，使用它们可以把文件当成单个的字符串来处理。但也有一定的局限性。把一个文件作为许多行的集合而不是一大段文本来处理，通常是很方便的。例如，如果想对每一行都加上行号，把每行作为一个单独的**string**对象会非常容易。要完成这项工作，我们需用别的方法。

2.7 vector简介

使用**string**，我们可以向**string**对象输入数据而不关心需要多少存储空间。但如果把每一
102 行读入一个**string**对象，我们就不知道需要多少**string**——只有读完整个文件后才知道。为了解决这一问题，我们需要有某种能够自动扩展的存放设施，用以包含所需数量的**string**对象。

实际上，为什么要限制我们自己只存放**string**对象呢？当编写程序时，很多情况下并不知道会用到多少什么东西。如果有某种“容器”对象，它能容纳所有的各种对象，这似乎更有用。幸运的是，标准C++库有一个现成的解决方法：标准容器（container）类。容器类是标准C++非常实用的强大工具之一。

人们经常会把标准C++库的“容器”与“算法”和被称为STL的东西相混淆。STL（标准模板类库，Standard Template Library）是1994年春天Alex Stepanov在加州San Diego的会议上把他的C++库提交给C++标准委员会时使用的名称（Alex Stepanov当时在惠普公司工作）。这个名称一直沿用下来，特别是惠普决定允许这个库公开下载后，使用的人就更多了。同时，

C++标准委员会对STL作了大量的修改，将它整合进标准C++类库。SGI公司(参见*http://www.sgi.com/Technology/STL*)不断对STL进行改进。SGI 的STL与标准的C++库在许多细节上是不同的。虽然人们经常产生误解，但实际上C++标准是不“包括”STL的。由于标准C++库的“容器”和“算法”与SGI的STL有相同的来源（通常同名），因此容易引起误会。所以，本书中，将使用“标准C++库”或“标准库容器”或其他类似的说法，避免使用“STL”这个术语。

虽然标准C++库容器和算法的实现所使用的某些概念较深奥，并且在本书第2卷中专门用了两章来讲解这些概念，但即使对这些概念不太了解，也不妨碍这些库的使用。最基本的标准容器—“**vector**”非常有用，在这里对它作一些介绍，以后会经常用到。我们会发现，使用 103
vector后，可以进行大量的工作而不用关心底层的实现（再强调一下，这就是面向对象编程的一个重要目标）。当读完第2卷中有关标准类库的章节后，我们会学到更多的关于**vector**和其他容器的知识。如果本书较早的程序中使用**vector**并不像有经验的C++程序员所做的那样，这是可以理解的。一般说来，这里的多数用法还是适当的。

vector类是一个*模板*（*template*），也就是说它可有效地用于不同的类型。就是说，我们可以创建**Shape**的**vector**、**Cat**的**vector**和**String**的**vector**等等。用模板几乎可以创建“任何事物的类”。把类型名输入到尖括号内，让编译器知道**vector**所用的类（在这种情况下就是vector将要保存的类）。所以，**string**的**vector**表示为**vector<string>**。这样，就定制了只装**string**对象的**vector**。如果试图在这个**vector**中加入其他类型，编译器会给出错误提示信息。

既然**vector**表达了“容器”的概念，就应该有一定的方法把东西放进容器中，并且能从容器里把东西取出来。为了在**vector**末尾后追加一个新元素，可以使用成员函数**push_back( )**(注意，对于一个具体的对象要用“.”号来调用它的成员函数)。“**push_back( )**”这个名字看上去似乎有些冗长，不如“put”简单，这样命名是因为还有别的容器和成员函数也要向容器添加新元素。例如，**insert( )**成员函数，它是在容器中间加入新元素，**vector**支持这个函数，但它的用法更复杂，第2卷再解释它。还有**push_front( )**函数(不属于**vector**)，它是把新元素加到**vector**的开头。在**vector**中，还有很多成员函数，在标准的C++类库中，还有很多容器，但是令人惊奇的是，仅仅知道一些简单的特征就能做许多事情了。

可以用**push_back( )**向**vector**内添加新元素，但怎样从**vector**取回这些元素呢？解决的方 104
法很巧妙——操作符重载，让**vector**像数组那样使用。几乎每一种编程语言都有数组这种数据类型（下一章将对它作更多的讨论）。数组是一个集合体，即它由许多元素构成。数组的一个显著特点是它所有的元素大小相同且逐个邻接。最重要的是元素可由“下标”（indexing）选定，这意味着，只要说“我要第n个元素”，就能找到这个元素，通常很快。除了某些特例，一般的编程语言下标都用方括号表示。比如，对于一个数组**a**，想提出第5个单元，就可以写成**a[4]**（注意下标总是从0开始）。

正如“<<”和“>>”可用于iostreams类一样，通过操作符重载也可把简单有效的下标记号用于**vector**类中。不必知道重载是如何实现的——它留到下一章讨论——但是，如果知道为了使**[ ]**与**vector**一起操作而隐藏了的某些技巧，这对于加深理解是有帮助的。

了解了上述内容，现在来看一个使用**vector**的程序。为使用**vector**，必须包含头文件**<vector>**:

```
//: C02:Fillvector.cpp
```

```
// Copy an entire file into a vector of string
#include <string>
#include <iostream>
#include <fstream>
#include <vector>
using namespace std;

int main() {
  vector<string> v;
  ifstream in("Fillvector.cpp");
  string line;
  while(getline(in, line))
    v.push_back(line); // Add the line to the end
  // Add line numbers:
105 for(int i = 0; i < v.size(); i++)
    cout << i << ": " << v[i] << endl;
} ///:~
```

程序大部分与前一个程序相同，打开文件并每次将一行读进 **string** 对象。不同的是，这些**string**对象被压入**vector v**的尾部。**while**循环完成时，整个文件存在于**v**内，并驻留内存。

while语句之后是**for**循环语句。它与**while**语句相似，不过它多了一些控制条件。**for**之后的括号内是控制表达式，这和**while**语句相同。但它有一个在括号内的控制表达式，由三部分组成：第一部分初始化；第二部分检测退出循环的条件；第三部分是改变控制分步通过一系列项目的某个值。程序中的这种**for**循环方式是非常通行的用法：初始化部分**int i=0**表示用一个整数**i**作循环计数器，并初始化为**0**；检测部分表明，要使循环继续，**i**的值必须小于**vector**对象**v**中的元素个数（元素个数由成员函数**size( )**得出）；最后一部分用到了C/C++中的自增操作符，使**i**加**1**。确切地说，**i++**表示取**i**的值加上**1**，并把结果返回给**i**。所以，整个**for**循环就是取控制变量**i**，使它从**0**逐渐递增至比**vector**对象的个数小**1**时结束。对于**i**的每一个值，执行一次**cout**语句，建立一行，它由**i**的值（由**cout**转化为字符数组）、分号、空格、文件中的一行句和由**endl**产生的一个换行符组成。编译和运行这个程序，可以看出其结果是给文件加上了行号。

因为在iostreams中能使用操作符“**>>**”，所以可以很容易地修改上面的程序，使之把输入分解成由空格分隔的单词而不是一些行。

```
//: C02:GetWords.cpp
// Break a file into whitespace-separated words
106 #include <string>
#include <iostream>
#include <fstream>
#include <vector>
using namespace std;

int main() {
  vector<string> words;
  ifstream in("GetWords.cpp");
  string word;
  while(in >> word)
    words.push_back(word);
  for(int i = 0; i < words.size(); i++)
    cout << words[i] << endl;
} ///:~
```

表达式：

```
while(in >> word)
```

意思是每次取输入的一个单词，当表达式的值为“false”时，就意味着文件读完了。当然，以空白来分隔单词是比较原始的办法，这里只是举一个简单例子。后面，会看到更复杂的例子，它们可以根据任何方式分割输入。

为了进一步说明使用可带任何类型的**vector**是很容易的事，下面给出一个创建**vector<int>**的例子：

```
//: C02:Intvector.cpp
// Creating a vector that holds integers
#include <iostream>
#include <vector>
using namespace std;

int main() {
  vector<int> v;
  for(int i = 0; i < 10; i++)
    v.push_back(i);
  for(int i = 0; i < v.size(); i++)
    cout << v[i] << ", ";
  cout << endl;
  for(int i = 0; i < v.size(); i++)
    v[i] = v[i] * 10; // Assignment
  for(int i = 0; i < v.size(); i++)
    cout << v[i] << ", ";
  cout << endl;
} ///:~
```

107

创建可以存放不同类型的**vector**，只需把类型当做模板参数（即在尖括号中的参数）输入即可。提供模板和设计完善的模板库正是为了使这种使用变得容易。

在这个例子中，我们还可以看到**vector**的另外一个重要特征。在表达式

```
v[i] = v[i] * 10;
```

中可以看到，**vector**不仅仅限于输入和取出，还可以通过使用方括号的下标操作符向**vector**的任何一个单元赋值（从而改变单元的值）。这说明**vector**是通用、灵活的“暂存器”，用来处理对象集。在后面几章我们将充分地利用它。

2.8 小结

本章主要说明，如果有人已经定义了我们所需要的类，则面向对象编程是很容易的事。这时，只需简单地包含一个头文件，创建对象，并向对象发送消息。如果所用的类功能很强而且设计完善，那么我们不需费很多的力气就能编写出很好的程序。

在显示使用库类使面向对象编程变得简单的过程中，本章也介绍了标准C++库中一些最基本的和十分有用的类型：一系列的输入输出流（特别是从文件和控制台进行读写的输入输出流）、**string**类和**vector**模板。可以看到使用这些库类是多么简单。现在可以想像用它们来编写程序完成许多工作，实际上，它们能做更多的事情[⊖]。虽然本书的前几章只用了这些工

108

⊖ 如果读者急于了解这些或者其他标准类库组件的功能，参见www.BruceEckel.com和www.dinkumware.com上的本书第2卷。

具很少的一部分功能，但对于用C这样的低级语言的编程方式已经是迈出了一大步。学习C语言的低层方面是为了教学目的，同时也很费时。如果用对象来管理低层的事务，最终会更有效。毕竟，面向对象编程就是要隐藏具体的细节，使我们着眼于程序设计更大的方面。

尽管面向对象编程尽可能使编程工作在较高的层次上进行，但C语言的某些基本知识是不能不知道的，这些将在下一章中讨论。

2.9 练习

部分练习题的答案可以在本书的电子文档 "*Thinking in C++ Annotated Solution Guide*" 中找到，只需支付很少的费用就可以从*http://www.BruceEckel.com*获得这个电子文档。

2-1 修改**Hello.cpp**，使它能打印你的名字和年龄（或者你的鞋码、爱犬的年龄等，只要你喜欢）。编译并运行修改后的程序。

2-2 以**Stream2.cpp**、**Numconv.cpp**为例，编一个程序，让它根据输入的半径值求出圆面积，并打印。可以用运算符 "*" 求半径的平方。注意，不要用八进制或十六进制格式打印（它们只适用于整数类型）。

2-3 编一个程序用来打开文件并统计文件中以空格隔开的单词数目。

2-4 编一个程序统计文件中特定单词的出现次数（要求使用**string**类的运算符 "==" 来查找单词）。

2-5 修改**Fillvector.cpp**使它能从后向前打印各行。

2-6 修改**Fillvector.cpp**使它能把**vector**中的所有元素连接成单独的一个字符串，并打印，但不要加上行号。

109 2-7 编一个程序，一次显示文件的一行，然后，等待用户按回车键后显示下一行。

2-8 创建一个**vector<float>**，并用一个**for**循环语句向它输入25个浮点数，显示**vector**的结果。

2-9 创建三个**vector<float>**对象，与第8题一样填写前两个对象。编一个**for**循环，把前两个**vector**的每一个相应元素相加起来，结果放入第三个**vector**的相应元素中。显示这三个**vector**的结果。

2-10 编一个程序，创建一个**vector<float>**，像前面的练习那样输入25个数。求每个数的
110 平方，并把它们放入**vector**的同样位置。显示运算前后的**vector**。

第3章 C++中的C

因为C++是以C为基础的，所以要用C++编程就必须熟悉C的语法，就像要解决微积分问题必须要对代数十分了解一样。

111

如果读者以前从没有接触过C，本章将会提供在C++中使用C风格的一个很好的背景知识。如果读者对Kernighan & Ritchie所著的C语言书（经常称之为K&R C）第1版中描述的C风格比较熟悉的话，就会发现在C++以及在标准C中有一些新的、不一样的特征。如果读者对标准C熟悉的话，则应当通览本章找出C++中与众不同的特点。注意这里介绍的是C++的一些基本特征，这些特征和C的特征很相似或者是对C进行的一些修改。C++的更为复杂的特征将会在后面各章中介绍。

本章结合读者对其他语言编程的经验，对C的构造和C++的一些基本结构作了简要介绍。更为详细的介绍参见本书的附带光盘，“*Thinking in C: Foundations for Java & C++*”（Chuck Allison著，MindView公司出版，也可以在www.MindView.net上得到）。这是一个在光盘上的讲座，其目的是让读者仔细地浏览C语言的基础知识。它着重于从C转向使用C++或Java语言所必需的知识，而不是试图让读者成为懂得C的所有细节的专家（使用C++或Java这样的高级语言的一个原因正是由于它们可以避免涉及许多这样的细节）。它也包括练习和解答指南。记住，光盘的内容不能代替本章，而只能作为本章和本书的准备知识，因为本章超出了这张光盘所包含的内容。

3.1 创建函数

在旧版本（标准化之前）的C中，我们可以用带任意个数和类型的参数调用函数，编译器都不会报告出错。运行程序之前每一件事情似乎都很好，但当运行时，我们却会得到一些奇怪的结果（更糟的是程序崩溃），并且没有说明为什么会这样的任何提示。缺乏对参数传递的协助以及会导致高深莫测的故障，可能是C被称为“高级汇编语言”的一个原因。以标准C前的程序员只能去适应这种情况。

112

标准C和C++有一个特征叫做函数原型（function prototyping）。用函数原型，在声明和定义一个函数时，必须使用参数类型描述。这种描述就是“原型”。调用函数时，编译器使用原型确保正确传递参数并且正确地处理返回值。如果调用函数时程序员出错了，编译器就会捕获这个错误。

实际上，在前一章中已经学习了函数原型（但并没有这样命名），因为在C++中函数声明的形式需要正确的原型。在函数原型中，参数表包含了应当传递给函数的参数类型和参数的标识符（对声明而言可以是任选的）。参数的顺序和类型必须在声明、定义和函数调用中相匹配。下面是一个声明函数原型的例子：

```
int translate(float x, float y, float z);
```

在函数原型中声明变量时，不能使用和定义一般变量同样的形式。就是说不能用**float x, y, z**。必须指明每一个参数的类型。在函数声明中，下面的形式是可以接受的：

```
int translate(float, float, float);
```

因为在调用函数时，编译器只是检查类型，所以使用标识符只是为了使别人阅读代码时更加清晰。

在函数定义中，因为参数是在函数内部引用的，所以需要命名。

113
```
int translate(float x, float y, float z) {
  x = y = z;
  // ...
}
```

这条规则只应用于C。在C++中，函数定义的参数表中可以使用未命名的参数。当然，因为它没有被命名，所以不能在函数体中使用它。允许不命名参数是为了给程序员提供在“参数列表中保留位置”的一种方式。不管谁调用函数都必须使用正确的参数。但是，创建函数的人将来可以使用这个参数，而不需要强制修改调用这个函数的代码。即使给出命名，在参数表中忽略这个参数也是可能的，但每次编译函数时，会得到这个值没有被使用这样一条令人讨厌的警告消息。如果删除这个名字，这个警告也会消除。

C和C++有两种其他声明参数列表的方式。如果有一个空的参数列表，可以在C++中声明这个函数为**func( )**，它告诉编译器，这里有0个参数。应该意识到这只意味着在C++中是空参数列表。在C中，它意味着不确定的参数数目（这是C中的漏洞，因为在这种情况下不能进行类型检查）。在C和C++中，声明**func(void)**都意味着空的参数列表。在这种情况下**void**这个关键词意味着“空”（在本章的后面将会看到，就指针而言它也可以表示“没有类型”）。

在不知道会有多少个参数或什么样类型的参数时，参数表的另一种选择是*可变的参数列表*。这个“不确定参数列表”用省略号（…）表示。定义一个带可变参数列表的函数比定义一个带固定参数列表的函数要复杂得多。如果（因为某种原因）不想使用函数原型的错误检查功能，可以对有固定参数表的函数使用可变参数列表。正因为如此，应该限制对C使用可变参数列表并且在C++中避免使用（正如我们将会看到的，在C++中有更好的选择）。在你的C
114 指南的库部分对使用可变参数列表做了描述。

3.1.1 函数的返回值

C++函数原型必须指明函数的返回值类型（在C中，如果省略返回值，表示默认为整型）。返回值的类型放在函数名的前面。为了表明没有返回值可以使用**void**关键字。如果这时试图从函数返回一个值会产生错误。下面有一些完整的函数原型：

```
int f1(void); // Returns an int, takes no arguments
int f2(); // Like f1() in C++ but not in Standard C!
float f3(float, int, char, double); // Returns a float
void f4(void); // Takes no arguments, returns nothing
```

要从一个函数返回值，我们必须使用**return**语句。**return**语句退出函数返回到函数调用后的那一点。如果**return**有参数，那个参数就是函数的返回值。如果函数规定返回一个特定类型的值，那么每一个**return**语句都必须返回这个类型。在一个函数定义中可以有多个**return**语句。

```
//: C03:Return.cpp
// Use of "return"
#include <iostream>
using namespace std;
```

```
char cfunc(int i) {
  if(i == 0)
    return 'a';
  if(i == 1)
    return 'g';
  if(i == 5)
    return 'z';
  return 'c';
}

int main() {
  cout << "type an integer: ";
  int val;
  cin >> val;
  cout << cfunc(val) << endl;
} ///:~
```

115

在函数**cfunc( )**中，第一个值为真的**if**语句，通过**return**语句退出函数。注意函数声明不是必须的，因为函数在**main( )**使用它之前定义，所以编译器从函数定义中知道它。

3.1.2 使用C的函数库

用C++编程时，当前C函数库中的所有函数都可以使用。在定义自己的函数之前，应该仔细地看一下函数库，可能有人已经解决了我们的问题，而且进行了更多的思考和调试。

注意，尽管很多编译器包含大量的额外函数可以使编程更加容易、吸引大家去使用，但是这并不是标准C库的一部分。如果我们肯定不想移植该应用程序到别的平台上（谁又能肯定呢？），那么就使用那些函数，让编程更加容易。如果希望该应用程序具有可移植性，就应该限制使用标准库函数。如果必须执行特定平台的活动，应当尽力把代码隔离在某一场所，以便移植到另一平台时容易进行修改。C++中，经常把特定平台的活动封装在一个类中，这是一个理想的解决办法。

使用库函数的方法如下：首先，在编程参考资料中查找函数（很多编程参考资料按字母顺序排序函数）。函数的描述应该包括说明代码语法的部分。这部分的头部通常至少有一**#include**行，表示包含函数原型的头文件。在程序文件中复制这个**#include**行，所以能正确声明函数。现在可以按照函数出现在语法部分的同样方式来调用它。如果出错了，编译器通过把函数调用和头文件中的函数原型相比较来报告错误。连接器通过默认路径查找标准库，所以在编程时需要做的就是包含这个头文件和调用这个函数。

116

3.1.3 通过库管理器创建自己的库

我们可以将自己的函数收集到一个库中。大多数编程包带有一个库管理器来管理对象模块组。每一个库管理器有它自己的命令，但有这样一个共同的想法：如果想创建一个库，那么就建立一个头文件，它包含库中的所有函数原型。把这个头文件放置在预处理器搜索路径中的某处，或者在当前目录中（以便能被**#include**“头文件”发现），或者在包含路径中（以便能被**#include**<头文件>发现）。现在把所有的对象模块连同建成后的库名传递给库管理器（大多数库管理器要求有一个共同的扩展名，例如**.lib**或**.a**）。把建成的库和其他库放置在同一个位置以便连接器能发现它。当使用自己的库时，必须向命令行添加一些东西，让连接器知

道为你调用的函数查找库。因为函数库随着系统而异，所以必须在你的系统手册中查找所有的细节。

3.2 执行控制语句

本节涵盖了C++中的执行控制语句。在读写C或C++代码之前，必须熟悉这些语句。

C++使用C的所有执行控制语句。这些语句包括**if-else**、**while**、**do-while**、**for**和**switch**选择语句。C++也允许使用声名狼藉的**goto**语句，在本书中会避免使用它。

3.2.1 真和假

所有的条件语句都使用条件表达式的真或假来判定执行路径。**A == B**是一个条件表达式
117 的例子。这里使用条件运算符“==”确定变量**A**是否等于变量**B**。表达式产生布尔值**true**（真）或**false**（假）（这只是C++中的关键字，在C中如果一个表达式等于非零值则为“真”）。其他的条件运算符有：**>**、**<**、**>=**等。条件语句在本章的后面会有更详细的介绍。

3.2.2 if-else语句

if-else语句有两种形式：用**else**或不用**else**。这两种形式是：

```
if(表达式)
    语句
```

或

```
if(表达式)
    语句
else
    语句
```

“表达式”的值为真或假。“语句”是以一个分号结束的简单语句，或一组包含在大括号里的简单语句构成的一个复合语句。不管什么时候使用“语句”，都意味着是简单语句或复合语句。注意这个语句也可能是另一个**if**语句，所以它们可连成一串。

```
//: C03:Ifthen.cpp
// Demonstration of if and if-else conditionals
#include <iostream>
using namespace std;

int main() {
  int i;
  cout << "type a number and 'Enter'" << endl;
  cin >> i;
  if(i > 5)
    cout << "It's greater than 5" << endl;
  else
    if(i < 5)
      cout << "It's less than 5 " << endl;
    else
      cout << "It's equal to 5 " << endl;

118 cout << "type a number and 'Enter'" << endl;
  cin >> i;
```

```
  if(i < 10)
    if(i > 5)  // "if" is just another statement
      cout << "5 < i < 10" << endl;
    else
      cout << "i <= 5" << endl;
  else // Matches "if(i < 10)"
    cout << "i >= 10" << endl;
} ///:~
```

缩进控制流语句体是一种习惯用法，以便读者可以很方便地知道它的起点和终点[⊖]。

3.2.3 while语句

while、**do-while**和**for**语句是循环控制语句。一个语句重复执行直到控制表达式的计值为假。**while**循环的形式是：

while(表达式)
    语句

循环一开始就对表达式进行计算。并在每次重复执行语句之前再次计算。

下面的例子一直在**while**循环体内执行，直到输入密码或按control-C键。

```
//: C03:Guess.cpp
// Guess a number (demonstrates "while")
#include <iostream>
using namespace std;

int main() {
  int secret = 15;
  int guess = 0;
  // "!=" is the "not-equal" conditional:
  while(guess != secret) { // Compound statement
    cout << "guess the number: ";
    cin >> guess;
  }
  cout << "You guessed it!" << endl;
} ///:~
```

119

while语句的条件表达式并不仅限于像上面的例子那样只进行一个简单的测试；它也可以像我们希望的那样复杂，只要能产生一个真或假的结果。我们甚至会看到没有循环体而只有一个分号代码：

```
while(/* Do a lot here */)
  ;
```

在这样的情况下，程序员写出了不但执行测试也可以进行自己工作的条件表达式。

3.2.4 do-while语句

do-while的形式是：

do
    语句
while(表达式)

⊖ 注意，在出现某种缩排约定后，所有的习惯用法都将终结。代码格式风格之间的争执是无休止的。对本书编码风格的描述请看附录A。

do-while语句与**while**语句的区别在于，即使表达式第一次计值就等于假，前面的语句也会至少执行一次。在一般的**while**语句中，如果条件第一次为假，语句一次也不会执行。

如果在程序**Guess.cpp**中使用**do-while**，变量**guess**不需要初始为**0**值，因为在它被检测之前就已被**cin**语句初始化了：

```
//: C03:Guess2.cpp
// The guess program using do-while
#include <iostream>
using namespace std;

int main() {
  int secret = 15;
  int guess; // No initialization needed here
  do {
120   cout << "guess the number: ";
    cin >> guess; // Initialization happens
  }   while(guess != secret);
  cout << "You got it!" << endl;
} ///:~
```

因为某种原因，大多数程序员更喜欢只使用**while**语句而避免使用**do-while**语句。

3.2.5 for语句

在第一次循环前，**for**循环执行初始化。然后它执行条件测试，并在每一次循环结束时执行某种形式的“步进”。**for**循环的形式是：

```
for(initialization; conditional; step)
    语句
```

表达式中的*initialization*、*conditional*或*step*都可能为空。一旦进入**for**循环，*initialization*代码就执行。在每一次循环之前，*conditional*被测试（如果它的计值一开始就为假，语句就不会执行）。每一次循环结束时，执行*step*。

for循环通常用于“计数”任务：

```
//: C03:Charlist.cpp
// Display all the ASCII characters
// Demonstrates "for"
#include <iostream>
using namespace std;

int main() {
  for(int i = 0; i < 128; i = i + 1)
    if (i != 26)  // ANSI Terminal Clear screen
      cout << " value: " << i
           << " character: "
           << char(i) // Type conversion
           << endl;
} ///:~
```

读者也许会注意到，变量**i**是在使用它的地方定义，而不是在‘{’所标注的程序块起始处
121 定义。这和传统的过程语言（包括C）形成了对照，过程语言要求在程序块的起始处定义所有的变量。这将在本章的后面讨论。

3.2.6 关键字break和continue

在任何一个**while**、**do-while**或**for**循环的结构体中，都能够使用**break**和**continue**控制循环的流程。**break**语句退出循环，不再执行循环中的剩余语句。**continue**语句停止执行当前的循环，返回到循环的起始处开始新的一轮循环。

作为**break**和**continue**语句的一个例子，下面程序是一个非常简单的菜单系统：

```
//: C03:Menu.cpp
// Simple menu program demonstrating
// the use of "break" and "continue"
#include <iostream>
using namespace std;

int main() {
  char c; // To hold response
  while(true) {
    cout << "MAIN MENU:" << endl;
    cout << "l: left, r: right, q: quit -> ";
    cin >> c;
    if(c == 'q')
      break; // Out of "while(1)"
    if(c == 'l') {
      cout << "LEFT MENU:" << endl;
      cout << "select a or b: ";
      cin >> c;
      if(c == 'a') {
        cout << "you chose 'a'" << endl;
        continue; // Back to main menu
      }
      if(c == 'b') {
        cout << "you chose 'b'" << endl;
        continue; // Back to main menu
      }
      else {
      cout << "you didn't choose a or b!"
           << endl;
      continue; // Back to main menu
    }
  }
  if(c == 'r') {
    cout << "RIGHT MENU:" << endl;
    cout << "select c or d: ";
    cin >> c;
    if(c == 'c') {
      cout << "you chose 'c'" << endl;
      continue; // Back to main menu
    }
    if(c == 'd') {
      cout << "you chose 'd'" << endl;
      continue; // Back to main menu
    }
    else {
      cout << "you didn't choose c or d!"
           << endl;
      continue; // Back to main menu
```

122

```
      }
    }
    cout << "you must type l or r or q!" << endl;
  }
  cout << "quitting menu..." << endl;
} ///:~
```

如果用户在主菜单中选择‘q’，则用关键字**break**退出，选择其他，程序则继续执行。在每一个子菜单选择后，关键字**continue**用于跳转到while循环的起始处。

while(true)语句等价于“永远执行这个循环”。当用户按‘q’时，**break**语句使程序跳出这个无限循环。

3.2.7 switch语句

switch语句根据一个整型表达式的值从几段代码中选择执行。它的形式是:

123

```
switch(selector) {
    case integral-value1 : statement; break;
    case integral-value2 : statement; break;
    case integral-value3 : statement; break;
    case integral-value4 : statement; break;
    case integral-value5 : statement; break;
    (...)
    default: statement;
}
```

选择器（*selector*）是一个产生整数值的表达式。**switch**语句把*选择器*（*selector*）的结果和每一个*整数值*（*integral-value*）比较。如果发现匹配，就执行对应的语句（简单语句或复合语句）。如果都不匹配，则执行**default**语句。

读者也许会注意到上面定义中的每一个**case**后面都以一个**break**语句作为结束，这个break语句使得执行跳转到**switch**语句体的结束处（完成**switch**的闭括号处）。这是建立**switch**语句的一种常用方式，但是**break**是可选的。如果省略它，**case**语句会顺序执行它后面的语句。也就是说，执行后面的各**case**语句代码，直到遇到一个**break**语句。尽管一般不需要这种举动，但是对于一个有经验的程序员来说这可能是有用的。

switch语句是一种清晰的实现多路选择的方式（即对不同的执行路径进行选择），但它需要一个能在编译时求得整数值的选择器。例如，如果想使用一个字符串类型的对象作为一个选择器，在**switch**语句中它是不能用的。对于字符串类型的选择器，必须使用一系列**if**语句并比较在条件中的字符串。

上面的菜单程序提供了一个特别好的**switch**语句例子:

```
//: C03:Menu2.cpp
// A menu using a switch statement
#include <iostream>
using namespace std;

int main() {
  bool quit = false;  // Flag for quitting
  while(quit == false) {
    cout << "Select a, b, c or q to quit: ";
    char response;
    cin >> response;
```

124

```
    switch(response) {
      case 'a' : cout << "you chose 'a'" << endl;
                 break;
      case 'b' : cout << "you chose 'b'" << endl;
                 break;
      case 'c' : cout << "you chose 'c'" << endl;
                 break;
      case 'q' : cout << "quitting menu" << endl;
                 quit = true;
                 break;
      default  : cout << "Please use a,b,c or q!"
                 << endl;
    }
  }
} ///:~
```

quit（退出）标志是**bool**（boolean的简写）型的，这种类型只有在C++中才会看到。它只能有**true**或**false**值。选择‘q’即设置**quit**标志为**true**。下一次计算选择器的值，**quit == false**返回**false**，所以不执行**while**循环体。

3.2.8 使用和滥用goto

因为关键字**goto**存在于C中，所以C++中也支持它。**goto**是一种不好的编程方式，经常避免使用**goto**。在多数情况下的确如此。想使用**goto**语句时，查一下程序代码，看是否有其他的解决方法。在少数情况下，可能会发现**goto**语句能够解决用别的方法不能解决的问题，但是尽管如此，还应仔细考虑一下。下面是一个例子，可能会作出似乎有理的选择：

```
//: C03:gotoKeyword.cpp
// The infamous goto is supported in C++
#include <iostream>
using namespace std;

int main() {
  long val = 0;
  for(int i = 1; i < 1000; i++) {
    for(int j = 1; j < 100; j += 10) {
      val = i * j;
      if(val > 47000)
        goto bottom;
        // Break would only go to the outer 'for'
    }
  }
  bottom: // A label
  cout << val << endl;
} ///:~
```


一个可供选择的方法是设置一个布尔值，在外层**for**循环对它进行测试，然后利用**break**从内层**for**循环跳出。然而，如果我们有几层**for**语句或**while**语句，可能会出现困难。

3.2.9 递归

递归是十分有趣的，有时也是非常有用的编程技巧，凭借递归可以调用我们所在的函数。当然，如果这是所做的全部，那么会一直调用下去，直到内存用完，所以一定要有一种确定

“达到底点”递归调用的方法。在下面的例子中，只要递归到**cat**的值超过‘Z’，递归就“达到底点”：[⊖]

```
//: C03:CatsInHats.cpp
// Simple demonstration of recursion
#include <iostream>
using namespace std;

void removeHat(char cat) {
  for(char c = 'A'; c < cat; c++)
    cout << "  ";
  if(cat <= 'Z') {
    cout << "cat " << cat << endl;
    removeHat(cat + 1); // Recursive call
  } else
    cout << "VOOM!!!" << endl;
}

126
int main() {
  removeHat('A');
} ///:~
```

在**removeHat( )**中，只要**cat**的值小于‘Z’，就会在**removeHat( )**中调用**removeHat( )**，从而实现递归。每次调用**removeHat( )**，它的参数比当前的**cat**值增加1，所以参数不断增加。

求解某些具有随意性的复杂问题经常使用递归，因为这时解的具体“大小”不受限制，函数可以一直递归调用，直到问题解决。

3.3 运算符简介

我们可以把运算符看做是一种特殊的函数（C++的运算符重载正是以这种方式对待运算符）。一个运算符带一个或更多的参数并产生一个新值。运算符参数和普通的函数调用参数相比在形式上不同，但是作用是一样的。

根据读者以前的编程经验，应该习惯于迄今使用的运算符。任何一种编程语言的加（+）、减和单目减（-）、乘（*）、除（/）和赋值（=）的概念都有同样的意义。本章后面列举出全部运算符集。

3.3.1 优先级

运算符优先级规定表达式中出现多个不同运算符时计值的运算顺序。C和C++中有具体的规则决定计值顺序。最容易记住的是先乘、除，后加、减。如果一个表达式的运算顺序对我们来说是不清晰的，那么对于任何一个读代码的人来说它都可能是不清晰的，所以应该使用
127 括号使计值次序更加清晰。例如：

```
A = X + Y - 2/2 + Z;
```

与带有一组特定的圆括号的同一语句：

```
A = X + (Y - 2)/(2 + Z);
```

⊖ 感谢Kris C. Matson建议这个练习题。

具有完全不同的含义。（令X = 1, Y = 2, Z = 3，试算一下结果。）

3.3.2 自增和自减

C有不少捷径，因此C++也有很多捷径。这些捷径使得更易于输入代码，但有时却不易于阅读。可能C语言的设计者认为如果程序员的眼睛不必浏览大范围的印刷区域，那么理解一段巧妙的代码可能是比较容易的。

其中一个较好的捷径是自增和自减运算符。经常使用它们去改变循环变量以控制循环执行的次数。

自减运算符是‘--’，意思是“减小一个单位”。自增运算符是‘++’，意思是“增加一个单位”。例如，如果A是一个整数，则++A等于（A = A + 1）。自增和自减产生一个变量的值作为结果。如果运算符在变量之前出现（即++A），则先执行运算，再产生结果值。如果运算符在变量之后出现（即A++），则产生当前值，再执行运算。例如：

```
//: C03:AutoIncrement.cpp
// Shows use of auto-increment
// and auto-decrement operators.
#include <iostream>
using namespace std;

int main() {
  int i = 0;
  int j = 0;
  cout << ++i << endl; // Pre-increment
  cout << j++ << endl; // Post-increment
  cout << --i << endl; // Pre-decrement
  cout << j-- << endl; // Post decrement
} ///:~
```

128

如果我们曾经对“C++”这个名字感到奇怪，那么现在应该明白了。C++隐含的意思就是“在C上更进一步”。

3.4 数据类型简介

在编写程序中，*数据类型*（*data type*）定义使用存储空间（内存）的方式。通过定义数据类型，告诉编译器怎样创建一片特定的存储空间，以及怎样操纵这片存储空间。

数据类型可以是内部的或抽象的。内部数据类型是编译器本来能理解的数据类型，直接与编译器关联。C和C++中的内部数据类型几乎是一样的。相反，用户定义的数据类型是我们和别的程序员创建的类型，作为一个类。它们一般被称为抽象数据类型。编译器启动时，知道怎样处理内部数据类型；编译器再通过读包含类声明的头文件（在后面几章我们会了解到这一点）认识怎样处理抽象数据类型。

3.4.1 基本内部类型

标准C的内部类型（由C++继承）规范不说明每一个内部类型必须有多少位。规范只规定内部类型必须能存储的最大值和最小值。如果机器基于二进制，则最大值可以直接转换成容纳这个值所需的最少位数。然而，例如，如果一个机器使用二进制编码的十进制（BCD）来

表示数字，在机器中容纳每一种数据类型的最大数值的空间是不同的。系统头文件**limits.h**和**float.h**中定义了不同的数据类型可能存储的最大值和最小值（在C++中，一般用**#include <climits>** 和**<cfloat>**代替）.

129

C和C++中有4个基本的内部数据类型，这里的描述是基于二进制的机器。**char**是用于存储字符的，使用最小的8位（一个字节）的存储，尽管它可能占用更大的空间。**Int**存储整数值，使用最小两个字节的存储空间。**float**和**double**类型存储浮点数，一般使用IEEE的浮点格式。**float**用于单精度浮点数，**double**用于双精度浮点数。

如前所述，我们可以在某一作用域的任何地方定义变量，可以同时定义和初始化它们。下面是怎样用这四种基本数据类型定义变量的例子:

```
//: C03:Basic.cpp
// Defining the four basic data
// types in C and C++

int main() {
  // Definition without initialization:
  char protein;
  int carbohydrates;
  float fiber;
  double fat;
  // Simultaneous definition & initialization:
  char pizza = 'A', pop = 'Z';
  int dongdings = 100, twinkles = 150,
    heehos = 200;
  float chocolate = 3.14159;
  // Exponential notation:
  double fudge_ripple = 6e-4;
} ///:~
```

程序的第一部分定义了4种基本数据类型的变量，没有对变量初始化。如果不初始化一个变量，标准会认为没有定义它的内容（通常，这意味着它们的内容是垃圾）。程序的第二部分同时定义和初始化变量（如果可能，最好在定义时提供初始值）。注意常量6e-4中指数符号的使用，意思是“6乘以10的负4次幂”。

130

3.4.2 bool类型与true和false

在**bool**类型成为标准C++的一部分之前，每个人都想使用不同的方法产生类似**bool**类型的行为。这产生了可移植性问题，可能会引入微妙的错误。

标准C++的**bool**类型有两种由内部的常量**true**（转换为整数1）和**false**（转换为整数0）表示的状态。这3个名字都是关键字。此外，一些语言元素也已经被采纳:

元　素	布尔类型的用法
&& || !	带布尔参数并产生**bool** 结果
< > <= >= == !=	产生**bool** 结果
if, for, while, do	条件表达式转换为**bool**值
? :	第一个操作数转换为**bool**值

因为有很多现存的代码使用整型**int**表示一个标志，所以编译器隐式转换**int**为**bool**（非零值为**true**而零值为**false**）。理想的情况下，编译器会给我们一个警告，建议纠正这种情况。

用++把一个标志设置为真是一种“糟糕的编程风格”。这样做依然是允许的，但受到抵制，意味着在将来的某个时候它可能是不合法的。问题在于从**bool**到**int**做了隐式类型转换，增加了值（可能超过了0和1的正常布尔值的范围），然后再做相反的隐式转换。

指针（本章的后面将会引入）在必要的时候也自动转换成**bool**值。 131

3.4.3 说明符

说明符（specifier）用于改变基本内部类型的含义并把它们扩展成一个更大的集合。有4个说明符：**long**、**short**、**signed**和**unsigned**。

long和**short**修改数据类型具有的最大值和最小值。一般的**int**必须至少有**short int**型的大小。整数类型的大小等级是：**short int**、**int**、**long int**。只要满足最小/最大值的要求，所有的大小可以看成是一样的。例如，在64位字的机器上，所有的数据类型都可能是64位的。

浮点数的大小等级是：**float**、**double**和**long double**。“long float”是不合法的类型，也没有**short**浮点数。

signed和**unsigned**修饰符告诉编译器怎样使用整数类型和字符的符号位（浮点数总含有一个符号）。**unsigned**数不保存符号，因此有一个多余的位可用，所以它能存储比**signed**数大两倍的正数。**signed**是默认的，只有**char**才一定要使用**signed**；**char**可以默认为**signed**，也可以不默认为**signed**。通过规定**signed char**，可以强制使用符号位。

下面的例子使用**sizeof**运算符显示用字节表示的数据类型的大小，该运算符在本章的后面介绍：

```
//: C03:Specify.cpp
// Demonstrates the use of specifiers
#include <iostream>
using namespace std;

int main() {
  char c;
  unsigned char cu;
  int i;
  unsigned int iu;
  short int is;
  short iis; // Same as short int
  unsigned short int isu;
  unsigned short iisu;
  long int il;
  long iil;  // Same as long int
  unsigned long int ilu;
  unsigned long iilu;
  float f;
  double d;
  long double ld;
  cout
    << "\n char= " << sizeof(c)
    << "\n unsigned char = " << sizeof(cu)
    << "\n int = " << sizeof(i)
    << "\n unsigned int = " << sizeof(iu)
    << "\n short = " << sizeof(is)
    << "\n unsigned short = " << sizeof(isu)
```

132

```
    << "\n long = " << sizeof(il)
    << "\n unsigned long = " << sizeof(ilu)
    << "\n float = " << sizeof(f)
    << "\n double = " << sizeof(d)
    << "\n long double = " << sizeof(ld)
    << endl;
} ///:~
```

因为标准中只规定每一种不同类型的最大值和最小值必须是一致的（如前所述），所以应该意识到在不同的机器/操作系统/编译器上运行这个程序得到的结果可能是不同的。

如上所示，当用**short**或**long**改变**int**时，关键字**int**是可选的。

3.4.4 指针简介

不管什么时候运行一个程序，都是首先把它装入（一般从磁盘装入）计算机内存。因此，程序中的所有元素都驻留在内存的某处。内存一般被布置成一系列连续的内存位置；我们通常把这些位置看做是8位字节，但实际上每一个空间的大小取决于具体机器的结构，一般称为
133 机器的字长（*word size*）。每一个空间可按它的地址与其他空间区分。为了便于讨论，我们认为所有机器都使用有连续地址的字节从零开始，一直到该计算机的内存的上限。

因为程序运行时驻留内存中，所以程序中的每一个元素都有地址。假设我们从一个简单的程序开始：

```
//: C03:YourPets1.cpp
#include <iostream>
using namespace std;

int dog, cat, bird, fish;

void f(int pet) {
  cout << "pet id number: " << pet << endl;
}

int main() {
  int i, j, k;
} ///:~
```

程序运行的时候，程序中的每一个元素在内存中都占有一个位置。甚至函数也占用内存。我们将会看到，定义什么样的元素和定义元素的方式通常决定元素在内存中放置的地方。

C和C++中有一个运算符会告诉我们元素的地址。这就是‘**&**’运算符。只要在标识符前加上‘**&**’，就会得出标识符的地址。可以修改程序**YourPets1.cpp**，用以打印所有元素的地址。修改如下：

```
//: C03:YourPets2.cpp
#include <iostream>
using namespace std;

int dog, cat, bird, fish;

void f(int pet) {
  cout << "pet id number: " << pet << endl;
134 }
int main() {
```

```
  int i, j, k;
  cout << "f(): " << (long)&f << endl;
  cout << "dog: " << (long)&dog << endl;
  cout << "cat: " << (long)&cat << endl;
  cout << "bird: " << (long)&bird << endl;
  cout << "fish: " << (long)&fish << endl;
  cout << "i: " << (long)&i << endl;
  cout << "j: " << (long)&j << endl;
  cout << "k: " << (long)&k << endl;
} ///:~
```

(long)是一种*类型转换*（*cast*）。意思是“不要把它看做是原来的类型，而是看做是**long**类型”。这个类型转换不是必须的，但是如果没有的话，地址是以十六进制的形式打印，所以转换为**long**类型会增加一些可读性。

这个程序的结果会随计算机、操作系统和各种其他的因素的不同而变化，但我们总会看到一些有趣的现象。在我的计算机上运行一次的结果如下：

```
f(): 4198736
dog: 4323632
cat: 4323636
bird: 4323640
fish: 4323644
i: 6684160
j: 6684156
k: 6684152
```

现在可以看到在函数**main( )**的内部和外部定义的变量存放在不同的区域；当对语言有更多的了解时，就会明白为什么如此。同样，**f( )**出现在它自己的区域，在内存中代码和数据一般是分开存放的。

另一个值得注意的有趣的事情是，相继定义的变量在内存中是连续存放的。它们根据各自的数据类型所要求的字节数分隔开。这个例子中只使用了整型数据类型，变量**cat**距离变量**dog** 135
4个字节，变量**bird**距离变量**cat** 4个字节，等等。所以在这台机器上，一个**int**占4个字节。

这个有趣的实验显示了怎样分配内存，那么利用地址能干什么呢?能做的最重要的事就是，把地址存放在别的变量中以便以后使用。C和C++有一个专门的存放地址的变量类型。这个变量叫做*指针*（*pointer*）。

定义指针的运算符和用于乘法的运算符‘*’是一样的。正如我们将看到的那样，编译器会根据它所在的上下文知道它表示的不是乘法。

定义一个指针时，必须规定它指向的变量类型。可以先给出一个类型名，然后不是立即给出变量的标识符，而是在类型和标识符之间插入一个星号，这就是说“等一等，它是一个指针”。一个指向**int**的指针如下所示：

```
int* ip; // ip points to an int variable
```

把‘*’和类型联系起来似乎是很明白且易读的，但是事实上可能容易产生错觉。有人可能更倾向于说“整型指针”好像它是一个单独的类型。可是，对于**int**或其他的基本数据类型，可以写成

```
int a, b, c;
```

而对于指针，可能想写成

```
int* ipa, ipb, ipc;
```

C的语法（并由C++语法继承）不允许像这样合乎情理的表达。在上面的定义中，只有**ipa**是一个指针，而**ipb**和**ipc**是一般的**int**（可以认为“*和标识符结合得更紧密”）。因此，最好是每一行定义一个指针；这样就能得到一个清晰的语法而不会混淆：

```
int* ipa;
int* ipb;
int* ipc;
```

136

C++编程的一般原则是在定义时进行初始化，事实上这种形式工作得很好。例如，上面的变量并没有初始化为任何一个特定的值，它们所具有的是一些无意义的值。如果写成下面的形式会更好：

```
int a = 47;
int* ipa = &a;
```

现在已经初始化了**a**和**ipa**，**ipa**存放**a**的地址。

一旦有一个初始化了的指针，我们能做的最基本的事就是利用指针来修改它指向的值。要通过指针访问变量，可以使用以前定义指针使用的同样的运算符来间接引用这个指针，如像：

```
*ipa = 100;
```

现在**a**的值是100而不是47。

这些是指针基础：可以保存地址，可以使用地址去修改原先的变量。但还是留下问题：为什么要通过另一个变量作为代理来修改一个变量？

通过对指针的介绍，我们可以把答案分为两大类：

1) 为了能在函数内改变“外部对象”。这可能是指针最基本的用途，并且在下一小节对它进行验证。
2) 为了获得许多灵活的编程技巧，而这些将在本书的其余部分见到。

3.4.5 修改外部对象

137

通常，向函数传递参数时，在函数内部生成该参数的一个拷贝。这称为**按值传递**（*pass-by-value*）。在下面的程序中能看到按值传递的效果：

```
//: C03:PassByValue.cpp
#include <iostream>
using namespace std;

void f(int a) {
  cout << "a = " << a << endl;
  a = 5;
  cout << "a = " << a << endl;
}

int main() {
  int x = 47;
  cout << "x = " << x << endl;
  f(x);
  cout << "x = " << x << endl;
} ///:~
```

在函数**f( )**中，**a**是一个局部变量（*local variable*），它只有在调用函数**f( )**期间存在。因为它是一个函数参数，所以调用函数时通过参数传递来初始化**a**的值；在**main( )**中参数是**x**，其值为47，所以当调用函数**f( )**时，这个值被拷贝到**a**中。

当运行这个程序时，我们会看到：

```
x = 47
a = 47
a = 5
x = 47
```

当然，最初，**x**的值是47。调用**f( )**时，在函数调用期间为变量**a**分配临时空间，拷贝**x**的值给**a**来初始化它，这可以通过打印结果得到验证。当然，我们可以改变**a**的值并显示它被改变。但是**f( )**调用结束时，分配给**a**的临时空间就消失了，我们可以看到，在**a**和**x**之间的曾经发生过的惟一联系，是在把**x**的值拷贝到**a**的时候。 138

当在函数**f( )**内部时，变量**x**就是外部对象（*outside object*）（我用的术语）。显然，改变局部变量并不会影响外部变量，因为它们分别放在存储空间的不同位置。但是，如果我们的确想修改外部对象那又该怎么办呢？这时指针就该派上用场了。在某种意义上，指针是另一个变量的别名。所以如果我们不是传递一个普通的值而是传递一个指针给函数，实际上就是传递外部对象的别名，使函数能修改外部对象，如像：

```
//: C03:PassAddress.cpp
#include <iostream>
using namespace std;

void f(int* p) {
  cout << "p = " << p << endl;
  cout << "*p = " << *p << endl;
  *p = 5;
  cout << "p = " << p << endl;
}

int main() {
  int x = 47;
  cout << "x = " << x << endl;
  cout << "&x = " << &x << endl;
  f(&x);
  cout << "x = " << x << endl;
} ///:~
```

现在函数**f( )**把指针作为参数，并且在赋值期间间接引用这个指针，这就使得外部对象**x**被修改。这时的输出是：

```
x = 47
&x = 0065FE00
p = 0065FE00
*p = 47
p = 0065FE00
x = 5
```

注意，**p**中的值就是变量**x**的地址，指针**p**的确是指向变量**x**。如果这还不够令人信服，当改变指针**p**指向的变量值并间接引用赋值为5，我们看到变量**x**的值现在已经改变为5了。 139

因此，通过给函数传递指针可以允许函数修改外部对象。后面我们将看到指针有很多其

他的用途，但是这是最基本的，可能也是最常用的用途。

3.4.6 C++引用简介

在C和C++中指针的作用基本上是一样的，但是C++增加了另外一种给函数传递地址的途径。这就是*按引用传递*（*pass-by-reference*），它也存在于一些其他的编程语言中，并不是C++的发明。

可能一开始我们会觉得没有必要使用引用，可以不用引用编写所有的程序。一般说来，除开在本书后面将要知道的一些重要地方，这是确实的。在后面我们将对引用有更多的了解，但是基本思想和前面所述的指针的使用是一样的：我们可以用引用传递参数地址。引用和指针的不同之处在于，带引用的函数调用比带指针的函数调用在语法构成上更清晰（在某种情况下，使用引用实质上的确只是语法构成上不同）。如果使用引用来修改程序**PassAddress.cpp**，我们能看到在**main( )**中函数调用的不同：

```
//: C03:PassReference.cpp
#include <iostream>
using namespace std;

void f(int& r) {
  cout << "r = " << r << endl;
  cout << "&r = " << &r << endl;
  r = 5;
  cout << "r = " << r << endl;
}

int main() {
  int x = 47;
  cout << "x = " << x << endl;
  cout << "&x = " << &x << endl;
140 f(x); // Looks like pass-by-value,
        // is actually pass by reference
  cout << "x = " << x << endl;
} ///:~
```

在函数**f( )**的参数列表中，不用**int***来传递指针，而是用**int&**来传递引用。在**f( )**中，如果仅仅写‘**r**’（如果**r**是一个指针，会产生一个地址值）会得到**r**引用的变量值。如果对**r**赋值，实际上是给**r**引用的变量赋值。事实上，得到**r**中存放的地址值的惟一方法是用‘**&**’运算符。

在函数**main( )**中，我们能看到引用在调用函数**f( )**中的重要作用，其语法形式还是**f(x)**。尽管这看起来像是一般的按值传递，但是实际上引用的作用是传递地址，而不是值的一个拷贝。输出结果是：

```
x = 47
&x = 0065FE00
r = 47
&r = 0065FE00
r = 5
x = 5
```

所以我们可以看到，以引用传递允许一个函数去修改外部对象，就像传递一个指针所做的那样（读者可能也注意到引用使得地址传递这个事实不太明显，这在本书的后面会得到检验）。因此，通过这个简单的介绍，我们可以认为引用仅仅是语法上的一种不同方法（有时称

为“语法糖”），它和指针完成同样的任务：允许函数去改变外部对象。

3.4.7 用指针和引用作为修饰符

迄今为止，我们已经看到了基本的数据类型**char**、**int**、**float**和**double**，看到了修饰符**signed**、**unsigned**、**short**和**long**，它们可以和基本的数据类型结合使用。现在我们增加了指针和引用（它们与基本数据类型和修饰符是独立的），所以可能产生三倍的结合： 141

```
//: C03:AllDefinitions.cpp
// All possible combinations of basic data types,
// specifiers, pointers and references
#include <iostream>
using namespace std;

void f1(char c, int i, float f, double d);
void f2(short int si, long int li, long double ld);
void f3(unsigned char uc, unsigned int ui,
  unsigned short int usi, unsigned long int uli);
void f4(char* cp, int* ip, float* fp, double* dp);
void f5(short int* sip, long int* lip,
  long double* ldp);
void f6(unsigned char* ucp, unsigned int* uip,
  unsigned short int* usip,
  unsigned long int* ulip);
void f7(char& cr, int& ir, float& fr, double& dr);
void f8(short int& sir, long int& lir,
  long double& ldr);
void f9(unsigned char& ucr, unsigned int& uir,
  unsigned short int& usir,
  unsigned long int& ulir);

int main() {} ///:~
```

当传递对象进出函数时，指针和引用也能工作；我们将会在后面的一章了解到这些内容。

这里有和指针一起工作的另一种类型：**void**。如果声明指针是**void***，它意味着任何类型的地址都可以间接引用那个指针（而如果声明**int***，则只能对**int**型变量的地址间接引用那个指针）。例如：

```
//: C03:VoidPointer.cpp
int main() {
  void* vp;
  char c;
  int i;
  float f;
  double d;
  // The address of ANY type can be
  // assigned to a void pointer:
  vp = &c;
  vp = &i;
  vp = &f;
  vp = &d;
} ///:~
```

142

一旦我们间接引用一个**void***，就会丢失关于类型的信息。这意味着在使用前，必须转换

为正确的类型:

```
//: C03:CastFromVoidPointer.cpp
int main() {
  int i = 99;
  void* vp = &i;
  // Can't dereference a void pointer:
  // *vp = 3; // Compile-time error
  // Must cast back to int before dereferencing:
  *((int*)vp) = 3;
} ///:~
```

转换**(int*)vp**告诉编译器把**void***当做**int***处理，因此可以成功地对它间接引用。读者可能注意到，这个语法很难看，的确如此，但是更糟的是，**void***在语言类型系统中引入了一个漏洞。也就是说，它允许甚至是提倡把一种类型看做另一种类型。在上面的例子中，通过把**vp**转换为**int***，把一个整型看做是一个整型，但是，并没有说不能把它转换为一个**char***或**double***，这将改变已经分配给**int**的存储空间的大小，可能会引起程序崩溃。一般来说，应当避免使用**void**指针，只有在一些少见的特殊情况下才用，到本书的后面才需要考虑这些。

我们不能使用**void**引用，其原因将在第11章说明。

3.5 作用域

作用域规则告诉我们一个变量的有效范围，它在哪里创建，在哪里销毁（也就是说，超
143 出了作用域）。变量的有效作用域从它的定义点开始，到和定义变量之前最邻近的开括号配对的第一个闭括号。也就是说，作用域由变量所在的最近一对括号确定。说明如下:

```
//: C03:Scope.cpp
// How variables are scoped
int main() {
  int scp1;
  // scp1 visible here
  {
    // scp1 still visible here
    //.....
    int scp2;
    // scp2 visible here
    //.....
    {
      // scp1 & scp2 still visible here
      //..
      int scp3;
      // scp1, scp2 & scp3 visible here
      // ...
    } // <-- scp3 destroyed here
    // scp3 not available here
    // scp1 & scp2 still visible here
    // ...
  } // <-- scp2 destroyed here
  // scp3 & scp2 not available here
  // scp1 still visible here
  //..
} // <-- scp1 destroyed here
///:~
```

上面的例子表明什么时候变量是可见的，什么时候变量是不可用的（即变量越出其作用域）。只有在变量的作用域内，才能使用它。作用域可以嵌套，即在一对大括号里面有其他的大括号对。嵌套意味着可以在我们所处的作用域内访问外层作用域的一个变量。上面的例子中，变量**scp1**在所有的作用域内都可用，而**scp3**只能在最里面的作用域内才可用。 144

实时定义变量

正如在本章前面提到的那样，定义变量时，C和C++有着显著的区别。这两种语言都要求变量使用前必须定义，但是C（和很多其他的传统过程语言）强制在作用域的开始处就定义所有的变量，以便在编译器创建一个块时，能给所有这些变量分配空间。

读C代码时，进入一个作用域，首先看到的是一个变量的定义块。在块的开始部分声明所有的变量，要求程序员以一种特定的方式写程序，因为语言的实现细节需要这样。大多数人在写代码之前并不知道他们将要使用的所有变量，所以他们必须不停地跳转回块的开头来插入新的变量，这是很不方便的，也会引起错误。这些变量定义对读者来说并没有很多含义，它们实际上只是容易引起混乱，因为它们出现的地方远离使用它们的上下文。

C++(不是C)允许在作用域内的任意地方定义变量，所以可以在正好使用它之前定义。此外，可以在定义变量时对它进行初始化以防止犯某种类型的错误。以这种方式定义变量使得编写代码更容易，减少了在一个作用域内不停地来回跳转造成的问题。因为可以在使用变量的上下文中看到所定义的变量，所以代码更容易理解。同时定义并初始化一个变量是非常重要的。通过使用变量的方式我们可以看到初始化一个变量值的意义。

我们还可以在**for**循环和**while**循环的控制表达式内定义变量，在**if**语句的条件表达式和**switch**的选择器语句内定义变量。下面是一个显示随时定义变量的例子：

```
//: C03:OnTheFly.cpp
// On-the-fly variable definitions
#include <iostream>
using namespace std;                                    145
int main() {
  //..
  { // Begin a new scope
    int q = 0; // C requires definitions here
    //..
    // Define at point of use:
    for(int i = 0; i < 100; i++) {
      q++; // q comes from a larger scope
      // Definition at the end of the scope:
      int p = 12;
    }
    int p = 1;  // A different p
  } // End scope containing q & outer p
  cout << "Type characters:" << endl;
  while(char c = cin.get() != 'q') {
    cout << c << " wasn't it" << endl;
    if(char x = c == 'a' || c == 'b')
      cout << "You typed a or b" << endl;
    else
      cout << "You typed " << x << endl;
  }
```

```
  cout << "Type A, B, or C" << endl;
  switch(int i = cin.get()) {
    case 'A': cout << "Snap" << endl; break;
    case 'B': cout << "Crackle" << endl; break;
    case 'C': cout << "Pop" << endl; break;
    default: cout << "Not A, B or C!" << endl;
  }
} ///:~
```

在最内层的作用域里，**p**是在作用域结束之前定义的，所以它只是一个毫无意义的表示（但它表明可以在任何地方定义一个变量）。在外层作用域中的**p**也是一样的情况。

在**for**循环的控制表达式中**i**的定义正是一个在需要的地方定义变量的例子（只能在C++中这样做）。**i**的作用域是**for**循环控制的表达式的作用域，所以可以轮到下一次**for**循环并重新使用**i**。这是在C++中一个非常方便和常用的用法；**i**是循环计数器的一个常用名字，我们不必费
146 神取新的名字。

尽管例子表明在**while**语句、**if**语句和**switch**语句中也可以定义变量，但是可能因为语法受到许多限制，这种定义不如在**for**的表达式中常用。例如，我们不能有任何插入括号。也就是说，不可以写出：

```
while((char c = cin.get()) != 'q')
```

附加的括号似乎是合理的，并且能做很有用的事，但因为无法使用它们，结果就不像所希望的那样。问题是因为‘!=’比‘=’的优先级高，所以**char c**最终含有的值是由**bool**转换为**char**的。当打印出来时，我们在很多终端上会看到一个笑脸字符。

通常，可以认为在**while**语句、**if**语句和**switch**语句中定义变量的能力是为了完备性，但是惟一使用这种变量定义的地方可能是在**for**循环中（在那里可能使用得十分频繁）。

3.6 指定存储空间分配

创建一个变量时，我们拥有指定变量生存期的很多选择，指定怎样给变量分配存储空间，以及指定编译器怎样处理这些变量。

3.6.1 全局变量

全局变量是在所有函数体的外部定义的，程序的所有部分（甚至其他文件中的代码）都可以使用。全局变量不受作用域的影响，总是可用的（也就是说，全局变量的生命期一直到程序的结束）。如果在一个文件中使用**extern**关键字来声明另一个文件中存在的全局变量，那么这个文件可以使用这个数据。例如：

```
//: C03:Global.cpp
//{L} Global2
// Demonstration of global variables
147 #include <iostream>
using namespace std;

int globe;
void func();
int main() {
  globe = 12;
```

```
  cout << globe << endl;
  func(); // Modifies globe
  cout << globe << endl;
} ///:~
```

下面的程序把**globe**作为一个外部变量来访问:

```
//: C03:Global2.cpp {O}
// Accessing external global variables
extern int globe;
// (The linker resolves the reference)
void func() {
  globe = 47;
} ///:~
```

变量**globe**的存储空间是由程序**Global.cpp**中的定义创建的，在**Global2.cpp**的代码中可以访问同一个变量。由于**Global2.cpp**和**Global.cpp**的代码是分段编译的，必须通过声明:

```
extern int globe;
```

告诉编译器变量存在哪里。

运行这个程序时，会看到函数**func( )**的调用的确影响**globe**的全局实例。

在**Global.cpp**中，可能看到下面这个特殊的注释标记（这是我自己的设计):

```
//{L} Global2
```

这是说要创建最后的程序，带有**Global2**名字的目标文件必须被连接进来（这里没有扩展名是因为目标文件的扩展名在不同的系统中是不一样的)。在**Global2.cpp**中，第一行有另一个特殊的注释标记**{O}**，意思是“不要从这个文件生成可执行文件，编译它是为了把它连接进一些其他的可执行文件中。” 本书第2卷中的**ExtractCode.cpp**程序（在*www.BruceEckel.com*上可以下载）阅读这些标记并生成适当的**makefile**使得每一个文件被正确地编译（在本章结束时将会了解**makefile**)。 148

3.6.2 局部变量

局部变量出现在一个作用域内，它们是局限于一个函数的。局部变量经常被称为*自动变量*（*automatic variable*)，因为它们在进入作用域时自动生成，离开作用域时自动消失。关键字**auto**可以显式地说明这个问题，但是局部变量默认为**auto**，所以没有必要声明为**auto**。

3.6.2.1 寄存器变量

寄存器变量是一种局部变量。关键字**register**告诉编译器“尽可能快地访问这个变量”。加快访问速度取决于实现，但是，正如名字所暗示的那样，这经常是通过在寄存器中放置变量来做到的。这并不能保证将变量放置在寄存器中，甚至也不能保证提高访问速度。这只是对编译器的一个暗示。

使用**register**变量是有限制的。不可能得到或计算**register**变量的地址。**register**变量只能在一个块中声明（不可能有全局的或静态的**register**变量)。然而可以在一个函数中（即在参数表中）使用**register**变量作为一个形式参数。

一般地，不应当推测编译器的优化器，因为它可能比我们做得更好。因此，最好避免使用关键字**register**。

3.6.3 静态变量

关键字**static**有一些独特的意义。通常，函数中定义的局部变量在函数作用域结束时消失。
149 当再次调用这个函数时，会重新创建该变量的存储空间，其值会被重新初始化。如果想使局部变量的值在程序的整个生命期里仍然存在，我们可以定义函数的局部变量为static（静态的），并给它一个初始值。初始化只在函数第一次调用时执行，函数调用之间变量的值保持不变。用这种方式，函数可以“记住”函数调用之间的一些信息片断。

我们可能奇怪为什么不使用全局变量。**static**变量的优点是在函数范围之外它是不可用的，所以它不可能被轻易地改变。这会使错误局部化。

下面是一个使用**static**变量的例子：

```
//: C03:Static.cpp
// Using a static variable in a function
#include <iostream>
using namespace std;

void func() {
  static int i = 0;
  cout << "i = " << ++i << endl;
}

int main() {
  for(int x = 0; x < 10; x++)
    func();
} ///:~
```

每一次在for循环中调用函数**func( )**时，它都打印不同的值。如果不使用关键字**static**，打印出的值总是‘1’。

static的第二层意思和前面的含义相关，即“在某个作用域外不可访问”。当应用**static**于函数名和所有函数外部的变量时，它的意思是“在文件的外部不可以使用这个名字”。函数名或变量是局部于文件的；我们说它具有文件作用域（*file scope*）。例如，编译和连接下面两个文件会引起连接器错误：

150
```
//: C03:FileStatic.cpp
// File scope demonstration. Compiling and
// linking this file with FileStatic2.cpp
// will cause a linker error

// File scope means only available in this file:
static int fs;

int main() {
  fs = 1;
} ///:~
```

尽管在下面的文件中变量**fs**被声明为**extern**，但是连接器不会找到它，因为在**FileStatic.cpp**中它被声明为**static**。

```
//: C03:FileStatic2.cpp {O}
// Trying to reference fs
extern int fs;
```

```
void func() {
  fs = 100;
} ///:~
```

static说明符也可能在一个类中使用。当在本书的后面了解了如何创建类的时候，再对此作出解释。

3.6.4 外部变量

前面已经简要地描述和说明了**extern**关键字。它告诉编译器存在着一个变量和函数，即使编译器在当前编译的文件中没有看到它。这个变量或函数可能在另一个文件中或者在当前文件的后面定义。下面是一个例子:

```
//: C03:Forward.cpp
// Forward function & data declarations
#include <iostream>
using namespace std;

// This is not actually external, but the
// compiler must be told it exists somewhere:
extern int i;
extern void func();
int main() {
  i = 0;
  func();
}
int i; // The data definition
void func() {
  i++;
  cout << i;
} ///:~
```

151

当编译器遇到‘**extern int i**’时，它知道**i**肯定作为全局变量存在于某处。当编译器看到变量**i**的定义时，并没有看到别的声明，所以知道它在文件的前面已经找到了同样声明的**i**。如果已经把变量**i**定义为**static**，又要告诉编译器，**i**是全局定义的（通过**extern**），但是，它也有文件作用域（通过**static**），所以编译器会产生错误。

3.6.4.1 连接

为了理解C和C++程序的行为，必须对连接（*linkage*）有所了解。在一个执行程序中，标识符代表存放变量或被编译过的函数体的存储空间。连接用连接器所见的方式描述存储空间。连结方式有两种：内部连接（*internal linkage*）和外部连接（*external linkage*）。

内部连接意味着只对正被编译的文件创建存储空间。用内部连接，别的文件可以使用相同的标识符或全局变量，连接器不会发现冲突——也就是为每一个标识符创建单独的存储空间。在C和C++中，内部连接是由关键字**static**指定的。

外部连接意味着为所有被编译过的文件创建一片单独的存储空间。一旦创建存储空间，连接器必须解决所有对这片存储空间的引用。全局变量和函数名有外部连接。通过用关键字**extern**声明，可以从其他文件访问这些变量和函数。函数之外定义的所有变量（在C++中除了**const**）和函数定义默认为外部连接。可以使用关键字**static**特地强制它们具有内部连接，也可以在定义时使用关键字**extern**显式指定标识符具有外部连接。在C中，不必用**extern**定义变量

152

或函数，但是在C++中对于**const**有时必须使用。

调用函数时，自动（局部）变量只是临时存在于堆栈中。连接器不知道自动变量，所以这些变量没有连接。

3.6.5 常量

在旧版本（标准前）的C中，如果想建立一个常量，必须使用预处理器：

```
#define PI 3.14159
```

无论在何地使用**PI**，都会被预处理器用值3.14159代替（在C和C++中都可以使用这个方法）。

当使用预处理器创建常量时，我们在编译器的范围之外能控制这些常量。对名字**PI**上不进行类型检查，也不能得到**PI**的地址（所以不能向**PI**传递一个指针和一个引用）。**PI**不能是用户定义的类型变量。**PI**的意义是从定义它的地方持续到文件结束的地方；预处理器并不识别作用域。

C++引入了命名常量的概念，命名常量就像变量一样，只是它的值不能改变。修饰符**const**告诉编译器这个名字表示常量。不管是内部的还是用户定义的数据类型都可以定义为**const**。如果定义了某对象为常量，然后试图修改它，编译器将会产生错误。

必须用下述方式说明一个常量类型：

```
const int x = 10;
```

153 在标准C和C++中，可以在参数列表中使用命名常量，即使列表中的参数是指针或引用（也就是说，可以获得**const**的地址）。**const**就像正常的变量一样有作用域，所以可以在函数中“隐藏”一个**const**，确保名字不会影响程序的其余部分。

const由C++采用，并加进标准C中，尽管它们很不一样。在C中，编译器对待**const**如同变量一样，只不过带有一个特殊的标记，意思是“不要改变我”。当在C中定义**const**时，编译器为它创建存储空间，所以如果在两个不同的文件中(或在头文件中)定义多个同名的**const**，连接器将生成发生冲突的错误消息。在C中使用**const**和在C++中使用**const**是完全不一样的（简而言之，在C++中使用得更好）。

3.6.5.1 常量值

在C++中，一个**const**必须有初始值（在C中不是这样）。内部类型的常量值可以表示为十进制、八进制、十六进制、浮点数（不幸的是，二进制数被认为是不重要的）或字符。

如果没有其他的线索，编译器会认为常量值是十进制。数值47、0和1101都被认为是十进制数。

常量值前带0被认为是八进制数（基数为8）。基数为8的数值只能含有数字0～7；编译器标记其他数字为错误。017是一个合法的八进制数（相当于基数为10的数值15）。

常量值前带0x被认为是十六进制数（基数为16）。基数为16的数值只能含有数字0～9和字母a～f或A～F。0x1fe是一个合法十六进制数（相当于基数为10的数值510）。

浮点数可以含有小数点和指数幂（用e表示，意思是“10的幂”）。小数点和e都可以任选。如果给一个浮点变量赋一个常量值，编译器会取得这个常量值并把它转换为浮点数（这个过
154 程是*隐式类型转换*（*implicit type conversion*）的一种形式）。但是，使用小数点或e对于提醒

读者当前正在使用的是浮点数是一个好主意；一些更旧的编译器也会需要这种暗示。

合法的浮点常量值包括：1e4、1.0001、47.0、0.0和－1.159e－77。我们可以对数加后缀强加浮点数类型：**f**或**F**强加**float**型，**L**或**l**强加**long double**型，否则是**double**型。

字符常量是用单引号括起来的字符，如‘**A**’、‘**0**’、‘ ’。注意字符‘**0**’（ASCII 96）和数值**0**之间存在巨大差别。用“反斜线”表示一些特殊的字符：‘**\n**’（换行），‘**\t**’（制表符），‘\\’（反斜线），‘**\r**’（回车），‘\"’（双引号），‘\'’（单引号），等等。也可以用八进制表示字符常量（如‘**\17**’）或用十六进制表示字符常量（如‘**\xff**’）。

3.6.6 volatile变量

限定词**const**告诉编译器“这是不会改变的”（这就允许编译器执行额外的优化）；而限定词**volatile**则告诉编译器“不知道何时会改变”，防止编译器依据变量的稳定性作任何优化。当读在代码控制之外的某个值时，例如读一块通信硬件中的寄存器，将使用这个关键字。无论何时需要**volatile**变量的值，都能读到，即使在该行之前刚刚读过。

“在代码的控制之外”的某个存储空间的一个特殊例子是在多线程程序中。如果正在观察被另一个线程或进程修改的特殊标识符，这个标识符应该是**volatile**的，所以编译器不会认为它能够对标识符的多次读入进行优化。

注意当编译器不进行优化时，**volatile**可能不起作用，但是当开始优化代码时（当编译器开始寻找冗余的读入时），可以防止出现重大的错误。

后面有一章将进一步阐述**const**和**volatile**关键字。 155

3.7 运算符及其使用

本节说明C和C++中的所有运算符。

所有的运算符都会从它们的操作数中产生一个值。除了赋值、自增、自减运算符之外，运算符所产生的值不会修改操作数。修改操作数被称为*副作用*(*side effect*)。一般使用修改操作数的运算符就是为了产生这种副作用，但是应该记住它们所产生的值就像没有副作用的运算符产生的值一样都是可以使用的。

3.7.1 赋值

赋值操作由运算符“=”实现。这意味着“取右边的值［通常称之为*右值*（*rvalue*）］并把它拷贝给左边［通常称之为*左值*（*lvalue*）］”。右值可以是任意的常量、变量或能产生值的表达式，但是左值必须是一个明确命名的变量（也就是说，应该有一个存储数据的物理空间）。例如，可以给一个变量赋值常量(**A = 4;**)，但是不能给常量赋任何值，因为它不能是左值（不能用**4 = A;**）。

3.7.2 数学运算符

基本的数学运算符和在大多数的编程语言中使用的一样：加（+）、减（－）、除（/）、乘（*）和取模（%；从整数相除得到余数）。整数相除会截取结果的整数部分（不舍入）。浮点数不能使用取模运算符。

C和C++也使用一种简化的符号来同时执行操作和赋值。这是由一个运算符后面跟着一个

等号来表示的，并且与语言中的各种运算符结合（只要有意义）。例如，要给变量x加4并赋值给x作为结果，可以写成**x += 4;**。

156 下面例子显示了数学运算符的使用：

```
//: C03:Mathops.cpp
// Mathematical operators
#include <iostream>
using namespace std;

// A macro to display a string and a value.
#define PRINT(STR, VAR) \
  cout << STR " = " << VAR << endl

int main() {
  int i, j, k;
  float u, v, w;  // Applies to doubles, too
  cout << "enter an integer: ";
  cin >> j;
  cout << "enter another integer: ";
  cin >> k;
  PRINT("j",j);  PRINT("k",k);
  i = j + k; PRINT("j + k",i);
  i = j - k; PRINT("j - k",i);
  i = k / j; PRINT("k / j",i);
  i = k * j; PRINT("k * j",i);
  i = k % j; PRINT("k % j",i);
  // The following only works with integers:
  j %= k; PRINT("j %= k", j);
  cout << "Enter a floating-point number: ";
  cin >> v;
  cout << "Enter another floating-point number:";
  cin >> w;
  PRINT("v",v); PRINT("w",w);
  u = v + w; PRINT("v + w", u);
  u = v - w; PRINT("v - w", u);
  u = v * w; PRINT("v * w", u);
  u = v / w; PRINT("v / w", u);
  // The following works for ints, chars,
  // and doubles too:
  PRINT("u", u); PRINT("v", v);
  u += v; PRINT("u += v", u);
  u -= v; PRINT("u -= v", u);
  u *= v; PRINT("u *= v", u);
  u /= v; PRINT("u /= v", u);
} ///:~
```

157 当然所有赋值的右值都可以更为复杂。

3.7.2.1 预处理宏介绍

注意，使用宏**PRINT( )**可以节省输入（和避免输入错误！）。传统上用大写字母来命名预处理宏以便突出它——后面我们很快会了解到宏有可能会变得危险（它们也可能非常有用）。

跟在宏名后面的括号中的参数会被闭括号后面的所有代码替代。只要在调用宏的地方，预处理程序就删除名字**PRINT**并替换代码，所以使用宏名时编译器不会报告任何错误信息，它并不对参数做任何类型检查（正如本章后面宏调试中显示的那样，后者可能是有益的）。

3.7.3 关系运算符

关系运算符在操作数之间建立一种关系。如果关系为真，则产生布尔（在C++中用关键字**bool**表示）值**true**；如果关系为假，则产生布尔值**false**。关系运算符有：小于(<)，大于 (>)，小于等于 (<=)，大于等于 (>=)，等于 (==)，不等于 (!=)。在C和C++中，它们可以使用所有的内部数据类型。在C++中，对用户所定义的数据类型可以给出它们的特殊定义（在第12章讨论运算符重载时将了解这些内容）。

3.7.4 逻辑运算符

逻辑运算符“与”(**&&**)和“或”(||)依据它们的参数的逻辑关系产生**true**或**false**。记住在C和C++中，如果语句是非零值则为**true**，如果是零则为**false**。如果打印一个**bool**值，一般会看到‘**1**’表示**true**、‘**0**’表示**false**。

下面例子使用了关系运算符和逻辑运算符：

```
//: C03:Boolean.cpp
// Relational and logical operators.
#include <iostream>
using namespace std;
int main() {
  int i,j;
  cout << "Enter an integer: ";
  cin >> i;
  cout << "Enter another integer: ";
  cin >> j;
  cout << "i > j is " << (i > j) << endl;
  cout << "i < j is " << (i < j) << endl;
  cout << "i >= j is " << (i >= j) << endl;
  cout << "i <= j is " << (i <= j) << endl;
  cout << "i == j is " << (i == j) << endl;
  cout << "i != j is " << (i != j) << endl;
  cout << "i && j is " << (i && j) << endl;
  cout << "i || j is " << (i || j) << endl;
  cout << " (i < 10) && (j < 10) is "
       << ((i < 10) && (j < 10))  << endl;
} ///:~
```

在上面的程序中，我们可以用**float**或**double**代替**int**定义。但是，注意浮点数和零的比较是很严格的，一个数和另一个数即使只有最小小数位不同仍然是“不相等”。一个最小小数位大于0的浮点数仍为真。

3.7.5 位运算符

位运算符允许在一个数中处理个别的位（因为浮点数使用一种特殊的内部格式，所以位运算符只适用于整型**char**、**int**和**long**）。位运算符对参数中的相应位做布尔代数运算来产生结果。

如果两个输入位都是1，则“与”运算符（&）在结果位上产生1，否则为0。如果两个输入位有一个是1，则“或”运算符（|）在结果位上产生1，只有当两个输入位都是0时，结果位才为0。如果两个输入位之一是1而不是同时为1，则位的*异或运算符xor*（^）的结果位为1。位的“非”运算符（~，也称为*补运算符*）是一个一元运算符，它只带一个参数（其他的运算

符都是二元运算符)。非运算符运算的结果和输入位相反，即输入位为0时结果位为1，输入位为1时结果位为0。

位运算符可以和“=”结合来统一运算和赋值：&=、|=和^=都是合法运算（因为~是一元运算符，所以不能和=结合)。

3.7.6 移位运算符

移位运算符也是对位的操纵。左移位运算符(<<)引起运算符左边的操作数向左移动，移动位数由运算符后面的操作数指定。右移位运算符(>>)引起运算符左边的操作数向右移动，移动位数由运算符后面的操作数指定。如果移位运算符后面的值比运算符左边的操作数的位数大，则结果是不定的。如果左边的操作数是无符号的，右移是逻辑移位，所以最高位补零。如果左边的操作数是有符号的，右移可能是也可能不是逻辑移位（也就是说，行为是不定的)。

移位可以和等号结合(<<=或>>=)。左值由左值按右值移位后的结果代替。

下面是一个例子，说明所有涉及位运算的运算符的使用。首先，这里单独创建了一个通用的函数，用二进制格式打印一个字节，所以这个函数很容易被重用。头文件声明了这个函数:

```
//: C03:printBinary.h
// Display a byte in binary
void printBinary(const unsigned char val);
///:~
```

下面是这个函数的实现:

```
//: C03:printBinary.cpp {O}
#include <iostream>
void printBinary(const unsigned char val) {
  for(int i = 7; i >= 0; i--)
    if(val & (1 << i))
      std::cout << "1";
    else
      std::cout << "0";
} ///:~
```


函数**printBinary( )**取出一个字节并一位一位地显示出来。表达式

```
(1 << i)
```

在每一个相继位的位置产生一个1，例如：00000001, 00000010,等等。如果这一位和变量**val**按位与并且结果不是零，就表明**val**的这一位为1。

最后，在例子中使用下面的函数显示位操作运算符:

```
//: C03:Bitwise.cpp
//{L} printBinary
// Demonstration of bit manipulation
#include "printBinary.h"
#include <iostream>
using namespace std;

// A macro to save typing:
#define PR(STR, EXPR) \
  cout << STR; printBinary(EXPR); cout << endl;
```

```
int main() {
  unsigned int getval;
  unsigned char a, b;
  cout << "Enter a number between 0 and 255: ";
  cin >> getval; a = getval;
  PR("a in binary: ", a);
  cout << "Enter a number between 0 and 255: ";
  cin >> getval; b = getval;
  PR("b in binary: ", b);
  PR("a | b = ", a | b);
  PR("a & b = ", a & b);
  PR("a ^ b = ", a ^ b);
  PR("~a = ", ~a);
  PR("~b = ", ~b);
  // An interesting bit pattern:
  unsigned char c = 0x5A;
  PR("c in binary: ", c);
  a |= c;
  PR("a |= c; a = ", a);
  b &= c;
  PR("b &= c; b = ", b);
  b ^= a;
  PR("b ^= a; b = ", b);
} ///:~
```

161

再一次使用预处理宏节省输入。它打印你选择的字符串，然后是一个表达式的二进制表示形式，再后是换行。

在**main( )**中，变量都是**unsigned**的。这是因为一般来说，在使用字节进行工作时并不希望用带符号数。对于变量**getval**而言，可能要使用**int**来替代**char**，因为语句“**cin >>**”以另一种方式把第一个数字看做是一个字符。通过把**getval**赋值给**a**和**b**，该值被转换为一个单独的字节（通过对它截尾）。

“**<<**”和“**>>**”实现位的移位功能，但是当移位越出数的一端时，那些位就会丢失（这就是通常所说的，那些位掉进了神秘的*位桶*（*bit bucket*）中，丢弃在这个桶中的位有可能需要重用）。操作位的时候，也可以执行*旋转*（*rotation*），即在一端移掉的位插入到另一端，好像它们在绕着一个回路旋转。尽管大多数计算机处理器提供了机器级的旋转命令（所以我们会在这种处理器的汇编语言中看到它），但在C和C++中，不直接支持旋转。大概C的设计者认为对“旋转”的处理应该适可而止（正如他们说的那样，他们的目标是建立最小的语言），因此我们可以建立自己的旋转命令。例如下面是实现左旋和右旋的函数：

```
//: C03:Rotation.cpp {O}
// Perform left and right rotations

unsigned char rol(unsigned char val) {
  int highbit;
  if(val & 0x80) // 0x80 is the high bit only
    highbit = 1;
  else
    highbit = 0;
  // Left shift (bottom bit becomes 0):
  val <<= 1;
```

162

```
  // Rotate the high bit onto the bottom:
  val |= highbit;
  return val;
}

unsigned char ror(unsigned char val) {
  int lowbit;
  if(val & 1) // Check the low bit
    lowbit = 1;
  else
    lowbit = 0;
  val >>= 1; // Right shift by one position
  // Rotate the low bit onto the top:
  val |= (lowbit << 7);
  return val;
} ///:~
```

试着在程序**Bitwise.cpp**中使用这些函数。注意，在使用这些函数前编译器必须在**Bitwise.cpp**中看到**rol( )**和**ror( )**的定义（或者至少是声明）。

通常情况下，使用位函数的效率非常高，因为它们被直接翻译成汇编语言语句。有时一个单独的C或C++语句会产生一行单独的汇编代码。

3.7.7 一元运算符

位的非运算不是惟一使用一个参数的运算符。和它一样，*逻辑非*(**!**)对一个**true**值得到一个**false**值。一元减(**-**)和一元加(**+**)是和二元减和二元加一样的运算符；根据表达式的书写方式，编译器能辨别属于哪一种用法。例如，语句

```
x = -a;
```

有明确的含义。

163 编译器可以理解

```
x = a * -b;
```

但是读者可能迷惑，所以写成

```
x = a * (-b);
```

更保险。

一元减得到一个负值。一元加实际上并不做任何事，只是和一元减相对应。

本章前面介绍了增量和减量运算符(**++**和**--**)。它们是涉及赋值的运算符中仅有的有副作用的运算符。这两个运算符使变量增加或减少一个单位，尽管对于不同的数据类型，“单位”可能有不同的含义——特别是对指针来说。

最后的一元运算符有C和C++中的地址运算符(**&**)，间接引用 (*****和**->**)和强制类型转换运算符，以及C++中的**new**和**delete**。在本章的叙述中，地址和间接引用只与指针一起使用。类型转换在本章后面叙述，**new**和**delete**将在第4章介绍。

3.7.8 三元运算符

三元运算符**if-else**与众不同，因为它有三个操作数。这的确是一个运算符因为它产生一个值，而不是像一般的**if-else**语句那样。它由三个表达式组成：如果第一个表达式（后面跟有一

个问号？）的计值为**true**，则对紧跟在问号后面的表达式求值，它的结果就是运算符的结果。如果第一个表达式为**false**，就执行第三个表达式（在冒号后面），它的结果就是运算符的结果。

可以使用**if-else**这个条件运算符的副作用或者它产生的值。下面的代码段说明了这两种情况：

```
a = --b ? b : (b = -99);
```

这里，条件产生右值。如果**b**自减运算的结果非零，则把**b**的值赋给**a**。如果**b**变为零，**a**和 164
b都被赋值为-99。**b**总是在递减，但是只有在**b**递减为0时，它才会被赋值为-99。可以使用如下不带“**a =**”的类似语句来利用它的副作用：

```
--b ? b : (b = -99);
```

在这里第二个b是多余的，因为运算符产生的值是无用的。但在“？”和“:”之间需要一个表达式。在这种情况下，这个表达式可以是一个常量，它能使代码运行得更快一点。

3.7.9 逗号运算符

逗号并不只是在定义多个变量时用来分隔变量，如像

```
int i, j, k;
```

当然，它也用于函数参数列表中。然而，它也可能作为一个运算符用于分隔表达式。在这种情况下，它只产生最后一个表达式的值。在逗号分隔的列表中，其余的表达式的计算只完成它们的副作用。下面的例子自增一串变量，并把最后一个作为右值：

```
//: C03:CommaOperator.cpp
#include <iostream>
using namespace std;
int main() {
  int a = 0, b = 1, c = 2, d = 3, e = 4;
  a = (b++, c++, d++, e++);
  cout << "a = " << a << endl;
  // The parentheses are critical here. Without
  // them, the statement will evaluate to:
  (a = b++), c++, d++, e++;
  cout << "a = " << a << endl;
} ///:~
```

通常，除了作为一个分隔符，逗号最好不作他用，因为人们不习惯把它看做是运算符。 165

3.7.10 使用运算符时的常见问题

如上所述，使用运算符时的一个问题是总不愿使用括号，即使在还不确定一个表达式如何计算时（可以查阅当前的C手册中表达式的计算顺序）。

另一个十分常见的错误如下所示：

```
//: C03:Pitfall.cpp
// Operator mistakes

int main() {
  int a = 1, b = 1;
  while(a = b) {
    // ....
```

```
  }
} ///:~
```

当**b**不为零时，语句**a = b**总是为真。把**b**的值赋给**a**，而**b**的值也是由运算符“=”产生的。一般在条件语句中，应当使用等值运算符“==”，而不是赋值。这是许多程序员经常犯的错误(但是，一些编译器会指出这个问题，这是有帮助的)。

一个相似的问题是使用位运算符中的“与”和“或”，而不是和它们相对应的逻辑运算符。位运算符中的“与”和“或”使用一个字符(**&** 或 **|**)，而逻辑“与”和“或”使用两个运算符(**&&**和**||**)。就像=和==一样，很容易会用一个字符替代两个字符。可以使用一种帮助记忆的方式“位比较小，所以在它们的运算符中不需要使用很多字符”。

3.7.11 转换运算符

转换（*cast*）这个词通常意为“浇铸成一个模型”。如果编译器能够明白的话，它会自动把一种数据类型转换为另一种类型。例如，如果赋一个整型值给一个浮点变量，编译器会暗地里调用一个函数（或更可能插入代码）来把整型转换为浮点型。转换允许使用这种显式类
166 型变换，或在转换没有正常情况下发生时强制它实现。

为了实现转换，要用括号把所想要转换的数据类型（包括所有的修饰符）括起来放在值的左边。这个值可以是一个变量、一个常量、由一个表达式产生的值或是一个函数的返回值。下面是一个例子:

```
//: C03:SimpleCast.cpp
int main() {
  int b = 200;
  unsigned long a = (unsigned long int)b;
} ///:~
```

转换是很有用的，但是它也造成了令人头痛的事，因为在某些情况下，它强制编译器把一个数据看做是比它实际上更大的类型，所以它占用了更多的内存空间，这可能会破坏其他数据。这种情况经常不是出现在上述简单的类型转换时，而在转换指针时发生。

C++有一个另外的转换语法，它遵从函数调用的语法。这个语法给参数加上括号而不是给数据类型加上括号，类似于函数调用:

```
//: C03:FunctionCallCast.cpp
int main() {
  float a = float(200);
  // This is equivalent to:
  float b = (float)200;
} ///:~
```

当然对于上面的情况，我们实际上不需要转换，只要写**200f**（实际上，一般编译器会对上面的表达式作转换)。转换一般用于变量，而不用于常量。

3.7.12 C++的显式转换

应该小心使用转换，因为转换实际上要做的就是对编译器说“忘记类型检查，把它看做是其他类型。”这也就是说，在C++类型系统中引入了一个漏洞，并阻止编译器报告在类型
167 方面出错了。更为糟糕的是，编译器会相信它，而不执行任何其他的检查来捕获错误。一

旦开始进行转换，程序员必须自己面对各种问题。事实上，无论什么原因，任何一个程序如果使用很多转换都值得怀疑。一般情况下，很少使用转换，它只是用于解决非常特殊的问题。

一旦理解了这一点，在遇上一个出故障的程序时，第一个反应应该是寻找作为嫌犯的转换。但是怎样确定C风格转换的位置呢？它们只是在括号中的类型名字，如果开始查找这些的话，我们会发现很难把它们和代码的其他部分区分开来。

标准C++包括一个显式的转换语法，使用它来完全替代旧的C风格的转换（当然，如果不破坏代码，是不会认为C风格的转换不合法，但是编译器的编写者很容易标出旧风格的转换）。显式类型转换语法使我们很容易发现它们，因为通过它们的名字就能找到：

static_cast	用于“良性”和“适度良性”转换，包括不用强制转换（例如自动类型转换）
const_cast	对“const”和/或“volatile”进行转换
reinterpret_cast	转换为完全不同的意思。为了安全使用它，关键必须转换回原来的类型。转换成的类型一般只能用于位操作，否则就是为了其他隐秘的目的。这是所有转换中最危险的
dynamic_cast	用于类型安全的向下转换（这种转换将在第15章介绍）

168

在下面的小节会更详细地叙述前面三个显式转换，而最后一个要在读者有了更多的了解之后，在第15章中阐述。

3.7.12.1 静态转换（static_cast）

static_cast全部用于明确定义的变换，包括编译器允许我们所做的不用强制转换的“安全”变换和不太安全但清楚定义的变换。**static_cast**包含的转换类型包括典型的非强制变换、窄化（有信息丢失）变换，使用**void***的强制变换、隐式类型变换和类层次的静态定位（因为还没有看到类和继承，这个主题会推延到第15章讨论）：

```
//: C03:static_cast.cpp
void func(int) {}

int main() {
  int i = 0x7fff; // Max pos value = 32767
  long l;
  float f;
  // (1) Typical castless conversions:
  l = i;
  f = i;
  // Also works:
  l = static_cast<long>(i);
  f = static_cast<float>(i);

  // (2) Narrowing conversions:
  i = l; // May lose digits
  i = f; // May lose info
  // Says "I know," eliminates warnings:
  i = static_cast<int>(l);
  i = static_cast<int>(f);
  char c = static_cast<char>(i);

  // (3) Forcing a conversion from void* :
  void* vp = &i;
  // Old way produces a dangerous conversion:
```

```
  float* fp = (float*)vp;
  // The new way is equally dangerous:
169 fp = static_cast<float*>(vp);
  // (4) Implicit type conversions, normally
  // performed by the compiler:
  double d = 0.0;
  int x = d; // Automatic type conversion
  x = static_cast<int>(d); // More explicit
  func(d); // Automatic type conversion
  func(static_cast<int>(d)); // More explicit
} ///:~
```

程序的第(1)部分，是C中习惯采用的几种变换，有的有强制转换，有的没有强制转换。把**int** 提升到**long**或**float**不会有问题，因为后者总是能容纳一个**int**所包含的值。尽管这是不必要的，但是可以使用**static_cast**来突出这些提升。

第(2)部分显示的是另一种变换方式。在这里可能会丢失数据，因为一个**int**和**long**或 **float**不是一样“宽”的；它不能容纳同样大小的数字。因此称为*窄化变换*（*narrowing conversion*）。编译器仍能执行这种转换，但是会经常给出一个警告。我们可以消除这种警告，表明我们真的想使用转换来实现它。

正如在第(3)部分看到的，C++中不用转换是不允许从**void***中赋值的（不像C）。这是很危险的，要求程序员知道他们正在做什么。至少，当查找故障的时候，**static_cast**比旧标准的转换更容易定位。

程序的第(4)部分显示编译器自动执行的几种隐式类型变换。这些变换是自动的，不需要强制转换，但是当我们要想清楚发生了什么或以后要查找转换，可以再次使用**static_cast**突出这个行为。

3.7.12.2 常量转换（const_cast）

如果从**const**转换为非**const**或从**volatile**转换为非**volatile**，可以使用**const_cast**。这是**const_cast**惟一允许的转换；如果进行别的转换就可能要使用单独的表达式或者可能会得到一个编译错误。

```
//: C03:const_cast.cpp
170 int main() {
  const int i = 0;
  int* j = (int*)&i; // Deprecated form
  j  = const_cast<int*>(&i); // Preferred
  // Can't do simultaneous additional casting:
//! long* l = const_cast<long*>(&i); // Error
  volatile int k = 0;
  int* u = const_cast<int*>(&k);
} ///:~
```

如果取得了**const**对象的地址，就可以生成一个指向**const**的指针，不用转换是不能将它赋给非**const**指针的。旧形式的转换能实现这样的赋值，但是**const_cast**是适用的。**volatile**也是这样。

3.7.12.3 重解释转换（reinterpret_cast）

这是最不安全的一种转换机制，最有可能出问题。**reinterpret_cast**把对象假想为模式（为了某种隐秘的目的），仿佛它是一个完全不同类型的对象。这是低级的位操作，C因此而名

声不佳。在使用**reinterpret_cast**做任何事之前，实际上总是需要**reinterpret_cast**回到原来的类型（或者把变量看做是它原来的类型）。

```
//: C03:reinterpret_cast.cpp
#include <iostream>
using namespace std;
const int sz = 100;

struct X { int a[sz]; };

void print(X* x) {
  for(int i = 0; i < sz; i++)
    cout << x->a[i] << ' ';
  cout << endl << "--------------------" << endl;
}

int main() {
  X x;
  print(&x);
  int* xp = reinterpret_cast<int*>(&x);
  for(int* i = xp; i < xp + sz; i++)
    *i = 0;
  // Can't use xp as an X* at this point
  // unless you cast it back:
  print(reinterpret_cast<X*>(xp));
  // In this example, you can also just use
  // the original identifier:
  print(&x);
} ///:~
```

171

在这个简单的例子中，**struct X**只包含一个整型数组，但是当用**X x**在堆栈中创建一个变量时，该结构体中的每一个整型变量的值都没有意义（通过使用函数**print( )**把结构体的每一个整型值显示出来可以表明这一点）。为了初始化它们，取得**X**的地址并转换为一个整型指针，该指针然后遍历这个数组置每一个整型元素为0。注意**i**的上限是如何通过计算**sz**加**xp**得到的。编译器知道我们实际上是希望**sz**的指针位置比**xp**更大，它替我们做了正确的指针算术运算。

reinterpret_cast的思想就是当需要使用的时候，所得到的东西已经不同了，以至于它不能用于类型的原来目的，除非再次把它转换回来。这里，我们在打印调用中转换回**X***，但是当然，因为我们还有原来的标识符，所以还可以使用它。但是**xp**只有作为**int***才有用，这真的是对原来的**X**的重新解释。

使用**reinterpret_cast**通常是一种不明智、不方便的编程方式，但是当必须使用它时，它是非常有用的。

3.7.13 sizeof——独立运算符

sizeof单独作为一个运算符是因为它满足不同寻常的需要。**sizeof**给我们提供对有关数据项目所分配的内存大小。正如在本章前面叙述的那样，**sizeof**告诉我们任何变量使用的字节数。它也可以给出数据类型的大小（不用变量名）。

```
//: C03:sizeof.cpp
#include <iostream>
using namespace std;
```

172
```
int main() {
  cout << "sizeof(double) = " << sizeof(double);
  cout << ", sizeof(char) = " << sizeof(char);
} ///:~
```

按照定义，任何**char**（**signed**、**unsigned**或普通的）类型的**sizeof**都是1，不管**char**潜在的存储空间是否实际上是一个字节。对于所有别的类型，结果都是以字节表示的大小。

注意**sizeof**是一个运算符，不是函数。如果把它应用于一个类型，必须要像上面所示的那样使用括号，但是如果对一个变量使用它，可以不要括号。

```
//: C03:sizeofOperator.cpp
int main() {
  int x;
  int i = sizeof x;
} ///:~
```

sizeof也可以给出用户定义的数据类型的大小。这在本书后面会介绍。

3.7.14 asm关键字

这是一种转义（escape）机制，允许在C++程序中写汇编代码。在汇编程序代码中经常可以引用C++的变量，这意味着可以方便地和C++代码通信，且限定汇编代码只是用于必要的高效调整，或使用特殊的处理器指令。编写汇编语言时所必须使用的严格语法是依赖于编译器的，在编译器的文档中可以发现有关语法。

3.7.15 显式运算符

这是用于位运算符和逻辑运算符的关键字。没有&、|、^这些键盘字符的非美国程序员被迫使用C的令人讨厌的三个*图形字符(trigraph)*，这使得不但在输入字符的时候令人烦恼，而
173 且在阅读时也含义模糊。在C++中用附加的关键字来修补这种情况。

关 键 字	含 义
and	**&&** (逻辑与)
or	**||** (逻辑或)
not	**!** (逻辑非)
not_eq	**!=** (逻辑不等)
bitand	**&** (位与)
and_eq	**&=** (位与-赋值)
bitor	**|** (位或)
or_eq	**|=** (位或-赋值)
xor	**^** (位异或)
xor_eq	**^=** (位异或-赋值)
compl	**~** (补)

如果读者的编译器遵从标准C++，它会支持这些关键字。

3.8 创建复合类型

基本的数据类型及其变体很重要，但也很简单。C和C++提供的工具允许把基本的数据类型组合成复杂的数据类型。正如我们将看到的那样，这些类型中最重要的是**struct**，在C++中

这是类的基础。但是，创建比较复杂的类型的最简单的一种方式，只需要通过**typedef**来命名一个名字为另一个名字。

3.8.1 用typedef命名别名

这个关键字从字面上看的作用比它实际所起的作用更大：**typedef**表示“类型定义”，但用“别名”来描述可能更精确，因为这正是它真正的作用。它的语法是：

174

typedef *原类型名 别名*

当数据类型稍微有点复杂时，人们经常使用**typedef**只是为了少敲几个键。下面是一种经常使用的**typedef**：

```
typedef unsigned long ulong;
```

现在如果写**ulong**，则编译器知道意思是**unsigned long**。我们可能认为使用预处理程序置换就可以很容易实现，但是在一些重要的场合，编译器必须知道我们正在将名字当做类型处理，所以**typedef**起了关键作用。

typedef经常会派上用场的地方是指针类型。如前所述，如果写出

```
int* x, y;
```

这实际上生成一个**int*x**和一个**int*y**（不是一个**int***）。也就是说，‘*’绑定右边，而不是左边。但是，如果使用一个**typedef**：

```
typedef int* IntPtr;
IntPtr x, y;
```

则**x**和**y**都是**int***类型。

有人可能争辩说避免使用**typedef**定义基本类型会更清楚，因此更可读，而使用大量**typedef**时，程序的确很快变得难以阅读。但是，在C中使用**struct**时，**typedef**是特别重要的。

3.8.2 用struct把变量结合在一起

struct（结构）是把一组变量组合成一个构造的一种方式。一旦创建了一个**struct**，就可以生成所建立的新类型变量的许多实例。例如：

```
//: C03:SimpleStruct.cpp
struct Structure1 {
  char c;
  int i;
  float f;
  double d;
};

int main() {
  struct Structure1 s1, s2;
  s1.c = 'a'; // Select an element using a '.'
  s1.i = 1;
  s1.f = 3.14;
  s1.d = 0.00093;
  s2.c = 'a';
  s2.i = 1;
  s2.f = 3.14;
```

175

```
  s2.d = 0.00093;
} ///:~
```

struct的声明必须以分号结束。在**main( )**中，创建了两个**Structure1**的实例：**s1**和**s2**。它们每一个都有各自独立的**c**、**i**、**f**和**d**版本。所以**s1**和**s2**表示了完全独立的变量块。要在**s1**或**s2**中选择一个元素，应该使用一个‘.’，使用**C++ class**对象的语法就是前面看到的那样，因为**class**对象是由**struct**演化而来的，所以**struct**是语法的来源。

注意这是使用**Structure1**的不便之处（正如所指出的那样，只是在C中需要，而不是C++）。在C中，当定义变量时，不能只说**Structure1**，必须说**struct Structure1**。这就是在C中使用**typedef**特别方便的地方。

```
//: C03:SimpleStruct2.cpp
// Using typedef with struct
typedef struct {
  char c;
  int i;
  float f;
  double d;
} Structure2;

int main() {
  Structure2 s1, s2;
  s1.c = 'a';
176 s1.i = 1;
  s1.f = 3.14;
  s1.d = 0.00093;
  s2.c = 'a';
  s2.i = 1;
  s2.f = 3.14;
  s2.d = 0.00093;
} ///:~
```

当定义**s1**和**s2**时（但是注意它只有数据和特征，并不包括行为，这就是在C++中得到的真正的对象），通过这样使用**typedef**，可以假定**Structure2**是一个像**int**或**float**一样的内部类型（这是在C中；而在C++中，可以试图去掉**typedef**），我们将会看到，**struct**标识符已经脱离了原来的目的，因为这里的目的是创造**typedef**。当然，有时候可能需要早定义结构是使用**struct**。这时，可以重复**struct**的名字，就像**struct**名和**typedef**一样：

```
//: C03:SelfReferential.cpp
// Allowing a struct to refer to itself

typedef struct SelfReferential {
  int i;
  SelfReferential* sr; // Head spinning yet?
} SelfReferential;

int main() {
  SelfReferential sr1, sr2;
  sr1.sr = &sr2;
  sr2.sr = &sr1;
  sr1.i = 47;
  sr2.i = 1024;
} ///:~
```

如果看一下这个程序，会看到**sr1**和**sr2**互相指向且每个都拥有一块数据。

实际上，**struct**的名字不必和**typedef**的名字相同，但是，一般使用相同的名字，为了使得事物更加简单。 177

3.8.2.1 指针和struct

在上面的例子中，所有的**struct**都当做对象处理。但是，像任何一片存储空间一样，可以取得一个**struct**的地址（正如在上面的程序**SelfReferential.cpp**中看到的那样）。如上所述，为了选择一个特定**struct**对象中的元素，应当使用‘.’。但是，如果有一个指向**struct**对象的指针，可以使用一个不同的运算符‘->’来选择对象中的元素。下面是一个例子:

```
//: C03:SimpleStruct3.cpp
// Using pointers to structs
typedef struct Structure3 {
  char c;
  int i;
  float f;
  double d;
} Structure3;

int main() {
  Structure3 s1, s2;
  Structure3* sp = &s1;
  sp->c = 'a';
  sp->i = 1;
  sp->f = 3.14;
  sp->d = 0.00093;
  sp = &s2; // Point to a different struct object
  sp->c = 'a';
  sp->i = 1;
  sp->f = 3.14;
  sp->d = 0.00093;
} ///:~
```

在**main( )**中，**struct**指针**sp**最初指向**s1**，用‘->’选择**s1**中的成员来初始化它们。随后**sp**指向**s2**，以同样的方式初始化那些变量。所以可以看到指针的另一个好处是可以动态地重定向它们，指向不同的对象，使编程更灵活。 178

到现在为止，这就是对**struct**需要了解的全部，但是随着本书的进展，我们会更自如地使用它们（特别是它们更有潜力的继任者——类）。

3.8.3 用enum提高程序清晰度

枚举数据类型是把名字和数字相联系的一种方式，从而对阅读代码的任何人给出更多的含义。**enum**关键字（来自C）通过为所给出的任何标识符表赋值0、1、2等值来自动地列举出它们。也可以声明**enum**变量（它们总是表示为整数值）。**enum**的声明和**struct**的声明很相似。

当想明了某种特征时，枚举数据类型是很有用的:

```
//: C03:Enum.cpp
// Keeping track of shapes

enum ShapeType {
  circle,
  square,
```

```
  rectangle
};  // Must end with a semicolon like a struct

int main() {
  ShapeType shape = circle;
  // Activities here....
  // Now do something based on what the shape is:
  switch(shape) {
    case circle:  /* circle stuff */ break;
    case square:  /* square stuff */ break;
    case rectangle:  /* rectangle stuff */ break;
  }
} ///:~
```

shape是被列举的数据类型**ShapeType**的变量，可以把它的值和列举的值相比较。因为**shape**实际上只是**int**，所以它可以具有任何一个**int**拥有的值（包括负数）。也可以把**int**变量和
179 枚举值比较。

读者可能意识到上面的类型转换例子对于程序有可能是一种值得怀疑的方式。C++对这类程序有一种更好的编码方式，对它的解释在本书的后面介绍。

如果不喜欢编译器赋值的方式，可以自己做，如：

```
enum ShapeType {
  circle = 10, square = 20, rectangle = 50
};
```

如果对某些名字赋给值，对其他的不赋给值，编译器会使用相邻的下一个整数值。例如，

```
enum snap { crackle = 25, pop };
```

编译器会把值26赋给**pop**。

使用枚举数据类型时，增强了代码的可读性。然而，在某种程度上，这只是试图（在C中）实现在C++中用类可以做到的事，所以在C++中很少看到使用**enum**。

3.8.3.1 枚举类型检查

C的枚举相当简单，只是把整数值和名字联系起来，但它们并不提供类型检查。在C++中，正如现在希望的那样，类型的概念是基础，对于枚举也是如此。当创建一个命名的枚举时，就像使用类一样有效地创建了一个新类型。在单元翻译期间，枚举名成为保留字。

此外，在C++中对枚举的类型检查比在C中更为严格。如果有一个**color**枚举类型的实例**a**，
180 我们就会特别注意到这个。在C中，可以写**a++**，但在C++中不能这样写。这是因为枚举的增量运算执行两种类型转换，其中一个在C++中是合法的，另一个是不合法的。首先，枚举的值隐式地从**color**强制转换为**int**，然后递增该值，再把**int**强制转换回**color**类型。在C++中，这是不允许的，因为**color**是一个独特的类型，并不等价于一个**int**。这一点是有意义的，因为我们怎么能知道在颜色表中**blue**的增量值会是什么？如果想对**color**进行增量运算，则它应该是一个类（按照增量运算）而不是一个**enum**，成为一个类会更安全。任何时候写代码对**enum**类型进行隐式转换，编译器都会标记这是一个危险活动。

在C++中，联合（在下面描述）有很相似的附加类型检查。

3.8.4 用union节省内存

有时一个程序会使用同一个变量处理不同的数据类型。对于这种情况，有两种选择：可

以创建一个**struct**，其中包含需要存储的所有可能的不同类型，或者可以使用**union**（联合）。**union**把所有的数据放进一个单独的空间内，它计算出放在**union**中的最大项所必需的空间数，并生成**union**的大小。使用**union**可以节省内存。

每当在**union**中放置一个值，这个值总是放在**union**开始的同一个地方，但是只使用必需的空间。因此，我们创建的是一个能容纳任何一个**union**变量的“超变量”。所有的**union**变量地址都是一样的（在类或**struct**中，地址是不同的）。

下面是一个使用**union**的例子。试着去掉不同的元素，看看对**union**的大小有什么影响。注意在**union**中声明某个数据类型的多个实例是没有意义的（除非就是要用不同的名字）。

```
//: C03:Union.cpp
// The size and simple use of a union
#include <iostream>
using namespace std;

union Packed { // Declaration similar to a class
  char i;
  short j;
  int k;
  long l;
  float f;
  double d;
  // The union will be the size of a
  // double, since that's the largest element
};  // Semicolon ends a union, like a struct

int main() {
  cout << "sizeof(Packed) = "
       << sizeof(Packed) << endl;
  Packed x;
  x.i = 'c';
  cout << x.i << endl;
  x.d = 3.14159;
  cout << x.d << endl;
} ///:~
```

181

编译器根据所选择的联合的成员执行适当的赋值。

一旦进行赋值，编译器并不关心用联合做什么。在上面的例子中，可以对x赋一个浮点值：

```
x.f = 2.222;
```

然后把它作为一个**int**输出。

```
cout << x.i;
```

结果是无用的信息。

3.8.5 数组

数组是一种复合类型，因为它们允许在一个单一的标识符下把变量结合在一起，一个接着一个。如果写出

```
int a[10];
```

就为10个**int**变量创建了一个接一个的存储空间，但是每一个变量并没有单独的标识符。相反，

182 它们都集结在名字a下。

要访问一个数组元素，可以使用定义数组时所使用的方括号语法：

```
a[5] = 47;
```

不过，必须记住，尽管a的大小是10，但是要从零开始选择数组元素（有时这被称为零指针），所以只可以选择数组元素0~9，如下所示：

```
//: C03:Arrays.cpp
#include <iostream>
using namespace std;

int main() {
  int a[10];
  for(int i = 0; i < 10; i++) {
    a[i] = i * 10;
    cout << "a[" << i << "] = " << a[i] << endl;
  }
} ///:~
```

访问数组是很快的。但是，如果下标超出数组的界限，这就不安全了，这可能会访问到别的变量。另一个缺陷是必须在编译期定义数组的大小；如果想在运行期改变大小，则不能使用上面的语法（C有一种动态创建数组的方式，但是这会造成严重的混乱）。在前面一章中介绍的C++向量提供了类似数组的对象，它能自动调整自身的大小，所以如果数组的大小在编译期不能确定的话，这是比较好的解决方法。

可以生成任何类型的数组，甚至是**struct**类型的：

```
//: C03:StructArray.cpp
// An array of struct

typedef struct {
  int i, j, k;
} ThreeDpoint;

int main() {
183   ThreeDpoint p[10];
  for(int i = 0; i < 10; i++) {
    p[i].i = i + 1;
    p[i].j = i + 2;
    p[i].k = i + 3;
  }
} ///:~
```

注意：**struct**中的标识符**i**如何与**for**循环中的**i**无关。

为了知道数组中的相邻元素之间的距离，可以打印出地址如下：

```
//: C03:ArrayAddresses.cpp
#include <iostream>
using namespace std;

int main() {
  int a[10];
  cout << "sizeof(int) = "<< sizeof(int) << endl;
  for(int i = 0; i < 10; i++)
    cout << "&a[" << i << "] = "
```

```
        << (long)&a[i] << endl;
} ///:~
```

当运行程序时，会看到每一个元素和前一个元素都是相距**int**大小的距离。也就是说，它们是一个接一个存放的。

3.8.5.1 指针和数组

数组的标识符不像一般变量的标识符。一方面，数组标识符不是左值，不能给它赋值。它只是一个进入方括号语法的手段，当给出数组名而没有方括号时，得到的就是数组的起始地址：

```
//: C03:ArrayIdentifier.cpp
#include <iostream>
using namespace std;

int main() {
  int a[10];
  cout << "a = " << a << endl;
  cout << "&a[0] =" << &a[0] << endl;
} ///:~
```

184

运行这个程序时，会看到这两个地址（因为没有转换为**long**，所以它以十六进制的形式打印出来）是一样的。

因此可以把数组标识符看做是数组起始处的只读指针。尽管不能改变数组标识符指向，但是可以另创建指针，使它在数组中移动。事实上，方括号语法和指针一样工作：

```
//: C03:PointersAndBrackets.cpp
int main() {
  int a[10];
  int* ip = a;
  for(int i = 0; i < 10; i++)
    ip[i] = i * 10;
} ///:~
```

当想给一个函数传递数组时，命名数组以产生它的起始地址的事实相当重要。如果声明一个数组为函数参数，实际上真正声明的是一个指针。所以在下面的例子中，**func1( )**和**func2( )**有一样的参数表：

```
//: C03:ArrayArguments.cpp
#include <iostream>
#include <string>
using namespace std;

void func1(int a[], int size) {
  for(int i = 0; i < size; i++)
    a[i] = i * i - i;
}

void func2(int* a, int size) {
  for(int i = 0; i < size; i++)
    a[i] = i * i + i;
}

void print(int a[], string name, int size) {
  for(int i = 0; i < size; i++)
```

185

```
    cout << name << "[" << i << "] = "
         << a[i] << endl;
}

int main() {
  int a[5], b[5];
  // Probably garbage values:
  print(a, "a", 5);
  print(b, "b", 5);
  // Initialize the arrays:
  func1(a, 5);
  func1(b, 5);
  print(a, "a", 5);
  print(b, "b", 5);
  // Notice the arrays are always modified:
  func2(a, 5);
  func2(b, 5);
  print(a, "a", 5);
  print(b, "b", 5);
} ///:~
```

尽管**func1( )**和**func2( )**以不同的方式声明它们的参数[⊖]，但是在函数内部的用法是一样的。这个例子暴露出了一些别的问题：数组不可以按值传递，也就是说，不会自动地得到传递给函数的数组的本地拷贝。因此，修改数组时，一直是在修改外部对象。如果想按照一般的参数那样提供按值传递，可能一开始会让人有点迷惑。

读者会注意到，**print( )**对数组参数使用方括号语法。尽管把数组作为参数传递时，指针语法和方括号语法是一样的，但是方括号语法使得读者更清楚它的意思是把这个参数看做是一个数组。

186

还要注意，在每一种情况传递了参数**size**。仅仅传递数组的地址还不能提供足够的信息，必须知道在函数中的数组有多大，这样就不会超出数组的界。

数组可以是任何一种类型，包括指针数组。事实上，想给程序传递命令行参数时，C和C++的函数**main( )**有特殊的参数表，其形式如像：

```
int main(int argc, char* argv[]) { // ...
```

第一个参数的值是第二个参数的数组元素个数。第二个参数总是**char***数组，因为数组中的元素来自作为字符数组的命令行（记住，数组只能作为指针传递）。命令行中的每一个用空格分隔的字符串被转换成单独的数组参数。通过遍历数组，下面的程序可以打印出所有的命令行参数：

```
//: C03:CommandLineArgs.cpp
#include <iostream>
using namespace std;

int main(int argc, char* argv[]) {
  cout << "argc = " << argc << endl;
```

⊖ 除非采取了严格的办法："在C/C++中的所有参数是通过值传递的，数组的'值'是由数组标识符产生的：它是一个地址。"从汇编语言的观点来看这可能是真的，但是当用更高层的概念工作时，我认为这是没有帮助的。在C++中附加的引用生成"所有的传递都是通过值"的说法更会使人混淆，对于这一点我认为按照与"以地址传递"相对的"以值传递"来思考更好。

```
  for(int i = 0; i < argc; i++)
    cout << "argv[" << i << "] = "
         << argv[i] << endl;
} ///:~
```

读者会注意到**argv[0]**是程序本身的路径和名字。它允许程序发现自己的信息。它也给程序参数数组增加一个或多个参数，所以一个常见的错误就是当想获取命令行参数**argv[1]**的值时，却去取**argv[0]**的值。

在函数**main( )**中，不要强制使用**argc**和**argv**为标识符；这些标识符只是习惯用法（如果不使用它们，可能会让别人迷惑）。还有另一种声明**argv**的方式：

187

```
int main(int argc, char** argv) { // ...
```

两种形式是等价的，但本书使用的版本更为直观，因为它直接表明“这是一个字符指针数组”。

从命令行中获得的是字符数组；如果想把数组看成是别的某种类型，应该在程序里负责转换它。为了便于转换为数值，在标准C库的**<cstdlib>**中声明了一些更有帮助的函数。最简单的是分别使用**atoi( )**、**atol( )**和**atof( )**把ASCII字符数组转换为**int**、**long**和**double**浮点值。下面是一个使用**atoi( )**的例子（另两个函数用同样的方式调用）：

```
//: C03:ArgsToInts.cpp
// Converting command-line arguments to ints
#include <iostream>
#include <cstdlib>
using namespace std;

int main(int argc, char* argv[]) {
  for(int i = 1; i < argc; i++)
    cout << atoi(argv[i]) << endl;
} ///:~
```

在这个程序中，可以在命令行中放置任意多个参数。读者会注意到**for**循环从值**1**开始，跳过了**argv[0]**中的程序名。如果在命令行上放置了一个包含小数点的浮点数，**atoi( )**只取得小数点前面的数字部分。如果在命令行中没有数值，**atoi( )**会返回零值。

3.8.5.2 探究浮点格式

本章已经介绍的**printBinary( )**函数对于研究不同数据类型的内部结构是很合适的。最令人感兴趣的就是浮点格式，它允许C和C++在有限的空间里存储非常大和非常小的数。尽管在这里不能完全显示其细节，但是在**float**和**double**里的数字位被分为段：指数、尾数和符号位，它用科学计数法来存储数值。下面的程序允许打印出不同浮点数的二进制形式，所以读者可以自己推断出编译器浮点格式的使用方案（一般这是浮点数的IEEE标准，但是有的编译器可能不遵守）。

188

```
//: C03:FloatingAsBinary.cpp
//{L} printBinary
//{T} 3.14159
#include "printBinary.h"
#include <cstdlib>
#include <iostream>
using namespace std;
```

```
int main(int argc, char* argv[]) {
  if(argc != 2) {
    cout << "Must provide a number" << endl;
    exit(1);
  }
  double d = atof(argv[1]);
  unsigned char* cp =
    reinterpret_cast<unsigned char*>(&d);
  for(int i = sizeof(double); i > 0 ; i -= 2) {
    printBinary(cp[i-1]);
    printBinary(cp[i]);
  }
} ///:~
```

首先，程序通过检查**argc**的值保证给定了参数，如果有一个参数，则**argc**的值应该为2（如果没有参数，则为1，因为程序名总是**argv**的第一个元素）。如果程序失败了，会打印出一个消息并调用标准C的库函数**exit( )**来终止程序。

189 程序从命令行中取得参数并使用函数**atof( )**把字符转换成**double**浮点数。然后通过取得地址并把该数转换为一个**unsigned char***指针作为一个字节数组。把其中的每一个字节传递给**printBinary( )**显示出来。

我在自己的机器上通过了这个程序，打印字节时符号位出现在前面。有的机器可能和我的不一样，所以可能需要重新安排打印的方式。读者应认识到理解浮点格式并不是微不足道的。例如，一般不把指数和尾数以字节划分的边界存放，而是为每一部分保留若干位数，并把它们尽可能紧密地压缩进内存。要真的看看发生了什么，应该把数值的每一部分的大小找出来（符号位总是一位，而指数和尾数的位数的大小不同），并把每一部分的位数分别打印出来。

3.8.5.3 指针算术

如果用指针所做的工作只是把它看做是数组的一个别名，那么指向数组的指针可能不太令人感兴趣。但是，指针比这个更灵活，因为可以修改它们指向任何别的地方（但是记住，不能修改数组标识符来指向别的地方）。

指针算术（*pointer arithmetic*）指的是对指针的某些算术运算符的应用。指针算术是一个源自普通算术的单独主题，其原因在于为了正确运行，指针必须遵守特定的约束。例如，指针常用的运算符是**++**——“给指针加1”。它的实际意义是改变指针移向“下一个值”。下面是一个例子：

```
//: C03:PointerIncrement.cpp
#include <iostream>
using namespace std;

int main() {
  int i[10];
  double d[10];
  int* ip = i;
  double* dp = d;
190 cout << "ip = " << (long)ip << endl;
  ip++;
  cout << "ip = " << (long)ip << endl;
  cout << "dp = " << (long)dp << endl;
  dp++;
  cout << "dp = " << (long)dp << endl;
} ///:~
```

我的机器上的运行输出是:

```
ip = 6684124
ip = 6684128
dp = 6684044
dp = 6684052
```

这里令人感兴趣的是尽管对**int***和**double***进行的都是同样的操作“++”，但是对**int***只改变了4个字节，而对**double***改变了8个字节。当然并非总是这样，这取决于**int**和**double**浮点数的大小。这就是指针算术的技巧：编译器计算出指针改变的正确值，使它指向数组中的下一个元素（指针算术只有在数组中才是有意义的）。甚至在**struct**数组中也能这样工作:

```
//: C03:PointerIncrement2.cpp
#include <iostream>
using namespace std;

typedef struct {
  char c;
  short s;
  int i;
  long l;
  float f;
  double d;
  long double ld;
} Primitives;

int main() {
  Primitives p[10];
  Primitives* pp = p;
  cout << "sizeof(Primitives) = "
       << sizeof(Primitives) << endl;
  cout << "pp = " << (long)pp << endl;
  pp++;
  cout << "pp = " << (long)pp << endl;
} ///:~
```

191

我的机器上的运行结果是:

```
sizeof(Primitives) = 40
pp = 6683764
pp = 6683804
```

所以可以看到编译器对于**struct** (以及**class**和**union**)指针也能正确地工作。

指针算术运算也可以使用运算符“--”、“+”和“-”，但是后面两个运算符的使用是有限制的：不能把两个指针相加，如果使指针相减，其结果是两个指针之间相隔的元素个数。不过，一个指针可以加上或减去一个整数。下面是一个说明指针算术运算用法的例子:

```
//: C03:PointerArithmetic.cpp
#include <iostream>
using namespace std;

#define P(EX) cout << #EX << ": " << EX << endl;

int main() {
  int a[10];
```

```
  for(int i = 0; i < 10; i++)
    a[i] = i; // Give it index values
  int* ip = a;
  P(*ip);
  P(*++ip);
  P(*(ip + 5));
  int* ip2 = ip + 5;
  P(*ip2);
  P(*(ip2 - 4));
  P(*--ip2);
  P(ip2 - ip); // Yields number of elements
} ///:~
```

这个程序以另一个宏开始，但是它使用了一个被称为字符串化的预处理器特征（在表达
192 式前用一个‘#’实现），其作用是获得任何一个表达式并把它转换成为一个字符数组。这是很方便的，因为它允许打印一个表达式，后面接一个冒号，再接一个表达式的值。在**main( )**中，可以看到这产生了一个有用的简化。

尽管++和--的前缀和后缀方式对指针来说都是有效的，但是在这个例子中只使用了前缀方式，因为在上面的表达式中指针间接引用之前先应用它们，所以它们允许看到运算的效果。注意只能加上和减去整数值，如果两个指针以这种方式结合，编译器是不允许的。

上面程序的输出是：

```
*ip: 0
*++ip: 1
*(ip + 5): 6
*ip2: 6
*(ip2 - 4): 2
*--ip2: 5
```

在各种情况下，指针算术根据所指元素的大小调整指针，使其指向“正确的地方”。

如果一开始指针算术运算看起来有点令人困扰，那么不必担心。大多数情况下只需要创建数组和用[ ]表示的数组下标，一般所需要的最为复杂的指针算术运算是++和--。指针运算一般都用于更为灵活和复杂的程序中，标准C++库中许多容器隐藏了大多数的灵活细节，所以不必担心这一点。

3.9 调试技巧

在理想环境下，因为有优秀的调试器能很容易使得程序的运行行为透明，所以可以很快
193 发现错误。但是，大多数的调试器都有盲点，这就需要在程序中插入小段代码来帮助理解发生了什么问题。此外，可能在没有调试器（例如一个嵌入式系统）或者可能只有少量的反馈（如一个单行的LED显示屏）的环境下进行开发。在这些情况下，就要用创造性的方法去发现和显示关于程序执行情况的信息。下一节对程序调试的技巧提出某些建议。

3.9.1 调试标记

如果在程序中加入调试代码，可能引起不便。一开始得到了太多的信息，这使得很难把故障孤立出来。当认为已经找到了故障时，我们开始删掉调试代码，却有可能发现再需要这些代码。我们可以用两种标记解决这类问题：预处理器调试标记和运行期调试标记。

3.9.1.1 预处理器调试标记

通过使用预处理器**#define**定义一个或更多的调试标记（在头文件中更适合），可以测试一个使用**#ifdef**语句和包含条件调试代码的标记。当认为调试完成了，只需使用**#undef**标记，代码就会自动消失（这会减少可执行文件的大小和运行时间）。

最好在开始建立工程前决定调试标记的名字，这样名字会一致。为了区分预处理器标记和变量，预处理器标记一般用大写字母书写。一个常用的标记名是**DEBUG**（但是小心，不能使用**NDEBUG**，它是C中的保留字）。语句序列可以是：

```
#define DEBUG // Probably in a header file
//...
#ifdef DEBUG // Check to see if flag is defined
/* debugging code here */
#endif // DEBUG
```

大多数C和C++的程序实现还允许在编译器的命令行中使用**#define**和**#undef**标记，所以可以用一个单独的命令重新编译代码并插入调试信息（最好使用makefile，这是后面要简要说明 194
的工具）。具体细节请看局部的文档。

3.9.1.2 运行期调试标记

在某些情况下，在程序执行期间打开和关闭调试标记会更加方便，特别是使用命令行在启动程序时设置它们。只是为了插入调试代码来重新编译一个大程序是很乏味的。

为了自动打开和关闭调试代码，可以建立一个如下的**bool**标记：

```
//: C03:DynamicDebugFlags.cpp
#include <iostream>
#include <string>
using namespace std;
// Debug flags aren't necessarily global:
bool debug = false;

int main(int argc, char* argv[]) {
  for(int i = 0; i < argc; i++)
    if(string(argv[i]) == "--debug=on")
      debug = true;
  bool go = true;
  while(go) {
    if(debug) {
      // Debugging code here
      cout << "Debugger is now on!" << endl;
    } else {
      cout << "Debugger is now off." << endl;
    }
    cout << "Turn debugger [on/off/quit]: ";
    string reply;
    cin >> reply;
    if(reply == "on") debug = true; // Turn it on
    if(reply == "off") debug = false; // Off
    if(reply == "quit") break; // Out of 'while'
  }
} ///:~
```

这个程序一直允许打开和关闭调试标记，直到输入“quit”告诉它想要退出。注意需要输入整个单词，而不仅仅是字母（如果想要的话，可以缩写它为字母）。在启动时，可以选择性 195

地使用命令行参数打开调试——这个参数可以出现在命令行的任意地方，因为**main( )**中的启动代码能看得到所有的参数。测试是相当简单的，因为表达式为：

```
string(argv[i])
```

取得**argv[i]**字符数组并创建一个**string**使得它容易和==右端比较。上面的程序查找整个字符串**--debug=on**。也可以寻找**--debug=**，然后看它后面有什么，以提供更多的选择。本书的第2卷（可从*www.BruceEckel.com*中获得）有专门的一章讲述标准C++ **string**类。

虽然调试标记是很少的几个领域之一，其中对于使用全局变量很有意义，但是，并不是说必须这样做。注意使用小写字母书写变量，用来提醒读者它不是一个预处理器标记。

3.9.2 把变量和表达式转换成字符串

写调试代码的时候，编写由包含变量名和后跟变量的字符数组组成的打印表达式是很乏味的。幸运的是，标准C具有字符串化运算符‘**#**’，它在本章前面使用过的。在一个预处理器宏中的参数前面使用一个**#**，预处理器会把这个参数转换为一个字符数组。把这一点与没有插入标点符号的若干个字符数组结合而连接成一个单独的字符数组，能够生成一个十分方便的宏用于调试期间打印出变量的值：

```
#define PR(x) cout << #x " = " << x << "\n";
```

如果调用宏**PR(a)**来打印变量**a**的值，它和下面的代码有同样的效果：

```
cout << "a = " << a << "\n";
```

196 整个表达式工作过程一样。下面的程序使用一个宏创建了一种速记方式打印出字符串化的表达式，然后计算表达式并打印出结果：

```
//: C03:StringizingExpressions.cpp
#include <iostream>
using namespace std;

#define P(A) cout << #A << ": " << (A) << endl;

int main() {
  int a = 1, b = 2, c = 3;
  P(a); P(b); P(c);
  P(a + b);
  P((c - a)/b);
} ///:~
```

可以看到像这样的技术是如何成为必不可少的，特别是在没有调试器（或者必须使用多个开发环境）的情况下。当不想调试时，也可以插入一个**#ifdef**使得定义的**P(A)**不起作用。

3.9.3 C语言assert( )宏

在标准头文件**<cassert>**中，会发现**assert( )**是一个方便的调试宏。当使用**assert( )**时，给它一个参数，即一个表示断言为真的表达式。预处理器产生测试该断言的代码。如果断言不为真，则在发出一个错误信息告诉断言是什么以及它失败之后，程序会终止。下面是一个例子：

```
//: C03:Assert.cpp
// Use of the assert() debugging macro
```

```
#include <cassert>  // Contains the macro
using namespace std;

int main() {
  int i = 100;
  assert(i != 100); // Fails
} ///:~
```

197

这个宏来源于标准C，所以在头文件**assert.h**中也可以使用。

当完成调试后，通过在程序的**#include<cassert>**之前插入语句行

```
#define NDEBUG
```

或者在编译器命令行中定义**ndebug**，可以消除宏产生的代码。在**<cassert>**中使用的**ndebug**是一个标记，用来改变宏产生代码的方式。

在本书后面，会看到对于**assert( )**有一些更复杂的可供选择的方式。

3.10 函数地址

一旦函数被编译并载入计算机中执行，它就会占用一块内存。这块内存有一个地址，因此函数也有地址。

可以通过指针使用函数地址，就像可以使用变量的地址一样。函数指针的声明和使用初看起来有点模糊，但是它同语言其余部分的格式一致。

3.10.1 定义函数指针

要定义一个指针指向一个无参无返回值的函数，可以写成：

```
void (*funcPtr)();
```

当看到像这样的一个复杂定义时，最好的处理方法是从中间开始和向外扩展。“从中间开始”的意思是从变量名开始，这里是指**funcPtr**。“向外扩展”的意思是先注意右边最近的项（在这个例子中没有该项，以右括号结束），然后注意左边（用星号表示的指针），注意右边（空参数表表示这个函数没有带任何参数），再注意左边（**void**指示函数没有返回值）。大多数声明都是以左－右－左动作的方式工作的。 198

回过头来看，“中间开始”（“**funcPtr**是一个...”），向右边走（没有东西，被右括号拦住了），向左边走并发现一个‘*’（“...指针指向一个...”），向右边走并发现一个空参数表（“...没有带参数的函数...”），向左边走并发现一个**void**（“**funcPtr**是一个指针，它指向一个不带参数并返回**void**的函数”）。

读者可能感到奇怪为什么***funcPtr**需要括号。如果不使用括号，编译器会看到：

```
void *funcPtr();
```

这可能是在声明一个函数（返回一个**void***）而不是定义一个变量。在了解一个声明和定义应该是什么的时候，可以想像编译器要经历同样的过程。所以要“遇到”这些括号，使得编译器会返回左边并发现‘*’，而不是一直向右发现一个空参数表。

3.10.2 复杂的声明和定义

另一方面，一旦知道C和C++声明语法是如何工作的，就能够创建许多复杂的条目。例如：

```
//: C03:ComplicatedDefinitions.cpp

/* 1. */      void * (*(*fp1)(int))[10];

/* 2. */      float (*(*fp2)(int,int,float))(int);

199 /* 3. */   typedef double (*(*(*fp3)())[10])();
              fp3 a;

/* 4. */      int (*(*f4())[10])();
int main() {} ///:~
```

对于每一条，使用先右后左的原则去推断。

第1行说明："**fp1**是一个指向函数的指针，该函数接受一个整型参数并返回一个指向含有10个**void**指针数组的指针。"

第2行说明："**fp2**是一个指向函数的指针，该函数接受三个参数（**int**、**int**和**float**）且返回一个指向函数的指针，该函数接受一个整型参数并返回一个**flaot**。"

如果创建许多复杂的定义，可以使用**typedef**。第3行显示了每次**typedef**是如何缩短复杂定义的。它说明："**fp3**是一个指向函数的指针，该函数无参数，且返回一个指向含有10个指向函数指针数组的指针，这些函数不接受参数且返回double值。"然后它又说明："**a**是**fp3**类型中的一个。"**typedef**在用简单描述构建复杂描述时通常是很有用的。

第4行不是变量定义而是一个函数定义。它说明："**f4**是一个返回指针的函数，该指针指向含有10个函数指针的数组，这些函数返回整型值。"

我们可能很少甚至是从未使用过如此复杂的声明和定义。但如果通过练习能把它搞清楚的话，就不会被在现实生活中可能遇到的稍微复杂的情况所困惑。

3.10.3 使用函数指针

一旦定义了一个函数指针，在使用前必须给它赋一个函数的地址。就像一个数组**arr[10]**的地址是由不带方括号的这个数组的名字(**arr**)产生的一样，函数**func( )**的地址也是由没有参数列表的函数名（**func**）产生的。也可以使用更加明显的语法**&func( )**。为了调用这个函数，
200 应当用与声明相同的方法间接引用指针。（记住，C和C++总是力图让引用看上去与使用它们的方法一样。）下面的例子表明如何定义和使用指向函数的指针：

```
//: C03:PointerToFunction.cpp
// Defining and using a pointer to a function
#include <iostream>
using namespace std;

void func() {
  cout << "func() called..." << endl;
}

int main() {
  void (*fp)();  // Define a function pointer
  fp = func;  // Initialize it
  (*fp)();    // Dereferencing calls the function
  void (*fp2)() = func;  // Define and initialize
```

```
  (*fp2)();
} ///:~
```

在定义了指向函数的指针**fp**之后，用**fp = func**使**fp**获得函数**func( )**的地址（注意在函数名后缺少了参数列表）。第二种情况显示了同时定义和初始化。

3.10.4 指向函数的指针数组

我们能够创建的一个更为有趣的结构是指向函数的指针数组。为了选择一个函数，只需要使用数组的下标，然后间接引用这个指针。这种方式支持*表格式驱动码*（*table-driven code*）的概念；可以根据状态变量（或者状态变量的组合值）去选择被执行函数，而不用条件语句或case语句。这种设计方式对于经常要从表中添加或删除函数（或者想动态地创建或改变表）十分有用。

下面的例子使用预处理宏创建了一些哑函数，然后使用自动集合初始化功能创建指向这些函数的指针数组。正如看到的那样，很容易从表中添加或删除函数（这样，这个程序就具有了函数功能）而只需改变少量的代码：

201

```
//: C03:FunctionTable.cpp
// Using an array of pointers to functions
#include <iostream>
using namespace std;

// A macro to define dummy functions:
#define DF(N) void N() { \
   cout << "function " #N " called..." << endl; }

DF(a); DF(b); DF(c); DF(d); DF(e); DF(f); DF(g);

void (*func_table[])() = { a, b, c, d, e, f, g };

int main() {
  while(1) {
    cout << "press a key from 'a' to 'g' "
      "or q to quit" << endl;
    char c, cr;
    cin.get(c); cin.get(cr); // second one for CR
    if ( c == 'q' )
      break; // ... out of while(1)
    if ( c < 'a' || c > 'g' )
      continue;
    (*func_table[c - 'a'])();
  }
} ///:~
```

当希望创建一些解释器或表处理程序时，可以想像这种技术是多么有用。

3.11 make: 管理分段编译

当使用*分段编译*（*separate compilation*）（把代码拆分为许多翻译单元）时，需要某种方法去自动编译每个文件并且告诉连接器把所有分散的代码段，连同适当的库和启动代码，构造成一个可执行的文件。许多编译器允许用一个简单的命令行语句完成。例如，对于GNU C++编译器，可能会用：

202
```
g++ SourceFile1.cpp SourceFile2.cpp
```

使用这种方法的问题是编译器事先要编译每个文件而不管文件是否需要重建。在具有多个文件的工程中，如果仅仅改变了一个文件，就可能阻止重新编译所有文件。

解决问题的方法是用一个称为**make**的程序。该程序是在UNIX上开发的，但某些形式到处都可使用。**make**工具按照一个名为**makefile**的文本文件中的指令去管理一个工程中的所有单个文件。当编辑了工程中的某些文件并使用**make**时，**make**程序会按照**makefile**中的说明去比较源代码文件与相应目标文件的日期，如果源代码文件的日期比它的目标文件的日期新，**make**会调用编译器对源代码进行编译。**make**仅仅编译已经改变了的源代码，以及其他受修改文件影响的源代码文件。使用**make**程序,每次修改程序时，不必重新编译工程中的所有文件，也不必核对所有生成的东西。**makefile**文件包含了组合工程的所有命令。学会使用**make**命令会节省大量时间，也会减少挫折。在Linux/Unix机器上安装新软件时使用**make**是一种典型的方式(虽然那些**makefile**比本书上出现的要复杂得多，而且作为安装过程的一部分，对于特定的机器，通常会自动地生成**makefile**文件)。

因为**make**实际上对所有C++编译器有某种可用的形式（即使没有，也可以在任何编译器上使用免费的**make**）,因此它将作为贯穿于本书的工具。然而，编译器提供商也创建了自己的工程构造工具。这些工具询问工程中包括哪些文件，然后它们确定所有的关系。这些工具使用与**makefile**相似的文件，通常称为*工程文件*（*project file*），程序环境会维护该文件因此不
203 必为它而担心。配置和使用工程文件随开发环境的改变而有所不同，因此必须找到怎样使用它们的相关文档（虽然工程文件工具由不同的厂商提供，但是使用都很简单）。

即使还使用特定厂商的构建工程工具，本书中所用的**makefile**仍然有效。

3.11.1 make的行为

当输入**make**(或你的“make”程序的其他名字)时，**make**程序在当前目录中寻找名为**makefile**的文件，该文件作为工程文件已经被建立。这个文件列出了源代码文件间的依赖关系。**make**程序观察文件的日期。如果一个依赖文件的日期比它所依赖的文件旧，**make**程序执行依赖关系之后列出的*规则*。

在**makefile**中的所有注释都从“#”开始一直延续到本行的末尾。

作为一个简单的例子，一个名为“hello”的程序的**makefile**文件可能包含:

```
# A comment
hello.exe: hello.cpp
        mycompiler hello.cpp
```

这就是说**hello.exe**(目标文件)依赖于**hello.cpp**。当**hello.cpp**比**hello.exe**文件日期新时，**make**执行“规则”**mycompiler hello.cpp**。可能会有多重依赖和多重规则。许多**make**程序要求所有规则以tab开头。这与空格通常被忽略的空格不一样，空格可以用于格式化以便于阅读。

规则不仅局限于调用编译器。在**make**中还可以调用想要调用的任何程序。通过创建相互依赖的规则集的分组，可以修改源代码文件，输入**make**，确信所有受影响的文件会重新正确
204 地重建。

3.11.1.1 宏

makefile可以包含某些宏（注意，这些宏完全不同于C/C++的预处理宏）。用宏进行字符

串替换是很方便的。本书中的**makefile**使用一个宏去调用C++编译器。例如，

```
CPP = mycompiler
hello.exe: hello.cpp
        $(CPP) hello.cpp
```

等号‘=’用来把**CPP**定义为一个宏，符号‘$’和圆括号扩展宏。在这里，扩展意味着宏调用**$(CPP)**将被字符串**mycompiler**取代。对于上面的宏，如果想改变到名为**cpp**的不同编译器，只需把宏改变为：

```
CPP = cpp
```

也可以在宏中加入编译器标志，或使用分开的宏加入编译器标志。

3.11.1.2 后缀规则

说明**make**怎样为工程中的每个单独的**cpp**文件调用编译器是很乏味的，特别当知道了每次相同的处理过程之后。因为**make**的设计注重节约时间，所以只要依赖于文件名字后缀，它就有一种简化操作的方式。这些简化称为*后缀规则*。一条后缀规则是一种教**make**怎样从一种类型文件（如**.cpp**）转化为另一种类型（如**.obj**或**.exe**）的方法。一旦有了**make**从一种文件转化为另外一种文件的规则，其他要做的只是告诉**make**哪些文件依赖于其他文件。当**make**发现一个文件比它依赖的文件旧，它就会使用规则创建一个新文件。

后缀规则告诉**make**可以根据文件的扩展名去考虑怎样构建程序而不需用显式规则去构建一切。在这种情况下它指出：“调用下面的命令从扩展名为**cpp**的文件去构造扩展名为**exe**的文件”。上述例子看起来如以下所示：

205

```
CPP = mycompiler
.SUFFIXES: .exe .cpp
.cpp.exe:
        $(CPP) $<
```

.SUFFIXES指令告诉**make**必须注意后面的扩展名，因为它们对于这个特定的**makefile**有特殊的意义。其后看到后缀规则**.cpp.exe**，说明“这里是怎样把任何扩展名为**cpp**的文件转化为一个扩展名为**exe**的文件的”（当**cpp**文件比**exe**文件新的时候）。和前面一样使用了宏**$(CPP)**，但是发现了某种新东西：**$<**。因为以‘$’开头，所以这是一个宏，但它是**make**内部的特殊的宏。符号**$<**只能用于后缀规则，意思是“无论怎样都要触发的规则”（有时称为*依赖*），在本例中表示“需要被编译的**cpp**文件。”

一旦建立了后缀规则，就能简单地说明，例如说明“**make Union.exe**”，后缀规则会展开，即使在整个**makefile**文件中从未提及“Union”。

3.11.1.3 默认目标

在宏和后缀规则之后，**make**在文件中查找第一个“目标”，并构建它，除非指定了不同的目标文件。因此对于**makefile**文件：

```
CPP = mycompiler
.SUFFIXES: .exe .cpp
.cpp.exe:
        $(CPP) $<
target1.exe:
target2.exe:
```

如果简单地输入‘**make**’，那么会生成**target1.exe**文件（使用默认的后缀规则），因为它是

make遇到的第一个目标。为了生成target2.exe我们不得不显式说明‘make target2.exe’。这样做就比较冗长，因此通常会创建一个依赖于所有其余目标文件的默认“哑元”目标，如像：

206
```
CPP = mycompiler
.SUFFIXES: .exe .cpp
.cpp.exe:
        $(CPP) $<
all: target1.exe target2.exe
```

在这里，‘all’并不存在，没有名为‘all’的文件，因此每次键入make，它会把‘all’作为第一个目标（这是默认的目标），然后发现‘all’不存在，所以它检查所有的依赖关系。因此它查看target1.exe并（使用后缀规则）判断：(1) target1.exe文件是否存在，(2) target1.cpp文件是否比target1.exe文件新。如果(1) (2)都成立，就使用后缀规则（除非为某个特定的文件提供了一个显式规则）。然后在默认的目标列表上查找下一个目标文件。因此通过建立一个默认的目标文件列表（按习惯通常称为‘all’，但可以随便起名），只需简单地键入make就能够生成在工程中的所有可执行文件。此外，可以定义其他的非默认目标文件列表用于其他目的，例如，当键入‘make debug’时会重新构建所有带有调试信息的文件。

3.11.2 本书中的makefile

使用本书第2卷的ExtractCode.cpp程序，本书中列出的所有代码会被自动地从本书的ASCII文本文件中抽取出来，并存放在相应章的子目录中。此外，ExtractCode.cpp程序会在每个子目录中创建一些makefile文件（具有不同的文件名），所以可以简单地进入子目录并输入make -f mycompiler.makefile（把编译器名mycompiler替换为，‘-f’标志说明跟在后面的是makefile文件）。最后ExtractCode.cpp程序在根目录中创建了一个“管理”makefile文件，在根目录中书中的文件已经被扩展，该makefile被传到各个子目录中且调用相应的makefile文件。这样发出一个make命令就能够编译本书中的所有代码，当编译器不能处理特别的文件
207 （注意，与标准C++兼容的编译器能够编译本书中的所有文件）时，编译过程会停止。make的实现会随系统而异，因而在生成的makefile文件中仅仅使用了make的最基本的特征。

3.11.3 makefile的一个例子

正如提到的那样，代码提取工具ExtractCode.cpp自动地为每章产生makefile文件。因为这个原因，makefile并未放在书中每一章（所有的makefile文件都和源代码一起打包，可以从*www.BruceEckel.com*下载）。

然而看一个makefile的例子是有意义的。以下是一个例子的简化版本，该例子由本书中的代码提取工具自动生成。可以在每个子目录中（它们有不同的名字，用‘make -f’调用）发现多个makefile文件。下面的例子是用于GNU C++的：

```
CPP = g++
OFLAG = -o
.SUFFIXES : .o .cpp .c
.cpp.o :
  $(CPP) $(CPPFLAGS) -c $<
.c.o :
  $(CPP) $(CPPFLAGS) -c $<
```

```
all: \
  Return \
  Declare \
  Ifthen \
  Guess \
  Guess2
# Rest of the files for this chapter not shown

Return: Return.o
  $(CPP) $(OFLAG)Return Return.o

Declare: Declare.o
  $(CPP) $(OFLAG)Declare Declare.o

Ifthen: Ifthen.o
  $(CPP) $(OFLAG)Ifthen Ifthen.o

Guess: Guess.o
  $(CPP) $(OFLAG)Guess Guess.o
Guess2: Guess2.o
  $(CPP) $(OFLAG)Guess2 Guess2.o

Return.o: Return.cpp
Declare.o: Declare.cpp
Ifthen.o: Ifthen.cpp
Guess.o: Guess.cpp
Guess2.o: Guess2.cpp
```

208

CPP宏被设置为编译器的名字。为了使用不同的编译器，可以编辑**makefile**文件，或者在命令行上修改宏的值，例如:

```
make CPP=cpp
```

注意，对于另外编译器，**ExtractCode.cpp**代码具有自动建立**makefile**的方案。

第二个宏**OFLAG**是一个标志，用于指定输出文件的名字。虽然许多编译器自动假定输出文件的名字与输入的文件名一致，但是还是有例外（如Linux/Unix编译器，它默认创建一个**a.out**的输出文件）。

可以看出本例有两条后缀规则，一条用于**cpp**文件，另一条用于**.c**文件(以防需要编译C代码)。默认的目标是**all**，对于目标的所有的行用反斜线符号表示继续，直到**Guess2**，它是目标列表中的最后一行，因此不再需要反斜线符。本章有许多文件，为简单起见，这里只列出了一些文件。

后缀规则管理从**cpp**文件创建目标文件（以**.o**作为扩展名），但是通常对创建可执行文件需要有显式说明的规则，因为一个可执行文件通常是通过连接许多不同的目标文件而产生的，而**make**程序不知道哪些是目标文件。同样，在某些情况（Linux/Unix）下，对于可执行文件并无标准扩展名，这种情况下，后缀规则将不能工作。所以，我们发现创建最终执行文件都显式说明了规则。

209

makefile采用最安全的路线，其中尽可能少地使用**make**特征；在宏中也使用了目标、依赖性和宏的最基本的**make**概念。这种方式实质上保证能与尽可能多的**make**程序共同工作。这可能会生成较大的**makefile**，但这不是很糟的事，因为它是通过**ExtractCode.cpp**自动产生的。

有许多本书中未使用的其他**make**特征，以及更新和更加灵活的**make**版本和变型，其中具有可以大量节约时间的高级的快捷用法。本地文档可以对特定的**make**做更加详尽的描述，也可以从Oram和Talbott所著的《*Managing Projects with Make*》(O'Reilly, 1993)一书中学到关于**make**的更多的知识。如果有的编译器提供商不能支持**make**或者它使用非标准的**make**，可以从Internet上搜索GNU文档（有许多GNU文档）找到GNU **make**程序，这种程序实际上支持已经存在的所有平台。

3.12 小结

本章相当集中地浏览了C++语法的基本特征，许多特征是从C中继承过来的，同C是共有的（由此导致C++自夸与C向后兼容）。在这里虽然介绍了C++的某些特征，由于主要是针对熟悉编程的人，因此仅限于介绍C和C++的基本语法。如果读者已经是C程序员，那么除了C++的特征对读者多半是新的以外，还可能会发现一两点关于C的不熟悉的知识。如果读者觉得本章有点难以接受的话，应该事先浏览在光盘上的教程“Thinking in C: Foundations for C++ and Java”（它包含了讲义、练习和题解），该光盘被装订到本书中，也可以从www.BruceEckel.com得到。

3.13 练习

210

部分练习题的答案可以在本书的电子文档“*Annotated Solution Guide for Thinking in C++*”中找到，只需支付很少的费用就可以从http://www.BruceEckel.com得到这个电子文档。

3-1 建立一个头文件（扩展名为‘**.h**’）。在该文件中，声明一组函数，具有可变参数，返回值包括**void**、**char**、**int**和**float**类型。建立一个包含上述头文件的**.cpp**文件，创建所有这些函数的定义。每个定义应该简单地输出函数名，参数列表，并返回类型以便知道它已经被调用。创建另外一个**.cpp**文件，它包含头文件且定义**int main( )**，在其中调用已经定义的所有函数。编译和运行这个程序。

3-2 编写一个程序使用两重**for**循环和模运算符(**%**)去寻找和输出质数（只能被1和它本身整除的整数）。

3-3 编写一个程序，使用一个**while**循环从标准输入(**cin**)中把单词读入到**string**中。这是一个“无穷”**while**循环，可以使用**break**语句中断（和退出程序）。对于读入的每个单词，先用一系列的**if**语句把该单词“映射”为一个整数值，然后用该整数值作为一个**switch**语句的选择条件（这些操作并不意味着是良好的设计风格，这仅仅是为练习这些控制流程）。在每个**case**中，输出一些有意义的信息。判定哪些是“有趣“的单词以及这些单词的意义。同时判定哪个单词是程序结束的标志。用文件作为输入来测试该程序（如果想节省输入，这个文件将作为程序的源文件）。

3-4 修改**Menu.cpp**程序，使用**switch**语句代替**if**语句。

3-5 编写一个程序计算在“优先级”一节中的两个表达式的值。

3-6 修改**YourPets2.cpp**程序以使用不同的数据类型（**char**、**int**、**float**、**double**和这些类型的变型）。运行该程序并画出结果内存分布图。如果能在多种机器、操作系统或者编译器上运行该程序，用尽可能多的变化进行这个试验。

211

3-7 创建两个函数，一个接受一个**string***参数，另一个接受一个**string&**参数。每个函数

必须用它特有的方式去改变外部的**string**对象。在**main( )**中，创建和初始化一个**string**对象，输出它，然后把它传给每个函数，输出结果。

3-8 编写一个使用所有三个图形字符（trigraph）的程序，看看你的编译器是否支持它们。

3-9 编译和运行**Static.cpp**程序。从代码中删除**static**关键词，再次编译和运行，解释发生的现象。

3-10 试编译**FileStatic.cpp**和**FileStatic2.cpp**程序并把它们连接起来。得到的错误消息的含义是什么？

3-11 修改**Boolean.cpp**程序，用**double**值代替**int**值。

3-12 修改**Boolean.cpp**和**Bitwise.cpp**程序，使用显式运算符（如果你的编译器与C++标准兼容，那么它会支持这些运算符）。

3-13 使用在**Rotation.cpp**程序中的函数去修改**Bitwise.cpp**程序。确保用这种方式能清楚地显示在旋转过程中的结果。

3-14 修改**Ifthen.cpp**程序，使用三重**if-else**运算符(**?:**)。

3-15 创建一个含有两个**string**对象和一个**int**对象的**struct**。使用**typedef**为该**struct**命名。创建**struct**的一个实例，初始化实例的三个值，然后输出它们。获得实例的地址，然后赋值给定义的**struct**类型的指针。改变实例的三个值，然后通过指针把它们打印出来。

3-16 编制一个使用颜色枚举类型的程序。创建一个**enum**类型的变量，然后用**for**循环输出与颜色名对应的数字。 212

3-17 用**Union.cpp**程序做一个试验，删除各种**union**元素，观察对**union**大小的影响。试给该**union**的一个元素赋值（属于某一类型），然后通过不同的元素（属于不同的类型）输出它的值，看看发生了什么情况。

3-18 编制一个程序，连续定义两个**int**数组。第二个数组的开始下标紧接第一个数组的结束下标。给两个数组赋值。打印出第二个数组观察由此引起的变化。再在两个数组定义之间定义一个**char**变量，重复上述操作。可以创建一个数组输出函数以简化程序。

3-19 修改**ArrayAddresses.cpp**程序，使之能处理**char**、**long**、**int**、**float**以及**double**类型数据。

3-20 运用**ArrayAddresses.cpp**程序中的技术，输出在**StructArray.cpp**程序中定义的**struct**的大小以及数组元素的地址。

3-21 创建一个**string**对象数组且对每一个元素赋一个字符串。用**for**循环输出该数组。

3-22 在**ArgsToInts.cpp**的基础上，编制两个新程序，它们各自使用**atol( )**和**atof( )**函数。

3-23 修改**PointerIncrement2.cpp**程序，其中用**union**代替**struct**。

3-24 修改**PointerArithmetic.cpp**程序，其中使用**long**和**long double**。

3-25 定义一个**float**变量。获得它的地址，把地址转化为**unsigned char**，赋值给一个**unsigned char**指针。使用指针和**[ ]**符号引用**float**变量中的下标，并用本章中定义的**printBinary( )**函数输出该**float**的内存映像。(从0到**sizeof(float)**)。改变该**float**变量的值看看是否能推算出下一步的情况（**float**包含编码的数据）。

3-26 定义一个**int**数组。获得该数组的起始地址，使用**static_cast**把它转化为**void***。写

213 一个带以下参数的函数：一个**void***、一个数字（表明字节的数目）和一个值（表明每个字节需要设定的值）。该函数必须为特定范围内的每个字节设定特定的值。在这个**int**数组上试验函数。

3-27 建立一个**const double**类型数组和一个**volatile double**类型数组。通过引用每个数组的下标且用**const_cast**把每个元素分别转换为non-**const**和non-**volatile**，然后对每个元素赋值。

3-28 建立一个函数，该函数接受一个指向**double**类型数组的指针和一个表明该数组大小的值。该函数应该输出数组中的每个元素值。现在建立一个**double**类型的数组，且初始化每个元素的值为0，然后使用你的函数输出该数组。接着使用**reinterpret_cast**关键字把数组的起始地址转化为**unsigned char***，把每个元素值设置为1（提示：必须用**sizeof**运算符计算一个**double**类型变量包含的字节数）。现在使用你的数组输出函数输出结果。想想为什么每个元素值不设成1.0?

3-29 （带有挑战性）修改**FloatingAsBinary.cpp**程序以便能够以单独的二进制位组输出**double**类型数据。为实现目标，必须用自己的特殊代码（可以从**printBinary( )**函数中衍生）去替换对**printBinary( )**的调用，还必须查阅并理解自己的编译器的浮点数字节格式（这是具有挑战性的部分）。

3-30 创建**makefile**文件，可以把编译**YourPets1.cpp**和**YourPets2.cpp**程序（用你特定的编译器）以及执行这两个程序作为默认的目标，确保使用后缀规则。

3-31 修改**StringizingExpressions.cpp**程序，通过设置一个命令行标志，使得**P(A)**能用条件**#ifdef**与调试代码分离开。需要参考编译器文档，了解在命令行上怎样定义和取消定义预处理的值。

3-32 定义一个函数，该函数接受一个**double**型参数且返回一个**int**值。创建和初始化一
214 个指向该函数的指针，通过这个指针调用这个函数。

3-33 声明一个函数，该函数接受一个**int**参数且返回指向另一个函数的指针，这个函数接受一个**char**变量且返回一个**float**值。

3-34 修改**FunctionTable.cpp**程序使每个函数返回一个**string**(而不是输出一个消息)以便在**main( )**函数中输出。

3-35 为前面某个练习（自己选择）建立一个**makefile**文件，允许键入**make**以构建这个
215 程序，并且键入**make debug**以构建带有调试信息的程序。

第4章 数据抽象

C++是一个能提高生产效率的工具。为什么我们要努力（不管我们试图做的转变多么容易，还是需要努力）使我们从已经熟悉且生产效率高的某种语言转到另一种新的语言上？而且使用这种新语言，我们会在确实掌握它之前的一段时间内降低生产效率。这是因为我们确信：通过使用新工具将会得到更大的好处。 217

用编程术语来讲，生产效率提高意味着较少的人能够在较少的时间内完成更复杂和更重要的程序。当然，选择语言时确实还有其他问题，例如运行效率（该语言的本质会引起运行速度减慢和代码臃肿吗？）、安全性（该语言能有助于确信我们的程序做我们计划的事情并具有很强的纠错能力吗？）、可维护性（该语言能帮助我们创建易理解、易修改和易扩展的代码吗？）。这些都是本书要介绍的重要因素。

简单地讲，提高生产效率，意味着本应当花费三个人一星期的程序，现在只需要花费一个人一两天的时间。这会涉及到经济学的多层次问题。生产效率提高了，我们很高兴，因为我们正在建造的东西其功能将会更强；我们的客户（或老板）很高兴，因为产品生产又快，用人又少；我们的顾客很高兴，因为他们得到的产品更便宜。而大幅度提高生产效率的惟一办法就是使用其他人的代码，即是去使用库。

库只是他人已经写好的一些代码，按某种方式包装在一起。通常，最小的包是带有扩展名（如**lib**）的文件和向编译器声明库中有什么的一个或多个头文件。连接器知道如何在库文件中搜索和提取相应的已编译的代码。但是，这只是提供库的一种方法。在跨越多种体系结构的平台（例如Linux/Unix）上，通常，提供库的最明智的方法是使用源代码，这样它就能在新的目标机上被重新配置和编译。

所以，库大概是改进生产效率的最重要的方法。C++的主要设计目标之一就是使库使用起来更加容易。这种说法暗示，在C中使用库有一些难度。理解这个因素将使我们对C++设计有一个初步的了解，并因而对如何使用它有更深入的认识。 218

4.1 一个袖珍C库

一个库通常以一组函数开始，但是，已经用过第三方C 库的程序员知道，通常还有比行为、动作和函数更多的东西。有一些特性（颜色、重量、纹理、亮度），它们都由数据表示。在C语言中，当处理一组特性时，可以方便地把它们放在一起，形成一个**struct**。特别是，如果我们想表示问题空间中的多个类似的东西时，可以对每件东西创建这个**struct**的一个变量。

这样，在大多数C库中都有一组**struct**和一组作用在这些**struct**之上的函数。现在看一个这样的例子。假设有一个编程工具，当创建时，它的表现像一个数组，但它的长度能在运行时建立。我称它为**CStash**。虽然它是用C++写的，但是它有C语言的风格：

```
//: C04:CLib.h
// Header file for a C-like library
// An array-like entity created at runtime
```

```
typedef struct CStashTag {
  int size;      // Size of each space
  int quantity;  // Number of storage spaces
  int next;      // Next empty space
  // Dynamically allocated array of bytes:
  unsigned char* storage;
} CStash;

void initialize(CStash* s, int size);
void cleanup(CStash* s);
int add(CStash* s, const void* element);
void* fetch(CStash* s, int index);
int count(CStash* s);
void inflate(CStash* s, int increase);
219 ///:~
```

像**CStashTag**这样的标签名一般用于需要在**struct**内部引用自身的情况。例如，如果创建一个链表（链表中的每个元素包含一个指向下一个元素的指针），这样就需要指向下一个**struct**变量的指针，所以需要一种方法，能辨别这个**struct**内部的指针的类型。在C库中，几乎总是可以在如上所示的每个**struct**体中看到**typedef**。这样做使得能把这个**struct**作为一个新类型处理，并且可以定义这个**struct**的变量，例如：

```
CStash A, B, C;
```

storage指针是一个**unsigned char***。这是 C 编译器支持的最小的存储单位，尽管在某些机器上它可能与最大的一般大，这依赖于具体实现，但一般占一个字节长。人们可能认为，因为**CStash**被设计用于存放任何类型的变量，所以**void***在这里应当更合适。然而，我们的目的并不是把它当做某个未知类型的块处理，而是作为连续的字节块。

这个实现文件的源代码（如果购买一个商品化的库，可能得到的只是编译好的**obj**或**lib**或**dll**等）如下：

```
//: C04:CLib.cpp {O}
// Implementation of example C-like library
// Declare structure and functions:
#include "CLib.h"
#include <iostream>
#include <cassert>
using namespace std;
// Quantity of elements to add
// when increasing storage:
const int increment = 100;

void initialize(CStash* s, int sz) {
  s->size = sz;
  s->quantity = 0;
  s->storage = 0;
  s->next = 0;
220 }
int add(CStash* s, const void* element) {
  if(s->next >= s->quantity) //Enough space left?
    inflate(s, increment);
  // Copy element into storage,
  // starting at next empty space:
```

```
  int startBytes = s->next * s->size;
  unsigned char* e = (unsigned char*)element;
  for(int i = 0; i < s->size; i++)
    s->storage[startBytes + i] = e[i];
  s->next++;
  return(s->next - 1); // Index number
}

void* fetch(CStash* s, int index) {
  // Check index boundaries:
  assert(0 <= index);
  if(index >= s->next)
    return 0; // To indicate the end
  // Produce pointer to desired element:
  return &(s->storage[index * s->size]);
}

int count(CStash* s) {
  return s->next;  // Elements in CStash
}

void inflate(CStash* s, int increase) {
  assert(increase > 0);
  int newQuantity = s->quantity + increase;
  int newBytes = newQuantity * s->size;
  int oldBytes = s->quantity * s->size;
  unsigned char* b = new unsigned char[newBytes];
  for(int i = 0; i < oldBytes; i++)
    b[i] = s->storage[i]; // Copy old to new
  delete [](s->storage); // Old storage
  s->storage = b; // Point to new memory
  s->quantity = newQuantity;
}

void cleanup(CStash* s) {
  if(s->storage != 0) {
   cout << "freeing storage" << endl;
   delete []s->storage;
  }
} ///:~
```

221

initialize( )通过设置内部变量为适当的值。完成对**struct CStash**的必要设置。最初，设置**storage**指针为零，表示不分配初始存储。

add( )函数在**CStash**的下一个可用位置上插入一个元素。首先，它检查是否有可用空间，如果没有，它就用后面介绍的**inflate( )**函数扩展存储空间。

因为编译器并不知道存放的特定变量的类型（函数返回的都是**void***），所以不能只做赋值，虽然这的确是很方便的事情。我们必须一个字节一个字节地拷贝这个变量，完成这项拷贝任务的最简单的方法是使用数组下标。典型的情况是，在**storage**中已经存放有数据字节，由**next**的值指明。为了从正确的字节偏移开始，**next**必须乘上每个元素的长度（按字节），产生**startBytes**，然后，参数**element**转换为一个**unsigned char***，所以这就能一个字节接着一个字节地寻址，拷贝进可用的**storage**存储空间中。增加后的**next**指向下一个可用的存储块，**fetch( )**能用指向这个数值存放点的“下标数”重新得到这个值。

fetch( )首先看**index**是否越界，如果没有越界，返回所希望的变量地址，地址采用**index**参数计算。因为**index**指出了相对于**CStash**的偏移元素数，所以必须乘上每个单元拥有的字节数，产生按字节计算的偏移量。当此偏移用于计算使用数组下标的**storage**的下标时，得到的不是地址，而是处于这个地址上的字节。为了产生地址，必须使用地址操作符**&**。

对于有经验的C程序员，**count( )**乍看上去可能有点奇怪，它好像是自找麻烦，做手工很
222 容易做的事情。例如，如果有一个**struct CStash**，称为**intStash**，那么通过使用**intStash.next**查明它已经有多少个元素，这种方法似乎更直接，而不是去做**count(&intStash)**函数调用（它有更多的花费）。但是，如果想改变**CStash**的内部表示和计数计算的方法，那么这个函数调用接口就具有必要的灵活性。并且，很多程序员不会为找出库的“更好”的设计而操心。如果他们能着眼于**struct**和直接取**next**的值，那么就有可能不经允许而改变**next**。是不是能有一些方法使得库设计者能更好地控制像这样的问题呢？（是的，这是可预见的。）

4.1.1 动态存储分配

我们不可能预先知道一个**CSatsh**需要的最大存储量是多少，所以从堆（*heap*）中分配由**storage**指向的内存。堆是很大的内存块，用以在运行时分配一些小的存储空间。在写程序时，如果还不知道所需内存的大小，就可以使用堆。这样，可以直到运行时才知道需要存放200个**Airplane**的空间，而不只是20个。在标准C中，动态内存分配函数包括**malloc( )**、**calloc( )**、**realloc( )**和**free( )**。然而，C++不是采用库调用方法，而是采用更高级的方法，即被集成进这个语言中的动态存储分配，使用关键字**new**和**delete**。

inflate( )函数使用**new**为**CStash**得到更大的空间块。在这种情况下，只扩展内存而不缩小它，**assert( )**保证不把负数传给**inflate( )**作为**increase**的值。能够存储的新元素数（**inflate( )**完成后）由计算**newQuantity**，再乘以每个元素的字节数得到**newBytes**，这是分配的字节数。因此，可以知道从旧的位置拷贝多少字节，**oldBytes**用旧的**quantity**计算。

223 实际的存储分配出现在*new*表达式中，它是包含**new**关键字的表达式：

```
new unsigned char[newBytes];
```

new表达式的一般形式是：

new Type;

其中**Type**表示希望在堆上分配的变量的类型。在这种情况下，我们希望一个长度为**newBytes**的**unsigned char**数组，这就是作为**Type**出现的变量。还可以分配简单类型的变量，例如**int**，表示为：

```
new int;
```

虽然很少这样做，但这可以使得形式一致。

new表达式返回指向所请求的准确类型对象的指针，因此，如果声称**new Type**，返回的是指向**Type**的指针。如果声称**new int**，返回指向一个**int**的指针。如果希望**new unsigned char**数组，返回的是指向这个数的第一个元素的指针。编译器确保把这个new表达式的返回值赋给一个正确类型的指针。

当然，任何时候申请内存都有可能失败，例如存储单元用完，正如我们看到的，C++有判断是否内存分配不成功的机制。

一旦分配了新内存块，旧内存块中的数据必须拷贝进这个新内存块，这又是通过数组下标完成的，在循环中一次拷贝一个字节。数据被拷贝以后，必须释放老的内存块，以便程序的其他部分在需要新内存块时使用。**delete**关键字是**new**的对应关键字，任何由**new**分配的内存块必须用**delete**释放（如果忘记了使用**delete**，这个内存块就不能用了，这称为*内存泄漏*（*memory leak*））。泄漏到一定程度，内存就耗尽了。另外，释放数组有特殊的语法形式，也就是必须提醒编译器，注意这是指向对象数组的指针，而不是仅仅指向一个对象的指针。该 224
语法形式是在被释放的指针前面加一对空方括号:

```
delete []myArray;
```

一旦释放了旧的内存块，指向这个新内存块的指针就可以赋给**storage**指针，再调整数量，**inflate**就完成了任务。

注意，堆管理器是相当简单的，它给出一块内存，而当用**delete**释放时又把它收回。这里没有提供能压缩堆获得较大的空闲块的*堆压缩*内部工具。如果程序反复分配和释放堆存储，最终将会产生大量的空闲内存碎片，但却没有足够大的块能分配所需要的内存。堆压缩器使程序更复杂，因为要前后移动内存块，所以指针应保持正确的值。一些操作环境有内置的堆压缩器，但是，要求使用特殊的内存*句柄*（*handle*）（它能临时转换为指针，锁定内存后堆压缩器就不能移动它了）。

当编译时，如果在栈上创建一个变量，那么这个变量的存储单元由编译器自动开辟和释放。编译器准确地知道需要多少存储容量，根据这个变量的活动范围知道这个变量的生命期。而对动态内存分配，编译器不知道需要多少存储单元，不知道它们的生命期，不能自动清除。因此，程序员应负责用**delete**释放这块存储，**delete**告诉堆管理器，这个存储可以被下一次调用的**new**重用。在这个库里合理的方法是使用**cleanup( )**函数，它做所有关闭的事情。

为了测试这个库，让我们创建两个**CStash**。第一个存放**int**，第二个存放由80个**char**组成的数组: 225

```
//: C04:CLibTest.cpp
//{L} CLib
// Test the C-like library
#include "CLib.h"
#include <fstream>
#include <iostream>
#include <string>
#include <cassert>
using namespace std;

int main() {
  // Define variables at the beginning
  // of the block, as in C:
  CStash intStash, stringStash;
  int i;
  char* cp;
  ifstream in;
  string line;
  const int bufsize = 80;
  // Now remember to initialize the variables:
  initialize(&intStash, sizeof(int));
  for(i = 0; i < 100; i++)
```

```
    add(&intStash, &i);
  for(i = 0; i < count(&intStash); i++)
    cout << "fetch(&intStash, " << i << ") = "
         << *(int*)fetch(&intStash, i)
         << endl;
  // Holds 80-character strings:
  initialize(&stringStash, sizeof(char)*bufsize);
  in.open("CLibTest.cpp");
  assert(in);
  while(getline(in, line))
    add(&stringStash, line.c_str());
  i = 0;
  while((cp = (char*)fetch(&stringStash,i++))!=0)
    cout << "fetch(&stringStash, " << i << ") = "
         << cp << endl;
  cleanup(&intStash);
  cleanup(&stringStash);
} ///:~
```

按照C语言的要求，所有的变量都在**main( )**范围的开头定义。当然，必须在这个程序块
226 的稍后通过调用**initialize( )**对**CStash**初始化。C库的问题之一是必须向用户认真地说明初始化和清除函数的重要性，如果这些函数未被调用，就会出现许多问题。遗憾的是，用户不总是记得初始化和清除是必须的。他们只知道他们想完成什么，并不关心我们反复说的："喂，等一等，您必须首先做这件事"。一些用户甚至认为初始化这些元素是自动完成的。在C中，的确没有机制能防止这种情况的发生（只有预示）。

intStash存放整型，**stringStash**存放字符数组。这些字符数组是通过打开源代码文件**CLibTest.cpp**和从中把这些行读到被称为**line**的**string**中形成的，然后使用成员函数**c_str( )**产生一个指向**line**字符的指针。

装载了这两个**Stash**之后，可以显示它们。**intStash**的打印用了一个**for**循环，用**count( )**确定它的限度。**stringStash**的打印用一个**while**语句，如果**fetch( )**返回零则表示打印越界,这时跳出循环。

还应当注意到下面的类型转换:

```
cp = (char*)fetch(&stringStash,i++)
```

这是因为C++有严格的类型检查，它不允许直接向其他类型赋**void***（C允许）。

4.1.2 有害的猜测

在考虑C库创建中的一般问题之前还应当了解一个更重要的问题。注意头文件**CLib.h**必须包含在所有涉及到**CStash**的文件中，因为编译器不能正确地猜测这个结构像什么。然而，它
227 能猜测一个函数像什么。这看上去像是一个特征，但实际上是C的一个主要缺陷。

虽然总是应当通过包含头文件声明函数，但是函数声明在C中不是基本的。调用没有声明的函数在C中是可以的（但是在C++中不可以）。一个好的编译器会告诫程序员应当首先声明函数，但是，按照C语言的标准，并不强迫这样。这是危险习惯，因为C编译器可能会假设，带有一个**int**参数的函数有包含**int**的参数表，尽管它实际上可能包含了一个**float**。正如我们将看到的，这会产生非常难发现的bug。

每个独立的C文件（带有扩展名**.c**的文件）是一个翻译单元（*translation unit*）。这就是说，

编译器在每个翻译单元上单独运行，这时它只知道这个单元。这样，由包含文件提供的任何信息都是相当重要的，因为它决定了编译器对程序的其他部分的理解。在头文件中的声明是特别重要的，因为在包含头文件的任何地方，编译器准确地知道做什么。例如，如果在头文件中有一个声明是**void func(float)**，编译器就知道，如果用一个整型参数调用这个函数，应当把这个**int**转换为**float**，作为传递参数［这被称为提升(*promotion*)］。如果没有声明，C编译器简单地假设有一个**func(int)**存在，它就不做提升，错误数据就悄悄地传给了**func( )**。

对于每个翻译单元，编译器创造一个目标文件，用**.o**或者**.obj**，或者其他类似的符号作为扩展名。这些目标文件，连同必要的启动代码，由连接器连接为可执行程序。在连接过程中，应当确定所有的外部引用。例如，在**CLibTest.cpp**中，声明和使用了**initialize( )**和**fetch( )**这样的函数（这就是，告诉编译器它们像什么），但在其中未定义。它们在别处定义，即在**CLib.cpp**中。这样，在**CLib.cpp**中的调用是外部引用。当连接器将所有的对象文件放在一起时，它必须取未确定的外部引用，找出它们实际访问的地址。在可执行程序中用这些地址替换这些外部引用。 228

在C中，连接器所要查找的外部引用是一些简单的函数名字，通常在它们的前面加下划线。因此，所有的连接器都必须匹配调用处的函数名和在对象文件中的函数体。如果在某处我们调用一个函数**func(int)**，而在某一目标文件中有**func(float)**的函数体，连接器将认为有**_func**在一处而且有**_func**在另一处，它认为这都对，在调用**func( )**的地方，把**int**置入栈中，而**func( )**函数体处认为**float**在栈中。如果这个函数只读这个值而不写，它不会破坏这个栈。事实上，如果它读取的这个**float**值可能刚好有某种意思，这是最坏的情况，因为这个bug很难找出。

4.2 哪儿出问题

我们通常有特别的适应能力，即使是对本不应该适应的事情。**CStash**库的风格对于C程序员已经是常用的了，但是如果观察它一会儿，就会发现它是相当笨拙的。因为在使用它时，必须向这个库中的每一个函数传递这个结构的地址。而当读这些代码时，这种库机制会和函数调用的含义相混淆，当试图理解这些代码时也会引起混乱。

在C中，使用库的最大的障碍之一是*名字冲突*（*name clashes*）。对于函数，C使用单个名字空间，当连接器查找一个函数名时，它在一个主表中查找，而且，当编译器编译一个单元时，它只能对带有指定名字的单个函数进行处理工作。

假设决定要从不同的厂商购买两个库，并且每一个库都有一个必须被初始化和清除的结构。两个厂商都认为**initialize( )**和**cleanup( )**是好名字。如果在某个处理单元中同时包含了这两个库文件，C 编译器怎么办呢？幸好，标准 C出错，报告声明函数有两个不同的参数表中 229
类型不匹配。即便不把它们包含在同一个处理单元中，连接器也会有问题。好的连接器会发现这里有名字冲突，但有些编译器仅仅通过查找目标文件表，按照在连接表中给出的次序，取第一个找到的函数名（实际上，这可以看做是一种功能，因为可以用自己的版本替换一个库函数）。

无论哪种情况，都不允许使用包含具有同名函数的两个C库。为了解决这个问题，C库厂商常常会在它们的所有函数名前加上一个独特字符串。所以，**initialize( )**和**cleanup( )**可能变为**CStash_initialize( )**和**CStash_cleanup( )**。这是合乎逻辑的，因为它“修饰了”这个**struct**的名字，而该函数以这样的函数名对这个**struct**操作。

现在到了迈向C++第一步的时候。我们知道，**struct**内部的标识符不会与全局标识符冲突。而当一些函数在特定**struct**上运算时，为什么不把这一优点扩展到函数名上呢？也就是，为什么不让函数成为**struct**的成员呢？

4.3 基本对象

C++的第一步正是这样，函数可以放在**struct**内部，作为“成员函数”。**CStash**的C版本翻译成C++的**Stash**后是：

```
//: C04:CppLib.h
// C-like library converted to C++

struct Stash {
  int size;      // Size of each space
  int quantity;  // Number of storage spaces
  int next;      // Next empty space
   // Dynamically allocated array of bytes:
  unsigned char* storage;
  // Functions!
230 void initialize(int size);
  void cleanup();
  int add(const void* element);
  void* fetch(int index);
  int count();
  void inflate(int increase);
}; ///:~
```

首先，注意到这里没有**typedef**，而是要求程序员创建一个**typedef**。C++编译器把结构名转变为这个程序的新类型名（就像**int**、**char**、**float**和**double**是类型名一样）。

所有的数据成员与以前完全相同，但现在这些函数在**struct**的内部了。另外，注意到，对应于这个库中的C版本中第一个参数已经去掉了。在C++中，不是硬性传递这个结构的地址作为在这个结构上运算的所有函数的第一个参数，而是编译器秘密地做这件事。现在，这些函数的仅有的参数与它们所做的事情有关，而不与这些函数的运算机制有关。

认识到这些函数代码与在C库中的那些同样有效，是很重要的。参数的个数是相同的（虽然看不到这个结构地址被传进来，实际上它在这里），每个函数只有一个函数体。正因为如此，书写

```
Stash A, B, C;
```

并不意味着每个变量得到不同的**add( )**函数。

那样产生的代码几乎和为C库写的一样。更有趣的是，这包括了“名字修饰”，在C中也许应当像**Stash_initialize( )**、**Stash_cleanup( )**等这样修饰。当函数在**struct**内时，编译器有效地做了同样的事情。因此，在**Stash**内部的**initialize( )**将不会与任何其他结构中的 **initialize( )**相
231 冲突，即便是与全局函数名**initialize( )**，也不会冲突。大部分时间都不必为函数名字修饰而担心——而是使用未修饰的函数名。但有时还必须能够指出这个**initialize( )**属于这个**struct Stash**而不属于任何其他的**struct**。特别是，当正在定义这个函数时，需要完全指定它是哪一个。为了完成这个指定任务，C++有一个新的运算符(**::**)，即*作用域解析运算符*（这样命名是因为名字现在能在不同的范围内：在全局范围内或在一个**struct**的范围内）。例如，如果希望

指定**initialize( )**属于**Stash**，就写**Stash::initialize(int size)**。可以看到，在下面函数定义中是如何使用作用域运算符的：

```
//: C04:CppLib.cpp {O}
// C library converted to C++
// Declare structure and functions:
#include "CppLib.h"
#include <iostream>
#include <cassert>
using namespace std;
// Quantity of elements to add
// when increasing storage:
const int increment = 100;

void Stash::initialize(int sz) {
  size = sz;
  quantity = 0;
  storage = 0;
  next = 0;
}

int Stash::add(const void* element) {
  if(next >= quantity) // Enough space left?
    inflate(increment);
  // Copy element into storage,
  // starting at next empty space:
  int startBytes = next * size;
  unsigned char* e = (unsigned char*)element;
  for(int i = 0; i < size; i++)
    storage[startBytes + i] = e[i];
  next++;
  return(next - 1); // Index number
}

void* Stash::fetch(int index) {
  // Check index boundaries:
  assert(0 <= index);
  if(index >= next)
    return 0; // To indicate the end
  // Produce pointer to desired element:
  return &(storage[index * size]);
}

int Stash::count() {
  return next; // Number of elements in CStash
}

void Stash::inflate(int increase) {
  assert(increase > 0);
  int newQuantity = quantity + increase;
  int newBytes = newQuantity * size;
  int oldBytes = quantity * size;
  unsigned char* b = new unsigned char[newBytes];
  for(int i = 0; i < oldBytes; i++)
    b[i] = storage[i]; // Copy old to new
  delete []storage; // Old storage
```

232

```
  storage = b; // Point to new memory
  quantity = newQuantity;
}

void Stash::cleanup() {
  if(storage != 0) {
    cout << "freeing storage" << endl;
    delete []storage;
  }
} ///:~
```

在C和C++之间有以下不同：首先，头文件中的声明是由编译器要求的。在C++中，不能调用未事先声明的函数，否则编译器将报告一个出错信息。这是确保这些函数调用在被调用点和被定义点之间一致的重要方法。通过强迫在调用函数之前必须声明它，C++ 编译器可以保证我们用包含这个头文件的方式完成这个声明。如果在这个函数被定义的地方还包含有同
233 样的头文件，则编译器将作一些检查以保证在这个头文件中的声明和这个定义匹配。这意味着，这个头文件变成了函数声明的有效的仓库，并且保证这些函数在项目中的所有处理单元中使用一致。

当然，全局函数仍然可以在定义和使用它的每个地方用手工方式声明（这是很乏味的，以致于变得不太可能）。然而，必须在定义和使用之前声明结构，而最习惯放置结构定义的位置是在头文件中，除非有意把它藏在代码文件中。

可以看到，除了作用域和来自这个库的 C 版本的第一个参数不再是显式的这一事实以外，所有这些成员函数实际上都与C版本中的一样。当然，这个参数仍然存在，因为这个函数必须工作在一个特定的**struct**变量上。但是，在成员函数内部，成员照常使用。这样，不写**s->size = sz**，而写**size = sz**。这就消除了多余的**s->**，它对我们所做的任何事情不能添加任何含义。当然，C++ 编译器必须为我们做这些事情。实际上，它取“秘密”的第一个参数（也就是先前用手工传递的这个结构的地址），并且当提到**struct**的数据成员的任何时候，应用成员选择器。这意味着，当在另一个**struct**的成员函数中时，通过简单地给出成员的名字，就可以使用任何成员（包括其他成员函数）。编译器在找出这个名字的全局版本之前先在局部结构的名字中搜索。这个性能意味着不仅代码更容易写，而且更容易阅读。

但是，如果因为某种原因，我们希望能够处理这个结构的地址，情况会怎么样呢？在这个库的C版本中，这是很容易的，因为每个函数的第一个参数是叫做**s**的一个**CStash***。在C++ 中，事情是更一致的。这里有一个特殊的关键字，称为**this**，它产生这个**struct**的地址。
234 它等价于这个库的C版本的‘**s**’。所以可以用下面语句恢复成C风格。

```
this->size = Size;
```

对这种书写形式进行编译所产生的代码是完全一样的，因此不需要像这样的方式使用**this**。有时，我们会看到有人在代码的各处都明显地使用**this->**，但是，这不能对代码增加任何意义。通常，不经常用**this**，而只是需要时才使用（稍后，本书中将有一些使用**this**的例子）。

最后需要提到，在C中，可以赋**void***给任何指针，例如：

```
int i = 10;
void* vp = &i; // OK in both C and C++
int* ip = vp; // Only acceptable in C
```

而且编译器能够通过。但在 C++ 中，这个语句是不允许的。为什么呢？因为C对类型信息不挑剔，所以它允许未明确类型的指针赋给一个明确类型的指针。而C++则不同。类型在C++中是严格的，当类型信息有任何违例时，编译器就不允许。这一点一直是很重要的，而对于C++尤其重要，因为在**struct**中有成员函数。如果能够在 C++ 中向**struct**传递指针而不被阻止，那么就能最终调用对于**struct**逻辑上并不存在的成员函数。这是防止灾难的一个实际的办法。因此，C++允许将任何类型的指针赋给 **void***（这是 **void*** 的最初的意图，它需要足够大，以存放任何类型的指针），但不允许将**void**指针赋给任何其他类型的指针。一个类型转换总是需要告诉读者和编译器，我们实际上要把它作为目标类型处理。

这就带来了一个有趣的问题，C++的最重要的目的之一是能编译尽可能多的已存在的 C 代码，以便能容易地向这个新语言过渡。然而，这并不意味着C允许的任何代码都能自动地被C++接受。有一些 C 编译器允许的东西是危险的和易出错的（本书中还会看到它们）。C++ 编 235
译器对于这些情况产生警告和出错信息，其优点远大于缺点。实际上，在C中有许多我们知道有错误只是不能找出它的情况，但是一旦用 C++ 重编译这个程序，编译器就能指出这些问题。在 C 中，我们常常发现能使程序通过编译，然后我们必须再花力气使它工作。在 C++ 中，常常是，程序编译正确了，它也就能工作了。这是因为该语言对类型要求更严格的缘故。

在下面的测试程序中，可以看到 **Stash** 的 C++ 版本所使用的另一些东西。

```
//: C04:CppLibTest.cpp
//{L} CppLib
// Test of C++ library
#include "CppLib.h"
#include "../require.h"
#include <fstream>
#include <iostream>
#include <string>
using namespace std;

int main() {
  Stash intStash;
  intStash.initialize(sizeof(int));
  for(int i = 0; i < 100; i++)
    intStash.add(&i);
  for(int j = 0; j < intStash.count(); j++)
    cout << "intStash.fetch(" << j << ") = "
         << *(int*)intStash.fetch(j)
         << endl;
  // Holds 80-character strings:
  Stash stringStash;
  const int bufsize = 80;
  stringStash.initialize(sizeof(char) * bufsize);
  ifstream in("CppLibTest.cpp");
  assure(in, "CppLibTest.cpp");
  string line;
  while(getline(in, line))
    stringStash.add(line.c_str());
  int k = 0;
  char* cp;
  while((cp =(char*)stringStash.fetch(k++)) != 0)
    cout << "stringStash.fetch(" << k << ") = "
```

236

```
        << cp << endl;
  intStash.cleanup();
  stringStash.cleanup();
} ///:~
```

我们可以注意到，变量是实时（on the fly）定义的（上一章介绍过）。也就是说，它们能在作用域内的任何点上定义，而不是像C语言限制的那样，只能在作用域的开头部分。

这段代码和**CLibTest.cpp**相似，但在调用成员函数时，在函数名字之前使用成员选择运算符“.”。这是一个传统的文法，它模仿了结构数据成员的使用。它们的不同在于这里是函数成员，有一个参数表。

当然，该编译器实际产生的调用，看上去更像原来的C库函数。如果考虑名字修饰和**this**传递，C++ 函数调用 **intStash.initialize(sizeof(int), 100)**就和**Stash_initialize(&intStash, sizeof(int), 100)**一样了。如果想知道在内部所进行的工作，可以回忆最早的C++ 编译器**cfront**，它由AT&T开发，它输出的是C代码，然后再由 C 编译器编译。这个方法意味着**cfront** 能使C++很快地移植到有C 编译器的机器上，有助于快速地传播 C++ 编译器技术。正因为这个C++编译器必须产生C，所以我们知道必然有方法用C语言描述C++文法（某些编译器仍然允许产生C代码）。

ClibTest.cpp的另一个改变是引入**require.h**头文件，这是为这本书创造的头文件，用来完成比**assert( )**更复杂的错误检查任务。它包含了几个函数，其中一个就是在这里为了检查文件而使用的**assure( )**。它检查这个文件是否已经成功地打开了，如果没有，它就报告一个标准错
237 误，告诉这个文件不能打开并且退出程序（这样，它就需要文件名作为第二个参数）。**require.h**函数在全书中都会用到，特别是为了保证命令行参数的个数正确和保证文件确实打开了。**require.h**取代了不断重复的和分散进行的检查出错代码，并且提供非常有用的出错信息。这些函数将在本书的后面解释。

4.4 什么是对象

我们已经看到了一个最初的例子，现在回过头来看一些术语。把函数放进结构中是从C到C++中的根本改变，这引起我们将结构作为新概念去思考。在 C 中，**struct**是数据的凝聚，它将数据捆绑在一起，使得我们可以将它们看做一个包。但这除了能使编程方便之外，别无其他。对这些结构进行操作的函数可以在别处。然而将函数也放在这个包内，结构就变成了新的创造物了，它既能描写属性（就像 C **struct** 能做的一样），又能描述行为，这就形成了对象的概念。对象是一个独立的捆绑的实体，有自己的记忆和活动。

在C++中，对象就是变量，它的最纯正的定义是“一块存储区”（更明确的说法是，“对象必须有惟一的标识”，在C++中是一个惟一的地址）。它是一块空间，在这里能存放数据，而且还隐含着有对这些数据进行处理的操作。

不幸的是，对于各种语言，当涉及这些术语时，并不完全一致，尽管它们是可以接受的。我们有时还会遇到关于面向对象语言是什么的争论，虽然到目前为止看起来已经相当调和了。有一些语言是基于对象的（*object-based*），意味着它们有像 C++ 的结构加函数这样的对象，
238 正如已经看到的。然而，这只是到达面向对象语言历程中的一部分，停留在把函数捆绑在数据结构内部的语言是基于对象的，而不是面向对象的。

4.5 抽象数据类型

将数据连同函数捆绑在一起的能力可以用于创建新的数据类型。这常常被称为*封装*（*encapsulation*）[⊖]。一个已存在的数据类型可能有几块数据封装在一起，例如 **float**，有一个指数，一个尾数和一个符号位。我们能够告诉它做事情：与另一个 **float** 或 **int** 相加，等等。它有属性和行为。

Stash 的定义创建了一个新数据类型，可以**add( )**、**fetch( )**和 **inflate( )**。由说明**Stash s**创建一个 **Stash** 就像由说明 **float f** 创建一个 **float** 一样。一个 **Stash** 也有属性和行为，甚至它的活动就像一个实数——一个内建的数据类型。称**Stash**为*抽象数据类型*（*abstract data type*），也许这是因为它能允许从问题空间抽象概念到解空间。另外，C++编译器把它看做一个新的数据类型，如果说一个函数需要一个 **Stash**，编译器就确保传递了一个 **Stash** 给这个函数。对抽象数据类型［有时称为*用户定义类型*（*user-defined type*）］的类型检查就像对内建类型的类型检查一样严格。

然而，我们会看到在对象上执行操作的方法有所不同。**object.member Function(arglist)** 是对一个对象“调用一个成员函数”。而在面向对象的用法中，也称之为“向一个对象发送消息”。这样，对于 **Stash s**，语句 **s.add(&i)** “发送消息给 **s**”，也就是说，“将它与自己 **add( )**”。事实上，面向对象编程可以总结为一句话，“*向对象发送消息*”。实际上，需要做的所有事情就是创建一束对象并且给它们发送消息。当然，技巧是勾画出对象和消息是什么，但如果完成了这些，用 C++ 的实现就直截了当了。 239

4.6 对象细节

在研讨会上经常提出的一个问题是“对象应当多大和它应当像什么”。回答是“和 C的 **struct**一样”。事实上，对于C **struct**（不带有C++的改进），由C编译器产生的代码和由C++ 编译器产生的完全相同，这可以使那些在代码中离不开结构的安排和大小细节的程序员放心，并且由于某种原因，他们直接访问结构的字节而不是使用标识符。（具体取决于不可移植的结构的特定大小和布局。）

一个**struct**的大小是它的所有成员大小的和。有时，当一个**struct** 被编译器处理时，会增加额外的字节以使得边界整齐，这主要是为了提高执行效率。在第15章中，将会看到如何在结构中增加“秘密”指针，但现在不必关心这些。

用 **sizeof** 运算符可以确定 **struct** 的长度。这里有一个小例子：

```
//: C04:Sizeof.cpp
// Sizes of structs
#include "CLib.h"
#include "CppLib.h"
#include <iostream>
using namespace std;

struct A {
  int i[100];
};

struct B {
  void f();
```

⊖ 这个词会引起争论，一些人采用此处的定义，还有一些人用它描述访问权限控制（在下一章讨论）。

```
};

240  void B::f() {}
int main() {
  cout << "sizeof struct A = " << sizeof(A)
       << " bytes" << endl;
  cout << "sizeof struct B = " << sizeof(B)
       << " bytes" << endl;
  cout << "sizeof CStash in C = "
       << sizeof(CStash) << " bytes" << endl;
  cout << "sizeof Stash in C++ = "
       << sizeof(Stash) << " bytes" << endl;
} ///:~
```

在我的机器上（你的机器可能有不同的结果），第一个打印语句产生的结果是200，因为每个**int**占两个字节。**struct B**是奇异的，因为它是没有数据成员的**struct**。在C中，这是不合法的，但在C++中，以这种选择方式创建一个**struct**，惟一的目的就是划定函数名的范围，所以这是允许的。尽管如此，由第二个打印语句产生的结果是一个有点奇怪的非零值。在该语言的较早的版本中，这个长度是零，但是，当创建这样的对象时出现了笨拙的情况：它们与紧跟着它们创建的对象有相同的地址，没有区别。对象的基本规则之一是每个对象必须有一个惟一的地址，因此，无数据成员的结构总应当有最小的非零长度。

最后两个**sizeof**语句表明在C++中的结构长度与C中等价版本的长度相同。C++尽力不增加任何不必要的开销。

4.7 头文件形式

当创建了一个包含有成员函数的**struct**时，也就创建了一个新数据类型。一般情况，希望这个类型对于我们和其他人都容易使用。另外，还希望将接口（声明）和实现（成员函数的定义）隔离开来，使得实现能在不需要重新编译整个系统的情况下可以改变。最后，将这个
241 新类型的声明放到头文件中。

当我第一次学习用C编程时，头文件对我是神秘的。许多有关C语言的书似乎不强调它，并且编译器也并不强调函数声明，所以它在大部分时间内似乎是可要可不要的，除非要声明结构时。在C++中，头文件的使用变得非常明显。它们对于很容易的程序开发实际上是强制，在它们中放入非常特殊的信息：声明。头文件告诉编译器在我们的库中哪些是可用的。即便程序员只拥有头文件和对象文件或库文件，他也能用这个库。因为对于**cpp**文件能够不要源代码而使用库。头文件是存放接口规范的地方。

虽然编译器不强迫这样做，但是，用C++建造大项目的最好的方法是采用库，收集相关的函数到同一个对象模块或库中，并且使用同一个头文件存放所有这些函数的声明。在C++中这是必须的；在C中，可以把所有的函数都放进C库中，但是在C++中，由抽象数据类型确定库中的函数，这些函数通过它们共同访问一个**struct**中数据而联系起来。任何成员函数必须在**struct**声明中声明，不能把它放在其他地方。在C中，鼓励使用函数库，而在C++中，这是一项制度。

4.7.1 头文件的重要性

当使用库函数时，C允许不用头文件，而是简单地随手声明这个函数。过去，人们有时候这样做是为了通过避免打开和包含这个文件而略微提高编译器的速度（这对于现代编译器一

般不是问题)。例如，这里有C函数**printf( )**的一个非常简单的声明（来自**<stdio.h>**）：

```
printf(...);
```

242

省略号表示可变的参数表[⊖]，说明**printf( )**有一些参数，每个参数有类型，但省略了它们。这样，无论什么参数都接受。使用这样的声明，就中止了对参数的检查。

这种习惯会引起问题，如果随手声明了一个函数，在一个文件中可能会留下错误。因为编译器只看到在这个文件中的手工声明，它可能会适应错误。然后这个程序会被正确地连接，但是这样使用函数，一个文件将会出现错误。这是很难发现的错误，而使用头文件就可以很容易地避免这种情况。

如果将所有的函数声明都放在一个头文件中，并且将这个头文件包含在使用这些函数和定义这些函数的任何文件中，就能确保在整个系统中声明的一致性。通过将这个头文件包含在定义文件中，还可以确保声明和定义匹配。

在C++中，如果在一个头文件中声明了一个**struct**，我们在使用**struct**的任何地方和定义这个**struct**成员函数的任何地方必须包含这个头文件。如果不经声明就调用常规函数，调用或定义成员函数，C++编译器会给出错误消息。通过强制正确地使用头文件，语言保证库中的一致性，并通过在各处强制使用相同的接口，可以减少程序错误。

头文件是我们和我们的库的用户之间的合约。这份合约描述了我们的数据结构，为函数调用规定了参数和返回值。它说，“这里是对我的库做什么的描述。”用户需要其中一些信息以开发应用程序，编译器需要所有这些信息以生成正确的代码。这个**struct**的用户简单地包含这个头文件，创建这个**struct**的对象（实例），连接到对象模块或库（也就是被编译的代码）中。 243

通过要求我们在使用结构和函数之前声明所有这些结构和函数，在定义成员函数之前声明这些成员函数，编译器强制履行这个合约。这样，就强制我们在头文件中放置这些声明，强制将这个头文件包含在定义成员函数的文件中和使用这些函数的文件中。因为描述库的单个头文件被包含在整个系统各处，所以编译器能确保一致性和防止错误。

我们必须认识到为了正确地组织代码和编写有效的头文件，需要考虑几个具体问题。第一个问题涉及应当放什么到头文件中。基本的原则是“只限于声明”，即只限于对编译器的信息，不涉及通过生成代码或创建变量而分配存储的任何信息。这是因为头文件一般会包含在项目的几个翻译单元中，如果一个标识符在多于一处被分配存储，那么连接器就报告多次定义错误（这是C++的一次定义规则：可以对事物声明任意多次，但是对于每个事物只能实际定义一次）。

这条规则不是呆板的。如果在头文件中定义了一个“文件静态”变量（仅在一个文件内可视的变量），那么在整个项目中会有该数据的多个实例，但连接器不会冲突[⊖]。基本上，我们不希望做任何会引起连接时歧义性的事情。

4.7.2 多次声明问题

头文件的第二个问题是：如果把一个**struct**声明放在一个头文件中，就有可能在一个编译程序中多次包含这个头文件。输入输出流就是一个很好的例子。每次一个**struct**做I/O都可能

⊖ 写一个带有可变参数列表的函数定义，必须应用*varargs*，虽然在C++中应避免这样使用。varargs的应用细节请参照C手册。

⊖ 然而，在标准C++文件中，static是一个不予推荐的特征。

包含一个输入输出流文件。如果我们正在开发的**cpp**文件使用多种**struct**（典型的是每种包含
244 一个头文件），这样就有多次包含**<iostream>**和重声明输入输出流的危险。

编译器认为重声明结构（包括**struct**和**class**）是一个错误，因为它还允许对不同的类型使用相同的名字。为了防止多次头文件包含引起的错误，需要在头文件中用预处理器建立一些智能功能（标准C++头文件中**<iostream>**等已经具有这样的“智能”）。

C和C++都允许重声明函数，只要两个声明匹配即可，但是两者都不允许重声明结构。在C++中，这条规则是特别重要的，因为如果编译器允许重声明一个结构而且这两个声明不同，那么应当使用哪一个声明呢？

重声明在C++中出现了问题，因为每个数据类型（带函数的结构）一般有它自己的头文件，如果想创造另一个数据类型（它使用第一个数据类型），则我们必须将第一个数据类型的头文件包含在这另一个数据类型中。在我们项目的任何**cpp**文件中，很可能包含几个已经包含了这个相同的头文件的文件。在一次编译过程中，编译器可能会多次看到这个相同的头文件。除非特别处理，否则编译器将发现结构的重声明，并报告编译时错误。为了解决这个问题，需要知道更多的预处理器的知识。

4.7.3 预处理器指示#define、#ifdef和#endif

预处理器指示**#define**可以用来创建编译时标记。你有两种选择：你可以简单地告诉预处理器这个标记被定义，但不指定特定的值：

```
#define FLAG
```

或者给它一个值（这是典型的定义常数的C方法）：

```
#define PI 3.14159
```

245 无论哪种情况，预处理器都能测试该标记，检查它是否已经被定义：

```
#ifdef FLAG
```

这将得到一个真值，**#ifdef**后面的代码将包含在发送给编译器的包中。当预处理器遇到语句

```
#endif
```

或

```
#endif // FLAG
```

时包含终止。

在同一行中，**#endif**之后无注释是不合规定的，尽管一些编译器可以接受这样的行。**#ifdef/#endif**对可以相互嵌套。

#define的反意是**#undef**（“un-define”的简写），它将使得使用相同变量的**#ifdef**语句得到假值。**#undef**还引起预处理器停止使用宏。**#ifdef**的反意是**#ifndef**，如果标记还没有定义，它得到真值（这是在头文件中使用的一种指示）。

在C预处理器中还有其他有用的特性，因此我们还应当检查我们文档中的全部设置。

4.7.4 头文件的标准

对于包含结构的每个头文件，应当首先检查这个头文件是否已经包含在特定的**cpp**文件中

了。这需要通过测试预处理器的标记来检查。如果这个标记没有设置，这个文件没有包含，则应当设置它（所以这个结构不会被重声明），并声明这个结构。如果这个标记已经设置，则表明这个类型已经声明了，所以应当忽略这段声明它的代码。下面显示头文件的样子：

```
#ifndef HEADER_FLAG
#define HEADER_FLAG
// Type declaration here...
#endif // HEADER_FLAG
```

246

正如已经看到的，头文件第一次被包含，这个头文件的内容（包括类型声明）将被包含在预处理器中。对于在单个编译单元中的所有后续的包含，该类型声明被忽略。HEADER_FLAG可以是任何惟一的名字，但沿用的可靠标准是大写这个头文件的名字并且用下划线替换句点（但是前面的下划线是为系统名保留的）。例如：

```
//: C04:Simple.h
// Simple header that prevents re-definition
#ifndef SIMPLE_H
#define SIMPLE_H

struct Simple {
  int i,j,k;
  initialize() { i = j = k = 0; }
};
#endif // SIMPLE_H ///:~
```

虽然**#endif**之后的**SIMPLE_H**是注释，并且预处理器忽略它，但它对于文档是有用的。

防止多次包含的这些预处理器语句常常称为*包含守卫*(*include guard*)。

4.7.5 头文件中的名字空间

我们将会注意到，在这本书的几乎所有**cpp**文件中都有*使用指令*（*using directive*）描述，通常的形式如下：

```
using namespace std;
```

因为**std**是环绕整个标准C++库的名字空间，所以这个特定的使用指令允许不用限定方式使用标准C++库中的名字。但是，在头文件中是决不会看到使用指令的（至少，不在一个范围之外）。原因是，这样的使用指令去除了对这个特定名字空间的保护，并且这个结果一直持续到当前编译单元结束。如果将一个使用指令放在一个头文件中（在一个范围之外），这就意味着“名字空间保护”将在包含这个头文件的任何文件中消失，这些文件常常是其他的头文件。这样，如果将使用指令放在头文件中，将很容易最终实际上在各处“关闭”名字空间，因此不能体现名字空间的好处。

247

简言之，不要在头文件中放置使用指令。

4.7.6 在项目中使用头文件

当用 C++ 建立项目时，我们通常要汇集大量不同的类型（带有相关函数的数据结构）。一般将每个类型或一组相关类型的声明放在一个单独的头文件中，然后在一个处理单元中定义这个类型的函数。当使用这个类型时必须包含这个头文件，执行正确的声明。

有时这个模式会在本书中使用，但是，更常见的情况是例子很小，所以结构声明、函数定义和 **main( )** 函数可以出现在同一个文件中。然而，应当记住，你想要实际使用的是隔离的文件和头文件。

4.8 嵌套结构

在全局名字空间之外为数据和函数取名字的好处可以扩展到结构中。我们可以将一个结构嵌套在另一个结构中，这就可以将相关联的元素放在一起。声明文法是我们所期望的形式，就像在下面结构中可以看到的那样，这个结构用简单链表方式实现了一个下推栈（push-down stack），所以它决不会越出内存。

```
//: C04:Stack.h
// Nested struct in linked list
#ifndef STACK_H
#define STACK_H

struct Stack {
248   struct Link {
        void* data;
        Link* next;
        void initialize(void* dat, Link* nxt);
      }* head;
      void initialize();
      void push(void* dat);
      void* peek();
      void* pop();
      void cleanup();
};
#endif // STACK_H ///:~
```

这个嵌套 **struct** 称为 **Link**，它包括一个指向这个表中的下一个**Link** 的指针和一个指向存放在 **Link** 中的数据的指针。如果 **next** 指针是零，这就意味着到了表尾。

注意：**head** 指针紧接在 **struct Link** 声明之后定义，而不是单独定义 **Link* head**。这是来自C 语言的一种文法，但它强调在结构声明之后的分号的重要性，分号表明这个结构类型用逗号分开的定义表结束（通常这个定义表是空的）。

正如到目前为止所有描述的结构一样，嵌套结构有它自己的 **initialize( )** 函数，以便确保正确的初始化。**Stack** 既有 **initialize( )** 函数又有 **cleanup( )** 函数，此外还有**push( )** 函数，它取一个指向希望存放的数据（假设已经分配在堆中）的指针；还有 **pop( )** 函数，它返回栈顶的**data**指针并去除栈顶元素。（注意，当**pop( )**一个元素时，我们有责任销毁由 **data**所指的对象）。**peek( )**函数也从栈顶返回 **data** 指针，但是它在栈（**Stack**）中保留这个栈顶元素。

下面是一些成员函数的定义：

```
//: C04:Stack.cpp {O}
// Linked list with nesting
#include "Stack.h"
#include "../require.h"
249   using namespace std;
      void
      Stack::Link::initialize(void* dat, Link* nxt) {
        data = dat;
```

```
  next = nxt;
}

void Stack::initialize() { head = 0; }

void Stack::push(void* dat) {
  Link* newLink = new Link;
  newLink->initialize(dat, head);
  head = newLink;
}

void* Stack::peek() {
  require(head != 0, "Stack empty");
  return head->data;
}

void* Stack::pop() {
  if(head == 0) return 0;
  void* result = head->data;
  Link* oldHead = head;
  head = head->next;
  delete oldHead;
  return result;
}

void Stack::cleanup() {
  require(head == 0, "Stack not empty");
} ///:~
```

第一个定义特别有趣，因为它表明如何去定义一个嵌套结构的成员。只需要使用一个额外的作用域解析层，说明外围 **struct** 的名字。**Stack::Link::initialize( )** 函数取参数并把参数赋给它的成员们。

Stack::initialize( ) 函数置 **head** 为零，使得这个对象知道它有一个空表。

Stack::push( ) 取参数，也就是一个指向希望用的变量的指针，并且把这个指针推入 250
Stack。首先，使用 **new**为 **Link** 分配空间，它将插入栈顶。然后调用 **Link**的**initialize( )** 函数对这个 **Link** 的成员赋相应的值。注意，给**next** 指针赋当前的 **head**，而给 **head** 赋新的**Link** 指针。这就有效地将**Link** 推向这个表的顶部了。

Stack::pop( )取出当前在该**Stack**顶部的 **data** 指针，然后向下移 **head** 指针，删除该**Stack** 的旧的栈顶元素，最后返回这个取出的指针。当**pop( )**取出了最后的元素后，**head**再次变为零，意味着**Stack**为空。

实际上，**Stack::cleanup( )**不做任何清除工作，而是确立一项硬性的策略，“你（即使用这个**Stack**对象的客户程序员）负责弹出这个**Stack**的所有元素并且删除它们。”**require( )**指出：如果**Stack**非空，就产生一个编程错误。

为什么**Stack**的析构函数不能对客户程序员不做**pop( )**的所有对象负责呢？问题是，**Stack** 存放的是**void**指针，而且在第13章中我们将了解对**void***调用**delete**不能正确地清除内容。“谁对内存负责”这个主题不是一个简单的问题，在后面章节中将会看到相关的内容。

下面是一个测试**Stack**的例子：

```
//: C04:StackTest.cpp
```

```
//{L} Stack
//{T} StackTest.cpp
// Test of nested linked list
#include "Stack.h"
#include "../require.h"
#include <fstream>
#include <iostream>
#include <string>
using namespace std;

int main(int argc, char* argv[]) {
  requireArgs(argc, 1); // File name is argument
  ifstream in(argv[1]);
  assure(in, argv[1]);
  Stack textlines;
  textlines.initialize();
  string line;
  // Read file and store lines in the Stack:
  while(getline(in, line))
    textlines.push(new string(line));
  // Pop the lines from the Stack and print them:
  string* s;
  while((s = (string*)textlines.pop()) != 0) {
    cout << *s << endl;
    delete s;
  }
  textlines.cleanup();
} ///:~
```

251

这个例子非常类似于前面一个例子，但是它把来自文件的行（作为**string**指针）存放到**Stack**中，然后弹出它们，这会使这个文件被逆序打印出来。注意**pop( )**成员函数返回一个**void***，并且必须在被用之前转换回**string***。间接引用指针以便打印**string**。

当填充**textlines**时，通过建立**new string(line)**为每个**Push( )**“复制”**line**的内容。从新表达式返回的值是指向这个新创建的**string**的指针，并且从**line**复制信息。如果简单地传递**line**地址给**push( )**，最终用相同的地址充填**Stack**，所有的指针都指向**line**。在本书的后面，我们将学习更多的“复制”过程。

文件名取自命令行。为了保证在命令行上有足够的参数，我们看**require.h**头文件中的第二个函数**requireArgs( )**，它比较**argc**与期望的参数个数，如果没有足够的参数，打印相应的错误信息并退出程序。

252

4.8.1 全局作用域解析

编译器默认选择的名字（“最接近”的名字）可能不是我们要用的名字，作用域解析运算符可以避免这种情况。例如，假设有一个结构，它的局域标识符为 **a** ，但是我们希望在成员函数内选用全局标识符 **a**。这时，编译器将默认选择局域的另一个标识符，因而必须告诉编译器应该选择哪个标识符。当你要用作用域解析运算符指定一个全局名字时，在运算符前面不加任何东西。下面是一个显示变量和函数的全局作用域解析的例子：

```
//: C04:Scoperes.cpp
// Global scope resolution
int a;
```

```
void f() {}

struct S {
  int a;
  void f();
};

void S::f() {
  ::f();  // Would be recursive otherwise!
  ::a++;  // Select the global a
  a--;    // The a at struct scope
}
int main() { S s; f(); } ///:~
```

如果在**S::f()**中没有作用域解析运算符，编译器会默认地选择成员函数的**f()**和**a**。

4.9 小结

在本章中，我们学习了使用C++的基本方法，也就是在结构的内部放入函数。结构的这种新类型称为*抽象数据类型*（*abstract data type*），用这种结构创建的变量称为这个类型的*对象*（*object*）或*实例*（*instance*）。调用对象的成员函数称为向这个对象*发消息*（*sending a message*）。在面向对象的程序设计中的主要动作就是向对象发消息。 253

虽然将数据和函数捆绑在一起有很大好处，并使得库更容易使用（因为这可以通过隐藏名字防止名字冲突），但是还有大量的工作可以使C++编程更安全。在下一章中，我们将学习如何保护 **struct** 的一些成员，以使得只有我们能对它们进行操作。这就在“什么是结构的用户可以改动的”和“什么只是程序员可以改动的”之间建立了明确的界线。

4.10 练习

部分练习题的答案可以在本书的电子文档“*Annotated Solution Guide for Thinking in C++*”中找到，只需支付很少的费用就可以从http://www.BruceEckel.com得到这个电子文档。

4-1 在标准C库中，函数**puts()**能显示字符数组到控制台上（所以能写**puts("hello")**。试写一个C语言程序，这个程序使用**puts()**，但不包含**<stdio.h>**，也不声明这个函数。用C编译器编译这个程序。（有些C++编译器并不与它们的C编译器分开；在这种情况下，可能需要使用一个强制C编译的命令行标记。）然后再用C++编译器对它编译，注意它们之间的区别。

4-2 创建一个 **struct** 声明，它有单个成员函数，然后为这个成员函数创建定义。创建这个新数据类型的对象，再调用这个成员函数。

4-3 改变练习2的答案，使得**struct**在合适的“防护”头文件中声明，同时，它的定义在一个**cpp**文件中，**main()**在另一个文件中。

4-4 创建一个**struct**，它有一个**int**数据成员，再创建两个全局函数，每个函数都接受一个指向该**struct**的指针。第一个函数有第二个**int**参数，并设置这个**struct**的**int**为它的参数值，第二个函数显示来自这个**struct**的**int**。测试这两个函数。

4-5 重写练习4，将两个函数改为这个**struct**的成员函数，再次测试。

4-6 创建一个类，它使用**this**关键字（冗余地）执行数据成员选择和成员函数调用。 254

(**this**表示当前对象的地址)。

4-7 让**Stash**存放**double**，存入25个**double**值，然后把它们显示到控制台上。

4-8 用**Stack**重写练习7。

4-9 创建一个文件，包含以**int**为参数的函数**f( )**，用**<stdio.h>**中的**printf( )**函数将参数**int**的值显示到控制台上，即写**printf("%d\n",i)**，这里**i**是希望显示的**int**。创建另外一个单独的文件，它包含**main( )**，在该文件中声明**f( )**接受**float**参数。从**main( )**中调用**f( )**。尝试用C++编译器编译和连接这个程序，看看会发生什么事情。再用C编译器编译和连接这个程序，观察运行时会发生什么事情。解释这里的行为。

4-10 发现如何由你的C编译器和C++编译器产生汇编语言。用C写一个函数和用C++写一个带有一个成员函数的**struct**。由每一个编译器产生汇编语言，找出由你的C函数和C++成员函数产生的函数名，这样，你能看到什么样的名字修饰出现在编译器内部。

4-11 写一个**main( )**中有条件编译代码的程序，使得当预处理器的值被定义时打印一条消息，而不被定义时则打印另外一条消息。编译这一代码段在程序中有**#define**的试验代码，然后找出你的编译器在命令行上定义预处理器的方法，对它进行试验。

4-12 写一个程序，它带有参数总是为假（零）的**assert ( )**，当运行时看发生什么现象。现在用**#define NDEBUG**编译它，再次运行它，看有什么不同。

4-13 创建一个抽象数据类型，它表示录像带租赁店中的录像带，试考虑在录像带租赁管
255 理系统中为使录像带（**Video**）类型运作良好而必须的所有数据与运算。包含一个能显示录像带**Video**信息的**print( )**的成员函数。

4-14 创建一个**Stack**对象，能存放练习13中的**Video**对象。创建几个**Video**对象，把它们存放在**Stack**中，然后用**Video::print( )**显示它们。

4-15 写一个程序，使用**sizeof**打印出你的编译器的所有基本数据类型的长度。

4-16 修改**Stash**，使用**vector<char>**作为它的底层数据结构。

4-17 使用**new**动态创建下面类型的存储块：**int**、**long**、一个能存放100个**char**的数组、一个能存放100个**float**的数组。打印它们的地址，然后用**delete**释放这些存储。

4-18 写一个带有**char***参数的函数。用**new**动态申请一个**char**数组，长度与传给这个函数的**char**数组同。使用数组下标，从参数中拷贝字符到这个动态申请的数组中（不要忘记null终结符）并且返回拷贝的指针。在**main( )**中，通过传递静态引用字符数组，测试这个函数。然后取这个结果，再传回这个函数。打印这两个字符串和这两个指针，这样我们可以看到它们是不同的存储。使用**delete**，清除所有的动态存储。

4-19 显示在一个结构中声明另一个结构的例子（*嵌套结构*），声明这两个**struct**的数据成员，声明和定义这两个**struct**的成员函数。写一个**main( )**，测试这两个新类型。

4-20 结构有多大？写一段代码，打印几个结构的长度。创建几个只有数据成员的结构和几个既有数据成员又有函数成员的结构，然后创建一个完全没有成员的结构。打印出所有这些结构的长度。解释产生完全没有成员的结构的长度结果的原因。

256 4-21 C++自动创建**struct**的**typedef**的等价物，正如在本章中看到的。对于枚举和联合类型也是如此。写一个小程序来证明这一点。

4-22 创建一个存放**Stash**的**Stack**，每个**Stash**存放来自输入文件的5行。使用**new**创建**Stash**，读文件进入**Stack**，然后从这个**Stack**中提取并按原来的形式打印出来。

4-23 修改练习22，使得创建一个**struct**，它封装几个**Stash**的**Stack**。用户只能通过成员函数添加和得到一行，在这个覆盖下，**struct**使用**Stash**的**Stack**。

4-24 创建一个**struct**，它存放一个**int**和一个指向相同**struct**的另一个实例的指针。写一个函数，它能取这些**struct**的地址，并且能取一个表示被创建的表的长度的**int**。这个函数产生这些**struct**的一个完整链（链表），链表的头指针是这个函数的参数，**struct**中的指针指向下一个**struct**。用**new**产生一些新**struct**，将计数（对象数目）放在这个**int**中，对这个链表的最后一个**struct**的指针栏置零值，表示链表结束。写第二个函数，它取这个链表的头指针，并且向后移动到最后，打印出每个**struct**的指针值和**int**值。

4-25 重复练习24，但是将这些函数放在一个**struct**内部，而不是用“原始”的**struct**和函数。

257

第5章 隐藏实现

一个典型的C语言库通常包含一个**struct**和一些作用在这个**struct**上面的相关函数。迄今为止，我们已经看到C++怎样处理那些在概念上相关联的函数，并使它们在语义
259 上真正关联起来，具体做法是：

把函数的声明放在一个**struct**的范围之内，改变这些函数的调用方法，在调用过程中不再把结构地址作为第一个参数传递，并增加一个新的数据类型到程序中（这样就不必为**struct**标记创建一个**typedef**之类的声明）。

这样做带来很多方便——有助于组织代码，使程序易于编写和阅读。然而，在使得C++中的库比以前更容易的同时，还存在一些其他问题，特别是在安全与控制方面。本章重点讨论结构中的边界问题。

5.1 设置限制

在任何关系中，设立相关各方都遵从的边界是很重要的。一旦建立了一个库，我们就与该库的*客户程序员*（*client programmer*）建立了一种关系，客户程序员需要用我们的库来编写应用程序或建立另外的库。

在C语言中，**struct**同其他数据结构一样，没有任何规则，客户程序员可以在**struct**中做他们想做的任何事情，没有什么途径来强制任何特殊的行为。比如，即使已经看到了上一章中提到的**initialize( )**函数和**cleanup( )**函数的重要性，但客户程序员有权决定是否调用它们（我们将在下一章看到更好的方法）。再比如，我们可能不愿意让客户程序员去直接操纵**struct**中的某些成员，但在C语言中没有任何方法可以阻止客户程序员这样做。一切都是暴露无遗的。

需要控制对结构成员的访问有两个理由：一是让客户程序员远离一些他们不需要使用的工具，这些工具对数据类型内部的处理来说是必需的，但对客户程序员解决特定问题的接口却不是必须的。这实际上是为客户程序员提供了方便，因为他们可以很容易地知道，对他们
260 来说什么是重要的，什么是可以忽略的。

访问控制的理由之二是允许库的设计者改变**struct**的内部实现，而不必担心会对客户程序员产生影响。在上一章的**Stack**例子中，我们想以大块的方式来分配存储空间以提高速度，而不是在每次增加元素时调用**malloc( )** 函数来重新分配内存。如果这些库的接口部分与实现部分是清楚地分离并保护的，那么就能达到上述目的并且只需要让客户程序员重新连接一遍就可以了。

5.2 C++的访问控制

C++语言引进了三个新的关键字，用于在结构中设置边界：**public**、**private**和**protected**。它们的用法和含义从字面上就能理解。这些*访问说明符*（*access specifier*）只在结构声明中，它们可以改变跟在它们之后的所有声明的边界。无论什么时候使用访问说明符，后面必须加

一个冒号。

public意味着在其后声明的所有成员可以被所有的人访问。**public**成员就如同一般的**struct**成员。比如，下面的**struct**声明是相同的:

```
//: C05:Public.cpp
// Public is just like C's struct

struct A {
  int i;
  char j;
  float f;
  void func();
};

void A::func() {}

struct B {
public:
  int i;
  char j;
  float f;
  void func();
};
void B::func() {}

int main() {
  A a; B b;
  a.i = b.i = 1;
  a.j = b.j = 'c';
  a.f = b.f = 3.14159;
  a.func();
  b.func();
} ///:~
```

261

相对地，**private**关键字则意味着，除了该类型的创建者和类的内部成员函数之外，任何人都不能访问。**private**在设计者与客户程序员之间筑起了一道墙。如果有人试图访问一个私有成员，就会产生一个编译错误。在上面的例子中，我们可以让**struct B**中的部分数据成员隐藏起来，只有我们自己能访问它们:

```
//: C05:Private.cpp
// Setting the boundary

struct B {
private:
  char j;
  float f;
public:
  int i;
  void func();
};

void B::func() {
  i = 0;
  j = '0';
  f = 0.0;
```

```
};

int main() {
  B b;
  b.i = 1;     // OK, public
//!  b.j = '1';  // Illegal, private
262 //!  b.f = 1.0;  // Illegal, private
} ///:~
```

虽然函数**func ( )**可以访问**B**的所有成员（因为**func ( )**是**B**的成员，所以自动获得访问的权限），但一般的全局函数如**main( )**却不能访问，当然其他结构的成员函数同样也不能访问。只有那些在结构声明（“合约”）中明确声明的函数才能访问这些**private**成员。

对访问说明符的顺序没有特别的要求，它们可以出现不止一次，可以影响在它们之后和下一个访问说明符之前声明的所有成员。

5.2.1 protected说明符

最后一种访问说明符是**protected**。**protected**与**private**基本相似，只有一点不同：继承的结构可以访问**protected**成员，但不能访问**private**成员。这个问题要到第14章才讨论继承，那时会更清楚。现在可以把这两种说明符等同看待。

5.3 友元

如果程序员想允许显式地不属于当前结构的一个成员函数访问当前结构中的数据，那该怎么办呢？他可以在该结构内部声明这个函数为**friend**（友元）。注意，一个**friend**必须在一个结构内声明，这一点很重要，因为程序员（和编译器）必须能读取这个结构的声明以理解这个数据类型的大小、行为等方面的所有规则。有一条规则在任何关系中都很重要，那就是“谁可以访问我的私有实现部分”。

类控制着哪些代码可以访问它的成员。如果不是一个**friend**的话，程序员没有办法从类外“破门而入”，他不能声明一个新类，然后说“嘿，我是类**Bob**的朋友（友元）”，不能指望这
263 样就可以访问类**Bob**的**private**成员和**protected**成员。

程序员可以把一个全局函数声明为**friend**，也可以把另一个结构中的成员函数甚至整个结构都声明为**friend**，请看下面的例子：

```
//: C05:Friend.cpp
// Friend allows special access

// Declaration (incomplete type specification):
struct X;

struct Y {
  void f(X*);
};

struct X { // Definition
private:
  int i;
public:
  void initialize();
```

```
  friend void g(X*, int); // Global friend
  friend void Y::f(X*);  // Struct member friend
  friend struct Z; // Entire struct is a friend
  friend void h();
};

void X::initialize() {
  i = 0;
}

void g(X* x, int i) {
  x->i = i;
}

void Y::f(X* x) {
  x->i = 47;
}

struct Z {
private:
  int j;
public:
  void initialize();
  void g(X* x);
};
void Z::initialize() {
  j = 99;
}

void Z::g(X* x) {
  x->i += j;
}

void h() {
  X x;
  x.i = 100; // Direct data manipulation
}

int main() {
  X x;
  Z z;
  z.g(&x);
} ///:~
```

264

struct Y有一个成员函数**f( )**，它将修改**X**类型的对象。这里有一个难题，因为C++的编译器要求在引用任一变量之前必须先声明，所以**struct Y**必须在它的成员**Y :: f(X*)**被声明为**struct X**的一个友元之前声明，但要声明**Y :: f(X*)**，又必须先声明**struct X**。

解决的办法：注意到**Y :: f(X*)**引用了一个**X**对象的地址（*address*）。这一点很关键，因为编译器知道如何传递一个地址，这一地址具有固定的大小，而不管被传递的是什么对象，即使它还没有完全知道这种对象类型大小。然而,如果试图传递整个对象，编译器就必须知道**X**的全部定义以确定它的大小以及如何传递，这就使得程序员无法去声明一个类似于**Y :: g(X)**的函数。

通过传递**X**的地址，编译器允许程序员在声明**Y :: f(X*)**之前做一个**X**的不完全的类型说明

(*incomplete type specification*)。这一点是用如下的声明时完成的:

265
```
struct X;
```

该声明仅仅是告诉编译器，有一个叫**X**的**struct**，所以当它被引用时，只要不涉及名字以外的其他信息，就不会产生错误。

这样，在**struct X**中，就可以成功地声明**Y :: f(X*)**为一个**friend**函数，如果程序员在编译器获得**Y**的全部说明信息之前声明它，就会产生一条错误，这种安全措施保证了数据的一致性，同时减少了错误的出现。

再来看看其他两个**friend**函数，第一个声明将一个全局函数**g( )**作为一个**friend**，但**g( )**在这之前并没有在全局范围内作过声明，这表明**friend**可以在声明函数的同时又将它作为**struct**的友元。这种扩展声明对整个结构同样有效:

```
friend struct Z;
```

是**Z**的一个不完全的类型说明，并把整个结构都当做一个**friend**。

5.3.1 嵌套友元

嵌套的结构并不能自动获得访问**private**成员的权限。要获得访问私有成员的权限，必须遵守特定的规则：首先声明（而不定义）一个嵌套的结构，然后声明它是全局范围使用的一个**friend**，最后定义这个结构。结构的定义必须与**friend**声明分开，否则编译器将不把它看做成员。请看下面的例子:

```
//: C05:NestFriend.cpp
// Nested friends
#include <iostream>
#include <cstring> // memset()
using namespace std;
const int sz = 20;

struct Holder {
private:
  int a[sz];
266 public:
  void initialize();
  struct Pointer;
  friend Pointer;
  struct Pointer {
  private:
    Holder* h;
    int* p;
  public:
    void initialize(Holder* h);
    // Move around in the array:
    void next();
    void previous();
    void top();
    void end();
    // Access values:
    int read();
    void set(int i);
  };
```

```
};

void Holder::initialize() {
  memset(a, 0, sz * sizeof(int));
}

void Holder::Pointer::initialize(Holder* rv) {
  h = rv;
  p = rv->a;
}

void Holder::Pointer::next() {
  if(p < &(h->a[sz - 1])) p++;
}

void Holder::Pointer::previous() {
  if(p > &(h->a[0])) p--;
}

void Holder::Pointer::top() {
  p = &(h->a[0]);
}

void Holder::Pointer::end() {
  p = &(h->a[sz - 1]);
}
```

267

```
int Holder::Pointer::read() {
  return *p;
}

void Holder::Pointer::set(int i) {
  *p = i;
}

int main() {
  Holder h;
  Holder::Pointer hp, hp2;
  int i;

  h.initialize();
  hp.initialize(&h);
  hp2.initialize(&h);
  for(i = 0; i < sz; i++) {
    hp.set(i);
    hp.next();
  }
  hp.top();
  hp2.end();
  for(i = 0; i < sz; i++) {
    cout << "hp = " << hp.read()
         << ", hp2 = " << hp2.read() << endl;
    hp.next();
    hp2.previous();
  }
} ///:~
```

一旦**Pointer**被声明，它就可以通过下面语句来获得访问**Holder**的私有成员的权限:

```
friend Pointer;
```

struct Holder包含一个**int**数组和一个**Pointer**，可以通过**Pointer**来访问这些整数。因为**Pointer**与**Holder**紧密联系，所以有必要将它作为结构**Holder**中的一个成员。但是，又因为**Pointer**是同**Holder**分开的，所以程序员可以在函数**main( )**中定义它们的多个实例，然后用它们来选择数组的不同部分。由于**Pointer**是一个结构而不是C语言中原始意义上的指针，因此
268 程序员可以保证它总是安全地指向**Holder**的内部。

使用标准C语言库函数 **memset( )** (在**<cstring>**中)可以使上面的程序变得容易。它把起始于某一特定地址的内存(该内存作为第一个参数)从起始地址直至其后的**n**（**n**作为第三个参数）个字节的所有内存都设置成同一个特定的值（该值作为第二个参数）。当然，程序员可以使用一个简单的循环来反复设置需要使用的所有内存，而且，**memset( )**是可用的，经过了很好的测试不太可能引入错误，而且比起手工编码来更有效。

5.3.2 它是纯面向对象的吗

这种类定义提供了有关权限的信息，通过查看该类可以知道哪些函数可以改变该类的私有部分。如果一个函数被声明为**friend**，就意味着它不是这个类的成员函数，却可以修改该类的私有成员，而且必须被列在该类的定义当中，因此可以认为它是一个特权函数。

C++不是完全的面向对象语言，而只是一个混合产品。增加**friend**关键字就是为了用来解决一些实际问题。这也说明了这种语言是不纯的。毕竟C++语言的设计目的是实用，而不是追求理想的抽象。

5.4 对象布局

第4章讲述了为C编译器而写的一个**struct**，然后一字不动地用C++编译器进行编译。这里分析**struct**的对象布局，也就是，各个变量放在分配给对象的内存的什么位置？如果C++编译器改变了C **struct**中的布局，那么在任何C语言代码中使用**struct**中变量的位置信息在C++中就会出错。

当开始使用访问说明符时，我们就已经完全进入了C++的领地，情况开始有所改变。在一个特定的“访问块”（被访问说明符限定的一组声明）内，这些变量在内存中肯定是连续存
269 放的，这和在C语言中一样，然而这些“访问块”本身可以不按声明的顺序在对象中出现。虽然编译器通常都是按访问块出现的顺序给它们分配内存，但并不是一定要这样，因为特定机器的体系结构和操作环境可对**private**成员和**protected**成员提供明确的支持，将其放在特定的内存位置上。C++语言的访问说明符并不想限制这种长处。

访问说明符是结构的一部分，它们并不影响从这个结构创建的对象。程序开始运行之前，所有的访问说明信息都消失了。访问说明信息通常是在编译期间消失的。在程序运行期间，对象变成了一个存储区域，别无他物，因此，如果有人真的想破坏这些规则并且直接访问内存中的数据，就如在C中所做的那样，那么C++并不能防止他做这种不明智的事，它只是提供给人们一个更容易、更方便的方法。

一般说来，在程序员编写程序时，依赖特定实现的任何东西都是不合适的。如确有必要，这些特定实现部分应封装在一个结构之内，这样当环境改变时，只需修改一个地方就行了。

5.5 类

访问控制通常是指实现细节的隐藏（*implementation hiding*）。将函数包含到一个结构内（常称为封装[⊖]）来产生一种带数据和操作的数据类型，由访问控制在该数据类型之内确定边界。这样做的原因有两个：首先是决定哪些客户程序员可以用，哪些客户程序员不能用。我们可以建立结构内部的机制，而不必担心客户程序员会把内部的数据机制当做他们可使用的接口的一部分来访问。 270

这就直接导出了第二个原因，那就是将具体实现与接口分离开来。如果该结构被用在一系列的程序中，而客户程序员只能对**public**的接口发送消息，这样就可以改变所有声明为**private**的成员而不必去修改客户程序员的代码。

同时采用封装和访问控制可以防止一些情况的发生，而这在C语言的**struct**类型中是做不到的。现在我们已经处在面向对象编程的世界中，在这里，结构就是由对象组成的类，就像人们可以描述由鱼或鸟组成的类一样，任何属于该类的对象都将共享这些特征和行为。也就是说，结构的声明已经变成该类型的所有对象看起来像什么以及将如何行动的描述。

在最初的面向对象编程语言Simula-67中，关键字**class**被用来描述一个新的数据类型。这显然启发了Stroustrup在C++中选用同样的关键字，以强调这是整个语言的关键所在。新的数据类型并非只是C中的带有函数的**struct**，这当然需要用一个新的关键字。

然而在C++中使用的**class**逐渐变成了一个非必要的关键字。它和**struct**的每个方面都是一样的，除了**class**中的成员默认为**private**，而**struct**中的成员默认为**public**。下面有两个结构，它们将产生相同的结果。

```
//: C05:Class.cpp
// Similarity of struct and class

struct A {
private:
  int i, j, k;
public:
  int f();
  void g();
};

int A::f() {
  return i + j + k;
}
void A::g() {                                    271
  i = j = k = 0;
}

// Identical results are produced with:

class B {
  int i, j, k;
public:
  int f();
  void g();
```

⊖ 正如前面说明的一样，访问控制有时被认为是一种封装。

```
};

int B::f() {
  return i + j + k;
}

void B::g() {
  i = j = k = 0;
}

int main() {
  A a;
  B b;
  a.f(); a.g();
  b.f(); b.g();
} ///:~
```

在C++中，**class**是面向对象编程的基本概念，它是一个关键字，但本书将不再用粗体字来表示——由于总要用到“class”，都标出来会令人厌烦。转换到类是如此重要，因此我怀疑Stroustrup偏向于将**struct**重新定义，但考虑到对C的向后兼容性而没有这样做。

许多人喜欢用一种更像**struct**的风格去创建一个类，因为可以通过不顾及类的“默认为**private**”的行为，而使用首选为**public**的原则。

```
class X {
public:
  void interface_function();
private:
  void private_function();
  int internal_representation;
};
```

272

之所以这样做，是因为这样可以让读者首先更清楚地看到他们所要关心的成员，然后可以忽略所有声明为**private**的成员。事实上，所有其他成员都必须在类中声明的惟一原因是让编译器知道对象有多大，以便为它们分配合适的存储空间，并保证它们的一致性。

但本书中的示例仍采用首先声明**private**成员的格式，如下例:

```
class X {
  void private_function();
  int internal_representation;
public:
  void interface_function();
};
```

有些人甚至不厌其烦地在他们的私有成员名字前加上私有标志:

```
class Y {
public:
  void f();
private:
  int mX;  // "Self-decorated" name
};
```

因为**mX**已经隐藏于**Y**的范围内，所以**m**（用它来指示成员）并不是必需的。然而在一个有许多全局变量的项目中（虽然极力想避免使用全局变量，但它们仍不可避免地在一些项目中出

现），这种命名有助于在一个成员函数的定义体内识别出哪些是全局变量，哪些是成员变量。

5.5.1 用访问控制来修改Stash

现在把第4章的例子用类及访问控制来改写一下。请注意客户程序员的接口部分现在已经很清楚地区分开了，完全不用担心客户程序员会偶然地访问到他们不该访问的内容了。 273

```
//: C05:Stash.h
// Converted to use access control
#ifndef STASH_H
#define STASH_H

class Stash {
  int size;      // Size of each space
  int quantity;  // Number of storage spaces
  int next;      // Next empty space
  // Dynamically allocated array of bytes:
  unsigned char* storage;
  void inflate(int increase);
public:
  void initialize(int size);
  void cleanup();
  int add(void* element);
  void* fetch(int index);
  int count();
};
#endif // STASH_H ///:~
```

inflate( )函数声明为**private**，因为它只被**add( )**函数调用，所以它属于内部实现部分，不属于接口部分。这就意味着以后可以调整这些实现的细节，使用不同的系统来管理内存。

在此例中，除了包含文件的名字之外，只有上面的头文件需要更改，实现文件和测试文件是相同的。

5.5.2 用访问控制来修改Stack

对于第二个例子，我们把**Stack**改写成一个类。现在嵌套的数据结构是**private**。这样做的好处是可以确保客户程序员既看不到，也不依赖于**Stack**的内部表示：

```
//: C05:Stack2.h
// Nested structs via linked list
#ifndef STACK2_H
#define STACK2_H

class Stack {
  struct Link {
    void* data;
    Link* next;
    void initialize(void* dat, Link* nxt);
  }* head;
public:
  void initialize();
  void push(void* dat);
  void* peek();
```

274

```
  void* pop();
  void cleanup();
};
#endif // STACK2_H ///:~
```

与上例一样，实现部分不需要改动，这里不再赘述。测试部分也一样，惟一改动了的地方是类的接口部分的健壮性。访问控制的真正价值体现在开发阶段中的防止越界。事实上，只有编译器知道类成员的保护级别，与成员关联的这些访问控制信息并没有被传递给连接器。所有的访问保护检查都是由编译器来完成的，在运行期间不再检查。

注意面向客户程序员的接口部分现在是一个压入式堆栈。它是用一个链表结构实现的，但可以换成其他的形式，而不会影响客户程序员处理问题，更重要的是，不需要改动客户程序员的代码。

5.6 句柄类

C++中的访问控制允许将实现部分与接口部分分开，但实现部分的隐藏是不完全的。编译器仍然必须知道一个对象的所有部分的声明，以便正确地创建和管理它。可以想像一种只
275 需声明一个对象的公共接口部分的编程语言，它将私有的实现部分隐藏起来。但C++要尽可能多地在编译期间作静态类型检查。这意味着尽早捕获错误，也意味着程序具有更高的效率。然而包含私有实现部分会带来两个影响：一是即使客户程序员不能轻易地访问私有实现部分，但可以看到它；二是造成一些不必要的重复编译。

5.6.1 隐藏实现

有些项目不可让最终客户程序员看到其实现部分。例如可能在一个库的头文件中显示一些策略信息，但公司不想让这些信息被竞争对手获得。我们可能在从事一个安全性很重要的系统——比如一个加密算法——我们不想在文件中暴露任何线索，以防有人破译代码。或许我们把库放在了一个“有敌意”的环境中，在那里程序员会不顾一切地用指针和类型转换来访问我们的私有成员。在所有这些情况下，就有必要把一个编译好的实际结构放在实现文件中，而不是让其暴露在头文件中。

5.6.2 减少重复编译

在我们的编程环境中，当一个文件被修改，或它所依赖的头文件被修改时，项目管理员需要重复编译该文件。这意味着程序员无论何时修改了一个类，无论修改的是公共的接口部分，还是私有成员的声明部分，他都必须再次编译包含头文件的所有文件。这就是通常所说的*易碎的基类问题*（*fragile base-class problem*）。对于一个大的项目而言，在开发初期这可能非常难以处理，因为内部实现部分可能需要经常改动。如果这个项目非常大，用于编译的时
276 间过多可能妨碍项目的快速转型。

解决这个问题的技术有时称为*句柄类*（*handle class*）或称为“Cheshire cat”[⊖]。有关实现的任何东西都消失了，只剩一个单指针“smile”。该指针指向一个结构，该结构的定义与其

⊖ 这个名字应该归功于John Carolan，他是C++语言早期的先驱之一，当然，还有Lewis Carroll（作家，《爱丽丝奇遇记》的作者——编辑注）。可以把该技术看做本书第2卷中论述的一种“桥”(bridge)设计模式。

所有的成员函数的定义一同出现在实现文件中。这样，只要接口部分不改变，头文件就不需变动。而实现部分可以按需要任意更改，完成后只需要对实现文件进行重新编译，然后重新连接到项目中。

这里有一个说明这一技术的简单例子。头文件中只包含公共的接口和一个单指针，该单指针指向一个没有完全定义的类。

```
//: C05:Handle.h
// Handle classes
#ifndef HANDLE_H
#define HANDLE_H

class Handle {
  struct Cheshire; // Class declaration only
  Cheshire* smile;
public:
  void initialize();
  void cleanup();
  int read();
  void change(int);
};
#endif // HANDLE_H ///:~
```

这是所有客户程序员都能看到的。下面这行

```
struct Cheshire;
```

是一个不完全的类型说明（*incomplete type specification*）或类声明（*class declaration*）［类定义（*class definition*）包含类的主体］。它告诉编译器，**Cheshire** 是一个结构名，但没有提供有关该**struct**的任何细节。这些信息对产生一个指向**struct**的指针来说已经足够了。但在提 277
供了一个结构的主体部分之前不能创建一个对象。在这种技术里，包含具体实现的结构体被隐藏在实现文件中。

```
//: C05:Handle.cpp {O}
// Handle implementation
#include "Handle.h"
#include "../require.h"

// Define Handle's implementation:
struct Handle::Cheshire {
  int i;
};

void Handle::initialize() {
  smile = new Cheshire;
  smile->i = 0;
}

void Handle::cleanup() {
  delete smile;
}

int Handle::read() {
  return smile->i;
}
```

```
void Handle::change(int x) {
  smile->i = x;
} ///:~
```

Cheshire 是一个嵌套结构，所以它必须用作用域符定义：

```
struct Handle::Cheshire {
```

在**Handle::initialize( )**中，为**Cheshire** 结构分配存储空间，在**Handle::cleanup( )**中，释放这些存储空间。这些内存被用来代替类的所有**private**部分。当编译**Handle.cpp**时，这个结构的定义被隐藏在目标文件中，没有人能看到它。如果改变了**Cheshire**的组成，惟一要重新
278 编译的是**Handle.cpp**，因为头文件并没有改动。

Handle的使用就像任何类的使用一样，包含头文件、创建对象、发送消息。

```
//: C05:UseHandle.cpp
//{L} Handle
// Use the Handle class
#include "Handle.h"

int main() {
  Handle u;
  u.initialize();
  u.read();
  u.change(1);
  u.cleanup();
} ///:~
```

客户程序员惟一能访问的就是公共的接口部分，因此，如果只修改了在实现中的部分，上面文件就不须重新编译。虽然这并不是完全对实现进行了隐藏，但毕竟是一大改进。

5.7 小结

在C++中，访问控制为类的创建者提供了很有价值的控制。类的客户程序员可以清楚地看到，什么可以用，什么应该忽略。更重要的是，它保证了类的客户程序员不会依赖类的任何实现细节。有了这些，我们就可以更改类的实现部分，没有客户程序员会因此而受到影响，因为他们并不能访问类的这一部分。

一旦拥有了更改实现部分的自由，就可以在以后的时间里改进我们的设计，而且允许犯错误。要知道，无论如何小心地计划和设计，都可能犯错误。犯些错误也是相对安全的，这
279 意味着我们会变得更有经验，会学得更快，就会更早完成项目。

一个类的公共接口部分是客户程序员能看到的。所以在分析设计阶段，保证接口的正确性更加重要。但这并不是说接口不能作修改。如果第一次没有正确地设计接口部分，可以再增加函数，这样就不需要删除那些已使用该类的程序代码。

5.8 练习

部分练习题的答案可以在本书的电子文档 "*Annotated Solution Guide for Thinking in C++*" 中找到，只需支付很少的费用就可以从http://www.BruceEckel.com得到这个电子文档。

5-1 创建一个类，具有**public**、**private** 和**protected**数据成员和函数成员。创建该类的一个对象，看看当试图访问所有的类成员时会得到一些什么编译信息。

5-2 写一个名为**Lib**的**struct**，包括三个**string**对象**a**、**b**和**c**。在函数 **main( )**中创建一个**Lib**对象 **x**，对**x.a**、**x.b**、**x.c**赋值并打印出这些值。再用数组**string s[3]**代替 **a**、**b**、**c**。你将会看到由于这种改变，函数 **main( )**中的代码出错。另外再创建一个名为**Libc**的类，有三个私有的 **string**对象**a**、**b**、**c**以及成员函数 **seta( )**、**geta( )**、**setb( )**、**getb( )**、**setc( )** 和**getc( )**，这些成员函数用来设置和得到三个私有成员的值。像上面那样写一个函数 **main( )**，现在把**private string**对象 **a**、**b**、**c**变成一个**private**数组**string s[3]**。这样你将会看到即使有这种改变，函数 **main( )**中代码照样执行。

5-3 创建一个类和一个全局**friend**函数来处理类的**private**数据。

5-4 编写两个类，每个类都有一个成员函数，该函数中把一个指针指向另一个类的一个对象。在函数**main( )**中创建两个实例对象，调用前面每一个类中的成员函数。

5-5 创建三个类。第一个类包括**private**数据，并且整个第二个类和第三个类的成员函数是它的友元，在函数**main( )**中演示一下它们是如何正确运行的。 280

5-6 创建一个**Hen**类，在该类中，嵌套一个**Nest**类，在**Nest**类中，有一个**Egg**类成员。每一个类都有一个成员函数**display( )**，在函数**main( )**中创建每一个类的实例，然后调用每一个类的**display( )**函数。

5-7 修改练习6中类**Nest**和类**Egg**，使它们都包含**private**数据，通过声明友元使套装类能够访问这些**private**数据。

5-8 创建一个有很多数据成员的类，这些数据成员分布在由**public**、**private** 和**protected**所指定的区域中。增加一个成员函数**showMap( )**，该成员函数打印这些数据成员的名字和它们的地址。如果有可能，在多个编译器、计算机或者操作系统中编译并运行这个程序，看目标代码中布局是否一样。

5-9 拷贝第4章中针对**Stash**的实现和测试文件，在本章中编译并测试**Stash.h**。

5-10 把来自练习6中**Hen**类的对象放到结构**Stash**中，取出并打印它们（如果还没有做，必须增加函数**Hen::print( )**）。

5-11 拷贝第4章中针对**Stack**的实现和测试文件，在本章中编译并测试**Stack2.h**。

5-12 把来自练习6中**Hen**类的对象放到结构**Stack**中，取出并打印它们（如果还没有做，必须增加函数**Hen::print( )**）。

5-13 修改**Handle.cpp** 中的结构**Cheshire**，检验工程管理员是否只对这个文件进行了重新编译和重新连接，而不重新编译 **UseHandle.cpp**。

5-14 使用 “Cheshire cat” 技术创建类**StackOfInt**(一个存放整型数的堆栈)，该技术隐藏了用于存储类**StackImp**中的元素的低级数据结构。实现**StackImp**有两种方法：一种是使用固定长的整型数组，另一种是使用**vector<int>**。在第一种方法中，由于通过预先调整设置了堆栈的最大尺寸，不必担心数组扩展。注意类**StackOfInt.h**不必随**StackImp**一起改变。 281 282

第6章 初始化与清除

第4章采用一个典型C语言库中所有分散的构件，并把它们封装进一个结构(一个
283 抽象数据类型，从现在起，称其为一个类)，从而在库的应用方面作出了重大改进。

这样不仅为访问库构件提供了统一的入口，也用类名隐藏了类内部的函数名。在第5章中，我们介绍了访问控制（实现隐藏）。这就为类的设计者提供了一种设立边界的途径，通过边界的设立来决定哪些是允许客户程序员处理的，哪些是禁止客户程序员处理的。这意味着，对某种数据类型进行操作的内部机制处于类的设计者控制之下，可以由他们斟酌决定，这样也可以让客户程序员清楚哪些成员是他们能够使用并应该加以注意的。

封装和访问控制在改进库的易用性方面取得了重大进展。它们提供的“新的数据类型”的概念在某些方面比来自C语言的现有的内置数据类型要好。现在，C++编译器可以为这种新的数据类型提供类型检查保证，从而在使用这种数据类型时就确保了一定程度的安全性。

当然，说到安全性，C++的编译器能比C编译器提供更多的功能。在本章及以后的章节中，我们将看到C++的另外一些特征。它们可以让程序中的错误充分暴露，有时甚至在编译这个程序之前，帮助查出错误，但通常是编译器的警告和出错信息。基于这个原因，我们不久就会习惯于这样一种情景：一个C++程序在第一次编译时就能正确运行。

安全性包括初始化和清除两个方面。在C语言中，如果程序员忘记了初始化或清除一个变量，就会出现一大段程序错误。这在一个C库中尤其如此，特别是当客户程序员不知如何初始化一个**struct**，或甚至不知道他们必须要初始化一个**struct**时。（库中通常不包含初始化函数，所以客户程序员不得不自己手工初始化**struct**。）清除是一个特殊问题，因为C程序员一旦用过一个变量后就会把它忘记，所以对于一个库的**struct**来说必要的清除工作往往会被遗忘。

284 在C++中，初始化和清除的概念是简化库的使用的关键所在，并可以减少那些在客户程序员忘记去完成这些操作时会引起的细微错误。本章就来讨论C++的这些特征，它们有助于保证正常的初始化和清除。

6.1 用构造函数确保初始化

在**Stash**和**Stack**类中都曾调用**initialize( )**函数，这个函数名暗示无论用什么方法使用这些对象都应当在对象使用之前调用这一函数。不幸的是，这要求客户程序员必须正确地初始化。而客户程序员在专注于用那令人惊奇的库来解决问题的时候，往往忽视了初始化的细节。在C++中，初始化实在太重要了，不应该留给客户程序员来完成。类的设计者可以通过提供一个叫做*构造函数*（*constructor*）的特殊函数来保证每个对象都被初始化。如果一个类有构造函数，编译器在创建对象时就自动调用这一函数，这一切在客户程序员使用他们的对象之前就已经完成了。是否调用构造函数不需要客户程序员来考虑，它是由编译器在对象定义时完成的。

接下来的问题是这个函数叫什么名字。这必须考虑两点，首先这个名字不能与类的其他成员函数冲突，其次，因为该函数是由编译器调用的，所以编译器必须总能知道调用哪个函

数。Stroustrup的方法似乎是最简单也最符合逻辑的：构造函数的名字与类的名字一样。这样的函数在初始化时会自动被调用。

下面是一个带构造函数的类的简单例子：

```
class X {
  int i;
public:
  X();  // Constructor
};
```

285

现在当一个对象被定义时：

```
void f() {
  X a;
  // ...
}
```

这时就好像**a**是一个**int**一样：为这个对象分配内存。但是当程序执行到**a**的序列点（*sequence point*）执行的点时，构造函数自动被调用，因为编译器已悄悄地在**a**的定义点处插入了一个**X::X( )**的调用。就像其他成员函数被调用一样。传递到构造函数的第一个（秘密）参数是**this**指针，也就是调用这一函数的对象的地址，不过，对构造函数来说，**this**指针指向一个没有被初始化的内存块，构造函数的作用就是正确的初始化该内存块。

像其他函数一样，也可以通过向构造函数传递参数，指定对象该如何创建或设定对象初始值，等等。构造函数的参数保证对象的所有部分都被初始化成合适的值。举例来说：如果类**Tree**有一个带整型参数的构造函数，用以指定树的高度，那么就必须这样来创建一个树对象：

```
Tree t(12);  // 12-foot tree
```

如果**Tree(int)**是惟一的构造函数，编译器将不会用任何其他方法来创建一个对象（在下一章将看到多个构造函数以及调用它们的不同方法）。

关于构造函数就全部介绍完了。构造函数有着特殊的名字，在每个对象创建时，编译器自动调用的函数。尽管构造函数简单，但是它解决了类的很多问题，并使得代码更容易读写。例如在前面的代码段中，对有些**initialize( )**函数并没有看到显式的调用，这些函数从概念上说是与定义分开的。在C++中，定义和初始化是集为一体的，不能只取其中之一。

286

构造函数和析构函数是两个非常特殊的函数：它们没有返回值。这与返回值为**void**的函数显然不同。后者虽然也不返回任何值，但还可以让它做点别的事情，而构造函数和析构函数则不允许。在程序中创建和消除一个对象的行为非常特殊，就像出生和死亡，而且总是由编译器来调用这些函数以确保它们被执行。如果它们有返回值，要么编译器必须知道如何处理返回值，要么就只能由客户程序员自己来显式地调用构造函数与析构函数，这样一来，安全性就被破坏了。

6.2 用析构函数确保清除

作为一个C程序员，可能经常想到初始化的重要性，但很少想到清除的重要性。毕竟，清除一个**int**时需要做什么？仅仅是忘记它。然而，在一个库中，对于一个曾经用过的对象，仅仅“忘记它”是不安全的。如果它修改了某些硬件参数，或在屏幕上显示了一些字符，或在堆中分配了一些内存，那么将会发生什么呢？如果只是“忘记它”，对象就永远不会消失。在

C++中，清除就像初始化一样重要。它通过析构函数来保证清除的执行。

析构函数的语法与构造函数一样，用类的名字作为函数名。然而析构函数前面加上一个代字号（~），以和构造函数区别。另外，析构函数不带任何参数，因为析构不需任何选项。下面是一个析构函数的声明：

```
class Y {
public:
  ~Y();
};
```

287

当对象超出它的作用域时，编译器将自动调用析构函数。可以看到，在对象的定义点处构造函数被调用，但析构函数调用的惟一证据是包含该对象的右括号。即使用**goto**语句跳出这一程序块（为了与C 语言向后兼容，**goto**在C++中仍然存在，当然有时也是为了方便），析构函数仍然被调用。应该注意*非局域的goto语句*（*nonlocal goto*），它们是用标准C语言库中的**setjmp( )** 和**longjmp( )**函数实现的，这些非局域的**goto**语句将不会引发析构函数的调用。（这是一种规范：但有的编译器可能并不用这种方法来实现。对那些不在规范中的特征的依赖性意味着这样的代码是不可移植的）。

下例说明了构造函数与析构函数的上述特征：

```
//: C06:Constructor1.cpp
// Constructors & destructors
#include <iostream>
using namespace std;

class Tree {
  int height;
public:
  Tree(int initialHeight);  // Constructor
  ~Tree();  // Destructor
  void grow(int years);
  void printsize();
};

Tree::Tree(int initialHeight) {
  height = initialHeight;
}

Tree::~Tree() {
  cout << "inside Tree destructor" << endl;
  printsize();
}

void Tree::grow(int years) {
  height += years;
}
```

288

```
void Tree::printsize() {
  cout << "Tree height is " << height << endl;
}

int main() {
  cout << "before opening brace" << endl;
  {
```

```
    Tree t(12);
    cout << "after Tree creation" << endl;
    t.printsize();
    t.grow(4);
    cout << "before closing brace" << endl;
  }
  cout << "after closing brace" << endl;
} ///:~
```

下面是上面程序的输出结果:

```
before opening brace
after Tree creation
Tree height is 12
before closing brace
inside Tree destructor
Tree height is 16
after closing brace
```

可以看到析构函数在包括它的右括号处被调用。

6.3 清除定义块

在C中，总是要在一个程序块的左括号一开始就定义好所有的变量，这在程序设计语言中不算少见，其理由无非是因为“这是一种好的编程风格”。在这点上，我有自己的看法。我认为它总是带来不方便。作为一个程序员，每当需要增加一个变量时我都得跳到块的开始，我发现如果变量定义紧靠着变量的使用点时，程序的可读性更强。

也许这些争论仅限于格式。在C++中，是否一定要在块的开头就定义所有变量成了一个 289
很突出的问题。如果存在构造函数，那么当对象产生时它必须首先被调用，如果构造函数带有一个或者更多个初始化参数，那么怎么知道在块的开头定义这些初始化信息呢？在一般的编程情况下将做不到这点，因为C中没有私有成员的概念。这样很容易将定义与初始化部分分开，然而C++要保证在一个对象产生时，它同时被初始化。这可以保证系统中没有未初始化的对象。C并不关心这些。事实上，C要求在块的开头定义变量，而这时还不知道一些必要的初始化信息[⊖]，这样就鼓励了不初始化变量的习惯。

通常，在C++中，在还不拥有构造函数的初始化信息时不能创建一个对象，所以不必在块的开头定义所有变量。事实上，这种语言风格似乎鼓励把对象的定义放得离使用点处尽可能近一点。在C++中，对一个对象适用的所有规则，对内部类型的对象也同样适用。这意味着任何类的对象或者内部类型的变量都可以在块的任何地方定义。这也意味着可以等到已经知道一个变量的必要信息时再去定义它，所以总是可以同时定义和初始化一个变量。

```
//: C06:DefineInitialize.cpp
// Defining variables anywhere
#include "../require.h"
#include <iostream>
#include <string>
using namespace std;

class G {
```

⊖ 在标准C的升级版本C99中，可以像C++一样，在某一块的任意地方定义变量。

```
  int i;
public:
  G(int ii);
};
G::G(int ii) { i = ii; }

int main() {
  cout << "initialization value? ";
  int retval = 0;
  cin >> retval;
  require(retval != 0);
  int y = retval + 3;
  G g(y);
} ///:~
```

290

上例中可以看到先是执行一些代码，然后**retval**被定义和初始化，接着是一条用来接受客户程序员输入的语句，最后定义**y**和**g**。然而，在C中这些变量都只能在块的开始处定义。

一般说来，应该在尽可能靠近变量的使用点处定义变量，并在定义时就初始化（这是对内部类型的一种格式上的建议，但在那里可以不做初始化）。这是出于安全性的考虑，通过减少变量在块中的生命周期，就可以减少该变量在块的其他地方被误用的机会。另外，程序的可读性也增强了，因为读者不需要跳到块的开头去确定变量的类型。

6.3.1 for循环

在C++中，经常看到**for**循环的计数器直接在**for**表达式中定义：

```
for(int j = 0; j < 100; j++) {
    cout << "j = " << j << endl;
}
for(int i = 0; i < 100; i++)
    cout << "i = " << i << endl;
```

上述这些语句是一种重要的特殊情况，这可能使那些刚接触C++的程序员感到迷惑不解。

291

变量**i**和**j**都是在**for**表达式中直接定义的（在C中不能这样做），然后它们就可以作为一个变量在**for**循环中使用。这给程序员带来很大的方便，因为从上下文中我们可以清楚地知道变量**i**、**j**的作用，所以不必再用诸如**i_loop_counter**之类的名字来定义一个变量，以清晰地表示这一变量的作用。

然而，如果想把变量**i**、**j**的生命期扩展到**for**循环之外，就会有一些问题[⊖]。

在第3章中指出，**while**语句和**switch**语句也允许在它们的表达式内定义变量，尽管这种用法远没有**for**循环重要。

要注意局部变量会屏蔽其封闭块内的其他同名变量。通常，使用与全局变量同名的局部变量会使人产生误解，并且也易于产生错误[⊖]。

小作用域是良好设计的指标。如果一个函数有好几页，也许正在试图让这个函数完成太

⊖ C++标准草案一个更早版本中，允许变量生命期扩展到包含**for**循环块的作用域。有些编译器仍旧这样实现，但它是不恰当的，所以我们的代码只有我们把块限制在**for**循环中时才可移植。

⊖ Java语言认为这种做法不妥，将其认为是出错。

多的工作。如果用更多细化的函数，不仅更有用，而且更容易发现错误。

6.3.2 内存分配

现在，一个变量可以在某个程序范围内的任何地方定义，所以在这个变量的定义之前是无法对它分配内存空间的。通常，编译器更可能像C编译器一样，在一个程序块的开头就分配所有的内存。这些对我们来说是无关紧要的，因为作为一个程序员，在变量定义之前总是无法访问这块存储空间（即该对象）[⊖]。即使存储空间在块的一开始就被分配，构造函数也仍然要到对象的定义时才会被调用，因为标识符只有到此时才有效。编译器甚至会检查有没有把 292
一个对象的定义（构造函数的调用）放到一个条件块中，比如在**switch**块中声明，或可能被**goto**跳过的地方。下例中解除注释的语句会导致一个警告或一个错误。

```
//: C06:Nojump.cpp
// Can't jump past constructors

class X {
public:
  X();
};

X::X() {}

void f(int i) {
  if(i < 10) {
   //! goto jump1; // Error: goto bypasses init
  }
  X x1;  // Constructor called here
 jump1:
  switch(i) {
    case 1 :
      X x2;  // Constructor called here
      break;
  //! case 2 : // Error: case bypasses init
      X x3;  // Constructor called here
      break;
  }
}

int main() {
  f(9);
  f(11);
}///:~
```

在上面的代码中，**goto**和**switch**都可能跳过构造函数调用的序列点，甚至构造函数没有被调用时，这个对象也会在后面的程序块中起作用，所以编译器给出了一条出错信息。这就确 293
保了对象在产生的同时被初始化。

当然，这里讨论的内存分配都是在堆栈中进行的。内存分配是通过编译器向下移动堆栈指针来实现的（这里的“向下”是相对而言的，实际指针值增加，还是减少，取决于机器）。也可以在堆栈中使用**new**为对象分配内存，这将在第13章中进一步介绍。

⊖ 当然，我们可以通过指针来访问这些存储空间，但这样做是非常有害的。

6.4 带有构造函数和析构函数的Stash

在前几章的例子中，都有一些很明显的函数对应为构造函数和析构函数：**initialize( )**和**cleanup( )**。下面是带有构造函数与析构函数的**Stash**头文件。

```
//: C06:Stash2.h
// With constructors & destructors
#ifndef STASH2_H
#define STASH2_H

class Stash {
  int size;      // Size of each space
  int quantity;  // Number of storage spaces
  int next;      // Next empty space
  // Dynamically allocated array of bytes:
  unsigned char* storage;
  void inflate(int increase);
public:
  Stash(int size);
  ~Stash();
  int add(void* element);
  void* fetch(int index);
  int count();
};
#endif // STASH2_H ///:~
```

下面是实现文件，这里只对**initialize( )**和**cleanup( )**的定义进行了修改，它们分别被构造函数与析构函数代替了。

294

```
//: C06:Stash2.cpp {O}
// Constructors & destructors
#include "Stash2.h"
#include "../require.h"
#include <iostream>
#include <cassert>
using namespace std;
const int increment = 100;

Stash::Stash(int sz) {
  size = sz;
  quantity = 0;
  storage = 0;
  next = 0;
}

int Stash::add(void* element) {
  if(next >= quantity) // Enough space left?
    inflate(increment);
  // Copy element into storage,
  // starting at next empty space:
  int startBytes = next * size;
  unsigned char* e = (unsigned char*)element;
  for(int i = 0; i < size; i++)
    storage[startBytes + i] = e[i];
  next++;
```

```
  return(next - 1); // Index number
}

void* Stash::fetch(int index) {
  require(0 <= index, "Stash::fetch (-)index");
  if(index >= next)
    return 0; // To indicate the end
  // Produce pointer to desired element:
  return &(storage[index * size]);
}

int Stash::count() {
  return next; // Number of elements in CStash
}

void Stash::inflate(int increase) {
  require(increase > 0,
    "Stash::inflate zero or negative increase");
  int newQuantity = quantity + increase;
  int newBytes = newQuantity * size;
  int oldBytes = quantity * size;
  unsigned char* b = new unsigned char[newBytes];
  for(int i = 0; i < oldBytes; i++)
    b[i] = storage[i]; // Copy old to new
  delete [](storage); // Old storage
  storage = b; // Point to new memory
  quantity = newQuantity;
}

Stash::~Stash() {
  if(storage != 0) {
   cout << "freeing storage" << endl;
   delete []storage;
  }
} ///:~
```

295

require.h中的函数是用来监视程序员错误的，代替函数**assert( )**的作用。但是函数**assert( )**对失败操作的输出不及**require.h**的函数有效（关于这一点将在本书后面说明）。

因为**inflate( )**是私有的，所以**require( )**不能正确执行的惟一情况就是：其他成员函数意外地把一些不正确的值传递给了**inflate( )**。如果能够确保这种情况不会发生，那么就考虑删除函数**require( )**，但是在类稳定之前不要删除，当把新的代码加入到类中时，出错的可能性就会存在。由此看来，使用**require( )**的代价很低（通过使用预处理器，有些代码能够被自动删除），这样代码也会具有很好的健壮性。

注意，在下面的测试程序中，**Stash**对象的定义放在紧靠使用对象的地方，对象的初始化通过构造函数的参数列表来实现，而对象的初始化似乎成了对象定义的一部分。

```
//: C06:Stash2Test.cpp
//{L} Stash2
// Constructors & destructors
#include "Stash2.h"
#include "../require.h"
#include <fstream>
#include <iostream>
```

296

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

int main() {
  Stash intStash(sizeof(int));
  for(int i = 0; i < 100; i++)
    intStash.add(&i);
  for(int j = 0; j < intStash.count(); j++)
    cout << "intStash.fetch(" << j << ") = "
         << *(int*)intStash.fetch(j)
         << endl;
  const int bufsize = 80;
  Stash stringStash(sizeof(char) * bufsize);
  ifstream in("Stash2Test.cpp");
  assure(in, " Stash2Test.cpp");
  string line;
  while(getline(in, line))
    stringStash.add((char*)line.c_str());
  int k = 0;
  char* cp;
  while((cp = (char*)stringStash.fetch(k++))!=0)
    cout << "stringStash.fetch(" << k << ") = "
         << cp << endl;
} ///:~
```

再看看**cleanup( )**调用已被取消，但当**intStash**和**stringStash**越出程序块的作用域时，析构函数被自动地调用了。

在**Stash**例子中需要注意的是：仅仅使用了内部类型，它们没有构造函数。如果试图将类对象拷贝到**Stash**中，就会出现很多问题，程序也不会正确执行。标准的**C++**库能够把对象正确地拷贝到使用它的容器中，但是，这是一个相当复杂的过程。在下面的**Stack**例子中，将会
297 看到使用指针可以避免出现这种问题，在后面的章节中将修改**Stash**以便使用指针。

6.5 带有构造函数和析构函数的Stack

重新实现含有构造函数和析构函数的链表（在**Stack**内），看看使用**new**和**delete**时，构造函数和析构函数怎样巧妙地工作。这是修改后的头文件：

```
//: C06:Stack3.h
// With constructors/destructors
#ifndef STACK3_H
#define STACK3_H

class Stack {
  struct Link {
    void* data;
    Link* next;
    Link(void* dat, Link* nxt);
    ~Link();
  }* head;
public:
  Stack();
  ~Stack();
  void push(void* dat);
  void* peek();
```

```
  void* pop();
};
#endif // STACK3_H ///:~
```

不仅**Stack**有构造函数与析构函数，而且被嵌套的类**Link**也有。

```
//: C06:Stack3.cpp {O}
// Constructors/destructors
#include "Stack3.h"
#include "../require.h"
using namespace std;

Stack::Link::Link(void* dat, Link* nxt) {
  data = dat;
  next = nxt;
}

Stack::Link::~Link() { }                                    298
Stack::Stack() { head = 0; }

void Stack::push(void* dat) {
  head = new Link(dat,head);
}

void* Stack::peek() {
  require(head != 0, "Stack empty");
  return head->data;
}

void* Stack::pop() {
  if(head == 0) return 0;
  void* result = head->data;
  Link* oldHead = head;
  head = head->next;
  delete oldHead;
  return result;
}

Stack::~Stack() {
  require(head == 0, "Stack not empty");
} ///:~
```

构造函数**Link::Link( )**只是简单地初始化**data**指针和 **next**指针，所以在**Stack::push( )**中下面这行：

```
head = new Link(dat,head);
```

不仅为一个新的链表分配内存（用第4章中介绍的关键字**new**动态创建对象），而且也巧妙地初始化该对象的指针成员。

读者也许想知道**Link**的析构函数为什么不做任何事情，尤其是为什么不删除**data**指针？这里存在两个问题：在第4章引入**Stack** 的地方，指出了如果**void**指针指向一个对象的话，就不能正确地将其删除（这种情况将在第13章中说明）。但是，除此之外，如果**Link**的析构函数删除**data**指针，**pop( )**将最终返回一个指向被删除对象的指针，很明显，这会引起错误。有时这被看做是所有权（*ownership*）问题：**Link**和**Stack**仅仅存放指针，但它们不负责清除这些 299

指针。这意味着必须非常小心，要知道由谁来负责这种工作。例如：如果不做**pop( )**而删除**Stack**中的所有指针，它们就不会自动被**Stack**的析构函数清除。这种问题不是独立的，它会导致内存泄漏，所以知道谁来负责清除一个对象，对于一个程序是成功还是失败来说是很关键的——这就是为什么如果**Stack**对象销毁时不为空，**Stack::~Stack( )**就会打印出错误信息的原因。

因为分配和清除**Link**对象的实现隐藏在类**Stack**中，它是内部实现的一部分，所以在测试程序中看不到它运行的结果，尽管从**pop( )**返回的指针由我们负责删除。

```
//: C06:Stack3Test.cpp
//{L} Stack3
//{T} Stack3Test.cpp
// Constructors/destructors
#include "Stack3.h"
#include "../require.h"
#include <fstream>
#include <iostream>
#include <string>
using namespace std;

int main(int argc, char* argv[]) {
  requireArgs(argc, 1); // File name is argument
  ifstream in(argv[1]);
  assure(in, argv[1]);
  Stack textlines;
  string line;
  // Read file and store lines in the stack:
  while(getline(in, line))
    textlines.push(new string(line));
  // Pop the lines from the stack and print them:
  string* s;
  while((s = (string*)textlines.pop()) != 0) {
    cout << *s << endl;
    delete s;
  }
} ///:~
```

300

既然这样，**textlines**中的所有行被弹出和删除，但是如果不出现这些操作的话，就会得到由**require( )**带回的信息，这些信息表明这里有内存泄漏。

6.6 集合初始化

顾名思义，集合（*aggregate*）就是多个事物聚集在一起。这个定义包括混合类型的集合：像**struct**和**class**等。数组就是单一类型的集合。

初始化集合往往既冗长又容易出错。而C++中集合*初始化*（*aggregate initialization*）却变得很方便而且很安全。当产生一个集合对象时，要做的只是指定初始值就行了，然后初始化工作就由编译器去承担了。这种指定可以用几种不同的风格，它取决于正在处理的集合类型。但不管是哪种情况，指定的初值都要用大括号括起来。比如一个内部类型的数组可以这样定义：

```
int a[5] = { 1, 2, 3, 4, 5 };
```

如果给出的初始化值多于数组元素的个数，编译器就会给出一条出错信息。但如果给的初始化值少于数组元素的个数，那将会怎么样呢？例如：

```
int b[6] = {0};
```

这时，编译器会把第一个初始化值赋给数组的第一个元素，然后用0赋给其余的元素。注意，如果定义了一个数组而没有给出一列初始值时，编译器并不会去做初始化工作。所以上面的表达式是将一个数组初始化为零的简洁方法，它不需要用一个**for**循环，也避免了“偏移1位”错误（它可能比**for**循环更有效，这取决于编译器）。

数组还有一种叫*自动计数*（*automatic counting*）的快速初始化方法，就是让编译器按初始化值的个数去决定数组的大小：

```
int c[] = { 1, 2, 3, 4 };
```

301

现在，如果决定增加另一个元素到这个数组上，只要增加一个初始化值即可，如果以此建立我们的代码，只需在一处作出修改即可，这样，在修改时出错的机会就减少了。但怎样确定这个数组的大小呢？用表达式**sizeof c/sizeof *c**(整个数组的大小除以第一个元素的大小)即可算出，这样，当数组大小改变时它不需要修改[⊖]。

```
for(int i = 0; i < sizeof c / sizeof *c; i++)
  c[i]++;
```

因为结构也是一种集合类型，所以它们也可以用同样的方式初始化。因为C风格的**struct**的所有成员都是**public**型的，所以它们的值可以直接指定。

```
struct X {
  int i;
  float f;
  char c;
};

X x1 = { 1, 2.2, 'c' };
```

如果有一个这种**struct**的数组，也可以用嵌套的大括号来初始化每一个对象。

```
X x2[3] = { {1, 1.1, 'a'}, {2, 2.2, 'b'} };
```

这里，第三个对象被初始化为零。

如果**struct**中有私有成员（典型的情况就是C++中设计良好的类），或即使所有成员都是公共成员，但有构造函数，情况就不一样了。在上例中，初始值被直接赋给了集合中的每个元素，但构造函数是通过正式的接口来强制初始化的。这里，构造函数必须被调用来完成初始化，因此，如果有一个下面的**struct**类型：

302

```
struct Y {
  float f;
  int i;
  Y(int a);
};
```

⊖ 在本书的第2卷中（可以在http:// www.BruceEckel.com上免费获得），我们将会看到：通过使用模板可以更方便地决定一个数组的大小。

必须指示构造函数调用，最好的方法像下面这样：

```
Y y1[] = { Y(1), Y(2), Y(3) };
```

这样就得到了三个对象和进行了三次构造函数调用。只要有构造函数，无论是所有成员都是公共的**struct**还是一个带私有成员的**class**，所有的初始化工作都必须通过构造函数来完成，即使正在对一个集合初始化。

下面是多构造函数参数的又一个例子：

```
//: C06:Multiarg.cpp
// Multiple constructor arguments
// with aggregate initialization
#include <iostream>
using namespace std;

class Z {
  int i, j;
public:
  Z(int ii, int jj);
  void print();
};

Z::Z(int ii, int jj) {
  i = ii;
  j = jj;
}

void Z::print() {
  cout << "i = " << i << ", j = " << j << endl;
}

int main() {
  Z zz[] = { Z(1,2), Z(3,4), Z(5,6), Z(7,8) };
303 for(int i = 0; i < sizeof zz / sizeof *zz; i++)
    zz[i].print();
} ///:~
```

注意：这看起来就好像对数组中的每个对象都调用显式的构造函数。

6.7 默认构造函数

默认构造函数（*default constructor*）就是不带任何参数的构造函数。默认的构造函数用来创建一个“原型（vanilla）对象”，当编译器需要创建一个对象而又不知任何细节时，默认的构造函数就显得非常重要。比如，有一个前面定义的**struct Y**，并用它来定义对象：

```
Y y2[2] = { Y(1) };
```

编译器就会报告找不到默认的构造函数。数组中的第二个对象想不带参数来创建，在这里编译器就去找默认的构造函数。实际上，如果只是简单地定义了一个**Y**对象的数组：

```
Y y3[7];
```

这样编译的时候，就会出现错误，因为它必须有一个默认的构造函数来初始化数组中的每一个对象。

如果像下面一样单独创建一个对象时，也会出现同样的错误：

```
Y y4;
```

记住，一旦有了一个构造函数，编译器就会确保不管在什么情况下它总会被调用。

默认的构造函数非常重要，所以当（且仅当）在一个结构（**struct** 或**class**）中没有构造函数时，编译器会自动为它创建一个。因此下面例子将会正常运行：

```
//: C06:AutoDefaultConstructor.cpp
// Automatically-generated default constructor

class V {
  int i;  // private
}; // No constructor

int main() {
  V v, v2[10];
} ///:~
```

304

然而，一旦有构造函数而没有默认构造函数，上面的对象定义就会产生一个编译错误。

读者可能会想，由编译器合成的构造函数应该可以做一些智能化的初始化工作，比如把对象的所有内存置零。但事实并非如此。因为这样会增加额外的负担，而且使程序员无法控制。如果想把内存初始化为零，那就得显式地编写默认的构造函数。

尽管编译器会创建一个默认的构造函数，但是编译器合成的构造函数的行为很少是我们期望的。我们应该把这个特征看成是一个安全网，但尽量少用它。一般说来，应该明确地定义自己的构造函数，而不让编译器来完成。

6.8 小结

由C++提供的细致精巧机制应给我们这样一个强烈的暗示：在这个语言中，初始化和清除是多么至关重要的。在Stroustrup设计C++时，他所作的第一个有关C语言效率的观察就是，从很大程度上说，有关程序难题是由于没有适当地初始化变量而引起的。这种错误很难发现。同样的问题也出现在变量的清除上。因为构造函数与析构函数让我们保证正确地初始化和清除对象（编译器将不允许没有调用构造函数与析构函数就直接创建与销毁一个对象），使我们得到了完全的控制与安全。

305

集合的初始化同样如此——它防止犯那种初始化内部数据类型集合时常犯的错误，使代码更简洁。

编码期间的安全性是C++中的一大问题，初始化和清除是这其中的一个重要部分，随着本书的深入学习，可以看到其他的安全性问题。

6.9 练习

部分练习题的答案可以在本书的电子文档 "*Annotated Solution Guide for Thinking in C++*" 中找到，只需支付很少的费用就可以从http://www.BruceEckel.com得到这个电子文档。

6-1 写一个简单的类**Simple**，其构造函数打印一些信息告诉我们它被调用。在函数**main( )**中定义对象。

6-2 在练习1的类中增加一个析构函数，让它打印一些信息告诉我们它被调用。

6-3 修改练习2中的类，让它包含一个**int**成员。修改它的构造函数，让其带一个**int**参数，该参数的值存放在类的**int**成员中，构造函数和析构函数打印该整数的值。这样当对象创建和销毁时我们就可以看到。

6-4 写一个程序演示一下这种情况：当用**goto**跳出一个循环时，析构函数仍然被调用。

6-5 写两个**for**循环，用他们打印出0到10的值。对于第一个，在**for**循环之前定义循环计数器，而对于第二个，在**for**循环控制表达式中定义循环计数器。作为本练习的第二部分，修改第二个**for**循环的标识符使它与第一个循环的计数器的名字相同，编译程序，看看会得到什么结果。

6-6 修改第5章最后的文件**Handle.h**、**Handle.cpp**和**UseHandle.cpp**，以使用构造函数和析构函数。

6-7 使用集合初始化创建一个**double**类型数组，指定其大小，但是并不提供所有的数组元素
306 值，使用**sizeof**确定数组的大小并打印出这个数组，然后通过使用集合初始化创建一个**double**类型数组并且自动地计算数组大小，然后打印这个数组。

6-8 使用集合初始化创建一个**string**类对象数组，创建一个**Stack**用来存储这些字符串，逐步把数组中的元素压入**Stack**中，最后，从**Stack**中弹出并打印它们。

6-9 利用练习3中创建的对象数组演示自动计数和集合初始化。在类中增加一个成员函数来打印一条信息。计算数组的大小，对数组的每个元素，调用新的成员函数。

6-10 创建一个没有构造函数的类，显示我们可以通过默认的构造函数创建对象。现在创
307 建类的一个非默认的构造函数(带一个参数)，编译试试看。解释所发生的情况。

第7章　函数重载与默认参数

能使名字方便使用，是任何程序设计语言的一个重要特征。 309

当我们创建一个对象（一个变量）时，要为存储区取一个名字。函数就是一个操作的名字。通过编制各种名字来描述身边的系统，我们可以产生易于被人们理解和修改的程序。这在很大程度上就像是写文章——其目的是与读者进行交流。

这里就产生了这样一个问题：如何把人类自然语言中有细微差别的概念映射到程序设计语言中。通常，自然语言中同一个词可以代表多种不同的含义，具体含义要依赖上下文来确定。这就是所谓的一词多义——该词被重载（*overload*）了。这点非常有用，特别是对于细微的差别。我们可以说“洗衬衫，洗汽车”。如果非得说成“衬衫－洗衬衫，汽车－洗汽车”，那将是很愚蠢的，就好像听话的人对指定的动作毫无辨别能力一样。人类语言都有内在的冗余，所以即使漏掉几个词，我们仍然可以知道其中的含义。我们不需要惟一标识符——我们可以从上下文中理解它的含义。

然而，大多数程序设计语言要求我们为每个函数设定一个惟一标识符。如果我们想打印三种不同类型的数据：**int**、**char**和**float**，通常不得不创建三个不同的函数名，如**print_int( )**、**print_char( )**和**print_float( )**，这些既增加了我们的编程工作量，也给读者理解程序增加了困难。

在C++中，还有另外一个因素会使函数名重载：构造函数。因为构造函数的名字预先由类的名字确定，所以看上去只能有惟一一个构造函数名。但如果我们想用多种方法来创建一个对象时该怎么办呢？例如假设创建一个类，这个类可以用标准的方法初始化自身，也可以通过从文件中读取信息来初始化，我们需要两个构造函数，一个不带参数（默认构造函数），另一个以一个字符串作为参数，这个字符串是初始化对象的文件的名字。两个都是构造函数，所以它们必须有相同的名字：类名。因此，函数重载对于允许函数同名是必不可少的。在这种情况下，构造函数是与不同的参数类型一起使用的。 310

尽管函数重载对构造函数来说是必须的，但是它仍是一个通用的方便手段，并且可以与任意函数（不仅包括类成员函数）一起使用。另外，函数重载意味着，我们有两个库，它们都有同名的函数，只要它们的参数列表不同就不会发生冲突。我们将在本章中详细讨论所有这些问题。

本章的主题就是方便地使用函数名。函数重载允许多个函数同名，但还有第2种方法使函数调用更方便。如果我们想以不同的方法调用同一个函数，该怎么办呢？当函数有一个长长的参数列表时，而大多数参数每次调用都一样时，书写这样的函数调用会使人厌烦，程序可读性也差。C++中有一个很通用的特征叫做默认参数（*default argument*）。默认参数就是在用户调用一个函数时没有指定参数值而由编译器插入参数值的参数。因此，**f(“hello”)**、**f(“hi”,1)**和**f(“howdy”,2, ‘c’)**可以用来调用同一个函数。它们也可能用来调用三个已重载的函数，但当参数列表相同时，我们通常希望调用同一个函数来完成相同的操作。

函数重载和默认参数实际上并不复杂。当我们学习完本章时，我们就会明白什么时候要用到它们，以及编译、连接时它们是怎样实现的。

7.1 名字修饰

在第4章中介绍了名字修饰（*name decoration*）的概念。在下面的代码中：

```
void f();
class X { void f(); };
```

class X内的函数**f( )**不会与全局的**f( )**发生冲突，编译器用不同的内部名**f( )**（全局函数）和
311 **X::f( )**（成员函数）来区分两个函数。在第4章中，我们建议在函数名前加类名的方法来命名函数，所以编译器使用的内部名字可能就是**_f**和**_X_f**。函数名不仅与类名关系密切，而且还跟其他因素有关。

为什么要这样呢？假设重载了两个函数名：

```
void print(char);
void print(float);
```

无论这两个函数是某个类的成员函数，还是全局函数都无关紧要。如果编译器只使用函数名的域，编译器并不能产生惟一的内部标识符，这两种情况下都得用**_print**结尾。重载函数的思想是让我们用同名的函数，但这些函数的参数列表应该不一样。所以，为了让重载函数正确工作，编译器要用不同的参数类型来修饰不同的函数名。上面的两个在全局范围定义的函数，可能会产生类似于**_print_char**和**_print_float**的内部名。因为，要注意编译器如何为这样的名字修饰没有统一的标准，所以不同的编译器可能会产生不同的内部名（让编译器产生汇编语言代码后就可以看到这个内部名是个什么样子了）。当然，如果想为特定的编译器和连接器购买编译过的库的话，这就会引起错误。但是即使名字修饰有统一的标准，因为编译器用不同的方式产生代码，也还会出现其他问题。

有关函数重载就讲到这里，可以对不同的函数用同样的名字，只要求函数的参数不同。编译器会修饰这些名字、范围和参数来产生内部名以供它和连接器使用。

7.1.1 用返回值重载

读了上面的介绍，我们自然会问："为什么只能通过范围和参数来重载，为什么不能通过返回值呢？"乍一听，似乎完全可行，而且还用内部函数名修饰了返回值，然后就可以用返回值重载了：

312
```
void f();
int f();
```

当编译器能从上下文中惟一确定函数的意思时，如**int x = f( );**这当然没有问题。然而，在C中，总是可以调用一个函数但忽略它的返回值，即调用了函数的副作用（*side effect*），在这种情况下，编译器如何知道调用哪个函数呢？更糟的是，读者怎么知道哪个函数会被调用呢？仅仅靠返回值来重载函数实在过于微妙了，所以在C++中禁止这样做。

7.1.2 类型安全连接

对名字修饰还可以带来一个额外的好处。在C中，如果用户错误地声明了一个函数，或者

更糟糕地，一个函数还没声明就调用了，而编译器则按函数被调用的方式去推断函数的声明。这是一个特别严重的问题。有时这种函数声明是正确的，但如果不正确，就会成为一个很难发现的错误。

在C++中，所有的函数在被使用前都必须事先声明，因此出现上述情况的机会大大减少了。编译器不会自动添加函数声明，所以我们应该包含一个合适的头文件。然而，假如由于某种原因还是错误地声明了一个函数，可能是通过自己手工声明，也可能是包含了一个错误的头文件（也许是一个过期的版本），名字修饰会给我们提供一个安全网，这也就是人们常说的*类型安全连接*（*type-safe linkage*）。

请看下面的几个例子。在第一个文件中，函数定义是：

```
//: C07:Def.cpp {O}
// Function definition
void f(int) {}
///:~
```

在第二个文件中，函数在错误的声明后调用：

```
//: C07:Use.cpp
//{L} Def
// Function misdeclaration
void f(char);

int main() {
//!  f(1); // Causes a linker error
} ///:~
```

313

即使知道函数实际上应该是**f(int)**，但编译器并不知道，因为它被告知——通过一个明确的声明——这个函数是**f(char)**。因此编译成功了，在C中，连接也能成功，但在C++中却不行。因为编译器会修饰这些名字，把它变成了诸如**f_int**之类的名字，而使用的函数则是**f_char**。当连接器试图引用**f_char**时，它只能找到**f_int**，所以它就会报告一条出错信息。这就是类型安全连接。虽然这种问题并不经常出现，但一旦出现就很难发现，尤其是在一个大项目中。这是利用C++编译器查找C语言程序中很隐蔽的错误的一个例子。

7.2 重载的例子

现在回过头来看看前面的例子，这里用重载函数来改写。如前所述，重载的一个很重要的应用是构造函数。可以在下面的**Stash**类中看到这一点。

```
//: C07:Stash3.h
// Function overloading
#ifndef STASH3_H
#define STASH3_H

class Stash {
  int size;      // Size of each space
  int quantity;  // Number of storage spaces
  int next;      // Next empty space
  // Dynamically allocated array of bytes:
  unsigned char* storage;
  void inflate(int increase);
public:
```

314

```
  Stash(int size); // Zero quantity
  Stash(int size, int initQuantity);
  ~Stash();
  int add(void* element);
  void* fetch(int index);
  int count();
};
#endif // STASH3_H ///:~
```

Stash( )的第一个构造函数与前面一样，但第二个有一个**Quantity**参数指明分配内存位置的初始大小。在这个定义中，可以看到**quantity**的内部值与**storage**指针一起被置零。在第二个构造函数中，调用**inflate(initQuantity)**增大**quantity**的值可以指示被分配的存储空间的大小。

```
//: C07:Stash3.cpp {O}
// Function overloading
#include "Stash3.h"
#include "../require.h"
#include <iostream>
#include <cassert>
using namespace std;
const int increment = 100;

Stash::Stash(int sz) {
  size = sz;
  quantity = 0;
  next = 0;
  storage = 0;
}

Stash::Stash(int sz, int initQuantity) {
  size = sz;
  quantity = 0;
  next = 0;
  storage = 0;
  inflate(initQuantity);
}

Stash::~Stash() {
  if(storage != 0) {
    cout << "freeing storage" << endl;
315 delete []storage;
  }
}

int Stash::add(void* element) {
  if(next >= quantity) // Enough space left?
    inflate(increment);
  // Copy element into storage,
  // starting at next empty space:
  int startBytes = next * size;
  unsigned char* e = (unsigned char*)element;
  for(int i = 0; i < size; i++)
    storage[startBytes + i] = e[i];
  next++;
  return(next - 1); // Index number
}
```

```
void* Stash::fetch(int index) {
  require(0 <= index, "Stash::fetch (-)index");
  if(index >= next)
    return 0; // To indicate the end
  // Produce pointer to desired element:
  return &(storage[index * size]);
}

int Stash::count() {
  return next; // Number of elements in CStash
}

void Stash::inflate(int increase) {
  assert(increase >= 0);
  if(increase == 0) return;
  int newQuantity = quantity + increase;
  int newBytes = newQuantity * size;
  int oldBytes = quantity * size;
  unsigned char* b = new unsigned char[newBytes];
  for(int i = 0; i < oldBytes; i++)
    b[i] = storage[i]; // Copy old to new
  delete []storage; // Release old storage
  storage = b; // Point to new memory
  quantity = newQuantity; // Adjust the size
} ///:~
```

当用第一个构造函数时，没有内存分配给**storage**，内存是在第一次调用**add( )**来增加一个 316
对象时分配的，另外，当执行**add( )**时，当前的内存块不够用时也会分配内存。

下面的测试程序中，两个构造函数都会被执行。

```
//: C07:Stash3Test.cpp
//{L} Stash3
// Function overloading
#include "Stash3.h"
#include "../require.h"
#include <fstream>
#include <iostream>
#include <string>
using namespace std;

int main() {
  Stash intStash(sizeof(int));
  for(int i = 0; i < 100; i++)
    intStash.add(&i);
  for(int j = 0; j < intStash.count(); j++)
    cout << "intStash.fetch(" << j << ") = "
         << *(int*)intStash.fetch(j)
         << endl;
  const int bufsize = 80;
  Stash stringStash(sizeof(char) * bufsize, 100);
  ifstream in("Stash3Test.cpp");
  assure(in, "Stash3Test.cpp");
  string line;
  while(getline(in, line))
    stringStash.add((char*)line.c_str());
```

```
  int k = 0;
  char* cp;
  while((cp = (char*)stringStash.fetch(k++))!=0)
    cout << "stringStash.fetch(" << k << ") = "
         << cp << endl;
} ///:~
```

对于**stringStash**调用构造函数，使用了第二个参数。假如知道需要解决的问题的一些情
317 况，就可以为**Stash**选择初始大小。

7.3 联合

正如前面所看到的一样,在C++中，**struct**和**class**惟一的不同之处就在于，**struct**默认为**public**，而**class**默认为**private**。很自然地，也可以让**struct**有构造函数和析构函数。另外，一个**union**（联合）也可以带有构造函数、析构函数、成员函数甚至访问控制。在下面的例子中，还能再一次看到使用重载的好处.

```
//: C07:UnionClass.cpp
// Unions with constructors and member functions
#include<iostream>
using namespace std;

union U {
private: // Access control too!
  int i;
  float f;
public:
  U(int a);
  U(float b);
  ~U();
  int read_int();
  float read_float();
};

U::U(int a) { i = a; }

U::U(float b) { f = b;}

U::~U() { cout << "U::~U()\n"; }

int U::read_int() { return i; }

float U::read_float() { return f; }

int main() {
  U X(12), Y(1.9F);
  cout << X.read_int() << endl;
  cout << Y.read_float() << endl;
318 } ///:~
```

从上面的代码中可以认为:**union**与**class**的惟一不同之处在于存储数据的方式(也就是说在**union**中**int**类型的数据和**float**类型的数据在同一内存区覆盖存放)，但是**union**不能在继承时作为基类使用，从面向对象设计的观点来看，这是一种极大的限制（有关继承将在第14章中讨论）。

尽管成员函数使客户程序员对**union**的访问在一定程度上变得规范，但是，一旦**union**被初始化，仍然不能阻止他们选择错误的元素类型。例如在上面的程序中，即使不恰当，我们也可以写**X.read_float( )**，然而，一个更安全的**union**可以封装在一个类中。在下面的例子中，注意**enum**是如何阐明代码的。以及重载是如何同构造函数一起出现的。

```
//: C07:SuperVar.cpp
// A super-variable
#include <iostream>
using namespace std;

class SuperVar {
  enum {
    character,
    integer,
    floating_point
  } vartype;  // Define one
  union {  // Anonymous union
    char c;
    int i;
    float f;
  };
public:
  SuperVar(char ch);
  SuperVar(int ii);
  SuperVar(float ff);
  void print();
};

SuperVar::SuperVar(char ch) {
  vartype = character;
  c = ch;
}
SuperVar::SuperVar(int ii) {
  vartype = integer;
  i = ii;
}

SuperVar::SuperVar(float ff) {
  vartype = floating_point;
  f = ff;
}

void SuperVar::print() {
  switch (vartype) {
    case character:
      cout << "character: " << c << endl;
      break;
    case integer:
      cout << "integer: " << i << endl;
      break;
    case floating_point:
      cout << "float: " << f << endl;
      break;
  }
}
```

319

```
int main() {
  SuperVar A('c'), B(12), C(1.44F);
  A.print();
  B.print();
  C.print();
} ///:~
```

在上面的代码中，**enum**没有类型名（它是一个没有加标记的枚举），如果想立即定义**enum**的一个实例时，上面的这种做法是可取的。在这里以后没有必要涉及枚举的类型名，所以说，枚举的类型名是可选的，不是必须的。

union没有类型名和标识符。这叫做*匿名联合*（*anonymous union*），为这个**union**创建空间，但并不需要用标识符的方式和以点操作符（'.'）方式访问这个**union**的元素。例如，如果匿名**union**是：

320

```
//: C07:AnonymousUnion.cpp
int main() {
  union {
    int i;
    float f;
  };
  // Access members without using qualifiers:
  i = 12;
  f = 1.22;
} ///:~
```

注意：我们访问一个匿名联合的成员就像访问普通的变量一样。惟一的区别在于：该联合的两个变量占用同一内存空间。如果匿名**union**在文件作用域内（在所有函数和类之外），则它必须被声明为**static**，以使它有内部的连接。

尽管**SuperVar**现在来说是安全的，但是，它的用途却有点值得怀疑，因为使用**union**的首要目的是为了节省空间，而增加**vartype**占用了**union**中很多与数据有关的空间。所以，节省的空间差不多就被抵消了。有两种选择可以使这种模式变得可行。如果**vartype**控制多个**union**实例——假如它们都是相同的数据类型——这样对于这一组实例，就仅仅只需要一个**vartype**，这样就不会占用更多的空间。一个更有效的方法是在所有**vartype**代码的前面加上**#ifdef**，这样就保证了在开发和测试中正确地使用。对于发行的代码，可以消除额外空间和时间开销。

7.4 默认参数

在**Stash3.h**中，比较了**Stash( )**的两个构造函数，它们似乎并没有多大不同，对不对？事实上，第一个构造函数只不过是第二个的一个特例——它的初始**size**为零。在这种情况下创建和管理同一函数的两个不同版本实在是浪费精力。

321

C++通过*默认参数*（*default argument*）提供了一种补救方法。默认参数是在函数声明时就已给定的一个值，如果在调用函数时没有指定这一参数的值，编译器就会自动地插上这个值。在**Stash**的例子中，可以把两个函数：

```
Stash(int size); // Zero quantity
Stash(int size, int initQuantity);
```

用一个函数声明来代替:

```
Stash(int size, int initQuantity = 0);
```

这样，**Stash(int)**定义就简化掉了——所需要的是一个单一的**Stash(int,int)**定义。

现在这两个对象的定义:

```
Stash A(100), B(100, 0);
```

将会产生完全相同的结果。它们将调用同一个构造函数。但对于**A**，它的第二个参数是由编译器在看到第一个参数是**int**而且没有第二个参数时自动加上去的。编译器能看到默认参数，所以它知道应该允许这样的调用，就好像它提供第二个参数一样，而这第二个参数值就是已经告知编译器的默认参数。

默认参数同函数重载一样，给程序员提供了很多方便，它们都使我们可以在不同的场合下使用同一函数名字。不同之处是，利用默认参数，当我们不想亲手提供这些值时，由编译器提供一个默认参数。上面的那个例子就是用默认参数而不用函数重载的一个很好的例子。否则，我们必然面临有几乎同样含义、同样操作的两个或更多的函数。当然，如果函数之间的行为差异较大，用默认参数就不合适了(对于这个问题,我们想知道两个差异较大的函数是否应当有相同的名字)。

在使用默认参数时必须记住两条规则。第一，只有参数列表的后部参数才是可默认的，也就是说，不可以在一个默认参数后面又跟一个非默认的参数。第二，一旦在一个函数调 322
用中开始使用默认参数，那么这个参数后面的所有参数都必须是默认的（这可以从第一条中导出)。

默认参数只能放在函数声明中，通常在一个头文件中。编译器必须在使用该函数之前知道默认值。有时人们为了阅读方便在函数定义处放上一些默认的注释值。如:

```
void fn(int x /* = 0 */) { // ...
```

7.4.1 占位符参数

函数声明时，参数可以没有标识符，当这些不带标识符的参数用做默认参数时，看起来很有意思。可以这样声明:

```
void f(int x, int = 0, float = 1.1);
```

在C++中，在函数定义时，并不一定需要标识符，如:

```
void f(int x, int, float flt) { /* ... */ }
```

在函数体中，**x**和**flt**可以被引用，但中间的这个参数值则不行，因为它没有名字。调用还必须为这个占位符（placeholder）提供一个值，有**f(1)**或**f(1,2,3.0)**。这种语法允许把一个参数用做占位符而不去用它。其目的在于以后可以修改函数定义而不需要修改所有的函数调用。当然，用一个有名字的参数也能达到同样的目的，但如果定义的这个参数在函数体内没有使用它，多数编译器会给出一条警告信息，并认为犯了一个逻辑错误。用这种没有名字的参数，我们就可以防止这种警告产生。

更重要的是，如果开始用了一个函数参数，而后来发现不需要用它，可以将它去掉而不会产生警告错误，而且不需要改动那些调用该函数以前版本的程序代码。 323

7.5 选择重载还是默认参数

函数重载和默认参数都给函数调用提供了方便。然而，有时它也会使人产生困惑：究竟该使用哪一种技术？例如，考虑下面的程序，它用来自动管理内存块。

```
//: C07:Mem.h
#ifndef MEM_H
#define MEM_H
typedef unsigned char byte;

class Mem {
  byte* mem;
  int size;
  void ensureMinSize(int minSize);
public:
  Mem();
  Mem(int sz);
  ~Mem();
  int msize();
  byte* pointer();
  byte* pointer(int minSize);
};
#endif // MEM_H ///:~
```

Mem对象包括一个**byte**块，以确保有足够的存储空间。默认的构造函数不分配任何的空间。第二个构造函数确保**Mem**对象中有**sz**大小的存储区，析构函数释放空间，**msize( )**告诉我们当前**Mem**对象中还有多少字节，**pointer( )**函数产生一个指向存储区起始地址的指针（**Mem**是一个相当底层的工具）。可以有一个重载版本的**pointer( )**函数，用这个函数，客户程序员可以将一个指针指向一块内存。该块内存至少有**minSize**大，有成员函数能够做到这一点。

324 构造函数和**pointer( )**成员函数都使用**private ensureMinSize( )**成员函数来增加内存块的大小（请注意，如果内存块要调整的话，存放**pointer( )**的结果是不安全的）。

下面是这个类的实现：

```
//: C07:Mem.cpp {O}
#include "Mem.h"
#include <cstring>
using namespace std;

Mem::Mem() { mem = 0; size = 0; }

Mem::Mem(int sz) {
  mem = 0;
  size = 0;
  ensureMinSize(sz);
}

Mem::~Mem() { delete []mem; }

int Mem::msize() { return size; }

void Mem::ensureMinSize(int minSize) {
  if(size < minSize) {
    byte* newmem = new byte[minSize];
```

```
    memset(newmem + size, 0, minSize - size);
    memcpy(newmem, mem, size);
    delete []mem;
    mem = newmem;
    size = minSize;
  }
}

byte* Mem::pointer() { return mem; }

byte* Mem::pointer(int minSize) {
  ensureMinSize(minSize);
  return mem;
} ///:~
```

可以看到，只有函数**ensureMinSize( )**负责内存分配，它在第二个构造函数和函数**pointer( )**的第二个重载形式中使用。如果**size**足够大的话，函数**ensureMinSize( )**什么也不需 325
要做，为了使块变得大一些（可能会有这种情况，当使用默认构造函数时，块的大小为零），必须分配新的存储空间，使用标准的C语言库函数**memset( )**把新分配的内存置零，关于这点已在第5章中作了介绍。接着调用标准C语言库函数**memcpy( )**，在这种情况下，把已经存在于**mem**中内容拷贝到**newmem**中（通常用一种有效的方式），最后，删除旧的内存，然后把新的内存和大小赋给适当的成员。

设计**Mem**类的目的是把它作为其他类的一种工具，以简化它们的内存管理（例如，它还可以隐藏由操作系统提供的更复杂的内存管理细节）。下面是一个测试程序，它创建了一个简单的“string”类：

```
//: C07:MemTest.cpp
// Testing the Mem class
//{L} Mem
#include "Mem.h"
#include <cstring>
#include <iostream>
using namespace std;

class MyString {
  Mem* buf;
public:
  MyString();
  MyString(char* str);
  ~MyString();
  void concat(char* str);
  void print(ostream& os);
};

MyString::MyString() {  buf = 0; }

MyString::MyString(char* str) {
  buf = new Mem(strlen(str) + 1);
  strcpy((char*)buf->pointer(), str);
}

void MyString::concat(char* str) {
  if(!buf) buf = new Mem;
```

326

```
    strcat((char*)buf->pointer(
      buf->msize() + strlen(str) + 1), str);
}

void MyString::print(ostream& os) {
  if(!buf) return;
  os << buf->pointer() << endl;
}

MyString::~MyString() { delete buf; }

int main() {
  MyString s("My test string");
  s.print(cout);
  s.concat(" some additional stuff");
  s.print(cout);
  MyString s2;
  s2.concat("Using default constructor");
  s2.print(cout);
} ///:~
```

用这个类，所能做的是创建一个**MyString**，连接文本，打印输出到一个**ostream**中。该类仅仅包含了一个指向**Mem**的指针，但是请注意设置指针为零的默认构造函数和第二个构造函数的区别，第二个构造函数创建了一个**Mem**并把一些数据拷贝给它。使用默认构造函数的好处，就是可以非常便利地创建空值**MyString**对象的大数组，因为每一个对象只是一个指针，默认构造函数的惟一开销是赋零值。当连接数据时，**MyString**的开销才会开始增长。在此情况下，只有**Mem**对象不存在的情况下才会被创建。但是，要是使用默认的构造函数，并且从未连接任何数据，调用析构函数仍然是安全的，因为为零调用的**delete**已经定义，这样它不会试图释放存储空间，或者另外导致一些问题。

如果观察这两个构造函数，乍一看，好像这是默认构造函数最好的候选，然而，如果删除默认构造函数，像下面用一个默认的参数来写另外一个构造函数：

327
```
MyString(char* str = "");
```

将会发现，它能正常工作，但是，我们将会失去宝贵的效率，因为**Mem**对象总是会被创建。为了获得效率，必须修改构造函数：

```
MyString::MyString(char* str) {
  if(!*str) { // Pointing at an empty string
    buf = 0;
    return;
  }
  buf = new Mem(strlen(str) + 1);
  strcpy((char*)buf->pointer(), str);
}
```

这也意味着默认值变成了一个标志：使用非默认值将导致需执行的一块代码被单独分离。这样构造一个小的构造函数，虽然看起来很合理，但是一般会导致错误。如果必须查看默认值而不是把它当做一个普通值的话，这就会意味着实际上是在单个函数体中使用两个不同的有效的函数版本：一个版本用于正常情况，另一个版本用于默认情况。我们也许会把它当成两个完全不同的函数体，由编译器来选择究竟使用哪一个。这种做法会稍微提高程序的效率（但是通常情况下不易察觉），因为额外的参数不会被传递，特定条件下的代码也不会被执行。

更重要的是，我们使用两个完全不相干的函数维护两个函数的代码，而不是使用默认参数把它们组合成一个函数。这样，维护起来就更容易，尤其是当函数特别大时。

另外一方面，考虑一下**Mem**类，如果审视两个构造函数和两个**pointer( )**函数时，可以发现：在两种情况下使用默认参数根本不会导致成员函数定义的改变。因此，类的定义可以如下面所示：

```
//: C07:Mem2.h
#ifndef MEM2_H
#define MEM2_H
typedef unsigned char byte;

class Mem {                                          328
  byte* mem;
  int size;
  void ensureMinSize(int minSize);
public:
  Mem(int sz = 0);
  ~Mem();
  int msize();
  byte* pointer(int minSize = 0);
};
#endif // MEM2_H ///:~
```

注意：调用**ensureMinSize(0)**总是非常有效。

尽管这两种情况都是基于效率问题作出决定的，但是应该注意不要陷入到只考虑效率的境地（这是诱人的）。设计类时，最重要的问题是类的接口（客户程序员可以使用的**public**成员）。如果产生的类容易使用和重用，那说明成功了。要是有必要，总是可以为了效率而作适当的调整。但是，如果程序员过分强调效率的话，设计的类的效果将是可怕的。应该主要关心的是接口清晰，使使用和阅读代码的人易于理解。注意**MemTest.cpp**文件中**MyString**的语法没有变化，不管一个默认的构造函数是否使用，以及效率是高还是低。

7.6 小结

不能把默认参数作为一个标志去决定执行函数的哪一块，这是基本原则。在这种情况下，只要能够，就应该把函数分解成两个或多个重载的函数。一个默认的参数应该是一个在一般情况下放在这个位置的值。这个值出现的可能比其他值要大，所以客户程序员可以忽略它或只在需要改变默认值时才去用它。 329

默认参数的引用是为了使函数调用更容易，特别是当这些函数的许多参数都有特定值时。它不仅使书写函数调用更容易，而且阅读也更方便，尤其是当类的创建者能够制定参数，以便把那些最不可能调整的默认参数放在参数表的最后面时。

默认参数的一个重要应用情况是在开始定义函数时用了一组参数，而使用了一段时间后发现要增加一些参数。通过把这些新增参数都作为默认的参数，就可以保证所有使用这一函数的客户代码不会受到影响。

7.7 练习

部分练习题的答案可以在本书的电子文档 *"Annotated Solution Guide for Thinking in C++"*

中找到，只需支付很小的费用就可以从http://www.BruceEckel.com得到这个电子文档。

7-1 创建一个包含一个**string**对象的**Text**类，来保存一个文件的内容。写两个构造函数：一个是默认的构造函数，另一个构造函数带有一个**string**参数，它是要打开的文件的名字。当使用第二个构造函数时，打开这个文件并把内容读到**string**成员对象中。增加一个成员函数**contents( )**用来返回**string**，以便可以打印。在**main( )**函数中，使用**Text**打开一个文件并打印该文件的内容。

7-2 创建一个**Message**类，其构造函数带有一个**string**型的默认参数。创建一个私有成员**string**，在构造函数中只是简单地把参数**string**赋值给内部的**string**。创建两个重载的成员函数**print( )**：一个不带参数，而只是显示存储在对象中的信息；另一个带有**string**型参数，它将显示该字符串加上对象内部信息。比较这种方法和使用构造函数
330 的方法，看哪种方法更合理？

7-3 确定您的编译器是怎样产生汇编输出代码的，并运行实验以观察名字修饰表。

7-4 创建带有4个成员函数的类，4个成员函数分别带有0、1、2、3个**int**参数。创建**main( )**函数，产生你的类对象并调用每一个成员函数。然后修改类，使它只有一个成员函数，并且都使用默认参数。你的**main( )**函数需要改变吗？

7-5 创建带有两个参数的函数，在**main( )**中调用它。然后让一个参数作为“占位符”（没有标识符），看看**main( )**中的调用是否改变。

7-6 用默认参数修改**Stash3.h**和**Stash3.cpp**中的构造函数，创建两个不同的**Stash**对象来测试构造函数。

7-7 创建一个新的**Stack**类（见第6章），默认构造函数如前面所述，还有第二个构造函数，它的参数是指向对象的指针数组和数组的大小。该构造函数应该遍历数组并把指针压入**Stack**中，用一个**string**数组测试你的程序。

7-8 修改**SuperVar**以便在所有**vartype**代码前有**#ifdef**，描述见前面关于**enum**的章节。让**vartype**成为一个常规的**public**枚举类型（没有实例），修改**print( )**，使得它要求**vartype**参数能告诉它做什么。

7-9 实现**Mem2.h**，确保修改的类仍旧能与**MemTest.cpp**一起工作。

7-10 使用**Mem**类来实现**Stash**。注意：由于该实现是**private**，因此用户看不到，测试代码不必修改。

7-11 在**Mem**类中，增加一个**bool**类型的成员函数**moved( )**。它引用**pointer( )**的结果，告诉指针是否已经移动（由于被重新分配）。写一个**main( )**函数来测试**moved( )**函数。每次需要访问**Mem**中的内存时，是使用像**moved( )**这样的函数好，还是简单地调用
331 **pointer( )**好？

第8章 常　量

常量概念（由关键字**const**表示）是为了使程序员能够在变和不变之间画一条界线。这在C++程序设计项目中提供了安全性和可控性。 333

自从常量概念出现以来，它就有多种不同的用途。与此同时，常量的概念慢慢地渗回到C语言中（在C语言中，它的含义已经改变）。在开始时，所有这些看起来是有点混淆。在本章里，将介绍什么时候、为什么和怎样使用关键字**const**。最后讨论关键字**volatile**，它是**const**的“近亲”（因为它们都关系到变化）并具有完全相同的语法。

const的最初动机是取代预处理器**#defines**来进行值替代。从这以后它曾被用于指针、函数变量、返回类型、类对象以及成员函数。所有这些用法都稍有区别，但它们在概念上是一致的，我们将在以下各节中说明这些用法。

8.1 值替代

当用C语言进行程序设计时，预处理器可以不受限制地建立宏并用它来替代值。因为预处理器只做些文本替代，它既没有类型检查概念，也没有类型检查功能，所以预处理器的值替代会产生一些微小的问题，这些问题在C++中可以通过使用**const**值而避免。

预处理器在C语言中用值替代名字的典型用法是这样的：

```
#define BUFSIZE 100
```

BUFSIZE是一个名字，它只是在预处理期间存在，因此它不占用存储空间且能放在一个头文件里，目的是为使用它的所有编译单元提供一个值。使用值替代而不是使用所谓的“不可思议的数”，这对于支持代码维护是非常重要的。如果代码中用到不可思议的数，读者不仅不清楚这个数字来自哪里，而且也不知道它代表什么。进而，当决定改变一个值时，程序员必须进行手工编辑，而且还不能跟踪以保证没有漏掉其中的一个（或者不小心改变了一个不应该改变的值）。 334

大多数情况，**BUFSIZE**的工作方式与普通变量类似；而且没有类型信息。这就会隐藏一些很难发现的错误。C++用**const**把值替代带进编译器领域来消除这些问题。那么可以这样写：

```
const int bufsize = 100;
```

这样就可以在编译时编译器需要知道这个值的任何地方使用**bufsize**，同时编译器还可以执行*常量折叠*（*constant folding*），也就是说，编译器在编译时可以通过必要的计算把一个复杂的常量表达式通过缩减简单化。这一点在数组定义里显得尤其重要：

```
char buf[bufsize];
```

可以为所有的内部数据类型（**char**、**int**、**float**和**double**型）以及由它们所定义的变量（也可以是类的对象，这将在以后章节里讲到）使用限定符**const**。因为预处理器会引入错误，所以我们应该完全用**const**取代**#define**的值替代。

8.1.1 头文件里的const

要使用**const**而非**#define**，同样必须把**const**定义放进头文件里。这样，通过包含头文件，可把**const**定义单独放在一个地方并把它分配给一个编译单元。C++中的**const**默认为内部连接(*internal linkage*)，也就是说，**const**仅在**const**被定义过的文件里才是可见的，而在连接时不能被其他编译单元看到。当定义一个**const**时，必须赋一个值给它，除非用**extern**作出了清楚的说明:

```
extern const int bufsize;
```

通常C++编译器并不为**const**创建存储空间，相反它把这个定义保存在它的符号表里。但
335 是，上面的**extern**强制进行了存储空间分配（另外还有一些情况，如取一个**const**的地址，也要进行存储空间分配），由于**extern**意味着使用外部连接，因此必须分配存储空间，这也就是说有几个不同的编译单元应当能够引用它，所以它必须有存储空间。

通常情况下，当**extern**不是定义的一部分时，不会分配存储空间。如果使用**const**，那么编译时会进行常量折叠。

当然，想绝对不为任何**const**分配存储是不可能的，尤其对于复杂的结构。在这种情况下，编译器建立存储，这会阻止常量折叠（因为没有办法让编译器确切地知道内存的值是什么——要是知道的话，它也不必分配内存了）。

由于编译器不能完全避免为**const**分配内存，所以**const**的定义必须默认内部连接，即连接仅在特定的编译单元内；否则，由于众多的**const**在多个**cpp**文件内分配存储，容易引起连接错误，连接程序在多个对象文件里看到同样的定义就会“抱怨”。然而，因为**const**默认内部连接，所以连接程序不会跨过编译单元连接那些定义，因此不会有冲突。在大部分场合使用内部数据类型的情况，包括常量表达式，编译都能执行常量折叠。

8.1.2 const的安全性

const的作用不仅限于在常数表达式里代替**#defines**。如果用运行期间产生的值初始化一个变量而且知道在变量生命期内是不变的，则用**const**限定该变量是程序设计中的一个很好的做法。如果偶然试图改变它，编译器会给出出错信息。下面是一个例子:

```
//: C08:Safecons.cpp
// Using const for safety
#include <iostream>
using namespace std;

const int i = 100;  // Typical constant
336 const int j = i + 10; // Value from const expr
long address = (long)&j; // Forces storage
char buf[j + 10]; // Still a const expression

int main() {
  cout << "type a character & CR:";
  const char c = cin.get(); // Can't change
  const char c2 = c + 'a';
  cout << c2;
  // ...
} ///:~
```

我们会发现，i是一个编译期间的**const**，但j是从i中计算出来的。然而，由于i是一个**const**，j的计算值来自一个常数表达式，而它自身也是一个编译期间的**const**。紧接下面的一行需要j的地址，所以迫使编译器给j分配存储空间。即使分配了存储空间，把j值保存在程序的某个地方，由于编译器知道j是**const**，而且知道j值是有效的，因此，这仍不能妨碍在决定数组**buf**的大小时使用j。

在主函数**main**( )里，对于标识符c有另一种**const**，因为其值在编译期间是不知道的。这意味着需要存储空间，而编译器不想保留它的符号表里的任何东西（和C语言的行为一样）。初始化必须在定义点进行，而且一旦初始化，其值就不能改变。我们看到c2由c的值计算出来，也会看到这类常量的作用域与其他任何类型**const**的作用域是一样的——这是对**#define**用法的另一种改进。

就实际来说，如果想让一个值不变，就应该使之成为**const**。这不仅为防止意外的更改提供安全措施，也消除了读存储器和读内存操作，使编译器产生的代码更有效。

8.1.3 集合

const可以用于集合，但必须保证编译器不会复杂到把一个集合保存到它的符号表中，所 337
以必须分配内存。在这种情况下，**const**意味着“不能改变的一块存储空间”。然而，不能在编译期间使用它的值，因为编译器在编译期间不需要知道存储的内容。这样，就能明白下面的代码是非法的：

```
//: C08:Constag.cpp
// Constants and aggregates
const int i[] = { 1, 2, 3, 4 };
//! float f[i[3]]; // Illegal
struct S { int i, j; };
const S s[] = { { 1, 2 }, { 3, 4 } };
//! double d[s[1].j]; // Illegal
int main() {} ///:~
```

在一个数组定义里，编译器必须能产生这样的代码，它们移动栈指针来存储数组。在上面这两种非法定义里，编译器给出“提示”是因为它不能在数组定义里找到一个常数表达式。

8.1.4 与C语言的区别

常量引进是在C++的早期版本中，当时标准C规范正在制定。那时，尽管C委员会决定在C中引入**const**，但是，不知何故，对他们来说，C中**const**的意思是“一个不能被改变的普通变量”，**const**常量总是占用存储而且它的名字是全局符。这样，C编译器不能把**const**看成一个编译期间的常量。在C中，如果写：

```
const int bufsize = 100;
char buf[bufsize];
```

尽管看起来好像做了一件合理的事，但这将得出一个错误。因为**bufsize**占用某块内存，所以C编译器不知道它在编译时的值。在C语言中可以选择这样书写：

```
const int bufsize;
```

这样写在C++中是不对的，而C编译器则把它作为一个声明，指明在别的地方有存储分配。
338 因为C默认**const**是外部连接的，所以这样做是合理的。C++默认**const**是内部连接的，这样，如果在C++中想完成与C中同样的事情，必须用**extern**明确地把连接改成外部连接：

```
extern const int bufsize; // Declaration only
```

这行代码也可用在C语言中。

在C++中，一个**const**不必创建内存空间，而在C中，一个**const**总是需要创建一块内存空间。在C++中，是否为**const**常量创建内存空间依赖于对它如何使用。一般说来，如果一个**const**仅仅用来把一个名字用一个值代替（如同使用**#define**一样），那么该存储空间就不必创建。要是存储空间没有创建的话（这依赖于数据类型的复杂性以及编译器的性能），在进行完数据类型检查之后，为了代码更加有效，值也许会折叠到代码中，这和以前使用**#define**不同。不过，如果取一个**const**的地址（甚至不知不觉地把它传递给一个带引用参数的函数）或者把它定义成**extern**，则会为该**const**创建内存空间。

在C++中，出现在所有函数之外的**const**的作用域是整个文件（也就是它只是在该文件外不可见），也就是说，它默认为内部连接，这和C++中的所有其他默认为外部连接标识符很不一样（也与C中的**const**不一样）。因此，如果在两个不同文件中声明同名的**const**不取它的地址，也不把它定义成**extern**，那么理想的C++编译器就不会为它分配内存空间，而只是简单地把它折叠到代码中。因为**const**在一个文件范围内有效，所以可以把它放在C++头文件中，在连接时不会造成任何冲突。

因为C++中的**const**默认为内部连接，所以不能在一个文件中定义一个**const**，而在另外一个文件中又把它作为**extern**来引用。为了使**const**成为外部连接以便让另外一个文件可以对它引用，必须明确地把它定义成**extern**，如下面这样：

```
extern const int x = 1;
```

339 注意，通过对它进行初始化并指定为**extern**，我们强迫给它分配内存（虽然编译器在这里仍然可以选择常量折叠）。初始化使它成为一个定义而不是一个声明。在C++中的声明：

```
extern const int x;
```

意味着在别处进行了定义（在C中，不一定这样）。现在明白为什么C++要求一个**const**定义时需要初始化：初始化把定义和声明区别开来（在C中，它总是一个定义，所以初始化不是必需的）。当进行了**extern const**声明时，编译器就不能够进行常量折叠了，因为它不知道具体的值。

在C语言中使用限定符**const**不是很有用的，如果希望在常数表达式里（必须在编译期间被求值）使用一个已命名的值，C总是迫使程序员在预处理器里使用**#define**。

8.2 指针

还可以使指针成为**const**。当处理**const**指针时，编译器仍将努力避免存储分配并进行常量折叠，但在这种情况下，这些特征似乎很少有用。更重要的是，如果程序员以后想在程序代码中改变**const**这种指针的使用，编译器将给出通知。这大大增加了安全性。

当使用带有指针的**const**时，有两种选择：**const**修饰指针正指向的对象，或者**const**修饰在指针里存储的地址。这些语法在开始时有点使人混淆，但实践之后就好了。

8.2.1 指向const的指针

正如任何复杂的定义一样，定义指针的技巧是在标识符的开始处读它并从里向外读。**const**修饰“最靠近”它的那个。这样，如果要使正指向的元素不发生改变，得写一个像这样的定义：

340

```
const int* u;
```

从标识符开始，是这样读的：“**u**是一个指针，它指向一个**const int**。”这里不需要初始化，因为**u**可以指向任何标识符（也就是说，它不是一个**const**），但它所指的值是不能被改变的。

这是一个容易混淆的部分。有人可能认为：要想指针本身不变，即包含在指针**u**里的地址不变，可简单地像这样把**const**从**int**的一边移向另一边：

```
int const* v;
```

并非所有的人都很肯定地认为：应该读成“**v**是一个指向**int**的**const**指针”。然而，实际上应读成“**v**是一个指向恰好是**const**的**int**的普通指针”。即**const**又把它自己与**int**结合在一起，效果与前面定义一样。两个定义是一样的，这一点容易使人混淆。为使程序更具有可读性，应该坚持用第一种形式。

8.2.2 const指针

使指针本身成为一个**const**指针，必须把**const**标明的部分放在*的右边，如：

```
int d = 1;
int* const w = &d;
```

现在它读成“**w**是一个指针，这个指针是指向**int**的**const**指针”。因为指针本身现在是**const**指针，编译器要求给它一个初始值，这个值在指针生命期间内不变。然而要改变它所指向的值是可以的，可以写

```
*w = 2;
```

也可以使用下面两种合法形式中的任何一种把一个**const**指针指向一个**const**对象：

341

```
int d = 1;
const int* const x = &d;  // (1)
int const* const x2 = &d; // (2)
```

现在，指针和对象都不能改变。

一些人认为第二种形式的一致性更好，因为**const**总是放在被修饰者的右边。但对于特定的编码风格来讲，程序员应当自己决定哪一种形式更清楚。

下面这个可编译的文件包含上面出现的一些语句

```
//: C08:ConstPointers.cpp
const int* u;
int const* v;
int d = 1;
int* const w = &d;
const int* const x = &d;  // (1)
int const* const x2 = &d; // (2)
int main() {} ///:~
```

8.2.2.1 格式

本书主张：只要可能，一行只定义一个指针，并尽可能在定义时初始化。正因为这一点，才可以把‘*’“附于”数据类型上：

```
int* u = &i;
```

int*本身好像是一个离散类型。这使代码更容易懂，可惜的是，实际上事情并非那样。事实上，‘*’与标识符结合，而不是与类型结合。它可以被放在类型名和标识符之间的任何地方。所以，可以这样做：

```
int *u = &i, v = 0;
```

它建立一个**int* u**和一个非指针**int v**。由于读者时常混淆这一点，因此最好用本书里所用
342 的表示形式（即一行里只定义一个指针）。

8.2.3 赋值和类型检查

C++关于类型检查是非常精细的，这一点也扩展到指针赋值。可以把一个非**const**对象的地址赋给一个**const**指针，因为也许有时不想改变某些可以改变的东西。然而，不能把一个**const**对象的地址赋给一个非**const**指针，因为这样做可能通过被赋值的指针改变这个对象的值。当然，总能用类型转换强制进行这样的赋值，但是，这是一个不好的程序设计习惯，因为这样就打破了对象的**const**属性以及由**const**提供的安全性。例如：

```
//: C08:PointerAssignment.cpp
int d = 1;
const int e = 2;
int* u = &d; // OK -- d not const
//! int* v = &e; // Illegal -- e const
int* w = (int*)&e; // Legal but bad practice
int main() {} ///:~
```

虽然C++有助于防止错误发生，但如果程序员自己打破了这种安全机制，它也是无能为力的。

8.2.3.1 字符数组的字面值

限定词**const**是很严格的，没有强调**const**的地方是字符数组的字面值。也许有人可以写：

```
char* cp = "howdy";
```

编译器将接受它而不报告错误。从技术上讲，这是一个错误，因为字符数组的字面值（这里是“**howdy**”）是被编译器作为一个常量字符数组建立的，所引用该字符数组得到的结果是它在内存里的首地址。修改该字符数组的任何字符都会导致运行时错误，当然，并不是所有的编译器都会做到这一点。

所以字符数组的字面值实际上是常量字符数组。当然，编译器把它们作为非常量看待，
343 这是因为有许多现有的C代码是这样做的。当然，改变字符数组的字面值的做法还未被定义，虽然可能在很多机器上是这样做的。

如果想修改字符串，就要把它放到一个数组中：

```
char cp[] = "howdy";
```

因为编译器常常不强调它们的差别，所以可以不使用后面这种形式，在这一点上已变得无关紧要了。

8.3 函数参数和返回值

用**const**限定函数参数及返回值是常量概念容易引起混淆的另一个地方。如果按值传递对象，对客户来讲，用**const**限定没有意义（它意味着传递的参数在函数里是不能被修改的）。如果按常量返回用户定义类型的一个对象的值，这意味着返回值不能被修改。如果传递并返回地址，**const**将保证该地址内容不会被改变。

8.3.1 传递const值

如果函数参数是按值传递，则可用指定参数是**const**的，如：

```
void f1(const int i) {
  i++; // Illegal -- compile-time error
}
```

这是什么意思呢？这是作了一个约定：变量初值不会被函数**f1( )**改变。然而，由于参数是按值传递的，因此要立即产生原变量的副本，这个约定对客户来说是隐式的。 344

在函数里，**const**有这样的意义：参数不能被改变。所以它其实是函数创建者的工具，而不是函数调用者的工具。

为了不使调用者混淆，在函数内部用**const**限定参数优于在参数表里用**const**限定参数。可以用一个指针来实现，但更好的语法形式是“引用”，这是第11章讨论的主题。简而言之，引用像一个被自动间接引用的常量指针，它的作用是成为对象的别名。为建立一个引用，在定义里使用&。所以，不引起混淆的函数定义应该是这样的：

```
void f2(int ic) {
  const int& i = ic;
  i++;  // Illegal -- compile-time error
}
```

这又会得到一个错误信息，但这时局部对象的常量性（**const**ness）不是函数特征标志的部分；它仅对函数实现有意义，所以它对客户来说是不可见的。

8.3.2 返回const值

对返回值来讲，存在一个类似的道理，即如果一个函数的返回值是一个常量（**const**）：

```
const int g();
```

这就约定了函数框架里的原变量不会被修改。另外。因为这是按值返回的，所以这个变量被制成副本，使得初值不会被返回值所修改。

首先，这使**const**看起来没有什么意义。可以从这个例子中看到：按值返回**const**明显失去作用：

```
//: C08:Constval.cpp
// Returning consts by value
// has no meaning for built-in types
```

345

```
int f3() { return 1; }
const int f4() { return 1; }

int main() {
  const int j = f3(); // Works fine
  int k = f4(); // But this works fine too!
} ///:~
```

对于内部类型来说，按值返回的是否是一个**const**，是无关紧要的，所以按值返回一个内部类型时，应该去掉**const**，从而不使客户程序员混淆。

当处理用户定义的类型时，按值返回常量是很重要的。如果一个函数按值返回一个类对象为**const**时，那么这个函数的返回值不能是一个左值（即它不能被赋值，也不能被修改）。例如：

```
//: C08:ConstReturnValues.cpp
// Constant return by value
// Result cannot be used as an lvalue

class X {
  int i;
public:
  X(int ii = 0);
  void modify();
};

X::X(int ii) { i = ii; }

void X::modify() { i++; }

X f5() {
  return X();
}

const X f6() {
  return X();
}

void f7(X& x) { // Pass by non-const reference
  x.modify();
}

int main() {
  f5() = X(1); // OK -- non-const return value
  f5().modify(); // OK
//!  f7(f5()); // Causes warning
// Causes compile-time errors:
//!  f6() = X(1);
//!  f6().modify();
//!  f7(f6());
} ///:~
```

346

f5( )返回一个非**const X**对象，然而**f6( )**返回一个**const X**对象。仅仅是非**const**返回值能作为一个左值使用，因此，当按值返回一个对象时，如果不让这个对象作为一个左值使用，则使用**const**很重要。

当按值返回一个内部类型时，**const**没有意义的原因是：编译器已经不让它成为一个左值

(因为它总是一个值而不是一个变量)。仅当按值返回用户定义的类型对象时，才会出现上述问题。

函数**f7( )**把它的参数作为一个非**const**引用（*reference*）(C++中另一种处理地址的办法，这是第11章讨论的主题)。从效果上讲，这与取一个非**const**指针一样，只是语法不同。在C++中不能编译通过的原因是会产生一个临时量。

8.3.2.1 临时量

有时候，在求表达式值期间，编译器必须创建临时对象（*temporary object*)。像其他任何对象一样，它们需要存储空间，并且必须能够构造和销毁。区别是从来看不到它们——编译器负责决定它们的去留以及它们存在的细节。但是关于临时量有这样一种情况：它们自动地成为常量。通常接触不到临时对象，改变临时量是错误的，因为这些信息应该是不可得的。编 347
译器使所有的临时量自动地成为**const**，这样当程序员犯那样的错误时，会向他发出错误警告。

在上面的例子中，**f5( )**返回一个非**const X**对象，但是在表达式：

```
f7(f5());
```

中，编译器必须产生一个临时对象来保存**f5( )**的返回值，使得它能传递给**f7( )**。如果**f7( )**的参数是按值传递的话，它能很好地工作，然后在**f7( )**中形成那个临时量的副本，不会对临时对象**X**产生任何影响。但是，如果**f7( )**的参数是按引用传递的，这意味着它取临时对象**X**的地址，因为**f7( )**所带的参数不是按**const**引用传递的，所以它允许对临时对象**X**进行修改。但是编译器知道：一旦表达式计算结束，该临时对象也会不复存在，因此，对临时对象**X**所作的任何修改也将丢失。由于把所有的临时对象自动设为**const**，这种情况导致编译期间错误，因此这种错误不难发现。

然而，下面的表达式是合法的：

```
f5() = X(1);
f5().modify();
```

尽管它们可以编译通过，但实际上存在问题。**f5( )**返回一个**X**对象，而且对编译器来说，要满足上面的表达式，它必须创建临时对象来保存返回值。于是，在这两个表达式中，临时对象也被修改，表达式被编译过之后，临时对象也将被清除。结果，丢失了所有的修改，从而代码可能存在问题——但是编译器不会有任何提示信息。对于用户来说，像这样的表达式很简单，他可以找出问题所在，但是，当事情变得复杂后，就可能在这方面出差错。

类对象常量是怎样保存起来的，将在本章的后面介绍。 348

8.3.3 传递和返回地址

如果传递或返回一个地址（一个指针或一个引用)，客户程序员去取地址并修改其初值是可能的。如果使这个指针或者引用成为**const**，就会阻止这类事的发生，这是非常重要的事情。事实上，无论什么时候传递一个地址给一个函数，都应该尽可能用**const**修饰它。如果不这样做，就不能以**const**指针参数的方式使用这个函数。

是否选择返回一个指向**const**的指针或者引用，取决于想让客户程序员用它干什么。下面这个例子表明了如何使用**const**指针作为函数参数和返回值：

```
//: C08:ConstPointer.cpp
// Constant pointer arg/return
```

```
void t(int*) {}

void u(const int* cip) {
//!  *cip = 2; // Illegal -- modifies value
  int i = *cip; // OK -- copies value
//!  int* ip2 = cip; // Illegal: non-const
}

const char* v() {
  // Returns address of static character array:
  return "result of function v()";
}

const int* const w() {
  static int i;
  return &i;
}

int main() {
  int x = 0;
  int* ip = &x;
  const int* cip = &x;
  t(ip);  // OK
//!  t(cip); // Not OK
  u(ip);  // OK
  u(cip); // Also OK
//!  char* cp = v(); // Not OK
  const char* ccp = v(); // OK
//!  int* ip2 = w(); // Not OK
  const int* const ccip = w(); // OK
  const int* cip2 = w(); // OK
//!  *w() = 1; // Not OK
} ///:~
```

函数**t( )**把一个普通的非**const**指针作为一个参数，而函数**u( )**把一个**const**指针作为参数。在函数**u( )**里，会看到试图修改**const**指针所指的内容是非法的。当然，可以把信息拷贝进一个非**const**变量中。编译器也不允许使用存储在**const**指针里的地址来建立一个非**const**指针。

函数**v( )**和**w( )**测试返回值的语义。函数**v( )**返回一个从字符数组的字面值中建立的**const char***。在编译器建立了它并把它存储在静态存储区之后，这个声明实际上产生这个字符数组的字面值的地址。像前面提到的一样，从技术上讲，这字符数组是一个常量，这个常量由函数**v( )**的返回值正确地表示。

w( )的返回值要求这个指针及这个指针所指向的对象均为常量。像函数**v( )**一样，仅仅因为它是静态的，所以在函数返回后由**w( )**返回的值是有效的。函数不能返回指向局部栈变量的指针，这是因为在函数返回后它们就无效了，而且栈也被清除了。可返回的另一个普通指针是在堆中分配的存储地址，在函数返回后它仍然有效。

在**main( )**中，函数被各种参数测试。函数**t( )**将接受一个非**const**指针参数。但是，如果想传给它一个指向**const**的指针，那么将不能防止**t( )**会丢下这个指针所指的内容不管，所以编译器会给出一个错误信息。函数**u( )**带一个**const**指针，所以它接受两种类型的参数。这样，带**const**指针参数的函数比不带**const**指针参数的函数更具一般性。

正如所期望的一样，函数v( )的返回值只可以被赋给一个const指针。编译器拒绝把函数w( )的返回值赋给一个非const指针，而接受一个const int* const，但令人奇怪的是它也接受一个const int*，这不是与返回类型恰好匹配的。又正如前面所讲的，因为这个值（包含在指针中的地址）正被拷贝，所以自动保持这样的约定：原始变量不能被改变。因此，只有当把const int*const中的第二个const当做一个左值使用时（编译器会阻止这种情况），它才能显示其意义所在。

8.3.3.1 标准参数传递

在C语言中，按值传递是最常见的。当想传递地址时，惟一的选择就是使用指针[⊖]。然而，在C++中这两种方法都受重视。相反，当传递一个参数时，首先选择按引用传递，而且是const引用。对于客户程序员来说，这样做语法与按值传递是一样的，所以不会像使用指针那样的混淆——他们甚至不必考虑指针。对于函数的创建者来说，传递地址总比传递整个类对象更有效，如果按const引用来传递，意味着函数将不改变该地址所指的内容，从客户程序员的观点来看，效果就像按值传递一样（只是更有效）。

由于引用的语法（对于调用者它看起来像按值传递）的原因，把一个临时对象传递给接受const引用的函数是可能的，但不能把一个临时对象传递给接受指针的函数——对于指针，它必须明确地接受地址。所以，按引用传递会产生一个从来不会在C中出现的新的情形：一个总是const的临时变量，它的地址可以被传递给一个函数。这就是为什么当临时变量按引用传递给一个函数时，这个函数的参数必须是const引用的原因。下面的例子说明了这一点： 351

```
//: C08:ConstTemporary.cpp
// Temporaries are const

class X {};

X f() { return X(); } // Return by value

void g1(X&) {} // Pass by non-const reference
void g2(const X&) {} // Pass by const reference

int main() {
  // Error: const temporary created by f():
//!  g1(f());
  // OK: g2 takes a const reference:
  g2(f());
} ///:~
```

函数f( )按值返回类X的一个对象。这意味着当立即取f( )的返回值并把它传递给另外一个函数时（正如g1( )和g2( )函数的调用），将建立一个临时量，该临时量是const。这样，函数g1( )中的调用是错误的，因为g1( )不接受const引用，但是函数g2( )中的调用是正确的。

8.4 类

本节介绍const用于类的两种办法。程序员可能想在一个类里建立一个局部const，将它用在常数表达式里，这个常数表达式在编译期间被求值。然而，const的意思在类里是不同的，所以为了创建类的const数据成员，必须了解这一选择。

⊖ 有些人甚至会说C中的一切都是按值来传递，因为当传递一个指针时，也会得到了一份副本（所以是通过值传递指针的）。但我认为，无论这种看法有多么准确，它都会使这个问题变得更加混乱而不易理解。

还可以使整个对象作为**const**（正如刚刚看到的，编译器总是将临时类对象作为常量）。但是，要保持类对象为常量却比较复杂。编译器能保证一个内部类型为常量，但不能控制类中的复杂性。为了保证一个类对象为常量，引进了**const**成员函数：**const**成员函数只能对于
352 **const**对象调用。

8.4.1 类里的const

常数表达式使用常量的地方之一是在类里。典型的例子是在一个类里建立一个数组，并用**const**代替**#define**设置数组大小以及用于有关数组的计算。数组大小一直隐藏在类里，这样，如果用**size**表示数组大小，就可以把**size**这个名字用在另一个类里而不发生冲突。然而所有的**#define**从定义的地方起就被预处理器看成是全局的，所以用**#define**就不会得到预期的效果。

读者可能认为合乎逻辑的选择是把一个**const**放在类里。但这不会产生预期的结果。在一个类里，**const**又部分地恢复到它在C语言中的含义。它在每个类对象里分配存储并代表一个值，这个值一旦被初始化以后就不能改变。在一个类里使用**const**意味着“在这个对象生命期内，它是一个常量”。然而，对这个常量来讲，每个不同的对象可以含有一个不同的值。

这样，在一个类里建立一个普通的（非**static**的）**const**时，不能给它初值。这个初始化工作必须在构造函数里进行，当然，要在构造函数的某个特别的地方进行。因为**const**必须在建立它的地方被初始化，所以在构造函数的主体里，**const**必定已被初始化了。否则，就只有等待，直到在构造函数主体以后的某个地方给它初始化，这意味着过一会儿才给**const**初始化。当然，无法防止在构造函数主体的不同地方改变**const**的值。

8.4.1.1 构造函数初始化列表

在构造函数里有个专门初始化的地方，这就是*构造函数初始化列表*（*constructor initializer list*），起初用在继承里（继承将在第14章介绍）。构造函数初始化表列表（顾名思义，只出现在构造函数的定义里）是一个出现在函数参数表和冒号后，但在构造函数主体开头的
353 花括号前的“函数调用列表”。这提醒人们，表里的初始化发生在构造函数的任何代码执行之前。这是初始化所有**const**的地方，所以类里的**const**的正确形式是：

```
//: C08:ConstInitialization.cpp
// Initializing const in classes
#include <iostream>
using namespace std;

class Fred {
  const int size;
public:
  Fred(int sz);
  void print();
};

Fred::Fred(int sz) : size(sz) {}
void Fred::print() { cout << size << endl; }

int main() {
  Fred a(1), b(2), c(3);
  a.print(), b.print(), c.print();
} ///:~
```

开始时，上面显示的构造函数初始化列表的形式容易使人们混淆，因为人们不习惯把一个内部类型看成好像也有一个构造函数。

8.4.1.2 内部类型的“构造函数”

随着语言的发展以及人们为使用户定义类型看起来像内部类型一样所作的努力，有时似乎使内部数据类型看起来像用户定义类型更好。在构造函数初始化列表里，可以把一个内部类型看成好像它有一个构造函数，就像下面这样：

```
//: C08:BuiltInTypeConstructors.cpp
#include <iostream>
using namespace std;

class B {
  int i;
public:
  B(int ii);
  void print();
};

B::B(int ii) : i(ii) {}
void B::print() { cout << i << endl; }

int main() {
  B a(1), b(2);
  float pi(3.14159);
  a.print(); b.print();
  cout << pi << endl;
} ///:~
```

354

这在初始化**const**数据成员时尤为关键，因为它们必须进入函数体前被初始化。

我们还可以把这个内部类型的“构造函数”（仅指赋值）扩展为一般的情形，这就是为什么要在上段代码中加入**float pi (3.14159)**定义的原因。

把一个内部类型封装在一个类里以保证用构造函数初始化，这是很有用的。例如，下面是一个**Integer**类：

```
//: C08:EncapsulatingTypes.cpp
#include <iostream>
using namespace std;

class Integer {
  int i;
public:
  Integer(int ii = 0);
  void print();
};

Integer::Integer(int ii) : i(ii) {}
void Integer::print() { cout << i << ' '; }

int main() {
  Integer i[100];
  for(int j = 0; j < 100; j++)
```

```
    i[j].print();
} ///:~
```

355

在**main( )**中的**Integer**数组元素都自动地初始化为零。与**for**循环和**memset( )**相比，这种初始化并不必付出更多的开销。很多编译器可以很容易地把它优化成一个很快的过程。

8.4.2 编译期间类里的常量

上面所使用的**const**是有趣的，也可能很有用，但是它没有解决最初的问题，这就是：如何让一个类有编译期间的常量成员？这就要求使用另外一个关键字**static**，在第10章才会对它进行详尽的介绍。在这种情形下，关键字**static**意味着“不管类的对象被创建多少次，都只有一个实例”，这正是所需要的：类中的一个常量成员，在该类的所有对象中它都一样。因此，一个内部类型的**static const**可以看做一个编译期间的常量。

必须在**static const**定义的地方对它进行初始化。这是在类中使用**static const**的特征之一，也显得有点与众不同。这种情况只会伴随**static const**一起出现：也许更喜欢把它用在其他情况下，但不行，因为所有其他的数据成员也必须在构造函数或其他成员函数里初始化。

下面有一个例子，它说明了在一个类里创建和使用一个叫做**size**的**static const**，这个类表示一个存放字符串指针的栈[⊖]。

```
//: C08:StringStack.cpp
// Using static const to create a
// compile-time constant inside a class
#include <string>
#include <iostream>
using namespace std;

class StringStack {
356   static const int size = 100;
  const string* stack[size];
  int index;
public:
  StringStack();
  void push(const string* s);
  const string* pop();
};

StringStack::StringStack() : index(0) {
  memset(stack, 0, size * sizeof(string*));
}

void StringStack::push(const string* s) {
  if(index < size)
    stack[index++] = s;
}

const string* StringStack::pop() {
  if(index > 0) {
    const string* rv = stack[--index];
    stack[index] = 0;
```

⊖ 在写本书时，并不是所有的编译器都支持该特征。

```
    return rv;
  }
  return 0;
}

string iceCream[] = {
  "pralines & cream",
  "fudge ripple",
  "jamocha almond fudge",
  "wild mountain blackberry",
  "raspberry sorbet",
  "lemon swirl",
  "rocky road",
  "deep chocolate fudge"
};

const int iCsz =
  sizeof iceCream / sizeof *iceCream;

int main() {
  StringStack ss;
  for(int i = 0; i < iCsz; i++)
    ss.push(&iceCream[i]);
  const string* cp;
  while((cp = ss.pop()) != 0)
    cout << *cp << endl;
} ///:~
```

357

因为**size**用来决定数组**stack**的大小，所以，它实际上是一个编译期间常量，但隐藏在类中。

注意**push( )**带有一个**const string***参数，**pop( )**返回一个**const string***，**StringStack**保存**const string***。否则，就不能用**StringStack**存放在**iceCream**中的指针。可是，它阻止程序员做改变包含在**StringStack**中的对象的任何事情。当然，这种限制不是普遍存在的。

8.4.2.1 旧代码中的“enum hack”

在旧版本的C++中，不支持在类中使用**static const**。这意味着**const**对在类中的常量表达式不起作用，不过，人们还是想做到这一点。于是，一个典型的解决办法就是使用不带实例的无标记**enum**（通常称为“enum hack”）。一个枚举在编译期间必须有值，它在类中局部出现，而且它的值对于常量表达式是可以使用的。所以有下面的代码：

```
//: C08:EnumHack.cpp
#include <iostream>
using namespace std;

class Bunch {
  enum { size = 1000 };
  int i[size];
};

int main() {
  cout << "sizeof(Bunch) = " << sizeof(Bunch)
       << ", sizeof(i[1000]) = "
       << sizeof(int[1000]) << endl;
} ///:~
```

358

这里使用的**enum**保证不占用对象的存储空间，编译期间得到枚举值。也可以明确地给枚举元素赋值。如下所示：

```
enum { one = 1, two = 2, three };
```

在一个完整的**enum**类型中，要是没有为枚举元素特别指定值的话，编译器会从最近的值开始计算，例如上面的**three**的值为3。

在上面**StringStack.cpp**中，可以把

```
static const int size = 100;
```

替换为

```
enum { size = 100 };
```

虽然会经常在以前的程序代码里看到使用**enum**技术，但在C++中增加了**static const**特性，正是为了解决这个问题。但是，没有绝对的理由说明一定要优先选择**static const**而尽量不用**enum** hack，本书使用**enum** hack是由于写这本书的时候大多数的编译器都支持这种特性。

8.4.3 const对象和成员函数

可以用**const**限定类成员函数，这是什么意思呢？为了搞清楚这一点，必须首先掌握**const**对象的概念。

用户定义类型和内部类型一样，都可以定义一个**const**对象。例如：

```
const int i = 1;
const blob b(2);
```

这里，**b**是类型**blob**的一个**const**对象。它的构造函数被调用，且其参数为“**2**”。由于编译器强调对象为**const**的，因此它必须保证对象的数据成员在其生命期内不被改变。它可以很容易地保证公有数据不被改变，但是它怎么知道哪些成员函数将会改变数据？它又如何知道哪
359 些成员函数对于**const**对象来说是“安全”的呢？

如果声明一个成员函数为**const**，则等于告诉编译器该成员函数可以为一个**const**对象所调用。一个没有被明确声明为**const**的成员函数被看成是将要修改对象中数据成员的函数，而且编译器不允许它为一个**const**对象所调用。

然而，不能到此为止。仅仅声明一个函数在类定义里是**const**的，还不能保证成员函数按声明的方式去做，所以编译器强迫程序员在定义函数时要重申**const**说明。（**const**已成为函数识别符的一部分，所以编译器和连接程序都要检查**const**。）为确保函数定义的常量性，如果我们改变对象中的任何成员或调用一个非**const**成员函数，编译器就将发出一个出错信息，这样，可以保证声明为**const**的任何成员函数能够按定义方式运行。

要理解声明**const**成员函数的语法，首先注意前面的带**const**的函数声明，它表示函数的返回值是**const**，但这不会产生想要的结果。相反，必须把修饰符**const**放在函数参数表的后面，例如：

```
//: C08:ConstMember.cpp
class X {
  int i;
public:
  X(int ii);
```

```
  int f() const;
};

X::X(int ii) : i(ii) {}
int X::f() const { return i; }

int main() {
  X x1(10);
  const X x2(20);
  x1.f();
  x2.f();
} ///:~
```

关键字**const**必须用同样的方式重复出现在定义里，否则编译器把它看成一个不同的函数，因为**f( )**是一个**const**成员函数，所以不管它试图以何种方式改变**i**或者调用另一个非**const**成员函数，编译器都把它标记成一个错误。 360

一个**const**成员函数调用**const**和非**const**对象是安全的，因此，可以把它看做成员函数的最一般形式（不幸的是，成员函数并不会自动地默认为**const**）。不修改数据成员的任何函数都应该把它们声明为**const**，这样它可以和**const**对象一起使用。

下面是一个比较**const**和非**const**成员函数的例子：

```
//: C08:Quoter.cpp
// Random quote selection
#include <iostream>
#include <cstdlib> // Random number generator
#include <ctime> // To seed random generator
using namespace std;

class Quoter {
  int lastquote;
public:
  Quoter();
  int lastQuote() const;
  const char* quote();
};

Quoter::Quoter(){
  lastquote = -1;
  srand(time(0)); // Seed random number generator
}

int Quoter::lastQuote() const {
  return lastquote;
}

const char* Quoter::quote() {
  static const char* quotes[] = {
    "Are we having fun yet?",
    "Doctors always know best",
    "Is it ... Atomic?",
    "Fear is obscene",
    "There is no scientific evidence "
    "to support the idea "
    "that life is serious",
```

361

```
    "Things that make us happy, make us wise",
  };
  const int qsize = sizeof quotes/sizeof *quotes;
  int qnum = rand() % qsize;
  while(lastquote >= 0 && qnum == lastquote)
    qnum = rand() % qsize;
  return quotes[lastquote = qnum];
}

int main() {
  Quoter q;
  const Quoter cq;
  cq.lastQuote(); // OK
//!  cq.quote(); // Not OK; non const function
  for(int i = 0; i < 20; i++)
    cout << q.quote() << endl;
} ///:~
```

构造函数和析构函数都不是**const**成员函数，因为它们在初始化和清除时，总是对对象作些修改。**quote( )**成员函数也不能是**const**函数，因为它要修改数据成员**lastquote**（请看**return**语句）。而**lastQuote( )**没做修改，所以它可以成为**const**函数，而且也可以被**const**对象**cq**安全地调用。

8.4.3.1 可变的：按位const和按逻辑const

如果想要建立一个**const**成员函数，但仍然想在对象里改变某些数据，这时该怎么办呢？这关系到*按位*（*bitwise*）**const**和*按逻辑*（*logical*）**const**（有时也称为*按成员*（*memberwise*）**const**）的区别。按位**const**意思是对象中的每个字节都是固定的，所以对象的每个位映像从不改变。按逻辑**const**意思是，虽然整个对象从概念上讲是不变的，但是可以以成员为单位改变。当编译器被告知一个对象是**const**对象时，它将绝对保护这个对象按位的常量性。要实现按逻辑**const**的属性，有两种由内部**const**成员函数改变数据成员的方法。

362

第一种方法已成为过去，称为“*强制转换常量性*（*casting away constness*）”。它以相当奇怪的方式执行。取**this**（这个关键字产生当前对象的地址）并把强制转换成指向当前类型对象的指针。看来**this**已经是所需的指针，但是，在**const**成员函数内部，它实际上是一个**const**指针，所以，还应把它强制转换成一个普通指针，这样就可以在那个运算中去掉常量性。下面是一个例子：

```
//: C08:Castaway.cpp
// "Casting away" constness

class Y {
  int i;
public:
  Y();
  void f() const;
};

Y::Y() { i = 0; }

void Y::f() const {
//!  i++; // Error -- const member function
  ((Y*)this)->i++; // OK: cast away const-ness
```

```
  // Better: use C++ explicit cast syntax:
  (const_cast<Y*>(this))->i++;
}

int main() {
  const Y yy;
  yy.f(); // Actually changes it!
} ///:~
```

这种方法是可行的，在过去的程序代码里可以看到这种用法，但这不是首选的技术。问题是：常量性的缺乏隐藏在成员函数的定义中，并且没有来自类接口的线索知道对象的数据实际上被修改，除非用户不能见到源代码（用户必然怀疑常量性被转换了，并寻找这一类型转换）。为了公开这一切，应当在类声明里使用关键字**mutable**，以指定一个特定的数据成员可以在一个**const**对象里被改变。 363

```
//: C08:Mutable.cpp
// The "mutable" keyword

class Z {
  int i;
  mutable int j;
public:
  Z();
  void f() const;
};

Z::Z() : i(0), j(0) {}

void Z::f() const {
//! i++; // Error -- const member function
    j++; // OK: mutable
}

int main() {
  const Z zz;
  zz.f(); // Actually changes it!
} ///:~
```

现在，类用户可从声明里看到哪个成员能够用**const**成员函数进行修改。

8.4.3.2 *只读存储能力*

如果一个对象被定义成**const**对象，它就成为被放进只读存储器（ROM）中的候选者，这经常是嵌入式系统程序设计中要考虑做的重要事情。然而，只建立一个**const**对象是不够的——只读存储能力所需要的条件要严格得多。当然，这个对象还应是按位**const**的，而不是按逻辑**const**的。如果只通过关键字**mutable**实现按逻辑常量化的话，就容易看出这一点。如果在一个**const**成员函数里的**const**被强制转换了，编译器可能检测不到这种情况。另外：

1) **class**或**struct**必须没有用户定义的构造函数或析构函数。

2) 这里不能有基类（将在第14章中谈到），也不能包含有用户定义构造函数或析构函数的成员对象。 364

在只读存储能力类型的**const**对象中的任何部分上，有关写操作的影响没有定义。虽然适当形成式的对象可被放进ROM里，但是目前还没有什么对象需要放进ROM里。

8.5 volatile

volatile的语法与**const**是一样的，但是**volatile**的意思是“在编译器认识的范围外，这个数据可以被改变”。不知何故，环境正在改变数据（可能通过多任务、多线程或者中断处理），所以，**volatile**告诉编译器不要擅自作出有关该数据的任何假定，优化期间尤其如此。

如果编译器说：“我已经把数据读进寄存器，而且再没有与寄存器接触”。一般情况下，它不需要再读这个数据。但是，如果数据是**volatile**修饰的，编译器就不能作出这样的假定，因为这个数据可能被其他进程改变了，它必须重读这个数据而不是优化这个代码来消除通常情况下那些冗余的读操作代码。

就像建立**const**对象一样，程序员也可以建立**volatile**对象，甚至还可以建立**const volatile**对象，这个对象不能被客户程序员改变，但可通过外部的代理程序改变。下面的例子描述了一个类，这个类涉及到通信硬件：

```
//: C08:Volatile.cpp
// The volatile keyword

class Comm {
  const volatile unsigned char byte;
  volatile unsigned char flag;
  enum { bufsize = 100 };
  unsigned char buf[bufsize];
  int index;
public:
365 Comm();
  void isr() volatile;
  char read(int index) const;
};

Comm::Comm() : index(0), byte(0), flag(0) {}

// Only a demo; won't actually work
// as an interrupt service routine:
void Comm::isr() volatile {
  flag = 0;
  buf[index++] = byte;
  // Wrap to beginning of buffer:
  if(index >= bufsize) index = 0;
}

char Comm::read(int index) const {
  if(index < 0 || index >= bufsize)
    return 0;
  return buf[index];
}

int main() {
  volatile Comm Port;
  Port.isr(); // OK
//!  Port.read(0); // Error, read() not volatile
} ///:~
```

就像**const**一样，我们可以对数据成员、成员函数和对象本身使用**volatile**，可以对**volatile**

对象调用**volatile**成员函数。

函数**isr( )**不能像中断服务程序那样使用的原因是：在一个成员函数里，当前对象（**this**）的地址必须被秘密地传递，而中断服务程序ISR一般根本不要参数。为解决这个问题，可以让**isr( )**是静态成员函数，这是第10章讨论的主题。

volatile的语法与**const**是一样的，所以对它们的讨论经常被放在一起。为指明可以选择两个中的任何一个，把它们连在一起通称为*c-v限定词*（*c-v qualifier*）。 366

8.6 小结

关键字**const**能将对象、函数参数、返回值和成员函数定义为常量，并能消除预处理器的值替代而不使预处理器的影响。所有这些都为程序设计提供了又一种非常好的类型检查形式以及安全性。使用所谓的*常量正确性*（*const correctness*）（在任何可能的地方使用**const**）已成为项目的救星。

尽管可以忽视**const**而继续使用旧的C代码习惯，但是它确实有帮助，第11章将改变他们的做法，在第11章中将开始大量使用引用，那时将看到对函数参数使用**const**是多么关键。

8.7 练习

部分练习题的答案可以在本书的电子文档 "*Annotated Solution Guide for Thinking in C++*" 中找到，只需支付很少的费用就可以从http://www.BruceEckel.com得到这个电子文档。

8-1 创建三个**const int**值，把它们加到一起得到一个值用来在一个数组定义中决定该数组的大小。在C中编译一遍相同的代码，看看会出现什么情况（通过使用命令行标记，可以将C++编译器改作为C编译器运行）。

8-2 自行证实C编译器和C++编译器对于**const**的处理是不同的。创建一个全局的**const**并将它用在一个全局的常量表达式中；然后分别用C和C++编译它。

8-3 为所有的内部类型创建**const**定义及其变量。和其他的**const**一起在表达式中使用定义新的**const**。并确保编译正确无误。

8-4 在一个头文件中创建一个**const**定义，包含这个头文件在两个**.cpp**文件中，然后编译这些文件并连接它们。要保证正确无误，再在C环境下试一遍。 367

8-5 创建一个**const**，当程序运行时，通过读时间决定它的值(必须使用标准的头文件**<ctime>**)，然后在这个程序中读时间的第二个值，并赋给**const**，看看会有什么结果。

8-6 创建一个**string**的**const**数组，然后尝试修改**string**数组中的某一个值。

8-7 在一个文件中创建一个**extern const**声明，该文件的**main( )**函数打印**extern const**的值，在另外一个文件中定义**extern const**，然后编译和连接这两个文件。

8-8 使用不同的声明形式创建两个指向**const long**的指针，一个指针指向一个**long**数组。演示能让指针增加和减少，但不能改变它所指向的值。

8-9 写一个指向**double**类型的**const**指针，让它指向**double**数组。显示能改变指针指向的内容，但不能增加或减小指针。

8-10 写一个指向**const**对象的**const**指针。显示只能读指针所指向的值，但不能改变该指针或它所指向的值。

8-11 删除**PointerAssignment.cpp**文件中代码的错误行前的注释，看看编译器会产生什么样的错误。

8-12 创建一个字符数组字面值和一个指向该数组开始点的指针，使用这个指针修改数组中的元素，看看编译器是否会报告出错，应当出错吗？如果没有，为什么会认为出错？

8-13 创建一个函数，它带有一个以**const**值传递的参数，然后在函数体中试图改变该参数。

8-14 创建一个函数，它带有一个按值传递的**float**参数。在函数体中，把**const float&**绑
368 定到函数的参数上，并且从那时起仅仅使用引用，以确保不改变参数。

8-15 修改**ConstReturnValues.cpp**文件，每次删除错误行前的注释，看看编译器会产生什么错误信息。

8-16 修改**ConstPointer.cpp**文件，每次删除错误行前的注释，看看编译器会产生什么错误信息。

8-17 制造文件**ConstPointer.cpp**的新版，名为**ConstReference.cpp**，其中把前者使用的指针用引用代替（也许需要用到第11章中的知识）。

8-18 修改**ConstTemporary.cpp**文件，删除错误行前的注释，看看编译器会产生什么错误信息。

8-19 创建一个包含**const** 和非**const float**成员的类。用构造函数的初始化列表进行初始化。

8-20 创建类**MyString**，它包含一个**string**成员、一个初始化该**string**成员的构造函数以及**print( )**函数。修改**StringStack.cpp**文件，以便让容器保存**MyString**对象，**main( )**函数打印它们。

8-21 创建包含一个**const**成员和一个枚举成员的类。在构造函数的初始化列表中初始化**const**成员，无标记的枚举成员用来决定数组大小。

8-22 在**ConstMember.cpp**文件中，删除成员函数定义前的**const**限定符，但是让**const**限定符出现在声明中，看看会得到何种类型的编译器错误信息。

8-23 创建一个类，它有一个**const**和非**const**成员函数。再创建该类的**cosnt**和非**const**对象，用不同类型的对象调用不同类型的成员函数。

8-24 创建一个类，它有一个**const**和非**const**成员函数。尝试从**const**成员函数中调用非
369 **const**成员函数，看看会得到何种类型的编译器错误消息。

8-25 在**Mutable.cpp**文件中，删除错误行前的注释，看看编译器会产生什么错误信息。

8-26 修改**Quoter.cpp**文件的函数**quote( )**，使它变为**const**成员函数和**lastquote mutable**。

8-27 创建一个类，它有一个**volatile**数据成员，创建一个**volatile**和一个非**volatile**成员函数用于修改**volatile**数据成员。看看编译器会出现什么情况。创建该类的**volatile**和非**volatile**对象，尝试调用**volatlie**和非**volatile**成员函数，看看哪一个调用会成功，哪一个调用不成功，以及编译器会产生什么样的错误信息。

8-28 创建一个具有成员函数**fly( )**的名为**bird**的类和一个不含**fly( )**的名为**rock**的类。建立一个**rock**对象，取它的地址，并把它赋给一个**void***。再取这个**void***，把它赋给一个**bird***（不必使用类型转换），通过指针调用函数**fly( )**。为什么C语言允许通过**void***（而不是类型转换）公开地赋值？这是C语言中的一个“缺陷”吗？不能把它
370 推广到C++中吗？

第9章 内联函数

C++从C中继承的一个重要特征是效率。假如C++的效率显著地低于C的效率，那么就会有很大一批程序员不去使用它。 371

在C中，保持效率的一个方法是使用宏(*macro*)。宏可以不要普通的函数调用代价就可使之看起来像函数调用。宏的实现是用预处理器而不是编译器。预处理器直接用宏代码代替宏调用，所以就没有了参数压栈、生成汇编语言的CALL、返回参数、执行汇编语言的RETURN等的开销。所有的工作由预处理器来完成，因此不用花费什么就具有了程序调用的便利和可读性。

在C++中，使用预处理器宏存在两个问题。第一个问题在C中也存在：宏看起来像一个函数调用，但并不总是这样。这样就隐藏了难以发现的错误。第二个问题是C++特有的：预处理器不允许访问类的成员数据。这意味着预处理器宏不能用作类的成员函数。

为了既保持预处理器宏的效率又增加安全性，而且还能像一般成员函数一样可以在类里访问自如，C++引入了内联函数(*inline function*)。本章将介绍C++中预处理器宏存在的问题、在C++中如何用内联函数解决这些问题以及使用内联函数的方针和内联函数的工作机制。

9.1 预处理器的缺陷

预处理器宏存在问题的关键是我们可能认为预处理器的行为和编译器的行为一样。当然，这是有意使宏在外观上和行为上与函数调用一样，因此容易被混淆。当微妙的差异出现时，问题就出现了。

考虑下面这个简单例子：

```
#define F (x) (x + 1)
```

现在假如有一个如下所示的F调用：

```
F(1)
```
372

预处理器展开它，出现下面不希望的情况：

```
(x) (x + 1)(1)
```

出现这个问题是因为在宏定义中F和括号之间存在空格。当这个空格取消后，调用宏时可以有空格空隙。像下面的调用：

```
F (1)
```

依然可以正确地展开为：

```
(1 + 1)
```

上面的例子虽然非常微不足道，但问题非常明显。当在宏调用中使用表达式作为参数时，真正的问题就出现了。

这里存在两个问题。第一个问题是表达式在宏内展开，所以它们的优先级不同于所期望的优先级。例如:

```
#define FLOOR(x,b) x>=b?0:1
```

现在假如用表达式作参数:

```
if(FLOOR(a&0x0f,0x07)) // ...
```

宏将展开成:

```
if(a&0x0f>=0x07?0:1)
```

因为&的优先级比>=的低，所以宏的展开结果将会使我们惊讶。一旦发现这个问题，可以通过在宏定义内的各个地方使用括弧来解决。(这是创建预处理器宏时使用的好方法。)上面的定义可改写成如下:

```
#define FLOOR(x,b) ((x)>=(b)?0:1)
```

373 然而，发现问题可能很难，我们可能一直认为宏的行为是正确的。在前面没有加括号的版本的例子中，大多数表达式将正确工作，因为>=的优先级比像+、/、--，甚至按位移动操作符的优先级都低。因此，很容易想到它对于所有的表达式都正确，包括那些位逻辑操作符。

前面的问题可以通过谨慎地编程来解决：在宏中将所有的内容都用括号括起来。第二个问题则复杂一些。不像普通函数，每次在宏中使用一个参数，都对这个参数求值。只要使用普通变量调用宏仅，求值就无危险。但假如参数求值有副作用，那么结果可能出乎预料，并肯定不能模仿函数行为。

例如，下面这个宏决定它的参数是否在一定范围:

```
#define BAND(x) (((x)>5 && (x)<10) ? (x) : 0)
```

只要使用一个“普通”参数，宏和真的函数的工作方式非常相似。但只要一松懈并开始相信它是一个真的函数时，问题就出现了。如下所示:

```
//: C09:MacroSideEffects.cpp
#include "../require.h"
#include <fstream>
using namespace std;

#define BAND(x) (((x)>5 && (x)<10) ? (x) : 0)

int main() {
  ofstream out("macro.out");
  assure(out, "macro.out");
  for(int i = 4; i < 11; i++) {
    int a = i;
    out << "a = " << a << endl << '\t';
    out << "BAND(++a)=" << BAND(++a) << endl;
    out << "\t a = " << a << endl;
  }
} ///:~
```

374

注意宏名中所有大写字母的使用。这是一种很有用的做法，因为大写的字母告诉读者这是一个宏而不是一个函数，所以如果出现问题，也可以起到一定的提示作用。

下面是这个程序的输出，它完全不是想从真正的函数期望得到的结果:

```
a = 4
  BAND(++a)=0
   a = 5
a = 5
  BAND(++a)=8
   a = 8
a = 6
  BAND(++a)=9
   a = 9
a = 7
  BAND(++a)=10
   a = 10
a = 8
  BAND(++a)=0
   a = 10
a = 9
  BAND(++a)=0
   a = 11
a = 10
  BAND(++a)=0
   a = 12
```

当**a**等于4时，仅测试了条件表达式第一部分，表达式只求值一次，所以宏调用的副作用是**a**等于5，这是在相同的情况下从普通函数调用所期望得到的。但当数字在值域范围内时，两个表达式都测试，产生两次自增操作。产生这个结果是由于再次对参数操作。一旦数字出了范围，两个条件仍然测试，所以也产生两次自增操作。根据参数不同产生的副作用也不同。 375

很清楚，这不是我们想从看起来像函数调用的宏中所希望得到的行为。在这种情况下，明显的解决方法是设计真正的函数。当然，如果多次调用函数将会增加额外的开销并可能降低效率。不幸的是，问题可能并不总是如此明显。可能不知不觉地得到一个包含混合函数和宏的库函数，所以像这样的问题可能隐藏了一些难以发现的错误。例如，在**cstdio**中的**putc( )**宏可能对它的第二个参数求值两次。这在标准C中作了详细说明。作为宏**toupper( )**不谨慎地执行也会对第二个参数求值多次。如在使用**toupper(*p++)**[⊖]时会产生不希望的结果。

9.1.1 宏和访问

当然，在C中需要对预处理器宏谨慎地编码和使用。要不是因为宏没有成员函数作用域这一要求，我们也会在C++中侥幸成功地使用它。预处理器只是简单地执行字符替代，所以不可能用下面这样或近似的形式写:

```
class X {
  int i;
public:
#define VAL(X::i) // Error
```

另外，这里没有指明正在使用哪个对象。在宏里简直没有办法表示类的范围。由于没有可以取代预处理器宏的方法，程序设计者出于效率考虑，不得不让一些数据成员成为**public**类型，这样就会暴露内部实现并妨碍在这个实现中的改变，从而消除了**private**提供的保护。 376

⊖ 更多的细节参见Andrew Koenig 的著作《*C Traps & Pitfalls*》(Addison-Wesley, 1989).

9.2 内联函数

在解决C++中宏访问**private**类成员的问题过程中，所有和预处理器宏有关的问题也随之排除了。这是通过使宏被编译器控制来实现的。在C++中，宏的概念是作为*内联函数*（*inline function*）来实现的，而内联函数无论从那一方面上说都是真正的函数。内联函数能够像普通函数一样具有我们所有期望的任何行为。惟一不同之处是内联函数在适当的地方像宏一样展开，所以不需要函数调用的开销。因此，应该（几乎）永远不使用宏，只使用内联函数。

任何在类中定义的函数自动地成为内联函数，但也可以在非类的函数前面加上**inline**关键字使之成为内联函数。但为了使之有效，必须使函数体和声明结合在一起，否则，编译器将它作为普通函数对待。因此

```
inline int plusOne(int x);
```

没有任何效果，仅仅只是声明函数（这不一定能够在稍后某个时候得到一个内联定义）。成功的方法如下:

```
inline int plusOne(int x) { return ++x; }
```

注意，编译器将检查函数参数列表使用是否正确，并返回值（进行必要的转换）。这些事情是预处理器无法完成的。假如对于上面的内联函数写成一个预处理器宏的话，将得到不想要的副作用。

一般应该把内联定义放在头文件里。当编译器看到这个定义时，它把函数类型（函数名+返回值）和函数体放到符号表里。当使用函数时，编译器检查以确保调用是正确的且返回值
377 被正确使用，然后将函数调用替换为函数体，因而消除了开销。内联代码的确占用空间，但假如函数较小，这实际上比为了一个普通函数调用而产生的代码（参数压栈和执行CALL）占用的空间还少。

在头文件中，内联函数处于一种特殊状态，因为在头文件中声明该函数，所以必须包含头文件和该函数的定义，这些定义在每个用到该函数的文件中，但是不会出现产生多个定义错误的情况（不过，在任何使用内联函数地方该内联函数的定义都必须是相同的）。

9.2.1 类内部的内联函数

为了定义内联函数，通常必须在函数定义前面放一个**inline**关键字。但这在类内部定义内联函数时并不是必须的。任何在类内部定义的函数自动地成为内联函数。如下例:

```
//: C09:Inline.cpp
// Inlines inside classes
#include <iostream>
#include <string>
using namespace std;

class Point {
  int i, j, k;
public:
  Point(): i(0), j(0), k(0) {}
  Point(int ii, int jj, int kk)
    : i(ii), j(jj), k(kk) {}
  void print(const string& msg = "") const {
```

```
    if(msg.size() != 0) cout << msg << endl;
    cout << "i = " << i << ", "
         << "j = " << j << ", "
         << "k = " << k << endl;
  }
};

int main() {
  Point p, q(1,2,3);
  p.print("value of p");
  q.print("value of q");
} ///:~
```

378

两个构造函数和**print( )**函数都默认为内联函数。注意在**main( )**函数中使用内联函数是自然而然的事。一个函数的逻辑行为必须相同（要不然会出现编译错误），不管它是否是内联函数，我们就会看到，惟一不同之处在于它们的效率不一样。

当然，因为类内部的内联函数节省了在外部定义成员函数的额外步骤，所以我们一定想在类声明内每一处都使用内联函数。但应记住，使用内联函数的目的是减少函数调用的开销。但是，假如函数较大，由于需要在调用函数的每一处重复复制代码，这样将使代码膨胀，在速度方面获得的好处就会减少（惟一可靠的办法就是在程序上试验，看看使用内联函数的效果如何）。

9.2.2 访问函数

在类中内联函数的最重要的使用之一是用做*访问函数*（*access function*）。这是一个小函数，它容许读或修改对象状态——即一个或几个内部变量。从下面的例子中，可以看访问函数为内联函数的原因。

```
//: C09:Access.cpp
// Inline access functions

class Access {
  int i;
public:
  int read() const { return i; }
  void set(int ii) { i = ii; }
};

int main() {
  Access A;
  A.set(100);
  int x = A.read();
} ///:~
```

379

这里，在类的设计者控制下，将类里面状态变量设计为私有，类的使用者就永远不会直接和它们发生联系了。对私有数据成员的所有访问只能通过成员函数接口进行。而且，这种访问是相当有效的。例如对于函数**read( )**，若没用内联函数，对**read( )**调用产生的代码将包括对**this**压栈和执行汇编语句CALL。对于大多数机器，产生的代码将比内联函数产生的代码大一些，执行的时间肯定要长。

不用内联函数，考虑效率的类设计者将忍不住简单地使**i**为公共成员，从而通过让用户直

接访问i来消除开销。从设计的角度看，这是很不好的。因为i将成为公共接口的一部分，所以意味着类设计者决不能修改它。我们将和称为i的一个int类型变量打交道。这是一个问题，因为可能在稍后觉得用一个float变量比用一个int 变量代表状态信息更有用一些，但因为int i是公共接口的一部分，所以不能改变它。同样，想在读或是设置i值时执行加法运算也是不允许的，另一方面，假如总是使用成员函数读和修改一个对象的状态信息，那么就可以满意地修改对象内部一些描述。

另外，使用成员函数控制数据成员的访问允许在成员函数中增加代码以检测数据什么时候改变。这在程序调试时非常有用。如果数据成员是public的，任何人就可以任意改变它的值。

9.2.2.1 访问器和修改器

一些人进一步把访问函数的概念分成访问器（*accessor*）（用于从一个对象读状态信息）
380 和修改器（*mutator*）（用于修改状态信息）。而且，可以用重载函数为访问器和修改器提供相同函数名，如何调用函数来决定是读还是修改状态信息。

```
//: C09:Rectangle.cpp
// Accessors & mutators

class Rectangle {
  int wide, high;
public:
  Rectangle(int w = 0, int h = 0)
    : wide(w), high(h) {}
  int width() const { return wide; } // Read
  void width(int w) { wide = w; } // Set
  int height() const { return high; } // Read
  void height(int h) { high = h; } // Set
};

int main() {
  Rectangle r(19, 47);
  // Change width & height:
  r.height(2 * r.width());
  r.width(2 * r.height());
} ///:~
```

构造函数使用构造函数初始化列表（这在第7章中做了简介，在第14章中将做详细介绍）来初始化wide和high值（对于内部数据类型使用伪构造函数调用形式）。

不能让成员函数名与数据成员名相同，于是我们也许想用下划线作为标识符的第一字符来区分这些数据成员。然而，第一个字符为下划线的标识符是保留的，所以不应该使用它们。

可以选用“get”和“set”来标识访问器和修改器。

```
//: C09:Rectangle2.cpp
// Accessors & mutators with "get" and "set"

class Rectangle {
  int width, height;
public:
  Rectangle(int w = 0, int h = 0)
    : width(w), height(h) {}
381 int getWidth() const { return width; }
  void setWidth(int w) { width = w; }
```

```
  int getHeight() const { return height; }
  void setHeight(int h) { height = h; }
};

int main() {
  Rectangle r(19, 47);
  // Change width & height:
  r.setHeight(2 * r.getWidth());
  r.setWidth(2 * r.getHeight());
} ///:~
```

当然，访问器和修改器对于内部变量来说，不必是简单的管道。有时，它们可以执行一些比较复杂的计算。下面的例子使用标准的C库函数中的时间函数来生成简单的**Time**类：

```
//: C09:Cpptime.h
// A simple time class
#ifndef CPPTIME_H
#define CPPTIME_H
#include <ctime>
#include <cstring>

class Time {
  std::time_t t;
  std::tm local;
  char asciiRep[26];
  unsigned char lflag, aflag;
  void updateLocal() {
    if(!lflag) {
      local = *std::localtime(&t);
      lflag++;
    }
  }
  void updateAscii() {
    if(!aflag) {
      updateLocal();
      std::strcpy(asciiRep,std::asctime(&local));
      aflag++;
    }
  }
public:
  Time() { mark(); }
void mark() {
  lflag = aflag = 0;
  std::time(&t);
}
const char* ascii() {
  updateAscii();
  return asciiRep;
}
// Difference in seconds:
int delta(Time* dt) const {
  return int(std::difftime(t, dt->t));
}
int daylightSavings() {
  updateLocal();
  return local.tm_isdst;
```

382

```
}
int dayOfYear() { // Since January 1
  updateLocal();
  return local.tm_yday;
}
int dayOfWeek() { // Since Sunday
  updateLocal();
  return local.tm_wday;
}
int since1900() { // Years since 1900
  updateLocal();
  return local.tm_year;
}
int month() { // Since January
  updateLocal();
  return local.tm_mon;
}
int dayOfMonth() {
  updateLocal();
  return local.tm_mday;
}
int hour() { // Since midnight, 24-hour clock
  updateLocal();
  return local.tm_hour;
}
int minute() {
  updateLocal();
  return local.tm_min;
}
int second() {
    updateLocal();
    return local.tm_sec;
  }
};
#endif // CPPTIME_H ///:~
```

383

标准C库函数对于时间有多种表示，它们都是类**Time**的一部分。但全部更新它们是没有必要的，所以**time_t t**被用作基本的表示法，**tm local** 和ASCII字符表示法**asciiRep**都有一个标记来显示它们是否已被更新为当前的时间**time_t**。两个私有函数**updateLocal( )**和**updateAscii( )**检查标记，并有条件地执行更新操作。

构造函数调用**mark( )**函数时（用户也可以调用它，强迫对象表示当前时间）也就清除了两个标记，这时当地时间和ASCII表示法是无效的。函数**ascii( )**调用**updateAscii( )**，因为函数**ascii( )**使用静态数据，假如它被调用，则这个静态数据被重写，所以**updateAscaii( )**把标准C库函数的结果拷贝到局部缓冲器里。函数**ascii( )**返回值就是内部缓冲器的地址。

所有以**daylightSavings( )**开始的函数都使用函数**updateLocal( )**，这就使得复合的内联函数变得相当大。这似乎不划算，尤其是考虑到可能不经常调用这些函数。但这不意味着所有的函数都应该用非内联函数。如果想让其他一些函数成为非内联函数的话，也至少让**updateLocal( )**为内联函数，这样它的代码将被复制在所有的非内联函数里，也能消除函数调用时额外的开销。

下面是一个小的测试程序:

```
//: C09:Cpptime.cpp
// Testing a simple time class
#include "Cpptime.h"
#include <iostream>
using namespace std;
int main() {                                                    384
  Time start;
  for(int i = 1; i < 1000; i++) {
    cout << i << ' ';
    if(i%10 == 0) cout << endl;
  }
  Time end;
  cout << endl;
  cout << "start = " << start.ascii();
  cout << "end = " << end.ascii();
  cout << "delta = " << end.delta(&start);
} ///:~
```

在这个例子里，创建了一个**Time**对象，然后执行一些时延动作，接着创建第2个**Time**对象来标记结束时间。这些用于显示开始时间、结束时间和消耗的时间。

9.3 带内联函数的Stash和Stack

引入了内联函数，现在，可以把**Stash**和**Stack**类变得更有效。

```
//: C09:Stash4.h
// Inline functions
#ifndef STASH4_H
#define STASH4_H
#include "../require.h"

class Stash {
  int size;      // Size of each space
  int quantity;  // Number of storage spaces
  int next;      // Next empty space
  // Dynamically allocated array of bytes:
  unsigned char* storage;
  void inflate(int increase);
public:
  Stash(int sz) : size(sz), quantity(0),
    next(0), storage(0) {}
  Stash(int sz, int initQuantity) : size(sz),
    quantity(0), next(0), storage(0) {
    inflate(initQuantity);                                      385
  }
  Stash::~Stash() {
    if(storage != 0)
      delete []storage;
  }
  int add(void* element);
  void* fetch(int index) const {
    require(0 <= index, "Stash::fetch (-)index");
    if(index >= next)
      return 0; // To indicate the end
    // Produce pointer to desired element:
    return &(storage[index * size]);
```

```
  }
  int count() const { return next; }
};
#endif // STASH4_H ///:~
```

很明显，小函数作为内联函数工作是理想的，但要注意：两个最大的函数仍旧保留为非内联函数，因为要是把它们作为内联使用的话，很可能在性能上得不到什么改善。

```
//: C09:Stash4.cpp {O}
#include "Stash4.h"
#include <iostream>
#include <cassert>
using namespace std;
const int increment = 100;

int Stash::add(void* element) {
  if(next >= quantity) // Enough space left?
    inflate(increment);
  // Copy element into storage,
  // starting at next empty space:
  int startBytes = next * size;
  unsigned char* e = (unsigned char*)element;
  for(int i = 0; i < size; i++)
    storage[startBytes + i] = e[i];
  next++;
  return(next - 1); // Index number
}

void Stash::inflate(int increase) {
  assert(increase >= 0);
  if(increase == 0) return;
386 int newQuantity = quantity + increase;
  int newBytes = newQuantity * size;
  int oldBytes = quantity * size;
  unsigned char* b = new unsigned char[newBytes];
  for(int i = 0; i < oldBytes; i++)
    b[i] = storage[i]; // Copy old to new
  delete [](storage); // Release old storage
  storage = b; // Point to new memory
  quantity = newQuantity; // Adjust the size
} ///:~
```

测试程序再一次表明一切都正常运行。

```
//: C09:Stash4Test.cpp
//{L} Stash4
#include "Stash4.h"
#include "../require.h"
#include <fstream>
#include <iostream>
#include <string>
using namespace std;

int main() {
  Stash intStash(sizeof(int));
  for(int i = 0; i < 100; i++)
    intStash.add(&i);
```

```
  for(int j = 0; j < intStash.count(); j++)
    cout << "intStash.fetch(" << j << ") = "
         << *(int*)intStash.fetch(j)
         << endl;
  const int bufsize = 80;
  Stash stringStash(sizeof(char) * bufsize, 100);
  ifstream in("Stash4Test.cpp");
  assure(in, "Stash4Test.cpp");
  string line;
  while(getline(in, line))
    stringStash.add((char*)line.c_str());
  int k = 0;
  char* cp;
  while((cp = (char*)stringStash.fetch(k++))!=0)
    cout << "stringStash.fetch(" << k << ") = "
         << cp << endl;
} ///:~
```

387

这个程序同上面的测试程序相同，所以输出结果也基本一样。

Stack类更好地使用了内联函数。

```
//: C09:Stack4.h
// With inlines
#ifndef STACK4_H
#define STACK4_H
#include "../require.h"

class Stack {
  struct Link {
    void* data;
    Link* next;
    Link(void* dat, Link* nxt):
      data(dat), next(nxt) {}
  }* head;
public:
  Stack() : head(0) {}
  ~Stack() {
    require(head == 0, "Stack not empty");
  }
  void push(void* dat) {
    head = new Link(dat, head);
  }
  void* peek() const {
    return head ? head->data : 0;
  }
  void* pop() {
    if(head == 0) return 0;
    void* result = head->data;
    Link* oldHead = head;
    head = head->next;
    delete oldHead;
    return result;
  }
};
#endif // STACK4_H ///:~
```

388 注意：**Link**析构函数在前面的**Stack**版本中是以空的形式出现的，而在这里被删除了。在**pop( )**中，表达式**delete oldHead**只是释放**Link**使用过的内存（它不销毁**Link**所指向的**data**对象）。

多数内联函数十分精细和明显，特别是对于**Link**尤其如此。甚至把**pop( )**作为内联函数看起来也是合理的，尽管条件表达式或者局部变量对于使用内联函数的好处不明显。这里，函数很小，可以使用内联函数提高效率而无负面影响。

如果所有的函数都是内联函数，那么使用库就会变得相当简单，因为就像在上面的测试程序中所看到的一样，不需要进行库连接（注意并没有**Stack4.cpp**）。

```
//: C09:Stack4Test.cpp
//{T} Stack4Test.cpp
#include "Stack4.h"
#include "../require.h"
#include <fstream>
#include <iostream>
#include <string>
using namespace std;

int main(int argc, char* argv[]) {
  requireArgs(argc, 1); // File name is argument
  ifstream in(argv[1]);
  assure(in, argv[1]);
  Stack textlines;
  string line;
  // Read file and store lines in the stack:
  while(getline(in, line))
    textlines.push(new string(line));
  // Pop the lines from the stack and print them:
  string* s;
  while((s = (string*)textlines.pop()) != 0) {
    cout << *s << endl;
    delete s;
  }
} ///:~
```

有时创建的类都是内联成员函数时，可以把整个类放在头文件中（我在本书中就跨越了
389 这条界线），在程序开发的过程中，这是有益的，尽管编译时可能会花费更多的编译时间。一旦程序稍微稳定后，就可以返回去，在适当的地方把函数改为非成员函数。

9.4 内联函数和编译器

为了理解内联何时有效，应该先理解当编译器遇到一个内联函数时将做什么。对于任何函数，编译器在它的符号表里放入函数类型（即包括名字和参数类型的函数原型及函数的返回类型）。另外，当编译器看到内联函数和对内联函数体的进行分析没有发现错误时，就将对应于函数体的代码也放入符号表。代码是以源程序形式存放还是以编译过的汇编指令形式存放取决于编译器。

当调用一个内联函数时，编译器首先确保调用正确，即所有的参数类型必须满足：要么与函数参数表中的参数类型一样，要么编译器能够将其转换为正确类型，并且返回值在目标表达式里应该是正确类型或可改变为正确类型。当然，编译器为任何类型函数都是这样做的，并且这是与预处理器显著的不同之处，因为预处理器不能检查类型和进行转换。

假如所有的函数类型信息符合调用的上下文的话，内联函数代码就会直接替换函数调用，这消除了调用的开销，也考虑了编译器的进一步优化。假如内联函数也是成员函数，对象的地址(**this**)就会被放入合适的地方，这个动作当然也是预处理器不能完成的。

9.4.1 限制

有两种编译器不能执行内联的情况。在这些情况下，它就像对非内联函数一样，根据内联函数定义和为函数建立存储空间，简单地将其转换为函数的普通形式。假如它必须在多重 390
编译单元里做这些（通常将产生一个多定义错误），连接器就会被告知忽略多重定义。

假如函数太复杂，编译器将不能执行内联。这取决于特定的编译器，但对于大多数编译器这时都会放弃内联方式，这时内联将可能不能提高任何效率。一般地，任何种类的循环都被认为太复杂而不扩展为内联函数。循环在函数里可能比调用要花费更多的时间。假如函数仅由简单语句组成，编译器可能没有任何内联的麻烦，但假如函数有许多语句，调用函数的开销将比执行函数体的开销少多了。记住，每次调用一个大的内联函数，整个函数体就被插入在函数调用的地方，所以很容易使代码膨胀，而程序性能上没有任何显著的改进。(在本书中的一些例子中使用的内联函数可能超过一定合理的内联尺寸。)

假如要显式地或隐式地取函数地址，编译器也不能执行内联。因为这时编译器必须为函数代码分配内存从而产生一个函数的地址。但当地址不需要时，编译器仍将可能内联代码。

内联仅是编译器的一个建议，编译器不会被强迫内联任何代码。一个好的编译器将会内联小的、简单的函数，同时明智地忽略那些太复杂的内联。这将给我们想要的结果——具有宏效率的函数调用的真正的语义学。

9.4.2 向前引用

如果猜想编译器执行内联函数时将会做什么事情，就可能会糊涂地认为限制比实际存在的要多。特别当一个内联函数在类中向前引用一个还没有声明的函数时，看起来好像编译器 391
不能处理。

```
//: C09:EvaluationOrder.cpp
// Inline evaluation order

class Forward {
  int i;
public:
  Forward() : i(0) {}
  // Call to undeclared function:
  int f() const { return g() + 1; }
  int g() const { return i; }
};

int main() {
  Forward frwd;
  frwd.f();
} ///:~
```

函数f( )调用**g( )**，但此时还没有声明**g( )**。这也能正常工作，因为C++语言规定：只有在类声明结束后，其中的内联函数才会被计算。

当然，如果g( )反过来调用f( )，就会产生递归调用，这对于编译器来说太复杂而不能执行内联。(应该在f( )和g( )中做一些测试，使其中一个有界可以退出，否则，递归将是无穷无尽的。)

9.4.3 在构造函数和析构函数里隐藏行为

在构造函数和析构函数中，可能易于认为内联的作用比它实际上更有效。构造函数和析构函数都可能隐藏行为，因为类可以包含子对象，子对象的构造函数和析构函数必须被调用。这些子对象可能是成员对象，或可能由于继承（继承将在第14章中介绍）而存在。下面是一个带成员对象的例子。

```
//: C09:Hidden.cpp
// Hidden activities in inlines
#include <iostream>
using namespace std;

class Member {
  int i, j, k;
public:
  Member(int x = 0) : i(x), j(x), k(x) {}
  ~Member() { cout << "~Member" << endl; }
};

class WithMembers {
  Member q, r, s; // Have constructors
  int i;
public:
  WithMembers(int ii) : i(ii) {} // Trivial?
  ~WithMembers() {
    cout << "~WithMembers" << endl;
  }
};

int main() {
  WithMembers wm(1);
} ///:~
```

Member的构造函数对于内联是足够简单的，它不做什么特别的事情。没有继承和成员对象会引起额外隐藏行为。但是在类**WithMembers**里，内联的构造函数和析构函数看起来似乎很直接和简单，但其实很复杂。成员对象**q**、**r**和**s**的构造函数和析构函数将被自动调用，这些构造函数和析构函数也是内联的，所以它们和普通的成员函数的差别是非常显著的。这并不是意味着应该使构造函数和析构函数定义为非内联的，只是在一些特定的情况下，这样做才是合理的。一般说来，快速地写代码来建立一个程序的初始“轮廓”时，使用内联函数经常是便利的。但假如要考虑效率，内联是值得注意的一个问题。

9.5 减少混乱

在本书里，把类里的内联定义做得简单和精练是非常有用的，因为这样更容易放在一页或一屏里，看起来更方便一些。但Dan Saks[⊖] 指出，在一个真正的工程里，这将造成类接口

⊖ 和Tom Plum 合著了《*C++ Programming Guidelines*》，Plum Hall, 1991.

混乱，因此使类难以使用。他用拉丁文*in situ*（在适当的位置上）来表示定义在类里的成员函数，并主张所有的定义都放在类外面以保持接口清楚。他认为这并不妨碍最优化。假如想优化，那么使用关键字**inline**。使用这个方法，前面（8.2.2节）的例子**Rectangle.cpp**修改如下：

```
//: C09:Noinsitu.cpp
// Removing in situ functions

class Rectangle {
  int width, height;
public:
  Rectangle(int w = 0, int h = 0);
  int getWidth() const;
  void setWidth(int w);
  int getHeight() const;
  void setHeight(int h);
};

inline Rectangle::Rectangle(int w, int h)
  : width(w), height(h) {}

inline int Rectangle::getWidth() const {
  return width;
}

inline void Rectangle::setWidth(int w) {
  width = w;
}

inline int Rectangle::getHeight() const {
  return height;
}

inline void Rectangle::setHeight(int h) {
  height = h;
}
int main() {
  Rectangle r(19, 47);
  // Transpose width & height:
  int iHeight = r.getHeight();
  r.setHeight(r.getWidth());
  r.setWidth(iHeight);
} ///:~
```

394

现在假如想比较一下内联函数与非内联函数的使用效果，可以简单地去掉关键字**inline**。（内联函数通常应该放在头文件里，但非内联函数必须放在它们自己的编译单元里。）假如想把函数放入文件，只用简单的剪切和粘贴操作就可完成。*in situ*函数需要更多的操作，且可能隐藏更多错误。这个方法的另外一个争论是可能总是对于函数定义使用一致的格式化类型，但有些并没有总是以*in situ*函数形式出现。

9.6 预处理器的更多特征

前面说过，我们几乎总是希望使用内联函数代替预处理器宏。然而当需要在标准C预处理器（通过继承也是C++预处理器）里使用3个特殊特征时却是例外：字符串定义、字符串拼接

和标志粘贴。字符串定义在本书的前面已作了介绍，字符串定义的完成是用#指示，它容许取一个标识符并把它转化为字符数组，然而字符串拼接在当两个相邻的字符串没有分隔符时发生，在这种情况下字符串组合在一起。在写调试代码时，这两个特征特别有用。

```
#define DEBUG(x) cout << #x " = " << x << endl
```

上面的这个定义可以打印任何变量的值。也可以得到一个跟踪信息，在此信息里打印出它们执行的语句。

395

```
#define TRACE(s) cerr << #s << endl; s
```

#s将输出语句字符。第2个**s**重申了该语句，所以这个语句被执行。当然，这可能会产生问题，尤其是在一行**for**循环中。

```
for(int i = 0; i < 100; i++)
  TRACE(f(i));
```

因为在**TRACE( )**宏里实际上有两个语句，所以一行**for**循环只执行第一个。解决办法是在宏中用逗号代替分号。

9.6.1 标志粘贴

标志粘贴直接用“**##**”实现，在写代码时是非常有用的。它允许设两个标识符并把它们粘贴在一起自动产生一个新的标识符。例如:

```
#define FIELD(a) char* a##_string; int a##_size
class Record {
  FIELD(one);
  FIELD(two);
  FIELD(three);
  // ...
};
```

每次调用**FIELD( )**宏，将产生一个保存字符数组的标识符和另一个保存字符数组长度的标识符。它不仅易读而且消除了编码出错，使维护更容易。

9.7 改进的错误检查

到目前为止，没有定义**require.h**中的函数却使用了它们（尽管**assert( )**也被用在适当的地方来检查程序错误），现在该定义这个头文件了。在这里使用内联函数是便利的，因为它们允许放在头文件中，这样简化了包的使用过程。只要包含头文件，就不必担心连接一个实现

396 文件。

应该注意异常处理机制（在本书的第2卷有详细的描述）为处理各种错误提供了一种更加有效的方法（特别是对于那些想恢复的错误），而不只是中止程序的运行。异常出现在诸如用户没有为一个文件提供足够的命令行参数，或者文件不能打开时。这时，程序不会继续运行。因此，可以调用标准的C库函数**exit( )**。

下面的头文件将放在本书的根目录中，所以它可以从所有的章节里访问。

```
//: :require.h
// Test for error conditions in programs
// Local "using namespace std" for old compilers
```

```
#ifndef REQUIRE_H
#define REQUIRE_H
#include <cstdio>
#include <cstdlib>
#include <fstream>
#include <string>

inline void require(bool requirement,
  const std::string& msg = "Requirement failed"){
  using namespace std;
  if (!requirement) {
    fputs(msg.c_str(), stderr);
    fputs("\n", stderr);
    exit(1);
  }
}

inline void requireArgs(int argc, int args,
  const std::string& msg =
    "Must use %d arguments") {
  using namespace std;
   if (argc != args + 1) {
     fprintf(stderr, msg.c_str(), args);
     fputs("\n", stderr);
     exit(1);
   }
}
inline void requireMinArgs(int argc, int minArgs,
  const std::string& msg =
    "Must use at least %d arguments") {
  using namespace std;
  if(argc < minArgs + 1) {
    fprintf(stderr, msg.c_str(), minArgs);
    fputs("\n", stderr);
    exit(1);
  }
}

inline void assure(std::ifstream& in,
  const std::string& filename = "") {
  using namespace std;
  if(!in) {
    fprintf(stderr, "Could not open file %s\n",
      filename.c_str());
    exit(1);
  }
}

inline void assure(std::ofstream& out,
  const std::string& filename = "") {
  using namespace std;
  if(!out) {
    fprintf(stderr, "Could not open file %s\n",
      filename.c_str());
    exit(1);
  }
```

397

```
}
#endif // REQUIRE_H ///:~
```

默认值提供合理信息，必要时可以改变。

从上面可以看到，没有使用**char***类型的参数，而是使用了**const string&**参数。这允许把**char***和**string**作为这些函数的参数，一般说来，这样做更有用（在我们自己编码时也可能想这样）。

398 在**requireArgs( )**和 **requireMinArgs( )**的定义中，增加了一个表示命令行中参数数目的参数，因为**argc**包括了总是作为第一个参数的程序名，所以**argc**比实际的命令行参数数目多一。

请注意在每一个函数中局部声明 “**using namespace std**” 的使用。这是因为声明不对时，编译器不会包含**namespace std**中标准的C库函数。这样将不能使用**namespace std**中的函数而导致编译错误。局部声明允许**require.h**同正确的和不正确的库一起工作，它不会为包含了这个头文件的任何人打开**namespace std**。

下面是一个测试**require.h**的简单程序。

```
//: C09:ErrTest.cpp
//{T} ErrTest.cpp
// Testing require.h
#include "../require.h"
#include <fstream>
using namespace std;

int main(int argc, char* argv[]) {
  int i = 1;
  require(i, "value must be nonzero");
  requireArgs(argc, 1);
  requireMinArgs(argc, 1);
  ifstream in(argv[1]);
  assure(in, argv[1]); // Use the file name
  ifstream nofile("nofile.xxx");
  // Fails:
//!  assure(nofile); // The default argument
  ofstream out("tmp.txt");
  assure(out);
} ///:~
```

为了打开文件也许想进一步地在**require.h**中加一个宏。

```
#define IFOPEN(VAR, NAME) \
  ifstream VAR(NAME); \
  assure(VAR, NAME);
```

399 可以像如下使用:

```
IFOPEN(in, argv[1])
```

刚开始，这种做法看起来是吸引人的，因为只要敲很少的代码。它虽然有一定的安全性，但最好还是避免这样做。应该注意：宏看起来像函数，但其行为方式不一样。它实际上创建一个对象（**in**），该对象的作用范围不仅仅在宏内。我们现在可以理解这一点，但是对于程序设计的新手和代码维护人员来说，令他们感到迷惑的就不止这一点。所以，只要有可能就尽量不去使用预编译宏。

9.8 小结

能够隐藏类的底层实现是关键的，因为在以后有可能想修改这一实现。我们可能为了效率这样做，或为了对问题有更好的理解，或因为有些新类变得可用而想在实现里使用这些新类。任何危害实现隐蔽性的东西都会减少语言的灵活性。这样，内联函数就显得非常重要，因为它实际上消除了预处理器宏和伴随的问题。通过用内联函数方式，成员函数可以和预处理器宏一样有效。

当然，内联函数也许会在类定义里被多次使用。因为它更简单，所以程序设计者都会这样做。但这不是什么大问题，因为以后期待程序规模减少时，可以将函数移出内联而不影响它们的功能。程序开发的原则应该是“首先是使它可以工作，然后优化。”

9.9 练习

部分练习题的答案可以在本书的电子文档 *“Annotated Solution Guide for Thinking in C++”* 中找到，只需支付很少的费用就可以从http://www.BruceEckel.com得到这个电子文档。

9-1 写一个使用本章开头出现的**F( )**宏的程序，证明它就像本章中所说的那样不能进行正确地扩展，修改宏并使程序能正确运行。 400

9-2 写一个使用本章开头出现的**FLOOR( )**宏的程序，说明它在什么情况下不能正常运行。

9-3 修改**MacroSideEffects.cpp**，使**BAND( )**能够正常运行。

9-4 创建两个功能相同的函数**f1( )**和**f2( )**，**f1( )**是内联函数，**f2( )**是非内联函数。使用**<ctime>**中的标准C库函数**clock( )**标记这两个函数的开始点和结束点，比较它们看哪一个运行得更快，为了得到有效的数字，也许需要在计时循环中重复调用这两个函数。

9-5 对练习4中的函数代码的复杂性和大小作一下试验，看看对于内联函数和非内联函数在时间的消耗上，能否找到一个平衡点。如果可能，再在不同的编译器上试一试，并注意它们之间的差异。

9-6 证明内联函数默认为内部连接。

9-7 创建一个类，它包含一个整型数组。增加一个内联构造函数和一个内联成员函数**print( )**。内联构造函数使用标准的C库函数**memset( )**初始化对应于构造函数的参数（默认时为零）的数组，内联成员函数**print( )**打印数组所有元素值。

9-8 把第5章中的例子**NestFriend.cpp**中的所有成员函数改成内联函数，并使它们为非*in situ*内联函数，也对于构造函数改造**initalize( )**函数。

9-9 使用内联函数修改第8章中的**StringStack.cpp**。

9-10 创建一个称为**Hue**的**enum**，它包含**red**、**blue**和**yellow**。创建一个**color**类，该类包含一个**Hue**类型的数据成员，其构造函数用参数设置这个数据成员的值。增加一个访问函数用来获取和设置**Hue**这个数据成员的值，注意所有的函数都使用内联函数。 401

9-11 使用访问器和修改器的方法修改练习10中的程序。

9-12 修改程序**Cpptime.cpp**，使它从程序开始运行时开始计时，直到用户按确认(Enter)

键或者回车键（Return）。

9-13 创建一个类，它带有两个内联成员函数，在类中定义的第一个成员函数调用第二个成员函数，而不需要提前声明。写一个主函数创建类的对象并调用第一个成员函数。

9-14 创建一个类**A**，它带有一个能声明自己的内联的默认的构造函数，再创建一个新类**B**，将**A**的一个对象作为**B**的成员，**B**的构造函数也是内联的，创建一个**B**类的对象数组，执行程序看看会出现什么情况。

9-15 从以前的练习的类中创建大量的对象并使用**Time**类来计算非内联构造函数和内联构造函数之间的时间差别（假如有剖析器(profiler)，也试着使用它。）

9-16 写一个带有一个**string**命令行参数的程序，写一个**for**循环，循环每执行一步就去掉**string**的一个字母并使用本章的**DEBUG( )**宏打印**string**。

9-17 正确地修改**TRACE( )**宏，使它成为本章所指定的特定宏，并使它能正确运行。

9-18 修改**FIELD( )**宏，使它含有一个索引（**index**）号，创建一个类，它的成员由一些对**FIELD( )**宏的调用组成，增加一个成员函数，它允许使用索引号查看域，写一个主函数**main( )**测试这个类。

9-19 修改**FIELD( )**宏，使它自动产生对每一个域访问的访问函数（数据应该仍旧是私有的）。创建一个类，它的成员由一些对**FIELD( )**宏的调用组成，写一个主函数**main( )**测试这个类。

9-20 写一个程序，它带两个命令行参数：第一个参数是一个整数，第二个参数是一个文
402 件名，使用**require.h**以确保参数数目正确，并且整数在5到10之间，文件能够被成功地打开。

9-21 写一个使用**IFOPEN( )**宏的程序，用它来打开一个文件并作为一个输入流，注意**ifstream**对象的创建以及它的作用域。

9-22 （高级）看看你的编译器怎样产生汇编代码。创建一个文件，它包含一个很小的函数和**main( )**函数，**main( )**调用这个小函数，分别产生这个小函数是内联和非内联
403 时的汇编代码，证明内联版本比非内联版本的函数调用的开销要小。

第10章 名字控制

创建名字是程序设计过程中一项最基本的活动，当一个项目很大时，它会不可避免地包含大量的名字。 405

C++允许我们对名字的产生和名字的可见性进行控制，包括这些名字的存储位置以及名字的连接。

static这个关键字早在人们知道“重载”这个词的含义之前就在C语言中被重载了，并且在C++中又增加了另外的含义。关于**static**的所有使用最基本的概念是指“位置不变的某个东西”（如“静电”），不管这里是指在内存中的物理位置还是指在文件中的可见性。

在本章里，我们将看到**static**如何控制存储和可见性，还将看到一种通过C++的名字空间特征来控制访问名字的改进方法。我们还将发现怎样使用已经采用C语言编写和编译过的函数。

10.1 来自C语言中的静态元素

在C和C++中，**static**都有两种基本的含义，并且这两种含义经常是互相冲突的：

1) 在固定的地址上进行存储分配，也就是说对象是在一个特殊的*静态数据区*（*static data area*）上创建的，而不是每次函数调用时在堆栈上产生的。这也是静态存储的概念。

2) 对一个特定的编译单位来说是局部的（就像在后面将要看到的，这在C++中局限于类的范围）。这样，**static**控制名字的*可见性*（*visibility*），所以这个名字在这个单元或类之外是不可见的。这也描述了连接的概念，它决定连接器将看到哪些名字。

本节将着重讨论**static**的这两个含义，这些都是从C中继承来的。

10.1.1 函数内部的静态变量

通常，在函数体内定义一个局部变量时，编译器在每次函数调用时使堆栈的指针向下移一个适当的位置，为这些局部变量分配内存。如果这个变量有一个初始化表达式，那么每当程序运行到此处，初始化就被执行。 406

然而，有时想在两次函数调用之间保留一个变量的值，可以通过定义一个全局变量来实现，但这样一来，这个变量就不仅仅只受这个函数的控制。C和C++都允许在函数内部定义一个**static**对象，这个对象将存储在程序的静态数据区中，而不是在堆栈中。这个对象只在函数第一次调用时初始化一次，以后它将在两次函数调用之间保持它的值。比如，下面的函数每次调用时都返回一个字符串中的下一个字符。

```
//: C10:StaticVariablesInfunctions.cpp
#include "../require.h"
#include <iostream>
using namespace std;

char oneChar(const char* charArray = 0) {
  static const char* s;
```

```
  if(charArray) {
    s = charArray;
    return *s;
  }
  else
    require(s, "un-initialized s");
  if(*s == '\0')
    return 0;
  return *s++;
}

char* a = "abcdefghijklmnopqrstuvwxyz";

int main() {
  // oneChar(); // require() fails
  oneChar(a); // Initializes s to a
  char c;
  while((c = oneChar()) != 0)
    cout << c << endl;
} ///:~
```

static char* s在每次**onechar( )**调用时保留它的值，因为它存放在程序的静态数据区而不是存储在函数的堆栈中。当用一个字符指针作参数**(char*)**调用**oneChar( )**时，参数值被赋给**s**，然后返回字符串的第一个字符。以后每次调用**oneChar( )**都不用带参数，函数将使用默认参数
407 **charArray**的默认值**0**，函数就会继续用以前初始化的**s**值取字符，直到它到达字符串的结尾标志——空字符为止，到这时，字符指针就不会再增加了，这样，指针不会越过字符串的末尾。

但是，如果调用**oneChar( )**时没有参数而且**s**以前也没有初始化，那会怎样呢？也许会在定义**s**时提供一个初始值：

```
static char* s = 0;
```

但如果没有为一个内部类型的静态变量提供一个初始值的话，编译器也会确保在程序开始时它被初始化为零（转化为适当的类型），所以在**oneChar( )**中，函数第一次调用时**s**将被赋值为零，这样**if(!s)**后面的程序就会被执行。

上例中**s**的初始化是很简单的，其实对一个静态对象的初始化（与其他对象的初始化一样）可以是任意的常量表达式，常量表达式中可以出现常量及在此之前已声明过的变量和函数。

应该知道：上面的函数很容易产生多线程问题；无论什么时候设计一个包含静态变量的函数时，都应该记住多线程问题。

10.1.1.1 函数内部的静态对象

关于一般的静态变量的规则同样适用于用户自定义的静态对象，而且它同样也必须有初始化操作。但是，零赋值只对内部类型有效，用户自定义类型必须用构造函数来初始化。因此，如果在定义一个静态对象时没有指定构造函数参数，这个类就必须有默认的构造函数。
408 请看下例：

```
//: C10:StaticObjectsInFunctions.cpp
#include <iostream>
using namespace std;

class X {
  int i;
```

```
public:
  X(int ii = 0) : i(ii) {} // Default
  ~X() { cout << "X::~X()" << endl; }
};

void f() {
  static X x1(47);
  static X x2; // Default constructor required
}

int main() {
  f();
} ///:~
```

在函数**f( )**内部定义一个静态的**X**类型的对象，它可以用带参数的构造函数来初始化，也可以用默认构造函数。程序控制第一次转到对象的定义点时，而且只有第一次时，才需要执行构造函数。

10.1.1.2 静态对象的析构函数

静态对象的析构函数（包括静态存储的所有对象，不仅仅是上例中的局部静态对象）在程序从**main( )**中退出时，或者标准的C库函数**exit( )**被调用时才被调用。多数情况下**main( )**函数的结尾也是调用**exit( )**来结束程序的。这意味着在析构函数内部使用**exit( )**是很危险的，因为这样导致了无穷的递归调用。但如果用标准的C库函数**abort( )**来退出程序，静态对象的析构函数并不会被调用。

可以用标准C库函数**atexit( )**来指定当程序跳出**main( )**（或调用**exit( )**）时应执行的操作。在这种情况下，在跳出**main( )**或调用**exit( )**之前，用**atexit( )**注册的函数可以在所有对象的析构函数之前被调用。

409

同普通对象的销毁一样，静态对象的销毁也是按与初始化时相反的顺序进行的。当然只有那些已经被创建的对象才会被销毁。幸运的是，开发工具会记录对象初始化的顺序和那些已被创建的对象 。全局对象总是在**main( )**执行之前被创建，在退出**main( )**时销毁。如果一个包含局部静态对象的函数从未被调用过，那么这个对象的构造函数也就不会执行，这样自然也不会执行析构函数。请看下例:

```
//: C10:StaticDestructors.cpp
// Static object destructors
#include <fstream>
using namespace std;
ofstream out("statdest.out"); // Trace file

class Obj {
  char c; // Identifier
public:
  Obj(char cc) : c(cc) {
    out << "Obj::Obj() for " << c << endl;
  }
  ~Obj() {
    out << "Obj::~Obj() for " << c << endl;
  }
};

Obj a('a'); // Global (static storage)
```

```
// Constructor & destructor always called

void f() {
  static Obj b('b');
}

void g() {
  static Obj c('c');
}

int main() {
  out << "inside main()" << endl;
  f(); // Calls static constructor for b
  // g() not called
410 out << "leaving main()" << endl;
} ///:~
```

在**Obj**中，**char c**的作用就像一个标识符，构造函数和析构函数就可以通过**c**显示出当前正在操作的对象信息。而**Obj a**是一个全局的**Obj**类的对象，所以构造函数总是在**main( )**函数之前就被调用。但函数**f( )**内的**Obj**类的静态对象**b**和函数**g( )**内的静态对象**c**的构造函数只在这些函数被调用时才起作用。

为了说明哪些构造函数与析构函数被调用，在**main( )**中只调用了**f( )**，程序的输出结果为：

```
Obj::Obj() for a
inside main()
Obj::Obj() for b
leaving main()
Obj::~Obj() for b
Obj::~Obj() for a
```

在执行**main( )**函数之前，对象**a**的构造函数即被调用，而**b**的构造函数只是因为**f( )**的调用而调用。当退出**main( )**函数时，所有被创建的对象的析构函数按创建时相反的顺序被调用。这意味着如果**g( )**被调用，对象**b**和**c**的析构函数的调用顺序依赖于**g( )**和**f( )**的调用顺序。

注意跟踪文件**ofstream**的对象**out**也是一个静态对象，因为它定义在所有函数之外，位于静态存储区。它的定义（因为不用**extern**定义）应该出现在文件的一开始，在**out**的任何可能的使用出现之前，这一点很重要，否则就可能在一个对象初始化之前使用它。

在**C++**中，全局静态对象的构造函数是在**main( )**之前调用的，所以现在有了一个在进入**main( )**之前执行一段代码的简单的、可移植的方法，并且可以在退出**main( )**之后用析构函数执行代码。在**C**中要做到这一点，就显得很繁琐，我们将不得不熟悉编译器开发商的汇编语言
411 的开始代码。

10.1.2 控制连接

一般情况下，在文件作用域（*file scope*）内的所有名字（即不嵌套在类或函数中的名字）对程序中的所有翻译单元来说都是可见的。这就是所谓的外部连接（*external linkage*），因为在连接时这个名字对连接器来说是可见的，对单独的翻译单元来说，它是外部的。全局变量和普通函数都有外部连接。

有时可能想限制一个名字的可见性。想让一个变量在文件范围内是可见的，这样这个文件中的所有函数都可以使用它，但不想让这个文件之外的函数看到或访问该变量，或不想这

个变量的名字与外部的标识符相冲突。

在文件作用域内，一个被明确声明为**static**的对象或函数的名字对翻译单元（用本书的术语来说也就是出现声明的**.cpp**文件）来说是局部于该单元的。这些名字有内部连接（*internal linkage*）。这意味着可以在其他的翻译单元中使用同样的名字，而不会发生名字冲突。

内部连接的一个好处是这个名字可以放在一个头文件中而不用担心连接时发生冲突。那些通常放在头文件里的名字，如常量、内联函数，在默认情况下都是内部连接的（当然常量只有在C++中默认情况下是内部连接的，在C中它默认为外部连接）。注意连接只引用那些在连接/装载期间有地址的成员，因此类声明和局部变量并不连接。

10.1.2.1 冲突问题

下面例子说明了**static**的两个含义是怎样彼此交叉的。所有的全局对象都是隐含为静态存储的，所以如果定义（在文件作用域）

```
int a = 0;
```

则**a**被存储在程序的静态数据区，在进入**main( )**函数之前，**a**即已初始化了。另外，**a**对所有的翻译单元都是全局可见的。用可见性术语来讲，**static**（只在翻译单元内可见）的反义是 412
extern，它明确地声明了这个名字对所有的翻译单元都是可见的。所以上面的定义和下面的定义是相同的。

```
extern int a = 0;
```

但如果这样定义：

```
static int a = 0;
```

只不过改变了**a**的可见性，现在**a**成了一个内部连接，但存储类型没有改变——对象总是驻留在静态数据区，而不管是**static**还是**extern**。

一旦进入局部变量，**static**就不会再改变变量的可见性（这时**extern**是没有意义的），而只是改变变量的存储类型。

如果把局部变量声明为**extern**，这意味着某处已经存在一个存储区（所以该变量对函数来说实际上是全局的），请看下面的例子。

```
//: C10:LocalExtern.cpp
//{L} LocalExtern2
#include <iostream>

int main() {
  extern int i;
  std::cout << i;
} ///:~

//: C10:LocalExtern2.cpp {O}
int i = 5;
///:~
```

对函数名（非成员函数），**static**和**extern**只会改变它们的可见性，所以如果说：

```
extern void f();
```

它和没有修饰时的声明是一样的：

```
void f();
```

413 如果定义:

```
static void f();
```

它意味着**f( )**只在本翻译单元内是可见的，这有时称作文件静态（*file static*）。

10.1.3 其他存储类型说明符

我们会看到**static**和**extern**用得很普遍。另外还有用得较少的两个存储类型说明符。一个是**auto**，人们几乎不用它，因为它告诉编译器这是一个局部变量。**auto**是“automatic”的缩写，它指明编译器自动为该变量分配存储空间的方法。实际上编译器总是可以从变量定义时的上下文中判断出这是一个局部变量，所以**auto**是多余的。

还有一个是**register**，它说明的也是局部（**auto**）变量，但它告诉编译器这个特殊的变量要经常用到，所以编译器应该尽可能地让它保存在寄存器中。它用于优化代码。但各种编译器对这种类型的变量处理方式也不尽相同，它们有时会忽略这种存储类型的指定。一般，如果要用到这个变量的地址，**register**指定符通常都会被忽略。应该避免用**register**类型，因为编译器在优化代码方面通常比我们做得更好。

10.2 名字空间

虽然名字可以嵌套在类中，但全局函数、全局变量以及类的名字还是在同一个全局名字空间中。虽然**static**关键字可以使变量和函数实行内部连接（使它们文件静态），从而做到一定的控制。但在一个大项目中，如果对全局的名字空间缺乏控制就会引起很多问题。为了解决这些问题，开发商常常使用冗长、难懂的名字，以使冲突减少，但这样我们不得不一个一个地敲这些名字（**typedef**常常用来简化这些名字）。但这不是一个很好的解决方法。

可以用C++的*名字空间*（*namespace*）特征，把一个全局名字空间分成多个可管理的小空
414 间。关键字**namespace**，如同**class**、**struct**、**enum**和**union**一样，把它们的成员的名字放到了不同的空间中去,尽管其他的关键字有其他的目的，但**namespace**惟一的目的是产生一个新的名字空间。

10.2.1 创建一个名字空间

创建一个名字空间与创建一个类非常相似:

```
//: C10:MyLib.cpp
namespace MyLib {
  // Declarations
}
int main() {} ///:~
```

这就产生了一个新的名字空间，其中包含了各种声明。然而，**namespace**与**class**、**struct**、**union** 和**enum**有着明显的区别:

- **namespace**只能在全局范围内定义，但它们之间可以互相嵌套。
- 在**namespace**定义的结尾，右花括号的后面不必跟一个分号。
- 一个**namespace**可以在多个头文件中用一个标识符来定义，就好像重复定义一个类一样。

```
//: C10:Header1.h
#ifndef HEADER1_H
```

```
#define HEADER1_H
namespace MyLib {
  extern int x;
  void f();
  // ...
}

#endif // HEADER1_H ///:~
//: C10:Header2.h
#ifndef HEADER2_H
#define HEADER2_H
#include "Header1.h"
// Add more names to MyLib
namespace MyLib { // NOT a redefinition!
  extern int y;
  void g();
  // ...
}

#endif // HEADER2_H ///:~
//: C10:Continuation.cpp
#include "Header2.h"
int main() {} ///:~
```

415

- 一个**namespace**的名字可以用另一个名字来作它的别名，这样就不必敲打那些开发商提供的冗长的名字了。

```
//: C10:BobsSuperDuperLibrary.cpp
namespace BobsSuperDuperLibrary {
  class Widget { /* ... */ };
  class Poppit { /* ... */ };
  // ...
}
// Too much to type! I'll alias it:
namespace Bob = BobsSuperDuperLibrary;
int main() {} ///:~
```

- 不能像类那样去创建一个名字空间的实例。

10.2.1.1 未命名的名字空间

每个翻译单元都可包含一个未命名的名字空间——可以不用标识符而只用“**namespace**”增加一个名字空间。

```
//: C10:UnnamedNamespaces.cpp
namespace {
  class Arm  { /* ... */ };
  class Leg  { /* ... */ };
  class Head { /* ... */ };
  class Robot {
    Arm arm[4];
    Leg leg[16];
    Head head[3];
    // ...
  } xanthan;
  int i, j, k;
}
int main() {} ///:~
```

416

在这个空间中的名字自动地在翻译单元内无限制地有效。但要确保每个翻译单元只有一个未命名的名字空间。如果把一个局部名字放在一个未命名的名字空间中，不需要加上**static**说明就可以让它们作内部连接。

10.2.1.2 友元

可以在一个名字空间的类定义之内插入（*inject*）一个友元（**friend**）声明：

```
//: C10:FriendInjection.cpp
namespace Me {
  class Us {
    //...
    friend void you();
  };
}
int main() {} ///:~
```

这样函数**you()**就成了名字空间**Me**的一个成员。

10.2.2 使用名字空间

在一个名字空间中引用一个名字可以采取三种方法：第一种方法是用作用域运算符，第二种方法是用**using**指令把所有名字引入到名字空间中，第三种方法是用**using**声明一次性引用名字。

10.2.2.1 作用域解析

名字空间中的任何名字都可以用作用域运算符作明确地指定，就像引用一个类中的名字
417 一样：

```
//: C10:ScopeResolution.cpp
namespace X {
  class Y {
    static int i;
  public:
    void f();
  };
  class Z;
  void func();
}
int X::Y::i = 9;
class X::Z {
  int u, v, w;
public:
  Z(int i);
  int g();
};
X::Z::Z(int i) { u = v = w = i; }
int X::Z::g() { return u = v = w = 0; }
void X::func() {
  X::Z a(1);
  a.g();
}
int main(){} ///:~
```

注意定义**X::Y::i**就像引用一个类**Y**的数据成员一样容易，**Y**如同被嵌套在类**X**中而不像是

被嵌套在名字空间**X**中。

到目前为止，名字空间看上去很像类。

10.2.2.2 使用指令

用**using** 关键字可以让我们立即进入整个名字空间，摆脱输入一个名字空间中完整标识符的烦恼。这种**using**和**namespace**关键字的搭配使用称为使用指令（*using directive*）。**using** 关键字声明了一个名字空间中的所有名字是在当前范围内，所以可以很方便地使用这些未限定的名字。如果以一个简单的名字空间开始:

```
//: C10:NamespaceInt.h
#ifndef NAMESPACEINT_H
#define NAMESPACEINT_H
namespace Int {
  enum sign { positive, negative };
  class Integer {
    int i;
    sign s;
  public:
    Integer(int ii = 0)
      : i(ii),
        s(i >= 0 ? positive : negative)
    {}
    sign getSign() const { return s; }
    void setSign(sign sgn) { s = sgn; }
    // ...
  };
}
#endif // NAMESPACEINT_H ///:~
```

418

using指令的用途之一就是把名字空间**Int**中的所有名字引入到另一个名字空间中，让这些名字嵌套在那个名字空间中。

```
//: C10:NamespaceMath.h
#ifndef NAMESPACEMATH_H
#define NAMESPACEMATH_H
#include "NamespaceInt.h"
namespace Math {
  using namespace Int;
  Integer a, b;
  Integer divide(Integer, Integer);
  // ...
}
#endif // NAMESPACEMATH_H ///:~
```

可以在一个函数中声明名字空间**Int**中的所有名字，但是让这些名字嵌套在这个函数中。

```
//: C10:Arithmetic.cpp
#include "NamespaceInt.h"
void arithmetic() {
  using namespace Int;
  Integer x;
  x.setSign(positive);
}
int main(){} ///:~
```

419

如果不用**using**指令，在这个名字空间的所有名字都需要被完全限定。

using 指令有一个缺点，那就是看起来不那么直观，引入名字的可见性的范围是在使用**using**的地方。可以不考虑使用**using** 指令的名字，就像它们已经被全局声明过，现在变为这个范围。

```
//: C10:NamespaceOverriding1.cpp
#include "NamespaceMath.h"
int main() {
  using namespace Math;
  Integer a; // Hides Math::a;
  a.setSign(negative);
  // Now scope resolution is necessary
  // to select Math::a :
  Math::a.setSign(positive);
} ///:~
```

如果有第二个名字空间，它包含了名字空间**Math**的某些名字:

```
//: C10:NamespaceOverriding2.h
#ifndef NAMESPACEOVERRIDING2_H
#define NAMESPACEOVERRIDING2_H
#include "NamespaceInt.h"
namespace Calculation {
  using namespace Int;
  Integer divide(Integer, Integer);
  // ...
}
#endif // NAMESPACEOVERRIDING2_H ///:~
```

因为这个名字空间也是用**using**指令来引入的，这样就可能产生冲突。不过，这种二义性出现在名字的使用时，而不是在**using**指令使用时。

```
//: C10:OverridingAmbiguity.cpp
#include "NamespaceMath.h"
#include "NamespaceOverriding2.h"
void s() {
420   using namespace Math;
  using namespace Calculation;
  // Everything's ok until:
  //! divide(1, 2); // Ambiguity
}
int main() {} ///:~
```

这样，即使永远不产生歧义性，使用**using**指令引入带名字冲突的名字空间也是可能的。

10.2.2.3 使用声明

可以用*使用声明*（*using declaration*）一次性引入名字到当前范围内。这种方法不像**using**指令那样把那些名字当成当前范围的全局名来看待，**using**声明是在当前范围之内进行的一个声明，这就意味着在这个范围内它可以不顾来自**using**指令的名字。

```
//: C10:UsingDeclaration.h
#ifndef USINGDECLARATION_H
#define USINGDECLARATION_H
namespace U {
```

```
  inline void f() {}
  inline void g() {}
}
namespace V {
  inline void f() {}
  inline void g() {}
}
#endif // USINGDECLARATION_H ///:~
```

```
//: C10:UsingDeclaration1.cpp
#include "UsingDeclaration.h"
void h() {
  using namespace U; // Using directive
  using V::f; // Using declaration
  f(); // Calls V::f();
  U::f(); // Must fully qualify to call
}
int main() {} ///:~
```

using声明给出了标识符的完整的名字，但没有了类型方面的信息。也就是说，如果名字空间中包含了一组用相同名字重载的函数，**using**声明就声明了这个重载的集合内的所有函数。 421

可以把**using**声明放在任何一般的声明可以出现的地方。**using**声明与普通声明只有一点不同：**using**声明可以引起一个函数用相同的参数类型来重载（这在一般的重载中是不允许的）。当然这种不确定性要到使用时才表现出来，而不是在声明时。

using声明也可以出现在一个名字空间内，其作用与在其他地方时一样：

```
//: C10:UsingDeclaration2.cpp
#include "UsingDeclaration.h"
namespace Q {
  using U::f;
  using V::g;
  // ...
}
void m() {
  using namespace Q;
  f(); // Calls U::f();
  g(); // Calls V::g();
}
int main() {} ///:~
```

一个**using** 声明是一个别名，它允许在不同的名字空间声明同样的函数。如果不想由于引入不同名字空间而导致重复定义一个函数时，可以使用**using**声明，它不会引起任何二义性和重复。

10.2.3 名字空间的使用

上面所介绍的一些规则刚开始时也许会使我们感到气馁，特别是当我们知道将来一直使用它们会有什么感觉时，尤其如此。一般说来，只要真正理解了它们的工作机理，使用它们也会变得非常简单。需要记住的关键问题是当引入一个全局**using**指令时(可以在任何范围之外通过使用**using namespace**)，就已经为那个文件打开了该名字空间。对于一个实现文件(一个**.cpp**文件)来说，这通常是一个好方法，因为只有在该文件编译结束时，**using**指令才会起作 422

用。也就是说，它不会影响任何其他的文件，所以可以每次在一个实现文件中调整对名字空间的控制。例如，如果发现由于在一个特定的实现文件中使用太多的**using**指令而产生名字冲突，就要对该文件做简单的改变，以致使用明确的限定或者**using**声明来消除名字冲突，这样不用修改其他的实现文件。

头文件的情况与此不同。不要把一个全局的**using**指令引入到一个头文件中，因为那将意味着包含这个头文件的任何其他头文件也会打开这个名字空间(头文件可以被另一个头文件包含)。

所以，在头文件中，最好使用明确的限定或者被限定在一定范围内的**using**指令和**using**声明。在本书中将讨论这种用法，通过这种方法，就不会“污染”全局名字空间和后退到C++的名字空间引入前的世界。

10.3 C++中的静态成员

有时需要为某个类的所有对象分配一个单一的存储空间。在C语言中，可以用全局变量，但这样很不安全。全局数据可以被任何人修改，而且，在一个大项目中，它很容易与其他的名字相冲突。如果可以把一个数据当成全局变量那样去存储，但又被隐藏在类的内部，并且清楚地与这个类相联系，这种处理方法当然是最理想的了。

这一点可以用类的静态数据成员来实现。类的静态成员拥有一块单独的存储区，而不管创建了多少个该类的对象。所有的这些对象的静态数据成员都共享这一块静态存储空间，这就为这些对象提供了一种互相通信的方法。但静态数据属于类，它的名字只在类的范围内有
423 效，并且可以是**public**（公有的）、**private**（私有的）或者**protected**（保护的）。

10.3.1 定义静态数据成员的存储

因为类的静态数据成员有着单一的存储空间而不管产生了多少个对象，所以存储空间必须在一个单独的地方定义。编译器不会分配存储空间。如果一个静态数据成员被声明但没有定义时，连接器会报告一个错误。

定义必须出现在类的外部（不允许内联）而且只能定义一次，因此它通常放在一个类的实现文件中。这种规定常常让人感到很麻烦，但它实际上是很合理的。例如，在一个类中定义一个静态数据成员如下:

```
class A {
  static int i;
public:
  //...
};
```

之后，必须在定义文件中为静态数据成员定义存储区:

```
int A::i = 1;
```

如果要定义了一个普通的全局变量，可以这样:

```
int i = 1;
```

在这里，类名和作用域运算符用于指定了**A::i**。

有些人对**A::i**是私有的这点感到疑惑不解，可是在这里似乎在公开地直接对它处理。这不

是破坏了类结构的保护性吗？有两个原因可以保证它绝对的安全。第一，这些变量的初始化惟一合法的地方是在定义时。事实上，如果静态数据成员是一个带构造函数的对象时，可以调用构造函数来代替“=”操作符；第二，一旦这些数据被定义了，最终的用户就不能再定义它——否则连接器会报告错误。而且这个类的创建者被迫产生这个定义，否则这些代码在测试时无法连接。这就保证了定义只出现一次并且它是由类的构造者来控制的。 424

静态成员的初始化表达式是在一个类的作用域内，请看下例：

```
//: C10:Statinit.cpp
// Scope of static initializer
#include <iostream>
using namespace std;

int x = 100;

class WithStatic {
  static int x;
  static int y;
public:
  void print() const {
    cout << "WithStatic::x = " << x << endl;
    cout << "WithStatic::y = " << y << endl;
  }
};

int WithStatic::x = 1;
int WithStatic::y = x + 1;
// WithStatic::x NOT ::x

int main() {
  WithStatic ws;
  ws.print();
} ///:~
```

这里，**withStatic::**限定符把**withStatic**的作用域扩展到全部定义中。

10.3.1.1 静态数组的初始化

第8章介绍了静态常量（**static const**）变量，它允许在一个类体中定义一个常量值。也可以创建静态对象数组，包括**const**数组与非**const**数组。这同前面的语法是一致的。

```
//: C10:StaticArray.cpp
// Initializing static arrays in classes
class Values {
  // static consts are initialized in-place:
  static const int scSize = 100;
  static const long scLong = 100;
  // Automatic counting works with static arrays.
  // Arrays, Non-integral and non-const statics
  // must be initialized externally:
  static const int scInts[];
  static const long scLongs[];
  static const float scTable[];
  static const char scLetters[];
  static int size;
  static const float scFloat;
```

425

```
  static float table[];
  static char letters[];
};

int Values::size = 100;
const float Values::scFloat = 1.1;

const int Values::scInts[] = {
  99, 47, 33, 11, 7
};

const long Values::scLongs[] = {
  99, 47, 33, 11, 7
};

const float Values::scTable[] = {
  1.1, 2.2, 3.3, 4.4
};

const char Values::scLetters[] = {
  'a', 'b', 'c', 'd', 'e',
  'f', 'g', 'h', 'i', 'j'
};

float Values::table[4] = {
  1.1, 2.2, 3.3, 4.4
};

char Values::letters[10] = {
  'a', 'b', 'c', 'd', 'e',
  'f', 'g', 'h', 'i', 'j'
};
```

426

```
int main() { Values v; } ///:~
```

利用全部类型的静态常量，可以在类内提供这些定义,但是对于其他的对象（包括全部类型的数组，甚至它们为静态常量），必须为这些成员提供专门的外部定义。这些定义是内部连接的，所以可以把它放在头文件中，初始化静态数组的方法与其他集合类型的初始化一样，包括自动计数。

也可以创建类的静态常量对象和这样的对象的数组。不过，不能使用“内联语法”初始化它们，这种语法对全部的内部类型的静态常量有效。

```
//: C10:StaticObjectArrays.cpp
// Static arrays of class objects
class X {
  int i;
public:
  X(int ii) : i(ii) {}
};

class Stat {
  // This doesn't work, although
  // you might want it to:
//!  static const X x(100);
  // Both const and non-const static class
```

```
  // objects must be initialized externally:
  static X x2;
  static X xTable2[];
  static const X x3;
  static const X xTable3[];
};

X Stat::x2(100);

X Stat::xTable2[] = {
  X(1), X(2), X(3), X(4)
};

const X Stat::x3(100);
const X Stat::xTable3[] = {
  X(1), X(2), X(3), X(4)
};

int main() { Stat v; } ///:~
```

类对象的常量和非常量静态数组的初始化必须以相同的方式执行，它们遵守典型的静态定义语法。

10.3.2 嵌套类和局部类

可以很容易地把一个静态数据成员放在另一个类的嵌套类中。这样的成员的定义显然是上节中情况的扩展——只须用另一种级别的作用域指定。然而不能在局部类（在函数内部定义的类）中有静态数据成员。因而，如下例：

```
//: C10:Local.cpp
// Static members & local classes
#include <iostream>
using namespace std;

// Nested class CAN have static data members:
class Outer {
  class Inner {
    static int i; // OK
  };
};

int Outer::Inner::i = 47;

// Local class cannot have static data members:
void f() {
  class Local {
  public:
//! static int i;  // Error
    // (How would you define i?)
  } x;
}

int main() { Outer x; f(); } ///:~
```

可以看到一个局部类中有与静态成员直接相关的问题。为了定义数据成员，怎样才能在文件范围描述它呢？实际上很少使用局部类。

10.3.3 静态成员函数

像静态数据成员一样，也可以创建一个静态成员函数，它为类的全体对象服务而不是为一个类的特殊对象服务。这样就不需要定义一个全局函数，减少了全局或局部名字空间的占用，把这个函数移到了类的内部。当产生一个静态成员函数时，也就表达了与一个特定类的联系。

可以用普通的方法调用静态成员函数，用点“.”和箭头“->”把它与一个对象相联系。然而，调用静态成员函数的一个更典型的方法是自我调用，这不需要任何具体的对象，而是像下面使用作用域运算符:

```
//: C10:SimpleStaticMemberFunction.cpp
class X {
public:
  static void f(){};
};

int main() {
  X::f();
} ///:~
```

当在一个类中看到静态成员函数时，要记住：类的设计者是想把这些函数与整个类在概念上关联起来。

静态成员函数不能访问一般的数据成员，而只能访问静态数据成员，也只能调用其他的静态成员函数。通常，当前对象的地址（**this**）是被隐式地传递到被调用的函数的。但一个静态成员函数没有**this**，所以它无法访问一般的成员。这样使用静态成员函数在速度上可以比全局函数有少许的增长，它不仅没有传递**this**所需的额外开销，而且还有使函数在类内的好处。

429

对于数据成员来说，**static**关键字指定它对类的所有对象来说，都只占有相同的一块存储空间。与定义对象的静态使用相对应，静态函数意味着对这个函数的所有调用来说，一个局部变量只有一份拷贝。

下面是一个静态数据成员和静态成员函数在一起使用的例子:

```
//: C10:StaticMemberFunctions.cpp
class X {
  int i;
  static int j;
public:
  X(int ii = 0) : i(ii) {
     // Non-static member function can access
     // static member function or data:
    j = i;
  }
  int val() const { return i; }
  static int incr() {
    //! i++; // Error: static member function
    // cannot access non-static member data
    return ++j;
```

```
  }
  static int f() {
    //! val(); // Error: static member function
    // cannot access non-static member function
    return incr(); // OK -- calls static
  }
};

int X::j = 0;

int main() {
  X x;
  X* xp = &x;
  x.f();
  xp->f();
  X::f(); // Only works with static members
} ///:~
```

因为静态成员函数没有**this**指针，所以它既不能访问非静态的数据成员，也不能调用非静态的成员函数。

430

注意在**main( )**中，一个静态成员可以用点或箭头来选取，把那个函数与一个对象联系起来，但也可以不与对象相联系（因为一个静态成员是与一个类相连，而不是与一个特定的对象相连），而是用类的名字和作用域运算符。

这里有一个有趣的特点：因为静态成员对象的初始化方法，所以可以把上述类的一个静态数据成员放到那个类的内部。下面是一个例子，它把构造函数变成私有的，这样**Egg**类只有一个惟一的对象存在，可以访问那个对象，但不能产生任何新的**Egg**对象。

```
//: C10:Singleton.cpp
// Static member of same type, ensures that
// only one object of this type exists.
// Also referred to as the "singleton" pattern.
#include <iostream>
using namespace std;

class Egg {
  static Egg e;
  int i;
  Egg(int ii) : i(ii) {}
  Egg(const Egg&); // Prevent copy-construction
public:
  static Egg* instance() { return &e; }
  int val() const { return i; }
};

Egg Egg::e(47);

int main() {
//!  Egg x(1); // Error -- can't create an Egg
  // You can access the single instance:
  cout << Egg::instance()->val() << endl;
} ///:~
```

E的初始化出现在类的声明完成后，所以编译器已有足够的信息为对象分配空间并调用构

431 造函数。

为了完全防止创建其他对象，还需要再做如下的工作：增加一个叫做拷贝构造函数(*copy constructor*)的私有构造函数。到目前为止，还不知道为什么必须这样做，因为在下章中才会讨论拷贝构造函数。然而，如果删除上面例子中定义的拷贝构造函数，那么就能像下面那样创建一个**Egg**对象。

```
Egg e = *Egg::instance();
Egg e2(*Egg::instance());
```

这两条语句都使用了拷贝构造函数，所以为了禁止这种可能性，拷贝构造函数声明为私有的（不需要定义，因为它不会被调用）。下一章的大部分内容是对拷贝构造函数的讨论，所以，通过下章的学习后，我们会明白是怎么一回事。

10.4 静态初始化的相依性

在一个指定的翻译单元中，静态对象的初始化顺序严格按照对象在该单元中定义出现的顺序。而清除的顺序则与初始化的顺序正好相反。

但是，对于作用域为多个翻译单元的静态对象来说，不能保证严格的初始化顺序，也没有办法来指定这种顺序。这可能会引起一些问题。下面的例子如果包含一个文件就会立即引起灾难（它会暂停一些简单的操作系统的运行，中止进程）。

```
// First file
#include <fstream>
std::ofstream out("out.txt");
```

另一个文件在它的初始表达式之一中用到了**out** 对象:

```
// Second file
#include <fstream>
extern std::ofstream out;
```

432

```
class Oof {
public:
  Oof() { std::out << "ouch"; }
} oof;
```

这个程序可能运行，也可能不能运行。如果在建立可执行文件时第一个文件先初始化，那么就不会有问题，但如果第二个文件先初始化，**Oof**的构造函数依赖于**out**的存在，而此时**out**还没有创建，于是就会引起混乱。

这种情况只会在相互依赖的静态对象的初始化时出现。在一个翻译单元内的一个函数的第一次调用之前，但在进入**main( )**之后，这个翻译单元内的静态对象都被初始化。如果静态对象位于不同的文件中，则不能确定这些静态对象的初始化顺序。

在**ARM**[⊖]中可以看到一个更微妙的例子，在一个文件中:

```
extern int y;
int x = y + 1;
```

在另一个文件中

```
extern int x;
```

⊖ 《*The Annotated C++ Reference Manual*》,Bjarne Stroustrup和Margaret Ellis著，1990年，20~21页。

```
int y = x + 1;
```

对所有的静态对象，连接装载机制在程序员指定的动态初始化发生前保证一个静态成员初始化为零。在前一个例子中，**fstream out**对象的存储空间赋零并没有特别的意义，所以它在构造函数调用前确实是未定义的。然而，对内部数据类型，初始化为零是有意义的，所以如果文件按上面的顺序被初始化，**y**开始被初始化为零，所以**x**变成**1**，而后**y**被动态初始化为**2**。然而，如果初始化的顺序颠倒过来，**x**被静态初始化为零，**y**被初始化为**1**，而后**x**被初始化为**2**。 433

程序员必须意识到这些，因为他们可能会在编程时遇到互相依赖的静态变量的初始化问题，程序可能在一个平台上工作正常，当把它移到另一个编译环境时，突然莫名其妙地不工作了。

10.4.1 怎么办

有三种方法来处理这一问题:

1) 不用它，避免初始化时的互相依赖。这是最好的解决方法。

2) 如果实在要用，就把那些关键的静态对象的定义放在一个文件中，这样只要让它们在文件中顺序正确就可以保证它们正确的初始化。

3) 如果确信把静态对象放在几个不同的翻译单元中是不可避免的——如在编写一个库时，这时无法控制那些使用该库的程序员——这可以通过两种程序设计技术加以解决。

10.4.1.1 技术一

这是由Jerry Schwarz在创建iostream库（因为**cin**、**cout**和**cerr**是静态的且定义在不同的文件中）时首创的一种技术。它实际上没有第二种技术好，但是因为它的生存期比较长，这样可能会遇到很多代码使用了它。知道它的工作原理还是很重要的。

这一技术要求在库头文件中加上一个额外的类。这个类负责库中的静态对象的动态初始化。下面是一个简单的例子:

```
//: C10:Initializer.h
// Static initialization technique
#ifndef INITIALIZER_H
#define INITIALIZER_H
#include <iostream>                                          434
extern int x; // Declarations, not definitions
extern int y;

class Initializer {
  static int initCount;
public:
  Initializer() {
    std::cout << "Initializer()" << std::endl;
    // Initialize first time only
    if(initCount++ == 0) {
      std::cout << "performing initialization"
                << std::endl;
      x = 100;
      y = 200;
    }
  }
  ~Initializer() {
```

```
    std::cout << "~Initializer()" << std::endl;
    // Clean up last time only
    if(--initCount == 0) {
      std::cout << "performing cleanup"
                << std::endl;
      // Any necessary cleanup here
    }
  }
};

// The following creates one object in each
// file where Initializer.h is included, but that
// object is only visible within that file:
static Initializer init;
#endif // INITIALIZER_H ///:~
```

x、y的声明只是表明这些对象的存在，并没有为它们分配存储空间。然而**initializer init**的定义为每个包含此头文件的文件分配那些对象的存储空间，因为名字是**static**的（这里控制可见性而不是指定存储类型，因为默认时是在文件作用域内）它只在本翻译单元可见，所以连接器不会报告一个多重定义错误。

435 下面是一个包含**x**、**y**和**init_Count**定义的文件:

```
//: C10:InitializerDefs.cpp {O}
// Definitions for Initializer.h
#include "Initializer.h"
// Static initialization will force
// all these values to zero:
int x;
int y;
int Initializer::initCount;
///:~
```

（当然，当一个文件包含头文件时，它的**init**静态实例也放在该文件中。）假设库的使用者产生了两个其他的文件:

```
//: C10:Initializer.cpp {O}
// Static initialization
#include "Initializer.h"
///:~
```

以及

```
//: C10:Initializer2.cpp
//{L} InitializerDefs Initializer
// Static initialization
#include "Initializer.h"
using namespace std;

int main() {
  cout << "inside main()" << endl;
  cout << "leaving main()" << endl;
} ///:~
```

现在哪个翻译单元先初始化都没有关系。当第一次包含**Initializer.h**的翻译单元被初始化时，**initCount**为零，这时初始化就已经完成了（这是由于任何动态初始化进行之前，静态存

储区已被设置为零)。对其余的翻译单元，**initCount**不会为零，并忽略初始化操作。清除将按相反的顺序发生，且**~Initializer( )**可确保它只发生一次。

这个例子用内部类型作为全局静态对象，这种方法也可以用于类，但其对象必须用 436
initializer类动态初始化。一种方法就是创建一个没有构造函数和析构函数的类，而是带有不同名字的用于初始化和清除的成员函数。当然更常用的做法是在**initializer( )**函数中，设定有指向对象的指针，用**new**创建它们。

10.4.1.2 技术二

在技术一使用很久之后才有人（我不知道是谁）提出了本小节将要说明的技术二。与技术一相比，这种技术更简单，也更清晰。之所以在技术一出现这么久之后才有技术二是因为C++太复杂。

这一技术基于这样的事实：函数内部的静态对象在函数第一次被调用时初始化，且只被初始化一次。需要记住的是，在这里真正想要解决的不是静态对象什么时候被初始化(这可以个别地加以控制)，而是确保正确的初始化顺序。

这种技术很灵巧。对于任何初始化依赖因素来说，可以把一个静态对象放在一个能返回对象引用的函数中。使用这种方法,访问静态对象的惟一途径就是调用这个函数。如果该静态对象需要访问其他依赖于它的静态对象时，就必须调用那些对象的函数。函数第一次被调用时，它强迫初始化发生。静态初始化的正确顺序是由设计的代码而不是由连接器任意指定顺序来保证的。

为了给出一个例子，这里有两个相互依赖的类。第一个类包含一个**bool**类型的成员，它只由构造函数初始化，所以能够知道该类的一个静态实例是否调用了构造函数(在程序开始时，静态存储区被初始化为零，如果没有调用构造函数的话，会对**bool**成员产生一个**false**值)。

```
//: C10:Dependency1.h
#ifndef DEPENDENCY1_H
#define DEPENDENCY1_H
#include <iostream>                                    437

class Dependency1 {
  bool init;
public:
  Dependency1() : init(true) {
    std::cout << "Dependency1 construction"
              << std::endl;
  }
  void print() const {
    std::cout << "Dependency1 init: "
              << init << std::endl;
  }
};
#endif // DEPENDENCY1_H ///:~
```

构造函数也显示它是什么时候被调用的，为了知道对象是否被初始化，可以通过**print( )**函数打印出对象的状态。

第二个类初始化由第一个类的一个对象来完成，这将会导致初始化相互依赖。

```
//: C10:Dependency2.h
#ifndef DEPENDENCY2_H
```

```
#define DEPENDENCY2_H
#include "Dependency1.h"

class Dependency2 {
  Dependency1 d1;
public:
  Dependency2(const Dependency1& dep1): d1(dep1){
    std::cout << "Dependency2 construction ";
    print();
  }
  void print() const { d1.print(); }
};
#endif // DEPENDENCY2_H ///:~
```

构造函数显示它自己并打印出对象**d1**的状态，所以能够知道当构造函数被调用时，**d1**是否已经初始化了。

438 为了说明会出现什么错误，下面的文件首先以一种不正确的顺序定义静态对象，如果在对象**Dependency1**之前连接器碰巧初始化对象**Dependency2**，错误就会出现。如果定义的顺序恰好正确，那么就会以相反的顺序的显示说明它是如何正常工作的。这样，说明技术二是可靠的。

为了有更多的可读性的输出，增加**separator( )**函数。诀窍就是不能全局地调用一个函数，除非该函数用来执行一个变量的初始化操作，所以**separator( )**函数返回一个哑元值用来初始化两个全局变量。

```
//: C10:Technique2.cpp
#include "Dependency2.h"
using namespace std;

// Returns a value so it can be called as
// a global initializer:
int separator() {
  cout << "----------------------" << endl;
  return 1;
}

// Simulate the dependency problem:
extern Dependency1 dep1;
Dependency2 dep2(dep1);
Dependency1 dep1;
int x1 = separator();

// But if it happens in this order it works OK:
Dependency1 dep1b;
Dependency2 dep2b(dep1b);
int x2 = separator();

// Wrapping static objects in functions succeeds
Dependency1& d1() {
  static Dependency1 dep1;
  return dep1;
}

Dependency2& d2() {
```

```
  static Dependency2 dep2(d1());
  return dep2;
}
int main() {
  Dependency2& dep2 = d2();
} ///:~
```

439

函数**d1( )**和**d2( )**包含类**Dependency1**和**Dependency2**的静态对象。现在，访问这些静态对象的惟一方法就是调用这两个函数，并在第一次函数调用时强迫进行静态初始化，这可以保证初始化的正确性，通过这种方法，可以知道程序什么时候运行以及输出什么结果。

下面的代码使用了技术二。通常，静态对象在单独的文件中定义（由于某些原因，必须这样做；不过要记住在单独的文件中定义静态对象也会出现问题），而不是在单独的文件中定义一个包含静态对象的函数。但是需要在头文件中声明。

```
//: C10:Dependency1StatFun.h
#ifndef DEPENDENCY1STATFUN_H
#define DEPENDENCY1STATFUN_H
#include "Dependency1.h"
extern Dependency1& d1();
#endif // DEPENDENCY1STATFUN_H ///:~
```

实际上，关键字“**extern**”对于函数声明来说是多余的。下面是第二个头文件：

```
//: C10:Dependency2StatFun.h
#ifndef DEPENDENCY2STATFUN_H
#define DEPENDENCY2STATFUN_H
#include "Dependency2.h"
extern Dependency2& d2();
#endif // DEPENDENCY2STATFUN_H ///:~
```

在前面的实现文件中，有静态对象定义，现在，改为在包装的函数定义中定义静态对象：

```
//: C10:Dependency1StatFun.cpp {O}
#include "Dependency1StatFun.h"
Dependency1& d1() {
  static Dependency1 dep1;
  return dep1;
} ///:~
```

440

其他的代码也可以放在这些头文件中，下面是另外一个文件：

```
//: C10:Dependency2StatFun.cpp {O}
#include "Dependency1StatFun.h"
#include "Dependency2StatFun.h"
Dependency2& d2() {
  static Dependency2 dep2(d1());
  return dep2;
} ///:~
```

现在有两个文件，这两个文件可以以任意的顺序连接。如果它们只包含普通的静态对象，那么可以产生任意顺序的初始化。在这里因为它们包含定义静态对象的函数，所以不会出现不正确的初始化：

```
//: C10:Technique2b.cpp
//{L} Dependency1StatFun Dependency2StatFun
```

```
#include "Dependency2StatFun.h"
int main() { d2(); } ///:~
```

当运行这个程序时，将会发现**Dependency1**类的静态对象的初始化总是发生在类**Dependency2**的静态对象的初始化之前。所以，从中可以看出这种方法要比第一种技术简单得多。

我们也许想在函数**d1( )**和**d2( )**的头文件中把它们声明为内联函数，但是我们必须明确地知道这样做不行。内联函数在它出现的每一个文件中都会有一份副本——这种副本包括静态对象的定义。因为内联函数自动地默认为内部连接，所以这将导致多个重复的静态对象，且它们作用域为多个编译单元，这当然会出现问题。所以必须确保每一个定义了静态对象的函
441 数只有一份定义，这就意味着不能把定义了静态对象的函数作为内联函数。

10.5 替代连接说明

如果在C++中编写一个程序需要用到C的库，那该怎么办呢？如果这样声明一个C函数：

```
float f(int a, char b);
```

C++的编译器就会将这个名字变成像**_f_int_char**之类的东西以支持函数重载（和类型安全连接）。然而，C编译器编译的库一般不做这样的转换，所以它的内部名为**_f**。这样，连接器将无法解释C++对**f( )**的调用。

C++中提供了一个替代连接说明（*alternate linkage specification*），它是通过重载**extern**关键字来实现的。**extern**后跟一个字符串来指定想声明的函数的连接类型，后面是函数声明。

```
extern "C" float f(int a, char b);
```

这就告诉编译器**f( )**是C连接，这样就不会转换函数名。标准的连接类型指定符有“**C**”和“**C++**”两种，但编译器开发商可选择用同样的方法支持其他语言。

如果有一组替代连接的声明，可以把它们放在花括号内：

```
extern "C" {
  float f(int a, char b);
  double d(int a, char b);
}
```

或在头文件中：

```
extern "C" {
#include "Myheader.h"
442 }
```

多数C++编译器开发商在他们的头文件中处理转换连接指定，包括C和C++，所以不用担心它们。

10.6 小结

static关键字很容易使人糊涂，因为有时它控制存储分配，而有时控制一个名字的可见性和连接。

随着C++名字空间的引入，我们有了更好的、更灵活的方法来控制一个大项目中名字的增长。

在类的内部使用**static**是在全程序中控制名字的另一种方法。这些名字不会与全局名冲突，并且可见性和访问也限制在程序内部，使得在维护代码时能有更多的控制。

10.7 练习

部分练习题的答案可以在本书的电子文档 "*Annotated Solution Guide for Thinking in C++*" 中找到，只需支付很少的费用就可以从http://www.BruceEckel.com得到这个电子文档。

10-1 创建一个函数（带一个默认值为零的参数），函数内有一个静态变量的，这个静态变量是一个指针。当调用者为这个参数提供值时，它就指向一个整形数组的起始地址。如果用默认的参数值调用该函数，那么这个函数就返回数组的下一个值，直到它访问到数组中的"－1"（在数组中，－1作为结束的标志），在函数**main( )**中调用这个函数。

10-2 创建一个这样的函数：每调用一次，它就返回Fibonacci序列中的下一个值。增加一个**bool**类型的参数，其默认值为**false**，当传递给该参数的值为**true**时重置函数使它指向Fibonacci序列的开头。在函数**main( )**中调用这个函数。 443

10-3 创建一个有一个整型数组的类。在类内部用静态整型常量设置数组的大小。增加一个**const int** 变量，并在构造函数初始化列表中初始化。构造函数是内联的。增加一个**static int** 成员变量并用特定值来初始化。增加一个内联的成员函数**print( )**，它打印数组中所有数组元素值并调用静态成员函数。在**main( )**中运用这样的类。

10-4 创建一个类**Monitor**，它能知道它的成员函数**incident( )**被调用了多少次。增加一个成员函数**print( )**显示**incident( )**被调用的次数，再创建一个包含一个静态的**Monitor**类的对象的函数。每次调用该函数时，它都会调用**print( )**成员函数显示**incident( )**被调用的次数。在主函数**main( )**中调用这个函数。

10-5 修改练习4中的**Monitor**类，使其成员函数**decrement( )**被调用时会减少记数。另创建一个类**Monitor2**，它的构造函数有一个指向**Monitor1**的指针参数，该构造函数存储指针值，调用**incident( )**以及**print( )**。**Monitor2**的析构函数调用**decrement( )**和**print( )**。写一个函数，在该函数中创建一个**Monitor2**的静态对象。在**main( )**中测试调用该函数和不调用该函数时，**Monitor2**的析构函数各会出现什么结果。

10-6 定义一个**Monitor2**类的全局对象，看看会得到什么结果。

10-7 创建一个类，它的析构函数打印信息并调用**exit( )**，定义该类的一个全局对象，看看会得到什么结果。

10-8 在文件**StaticDestructors.cpp**中，在**main( )**内用不同的顺序调用**f( )**、**g( )**来检验构造函数与析构函数的调用顺序，你的编译器能正确地编译它们吗？

10-9 在文件**StaticDestructors.cpp**中，把**out**的最初定义变为一个**extern**声明，并把实际定义放到**a**（它的**Obj**构造函数传送信息给**out**）的定义之后，看看默认的错误处理是怎样工作的。运行程序时应确保没有其他重要程序在运行，否则机器会出现错误。 444

10-10 验证当带有多个静态变量的头文件被多个**cpp**文件包含时，不会有名字冲突。

10-11 创建一个简单的类，它包含一个整型数据成员，一个用自身参数初始化该数据成员的构造函数，还有一个用自身参数设置该成员值的成员函数，以及打印该成员值的**print( )**函数。把该类放到头文件中去，在两个**cpp**文件中包含该头文件，在

一个头文件中创建类的一个实例，在另外一个类中用**extern**声明，并在**main( )**中测试。记住必须连接两个对象文件，否则连接器将找不到所要连接的目标。

10-12 创建练习11中的类的静态实例，并验证：由于不存在**this**指针，连接器找不到它。

10-13 在一个头文件中声明一个函数。在另一个**cpp**文件中定义它，在第二个**cpp**文件的**main( )**中调用这个函数，编译并验证它能正常运行。然后改变函数的定义，使它变为静态，验证连接器将找不到这个函数。

10-14 修改第8章中的**Volatile.cpp**文件，使**comm::isr( )**能够像中断服务例程一样运行。注意：中断服务例程不带任何参数。

10-15 写一个使用**auto**和**register**关键字的简单程序，然后编译它。

10-16 创建一个包含一个名字空间的头文件。在名字空间里声明几个函数。再创建另一个头文件，它包含第一个头文件，并在先前的名字空间的基础上再增加几个函数声明。写一个包含第二个头文件的**cpp**文件。把名字空间用一个短的别名代替。在函数定义里使用作用域运算符调用这些函数。在另外一个单独的函数里，通过
445 **using**指令把名字空间引入到函数中。并证实：这时并不需要作用域运算符调用名字空间里的函数。

10-17 创建一个带无名的名字空间的头文件。在两个单独的**cpp**文件中包含这个头文件，验证这个无名的名字空间对于这两个翻译单元来说都是一致的。

10-18 使用练习17的头文件，验证无名名字空间中的名字在一个翻译单元里即使不加指定也是可见的。

10-19 修改**FriendInjection.cpp**文件，增加一个友元函数的定义，在主函数**main( )**中调用它。

10-20 在文件**Arithmetic.cpp**中，说明在一个函数中使用的**using**指令并不能扩展到这个函数的范围之外。

10-21 修改文件**OverridingAmbiguity.cpp**，先使用作用域运算符，然后用**using**声明代替作用域运算符来强迫编译器选择其中某个同名的函数名。

10-22 在两个头文件中，创建两个名字空间，每一个名字空间都包含一个类，且类名相同。创建一个包含这两个头文件的**cpp**文件。定义一个函数，在该函数中用**using**指令引入两个名字空间，然后创建类的一个对象，看看会有什么发生。再改变**using**指令的使用，使它为全局使用（在函数之外），看看结果是否不同。另外再使用作用域运算符，并创建两个类的对象。

10-23 用**using**声明修改练习22的程序，强迫编译器选择其中某个同名的类名。

10-24 去掉文件**BobsSuperDuperLibrary.cpp**和**UnnamedNamespaces.cpp**中的名字空间声明，把这些声明放到一个单独的头文件中，在处理过程中给这个无名的名字空间一个名字。在第三个文件中创建一个新的名字空间，该名字空间使用**using**声明
446 包含其他两个名字空间。在主函数**main( )**中使用**using**指令引用这个新的名字空间并访问所有的名字空间。

10-25 创建一个包含**<string>**和**<iostream>**的头文件，但不使用任何**using**指令和**using**声明。就像本书中所看到的一样，这里使用“include”。创建一个带有内联函数的类，它含有一个**string**成员，一个用自身参数初始化该成员的构造函数，还含有一个

print( )函数，它显示**String**成员的值，写一个**cpp**文件并在**main( )**中运用这个类。

10-26 创建一个带**static double**和**long**类型的成员的类，写一个静态的成员函数并打印出这些静态数据成员的值。

10-27 创建一个类，它包含一个整型数据成员，一个通过自身参数初始化该整型数据成员的构造函数，还有一个显示这个整型数据成员的**print( )**函数。再创建一个类，它包含第一个类的静态对象，增加一个静态成员函数并调用这个静态对象的**print( )**函数，在主函数**main( )**中运用这个类。

10-28 创建一个类，包含常量的和非常量的静态整型数组。写静态的方法来打印这些数组的值。在**main( )**函数中运用这些类。

10-29 创建一个类，它包含一个**string**类型的数据成员，一个通过自身参数初始化该数据成员的构造函数，还有一个显示这个数据成员的**print( )**函数，再创建一个类，它包含第一个类的对象的**const**和非**const**的静态对象数组，还有打印这些数组的静态方法。在**main( )**函数中运用第二个类。

10-30 创建一个带整型成员和一个默认构造函数的结构（**struct**），默认的构造函数把整型成员初始化为零。让这个结构局部于一个函数。在该函数中，创建一个该结构的对象数组，并演示这个数组中的整型被自动初始化为零。

10-31 创建一个类，它体现指针连接，但只允许使用一个指针。

10-32 在一个头文件中，创建一个类**Mirror**，它包含两个数据成员：一个是指向**Mirror**对象的指针和一个**bool**类型的数据成员，写两个构造函数：一个是默认的构造函数，它把**bool**成员初始化为**true**，使**Mirror**指向零值。第二个构造函数带有一个指向**Mirror**对象的指针参数，用该参数给对象的指针赋值。并把**bool**类型的数据成员设置成**false**。再增加一个成员函数**test( )**，如果对象的指针成员为非零，则通过指针调用**test( )**并返回它的值。如果指针是零，就返回**bool**类型的数据成员的值。然后写5个**cpp**文件，每一个都包含**Mirror**头文件。第一个**cpp**文件通过使用默认构造函数定义一个全局的**Mirror**对象。第二个**cpp**文件把第一个文件中定义的对象声明为**extern**，并通过使用第二个构造函数定义一个全局的**Mirror**对象，用一个指针指向第一个对象，在第三、四、五个文件中也做同样的处理。在最后一个文件中当然也包括一个全局对象的定义，并且**main( )**应该调用**test( )**函数并报告结果。如果结果为**true**，找出该如何改变连接器的连接顺序来使返回的结果为**false**。 447

10-33 用本书中介绍的技术一修改练习32中的程序。

10-34 用本书中介绍的技术二修改练习32中的程序。

10-35 写一个不包含任何头文件的程序，用标准的C库函数声明**puts( )**，在主函数**main( )**中调用这个函数。 448

第11章 引用和拷贝构造函数

449 引用就像是能自动地被编译器间接引用的常量型指针。

虽然引用Pascal语言中也有，但C++中引用的思想来自于Algol语言。在C++中，引用是支持运算符重载语法的基础（见第12章），也为函数参数的传入和传出控制提供了便利。

本章首先简单地介绍一下C 和C++的指针的差异，然后介绍引用。但本章的大部分内容将研究对于C++新手来说比较含混的问题：*拷贝构造函数*(*copy-constructor*)。它是一种特殊的构造函数，需要用引用来实现从现有的相同类型的对象中产生新的对象。编译器使用拷贝构造函数通过*按值传递*（*by value*）的方式在函数中传递和返回对象。

本章最后将阐述有点难以理解的C++的*成员指针*(*pointer-to-member*)这个概念。

11.1 C++中的指针

C和C++指针的最重要的区别在于C++是一种类型要求更强的语言。就**void***而言，这一点表现得更加突出。C不允许随便地把一个类型的指针赋值给另一个类型，但允许通过**void***来实现。例如:

```
bird* b;
rock* r;
void* v;
v = r;
b = v;
```

由于C的这种功能允许把任何一种类型看做别的类型处理，这就在类型系统中留下了一个大的漏洞。C++不允许这样做，其编译器将会给出一个出错信息。如果真想把某种类型当做别的类型处理，则必须显式地使用类型转换，通知编译器和读者（第3章已经介绍了C++的经
450 过改进“显式”类型转换语法）。

11.2 C++中的引用

引用（*reference*）**(&)** 就像能自动地被编译器间接引用的常量型指针。它通常用于函数的参数表中和函数的返回值，但也可以独立使用。例如:

```
//: C11:FreeStandingReferences.cpp
#include <iostream>
using namespace std;

// Ordinary free-standing reference:
int y;
int& r = y;
// When a reference is created, it must
// be initialized to a live object.
// However, you can also say:
const int& q = 12;  // (1)
```

```
// References are tied to someone else's storage:
int x = 0;          // (2)
int& a = x;         // (3)
int main() {
  cout << "x = " << x << ", a = " << a << endl;
  a++;
  cout << "x = " << x << ", a = " << a << endl;
} ///:~
```

在行（1）中，编译器分配了一个存储单元，它的值被初始化为12，于是这个引用就和这个存储单元联系上了。应用要点是任何引用必须和存储单元联系。访问引用时，就是在访问那个存储单元。因而，如果写行（2）和（3），那么增加**a**事实上就是增加**x**，这个可在**main( )**函数中显示出来。思考一个引用的最简单的方法是把它当做一个奇特的指针。这个指针的一个优点是不必怀疑它是否被初始化了（编译器强迫它初始化），也不必知道怎样对它间接引用（这由编译器做）。

使用引用时有一定的规则：

1) 当引用被创建时，它必须被初始化（指针则可以在任何时候被初始化）。 451

2) 一旦一个引用被初始化为指向一个对象，它就不能改变为另一个对象的引用（指针则可以在任何时候指向另一个对象）。

3) 不可能有NULL引用。必须确保引用是和一块合法的存储单元关联。

11.2.1 函数中的引用

最经常看见引用的地方是在函数参数和返回值中。当引用被用做函数参数时，在函数内任何对引用的更改将对函数外的参数产生改变。当然，可以通过传递一个指针来做相同的事情，但引用具有更清晰的语法。（如果愿意的话，可以把引用看做一个使语法更加便利的工具。）

如果从函数中返回一个引用，必须像从函数中返回一个指针一样对待。当函数返回时，无论引用关连的是什么都应该存在，否则，将不知道指向哪一个内存。

下面有一个例子：

```
//: C11:Reference.cpp
// Simple C++ references

int* f(int* x) {
  (*x)++;
  return x; // Safe, x is outside this scope
}

int& g(int& x) {
  x++; // Same effect as in f()
  return x; // Safe, outside this scope
}

int& h() {
  int q;
//!  return q;  // Error
  static int x;                                    452
  return x; // Safe, x lives outside this scope
}
```

```
int main() {
  int a = 0;
  f(&a); // Ugly (but explicit)
  g(a);  // Clean (but hidden)
} ///:~
```

对函数**f( )**的调用缺乏使用引用的方便性和清晰性，但很清楚这是传递一个地址。在函数**g ( )**的调用中，地址通过引用被传递，但表面上看不出来。

11.2.1.1 常量引用

仅当在**Reference.cpp**中的参数是非常量对象时，这个引用参数才能工作。如果是常量对象，函数**g ( )**将不接受这个参数，这样做是一件好事，因为这个函数将改变外部参数。如果知道这函数不妨碍对象的不变性的话，让这个参数是一个常量引用将允许这个函数在任何情况下使用。这意味着，对于内部类型，这个函数不会改变参数，而对于用户定义的类型，该函数只能调用常量成员函数，而且不应当改变任何公共的数据成员。

在函数参数中使用常量引用特别重要。这是因为我们的函数也许会接受临时对象，这个临时对象是由另一个函数的返回值创立或由函数使用者显式地创立的。临时对象总是不变的，因此如果不使用常量引用，参数将不会被编译器接受。看下面一个非常简单的例子：

```
//: C11:ConstReferenceArguments.cpp
// Passing references as const

void f(int&) {}
void g(const int&) {}

int main() {
//!  f(1); // Error
453   g(1);
} ///:~
```

调用**f(1)**会产生编译期间错误，这是因为编译器必须首先建立一个引用，即编译器为一个**int**类型分派存储单元，同时将其初始化为**1**并为其产生一个地址和引用捆绑在一起。存储的内容必须是常量，因为改变它没有任何意义——我们再不能对它进行操作。对于所有的临时对象，必须同样假设它们是不可存取的。当改变这种数据的时候，编译器会指出错误，这是非常有用的提示，因为这个改变会导致信息丢失。

11.2.1.2 指针引用

在C语言中，如果想改变指针本身而不是它所指向的内容，函数声明可能像这样：

```
void f(int**);
```

当传递它时，必须取得指针的地址：

```
int i = 47;
int* ip = &i;
f(&ip);
```

对于C++中的引用，语法清晰多了。函数参数变成指针的引用，用不着取得指针的地址。因此，

```
//: C11:ReferenceToPointer.cpp
#include <iostream>
```

```
using namespace std;

void increment(int*& i) { i++; }

int main() {
  int* i = 0;
  cout << "i = " << i << endl;
  increment(i);
  cout << "i = " << i << endl;
} ///:~
```

通过运行这个程序，将会看到指针本身增加了，而不是它指向的内容增加了。 454

11.2.2 参数传递准则

当给函数传递参数时，人们习惯上是通过常量引用来传递。虽然最初看起来似乎仅是出于效率考虑（通常在设计和装配程序时并不考虑效率），但像本章以后部分介绍的，这里将会带来很多的危险。拷贝构造函数需要通过传值方式来传递对象，但这并不总是可行的。

这种简单习惯可以大大提高效率：传值方式需要调用构造函数和析构函数，然而如果不想改变参数，则可通过常量引用传递，它仅需要将地址压栈。

事实上，只有一种情况不适合用传递地址方式，这就是当传值是惟一安全的途径，否则将会破坏对象时（不想修改外部对象，这不是调用者通常期望的）。这是下一节的主题。

11.3 拷贝构造函数

介绍了C++中引用的基本概念后，我们将讲述一个更令人混淆的概念：拷贝构造函数，它常被称为**X(X&)**（“X引用的X”）。在函数调用时，这个构造函数是控制通过传值方式传递和返回用户定义类型的根本所在。事实上，我们将会看到，这是很重要的，以至于编译器在没有提供拷贝构造函数时将会自动地创建。

11.3.1 按值传递和返回

为了理解拷贝构造函数的需要，看一下C语言在调用函数时处理通过按值传递和返回变量的方法。如果声明了一个函数并调用它：

```
int f(int x, char c);
int g = f(a, b);
```

455

编译器如何知道怎样传递和返回这些变量？其实它天生就知道！因为它必须处理的类型的范围是如此之小（**char**、**int**、**float**、**double**和它们的变量），这些信息都被内置在编译器中。

如果能了解编译器怎样产生汇编代码和确定调用函数**f ( )**而产生的语句，以上语句就相当于：

```
push  b
push  a
call  f()
add   sp,4
mov   g, register a
```

这个代码已被认真整理过，使之具有普遍意义；**b**和**a**的表达式根据变量是全局变量（在

这种情况下它们是_b和_a）或局部变量（编译器将在堆栈上对其索引）将有差异。g表达式也是这样。对f( )调用的形式取决于名字修饰表，“寄存器a”取决于CPU寄存器在汇编程序中是如何命名的。但不管代码如何，逻辑是相同的。

在C和C++中，参数是从右向左进栈的，然后调用函数，调用代码负责清理栈中的参数（这一点说明了**add sp,4**的作用）。但是要注意，通过按值传递方式传递参数时，编译器简单地将参数拷贝压栈——编译器知道拷贝有多大，并知道如何对参数压栈，对它们正确地拷贝。

f( )的返回值放在寄存器中。编译器同样知道返回值的类型，因为这个类型是内置于语言中的，于是编译器可以通过把返回值放在寄存器中返回它。在C的基本数据类型中，拷贝这个
456 值的位的行为就等同于拷贝这个对象。

11.3.1.1 传递和返回大对象

现在来考虑用户定义的类型。如果创建了一个类，希望通过传值方式传递该类的一个对象，编译器怎样知道做什么？这是编译器所不知的非内部数据类型，是别人创建的类型。

为了研究这个问题，首先从一个简单的结构开始，这个结构太大以至于不能在寄存器中返回：

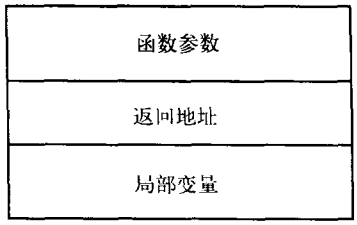
```
//: C11:PassingBigStructures.cpp
struct Big {
  char buf[100];
  int i;
  long d;
} B, B2;

Big bigfun(Big b) {
  b.i = 100; // Do something to the argument
  return b;
}

int main() {
  B2 = bigfun(B);
} ///:~
```

在这里列出汇编代码有点复杂，因为大多数编译器使用辅助（helper）函数而不是简单插入功能性的语句。在**main( )**函数中，正如我们猜测的，首先调用函数**bigfun( )**，整个**B**的内容被压栈（我们可能发现有些编译器把B的地址和大小装入寄存器，然后调用辅助函数把它压栈）。

在先前的例子中，调用函数之前要把参数压栈。然而，在**PassingBigStructures.cpp**中，将看到附加的操作：在函数调用之前，**B2**的地址压栈，虽然它明显不是一个参数。为了理解
457 这里发生的事，必须了解当编译器调用函数时对编译器的约束。

11.3.1.2 函数调用栈框架

当编译器为函数调用产生代码时，它首先把所有的参数压栈，然后调用函数。在函数内部，产生代码，向下移动栈指针为函数局部变量提供存储单元。（在这里“下”是相对的，在压栈时，机器的栈指针可能增加也可能减小。）但是在汇编语言CALL中，CPU把程序代码中的函数调用指令的地址压栈，所以汇编语言RETURN可以使用这个地址返回到调用点。当然，这个地址是非常重要的，因为没有它程序将迷失方向。这里提供一个在CALL后栈框架的样子，此时在函数中已为局部变量分配了存储单元。



函数的其他部分产生的代码希望能完全按照这个方法安排内存，因此它可以谨慎地从函数参数和局部变量中存取而不触及返回地址。我称在函数调用过程中被函数使用的这块内存为*函数框架*(*function frame*)。

另外，试图从栈中得到返回值是合理的。因为编译器简单地把返回值压栈，函数可以返回一个偏移值，它告诉返回值的开始在栈中所处的位置。

11.3.1.3 重入

因为在C和C++中的函数支持中断，所以这将出现语言重入的难题。同时，它们也支持函数递归调用。这就意味着在程序执行的任何时候，中断都可以发生而不打乱程序。当然，编写中断服务程序（ISR）的作者负责存储和还原所使用的所有的寄存器（可以把ISR看成没有 458
参数和返回值是**void**的普通函数，它存储和还原CPU的状态。有些硬件事件触发一个ISR函数的调用，而不是在程序中显式地调用）。

现在来想像一下，如果普通函数试着在堆栈中返回值，将会发生什么。因为不能触及堆栈返回地址以上任何部分，所以函数必须在返回地址以下将值压栈。但当汇编语言RETURN执行时，堆栈指针必须指向返回地址（或正好位于它下面，这取决于机器。），所以恰好在RETURN语句之前，函数必须将堆栈指针向上移动，这便清除了所有局部变量。但如果试图从堆栈中的返回地址下返回数值，因为中断可能此时发生，此时是最易被攻击的时候。这个时候ISR将向下移动堆栈指针，保存返回地址和局部变量，这样就会覆盖掉返回值。

为了解决这个问题，在调用函数之前，调用者应负责在堆栈中为返回值分配额外的存储单元。然而，C不是按照这种方法设计的，C++也一样。正如不久将看到的，C++编译器使用更有效的方案。

下一个想法可能是在全局数据区域返回数值，但这不可行。重入意味着任何函数可以中断任何其他的函数，*包括当前所处的相同函数*。因此，如果把返回值放在全局区域，可能又返回到相同的函数中，这将重写返回值。对于递归也是同样的道理。

惟一安全的返回场所是寄存器，问题是当寄存器没有用于存放返回值的足够大小时该怎么做。答案是把返回值的地址像一个函数参数一样压栈，让函数直接把返回值信息拷贝到目 459
的地。这也是在**PassingBigStructures.cpp**的**main( )**中**bigfun( )**调用之前将**B2**的地址压栈的原因。如果看了**bigfun( )**的汇编输出，可以看到它接收这个隐藏的参数并在函数内完成向目的地的拷贝。

11.3.1.4 *位拷贝与初始化*

迄今为止，一切都很顺利。对于传递和返回大的简单结构有了可使用的方法。但注意所用的方法是从一个地方向另一个地方拷贝位，这对于C考虑的变量的原始方法当然进行得很好。但在C++中，对象比一组比特位要复杂得多，因为对象具有含义。这个含义也许不能由它具有的位拷贝来很好地反映。

下面来考虑一个简单的例子：一个类在任何时候都知道它存在多少个对象。从第10章了解到可以通过包含一个静态数据成员的方法来做到这点。

```
//: C11:HowMany.cpp
// A class that counts its objects
#include <fstream>
#include <string>
using namespace std;
ofstream out("HowMany.out");

class HowMany {
  static int objectCount;
public:
  HowMany() { objectCount++; }
  static void print(const string& msg = "") {
    if(msg.size() != 0) out << msg << ": ";
    out << "objectCount = "
        << objectCount << endl;
  }
  ~HowMany() {
    objectCount--;
    print("~HowMany()");
  }
};
int HowMany::objectCount = 0;

// Pass and return BY VALUE:
HowMany f(HowMany x) {
  x.print("x argument inside f()");
  return x;
}

int main() {
  HowMany h;
  HowMany::print("after construction of h");
  HowMany h2 = f(h);
  HowMany::print("after call to f()");
} ///:~
```

460

HowMany类包括一个静态变量**int objectCount**和一个用于报告这个变量的静态成员函数**print( )**，这个函数有一个可选择的消息参数。每当一个对象产生时，构造函数增加记数，而对象销毁时，析构函数减小记数。

然而，输出并不是所期望的那样：

```
after construction of h: objectCount = 1
x argument inside f(): objectCount = 1
~HowMany(): objectCount = 0
after call to f(): objectCount = 0
~HowMany(): objectCount = -1
~HowMany(): objectCount = -2
```

在**h**生成以后，对象数是**1**，这是对的。我们希望在**f( )**调用后对象数是**2**，因为**h2**也在范围内。然而，对象数是**0**，这意味着发生了严重的错误。这从结尾两个析构函数执行后使得对象数变为负数的事实得到确认，有些事根本就不应该发生。

让我们来看一下函数**f( )**通过按值传递方式传入参数那一处。原来的对象**h**存在于函数框架之外，同时在函数体内又增加了一个对象，这个对象是通过传值方式传入的对象的拷贝。然而，参数的传递是使用C的原始的位拷贝的概念，但C++ **HowMany**类需要真正的初始化来 461
维护它的完整性。所以，默认的位拷贝不能达到预期的效果。

在对**f( )**的调用的最后，当局部对象出了其范围时，析构函数就被调用，析构函数使**objectCount**减小。所以，在函数外面，**objectCount**等于**0**。**h2**对象的创建也是用位拷贝产生的，所以，构造函数在这里也没有调用。当对象**h**和**h2**出了它们的作用范围时，它们的析构函数就使**objectCount**值变为负值。

11.3.2 拷贝构造函数

出现上述问题是因为编译器对如何从现有的对象产生新的对象进行了假定。当通过按值传递的方式传递一个对象时，就创立了一个新对象，函数体内的对象是由函数体外的原来存在的对象传递的。从函数返回对象也是同样的道理。在表达式中：

```
HowMany h2 = f(h);
```

先前未创立的对象**h2**是由函数**f( )**的返回值创建的，所以又从一个现有的对象中创建了一个新对象。

编译器假定我们想使用位拷贝来创建对象。在许多情况下，这是可行的。但在**HowMany**类中就行不通，因为初始化不是简单的拷贝。如果类中含有指针又将出现另一个问题：它们指向什么内容，是否拷贝它们或它们是否与一些新的内存块相连？

幸运的是，可以介入这个过程，并可以防止编译器进行位拷贝。每当编译器需要从现有的对象创建新对象时，可以通过定义自己的函数做这些事。因为是在创建新对象，所以，这个函数应该是构造函数，并且传递给这个函数的单一参数必须是创立的对象的源对象。但是这个对象不能通过按值传递方式传入构造函数，因为正在试图定义的函数就是为了处理按值 462
传递方式的，而且按句法传递一个指针是没有意义的，毕竟我们正在从现有的对象创建新对象。这里，引用就起作用了，可以使用源对象的引用。这个函数被称为拷贝构造函数，它经常被称为**X(X&)**（它叫做类**X**的外在表现）。

如果设计了拷贝构造函数，当从现有的对象创建新对象时，编译器将不使用位拷贝。编译器总是调用我们的拷贝构造函数。所以，如果没有设计拷贝构造函数，编译器将做一些判断，但可以选择完全接管这个过程的控制。

现在可以解决**HowMany.cpp**中的问题。

```
//: C11:HowMany2.cpp
// The copy-constructor
#include <fstream>
#include <string>
using namespace std;
ofstream out("HowMany2.out");

class HowMany2 {
  string name; // Object identifier
  static int objectCount;
public:
  HowMany2(const string& id = "") : name(id) {
```

```
    ++objectCount;
    print("HowMany2()");
  }
  ~HowMany2() {
    --objectCount;
    print("~HowMany2()");
  }
  // The copy-constructor:
  HowMany2(const HowMany2& h) : name(h.name) {
    name += " copy";
    ++objectCount;
    print("HowMany2(const HowMany2&)");
  }
  void print(const string& msg = "") const {
    if(msg.size() != 0)
463   out << msg << endl;
    out << '\t' << name << ": "
        << "objectCount = "
        << objectCount << endl;
  }
};

int HowMany2::objectCount = 0;

// Pass and return BY VALUE:
HowMany2 f(HowMany2 x) {
  x.print("x argument inside f()");
  out << "Returning from f()" << endl;
  return x;
}

int main() {
  HowMany2 h("h");
  out << "Entering f()" << endl;
  HowMany2 h2 = f(h);
  h2.print("h2 after call to f()");
  out << "Call f(), no return value" << endl;
  f(h);
  out << "After call to f()" << endl;
} ///:~
```

这儿加入一些新的方法，使我们能很好地理解发生过程。首先，当对象的信息被打印出来时，**string name**起着对象识别作用。在构造函数内，可以设置一个标识符字串（通常是对象的名字），它通过**string**构造函数拷贝至**name**中。默认值" "构造了一个空字符串。同样，构造函数将增加而析构函数减少**objectCount**的值。

其次是拷贝构造函数**HowMany2(const HowMany2&)**。拷贝构造函数可以仅从现有的对象创立新对象，所以，现有的对象的名字被拷贝给**name**，这样就能了解它是从哪里拷贝来的。如果深入了解，将会看到在构造函数的初始化表上对**name(h.name)**的调用事实上就是调用了
464 **string**拷贝构造函数。

在拷贝构造函数内部，对象数目会像普通构造函数一样的增加。这意味着当参数通过按值传递方式传递和返回时，我们能得到准确的对象数目。

print( )函数已经被修改，用于打印消息、对象标识符和对象数目。现在**print( )**函数必须

访问具体对象的**name**数据，所以不再是静态成员函数。

在**main( )**函数内部，可以看到又增加了一次函数**f( )**的调用。但这次使用了普通的C语言调用方式，且忽略了函数的返回值。既然现在知道了值是如何返回的(即在函数体内，代码处理返回过程并把结果放在目的地，目的地的地址作为一个隐藏的参数传递)。返回值被忽略将会发生什么，程序的输出将对此作出解释。

在显示输出之前,这里有一个小程序,它使用了iostreams可为任何文件加入行号。

```
//: C11:Linenum.cpp
//{T} Linenum.cpp
// Add line numbers
#include "../require.h"
#include <vector>
#include <string>
#include <fstream>
#include <iostream>
#include <cmath>
using namespace std;

int main(int argc, char* argv[]) {
  requireArgs(argc, 1, "Usage: linenum file\n"
    "Adds line numbers to file");
  ifstream in(argv[1]);
  assure(in, argv[1]);
  string line;
  vector<string> lines;
  while(getline(in, line)) // Read in entire file
    lines.push_back(line);
  if(lines.size() == 0) return 0;
  int num = 0;
  // Number of lines in file determines width:
  const int width = int(log10(lines.size())) + 1;
  for(int i = 0; i < lines.size(); i++) {
    cout.setf(ios::right, ios::adjustfield);
    cout.width(width);
    cout << ++num << ") " << lines[i] << endl;
  }
} ///:~
```

465

整个文件被读入**vector<string>**，这使用了本书前面同样的代码。当打印行号时，我们希望所有的行都能彼此对齐，这就要求在文件中调整行的数目，以使得各行号所允许的宽度是一致的。我们可以轻松地通用**vector::size( )**决定行的数目，但我们真正所需要知道的是它们是否超过了10行、100行、1000行等。如果对文件的行数取以10为底的对数，把它转为整型并再加1，这样就可得到行的最大宽度。

我们将会注意到，在**for**循环的内部有两个特殊的调用:**setf( )**和**width( )**。在这方面，**ostream**调用允许控制对齐方式和输出的宽度。但是它们必须在每一行被输出时都要调用，这也就是为什么它们被置于**for**循环内的原因。在本书的第2卷有一章是说明输出流的，它将介绍更多有关控制输出流的调用和其他的一些方法。

当**Linenum.cpp**被应用于**HowMany2.out**时，结果如下:

```
1) HowMany2()
2)   h: objectCount = 1
```

```
 3) Entering f()
 4) HowMany2(const HowMany2&)
 5)   h copy: objectCount = 2
 6) x argument inside f()
 7)   h copy: objectCount = 2
 8) Returning from f()
 9) HowMany2(const HowMany2&)
10)   h copy copy: objectCount = 3
11) ~HowMany2()
12)   h copy: objectCount = 2
13) h2 after call to f()
14)   h copy copy: objectCount = 2
15) Call f(), no return value
16) HowMany2(const HowMany2&)
17)   h copy: objectCount = 3
18) x argument inside f()
19)   h copy: objectCount = 3
20) Returning from f()
21) HowMany2(const HowMany2&)
22)   h copy copy: objectCount = 4
23) ~HowMany2()
24)   h copy: objectCount = 3
25) ~HowMany2()
26)   h copy copy: objectCount = 2
27) After call to f()
28) ~HowMany2()
29)   h copy copy: objectCount = 1
30) ~HowMany2()
31)   h: objectCount = 0
```

正如所希望的，第一件发生的事是为**h**调用普通的构造函数，对象数增加为1。但在进入函数**f( )**时，拷贝构造函数被编译器调用，完成传值过程。在**f( )**内创建了一个新对象，它是**h**的拷贝（因此被称为“**h**拷贝”），所以对象数变成2，这是拷贝构造函数的作用结果。

第8行显示了从**f( )**返回的开始情况。但在局部变量“**h**拷贝”销毁以前（在函数结尾这个局部变量便出了范围），它必须被拷入返回值，也就是**h2**。先前未创建的对象（**h2**）是从现有的对象（在函数**f( )**内的局部变量）创建的，所以在第9行拷贝构造函数当然又被使用。现在，对于**h2**的标识符，名字变成了“**h**拷贝的拷贝”。因为它是从拷贝拷过来的，这个拷贝是函数**f( )**内部对象。在对象返回之后，函数结束之前，对象数暂时变为3，但此后内部对象“**h**拷贝”被销毁。在13行完成对**f( )**的调用后，仅有2个对象**h**和**h2**。这时可以看到**h2**最终是“**h**拷贝的拷贝”。

11.3.2.1 临时对象

第15行开始调用**f(h)**，这次调用忽略了返回值。在16行可以看到恰好在参数传入之前，拷贝构造函数被调用。和前面一样，21行显示了为了返回值而调用拷贝构造函数。但是，拷贝构造函数必须有一个作为它的目的地（**this**指针）的工作地址。但这个地址从哪里获得呢？

每当编译器为了正确地计算一个表达式而需要一个临时对象时，编译器可以创建一个。在这种情况下，编译器创建一个看不见的对象作为函数**f( )**忽略了的返回值的目标地址。这个临时对象的生存期应尽可能的短，这样，空间就不会被这些等待被销毁且占用珍贵资源的临时对象搞乱。在一些情况下，临时对象可能立即传递给另外的函数。但在现在这种情况下，临时对象在函数调用之后不再需要，所以一旦函数调用完结就对内部对象调用析构函数（23

和24行），这个临时对象就被销毁（25和26行）。

在28-31行，对象**h2**被销毁了，接着对象**h**被销毁。对象记数非常正确地回到了0。

11.3.3 默认拷贝构造函数

因为拷贝构造函数实现按值传递方式的参数传递和返回，所以在这种简单结构情况下，编译器将有效地创建一个默认拷贝构造函数，这非常重要。在C中也是这样。然而，直到目前所看到的一切默认的都是原始行为：位拷贝。

当包括更复杂的类型时，如果没有创建拷贝构造函数，C++编译器也将自动地创建拷贝构造函数。然而，又一次的，位拷贝没有意义，它并不能达到我们的意图。 468

这儿有一个例子显示编译器采取的更聪明的方法。设想创建了一个新类，它是由某些现有类的对象组成的。这个创建类的方法被称为组合（*composition*），它是从现有类创建新类的方法之一。现在，假设用这个方法快速创建一个新类来解决某个问题。因为还不知道拷贝构造函数，所以没有创建它。下面的例子演示了当编译器为新类创建默认拷贝构造函数时，编译器做了哪些事。

```
//: C11:DefaultCopyConstructor.cpp
// Automatic creation of the copy-constructor
#include <iostream>
#include <string>
using namespace std;

class WithCC { // With copy-constructor
public:
  // Explicit default constructor required:
  WithCC() {}
  WithCC(const WithCC&) {
    cout << "WithCC(WithCC&)" << endl;
  }
};

class WoCC { // Without copy-constructor
  string id;
public:
  WoCC(const string& ident = "") : id(ident) {}
  void print(const string& msg = "") const {
    if(msg.size() != 0) cout << msg << ": ";
    cout << id << endl;
  }
};

class Composite {
  WithCC withcc; // Embedded objects
  WoCC wocc;
public:
  Composite() : wocc("Composite()") {}
  void print(const string& msg = "") const {
    wocc.print(msg);
  }
};

int main() {
```

469

```
  Composite c;
  c.print("Contents of c");
  cout << "Calling Composite copy-constructor"
       << endl;
  Composite c2 = c;  // Calls copy-constructor
  c2.print("Contents of c2");
} ///:~
```

类**WithCC**有一个拷贝构造函数，这个函数只是简单地宣布它被调用,这引出了一个有趣的问题。在类**Composite**中，使用默认的构造函数创建一个**WithCC**类的对象。如果在类**WithCC**中根本没有构造函数，编译器将自动地创建一个默认的构造函数。不过在这种情况下，这个构造函数什么也不做。然而，如果加了一个拷贝构造函数，我们就告诉了编译器我们将自己处理构造函数的创建，编译器将不再创建默认的构造函数，并且，除非我们显式地创建一个默认的构造函数，就如同为类**WithCC**所做的那样，否则将指示出错。

类**WoCC**没有拷贝构造函数，但它的构造函数将在内部**string**中存储一个信息，这个信息可以使用**print( )**函数打印出来。这个构造函数在类**Composite**构造函数的初始化表达式表（初始化表达式表已在第8章简单地介绍过了，并将在第14章中全面介绍）中被显式地调用。这样做的原因在稍后将会明白。

类**Composite**既含有**WithCC**类的成员对象又含有**WoCC**类的成员对象（注意因为必须如此做，内嵌的对象**WoCC**在构造函数初始化表中被初始化了）。类**Composite**没有显式定义的拷贝构造函数。然而，在**main( )**函数中，按下面的定义使用拷贝构造函数创建了一个对象。

```
Composite c2 = c;
```

类**Composite**的拷贝构造函数由编译器自动创建，程序的输出显示了它是如何被创建的。

470

```
Contents of c: Composite()
Calling Composite copy-constructor
WithCC(WithCC&)
Contents of c2: Composite()
```

为了对使用组合（和继承的方法，将在第14章介绍）的类创建拷贝构造函数，编译器递归地为所有的成员对象和基类调用拷贝构造函数。如果成员对象还含有别的对象，那么后者的拷贝构造函数也将被调用。所以，在这里，编译器也为类**WithCC**调用拷贝构造函数。程序的输出显示了这个构造函数被调用。因为**WoCC**没有拷贝构造函数，编译器为它创建一个，该拷贝构造函数仅执行了位拷贝。编译器在类**Composite**的拷贝构造函数内部调用了这个刚创建的拷贝构造函数，这可由在**main**中调用**Composite::print( )**显示出来，因为**c2.wocc**的内容与**c.wocc**内容是相同的。编译器获得一个拷贝构造函数的过程被称为*成员方法初始化*（*memberwise initialization*）。

最好的方法是创建自己的拷贝构造函数而不让编译器创建。这样就能保证程序在我们的控制之下。

11.3.4 替代拷贝构造函数的方法

现在，我们可能头已发晕了。我们可能想，怎样才能不必了解拷贝构造函数就能写一个具有一定功能的类。但是别忘了：仅当准备用按值传递的方式传递类对象时，才需要拷贝构造函数。如果不那么做时，就不需要拷贝构造函数。

11.3.4.1 防止按值传递

我们也许会说："如果我自己不写拷贝构造函数，编译器将为我创建。所以，我怎么能保证一个对象将永远不会被通过按值传递方式传递呢？"

有一个简单的技术防止通过按值传递方式传递：声明一个私有拷贝构造函数。甚至不必去定义它，除非成员函数或友元函数需要执行按值传递方式的传递。如果用户试图用按值传递方式传递或返回对象，编译器将会发出一个出错信息。这是因为拷贝构造函数是私有的。因为已显式地声明我们接管了这项工作，所以编译器不再创建默认的拷贝构造函数。例如：

471

```
//: C11:NoCopyConstruction.cpp
// Preventing copy-construction

class NoCC {
  int i;
  NoCC(const NoCC&); // No definition
public:
  NoCC(int ii = 0) : i(ii) {}
};

void f(NoCC);

int main() {
  NoCC n;
//! f(n); // Error: copy-constructor called
//! NoCC n2 = n; // Error: c-c called
//! NoCC n3(n); // Error: c-c called
} ///:~
```

注意使用的很普遍的形式

```
NoCC(const NoCC&);
```

这里使用了**const**。

11.3.4.2 改变外部对象的函数

引用语法比指针语法好用，但对于读者来说，它却使意思变得模糊。例如，在iostreams库函数中，一个重载版函数**get( )**是用一个**char&**作为参数，这个函数的作用是通过插入**get( )**的结果而改变它的参数。然而，当阅读使用这个函数的代码时，我们不会立即明白外面的对象正被改变：

```
char c;
cin.get(c);
```

472

相反,此函数调用看起来更像是按值传递方式传递，暗示着外部对象没有被改变。

正因为如此，当传递一个可被修改的参数地址时，从代码维护的观点看，使用指针可能更安全些。如果总是应用**const**引用传递地址，除非打算通过地址修改外部对象（这个地址通过非**const**指针传递），这样读者更容易读懂我们的代码。

11.4 指向成员的指针

指针是指向一些内存地址的变量，既可以是数据的地址也可以是函数的地址。所以，可以在运行时改变指针指向的内容。C++的成员指针（*pointer-to-member*）遵从同样的概念，除

了所选择的内容是在类中之内的成员指针。这里麻烦的是所有的指针需要地址，但在类内部是没有地址的；选择一个类的成员意味着在类中偏移。只有把这个偏移和具体对象的开始地址结合，才能得到实际地址。成员指针的语法要求选择一个对象的同时间接引用成员指针。

为了理解这个语法，先来考虑一个简单的结构：如果有一个这样结构的指针**sp**和对象**so**，可以通过下面方法选择成员：

```
//: C11:SimpleStructure.cpp
struct Simple { int a; };
int main() {
  Simple so, *sp = &so;
  sp->a;
  so.a;
} ///:~
```

现在，假设有一个普通的指向integer的指针**ip**。为了取得**ip**指向的内容，用一个*号间接引用这个指针。

473
```
*ip = 4;
```

最后，考虑如果有一个指向一个类对象成员的指针，如果假设它代表对象内一定的偏移，将会发生什么？为了取得指针指向的内容，必须用*号间接引用。但是，它只是一个对象内的偏移，所以必须也要指定那个对象。因此，*号要和间接引用的对象结合。所以，对于指向一个对象的指针，新的语法变为->*，对于一个对象或引用，则为.*，如下所示。

```
objectPointer->*pointerToMember = 47;
object.*pointerToMember = 47;
```

现在，让我们看看定义**pointerToMember**的语法是什么？其实它像任何一个指针，必须说出它指向什么类型。并且，在定义中也要使用一个‘*’号。惟一的区别只是它必须说出这个成员指针使用什么类的对象。当然，这是用类名和作用域运算符实现的。因此，可表示如下：

```
int ObjectClass::*pointerToMember;
```

定义一个名字为**pointerToMember**的成员指针，该指针可以指向在**ObjectClass**类中的任一**int**类型的成员。还可以在定义的时候初始化这个成员指针。

```
int ObjectClass::*pointerToMember = &ObjectClass::a;
```

因为仅仅提到了一个类而非那个类的对象，所以没有**ObjectClass::a**的确切“地址”。因而，**&ObjectClass::a**仅是作为成员指针的语法被使用。

下面例子说明了如何建立和使用指向数据成员的指针：

```
//: C11:PointerToMemberData.cpp
#include <iostream>
using namespace std;

class Data {
public:
  int a, b, c;
474
  void print() const {
    cout << "a = " << a << ", b = " << b
         << ", c = " << c << endl;
  }
```

```
};

int main() {
  Data d, *dp = &d;
  int Data::*pmInt = &Data::a;
  dp->*pmInt = 47;
  pmInt = &Data::b;
  d.*pmInt = 48;
  pmInt = &Data::c;
  dp->*pmInt = 49;
  dp->print();
} ///:~
```

显然，除了对于一些特例（即需要精确地指向的），这里就显得有些过于难用而无法随处使用。

另外，成员指针是受限制的，它们仅能被指定给在类中的确定的位置。例如，我们不能像使用普通指针那样增加或比较成员指针。

11.4.1 函数

一个类似的练习产生指向成员函数的指针语法。指向函数的指针（参见第3章的最后部分）定义如下：

```
int (*fp)(float);
```

(*fp)的圆括号用来迫使编译器正确判断定义。没有圆括号，这个表达式就是一个返回**int***值的函数。

为了定义和使用成员函数的指针，圆括号扮演同样重要的角色。假设在一个结构内有一个函数，通过给普通函数插入类名和作用域运算符就可以定义一个指向成员函数的指针。

```
//: C11:PmemFunDefinition.cpp
class Simple2 {
public:
  int f(float) const { return 1; }
};
int (Simple2::*fp)(float) const;
int (Simple2::*fp2)(float) const = &Simple2::f;
int main() {
  fp = &Simple2::f;
} ///:~
```

475

从对**fp2**定义可以看出，一个成员指针可以在它创建的时候被初始化，或者也可在其他任何时候。不像非成员函数，当获取成员函数的地址时，符号**&**不是可选的。但是，可以给出不含参数列表的函数标识符，因为重载方案可以由成员指针的类型所决定。

11.4.1.1 一个例子

在程序运行时，我们可以改变指针所指的内容。因此在运行时就可以通过指针选择或改变我们的行为，这就为程序设计提供了重要的灵活性。成员指针也一样，它允许在运行时选择一个成员。特别的，当类只有公有成员函数（数据成员通常被认为是内部实现的一部分）时，就可以用指针在运行时选择成员函数，下面的例子正是这样：

```
//: C11:PointerToMemberFunction.cpp
```

```
#include <iostream>
using namespace std;

class Widget {
public:
  void f(int) const { cout << "Widget::f()\n"; }
  void g(int) const { cout << "Widget::g()\n"; }
  void h(int) const { cout << "Widget::h()\n"; }
  void i(int) const { cout << "Widget::i()\n"; }
};

int main() {
  Widget w;
  Widget* wp = &w;
  void (Widget::*pmem)(int) const = &Widget::h;
476 (w.*pmem)(1);
  (wp->*pmem)(2);
} ///:~
```

当然，期望一般用户创建如此复杂的表达式不是很合乎情理的。如果用户必须直接操作成员指针，那么**typedef**是适合的。为了安排得当，可以使用成员指针作为内部执行机制的一部分。现在回到先前的那个在类中使用成员指针的例子上来。用户所要做的是传递一个数字以选择一个函数[⊖]。

```
//: C11:PointerToMemberFunction2.cpp
#include <iostream>
using namespace std;

class Widget {
  void f(int) const { cout << "Widget::f()\n"; }
  void g(int) const { cout << "Widget::g()\n"; }
  void h(int) const { cout << "Widget::h()\n"; }
  void i(int) const { cout << "Widget::i()\n"; }
  enum { cnt = 4 };
  void (Widget::*fptr[cnt])(int) const;
public:
  Widget() {
    fptr[0] = &Widget::f; // Full spec required
    fptr[1] = &Widget::g;
    fptr[2] = &Widget::h;
    fptr[3] = &Widget::i;
  }
  void select(int i, int j) {
    if(i < 0 || i >= cnt) return;
    (this->*fptr[i])(j);
  }
  int count() { return cnt; }
};

int main() {
  Widget w;
  for(int i = 0; i < w.count(); i++)
    w.select(i, 47);
477 } ///:~
```

⊖ 感谢Owen Mortensen提供了本例。

在类接口和**main( )**函数里，可以看到，包括函数本身在内的整个实现被隐藏了。代码甚至必须请求对函数的**Count( )**。用这个方法，类执行者可以在内部执行时改变函数的数量而不影响使用这个类的代码。

在构造函数中，成员指针的初始化似乎过分指定了。是否可以这样写:

```
fptr[1] = &g;
```

因为名字**g**在成员函数中出现，这是否可以自动地认为在这个类范围内呢？问题是这不符合成员函数的语法，它的语法要求每个人尤其编译器能够判断将要进行什么。相似地，当成员指针被间接引用时，它看起来像这样:

```
(this->*fptr[i])(j);
```

它仍是过分指定的，**this**似乎多余。正如前面所讲的，当它被间接引用时，语法也需要成员指针总是和一个对象绑定在一起。

11.5 小结

C++的指针和C中的指针是几乎相等的，这是非常好的。否则，许多C代码在C++中将不会被正确地编译。仅在出现危险赋值的地方，编译器才会产生出错信息。假设确实想这样赋值，编译器的出错可以用简单的（和显式的！）类型转换清除。

C++还从Algol和Pascal中引进引用（*reference*）的概念，引用就像一个能自动被编译器间接引用的常量指针一样。引用占有一个地址，但可以把它看成一个对象。引用是运算符重载 478
语法（下一章的主题）的重点，它也为普通函数按值传递方式传递和返回对象增加了语法上的便利。

拷贝构造函数采用相同类型的已存在对象的引用作为它的参数，它可以被用来从现有的对象创建新对象。当用按值传递方式传递或返回一个对象时，编译器自动调用这个拷贝构造函数。虽然，编译器将自动地创建一个拷贝构造函数，但是，如果认为需要有一个拷贝构造函数，应当自己定义一个，以确保完成正确的操作。如果不想通过按值传递方式传递和返回对象，应该创建一个私有的拷贝构造函数。

指向成员的指针和普通指针一样具有相同的功能：可以在运行时选取特定存储单元（数据或函数）。指向成员的指针只和类成员一起工作，而不是和全局数据或函数。通过使用指向成员的指针，我们的程序设计可以在运行时灵活地改变行为。

11.6 练习

部分练习题的答案可以在本书的电子文档 "*Annotated Solution Guide for Thinking in C++*" 中找到，只需支付很少的费用就可以从http://www.BruceEckel.com得到这个电子文档。

11-1 把本章开头的 "bird & rock" 代码段写为C程序（对数据类型使用**structs**），并对它编译，试着用C++的编译器对它进行编译，看看会有什么发生？

11-2 把标题为 "C++中的引用" 的小节的开头部分代码段放入**main( )**中，在输出时增加一些说明，以证明引用就相当于被自动间接引用的指针。

11-3 写一个程序，在其中尝试（1）创建一个引用，在其创建时没有被初始化。(2)在一个引用被初始化后，改变它的指向，使之指向另一个对象。(3)创建一个NULL引用。 479

11-4 写一个函数，该函数使用指针作为参数，修改指针所指内容，然后用引用返回指针所指的内容。

11-5 创建一个包含若干成员函数的类，再用这个类创建一个对象，该对象被练习4中的参数所指向。让这个指针是**const**的和这些成员函数是**const**的，证明仅能在自己的函数内调用**const**成员函数。让函数参数是引用而不是指针。

11-6 把标题为“指针引用”小节的开头部分的代码段写成为一段程序。

11-7 创建一个函数，使之参数为一个指向指针的指针的引用，要求该函数对其参数进行修改。然后，在**main**( )中，调用这个函数。

11-8 创建一个函数，使其用**char&**作参数并且修改该参数。在**main**( )函数里，打印一个**char**变量，使用这个变量做参数，调用我们设计的函数。然后，再次打印此变量以证明它已被改变。请问这影响了程序的可读性吗？

11-9 写一个包含了一个**const**成员函数和一个非**const**成员函数的类，再写三个使用刚创建类的对象作为参数的函数：第一个是通过按值传递方式传递参数，第二个是通过引用方式，第三个是通过**const**引用方式。在这些数的内部，试着调用所创建类的两个成员函数并解释其结果。

11-10 （有点挑战性）写一个简单的函数，该函数使用一个**int**作为其参数，增加参数的值并返回它。在**main**( )中，调用这个函数。现在观察编译器如何产生汇编代码并且通过汇编描述来追踪，以理解参数是如何被传递和返回的，以及局部变量是如何从栈中索引的。

11-11 写一个函数，该函数使用了**char**、**int**、**float**和**double**作为其参数。用编译器产生汇编代码并找出在函数调用之前把参数压入栈的指令。

480 11-12 写一个返回**double**的函数，产生汇编代码并确定该值是如何被返回的。

11-13 产生**PassingBigStructures.cpp**的汇编代码，追踪并了解编译器产生代码传送和返回大型结构的方法。

11-14 写一个简单的递归函数，该函数减少参数的值，如果参数变为0则返回0，否则调用它本身。产生这个函数的汇编代码，解释编译器创建汇编代码的过程是如何支持递归的。

11-15 编写代码用来证明当自己没有创建一个拷贝构造函数时，编译器将自动地生成拷贝构造函数。并证明生成的拷贝构造函数将对基本类型执行位拷贝，而对用户定义的类型执行拷贝构造函数。

11-16 创建一个包含拷贝构造函数的类，该类向**cout**说明它被执行。然后创建一个函数，该函数用按值传递方式传递刚创建类的一个对象。再创建一个函数，此函数产生一个刚创建类的局部对象并通过按值传递方式返回它。调用这些函数以证明当通过按值传递方式传递和返回对象时，实际上是调用了拷贝构造函数。

11-17 创建一个包含**double***的类，其构造函数通过调用**new double**来对**double***进行初始化,并将构造函数的参数中的值赋给结果存储单元。析构函数打印出所指向的值，并把该值设为－1，对存储单元调用**delete**，然后将指针置0。现在创建一个函数，该函数可通过按值传递方式获取刚创建类的一个对象。在**main**( )中调用这个函数。看看会有什么问题发生。通过创建一个拷贝构造函数来解决这个问题。

11-18 创建一个类，该类中的构造函数就像是一个拷贝构造函数，但它有一个额外的带有默认值的参数。说明这将仍然是作为拷贝构造函数被使用的。

11-19 创建一个带有能显示信息的拷贝构造函数的类。再创建第二个类，该类的成员含有一个由第一个类创建的对象，但不创建拷贝构造函数。验证第二个类中自动生成的拷贝构造函数将调用第一个类的拷贝构造函数。 481

11-20 创建一个非常简单的类和一个函数，该函数通过按值传递方式返回所创建类的一个对象。再创建第二个函数，它以一个所创建类的对象的引用为参数。作为第二个函数的参数，调用第一个函数，并说明第二个函数必须在它的参数中使用**const**引用。

11-21 创建一个没有拷贝构造函数的简单的类和一个简单的函数，此函数通过按值传递方式接收的参数是所创建类的一个对象。现在通过（仅）对拷贝构造函数增加一个私有声明来改变你的类。请解释当创建的函数被编译时将会发生什么。

11-22 本练习会创建一个拷贝构造函数的替代物。创建一个类**X**并声明（但不定义）一个私有类型拷贝构造函数。创建一个公有函数**clone( )**以作为一个**const**成员函数，该成员函数返回一个用**new**创建的对象的拷贝。现在写一个函数，使用**const X&**作参数并且复制了一个能被修改的局部拷贝。这种方法的缺点是当你这样做时，必须确保显式地销毁（使用**delete**）被复制的对象。

11-23 解释第7章中**Mem.cpp**和**MemTest.cpp**的错误，并解决其问题。

11-24 创建一个类，它含有一个**double**类型数据成员和一个打印**double**的**print( )**函数。在**main( )**中，再分别创建指向所创建类中的数据成员和函数的成员的指针。创建类的一个对象和指向该对象的一个指针，通过指向成员的指针，再使用对象和指向对象的指针，来操纵类的这两种成员。

11-25 创建包含一个整型数组的类。能否通过使用指向成员的指针对这个数组进行索引？

11-26 通过增加一个重载的成员函数**f( )**（你可以决定重载的参数表），修改**PmemFunDefinition.cpp**。再创建一个成员指针，使它指向**f ( )**的重载版本，然后通过这个指针调用此函数。在这种情况下，重载的结果会如何发生？ 482

11-27 根据第3章的**FunctionTable.cpp**，创建包含了一组函数指针的**vector**向量的类，用**add( )**和**remove( )**成员函数来增加和减少函数指针。再增加一个**run( )**函数，该函数可在**vector**中移动，并可调用所有的函数。

11-28 修改练习27，使它能够用指向成员函数的指针来完成上述工作。 483

第12章　运算符重载

运算符重载（operator overloading）只是一种“语法上的方便”（syntactic sugar），
485 *也就是说它只是另一种函数调用的方式。*

其中的不同之处在于函数的参数不是出现在圆括号内，而是紧贴在一些字符旁边，这些字符我们一般认为是不可变的运算符。

运算符的使用和普通的函数调用有两点不同。首先语法上是不同的，“调用”运算符时要把运算符放置在参数之间，有时在参数之后。第二个不同是由编译器决定调用哪一个“函数”。例如，如果对参数为浮点类型使用运算符“+”，编译器会“调用”执行浮点类型加法的函数（这种调用通常是插入内联代码，或者一段浮点处理器指令）。如果对一个浮点数和一个整数使用运算符“+”，编译器将“调用”一个特殊的函数，把**int**类型转化为**float**类型，然后再“调用”浮点加法代码。

但在C++中，可以定义一个处理类的新运算符。这种定义很像一个普通函数的定义，只是函数的名字由关键字**operator**及其后紧跟的运算符组成。差别仅此而已。它像任何其他函数一样也是一个函数，当编译器遇到适当的模式时，就会调用这个函数。

12.1　两个极端

有些人很容易滥用运算符重载。它确实是一个有趣的工具。但应注意，它仅仅只是一种*语法上的方便*，是另外一种调用函数的方式而已。从这个角度看，只有在能使涉及类的代码更易写，尤其是更易读时（请记住，读代码的机会比写代码多多了）才有理由重载运算符。如果不是这样，就不庸人自扰了。

对于运算符重载，另外一个常见的反应是恐慌：突然之间，C运算符的含义变得不同寻常了。“一切都变了，所有C代码的功能都要改变！”并非如此。在仅包含内置数据类型的表达
486 式中的所有运算符是不可能被改变的。我们不能重载如下的运算符改变其行为。

```
1 << 4;
```

或者重载运算符使得下面的表达式有意义。

```
1.414 << 2;
```

只有那些包含用户自定义类型的表达式才能有重载的运算符。

12.2　语法

定义重载的运算符就像定义函数，只是该函数的名字是**operator@**，这里@代表了被重载的运算符。函数参数表中参数的个数取决于两个因素：

1) 运算符是一元的（一个参数）还是二元的（两个参数）。

2) 运算符被定义为全局函数（对于一元是一个参数，对于二元是两个参数）还是成员函数

（对于一元没有参数，对于二元是一个参数——此时该类的对象用做左侧参数）。

下面的简单类说明了运算符重载的语法：

```
//: C12:OperatorOverloadingSyntax.cpp
#include <iostream>
using namespace std;

class Integer {
  int i;
public:
  Integer(int ii) : i(ii) {}
  const Integer
  operator+(const Integer& rv) const {
    cout << "operator+" << endl;
    return Integer(i + rv.i);
  }
  Integer&
  operator+=(const Integer& rv) {
    cout << "operator+=" << endl;
    i += rv.i;
    return *this;
  }
};

int main() {
  cout << "built-in types:" << endl;
  int i = 1, j = 2, k = 3;
  k += i + j;
  cout << "user-defined types:" << endl;
  Integer ii(1), jj(2), kk(3);
  kk += ii + jj;
} ///:~
```

487

这两个重载的运算符被定义为内联成员函数，在它们被调用时会显示信息。对于二元运算符，惟一的参数是出现在运算符右侧的那个操作数。当一元运算符被定义为成员函数时，是没有参数的。所调用的成员函数属于运算符左侧的那个对象。

对于非条件运算符（条件运算符通常返回一个布尔值），如果两个参数是相同的类型，总是希望返回相同类型的对象或引用吧（如果它们不是相同类型，结果就取决于程序设计者了）。用这种方法可以构造复杂的表达式：

```
kk += ii + jj;
```

operator +产生一个新的**Integer**（临时的），它用做**operator +=**的**rv**（右）参数。一旦这个临时变量不再需要就会销毁。

12.3 可重载的运算符

虽然几乎所有C中的运算符都可以重载，但运算符重载的使用是相当受限制的。特别是不能使用C中当前没有意义的运算符（例如用**代表求幂），不能改变运算符的优先级，不能改变运算符的参数个数。这样限制有意义，否则，所有这些行为产生的运算符只会混淆而不是澄清语意。

488

下面两个小节给出重载所有“常规”运算符的例子，重载的形式都是最可能用到的。

12.3.1 一元运算符

下面的例子显示了重载所有一元运算符的语法，有全局函数形式（非成员的友元函数）也有成员函数形式。它们将扩充前面给出的类**Integer**并且增加一个新类**byte**。具体运算符的含义取决于使用它们的方式，但在设计特殊操作之前要为未来使用这些类的程序员好好想一想。

这里是所有一元函数的目录:

```
//: C12:OverloadingUnaryOperators.cpp
#include <iostream>
using namespace std;

// Non-member functions:
class Integer {
  long i;
  Integer* This() { return this; }
public:
  Integer(long ll = 0) : i(ll) {}
  // No side effects takes const& argument:
  friend const Integer&
    operator+(const Integer& a);
  friend const Integer
    operator-(const Integer& a);
  friend const Integer
    operator~(const Integer& a);
  friend Integer*
    operator&(Integer& a);
  friend int
    operator!(const Integer& a);
  // Side effects have non-const& argument:
489 // Prefix:
  friend const Integer&
    operator++(Integer& a);
  // Postfix:
  friend const Integer
    operator++(Integer& a, int);
  // Prefix:
  friend const Integer&
    operator--(Integer& a);
  // Postfix:
  friend const Integer
    operator--(Integer& a, int);
};

// Global operators:
const Integer& operator+(const Integer& a) {
  cout << "+Integer\n";
  return a; // Unary + has no effect
}
const Integer operator-(const Integer& a) {
  cout << "-Integer\n";
  return Integer(-a.i);
}
const Integer operator~(const Integer& a) {
  cout << "~Integer\n";
  return Integer(~a.i);
```

```
}
Integer* operator&(Integer& a) {
  cout << "&Integer\n";
  return a.This(); // &a is recursive!
}
int operator!(const Integer& a) {
  cout << "!Integer\n";
  return !a.i;
}
// Prefix; return incremented value
const Integer& operator++(Integer& a) {
  cout << "++Integer\n";
  a.i++;
  return a;
}
// Postfix; return the value before increment:
const Integer operator++(Integer& a, int) {
  cout << "Integer++\n";
  Integer before(a.i);
  a.i++;
  return before;
}
// Prefix; return decremented value
const Integer& operator--(Integer& a) {
  cout << "--Integer\n";
  a.i--;
  return a;
}
// Postfix; return the value before decrement:
const Integer operator--(Integer& a, int) {
  cout << "Integer--\n";
  Integer before(a.i);
  a.i--;
  return before;
}

// Show that the overloaded operators work:
void f(Integer a) {
  +a;
  -a;
  ~a;
  Integer* ip = &a;
  !a;
  ++a;
  a++;
  --a;
  a--;
}

// Member functions (implicit "this"):
class Byte {
  unsigned char b;
public:
  Byte(unsigned char bb = 0) : b(bb) {}
  // No side effects: const member function:
  const Byte& operator+() const {
```

490

```
    cout << "+Byte\n";
    return *this;
  }
  const Byte operator-() const {
    cout << "-Byte\n";
    return Byte(-b);
  }
  const Byte operator~() const {
    cout << "~Byte\n";
    return Byte(~b);
  }
  Byte operator!() const {
    cout << "!Byte\n";
    return Byte(!b);
  }
  Byte* operator&() {
    cout << "&Byte\n";
    return this;
  }
  // Side effects: non-const member function:
  const Byte& operator++() { // Prefix
    cout << "++Byte\n";
    b++;
    return *this;
  }
  const Byte operator++(int) { // Postfix
    cout << "Byte++\n";
    Byte before(b);
    b++;
    return before;
  }
  const Byte& operator--() { // Prefix
    cout << "--Byte\n";
    --b;
    return *this;
  }
  const Byte operator--(int) { // Postfix
    cout << "Byte--\n";
    Byte before(b);
    --b;
    return before;
  }
};

void g(Byte b) {
  +b;
  -b;
  ~b;
  Byte* bp = &b;
  !b;
  ++b;
  b++;
  --b;
  b--;
}
```

```
int main() {
  Integer a;
  f(a);
  Byte b;
  g(b);
} ///:~
```

函数是根据其参数传递的方法分组的。如何传递和返回参数的指导方针到后面再讲。上面的形式（和下一小节的形式）是典型的使用形式，所以在你自己重载运算符时可以从这些范式开始。

12.3.1.1 自增和自减

重载的++和--运算符有点让人进退维谷，因为我们总是希望能根据它们出现在所作用的对象的前面（前缀）还是后面（后缀）来调用不同的函数。解决方法很简单，但有些人一开始会觉得容易混淆。例如当编译器看到**++a**（先自增）时，它就调用**operator++(a)**；但当编译器看到**a++**时，它就调用**operator++(a,int)**。即编译器通过调用不同的重载函数区别这两种形式。在成员函数版本的**OverloadingUnaryOperators.cpp**中，如果编译器看到**++b**，它就产生一个对**B::operator++( )**的调用；如果编译器看到**b++**，它就产生一个对**B::operator++(int)**的调用。

用户所见到的是对前缀和后缀版本调用不同的函数。然而，实质上这两个函数调用有着不同的标记，所以它们指向两个不同的函数体。编译器为**int**参数传递一个哑元常量值（因为这个值永远不被使用，所以它永远不会给出一个标识符）用来为后缀版产生不同的标记。

12.3.2 二元运算符

下面的清单为二元运算符重复了**OverloadingUnaryOperators.cpp**，于是就有了所有可重载运算符的例子。全局版本和成员函数版本都在里面。 493

```
//: C12:Integer.h
// Non-member overloaded operators
#ifndef INTEGER_H
#define INTEGER_H
#include <iostream>

// Non-member functions:
class Integer {
  long i;
public:
  Integer(long ll = 0) : i(ll) {}
  // Operators that create new, modified value:
  friend const Integer
    operator+(const Integer& left,
              const Integer& right);
  friend const Integer
    operator-(const Integer& left,
              const Integer& right);
  friend const Integer
    operator*(const Integer& left,
              const Integer& right);
  friend const Integer
    operator/(const Integer& left,
```

```
            const Integer& right);
friend const Integer
  operator%(const Integer& left,
            const Integer& right);
friend const Integer
  operator^(const Integer& left,
            const Integer& right);
friend const Integer
  operator&(const Integer& left,
            const Integer& right);
friend const Integer
  operator|(const Integer& left,
            const Integer& right);
friend const Integer
  operator<<(const Integer& left,
             const Integer& right);
friend const Integer
  operator>>(const Integer& left,
494          const Integer& right);
// Assignments modify & return lvalue:
friend Integer&
  operator+=(Integer& left,
             const Integer& right);
friend Integer&
  operator-=(Integer& left,
             const Integer& right);
friend Integer&
  operator*=(Integer& left,
             const Integer& right);
friend Integer&
  operator/=(Integer& left,
             const Integer& right);
friend Integer&
  operator%=(Integer& left,
             const Integer& right);
friend Integer&
  operator^=(Integer& left,
             const Integer& right);
friend Integer&
  operator&=(Integer& left,
             const Integer& right);
friend Integer&
  operator|=(Integer& left,
             const Integer& right);
friend Integer&
  operator>>=(Integer& left,
              const Integer& right);
friend Integer&
  operator<<=(Integer& left,
              const Integer& right);
// Conditional operators return true/false:
friend int
  operator==(const Integer& left,
             const Integer& right);
friend int
  operator!=(const Integer& left,
```

```
                const Integer& right);
  friend int
    operator<(const Integer& left,
              const Integer& right);
  friend int
    operator>(const Integer& left,
              const Integer& right);
  friend int
    operator<=(const Integer& left,
               const Integer& right);
  friend int
    operator>=(const Integer& left,
               const Integer& right);
  friend int
    operator&&(const Integer& left,
               const Integer& right);
  friend int
    operator||(const Integer& left,
               const Integer& right);
  // Write the contents to an ostream:
  void print(std::ostream& os) const { os << i; }
};
#endif // INTEGER_H ///:~
```

```
//: C12:Integer.cpp {O}
// Implementation of overloaded operators
#include "Integer.h"
#include "../require.h"

const Integer
  operator+(const Integer& left,
            const Integer& right) {
  return Integer(left.i + right.i);
}
const Integer
  operator-(const Integer& left,
            const Integer& right) {
  return Integer(left.i - right.i);
}
const Integer
  operator*(const Integer& left,
            const Integer& right) {
  return Integer(left.i * right.i);
}
const Integer
  operator/(const Integer& left,
            const Integer& right) {
  require(right.i != 0, "divide by zero");
  return Integer(left.i / right.i);
}
const Integer
  operator%(const Integer& left,
            const Integer& right) {
  require(right.i != 0, "modulo by zero");
  return Integer(left.i % right.i);
}
```

495

496

```
const Integer
  operator^(const Integer& left,
            const Integer& right) {
  return Integer(left.i ^ right.i);
}
const Integer
  operator&(const Integer& left,
            const Integer& right) {
  return Integer(left.i & right.i);
}
const Integer
  operator|(const Integer& left,
            const Integer& right) {
  return Integer(left.i | right.i);
}
const Integer
  operator<<(const Integer& left,
             const Integer& right) {
  return Integer(left.i << right.i);
}
const Integer
  operator>>(const Integer& left,
             const Integer& right) {
  return Integer(left.i >> right.i);
}
// Assignments modify & return lvalue:
Integer& operator+=(Integer& left,
                    const Integer& right) {
497   if(&left == &right) {/* self-assignment */}
   left.i += right.i;
   return left;
}
Integer& operator-=(Integer& left,
                    const Integer& right) {
   if(&left == &right) {/* self-assignment */}
   left.i -= right.i;
   return left;
}
Integer& operator*=(Integer& left,
                    const Integer& right) {
   if(&left == &right) {/* self-assignment */}
   left.i *= right.i;
   return left;
}
Integer& operator/=(Integer& left,
                    const Integer& right) {
   require(right.i != 0, "divide by zero");
   if(&left == &right) {/* self-assignment */}
   left.i /= right.i;
   return left;
}
Integer& operator%=(Integer& left,
                    const Integer& right) {
   require(right.i != 0, "modulo by zero");
   if(&left == &right) {/* self-assignment */}
   left.i %= right.i;
```

```
  return left;
}
Integer& operator^=(Integer& left,
                    const Integer& right) {
  if(&left == &right) {/* self-assignment */}
  left.i ^= right.i;
  return left;
}
Integer& operator&=(Integer& left,
                    const Integer& right) {
  if(&left == &right) {/* self-assignment */}
  left.i &= right.i;
  return left;
}
Integer& operator|=(Integer& left,
                    const Integer& right) {
  if(&left == &right) {/* self-assignment */}
  left.i |= right.i;
  return left;
}
Integer& operator>>=(Integer& left,
                     const Integer& right) {
  if(&left == &right) {/* self-assignment */}
  left.i >>= right.i;
  return left;
}
Integer& operator<<=(Integer& left,
                     const Integer& right) {
  if(&left == &right) {/* self-assignment */}
  left.i <<= right.i;
  return left;
}
// Conditional operators return true/false:
int operator==(const Integer& left,
               const Integer& right) {
    return left.i == right.i;
}
int operator!=(const Integer& left,
               const Integer& right) {
    return left.i != right.i;
}
int operator<(const Integer& left,
              const Integer& right) {
    return left.i < right.i;
}
int operator>(const Integer& left,
              const Integer& right) {
    return left.i > right.i;
}
int operator<=(const Integer& left,
               const Integer& right) {
    return left.i <= right.i;
}
int operator>=(const Integer& left,
               const Integer& right) {
    return left.i >= right.i;
```

498

```
}
int operator&&(const Integer& left,
               const Integer& right) {
    return left.i && right.i;
}
int operator||(const Integer& left,
               const Integer& right) {
    return left.i || right.i;
} ///:~

//: C12:IntegerTest.cpp
//{L} Integer
#include "Integer.h"
#include <fstream>
using namespace std;
ofstream out("IntegerTest.out");

void h(Integer& c1, Integer& c2) {
499   // A complex expression:
  c1 += c1 * c2 + c2 % c1;
  #define TRY(OP) \
    out << "c1 = "; c1.print(out); \
    out << ", c2 = "; c2.print(out); \
    out << ";  c1 " #OP " c2 produces "; \
    (c1 OP c2).print(out); \
    out << endl;
  TRY(+) TRY(-) TRY(*) TRY(/)
  TRY(%) TRY(^) TRY(&) TRY(|)
  TRY(<<) TRY(>>) TRY(+=) TRY(-=)
  TRY(*=) TRY(/=) TRY(%=) TRY(^=)
  TRY(&=) TRY(|=) TRY(>>=) TRY(<<=)
  // Conditionals:
  #define TRYC(OP) \
    out << "c1 = "; c1.print(out); \
    out << ", c2 = "; c2.print(out); \
    out << ";  c1 " #OP " c2 produces "; \
    out << (c1 OP c2); \
    out << endl;
  TRYC(<) TRYC(>) TRYC(==) TRYC(!=) TRYC(<=)
  TRYC(>=) TRYC(&&) TRYC(||)
}

int main() {
  cout << "friend functions" << endl;
  Integer c1(47), c2(9);
  h(c1, c2);
} ///:~

//: C12:Byte.h
// Member overloaded operators
#ifndef BYTE_H
#define BYTE_H
#include "../require.h"
#include <iostream>
// Member functions (implicit "this"):
class Byte {
  unsigned char b;
```

```
public:
  Byte(unsigned char bb = 0) : b(bb) {}
  // No side effects: const member function:
  const Byte
    operator+(const Byte& right) const {
    return Byte(b + right.b);
  }
const Byte
  operator-(const Byte& right) const {
  return Byte(b - right.b);
}
const Byte
  operator*(const Byte& right) const {
  return Byte(b * right.b);
}
const Byte
  operator/(const Byte& right) const {
  require(right.b != 0, "divide by zero");
  return Byte(b / right.b);
}
const Byte
  operator%(const Byte& right) const {
  require(right.b != 0, "modulo by zero");
  return Byte(b % right.b);
}
const Byte
  operator^(const Byte& right) const {
  return Byte(b ^ right.b);
}
const Byte
  operator&(const Byte& right) const {
  return Byte(b & right.b);
}
const Byte
  operator|(const Byte& right) const {
  return Byte(b | right.b);
}
const Byte
  operator<<(const Byte& right) const {
  return Byte(b << right.b);
}
const Byte
  operator>>(const Byte& right) const {
  return Byte(b >> right.b);
}
// Assignments modify & return lvalue.
// operator= can only be a member function:
Byte& operator=(const Byte& right) {
  // Handle self-assignment:
  if(this == &right) return *this;
  b = right.b;
  return *this;
}
Byte& operator+=(const Byte& right) {
  if(this == &right) {/* self-assignment */}
  b += right.b;
```

500

501

```
  return *this;
}
Byte& operator-=(const Byte& right) {
  if(this == &right) {/* self-assignment */}
  b -= right.b;
  return *this;
}
Byte& operator*=(const Byte& right) {
  if(this == &right) {/* self-assignment */}
  b *= right.b;
  return *this;
}
Byte& operator/=(const Byte& right) {
  require(right.b != 0, "divide by zero");
  if(this == &right) {/* self-assignment */}
  b /= right.b;
  return *this;
}
Byte& operator%=(const Byte& right) {
  require(right.b != 0, "modulo by zero");
  if(this == &right) {/* self-assignment */}
  b %= right.b;
  return *this;
}
Byte& operator^=(const Byte& right) {
  if(this == &right) {/* self-assignment */}
  b ^= right.b;
  return *this;
}
Byte& operator&=(const Byte& right) {
  if(this == &right) {/* self-assignment */}
  b &= right.b;
  return *this;
}
Byte& operator|=(const Byte& right) {
  if(this == &right) {/* self-assignment */}
  b |= right.b;
  return *this;
}
Byte& operator>>=(const Byte& right) {
  if(this == &right) {/* self-assignment */}
502
   b >>= right.b;
   return *this;
 }
 Byte& operator<<=(const Byte& right) {
   if(this == &right) {/* self-assignment */}
   b <<= right.b;
   return *this;
 }
 // Conditional operators return true/false:
 int operator==(const Byte& right) const {
     return b == right.b;
 }
 int operator!=(const Byte& right) const {
     return b != right.b;
 }
```

```
  int operator<(const Byte& right) const {
      return b < right.b;
  }
  int operator>(const Byte& right) const {
      return b > right.b;
  }
  int operator<=(const Byte& right) const {
      return b <= right.b;
  }
  int operator>=(const Byte& right) const {
      return b >= right.b;
  }
  int operator&&(const Byte& right) const {
      return b && right.b;
  }
  int operator||(const Byte& right) const {
      return b || right.b;
  }
  // Write the contents to an ostream:
  void print(std::ostream& os) const {
    os << "0x" << std::hex << int(b) << std::dec;
  }
};
#endif // BYTE_H ///:~
```

```
//: C12:ByteTest.cpp
#include "Byte.h"
#include <fstream>
using namespace std;
ofstream out("ByteTest.out");
void k(Byte& b1, Byte& b2) {
  b1 = b1 * b2 + b2 % b1;

  #define TRY2(OP) \
    out << "b1 = "; b1.print(out); \
    out << ", b2 = "; b2.print(out); \
    out << ";  b1 " #OP " b2 produces "; \
    (b1 OP b2).print(out); \
    out << endl;

  b1 = 9; b2 = 47;
  TRY2(+) TRY2(-) TRY2(*) TRY2(/)
  TRY2(%) TRY2(^) TRY2(&) TRY2(|)
  TRY2(<<) TRY2(>>) TRY2(+=) TRY2(-=)
  TRY2(*=) TRY2(/=) TRY2(%=) TRY2(^=)
  TRY2(&=) TRY2(|=) TRY2(>>=) TRY2(<<=)
  TRY2(=) // Assignment operator

  // Conditionals:
  #define TRYC2(OP) \
    out << "b1 = "; b1.print(out); \
    out << ", b2 = "; b2.print(out); \
    out << ";  b1 " #OP " b2 produces "; \
    out << (b1 OP b2); \
    out << endl;
```

503

```
  b1 = 9; b2 = 47;
  TRYC2(<) TRYC2(>) TRYC2(==) TRYC2(!=) TRYC2(<=)
  TRYC2(>=) TRYC2(&&) TRYC2(||)

  // Chained assignment:
  Byte b3 = 92;
  b1 = b2 = b3;
}

int main() {
  out << "member functions:" << endl;
  Byte b1(47), b2(9);
  k(b1, b2);
} ///:~
```

504 可以看到**operator=**只允许作为成员函数。这将在后面解释。

请注意在运算符重载中所有赋值运算符都有代码检测自赋值(self-assignment)，这是总原则。在某些情况下，这是不需要的。例如，对于**operator+=**，我们总是习惯写**A+=A**，让A与自己相加。检测自赋值最重要的地方是**operator=**，因为复杂的对象可能因为它而发生灾难性的结果（在一些情况下这不会有问题，但不管怎么说，在写**operator=**时，应该小心一些）。

在前两个例子中重载的运算符处理的是单一类型。也可以重载运算符处理混合类型，所以可以“将苹果与橙子相加”。然而，在开始进行运算符重载之前，应该看一下本章后面有关自动类型转换的一节。在适当的地方使用类型转换可以减少许多运算符重载。

12.3.3 参数和返回值

在**OverloadingUnaryOperators.cpp**、**Integer.h**和**Byte.h**例子中可以见到各种不同的参数传递和返回方法，乍一看让人有些摸不着头脑。虽然可以用任何需要的方式传递和返回参数，但在这些例子中所用的方式却不是随便选的。它们遵守一种合乎逻辑的模式，我们在大部分情况下都应选择这种模式：

1) 对于任何函数参数，如果仅需要从参数中读而不改变它，默认地应当作为**const**引用来传递它。普通算术运算符（像“+”和“-”等）和布尔运算符不会改变参数，所以以**const**引用传递是主要的使用方式。当函数是一个类成员的时候，就转换为**const**成员函数。只有会改变左侧参数的运算符赋值(operator-assignment)（如“+”、“=”）和**operator=**，左侧参数不是常量，但因为参数将被改变，所以参数仍然按地址传递。

505 2) 返回值的类型取决于运算符的具体含义（我们可以对参数和返回值做任何想做的事）。如果使用该运算符的结果是产生一个新值，就需要产生一个作为返回值的新对象。例如，**Integer::operator+**必须生成一个操作数之和的**Integer**对象。这个对象作为一个常量通过传值方式返回，所以作为一个左值结果不会被改变。

3) 所有赋值运算符均改变左值。为了使赋值结果能用于链式表达式（如**a=b=c**），应该能够返回一个刚刚改变了的左值的引用。但这个引用应该是常量还是非常量呢？虽然我们是从左向右读表达式**a=b=c**，但编译器是从右向左分析这个表达式，所以并非一定要返回一个非常量值来支持链式赋值。然而人们有时希望能够对刚刚赋值的对象进行运算，例如**(a=b).func( )**，这是**b**赋值给**a**后调用**func ( )**。因此所有赋值运算符的返回值对于左值应该

是非常量引用。

4) 对于逻辑运算符，人们希望至少得到一个**int**返回值，最好是**bool**返回值。(在大多数编译器支持C++内置**bool**类型之前开发的库函数使用**int**或者用**typedef**产生的等价类型)。

因为有前缀和后缀版本，所以自增和自减运算符出现了两难局面。由于两个版本都改变了对象，所以这个对象不能作为常量类型。在对象被改变后，前缀版本返回其值，我们希望返回改变后的对象。这样，用前缀版本只需作为一个引用返回***this**。因为后缀版本返回改变之前的值，所以必须创建一个代表这个值的独立对象并返回它。因此，如果想保持本意，对于后缀必须通过传值方式返回。(注意，我们经常会发现自增和自减运算返回一个**int** 值或**bool**值，表示诸如是否在列表上移动的对象到达了列表尾部这样的情况)。现在的问题是：它们应该按常量还是按非常量返回？如果允许对象被改变，而有的人写了表达式 506
(++a).func( )，那么**func( )**作用在**a**上。但对于表达式**(a++).func( ), func( )**作用在通过后缀**operator++**返回的临时对象上。临时对象自动定为常量，所以这一操作会被编译器阻止。但为了一致性，两者都是常量更有意义，这里就是如此。我们可以选择让前缀版本是非常量的，而后缀版本是常量的。因为想给自增和自减运算符赋予各种含义，所以它们需要就事论事考虑。

12.3.3.1 作为常量通过传值方式返回

作为常量通过传值方式返回，开始看起来有些微妙，所以值得多加解释。现在考虑二元运算符+。假设在表达式**f(a+b)**中使用它，**a+b**的结果变为一个临时对象，这个对象用于对**f( )**的调用中。因为它是临时的，自动被定为常量，所以无论是否使返回值为常量都没有影响。

然而，也可能发送一个消息给**a+b**的返回值而不是仅传递给一个函数。例如，可以写表达式**(a+b).g( )**，其中**g( )**是**Integer**的成员函数。这里，通过设返回值为常量，规定了对于返回值只有常量成员函数才可以被调用。用常量是恰当的，这是因为这样可以使我们不用在对象中存储可能有价值的信息，而该信息很可能是会被丢失的。

12.3.3.2 返回值优化

通过传值方式返回要创建新对象时，应注意使用的形式。例如在**operator+**:

```
return Integer(left.i + right.i);
```

乍看起来这像是一个“对一个构造函数的调用”，其实并非如此。这是临时对象语法，它是在说：“创建一个临时**Integer**对象并返回它”。据此我们可能认为如果创建一个有名字的局部对象并返回它结果将会是一样的。其实不然。如果如下编写， 507

```
Integer tmp(left.i + right.i);
return tmp;
```

将发生三件事。首先，创建**tmp**对象，其中包括构造函数的调用。然后，拷贝构造函数把**tmp**拷贝到外部返回值的存储单元里。最后，当**tmp**在作用域的结尾时调用析构函数。

相反，“返回临时对象”的方式是完全不同的。当编译器看到我们这样做时，它明白对创建的对象没有其他需求，只是返回它，所以编译器直接地把这个对象创建在外部返回值的内存单元。因为不是真正创建一个局部对象，所以仅需要一个普通构造函数调用（不需要拷贝构造函数），且不会调用析构函数。这种方法不需要什么花费，因此效率是非常高的，但程序员要理解这些。这种方式常被称作返回值优化（*return value optimization*）。

12.3.4 不常用的运算符

还有一些运算符的重载语法有一点不同。

下标运算符**operator[ ]**，必须是成员函数并且它只接受一个参数。因为它所作用的对象应该像数组一样操作，可以经常从这个运算符返回一个引用，所以它可以被很方便地用于等号左侧。这个运算符经常被重载，可以在本书其他部分看到相关的例子。

运算符**new** 和**delete** 用于控制动态存储分配并能按许多种不同的方法进行重载，这将在第13章中讨论。

12.3.4.1 operator,

当逗号出现在一个对象左右，而该对象的类型是逗号定义所支持的类型时，将调用逗号
508 运算符。然而，“**operator,**”调用的目标不是函数参数表，而是被逗号分隔开的、没有被括号括起来的对象。除了使语言保持一致性外，这个运算符似乎没有许多实际用途。下面的例子说明了当逗号出现在对象前面以及后面时，逗号函数调用的方式：

```
//: C12:OverloadingOperatorComma.cpp
#include <iostream>
using namespace std;

class After {
public:
  const After& operator,(const After&) const {
    cout << "After::operator,()" << endl;
    return *this;
  }
};

class Before {};

Before& operator,(int, Before& b) {
  cout << "Before::operator,()" << endl;
  return b;
}

int main() {
  After a, b;
  a, b;  // Operator comma called

  Before c;
  1, c;  // Operator comma called
} ///:~
```

全局函数允许逗号放在被讨论的对象的前面。这里的用法既晦涩又可疑。虽然还可以再把一个逗号分隔的参数表当做更加复杂的表达式的一部分，但这太灵活了，大多数情况下不能使用。

12.3.4.2 operator->

当希望一个对象表现得像一个指针时，通常就要用到**operator->**。由于这样一个对象比一般的指针有着更多与生俱来的灵巧性，于是常被称作*灵巧指针*（*smart pointer*）。如果想用
509 类包装一个指针以使指针安全，或是在*迭代器*（*iterator*）普通的用法中，这样做会特别有用。迭代器是一个对象，这个对象可以作用于其他对象的容器或集合上，每次选择它们中的一个，

而不用提供对容器的直接访问。(在类函数里经常发现容器和迭代器，例如本书第2卷中描述的标准C++库。)

指针间接引用运算符一定是一个成员函数。它有着额外的、非典型的限制：它必须返回一个对象（或对象的引用），该对象也有一个指针间接引用运算符；或者必须返回一个指针，被用于选择指针间接引用运算符箭头所指向的内容。下面是一个简单的例子：

```
//: C12:SmartPointer.cpp
#include <iostream>
#include <vector>
#include "../require.h"
using namespace std;

class Obj {
  static int i, j;
public:
  void f() const { cout << i++ << endl; }
  void g() const { cout << j++ << endl; }
};

// Static member definitions:
int Obj::i = 47;
int Obj::j = 11;

// Container:
class ObjContainer {
  vector<Obj*> a;
public:
  void add(Obj* obj) { a.push_back(obj); }
  friend class SmartPointer;
};

class SmartPointer {
  ObjContainer& oc;
  int index;
public:
  SmartPointer(ObjContainer& objc) : oc(objc) {
    index = 0;
  }
  // Return value indicates end of list:
  bool operator++() { // Prefix
    if(index >= oc.a.size()) return false;
    if(oc.a[++index] == 0) return false;
    return true;
  }
  bool operator++(int) { // Postfix
    return operator++(); // Use prefix version
  }
  Obj* operator->() const {
    require(oc.a[index] != 0, "Zero value "
      "returned by SmartPointer::operator->()");
    return oc.a[index];
  }
};
```

510

```
int main() {
  const int sz = 10;
  Obj o[sz];
  ObjContainer oc;
  for(int i = 0; i < sz; i++)
    oc.add(&o[i]); // Fill it up
  SmartPointer sp(oc); // Create an iterator
  do {
    sp->f(); // Pointer dereference operator call
    sp->g();
  } while(sp++);
} ///:~
```

类**Obj**定义了程序中使用的一些对象。函数**f( )**和**g( )**用静态数据成员打印令人感兴趣的值。使用**ObjContainer**的函数**add( )**将指向这些对象的指针存储在类型为**ObjContainer**的容器中。**ObjContainer**看起来像一个指针数组，但却没有办法取回这些指针。然而，类**SmartPointer**被声明为友元类，所以它允许进入这个容器内。类**SmartPointer**看起来像一个聪明的指针——可以使用运算符++向前移动它（也可以定义一个**operator--**），它不会超出它所指向的容器的
511 范围，它可以返回它指向的内容（通过这个指针间接引用运算符）。注意，不像一个基本指针，**SmartPointer** 是和所创建的容器的配套使用的，不存在一个具有“通用目的”的灵巧指针。我们将在本书最后一章和第2卷中了解更多被称为“迭代器”的灵巧指针的内容。

在**main( )**中，一旦**Obj**对象装入容器**oc**，一个**SmartPointer**类的**SP**就创建了。灵巧指针按下面的表达式进行调用：

```
sp->f(); // Smart pointer calls
sp->g();
```

这里，尽管**sp**实际上并没有成员函数**f( )**和**g( )**，但指针间接引用运算符自动地为用**SmartPointer::operator->**返回的**Obj***调用那些函数。编译器进行所有检查以保证函数调用正确。

虽然，指针间接运算符的底层机制比其他运算符复杂一些，但目的是一样的——为类的用户提供更为方便的语法。

12.3.4.3 嵌入的迭代器

更常见的是，“灵巧指针”和“迭代器”类嵌入它所服务的类中。前面的例子可按如下重写，以在**ObjContainer**中嵌入**SmartPointer**。

```
//: C12:NestedSmartPointer.cpp
#include <iostream>
#include <vector>
#include "../require.h"
using namespace std;

class Obj {
  static int i, j;
public:
  void f() { cout << i++ << endl; }
  void g() { cout << j++ << endl; }
512 };
// Static member definitions:
int Obj::i = 47;
```

```
int Obj::j = 11;

// Container:
class ObjContainer {
  vector<Obj*> a;
public:
  void add(Obj* obj) { a.push_back(obj); }
  class SmartPointer;
  friend SmartPointer;
  class SmartPointer {
    ObjContainer& oc;
    unsigned int index;
  public:
    SmartPointer(ObjContainer& objc) : oc(objc) {
      index = 0;
    }
    // Return value indicates end of list:
    bool operator++() { // Prefix
      if(index >= oc.a.size()) return false;
      if(oc.a[++index] == 0) return false;
      return true;
    }
    bool operator++(int) { // Postfix
      return operator++(); // Use prefix version
    }
    Obj* operator->() const {
      require(oc.a[index] != 0, "Zero value "
        "returned by SmartPointer::operator->()");
      return oc.a[index];
    }
  };
  // Function to produce a smart pointer that
  // points to the beginning of the ObjContainer:
  SmartPointer begin() {
    return SmartPointer(*this);
  }
};

int main() {
  const int sz = 10;
  Obj o[sz];
  ObjContainer oc;
  for(int i = 0; i < sz; i++)
    oc.add(&o[i]); // Fill it up
  ObjContainer::SmartPointer sp = oc.begin();
  do {
    sp->f(); // Pointer dereference operator call
    sp->g();
  } while(++sp);
} ///:~
```

513

除了实际上嵌入了类中，另有两点不同之处。首先是在类的声明中说明它是一个友元类。

```
class SmartPointer;
friend SmartPointer;
```

编译器首先在被告知类是友元的之前，必须知道该类是存在的。

第二个不同之处是在**ObjContainer**的成员函数**begin( )**中，**begin( )**产生一个指向**ObjContainer**序列开头的**SmartPointer**。虽然实际上仅是方便了，但由于它遵循了在标准C++库中使用的部分形式，所以还是值得的。

12.3.4.4 **operator->***

operator->*是一个二元运算符，其行为与所有其他二元运算符类似。它是专为模仿前一章介绍的内部数据类型的成员指针行为而提供的。

与**operator->***一样，指向成员的指针间接引用运算符通常同某种代表“灵巧指针”的对象一起使用。这里的例子将简单些以便于理解。在定义**operator->***时要注意它必须返回一个对象，对于这个对象，可以用正在调用的成员函数为参数调用**operator( )**。

operator( )的函数调用必须是成员函数，它是惟一的允许在它里面有任意个参数的函数。
514 这使得对象看起来像一个真正的函数。虽然可以定义一些重载的带不同参数的**operator( )**函数，但这常被用于仅有一个单一操作数或至少是一个特别优先的类型。在第2卷中，可以看到标准C++库使用函数调用运算符以创建“函数对象”。

要想创建一个**operator->***，必须首先创建带有**operator( )**类，这是**operator->***将返回对象的类。该类必须获取一些必要的信息，以使当**operator( )**被调用时，指向成员的指针可以对对象进行间接引用。在下面的例子中，**FunctionObject**的构造函数得到并储存指向对象的指针和指向成员函数的指针，然后**operator( )**使用这些指针进行实际指向成员的指针的调用。

```
//: C12:PointerToMemberOperator.cpp
#include <iostream>
using namespace std;

class Dog {
public:
  int run(int i) const {
    cout << "run\n";
    return i;
  }
  int eat(int i) const {
     cout << "eat\n";
     return i;
  }
  int sleep(int i) const {
    cout << "ZZZ\n";
    return i;
  }
  typedef int (Dog::*PMF)(int) const;
  // operator->* must return an object
  // that has an operator():
  class FunctionObject {
    Dog* ptr;
    PMF pmem;
  public:
    // Save the object pointer and member pointer
    FunctionObject(Dog* wp, PMF pmf)
      : ptr(wp), pmem(pmf) {
515   cout << "FunctionObject constructor\n";
    }
    // Make the call using the object pointer
```

```
    // and member pointer
    int operator()(int i) const {
      cout << "FunctionObject::operator()\n";
      return (ptr->*pmem)(i); // Make the call
    }
  };
  FunctionObject operator->*(PMF pmf) {
    cout << "operator->*" << endl;
    return FunctionObject(this, pmf);
  }
};

int main() {
  Dog w;
  Dog::PMF pmf = &Dog::run;
  cout << (w->*pmf)(1) << endl;
  pmf = &Dog::sleep;
  cout << (w->*pmf)(2) << endl;
  pmf = &Dog::eat;
  cout << (w->*pmf)(3) << endl;
} ///:~
```

Dog有三个成员函数，它们的参数和返回类型都是**int**。**PMF**是一个**typedef**，用于简化定义一个指向**Dog**成员函数的指向成员的指针。

operator->*创建并返回一个**FunctionObject**对象。注意**operator->***既知道指向成员的指针所调用的对象（**this**），又知道这个指向成员的指针，并把它们传递给存储这些值的**FunctionObject**构造函数。当**operator->***被调用时，编译器立刻转而对**operator->***返回的值调用**operator()**，把已经给**operator->***的参数传递进去。**FunctionObject::operator()**得到参数，然后使用存储的对象指针和指向成员的指针间接引用“真实的”指向成员的指针。

注意，正如**operator->**，这里操作的内容正插入到调用**operator->***的中间。如果需要的话，这允许我们执行某些额外的操作。 516

此处执行的**operator->***机制仅作用于参数和返回值是**int**的成员函数。这是一个局限，但是，如果试着为每一个不同的可能性进行重载，这就像是一个禁止的行为。幸运的是，C++的**template**机制（在本书的最后一章和第2卷中讲述）被设计用来处理这个问题。

12.3.5 不能重载的运算符

在可用的运算符集合里存在一些不能重载的运算符。这样限制的通常原因是出于对安全的考虑：如果这些运算符也可以被重载，将会造成危害或破坏安全机制，使得事情变得困难或混淆现有的习惯。

1) 成员选择**operator.**。点在类中对任何成员都有一定的意义。但如果允许它重载，就不能用普通的方法访问成员，只能用指针和指针**operator->**访问。
2) 成员指针间接引用**operator.***，因为与**operator.**同样的原因而不能重载。
3) 没有求幂运算符。通常的选择是来自Fotran语言的**operator****，但这出现了难以分析的问题。C没有求幂运算符，C++似乎也不需要，因为这可以通过函数调用来实现。求幂运算符增加了使用的方便，但没有增加新的语言功能，反而为编译器增加了复杂性。
4) 不存在用户定义的运算符，即不能编写目前运算符集合中没有的运算符。不能这样做

517 的部分原因是难以决定其优先级，另一部分原因是没有必要增加麻烦。

5) 不能改变优先级规则。否则人们很难记住它们。

12.4 非成员运算符

在前面的一些例子里，运算符可能是成员运算符或非成员运算符，这似乎没有多大差异。这样就会出现一个问题："应该选择哪一种？"总的来说，如果没有什么差异，它们应该是成员运算符。这样做强调了运算符和类的联合。当左侧操作数是当前类的对象时，运算符会工作得很好。

但有时左侧运算符是别的类的对象。这种情况通常出现在为输入输出流重载**operator<<**和**>>**时。因为输入输出流是一个基本C++库，我们将有可能想为定义的大部分类重载运算符，所以这个过程是值得记住的：

```
//: C12:IostreamOperatorOverloading.cpp
// Example of non-member overloaded operators
#include "../require.h"
#include <iostream>
#include <sstream> // "String streams"
#include <cstring>
using namespace std;

class IntArray {
  enum { sz = 5 };
  int i[sz];
public:
  IntArray() { memset(i, 0, sz* sizeof(*i)); }
  int& operator[](int x) {
    require(x >= 0 && x < sz,
      "IntArray::operator[] out of range");
    return i[x];
  }
  friend ostream&
    operator<<(ostream& os, const IntArray& ia);
518 friend istream&
    operator>>(istream& is, IntArray& ia);
};

ostream&
operator<<(ostream& os, const IntArray& ia) {
  for(int j = 0; j < ia.sz; j++) {
    os << ia.i[j];
    if(j != ia.sz -1)
      os << ", ";
  }
  os << endl;
  return os;
}

istream& operator>>(istream& is, IntArray& ia){
  for(int j = 0; j < ia.sz; j++)
    is >> ia.i[j];
  return is;
}
```

```
int main() {
  stringstream input("47 34 56 92 103");
  IntArray I;
  input >> I;
  I[4] = -1; // Use overloaded operator[]
  cout << I;
} ///:~
```

这个类还包含重载**operator[ ]**，这个运算符在数组里返回了一个合法值的引用。因为一个引用被返回，所以下面的表达式:

```
I[4] = -1;
```

看起来不仅比使用指针更规范些，而且它也达到了预期的效果。

被重载的移位运算符通过引用方式传递和返回，所以运算将影响外部对象。在函数定义中，表达式如

```
os << ia.i[j];
```

519

会使现有的重载运算符函数被调用（即那些定义在**<iostream>**中的）。在这种情况下，被调用的函数是**ostream& operator<<(ostream&,int)**，这是因为**ia.i[j]**是一个**int**值。

一旦所有的动作在**istream**或**ostream**上完成，它将被返回，因此它可被用于更复杂的表达式。

在**main( )**中，**iostream**的一种新类型被使用: **stringstream**（在**<sstream>**中声明）。该类包含一个**string**（正如此处显示的，它可以由一个**char**数组创建）并且把它转化为一个**iostream**。在上面的例子中，这意味着在不打开一个文件或在命令行中键入数据的情况下，移位运算符可以被测试。

这个例子使用的是插入符和提取符的标准形式。如果想为自己的类创建一个集合，可以拷贝这个函数署名并返回以上的类型，且遵从该函数体的形式。

12.4.1 基本方针

Murray[⊖]为在成员和非成员之间的选择提出了如下的方针:

运算符	建议使用
所有的一元运算符	成员
= ( ) [ ] -> ->*	必须是成员
+= -= /= *= ^= &= |= %= >>= <<=	成员
所有其他二元运算符	非成员

520

12.5 重载赋值符

赋值符常引起C++程序员初学者的混淆。这是毫无疑问的，因为‘=’在编程中是最基本的运算符，是在机器层上拷贝寄存器。另外，当使用‘=’时也能引起拷贝构造函数（上一章内容）调用:

⊖ 参见Rob Murray所著《*C++ Strategies & Tactics*》(Addison-Wesley)，1993, 第47页。

```
MyType b;
MyType a = b;
a = b;
```

第2行定义了对象**a**。一个新对象先前不存在，现在正被创建。因为现在知道了C++编译器关于对象初始化是如何保护的，所以知道在对象被定义的地方构造函数总是必须被调用。但是是调用哪个构造函数呢？**a**是从现有的**MyType**对象创建的（**b**在等号的右侧），所以只有一个选择：拷贝构造函数。所以虽然这里包括一个等号，但拷贝构造函数仍被调用。

第3行情况就不同了。在等号左侧有一个以前初始化了的对象。很清楚，不用为一个已经存在的对象调用构造函数。在这种情况下，为**a**调用**MyType::operator =**，把出现在右侧的任何东西作为参数（可以有多种取不同右侧参数的**operator=**函数）。

对于拷贝构造函数则没有这个限制。在任何时候使用一个“=”代替普通形式的构造函数调用来初始化一个对象时，无论等号右侧是什么，编译器都会寻找一个接受右边类型的构造函数：

```
//: C12:CopyingVsInitialization.cpp
class Fi {
public:
  Fi() {}
};
class Fee {
public:
  Fee(int) {}
  Fee(const Fi&) {}
};

int main() {
  Fee fee = 1; // Fee(int)
  Fi fi;
  Fee fum = fi; // Fee(Fi)
} ///:~
```

521

当处理“=”时，记住这个差别是非常重要的：如果对象还没有被创建，初始化是需要的，否则使用赋值**operator=**。

对于初始化，使用“=”可以避免写代码。不用总是用显式的构造函数形式。等号的两种构造形式变为：

```
Fee fee(1);
Fee fum(fi);
```

这个方法可以避免使读者混淆。

12.5.1 operator=的行为

在**Integer.h** 和 **Byte.h**中，可以看到**operator=**仅是成员函数，它密切地与“=”左侧的对象相联系。如果允许定义**operator=**为全局的，那么我们就会试图重新定义内置的“=”：

```
int operator=(int, MyType); // Global = not allowed!
```

编译器通过强制**operator=**为成员函数而避开这个问题。

当创建一个**operator=**时，必须从右侧对象中拷贝所有需要的信息到当前的对象（即调用

运算符的对象）以完成为类的“赋值”，对于简单的对象，这是显然的: 522

```
//: C12:SimpleAssignment.cpp
// Simple operator=()
#include <iostream>
using namespace std;

class Value {
  int a, b;
  float c;
public:
  Value(int aa = 0, int bb = 0, float cc = 0.0)
    : a(aa), b(bb), c(cc) {}
  Value& operator=(const Value& rv) {
    a = rv.a;
    b = rv.b;
    c = rv.c;
    return *this;
  }
  friend ostream&
  operator<<(ostream& os, const Value& rv) {
    return os << "a = " << rv.a << ", b = "
      << rv.b << ", c = " << rv.c;
  }
};

int main() {
  Value a, b(1, 2, 3.3);
  cout << "a: " << a << endl;
  cout << "b: " << b << endl;
  a = b;
  cout << "a after assignment: " << a << endl;
} ///:~
```

这里，“=”左侧的对象拷贝了右侧对象中的所有内容，然后返回它的引用，所以还可以创建更加复杂的表达式。

这个例子犯了一个常见的错误。当准备给两个相同类型的对象赋值时，应该首先检查一下自赋值(self-assignment)：这个对象是否对自身赋值了？在一些情况下，例如本例，无论如何执行这些赋值运算都是无害的，但如果对类的实现进行了修改，那么将会出现差异。如果我们习惯于不做检查，就可能忘记并产生难以发现的错误。 523

12.5.1.1 类中指针

如果对象不是如此简单时将会发生什么问题？例如，如果对象里包含指向别的对象的指针将如何？简单地拷贝一个指针意味着以指向相同的存储单元的对象而结束。在这种情况下，就需要自己做簿记。

这里有两个解决办法。当做一个赋值运算或一个拷贝构造函数时，最简单的方法是拷贝这个指针所涉及的一切，这是非常直接的。

```
//: C12:CopyingWithPointers.cpp
// Solving the pointer aliasing problem by
// duplicating what is pointed to during
// assignment and copy-construction.
#include "../require.h"
```

```
#include <string>
#include <iostream>
using namespace std;

class Dog {
  string nm;
public:
  Dog(const string& name) : nm(name) {
    cout << "Creating Dog: " << *this << endl;
  }
  // Synthesized copy-constructor & operator=
  // are correct.
  // Create a Dog from a Dog pointer:
  Dog(const Dog* dp, const string& msg)
    : nm(dp->nm + msg) {
    cout << "Copied dog " << *this << " from "
         << *dp << endl;
  }
  ~Dog() {
    cout << "Deleting Dog: " << *this << endl;
  }
  void rename(const string& newName) {
    nm = newName;
    cout << "Dog renamed to: " << *this << endl;
  }
  friend ostream&
524 operator<<(ostream& os, const Dog& d) {
    return os << "[" << d.nm << "]";
  }
};

class DogHouse {
  Dog* p;
  string houseName;
public:
  DogHouse(Dog* dog, const string& house)
   : p(dog), houseName(house) {}
  DogHouse(const DogHouse& dh)
    : p(new Dog(dh.p, " copy-constructed")),
      houseName(dh.houseName
        + " copy-constructed") {}
  DogHouse& operator=(const DogHouse& dh) {
    // Check for self-assignment:
    if(&dh != this) {
      p = new Dog(dh.p, " assigned");
      houseName = dh.houseName + " assigned";
    }
    return *this;
  }
  void renameHouse(const string& newName) {
    houseName = newName;
  }
  Dog* getDog() const { return p; }
  ~DogHouse() { delete p; }
  friend ostream&
  operator<<(ostream& os, const DogHouse& dh) {
```

```
    return os << "[" << dh.houseName
      << "] contains " << *dh.p;
  }
};

int main() {
  DogHouse fidos(new Dog("Fido"), "FidoHouse");
  cout << fidos << endl;
  DogHouse fidos2 = fidos; // Copy construction
  cout << fidos2 << endl;
  fidos2.getDog()->rename("Spot");
  fidos2.renameHouse("SpotHouse");
  cout << fidos2 << endl;
  fidos = fidos2; // Assignment
  cout << fidos << endl;
  fidos.getDog()->rename("Max");
  fidos2.renameHouse("MaxHouse");
} ///:~
```

525

Dog是一个简单的类，仅包含一个用来说明**dog**名字的**string**成员。但是，由于构造函数和析构函数在它们被调用的时候打印信息，所以就可以知道对**Dog**进行操作的时间。注意这第二个构造函数有点像拷贝构造函数，除了它的参数是一个指向**Dog**对象的指针而不是一个引用外，并且它还有第二个参数，是同**Dog**参数的名字相关联的信息。这被用于帮助追踪程序的执行。

可以看到，无论何时成员函数打印信息，它都不是直接获取这些信息的，而是把***this** 传送给**cout**。进而调用**ostream operator<<**。用这种方式进行操作是值得的，因为如果想重新格式化**Dog**信息的显示方式（正如通过增加“[”和“]”所做的），仅需要在一处进行操作。

当类中包含指针时，**DogHouse**含有一个**Dog***并说明了需要定义的4个函数：所有必需的普通构造函数、拷贝构造函数、**operator=**（无论定义它还是不允许它）和析构函数。对**operator=**当然要检查自赋值，虽然这儿并不一定需要，这实际上减少了改变代码而忘记检查自赋值的可能性。

12.5.1.2 引用计数

在上面的例子中，拷贝构造函数和**operator=**对指针所指向的内容作了一个新的拷贝，并由析构函数删除它。但是，如果对象需要大量的内存或过高的初始化，我们也许想避免这种拷贝。解决这个问题的通常方法称为*引用计数*(*reference counting*)。可以使一块存储单元具有智能，它知道有多少对象指向它。拷贝构造函数或赋值运算意味着把另外的指针指向现在的存储单元并增加引用记数。消除意味着减小引用记数，如果引用记数为0则意味销毁这个对象。

526

但如果向这个对象（上例中的**Dog**）执行写入操作将会如何呢？因为不止一个对象使用这个**Dog**，所以当修改自己的**Dog**时，也等于也修改了他人的**Dog**。为了解决这个“别名”问题，经常使用另外一个称为*写拷贝*(*copy-on-write*)的技术。在向这块存储单元写之前，应该确信没有其他人使用它。如果引用记数大于1，在写之前必须拷贝这块存储单元，这样就不会影响他人了。这儿提供了一个简单的引用记数和关于写拷贝的例子：

```
//: C12:ReferenceCounting.cpp
// Reference count, copy-on-write
#include "../require.h"
#include <string>
#include <iostream>
```

```
using namespace std;

class Dog {
  string nm;
  int refcount;
  Dog(const string& name)
    : nm(name), refcount(1) {
    cout << "Creating Dog: " << *this << endl;
  }
  // Prevent assignment:
  Dog& operator=(const Dog& rv);
public:
  // Dogs can only be created on the heap:
  static Dog* make(const string& name) {
    return new Dog(name);
  }
  Dog(const Dog& d)
    : nm(d.nm + " copy"), refcount(1) {
    cout << "Dog copy-constructor: "
         << *this << endl;
  }
  ~Dog() {
    cout << "Deleting Dog: " << *this << endl;
  }
  void attach() {
    ++refcount;
    cout << "Attached Dog: " << *this << endl;
  }
  void detach() {
    require(refcount != 0);
    cout << "Detaching Dog: " << *this << endl;
    // Destroy object if no one is using it:
    if(--refcount == 0) delete this;
  }
  // Conditionally copy this Dog.
  // Call before modifying the Dog, assign
  // resulting pointer to your Dog*.
  Dog* unalias() {
    cout << "Unaliasing Dog: " << *this << endl;
    // Don't duplicate if not aliased:
    if(refcount == 1) return this;
    --refcount;
    // Use copy-constructor to duplicate:
    return new Dog(*this);
  }
  void rename(const string& newName) {
    nm = newName;
    cout << "Dog renamed to: " << *this << endl;
  }
  friend ostream&
  operator<<(ostream& os, const Dog& d) {
    return os << "[" << d.nm << "], rc = "
      << d.refcount;
  }
};
```

527

```
class DogHouse {
  Dog* p;
  string houseName;
public:
  DogHouse(Dog* dog, const string& house)
   : p(dog), houseName(house) {
    cout << "Created DogHouse: "<< *this << endl;
  }
  DogHouse(const DogHouse& dh)
    : p(dh.p),
      houseName("copy-constructed " +
        dh.houseName) {
    p->attach();
    cout << "DogHouse copy-constructor: "
         << *this << endl;
  }
  DogHouse& operator=(const DogHouse& dh) {
    // Check for self-assignment:
    if(&dh != this) {
      houseName = dh.houseName + " assigned";
      // Clean up what you're using first:
      p->detach();
      p = dh.p; // Like copy-constructor
      p->attach();
    }
    cout << "DogHouse operator= : "
         << *this << endl;
    return *this;
  }
  // Decrement refcount, conditionally destroy
  ~DogHouse() {
    cout << "DogHouse destructor: "
         << *this << endl;
    p->detach();
  }
  void renameHouse(const string& newName) {
    houseName = newName;
  }
  void unalias() { p = p->unalias(); }
  // Copy-on-write. Anytime you modify the
  // contents of the pointer you must
  // first unalias it:
  void renameDog(const string& newName) {
    unalias();
    p->rename(newName);
  }
  // ... or when you allow someone else access:
  Dog* getDog() {
    unalias();
    return p;
  }
  friend ostream&
  operator<<(ostream& os, const DogHouse& dh) {
    return os << "[" << dh.houseName
      << "] contains " << *dh.p;
  }
```

528

```
};

int main() {
  DogHouse
529   fidos(Dog::make("Fido"), "FidoHouse"),
    spots(Dog::make("Spot"), "SpotHouse");
  cout << "Entering copy-construction" << endl;
  DogHouse bobs(fidos);
  cout << "After copy-constructing bobs" << endl;
  cout << "fidos:" << fidos << endl;
  cout << "spots:" << spots << endl;
  cout << "bobs:" << bobs << endl;
  cout << "Entering spots = fidos" << endl;
  spots = fidos;
  cout << "After spots = fidos" << endl;
  cout << "spots:" << spots << endl;
  cout << "Entering self-assignment" << endl;
  bobs = bobs;
  cout << "After self-assignment" << endl;
  cout << "bobs:" << bobs << endl;
  // Comment out the following lines:
  cout << "Entering rename(\"Bob\")" << endl;
  bobs.getDog()->rename("Bob");
  cout << "After rename(\"Bob\")" << endl;
} ///:~
```

类**Dog**是**DogHouse**指向的对象。包含了一个引用记数及控制和读引用记数的函数。同时这里存在一个拷贝构造函数，所以可以从现有的对象创建一个新的**Dog**。

函数**attach( )**增加一个**Dog**的引用记数用以指示有另一个对象使用它。函数**detach( )**减少引用记数。如果引用记数为0，则说明没有对象使用它，所以通过表达式**delete this**，成员函数销毁它自己的对象。

在进行任何修改（例如为一个**Dog**重命名）之前，必须保证所修改的**Dog**没有被别的对象正在使用。这可以通过调用**DogHouse::unalias( )**，它又进而调用**Dog::unalias( )**来做到这点。如果引用记数为1（意味着没有别的对象指向这块存储单元），后面这个函数将返回存在的**Dog**指针，但如果引用记数大于1（意味着不只一个对象指向这个**Dog**）就要复制这个**Dog**。

530 拷贝构造函数给源对象**Dog**赋值**Dog**，而不是创建它自己的存储单元。然后因为现在增加了使用这个存储单元的对象，所以通过调用**Dog::attach( )**增加引用记数。

operator=处理等号左侧已创建的对象，所以它首先必须通过为**Dog**调用**detach( )**来整理这个存储单元。如果没有其他对象使用它，这个老的**Dog**将被销毁。然后**operator=**重复拷贝构造函数的行为。注意它首先检查是否给它本身赋予相同的对象。

析构函数调用**detach( )**有条件地销毁**Dog**。

为了实现写拷贝，必须控制所有写存储单元的动作。例如成员函数**renameDog( )**允许对这个存储单元修改数值。但它首先必须使用**unalias( )**防止修改一个已别名化了的存储单元（超过一个对象使用的存储单元）。如果想从**DogHouse**中产生一个指向**Dog**的指针，首先要对指针调用**unalias( )**。

在**main( )**中测试了几个必须正确实现引用记数的函数：构造函数、拷贝构造函数、**operator=**和析构函数。在**main( )**中也通过C调用**renameDog( )**测试了写拷贝。

下面是（一部分重新格式化后的）输出结果:

```
Creating Dog: [Fido], rc = 1
Created DogHouse: [FidoHouse]
  contains [Fido], rc = 1
Creating Dog: [Spot], rc = 1
Created DogHouse: [SpotHouse]
  contains [Spot], rc = 1
Entering copy-construction
Attached Dog: [Fido], rc = 2
DogHouse copy-constructor:
  [copy-constructed FidoHouse]
    contains [Fido], rc = 2
After copy-constructing bobs
fidos:[FidoHouse] contains [Fido], rc = 2
spots:[SpotHouse] contains [Spot], rc = 1
bobs:[copy-constructed FidoHouse]
  contains [Fido], rc = 2
Entering spots = fidos
Detaching Dog: [Spot], rc = 1
Deleting Dog: [Spot], rc = 0
Attached Dog: [Fido], rc = 3
DogHouse operator= : [FidoHouse assigned]
  contains [Fido], rc = 3
After spots = fidos
spots:[FidoHouse assigned] contains [Fido],rc = 3
Entering self-assignment
DogHouse operator= : [copy-constructed FidoHouse]
  contains [Fido], rc = 3
After self-assignment
bobs:[copy-constructed FidoHouse]
  contains [Fido], rc = 3
Entering rename("Bob")
After rename("Bob")
DogHouse destructor: [copy-constructed FidoHouse]
  contains [Fido], rc = 3
Detaching Dog: [Fido], rc = 3
DogHouse destructor: [FidoHouse assigned]
  contains [Fido], rc = 2
Detaching Dog: [Fido], rc = 2
DogHouse destructor: [FidoHouse]
  contains [Fido], rc = 1
Detaching Dog: [Fido], rc = 1
Deleting Dog: [Fido], rc = 0
```

531

通过研究输出结果、跟踪源代码和程序的测试，将加深对这些技术的理解。

12.5.1.3 自动创建operator=

因为将一个对象赋给另一个相同类型的对象是大多数人可能做的事情，所以如果没有创建**type::operator=(type)**，编译器将自动创建一个。这个运算符行为模仿自动创建的拷贝构造函数的行为：如果类包含对象（或是从别的类继承的），对于这些对象，**operator=**被递归调用。这被称为成员赋值(*memberwise assignment*)。见如下例子:

```
//: C12:AutomaticOperatorEquals.cpp
#include <iostream>
using namespace std;
```

532

```
class Cargo {
public:
  Cargo& operator=(const Cargo&) {
    cout << "inside Cargo::operator=()" << endl;
    return *this;
  }
};

class Truck {
  Cargo b;
};

int main() {
  Truck a, b;
  a = b; // Prints: "inside Cargo::operator=()"
} ///:~
```

为**Truck**自动生成的**operator=**调用**Cargo::operator=**。

一般我们不会想让编译器做这些。对于复杂的类（尤其是它们包含指针时），应该显式地创建一个**operator=**。如果真的不想让人执行赋值运算，可以把**operator=**声明为**private**函数（除非在类内使用它，否则不必定义它）。

12.6 自动类型转换

在C和C++中，如果编译器看到一个表达式或函数调用使用了一个不合适的类型，它经常会执行一个自动类型转换，从现在的类型到所要求的类型。在C++中，可以通过定义自动类型转换函数来为用户定义类型达到相同效果。这些函数有两种类型：特殊类型的构造函数和
533 重载的运算符。

12.6.1 构造函数转换

如果定义一个构造函数，这个构造函数能把另一类型对象（或引用）作为它的单个参数，那么这个构造函数允许编译器执行自动类型转换。如下例：

```
//: C12:AutomaticTypeConversion.cpp
// Type conversion constructor
class One {
public:
  One() {}
};

class Two {
public:
  Two(const One&) {}
};

void f(Two) {}

int main() {
  One one;
  f(one); // Wants a Two, has a One
} ///:~
```

当编译器看到f( )以类One的对象为参数调用时，编译器检查f( )的声明并注意到它需要一个类Two的对象作为参数。然后，编译器检查是否有从对象One到Two的方法。它发现了构造函数Two::Two(One)，Two::Two(One)被悄悄地调用，结果对象Two被传递给f( )。

在这种情况下，自动类型转换避免了定义两个f( )重载版本的麻烦。然而，代价是调用Two的隐藏构造函数，如果关心f( )的调用效率的话，那就不要使用这种方法。

12.6.1.1 阻止构造函数转换

有时通过构造函数自动转换类型可能出现问题。为了避开这个麻烦，可以通过在前面加关键字explicit（只能用于构造函数）来对上例类Two的构造函数进行修改: 534

```
//: C12:ExplicitKeyword.cpp
// Using the "explicit" keyword
class One {
public:
  One() {}
};

class Two {
public:
  explicit Two(const One&) {}
};

void f(Two) {}

int main() {
  One one;
//!  f(one); // No auto conversion allowed
  f(Two(one)); // OK -- user performs conversion
} ///:~
```

通过使类Two的构造函数显式化，编译器被告知不能使用那个构造函数执行任何自动转换（那个类中其他非显式化的构造函数仍可以执行自动类型转换）。如果用户想进行转换必须写出代码。上面代码f(Two(One))创建一个从类型One到Two的临时对象，就像编译器在前面版本中所做的那样。

12.6.2 运算符转换

第二种自动类型转换的方法是通过运算符重载。可以创建一个成员函数，这个函数通过在关键字operator后跟随想要转换到的类型的方法，将当前类型转换为希望的类型。这种形式的运算符重载是独特的，因为没有指定一个返回类型——返回类型就是正在重载的运算符的名字。下面是一个例子:

```
//: C12:OperatorOverloadingConversion.cpp
class Three {
  int i;
public:
  Three(int ii = 0, int = 0) : i(ii) {}
};

class Four {
  int x;
public:
```

535

```
  Four(int xx) : x(xx) {}
  operator Three() const { return Three(x); }
};

void g(Three) {}

int main() {
  Four four(1);
  g(four);
  g(1);  // Calls Three(1,0)
} ///:~
```

用构造函数技术，目的类执行转换。然而使用运算符技术，是源类执行转换。构造函数技术的价值是在创建一个新类时为现有系统增加了新的转换途径。然而，创建一个单一参数的构造函数总是定义一个自动类型转换（即使它有不止一个参数也是一样，因为其余的参数将被默认处理），这可能并不是我们所想要的（那种情况下可使用**explicit**来避免）。另外，使用构造函数技术没有办法实现从用户定义类型向内置类型转换，这只有运算符重载可能做到。

12.6.2.1 反身性

使用全局重载运算符而不用成员运算符的最便利的原因之一是在全局版本中的自动类型转换可以针对左右任一操作数，而成员版本必须保证左侧操作数已处于正确的形式。如果想两个操作数都被转换，全局版本可以节省很多代码。下面有一个小例子：

```
//: C12:ReflexivityInOverloading.cpp
class Number {
  int i;
536 public:
  Number(int ii = 0) : i(ii) {}
  const Number
  operator+(const Number& n) const {
    return Number(i + n.i);
  }
  friend const Number
    operator-(const Number&, const Number&);
};

const Number
  operator-(const Number& n1,
            const Number& n2) {
    return Number(n1.i - n2.i);
}

int main() {
  Number a(47), b(11);
  a + b; // OK
  a + 1; // 2nd arg converted to Number
//! 1 + a; // Wrong! 1st arg not of type Number
  a - b; // OK
  a - 1; // 2nd arg converted to Number
  1 - a; // 1st arg converted to Number
} ///:~
```

类**Number**有一个成员**operator+**和一个**friend operator-**。因为有一个使用单一**int**参数的

构造函数，在正确的条件下，**int**可以自动转换为**Number**。在**main( )**里，可以看到增加一个**Number**到另一个**Number**进行得很好，这是因为它重载的运算符非常相匹配。当编译器看到一个**Number**后跟一个**+**号和一个**int**时，它也能和成员函数**Number::operator+**相匹配并且构造函数把**int**参数转换为**Number**。但当编译器看到一个**int**、一个**+**号和一个**Number**时，它就不知道如何去做，因为它所拥有的是**Number::operator+**，需要左侧的操作数是**Number**对象。因此，编译器发出一个出错信息。

对于**friend operator-**，情况就不同了。编译器需要填满两个参数，它不是限定**Number**作为左侧参数。因此，如果看到表达式

```
1 - a
```

537

编译器就使用构造函数把第一个参数转换为**Number**。

有时也许想通过把它们设成成员函数来限定运算符的使用。例如当用一个矢量与矩阵相乘，矢量必须在右侧。但如果想让运算符转换任一个参数，就要使运算符为友元函数。

幸运的是，编译器不会把表达式**1-1**的两个参数转换为**Number**对象，然后调用**operator-**。那将意味着现有的C代码可能突然执行不同的工作了。编译器首先匹配“最简单的”可能性，对于表达式**1-1**将优先使用内置运算符。

12.6.3 类型转换例子

本例中的自动类型转换对于任一含有字符串的类（本例中，因为是简单的，所以使用的是标准**C++ string**类）是非常有帮助的。如果不用自动类型转换就想从标准的C库函数中使用所有的字符串函数，那么就得为每一个函数写一个相应的成员函数，就像下面的例子：

```
//: C12:Strings1.cpp
// No auto type conversion
#include "../require.h"
#include <cstring>
#include <cstdlib>
#include <string>
using namespace std;

class Stringc {
  string s;
public:
  Stringc(const string& str = "") : s(str) {}
  int strcmp(const Stringc& S) const {
    return ::strcmp(s.c_str(), S.s.c_str());
  }
  // ... etc., for every function in string.h
};

int main() {
  Stringc s1("hello"), s2("there");
  s1.strcmp(s2);
} ///:~
```

538

这里只写了一个**strcmp( )**函数，但必须为可能需要的**<cstring>**中的每一个写一个相应的函数。幸运的是，可以提供一个允许访问**<cstring>**中所有函数的自动类型转换：

```
//: C12:Strings2.cpp
```

```
// With auto type conversion
#include "../require.h"
#include <cstring>
#include <cstdlib>
#include <string>
using namespace std;

class Stringc {
  string s;
public:
  Stringc(const string& str = "") : s(str) {}
  operator const char*() const {
    return s.c_str();
  }
};

int main() {
  Stringc s1("hello"), s2("there");
  strcmp(s1, s2); // Standard C function
  strspn(s1, s2); // Any string function!
} ///:~
```

因为编译器知道如何从**String c**转换到**char***，所以现在任何一个接受**char***参数的函数也可以接受**Stringc**参数。

12.6.4 自动类型转换的缺陷

因为编译器必须选择如何执行类型转换，所以如果没有正确地设计出转换，编译器会产
539 生麻烦。类**X**可以用**operator Y( )**将它本身转换到类**Y**，这是一个简单且明显的情况。如果类**Y**有一个单个参数为**X**的构造函数，这也表示同样的类型转换。现在编译器有两个从**X**到**Y**的转换方法，所以当发生转换时，编译器会产生一个不明确指示的出错信息：

```
//: C12:TypeConversionAmbiguity.cpp
class Orange; // Class declaration

class Apple {
public:
  operator Orange() const; // Convert Apple to Orange
};

class Orange {
public:
  Orange(Apple); // Convert Apple to Orange
};

void f(Orange) {}

int main() {
  Apple a;
//! f(a); // Error: ambiguous conversion
} ///:~
```

这个问题的解决方法是不要那样做，而是仅提供单一的从一个类型到另一个类型的自动转换方法。

当提供了转换到不止一种类型的自动转换时，会发生一个引起出错的更困难的问题。有时，这个问题被称为扇出(*fan-out*):

```
//: C12:TypeConversionFanout.cpp
class Orange {};
class Pear {};

class Apple {
public:
  operator Orange() const;
  operator Pear() const;
};
// Overloaded eat():
void eat(Orange);
void eat(Pear);

int main() {
  Apple c;
//! eat(c);
  // Error: Apple -> Orange or Apple -> Pear ???
} ///:~
```

540

类**Apple**有向**Orange**和**Pear**的自动转换。这样存在一个隐藏的缺陷：使用了创建的两种版本的重载运算符**eat( )**时就出现问题了（只有一个版本时，**main( )**里的代码会正常运行）。

通常，对于自动类型的解决方法是只提供一个从某类型向另一个类型转换的自动转换版本。当然也可以有多个向其他类型的转换，但它们不应该是自动转换，而应该用如**makeA( )**和**makeB( )**这样的名字来创建显式的函数调用。

12.6.4.1 隐藏的行为

自动类型转换会引入比所希望的更多的潜在行为。下面要费点力去理解了，看看**CopyingVsInitialization.cpp**修改后的例子:

```
//: C12:CopyingVsInitialization2.cpp
class Fi {};

class Fee {
public:
  Fee(int) {}
  Fee(const Fi&) {}
};

class Fo {
  int i;
public:
  Fo(int x = 0) : i(x) {}
  operator Fee() const { return Fee(i); }
};
int main() {
  Fo fo;
  Fee fee = fo;
} ///:~
```

541

这里没有从**Fo**对象创建**Fee fee**的构造函数。然而，**Fo**有一个到**Fee**的自动类型转换。这

里也没有从Fee对象创建Fee的拷贝构造函数，但这是一种能由编译器帮助我们创建的特殊函数之一（默认的构造函数、拷贝构造函数、operator=和析构函数可自动创建）。对于下面正确的声明：

```
Fee fee = fo;
```

自动类型转换运算符被调用并创建一个拷贝函数：

自动类型转换应小心使用。同所有重载的运算符相比，它在减少代码方面是非常出色的，但不值得无缘无故地使用。

12.7 小结

运算符重载存在的原因是为了使编程容易。运算符重载没有什么神秘的，它只不过是拥有有趣名字的函数。当它以正确的形式出现时，编译器调用这个函数。但如果运算符重载对于类的设计者或类的使用者不能提供特别显著的益处，则最好不要使用，因为增加运算符重载会使问题混淆。

12.8 练习

部分练习题的答案可以在本书的电子文档 *"Annotated Solution Guide for Thinking in C++"* 中找到，只需支付很少的费用就可以从http://www.BruceEckel.com得到这个电子文档。

542

12-1 写一个有重载**operator++**的类。试着用前缀和后缀两种形式调用此运算符，看看编译器会给出什么警告。

12-2 创建含有一个**int**成员的简单的类，以成员函数的形式重载**operator+**。同时提供一个**print( )**成员函数，以**ostream&**作为参数并打印出该**ostream&**。测试该类表明它可以正确运行。

12-3 用成员函数形式，在练习2的类中增加一个二元**operator-**。要求可以在**a+b-c**这样复杂的表达式中使用该类的对象。

12-4 在练习2的例子中增加**operator++**和**operator--**，要包括前缀和后缀版本，使得它们返回自增和自减对象。确保后缀版本返回正确的值。

12-5 修改练习4的自增和自减运算符，以使得前缀版本使用非常量而后缀版本使用常量。显示它们运行正确并解释为什么实际中要这么做。

12-6 改变练习2中的**print( )**函数，使得它是重载**operator<<**，就像在**IostreamOperatorOverloading.cpp**中一样。

12-7 修改练习3，使得**operator+**和**operator-**是非成员函数。表明它们仍能正确运行。

12-8 对练习2的类中增加一元**operator-**，并表明它可以正确运行。

12-9 写一个只含有单个**private char**成员的类。重载iostream **operator<<**和**>>**（像在**IostreamOperatorOverloading.cpp**中的一样）并测试它们，可以用**fstreams**、**stringstreams**和 **cin** 与**cout**测试它们。

12-10 测定为了前缀**operator++**和**operator--**，编译器传递的哑常量值。

543

12-11 写一个包含一个**double**成员的**Number**类，并增添重载的**operator+**、**-**、*、/和赋值符。为这些函数合理地选择返回值以便可以链式写表达式，以提高效率。写一个自动类型转换**operator double( )**。

12-12 如果返回值优化还没有被使用，修改练习11，使用返回值优化。

12-13 创建一个包含指针的类，表明如果允许编译器生成**operator=**，结果将得到具有不同别名、指向同一存储区域的指针。现在通过定义自己的**operator=**解决这个问题，并表明它纠正了别名。确保检查自赋值并完全处理了此问题。

12-14 写一个包含**string**和**static int**成员的类**Bird**。在预设的构造函数中，根据类的名字(**Bird #1**，**Bird #2**，等等)，使用**int**成员自动地生成一个置于**string**中的标识符。为**ostream**增加**operator<<**以打印**Bird**对象。写一个赋值**operator=**和一个拷贝构造函数。在**main( )**中，验证它们都可正确运行。

12-15 写一个类**BirdHouse**，包含一个对象、一个指针和一个练习14中类**Bird**的引用。构造函数的参数是三个**Bird**类对象。为**BirdHouse**增加**ostream operator<<**。写一个赋值**operator=**和一个拷贝构造函数。在**main( )**中，验证它们都正确地运行。确保能对**BirdHouse**对象链接赋值运算，并生成包括多种操作的表达式。

12-16 对练习15中的类**Bird**和**BirdHouse**增加一个**int**数据成员。增加成员运算符**+**、**-**、*和 /，它们使用**int**成员在各自的成员上进行操作。验证这些工作。

12-17 用非成员运算符进行练习16的操作。

12-18 增加**operator--**到**SmartPointer.cpp**和**NestedSmartPointer.cpp**中。

12-19 修改**CopyingVsInitialization.cpp**，使得所有的构造函数打印出正在进行操作的信息。现在验证拷贝构造函数的两种调用形式是相等的。 544

12-20 试着为一个类创建一个非成员**operator=**，并看看能得到编译器的何种信息。

12-21 用拷贝构造函数创建一个类，该拷贝构造函数的第二个参数是一个预设值为“op=call.”的**string**成员。创建一个函数，它通过传值方式接受此类的对象，并表明拷贝构造函数被正确地调用了。

12-22 在**CopyingWithPointers.cpp**中，删除**DogHouse**中的**operator=**，表明编译器生成的**operator=**正确地拷贝了**string**成员，但简单地为**Dog**指针起了别名。

12-23 在**ReferenceCounting.cpp**中，增加一个**static int**成员和一个基本**int**成员作为**Dog**和**DogHouse**的数据成员。在两个类的所有构造函数中，把**static int**成员加1并把结果赋于基本**int**成员以记录所创建的对象的数目。进行必要的修改，打印出涉及到的对象的**int**标识符。

12-24 创建包含一个**string**数据成员的类。在构造函数中初始化**string**成员，但不用创建拷贝构造函数或**operator=**。创建第二个类，其成员对象是第一个类的对象；同样不要为这个类创建拷贝构造函数或**operator=**。表明拷贝构造函数和**operator=**可被编译器正确地生成。

12-25 合并在**OverloadingUnaryOperators.cpp**和**Integer.cpp** 中的类。

12-26 通过新增加两个**Dog**中的成员函数，它们无参数并返回为**void**值，来修改**PointerToMemberOperator.cpp**。创建并测试作用于这两个新函数上的重载**operator->***。

12-27 在**NestedSmartPointer.cpp**中增加**operator->***。

12-28 创建两个类**Apple**和**Orange**。在类**Apple**中，创建一个构造函数，其参数是**Orange**类的对象。然后创建一个函数，它的参数是**Apple**类对象，并用**Orange**类 545

对象调用该函数。现在显式应用**Apple**类的构造函数以表明自动类型转换被禁止。修改对函数的调用，使得能成功地做显式转换。

12-29 在**ReflexivityInOverloading.cpp**中增加一个全局**operator***并表明它具有反身性。

12-30 创建两个类并创建**operator+**和转换函数，使得对于这两个类，加法具有反身性。

12-31 不用自动转换运算符，而通过创建一个显式的执行类型转换的函数修改**TypeConversionFanout.cpp**。

12-32 写一段简单的代码，在其中对**double**类型使用**operator+**、**-**、***和 **/**。看看编译器如何生成汇编代码并根据生成的汇编语言找出并解释发生了什么。

546

第13章 动态对象创建

有时我们能知道程序中对象的确切数量、类型和生命期。但情况并不总是这样。 547

空中交通指挥系统将会需要处理多少架飞机？一个CAD系统将会需要多少个形体？在一个网络中将会有多少个节点？

为了解决这个普通的编程问题，在运行时可以创建和销毁对象是最基本的要求。当然，C早就提供了*动态内存分配*（*dynamic memory allocation*）函数**malloc( )**和**free( )**（以及**malloc( )**的变体），这些函数在运行时从*堆*（也称*自由内存*）中分配存储单元。

然而，在C++中这些函数将不能很好地运行。因为构造函数不允许我们向它传递内存地址来进行初始化。如果那么做了，我们可能：

1) 忘记了。则在C++中有保证的对象初始化将会难以保证。

2) 期望发生正确的事，但在对对象进行初始化之前意外地对对象进行了某种操作。

3) 把错误规模的对象传递给它。

当然，即使我们正确地完成了每件事，修改我们程序的人也容易犯同样的错误。不正确的初始化要对大部分编程问题承担责任，所以在堆上创建对象时确保构造函数调用是特别重要的。

C++是如何保证正确的初始化和清理，又允许我们在堆上动态创建对象呢？

答案是，使动态对象创建成为语言的核心。**malloc( )**和**free( )**是库函数，因此不在编译器控制范围之内。然而，如果我们有一个完成动态内存分配及初始化组合动作的运算符和另一个完成清理及释放内存组合动作的运算符，编译器仍可以保证所有对象的构造函数和析构函数都会被调用。 548

在本章中，我们将明白C++的**new**和**delete**是如何通过在堆上安全地创建对象来出色地解决这个问题。

13.1 对象创建

当创建一个C++对象时，会发生两件事：

1) 为对象分配内存。

2) 调用构造函数来初始化那个内存。

到目前为止，应该确保步骤2一定发生。C++强迫这样做是因为未初始化的对象是程序出错的主要原因。对象在哪里和如何被创建无关紧要——构造函数总是需要被调用。

然而，步骤1可以用几种方式或在可选择的时间发生：

1) 在静态存储区域，存储空间在程序开始之前就可以分配。这个存储空间在程序的整个运行期间都存在。

2) 无论何时到达一个特殊的执行点（左大括号）时，存储单元都可以在栈上被创建。出了执行点（右大括号），这个存储单元自动被释放。这些栈分配运算内置于处理器的指令集中，

非常有效。然而，在写程序的时候，必须知道需要多少个存储单元，以便编译器生成正确的指令。

3) 存储单元也可以从一块称为堆（也被称为自由存储单元）的地方分配。这被称为动态内存分配。在运行时调用程序分配这些内存。这意味着可以在任何时候决定分配内存及分配多少内存。当然也需负责决定何时释放内存。这块内存的生存期由我们选择决定——而不受
549 范围决定。

这三个区域经常被放在一块连续的物理存储单元里：静态内存、栈和堆（由编译器的开发者决定它们的顺序），但没有一定的规则。堆栈可以在特定的地方，堆的实现可以通过调用由运算系统分配的一块存储单元。这三件事无需程序设计者来完成。当申请内存的时候，只要知道它们在哪里就行了。

13.1.1 C从堆中获取存储单元的方法

为了在运行时动态分配内存，C在它的标准库函数中提供了一些函数：从堆中申请内存的函数**malloc( )**以及它的变种**calloc( )**和**realloc( )**、释放内存返回给堆的函数**free( )**。这些函数是有效的但较原始的，需要编程人员理解和小心使用。为了使用C的动态内存分配函数在堆上创建一个类的实例，我们必须这样做：

```
//: C13:MallocClass.cpp
// Malloc with class objects
// What you'd have to do if not for "new"
#include "../require.h"
#include <cstdlib> // malloc() & free()
#include <cstring> // memset()
#include <iostream>
using namespace std;

class Obj {
  int i, j, k;
  enum { sz = 100 };
  char buf[sz];
public:
  void initialize() { // Can't use constructor
    cout << "initializing Obj" << endl;
    i = j = k = 0;
    memset(buf, 0, sz);
  }
  void destroy() const { // Can't use destructor
    cout << "destroying Obj" << endl;
  }
550 };
int main() {
  Obj* obj = (Obj*)malloc(sizeof(Obj));
  require(obj != 0);
  obj->initialize();
  // ... sometime later:
  obj->destroy();
  free(obj);
} ///:~
```

在下面这行代码中，使用了**malloc( )**为对象分配内存：

```
Obj* obj = (Obj*)malloc(sizeof(Obj));
```

这里用户必须决定对象的长度（这也是程序出错原因之一）。由于**malloc( )**只是分配了一块内存而不是生成一个对象，所以它返回了一个**void***类型指针。而C++不允许将一个**void***类型指针赋予任何其他指针，所以必须做类型转换。

因为**malloc( )**可能找不到可分配的内存（在这种情况下它返回0），所以必须检查返回的指针以确保内存分配成功。

但这一行最易出现问题：

```
Obj->initialize();
```

用户在使用对象之前必须记住对它初始化。注意构造函数没有被使用，这是因为构造函数不能被显式地调用[⊖]——它是在对象创建时由编译器调用。问题是现在用户可能在使用对象时忘记执行初始化，因此这又是一个程序出错的主要来源。

许多程序设计者发现C的动态内存分配函数太复杂，容易令人混淆。所以，C程序设计者常常在静态内存区域使用虚拟内存机制分配很大的变量数组以避免使用动态内存分配。为了在C++中使得一般的程序员可以安全使用库函数而不费力，所以C的动态内存方法是不可接受的。 551

13.1.2 operator new

C++中的解决方案是把创建一个对象所需的所有动作都结合在一个称为**new**的运算符里。当用**new**（**new**的表达式）创建一个对象时，它就在堆里为对象分配内存并为这块内存调用构造函数。因此，如果写出下面的表达式

```
MyType *fp = new MyType(1,2);
```

在运行时等价于调用**malloc(sizeof(MyType))**(常常,就是精确地调用**malloc( )**)，并使用（**1, 2**）作为参数表来为**MyType**调用构造函数，**this**指针指向返回值的地址。在指针被赋给**fp**之前，它是不定的、初始化的对象——在这之前我们甚至不能触及它。它自动地被赋予正确的**MyType**类型，所以不必进行映射。

默认的**new**还进行检查以确信在传递地址给构造函数之前内存分配是成功的，所以不必显式地确定调用是否成功。在本章后面，我们将会发现，如果没有可供分配的内存会发生什么事情。

我们可以为类使用任何可用的构造函数而写一个**new**表达式。如果构造函数没有参数，可以写没有构造函数参数表的**new**表达式：

```
MyType *fp = new MyType;
```

我们已经注意到了，在堆里创建对象的过程变得简单了——只是一个简单的表达式，它带有内置的长度计算、类型转换和安全检查。这样在堆里创建一个对象和在栈里创建一个对象一样容易。 552

13.1.3 operator delete

new表达式的反面是**delete**表达式。**delete**表达式首先调用析构函数，然后释放内存（经常是调用**free( )**）。正如**new**表达式返回一个指向对象的指针一样，**delete**表达式需要一个对象的地址。

⊖ 这里，称为*定位new*（*placement new*）的特殊语法可用来在一块预先分配好的内存上调用构造函数。这将在后面的章节中加以介绍。

```
delete fp;
```

上面的表达式清除了早先创建的动态分配的**MyType**类型对象。

delete只用于删除由**new**创建的对象。如果用**malloc( )**（或**calloc( )**或**realloc( )**）创建一个对象，然后用**delete**删除它，这个动作行为是未定义的。因为大多数默认的**new**和**delete**实现机制都使用了**malloc( )**和**free( )**，所以很可能会没有调用析构函数就释放了内存。

如果正在删除的对象的指针是0，将不发生任何事情。为此，人们经常建议在删除指针后立即把指针赋值为0以免对它删除两次。对一个对象删除两次可能会产生某些问题。

13.1.4 一个简单的例子

这个例子显示了初始化发生的情况：

```
//: C13:Tree.h
#ifndef TREE_H
#define TREE_H
#include <iostream>

class Tree {
  int height;
public:
  Tree(int treeHeight) : height(treeHeight) {}
  ~Tree() { std::cout << "*"; }
  friend std::ostream&
  operator<<(std::ostream& os, const Tree* t) {
    return os << "Tree height is: "
              << t->height << std::endl;
  }
};
#endif // TREE_H ///:~
```

553

```
//: C13:NewAndDelete.cpp
// Simple demo of new & delete
#include "Tree.h"
using namespace std;

int main() {
  Tree* t = new Tree(40);
  cout << t;
  delete t;
} ///:~
```

我们通过打印**Tree**的值得知构造函数被调用了。这里是通过调用参数为**ostream**和**Tree***类型的重载**operator<<**来实现这个运算的。注意，虽然这个函数被声明为一个友元（**friend**），但它还是被定义为一个内联函数。这仅仅是出于方便考虑——定义一个友元函数为内联函数不会改变友元状态，而且它仍是全局函数而不是一个类的成员函数。也要注意返回值是整个输出表达式的结果，它本身是一个**ostream&**（为了满足函数返回值类型，它必须是**ostream&**）。

13.1.5 内存管理的开销

当在堆栈里自动创建对象时，对象的大小和它们的生存期被准确地内置在生成的代码里，

这是因为编译器知道确切的类型、数量和范围。而在堆里创建对象还包括另外的时间和空间的开销。以下是一个典型的情况。(可以用**calloc( )**或**realloc( )**代替**malloc( )**)

调用**malloc( )**，这个函数从堆里申请一块内存（这实际上是用了**malloc( )**代码的一部分）。

从堆里搜索一块足够大的内存来满足请求。这可以通过检查按某种方式排列的映射或目录来实现，这样的映射或目录用以显示内存的使用情况。这个过程很快但可能要试探几次， 554
所以它可能是不确定的——即每次运行**malloc( )**并不是花费了完全相同的时间。

在指向这块内存的指针返回之前，这块内存的大小和地址必须记录下来，这样以后调用**malloc( )**就不会使用它，而且当调用**free( )**时，系统就会知道释放多大的内存。

实现这些运算的方法可能变化很大。例如，不能阻止处理器中的内存分配原语的执行。如果好奇的话，可以写一个测试程序来估计**malloc( )**实现的方法。如果有的话，当然也可以读库函数的源代码。(GNU C 的源代码总是有的。)

13.2 重新设计前面的例子

使用**new**和**delete**，对于本书前面介绍的**Stash**例子，可以使用到目前为止讨论的所有的技术来重写。检查这个新代码将有助于对这些主题的复习。

在本书的此处，类**Stash**和**Stack**自己都将不“拥有”它们指向的对象。即当**Stash**或**Stack**对象出了范围，它也不会为它指向的对象调用**delete**。试图使它们成为普通的类是不可能的，原因是它们是**void**指针。如果**delete**一个**void**指针，惟一发生的事就是释放了内存，这是因为既没有类型信息也没有办法使得编译器知道要调用哪个析构函数。

13.2.1 使用delete void*可能会出错

如果想对一个**void***类型指针进行**delete**操作，要注意这将可能成为一个程序错误，除非指针所指的内容是非常简单的，因为，它将不执行析构函数。下面的例子将显示发生的情况:

```
//: C13:BadVoidPointerDeletion.cpp
// Deleting void pointers can cause memory leaks                    555
#include <iostream>
using namespace std;

class Object {
  void* data; // Some storage
  const int size;
  const char id;
public:
  Object(int sz, char c) : size(sz), id(c) {
    data = new char[size];
    cout << "Constructing object " << id
         << ", size = " << size << endl;
  }
  ~Object() {
    cout << "Destructing object " << id << endl;
    delete []data; // OK, just releases storage,
    // no destructor calls are necessary
  }
};
```

```
int main() {
  Object* a = new Object(40, 'a');
  delete a;
  void* b = new Object(40, 'b');
  delete b;
} ///:~
```

类**Object**包含了一个**void***指针，它被初始化指向“元”数据(它没有指向含有析构函数的对象)。在**Object**的析构函数中，对这个**void***指针调用**delete**并不会发生什么错误，因为所需要的仅是释放这块内存。

但在**main( )**中，我们看到使**delete**知道它所操作的对象的类型是十分有必要的。输出如下:

```
Constructing object a, size = 40
Destructing object a
Constructing object b, size = 40
```

因为**delete a**知道**a**指向一个**Object**对象，所以析构函数将会被调用，从而释放了分配给
556 **data**的内存。但是，正如在进行**delete b** 的操作中，如果通过**void***类型的指针对一个对象进行操作，则只会释放**Object**对象的内存，而不会调用析构函数，也就不会释放**data**所指向的内存。编译这个程序时，编译器会认为我们知道所做的一切。于是我们不会看到任何警告信息。但因此我们会丢失大量的可用内存。

如果在程序中发现内存丢失的情况，那么就搜索所有的**delete**语句并检查被删除指针的类型。如果是**void***类型，则可能发现了引起内存丢失的某个因素（因为C++还有很多其他的引起内存丢失的因素)。

13.2.2 对指针的清除责任

为了使**Stash**和**Stack**容器具有灵活性（可以包含任意类型的对象)，要使用**void**指针。这意味着当一个指针从**Stash**或**Stack**对象返回时，必须在使用之前把它转换为适当的类型。如上所示，在删除它之前也必须把它转换为适当的类型，否则将会丢失内存。

解决内存泄漏的另一个工作在于确保对容器中的每一个对象调用**delete**。容器含有**void***类型指针，因此不能正确地执行清除，所以容器自己不能“管理“指针。于是用户必须负责清除这些对象。如果把指向在栈上创建的对象的指针和指向在堆上创建的对象的指针都存放在同一个容器中，将会发生严重的问题。(当从容器中取回一个指针时，我们如何才能知道它所指向的对象是被分配在哪块内存上的呢？）因此不管是通过精心的设计或是通过只作用在堆上的类创建，我们都必须保证存储在如下版本的**Stash**和**Stack**上的对象仅是在堆上创建的。

557 保证由客户程序员负责清除容器中的所有指针同样是很重要的。在前面的例子中，已经看到**Stack**类是如何在它的析构函数中检查所有的**Link**对象已经出栈了的。但对于**Stash**，需要使用另一种方法。

13.2.3 指针的Stash

Stash的新版本称为**PStash**，它含有在堆中本来就存在的对象的指针。而前面章节中旧的**Stash**则是通过传值方式拷贝对象到**Stash**的容器。使用**new**和**delete**，控制指向在堆中创建的

对象的指针就变得安全、容易了。

下面提供了“pointer **Stash**”的头文件:

```
//: C13:PStash.h
// Holds pointers instead of objects
#ifndef PSTASH_H
#define PSTASH_H

class PStash {
  int quantity; // Number of storage spaces
  int next; // Next empty space
   // Pointer storage:
  void** storage;
  void inflate(int increase);
public:
  PStash() : quantity(0), storage(0), next(0) {}
  ~PStash();
  int add(void* element);
  void* operator[](int index) const; // Fetch
  // Remove the reference from this PStash:
  void* remove(int index);
  // Number of elements in Stash:
  int count() const { return next; }
};
#endif // PSTASH_H ///:~
```

基本的数据成分是非常相似的，但现在**storage**是一个**void**指针数组，并且用**new**代替**malloc( )**为这个数组分配内存。在下面这个表达式中，

```
void** st = new void*[quantity + increase];
```

558

对象的类型是**void***，所以这个表达式表示分配了一个**void**指针的数组。

析构函数删除**void**指针本身，而不是试图删除它们所指向的内容（正如前面所指出的，释放它们的内存但不调用析构函数，这是因为一个**void**指针没有类型信息）。

其他方面的变化是用**operator[ ]**代替了函数**fetch( )**，这在语句构成上显得更有意义。因为返回一个**void***指针，所以用户必须记住在容器内存储的是什么类型，在取回它们时要对这些指针进行类型转换（这是在以后章节中将要修改的问题）。

下面是成员函数的定义:

```
//: C13:PStash.cpp {O}
// Pointer Stash definitions
#include "PStash.h"
#include "../require.h"
#include <iostream>
#include <cstring> // 'mem' functions
using namespace std;

int PStash::add(void* element) {
  const int inflateSize = 10;
  if(next >= quantity)
    inflate(inflateSize);
  storage[next++] = element;
  return(next - 1); // Index number
}
```

```
// No ownership:
PStash::~PStash() {
  for(int i = 0; i < next; i++)
    require(storage[i] == 0,
      "PStash not cleaned up");
  delete []storage;
}

// Operator overloading replacement for fetch
559 void* PStash::operator[](int index) const {
  require(index >= 0,
    "PStash::operator[] index negative");
  if(index >= next)
    return 0; // To indicate the end
  // Produce pointer to desired element:
  return storage[index];
}

void* PStash::remove(int index) {
  void* v = operator[](index);
  // "Remove" the pointer:
  if(v != 0) storage[index] = 0;
  return v;
}

void PStash::inflate(int increase) {
  const int psz = sizeof(void*);
  void** st = new void*[quantity + increase];
  memset(st, 0, (quantity + increase) * psz);
  memcpy(st, storage, quantity * psz);
  quantity += increase;
  delete []storage; // Old storage
  storage = st; // Point to new memory
} ///:~
```

除了用储存指针代替整个对象的拷贝外，函数**add( )**的效果和以前是一样的。

inflate( )的代码被修改为能处理**void***指针数组的存储，而不是先前的设计，只处理元比特。这里没有优先使用数组索引的拷贝方法，而是使用标准C库函数中的**memset( )**来使所有新的内存置0(并不是一定要如此,因为**PStash**有可能正确地管理所有的内存，但小心点是没有害处的)，然后用**memcpy( )**把存在的数据从原来的地方移到一个新的地方。通常类似于**memset( )** 和**memcpy( )** 的函数随着时间会逐渐优化，所以它们会比前面所示的循环更快。但由于类似**inflate( )**的函数可能没有被使用，所以一般看不出性能上的差异。然而这种比循环
560 更简炼的函数调用有助于防止编码错误。

为了由客户程序完全负责对象的清除，有两种方法可以获得**PStash**中的指针：其一是使用**operator[ ]**，它简单地返回作为一个容器成员的指针。第二种方法是使用成员函数**remove( )**，它返回指针，并且通过置**0**的方法从容器中删除该指针。当**PStash**的析构函数被调用时，它进行检查以确信所有的对象指针已被删除。如果注意到指针还没有被删除，则可以通过删除它来防止内存丢失。(后面的章节中有更加智能的方法)。

13.2.3.1 一个测试程序

为了测试**PStash**，我们重写了**Stash**的测试程序:

```
//: C13:PStashTest.cpp
//{L} PStash
// Test of pointer Stash
#include "PStash.h"
#include "../require.h"
#include <iostream>
#include <fstream>
#include <string>
using namespace std;

int main() {
  PStash intStash;
  // 'new' works with built-in types, too. Note
  // the "pseudo-constructor" syntax:
  for(int i = 0; i < 25; i++)
    intStash.add(new int(i));
  for(int j = 0; j < intStash.count(); j++)
    cout << "intStash[" << j << "] = "
         << *(int*)intStash[j] << endl;
  // Clean up:
  for(int k = 0; k < intStash.count(); k++)
    delete intStash.remove(k);
  ifstream in ("PStashTest.cpp");
  assure(in, "PStashTest.cpp");
  PStash stringStash;
  string line;
  while(getline(in, line))
    stringStash.add(new string(line));
  // Print out the strings:
  for(int u = 0; stringStash[u]; u++)
    cout << "stringStash[" << u << "] = "
         << *(string*)stringStash[u] << endl;
  // Clean up:
  for(int v = 0; v < stringStash.count(); v++)
    delete (string*)stringStash.remove(v);
} ///:~
```

561

与前面一样，我们创建了**Stash**对象，并且为它们加入了内容。不同的是这次的内容是由**new**表达式产生的指针。首先请注意这一行：

```
intStash.add(new int(i));
```

这个表达式**new int(i)**使用了伪构造函数形式，因此将在堆上创建了一块区域用来存储这个新的**int**对象，同时这个**int**对象被初始化为**i**。

打印时，由**PStash::operator[ ]**返回的值必须被转换为正确的类型，对于这个程序其余的**PStash**对象，也将重复这个动作。这是使用**void**指针的缺点，将在后面的章节中解决。

测试的第2步是打开源程序文件，并逐行把它读到每一个**PStash**里。首先用**getline( )**把每一行读入一个**String**对象，然后对**line**进行**new string**操作，将这一行的内容拷贝下来。如果每次只是传送**line**的地址，将会得到指向**line**的一些指针，而此时**line**仅包含了所读文件的最后一行的内容。

当取回指针时，我们可以看到表达式：

```
*(string*)stringStash[v]
```

为了使**operator[ ]**返回的指针具有正确的类型，它们必须被转换为**String***。然后，**String***被间接引用，所以此表达式的计算结果相当于一个对象，这时编译器也认为一个**string**对象被发送给了**cout**。

562 在堆上创建的对象必须通过**remove( )**语句进行注销，否则将会实时地得到一个信息，告诉我们并没有完全清除那些在**PStash**中的对象。注意，对于**int**指针，类型转换不是必需的，因为**int**类的对象没有析构函数，我们所需要的仅是释放内存。

```
delete intStash.remove(k);
```

但是，对于**string**指针，如果忘记了类型转换，则会出现内存泄漏的情况。所以说进行类型转换是十分重要的。

```
delete (string*)stringStash.remove(k);
```

这些问题的一部分（但不是全部）可以使用模板进行解决。（我们将在第16章中学习模板）。

13.3 用于数组的new和delete

在栈或堆上创建一个对象数组是同样容易的。当然，应当为数组里的每一个对象调用构造函数。但这里有一个限制条件：由于不带参数的构造函数必须被每一个对象调用，所以除了在栈上整体初始化（见第6章）外还必须有一个默认的构造函数。

当使用**new**在堆上创建对象数组时，还必须多做一些操作。下面是一个创建对象数组的例子：

```
MyType* fp = new MyType[100];
```

这样在堆上为100个**MyType**对象分配了足够的内存并为每一个对象调用了构造函数。但是现在，仅拥有一个**MyType***。它和用下面的表达式创建单个对象得到的结果是一样的：

```
MyType* fp2 = new MyType;
```

因为这是我们写的代码，所以我们知道**fp**实际上是一个数组的起始地址，所以可以使用类似于**fp[3]**的形式来选择数组的元素。但销毁这个数组时发生了什么呢？下面的语句看起来是完全一样的：

563

```
delete fp2; // OK
delete fp;  // Not the desired effect
```

并且它们的效果也应该是一样：为所给地址指向的**MyType**对象调用析构函数，然后释放内存。对于**fp2**，这样是正确的，但对于**fp**，另外99个析构函数没有调用。适当数量的存储单元会被释放，但是，由于它们被分配在一个整块的内存中，所以，整个内存块的大小被分配程序在某处中断了。

解决办法是给编译器一个信息，说明它实际上是一个数组的起始地址。这可以用下面的语法来实现：

```
delete []fp;
```

空的方括号告诉编译器产生代码，该代码的任务是将从数组创建时存放在某处的对象数量取回，并为数组的所有对象调用析构函数。这实际上是对以前形式的改良，我们偶尔仍可

以在旧版本中看到如下的代码:

```
delete [100]fp;
```

这个语法强迫程序设计者加入数组中对象的数量，但程序设计者有可能把对象的数量弄错。而让编译器处理这件事引起的附加代价是很低的，所以只在一个地方指明对象数量要比在两个地方指明好些。

13.3.1 使指针更像数组

作为题外话，上面定义的**fp**可以被修改指向任何类型，但这对于一个数组的起始地址来讲没有什么意义。一般讲来，把它定义为常量会更好些，因为这样任何修改指针的企图都会被认为出错。为了得到这个效果，可以试着用下面的表达式:

```
int const* q = new int[10];
```

或

564

```
const int* q = new int[10];
```

上面的这两种表达式都把**const**和被指针指向的**int**捆绑在一起，而不是指针本身。如果使用下面的表达式:

```
int* const q = new int[10];
```

则现在**q**中的数组元素可以被修改了，但对**q**本身的修改（例如**q++**）是不合法的，因为它是一个普通数组标识符。

13.4 耗尽内存

当**operator new( )**找不到足够大的连续内存块来安排对象时，将会发生什么事情呢？一个称为*new-handler*的特殊函数将会被调用。首先，检查指向函数的指针，如果指针非0，那么它指向的函数将被调用。

new-handler的默认动作是产生一个异常(*throw an exception*)，这个主题将在第2卷中介绍。然而，如果我们在程序里用堆分配，至少要用“内存已耗尽”的信息代替new-handler，并异常中断程序。用这个办法，在调试程序时会得到程序出了什么错误的线索。对于最终的程序，我们总想使之具有很强的容错恢复性。

通过包含**new.h**来替换new-handler，然后以想装入的函数地址为参数调用**set_new_handler( )**函数。

```
//: C13:NewHandler.cpp
// Changing the new-handler
#include <iostream>
#include <cstdlib>
#include <new>
using namespace std;

int count = 0;

void out_of_memory() {
  cerr << "memory exhausted after " << count
    << " allocations!" << endl;
```

565

```
  exit(1);
}

int main() {
  set_new_handler(out_of_memory);
  while(1) {
    count++;
    new int[1000]; // Exhausts memory
  }
} ///:~
```

new-handler 函数必须不带参数且其返回值为**void**。**while**循环将持续分配**int**对象（并丢掉它们的返回地址）直到空的内存被耗尽。在紧接下去的下一次对**new**的调用时，将没有内存可被调用，所以调用new-handler。

new-handler试着调用**operator new( )**，如果已经重载了**operator new( )**（在下一节中介绍），则new-handle将不会按默认调用。如果仍想调用new-handler，则我们不得不在重载了的**operator new( )**的代码中加上做这些工作的代码。

当然，可以写更复杂的new-handler，甚至它可以回收内存［通常叫做*无用单元收集器*（*garbage collector*）］。但这不是编程新手的工作。

13.5 重载new和delete

当我们创建一个**new**表达式时，会发生两件事。首先，使用**operator new( )**分配内存，然后调用构造函数。在delete表达式里，调用了析构函数，然后使用**operator delete( )**释放内存。我们无法控制构造函数和析构函数的调用（否则可能会意外地搅乱它们），但可以改变内存分配函数**operator new( )**和**operator delete( )**。

566 使用了**new**和**delete**的内存分配系统是为通用目的而设计的。但在特殊的情形下，它并不能满足需要。最常见的改变分配系统的原因是出于效率考虑：也许要创建和销毁一个特定的类的非常多的对象以至于这个运算变成了速度的瓶颈。C++允许重载**new**和**delete**来实现我们自己的存储分配方案，所以可以用它来处理问题。

另一个问题是堆碎片：分配不同大小的内存可能会在堆上产生很多碎片，以至于很快用完内存。虽然内存可能还有，但由于都是碎片，也就找不到足够大的内存块满足需要。通过为特定类创建自己的内存分配器，可以确保这种情况不会发生。

在嵌入和实时系统里，程序可能必须在有限的资源情况下运行很长时间。这样的系统也可能要求分配内存花费相同的时间且不允许出现堆内存耗尽或出现很多碎片的情况。由客户定制的内存分配器是一种解决办法，否则程序设计者在这种情况下要避免使用new和delete，而这将错过了C++很有价值的优点。

当重载**operator new( )**和**operator delete( )**时，我们只是改变了原有的内存分配方法，记住这一点是很重要的。编译器将用重载的**new**代替默认的版本去分配内存，然后为那个内存调用构造函数。所以，虽然当编译器看到**new**时，编译器分配内存并调用构造函数，但是当重载**new**时，可以改变的只是内存分配部分（**delete**也有相似的限制。）。

当重载**operator new( )**时，也可以替换它用完内存时的行为，所以必须在**operator new( )**里决定做什么：返回0、写一个调用new-handler的循环、再试着分配或者（典型的）产生一个**bad_alloc**的异常信息（在第2卷中讨论，可从www.BruceEckel.com处获得）。

重载**new**和**delete**与重载任何其他运算符一样。但可以选择重载全局内存分配函数或者是针对特定类的分配函数。 567

13.5.1 重载全局new和delete

当全局版本的**new**和**delete**不能满足整个系统时，对其重载是很极端的方法。如果重载全局版本，就使默认版本完全不能被访问——甚至在这个重新定义里也不能调用它们。

重载的**new**必须有一个**size_t**参数（sizes的标准C类型）。这个参数由编译器产生并传递给我们，它是要分配内存的对象的长度。必须返回一个指向等于这个长度（或大于这个长度，如果有这样做的原因）的对象的指针，如果没有找到存储单元（在这种情况下，构造函数不被调用），则返回一个0。然而如果找不到存储单元，不能仅仅返回0，也许还应该做一些诸如调用new-handler或产生一个异常信息之类的事，通知这里存在问题。

operator new( )的返回值是一个**void***，而不是指向任何特定类型的指针。所做的是分配内存，而不是完成一个对象建立——直到构造函数调用了才完成对象的创建，它是编译器确保做的动作，不在我们的控制范围之内。

operator delete( )的参数是一个指向由**operator new( )**分配的内存的**void***。参数是一个**void***是因为它是在调用析构函数后得到的指针。析构函数从存储单元里移去对象。**operator delete( )**的返回类型是**void**。

下面提供了一个如何重载全局**new**和**delete**的简单的例子：

```
//: C13:GlobalOperatorNew.cpp
// Overload global new/delete
#include <cstdio>
#include <cstdlib>
using namespace std;

void* operator new(size_t sz) {
  printf("operator new: %d Bytes\n", sz);
  void* m = malloc(sz);
  if(!m) puts("out of memory");
  return m;
}

void operator delete(void* m) {
  puts("operator delete");
  free(m);
}

class S {
  int i[100];
public:
  S() { puts("S::S()"); }
  ~S() { puts("S::~S()"); }
};

int main() {
  puts("creating & destroying an int");
  int* p = new int(47);
  delete p;
  puts("creating & destroying an s");
```

568

```
  S* s = new S;
  delete s;
  puts("creating & destroying S[3]");
  S* sa = new S[3];
  delete []sa;
} ///:~
```

这里可以看到重载**new**和**delete**的通常形式。这里的内存分配使用了标准C库函数**malloc( )**和**free( )**（可能默认的**new**和**delete**也使用这些函数）。并且,它们还打印出了有关正在做什么的信息。注意，这里使用**printf( )**和**puts( )**而不是**iostreams**。因此，当创建了一个**iostream**对象时（像全局的**cin**、**cout**和**cerr**），它们调用**new**去分配内存。用**printf( )**不会进入死锁状态，因为它不调用**new**来初始化本身。

在**main( )**里，创建内部数据类型对象以证明在这种情况下也调用重载的**new**和**delete**。然后创建一个类型**S**的单个对象，接着创建一个类型**S**的数组。对于这个数组，从所需要的字节数目中可以看到,额外的内存被分配用于存放它所包含对象的数量的信息。在所有情况下，都
569 使用了全局重载版本的**new**和**delete**。

13.5.2 对于一个类重载new和delete

为一个类重载**new**和**delete**时，尽管不必显式地使用**static**，但实际上仍是在创建**static**成员函数。它的语法也和重载任何其他运算符一样。当编译器看到使用**new**创建自己定义的类的对象时，它选择成员版本的**operator new( )**而不是全局版本的**new( )**。但全局版本的**new**和**delete**仍为所有其他类型对象使用（除非它们有自己的**new**和**delete**）。

在下面的例子里为类**Framis**创建了一个非常简单的内存分配系统。程序开始时在静态数据区域留出一块存储单元。这块内存被用来为**Framis**类型的对象分配存储空间。为了标明哪块存储单元已被使用，这里使用了一个字节（byte）数组，一个字节代表一块存储单元。

```
//: C13:Framis.cpp
// Local overloaded new & delete
#include <cstddef> // Size_t
#include <fstream>
#include <iostream>
#include <new>
using namespace std;
ofstream out("Framis.out");

class Framis {
  enum { sz = 10 };
  char c[sz]; // To take up space, not used
  static unsigned char pool[];
  static bool alloc_map[];
public:
  enum { psize = 100 };  // frami allowed
  Framis() { out << "Framis()\n"; }
  ~Framis() { out << "~Framis() ... "; }
  void* operator new(size_t) throw(bad_alloc);
  void operator delete(void*);
};
unsigned char Framis::pool[psize * sizeof(Framis)];
bool Framis::alloc_map[psize] = {false};
```

570

```
// Size is ignored -- assume a Framis object
void*
Framis::operator new(size_t) throw(bad_alloc) {
  for(int i = 0; i < psize; i++)
    if(!alloc_map[i]) {
      out << "using block " << i << " ... ";
      alloc_map[i] = true; // Mark it used
      return pool + (i * sizeof(Framis));
    }
  out << "out of memory" << endl;
  throw bad_alloc();
}

void Framis::operator delete(void* m) {
  if(!m) return; // Check for null pointer
  // Assume it was created in the pool
  // Calculate which block number it is:
  unsigned long block = (unsigned long)m
    - (unsigned long)pool;
  block /= sizeof(Framis);
  out << "freeing block " << block << endl;
  // Mark it free:
  alloc_map[block] = false;
}

int main() {
  Framis* f[Framis::psize];
  try {
    for(int i = 0; i < Framis::psize; i++)
      f[i] = new Framis;
    new Framis; // Out of memory
  } catch(bad_alloc) {
    cerr << "Out of memory!" << endl;
  }
  delete f[10];
  f[10] = 0;
  // Use released memory:
  Framis* x = new Framis;
  delete x;
  for(int j = 0; j < Framis::psize; j++)
    delete f[j]; // Delete f[10] OK
} ///:~
```

571

通过创建一个能够容纳**psize**个**Framis**对象的字节数组的方法，为**Framis**堆分配了内存。相应地,分配表中也会含有**psize**个成员，其中每一**bool**类型成员对应一块内存。初始化时，分配表中所有的值都被置为**false**，这可以使用设置首元素的集合初始化技巧，因为编译器能够自动地以常规的预设值来初始化其余的元素（对于**bool**类型来说，就是初始化为**false**）。

局部**operator new( )**和全局**operator new( )**具有相同的语法。首先对分配表进行搜索,寻找值为**false**的成员。找到后将该成员设置为**ture**，以此声明对应的存储单元已经被分配了,并且返回这个存储单元的地址。如果找不到任何空闲内存，将会给跟踪文件发送一个消息,并且产生一个**bad_alloc**类型的异常信息.

这是在这本书中看到的第一个含有异常情况的例子。因为有关异常情况的详细的讨论被

放在了第2卷，所以这里只是一个简单的例子。在**operator new( )**中，可以看到两个异常情况处理的标志。首先是在函数参数表后面的**throw(bad_alloc)**，它通知了编译器和读者这个函数可以产生一个**bad_alloc**的异常信息。其次,如果没有内存可供使用了，则此函数会由**throw bad_alloc**语句产生一个异常信息。当此异常信息产生时，函数停止执行并且把控制权交给表示为一个**catch**子句的*异常处理*（*exception handler*）。

在**main( )**中，可以看到异常处理的其余部分，也就是*try-catch*子句。被大括号围起的***Try***部分包含了可以产生异常信息的所有代码——在这里就是指任何对含有**Framis**对象的**new**的调用。跟在**try**部分后面的是一个或多个**catch**子句，每一个都指明了它们获取的异常信息的类型。在本例中，**catch(bad_alloc)**指明了**bad_alloc**类型的异常信息在此可被获取。这里的**catch**子句仅当**bad_alloc**类型异常信息生成时才执行，并且执行是在这组**catch**子句的最后一个结束后开始的（这里只有一个**catch**子句，但在别的程序中可以有多个）。

572 在本例中，因为没有涉及到全局**operator new( )**和**delete( )**，所以使用iostreams是可行的。

operator delete( )假设**Framis**的地址是在这个堆里创建的，这是一个正确的假设。因为无论何时我们在堆上创建单个的**Framis**对象——不是一个数组，都将调用局部**operator new( )**。而全局版本的**new( )**在创建数组时使用。因此用户可能会在用**operator delete( )**删除一个数组时，偶然地忘记了使用空方括号语法，而这就会出现问题。用户也可能删除了在栈上创建的指向对象的指针。如果考虑到这样的事情可能发生，应该加入一行代码以确保地址是在这个堆内并是在正确的地址范围内（也可以考虑重载**new**和**delete**对于防止内存丢失的潜力。）。

operator delete( )计算出该指针所代表的那块内存，并在分配表中将对应部分置为false，以表明这块内存已经被释放了。

在**main( )**中，动态地分配足够多的**Framis**对象，把可用内存消耗掉。这用来检查无内存可供分配的情况。然后释放一个对象，再创建一个对象以表明释放的内存可被重新使用。

因为这个内存分配方案是针对**Framis**对象的，所以可能比使用默认的**new**和**delete**的通用内存分配方案效率要高一些。但是，应当注意，如果使用继承，该分配方案不能自动继承使用（继承将在第14章中介绍。）。

13.5.3 为数组重载new和delete

如果为一个类重载了**operator new( )**和**operator delete( )**，那么无论何时创建这个类的一个对象都将调用这些运算符。但如果要创建这个类的一个对象数组时，全局**operator new( )**就会被立即调用，用来为这个数组分配足够的内存。对此，可以通过为这个类重载运算符的数组版本，

573 即**operator new[ ]**和**operator delete[ ]**，来控制对象数组的内存分配。下面的例子显示了何时这两个不同的版本会被调用:

```
//: C13:ArrayOperatorNew.cpp
// Operator new for arrays
#include <new> // Size_t definition
#include <fstream>
using namespace std;
ofstream trace("ArrayOperatorNew.out");

class Widget {
  enum { sz = 10 };
  int i[sz];
```

```
public:
  Widget() { trace << "*"; }
  ~Widget() { trace << "~"; }
  void* operator new(size_t sz) {
    trace << "Widget::new: "
          << sz << " bytes" << endl;
    return ::new char[sz];
  }
  void operator delete(void* p) {
    trace << "Widget::delete" << endl;
    ::delete []p;
  }
  void* operator new[](size_t sz) {
    trace << "Widget::new[]: "
          << sz << " bytes" << endl;
    return ::new char[sz];
  }
  void operator delete[](void* p) {
    trace << "Widget::delete[]" << endl;
    ::delete []p;
  }
};

int main() {
  trace << "new Widget" << endl;
  Widget* w = new Widget;
  trace << "\ndelete Widget" << endl;
  delete w;
  trace << "\nnew Widget[25]" << endl;
  Widget* wa = new Widget[25];
  trace << "\ndelete []Widget" << endl;
  delete []wa;
} ///:~
```

574

这里，全局版本的**new**和**delete**被调用，除了加入了跟踪信息以外，它们和没有**new**和**delete**的重载版本效果是一样的。当然，可以在重载的**new**和**delete**里使用任意的内存分配方案。

可以看到，在语法上，除了多一对括号外，数组版本的**new**和**delete**与单个对象版本的是一样的。不管是哪种版本，我们都要决定所要分配内存的大小。数组版本中的大小指的是整个数组的大小。应该记住，重载**operator new( )**惟一需要做的是返回一个足够大的内存块的指针。虽然可以初始化那块内存，但通常编译器将自动地调用构造函数来对该内存块进行初始化。

这里构造函数和析构函数只是打印出字符，因此可以看到什么时候它们被调用。下面是某个编译器生成的跟踪文件的输出信息：

```
new Widget
Widget::new: 40 bytes
*
delete Widget
~Widget::delete

new Widget[25]
Widget::new[]: 1004 bytes
*************************
delete []Widget
~~~~~~~~~~~~~~~~~~~~~~~~~Widget::delete[]
```

正如所预计的，创建单个对象需要40个字节（本机为**int**类型分配 4 个字节）。首先调用**operator new( )**，接着调用了构造函数（这可从输出信息*看出）。在后面的部分，对**delete**的调用首先引起了析构函数的调用，然后是对**operator delete( )**的调用。

575 当创建一个**Widget**类型的对象数组时，使用了数组版本的**operator new( )**。但请注意，需要的长度比期望的多了4个字节。这额外的4个字节是系统用来存放数组信息的，特别是数组中对象的数量。当用下面的表达式时，

```
delete []Widget;
```

方括号就告诉编译器它是一个对象数组，所以编译器产生寻找数组中对象的数量的代码，然后多次调用析构函数。可以看到，即使数组**operator new( )**和**operator delete( )**只为整个数组调用一次，但对于数组中的每一个对象，都调用了默认的构造函数和析构函数。

13.5.4 构造函数调用

分析下面语句:

```
MyType* f = new MyType;
```

调用**new**分配了一个大小等于**MyType**类型的内存，然后在那个内存上调用了**MyType**构造函数。但如果使用了**new**的内存分配没有成功，将会出现什么状况呢？在那种情况下，构造函数不会被调用，所以虽然没能成功地创建对象，但至少没有调用构造函数并传给它一个为0的**this**指针。下面的例子说明了这一点:

```
//: C13:NoMemory.cpp
// Constructor isn't called if new fails
#include <iostream>
#include <new> // bad_alloc definition
using namespace std;

class NoMemory {
public:
  NoMemory() {
    cout << "NoMemory::NoMemory()" << endl;
  }
  void* operator new(size_t sz) throw(bad_alloc){
    cout << "NoMemory::operator new" << endl;
    throw bad_alloc(); // "Out of memory"
  }
576 };
int main() {
  NoMemory* nm = 0;
  try {
    nm = new NoMemory;
  } catch(bad_alloc) {
    cerr << "Out of memory exception" << endl;
  }
  cout << "nm = " << nm << endl;
} ///:~
```

当程序运行时，并没有打印出构造函数的信息，仅仅是打印了**operator new( )**和异常处理的信息。因为**new** 没有返回，构造函数也没有被调用，当然它的信息就不会被打印出来。

nm被初始化为0是很重要的，因为**new**表达式没有执行完毕，指针被置为0可以确保我们没有误用它。但是,在异常处理中，我们除了打印出一条信息以外，还应当多做一些事情，使得程序继续执行，就像该对象已经被成功地创建了一样。理想情况下，我们所做的将使程序从问题中恢复过来，或者至少可以在记录下错误后退出。

在以前的C++版本中，如果内存分配失败，则一般是返回0。它将使构造函数不被调用。但是，如果试着在一个标准的编译器中由**new**返回0值，则会被告之应该产生一个**bad_alloc**。

13.5.5 定位new和delete

重载**operator new( )**还有其他两个不常见的用途。

1) 我们也许会想在内存的指定位置上放置一个对象。这对于面向硬件的内嵌系统特别重要，在这个系统中，一个对象可能和一个特定的硬件是同义的。

2) 我们也许会想在调用**new**时，能够选择不同的内存分配方案。 577

这两个特性可以用相同的机制实现：重载的**operator new( )**可以带一个或多个参数。正如前面所看到的，第一个参数总是对象的长度，它在内部计算出来并由编译器传递给**new**。但其他参数可由我们自己定义：一个放置对象的地址、一个是对内存分配函数或对象的引用,或其他任何使我们方便的设置。

最初在调用过程中传递额外的参数给**operator new( )**的方法看起来似乎有点古怪：在关键字**new**后是参数表（没有**size_t**参数，它由编译器处理），参数表后面是正在创建的对象的类名字。例如:

```
X* xp = new(a) X;
```

将**a**作为第二个参数传递给**operator new( )**。当然，这是在**operator new( )**已经声明的情况下才是有效的。

下面的例子显示了如何在一个特定的存储单元里放置一个对象。

```
//: C13:PlacementOperatorNew.cpp
// Placement with operator new()
#include <cstddef> // Size_t
#include <iostream>
using namespace std;

class X {
  int i;
public:
  X(int ii = 0) : i(ii) {
    cout << "this = " << this << endl;
  }
  ~X() {
    cout << "X::~X(): " << this << endl;
  }
  void* operator new(size_t, void* loc) {
    return loc;
  }
};

int main() {
  int l[10];
```

578

```
  cout << "l = " << l << endl;
  X* xp = new(l) X(47); // X at location l
  xp->X::~X(); // Explicit destructor call
  // ONLY use with placement!
} ///:~
```

注意：**operator new( )**仅返回了传递给它的指针。因此，调用者可以决定将对象存放在哪里，这时在该指针所指向的那块内存上，作为**new**表达式一部分的构造函数将被调用。

虽然本例只是使用了一个外加的参数，但如果需要实现其他的目的时，使用更多的参数同样也是可以的。

在销毁对象时将会出现两难选择的局面。因为仅有一个版本的**operator delete**，所以没有办法说“对这个对象使用我的特殊内存释放器”。可以调用析构函数，但不能用动态内存机制释放内存，因为内存不是在堆上分配的。

解决方法是用非常特殊的语法：我们可以显式地调用析构函数。例如：

```
xp->X::~X(); // Explicit destructor call
```

这里要严重警告一下。因为当某些人想要实时地决定对象的生存时间时，他们使用这种方法在作用范围结束之前的任意时刻销毁对象，而不是调节作用范围或者使用动态对象创建（这样做会更正确）。而如果用这种方法为在栈上创建的对象调用析构函数时，将会出现严重的问题，这是因为析构函数在对象超出作用范围时又会被调用一次。如果为在堆上创建的对象用这种方法调用析构函数，析构函数将被执行，但内存不释放，这是我们所不希望的。用这种方法显式地调用析构函数，其实只有一个原因，即支持**operator new( )**的定位语法。

579 还有一个定位**operator delete**，它仅在一个定位**operator new**表达式的构造函数产生一个异常信息时才被调用（因此该内存在异常处理操作中被自动地清除了）。定位**operator delete**有一个和定位**operator new**相对应的参数表，该定位**operator new**是指在构造函数产生异常信息之前被调用的那一个。这个主题将放在第2卷的异常处理章节中。

13.6 小结

在栈上创建自动对象既方便又理想，但为了解决常见的程序问题，必须在程序执行的任何时候，特别是需要对来自程序外部信息作出反应时，能够创建和销毁对象。虽然C的动态内存分配可以从堆上得到内存，但它在C++上不易使用并且不能够保证安全。使用**new**和**delete**进行动态对象创建，这已经成为语言的核心，它可以使我们在堆上创建对象像在栈上创建对象一样容易。另外，它还增加了程序的灵活性。如果**new**和**delete**不能满足要求，尤其是它们的效率不高时，程序员可以改变它们的行为，而且在堆的内存用完时可以修改它们所执行的操作。

13.7 练习

部分练习题的答案可以在本书的电子文档“*Annotated Solution Guide for Thinking in C++*”中找到，只需支付很少的费用就可以从http://www.BruceEckel.com得到这个电子文档。

13-1 创建一个**class Counted**，它包含一个**int**类型的成员变量**id**和一个**static int**类型的成员变量**count**。默认构造函数的开头为**Counted( ) : id(count++) {**。要求构造函数打印**id**值并且输出“it's being created”。另外析构函数也打印出**id**值并且输出“it's

being destroyed"。测试这个类。

13-2 通过使用**new**创建一个（练习 1 中的）**class Counted**的对象，并且用**delete**销毁它，来证明**new**和**delete**总是调用了构造函数和析构函数。在堆上创建和销毁这些对象的一个数组。 580

13-3 创建一个**PStash**对象，在此对象中用**new**创建练习1的对象。观察当这个对象超出了范围和它的析构函数被调用时有什么情况发生。

13-4 创建一个**vector< Counted*>**，对它使用**new**创建(练习1中的)。**Counted**对象时返回的指针填充。扫描这个**vector**并输出**Counted**对象，然后再次扫描，并删除每一个对象。

13-5 重复练习4的操作，只是增加一个**Counted**的成员函数**f( )**，该函数可以输出一条信息。然后扫描这个**vector**并对每一个对象调用函数**f( )**。

13-6 使用**PStash**重复练习5的操作。

13-7 使用第9章的**Stack4.h**重复练习5的操作。

13-8 动态创建一个（练习1中的）**class Counted**的对象数组。不使用方括号对返回指针调用**delete**。对此操作的结果进行解释。

13-9 使用**new**创建一个（练习1中的）**class Counted**的对象，对**void***类型的返回指针进行类型转换，然后再删除它。对此运算结果进行解释。

13-10 在计算机上执行**NewHandler.cpp**，观察变量**count**的最终结果。计算可供我们的程序使用的空闲内存的数量。

13-11 创建一个类,带有重载运算符**new**和**delete**，要求含有对于单个对象和数组的两个版本。演示这两个版本的工作情况。

13-12 设计一个对**Framis.cpp**进行测试的程序来显示定制的**new**和**delete**比全局的**new**和**delete**大约快多少。

13-13 修改**NoMemory.cpp**让它含有一个**int**类型的数组。使它实际上没有产生**bad_alloc**，而是分配了内存。在**main( )**中，建立一个像在**NewHandler.cpp**中的**while**循环，用来消耗完内存，观察一下当**operator new**没有测试内存是否被成功地分配时会有什么发生。然后在**operator new**中加入测试并产生**bad_alloc**。

13-14 创建一个含有定位**new**运算符的类，定位**new**运算符的第二个参数是一个**string**类型值。这个类还包括一个**static vector<string>**，用来存放第二个参数。该定位 581
new运算符同常规的一样用来分配内存。在**main( )**中，调用定位**new**并且以描述该调用的字符串作为**string**参数（可能要用到预处理的**__FILE__**和**__LINE__**宏）。

13-15 通过增加**static vector<Widget*>**来修改**ArrayOperatorNew.cpp**，即加入每一个**Widget**地址。该**Widget**地址是由**operator new( )**分配内存并且当它被释放时可以通过**operator delete( )**删除。（我们可能需要在标准C++库文件或者在本书的第2卷（可从**Web**站点中获得）中查找有关**vector**的信息。）创建第二个类**MemoryChecker**，含有可以打印出**vector**中**Widget**指针数目的析构函数。再创建一个含有**MemoryChecker**对象的程序。在**main( )**中，动态地分配且销毁一些**Widget**的对象和数组。显示**MemoryChecker**可以阻止内存丢失。 582

第14章 继承和组合

C++最重要的特征之一是代码重用。但是如果希望更进一步，就不能仅仅用拷贝
583 代码和修改代码的方法，而是要做更多的工作。

在C中，这个问题未能得到很好的解决。而在C++中，这可以通过类的方法解决。我们通过创建新类来重用代码，而不是从头创建它们。这样，便可以使用别人已经创建好并经过调试的类。

关键技巧是使用这些类，但不修改已存在的代码。在本章中，我们将看到两种完成这项任务的方法。第一种方法很直接：我们简单地在新类中创建已存在类的对象。因为新类是由已存在类的对象组合而成，所以这种方法称为组合（*composition*）。

第二种方法要复杂些。我们创建一个新类作为一个已存在类的类型。我们不修改已存在的类，而是采取这个已存在类的形式，并将代码加入其中。这种巧妙的方法称为继承（*inheritance*），其中大量的工作是由编译器完成。继承是面向对象程序设计的基石，而且它还有另外的含义，我们将在下一章中探讨它的另外含义。

在语法上和行为上，组合和继承大部分是相似的（它们都是在已存在类型的基础上创建新类型的方法）。在本章中，我们将学习这些代码重用机制。

14.1 组合语法

实际上，我们一直都在用组合创建类，只不过是在用内部数据类型（有时用**string**）组合新类。其实使用用户定义类型组合新类同样很容易。

考虑下面这个在某种意义上是有价值的类：

```
//: C14:Useful.h
// A class to reuse
#ifndef USEFUL_H
#define USEFUL_H

584 class X {
  int i;
public:
  X() { i = 0; }
  void set(int ii) { i = ii; }
  int read() const { return i; }
  int permute() { return i = i * 47; }
};
#endif // USEFUL_H ///:~
```

在**X**类中，数值成员是私有的，所以将类型**X**的一个对象作为公共对象嵌入到一个新类内部，这是绝对安全的。这样就使得新类的接口很简单，

```
//: C14:Composition.cpp
// Reuse code with composition
```

```
#include "Useful.h"

class Y {
  int i;
public:
  X x; // Embedded object
  Y() { i = 0; }
  void f(int ii) { i = ii; }
  int g() const { return i; }
};

int main() {
  Y y;
  y.f(47);
  y.x.set(37); // Access the embedded object
} ///:~
```

访问嵌入对象（称为子对象）的成员函数只需再一次的成员选择。

更常见的是把嵌入的对象设为私有，因此它们将成为内部实现的一部分（这意味着如果我们愿意，可以改变这个实现）。新类的公有接口函数包括了对嵌入对象的使用，但没有必要模仿这个对象的接口。

```
//: C14:Composition2.cpp
// Private embedded objects
#include "Useful.h"
class Y {
  int i;
  X x; // Embedded object
public:
  Y() { i = 0; }
  void f(int ii) { i = ii; x.set(ii); }
  int g() const { return i * x.read(); }
  void permute() { x.permute(); }
};

int main() {
  Y y;
  y.f(47);
  y.permute();
} ///:~
```

585

这里，**permute( )**函数是通过新类的接口执行的，但**X**的其他的成员函数是在新类的成员**Y**内执行的。

14.2 继承语法

组合的语法是清晰的，而对于继承，则有新的不同的形式。

当继承时，我们会发现“这个新类很像原来的类”。我们规定，在代码中和原来一样给出该类的名字，但在类的左括号的前面，加一个冒号和基类的名字（对于多重继承，要给出多个基类名，它们之间用逗号分开）。当做完这些时，将会自动地得到基类中的所用数据成员和成员函数。下面是一个例子：

```
//: C14:Inheritance.cpp
```

```
// Simple inheritance
#include "Useful.h"
#include <iostream>
using namespace std;

class Y : public X {
  int i; // Different from X's i
public:
  Y() { i = 0; }
  int change() {
    i = permute(); // Different name call
    return i;
  }
  void set(int ii) {
    i = ii;
    X::set(ii); // Same-name function call
  }
};

int main() {
  cout << "sizeof(X) = " << sizeof(X) << endl;
  cout << "sizeof(Y) = "
       << sizeof(Y) << endl;
  Y D;
  D.change();
  // X function interface comes through:
  D.read();
  D.permute();
  // Redefined functions hide base versions:
  D.set(12);
} ///:~
```

586

我们可以看到Y对X进行了继承，这意味着Y将包含X中的所有数据成员和成员函数。实际上，正如没有对X进行继承，而在Y中创建了一个X的成员对象一样，Y是包含了X的一个子对象。无论是成员对象还是基类存储，都被认为是子对象。

所有X中的私有成员在Y中仍然是私有的，这是因为Y对X进行了继承并不意味着Y可以不遵守保护机制。X中的私有成员仍然占有存储空间，只是不可以直接地访问它们罢了。

在**main( )**中，从**sizeof(Y)**是**sizeof(X)**的两倍可以看出，Y的数据成员是同X的成员结合在一起了。

587

我们注意到，本例中的基类前面是**public**。由于在继承时，基类中所有的成员都是被预设为私有的，所以如果基类的前面没有**public**，这意味着基类的所有公有成员将在派生类中变为私有的。这显然不是所希望的[⊖]，我们希望基类中的所有公有成员在派生类中仍是公有的。这可以在继承时通过使用关键字**public**来实现。

在**change( )**中，基类的**permute( )**函数被调用。即派生类可以直接访问所有基类的公有函数。

派生类中的**set( )**函数重新定义了基类中**set( )**函数。这即是说，如果调用一个Y类型对象的**read( )**和**permute( )**函数，将会使用基类中的这些函数（这可在**main( )**中表现出来）。但如果调用一个Y类型对象的**set( )**函数，将会使用派生类中的重定义版本。这意味着如果不想使用某个继承而来的函数，我们可以改变它的内容（当然我们也可以增加全新的函数，例如

⊖ 在Java中，编译器不会因继承而让程序员减少对成员的访问能力。

change( )。

然而，当我们重新定义了一个函数的后，仍可能想调用基类的函数。但如果对于**set( )**，只是简单地调用**set( )**函数，将得到这个函数的本地版本——一个递归的函数调用。为了调用基类的**set( )**函数，必须使用作用域运算符来显式地标明基类名。

14.3 构造函数的初始化表达式表

已经看到，在C++中保证正确的初始化是多么重要，这一点在组合和继承中也是一样。当创建一个对象时，编译器确保调用了所有子对象的构造函数。到目前为止，在已有的例子中，所有子对象都有默认的构造函数，编译器可以自动调用它们。但是，如果子对象没有默认构造函数或如果想改变构造函数的某个默认参数，情况怎么样呢？这会出现问题的，因为这个 588
新类的构造函数没有权利访问这个子对象的私有数据成员，所以不能直接地对它们初始化。

解决的方法很简单：对于子对象调用构造函数，C++为此提供了专门的语法，即构造函数的初始化表达式表。构造函数的初始化表达式表的形式模仿继承活动。对于继承，我们把基类置于冒号和这个类体的左括号之间。而在构造函数的初始化表达式表中，可以将对子对象构造函数的调用语句放在构造函数参数表和冒号之后，在函数体的左括号之前。对于从**Bar**继承来的类**MyType**，如果**Bar**的构造函数只有一个**int**型参数，则可以表示为：

```
MyType::MyType(int i) : Bar(i) { // ...
```

14.3.1 成员对象初始化

显然，对于组合，也可以对成员对象使用同样语法，只是所给出的不是类名，而是对象的名字。如果在初始化表达式表中有多个构造函数的调用，应当用逗号加以隔开：

```
MyType2::MyType2(int i) : Bar(i), m(i+1) { // ...
```

这是类**MyType2**构造函数的开头，该类是从**Bar**继承来的，并且包含一个称为**m**的成员对象。请注意，虽然可以在这个构造函数的初始化表达式表中看到基类的类型，但只能看到成员对象的标识符。

14.3.2 在初始化表达式表中的内部类型

构造函数的初始化表达式表允许我们显式地调用成员对象的构造函数。事实上，也没有其他方法可以调用那些构造函数。它的主要思想是，在进入新类的构造函数体之前调用所有其他的构造函数。这样，对子对象的成员函数所做的任何调用都总是转到了这个被初始化的对象中。即使编译器可以隐藏地调用默认的构造函数，但在没有对所有的成员对象和基类对 589
象的构造函数进行调用之前，就没有办法进入该构造函数体。这是C++的一个强化的机制，它确保了，如果没有调用对象 (或对象的一部分)的构造函数，就别想向下进行。

所有的成员对象在构造函数的左括号之前就被初始化了，这种方法对于程序设计很有帮助。一旦遇到左括号，就认为所有的子对象已被正确地初始化了，我们的精力就可以集中在想要完成的任务上面。然而，还有一个问题：对于那些没有构造函数的内部类型嵌入对象，这一切又将怎样呢？

为了使语法一致，可以把内部类型看做这样一种类型，它只有一个取单个参数的构造函

数，而这个参数与正在初始化的变量的类型相同。于是，可以这样写：

```
//: C14:PseudoConstructor.cpp
class X {
  int i;
  float f;
  char c;
  char* s;
public:
  X() : i(7), f(1.4), c('x'), s("howdy") {}
};

int main() {
  X x;
  int i(100);  // Applied to ordinary definition
  int* ip = new int(47);
} ///:~
```

这些“伪构造函数调用”操作可以进行简单的赋值。这种方法很方便，并且具有良好的编码风格，所以常能看到它使用。

590 甚至当在类之外创建内部类型的变量时，也可以使用伪构造函数语法：

```
int i(100);
int* ip = new int(47);
```

这使得内部类型的操作有点类似于对象。但要记住，这些并不是真正的构造函数。特别地，如果没有显式的进行伪构造函数调用，初始化是不会执行的。

14.4 组合和继承的联合

当然，还可以把组合和继承放在一起使用。下面的例子中通过继承和组合两种方法创建了一个更复杂的类。

```
//: C14:Combined.cpp
// Inheritance & composition

class A {
  int i;
public:
  A(int ii) : i(ii) {}
  ~A() {}
  void f() const {}
};

class B {
  int i;
public:
  B(int ii) : i(ii) {}
  ~B() {}
  void f() const {}
};

class C : public B {
  A a;
public:
```

```
  C(int ii) : B(ii), a(ii) {}
  ~C() {} // Calls ~A() and ~B()
  void f() const {  // Redefinition
    a.f();
    B::f();
  }
};
int main() {
  C c(47);
} ///:~
```

591

C对**B**进行了继承并且有一个类型**A**的成员对象（“由类型**A**的成员对象组合而成”）。可以看到，构造函数的初始化表达式表中调用了基类构造函数和成员对象构造函数。

函数 **C::f( )**重定义了它所继承的**B::f( )**，但同时还调用基类版本。另外，它还调用了**a.f( )**。注意，只有通过继承，才能重新定义它的函数。而对于成员对象，只能操作这个对象的公共接口，而不能重定义它。另外，如果**C::f( )**还没有被定义，则对类型**C**的一个对象调用**f( )**就不会调用**a.f( )**，而会调用**B::f( )**。

自动析构函数调用

虽然常常需要在初始化表达式表中做显式构造函数调用，但并不需要做显式的析构函数调用，因为对于任何类型只有一个析构函数，并且它并不取任何参数。然而，编译器仍要保证所有的析构函数被调用，这意味着，在整个层次中的所有析构函数中，从派生最底层的析构函数开始调用，一直到根层。

重点要注意构造函数和析构函数与众不同之处在于每一层函数都被调用。然而对于通常的成员函数，只是这个函数被调用，而它的那些基类版本并不会被调用。如果还想调用重新定义过的成员函数的基类版本，则必须显式地去做。

14.4.1 构造函数和析构函数调用的次序

当一个对象有许多子对象时，了解构造函数和析构函数的调用次序是很有意思的。下面的例子清楚地表明了调用的次序：

```
//: C14:Order.cpp
// Constructor/destructor order
#include <fstream>
using namespace std;
ofstream out("order.out");

#define CLASS(ID) class ID { \
public: \
  ID(int) { out << #ID " constructor\n"; } \
  ~ID() { out << #ID " destructor\n"; } \
};

CLASS(Base1);
CLASS(Member1);
CLASS(Member2);
CLASS(Member3);
CLASS(Member4);

class Derived1 : public Base1 {
```

592

```
  Member1 m1;
  Member2 m2;
public:
  Derived1(int) : m2(1), m1(2), Base1(3) {
    out << "Derived1 constructor\n";
  }
  ~Derived1() {
    out << "Derived1 destructor\n";
  }
};

class Derived2 : public Derived1 {
  Member3 m3;
  Member4 m4;
public:
  Derived2() : m3(1), Derived1(2), m4(3) {
    out << "Derived2 constructor\n";
  }
  ~Derived2() {
    out << "Derived2 destructor\n";
  }
};

int main() {
  Derived2 d2;
} ///:~
```

593

首先，创建一个**ofstream**对象，用来把所有的输出发送到一个文件中。然后为了在书中少敲一些字符也为了演示一种宏技术（这个技术将在第16章中被一个更好的技术代替），这里使用了宏以建立一些类（这些类将被用于继承和组合）。每个构造函数和析构函数向这个跟踪文件报告它们自己的行动。注意，这些是构造函数，而不是默认构造函数，它们每一个都有一整型参数。这个参数本身没有标识符，它的惟一的任务就是强迫在初始化表达式表中显式调用这些构造函数（消除标识符防止编译器警告信息）。

这个程序的输出是：

```
Base1 constructor
Member1 constructor
Member2 constructor
Derived1 constructor
Member3 constructor
Member4 constructor
Derived2 constructor
Derived2 destructor
Member4 destructor
Member3 destructor
Derived1 destructor
Member2 destructor
Member1 destructor
Base1 destructor
```

可以看出，构造是从类层次的最根处开始，而在每一层，首先会调用基类构造函数，然后调用成员对象构造函数。调用析构函数则严格按照构造函数相反的次序——这是很重要的，因为要考虑潜在的相关性(对于派生类中的构造函数和析构函数，必须假设基类子对象仍然可

供使用,并且已经被构造了——或者还未被消除)。

另一个有趣现象是，对于成员对象，构造函数调用的次序完全不受构造函数的初始化表达式表中的次序影响。该次序是由成员对象在类中声明的次序所决定的。如果能通过构造函数的初始化表达式表改变构造函数调用次序，那么就会对两个不同的构造函数有两种不同的调用顺序。而析构函数将不能知道如何相应逆序地执行析构，这就产生了相关性问题。 594

14.5 名字隐藏

如果继承一个类并且对它的成员函数重新进行定义，可能会出现两种情况。第一种是正如在基类中所进行的定义一样，在派生类的定义中明确地定义操作和返回类型。这称之为对普通成员函数的重定义（*redefining*），而如果基类的成员函数是虚函数的情况，又可称之为重写（*overriding*）（虚函数是一种常见的函数，我们将在第15章详细地进行介绍）。但是如果在派生类中改变了成员函数参数列表和返回类型，会发生什么情况呢？这里有一个例子:

```
//: C14:NameHiding.cpp
// Hiding overloaded names during inheritance
#include <iostream>
#include <string>
using namespace std;

class Base {
public:
  int f() const {
    cout << "Base::f()\n";
    return 1;
  }
  int f(string) const { return 1; }
  void g() {}
};

class Derived1 : public Base {
public:
  void g() const {}
};

class Derived2 : public Base {
public:
  // Redefinition:
  int f() const {
    cout << "Derived2::f()\n";
    return 2;
  }
};

class Derived3 : public Base {
public:
  // Change return type:
  void f() const { cout << "Derived3::f()\n"; }
};

class Derived4 : public Base {
```

595

```
public:
  // Change argument list:
  int f(int) const {
    cout << "Derived4::f()\n";
    return 4;
  }
};

int main() {
  string s("hello");
  Derived1 d1;
  int x = d1.f();
  d1.f(s);
  Derived2 d2;
  x = d2.f();
//!  d2.f(s); // string version hidden
  Derived3 d3;
//!  x = d3.f(); // return int version hidden
  Derived4 d4;
//!  x = d4.f(); // f() version hidden
  x = d4.f(1);
} ///:~
```

在**Base**类中有一个可被重载的函数**f( )**，类**Derived1**并没有对函数**f( )**进行任何改变，但它重新定义了函数**g( )**。在**main( )**中，可以看到函数**f( )**的两个重载版本在类**Derived1**中都是可以使用的。但是，由于类**Derived2**重新定义了函数**f( )**的一个版本，而对另一个版本没有进行
596 重定义，因此这第二个重载形式是不可以使用的。在类**Derived3**中，通过改变返回类型隐藏了基类中的两个函数版本，而在类**Derived4**中，通过改变参数列表同样隐藏了基类中的两个函数版本。总体上，可以得出，任何时候重新定义了基类中的一个重载函数，在新类之中所有其他的版本则被自动地隐藏了。在第15章，我们将会看到加上**virtual**这个关键字会对函数的重载有一点影响。

如果通过修改基类中一个成员函数的操作与/或返回类型来改变了基类的接口，我们就没有使用继承通常所提供的功能，而是按另一种方式来重用了该类。这并不一定意味做错了，只是由于继承的最终目标是为了实现多态性（*polymorphism*）。并且如果我们改变了函数特征或返回类型，实际上便改变了基类的接口。如果这便是想要做的，我们就主要通过继承来重用代码，而无需维护基类的通用接口（这是多态性的一个很重要的方面）。总体上说，当按这种方式使用继承，就意味着我们有一个具有通用目的的类，对于特定的需要再对它进行具体化——虽然不总是这样，但通常这被认为是属于组合的范围。

例如，对于第9章中的**Stack**类，该类的一个问题是我们不得不在每次从容器中取回指针时进行类型变换。这不仅仅很麻烦，而且不安全。我们可以把指针类型转换为指向想要的任何对象上。

初看较好的解决方法是通过继承的方法来具体化通用类**Stack**。下面的例子使用了第9章中的类：

```
//: C14:InheritStack.cpp
// Specializing the Stack class
#include "../C09/Stack4.h"
#include "../require.h"
#include <iostream>
```

```
#include <fstream>
#include <string>
using namespace std;

class StringStack : public Stack {                                    597
public:
  void push(string* str) {
    Stack::push(str);
  }
  string* peek() const {
    return (string*)Stack::peek();
  }
  string* pop() {
    return (string*)Stack::pop();
  }
  ~StringStack() {
    string* top = pop();
    while(top) {
      delete top;
      top = pop();
    }
  }
};

int main() {
  ifstream in("InheritStack.cpp");
  assure(in, "InheritStack.cpp");
  string line;
  StringStack textlines;
  while(getline(in, line))
    textlines.push(new string(line));
  string* s;
  while((s = textlines.pop()) != 0) { // No cast!
    cout << *s << endl;
    delete s;
  }
} ///:~
```

因为所有**Stack4.h**的成员函数都是内联的，所以不再需要进行链接。

StringStack类具体化了**Stack**类，所以**push( )**将仅接收**String**类型指针。以前，**Stack**类接收**void**类型指针，但用户没有进行类型检查以确保插入了正确的指针。另外，**peek( )**和**pop( )**现在返回了**String**类型指针，而不是**void**类型指针，所以使用这些指针就无需进行类型转换了。

598

很不可思议，在**push( )**、**peek( )**和**pop( )**中不需要额外的类型安全检查！在编译的时候，编译器会收到额外的类型信息，但这些函数是内联的并且不会产生额外的代码。

名字隐藏在这里起了作用，这主要是因为**push( )**函数有着不同的特征：参数列表是不同的。如果在同一个类中含有两个版本的**push( )**，它们将会被重载，但在这里并不希望进行重载，这是由于它仍将允许我们把任何类型的指针作为**void***类型传送给**push( )**。幸运的是，当**push( )**函数的新版本在派生类中被定义后，C++将把基类中的**push(void*)**版本隐藏起来，因此这时仅允许向**StringStack**中推入**push( ) string**指针。

由于现在可以确保我们清楚地知道在容器中的是何种类型的对象，所以析构函数能正

确地执行，同时对象的所属问题也就被解决了——或者说至少是解决对象所属问题的一种方法。这里，如果**push( )**一个**string**指针进入**StringStack**，然后（根据**StringStack**的语义）我们也可以传递该指针的所属类给**StringStack**。如果**pop( )**这个指针，将不仅可以得到该指针，再且还可以得到该指针的所属类。当调用析构函数时，任何留在**StringStack**中的指针将会被其析构函数消除。由于这里的都是**string**类型的指针，所以**delete**语句将作用于**string**指针，而不会对**void**指针进行操作，于是正确地执行了析构操作，这时一切都正确地运行。

这里有一个缺点：就是这个类仅仅可对**string**指针进行操作。如果想要一个可对某一其他类型的对象进行操作的**Stack**类，我们必须写一个该类的新版本，而它也仅可作用于这个新类型的对象，这将变得很麻烦。我们可在第16章中看到，这个问题最终将通过模板解决。

我们可以对这个例子多做一些观察：在继承的过程中，它改变了**Stack**类的接口。如果接口是不同的，一个**StringStack**类将不同于**Stack**类，我们也不能把**StringStack**类当做**Stack**类
599 进行使用。这里充分表露了继承的不可靠性，如果我们不创建一个**Stack**类型的**StringStack**，那为什么要继承呢？更为恰当的**StringStack**版本将在本章后面给出。

14.6 非自动继承的函数

不是所有的函数都能自动地从基类继承到派生类中的。构造函数和析构函数用来处理对象的创建和析构操作，但它们只知道对它们的特定层次上的对象做些什么。所以，在该类以下各个层次中的所有的构造函数和析构函数都必须被调用，也就是说，构造函数和析构函数不能被继承，必须为每一个特定的派生类分别创建。

另外，**operator=** 也不能被继承，因为它完成类似于构造函数的活动。这就是说，尽管我们知道如何由等号右边的对象初始化左边的对象的所有成员，但这并不意味着这个初始化在继承后仍然具有同样的意义。

在继承过程中，如果不亲自创建这些函数，编译器就会生成它们（至于构造函数，我们不能创建任何的构造函数，因为编译器创建默认的构造函数和拷贝构造函数），这在第6章中已经简要地讲过了。被生成的构造函数使用成员方式的初始化，而被生成的 **operator=** 使用成员方式的赋值。下面是由编译器创建的函数的例子。

```
//: C14:SynthesizedFunctions.cpp
// Functions that are synthesized by the compiler
#include <iostream>
using namespace std;

class GameBoard {
public:
  GameBoard() { cout << "GameBoard()\n"; }
  GameBoard(const GameBoard&) {
600   cout << "GameBoard(const GameBoard&)\n";
  }
  GameBoard& operator=(const GameBoard&) {
    cout << "GameBoard::operator=()\n";
    return *this;
  }
  ~GameBoard() { cout << "~GameBoard()\n"; }
};
```

```
class Game {
  GameBoard gb; // Composition
public:
  // Default GameBoard constructor called:
  Game() { cout << "Game()\n"; }
  // You must explicitly call the GameBoard
  // copy-constructor or the default constructor
  // is automatically called instead:
  Game(const Game& g) : gb(g.gb) {
    cout << "Game(const Game&)\n";
  }
  Game(int) { cout << "Game(int)\n"; }
  Game& operator=(const Game& g) {
    // You must explicitly call the GameBoard
    // assignment operator or no assignment at
    // all happens for gb!
    gb = g.gb;
    cout << "Game::operator=()\n";
    return *this;
  }
  class Other {}; // Nested class
  // Automatic type conversion:
  operator Other() const {
    cout << "Game::operator Other()\n";
    return Other();
  }
  ~Game() { cout << "~Game()\n"; }
};

class Chess : public Game {};

void f(Game::Other) {}

class Checkers : public Game {
public:
  // Default base-class constructor called:
  Checkers() { cout << "Checkers()\n"; }
  // You must explicitly call the base-class
  // copy constructor or the default constructor
  // will be automatically called instead:
  Checkers(const Checkers& c) : Game(c) {
    cout << "Checkers(const Checkers& c)\n";
  }
  Checkers& operator=(const Checkers& c) {
    // You must explicitly call the base-class
    // version of operator=() or no base-class
    // assignment will happen:
    Game::operator=(c);
    cout << "Checkers::operator=()\n";
    return *this;
  }
};

int main() {
  Chess d1;  // Default constructor
```

601

```
  Chess d2(d1); // Copy-constructor
//! Chess d3(1); // Error: no int constructor
  d1 = d2; // Operator= synthesized
  f(d1); // Type-conversion IS inherited
  Game::Other go;
//!  d1 = go; // Operator= not synthesized
           // for differing types
  Checkers c1, c2(c1);
  c1 = c2;
} ///:~
```

GameBoard 和 **Game**中的构造函数和 **operator=** 都自己作了声明,所以我们能知道编译器何时使用它们。另外，**operator Other( )**从 **Game**对象到被嵌入的类 **Other** 的对象完成自动类型变换。类 **Chess**简单地从 **Game** 继承，并没有创建函数（观察编译器如何反应)。函数 **f( )** 接收一个 **Other** 对象以测试这个自动类型变换函数。

在 **main( )** 中，调用了为派生类**Class**生成的默认构造函数和拷贝构造函数。调用这些构
602 造函数的**Game**版本作为构造函数调用继承的一部分，尽管这看上去像是继承，但新的构造函数实际上是创建的。正如所预料的，自动创建带参数的构造函数是不可能的，因为这样对于编译器来说需要靠直觉知道太多东西。

在 **Chess**中，使用成员函数赋值，**operator=** 也被作为一个新的函数生成，(因此,调用了基类版本),这是因为该函数在新类中没有被显式地写出。当然，析构函数也会被编译器自动地生成。

鉴于有关处理对象创建的重写函数的所有原则，我们也许会觉得奇怪，为什么自动类型变换运算也能被继承。但其实这不足为奇——如果在 **Game**中有足够的块建立一个 **Other** 对象，那么在从 **Game**中派生出的任何东西中，这些块仍在原地，类型变换当然也就仍然有效(尽管实际上我们可能想重定义它)。

生成的**operator=**仅仅作用于同种类型对象。如果想把一种类型赋于另一种类型，则这个**operator=**必须由自己写出。

如果仔细地观察**Game**，将会看到拷贝构造函数和赋值运算符显式地调用了成员对象的拷贝构造函数和赋值运算符。我们通常会想这么做的，因为如果不这样做的话，将会代替拷贝构造函数调用默认的成员对象构造函数，至于赋值运算符，则根本就不会对成员对象有赋值操作执行。

最后，观察一下**Checkers**，它显示地写了默认构造函数、拷贝构造函数和赋值运算符。在默认构造函数中，默认的基类构造函数被自动地调用，这正是我们所希望的。但是，有一点很重要，一旦决定写自己的拷贝构造函数和赋值运算符，编译器就会假定我们已知道所做的一切，并且不再像在生成的函数中那样自动地调用基类版本。而如果想调用基类版本，那我们就必须亲自显式地调用它们。**Checkers**的拷贝构造函数中，这个调用出现在构造函数的
603 初始化列表中:

```
Checkers(const Checkers& c) : Game(c) {
```

至于**Checkers**赋值运算符，基类的调用在函数体的第一行中:

```
Game::operator=(c);
```

无论何时我们继承了一个类，这些调用都将成为我们使用的规范形式的一部分。

14.6.1 继承和静态成员函数

静态（**static**）成员函数与非静态成员函数的共同点：

1) 它们均可被继承到派生类中。

2) 如果我们重新定义了一个静态成员，所有在基类中的其他重载函数会被隐藏。

3) 如果我们改变了基类中一个函数的特征，所有使用该函数名字的基类版本都将会被隐藏。

然而，静态（**static**）成员函数不可以是虚函数（**virtual**）（第15章将详细介绍这个主题）。

14.7 组合与继承的选择

无论组合还是继承都能把子对象放在新类型中。两者都使用构造函数的初始化表达式表去构造这些子对象。现在我们可能会奇怪，这两者之间到底有什么不同？该如何选择？

组合通常是在希望新类内部具有已存在类的功能时使用，而不是希望已存在类作为它的接口。这就是说，嵌入一个对象用以实现新类的功能，而新类的用户看到的是新定义的接口而不是来自老类的接口。为此，在新类的内部嵌入已存在类的 **private** 对象。 604

有时，又希望允许类用户直接访问新类的组成，这就让成员对象是 **public**。由于成员对象使用自己的访问控制，所以是安全的，而当用户了解了我们所做的组装工作时，会更容易理解接口。**Car**类是一个很好的例子：

```
//: C14:Car.cpp
// Public composition

class Engine {
public:
  void start() const {}
  void rev() const {}
  void stop() const {}
};

class Wheel {
public:
  void inflate(int psi) const {}
};

class Window {
public:
  void rollup() const {}
  void rolldown() const {}
};

class Door {
public:
  Window window;
  void open() const {}
  void close() const {}
};

class Car {
public:
  Engine engine;
  Wheel wheel[4];
```

```
  Door left, right; // 2-door
};

int main() {
  Car car;
  car.left.window.rollup();
  car.wheel[0].inflate(72);
} ///:~
```

605

因为**Car**的组合是分析这个问题的一部分（并不是基本设计的部分），所以让成员是**public**，有助于客户程序员理解如何使用这个类，而且能使类的实例具有更小的代码复杂性。

稍加思考就会看到，用“车辆”对象组合一个**Car**是毫无意义的——小汽车不能包含车辆，它本身就是一种车辆。这种*is-a*关系用继承表达，而 *has-a* 关系用组合表达。

14.7.1 子类型设置

现在假设想创建 **ifstream** 对象的一个类，它不仅打开一个文件，而且还保存文件名。这时可以使用组合并把 **ifstream** 及**string**都嵌入这个新类中：

```
//: C14:FName1.cpp
// An fstream with a file name
#include "../require.h"
#include <iostream>
#include <fstream>
#include <string>
using namespace std;

class FName1 {
  ifstream file;
  string fileName;
  bool named;
public:
  FName1() : named(false) {}
  FName1(const string& fname)
    : fileName(fname), file(fname.c_str()) {
    assure(file, fileName);
    named = true;
  }
  string name() const { return fileName; }
  void name(const string& newName) {
    if(named) return; // Don't overwrite
    fileName = newName;
    named = true;
  }
  operator ifstream&() { return file; }
};

int main() {
  FName1 file("FName1.cpp");
  cout << file.name() << endl;
  // Error: close() not a member:
//!  file.close();
} ///:~
```

606

然而这里存在一个这样的问题：我们也许想通过包含一个从**FName1**到**ifstream &**的自动类型转换运算，在任何使用 **ifstream** 的地方都使用 **FName1** 对象，但在 main 中，

```
file.close();
```

这一行将不编译，因为自动类型转换只发生在函数调用中，而不在成员选择期间。所以，这个方法不可行。

第二个方法是对 **FName1** 增加 **close( )**的定义：

```
void close() { file.close(); }
```

如果只有很少的函数要从**ifstream** 类中拿来，这是可行的。在这种情况下，我们只是用了这个类的一部分，并且组合是适用的。

但是，如果希望这个类中的每件东西都进来，应该做什么呢？这称为子类型化（*subtyping*），因为我们正由已存在的类创建一个新类，并且希望这个新类与已存在的类有着严格相同的接口（希望增加任何我们想要加入的其他成员函数），所以能在已经用过这个已存在类的任何地方使用这个新类，这就是必须使用继承的地方。可以看到，子类型设置很好地解决了先前例子中的问题。 607

```
//: C14:FName2.cpp
// Subtyping solves the problem
#include "../require.h"
#include <iostream>
#include <fstream>
#include <string>
using namespace std;

class FName2 : public ifstream {
  string fileName;
  bool named;
public:
  FName2() : named(false) {}
  FName2(const string& fname)
    : ifstream(fname.c_str()), fileName(fname) {
    assure(*this, fileName);
    named = true;
  }
  string name() const { return fileName; }
  void name(const string& newName) {
    if(named) return; // Don't overwrite
    fileName = newName;
    named = true;
  }
};

int main() {
  FName2 file("FName2.cpp");
  assure(file, "FName2.cpp");
  cout << "name: " << file.name() << endl;
  string s;
  getline(file, s); // These work too!
  file.seekg(-200, ios::end);
  file.close();
} ///:~
```

现在，能与**ifstream**对象一起工作的任何成员函数也能与**FName2**对象一起工作。我们也可以看见，需要使用**ifstream**对象的非成员函数，例如**getline( )**，同样可以使用**FName2**对象。这是因为，一个**FName2** 是**ifstream**的一个类型。它并不只是简单地包含了一个**ifstream**。这
608 一点非常重要，我们将在本章最后和在下一章中进行讨论。

14.7.2 私有继承

通过在基类表中去掉 **public** 或者通过显式地声明**private**，可以私有地继承基类（后者可能是更好的策略，因为可以让用户明白它的含义）。当私有继承时，我们是“照此实现”；也就是说，创建的新类具有基类的所有数据和功能，但这些功能是隐藏的，所以它只是部分的内部实现。该类的用户访问不到这些内部功能，并且一个对象不能被看做是这个基类的实例（正如在**FName2.Cpp**中的）。

我们可能奇怪，**private** 继承的目的是什么，因为在这个新类中使用组合创建一个**private**对象的选择似乎更合适。为了完整性，**private**继承被包含在该语言中。但是，如果不为了其他理由，则应当减少混淆，所以通常希望使用组合而不是**private**继承。然而，这里可能偶然有这种情况，即可能想产生像基类接口一样的接口部分，而不允许该对象的处理像一个基类对象。**private**继承提供了这个功能。

14.7.2.1 对私有继承成员公有化

当私有继承时，基类的所有**public**成员都变成了**private**。如果希望它们中的任何一个是可视的，只要用派生类的**public**部分声明它们的名字即可：

```
//: C14:PrivateInheritance.cpp
class Pet {
public:
  char eat() const { return 'a'; }
  int speak() const { return 2; }
  float sleep() const { return 3.0; }
  float sleep(int) const { return 4.0; }
};

class Goldfish : Pet { // Private inheritance
public:
  using Pet::eat; // Name publicizes member
609 using Pet::sleep; // Both members exposed
};

int main() {
  Goldfish bob;
  bob.eat();
  bob.sleep();
  bob.sleep(1);
//! bob.speak();// Error: private member function
} ///:~
```

这样，如果想要隐藏基类的部分功能，则**private**继承是有用的。

注意给出一个重载函数的名字将使基类中所有它的重载版本公有化。

在使用**private**继承取代组合之前，应当仔细考虑，当与运行时类型标识相连时，私有继承特别复杂（这是本书第2卷中一章的主题，可从*www.BruceEckel.com*处下载）。

14.8 protected

我们已经学习了继承，而关键字**protected**对于继承有特殊的意义。在理想情况下，**private**成员总是严格私有的，但在实际项目中，有时希望某些东西隐藏起来，但仍允许其派生类的成员访问。于是关键字**protected**派上了用场。它的意思是："就这个类的用户而言，它是**private**的，但它可被从这个类继承来的任何类使用"。

最好让数据成员是**private**，因为我们应该保留改变内部实现的权利。然后才能通过**protected**成员函数控制对该类的继承者的访问。

```
//: C14:Protected.cpp
// The protected keyword
#include <fstream>
using namespace std;

class Base {
  int i;
protected:
  int read() const { return i; }
  void set(int ii) { i = ii; }
public:
  Base(int ii = 0) : i(ii) {}
  int value(int m) const { return m*i; }
};

class Derived : public Base {
  int j;
public:
  Derived(int jj = 0) : j(jj) {}
  void change(int x) { set(x); }
};

int main() {
  Derived d;
  d.change(10);
} ///:~
```

610

在本书的后面以及第2卷中，可以看到需要**protected**的例子。

14.8.1 protected继承

当继承时，基类默认为**private**，这意味着所有**public**成员函数对于新类的用户是**private**的。通常我们都会按**public**进行继承，从而使得基类的接口也是派生类的接口。然而在继承期间，也可以使用**protected**关键字。

保护继承的派生类意味着对其他类来说是"照此实现"，但它是对于派生类和友元是"**is-a**"。它是不常用的，它的存在只是为了语言的完备性。

611

14.9 运算符的重载与继承

除了赋值运算符以外，其余的运算符可以自动地继承到派生类中。这个可以通过**C12:Byte.h**中的继承加以说明：

```
//: C14:OperatorInheritance.cpp
// Inheriting overloaded operators
#include "../C12/Byte.h"
#include <fstream>
using namespace std;
ofstream out("ByteTest.out");

class Byte2 : public Byte {
public:
  // Constructors don't inherit:
  Byte2(unsigned char bb = 0) : Byte(bb) {}
  // operator= does not inherit, but
  // is synthesized for memberwise assignment.
  // However, only the SameType = SameType
  // operator= is synthesized, so you have to
  // make the others explicitly:
  Byte2& operator=(const Byte& right) {
    Byte::operator=(right);
    return *this;
  }
  Byte2& operator=(int i) {
    Byte::operator=(i);
    return *this;
  }
};

// Similar test function as in C12:ByteTest.cpp:
void k(Byte2& b1, Byte2& b2) {
  b1 = b1 * b2 + b2 % b1;

  #define TRY2(OP) \
    out << "b1 = "; b1.print(out); \
    out << ", b2 = "; b2.print(out); \
    out << ";  b1 " #OP " b2 produces "; \
    (b1 OP b2).print(out); \
    out << endl;

612 b1 = 9; b2 = 47;
  TRY2(+) TRY2(-) TRY2(*) TRY2(/)
  TRY2(%) TRY2(^) TRY2(&) TRY2(|)
  TRY2(<<) TRY2(>>) TRY2(+=) TRY2(-=)
  TRY2(*=) TRY2(/=) TRY2(%=) TRY2(^=)
  TRY2(&=) TRY2(|=) TRY2(>>=) TRY2(<<=)
  TRY2(=) // Assignment operator

  // Conditionals:
  #define TRYC2(OP) \
    out << "b1 = "; b1.print(out); \
    out << ", b2 = "; b2.print(out); \
    out << ";  b1 " #OP " b2 produces "; \
    out << (b1 OP b2); \
    out << endl;

  b1 = 9; b2 = 47;
  TRYC2(<) TRYC2(>) TRYC2(==) TRYC2(!=) TRYC2(<=)
  TRYC2(>=) TRYC2(&&) TRYC2(||)
```

```
  // Chained assignment:
  Byte2 b3 = 92;
  b1 = b2 = b3;
}

int main() {
  out << "member functions:" << endl;
  Byte2 b1(47), b2(9);
  k(b1, b2);
} ///:~
```

除了使用**Byte2**代替了**Byte**以外，该测试代码同**C12:ByteTest.cpp**中代码是一样的。这种方法通过继承检测了所有运算符是否可以对**Byte2**进行操作。

当检测类**Byte2**时，我们将看到必须显式定义构造函数，同时仅仅生成了可以把**Byte2**赋值于**Byte2**类型的**operator=**，而任何我们需要的赋值运算符将由我们自己生成。

14.10 多重继承

既然我们已可以从一个类继承，那么我们也就应该能同时从多个类继承。实际上这是可 613
以做到的，但是否它像设计部分一样有意义仍是有争议的。不过有一点是肯定的：直到我们已经很好地学会程序设计并完全理解这个语言时，我们才能试着去用它。这时，我们大概会认识到，不管我们如何认为我们必须用多重继承，我们总是能通过单重继承完成。

开始时，多重继承看起来似乎很简单：在继承时，只需在基类列表中增加多个类，用逗号隔开。然而，多重继承引起很多含糊的可能性，这就是为什么要在第2卷中专门有一章讨论这个问题的原因。

14.11 渐增式开发

继承和组合的优点之一是它支持*渐增式开发*（*incremental development*），它允许在已存在的代码中引进新代码，而不会给原来的代码带来错误，即使产生了错误，这个错误也只与新代码有关。也就是说通过继承（或通过组合）已存在的功能类并在其基础上增加数据成员和成员函数（并重定义已存在的成员函数）时，已存在类的代码——可能某人仍在使用——并不会被改变，更不会产生错误。如果错误出现，我们就会知道它肯定是在新派生代码中。相对于修改已存在代码体的做法来说，这些新代码很短也很易读。

相当奇怪的是，这些类如何清楚地被隔离。为了重用这些代码，甚至不需要这些成员函数的源代码，只需要表示类的头文件和目标文件或带有已编译成员函数的库文件（对于继承和组合都是这样）。

认识到程序开发就像人的学习过程一样，是一个渐增过程，这是很重要的。我们能做尽可能多的分析，但当开始一个项目时，我们仍不可能知道所有的答案。如果开始把项目作为一个有机的、可进化的生物来“培养”，而不是完全一次性地构造它，像一个玻璃盒子式的摩 614
天大楼，那么我们就会获得更大的成功和更直接的反馈[⊖]。

虽然继承对于实验是有用的技术，但在事情稳定之后，我们需要用新眼光重新审视一下

⊖ 为了学习这方面的更多思想，请参见Kent Beck所著的《*Extreme Programming Explained*》(Addison-Wesley 2000)。

我们的类层次，把它看成一个可感知的结构[⊖]。记住，继承首先是表示一种关系，即“新类属于老类的类型（*a type of*）”。我们的程序不应当关心怎样摆布位，而应当关心如何创建和处理各类型的对象，以便用问题空间的术语表示模型。

14.12 向上类型转换

在这一章的前面，我们已经看到了由**ifstream**派生而来的类的对象如何有**ifstream**对象的所有特性和行为。在**FName2.cpp**中，任何**ifstream**成员函数应当能被**FName2**对象调用。

继承的最重要的方面不是它为新类提供了成员函数，而是它是基类与新类之间的关系，这种关系可被描述为：“新类属于原有类的类型”。

这个描述不仅仅是一种想像的解释继承的方法——它直接由编译器支持。举个例子来说，考虑称为**Instrument**的基类（它表示乐器）和派生类**Wind**（管乐器）。因为继承意味着在基类中的所有函数在派生类中也是可行的，可以发送给基类的消息也可以发送给这个派生类。所
615 以，如果**Instrument**类有**play( )**成员函数，那么**Wind** 也有。这意味着，可以确切地说，**Wind** 对象也就是**Instrument** 类型的一个对象。下面的例子表明编译器是如何支持这个概念的。

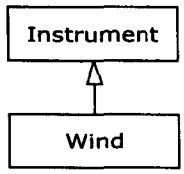
```
//: C14:Instrument.cpp
// Inheritance & upcasting
enum note { middleC, Csharp, Cflat }; // Etc.

class Instrument {
public:
  void play(note) const {}
};

// Wind objects are Instruments
// because they have the same interface:
class Wind : public Instrument {};

void tune(Instrument& i) {
  // ...
  i.play(middleC);
}

int main() {
  Wind flute;
  tune(flute); // Upcasting
} ///:~
```

在这个例子中，有趣的是**tune( )**函数，它接受一个**Instrument**类型的引用。然而，在**main( )**中，在**tune( )**函数的调用中却被传递了一个**Wind**参数 。我们可能会感到奇怪，C++对于类型检查应该是非常严格的，然而接受一个类型的函数为什么会这么容易地接受另一个类型。直到人们认识到**Wind**对象也是一个**Instrument**对象，这里**tune( )**函数对于**Instrument**的所有调用对于**Wind**也都是可调用的（这是由继承所保证的）。在**tune( )**中，这些代码对**Instrument**和从**Instrument**派生来的任何类型都有效，这种将**Wind**的引用或指针转变成
616 **Instrument**引用或指针的活动被称为向上类型转换（*upcasting*）。

⊖ 参见 Martin Fowler所著的《*Refactoring: Improving the Design of Existing Code*》(Addison-Wesley, 1999)。

14.12.1 为什么要“向上类型转换”

这个术语的引入是有其历史原因的，而且它也与类继承图的传统画法有关：在顶部是根，向下生长（当然我们可以用任何我们认为方便的方法画我们的图）。对于**Instrument.cpp**的继承图是



从派生类到基类的类型转换，在继承图上是上升的，所以它一般称为向上类型转换。向上类型转换总是安全的。因为是从更专门的类型到更一般的类型——对于这个类接口可能出现的惟一的事情是它失去成员函数，而不是获得它们。这就是编译器允许向上类型转换而不需要显式地说明或做其他标记的原因。

14.12.2 向上类型转换和拷贝构造函数

如果允许编译器为派生类生成拷贝构造函数，它将首先自动地调用基类的拷贝构造函数，然后再是各成员对象的拷贝构造函数（或者在内部类型上执行位拷贝），因此可以得到正确的操作：

```
//: C14:CopyConstructor.cpp
// Correctly creating the copy-constructor
#include <iostream>
using namespace std;

class Parent {
  int i;
public:
  Parent(int ii) : i(ii) {
    cout << "Parent(int ii)\n";
  }
  Parent(const Parent& b) : i(b.i) {
    cout << "Parent(const Parent&)\n";
  }
  Parent() : i(0) { cout << "Parent()\n"; }
  friend ostream&
    operator<<(ostream& os, const Parent& b) {
    return os << "Parent: " << b.i << endl;
  }
};

class Member {
  int i;
public:
  Member(int ii) : i(ii) {
    cout << "Member(int ii)\n";
  }
  Member(const Member& m) : i(m.i) {
    cout << "Member(const Member&)\n";
```

617

```
  }
  friend ostream&
    operator<<(ostream& os, const Member& m) {
    return os << "Member: " << m.i << endl;
  }
};

class Child : public Parent {
  int i;
  Member m;
public:
  Child(int ii) : Parent(ii), i(ii), m(ii) {
    cout << "Child(int ii)\n";
  }
  friend ostream&
    operator<<(ostream& os, const Child& c){
    return os << (Parent&)c << c.m
              << "Child: " << c.i << endl;
  }
};

int main() {
  Child c(2);
  cout << "calling copy-constructor: " << endl;
  Child c2 = c; // Calls copy-constructor
  cout << "values in c2:\n" << c2;
} ///:~
```

618

从对其中**Parent**部分调用**operator<<**的方式可以看出，**Child** 中的**operator<<**很有意思，它通过将**Child**对象类型转换为**Parent&**（但如果是类型转换了一个基类对象，而不是一个引用的话，将得不到所需要的结果）：

```
return os << (Parent&)c << c.m
```

这时编译器把它当做一个**Parent**类型，将调用**operator<<**的**Parent**版本。

我们可以看到**Child**没有显式定义的拷贝构造函数。编译器将通过调用**Parent** 和**Member**的拷贝构造函数来生成它的拷贝构造函数（如果我们没有创建任何构造函数，它是生成的四个函数之一，其他还有默认构造函数、**operator=**和析构函数）。这可从下面的输出中显示出来：

```
Parent(int ii)
Member(int ii)
Child(int ii)
calling copy-constructor:
Parent(const Parent&)
Member(const Member&)
values in c2:
Parent: 2
Member: 2
Child: 2
```

然而，如果试着为**Child**写自己的拷贝构造函数，并且出现错误：

```
Child(const Child& c) : i(c.i), m(c.m) {}
```

这时将会为**Child**中的基类部分调用默认的构造函数，这是在没有其他的构造函数可供选择调用的情况下，编译器回溯搜索的结果。（记住某些构造函数总是必须为每个对象所调用，

而不管它是否是一个其他类的子对象)。这样输出将会是:

```
Parent(int ii)
Member(int ii)
Child(int ii)
calling copy-constructor:
Parent()
Member(const Member&)
values in c2:
Parent: 0
Member: 2
Child: 2
```

619

这可能并不是我们所希望的，因为通常我们会希望基类部分从已存在对象拷贝至一个新的对象，以作为拷贝构造函数的一部分。

为了解决这个问题，必须记住无论何时我们在创建了自己的拷贝构造函数时，都要正确地调用基类拷贝构造函数（正如编译器所作的)。这猛一看可能有点奇怪，但它是向上类型转换的另一种情况:

```
Child(const Child& c)
  : Parent(c), i(c.i), m(c.m) {
  cout << "Child(Child&)\n";
}
```

奇怪的部分在于调用**Parent**的拷贝构造函数的地方: **Parent(c)**。传送一个**Child**对象给**Parent**构造函数意味着什么？因为**Child**是由**Parent**继承而来，所以**Child**的引用也就相当于**Parent**的引用。基类拷贝构造函数的调用将一个**Child**的引用向上类型转换为一个**Parent**的引用，并且使用它来执行拷贝构造函数。当我们创建自己的拷贝构造函数时，也总会做同样的事情。

14.12.3 组合与继承（再论）

确定应当用组合还是用继承，最清楚的方法之一是询问是否需要从新类向上类型转换。在本章的前面，**Stack**类通过继承被专门化，然而，**StringStack**对象仅作为**string**容器，不需向上类型转换，所以更合适的方法可能是组合:

620

```
//: C14:InheritStack2.cpp
// Composition vs. inheritance
#include "../C09/Stack4.h"
#include "../require.h"
#include <iostream>
#include <fstream>
#include <string>
using namespace std;

class StringStack {
  Stack stack; // Embed instead of inherit
public:
  void push(string* str) {
    stack.push(str);
  }
  string* peek() const {
    return (string*)stack.peek();
```

```
  }
  string* pop() {
    return (string*)stack.pop();
  }
};

int main() {
  ifstream in("InheritStack2.cpp");
  assure(in, "InheritStack2.cpp");
  string line;
  StringStack textlines;
  while(getline(in, line))
    textlines.push(new string(line));
  string* s;
  while((s = textlines.pop()) != 0) // No cast!
    cout << *s << endl;
} ///:~
```

这个文件与**InheritStack.cpp**是一样的，只不过**Stack**对象被嵌入在**StringStack**内，并且成员函数是由被嵌入对象调用的。这里没有时间和空间的开销，因为其子类占用相同量的空间，而且所有另外的类型检查是发生在编译时。

虽然这可能会变得更加复杂，但我们可以用**private**继承以表示“照此实现”，这也将很好
621 地解决了这个问题。然而，一个重要的方面是确保多重继承。在这种情况下，如果发现一个程序中可以使用组合来代替继承，我们便可以消除对多重继承的需要。

14.12.4 指针和引用的向上类型转换

在**Instrument.cpp**中，向上类型转换发生在函数调用期间——在函数外的**Wind**对象被引用并且变成一个在这个函数内的**Instrument**的引用。向上类型转换还能出现在对指针或引用简单赋值期间：

```
Wind w;
Instrument* ip = &w; // Upcast
Instrument& ir = w; // Upcast
```

和函数调用一样，这两个例子都不要求显式地类型转换。

14.12.5 危机

当然，任何向上类型转换都会损失对象的类型信息，如果如下编写：

```
Wind w;
Instrument* ip = &w;
```

则编译器只能把**ip**作为一个**Instrument**指针处理。这就是，它不能知道**ip**实际上可能是指向**Wind**的对象。所以，当调用**play( )**成员函数时，使用

```
ip->play(middleC);
```

编译器只能知道它正在对于一个**Instrument**指针调用**play( )**，并调用**Instrument::play( )**的基本版本，而不是它应该做的调用**wind::play( )**。这样将会得到不正确的结果。

这是一个重要的问题，将在第15章通过介绍面向对象编程的第三块基石：多态性（在
622 C++中用**virtual**函数实现）来解决。

14.13 小结

继承和组合都允许由已存在的类型创建新类型，两者都是在新类型中嵌入已存在的类型的子对象。然而，如果想重用已存在类型作为新类型的内部实现的话，我们最好用组合；如果想使新的类型和基类的类型相同（类型一样可确保接口一样），则应使用继承。如果派生类有基类的接口，它就能向上类型转换到这个基类，这一点对第15章中介绍的多态性很重要。

虽然通过组合和继承进行代码重用对于快速项目开发有帮助，但通常我们会希望在允许其他程序员依据它开发之前重新设计类层次。我们的类层次必须有这样的特性：它的每个类有专门的用途，它不能太大（包含太多不利于重用的功能），也不能太小（太小如不对它本身增加功能就不能使用）。

14.14 练习

部分练习题的答案可以在本书的电子文档 *"Annotated Solution Guide for Thinking in C++"* 中找到，只需支付很少的费用就可以在http: //www.BruceEckel.com得到这个电子文档。

14-1 修改**Car.cpp**，使它也继承**Vehicle**类，在**Vehicle**中放置合适的成员函数（也就是说，补充一些成员函数）。对**Vehicle**增加一个非默认的构造函数，在**Car**的构造函数内部必须调用它。

14-2 创建两个类，**A**和**B**，带有默认的构造函数。从**A**继承出一个新类，称为**C**，并且在**C**中创建**B**的一个成员对象，而不对**C**创建构造函数。创建类**C**的一个对象，观察结果。

14-3 创建一个三层的类结构，带有默认的构造函数和析构函数，它们都对**cout**做了声明。对于最底层的派生类对象，验证所有三个构造函数和析构函数都自动被调用。解释这里调用的顺序。 623

14-4 修改**Combined.cpp**，再多继承一层并且增加一个新的成员对象。添加代码来显示何时调用构造函数和析构函数。

14-5 在**Combined.cpp**中，创建从类**B**继承来的类**D**，它含有一个类**C**的成员对象。添加代码来显示何时调用构造函数和析构函数。

14-6 修改**Order.cpp**，再继承出一层，即**Derived3**，它含有类**Member4**和类**Member5**的成员对象。跟踪程序的输出。

14-7 在**NameHiding.cpp**中验证，**Derived2**、**Derived3** 和 **Derived4** 中**f( )**的基类版本都是不可用的。

14-8 修改**NameHiding.cpp**，在**Base**中增加三个名为**h( )**的重载函数。然后显示在派生类中重新定义其中的一个函数，则会隐藏其余的函数。

14-9 从**vector<void*>**中继承出类**StringVector**，重新定义**push_back( )** 和 **operator[ ]**成员函数以接收和生成**string***。如果试着**push_back( )**一个**void***，会发生什么情况？

14-10 创建一个包含**long**型成员的类，对构造函数使用psuedo构造函数调用语法来初始化这个**long**成员。

14-11 创建类**Asteroid**，使用继承，具体化在第13章（**PStash. h & PStash. cpp**）中的**PStash**类，使得它接受和返回**Asteroid**指针。修改**PStashtest.cpp**并测试该类。改

变这个类使得**PStash**是一个成员对象。

14-12 使用**vector**来代替**PStash**，重复练习11。

14-13 在**SynthesizedFunctions.cpp**中，修改**Chess**，使它有一个默认构造函数、拷贝构造函数和赋值运算符。显示我们进行了正确的修改。

14-14 创建两个不含有默认构造函数的类**Traveler** 和 **Pager**，但具有一个参数为**string**的
624 构造函数，该构造函数只是简单地把**string**参数拷贝至一个内部变量中。对于每个类，创建正确的拷贝构造函数和赋值运算符。现在从**Traveler**中继承出类**BusinessTraveler**，并使其包含一个类**Pager**的对象。创建正确的默认构造函数、参数为**string**的构造函数、拷贝构造函数和赋值运算符。

14-15 创建含有两个**static**成员函数的类。继承这个类，并且重新定义其中一个成员函数，显示出另一个函数在派生类中被隐藏。

14-16 找出**ifstream**更多的成员函数。在**FName2.cpp**中，尝试将它们输出在**file**对象上。

14-17 使用**private**和**protected**继承方式从基类中创建两个新类。然后试着把派生类的对象向上类型转换为基类对象。解释所发生的事情。

14-18 在**Protected.cpp**中，在**Derived**里增加一个成员函数，它用来调用**Base**类中的**protected**成员函数**read( )**。

14-19 修改**Protected.cpp**使得**Derived**是按**protected**方式继承来的。看看一个**Derived**对象是否可以调用**value( )**。

14-20 创建一个含有**fly( )**方法的类**SpaceShip**。从**SpaceShip** 中继承出**Shuttle**，并且增加一个方法**land( )**。创建一个新的类**Shuttle**，通过一个**SpaceShip**对象的指针或引用向上类型转换，并且试着调用**land( )**方法。解释操作的结果。

14-21 修改**Instrument.cpp**，对**Instrument**增加一个**prepare( )**方法。在**tune( )**中调用**prepare( )**。

14-22 修改**Instrument.cpp**，使**play( )**向**cout**打印出消息，并且**Wind**重新定义了**play( )**，使之向**cout**打印出不同的消息。运行这个程序并解释为什么我们不希望有这样的结果。然后在**Instrument**中**play( )**的声明前加上**virtual**关键字（我们将在第15章中学习），观察结果有什么不同。

14-23 在**CopyConstructor.cpp**中，由**Child**继承出一个新类，使其具有一个**Member m**。并且创建适当的构造函数、拷贝构造函数、**operator=**和用于ostreams的
625 **operator<<**。在**main( )**中测试这个类。

14-24 修改例子**CopyConstructor.cpp**，对**Child**使用我们自己的拷贝构造函数，它不调用基类拷贝构造函数，看看有什么情况发生。在**Child**的拷贝构造函数中的初始化列表里，通过进行适当的显式地对基类拷贝构造函数的调用来分析和解决这个问题。

14-25 使用**vector<string>**代替**Stack**，来对**InheritStack2.cpp**进行修改。

14-26 创建一个类**Rock**，含有默认构造函数、拷贝构造函数、赋值运算符和析构函数，它们都向**cout**声明它们已经被调用。在**main( )**中，创建一个**vector<Rock>**（即通过传值方式得到**Rock**对象）并且添加一些**Rock**对象。运行这个程序并解释我们得到的输出。注意**vector**里所有**Rock**的析构函数是否都被调用了。然后用**vector<Rock*>**来重复上面的操作。可以创建**vector<Rock&>**吗？

14-27 本练习创建一个名为代理（*proxy*）的设计模式。首先建立一个基类**Subject**，使其包含三个函数**f( )**、**g( )**和 **h( )**。现在从中继承出类**Proxy**和另外两个类**Implementation1**和**Implementation2**。**Proxy**中应包含指向**Subject**的指针，于是它的所有成员函数可以通过该**Subject**指针指回，调用**Subject**的函数。**Proxy**的构造函数的参数是指向**Subject**的指针，它被包含在**Proxy**内（通常这由构造函数完成）。在**main( )**中，使用两种不同的实现方式创建两个不同的**Proxy**对象。然后修改**Proxy**使得我们可以动态地改变实现方式。

14-28 修改第13章的**ArrayOperatorNew.cpp**以显示，如果我们继承**Widget**，则仍将可以正确地执行分配。解释为什么不能正确地执行第13章中**Framis.cpp**的继承。

14-29 修改第13章的**Framis.cpp**，对**Framis**进行继承，并且为我们的派生类创建新版本的**new** 和 **delete**。表明可以正确地运行它们。 626

第15章 多态性和虚函数

多态性（在C++中通过虚函数来实现）是面向对象程序设计语言中数据抽象和继
627 承之外的第三个基本特征。

多态性（polymorphism）提供了接口与具体实现之间的另一层隔离，从而将“*what*”与“*how*”分离开来。多态性改善了代码的组织性和可读性，同时也使创建的程序具有可扩展性，程序不仅在项目的最初创建期可以“扩展”，而且当在项目需要有新的功能时也能“扩展”。

封装（encapsulation）通过组合特性和行为来生成新的数据类型。访问控制通过使细节数据设为**private**，将接口从具体实现中分离开来。这类机制对于具有过程化程序设计背景的人来说是非常有意义的。而虚函数则根据类型来处理解耦。在第14章中，我们已经看到，如何继承通过把对象作为它自己的类型或它的基类型来处理。这种能力非常关键，因为它允许很多类型（从同一个基类型派生）被看做是一个类型，一段代码可以同样地工作在所有这些不同类型上。虚函数允许一个类型表达自己与另一个相似类型之间的区别，只要这两个类型都是从同一个基类型派生的。这种区别是通过从基类调用的那些函数行为的不同来表达的。

在本章中，我们将从基本知识开始学习虚函数，为了简单起见，本章所用的例子经过简化，只保留了程序的“虚”性质。

15.1 C++程序员的演变

C程序员可以用三步演变为C++程序员。第一步：简单地把C++作为一个“更好的C”，因为C++要求在使用任何函数之前必须声明它，并且对于如何使用变量有更苛刻的要求。简单地用C++编译器编译C程序常常会发现错误。

第二步：进入“基于对象”的C++。这意味着，很容易看到将数据结构和在它上面活动
628 的函数捆绑在一起的代码组织好处，还可以看到构造函数和析构函数的价值，也许还会看到一些简单的继承。大多数用过C工作的程序员很快就看到这个步骤是有用的，因为无论何时，当他们创建库时，这个步骤都是他们努力要去做的。然而在C++中，将由编译器来帮助我们完成这个步骤。

在基于对象的层面上，我们容易产生错觉，因为我们可以很快成功，并且无需花费太多精力就能得到很多好处。感觉就好像我们正在创建数据类型——制造类和对象，向这些对象发送消息，一切恰到好处并且干净利落。

但是，不要犯傻。如果我们停留在这里，我们就会错失这个语言最重要的部分。这个最重要的部分才是通向真正的面向对象程序设计的飞跃。要做到这一点，只有靠虚函数。

虚函数增强了类型概念，而不是只在结构内部隐蔽地封装代码，所以毫无疑问，对于新的C++程序员来说，这些概念是最困难的。然而，它们也是理解面向对象程序设计的转折点。如果不用虚函数，就等于还不懂得面向对象程序设计（OOP）。

因为虚函数是与类概念紧密联系的，而类是面向对象程序设计的核心，所以在传统的过程型语言中没有类似于虚函数的东西。作为一个过程型程序员，没有什么事物可以帮助他思考虚函数，因为他接触的是这个语言的其他特征。过程型语言中的特征可以在算法层上来理解，而虚函数只能从设计的观点来理解。

15.2 向上类型转换

在第14章中，我们已经看到对象如何能作为它自己的类或作为它的基类的对象来使用。另外，还能通过基类的地址操作它。取一个对象的地址（指针或引用），并将其作为基类的地址来处理，这被称为向上类型转换（*upcasting*），因为继承树的绘制方式是以基类为顶点的。 629

我们还看到出现一个问题，它体现在如下的代码段中：

```
//: C15:Instrument2.cpp
// Inheritance & upcasting
#include <iostream>
using namespace std;
enum note { middleC, Csharp, Eflat }; // Etc.

class Instrument {
public:
  void play(note) const {
    cout << "Instrument::play" << endl;
  }
};

// Wind objects are Instruments
// because they have the same interface:
class Wind : public Instrument {
public:
  // Redefine interface function:
  void play(note) const {
    cout << "Wind::play" << endl;
  }
};

void tune(Instrument& i) {
  // ...
  i.play(middleC);
}

int main() {
  Wind flute;
  tune(flute); // Upcasting
} ///:~
```

函数**tune( )**（通过引用）接受一个**Instrument**，但也不拒绝任何从**Instrument**派生的类。在**main( )**中，可以看到，无需类型转换，就能将**Wind**对象传给**tune( )**。这是可接受的；在**Instrument** 中的接口必然存在于**Wind**中，因为**Wind**是从**Instrument**中按公有方式继承而来的。**Wind**到**Instrument**的向上类型转换会使**Wind**的接口“变窄”，但不会窄过**Instrument**的整个接口。 630

处理指针时采用相同的参数；惟一的不同是用户必须显式地取对象的地址传给函数。

15.3 问题

运行程序**Instrument2.cpp**可以看到这个程序中的问题。调用输出的是**Instrument::play**。显然，这不是所希望的输出，因为我们知道这个对象实际上是**Wind**而不是一个**Instrument**。应当调用的是**Wind::play**。为此，由**Instrument**派生的任何对象不论它处于什么位置都应当使用它的**play( )**版本。

然而，当对函数用C方法时，**Instrument2.cpp**的行为并不使人惊奇。为了理解这个问题，需要知道捆绑（*binding*）的概念。

15.3.1 函数调用捆绑

把函数体与函数调用相联系称为捆绑（*binding*）。当捆绑在程序运行之前（由编译器和连接器）完成时，这称为早捆绑（*early binding*）。我们可能没有听过这个术语，因为在过程型语言中不会有这样的选择：C编译只有一种函数调用方式，就是早捆绑。

上面程序中的问题是早捆绑引起的，因为编译器在只有**Instrument**地址时它并不知道要调用的正确函数。

解决方法被称为晚捆绑（*late binding*），这意味着捆绑根据对象的类型，发生在运行时。晚捆绑又称为动态捆绑（*dynamic binding*）或运行时捆绑(*runtime binding*)。当一个语言实现
631 晚捆绑时，必须有某种机制来确定运行时对象的类型并调用合适的成员函数。对于一种编译语言，编译器并不知道实际的对象类型，但它插入能找到和调用正确函数体的代码。晚捆绑机制因语言而异，但可以想像，某些种类的类型信息必须装在对象自身中。稍后将会看到它是如何工作的。

15.4 虚函数

对于特定的函数，为了引起晚捆绑，C++要求在基类中声明这个函数时使用**virtual**关键字。晚捆绑只对**virtual**函数起作用，而且只在使用含有**virtual**函数的基类的地址时发生，尽管它们也可以在更早的基类中定义。

为了创建一个像**virtual**这样的成员函数，可以简单地在这个函数声明的前面加上关键字**virtual**。仅仅在声明的时候需要使用关键字**virtual**，定义时并不需要。如果一个函数在基类中被声明为**virtual**，那么在所有的派生类中它都是**virtual**的。在派生类中**virtual**函数的重定义通常称为重写(*overriding*)。

注意，仅需要在基类中声明一个函数为**virtual**。调用所有匹配基类声明行为的派生类函数都将使用虚机制。虽然可以在派生类声明前使用关健字**virtual**（这也是无害的），但这样会使程序段显得冗余和混乱。

为了从**Instrument2.cpp**中得到所希望的结果，只需简单地在基类中的**play( )**之前增加**virtual**关键字：

```
//: C15:Instrument3.cpp
// Late binding with the virtual keyword
#include <iostream>
using namespace std;
enum note { middleC, Csharp, Cflat }; // Etc.
```

```
class Instrument {                                          632
public:
  virtual void play(note) const {
    cout << "Instrument::play" << endl;
  }
};

// Wind objects are Instruments
// because they have the same interface:
class Wind : public Instrument {
public:
  // Override interface function:
  void play(note) const {
    cout << "Wind::play" << endl;
  }
};

void tune(Instrument& i) {
  // ...
  i.play(middleC);
}

int main() {
  Wind flute;
  tune(flute); // Upcasting
} ///:~
```

这个文件除了增加了**virtual**关键字之外，一切与**Instrument2.cpp**相同，但结果明显不一样。现在输出调用的是**Wind::play**。

15.4.1 扩展性

通过将**play( )**在基类中定义为**virtual**，不用改变**tune( )**函数就可以在系统中随意增加新函数。在一个设计风格良好的 OOP程序中，大多数甚至所有的函数都沿用**tune( )**模型，只与基类接口通信。这样的程序是可扩展的（*extensible*），因为可以通过从公共基类继承新数据类型而增加新功能。操作基类接口的函数完全不需要改变就可以适合于这些新类。 633

这里有一个instrument例子，它有更多的虚函数和一些新类，它们都能与老的版本一起正确工作，而不用改变**tune( )**函数：

```
//: C15:Instrument4.cpp
// Extensibility in OOP
#include <iostream>
using namespace std;
enum note { middleC, Csharp, Cflat }; // Etc.

class Instrument {
public:
  virtual void play(note) const {
    cout << "Instrument::play" << endl;
  }
  virtual char* what() const {
    return "Instrument";
  }
```

```
  // Assume this will modify the object:
  virtual void adjust(int) {}
};

class Wind : public Instrument {
public:
  void play(note) const {
    cout << "Wind::play" << endl;
  }
  char* what() const { return "Wind"; }
  void adjust(int) {}
};

class Percussion : public Instrument {
public:
  void play(note) const {
    cout << "Percussion::play" << endl;
  }
  char* what() const { return "Percussion"; }
  void adjust(int) {}
};

class Stringed : public Instrument {
public:
  void play(note) const {
634 cout << "Stringed::play" << endl;
  }
  char* what() const { return "Stringed"; }
  void adjust(int) {}
};

class Brass : public Wind {
public:
  void play(note) const {
    cout << "Brass::play" << endl;
  }
  char* what() const { return "Brass"; }
};

class Woodwind : public Wind {
public:
  void play(note) const {
    cout << "Woodwind::play" << endl;
  }
  char* what() const { return "Woodwind"; }
};

// Identical function from before:
void tune(Instrument& i) {
  // ...
  i.play(middleC);
}

// New function:
void f(Instrument& i) { i.adjust(1); }
```

```
// Upcasting during array initialization:
Instrument* A[] = {
  new Wind,
  new Percussion,
  new Stringed,
  new Brass,
};

int main() {
  Wind flute;
  Percussion drum;
  Stringed violin;
  Brass flugelhorn;
  Woodwind recorder;
  tune(flute);
  tune(drum);
  tune(violin);
  tune(flugelhorn);
  tune(recorder);
  f(flugelhorn);
} ///:~
```

635

可以看到，这个例子已在**Wind**之下增加了另外的继承层，但不管这里有多少层，**virtual**机制仍会正确工作。针对**Brass**和**Woodwind**，**adjust( )**函数没有重写（重新定义）。当出现这种情况时，将会自动地调用继承层次中“最近”的定义——编译器保证对于虚函数总是有某种定义，所以决不会出现最终调用不与函数体捆绑的情况（这种情况将导致灾难）。

数组**A[ ]**存放指向基类**Instrument**的指针，所以在数组初始化过程中发生向上类型转换。这个数组和函数**f( )**将在稍后的讨论中用到。

在对**tune( )**的调用中，向上类型转换在对象的每一个不同的类型上完成。总能得到期望的结果。这可以被描述为“发送一条消息给一个对象，让这个对象考虑用它来做什么”。**virtual**函数使我们在分析项目时可以初步确定：基类应当出现在哪里？应当如何扩展这个程序？在程序最初创建时，即便我们没有发现合适的基类接口和虚函数，但在稍后或者更晚，当决定扩展或维护这个程序时，也常常会发现它们。这不是分析或设计错误，它只意味着一开始我们还没有所有的信息。由于C++中严格的模块化，因此这并不是大问题。因为当我们对系统的一部分进行修改时，往往不会像C那样波及系统的其他部分。

15.5 C++如何实现晚捆绑

晚捆绑如何发生？所有的工作都由编译器在幕后完成。当告诉编译器要晚捆绑时（通过创建虚函数来告诉），编译器安装必要的晚捆绑机制。因为程序员常常从理解C++虚函数机制中受益，所以这一节将详细阐述编译器实现这一机制的方法。

636

关键字**virtual**告诉编译器它不应当执行早捆绑，相反，它应当自动安装对于实现晚捆绑必需的所有机制。这意味着，如果对**Brass**对象通过基类**Instrument**地址调用**play( )**，将得到恰当的函数。

为了达到这个目的，典型的编译器[⊖]对每个包含虚函数的类创建一个表（称为VTABLE）。

⊖ 编译器可以按它们希望的任何方式执行虚操作，但是这里所讨论的方法是一种相当通用的方法。

在VTABLE中，编译器放置特定类的虚函数的地址。在每个带有虚函数的类中，编译器秘密地放置一个指针，称为*vpointer*（缩写为VPTR），指向这个对象的VTABLE。当通过基类指针做虚函数调用时（也就是做多态调用时），编译器静态地插入能取得这个VPTR并在VTABLE表中查找函数地址的代码，这样就能调用正确的函数并引起晚捆绑的发生。

为每个类设置VTABLE、初始化VPTR、为虚函数调用插入代码，所有这些都是自动发生的，所以不必担心。利用虚函数，即使在编译器还不知道这个对象的特定类型的情况下，也能调用这个对象中正确的函数。

下面几节将进行更详细的阐述。

15.5.1 存放类型信息

637 可以看到，在任何类中不存在显式的类型信息。而先前的例子和简单的逻辑告诉我们，必须有一些类型信息放在对象中；否则，类型将不能在运行时被建立。确实是这样的,但类型信息被隐藏了。为了看到这些信息，这里举一个例子，以便检查使用虚函数的类的长度，并与没有虚函数的类进行比较。

```
//: C15:Sizes.cpp
// Object sizes with/without virtual functions
#include <iostream>
using namespace std;

class NoVirtual {
  int a;
public:
  void x() const {}
  int i() const { return 1; }
};

class OneVirtual {
  int a;
public:
  virtual void x() const {}
  int i() const { return 1; }
};

class TwoVirtuals {
  int a;
public:
  virtual void x() const {}
  virtual int i() const { return 1; }
};

int main() {
  cout << "int: " << sizeof(int) << endl;
  cout << "NoVirtual: "
       << sizeof(NoVirtual) << endl;
  cout << "void* : " << sizeof(void*) << endl;
  cout << "OneVirtual: "
       << sizeof(OneVirtual) << endl;
  cout << "TwoVirtuals: "
       << sizeof(TwoVirtuals) << endl;
} ///:~
```

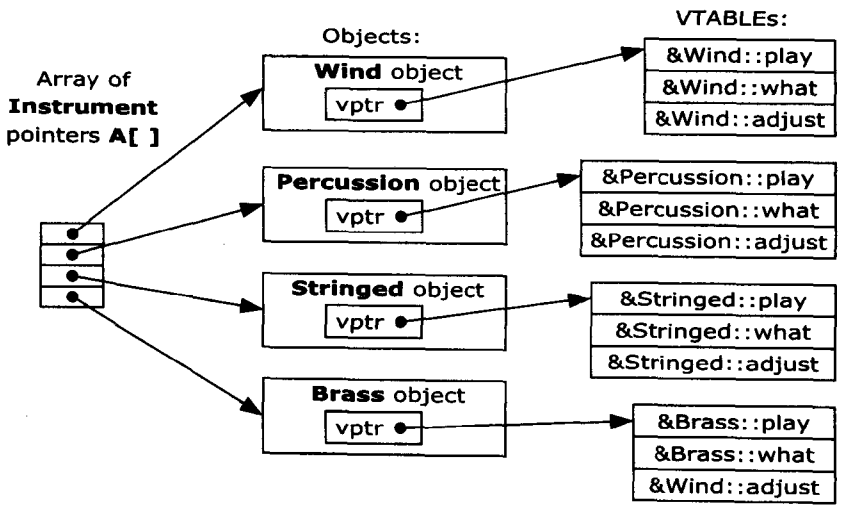
638

不带虚函数，对象的长度恰好就是所期望的长度：单个[⊖]**int**的长度。而带有单个虚函数的**OneVirtual**，对象的长度是**NoVirtual**的长度加上一个**void**指针的长度。它反映出，如果有一个或多个虚函数，编译器都只在这个结构中插入一个单个指针（VPTR）。因此**OneVirtual**和**TwoVirtuals**的长度没有区别。这是因为VPTR指向一个存放函数地址的表。我们只需要一个表，因为所有虚函数地址都包含在这个单个表中。

这个例子至少要求一个数据成员。如果没有数据成员，C++编译器会强制这个对象是非零长度，因为每个对象必须有一个互相区别的地址。如果我们想像在一个零长度对象的数组中索引寻址，就能理解这一点。把一个“哑”成员插入到对象中，否则这个对象就会是零长度。当类型信息由于存在这个关键字**virtual**而被插入时，这个“哑”成员的位置就被占用。在上例中，用注释符号将**int a**这一行去掉，就会看到这种情况。

15.5.2 虚函数功能图示

下面是**Instrument4.cpp**中的指针数组**A[ ]**的图，它可以帮助我们准确地理解当使用虚函数时编译器进行的内部活动。 639



这个**Instrument**指针数组没有特殊类型信息，它的每一个元素都指向一个类型为**Instrument**的对象。**Wind**、**Percussion**、**Stringed**和**Brass**都可以归入这个类别之中，因为它们都是从**Instrument**派生来的（并因而与**Instrument**有相同的接口和可以响应相同的消息），所以它们的地址自然被放进这个数组。然而，编译器并不知道它们是比**Instrument**对象具有更多内容的东西，所以，就将它们留给其自己的设备处理，而通常调用所有函数的基类版本。但在这里，所有这些函数都被用**virtual**声明，所以出现了不同的情况。

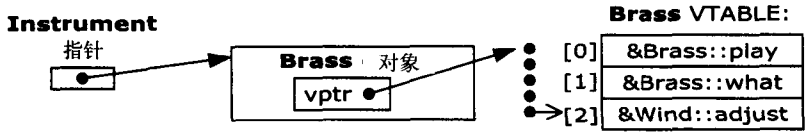
每当创建一个包含有虚函数的类或从包含有虚函数的类派生一个类时，编译器就为这个类创建一个惟一的VTABLE，如这个图的右面所示。在这个表中，编译器放置了在这个类中或在它的基类中所有已声明为virtual的函数的地址。如果在这个派生类中没有对在基类中声明为virtual的函数进行重新定义，编译器就使用基类的这个虚函数地址。（在**Brass**的VTABLE中，

⊖ 这里某些编译器可能含有长度功能，但是并不多见。

adjust的入口就是这种情况。）然后编译器在这个类中放置VPTR（可在Sizes.ccp中发现）。当
640 使用简单继承时，对于每个对象只有一个VPTR。VPTR必须被初始化为指向相应的VTABLE的起始地址。（这在构造函数中发生，在稍后会看得更清楚。）

一旦VPTR被初始化为指向相应的VTABLE，对象就“知道”它自己是什么类型。但只有当虚函数被调用时这种自我认知才有用。

当通过基类地址调用一个虚函数时（此时编译器没有能完成早捆绑所需的所有信息），要特殊处理。它不是实现典型的函数调用，那样只是简单地用汇编语言CALL特定的地址，而是编译器为完成这个函数调用而产生不同的代码。下面看到的是通过Instrument指针对于Brass调用adjust()（Instrument引用产生同样的结果）。



编译器从这个Instrument指针开始，这个指针指向这个对象的起始地址。对于所有的Instrument对象或由Instrument派生的对象，它们的VPTR都在对象的相同位置（常常在对象的开头），所以编译器能够取出这个对象的VPTR。VPTR指向VTABLE的起始地址。所有的VTABLE都具有相同的顺序，不管何种类型的对象。play()是第一个，what()是第二个，adjust()是第三个。所以无论什么特殊的对象类型，编译器都知道adjust()函数必在VPTR+2处。这样，不是“以Instrument::adjust地址调用这个函数”（这是早捆绑，是错误动作），而是产生代码，即实际上“在VPTR+2处调用这个函数”。因为获取VPTR和确定实际函数地址发生在运行时，所以这样就得到了所希望的晚捆绑。我们向这个对象发送消息，随后这个对
641 象能断定它应当做什么。

15.5.3 撩开面纱

如果能看到由虚函数调用而产生的汇编语言代码，这将是很有帮助的，这样我们可以看到晚捆绑实际上是如何发生的。下面是在函数f(Instrument&i)内部调用

```
i.adjust(1);
```

某个编译器所产生的输出：

```
push  1
push  si
mov   bx, word ptr [si]
call  word ptr [bx+4]
add   sp, 4
```

C++函数调用的参数与C函数调用一样，是从右向左进栈的（这个顺序是为了支持C的变量参数表），所以参数1首先压栈。对于这个函数，寄存器si（Intel x86处理器的一部分）存放i的地址。因为它是被选中的对象的首地址，它也被压进栈。记住，这个首地址对应于this的值，正因为调用每个成员函数时this都必须作为参数压进栈，所以成员函数知道它工作在哪个特殊对象上。这样，我们总能看到，在成员函数调用之前压栈的次数等于参数个数加1（除了static成员函数，它没有this）。

现在，必须实现实际的虚函数调用。首先，必须产生VPTR，使得能找到VTABLE。对于

这个编译器，VPTR在对象的开头，所以**this**的内容对应于VPTR。下面这一行

```
mov bx, word ptr [si]
```

取出**si**（即**this**）所指的字，它就是VPTR。将这个VPTR放入寄存器**bx**中。

放在**bx**中的这个VPTR指向这个VTABLE的首地址，但被调用的函数不是在VTABLE中第 642
0个位置，而是在第2个位置（因为它是这个表中的第3个函数）。对于这种内存模式，每个函数指针是两个字节长，所以编译器对VPTR加4，计算相应的函数地址所在的地方。注意，这是编译时建立的常值，所以惟一要做的事情就是保证在第2个位置上的指针恰好指向**adjust( )**。幸好编译器仔细处理，并保证在VTABLE中的所有函数指针都以相同的次序出现，而不论我们在派生类中是以什么次序重载它们。

一旦在VTABLE中相应函数指针的地址被计算出来，就调用这个函数。所以取出这个地址并马上在这个句子中调用：

```
call word ptr [bx+4]
```

最后，栈指针移回去，以清除在调用之前压入栈的参数。在C和C++汇编代码中，将经常看到调用者清除这些参数，但这可能依据处理器和编译器的实现而有所不同。

15.5.4 安装vpointer

因为VPTR决定了对象的虚函数的行为，所以我们可以看到VPTR总是指向相应的VTABLE是多么重要。在VPTR适当初始化之前绝对不能调用虚函数。当然，可以保证初始化的地点是在构造函数中，但是在**Instrument**例子中没有一个是有构造函数的。

这样，默认构造函数的创建是很关键的。在**Instrument**例子中，编译器创建了一个默认构造函数，它只做初始化VPTR的工作。在使用任何**Instrument**对象之前，对于**Instrument**对象自动调用这个构造函数。所以，可以安全地调用虚函数。 643

在下一节中我们将讨论在构造函数内部自动初始化VPTR的含义。

15.5.5 对象是不同的

认识到向上类型转换仅处理地址，这是重要的。如果编译器有一个它知道确切类型的对象，那么（在C++中）对任何函数的调用将不再使用晚捆绑，或至少编译器不必使用晚捆绑。因为编译器知道对象的确切类型，为了提高效率，当调用这些对象的虚函数时，很多编译器使用早捆绑。下面是一个例子：

```
//: C15:Early.cpp
// Early binding & virtual functions
#include <iostream>
#include <string>
using namespace std;

class Pet {
public:
  virtual string speak() const { return ""; }
};

class Dog : public Pet {
public:
```

```
  string speak() const { return "Bark!"; }
};

int main() {
  Dog ralph;
  Pet* p1 = &ralph;
  Pet& p2 = ralph;
  Pet p3;
  // Late binding for both:
  cout << "p1->speak() = " << p1->speak() <<endl;
  cout << "p2.speak() = " << p2.speak() << endl;
  // Early binding (probably):
  cout << "p3.speak() = " << p3.speak() << endl;
} ///:~
```

644 在**p1->speak( )**和**p2.speak( )**中，使用地址，就意味着信息不完全：**p1**和**p2**可能表示**Pet**的地址，也可能表示其派生对象的地址，所以必须使用虚函数。而当调用**p3.speak( )**时不存在含糊，编译器知道确切的类型且知道它是一个对象，所以它不可能是由**Pet**派生的对象，而确切的只是一个**Pet**。这样，可以使用早捆绑。但是，如果不希望编译器的工作如此复杂，仍然可以使用晚捆绑，并且会产生相同的行为。

15.6 为什么需要虚函数

在这个问题上，我们可能会问："如果这个技术如此重要，并且使得任何时候都能调用'正确'的函数，那么为什么它是可选的呢？为什么我甚至还需要知道它呢？"

问得好。回答关系到C++的基本哲学："因为它不是相当高效的"。从前面的汇编语言输出可以看出，它并不是对于绝对地址的一个简单的CALL，而是为设置虚函数调用需要两条以上的复杂的汇编指令。这既需要代码空间，又需要执行时间。

一些面向对象的语言已经接受了这种途径，即晚捆绑对于面向对象程序设计是性质所固有的，所以应当总是出现，它不应当是可选的，而且用户并不一定需要知道它。这是在创造语言的设计时决定的，而这种特殊的方法对于许多语言是合适的。[⊖]而C++来自于C，在C中，效率是重要的。创造C完全是为了代替汇编语言以实现操作系统（从而改写操作系统——UNIX——使得比它的先驱更轻便）。发明C++的主要原因之一是让C程序员的工作具有更高效
645 率。[⊖]当C程序员遇到C++时要问的第一个问题是"我将得到什么样的规模和速度效果"？如果回答是"除了函数调用时需要有一点额外的开销外，一切皆好"，那么许多人就会仍使用C，而不会改变到C++。另外，内联函数是不可能的，因为虚函数必须有地址放在VTABLE中。所以虚函数是可选的，而且该语言的默认是非虚拟的，这是最快的配置。Stroustrup声明他的方针是，"如果我们不用它，我们就不会为它花费额外的开销。"

因此，**virtual**关键字可以改变程序的效率。然而，当设计类时，我们不应当为效率问题担心。如果想使用多态，就在每处使用虚函数。当试图加速代码时，只需寻找可以不使用虚函数的函数（而且通常可能在其他方面获得更大收益——好的编程者会在查找瓶颈方面，而不是在猜测方面投入更多的工作）。

⊖ 例如，Smalltalk、Java及Python语言都成功地使用了这种方法。

⊖ 在C++的发源地——贝尔实验室中，汇集着大量的C程序员。尽力使这些C程序员的工作更加有效率，即使是改善一点点，也会为公司节省数百万美元的开销。

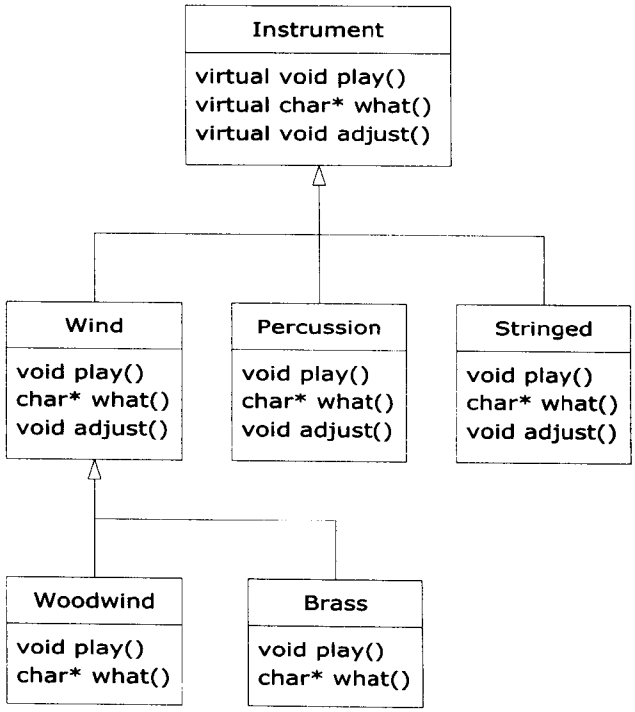
有些证据表明，C++中的规模和速度改进效果是在C的规模和速度的10%之内，并且常常更接近。能够得到更小的规模和更高速度的原因是C++可以有比用C更快的方法设计程序，而且设计的程序更小。

15.7 抽象基类和纯虚函数

在设计时，常常希望基类仅仅作为其派生类的一个接口。这就是说，仅想对基类进行向上类型转换，使用它的接口，而不希望用户实际地创建一个基类的对象。要做到这点，可以在基类中加入至少一个*纯虚函数*（*pure virtual function*），来使基类成为*抽象*（*abstract*）类。纯虚函数使用关键字**virtual**，并且在其后面加上= **0**。如果某人试着生成一个抽象类的对象，编译器会制止他。这个工具允许生成特定的设计。 646

当继承一个抽象类时，必须实现所有的纯虚函数，否则继承出的类也将是一个抽象类。创建一个纯虚函数允许在接口中放置成员函数，而不一定要提供一段可能对这个函数毫无意义的代码。同时，纯虚函数要求继承出的类对它提供一个定义。

在所有的instrument的例子中，基类**Instrument**中的函数总是“哑”函数。如果调用这些函数，就会出错。这是因为，**Instrument**的目的是对所有从它派生出来的类创建公共接口。



建立公共接口的惟一原因是它能对于每个不同的子类有不同的表示。它建立一个基本的格式，用来确定什么是对于所有派生类是公共的——除此之外，别无用途。所以，把 647
Instrument设计为抽象类就比较合适。当仅希望通过一个公共接口来操纵一组类，且这个公共接口不需要实现（或者不需要完全实现）时，可以创建一个抽象类。

如果有一个作用类似于抽象类的类（就像**Instrument**），则这个类的对象几乎总是没有意义的。也就是说，**Instrument**的含义只表示接口，不表示特例实现，所以创建一个

Instrument对象没有意义。我们也许想防止用户这样做，这可以通过让**Instrument**的所有虚函数打印出错信息而完成，但这种方法到运行时才能获得出错信息，并且要求用户进行可靠而详尽的测试。所以最好是在编译时就能发现这个问题。

下面是用于纯虚函数声明的语法：

```
virtual void f() = 0;
```

这样做，等于告诉编译器在VTABLE中为函数保留一个位置，但在这个特定位置中不放地址。只要有一个函数在类中被声明为纯虚函数，则VTABLE就是不完全的。

如果一个类的VTABLE是不完全的，当某人试图创建这个类的对象时，编译器做什么呢？它不能安全地创建一个纯抽象类的对象，所以如果试图创建一个纯抽象类的对象，编译器就发出一个出错信息。这样，编译器就保证了抽象类的纯洁性，它就不会被误用了。

下面是修改后的**Instrument4.cpp**，它使用了纯虚函数。因为这个类中全是纯虚函数，所以我们称之为*纯抽象类*（*pure abstract class*）：

```
//: C15:Instrument5.cpp
// Pure abstract base classes
#include <iostream>
using namespace std;
enum note { middleC, Csharp, Cflat }; // Etc.

class Instrument {
public:
  // Pure virtual functions:
  virtual void play(note) const = 0;
  virtual char* what() const = 0;
  // Assume this will modify the object:
  virtual void adjust(int) = 0;
};
// Rest of the file is the same ...

class Wind : public Instrument {
public:
  void play(note) const {
    cout << "Wind::play" << endl;
  }
  char* what() const { return "Wind"; }
  void adjust(int) {}
};

class Percussion : public Instrument {
public:
  void play(note) const {
    cout << "Percussion::play" << endl;
  }
  char* what() const { return "Percussion"; }
  void adjust(int) {}
};

class Stringed : public Instrument {
public:
  void play(note) const {
    cout << "Stringed::play" << endl;
```

648

```
  }
  char* what() const { return "Stringed"; }
  void adjust(int) {}
};

class Brass : public Wind {
public:
  void play(note) const {
    cout << "Brass::play" << endl;
  }
  char* what() const { return "Brass"; }
};

class Woodwind : public Wind {
public:
  void play(note) const {
    cout << "Woodwind::play" << endl;
  }
  char* what() const { return "Woodwind"; }
};

// Identical function from before:
void tune(Instrument& i) {
  // ...
  i.play(middleC);
}

// New function:
void f(Instrument& i) { i.adjust(1); }

int main() {
  Wind flute;
  Percussion drum;
  Stringed violin;
  Brass flugelhorn;
  Woodwind recorder;
  tune(flute);
  tune(drum);
  tune(violin);
  tune(flugelhorn);
  tune(recorder);
  f(flugelhorn);
} ///:~
```

649

纯虚函数是非常有用的，因为它们使得类有明显的抽象性，并告诉用户和编译器打算如何使用。

注意，纯虚函数禁止对抽象类的函数以传值方式调用。这也是防止对象切片（*object slicing*）（这将会被简单地介绍）的一种方法。通过抽象类，可以保证在向上类型转换期间总是使用指针或引用。

650

纯虚函数防止产生完全的VTABLE，但这并不意味着我们不希望对其他一些函数产生函数体。我们常常希望调用一个函数的基类版本，即便它是虚拟的。把公共代码放在尽可能靠近我们的类层次根的地方，这是很好的想法。这不仅节省了代码空间，而且使得改变的传播更加容易。

15.7.1 纯虚定义

在基类中，对纯虚函数提供定义是可能的。我们仍然告诉编译器不允许产生抽象基类的对象，而且如果要创建对象，则纯虚函数必须在派生类中定义。然而，我们可能希望一段公共代码，使一些或所有派生类定义都能调用，而不必在每个函数中重复这段代码。

正如下面的纯虚定义：

```
//: C15:PureVirtualDefinitions.cpp
// Pure virtual base definitions
#include <iostream>
using namespace std;

class Pet {
public:
  virtual void speak() const = 0;
  virtual void eat() const = 0;
  // Inline pure virtual definitions illegal:
  //!  virtual void sleep() const = 0 {}
};

// OK, not defined inline
void Pet::eat() const {
  cout << "Pet::eat()" << endl;
}

void Pet::speak() const {
  cout << "Pet::speak()" << endl;
}
class Dog : public Pet {
public:
  // Use the common Pet code:
  void speak() const { Pet::speak(); }
  void eat() const { Pet::eat(); }
};

int main() {
  Dog simba;  // Richard's dog
  simba.speak();
  simba.eat();
} ///:~
```

651

Pet的VTABLE表仍然空着，但在这个派生类中刚好有一个函数，可以通过名字调用它。

这个特点的另一个好处是，它允许我们实现从常规虚函数到纯虚函数的改变，而无需打乱已存在的代码。（这是一个处理不用重新定义虚函数的类的方法。）

15.8 继承和VTABLE

可以想像，当实现继承和重新定义一些虚函数时，会发生什么事情？编译器对新类创建一个新VTABLE表，并且插入新函数的地址，对于没有重新定义的虚函数使用基类函数的地址。无论如何，对于可被创建的每个对象(即它的类不含有纯虚函数)，在VTABLE中总有一个函数地址的全集，所以绝对不能对不在其中的地址进行调用（否则结果将会是灾难性的）。

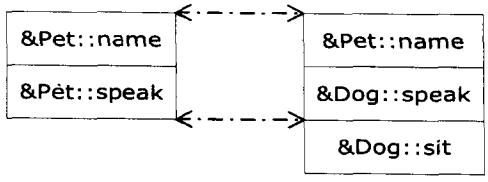
但是在派生（*derived*）类中继承或增加新的虚函数时会发生什么呢？下面有一个简单的例子：

```
//: C15:AddingVirtuals.cpp
// Adding virtuals in derivation
#include <iostream>
#include <string>
using namespace std;
class Pet {
  string pname;
public:
  Pet(const string& petName) : pname(petName) {}
  virtual string name() const { return pname; }
  virtual string speak() const { return ""; }
};

class Dog : public Pet {
  string name;
public:
  Dog(const string& petName) : Pet(petName) {}
  // New virtual function in the Dog class:
  virtual string sit() const {
    return Pet::name() + " sits";
  }
  string speak() const { // Override
    return Pet::name() + " says 'Bark!'";
  }
};

int main() {
  Pet* p[] = {new Pet("generic"),new Dog("bob")};
  cout << "p[0]->speak() = "
       << p[0]->speak() << endl;
  cout << "p[1]->speak() = "
       << p[1]->speak() << endl;
//! cout << "p[1]->sit() = "
//!      << p[1]->sit() << endl; // Illegal
} ///:~
```

652

类**Pet**中含有2个虚函数: **speak( )** 和**name( )**，而在类**Dog**中又增加了第3个称为**sit( )**的虚函数，并且重新定义了**speak( )**的含义。下图有助于显示发生的事情。这是由编译器为**Pet**和**Dog**创建的VTABLE。

653



注意，编译器在**Dog**的VTABLE中把**speak( )**的地址准确地映射到和**Pet**的VTABLE 中同样的位置。类似地，如果类**Pug**从**Dog**中继承而来，则在它的VTABLE 中**sit( )**也将会被放置在和**Dog**的VTABLE中相同的位置。这是因为（正如通过汇编语言例子看到的）编译器产生的代码在VTABLE中使用一个简单的偏移来选择虚函数。不论对象属于哪个特殊的类，它的VTABLE都是以同样的方法设置，所以对虚函数的调用将总是使用同样的方法。

然而在这里，编译器只对指向基类对象的指针工作。而这个基类只有**speak( )** 和 **name( )**

函数，所以它就是编译器惟一允许调用的函数。那么，如果只有基类对象的指针，那么编译器怎么可能知道自己正在对**Dog**对象工作呢？这个指针可能指向其他一些没有**sit( )**函数的类。在VTABLE中，可能有，也可能没有一些其他函数的地址，但无论何种情况，对这个VTABLE地址做虚函数调用都不是我们想要做的。所以编译器通过防止我们对只存在于派生类中的函数做虚函数调用来完成其工作。

有一些比较少见的情况，可能我们知道指针实际上指向哪一种特殊子类的对象。这时如果想调用只存在于这个子类中的函数，则必须类型转换这个指针。下面的语句可以消除由前面程序产生的出错信息：

```
((Dog*)p[1])->sit()
```

654 这里，我们碰巧知道**p[1]**指向**Dog**对象，但通常情况下我们并不知道。如果你的问题是必须知道所有对象的确切类型，那么我们应当重新考虑这个问题，因为我们可能在进行不正确的虚函数调用。然而对于有些情况，如果知道保存在一般容器中的所有对象的确切类型，会使我们的设计工作在最佳状态（或者没有选择）。这就是*运行时类型辨认*（*Run-Time Type Identification*，RTTI）问题。

RTTI是有关向下类型转换基类指针到派生类指针的问题（“向上”和“向下”是相对典型类图而言的，典型类图以基类为顶点）。向上类型转换是自动发生的，不需强制，因为它是绝对安全的。向下类型转换是不安全的，因为这里没有关于实际类型的编译时信息，所以必须准确地知道这个类实际上是什么类型。如果把它转换成错误的类型，就会出现麻烦。

在本章的后面将讨论RTTI，而且本书的第2卷中也有一章专门讨论这个主题。

15.8.1 对象切片

当多态地处理对象时，传地址与传值有明显的不同。所有在这里已经看到的例子和将会看到的例子都是传地址的，而不是传值的。这是因为地址都有相同的长度[⊖]，传递派生类（它通常稍大一些）对象的地址和传递基类（它通常更小一点）对象的地址是相同的。如前面所述，这是使用多态的目的，即让对基类对象操作的代码也能透明地操作派生类对象。

如果对一个对象进行向上类型转换，而不使用地址或引用，发生的事情将会使我们吃惊：这个对象被“切片”，直到剩下来的是适合于目的的子对象。在下面例子中可以看到当一个对
655 象被“切片”后发生了什么。

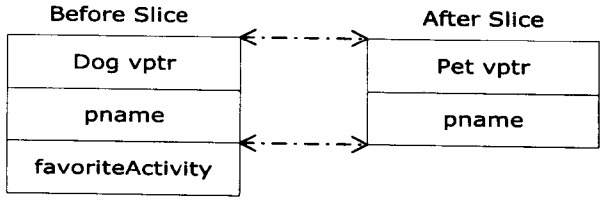
```
//: C15:ObjectSlicing.cpp
#include <iostream>
#include <string>
using namespace std;

class Pet {
  string pname;
public:
  Pet(const string& name) : pname(name) {}
  virtual string name() const { return pname; }
  virtual string description() const {
    return "This is " + pname;
  }
```

⊖ 实际上，并不是所有机器上的指针都具有同样的长度。然而，在我们的讨论范围内，认为它们是相同的。

```
};

class Dog : public Pet {
  string favoriteActivity;
public:
  Dog(const string& name, const string& activity)
    : Pet(name), favoriteActivity(activity) {}
  string description() const {
    return Pet::name() + " likes to " +
      favoriteActivity;
  }
};

void describe(Pet p) { // Slices the object
  cout << p.description() << endl;
}

int main() {
  Pet p("Alfred");
  Dog d("Fluffy", "sleep");
  describe(p);
  describe(d);
} ///:~
```

函数**describe( )**通过传值方式传递一个类型为**Pet**的对象。然后对于这个**Pet**对象调用虚函数**description( )**。我们可能希望第一次调用产生“This is Alfred”，而第二次调用产生“Fluffy likes to sleep”。实际上，两次调用都是调用了基类版本的**description( )**。 656

在这个程序中，发生了两件事情。第一，**describe( )**接受的是一个**Pet**对象（而不是指针或引用），所以**describe( )**中的任何调用都将引起一个与**Pet**大小相同的对象压栈并在调用后清除。这意味着，如果一个由**Pet**派生来的类的对象被传给**describe( )**，则编译器会接受它，但只拷贝这个对象的对应于**Pet**的部分，切除这个对象的派生部分，如下图所示：



现在，我们可能对这个虚函数调用有这样的疑问：如果**Dog::description( )**使用了**Pet**（它仍存在）和**Dog**（它不再存在，因为已被切掉），当调用它时，会发生什么呢？

其实我们已经从灾难中被解救出来，这个对象正安全地按值传递。这是因为派生类对象已经被强迫地变为基类对象，所以编译器知道这个对象的确切类型。另外，当按值传递时，**Pet**对象的拷贝构造函数被调用，该构造函数初始化VPTR指向**Pet**的VTABLE，并且只拷贝这个对象的**Pet**部分。这里没有显式的拷贝构造函数，所以编译器自动地生成一个。由于所有上述原因，因此这个对象在切片过程中真的变成了一个**Pet**对象。

对象切片实际上是当它拷贝到一个新的对象时，去掉原来对象的一部分，而不是像使用指针或引用那样简单地改变地址的内容。因此，不常使用对象向上类型转换，事实上，通常是要提防或防止这种操作。注意，在本例中，如果**description( )**在基类中是一个纯虚函数

657 （这并不是毫无理由的，因为它在基类中实际上也并没有做什么事情），因为编译器不会允许我们“创建”基类对象（这就是我们通过传值向上类型转换所发生的事情），所以它将阻止对对象进行“切片”。这可能会是纯虚函数最重要的作用：如果某人试着这么做，将通过生成一个编译错误来阻止对象切片。

15.9 重载和重新定义

在第14章中，我们看到重新定义一个基类中的重载函数将会隐藏所有该函数的其他基类版本。而当对虚函数进行这些操作时，情况会有点不同。考虑下面这个例子，它对第14章中的例子**NameHiding.cpp**进行了修改：

```
//: C15:NameHiding2.cpp
// Virtual functions restrict overloading
#include <iostream>
#include <string>
using namespace std;

class Base {
public:
  virtual int f() const {
    cout << "Base::f()\n";
    return 1;
  }
  virtual void f(string) const {}
  virtual void g() const {}
};

class Derived1 : public Base {
public:
  void g() const {}
};

class Derived2 : public Base {
public:
  // Overriding a virtual function:
  int f() const {
    cout << "Derived2::f()\n";
    return 2;
658 }
};

class Derived3 : public Base {
public:
  // Cannot change return type:
  //! void f() const{ cout << "Derived3::f()\n";}
};

class Derived4 : public Base {
public:
  // Change argument list:
  int f(int) const {
    cout << "Derived4::f()\n";
    return 4;
  }
```

```
};

int main() {
  string s("hello");
  Derived1 d1;
  int x = d1.f();
  d1.f(s);
  Derived2 d2;
  x = d2.f();
//!  d2.f(s); // string version hidden
  Derived4 d4;
  x = d4.f(1);
//!  x = d4.f(); // f() version hidden
//!  d4.f(s); // string version hidden
  Base& br = d4; // Upcast
//!  br.f(1); // Derived version unavailable
  br.f(); // Base version available
  br.f(s); // Base version abailable
} ///:~
```

首先注意到，在**Derived3**中，编译器不允许我们改变重新定义过的函数的返回值（如果**f( )**不是虚函数，则是允许的）。这是一个非常重要的限制，因为编译器必须保证我们能够多态地通过基类调用函数，并且如果基类希望**f( )**返回一个**int**值，则**f( )**的派生类版本必须保持约定，否则将会出问题。

659

在第14章中的规则仍将有效：如果重新定义了基类中的一个重载成员函数，则在派生类中其他的重载函数将会被隐藏。这可由**main( )**中测试**Derived4**的代码显示出来，即使**f( )**的新版本实际上并没有重新定义一个已存在的虚函数的接口，**f( )**的两个基类版本会被**f(int)**隐藏。然而，如果把**d4**向上类型转换到**Base**，则只有基类版本是可行的（因为基类约定允许），而派生类版本是不可行的（因为在基类中没有特定的方法）。

15.9.1 变量返回类型

上例的类**Derived3**显示了我们不能在重新定义过程中修改虚函数的返回类型。通常是这样的，但也有特例，我们可以稍稍修改返回类型。如果返回一个指向基类的指针或引用，则该函数的重新定义版本将会从基类返回的内容中返回一个指向派生类的指针或引用。例如：

```
//: C15:VariantReturn.cpp
// Returning a pointer or reference to a derived
// type during ovverriding
#include <iostream>
#include <string>
using namespace std;

class PetFood {
public:
  virtual string foodType() const = 0;
};

class Pet {
public:
  virtual string type() const = 0;
  virtual PetFood* eats() = 0;
```

```
};

class Bird : public Pet {
public:
660   string type() const { return "Bird"; }
  class BirdFood : public PetFood {
  public:
    string foodType() const {
      return "Bird food";
    }
  };
  // Upcast to base type:
  PetFood* eats() { return &bf; }
private:
  BirdFood bf;
};

class Cat : public Pet {
public:
  string type() const { return "Cat"; }
  class CatFood : public PetFood {
  public:
    string foodType() const { return "Birds"; }
  };
  // Return exact type instead:
  CatFood* eats() { return &cf; }
private:
  CatFood cf;
};

int main() {
  Bird b;
  Cat c;
  Pet* p[] = { &b, &c, };
  for(int i = 0; i < sizeof p / sizeof *p; i++)
    cout << p[i]->type() << " eats "
         << p[i]->eats()->foodType() << endl;
  // Can return the exact type:
  Cat::CatFood* cf = c.eats();
  Bird::BirdFood* bf;
  // Cannot return the exact type:
//!  bf = b.eats();
  // Must downcast:
  bf = dynamic_cast<Bird::BirdFood*>(b.eats());
} ///:~
```

661 成员函数**Pet::eats( )**返回一个指向**PetFood**的指针。在**Bird**中，完全按基类中的形式重载这个成员函数，并且包含了返回类型。也就是说，**Bird::eats( )**把**BirdFood**向上类型转换到**PetFood**。

但在**Cat**中，**eats( )**的返回类型是指向**CatFood**的指针，而**CatFood**是派生于**PetFood**的类。编译它的惟一原因是，返回类型是从基类函数的返回类型中继承而来的。这样，合约仍被遵守；**eats( )**还是返回了一个**PetFood**指针。

如果考虑多态性的话，这看上去就并不是必需的。为什么不把所有的返回类型向上类型

转换为**PetFood***，正如**Bird::eats( )**所做的那样呢？这是个好建议，但在**main( )**的结束部分，我们看到了不同之处：**Cat::eats( )**可以返回**PetFood**的确切类型，而**Bird::eats( )**的返回值必须被向下类型转换为确切的类型。

所以说，能返回确切的类型要更通用些，而且在自动地进行向上类型转换时不丢失特定的信息。然而，返回基类类型通常会解决我们的问题，所以这是一个特殊的功能。

15.10 虚函数和构造函数

当创建一个包含有虚函数的对象时，必须初始化它的VPTR以指向相应的VTABLE。这必须在对虚函数进行任何调用之前完成。正如我们可能猜到的，因为生成一个对象是构造函数的工作，所以设置VPTR也是构造函数的工作。编译器在构造函数的开头部分秘密地插入能初始化VPTR的代码。正如第14章所述，如果我们没有为一个类显式创建构造函数，则编译器会为我们生成构造函数。如果该类含有虚函数，则生成的构造函数将会包含相应的VPTR初始化代码。这有几个含义。

首先，这涉及效率。内联（**inline**）函数的作用是对小函数减少调用代价。如果C++不提供内联函数，则预处理器就可能被用来创建这些“宏”。然而，预处理器没有通道或类的概念，因此不能被用来创建成员函数宏。另外，有了由编译器插入的隐藏代码的构造函数，预处理宏根本不能工作。

当寻找效率漏洞时，我们必须明白，编译器正在插入隐藏代码到我们的构造函数中。这些隐藏代码不仅必须初始化VPTR，而且还必须检查**this**的值（以免**operator new**返回零）和 662
调用基类构造函数。放在一起，这些代码可以影响我们认为是一个小内联函数的调用。特别是，构造函数的规模会抵消函数调用代价的减少。如果做大量的内联构造函数调用，代码长度就会增长，而在速度上没有任何好处。

当然，也许并不会立即把所有这些小构造函数都变成非内联，因为它们更容易写为内联构造函数。但是，当我们正在调整我们的代码时，记住，务必去掉这些内联构造函数。

15.10.1 构造函数调用次序

构造函数和虚函数的第二个有趣的方面涉及构造函数的调用顺序和在构造函数中虚函数调用的方法。

所有基类构造函数总是在继承类构造函数中被调用。这是有意义的，因为构造函数有一项专门的工作：确保对象被正确地建立。派生类只访问它自己的成员，而不访问基类的成员。只有基类构造函数能正确地初始化它自己的成员。因此，确保所有的构造函数被调用是很关键的，否则整个对象不会适当地被构造。这就是为什么编译器强制为派生类的每个部分调用构造函数的原因。如果不在构造函数初始化表达式表中显式地调用基类构造函数，它就调用默认构造函数。如果没有默认构造函数，编译器将报告出错。

构造函数调用的顺序是重要的。当继承时，必须知道基类的全部成员并能访问基类的任 663
何**public** 和**protected**成员。这意味着，当在派生类中时，必须能肯定基类的所有成员都是有效的。在通常的成员函数中，构造已经发生，所以这个对象的所有部分的成员都已经建立。然而，在构造函数内，必须想办法保证所有成员都已经建立。保证它的惟一方法是让基类构造函数首先被调用。这样，当在派生类构造函数中时，在基类中能访问的所有成员都已经被

初始化。在构造函数中，“知道所有成员对象是有效的”也是下面做法的原因：只要可能，我们应当在这个构造函数初始化表达式表中初始化所有的成员对象（即对象通过组合被置于类中）。只要遵从这个做法，我们就能保证初始化所有基类成员和当前对象的成员对象。

15.10.2 虚函数在构造函数中的行为

构造函数调用层次会导致一个有趣的两难选择。试想：如果我们在构造函数中并且调用了虚函数，那么会发生什么现象呢？在普通的成员函数中，我们可以想像所发生的情况——虚函数的调用是在运行时决定的，这是因为编译时这个对象并不能知道它是属于这个成员函数所在的那个类，还是属于由它派生出来的某个类。于是，我们也许会认为在构造函数中也会发生同样的事情。

然而，情况并非如此。对于在构造函数中调用一个虚函数的情况，被调用的只是这个函数的本地版本。也就是说，虚机制在构造函数中不工作。

这种行为有两个理由。在概念上，构造函数的工作是生成一个对象。在任何构造函数中，可能只是部分形成对象——我们只能知道基类已被初始化，但并不能知道哪个类是从这个基
664 类继承来的。然而，虚函数在继承层次上是“向前”和“向外”进行调用。它可以调用在派生类中的函数。如果我们在构造函数中也这样做，那么我们所调用的函数可能操作还没有被初始化的成员，这将导致灾难的发生。

第二个理由是机械的。当一个构造函数被调用时，它做的首要的事情之一就是初始化它的VPTR。然而，它只能知道它属于“当前”类——即构造函数所在的类。于是它完全忽视这个对象是否是基于其他类的。当编译器为这个构造函数产生代码时，它是为这个类的构造函数产生代码——既不是为基类，也不是为它的派生类（因为类不知道谁继承它）。所以它使用的VPTR必须是对于这个类的VTABLE。而且，只要它是最后的构造函数调用，那么在这个对象的生命期内，VPTR将保持被初始化为指向这个VTABLE。但如果接着还有一个更晚派生的构造函数被调用，那么这个构造函数又将设置VPTR指向它的VTABLE，以此类推，直到最后的构造函数结束。VPTR的状态是由被最后调用的构造函数确定的。这就是为什么构造函数调用是按照从基类到最晚派生的类的顺序的另一个理由。

但是，当这一系列构造函数调用正发生时，每个构造函数都已经设置VPTR指向它自己的VTABLE。如果函数调用使用虚机制，它将只产生通过它自己的VTABLE的调用，而不是最后派生的VTABLE（所有构造函数被调用后才会有最后派生的VTABLE）。另外，许多编译器认识到，如果在构造函数中进行虚函数调用，应该使用早捆绑，因为它们知道晚捆绑将只对本地函数产生调用。无论哪种情况，在构造函数中调用虚函数都不能得到预期的结果。

15.11 析构函数和虚拟析构函数

构造函数是不能为虚函数的。但析构函数能够且常常必须是虚的。

构造函数有一项特殊工作，即一块一块地组合成一个对象。它首先调用基类构造函数，
665 然后调用在继承顺序中的更晚派生的构造函数（同样，它也必须按此方法调用成员对象构造函数）。类似地，析构函数也有一项特殊工作，即它必须拆卸属于某层次类的对象。为了做这些工作，编译器生成代码来调用所有的析构函数，但它必须按照与构造函数调用相反的顺序。这就是，析构函数自最晚派生的类开始，并向上到基类。这是安全且合理的：当前的析构函

数一直知道基类成员仍是有效的。如果需要在析构函数中调用某一基类的成员函数，进行这样的操作是安全的。因此，析构函数能够对其自身进行清除，然后它调用下一个析构函数，该析构函数又将执行它的清除工作，以此类推。每个析构函数知道它所在类从哪一个类派生而来，但不知道从它派生出哪些类。

应当记住，构造函数和析构函数是类层次进行调用的惟一地方（因此，编译器自动地生成适当的类层次）。在所有其他函数中，只有这个函数会被调用（非基类版本），而无论它是虚的还是非虚的。同一个函数的基类版本在普通函数中被调用（无论它是虚的还是非虚的）的惟一方法是显式地调用这个函数。

通常，析构函数的执行是相当充分的。但是，如果想通过指向某个对象基类的指针操纵这个对象（也就是，通过它的一般接口操纵这个对象），会发生什么现象呢？这在面向对象的程序设计中确实很重要。当我们想**delete**在栈中已经用**new**创建的对象的指针时，就会出现这个问题。如果这个指针是指向基类的，在**delete**期间，编译器只能知道调用这个析构函数的基类版本。这听起来很耳熟，虚函数被创建恰恰是为了解决同样的问题。幸运的是，就像除了构造函数以外的所有其他函数一样，析构函数可以是虚函数。

```
//: C15:VirtualDestructors.cpp
// Behavior of virtual vs. non-virtual destructor
#include <iostream>
using namespace std;
class Base1 {
public:
  ~Base1() { cout << "~Base1()\n"; }
};

class Derived1 : public Base1 {
public:
  ~Derived1() { cout << "~Derived1()\n"; }
};

class Base2 {
public:
  virtual ~Base2() { cout << "~Base2()\n"; }
};

class Derived2 : public Base2 {
public:
  ~Derived2() { cout << "~Derived2()\n"; }
};

int main() {
  Base1* bp = new Derived1; // Upcast
  delete bp;
  Base2* b2p = new Derived2; // Upcast
  delete b2p;
} ///:~
```

666

当运行这个程序时，将会看到**delete bp**只调用基类的析构函数。而当**delete b2p**调用时，在基类的析构函数执行后，派生类析构函数将会执行，这正是我们所希望的。不把析构函数设为虚函数是一个隐匿的错误，因为它常常不会对程序有直接的影响。但要注意它不知不觉地引

入存储器泄漏（关闭程序时内存未释放）。同样，这样的析构操作还有可能掩盖发生的问题。

即使析构函数像构造函数一样，是“例外”函数，但析构函数可以是虚的，这是因为这个对象已经知道它是什么类型（而在构造期间则不然）。一旦对象已被构造，它的VPTR就已
667 被初始化，所以能发生虚函数调用。

15.11.1 纯虚析构函数

尽管纯虚析构函数在标准C++中是合法的，但在使用时有一个额外的限制：必须为纯虚析构函数提供一个函数体。这看起来有点违反常规；如果它需要一个函数体，那它又如何称之为“纯”？但如果我们记得构造函数和析构函数是具有特别意义的操作，特别是如果我们记得在一个类层次中总是会调用所有的析构函数，就会有所体会。如果我们不对一个纯虚析构函数进行定义，在析构期间将会调用什么函数体呢？因此，编译器和链接程序强迫纯虚析构函数一定要有一个函数体，这是十分必要的。

如果它是纯虚的，而且不得不有一个函数体，那么它的价值是什么呢？我们可以看到纯虚析构函数和非纯虚析构函数之间惟一的不同之处在于纯虚析构函数使得基类是抽象类，所以不能创建一个基类的对象（虽然如果基类的任何其他函数是纯虚函数，也是具有同样的效果）。

然而，当从某个含有虚析构函数的类中继承出一个类，情况变得有点复杂。不像其他的纯虚函数，我们不要求在派生类中提供纯虚函数的定义。下面的编译和链接便是证明。

```
//: C15:UnAbstract.cpp
// Pure virtual destructors
// seem to behave strangely

class AbstractBase {
public:
  virtual ~AbstractBase() = 0;
};

AbstractBase::~AbstractBase() {}

class Derived : public AbstractBase {};
// No overriding of destructor necessary?
668
int main() { Derived d; } ///:~
```

一般来说，如果在派生类中基类的纯虚函数（和所有其他纯虚函数）没有重新定义，则派生类将会成为抽象类。但这里，看起来好像并不是这样。然而，如果不进行析构函数定义，编译器将会自动地为每个类生成一个析构函数定义。那就是这里所发生的——基类的析构函数被重写（重新定义），因此编译器会提供定义并且派生类实际上不会成为抽象类。

这会产生一个有趣的问题：纯虚析构函数的目的是什么？它不像普通的纯虚函数，我们必须提供一个函数体。在派生类中，由于编译器为我们生成了析构函数，所以我们并非一定要提供一个定义。那么，常规的析构函数和纯析构函数的差别是什么呢？

当我们的类仅含有一个纯虚函数时，就会发现这个惟一的差别：析构函数。在这一点上，析构函数的纯虚性的惟一效果是阻止基类的实例化。如果有其他的纯虚函数，则它们会阻止基类的实例化，但如果没有那些纯虚函数，则纯虚析构函数将会执行这项操作。所以，当虚析构函数是十分必要时，则它是不是纯虚的就不是那么重要了。

运行下面的程序，可以看到在派生类版本之后，随着任何其他的析构函数，调用了纯虚函数体。

```
//: C15:PureVirtualDestructors.cpp
// Pure virtual destructors
// require a function body
#include <iostream>
using namespace std;

class Pet {
public:
  virtual ~Pet() = 0;
};

Pet::~Pet() {
  cout << "~Pet()" << endl;
}

class Dog : public Pet {
public:
  ~Dog() {
    cout << "~Dog()" << endl;
  }
};

int main() {
  Pet* p = new Dog; // Upcast
  delete p; // Virtual destructor call
} ///:~
```

669

作为一个准则，任何时候我们的类中都要有一个虚函数，我们应当立即增加一个虚析构函数（即使它什么也不做）。这样，我们保证在后面不会出现问题。

15.11.2 析构函数中的虚机制

在析构期间，有一些我们可能不希望马上发生的情况。如果正在一个普通的成员函数中，并且调用一个虚函数，则会使用晚捆绑机制来调用这个函数。而对于析构函数，这样不行，不论是虚的还是非虚的。在析构函数中，只有成员函数的“本地”版本被调用；虚机制被忽略。

```
//: C15:VirtualsInDestructors.cpp
// Virtual calls inside destructors
#include <iostream>
using namespace std;

class Base {
public:
  virtual ~Base() {
    cout << "Base1()\n";
    f();
  }
  virtual void f() { cout << "Base::f()\n"; }
};

class Derived : public Base {
```

670

```
public:
  ~Derived() { cout << "~Derived()\n"; }
  void f() { cout << "Derived::f()\n"; }
};

int main() {
  Base* bp = new Derived; // Upcast
  delete bp;
} ///:~
```

在析构函数的调用中，**Derived::f()**没有被调用，即使**f()**是一个虚函数。

为什么是这样呢？假设在析构函数中使用虚机制，那么调用下面这样的虚函数是可能的：这个函数是在继承层次中比当前的析构函数“更靠外的”（更晚派生的）。但是，有一点要注意，析构函数从“外层”（从最晚派生的析构函数向基类析构函数）被调用。所以，实际上被调用的函数就可能操作在已被删除的对象上。因此，编译器决定在编译时只调用这个函数的“本地”版本。注意，对于构造函数也是如此（这在前面已讲到），但在构造函数的情况下，这样做是因为类型信息还不可用，然而在析构函数中，这样做是因为信息（也就是**VPTR**）虽存在，但不可靠。

15.11.3 创建基于对象的继承

本书中，在对容器类**Stack** 和 **Stash**的描述中，有一点是重复出现的，这就是“所有权问题”。负责对动态创建（使用**new**）的对象进行**delete**调用的称之为“所有者”。在使用容器时的问题是，它们需要足够的灵活性用来接收不同类型的对象。为了做到这一点，容器使用**void**指针，因此它们并不知道所包容对象的类型。删除一个**void**指针并不调用析构函数，所
671 以容器并不负责清除它的对象。

在第14章的例子**InheritStack.cpp**中提出了一种解决办法，从**Stack**继承出一个仅可以接收和生成**string**指针的类。所以它知道它只包容了指向**string**对象的指针，因此它可以正确地删除它们。这是一个不错的解决办法，但是它要求我们要为想在容器中容纳的每一种类型都派生出一个新类。（虽然现在看起来有点冗余，但在第16章中介绍过模板后，它运行得相当不错。）

问题是我们希望容器可以容纳更多的类型，但我们不想使用**void**指针。另外一种解决方法是使用多态性，它通过强制容器内的所有对象从同一个基类继承而来。这就是说，容器容纳了具有同一基类的对象，并随后调用虚函数——特别地，我们可以调用虚析构函数来解决所有权问题。

这种解决方法使用*单根继承*（*singly-rooted hierarchy*）或*基于对象的继承*（*object-based hierarchy*）（这是因为继承的根类通常称为“对象”）。可以看到使用单根继承还有其他一些优点。事实上，除了C++，每种面向对象的语言都强制使用这样的体系——当创建一个类时，都会直接或间接地从一个公共基类中继承出它，这个基类是由该语言的创建者生成的。C++中认为，强制地使用这个公共基类会引起太多的开销，所以便没有使用它。然而，我们可以在自己的项目中选择是否使用它，在本书的第2卷中将进一步讨论这个主题。

为了解决所有权问题，可以创建一个相当简单的类**Object**作为基类，它仅包含一个虚析构函数。**Stack**于是可以容纳继承自**Object**的类。

```
//: C15:OStack.h
```

```
// Using a singly-rooted hierarchy
#ifndef OSTACK_H
#define OSTACK_H

class Object {
public:
  virtual ~Object() = 0;
};

// Required definition:
inline Object::~Object() {}

class Stack {
  struct Link {
    Object* data;
    Link* next;
    Link(Object* dat, Link* nxt) :
      data(dat), next(nxt) {}
  }* head;
public:
  Stack() : head(0) {}
  ~Stack(){
    while(head)
      delete pop();
  }
  void push(Object* dat) {
    head = new Link(dat, head);
  }
  Object* peek() const {
    return head ? head->data : 0;
  }
  Object* pop() {
    if(head == 0) return 0;
    Object* result = head->data;
    Link* oldHead = head;
    head = head->next;
    delete oldHead;
    return result;
  }
};
#endif // OSTACK_H ///:~
```

672

通过把所有的东西放在头文件中来简化问题，纯虚析构函数（所要求的）的定义以内联形式置于头文件中，并且**pop( )**也是内联的（对于内联形式来说，它可能太大了）。

Link对象现在是指向**Object**指针，而不是**void**指针，并且**Stack**也将仅仅接收和返回**Object**指针。现在，**Stack**更具有灵活性，因为它容纳了大量不同的类型，而且也可以消除被置于**Stack**中的任一对象。新的限制（在第16章中，当对这个问题运用模板时，将不具有这个限制）是置于**Stack**中的所有内容都必须继承自**Object**。如果新建一个类，这还是可行的，但如果已经有了一个类（例如**string**），并且希望把它置于**Stack**中，又会如何呢？这种情况下，新类必须具备**string** 和 **Object**的特点，即它必须继承自这两个类。这称之为多重继承（*multiple inheritance*），在本书第2卷（可从*www.BruceEckel.com*处下载）中有一整章是关于这个主题的。当我们阅读该章时，将会看到多重继承是非常复杂的，应尽量少用这一功能。

673

然而，在这里，所有的一切都是很简单的，所以无需考虑多重继承的任何缺点。

```
//: C15:OStackTest.cpp
//{T} OStackTest.cpp
#include "OStack.h"
#include "../require.h"
#include <fstream>
#include <iostream>
#include <string>
using namespace std;

// Use multiple inheritance. We want
// both a string and an Object:
class MyString: public string, public Object {
public:
  ~MyString() {
    cout << "deleting string: " << *this << endl;
  }
  MyString(string s) : string(s) {}
};

int main(int argc, char* argv[]) {
  requireArgs(argc, 1); // File name is argument
  ifstream in(argv[1]);
  assure(in, argv[1]);
  Stack textlines;
  string line;
  // Read file and store lines in the stack:
674 while(getline(in, line))
    textlines.push(new MyString(line));
  // Pop some lines from the stack:
  MyString* s;
  for(int i = 0; i < 10; i++) {
    if((s=(MyString*)textlines.pop())==0) break;
    cout << *s << endl;
    delete s;
  }
  cout << "Letting the destructor do the rest:"
    << endl;
} ///:~
```

虽然这个代码段与**Stack**以前的测试程序版本很相似，但我们注意到仅有10个元素从栈中弹出，这意味着还保留了一些对象。因为**Stack**知道它包容了**Object**，并且析构函数可以正确地把它们清除掉。因为**MyString**对象在它们被清除时打印信息，所以我们可从程序的输出中知道这一点。

创建包容**Object**的容器是一种合理的方法——如果使用单根继承（由于语言本身或需要的缘故，强制每个类继承自**Object**）。这时，保证一切都是一个**Object**，因此在使用容器时并不是十分复杂。然而，在C++中，不能期望这适用于每个类，所以如果有多重继承会出现问题。在第16章中会看到模板可以使用更简单、更灵巧的方法来处理这个问题。

15.12 运算符重载

就像对成员函数那样，我们可以使用**virtual**运算符。然而，因为我们可能对两个不知道

类型的对象进行操作，所以实现**virtual**运算符通常会很复杂。这通常用于处理数学部分（对于它们，我们常常重载运算符）。例如，对于一个处理矩阵、向量和标量的系统，这3个成分都是派生自**Math**类。

675

```
//: C15:OperatorPolymorphism.cpp
// Polymorphism with overloaded operators
#include <iostream>
using namespace std;

class Matrix;
class Scalar;
class Vector;

class Math {
public:
  virtual Math& operator*(Math& rv) = 0;
  virtual Math& multiply(Matrix*) = 0;
  virtual Math& multiply(Scalar*) = 0;
  virtual Math& multiply(Vector*) = 0;
  virtual ~Math() {}
};

class Matrix : public Math {
public:
  Math& operator*(Math& rv) {
    return rv.multiply(this); // 2nd dispatch
  }
  Math& multiply(Matrix*) {
    cout << "Matrix * Matrix" << endl;
    return *this;
  }
  Math& multiply(Scalar*) {
    cout << "Scalar * Matrix" << endl;
    return *this;
  }
  Math& multiply(Vector*) {
    cout << "Vector * Matrix" << endl;
    return *this;
  }
};

class Scalar : public Math  {
public:
  Math& operator*(Math& rv) {
    return rv.multiply(this); // 2nd dispatch
  }
  Math& multiply(Matrix*) {
    cout << "Matrix * Scalar" << endl;
    return *this;
  }
  Math& multiply(Scalar*) {
    cout << "Scalar * Scalar" << endl;
    return *this;
  }
  Math& multiply(Vector*) {
```

676

```
    cout << "Vector * Scalar" << endl;
    return *this;
  }
};

class Vector : public Math  {
public:
  Math& operator*(Math& rv) {
    return rv.multiply(this); // 2nd dispatch
  }
  Math& multiply(Matrix*) {
    cout << "Matrix * Vector" << endl;
    return *this;
  }
  Math& multiply(Scalar*) {
    cout << "Scalar * Vector" << endl;
    return *this;
  }
  Math& multiply(Vector*) {
    cout << "Vector * Vector" << endl;
    return *this;
  }
};

int main() {
  Matrix m; Vector v; Scalar s;
  Math* math[] = { &m, &v, &s };
  for(int i = 0; i < 3; i++)
    for(int j = 0; j < 3; j++) {
      Math& m1 = *math[i];
      Math& m2 = *math[j];
      m1 * m2;
    }
} ///:~
```

为了简单起见，这里仅重载了**operator***。重载的目的是使任意两个**Math**对象相乘并且生
677 成所需的结果——注意矩阵乘以向量和向量乘以矩阵是两个完全不同的操作。

main( )中的问题在于，表达式**m1 * m2**包含了两个向上类型转换的**Math**引用，因此不知道这两个对象的类型。一个虚函数仅能进行单一指派——即判定一个未知对象的类型。本例中所使用的判定两个对象类型的技术称之为*多重指派*（*multiple dispatching*），一个单一虚函数调用引起了第二个虚函数调用。在完成第二个调用时，已经得到了这两个对象的类型，于是可以执行正确的操作。我们开始时会有点不清楚，但如果多看些例子，就会理解的。这个主题在本书的第2卷（可从*www.BruceEckel.com*处下载）的“设计风格”一章中有更深入的探讨。

15.13 向下类型转换

我们可能猜测，既然存在向上类型转换——在类层次中向上移动，那也应该存在可以向下移动的*向下类型转换*（*downcasting*）。但是由于在一个继承层次上向上移动时，类总是集中于更一般的类，因此向上类型转换是容易的。这就是说，当进行向上类型转换时，总是清楚地派生自祖先类（典型地总是一个，除了多重继承的情况），而当向下类型转换时，通常会有多种选择让我们进行类型转换。更特殊些，**Circle**是**Shape**的一种类型（这是向上类型转换），但如果对一

个**Shape**进行向下类型转换，它可能会是**Circle**、**Square**、**Triangle**等。因此，对于安全地进行向下类型转换，就出现了两难的选择。(但更重要的是，要问问自己，为什么首先使用向下类型转换而不用多态性来自动地获取正确的类型。在本书的第2章中介绍了向下类型转换的避免。)

C++提供了一个特殊的称为**dynamic_cast**的*显示类型转换*(*explicit cast*)(在第3章中介绍过)，它就是一种*安全类型向下类型转换*(*type-safe downcast*)的操作。当使用**dynamic_cast**来试着向下类型转换一个特定的类型，仅当类型转换是正确的并且是成功的时，返回值会是一个指向所需类型的指针，否则它将返回0来表示这并不是正确的类型。下面有一个小例子。

678

```
//: C15:DynamicCast.cpp
#include <iostream>
using namespace std;

class Pet { public: virtual ~Pet(){}};
class Dog : public Pet {};
class Cat : public Pet {};

int main() {
  Pet* b = new Cat; // Upcast
  // Try to cast it to Dog*:
  Dog* d1 = dynamic_cast<Dog*>(b);
  // Try to cast it to Cat*:
  Cat* d2 = dynamic_cast<Cat*>(b);
  cout << "d1 = " << (long)d1 << endl;
  cout << "d2 = " << (long)d2 << endl;
} ///:~
```

当使用**dynamic_cast**时，必须对一个真正多态的层次进行操作——它含有虚函数——这因为**dynamic_cast**使用了存储在VTABLE中的信息来判断实际的类型。这里，基类含有一个虚析构函数并定义了它。**main( )**中，一个**Cat**指针被向上类型转换到**Pet**，然后又试着向下类型转换到一个**Dog**指针和一个**Cat**指针。运行这个程序时，打印出这两个指针，可以看到不正确的向下类型转换返回了0值。当然，无论何时进行向下类型转换，我们都有责任进行检验以确保类型转换的返回值为非0。但我们不用确保指针要完全一样，这是因为通常在向上类型转换和向下类型转换时指针会进行调整(特别是在多重继承的情况下)。

dynamic_cast运行时需要一点额外的开销；不多，但如果执行大量的**dynamic_cast**(这时我们的程序设计就有严重的问题)，就会影响性能。有时，在进行向下类型转换时，我们可以知道正在处理的是何种类型，这时使用**dynamic_cast**产生的额外开销就没有必要，可以通过使用**static_cast**来代替它。

679

```
//: C15:StaticHierarchyNavigation.cpp
// Navigating class hierarchies with static_cast
#include <iostream>
#include <typeinfo>
using namespace std;

class Shape { public: virtual ~Shape() {}; };
class Circle : public Shape {};
class Square : public Shape {};
class Other {};
```

```
int main() {
  Circle c;
  Shape* s = &c; // Upcast: normal and OK
  // More explicit but unnecessary:
  s = static_cast<Shape*>(&c);
  // (Since upcasting is such a safe and common
  // operation, the cast becomes cluttering)
  Circle* cp = 0;
  Square* sp = 0;
  // Static Navigation of class hierarchies
  // requires extra type information:
  if(typeid(s) == typeid(cp)) // C++ RTTI
    cp = static_cast<Circle*>(s);
  if(typeid(s) == typeid(sp))
    sp = static_cast<Square*>(s);
  if(cp != 0)
    cout << "It's a circle!" << endl;
  if(sp != 0)
    cout << "It's a square!" << endl;
  // Static navigation is ONLY an efficiency hack;
  // dynamic_cast is always safer. However:
  // Other* op = static_cast<Other*>(s);
  // Conveniently gives an error message, while
  Other* op2 = (Other*)s;
  // does not
} ///:~
```

在这个程序中，使用了一个新的特征，本书第2卷会有一章完全介绍这一主题：C++的运行时类型识别（*Run-Time Type Information,* RTTI）机制。RTTI允许我们得到在进行向上类型转换时丢失的类型信息。**dynamic_cast**实际上就是RTTI的一种形式。这里，**typeid**关键字
680 （在头文件**<typeinfo>**中声明）用来检测指针的类型。可以看到，向上类型转换的**Shape**指针的类型相继与**Circle**指针和**Square**指针相比较，来判断它们是否匹配。RTTI的内容远远不止**typeid**，我们也可以想像它能通过虚函数简单合理地实现我们自己的类型信息系统。

程序创建了一个**Circle**对象，它的地址被向上类型转换为**Shape**指针；第二个表达式显示了我们如何使用**static_cast**来进行更加显式地向上类型转换。然而，由于向上类型转换总是安全的并且是通用的，因此我认为用一个显式类型转换来进行向上类型转换将是混乱和没有必要的。

RTTI用于判定类型，**static_cast**用于执行向下类型转换。但要注意，在这个设计中，处理效率同使用**dynamic_cast**是一样的，并且客户程序员必须进行检测来发现那些实际成功的类型转换。我们希望在不使用**dynamic_cast**而使用**static_cast**之前，有一个比上面例子更加确定的环境（并且在使用**dynamic_cast**之前，我们希望可以再一次仔细地检查我们的设计）。

如果类层次中没有虚函数（这是一个有问题的设计），或者如果有其他的需要，要求我们安全地进行向下类型转换，与使用**dynamic_cast**相比，静态地执行向下类型转换会稍微快一点。另外，**static_cast**不允许类型转换到该类层次的外面，而传统的类型转换是允许的，所以它们会更安全。但是静态地浏览类层次总是有风险的，所以除非特殊情况，我们一般使用**dynamic_cast**。

15.14 小结

多态性在C++中用虚函数实现，它意味着“具有不同的形式”。在面向对象的程序设计中，

有相同的功能（即基类中的公共接口）和使用这个功能的不同的形式：虚函数的不同版本。

在本章中，我们已经看到，不用数据抽象和继承，理解甚至创建一个多态的例子，是不 681
可能的。多态是不能独立看待的特征（例如像**const**或**switch**这样的语句），必须协同工作，它是类关系大家庭中的一部分。人们常常被C++的其他非面向对象的特征（例如重载和默认参数）所混淆，它们有时被作为面向对象的特征描述。不要被迷惑，如果它们没有进行晚捆绑，就没有多态性。

为了在程序中有效地使用多态等面向对象的技术，不能只知道让程序包含单个类的成员和消息，而且还应知道类的共性和它们之间的关系。虽然这需要很大的努力，但这是值得的，因为将得到更快的程序开发和更好的代码组织、可扩充的程序和更容易维护的代码。

多态性完善了语言的面向对象特征，但在C++中，还有两个更重要的特征：模板（第16章的内容，并且在第2卷中有更为详细介绍）和异常处理（在第2卷中介绍）。就像面向对象的其他特征（抽象数据类型、继承和多态）一样，这些特征使我们的编程能力有很大的提高。

15.15 练习

部分练习题的答案可以在本书的电子文档 *"Annotated Solution Guide forThinking in C++"* 中找到，只需支付很少的费用就可以从http://www.BruceEckel.com得到这个电子文档。

15-1 创建一个非常简单的"shape"层次：基类称为**Shape**，派生类称为**Circle**、**Square**和**Triangle**。在基类中定义一个虚函数**draw( )**，再在这些派生类中重定义它。在堆中创建**Shape**对象，并且建立一个指向这些**Shape**对象的指针数组（这样就形成了指针向上类型转换）。并且通过基类指针调用**draw( )**，检验虚函数的行为。如果调试器支持，就用单步执行这个例子。 682

15-2 修改练习1，使得**draw( )**是纯虚函数。尝试创建一个类型为**Shape**的对象。并试着在构造函数内调用这个纯虚函数，看看结果如何。保留它的纯虚性，对**draw( )**进行定义。

15-3 在练习2的基础上进一步，创建一个通过传值方式接收**Shape**对象参数的函数，并试着向上类型转换一个派生类对象作为参数。看看结果如何。通过把参数设为**Shape**对象的引用来修改这个函数。

15-4 修改**C14:Combined.cpp**，把基类中的**f( )**设为虚函数。在**main( )**中执行向上类型转换并且调用虚函数。

15-5 通过增加一个虚函数**prepare( )**来修改**Instrument3.cpp**。在**tune( )**中调用**prepare( )**。

15-6 创建一个含有**Rodent**类的继承层次，它包括**Mouse**、**Gerbil**、**Hamster**等。在基类中提供对所有**Rodent**都适用的方法，并根据**Rodent**的特定类型，在派生类中执行不同的行为。创建一个**Rodent**指针数组，使它们指向**Rodent**不同的特定类型，并且调用基类中的方法，看看结果如何。

15-7 修改练习6，用**vector<Rodent*>**来代替指针数组。确保内存可以被正确地清除掉。

15-8 根据前面的**Rodent**类层次，从**Hamster**中继承出**BlueHamster**（是的，当我还是小孩时，我就有这么一只鼠），重新定义基类中的方法，并显示调用基类方法的代码不需要进行修改就可以在新类中使用。

15-9 在前面的**Rodent**类层次中，增加一个虚析构函数，并且使用**new**创建一个**Hamster**

对象，向上类型转换成为一个**Rodent***，然后**delete**该指针，显示它并不能调用层次中所有的析构函数。把析构函数改为虚函数，显示这样就可以正确地调用所有的析构函数。

683 15-10 在前面的**Rodent**类层次中，修改**Rodent**，使它成为一个纯抽象基类。

15-11 使用基类**Aircraft**和它的不同的派生类，创建一个空中交通系统。使用**vector<Aircraft*>**建立类**Tower**，给在它控制下的不同飞行器发送适当的信息。

15-12 通过从**Plant**中继承各种类型来建立一个温室的模型，并且在该温室中创建可以照看植物的机制。

15-13 在**Early.cpp**中，使**Pet**成为一个纯抽象基类。

15-14 在**AddingVirtuals.cpp**中，把**Pet**所有的成员函数改为纯虚函数，并对**name( )**进行定义。使用**name( )**的基类定义，对**Dog**进行必要的修改。

15-15 写出一个小程序以显示在普通成员函数中调用虚函数和在构造函数中调用虚函数的不同。这个程序应当表明两种调用会产生不同的结果。

15-16 通过从**Derived**中继承出一个类并且重新定义它的**f( )**和析构函数来修改**VirtualsInDestructors.CPP**。在**main( )**中，向上类型转换我们的新类，然后**delete**它。

15-17 在练习16的基础上，在每一个析构函数中增加对函数**f( )**的调用。解释运行的结果。

15-18 创建含有一个数据成员的类和含用另一个数据成员的派生类。编写一个非成员函数，它通过传值方式接收一个基类的对象，并且使用**sizeof**打印出该对象的大小。在**main( )**中创建一个派生类的对象，打印出它的大小，然后调用我们的函数。解释运行的结果。

15-19 创建一个虚函数调用的简单例子，并且输出其汇编代码。找出虚函数调用的汇编代码，跟踪运行并解释这些代码。

15-20 编写一个类，含有一个虚函数和一个非虚函数。继承出一个新类，并生成该类的
684 对象，然后向上类型转换为基类的指针。使用**<ctime>**中的**clock( )**函数（需要在本地C库指南中找到它）来测出虚函数调用和非虚函数调用的区别。为了看到区别，需要在时间循环中对每个函数进行多次调用。

15-21 通过在**CLASS**宏的基类中增加一个虚函数（使它打印些信息）并且把析构函数改为虚函数来修改**C14:Order.cpp**。生成不同子类的对象，然后把它们向上类型转换为基类对象。检验虚操作的运行以及发生的适当的构造操作和析构操作。

15-22 编写一个含有3个重载虚函数的类。在新建类中继承出一个新类，并且重新定义其中一个函数。生成派生类的一个对象。我们是否可以通过派生类对象调用所有的基类函数呢？把该对象的地址向上类型转换为基类对象。我们是否可以通过此基类对象调用所有的3个函数呢？删去在派生类中所做的重写定义。现在我们又是否可以通过派生类对象调用所有的基类函数呢？

15-23 修改**VariantReturn.cpp**，显示它的行为可以使用引用和指针来进行工作。

15-24 在**Early.cpp**中，如何才能分辨出编译器的调用是使用了早捆绑还是晚捆绑？判断我们自己的编译器的调用属于哪种情况？

15-25 创建一个基类，含有一个**clone( )**函数，它返回指向当前对象拷贝的指针。派生出

两个子类，同时重新定义**clone( )**，它返回它们各自类型拷贝的指针。在**main( )**中，生成并且向上类型转换两个派生类型的对象，然后分别调用它们的**clone( )**，并检验所克隆的拷贝是正确的子类型。试验我们的**clone( )**函数，使得返回的类型是基类，再试着返回准确的派生类型。我们能否考虑到后一种方法所必需的环境？

15-26 通过创建自己的类，然后对它和**Object**进行多重继承，生成的对象置于**Stack**中来修改**OStackTest.cpp**。在**main( )**中测试我们的类。

15-27 在**OperatorPolymorphism.cpp**中增加一个类**Tensor**。 685

15-28 (中级)创建一个不带数据成员和构造函数而只有一个虚函数的基类**X**，从**X**继承出类**Y**，它没有显式的构造函数。产生汇编代码并检验它，以确定**X**的构造函数是否被创建和调用，如果是的，这些代码做什么？解释我们的发现。**X**没有默认的构造函数，但是为什么编译器不报告出错？

15-29 (中级)修改练习28，为这两个类创建构造函数，让每个构造函数调用一个虚函数。产生汇编代码。确定在每个构造函数内VPTR在何处被赋值。在构造函数内编译器使用虚函数机制吗？确定为什么这些函数的本地版本仍被调用。

15-30 (高级)如果对象的参数为传值方式传递的函数调用不用早捆绑，则虚调用可能会访问不存在的部分。这可能吗？编写一些代码强制进行虚调用，看看是否会引起冲突。解释这个行为，检验当对象以传值方式传递时会发生什么现象。

15-31 (高级)通过我们处理器的汇编语言信息或者其他技术，找出简单调用所需的时间数及虚函数调用所需的时间数，从而得出虚函数调用需要多出多少时间。

15-32 确定执行时VPTR的**Sizeof**。现在对两个含有虚函数的类进行多重继承。在派生类中可以得到一个还是两个VPTR？

15-33 创建一个含有数据成员和虚函数的类。编写一个监视我们类对象的内存的函数，它打印出变化的部分。要做到这一点，我们需要进行试验并且不断地找出对象中VPTR的所在位置。 686

15-34 假设不存在虚函数，修改**Instrument4.cpp**，使得它使用**dynamic_cast**来代替虚函数调用。解释为什么这不是一个好的方法。

15-35 修改**StaticHierarchyNavigation.cpp**，不使用C++ RTTI，而是通过基类中的虚函数**whatAmI( )**和**enum type { Circles, Squares }**，来创建我们自己的RTTI。

15-36 在第12章的**PointerToMemberOperator.cpp**中，显示即使重载了**operator->***，多态性依旧适用于成员指针。 687

第16章 模板介绍

继承和组合提供了重用对象代码的方法，而C++的模板特征提供了重用源代码的
689 *方法。*

虽然 C++模板是通用的程序设计工具，但当它们引入了C++后，似乎就不再鼓励使用基于对象的容器类层次结构(在第15章的最后论证)了。例如,标准C++容器类和算法(在本书的第2卷中有两章对此问题进行解释,可以从*www.BruceEckel.com*下载本书的第2卷。)是完全应用模板完成的，对程序员来说相对易于使用。

本章不仅阐述模板的基础，而且还介绍容器，它是面向对象程序设计的基本构件，几乎可以完全通过标准C++库中的容器实现。可以看到，本书使用的容器的例子——**Stash** 和 **Stack**——刚好适合于学习容器。在本章中，增加了迭代器（*iterator*）的概念。虽然容器是与模板一起使用的理想的例子，但是在第2卷中（其中有一章是专门讨论高级模板）将会学到模板的许多别的用法。

16.1 容器

假定想创建一个栈，正如全书所做的这样。为了简单，这个栈类只存放**int**类型的值。

```
//: C16:IntStack.cpp
// Simple integer stack
//{L} fibonacci
#include "fibonacci.h"
#include "../require.h"
#include <iostream>
using namespace std;

class IntStack {
  enum { ssize = 100 };
  int stack[ssize];
  int top;
public:
690 IntStack() : top(0) {}
  void push(int i) {
    require(top < ssize, "Too many push()es");
    stack[top++] = i;
  }
  int pop() {
    require(top > 0, "Too many pop()s");
    return stack[--top];
  }
};

int main() {
  IntStack is;
  // Add some Fibonacci numbers, for interest:
```

```
  for(int i = 0; i < 20; i++)
    is.push(fibonacci(i));
  // Pop & print them:
  for(int k = 0; k < 20; k++)
    cout << is.pop() << endl;
} ///:~
```

类**IntStack**是最为常见的下推栈的例子。为了简化，此处栈的尺寸是固定的，但是也可以对其进行修改，使得它能通过在堆中分配内存而自动扩展，如同在本书中到处被考查的**Stack**类一样。

main( )向这个栈添加一些整数，然后再弹出它们。为了让这个例子更有趣，这些整数用**fibonacci( )**函数生成，它生成传统的兔子繁殖数。下面是声明这个函数的头文件。

```
//: C16:fibonacci.h
// Fibonacci number generator
int fibonacci(int n); ///:~
```

下面是实现：

```
//: C16:fibonacci.cpp {O}
#include "../require.h"

int fibonacci(int n) {
  const int sz = 100;
  require(n < sz);
  static int f[sz]; // Initialized to zero
  f[0] = f[1] = 1;
  // Scan for unfilled array elements:
  int i;
  for(i = 0; i < sz; i++)
    if(f[i] == 0) break;
  while(i <= n) {
    f[i] = f[i-1] + f[i-2];
    i++;
  }
  return f[n];
} ///:~
```

691

这是一个相当有效的实现，因为它决不会多次生成这些数。它使用**int**的**static**数组，编译器将这个**static**数组初始化为零。第一个**for**循环把下标**i**移到第一个数组元素为零的地方，然后**while**循环向这个数组添加斐波纳契数，直到期望的元素达到。但是注意，如果经过元素**n**的斐波纳契数（Fibonacci number）都已经被初始化，则完全跳过这个**while**循环。

16.1.1 容器的需求

很明显，一个整数栈不是一个重要的工具。容器类的真正需求是在堆上使用**new**创建对象和使用**delete**销毁对象的时候体现的。在一般程序设计问题中，程序员在编写程序时并不知道将来需要创建多少个对象。例如在设计空中交通指挥系统时不应限制这个系统能处理的飞机数目。我们不希望由于实际飞机的数目超过设计值而导致这个系统失败。在计算机辅助设计系统中，可以处理许多造型，只有用户能够（在运行时）确定到底需要多少造型。我们一旦注意到上述问题，便可以在程序开发中发现许多这样的例子。

依赖虚拟存储去处理“存储器管理”的C程序员常常发现**new**、**delete**和容器类的思想的
692 混乱。表面上看，创建一个足够大的能包括任何可能需求的巨型全局数组是可行的。这可能不需要太多思考（或者并不需要弄清楚**malloc( )**和**free( )**），但是这样的程序接口性能较差，而且暗藏着难以捕捉的错误。

另外，如果我们创建一个巨型的C++对象的全局数组，那么构造函数和析构函数的开销会使系统效率显著地下降。C++中有更好的解决方法：用**new**创建需要的对象，将其指针放入容器中，待实际使用时将其取出并进行处理。用这种方法，所创建的只是确实需要的对象。通常，在启动程序时没有可用的初始化条件。**new**允许等待，直到在环境中相关事件发生后，再实际地创建这个对象。

在大多数情况下，应当创建用来存放感兴趣对象指针的容器。应当用**new**创建这些对象，然后把结果指针放在容器中（在这个过程中这是向上类型转换），当需要用到这些对象时再将指针从容器中取出。这项技术使得程序更具灵活性和一般性。

16.2 模板综述

现在出现了一个问题。**IntStack**可存放整数，但是也可能希望有一个栈可存放造型、航班、植物等数据对象。用强调重用性的语言每次从头重新开发代码，不是一个明智的办法。应该有更好的方法。

有3种源代码重用的方法：C方法，这里列出是为了对照；对C++产生过重大影响的Smalltalk方法；C++的模板方法。

（1）C方法

毫无疑问，应该摒弃C方法，这是由于它表现繁琐、易发生错误、缺乏美感。用这种方法，
693 需要拷贝**Stack**的源码并对其进行手工修改，这样就会引进新的错误。这是非常低效的技术。

（2）**Smalltalk** 方法

Smalltalk（以及之后的Java）方法是通过继承来实现代码重用的，既简单又直观。为此，每个容器类包含通用的基类**Object**的项目（类似于第15章最后的例子）。Smalltalk的基类库十分重要，完全不需要从头创建类。相反，创建一个新类必须从已有类中继承，不能随意创建。可以从类库中选择功能和需求尽可能接近的一个已有类作为父类，并在对父类的继承中加以修正从而创建一个新类。很明显，这种方法由于可以减少我们的工作量，因而提高了我们的效率（这也说明了为什么需要花大量的时间去学习Smalltalk类库才能成为熟练的Smalltalk程序员）。

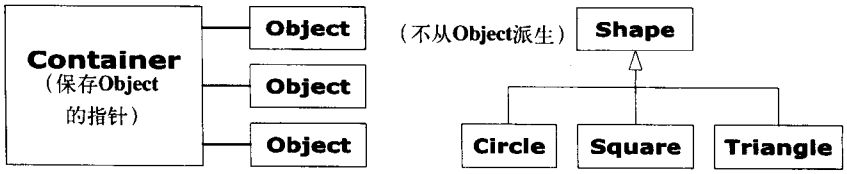
但是，这也意味着Smalltalk的所有类都是单个继承树的一部分。当创建新类时必须继承树的某一枝。树的大部分已经存在（它是Smalltalk的类库），树的根称为Object——这是每个Smalltalk容器所包含的同一个类。

这是一种单纯的技巧，因为Smalltalk（和Java[⊖]）类层次上的任何类都源于**Object**的派生，所以任何容器可容纳任何类（包括容器本身）。这种基于通用的基类（常称为**Object**，在Java中也有类似情况）的单树形层次类型称为“基于对象的层次结构”。我们可能听说过这个概念，并猜想这是另一个OOP的基本概念，就像“多态性”一样。但实际上，这仅仅意味着以**Object**（或相近的名称）为根的树形类结构和包含**Object**的容器类。

⊖ 在Java中，基本数据类型是一个例外，出于效率的考虑，这里有一些非**Object**类型。

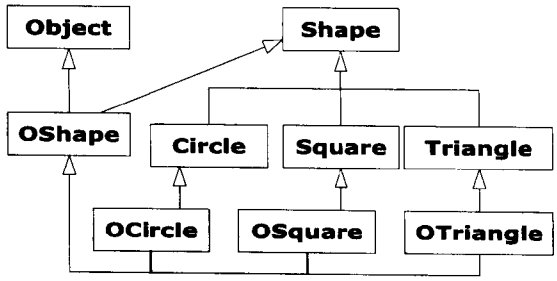
因为Smalltalk类库的发展史比C++更长久，且早期的C++编译器没有容器类库，所以C++能将Smalltalk类库的良好思想加以借鉴。这种借鉴出现在早期的C++实现中[⊖]，由于它表现为 694
一个有效的代码实体，因此许多人开始使用它，但在使用容器类的过程中发现了一个问题。

该问题在于，在Smalltalk（和我所知道的许多其他OOP语言）中，所有的类都自动地从单个层次结构中派生而来，但在C++中则不行。我们可能本来已经拥有了完善的基于对象的层次结构以及它的容器类，而且还可能从其他不用这种层次结构的供应商那里购买到一组类，如形体类、航班类等（为了使用层次结构而增加了开销，这是C程序员不愿意做的事情。）我们如何把一个单独的类树插入到我们的基于对象的层次结构中的容器类之中呢？这个问题如下所示：



因为C++支持多个独立层次结构，所以Smalltalk的“基于对象的层次结构”在此不适用。

解决方案似乎是明显的。如果我们有许多继承层次结构，就应当能从多个类继承：多重继承可以解决上述问题。所以我们应该按下述的方法去实施（一个类似的例子已在第15章结尾处给出）：



现在，**OShape**具有了**Shape**的特点和行为，但它也是从**Object**派生而来的，所以可将其 695
置于**Container**内。额外的继承也必然进入了**OCircle**和**OSquare**等，这样这些类才能向上类型转换为**OShape**，并因而保持正确的行为。我们可以看到，事情正在迅速变得混乱。

编译器供应商发明了他们自己的基于对象的容器类层次结构，并将它们加入到他们的编译系统中，这些层次结构中的大多数可以用模板版本替代。我们可以对多重继承是否可以解决大多数编程问题进行争论，但是在本书的第2卷中将会看到，除某些特殊情况外，它的复杂性是可以很好避免的。

16.2.1 模板方法

尽管具有多重继承的基于对象的层次结构在概念上是直观的，但是它在实践上较为困难。

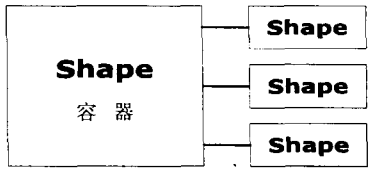
⊖ OOPS库，由Keith Gorlen在NIH时创建。

在Stroustrup的最初著作[⊖]中阐述了替换基于对象层次的一种更可取的选择。创造容器类作为参数化类型的大型预处理宏，这些参数能替代为所希望的类型。当我们打算创建一个容器存放某个特别类型时，应当使用一对宏调用。

不幸的是，这种方法在当时被所有已有的Smalltalk文献和程序设计经验弄混淆了，加之它确实有点难处理，所以当时基本上没有什么人用它。

在此期间，Stroustrup和贝尔实验室的C++小组对原先的宏方法进行了修正，对其进行了简化并将它从预处理器域移入到编译器中。这种新的代码替换装置被称为模板[⊖]，而且它表现了完全不同的代码重用方法：模板对源代码进行重用，而不是通过继承和组合重用目标代
696 码。容器不再存放称为**Object**的通用基类，而是存放一个未指明的参数。当用户使用模板时，参数由编译器（*by the compiler*）来替换，这非常像原来的宏方法，但却更清晰、更容易使用。

现在，可以不必为使用容器类时的继承和组合担忧了，可以采用容器的模板并且为具体问题复制出特定的版本，就像下图所示：



编译器会为我们做这些工作，而我们最终是以所需要的容器去做我们的工作，而不是用那些令人头疼的继承层次。在C++中，模板实现了参数化类型（*parameterized type*）的概念。模板方法的另一个优点是，使对继承不熟悉、不适应的新程序员也能正确地使用密封的容器类（就像在本书中对**vector**所做的那样）。

16.3 模板语法

template这个关键字会告诉编译器，随后的类定义将操作一个或更多未指明的类型。当由这个模板产生实际类代码时，必须指定这些类型以使编译器能够替换它们。

下面是一个说明模板语法的小例子，它产生一个带有越界检查的数组。

```
//: C16:Array.cpp
#include "../require.h"
#include <iostream>
using namespace std;

template<class T>
697 class Array {
  enum { size = 100 };
  T A[size];
public:
  T& operator[](int index) {
    require(index >= 0 && index < size,
      "Index out of range");
```

⊖ 《C++程序设计语言》（*The C++ Programming Language*），由Bjarne Stroustrup著（第1版，Addison-Wesley公司1986年出版）。

⊖ 模板的思想类似于ADA的泛型（generic）。

```
    return A[index];
  }
};

int main() {
  Array<int> ia;
  Array<float> fa;
  for(int i = 0; i < 20; i++) {
    ia[i] = i * i;
    fa[i] = float(i) * 1.414;
  }
  for(int j = 0; j < 20; j++)
    cout << j << ": " << ia[j]
         << ", " << fa[j] << endl;
} ///:~
```

它看上去像一个普通的类，除了下面一行以外：

```
template<class T>
```

这里**T**是替换参数，它代表一个类型名称。在容器类中，它将出现在那些原本由某一特定类型出现的地方。

在**Array**中，其元素的插入和取出都用相同的函数——即重载的**operator[ ]**来实现。它返回一个引用，因此可被用于等号的两边（即，可以是*左值*也可以是*右值*）。注意，当下标值越界时，用**require( )**函数输出提示信息。因为**operator[ ]**是**内联的**，所以用这种方法来保证不发生数组下标越界现象，随后在提交代码时去掉**require( )**。

在**main( )**中，我们看到可以非常容易地创建包含不同类型的**Array**。代码如下： 698

```
Array<int> ia;
Array<float> fa;
```

这时，编译器两次扩展了**Array**模板［这被称为*实例化*（*instantiation*）］，创建两个新的*生成类*（*generated class*），可以把它们看做**Array_int**和**Array_float**（不同的编译器对名称有不同的修饰方法）。这些类就像手工创建的一样，只是这里是当定义了对象**ia**和**fa**后由编译器来创建这些类。我们还会注意到，编译器避免了或者连接器合并了类的重复定义。

16.3.1 非内联函数定义

当然，有时我们希望有非内联成员函数的定义。这时编译器需要在成员函数定义之前看到**template**声明。下面在前述例子的基础上加以修正来说明非内联函数的定义。

```
//: C16:Array2.cpp
// Non-inline template definition
#include "../require.h"

template<class T>
class Array {
  enum { size = 100 };
  T A[size];
public:
  T& operator[](int index);
};
```

```
template<class T>
T& Array<T>::operator[](int index) {
  require(index >= 0 && index < size,
    "Index out of range");
  return A[index];
}

int main() {
  Array<float> fa;
  fa[0] = 1.414;
} ///:~
```

699

注意在引用模板的类名的地方，必须伴有该模板的参数列表，例如在**Array<T>::operator[ ]**中。可以想像，在内部，使用模板参数列表中的参数修饰类名，以便为每一个模板实例产生惟一的类名标识符。

16.3.1.1 头文件

即使是在创建非内联函数定义时，我们还是通常想把模板的所有声明和定义都放入一个头文件中。这似乎违背了通常的头文件规则："不要放置分配存储空间的任何东西"（这条规则是为了防止在连接期间的多重定义错误），但模板定义很特殊。在**template<…>**之后的任何东西都意味着编译器在当时不为它分配存储空间，而是一直处于等待状态直到被一个模板示例告知。在编译器和连接器中有机制能去掉同一模板的多重定义。所以为了使用方便，几乎总是在头文件中放置全部的模板声明和定义。

有时，也可能为了满足特殊的需要（例如，强制模板示例仅存在于单个的Windows **dll**文件中）而要在一个独立的**cpp**文件中放置模板的定义。大多数编译器有一些机制允许这么做；我们将必须检查我们的特定编译器的说明文档以便使用它。

有些人认为，在实现中将所有源代码放在头文件中，如果有人从我们这里买到库，则他们就有条件盗窃和修改代码。这可能是一个问题，但它依赖于我们看待这个问题的方法：他们买的是产品还是服务？如果是产品，我们就必须为保护它做一些事情，或许我们不想给出源代码，而只给出编译过的代码。但是许多人把软件看做服务，甚至是预约服务。消费者想
700 要我们的专门技术，想要我们继续维护这段可重用的代码，所以他们没有必要这样做，因此他们可以集中精力做他们的事情。我个人认为，大多数消费者将我们看做有价值的资源，不希望危害他们与我们之间的关系。至于少数想盗窃而不是购买或做独创工作的人，他们大概无论如何也不能与我们相处。

16.3.2 作为模板的IntStack

下面是来自**IntStack.cpp**的容器和迭代器，是作为一般的容器类使用模板来实现的：

```
//: C16:StackTemplate.h
// Simple stack template
#ifndef STACKTEMPLATE_H
#define STACKTEMPLATE_H
#include "../require.h"

template<class T>
class StackTemplate {
  enum { ssize = 100 };
```

```
  T stack[ssize];
  int top;
public:
  StackTemplate() : top(0) {}
  void push(const T& i) {
    require(top < ssize, "Too many push()es");
    stack[top++] = i;
  }
  T pop() {
    require(top > 0, "Too many pop()s");
    return stack[--top];
  }
  int size() { return top; }
};
#endif // STACKTEMPLATE_H ///:~
```

注意，模板会对它包含的对象做一定的假设。例如，**StackTemplate**假设在**push( )**函数中有一些对**T**的赋值运算。可以说，模板对于它可以包含的类型“隐含着一个界面”。

表述它的另一种方法是认为模板为C++提供了一种*弱类型*（*weak typing*）机制，C++通常是强类型语言。弱类型不是坚持一个类型是某个可接受的确切类型，而是只要求它想调用的成员函数对于一个特定对象可用就行了。这样，弱类型代码适用于可以接受这些成员函数调用的任何对象，因此更灵活[⊖]。 701

这里有一个用于检测模板的修正过的例子：

```
//: C16:StackTemplateTest.cpp
// Test simple stack template
//{L} fibonacci
#include "fibonacci.h"
#include "StackTemplate.h"
#include <iostream>
#include <fstream>
#include <string>
using namespace std;

int main() {
  StackTemplate<int> is;
  for(int i = 0; i < 20; i++)
    is.push(fibonacci(i));
  for(int k = 0; k < 20; k++)
    cout << is.pop() << endl;
  ifstream in("StackTemplateTest.cpp");
  assure(in, "StackTemplateTest.cpp");
  string line;
  StackTemplate<string> strings;
  while(getline(in, line))
    strings.push(line);
  while(strings.size() > 0)
    cout << strings.pop() << endl;
} ///:~
```

惟一的不同是在实例**is**的创建中。在这个模板参数列表中，我们指明了栈和迭代器应当存

⊖ 在Smalltalk和Python语言中的所有方法都是弱类型，所以这些语言不需要模板机制。实际上，我们得到了无模板的模板。

[702] 放的类型。为了显示这个模板的一般性，我们还创建了一个**StackTemplate**来存放**string**。这是通过读入来自源代码文件的代码行来检测的。

16.3.3 模板中的常量

模板参数并不局限于类定义的类型，可以使用编译器内置类型。这些参数值在编译期间变成模板的特定示例的常量。我们甚至可以对这些参数使用默认值。下面的例子允许我们在实例化时设置**Array**类的长度,并且还可以提供默认值。

```
//: C16:Array3.cpp
// Built-in types as template arguments
#include "../require.h"
#include <iostream>
using namespace std;

template<class T, int size = 100>
class Array {
  T array[size];
public:
  T& operator[](int index) {
    require(index >= 0 && index < size,
      "Index out of range");
    return array[index];
  }
  int length() const { return size; }
};

class Number {
  float f;
public:
  Number(float ff = 0.0f) : f(ff) {}
  Number& operator=(const Number& n) {
    f = n.f;
    return *this;
  }
  operator float() const { return f; }
  friend ostream&
    operator<<(ostream& os, const Number& x) {
      return os << x.f;
[703] }
};

template<class T, int size = 20>
class Holder {
  Array<T, size>* np;
public:
  Holder() : np(0) {}
  T& operator[](int i) {
    require(0 <= i && i < size);
    if(!np) np = new Array<T, size>;
    return np->operator[](i);
  }
  int length() const { return size; }
  ~Holder() { delete np; }
```

```
};

int main() {
  Holder<Number> h;
  for(int i = 0; i < 20; i++)
    h[i] = i;
  for(int j = 0; j < 20; j++)
    cout << h[j] << endl;
} ///:~
```

如前所述，**Array**是被检查的对象数组，并且防止下标越界。类**Holder**很像**Array**，只是它有一个指向**Array**的指针，而不是指向类型**Array**的嵌入对象。该指针在构造函数中不被初始化，而是推迟到第一次访问时。这称为*懒惰初始化*（*lazy initialization*）。如果创造大量的对象，但不访问每一个对象，为了节省存储，可以用懒惰初始化技术。

注意，在这两个模板中，**size**值决不存放在类中，但对它的使用就如同是成员函数中的数据成员。

704

16.4 作为模板的Stash和Stack

贯穿本书反复讨论的**Stash**和**Stack**容器类面临的“所有权”问题，源于我们还不能确切地知道这些容器包含的是什么类型。最近出现的是**Object**的**Stack**容器，这在第15章最后的**OStackTest.cpp**中已经看到了。

如果客户程序员不显式地移去所有指向存放在容器中对象的指针，则容器应当能正确地删除这些指针。这就是说，容器“拥有”不被移走的对象，负责清除它们。问题是这个清除要求关于对象类型的知识，而创造一个一般性的容器类不要求关于对象类型的知识。然而，利用模板，我们可以编写不知道对象类型的代码，并且对于我们希望包含的每种类型，我们可以更容易地实例化这个容器的新版本。个别的已实例化的容器不知道它们保存的对象的类型，但能调用正确的析构函数（假定在典型情况下包含多态性，这时已提供了虚析构函数）。

对于**Stack**，结果很简单，因为所有成员函数都能合理地内联。

```
//: C16:TStack.h
// The Stack as a template
#ifndef TSTACK_H
#define TSTACK_H

template<class T>
class Stack {
  struct Link {
    T* data;
    Link* next;
    Link(T* dat, Link* nxt):
      data(dat), next(nxt) {}
  }* head;
public:
  Stack() : head(0) {}
  ~Stack(){
    while(head)
      delete pop();
  }
  void push(T* dat) {
```

705

```
    head = new Link(dat, head);
  }
  T* peek() const {
    return head ? head->data : 0;
  }
  T* pop(){
    if(head == 0) return 0;
    T* result = head->data;
    Link* oldHead = head;
    head = head->next;
    delete oldHead;
    return result;
  }
};
#endif // TSTACK_H ///:~
```

如果将它与第15章最后的例子**OStack.h**相比较，我们可以看到**Stack**实际上相同，除了**Object**已经用**T**替换以外。测试程序也近似相同，除了消除了从**string**和**Object**多重继承的必要性（甚至对于**Object**本身的需要）以外。现在，没有**MyString**类宣布它的销毁，所以增加了一个小的新类来显示**Stack**容器清除它的对象。

```
//: C16:TStackTest.cpp
//{T} TStackTest.cpp
#include "TStack.h"
#include "../require.h"
#include <fstream>
#include <iostream>
#include <string>
using namespace std;

class X {
public:
706   virtual ~X() { cout << "~X " << endl; }
};

int main(int argc, char* argv[]) {
  requireArgs(argc, 1); // File name is argument
  ifstream in(argv[1]);
  assure(in, argv[1]);
  Stack<string> textlines;
  string line;
  // Read file and store lines in the Stack:
  while(getline(in, line))
    textlines.push(new string(line));
  // Pop some lines from the stack:
  string* s;
  for(int i = 0; i < 10; i++) {
    if((s = (string*)textlines.pop())==0) break;
    cout << *s << endl;
    delete s;
  } // The destructor deletes the other strings.
  // Show that correct destruction happens:
  Stack<X> xx;
  for(int j = 0; j < 10; j++)
    xx.push(new X);
} ///:~
```

X的析构函数是虚的，这里不是因为需要如此，而是因为xx稍后能用来存放从X派生的对象。

注意，对于string和对于X创造不同种类的Stack是多么容易。由于模板的存在，我们可以得到两方面的好处——即Stack类容易使用和正确清除。

16.4.1 模板化的指针Stash

重新组织PStash代码成为模板并不简单，因为有一些成员函数不应当内联。但是，作为一个模板，这些函数定义仍然存放在头文件中（编译器和连接器处理多定义问题）。代码看上去非常类似于通常的PStash，除了增量的大小（由inflate( )使用）已经被模板化为具有默认值的无类参数以外，所以这个增量的大小能在实例化时修改（注意，这意味着增量大小是固定的，尽管有人会争辩增量大小应当在对象的整个生命期中都是可以改变的）。 707

```
//: C16:TPStash.h
#ifndef TPSTASH_H
#define TPSTASH_H

template<class T, int incr = 10>
class PStash {
  int quantity; // Number of storage spaces
  int next; // Next empty space
  T** storage;
  void inflate(int increase = incr);
public:
  PStash() : quantity(0), next(0), storage(0) {}
  ~PStash();
  int add(T* element);
  T* operator[](int index) const; // Fetch
  // Remove the reference from this PStash:
  T* remove(int index);
  // Number of elements in Stash:
  int count() const { return next; }
};

template<class T, int incr>
int PStash<T, incr>::add(T* element) {
  if(next >= quantity)
    inflate(incr);
  storage[next++] = element;
  return(next - 1); // Index number
}

// Ownership of remaining pointers:
template<class T, int incr>
PStash<T, incr>::~PStash() {
  for(int i = 0; i < next; i++) {
    delete storage[i]; // Null pointers OK
    storage[i] = 0; // Just to be safe
  }
  delete []storage;
}

template<class T, int incr>
T* PStash<T, incr>::operator[](int index) const {
```

708

```
  require(index >= 0,
    "PStash::operator[] index negative");
  if(index >= next)
    return 0; // To indicate the end
  require(storage[index] != 0,
    "PStash::operator[] returned null pointer");
  // Produce pointer to desired element:
  return storage[index];
}

template<class T, int incr>
T* PStash<T, incr>::remove(int index) {
  // operator[] performs validity checks:
  T* v = operator[](index);
  // "Remove" the pointer:
  if(v != 0) storage[index] = 0;
  return v;
}

template<class T, int incr>
void PStash<T, incr>::inflate(int increase) {
  const int psz = sizeof(T*);
  T** st = new T*[quantity + increase];
  memset(st, 0, (quantity + increase) * psz);
  memcpy(st, storage, quantity * psz);
  quantity += increase;
  delete []storage; // Old storage
  storage = st; // Point to new memory
}
#endif // TPSTASH_H ///:~
```

在这里使用的默认增量大小是很小的，以便保证能发生对**inflate( )**的调用。我们采用的这种方法可以确保工作正确。

为了测试模板化的**PStash**的控制权，下面的类将报告自身的创建和销毁，并保证被创建的对象都能被销毁。**AutoCounter**只允许它的类型的对象在栈上创建。

```
//: C16:AutoCounter.h
#ifndef AUTOCOUNTER_H
#define AUTOCOUNTER_H
#include "../require.h"
#include <iostream>
#include <set> // Standard C++ Library container
#include <string>

class AutoCounter {
  static int count;
  int id;
  class CleanupCheck {
    std::set<AutoCounter*> trace;
  public:
    void add(AutoCounter* ap) {
      trace.insert(ap);
    }
    void remove(AutoCounter* ap) {
      require(trace.erase(ap) == 1,
```

```
        "Attempt to delete AutoCounter twice");
    }
    ~CleanupCheck() {
      std::cout << "~CleanupCheck()"<< std::endl;
      require(trace.size() == 0,
       "All AutoCounter objects not cleaned up");
    }
  };
  static CleanupCheck verifier;
  AutoCounter() : id(count++) {
    verifier.add(this); // Register itself
    std::cout << "created[" << id << "]"
              << std::endl;
  }
  // Prevent assignment and copy-construction:
  AutoCounter(const AutoCounter&);
  void operator=(const AutoCounter&);
public:
  // You can only create objects with this:
  static AutoCounter* create() {
    return new AutoCounter();
  }
  ~AutoCounter() {
    std::cout << "destroying[" << id
              << "]" << std::endl;
    verifier.remove(this);
  }
  // Print both objects and pointers:
  friend std::ostream& operator<<(
    std::ostream& os, const AutoCounter& ac){
    return os << "AutoCounter " << ac.id;
  }
  friend std::ostream& operator<<(
    std::ostream& os, const AutoCounter* ac){
    return os << "AutoCounter " << ac->id;
  }
};
#endif // AUTOCOUNTER_H ///:~
```

710

AutoCounter类做两件事。第一，它继续对**AutoCounter**的每个实例编号：这个编号的值保存在**id**中，并且使用**static**数据成员**count**来生成这个编号。

第二，更复杂，嵌套类**CleanupCheck**的一个静态实例（称为**verifier**）跟踪被创建和销毁的所有的**AutoCounter**对象，如果程序员没有完全清除它们，它就向程序员报告（也就是假定这里有一个内存泄漏）。这个行为是使用标准C++类库中的**set**类完成的，这是良好设计的模板如何能方便使用的极好例子（在本书的第2卷，我们可以学习C++标准类库中的所有容器）。

set类是按照它所包含的类型建立模板的；在这里，它被实例化为包含**AutoCounter**指针的实例。一个**set**只允许每个不同对象的一个实例被添加；在**add( )**中，这由**set::insert( )**函数完成。如果我们正在试图添加先前已经添加过的内容，**insert( )**就用它的返回值通知我们。然而，因为对象地址被添加，所以我们可以依靠C++保证所有对象有惟一的地址。

在**remove( )**中，使用**set::erase( )**从**set**中移出**AutoCounter**指针。返回值告诉我们这个元素的多少个实例被移出。在这种情况下，我们只希望返回0或1。如果返回值是0，表示这个对

象已经从**set**中删除，并且这是第二次试图删除它，这是一个程序设计错误，可以通过
711 **require( )**报告这个错误。

CleanupCheck的析构函数最后检查**set**的长度是否确实是0。如果是0，表示它的所有对象都已经被完全清除。如果不是0，说明有内存泄漏，可以通过**require( )**报告这个错误。

AutoCounter的构造函数和析构函数用**verifier**对象注册和注销它们自己。注意，构造函数、拷贝构造函数以及赋值运算符都是**private**的，所以创建对象的惟一方法是用**static create( )**成员函数，这是*factory*的一个简单例子，它保证所有的对象都在堆上创建，所以**verifier**对于赋值和拷贝构造不会混淆。

因为所有的成员函数都是内联的，所以使用实现文件的惟一原因是为了包含静态数据成员的定义。

```
//: C16:AutoCounter.cpp {O}
// Definition of static class members
#include "AutoCounter.h"
AutoCounter::CleanupCheck AutoCounter::verifier;
int AutoCounter::count = 0;
///:~
```

利用手边的**AutoCounter**，我们现在可以测试**PStash**的功能。下面的例子不仅表明**PStash**析构函数清除了它现在所拥有的对象，而且还表明**AutoCounter**类如何检测到还没有被清除的对象。

```
//: C16:TPStashTest.cpp
//{L} AutoCounter
#include "AutoCounter.h"
#include "TPStash.h"
#include <iostream>
#include <fstream>
using namespace std;

int main() {
  PStash<AutoCounter> acStash;
712 for(int i = 0; i < 10; i++)
    acStash.add(AutoCounter::create());
  cout << "Removing 5 manually:" << endl;
  for(int j = 0; j < 5; j++)
    delete acStash.remove(j);
  cout << "Remove two without deleting them:"
       << endl;
  // ... to generate the cleanup error message.
  cout << acStash.remove(5) << endl;
  cout << acStash.remove(6) << endl;
  cout << "The destructor cleans up the rest:"
       << endl;
  // Repeat the test from earlier chapters:
  ifstream in("TPStashTest.cpp");
  assure(in, "TPStashTest.cpp");
  PStash<string> stringStash;
  string line;
  while(getline(in, line))
    stringStash.add(new string(line));
  // Print out the strings:
```

```
  for(int u = 0; stringStash[u]; u++)
    cout << "stringStash[" << u << "] = "
         << *stringStash[u] << endl;
} ///:~
```

当从**PStash**中移出**AutoCounter**元素5和元素6时，它们就变成了调用者的责任，但是因为调用者没有清除它们，所以就引起了内存泄漏，它们随后在运行时被**AutoCounter**检测到。

当我们运行这个程序时，会看到错误信息不像希望的那样详细。如果在系统中使用**AutoCounter**中所描述的方案去发现内存泄漏，也许希望打印出关于未被清除对象的更详细的信息。本书的第2卷表明了做这件事情的更好方法。

16.5 打开和关闭所有权

让我们回到所有权问题上来。以值包含对象的容器通常无需担心所有权问题，因为它们清晰地拥有它们所包含的对象。但是，如果容器内包含指向对象的指针（这种情况在C++中 713
相当普遍，尤其在多态情况下），而这些指针很可能用于程序的其他地方，那么删除该指针指向的对象会导致在程序的其他地方的指针对已销毁的对象进行引用。为了避免上述情况，在设计和使用容器时必须考虑所有权问题。

许多程序都比这个问题更简单，并且不会遇到所有权问题：一个容器所包含的指针指向仅由这个容器使用的那些对象。在这种情况下，所有权简单而直观：该容器拥有它自己的对象。

处理所有权问题的最好方法是由客户程序员来选择。这常常通过构造函数的一个参数来完成，它默认地指明所有权（简单情况）。另外还有“读取”和“设置”函数用来查看和修正容器的所有权。如果容器内有用于删除对象的函数，容器所有权的状态通常会影响这个删除，所以我们还可以找到在删除函数中控制销毁的选项。我们可以对容器中的每一个成员添加所有权数据，这样每个位置都知道它是否需要被销毁；这是一个引用计数的变体，在这里是容器而不是对象知道所指对象的引用数。

```
//: C16:OwnerStack.h
// Stack with runtime conrollable ownership
#ifndef OWNERSTACK_H
#define OWNERSTACK_H

template<class T> class Stack {
  struct Link {
    T* data;
    Link* next;
    Link(T* dat, Link* nxt)
      : data(dat), next(nxt) {}
  }* head;
  bool own;
public:
  Stack(bool own = true) : head(0), own(own) {}
  ~Stack();
  void push(T* dat) {
    head = new Link(dat,head);
  }                                                    714
  T* peek() const {
    return head ? head->data : 0;
```

```
  }
  T* pop();
  bool owns() const { return own; }
  void owns(bool newownership) {
    own = newownership;
  }
  // Auto-type conversion: true if not empty:
  operator bool() const { return head != 0; }
};

template<class T> T* Stack<T>::pop() {
  if(head == 0) return 0;
  T* result = head->data;
  Link* oldHead = head;
  head = head->next;
  delete oldHead;
  return result;
}

template<class T> Stack<T>::~Stack() {
  if(!own) return;
  while(head)
    delete pop();
}
#endif // OWNERSTACK_H ///:~
```

默认行为是让容器去销毁它的对象，但我们可以通过修改构造函数的参数或者使用**owns( )**读/写成员函数来改变这个行为。

正如我们可能看到的大多数模板那样，整个实现包含在头文件中。下面是一个检验所有权能力的小测试。

```
//: C16:OwnerStackTest.cpp
//{L} AutoCounter
#include "AutoCounter.h"
#include "OwnerStack.h"
#include "../require.h"
#include <iostream>
#include <fstream>
#include <string>
using namespace std;

int main() {
  Stack<AutoCounter> ac; // Ownership on
  Stack<AutoCounter> ac2(false); // Turn it off
  AutoCounter* ap;
  for(int i = 0; i < 10; i++) {
    ap = AutoCounter::create();
    ac.push(ap);
    if(i % 2 == 0)
      ac2.push(ap);
  }
  while(ac2)
    cout << ac2.pop() << endl;
  // No destruction necessary since
  // ac "owns" all the objects
} ///:~
```

ac2对象不拥有放在它里面的对象，因而**ac**就是对所有权员有责任的“主”容器。在容器生命期内如果希望改变一个容器拥有它的对象，我们可以用**owns( )**做这件事。

我们还有可能改变所有权的粒度，使得它以“object-by-object”为基础，但是这将可能会使所有权问题的解决更趋复杂。

16.6 以值存放对象

实际上，如果我们没有模板，那么在一个一般的容器内创建对象的一个拷贝是一个复杂的问题。使用模板，事情就相对简单了，只要说我们存放对象而不是指针就行了。

```
//: C16:ValueStack.h
// Holding objects by value in a Stack
#ifndef VALUESTACK_H
#define VALUESTACK_H
#include "../require.h"
template<class T, int ssize = 100>
class Stack {
  // Default constructor performs object
  // initialization for each element in array:
  T stack[ssize];
  int top;
public:
  Stack() : top(0) {}
  // Copy-constructor copies object into array:
  void push(const T& x) {
    require(top < ssize, "Too many push()es");
    stack[top++] = x;
  }
  T peek() const { return stack[top]; }
  // Object still exists when you pop it;
  // it just isn't available anymore:
  T pop() {
    require(top > 0, "Too many pop()s");
    return stack[--top];
  }
};
#endif // VALUESTACK_H ///:~
```

716

用于被包含对象的拷贝构造函数通过按值传递和返回对象来做大部分工作。在**push( )**内，对象在**Stack**数组上的对象存储是用**T::operator=**完成的。为了保证工作，一个称为**SelfCounter**的类将跟踪对象创建和拷贝构造。

```
//: C16:SelfCounter.h
#ifndef SELFCOUNTER_H
#define SELFCOUNTER_H
#include "ValueStack.h"
#include <iostream>

class SelfCounter {
  static int counter;
  int id;
public:
  SelfCounter() : id(counter++) {
```

```
    std::cout << "Created: " << id << std::endl;
  }
717 SelfCounter(const SelfCounter& rv) : id(rv.id){
    std::cout << "Copied: " << id << std::endl;
  }
  SelfCounter operator=(const SelfCouncer& rv) {
    std::cout << "Assigned " << rv.ld << " to "
              << id << std::endl;
    return *this;
  }
  ~SelfCounter() {
    std::cout << "Destroyed: "<< id << std::endl;
  }
  friend std::ostream& operator<<(
    std::ostream& os, const SelfCounter& sc){
    return os << "SelfCounter: " << sc.id;
  }
};
#endif // SELFCOUNTER_H ///:~

//: C16:SelfCounter.cpp {O}
#include "SelfCounter.h"
int SelfCounter::counter = 0; ///:~

//: C16:ValueStackTest.cpp
//{L} SelfCounter
#include "ValueStack.h"
#include "SelfCounter.h"
#include <iostream>
using namespace std;

int main() {
  Stack<SelfCounter> sc;
  for(int i = 0; i < 10; i++)
    sc.push(SelfCounter());
  // OK to peek(), result is a temporary:
  cout << sc.peek() << endl;
  for(int k = 0; k < 10; k++)
    cout << sc.pop() << endl;
} ///:~
```

当创建一个**Stack**容器时，对于数组中的每个对象调用被包含对象的默认构造函数。最初将看到100个**SelfCounter**对象被创建，但是，这只是这个数组的初始化。这样的代价可能有
718 点昂贵，但是在像这样的简单设计中没有办法。如果允许**Stack**的规模动态增长，让它更一般化，就会出现更复杂的情况，因为在上面显示的实现中会包括：创建一个新的（更大）的数组，拷贝老的数组给新的数组，销毁这个老的数组（事实上，这是标准C++库函数**vector**类所做的事情）。

16.7 迭代器简介

迭代器（*iterator*）是一个对象，它在其他对象的容器上遍历，每次选择它们中的一个，不需要提供对这个容器的实现的直接访问。迭代器提供了一种访问元素的标准方法，无论容器是否提供了直接访问元素的方法。迭代器常常与容器类联合使用，而且迭代器在标准C++

容器的设计和使用中是一个基本概念，这方面的知识在本书的第2卷（可从*www.BruceEckel.com*下载）中有全面的描述。迭代器也是一种设计模式（*design pattern*），这是第2卷中的有一章的主题。

在许多情况下，迭代器是一个“灵巧指针”；并且事实上，我们会注意到：迭代器通常模仿大多数指针的运算。然而，不同的是，迭代器的设计更安全，所以数组越界的可能性更小（或者说，如果有数组越界，就会更早被发现）。

考虑本章的第一个例子，这里增加了一个简单的迭代器。

```
//: C16:IterIntStack.cpp
// Simple integer stack with iterators
//{L} fibonacci
#include "fibonacci.h"
#include "../require.h"
#include <iostream>
using namespace std;

class IntStack {
  enum { ssize = 100 };
  int stack[ssize];
  int top;
public:
  IntStack() : top(0) {}
  void push(int i) {
    require(top < ssize, "Too many push()es");
    stack[top++] = i;
  }
  int pop() {
    require(top > 0, "Too many pop()s");
    return stack[--top];
  }
  friend class IntStackIter;
};

// An iterator is like a "smart" pointer:
class IntStackIter {
  IntStack& s;
  int index;
public:
  IntStackIter(IntStack& is) : s(is), index(0) {}
  int operator++() { // Prefix
    require(index < s.top,
      "iterator moved out of range");
    return s.stack[++index];
  }
  int operator++(int) { // Postfix
    require(index < s.top,
      "iterator moved out of range");
    return s.stack[index++];
  }
};

int main() {
  IntStack is;
  for(int i = 0; i < 20; i++)
```

719

```
    is.push(fibonacci(i));
  // Traverse with an iterator:
  IntStackIter it(is);
  for(int j = 0; j < 20; j++)
    cout << it++ << endl;
} ///:~
```

720

创建**IntStackIter**，以便只与**IntStack**一起工作。注意，**IntStackIter**是**IntStack**的友元，这就允许访问**IntStack**的所有私有成员。

像指针一样，**IntStackIter**的工作是遍历**IntStack**，并提取值。在这个简单的例子中，**IntStackIter**只能向前移动（用**operator++**的前缀和后缀形式）。然而，迭代器定义没有限制，而只是有与它一起工作的容器的约束限制。在与迭代器相联系的容器中，迭代器可以用任何方式移动，并且可以通过它修改被包含的值。

习惯上，用构造函数来创建迭代器，并把它与一个容器对象联系，并且在它的生命期中，不把它与不同的容器联系。（迭代器通常是小的和廉价的，所以可以很容易地再做一个。）

使用迭代器，我们可以扫描栈的元素而不用弹出它们，这就像指针遍历数组的元素一样。然而，迭代器知道栈的下层结构，并知道如何遍历栈的元素，所以即便我们正在以“向前移动指针”的方式遍历栈的元素，我们也应该使用迭代器。下面将介绍更多的内容。迭代器的关键是：从一个容器元素移动到下一个元素的复杂过程被抽象为就像一个指针一样。目标是使程序中的每个迭代器都有相同的接口，使得使用这个迭代器的任何代码都不用关心它指向什么，只需要知道它能用同样的方法重新配置所有的迭代器，所以这个迭代器所指向的容器并不重要。用这种方法，我们可以编写更一般性的代码。标准C++库的所有容器和算法都基于迭代器的这一原则。

为了让事情更一般化，最好能说“每一个容器有一个相关的名为**iterator**的类”，但是这引起典型的名字问题。解决的办法是为每个容器增加一个嵌套的**iterator**类（注意，在这种情

721

况下，“**iterator**”以小写字母开始，使得它与标准C++库的风格一致）。下面是**IterIntStack.cpp**，带有一个嵌套的**iterator**。

```
//: C16:NestedIterator.cpp
// Nesting an iterator inside the container
//{L} fibonacci
#include "fibonacci.h"
#include "../require.h"
#include <iostream>
#include <string>
using namespace std;

class IntStack {
  enum { ssize = 100 };
  int stack[ssize];
  int top;
public:
  IntStack() : top(0) {}
  void push(int i) {
    require(top < ssize, "Too many push()es");
    stack[top++] = i;
  }
  int pop() {
```

```
    require(top > 0, "Too many pop()s");
    return stack[--top];
  }
  class iterator;
  friend class iterator;
  class iterator {
    IntStack& s;
    int index;
  public:
    iterator(IntStack& is) : s(is), index(0) {}
    // To create the "end sentinel" iterator:
    iterator(IntStack& is, bool)
      : s(is), index(s.top) {}
    int current() const { return s.stack[index]; }
    int operator++() { // Prefix
      require(index < s.top,
        "iterator moved out of range");
      return s.stack[++index];
    }
    int operator++(int) { // Postfix
      require(index < s.top,                                    722
        "iterator moved out of range");
      return s.stack[index++];
    }
    // Jump an iterator forward
    iterator& operator+=(int amount) {
      require(index + amount < s.top,
        "IntStack::iterator::operator+=() "
        "tried to move out of bounds");
      index += amount;
      return *this;
    }
    // To see if you're at the end:
    bool operator==(const iterator& rv) const {
      return index == rv.index;
    }
    bool operator!=(const iterator& rv) const {
      return index != rv.index;
    }
    friend ostream&
    operator<<(ostream& os, const iterator& it) {
      return os << it.current();
    }
  };
  iterator begin() { return iterator(*this); }
  // Create the "end sentinel":
  iterator end() { return iterator(*this, true);}
};

int main() {
  IntStack is;
  for(int i = 0; i < 20; i++)
    is.push(fibonacci(i));
  cout << "Traverse the whole IntStack\n";
  IntStack::iterator it = is.begin();
  while(it != is.end())
```

```
    cout << it++ << endl;
  cout << "Traverse a portion of the IntStack\n";
  IntStack::iterator
    start = is.begin(), end = is.begin();
  start += 5, end += 15;
  cout << "start = " << start << endl;
  cout << "end = " << end << endl;
  while(start != end)
    cout << start++ << endl;
} ///:~
```

723

当创建一个嵌套**friend**类的时候，我们必须经过首先声明这个类的名称，然后声明它是友元，最后定义这个类的过程。否则，编译器将会产生混淆。

我们在迭代器中增加了一些新的手法。**current( )** 成员函数产生容器中的由迭代器当前选择的元素。我们可以用**operator+=**使迭代器向前“跳跃”任意个元素。而且，我们还会看到两个重载运算符：**==**和**!=**，它们将比较两个迭代器。它们能比较任意两个**IntStack::iterator**，但是它们的最初意图是测试这个迭代器是否已经到达了序列的终点，采用“实际的”标准C++库迭代器所用的相同方法。其思想是，两个迭代器定义了一个范围，第一个迭代器指向第一个元素，第二个迭代器指向最后一个元素后面的位置。如果希望遍历由这两个迭代器所定义的范围，可以写为如下形式：

```
while(start != end)
  cout << start++ << endl;
```

这里，**start**和**end**是在这个范围内的两个迭代器。注意，**end**迭代器并不反向引用，只是告诉我们已经到了这个范围的终点，我们称之为“*终止哨兵*”（*end sentinel*）。因而它代表“终点后面的一个”。

大多数情况下我们希望在容器中遍历整个序列，所以这个容器需要某种方法产生表示这个序列的开始和终止哨兵的迭代器。在此，就像在标准C++库中一样，这些迭代器由容器的成员函数**begin( )** 和**end( )**产生。**begin( )**使用第一个迭代器构造函数，它默认指向这个容器的开始（这就是压入这个栈的第一个元素）。然而，第二个构造函数，由**end( )**使用，对于创建终止哨兵迭代器是必需的。“在终点”的意思是指向这个栈的顶部，因为**top**允许指向下一个可用的但是尚未使用的位置。这个迭代器构造函数采用第二个类型为**bool**的参数，它是哑元，

724

以区别两个构造函数。

在**main( )**中再次使用斐波纳契数来填充**IntStack**，用迭代器遍历整个**IntStack**并且还遍历序列的一个小范围。

当然，下一步是通过对它所包含的类型模板化来让代码一般化，所以不是强迫仅能存放**int**，而是可以存放任何类型。

```
//: C16:IterStackTemplate.h
// Simple stack template with nested iterator
#ifndef ITERSTACKTEMPLATE_H
#define ITERSTACKTEMPLATE_H
#include "../require.h"
#include <iostream>

template<class T, int ssize = 100>
class StackTemplate {
```

```
  T stack[ssize];
  int top;
public:
  StackTemplate() : top(0) {}
  void push(const T& i) {
    require(top < ssize, "Too many push()es");
    stack[top++] = i;
  }
  T pop() {
    require(top > 0, "Too many pop()s");
    return stack[--top];
  }
  class iterator; // Declaration required
  friend class iterator; // Make it a friend
  class iterator { // Now define it
    StackTemplate& s;
    int index;
  public:
    iterator(StackTemplate& st): s(st),index(0){}
    // To create the "end sentinel" iterator:
    iterator(StackTemplate& st, bool)
      : s(st), index(s.top) {}
    T operator*() const { return s.stack[index];}
    T operator++() { // Prefix form
      require(index < s.top,
        "iterator moved out of range");
      return s.stack[++index];
    }
    T operator++(int) { // Postfix form
      require(index < s.top,
        "iterator moved out of range");
      return s.stack[index++];
    }
    // Jump an iterator forward
    iterator& operator+=(int amount) {
      require(index + amount < s.top,
        " StackTemplate::iterator::operator+=() "
        "tried to move out of bounds");
      index += amount;
      return *this;
    }
    // To see if you're at the end:
    bool operator==(const iterator& rv) const {
      return index == rv.index;
    }
    bool operator!=(const iterator& rv) const {
      return index != rv.index;
    }
    friend std::ostream& operator<<(
      std::ostream& os, const iterator& it) {
      return os << *it;
    }
  };
  iterator begin() { return iterator(*this); }
  // Create the "end sentinel":
  iterator end() { return iterator(*this, true);}
```

725

```
};
#endif // ITERSTACKTEMPLATE_H ///:~
```

可以看到，从正规类到模板的转换是适度透明的。首先创建和调试一个普通类，然后让它成为模板，一般认为这种方法比一开始就创建模板更容易。

注意，不是只写：

```
friend iterator; // Make it a friend
```

这段代码是：

726
```
friend class iterator; // Make it a friend
```

这是重要的，因为名字“iterator”已经在一个范围内，来自一个被包含的文件。

不是用**current( )**成员函数，而是**iterator**有一个**operator***，用来选择当前的元素，这使**iterator**看上去更像一个指针，这是一个普通的习惯。

下面是一个用来测试模板的修改过的例子：

```
//: C16:IterStackTemplateTest.cpp
//{L} fibonacci
#include "fibonacci.h"
#include "IterStackTemplate.h"
#include <iostream>
#include <fstream>
#include <string>
using namespace std;

int main() {
  StackTemplate<int> is;
  for(int i = 0; i < 20; i++)
    is.push(fibonacci(i));
  // Traverse with an iterator:
  cout << "Traverse the whole StackTemplate\n";
  StackTemplate<int>::iterator it = is.begin();
  while(it != is.end())
    cout << it++ << endl;
  cout << "Traverse a portion\n";
  StackTemplate<int>::iterator
    start = is.begin(), end = is.begin();
  start += 5, end += 15;
  cout << "start = " << start << endl;
  cout << "end = " << end << endl;
  while(start != end)
    cout << start++ << endl;
  ifstream in("IterStackTemplateTest.cpp");
  assure(in, "IterStackTemplateTest.cpp");
  string line;
  StackTemplate<string> strings;
  while(getline(in, line))
    strings.push(line);
727
  StackTemplate<string>::iterator
    sb = strings.begin(), se = strings.end();
  while(sb != se)
    cout << sb++ << endl;
} ///:~
```

迭代器的第一个应用只移动它从开始到最后（可以看到终止哨兵工作正常）。在第二个应用中，我们可以看到迭代器如何允许我们容易地指定元素的范围（在标准C++库中，容器和迭代器随处使用范围的概念）。重载**operator+=**移动**start**和**end**迭代器到**is**中元素范围的中间位置，打印出这些元素。注意，在输出中，终止哨兵不在范围内，这样，它可以是范围终点后面的一个，可以让程序员知道他已经越过了终点，但是，不反向引用终止哨兵，否则就相当于反向引用空指针。（在**StackTemplate::iterator**中我已经做了防护，但是在标准C++库中的容器和迭代器中，出于效率的原因，没有这样的代码，所以必须注意。）

最后，为了验证**StackTemplate**与类对象一起工作，采用一个**string**的实例，它用源代码文件中的行填充这些字串，然后打印出它们。

16.7.1 带有迭代器的栈

重复具有动态长度**Stack**类的过程，这是贯穿本书的例子。这里**Stack**类带有一个嵌套的迭代器。

```
//: C16:TStack2.h
// Templatized Stack with nested iterator
#ifndef TSTACK2_H
#define TSTACK2_H

template<class T> class Stack {
  struct Link {
    T* data;
    Link* next;
    Link(T* dat, Link* nxt)
      : data(dat), next(nxt) {}
  }* head;
public:
  Stack() : head(0) {}
  ~Stack();
  void push(T* dat) {
    head = new Link(dat, head);
  }
  T* peek() const {
    return head ? head->data : 0;
  }
  T* pop();
  // Nested iterator class:
  class iterator; // Declaration required
  friend class iterator; // Make it a friend
  class iterator { // Now define it
    Stack::Link* p;
  public:
    iterator(const Stack<T>& tl) : p(tl.head) {}
    // Copy-constructor:
    iterator(const iterator& tl) : p(tl.p) {}
    // The end sentinel iterator:
    iterator() : p(0) {}
    // operator++ returns boolean indicating end:
    bool operator++() {
      if(p->next)
        p = p->next;
```

728

```
      else p = 0; // Indicates end of list
      return bool(p);
    }
    bool operator++(int) { return operator++(); }
    T* current() const {
      if(!p) return 0;
      return p->data;
    }
    // Pointer dereference operator:
    T* operator->() const {
      require(p != 0,
        "PStack::iterator::operator->returns 0");
      return current();
    }
    T* operator*() const { return current(); }
    // bool conversion for conditional test:
    operator bool() const { return bool(p); }
    // Comparison to test for end:
    bool operator==(const iterator&) const {
729   return p == 0;
    }
    bool operator!=(const iterator&) const {
      return p != 0;
    }
  };
  iterator begin() const {
    return iterator(*this);
  }
  iterator end() const { return iterator(); }
};

template<class T> Stack<T>::~Stack() {
  while(head)
    delete pop();
}

template<class T> T* Stack<T>::pop() {
  if(head == 0) return 0;
  T* result = head->data;
  Link* oldHead = head;
  head = head->next;
  delete oldHead;
  return result;
}
#endif // TSTACK2_H ///:~
```

我们已经注意到，这个类已经被修改以支持所有权，它能工作是因为这个类知道确切类型（或者至少知道基本类型，这是基于使用虚构造函数的假设而工作的）。对于容器的默认是销毁它的对象，但是我们要负责处理我们**pop( )**的任何指针。

迭代器是简单的，体积非常小，即单个指针的大小。当创建一个迭代器时，它被初始化为指向链表的头，只能沿着链表向前递增。如果希望指向起点之后，就创建一个新迭代器，如果希望记住表中的一点，就从已存在的迭代器中创建一个新迭代器，指向这一点（使用迭代器的拷贝构造函数）。

为了对由迭代器指向的对象调用函数，我们可以使用**current( )**函数、**operator***和指针反向引用**operator->**（迭代器中的一个共同点）。后者的实现看上去与**current( )**一样，因为它返回一个指向当前对象的指针，但是实际上不同，因为这个指针反向引用运算符完成反向引用的外层（参见第12章）。 730

iterator类遵循前面例子中的形式。**class iterator**嵌套在容器类中，它包含构造函数，可以创建指向容器中一个元素的一个迭代器和一个“终止哨兵”迭代器，并且容器类有用来产生这些迭代器的**begin( )**和**end( )**方法。（当我们对标准C++库的学习更加深入之后，就会看到：在这里用的名字**iterator**、**begin( )**和**end( )**已经明确地被推举为标准容器类。在本章的最后将会看到，使用这些容器类就像使用标准C++库容器类一样。）

全部实现都包含在头文件中，所以这里没有单独的**cpp**文件。下面是用来检验迭代器的一个小测试：

```
//: C16:TStack2Test.cpp
#include "TStack2.h"
#include "../require.h"
#include <iostream>
#include <fstream>
#include <string>
using namespace std;

int main() {
  ifstream file("TStack2Test.cpp");
  assure(file, "TStack2Test.cpp");
  Stack<string> textlines;
  // Read file and store lines in the Stack:
  string line;
  while(getline(file, line))
    textlines.push(new string(line));
  int i = 0;
  // Use iterator to print lines from the list:
  Stack<string>::iterator it = textlines.begin();
  Stack<string>::iterator* it2 = 0;
  while(it != textlines.end()) {
    cout << it->c_str() << endl;
    it++;
    if(++i == 10) // Remember 10th line
      it2 = new Stack<string>::iterator(it);
  }
  cout << (*it2)->c_str() << endl;
  delete it2;
} ///:~
```

731

Stack是一个示例，用来存放**string**对象，并用来自一个文件的行填充。然后创建一个迭代器，用于遍历这个序列。通过从第一个迭代器拷贝构造第二个迭代器，来记住第10行，然后打印这一行，并且销毁动态创建的迭代器。这里，使用动态对象创建来控制该对象的生命期。

16.7.2 带有迭代器的PStash

对于大多数容器类，有迭代器是有意义的。这里对**PStash**类添加一个迭代器：

```
//: C16:TPStash2.h
// Templatized PStash with nested iterator
#ifndef TPSTASH2_H
#define TPSTASH2_H
#include "../require.h"
#include <cstdlib>

template<class T, int incr = 20>
class PStash {
  int quantity;
  int next;
  T** storage;
  void inflate(int increase = incr);
public:
  PStash() : quantity(0), storage(0), next(0) {}
  ~PStash();
  int add(T* element);
  T* operator[](int index) const;
  T* remove(int index);
  int count() const { return next; }
  // Nested iterator class:
  class iterator; // Declaration required
  friend class iterator; // Make it a friend
  class iterator { // Now define it
732 PStash& ps;
  int index;
public:
  iterator(PStash& pStash)
    : ps(pStash), index(0) {}
  // To create the end sentinel:
  iterator(PStash& pStash, bool)
    : ps(pStash), index(ps.next) {}
  // Copy-constructor:
  iterator(const iterator& rv)
    : ps(rv.ps), index(rv.index) {}
  iterator& operator=(const iterator& rv) {
    ps = rv.ps;
    index = rv.index;
    return *this;
  }
  iterator& operator++() {
    require(++index <= ps.next,
      "PStash::iterator::operator++ "
      "moves index out of bounds");
    return *this;
  }
  iterator& operator++(int) {
    return operator++();
  }
  iterator& operator--() {
    require(--index >= 0,
      "PStash::iterator::operator-- "
      "moves index out of bounds");
    return *this;
  }
  iterator& operator--(int) {
```

```
    return operator--();
  }
  // Jump interator forward or backward:
  iterator& operator+=(int amount) {
    require(index + amount < ps.next &&
      index + amount >= 0,
      "PStash::iterator::operator+= "
      "attempt to index out of bounds");
    index += amount;
    return *this;
  }
  iterator& operator-=(int amount) {
    require(index - amount < ps.next &&
      index - amount >= 0,
        "PStash::iterator::operator-= "
        "attempt to index out of bounds");
      index -= amount;
      return *this;
    }
    // Create a new iterator that's moved forward
    iterator operator+(int amount) const {
      iterator ret(*this);
      ret += amount; // op+= does bounds check
      return ret;
    }
    T* current() const {
      return ps.storage[index];
    }
    T* operator*() const { return current(); }
    T* operator->() const {
      require(ps.storage[index] != 0,
        "PStash::iterator::operator->returns 0");
      return current();
    }
    // Remove the current element:
    T* remove(){
      return ps.remove(index);
    }
    // Comparison tests for end:
    bool operator==(const iterator& rv) const {
      return index == rv.index;
    }
    bool operator!=(const iterator& rv) const {
      return index != rv.index;
    }
  };
  iterator begin() { return iterator(*this); }
  iterator end() { return iterator(*this, true);}
};

// Destruction of contained objects:
template<class T, int incr>
PStash<T, incr>::~PStash() {
  for(int i = 0; i < next; i++) {
    delete storage[i]; // Null pointers OK
    storage[i] = 0; // Just to be safe
```

733

```
  }
  delete []storage;
}
template<class T, int incr>
int PStash<T, incr>::add(T* element) {
  if(next >= quantity)
    inflate();
  storage[next++] = element;
  return(next - 1); // Index number
}

template<class T, int incr> inline
T* PStash<T, incr>::operator[](int index) const {
  require(index >= 0,
    "PStash::operator[] index negative");
  if(index >= next)
    return 0; // To indicate the end
  require(storage[index] != 0,
    "PStash::operator[] returned null pointer");
  return storage[index];
}

template<class T, int incr>
T* PStash<T, incr>::remove(int index) {
  // operator[] performs validity checks:
  T* v = operator[](index);
  // "Remove" the pointer:
  storage[index] = 0;
  return v;
}

template<class T, int incr>
void PStash<T, incr>::inflate(int increase) {
  const int tsz = sizeof(T*);
  T** st = new T*[quantity + increase];
  memset(st, 0, (quantity + increase) * tsz);
  memcpy(st, storage, quantity * tsz);
  quantity += increase;
  delete []storage; // Old storage
  storage = st; // Point to new memory
}
#endif // TPSTASH2_H ///:~
```

734

735

这个文件的大部分是先前**PStash**和嵌套**iterator**直接翻译成的模板。然而，这时运算符返回对当前迭代器的引用，这是更典型和更灵活的方法。

析构函数对于所有被包含的指针调用**delete**，并且因为类型由模型获取，所以将发生适当的销毁。应当知道，如果容器存放指向基类类型的指针，那么这个类型应当有虚析构函数，以保证正确地清除派生对象，当将这些派生对象放进该容器时，它们的地址已经发生了向上类型转换。

PStash::iterator遵循迭代器在其生命期内只结合单个容器对象这一模式。另外，拷贝构造函数允许让新迭代器指向已存在迭代器指向的同一位置，这就像在容器内夹了书签。**operator+=**和**operator-=**成员函数允许移动迭代器一些距离，但与容器的边界有关。重载的增

加和减小运算符移动迭代器一个位置。**operator+**生成新迭代器，它向前移动加数个位置。像前面的例子一样，指针反向引用运算符被用于在迭代器涉及的元素上进行运算，**remove( )**通过调用容器的**remove( )**来销毁当前对象。

就像前面的同样代码（按照标准C++库容器方式），被用来创建终止哨兵：第二构造函数、容器的**end( )**成员函数和用于比较的**operator==**与**operator!=**。

下面的例子创建和测试两个不同种类的**Stash**对象：一个成为名为**Int**的新类，它宣布它的构造和析构；另一个存放标准库**string**类的对象。

```
//: C16:TPStash2Test.cpp
#include "TPStash2.h"
#include "../require.h"
#include <iostream>
#include <vector>
#include <string>
using namespace std;
class Int {
  int i;
public:
  Int(int ii = 0) : i(ii) {
    cout << ">" << i << ' ';
  }
  ~Int() { cout << "~" << i << ' '; }
  operator int() const { return i; }
  friend ostream&
    operator<<(ostream& os, const Int& x) {
      return os << "Int: " << x.i;
  }
  friend ostream&
    operator<<(ostream& os, const Int* x) {
      return os << "Int: " << x->i;
  }
};

int main() {
  { // To force destructor call
    PStash<Int> ints;
    for(int i = 0; i < 30; i++)
      ints.add(new Int(i));
    cout << endl;
    PStash<Int>::iterator it = ints.begin();
    it += 5;
    PStash<Int>::iterator it2 = it + 10;
    for(; it != it2; it++)
      delete it.remove(); // Default removal
    cout << endl;
    for(it = ints.begin();it != ints.end();it++)
      if(*it) // Remove() causes "holes"
        cout << *it << endl;
  } // "ints" destructor called here
  cout << "\n--------------------\n";
  ifstream in("TPStash2Test.cpp");
  assure(in, "TPStash2Test.cpp");
  // Instantiate for String:
```

736

```
  PStash<string> strings;
  string line;
  while(getline(in, line))
    strings.add(new string(line));
  PStash<string>::iterator sit = strings.begin();
737 for(; sit != strings.end(); sit++)
    cout << **sit << endl;
  sit = strings.begin();
  int n = 26;
  sit += n;
  for(; sit != strings.end(); sit++)
    cout << n++ << ": " << **sit << endl;
} ///:~
```

为了方便，**Int**有一个相关的**ostream operator<<**，用于**Int&**和**Int***。

在**main( )**中的第一个代码段用花括号括起来，用来强迫**PStash<Int>**的销毁，并因而由析构函数自动清除。用手工移走和删除元素的范围，以表明**PStash**清除了剩余的元素。

对于**PStash**的这两个实例，创建一个迭代器并用于遍历容器。注意由使用这些构造函数所产生的简洁性，我们不会遭受使用数组的实现细节的困扰。只要告诉容器和迭代器对象做什么，而无需告诉它们如何做。这就使得这个解决方法容易概念化、建立和修改。

16.8 为什么使用迭代器

到目前为止，我们已经看到了迭代器的机制，但是要理解为什么它们如此重要还需要采用更复杂的例子。

在一个真实的面向对象程序中，经常可以看到多态性、动态对象创建和容器在一起使用。容器和动态对象创建解决了不知道我们需要多少对象以及对象是什么类型的这样的问题。如果配置一个容器存放指向基类对象的指针，每次放置一个派生类指针进入容器时，就发生向上类型转换。正如本书第1卷中最后的代码，这个例子还将我们至今已经学习过的各个不同方
738 面放在一起，如果我们能理解这个例子，我们就为学习第2卷做好了准备。

假设我们正在创建一个程序，这个程序允许用户编辑和产生不同种类的图画。每个图画都是一个包含一组**Shape**对象的对象。

```
//: C16:Shape.h
#ifndef SHAPE_H
#define SHAPE_H
#include <iostream>
#include <string>

class Shape {
public:
  virtual void draw() = 0;
  virtual void erase() = 0;
  virtual ~Shape() {}
};

class Circle : public Shape {
public:
  Circle() {}
  ~Circle() { std::cout << "Circle::~Circle\n"; }
```

```
  void draw() { std::cout << "Circle::draw\n";}
  void erase() { std::cout << "Circle::erase\n";}
};

class Square : public Shape {
public:
  Square() {}
  ~Square() { std::cout << "Square::~Square\n"; }
  void draw() { std::cout << "Square::draw\n";}
  void erase() { std::cout << "Square::erase\n";}
};

class Line : public Shape {
public:
  Line() {}
  ~Line() { std::cout << "Line::~Line\n"; }
  void draw() { std::cout << "Line::draw\n";}
  void erase() { std::cout << "Line::erase\n";}
};
#endif // SHAPE_H ///:~
```

这段代码使用了基类中虚函数的典型结构，这此虚函数在派生类中被重新定义。注意， 739
Shape类包含一个虚析构函数，应当将有些东西自动添加到具有虚函数的任何类中。如果一个容器存放指向**Shape**对象的指针或引用，则当对这些对象调用这个虚析构函数时，所有的相关数据都将被正确地清除。

在下面例子中，每一个不同类型的图画都使用了不同种类的模板化容器类：已经在本章定义的**PStash**和**Stack**，以及来自标准C++库的**vector**类。容器的“使用”是极其简单的，并且通常情况下，继承可能不是最好的方法（组合可能更有意义），但是，在这种情况下，继承是一个简单的方法，并没有从这个例子中去掉。

```
//: C16:Drawing.cpp
#include <vector> // Uses Standard vector too!
#include "TPStash2.h"
#include "TStack2.h"
#include "Shape.h"
using namespace std;

// A Drawing is primarily a container of Shapes:
class Drawing : public PStash<Shape> {
public:
  ~Drawing() { cout << "~Drawing" << endl; }
};

// A Plan is a different container of Shapes:
class Plan : public Stack<Shape> {
public:
  ~Plan() { cout << "~Plan" << endl; }
};

// A Schematic is a different container of Shapes:
class Schematic : public vector<Shape*> {
public:
  ~Schematic() { cout << "~Schematic" << endl; }
};
```

```
// A function template:
template<class Iter>
740 void drawAll(Iter start, Iter end) {
  while(start != end) {
    (*start)->draw();
    start++;
  }
}

int main() {
  // Each type of container has
  // a different interface:
  Drawing d;
  d.add(new Circle);
  d.add(new Square);
  d.add(new Line);
  Plan p;
  p.push(new Line);
  p.push(new Square);
  p.push(new Circle);
  Schematic s;
  s.push_back(new Square);
  s.push_back(new Circle);
  s.push_back(new Line);
  Shape* sarray[] = {
    new Circle, new Square, new Line
  };
  // The iterators and the template function
  // allow them to be treated generically:
  cout << "Drawing d:" << endl;
  drawAll(d.begin(), d.end());
  cout << "Plan p:" << endl;
  drawAll(p.begin(), p.end());
  cout << "Schematic s:" << endl;
  drawAll(s.begin(), s.end());
  cout << "Array sarray:" << endl;
  // Even works with array pointers:
  drawAll(sarray,
    sarray + sizeof(sarray)/sizeof(*sarray));
  cout << "End of main" << endl;
} ///:~
```

不同类型的容器都存放指向**Shape**的指针和指向**Shape**派生类的向上类型转换对象的指
741 针。然而，因为多态性，当调用虚函数时，仍然出现正确的行为。

注意，**Shape***的数组**sarray**也可以被看做一个容器。

16.8.1 函数模板

在**drawAll( )**中，我们已经看到了一些新东西。但是，到本章为止，我们仅仅使用了类模板，它们实例化基于一个或多个类型参数的新表。然而，我们可以同样容易地创建函数模板，它们创建基于类型参数的新函数。创建函数模板的理由与使用类模板的理由相同：我们试图创建一般性的代码，我们可以通过延迟规定一个或多个类型的方法来创建这样的代码。我们只想写明这些类型参数支持特定运算，并不确切地说明它们是什么类型。

函数模板**drawAll( )**可以看做是一个*算法*（在标准C++库中大部分函数模板被称为算法）。它只是给出描述元素的一个区域的迭代器，说明如何做某件事情，只要这些迭代器能被反向引用、增加和比较。在本章中，我们已经开发出这种迭代器，但这也不是巧合，这种迭代器由标准C++库中的容器生成，在这个例子中由使用**vector**证实。

我们还希望**drawAll( )**是一个*泛型算法*（*generic algorithm*），所以容器可以是任意类型的，我们没有必要为每个不同类型的容器编写这个算法的新版本。在此，函数模板是基本的，因为它们能自动地为每个不同类型的容器产生特殊代码。但是，如果没有由迭代器提供的另外的间接性，这种泛型（genericness）就没有可能。这就是迭代器为什么如此重要的原因；它们允许用户编写涉及容器的通用代码，而用户并不知道容器的下层结构。（注意，在C++中，为了正确工作，迭代器和泛型算法都需要函数模板。）

在**main( )**中可以看到这点的证明，因为**drawAll( )**的工作不随着容器类型的不同而改变。更有趣的是，**drawAll( )**对于指向数组**sarray**的开始和结尾的指针也能工作。这种将数组作为容器处理的能力是标准C++库设计的一部分，它们的算法很像**drawAll( )**。 742

因为容器类模板很少关系到普通类所具有的继承和向上类型转换，所以不会在容器类中看到虚函数。容器的重用是用模板，而不是用继承实现的。

16.9 小结

容器类是面向对象程序设计的一个基本部分。它们是简化和隐藏实施细节、提高开发效率的另一种方法。另外，它们通过替换C语言中发现的原始数组和相对粗糙的数据结构技术从而大大地提高了程序的灵活性和安全性。

因为客户程序员需要容器，所以容器的便于使用是它的基本特征。这样，模板就被引入。使用模板语法，对源代码进行的重用（相反的是，由继承和组合提供的对对象代码进行的重用）对初学者来说变得十分平常。实际上，使用模板实施代码重用比使用继承和组合实施代码重用容易得多。

虽然在本书中我们已经学习了创建容器和迭代器类的相关知识，但实际上，更有用的是学习了在标准C++库中的容器和迭代器，因为可以期望在每个编译器中使用它们。正如我们将会在本书的第2卷（可从*www.BruceEckel.com*处下载）中看到的，标准C++库中的容器和算法实际上总能满足我们的需要，因此不需要自己创建新的容器类。

本章已经涉及与容器类设计有关的问题，但我们可能希望学习更多的内容。一个更加复杂的容器类库可以覆盖所有的其他问题，包括多线程、持久存储和无用单元收集。 743

16.10 练习

部分练习题的答案可以在本书的电子文档 "*Annotated Solution Guide for Thinking in C++*" 中找到，只需支付很少的费用就可以从http://www.BruceEckel.com得到这个电子文档。

16-1 实现本章中**OShape**图的继承层次。

16-2 修改第15章练习1的结果，以便使用**TStack2.h**中的**Stack**和**iterator**替代一个**Shape**指针的数组。增加针对类层次的析构函数，使得我们可以观察到：在**Stack**超出范围时**Shape**对象的销毁。

16-3 修改**TPStash.h**，使得由**inflate( )**使用的增量值能在特定容器对象的生命期内改变。

16-4 修改**TPStash.h**，使得由**inflate( )**使用的增量值能自动地调整自身大小，以减少它需要被调用的次数。例如，每次调用它，它都能为下一次调用而加倍这个增量值。通过报告是否**inflate( )**被调用来证明这个功能，并且在**main( )**中编写测试代码。

16-5 对于**fibonacci( )**函数产生的值的类型，模板化**fibonacci( )**函数（使得它能产生**long**、**float**等类型的值，而不只产生**int**型值）。

16-6 使用标准C++库**vector**作为下层实现，创建一个**Set**模板类，对于放入这个模板类中的每一种对象，它只接受一个。创建一个嵌套**iterator**类，它支持本章中的“终止哨兵”思想。在**main( )**中编写测试代码，并随后代替标准C++库的**Set**模板以验证其行为是正确的。

16-7 修改**AutoCounter.h**使得它能在任何类中被用作成员对象，我们希望能跟踪它的创建和销毁。增加一个**string**成员，用来保存这个类的名称。在我们自己的一个类中测试这个工具。

744 16-8 创建**OwnerStack.h**的一个版本，它使用标准C++库**vector**作为它的下层实现。为此，我们可能需要查寻**vector**的一些成员函数（或只考虑**<vector>**头文件）。

16-9 修改**ValueStack.h**，使得当我们**push( )**更多的对象并且超出空间时它能自动地扩展。改动**ValueStackTest.cpp**以测试新的功能性。

16-10 重复练习9，但使用标准C++库**vector**作为**ValueStack**的内部实现。注意，这种做法容易多了。

16-11 修改**ValueStackTest.cpp**，使得它在**main( )**中使用标准C++库**vector**而不使用**Stack**。注意运行时的行为：当**vector**创建时它自动创建一系列默认对象吗？

16-12 修改**TStack2.h**，使得它使用标准C++库**vector**作为它的下层实现。确信不要改变接口就能让**TStack2Test.cpp**照常工作。

16-13 使用标准C++库**Stack**而不使用**vector**重复练习12（可能需要查寻关于**stack**的信息，或者搜索**<stack>**头文件）。

16-14 修改**TPStash2.h**，使得它使用标准C++库**vector**作为它的下层实现。确信不要改变接口就能让**TPStash2Test.cpp**照常工作。

16-15 在**IterIntStack.cpp**中，修改**IntStackIter**，给它一个“终止哨兵”构造函数，添加**operator ==**和**operator!=**。在**main( )**中，使用一个迭代器遍历这个容器的元素，直到我们到达“终止哨兵”。

16-16 使用**TStack2.h**、**TPStash2.h**和**Shape.h**，为**Shape***实例化**Stack**和**PStash**容器，对它们填充向上类型转换的**Shape**指针，然后用迭代器遍历每个容器，并为每个对象调用**draw( )**。

16-17 模板化**TPStash2Test.cpp**中的**Int**类，使得它存放任意类型的对象（可以改变这个
745 类的名称，使之更确切）。

16-18 模板化来自第12章的**IostreamOperatorOverloading.cpp**中的**IntArray**类，模型化它包含的对象的类型和内部数组的长度。

16-19 将来自第12章的**NestedSmartPointer.cpp**中的**ObjContainer**翻译成一个模板。用两个不同的类测试它。

16-20 通过模板化**class Stack**来修改**C15:OStack.h**和**C15:OStackTest.cpp**，使得它自动

地从被包含类和从**Object**多重继承。被产生的**Stack**应当只接受和生成被包含类型的指针。

16-21 使用**vector**而不使用**Stack**重复练习20。

16-22 从**vector<void*>**继承一个类**StringVector**，并且重新定义**push_back( )**和**operator[]**成员函数，使得它只接受和生成**string***（并执行适当的类型转换）。现在，创建一个模板，它将自动地产生一个容器类以便对任何类型的指针做同样的事情。这个技术常常用于减少代码膨胀，防止过多的模板实例化。

16-23 在**TPStash2.h**中，对**PStash::iterator**添加和测试**operator−**，仿照**operator+**的逻辑。

16-24 在**Drawing.cpp**中，添加和测试一个函数模板，用来调用**erase( )**成员函数。

16-25 （高级）修改**TStack2.h**中的**Stack**类以允许所有权的所有粒度：为每一个链表增加一个标志以表明它是否拥有其指向的对象，并在**push( )**函数和析构函数中支持这一信息。增加用于读取和改变每一个链表所有权的成员函数。

16-26 （高级）修改来自第12章的**PointerToMemberOperator.cpp**，使得**FunctionObject**和**operator->***被模板化，以便与任何返回类型工作（对于**operator->***，必须用成员模板，这将在第2卷中介绍）。在**Dog**成员函数中，添加和测试对于零个、一个和两个参数的支持。

746

附录A 编码风格

这个附录不是关于圆括号和大括号的缩排和布置，虽然这也将谈到。它是在本书
747 中使用的组织代码清单的一般性指导方针。

虽然这里的大部分问题已在本书中介绍过，但是，放在最后的这个附录将涉及到每一个主题，如果我们对某些方面不太理解，可以在相应的章节中查找。

本书中的所有的编码风格都已认真考虑和实施，有的在一年以上。当然，每一个人都有他自己的组织代码所用方法的理由，而我只是试图告诉读者我是如何形成我的风格的，约束和环境的因素如何引导我做出这样的决定的。

A.1 常规

在本书的正文中，标识符（函数、变量和类名）用**黑体字**。大部分关键字也用黑体，除了那些经常用的关键字，它们再用黑体就变得乏味了，例如“class” 和 “virtual”。

在这本书的例子中，我使用特殊的编码风格，它已经发展几年了，并部分地受到了Bjarne Stroustrup在他的原作《*C++程序设计语言*》（*The C++ Programming Language*）中的风格的影响。格式风格的话题可以争论几个小时，我只能说我不是试图通过我的例子规定正确的风格、我有自己使用我所创造的风格的动机。因为C++是一个自由格式的程序设计语言，所以任何人可以使用他已经适应了的任何风格。

这就是说，我会注意到，重要的是在一个项目中有一致的风格。如果读者搜索因特网，会发现很多工具能用来重新格式化项目的代码，使得它们达到一致，这种一致是有意义的。

748 本书中的程序是能自动从本书正文中摘取出来的文件。能测试它们，保证它们正确地运行，如果使用的是与标准C++一致的编译器的话（注意，不是每个编译器都支持所有的语言性能），打印在书中的代码文件应当允许没有编译时错误。会引起编译时错误信息的语句用注释符//! 注释出来了，所以它们能很容易被发现和用自动的手段测试。对于作者的错误发现和报告最早出现在本书的电子版（*www.BruceEckel.com*）及稍后的修正版中。

本书的标准之一是所有的程序都将无错误编译和连接（虽然有时引起警告）。最后，一些程序仅仅示范编码例子，并不表示独立的程序，它有空**main( )**函数，例如

```
int main() {}
```

这使完整程序的连接没有错误。

标准的**main( )**返回**int**，而标准C++规定，如果在**main( )**中没有**return**语句，编译器将自动地产生**return 0**代码。本书中使用了这个选项（在**main( )**中没有**return** 语句）（一些编译器可能仍然产生警告，但标准C++不会）。

A.2 文件名

在C中，惯例是，名字头文件（包含声明）以**.h**为扩展名，实现文件（引起内存分配和代

码生成）以.c为扩展名。C++经历了一个发展过程，它首先在Unix上开发，这个操作系统能分辨文件名的大小写。最初的文件名简单地使用相对于C扩展名的大写.H和.C。这当然对于不能区分大小写的操作系统不行，例如DOS。DOS C++对于头文件和实现文件分别使用扩展名hxx和cxx，或者hpp和cpp。后来，有人分析了必须使用不同的扩展名的惟一的原因，是使得编 749
译器能区分是作为C文件编译还是作为C++文件编译。因为编译器绝对不会直接编译头文件，所以只有实现文件的扩展名需要改变。这个习惯风行于所有的系统，现在已经变成对于实现文件用cpp和对于头文件用h。注意，当包含标准C++头文件时，可以不用扩展名，也就是#include <iostream>。

A.3 开始和结束注释标记

非常重要的问题是在本书中看到的所有代码必须被验证是正确的（至少用一个编译器）。这是通过自动地从本书中提取这些文件完成的。简单地说，所有的列出的代码都被编译过（少数代码片段除外），它们在开始和结尾处有注释标记。这些标记是本书第2卷中的提取工具ExtractCode.cpp要使用的，它将代码清单从本书的无格式的ASCII文本中抽取出来。

列表结尾处的标记符简单地告诉ExtractCode.cpp这是清单的结尾处，而开始处的标记后面跟着关于这些文件所在子目录的信息（一般以章组织，例如，一个文件在第8章，则有标记符C08），再跟一个逗号，然后是这个清单文件名。

因为ExtractCode.cpp还为每个子目录创建一个makefile，所以关于程序如何制作和如何用命令行测试的信息也放在清单里。如果一个程序是单独的（不需要与任何其他程序连接），则它就没有附加信息。对于头文件也是如此。如果程序中不包含main( )，这必须与其他部分连接，则在文件名后有{O}。如果清单是主程序，但需要与其他部分连接，则有独立的以//{L}开头的行，接着是需要连接的所有文件名（不带扩展名，因为它们会随着平台变化而变化。）

我们可以在全书中找到例子。 750

如果要提取一个文件，但开始和结束标记不需要包含在被提取的文件中（例如这是一个测试数据文件）则它的开始标记后面直接跟随‘!’。

A.4 圆括号、大括号和缩排

我们可能注意到在这本书中的格式风格不同于许多传统的C风格。当然，每个人都认为他自己的风格是最自然的。但是这里使用的风格有简单的逻辑基础，在这里将结合其他先进的风格思想一起介绍。

格式风格只有一个动机：表达，无论是打印形式还是现场讨论形式。都会有人感到因为没有表达清楚,从而得到的并非是需要的。然而对代码来说是读多于写，所以应当让读者感到容易。我的两个最重要的标准是“可扫视性”（读者扫视一行的含义有多么容易）和一页中能容纳的行数。后者可能更有趣，当我们做演示的时候，如果必须在幻灯片之间翻来翻去，会严重分散听众的注意。引起这种情形的原因是其中包含了一些没有用的文字行。

每个人都同意代码内的括号应当是错位的，不同意见，也就是在格式风格中的最不一致的地方是：左括号在什么地方？我认为，这是一个引起代码风格不同的问题（编码风格的列表，参看Tom Plum 和Dan Saks的《*C++ Programming Guidelines*》，Plum Hall，1991）。今天的许多代码风格来自标准C之前的约束（函数原型之前），但现在不适用了。

首先，我对这个关键问题的回答是，左括号总是应与“前导元素”（指“任何代码体：类、
751 函数、对象定义或条件语句，等等）处于同一行。这是我写所有代码使用的一贯规则，它使得格式更简单，使得当我看到这一行时 “可扫视性”更好。

```
int func(int a);
```

我们知道，这一行最后的分号表示这是一个声明，后边不再有什么东西，但当看到下面行时

```
int func(int a) {
```

马上就知道它是定义，因为这一行是以开括号结束的，没有分号。使用这种方法，为多行定义放开括号也是一样的。

```
int func(int a) {
  int b = a + 1;
  return b * 2;
}
```

而为单行定义常常用于内联函数。

```
int func(int a) { return (a + 1) * 2; }
```

类似的，对于类:

```
class Thing;
```

是一个类名声明，而

```
class Thing {
```

是类定义。我们可以通过扫视这个类的这单独一行，就能知道这是声明还是定义。当然，将开括号放在这同一行而不是单独作为一行，可以在一页中放更多的程序行。

那么为什么我们还有许多其他的风格呢？特别是，我们注意到，许多人创建类沿用上面的风格［Stroustrup在他编写的由Addison-Wesley出版的《*C++程序设计语言*》（*The C++ Programming Language*）一书中使用了上述风格］，但在创建函数定义时却将左大括号构
752 成单独一行（这又形成了许多不同的独立风格）。Stroustrup让短的内联函数例外。用我在这里描述的方法，一切都是一致的，无论是何种名字（类、函数、枚举等），我们都将左大括号放在同一行中，指明下面是相应的程序体。而且，对于短内联函数和一般函数定义，左括号的格式也是一样的。我断定，许多人使用的函数定义的分割来源于先前的C函数原型。在其中，我并不在圆括号之内声明参数，而是在右圆括号和左大括号之间（这表明了C 的汇编语言的根）。

```
void bar()
 int x;
 float y;
{
 /* body here */
}
```

这样，如果将左大括号放在与前导元素同一行中，这很难看。所以没有人这样做。但是，大括号是应当独立于代码体还是应当在“前导元素”层上有不同意见，这样，我们就得到了不同的格式风格。

对于直接把左大括号放在（类、结构、函数等的）声明之后，有一些争论。下面的意见来自读者，列举出来，使得我们知道问题是什么：

> 有经验的‘vi’（vim）用户知道，击‘]’键两次，将使用户到下一个首列为‘{’（或^L）处。这个性能对于浏览代码非常有用（跳到下一个函数或类定义处）。[我的解释是，当我最初在Unix上工作时，GNU Emacs刚刚出现，我迷上了它。结果，‘vi’没有引起我的兴趣，这样，我就没有想到“首列位置问题”。但是，有相当多的‘vi’用户受这个问题的影响。]
>
> 置‘{’于下一行，消除了一些在复杂条件下的混淆代码，有助于扫视性。例如：

753

```
if(cond1
   && cond2
   && cond3) {
   statement;
}
```

> 上面代码扫视性很差[读者断言]。但是

```
if (cond1
&& cond2
&& cond3)
{
statement;
}
```

> 将‘if’与代码体隔开，导致了最好的可读性。[读者是否同意这一观点与读者的习惯有关]。
>
> 最后，当将大括号安排在上述列上时，很容易有视觉上的效果。它们在视觉上很好地“突出”出来。[读者的评论结束]

大括号放在何处的问题大概是意见最不一致的问题。我研究了这两种格式，最终归结为这依赖于程序员逐渐适应了什么。但是，我注意到，正式的Java编码标准（在Sun的Web站点上）与我在这里描述的一样，因为越来越多的人开始用这两种语言编码，二者之间编码风格一致可能是有好处的。

我所用的方法去除了所有的异常和特殊情况，理论上形成了独立风格。即便在函数体内，也保持一致性，例如

```
for(int i = 0; i < 100; i++) {
  cout << i << endl;
  cout << x * i << endl;
}
```

754

这种风格容易讲授和记忆，对所有的格式用单一和一致的规则，不是对于类用一种，对于函数用第二种（内联用一行，其他用多行），对于**for**循环和**if**语句用另外一些等等。我认为，一致性是值得考虑的。最重要的是，C++是比C更新的语言，虽然我们必须照顾C，但是不应当将太多的旧的东西保留下来，以免在未来引起问题。由代码多行引起的问题变成了大问题，即使是在C中。为了彻底考察这一问题，请参看David Straker编写的《*C Style: Standards and Guidelines*》(Prentice-Hall, 1992)。

我必须遵从的其他约束是行宽，因为本书限定50个字符。当某一行太长时会怎么样呢？

我再一次努力用统一的策略断开行，使得它们很容易查看。只要某些行是一个定义、参数表等等的一部分，继续行就应当相对于定义、参数表等等后退一层。

A.5 标识符名

熟悉Java的读者会注意到，对于所有的标识符名，我已经转向使用标准Java风格。但是，我不能做到完全一致，因为在标准C和C++库中的标识符不遵从这种风格。

这种风格是相当直截了当的。如果这个标识符是一个类，它的第一个字母必须大写，如果是函数或变量，则第一个字母是小写。标识符的其余部分由一个或多个单词组成，连在一起，但每个单词的第一个字母用大写，以示区分。所以类如下所示：

```
class FrenchVanilla : public IceCream {
```

对象标识符如下：

```
FrenchVanilla myIceCreamCone(3);
```

函数如下：

755

```
void eatIceCreamCone();
```

（无论是成员函数还是常规函数）。

一个例外是编译时常量（**const**或**#define**），其中所有的字母都用大写。

这种风格的价值在于大写字母有意义，能从第一个字母看出是一个类还是一个对象或方法。当访问静态（**static**）类成员时特别有用。

A.6 头文件的包含顺序

头文件被包含的顺序是从“最特殊到最一般”。这就是，在本地目录中的任何头文件首先被包含，然后是我们自己的所有“工具”头文件，例如**require.h**，随后是第三方库头文件，接着是标准C++库头文件和C库头文件。

要了解其原因，可以看John Lakos在《*Large-Scale C++ Software Design*》(Addison-Wesley, 1996)中的一段话：

> 保证.h文件的组成部分不被它自身解析（parse），这可以避免潜在的使用错误。因为被自身解析缺乏明确提供的声明或定义。在.c文件的第一行包含.h文件能确保所有对于构件的物理界面重要的内部信息块都在.h中（如果的确是缺少了某些信息块，一旦编译这个.c文件时就可以发现这个问题）。

如果包含头文件的顺序是“从最特殊到最一般”，如果我们的头文件不被它自己解析，我
756 们将马上找到它，防止麻烦事情发生。

A.7 在头文件上包含警卫

总是在头文件中使用包含警卫（*include guard*），防止编译单个**.cpp**文件期间一个头文件被多次包含。包含警卫是由一个预处理**#define**实现的，检查这个名字是否已经定义。警卫使用的名字是以头文件的名字为基础的，将文件名的字母大写，用下划线替换‘.’，例如：

```
// IncludeGuard.h
```

```
#ifndef INCLUDEGUARD_H
#define INCLUDEGUARD_H
// Body of header file here...
#endif // INCLUDEGUARD_H
```

包含在最后一行中的标识符是为了清晰。虽然一些预处理器忽略**#endif**之后的任何字符，但这不是标准写法，这个标识符用于注释。

A.8 使用名字空间

在头文件中，必须小心保证其中没有包含任何有“污染” 的名字空间，这就是，如果改变函数或类外面的名字空间，将引起包含这个头文件的任何文件的改变，这可能导致各种问题。不允许在函数声明外面有任何**using**声明，不允许在头文件中有全局**using**声明。

在**.cpp**文件中，任何全局**using**指示将只影响这个文件，因此，在本书中，一般用它们生成更容易阅读的代码，特别是在小程序中。

A.9 require( )和assure( )的使用

require( )和**assure( )**函数在**require.h**中定义，本书的各处都有应用，它们能报告某些问题。如果我们熟悉了*前置条件*和*后置条件*概念（由Bertrand Meyer提出），我们会认识到， 757
require( )和**assure( )**能或多或少地提供前置条件（通常地）和后置条件（有时）。这样，在函数开始，即函数“核心”执行之前，前置条件被检查，以保证所有的必要条件都正确。然后执行函数的这个“核心”，有时检查后置条件，以确保数据的新的状态在预料之中。我们将注意到，后置条件的检查在本书中很少，**assure( )**主要用于保证文件已经成功地打开。 758

附录B 编程准则

这个附录收集了C++编程的一些建议，它们是我在教学和实践过程中收集而成的，
759 当然还有：

还有一些忠告来自我的一些朋友，包括Dan Saks（与Tom Plum合著了《*C++ Programming Guidelines*》，1991）、Scott Meyers(《*Effective C++*》一书的作者，Addison-Wesley, 1998)和Rob Murray（《*C++Strategies & Tactics*》的作者，Addison-Wesley,1993）。而且，有许多条目都是从本书中摘录下来的。

1. 首先让程序运行，然后再追求速度。即使我们确定这一段程序非常重要，而且是我们系统中的瓶颈。不要优化，首先用尽可能简单的设计使程序可以运行，如果速度不满足要求，再对其进行分析。我们总是能够发现“我们的”瓶颈并不是问题所在。节省我们的时间做真正有意义的事。
2. 编写简洁优美的程序有很多潜在的好处。这不是可有可无的。简洁优美的程序不仅易读，易调试，而且易于理解和维护，这正是能够带来经济利益的地方。这一点只有通过实践才可能体会，因为初看来，使程序简洁优美会影响程序的生产效率，但是，当我们的程序能够无缝地集成进我们的系统，甚至我们的程序需要修改时，就会体现出优点。
3. 记住要“分而治之”。如果感到问题复杂，试着猜测程序的最基本操作，为最难的部分创造一个对象——书写代码并且应用这个对象，然后将这个最难的部分嵌入其他的对象，等等。
4. 不要用C++主动重写我们已有的C代码，除非我们需要对它的功能做较大的调整，（也就是说，如果能用就不要重做）。用C++重新编译是很有价值的，因为这可以发现隐藏的错误。把一段运行得很好的C代码用C++重写可能是在浪费时间，除非C++的版本以
760 类的形式提供许多重用的机会。
5. 如果有许多C代码需要改变，首先隔离不需要修改的代码，最好将那些函数打包成“API 类”的静态成员函数。然后集中精力到要修改的代码，将它们精化成类以使以后的维护修改更容易。
6. 要区别类的创建者和类的使用者（客户程序员）。类的使用者才是“顾客”，他们并不需要或许也不想知道类的内部是怎样运作的。类的创建者必须是设计类和编写类的专家，以使得被创建的类可以被最没有经验的程序员使用，而且在应用程序中工作良好。库只是在透明的情况下才会容易使用。
7. 当我们创建一个类时，要尽可能用有意义的名字来命名类。我们的目标应该是使用户接口要领简单。可以用函数重载和默认参数来创建一个清楚、易用的接口。
8. 数据隐藏允许我们（类的创建者）将来在不破坏用户代码（代码使用了该类）的情况下随心所欲地修改代码。为实现这一点，应把对象的成员尽可能定义为**private**，而只让

接口部分为**public**，而且总是使用函数而不是数据。只有在迫不得已时才让数据为**public**。如果类的使用者不需要调用某个函数，就让这个函数成为**private**。如果类的一部分要让派生类可见，就定义成**protected**，并提供一个函数接口而不是直接暴露数据，这样，实现部分的改变将对派生类产生最小的影响。

9. 不要陷入分析瘫痪中。有些东西只有在编程时才能学到并使各种系统正常。C++有内建的防火墙，让它们为我们服务。在类或一组类中的错误不会破坏整个系统的完整性。 761
10. 我们的分析和设计至少要在系统中创建类、它们的公共接口、它们与其他类的关系、特殊的基类。如果我们的方法产生的东西比这些更多，就应当问问自己，是不是所有的成分在程序的整个生命期中都是有价值的，如果不是，将会增加我们对它们的维护开销。开发小组的人都认为不应该维护对他们的产品没有用的东西。许多设计方法并不大奏效，这是事实。
11. 首先写测试代码（在写类之前），并和类代码一起提交，运用makefile或其他工具使运行测试自动化。这样，在运行测试代码之前就可以自动校验改变，迅速发现错误。因为我们拥有检测错误的体系，所以当发现需要修改代码时，会更大胆地进行尝试。在语言的发展中，最大的进步就是在语言内部建立了类型检查等测试、例外处理等机制。但是这些只能提供给我们这么多，我们应该针对自己的类或程序的特殊性进行测试保证程序的鲁棒性。
12. 首先书写测试代码（在写类代码之前）可以保证类设计的完整性。如果不写测试代码，就不知道我们的类能够做什么。另外，写测试代码的过程会使我们想到类中所需的其他特性或约束条件——这些特性或约束通常在分析和设计阶段不易察觉。
13. 记住软件工程的基本原则：所有的问题都可以通过引进一个额外的间接层来简化[⊖]。这是抽象方法的基础，而抽象是面向对象编程的首要特征。 762
14. 尽可能地原子化类。也就是每个类有一个单一、清楚的目的。如果我们的类或我们设计的系统过于复杂，就应当将所有复杂的类分解成多个简单的类。
15. 注意较长的成员函数定义，长的复杂的函数难于维护，而且很可能这个函数自己做了太多的事情。如果看到这样一个函数，至少预示着应该分解成几个函数，甚至预示着应该创造一个新类。
16. 注意长的参数表，这样函数调用会难写、难读、难于维护。应该把这个成员函数改成一个合适的类，用对象作为参数传递。
17. 不要自我重复。如果一段代码在派生类的许多函数中重复出现，就把这段代码放在基类的一个单一的函数中然后在派生类中调用它。这样我们不仅节省了代码空间，也使将来的修改容易传播。我们可以用内联函数来提高效率。有时发现这种通用代码会为我们的接口添加有用的功能。
18. 注意**switch**和**if-else**语句。它们是典型的*类型检查编码*的指示符。意味着程序运行的情况和我们的类型信息有关。（实际的类型也许不是最初看起来的类型）我们通常可以将这些代码换成继承和多态，多态函数会为我们进行类型检查，使程序更可靠而且易于扩展。
19. 从设计的角度，寻找并区分那些变化和不变的成分。也就是在系统中寻找那些修改时 763

⊖ Andrew Koenig向我解释了这一点。

不需要重新设计的成分，把它们封装到一个类中。我们可以在本书第2卷的“Design Patterns”一章中学到更多，它可在*www.BruceEckel.com*得到。

20. 注意不同点。两个语义上不同的对象可能有同样的操作或反应，自然就会试着把一个作为另一个的子类以便利用继承性的好处。这就叫差异，但并没有充分的理由来强制这种并不存在的父子关系。一个好的解决办法是产生一个共同的父类：它包含两个子类——这可能要多占一点空间，但我们可以从继承中获益，并且可能对这种设计有重要发现。
21. 注意在继承过程中的限制。最清晰的设计是向被继承者加入新的功能，而如果在继承过程删除了原有功能，而不是加入新功能，那这个设计就值得怀疑了。但这也不是绝对的，如果我们正在与一个老的类库打交道，对已有的类在子类中进行限制可能更有效，而不必重建一套类层次来使我们的新类适应新的应用。
22. 不要用子类去扩展基类的功能。如果一个类接口部分很关键的话，应当把它放在基类中，而不是在继承时加入。如果我们正在用继承来添加成员函数，我们可能应该重新考虑我们的设计。
23. 一个类一开始时接口部分应尽可能小而精。在类使用过程中，我们会发现需要扩展类的接口。然而一个类一旦投入使用，我们要想减少接口部分，就会影响那些使用了该
764 类的代码，但如果我们需要增加函数则不会有影响，一切正常，只需重新编译一下即可。但即使用新的成员函数取代了原来的功能，也不要去改正原有接口（如果我们愿意的话，可以在低层将两个函数合并）。如果我们需要对一个已有的函数增加参数，我们可以让原来的参数保持不变，把所有新参数作为默认参数，这样不会妨碍对该函数已有的调用。
24. 大声朗读我们的类，确保它们是合理的。读基类时用“is-a”，读成员对象时用“has-a”。
25. 在决定是用继承还是用组合时，问问自己是不是需要向上类型转换到基类。如果不需要，就用组合（成员对象）而不用继承。这样可以减少多重继承的可能。如果我们选择继承，用户会认为他们被假设向上类型转换。
26. 有时我们为了访问基类中的**protected**成员而采用继承。这可能导致一个可察觉的对多重继承的需求。如果我们不需要向上类型转换，首先导出一个新类来完成保护成员的访问，然后把这个新类作为一个成员对象，放在需要用到它的所有对象中去。
27. 一个典型的基类仅仅是它的派生类的一个接口。当我们创建一个基类时，默认情况下让成员函数都成为纯虚函数。析构函数也可以是纯虚函数（强制派生类对它重新定义），但记住要给析构函数一个函数体，因为继承关系中所有的析构函数总是被调用。
28. 当我们在类中放一个虚函数时，让这个类的所有函数都成为虚函数，并在类中定义一个虚析构函数。只有当我们要求高效时，而且分析工具指出应该这样做时，再把
765 **virtual**关键字去掉。
29. 用数据成员表示值的变化，用虚函数表示行为的变化。如果我们发现一个类中有几个状态变量和几个成员函数，而成员函数在这些变量的作用下改变行为，我们可能要重新设计它，用子类和虚函数来区分这种不同的作用。
30. 如果我们必须做一些不可移植的事，对这种服务做一个抽象并将它定位在一个类的内

部，这个额外的间接层可防止这种不可移植性影响我们的整个程序。

31. 尽量不用多重继承。这可帮我们摆脱困境，尤其是修复我们无法控制的类的接口时。除非我们是一个经验相当丰富的程序员，否则不要在系统中设计多重继承。
32. 不要用私有继承。虽然C++中可以有私有继承，而且似乎在某些场合下很有用，但它和运行时类型识别一起使用时，常常引起语义的模棱两可。我们可以用一个私有成员对象来代替私有继承。
33. 如果两个类因为一些函数的关系（如容器和迭代器）而联系在一起，使一个类设为公有并将另一个类包含成友元，像标准C++库将迭代器嵌入容器所做的一样（这样做的例子在第16章后部）。这不仅强调二者之间的联系，而且允许一个类的名字嵌入到另一个类中复用。标准C++通过在每个容器类中定义嵌入的迭代器类，为容器提供了通用接口。嵌入的另一个原因是可以作为私有运行的一部分。这里，嵌入比类之间的联系提供了更大的运行隐藏，而且防止出现上面提到的名字空间污染。 766
34. 运算符重载仅仅是“语法糖”：另一种函数调用方法。如果重载一个运算符不会使类的接口更清楚、更易于使用，就不要重载它。一个类只创建一个自动类型转换运算符，一般情况下，重载运算符应遵循第11章介绍的原则和格式。
35. 首先保证程序能运行，然后再考虑优化。特别是，不要急于写内联函数、使一些函数为非虚函数或者紧缩代码以提高效率。这些在我们开始构建系统时都不用考虑。我们开始的目标应该是证明设计的正确性，除非设计要求一定的效率。
36. 不要让编译器来为我们产生构造函数、析构函数或**operator=**。这些是训练我们的机会。类的设计者应该明确地说出类应该做什么，并完全控制这个类。如果我们不想要拷贝构造函数或**operator=**，就把它们声明为**私有的**。记住，只要我们产生了任何构造函数，就防止了默认构造函数被生成。
37. 如果我们的类中包含指针，我们必须产生拷贝构造函数、**operator=**和析构函数，以使类运行正常。
38. 当为派生类写拷贝构造函数时，记住要显式调用基类的拷贝构造函数（和成员对象版本，参见14章）。如果不这样做，基类（或者成员对象）会调用默认构造函数，可能这不是我们所想要的情形。要调用基类拷贝构造函数，用以下方式将它传给派生类：

```
Derived(const Derived& d) : Base(d) { // ...
```

39. 当为派生类写赋值操作符时，记住显式调用基类版本（参见第14章）。如果不如此，就不会起作用（成员对象也是如此）。应用基类的名字和作用域操作符，调用基类的 767
赋值运算符：

```
Derived& operator=(const Derived& d) {
  Base::operator=(d);
```

40. 为了减少大项目开发过程中的重复编译，应使用第5章介绍的句柄类/Cheshire cat技术，只有需要提高运行效率时才把它去掉。
41. 避免用预处理器。可以用常量来代替值，用内联函数代替宏。
42. 保持范围尽可能地小，这样我们的对象的可见性和生命期也就尽可能地小。这就减少了错用对象和隐藏难以发现的错误的可能性。比方说，假设我们有一个容器和一段扫描这个容器的代码，如果我们拷贝这些代码用于一个新的容器，我们可能无意间用原

有的容器的大小作为新容器的边界。然而，如果原来的容器超出了使用范围，这个错误就会被编译器发现。

43. 避免使用全局变量。尽可能把数据放在类中。全局函数的存在可能性要比全局变量大，虽然后来我们可能发现一个全局函数作为一个类的静态成员更合适。
44. 如果我们需要声明一个来自库中的类或函数，应该用包含一个头文件的方法。比如，如果我们想创建一个函数来写到**ostream**中，不要用一个不完全类型指定的方法自己来声明**ostream**，如

```
class ostream;
```

这样做会使我们的代码变得很脆弱。（例如**ostream**实际上可能是一个**typedef**。）我们可以用头文件的形式，例如：

```
#include <iostream>
```

768 当创建我们自己的类时，在只需要用到指针的情况下，如果一个库很大，应提供给用户一个头文件的简写形式，文件中包含有不完全的类型说明（就是类型名声明），它可以提高编译速度。

45. 当选择重载运算符的返回值类型时，要考虑表达式连成一串时可能出现的情况：当定义**operator=**时，应记住**x=x**。要对左值返回一个拷贝或一个引用（**return*this**），这样才能用在串连表达式（**A=B=C**）中。
46. 当写一个函数时，我们的第一选择是用**const**引用来传递参数。只要我们不需要修改正在被传递进入的对象，这种方式是最好的。因为它有着传值方式的简单，但不需要费时的构造和析构来产生局部对象，而这在传值方式时是不可避免的。通常我们在设计和构建我们的系统时不用注意效率问题，但养成这种习惯仍是件好事。
47. 当心临时变量。当调整效率时，要注意临时创建的对象，尤其是用运算符重载时。如果我们的构造函数和析构函数很复杂，创建和销毁临时对象就很费时。当从一个函数返回一个值时，总是应在return语句中调用构造函数来“就地”产生一个对象。

```
return MyType(i, j);
```

这优于

```
MyType x(i, j);
return x;
```

前一个返回语句（即所谓的*返回值优化*）避免了拷贝构造函数和析构函数的调用。

48. 当产生构造函数时，要考虑到异常情况，在最好的情况下，构造函数只是抛出异常，其次是：类只从健壮的类被组合和继承，所以当抛出异常时它们会自动清除它们所做

769 的一切。如果我们必须使用裸指针，我们应该负责捕获自己的异常，然后在我们的构造函数抛出异常之前释放所有指针指向的资源。如果一个构造函数无法避免失败，最好的方法是抛出异常。

49. 在我们的构造函数中只做一些最必要的事情，这不仅使构造函数的调用有较低的时间花费（这中间有许多可能不受我们控制），而且我们的构造函数更少地抛出异常和引起问题。
50. 析构函数的作用是释放在对象的整个生命期内分配的所有资源，而不仅仅是在创建

期间。

51. 使用异常层次，最好从标准C++异常层次中继承，并作为公共类嵌入能抛出异常的类中。捕获异常的人然后可以确定异常的类型。如果我们加上新的派生异常，已存在的客户代码还是通过基类来捕获这个异常。
52. 用值来抛出异常，用引用来捕获异常。让异常处理机制处理内存管理。如果我们抛出一个指向在堆上产生的异常的指针，则捕获者必须破坏这个异常，这是一种不利的耦合。如果我们用值来捕获异常，我们需要额外的构造和析构，更糟的是，我们的异常对象的派生部分可能在以值向上类型转换时被切片。
53. 除非确有必要，否则不要写自己的类模板。先查看一个标准模板库，然后查问创建特殊工具的开发商。当我们熟悉了这些产品后，我们就可大大提高我们的生产效率。
54. 当创建模板时，留心那些带类型的代码并把它们放在非模板的基类中，以防不必要的代码膨胀。用继承或组合，我们可以产生自己的模板，模板中包含的大量代码都应是必要的，类型有关的。 770
55. 不要用**<cstdio>**函数，例如**printf( )**。学会用输入输出流来代替，它们是安全和可扩展类型，而且功能也更强。我们在这上面花费的时间肯定不会白费。一般情况下都要尽可能用C++中的库而不要用C库。
56. 不要用C的内部数据类型，虽然C++为了向后兼容仍然支持它们，但它们不像C++的类那样强壮，所以这会增加我们查找错误的时间。
57. 无论何时，如果我们用一个内部数据类型作为一个全局或自动变量，在我们可以初始化它们之前不要定义它们。每一行定义一个变量，并同时对它初始化。当定义指针时，把‘*’紧靠在类型的名字一边。如果我们每个变量占一行，我们就可以很安全地定义它们。这种风格也使读者更易于理解。
58. 保证在所有代码前面初始化。在构造函数初始化表中完成所有成员的初始化，甚至包括内部数据类型（用伪构造函数调用）。在初始化子对象时用构造函数初始化表常常更有效；否则调用了默认构造函数，而不再调用使初始化正确的其他成员函数（可能是**operator=**）了。
59. 不要用“**MyType a = b;**”的形式来定义一个对象。这是常常引起混乱的原因。因为它调用构造函数来代替**operator=**。为了清楚起见，可以用“**MyType a(b);**”来代替。这个语句结果是一样的，但不会引起混乱。
60. 使用第3章中的C++显式类型转换。类型转换重载了正常的类型系统，它往往是潜在的错误点。通过把C中一个类型转换负责一切的情况改成多种表达清楚的转换，任何人来调试和维护这些代码时，都可以很容易地发现这些最容易发生逻辑错误的地方。 771
61. 为了使一个程序更强壮，每个组件都必须是很强壮的。在我们创建的类中运用C++中提供的所有工具：隐藏实现、异常、常量更正、类型检查等等。用这些方法我们可以在构造系统时安全地转移到下一个抽象层次。
62. 建立常量更正。这允许编译器指出一些非常细微且难以发现的错误。这项工作需要经过一定的训练，而且必须在类中协调使用，但这是值得的。
63. 充分利用编译器的错误检查功能，用完全警告方式编译我们的全部代码，修改我们的代码，直到消除所有的警告为止。在我们的代码中宁可犯编译错误也不要犯运行错误

(比如不要用变参数列表，这会使所有类型检查无效)。用**assert( )**来调试，对运行时错误要进行异常处理。

64. 宁可犯编译错误也不要犯运行错误。处理错误的代码离出错点越近越好。尽量就地处理错误而不要抛出异常。用最近的异常处理器处理所有的异常，这里它有足够的信息处理它们。在当前层次上处理我们能解决的异常，如果解决不了，重新抛出这个异常(参看第2卷)。
65. 如果我们用异常说明（参看第2卷，可从*www.BruceEckel.com*上下载，来学习有关异常处理的内容），用**set_unexpected( )** 函数安装我们自己的**unexpected( )**函数。我们的**unexpected( )**应该记录这个错误并重新抛出当前的异常。这样的话，如果一个已存在的函数被重写并且开始引起异常时，我们可以获得记录，从而修改调用代码处理异常。
66. 建立一个用户定义的**terminate( )**函数（指出一个程序员的错误）来记录引起异常的错误，然后释放系统资源，并退出程序。 772
67. 如果一个析构函数调用了任何函数，这些函数都可能抛出异常。一个析构函数不能抛出异常（这会导致**terminate( )**调用，它指出一个程序设计错误)。所以任何调用了其他函数的析构函数都应该捕获和管理它自己的异常。
68. 不要自己创建私有数据成员名字“修饰”(前面的下划线，匈牙利记号等等)，除非我们有许多已在的全局值，否则让类和名字空间来为我们做这些事。
69. 注意重载，一个函数不应该用某一参数的值来决定执行哪段代码，如果遇到这种情况，应该产生两个或多个重载函数来代替。
70. 把指针隐藏在容器类中。只有当我们要对它们执行一个立即可以完成的操作时才把它们带出来。指针已经成为出错的一大来源，当用**new**运算符时，应试着把结果指针放到一个容器中。让容器拥有它的指针，这样它就会负责清除它们。更好的方法是把指针打包进类中。如果我们仍想让它看起来像一个指针，就重载**operator->**和**operator***，如果我们必须有一个游离状态的指针，记住初始化它，最好是指向一个对象的地址，必要时让它等于0。当我们删除它时把它置0，以防意外的多次删除。
71. 不要重载全局**new**和**delete**，可以在类的基础上去重载它们。重载全局**new**和**delete**会影响整个客户程序员的项目，有些事只能由项目的创建者来控制。当为类重载**new**和**delete**时，不要假定我们知道对象的大小，有些人可能是从我们的类中继承的。用提供的参数的方法。如果我们做任何特殊的事，要考虑到它可能对继承者产生的影响。
72. 防止对象切片。实际上以值向上类型转换到一个对象毫无意义。为了防止这一点，在我们的基类中放入一些纯虚函数。 773
73. 有时简单的集中会很管用。一个航空公司的“旅客舒适系统”由一系列相互无关的因素组成：座位、空调、电视等，而我们需要在一架飞机上创建许多这样的东西。我们要创建私有成员并建立一个全部的接口吗？不，在这种情况下组件本身也是公开接口的一部分，所以我们应该创建公共成员对象。这些对象有它们自己的私有实现，所以也是很安全的。注意简单的集合不是常用的方法，但也会用到。 774

附录C 推荐读物

进一步学习的资源。

C.1 C

Thinking in C: Foundations for Java & C++，作者为Chuck Allison(该书是在本书后面的CD ROM上的MindView公司课程讨论材料，© 2000，可在*www.BruceEckel.com*找到)。这是一部包括讲演、演示幻灯片的课程，讲述准备学习Java或者C++所需的C语言的基础知识。这本书不是完全的C课程；仅仅包括学习其他语言所必须拥有的知识。附加的具体语言章节介绍要成为C++或Java程序员所必须具有的专业素质。建议读者在阅读本课程之前最好具备一种高级编程语言（如Pascal、BASIC、Fortran或者LISP）的一些实践经验（如果没有这种背景，读者也可以通过CD学习，但是本课程的设计初衷并不是要成为介绍编程基础知识的入门级读物)。

C.2 基本C++

The C++ Programming Language，第3版，作者为Bjarne Stroustrup (Addison-Wesley, 1997)[⊖]。在某种程度上，我们现在所读的这本《C++编程思想》的目的就是想让读者参考Bjarne的书。由于Bjarne的书是他作为C++这门语言的作者对C++语言所进行的描述，因此当读者想解决关于C++能做什么或不能做什么的任何不能确定的问题时，就应该去看这本书。当掌握了C++语言的诀窍并准备进一步深入探讨，就需要这本书。

C++ Primer，第3版，作者为Stanley Lippman和Josee Lajoie (Addison-Wesley, 1998)[⊖]。再没有比这更好的教学范本了。这本厚厚的书里充满了大量的细节。每当解决问题时，我就会应用这本C++ Primer和Stroustrup的书。《C++编程思想》（*Thinking in C++*）应该为理解*C++ Primer*和Stroustrup的书提供了基础。

C & C++ Code Capsules, 作者为 Chuck Allison (Prentice-Hall, 1998)。这本书假定读者已经知道C和C++，并且涉及一些读者可能已经生疏的问题，或者涉及一些读者可能以前没有正确认识到的问题。这本书填补了C和C++的一些空白。

The C++ Standard。这是委员会经过多年的努力工作所做的文件。可惜它不是免费的。但至少能够在*www.cssinfo.com*上花18美元购买PDF格式的电子版。

C.2.1 我自己的书

按照出版时间列出我所著的一些书，其中有些书因为年代久远和技术发展，现在已经不再适用。

Computer Interfacing with Pascal & C (1988年通过Eisys品牌自行出版。只有在

⊖ 本书的英文影印版已由高等教育出版社出版。本书的中文版已由机械工业出版社出版。——编辑注

⊖ 本书的中文版已由中国电力出版社出版。——编者注

*www.BruceEckel.com*上可以得到这本书)。这本书所介绍的电子学可回溯到CP/M仍然是国王而DOS已成为一步登天的新贵的时候，那时我应用高级语言并经常用计算机并口来做不同的电子项目。本书是根据我为第一个也是最好的杂志所写的专栏*Micro Cornucopia*改编的。(Larry O'Brien对它进行过解释，他是最好的计算机杂志"*Software Development Magazine*"的资深编辑——他们甚至计划在花瓶里建造一个机器人)。唉，Micro C在 Internet 出现之前早已消失。创造这本书的过程是一个非常满意的出版经历。

Using C++ (Osborne/McGraw-Hill, 1989)。这是最先讲解C++语言的图书之一。它已绝版并被第2版代替，第2版更名为**C++ Inside & Out**。

C++ Inside & Out (Osborne/McGraw-Hill, 1993)。正如上面一段所指出的，这本书是**Using C++**的第2版。这本书对C++讲解更为精确，但它毕竟是1992~1993年期间所写的书，*Thinking in C++*后来取代了它。从*www.BruceEckel.com*可以发现更多有关本书的信息并可以下载本书的源代码。

Thinking in C++, 第1版 (Prentice-Hall, 1995)。

777 **Black Belt C++, the Master's Collection**, Bruce Eckel 编辑(M&T Books, 1994)。绝版。这本书是我主持的软件开发会议C++分会上专家陈述的观点的汇总。这本书的封面鼓励我从此以后要把握自己所著的图书的封面设计。

Thinking in Java, 第2版(Prentice-Hall, 2000)[⊖]。本书第1版荣获1999年度"*Software Development Magazine*"的生产效率奖和"Java Developer's Journal"杂志的编辑选择奖。可以从*www.BruceEckel.com*下载本书的电子版。

C.3 深入研究和死角分析

本小节介绍的书深入地研究语言主题，帮助读者避免开发C++程序过程中固有的一些典型缺陷。

Effective C++(第2版，Addison-Wesley 1998)[⊖]和**More Effective C++** (Addison-Wesley, 1996)，作者为Scott Meyers。这两本书是解决C++难点问题和代码设计方面必备的经典教材。我曾经试着在《*Thinking in C++*》这本书里捕捉和表述这两本书中的思想，但是我不能自欺欺人地说我已经成功了。如果读者在学习和研究C++上花了很多时间，那么最终他们会选用这两本书。还可以从CD ROM上得到这两本书的电子版。

Ruminations on C++，作者为 Andrew Koenig和Barbara Moo (Addison-Wesley, 1996)。Andrew 在C++语言的许多方面和Stroustrup一起工作，是一个非常可靠的权威。多年来，从他的出版物和与他的接触中，我发现他不断更新的洞察力是多么的深刻，我也从他那里学到了许多东西。

Large-Scale C++ Software Design，作者为John Lakos (Addison-Wesley, 1996)。这本书讨论了做大项目时通常会碰到的问题及其答案，但是小项目也常会遇到一些类似问题，因此这本书对小项目开发也有一定的适用性。

C++ Gems，编辑为Stan Lippman (SIGS publications, 1996)。这本书包含摘选自The *C++ Report*中的一些文章。

⊖ 本书的中文版已由机械工业出版社出版。——编辑注

⊖ 本书的中文版已由华中科技大学出版社出版。——编辑注

The Design & Evolution of C++，作者为 Bjarne Stroustrup (Addison-Wesley 1994)[⊖]。本书的作者从C++创建者的角度说明为什么会作出不同的设计决策。这些内容不是不可缺少的，但是却很有趣。 778

C.4 分析和设计

Extreme Programming Explained，作者为Kent Beck (Addison-Wesley 2000)[⊖]。我喜欢这本书。是的，我倾向于采取较激进的方法，但我总觉得应该有不同的、更好的开发方法，我认为XP是很好的方法。对我有类似影响的惟一的书是*PeopleWare* (下面会进行描述)，它主要讨论环境和合作的文化。*Extreme Programming Explained*讨论编程以及许多事情，甚至讨论听说的最新“发现”。他们甚至宣称只要你不准备花费很多时间，而且愿意抛弃所做工作时，就成功了。(注意这本书封面上没有“UML stamp of approval”标志。)我可以看到这样的情况，决定某些人是否为某公司工作只是取决于他们是否使用过XP。简短的书，简短的章节，不费力地阅读，兴奋地思考。读者可以想像自己在这样的氛围里工作，会带来一个全新的视野。

UML Distilled，作者为Martin Fowler (第2版，Addison-Wesley, 2000)[⊜]。当读者第一次遇到UML，可能会感到困难，因为它有那么多的图表和细节。根据Fowler的观点，许多内容都不是必须的，所以这本书中删掉了一些内容，只留下基本部分。对于大多数项目，只需要知道一些图形工具。Fowler的目标是得到好设计而不是为如何使用工具担忧。这是一本行文简练，可读性很强的好书；如果读者想了解UML，应该首先读这本书。

The Unified Software Development Process，作者为Ivar Jacobsen、Grady Booch和James Rumbaugh (Addison-Wesley 1999)[⊛]。我本来不喜欢这本书，它看起来像恼人的大学课本。然而我惊喜地发现，书中只有一些看起来作者认为不清楚的概念解释。书的大部分不仅清楚，而且令人愉快。最重要的是，书里的过程具有实践意义。它不是极限编程（并且没有清晰的测试），但是它是UML的核心和主宰，即使读者不应用XP，他们中的大多数人也会赞同“UML是好的”（不管他们实际中应用了多少），所以读者可以采用这本书。我认为这本书堪称UML的旗舰，当读者想了解更多细节时，可以读完在Fowler的《*UML Distilled*》之后阅读它。 779

当我们选择一种方法之前，最好先听听那些不是想把这种方法出售给我们的人的观点。不真正了解我们想从中得到什么或它能为我们做什么就决定采用一种方法是很容易的。其他很多人正在用它，这看起来是一个令人信服的理由。然而，人们有一种奇怪的心理怪癖：如果他们想相信一些东西能够解决他们的问题，他们就去试（这是试验，是很好的）。但是如果它不能解决他们的问题，他们就加倍努力并大声宣布他们的伟大实践（这是一种否认事实的态度，是不好的）。这里可以做一个假设来说明问题，即如果他们将其他的人置入同一艘船中，即使什么也得不到（或者船马上就要沉没），他们也不会感到孤独。

这并不是说所有的方法学都不发挥作用，但是我们应当用一些思想工具武装起来，以帮助我们保持试验状态（“这不行，让我们试试其他的方法”），不要陷入否认事实的状态（“不，这不是一个问题，一切都挺好的，我们不需要改变”）。我认为下面的这本书，在选择一种方

⊖ 本书的中文版以及英文影印版都已由机械工业出版社出版。——编辑注
⊜ 本书的中文版已由人民邮电出版社出版。——编辑注
⊜ 本书的中文版已由清华大学出版社出版。——编辑注
⊛ 本书的中文版已由机械工业出版社出版。——编辑注

法之前，为我们提供了这些工具。

Software Creativity，作者为Robert Glass (Prentice-Hall, 1995)。这是我所见到的最好的讨论关于整个方法学问题的书。它是Glass写的或他得到的（有P.J. Plauger的贡献）一些评论和文章的汇集，反映了他多年来在这个问题上的想法和研究成果。这本书很有趣并且篇幅不
780 长，它不是四处漫游，使人厌烦的。它列出了很多的文章和研究作为参考文献。所有的程序员和管理者在陷入方法论的泥潭之前都应该读读这本书。

Software Runaways: Monumental Software Disasters，作者为Robert Glass (Prentice-Hall, 1997)。这本书的不同之处是它明确讨论了我们没有讨论过的东西：多少项目不仅失败了，而且失败得很“壮烈”。我发现大多人仍然相信“那不会发生在我身上”（或“那不会再发生了”）。我认为这样会使我们处于劣势。在头脑中时刻想着事情很可能会出错，会使我们更好地控制事情向正确的方向发展。

Object Lessons，作者为Tom Love (SIGS Books, 1993)，另一本“观点新颖”的好书。

Peopleware，作者为Tom Demarco 和Timothy Lister (Dorset House, 第2版，1999)。虽然这两位作者有软件开发的背景，但这本书是从总体上讨论项目和项目组的书。其重点是人及他们的需要，而不是技术和它的需要。这本书讨论创建一个令人愉快和高效的环境，而不是制定多少必须遵守的规则，使人成为机器上的零件。我认为，后一种态度，其最大的贡献在于，当应用XYZ方法时，使程序员微笑点头，安静地做他们总是在做的那些事情。

Complexity，作者为M. Mitchell Waldrop，(Simon & Schuster, 1992)。这本书记载了在新墨西哥州Santa Fe市，有一群不同领域的科学家聚在一起讨论的个人无法解决的实际问题（经济股票市场、生命的最初形式、为什么人们在社会中如此表现等）。通过交叉物理、经济、化学、数学、计算机科学、社会学和其他学科，建立了包含各学科的对这些问题的解决方法。但是更重要的是，一种思考这些超复杂问题的不同方法出现了：不是通过数学处理并幻想能够写出方程式，预测所有行为，而是首先观察，寻找一种模式，并且尽可能运用各种方法来模仿这个模式。（例如，这本书记载了遗传算法的出现过程。）我认为，这种思考方式是很有
781 用的，可以作为我们管理越来越复杂的软件项目的观察方法。

索　引

索引中的页码为英文原书的页码，与书中边栏的页码一致。

-, 156, 163
--, 164
!, 163
!=, 158
preprocessor stringize operator（预处理器字符串化运算符），196
#define, 194, 245, 335, 353
#endif, 245, 757
#ifdef, 194, 245
#ifndef, 246
#include , 85
#undef, 194
$<, in makefiles, 206
%, 156
&, 134, 164
&&, logical and（逻辑与），158
&, bitwise and（按位与），159
&= bitwise（按位），160
( ), overloading the function call operator（重载函数调用运算符），514
*, 156
  overloaded operator（重载的运算符），727, 730
  pointer dereference（指针间接引用），136
-, with pointers（按指针方式），192
--, with pointers（按指针方式），192
. member selection operator（成员选择运算符），237
... variable argument list（变量参数表），114
  varargs, （变量参数），243
/, 156
::, 232, 429
  scope resolution operator, and namespaces（作用域解析运算符，名字空间），417
?: ternary if-else（三重if-else），164
[ ]
  array indexing（数组下标），105
  overloaded indexing operator（重载的下标运算符），519, 698
^ bitwise exclusive-or（按位异或），159
^= bitwise（按位），160
|, bitwise or（按位或），159
||, logical or（逻辑或），158
|= bitwise（按位），160
~ bitwise not/ones complement（按位非/补），159
~, destructor（析构函数），287
+, 156, 163
  with pointers（按指针方式），192
++, 164
  with pointers（按指针方式），190
<, 158
<<, 160
  overloading for iostreams（输入输出流重载），518
<<=, 160
<=, 158
=, 166
  operator（运算符）
  as a private function（作为私有函数），533
  automatic creation（自动创建），532
  operator, as a private function（运算符，作为私有函数），709
  overloading（重载），521
==, 158, 166
>, 158
->
  overloading the smart pointer operator（重载灵巧指针运算符），509
  struct member selection via pointer（通过指针选择结

构成员），178
->*, overloading（重载），514
>=, 158
>>, 160
iostreams（输入输出流）
operator（运算符），106
overloading（重载），518
>>=, 160

A

abort( ), 409
abstract（抽象）
base classes and pure virtual functions（基类和纯虚函数），646
data type（数据类型），129, 239
abstraction,（抽象），22
access（访问，存取）
control（控制），260
run-time（运行时），275
function（函数），379
specifiers（说明符），29, 261
and object layout（对象分布），269
order for（顺序），263
accessors（访问器），380
actor, in use cases（用例中的执行者），50
addition (+)（加），156
address（地址）
const（常量），339
each object must have a unique address（每个对象必须有一个惟一的地址），241
element（元素），134
function（函数），198, 391
memory（存储器），133
object（对象），265
pass as const references（按常量引用方式传递），473
passing and returning with const（按常量方式传递和返回），349
struct object（结构对象），178
address-of (&)（...的地址），164
aggregate（集合），105
const aggregates（常量集合），337
initialization（初始化），201, 301
and structures（和结构），302
aggregation（集合），30
algorithms, Standard C++ Library（算法，标准C++库），742
aliasing（别名）
namespace（名字空间），415
solving with reference counting and copy-on-write（用引用计数和写拷贝求解），527
Allison, Chuck, 2, 776
allocation（分配）
dynamic memory allocation（动态内存分配），223, 548
memory, and efficiency（内存和效率），566
storage（存储），292
alternate linkage specification（替代连接说明），442
ambiguity（歧义性），244
during automatic type conversion（在自动类型转换期间），540
with namespaces（按照名字空间方式），420
analysis（分析）
and design, object-oriented（分析和设计，面向对象），44
paralysis（瘫痪），45
requirements analysis（需求分析），48
and（与）
& bitwise（按位），159, 166
&& logical（逻辑），158, 166
&& logical and（逻辑与），173
and_eq, &= (bitwise and-assignment)（按位与和赋值），173
anonymous union（匿名联合），320
ANSI Standard C++（ANSI标准C++），14
argc, 187
arguments（参数）
argument-passing guidelines（参数传递准则），455
command line（命令行），187, 252
const（常量），344
constructor（构造函数），286
default（默认），310, 311, 321

argument as a flag（作为标记的参数），329
destructor（析构函数），287
empty argument list, C vs.C++（空参数表，C对比C++），114
function（函数），81. 138
indeterminate list（不确定的表），114
macro（宏），374
mnemonic names（助记名），83
name decoration（名字修饰），312
overloading vs. default arguments（重载和默认参数），324
passing（传递），450
placeholder（占位符），323
preferred approach to argument passing（参数传递的首选方法），351
references（引用），451
return values, operator overloading（返回值，运算符重载），505
trailing and defaults（跟踪和默认），322
unnamed（未命名），114
variable argument list（变量参数表），114, 243
without identifiers（无标识符），323
argv, 187
arithmetic, pointer（算术，指针），190
array（数组），182
automatic counting（自动计数），301
bounds-checked, using templates（边界检查，使用模板），697
calculating size（计算长度），302
definition, limitations（定义，限制），338
indexing, overloaded operator []（索引，重载的运算符[]），698
initializing to zero（初始化为零），301
inside a class（类内部），353
making a pointer look like an array（让指针像数组），564
new & delete, 563
of pointers（指针的），187
of pointers to functions（指向函数的指针的），201
off-by-one error（"偏移1位"错误），301
overloading new and delete for arrays（为数组重载new和delete），573
pointers and（指针与），184
static（静态），692
static initialization（静态初始化），425
asctime( ), 384
assembly-language（汇编语言）
asm in-line assembly-language keyword（asm行汇编语言关键字），173
CALL, 458
code for a function call（为函数调用的代码），456
code generated by a virtual function（由虚函数生成的代码），642
RETURN, 458
assert( )
macro in Standard C（标准C中的宏），197, 223, 396
assignment（赋值），156, 301:
disallowing（不接收），522
memberwise（按成员），532, 600
operator（运算符），505
overloading（重载），521
pointer, const and non-const（指针，常量和非常量），343
self-assignment in operator overloading（在运算符重载中的自赋值），523
assure( ), 757
from require.h（来自require.h），237
atexit( ), 409
atof( ), 188, 189
atoi( ), 188
atol( ), 188
auto keyword（自动关键字），149, 414
auto-decrement operator（自减运算符），128
auto-increment operator（自增运算符），106, 128
automatic（自动）
counting, and arrays（计数和数组），301
creation of operator=（运算符=的创建），532
destructor calls（析构函数调用），297
type conversion（类型转换），228, 533
pitfalls（缺陷），539
preventing with the keyword explicit（用关键字显式防止），534

variable（变量），42, 149, 153

B

Backslash（反斜线符号），95
Backspace（后退一格），95
bad_alloc, 572
base（基）
  abstract base classes and pure virtual functions（抽象基类和纯虚函数），646
  base-class interface（基类接口），633
  fragile base-class problem（易碎的基类问题），276
  types（类型），32
  virtual keyword in derived-class declarations（在派生类声明中的虚关键字），632
basic concepts of object-oriented programming (OOP)（面向对象程序设计的基类概念），22
BASIC language（BASIC语言），68, 77
Beck, Kent, 779
behavior（行为），219
binary operators（二元运算符），160
  examples of all overloaded（所有重载的例子），493
  overloaded（重载），487
binding（捆绑）
  dynamic binding（动态捆绑），631
  early（早），38, 644
  function call binding（函数调用捆绑），631, 641
  late（晚），38, 631
  run-time binding（运行时捆绑），631
bit bucket（位桶），162
bit-shifting（位移），162
bitand, & (bitwise and)（位与），173
bitcopy（位拷贝），468
bitcopy, vs. Initialization（位拷贝，对初始化），460
bitor, | (bitwise or)（位或），173
bitwise（位或）
  and operator &（与运算符），159, 166
  const（常量），362
  exclusive-or, xor ^（异或，xor ^），159
  explicit bitwise and logical operators（显式位和逻辑运算符），173
  not ~（非~），159
  operators（运算符），159
  or operator（或运算符），159, 166
bloat, code（膨胀，代码），391
block（块）
  access（访问，存取），269
  and storage allocation（存储分配），292
  definition（定义），289
Booch, Grady, 779
book（书）
  design & production（设计和产品），18
  errors, reporting（错误，报告），16
bool（布尔），125, 195
Boolean（布尔），117, 158, 163
  algebra（代数），159
  and floating point（浮点数），159
  bool, true and false（布尔，真和假），131
bounds-checked array, with templates（边界检查的数组，用模板），697
break, keyword break（关键字），122
bucket, bit（桶，位），162
bugs（错误）
  common pitfalls with operators（与运算符有关的一般性缺陷），166
  finding（发现），292
  from casts（从类型转换），168
  with temporaries（用临时变量），348
built-in type（内部类型），129
  basic（基本的），129
  initializer for a static variable（静态变量的初始化器），408
  pseudoconstructor（伪构造函数）
  calls for（要求），589
  form for built-in types（内部类型的形式），381
byte（字节），133

C

C, 289
  #define, 340
  backward compatibility（向后兼容），73

C programmers learning C++（C程序员学习C++），628
C++ compatibi
function library（函数库），116
fundamentals（基本原理），112
heap（堆），550
hole in the type system, via void*（由于void*而产生的类型系统的漏洞），450
ISO Standard C（ISO标准C），14
Libraries（库），89, 219
Linkage（连接），338
linking compiled C code with C++（将编译过的C代码与C++连接）
name collisions（名字冲突），68
operators and their use（运算符与它们的使用），156
passing and returning variables by value（以值方式传递和返回变量），455
pitfalls（缺陷），227
preprocessor（预处理器），334
safety hole during linking（连接中的安全缺陷），314
Standard library function（标准库函数）
abort( ), 409
atexit( ), 409
exit( ), 409
Thinking in C CD ROM, 776
C++
automatic typedef for struct and class（结构和类的自动类型定义），231
C compatibility（C的兼容性），235
C programmers learning C++（C程序员学习C++），628
cfront, original C++ compiler（cfront，最早的C++编译器），237
compiling C（编译C），305
converting from C to C++（从C转变为C++），230, 760
data（数据），129
difference with C when defining variables（当定义变量时与C的差别），145
efficiency（效率），66
empty argument list, C vs. C++（空参数表，C与C++比较），114
explicit casts（显式类型转换），167
finding C errors by recompiling in C++（通过用C++重新编译而发现C错误），314
first program（第一个程序），90
GNU Compiler（GNU编译器），71
hybrid object-oriented language, and friend（混合的面向对象语言，友元），269
implicit structure address passing（隐含的结构地址传递），231
linking compiled C code with C++（用C++连接编译过的C代码），442
major language features（主要的语言特征），682
meaning of the language name（语言名字的含义），129
object-based C++（基于对象的C++），628
one definition rule（一个定义规则），244
operators and their use（运算符和它们的使用），156
programming guidelines（程序设计指导方针），760
Standard C++（标准C++），14
Standards Committee（标准委员会），14
strategies for transition to（对...的转换策略），68
stricter type checking（严格类型检查），227
strongly typed language（强类型语言），450
why it succeeds（为什么它成功），64
calculating array size（计算数组长度），302
CALL, assembly-language（CALL，汇编语言），458
calling a member function for an object（对于一个对象调用成员函数），239
calloc( ), 223, 550, 554
Carolan, John, 277
Carroll, Lewis, 277
case（案例），124
cassert standard header file（cassert标准头文件），197
cast（类型转换），40, 135, 164, 276, 552, 630
C++ explicit casts（C++显式类型转换），167
casting away constness（强制转换const），363
casting void pointers对void（指针进行类型转换），235
const_cast（常量类型转换），170
explicit cast for upcasting（显式向上类型转换），681

explicit keyword（显式关键字），678
operators（运算符），166
pointer assignment（指针参数），343
reinterpret cast（重新解释类型转换），171
static_cast（静态类型转换），169
cat, Cheshire, 277
catch clauses catch（子句），572
CD ROM
seminars on CD-ROM from MindView（MindView 的 CD-ROM 上的研讨会），16
Thinking in C, Foundations for Java & C++ (packaged with book), 2, 15, 112
cfront, original C++ compiler（cfront最早的C++编译器），237
chapter overviews（章节总览），7
char, 96, 130, 132
sizeof, 173
character（字符），154
array literals（数组的字面值），343
character array concatenation（字符数组的拼接），96
constants（常量），155
characteristics（特性），219
check for self-assignment in operator overloading（运算符重载的自赋值检查），505
Cheshire cat, 277
cin, 97
clashes, name（冲突，名字），229
class（类），25, 76, 271
abstract base classes and pure virtual functions（抽象基类和纯虚函数），646
adding new virtual functions in the derived class（在派生类中增加新的虚函数），652
aggregate initialization（集合初始化），302
class definition and inline functions（类定义和内联函数），378
compile-time constants inside（编译时内部常量），353, 356, 358
composition, and copy-constructor（组合和拷贝构造函数），369
const and enum in（常量和枚举），353
container class templates and virtual functions（容器类模板和虚函数），743
creators（创建者），28
declaration（声明），277
of a nested friend class（一个嵌套的友元类的），514
defining the interface（定义接口），62
definition（定义），277
difference between a union and a class（联合与类之间的区别），319
duplicate class definitions and templates（复制类定义和模板），699
fragile base-class problem（易碎的基类问题），276
generated by macro（由宏生成的），594
generated classes for templates（为模板生成的类），699
handle class（句柄类），275
inheritance and copy-constructor（继承和拷贝构造函数），471
diagrams（图表，图），617
initialization, memberwise（初始化，按成员），471
instance of（为例），24
keyword（关键字），31
local（局部），428
nested（嵌套的），428
iterator（迭代器），512, 721
overloading new and delete for a class（为类重载new和delete），570
pointers in, and overloading operator=（在...中的指针，重载运算符=），524
static class objects inside functions（函数内的静态类对象），408
static data members（静态数据成员），423
static member functions（静态成员函数），429
templates（模板），742
using const with（用常量），352
class-responsibility-collaboration (CRC) cards（类职责协同（CRC）卡片），52
cleanup（清除），227, 666
automatic destructor calls with inheritance and composition（带有继承和组合的自动析构函数调用），592
initialization and cleanup on the heap（在堆上的初始

化和清除），548
client programmer（客户程序员），28, 260
code（代码）
source availability（源代码的可用性），12
table-driven（表驱动的），201
assembly for a function call（用于函数调用的汇编），456
bloat（膨胀），391
comment tags in listings（在清单中的注释标记），750
consulting, mentoring, and design and code walkthroughs from MindView（来自MindView公司的咨询、指导和代码演练），16
generator（生成器），79
organization（组织机构），248
header files（头文件），244
program structure when writing code（写代码时的程序结构），93
re-use（重用），583
collection（收集），510, 719
collector, garbage（收集器，无用单元），42
collision, linker（冲突，连接器），244
comma operator（逗号运算符），165, 508
command line（命令行），252
arguments（参数），187
comment tag（注释标记）
for linking为了连接），148
in source-code listings（在源代码清单中），750
comments, makefile（注释，makefile文件，），204
committee, C++ Standards（委员会，C++标准），14
common interface（公共接口），647
compaction, heap（压缩，堆），225
compatibility（兼容性）
C & C++, 235
with C（用C），98
compilation（编译）
needless（不需要的），276
process（过程，进程），79
separate（分开），78
separate, and make（分开和安排），202
compile time constants（编译时常量），335
compiler（编译器），76, 77
creating default constructor（创建默认构造函数），304
original C++ compiler cfront（最早的C++编译器cfront），237
running（运行），95
support（支持），15
compiling C with C++（用C++编译C），305
compl, ~ ones complement（补），173
complicated（复杂的）
declarations & definitions（声明和定义），199
expressions, and operator overloading（表达式，运算符重载），488
composite（组合）
array（数组），182
type creation（类型创建），174
composition（组合），30, 584, 607
combining composition & inheritance（组合与继承结合），591
copy-constructor（拷贝构造函数），469
member object initialization（成员对象初始化），589
vs. inheritance（对应于继承），604, 620, 740
concatenation, character array（拼接，字符数组），96
concept, high（概念，高），48
conditional operator（条件运算符），164
conditional, in for loop（有条件的，在for循环），121
const（常量），153, 334
address of（...的地址），339
aggregates（集合），337
casting away（强制转换），363
character array literals（字符数组字面值），343
compile-time constants in classes（在类中的编译时常量），356
const reference function arguments（常量引用函数参数），351
correctness（正确性），367
enum in classes（在类中的枚举），353
evaluation point of（...的评价点），337
extern（外部的），339
function arguments and return values（函数参数和返回值），344
in C（在C中），338

initializing data members（初始化数据成员），355
logical（逻辑的），362
member function（成员函数），352
and objects（和对象），359
mutable（易变的），362
pass addresses as const references（作为常量引用传递地址），473
pointer to const（指向常量的指针），171
pointers（指针），340
reference（引用），345, 453
and operator overloading（运算符重载），505
return by value as const（以值作为常量返回），345
and operator overloading（运算符重载），507
safety（安全），336
temporaries are automatically const（临时和自动常量），347
const_cast（常量类型转换），170
constant（常量），153
character（字符），155
compile-time（编译时），335
inside classes（类内），358
folding（折叠），335, 339
named（命名），153
templates, constants in（模板，编译时常量），703
values（值），154
constructor（构造函数），285, 548, 551, 665
arguments（参数），286
automatic type conversion（自动类型转换），534
behavior of virtual functions inside constructors（在构造函数内的虚函数行为），664
copy-constructor（拷贝构造函数），432, 450, 455, 463, 657
alternatives to（选择），471
vs. operator=（和，运算符=），521
creating a new object from an existing object（由已经存在的对象创建一个新对象），462
default（默认），304, 327, 408, 470, 563
inheritance（继承），663
synthesized by the compile（由编译器综合），304
doesn't automatically inherit（不能自动继承），600
efficiency（效率），663
global object（全局对象），410
initialization and cleanup on the heap（在堆上的初始化和清除），548
initializer list（初始化表），353, 589, 664
pseudoconstructors（伪构造函数），589
inline（内联），392
installing the VPTR（安装VPTR），643
memberwise initialization（按成员初始化），600
name（名字），285
new operator, memory exhaustion（new运算符，内存用完），576
order of construction with inheritance（继承构造函数的顺序），665
order of constructor call（构造函数调用的顺序），663
and destructor calls（和析构函数调用顺序），592
overloading（重载），310, 319
private（私有），709
pseudo-constructor（伪构造函数），562
return value（返回值），287
tracking creations and destructions（跟踪创建和销毁），709
virtual functions & constructors（虚函数和构造函数），662
consulting, mentoring, and design and code walkthroughs from MindView（来自MindView公司的咨询、指导和代码演练），16
container（容器），510, 719
container class templates and virtual functions（容器类模板和虚函数），743
delete（删除），671
iterators（迭代器），690
new, delete, and containers（new、delete和容器），692
ownership（所有权），671, 713
polymorphism（多态的），738
Standard C++ Library（标准C++库），104
Vector（矢量），102
context, and overloading（上下文，和重载），310
continuation, namespace（继续，名字空间），415
continue, keyword （continue, 关键字），122
control（控制）

access（访问，存取），29, 260
run-time（运行时），275
access specifiers（访问说明符），261
expression, used with a for loop（表达式，用for循环），106
controlling（控制）
execution（执行），117
linkage（连接），412
conversion（转换）
automatic type conversion（自动类型转换），533
narrowing conversions（窄变换），170
pitfalls in automatic type conversion（自动类型转换中的缺陷），539
preventing automatic type conversion with the keyword explicit（用关键字显式防止自动类型转换），534
to numbers from char*（从char*到数字），188
converting from C to C++（从C到C++的转变），230, 760
copy-constructor（拷贝构造函数），432, 450, 455, 463, 508, 657, 730
alternatives（选择），471
composition（组合），469
default（默认），468
inheritance（继承），471
private（私有的），471, 709
upcasting and the copy-constructor（向上类型转换和拷贝构造函数），617
vs. operator=（和运算符=），521
copy-on-write (COW)（写拷贝），527
copying pointers inside classes（在类内拷贝指针），524
copyright notice, source code（版权注意，源代码），12
correctness, const（正确性，常量），367
costs, startup（成本，启动），71
counting（计数）：
automatic, and arrays（自动的，和数组），301
reference（引用），526
cout, 90, 91
cover design, book（封面设计，书），17
CRC, class-responsibility-collaboration cards（CRC，类职责协同卡片），52
creating（创建）
functions in C and C++（C和C++中的函数），112
new object from an existing object（由已经存在的对象生成的新对象），462
objects on the heap（在堆上的对象），554
crisis, software（危机，软件），8
cstdlib standard header file cstdlib（cstdlib标准头文件），188
cstring standard header file cstring（cstring标准头文件），269
c-v qualifier（c-v限定词），366

D

data（数据）
defining storage for static members（为静态成员定义存储），424
initializing const members（初始化常量成员），355
static area（静态区域），406
static members inside a class（在类中的静态成员），423
data type（数据类型）
abstract（抽象），129, 239
built-in（内部的），129
equivalence to class（与类相同），26
user-defined（用户定义的），129
debugging（调试），78
assert() macro（assert()宏），197
flags（标记），194
preprocessor flags（预处理器标记），194
require.h, 396
run-time（运行时），195
using the preprocessor（使用预处理器），395
decimal（十进制），154
declaration（声明），81
all possible combinations（所有可能的结合），141
analyzing complex（分析复杂性），199
and definition（和定义），243
class（类），277
nested friend（嵌套的友元），514
const（常量），340
forward（向前），151
function（函数），116, 233, 313

declaration syntax（声明语法），82
not essential in C（在C中不是基本的），228
header files（头文件），242, 244
structure（结构），265
using, for namespaces using（使用，使用名字空间），421
variable（变量）
declaration syntax（声明语法），83
point of declaration & scope（声明指针和作用域），145
virtual（虚），632
base-class declarations（基类声明），632
derived-class declarations（派生类声明），632
decoration, name（修饰，名字），230, 231, 237, 442
overloading（重载），311
decoupling（隔离），628
via polymorphism（通过多态），69
decrement（减一），128, 164
and increment operators（加一运算），506
overloading operator（重载运算符），493
default（默认）
argument（参数），310, 311, 321
as a flag（作为标记），329
vs. overloading（重载），324
constructor（构造函数），304, 327, 408, 470, 563
inheritance（继承），663
copy-constructor（拷贝构造函数），468
default values in templates（模板中的默认值），703
keyword（关键字），124
defining（定义）
function pointer（函数指针），198
initializing at the same time（同时初始化），290
initializing variables（初始化变量），130
variable（变量），145
anywhere in the scope（在作用域的无论何处），145
definition（定义），81
array（数组），338
block（块），289
class（类），277
complex function definitions（复杂函数定义），198
const（常量），340
declaration（声明），243
duplicate class definitions and templates（重复类定义和模板），699
formatting pointer definitions（格式化指针定义），342
function（函数），83
non-inline template member function definitions（非内联模板成员函数定义），699
object（对象），285
pure virtual function definitions（纯虚函数定义），651
storage for static data members（静态数据成员的存储），424
structure definition in a header file（在头文件中的结构定义），234
delete（删除），164, 223, 553
calling delete for zero（对零调用删除），327
delete-expression（删除表达式），553, 566
keyword（关键字），42
multiple deletions of the same object（同一对象的多次删除），553
new
and containers（和容器），692
for arrays（为数组），563
overloading new and delete（重载new和 delete），566
array（数组），573
class（类），570
global（全局的），568
void*, deleting is a bug（void*,删除是错误的），555
zero pointer（零指针），553
Demarco, Tom, 781
dependency（相关性，依赖性）
makefile（程序的描述文件），204
static initialization（静态初始化），432
deprecation, of ++ with a bool flag（反对，带有布尔标记的++），131
dereference（间接引用）
*, 164
dereferencing function pointers（间接引用函数指针），200
pointer（指针），137
derived（派生）
adding new virtual functions in the derived class（在派

生类中增加新虚函数），652
types（类型），32
virtual keyword in derived-class declarations（在派生类声明中的虚关键字），632
design（设计）
analysis and design, object-oriented（分析和设计，面向对象），44
book（书）
cover（封面），17
design and production（设计和生产），18
consulting, mentoring, and design and code walkthroughs from MindView（来自MindView公司的咨询、指导和代码演练），16
five stages of object design（对象设计的五个步骤），54
inlines（内联），380
mistakes（错误），279
pattern, iterator（模式，迭代器），719
patterns（模式），59, 70
destructor（析构函数），287
automatic destructor calls（自动析构函数调用），297
with inheritance and composition（用继承和组合），592
doesn't automatically inherit（不能自动继承），600
explicit destructor call（显式析构函数调用），579
initialization and cleanup on the heap（在堆上的初始化和清除），548
inlines（内联），392
order of constructor and destructor calls（构造函数和析构函数调用的顺序），592
pure virtual destructor（纯虚析构函数），668
scope（作用域），288
static objects（静态对象），410
tracking creations and destructions（跟踪创建和销毁，709
virtual destructor（虚构造函数），665, 707, 736, 740
virtual function calls in destructors（在析构函数中的虚函数调用），670
development, incremental（开发，渐增式的），614
diagram（图表，图）
class inheritance diagrams（类继承图），617
inheritance（继承），40
use case（用例），49
directive（指令）
preprocessor（预处理器），79
using, namespaces（使用，名字空间），92, 418
header files（头文件），247
directly accessing structure（直接访问结构），240
disallowing assignment（不允许赋值），533
dispatching, double/multiple（指派，两重/多重），675
division (/)（除法），156
do-while, 120
double（两重），155
dispatching, and multiple dispatching（指派，和多重指派），675
double precision floating point（双精度浮点），130
internal format（内部格式），189
downcast（向下类型转换）
static_cast（静态类型转换），681
type-safe（类型安全），678
duplicate class definitions and templates（重复类型定义和模板），699
dynamic（动态的）
binding（捆绑），631
memory allocation（内存分配），223, 548
object creation（对象创建），42, 547, 732, 738
type checking（类检查），80
dynamic_cast（动态类型转换），678

E

early binding（早捆绑），38, 631, 641, 644
edition, 2nd, what's new in（版本，第2版，其中什么是新的），2
efficiency（效率），371
C++, 66
constructor（构造函数），663
creating and returning objects（创建和返回对象），507
inlines（内联），392
memory allocation（内存分配），567
references（引用），455

trap of premature optimization（过早优化的陷阱），329
virtual functions（虚函数），645
elegance, in programming（优美的，在程序设计中），60
Ellis, Margaret, 433
else（另外），118
embedded（嵌入的）
Object（对象），585
systems（系统），577
encapsulation（封装），239, 270
end sentinel, iterator（终止哨兵，迭代器），724, 728, 736
enum（枚举）
and const in classes（和类中常量），353
clarifying programs with（用...阐明程序），179
hack, 358
incrementing（增量，增加），180
keyword（关键字），179
type checking（类型检查），180
untagged（无标记的），320, 358
equivalence（等价），166
==, 158
error（错误）
exception handling（异常处理），43
off-by-one（偏移1位），301
preventing with common header files（用公共头文件防止），244
reporting errors in book（报告书中的错误），16
structure redeclaration（结构重声明），245
escape sequences（转义序列），94
evaluation order, inline（赋值顺序,内联），391
evolution, in program development（进化，在程序开发中），58
exception handling（异常处理），43, 565
simple use（简单使用），572
executing code（执行代码）
after exiting main()（退出main()之后），411
before entering main()（进入main()之前），411
execution（执行）
controlling（控制），117
point（点），549
exercise solutions（练习答案），12
exit(), 397, 409
explicit（显式的）
cast（类型转换），678
C++, 167
for upcasting（向上类型转换），681
keyword to prevent automatic type conversion（防止自动类型转换的关键字），534
exponential（指数），154
notation（符号），130
exponentiation, no operator（求幂，无运算符），517
expressions, complicated, and operator overloading（表达式，复杂的和运算符重载），488
extending a class during inheritance（在继承中扩展类），34
extensible program（可扩展的程序），633
extern（外部的），84, 147, 151, 335, 339, 442
const（常量），335, 340
to link C code（连接C代码），442
external（外部的）
linkage（连接），152, 338, 339, 412
references, during linking（引用，连接期间），228
extractor and inserter, overloading for iostreams（提取器和插入器，输入输出流重载），518
eXtreme Programming (XP)（极限编程），61, 615, 779

F

factory, design pattern（工厂，设计模式），712
false（假），158, 163, 246
and true, in conditionals（和真，在条件中），117
bool, true and false（布尔，真和假），131
fan-out, automatic type conversion（扇出，自动类型转换），540
Fibonacci（斐波纳契），725
fibonacci(), 691
file（文件）
header（头），233, 242, 323
code organization（代码组织），248
const（常量），335
namespaces（名字空间），423

names（名字），749
reading and writing（读和写），100
scope（作用域），150, 152, 412
static（静态的），150, 244, 414
structure definition in a header file（在头文件中的结构定义），234
flags, debugging（标记，调试），194
floating point（浮点）
float（流），130, 155
float.h, 129
internal format（内部格式），189
number size hierarchy（数字长度层次），132
numbers（数字），130, 154
true and false（真和假），159
for
defining variables inside the control expression（在控制表达式中定义变量），145
loop（循环），106, 121
loop counter, defined inside control expression（循环计数器，在控制表达式中定义的），291
variable lifetime in for loops（在for 循环中的变量生命期），292
formatting pointer definitions（格式化指针定义），342
forward（向前/前向）
declaration（声明），151
reference, inline（引用，内联），391
Fowler, Martin, 45, 58, 779
fragile base-class problem（易碎的基类问题），276
fragmentation, heap（碎片，堆），225, 567
free store（释放存储），549
free( ), 223, 550, 553, 555, 569
free-standing reference（独立引用），451
friend（友元），263, 554
declaration of a nested friend class（嵌套定义类的声明），514
global function（全局函数），264
injection into namespace（加入名字空间），417
member function（成员函数），264
nested structure（嵌套结构），266
structure（结构），264
fstream, 100
function（函数），81
abstract base classes and pure virtual functions（抽象基类和纯虚函数），646
access（存取，访问），379
adding more to a design（向设计增加更多的内容），280
adding new virtual functions in the derived class（在派生类中增加新的虚函数），652
address（地址），198, 391
argument（参数），138
const（常量），344
const reference（常量引用），341
reference（引用），451
array of pointers to（指向...的指针数组），201
assembly-language code generated（生成的汇编语言代码）
function call（函数调用），456
virtual function call（虚函数调用），642
binding, for a function call（捆绑，为函数调用）631, 641
body（体），83
C library（C库），116
call operator( )（call 运算符( )），614
call overhead（调用开销），372, 377
called for side effect（有副作用的调用），313
complicated function definitions（复杂函数定义），198
constructors, behavior of virtual functions inside（构造函数，虚函数内部的行为），664
creating（创建），112
declaration（声明），116, 245, 313
not essential in C（不是C中基本的），228
required（要求），233
syntax（语法，句法），82
definition（定义），83
empty argument list, C vs.C++（空参数表，C与C++比较），114
expanding the function interface（扩展函数接口），330
global（全局的），234
friend（友元），264

helper, assembly（协助，汇编），457
inline（内联），372, 377, 646
header files（头文件），396
local class (class defined inside a function)（局部类（在函数内定义的类）），428
member function（成员函数），28, 230
calling（调用）
a member function（成员函数），239
another member function from within a member function（在成员函数中的另一个成员函数），234
base-class functions（基类函数），588
const（常量），352, 359
friend（友元），264
inheritance and static member functions（继承和静态成员函数），604
overloaded operator（重载运算符），487
selection（选择），234
objects（对象），515
overloading（重载），310
operator（运算符），486
using declaration, namespaces（使用声明，名字空间），421
overriding（重写），35
pass-by reference & temporary objects（按照引用方式传递，临时对象），453
pointer（指针）
defining（定义），198
to member function（对成员函数），475
using a function pointer（使用函数指针），200
polymorphic function call（多态函数调用），637
prototyping（原型化），113
pure virtual function definitions（纯虚函数定义），651
redefinition during inheritance（在继承期间的重定义），588
return value（返回值）
by reference（按照引用方式），451
returning a value（返回值），115
type（类型），597
void, 115
signature（特征），597
stack frame for a function call（函数调用的栈结构），458
static（静态）
class objects inside functions（在函数中的类对象），408
member（成员），366, 429, 465
objects inside functions（在函数中的对象），437
variables inside functions（在函数中的变量），406
templates（模板），742
type（类型），390
unique identifier for each（每个有惟一的标识符），310
variable argument list（变量参数表），114
virtual function（虚函数），627, 629
constructor（构造函数），662
overriding（重写），632
picturing（画），639

G

garbage collector（无用单元收集器），42, 566
generic algorithm（泛型算法），742
get and set functions（get和set函数），381
get( ), 472
getline( )
and string, 562
from iostreams library（从输入输出流库），100
Glass, Robert, 780
global（全局）
friend function（友元函数），264
functions（函数），234
new and delete, overloading（new和delete重载），568
object constructor（对象构造函数），410
operator, overloaded（运算符，重载），487
scope resolution（作用域解析），253
static initialization dependency of global objects（全局对象的静态初始化依赖），432
variables（变量），147
GNU C++, 71
Gorlen, Keith, 694
goto, 125, 288, 293

non-local（非局部的），288
greater than（大于）
>, 158
or equal to (>=)（或等于），158
guaranteed initialization（保证初始化），294, 548
guards, include, on header files（防护，包含，在头文件上），757
guidelines（指导方针）
argument passing（参数传递），455
C++ programming guidelines（C++程序设计指导方针），760
object development（对象开发），56

H

hack, enum（hack,枚举），358
handle classes（句柄类），275, 277
has-a（有一个），30
composition（组合），604
header file（头文件），85, 116, 129, 233, 242, 323, 335
code organization（代码组织），248
enforced use of in C++（在C++中强制使用），243
formatting standard（格式化标准），246
include guards（包含防护），246
inline definitions（内联定义），377
internal linkage（内部连接），412
namespaces（名字空间），423
new file include format（新文件包含格式），86
order of inclusion（包含顺序），756
templates（模板），700, 707
using directives（使用指令，using指令），248
importance of using a common header file（使用公共头文件的重要性），242
multiple inclusion（多次包含），244
structure definition in a header file（在头文件中的结构定义），242
heap（堆），42, 223
C heap（C堆），550
compactor（压缩器），225
creating objects（创建对象），554
fragmentation（分裂，碎片），225, 567
guaranteeing that all objects are created on the heap（保证所有的对象在堆上创建），712
storage allocation（存储分配），549
simple example system（简单例子系统），570
helper function, assembly（协助函数，汇编），457
hexadecimal（十六进制），154
hiding（隐藏）
function names inside a struct（在结构中的函数名），230
implementation（实现），28, 260, 270, 275
names（名字）
during inheritance（在继承期间），595
during overloading（在重载期间），658
variables from the enclosing scope（在封闭作用域的变量），292
hierarchy, singly-rooted/object-based（层次结构，单个根/基于对象），672, 694
high concept（高级概念），48
high-level assembly language（高级汇编语言），113
hostile programmers（不友好的程序员），276
hybrid（混合）
C++, hybrid object-oriented language, and friend（C++混合的面向对象语言，和友元），269
object-oriented programming language（面向对象程序设计），7

I

Identifier（标识符）
unique for each function（每个函数有惟一的），310
unique for each object（每个对象有惟一的），238
IEEE standard for floating-point numbers（浮点数的IEEE标准），130, 189
if-else, 118
defining variables inside the conditional statement（在条件语句中定义变量），145
ternary ?:（三元操作符 ?:），164
ifstream, 100, 606
implementation（实现），27, 241
and interface, separating（和接口，分隔），29, 261, 271, 380

hiding（隐藏），28, 260, 270, 275
compile-time only（只在编译时），275
implicit type conversion（隐含的类型转换），154
in situ inline functions（in situ内联函数），394
in-memory compilation（在内存中编译），78
include（包含），85
include guards, in header files（包含防止，在头文件中），246, 757
new include format（新包含格式），86
incomplete type specification（不完全类规范说明），265, 277
increment（加一），128, 164
and decrement operators（和减一运算），506
incrementing and enumeration（增加和枚举），180
overloading operator ++（重载运算符++），493
incremental（加一）
development（开发），614
programming（程序设计），614
indeterminate argument list（不确定参数表），114
indexing（索引，下标）
array, using [ ]（数组,用[ ]），105, 183
zero（零），183
inheritance（继承），31, 584, 586, 615
choosing composition vs. Inheritance（选择组合与继承），604
class inheritance diagrams（类继承图），617
combining composition & inheritance（结合组合与继承），591
copy-constructor（拷贝构造函数），471
diagram（图），40
extending a class during（在…中扩展类），34
extensibility（可扩展性），633
function redefinition（函数重定义），588
initialization（初始化），663
is-a（是一个），600, 615
multiple（多个），586, 613, 621, 673, 695
name hiding（名字隐藏），658
operator overloading & inheritance（运算符重载和继承），612
order of construction（构造顺序），665
private inheritance（私有继承），609
protected inheritance（保护继承），611
public inheritance（公共继承），587
static member functions（静态成员函数），604
subtyping（子类型定义），606
virtual function calls in destructors（在析构函数中的虚函数调用），670
vs. Composition（和组合），620, 740
VTABLE, 652
initialization（初始化），227, 356
aggregate（集合），201, 301
array（数组）
elements（元素），301
to zero（为零），301
const data members（常量数据成员），355
const inside class（在类中的常量），353
constructor（构造函数），285
constructor initializer list（构造函数初始化表），353, 589, 664
definition, simultaneous（定义，并发），290
for loop（for循环），106, 121
guaranteed（保证），294, 548
during inheritance（在继承期间），663
initialization and cleanup on the heap（在堆上的初始化和清除），548
initializer for a static variable of a built-in type（内部类型的静态变量的初始化器），408
lazy（懒惰），704
member object initialization（成员对象初始化），589
memberwise（按成员），471, 600
object using =（对象using =），521
static（静态）
array（数组），425
const（常量），356
dependency（依赖性），432
member（成员），425
zero initialization by the linking-loading mechanism（用连接分配机制初始化为零），433
variables at point of definition（在定义点上的变量），130
vs. Bitcopy（和位拷贝），460
injection, friend into namespace（插入，友元进入名字空

间），417
inline（内联），394, 662
class definition（类定义），1378
constructor efficiency（构造函数效率），663
constructors（构造函数），392
convenience（方便），393
definitions and header files（定义和头文件），377
destructors（析构函数），392
effectiveness（有效性），390
efficiency（效率），392
function（函数），372, 377, 646
header files（头文件），396
in situ, 394
limitations（限制），390
non-inline template member function definitions（非内联模板成员函数定义），699
order of evaluation（赋值顺序），391
templates（模板），707
input（输入）
reading by words（逐字阅读），106
standard（标准），97
insert( ), 104
inserter and extractor（插入器和提取器），overloading for iostreams（重载输入输出流），518
instance of a class（类的实例），24
instantiation, template（实例化，模板），699
int, 130
interface（接口），241
base-class interface（基类接口），633
common interface（公共接口），647
defining the class（定义类），62
expanding function interface（扩展函数接口），330
for an object（对于对象），25
implementation, separation of（实现，分隔），29, 261, 271, 380
implied by a template（由模板暗示），701
user（用户），51
internal linkage（内部连接），152, 335, 339, 412
interpreters（解释器），77
interrupt service routine (ISR)（中断服务程序），366, 458
iostreams（输入输出流），90
get( ), 472
getline( ), 100
global overloaded new & delete（全局重载new和delete）
interaction with（与...交互），572
limitations of（...的限制），569
manipulators（操作者），96
overloading << and >>（重载<<和>>），518
reading and writing files（读和写文件），100
reading input（读输入），97
setf( ), 466
strings with iostreams（带输入输出流的字串），100
width( ), 466
is-a（是一个）
inheritance（继承），604, 615
vs. is-like-a relationships（像…的关系），35
ISO Standard（ISO标准）
C, 14
fundamentals（基本的），112
C++, 14
header files（头文件），245
istream, overloading operator >>（输入流，重载运算符>>），520
iteration, in program development（迭代，在程序开发中），57
iterator（迭代器），509, 719, 730
containers（容器），690
motivation（动机），738
nested class（嵌套的类），512
Standard C++ Library（标准C++库），724

J

Jacobsen, Ivar, 779
Java, 3, 15, 65, 71, 74, 588, 645, 694, 816

K

K&R C, 112

keywords（关键字）
#define, 245, 335
#endif, 245, 757
#ifdef, 245
#include, 85
&, 134
(), function call operator overloading（函数调用操作符重载）, 514
*, 136, 164
'.' (member selection operator)（成员选择运算符）, 237
=, 156
overloading（重载）, 505, 521
->, 164
overloading（重载）, 509
struct member selection via pointer（通过指针的结构成员选择）, 178
->*, 474
overloading（重载）, 514
.*, 474
::, 232, 253
asm, for in-line assembly language（asm, 在线汇编语言）, 173
auto（自动）, 109, 414
bool（布尔）, 125
true and false（真和假）, 131
break, 122
case, 124
catch, 572
char, 96, 130, 132
class（类）, 25, 31, 271
const（常量）, 153, 333, 453
const_cast（常量类型转换）, 170
continue（继续）, 122
default（默认）, 124
delete（删除）, 42, 223
do, 120
double（加倍）, 130, 132
dynamic_cast（动态类型转换）, 678
else, 118
enum（枚举）, 179, 358
untagged（无标记）, 320
explicit（显式的）, 534
extern（外部的）, 84, 147, 151, 335, 339, 412
for alternate linkage（另外的连接）, 442
false（假）, 117, 131
float（浮点）, 130, 132
for, 106, 121
friend（友元）, 263
goto, 125, 288, 293
if, 118
inline（内联）, 394, 662
int, 130
long（长）, 132
long double（长双精度）, 132
long float (not legal)（长浮点数(不合法)）, 132
mutable（可变的）, 363
namespace（名字空间）, 91, 414, 757
new, 42, 223
operator（运算符）, 486
private（私有的）, 262, 270, 380, 610
protected（保护）, 263, 270, 610
public（公共）, 261
register（寄存器）, 149, 414
reinterpret_cast（重解释类型转换）, 171
return（返回）, 115
short, 132
signed（有符号的）, 132
signed char（有符号的字符）, 132
sizeof, 132, 172, 587
with struct, 240
static（静态）, 149, 350, 406
static_cast（静态类型转换）, 169, 679
struct（结构）, 175, 20
switch, 123, 293
template（模板）, 689, 696
this, 234, 286, 363, 380, 429
throw, 572
true（真）, 117, 131
try, 572
typedef（类型定义）, 174
typeid（定义类型）, 680

union（联合），181, 318
anonymous（匿名），320
unsigned（无符号的），132
using使用, 92, 417
virtual（虚），39, 595, 627, 632, 637, 646, 665
void, 114
void& (illegal)（void(不合法)），143
void*, 142, 450
volatile（可变的/易变的），155
while, 101, 119
Koenig, Andrew, 376, 762, 778

L

Lajoie, Josee, 776
Lakos, John, 756, 778
Language（语言）
C++ is a more strongly typed language（C++是类型更严格的语言），450
C++, hybrid object-oriented language, and friend（C++，混合的面向对象语言，和友元），269
hybrid object-oriented programming language（混合的面向对象程序设计语言），7
large programs, creation of（大程序，的创造），78
late binding（晚捆绑），38, 631
implementing（实现），636
layout, object, and access control（规划，对象，和访问控制），269
lazy initialization（懒惰初始化），704
leading underscore, on identifiers (reserved)（下划线前面，在标识符上（保留名）），381
leaks, memory（泄漏，内存），224, 300
left-shift operator（<<左移运算符<<），160
less than
<, 158
or equal to <=（小于或等于），158
library（库），76, 80, 88, 218
C, 219
code（代码），78
creating your own with the librarian（用库管理程序创建自己的库），117
issues with different compilers（不同编译器有关的问题），312
Standard C function（标准C函数）
abort( ), 409
atexit( ), 409
exit( ), 409
lifetime（生命期）
for loop variables（for循环变量），292
object（对象），42, 547
temporary objects（临时对象），468
limits.h, 129
linkage（连接），152, 406
alternate linkage specification（替代连接说明），442
controlling（控制），412
external（外部的），335, 339, 412
internal（内部的），335, 339, 412
no linkage（不连接），153, 412
type-safe（类型安全），313
linked list（连接表），248, 275, 298
linker（连接器），78, 79, 87
collision（冲突），244
external references（外部引用），228
object file order（目标文件顺序），88
searching libraries（库搜索），88, 117
unresolved references（未解决的引用），88
Lippman, Stanley, 776
list（表）
constructor initializer（构造函数初始化器），353, 589
linked（连接的），248, 275, 298
Lister, Timothy, 781
local（局部的）
array（数组），186
classes（类），428
static object（静态对象），410
variable（变量），138, 149
logarithm（对数），466
logical（逻辑）
and &&, 166
const（常量），362
explicit bitwise and logical operators（显式的按位和按

逻辑运算符），173
not !, 163
operators（非运算符），158, 505
or ||, 166
long（长），132, 135
long double（长双精度），132, 155
longjmp( ), 288
loop（循环）
for, 106
loop counter, defined inside control expression（循环计数器，定义的内部控制表达式），291
variable lifetime in for loops（在for循环中的变量生命期），292
while, 101
Love, Tom, 781
lvalue（左值），156, 346, 698

M

machine instructions（机器指令），76
macro（宏）
argument（参数），374
makefile（程序的描述文件），205
preprocessor（预处理器），158, 192, 372
macros for parameterized types, instead of templates（针对参数化类型的宏，而不用模板），696
unsafe（不安全），399
to generate classes（生成类），594
magic numbers, avoiding（幻数，避免），334
main( )
basic form（基本形式），93
executing code after exiting（退出后的执行代码），411
executing code before entering（进入前的执行代码），411
maintenance, program（维护，程序），58
make, 202
dependencies（相关性，依赖性），204
suffix rules（后缀规则），205
SUFFIXES, 206
macros（宏），205
makefile, 203, 750
malloc( ), 223, 550, 552, 554, 569
behavior, not deterministic in time（行为，在时间上非确定的），555
management obstacles（管理障碍），71
mangling, name（破坏，名字），230, 231, 237
and overloading（和重载），311
mathematical operators（数学运算符），156
Matson, Kris C., 126
member（成员）
defining storage for static data member（为静态数据成员定义存储），424
initializing const data members（初始化常量数据成员），355
member function（成员函数），28, 230
calling（调用），239
calling another member function from within a member function（在一个成员函数内调用另一个成员函数），234
const（常量），352, 359
four member functions the compiler synthesizes（编译器综合的四个成员函数），619
friend（友元），264
non-inline template member function definitions（非内联模板成员函数定义），699
return type（返回类型），597
selection（选择），234
signature（特征），597
static（静态的），366, 429, 465
and inheritance（和继承），604
object（对象），30
object initialization（对象初始化），589
overloaded member operator（重载成员运算符），487
pointers to members（指向成员的指针），473
selection operator（选择运算符），237
static data member inside a class（在类中的静态数据成员），423
vs. non-member operators（和非成员运算符），518
memberwise（按成员）
assignmen（赋值），532, 600
const（常量），362

initialization（初始化），471, 600
memcpy( ), 560
standard C library function（标准C库函数），326
memory（内存，存储器），133
allocation and efficiency（分配和效率），566
dynamic memory allocation（动态内存分配），223, 548
leak（泄漏），224, 300
finding with overloaded new and delete（用重载的new和delete发现），573
from delete void*（由删除void*），557
management（管理）
example of（...的例子），324
reference counting（引用，计数），326
memory manager overhead（内存管理开销），554
read-only (ROM)（只读存储器），364
simple storage allocation system（简单的存储分配系统），570
memset( ), 269, 326, 356, 560
mentoring（顾问）
and training（训练），71, 73
consulting, mentoring, and design and code walkthroughs from MindView（来自MindView的咨询、指导和代码演练），16
message, sending（发送，消息），25, 239, 636
methodology, analysis and design（方法学，分析与设计），44
Meyers, Scott, 28, 760, 778
MindView
public hands-on training seminars（公共的手把手的课堂培训），16
seminars-on-CD-ROM（CD-ROM上的课堂讨论），16
minimum size of a struct（结构的最小长度），241
mission statement（任务声明），47
mistakes, and design（错误，和设计），279
modulus (%)（模数），156
Moo, Barbara, 778
Mortensen, Owen, 477
multi-way selection（多路选择），124
multiparadigm programming（多范式程序设计程），24
multiple（多重）
dispatching（分派），675
inclusion of header files（头文件的包含），244
inheritance（继承），586, 613, 621, 673, 695
multiple-declaration problem（多次声明问题），244
multiplication (*)（乘法），156
multitasking and volatile（多任务处理和易变的），365
Murray, Rob, 520, 760
mutable（可变的），363
bitwise vs. logical const（按位和按逻辑的常量），362
mutators（修改器），380

N

name（名字）
clashes（冲突），229
collisions, in C（冲突，在C中），68
decoration（修饰），230, 231, 237, 442
no standard for（无标准），312
overloading and（重载与），311
file（文件），749
hiding, during inheritance（隐藏，在继承期间），595
mangling（损坏），230, 231, 237
and overloading（和重载），311
named constant（名字的常量），153
namespace（名字空间），91, 414, 757
aliasing（别名），415
ambiguity（含糊/歧义性），420
continuation（继续），415
header files（头文件），399
injection of friends（友元的插入），417
referring to names in（对名字引用），417
single name space for functions in C（在C中函数的单一名字空间），229
std, 92
unnamed（未命名的），416
using（使用），417
declaration（声明），421
and overloading（和重载），422
directive（指令），418
and header files（和头文件），247
naming the constructor（命名构造函数），285

narrowing conversions（窄变换），170
NDEBUG, 198
needless recompilation（不需要的重编译），276
nested（嵌套的）
  class（类），428
  friend structure（友元结构），266
  iterator class（迭代器类），512, 721
  scopes（作用域），144
  structures（结构），248
new, 164, 223
  and delete for arrays（对于数组的new和delete），563
  array of pointers（指针数组），558
  delete and containers（删除和容器），692
  keyword（关键字），42
  new-expression（new表达式），223, 552, 566
  new-handler（new句柄），565
  operator new（运算符new），552
  constructor, memory exhaustion（构造函数，内存用尽），576
  exhausting storage（用尽存储），565
  placement specifier（定位符），577
  overloading（重载）
  can take multiple arguments（能取多个参数），577
  new and delete, 566
  for a class（对于类的），570
  for arrays（对于数组的），573
  global（全局的），568
newline（新行），94
no linkage（不连接），153, 412
non-local goto（非局部goto），288
not（非）
  bitwise（按位），159
  equivalent !=（等于），158
  logical not !（逻辑非），173
not_eq, != (logical not-equivalent)（不等于(逻辑不等)），173
nuance, and overloading（细微差别，和重载），310
NULL references（NULL引用），451, 479
number, conversion to numbers from char*（数，从char*变换为数字），188

O

object（对象），23, 79
  address of（...的地址），265
  const member functions（常量成员函数），359
  creating a new object from an existing object（从已经存在的对象创建新的对象），462
  creating on the heap（在堆上创建），554
  definition of（...的定义），238
  definition point（定义点），285
  destruction of static（静态的析构），410
  dynamic object creation（动态对象创建, 42, 738
  file（文件），228
  order during linking（连接期间的顺序），88
  five stages of object design（对象设计的五个阶段），54
  function objects（函数对象），515
  global constructor（全局构造函数），410
  guidelines for object development（对象开发的指导方针），56
  interface to（与...的接口），25
  layout, and access control（规划，和访问控制），269
  lifetime of an object（对象的生命期），42, 547
  local static（局部静态），410
  member（成员），30
  module（模块），79
  object-based（基于对象），238
  object-based C++（基于对象的C++），628
  outside（外面），139
  pass by value（按值方式传递，以值传递），462
  passing and returning large objects（传递和返回大对象），457
  scope, going out of（作用域，超出），143
  size（长度），554
  forced to be non-zero（强制为非零），639
  slicing（切片），650, 655
  static（静态）
  class objects inside functions（在函数中的类对象），408, 437
  initialization dependency（初始化依赖性），432
  temporary（临时的），347, 453, 468, 535

unique address, each object（惟一地址，每个对象），241
object-based/singly-rooted hierarchy（基于对象/单根层次），672, 694
object-oriented（面向对象）
analysis and design（设计和分析），44
basic concepts of object-oriented programming (OOP)（面向对象程序设计的基本概念），22
C++, hybrid object-oriented language, and friend（C++，混合的面向对象语言，和友元C++），269
hybrid object-oriented programming language（混合的面向对象程序设计语言），7
obstacles, management（障碍，管理），71
octal（八进制），154
off-by-one error（"偏移1位"错误），301
ofstream（输出文件流），100, 594
as a static object（作为静态对象），411
one-definition rule（一次定义原则），82, 244
ones complement operator（补运算符），159
OOP（面向对象程序设计），271
analysis and design（分析和设计），44
basic characteristics（基本特征），24
basic concepts of object-oriented programming（面向对象程序设计的基本概念），22
Simula programming language（Simula 程序设计语言），25
substitutability（可替换性），24
summarized（总结），239
operator（运算符），156
&, 134
( ), function call（(), 函数调用），514
*, 136, 727, 730
?: ternary if-else（?:三元运算符if-else），164
[ ], 508, 559, 698
++, 493
<< overloading to use with ostream（<<重载用作输出流），554
=, 505
as a private function（作为私有函数），533
automatic creation（自动创建），532
behavior of（...的行为），522
doesn't automatically inherit（不自动继承），600
memberwise assignment（按成员赋值），600
private（私有），709
vs. copy-constructor（对比拷贝构造函数），521
-> smart pointer（灵巧指针），509
->* pointer to member（指向成员），514
>> and iostreams（>>和输入输出流），106
assignment（赋值），505
auto-increment ++（自动增加），106
binary（二进制）
operators（运算符），160
overloaded（重载），487
overloading examples（重载例子），493
bitwise（按位），159
bool behavior with built-in operators（内部运算符布尔行为），131
C & C++, 127
casting（类型转换），166
choosing between member and non-member overloading, guidelines（在成员和非成员重载之间选择,指导方针），520
comma（逗号），165, 508
complicated expressions with operator overloading（带运算符重载的复杂表达式），488
explicit bitwise and logical operators（显式的按位和逻辑运算符），173
fan-out in automatic type conversion,（在自动类型转换中的扇出），540
global（全局）
overloaded（重载），487
scope resolution ::（作用域解析::），253
increment ++ and decrement --（增加++和减少--），506
logical（逻辑的），158, 505
member function（成员选择），237
member vs. non-member（成员和非成员），518
new, 552
exhausting storage（耗尽内存），565
new-expression（new-表达式），552
placement specifier（替换说明符），577
no exponentiation（非指数），517
no user-defined（非用户定义），517

ones-complement（补运算），159
operators you can't overload（不能重载的运算符），517
overloading（重载），91, 450, 485, 732
arguments and return values（参数和返回值），505
check for self-assignment（自赋值检查），505
inheritance（继承），612
member function（成员函数），487
operators that can be overloaded（可以重载的运算符），488
reflexivity（反身性），536
return type（返回类型），488
virtual functions（虚函数），675
[ ], 519
pitfalls（缺陷），166
postfix increment & decrement（后缀加一和减一），493
precedence（优先级），127
prefix increment & decrement（前缀加一和减一），493
preprocessor strinsize operator #（预处理器字符串长度运算符#），196
relational（关系），158
scope resolution ::（作用域解析::），232, 253, 429
and namespaces（和名字空间），417
for calling base-class functions（调用基类函数），588
shift（移），601
sizeof, 172
type conversion overloading（类型转换重载），535
unary（一元运算符），159, 163
overloaded（重载），487
overloading examples（重载例子），489
unusual overloaded（与众不同的重载），508
optimization（优化）
inlines（内联），379
return value optimization（返回值优化），507
optimizer（优化器）
peephole（窥孔），79
global（全局），79
or
| bitwise（按位或），159
|| logical（逻辑或），158, 166, 173
order（顺序）
access specifiers（访问说明符），173
constructor and destructor calls（构造函数和析构函数调用），263
constructor calls（构造函数调用），592
organization, code（组织,代码），248
header files（头文件），244
or_eq, |= (bitwise or-assignment)（或相等，|=(按位或-赋值)），173
ostream（输出流），327
overloading operator <<（重载运算符<<），520, 554
output, standard（输出,标准），90
outside object（对象外），139
overhead（开销）
assembly-language code generated by a virtual function（由虚函数生成的汇编语言代码），642
function call（函数调用），372, 377
memory manager（内存管理），554
size overhead of virtual functions（虚函数的大小开销），637
overloading（重载），95
<< and >> for iostreams（输入输出流的<< 和 >>），518
assignment（赋值），521
choosing between members and non-members, guidelines（在成员和非成员之间选择,指导方针），520
constructor（构造函数），319
default arguments, difference with overloading（默认参数,和重载的不同），324
fan-out in automatic type conversion（自动类型转换的扇出），540
function（函数），310
function call operator( )（函数调用操作符()），514
global operators vs. member operators（全局操作符和成员操作符），536
namespaces, using declaration（名字空间, using 声明），421
new & delete, 566
new and delete

array（数组），573
class（类），570
global（全局），568
on return values（返回值），312
operator（运算符），91
[], 519
++, 493
<< to use with ostream（<<用作输出流），554
-> smart pointer operator（灵巧指针运算符），509
->* pointer-to-member（指向成员），514
inheritance（继承），612
operators that can be overloaded（可以被重载的运算符），488
operators that can't be overloaded, 不能重载的运算符），517
overloading reflexivity（重载反身性），536
type conversion（类型转换），535
virtual functions（虚函数），675
operator（操作符，运算符），450
overriding, difference（重载,区别），658
pitfalls in automatic type conversion（自动类型转换的缺陷），539
overriding（重写），632
and overloading（和重载），658
during inheritance（在继承期间），595
function（函数），35
overview, chapters（总览,章节），7
ownership（所有权），599, 709, 713, 730
and containers（容器），299, 555, 671, 705

P

pair programming（结对编程），63
paralysis, analysis（瘫痪，分析），45
parsing（语法分析），79
parse tree（语法分析树），79
pass-by-reference（按引用方式传递，传引用），140
pass-by-value（按值方式传递，传值），137, 462
and arrays（和数组），186
passing（传递）
and returning（和返回）
addresses（地址），344
addresses, with const（地址，用常数），349
by value, C（按值方式, C），455
large objects（大对象），457
by value（按值方式），344, 450, 657
temporaries（临时），351
patterns, design（模式，设计），59, 70
iterator（迭代器），719
performance issues（性能方面），72
Perl, 89
pitfall（缺陷）
automatic type conversion（自动类型转换），539
C, 227
operators（运算符），166
preprocessor（预处理器），327
placeholder arguments（占位符参数），323
placement, operator new placement specifier（定位，operator new 定位说明符），577
planning, software development（计划,软件开发），47
Plauger, P.J., 780
Plum, Tom, 394, 751, 760
point, sequence（点,序列），286, 293
pointer（指针），136, 153, 164, 276, 450
argument passing, vs. references（参数传递和引用），341
arithmetic（算术），190
array（数组），184
making a pointer look like an array（使指针看起来像数组），564
of pointers（...的指针），187
assignments, const and non-const（赋值,常量和非常量），343
classes containing, and overloading operator=（类包含，重载运算符=），524
const（常量），171, 340
formatting definitions（格式化定义），342
introduction（简介），133
member, pointer to（成员,指向），473
function（函数），475
overloading（重载），514

pointer & reference upcasting（指针和引用向上类型转换），622
pointer to function（函数指针）
array of（...的数组），201
defining（定义），198
using（使用），200
reference to pointer（指针引用），454
reference, difference（引用,区别），140
smart pointer（灵巧指针），730
square brackets（方括号），185
stack（栈），294
struct, member selection with ->（结构，用->的成员选择符），178
upcasting（向上类型转换），631
void, 450, 455, 559, 562
void*, 142
vs. reference when modifying outside objects（对比在对象外修改时引用），472
polymorphism（多态），37, 597, 627, 681, 713, 741
containers（容器），738
polymorphic function call（多态函数调用），637
vs. downcasting（和向下类型转换），678
post-decrement --（后减一），128
post-increment ++（后加一），128
postconditions（后置条件），758
postfix operator increment & decrement（后缀运算符加一和减一），493
pre-decrement --（前减一），128
pre-increment ++（前加一），128
precedence, operator（优先，运算符），127
preconditions（前置条件），758
prefix operator increment & decrement,（前缀运算符加一和减一），493
preprocessor（预处理器），79, 85, 153
#define, #ifdef and #endif, 245
and scoping（和作用域），376
debugging flags（调试标志），194
macro（宏），158, 192, 372
unsafe（不安全），399
pitfall（缺陷），372
problems（问题），372
string concatenation（字符串拼接），395
stringsizing（字符串长度），395
token pasting（标志粘贴），395
value substitution（值代替），334
prerequisites, for this book（预备知识,本书），22
preventing automatic type conversion with the keyword explicit（用关键字显式阻止自动类型转换），534
printf( ), 569
private（私有），29, 262, 270, 277, 380, 610
copy-constructor（拷贝构造函数），471
private inheritance（私有继承），609
problem space（问题空间），23
process（过程），365
production, and book design（生产，和书设计），18
program（程序）
maintenance（维护），58
structure when writing code（写代码时的结构），93
programmer, client（程序员，客户），28, 260
programming（程序设计）：
basic concepts of object-oriented programming (OOP)（面向对象程序设计的基本概念），22
eXtreme Programming (XP)（极限编程），61, 615, 779
in the large（大型），68
incremental process（渐增过程），614
multiparadigm（多范式），24
pair（结对），63
programs, calling other（程序，调用其它），98
project building tools（项目创建工具），203
promotion（提升），228
automatic type conversion（自动类型转换），533
protected（保护的），29, 263, 270, 610
inheritance（继承），611
prototyping（原型）
function（函数），113
rapid（快速的），59
pseudoconstructor, for built-in types（伪构造函数，内部类型），381, 562, 589
public（公共的），29, 261
inheritance（继承），587
seminars（课堂讨论），5

pure（纯）
abstract base classes and pure virtual functions（抽象基类和纯虚函数），646
C++, hybrid object-oriented language, and friend,（C++, 混合的面向对象语言, 和友元），269
substitution（替代），35
virtual destructor（虚析构函数），668
virtual function definitions（虚函数定义），651
push-down stack（下推栈），104
push_back( ), for vector（push_back( ), 向量），275
putc( ), 376
puts( ), 569
Python, 54, 74, 77, 78, 89, 645, 702

Q

qualifier, c-v（限定词, c-v），366

R

ranges, used by containers and iterators in the Standard C++ Library（范围, 由标准C++库的容器和迭代器, 使用的），728
rapid prototyping（快速原型法），59
re-declaration of classes, preventing（类的再声明, 防止），244
re-entrant（重入），458
read-only memory (ROM)（只读内存），364
reading（读）
file, 100
put by words（输入单词），106
realloc( ), 223, 550, 554
recompiling C programs in C++（用C++再编译C程序），236
recursion（递归），126, 459
and inline functions（内联函数），392
redefining during inheritance（在继承中重定义），595
reducing recompilation（减少再次编译），276
refactoring（精化, 重构），58
reference（引用），153, 450, 451
C++, 140
const（常量），345, 453
and operator overloading（和运算符重载），505
for argument passing（参数传递），351
efficiency（效率），455
external, during linking（外部, 在连接时），228
free-standing（单独使用），451
function（函数），452
NULL, 451, 479
passing const（传常量, 按常量传递），473
pointer & reference upcasting（指针和引用向上类型转换），622
pointer, reference to a pointer（指针, 指针的引用），454
reference counting（引用计数），526, 714
rules（规则），451
upcasting（向上类型转换），630
void reference (illegal)（空引用(不合法)），143
vs. pointer when modifying outside objects（对比对象外修改时的指针），472
reflexivity, in operator overloading（反身性, 运算符重载），536
register（寄存器），414
variables（变量），149
reinterpret_cast（重解释转换），171
relational operators（关系运算符），158
reporting errors in book（报告书中的错误），16
request, in OOP（请求, OOP中的），25
require( ), 698, 711, 757
require.h, 237, 252, 756, 757
function definitions（函数定义），396
requireArgs( ), from require.h,（requireArgs( ), 来自require.h），252
requirements analysis（需求分析），48
resolution, scope（解析, 作用域）
global（全局），253
nested structures（嵌套结构），278
operator ::（运算符::），232
resolving references（可分辨引用），80
return（返回）
by value（通过值, 按照值的方式），450

by value as const, and operator overloading（作为常量传值，和运算符重载），507
const value（常量值），345
constructor return value（构造函数返回值），287
efficiency when creating and returning objects（创建和返回对象时的效率），507
function return values, references（函数返回值，引用），451
keyword（关键字），115
operator（运算符）
overloaded return type（重载返回类型），488
overloading arguments and return values（重载参数和返回值），505
overloading on return values（重载返回值），312
passing and returning by value, C（通过值传递和返回，C），455
passing and returning large objects（传递和返回大对象），457
references to local objects（局部对象引用），452
type（类型），597
value（值），81
from a function（从函数），115
optimization（优化），507
semantics（语义），350
void（空），115
RETURN, assembly-language（RETURN，汇编语言），458
reusability（可重用性），29
reuse（重用），55
code reuse（代码重用），583
existing class libraries（现有的类库），70
source code reuse with templates（使用模板进行源代码重用），696
templates（模板），689
right-shift operator (>>)（右移运算符(>>)），160
ROM, read-only memory, ROMability（ROM，只读存储），364
rotate（旋转），162
bit manipulation（位操作），162
RTTI, run-time type identification（RTTI，运行时类型识别），655, 680
rule, makefile（规则，makefile），204
Rumbaugh, James, 779
run-time（运时时），
access control（访问控制），275
binding（捆绑），631
debugging flags（调试标志），195
type identification (RTTI)（类型识别(RTTI)），655, 680
rvalue（右值），156, 698

S

safe union（安全联合），319
Saks, Dan, 66, 394, 751, 760
scenario（情节），49
scheduling（计划进度），51
Schwarz, Jerry, 434
scope（作用域），143, 288, 339, 554
consts（常量），337
file（文件），339, 412
going out of（在...之外），143
hide variables from the enclosing scope（在封闭作用域隐藏变量），292
preprocessor（预处理器），376
resolution（分解）
global（全局），253
nested structures（嵌套结构），278
operator ::（运算符::），232, 429
and namespaces（名字空间），417
for calling base-class functions（调用基类函数），588
scoped variable（限定作用域的变量），42
static member initialization（静态成员初始化），425
storage allocation,（存储分配），549
use case（用例），57
second edition, what's new（第2版，新增内容），2
security（安全），276
selection（选择），
member function（成员函数），234
multi-way（多路），124
self-assignment, checking for in operator overloading（自赋值，检查运算符重载），505, 523

semantics, return value（语义，返回值），350
seminars（课堂讨论）
on CD-ROM, from MindView（来自MindView公司，在CDROM上），16
public（公共的），5
training seminars from MindView（来自MindView 公司的课堂培训），16
sending a message（送消息），25, 239, 636
sentinel, end（哨兵，终止），728, 736
separate compilation（分别编译），78, 80
and make（和make），702
separation of interface and implementation（接口和实现的隔离），29, 261, 271
sequence point（序列点），286, 293
set
<set> standard header file（<set>标准头文件），711
and get functions（和get函数），381
container class from the Standard C++ Library（来自标准C++库的容器类），711
setf( ), iostreams（setf( )，输入输出流），466
setjmp( ), 288
SGI (Silicon Graphics) STL project（SGI, STL项目），103
shape（形状）
example（例子），32
hierarchy（层次），682
shift operators（shift操作符），160
short（短），132
side effect（副作用），156, 164
signature（签名/特征），597
signed（带符号的），132
char, 132
Silicon Graphics (SGI) STL project（SGI, STL项目），103
Simula programming language（Simula编程语言），25, 271
single-precision floating point（单精度浮点），130
singly-rooted/object-based hierarchy（单根/基于对象的继承），672, 694
size（尺寸/规模）
built-in types（内部类型），129
object（对象），554
forced to be nonzero（强制非零），639
size_t, 568
storage（存储），220
struct（结构），240
word, 133
sizeof, 132, 172, 302, 587
char, 173
struct（结构），240
slicing（切片）
object slicing（对象切片），655
Smalltalk, 24, 80, 645, 694, 702
smart pointer operator ->（灵巧指针运算符->），509, 730
software（软件）
crisis（危机），8
development methodology（开发方法学），45
solution space（解空间），23
solutions, exercise（答案，练习），12
source code availability（提供源代码），12
source-level debugger（源代码级调试），78
space（空间）
problem（问题），23
solution（解），23
specification（说明，规范说明），
incomplete type（不完全类型），265, 277
system specification（系统说明），48
specifier（说明符，指定符）
access specifiers（访问说明符），29, 261
no required order in a class（在类中不要求顺序），263
to modify basic built-in types（修改基本内部类型），132
specifying storage allocation（指定存储分配），147
sstream standard header file（sstream标准头文件），520
stack（栈），41, 248, 294, 549
function-call stack frame（函数调用栈框架），458
pointer（指针），406
push-down（下推），275
storage allocation（存储分配），549
variable on the stack（栈中变量），225

Stack example class（栈例子类），248, 274, 298, 388, 597, 672, 690, 705, 728
Standard C++ Library（标准C++库），
algorithms（算法），742
insert( ), 104
push_front( ), 104
ranges, used by containers and iterators（范围，用于容器和迭代器），728
standard for each class header file（每个类头文件的标准），246
standard input（标准输入），97
standard library（标准库），89
standard library header file（标准库头文件），
cassert, 197
cstdlib, 188
cstring, 269
set, 711
sstream, 520
typeinfo, 680
standard output（标准输出），90
Standard Template Library (STL)（标准模板库），103
standards, C++ Committee（标准，C++委员会），14
startup costs（启动开销），71
startup module（启动模块），89
Stash example class（Stash例子类），219, 230, 274, 294, 314, 322, 385, 558, 707
statement（语句），
continuation over several lines（在一些行继续），97
mission（任务），47
static（静态），149, 406, 711
array（数组），692
initialization（初始化），425
class objects inside functions（函数中的类对象），408
confusion when using（当使用using指令时的混乱），412
const（常量），356
data（数据），
area（区域），406
members inside a class（类中成员），423, 430
defining storage for（定义存储），424
destruction of objects（对象的销毁），410
file（文件），414
initialization dependency（初始化依赖），432
initialization to zero（初始化为零），433
initializer for a variable of a built-in type（内部类型的变量初始化），408
local object（局部对象），410
member functions（成员函数），366, 429, 465
inheritance and（继承），604
objects inside functions（函数内对象），437
storage（存储），41, 406
area,（区域），549
type checking（类型检查），80
variables in functions as return values（作为返回值的函数变量），350
variables inside functions（函数内变量），406
static_cast（静态类型转换），169, 679
downcast（向下类型转化），681
std namespace（std 名字空间），92
step, in for loop（for循环中的步骤），121
STL
Silicon Graphics (SGI) STL project（SGI,STL项目），103
Standard Template Library（标准模板库），103
storage（存储），
allocation（分配），292
const and extern（常量和外部的），336
auto storage class specifier（自动存储类型指定符），414
const, in C vs. C++（常量，在C与C++中），339
defining storage for static data members（定义静态数据成员的存储），424
extern storage class specifier（外部存储类型指定符），412
register storage class specifier（寄存器存储类型指定符），414
running out of（运行超出），565
simple allocation system（简单分配系统），570
sizes（大小），220
static（静态），41, 406
area（区域），549
storage class specifier（存储类型指定符），412

storage class（存储类），412
storing type information（存储类型信息），637
Straker, David, 755
string（字符串），94, 227
class, Standard C++（类，标准C++），99
concatenation（拼接），96
copying a file into（把文件拷入...），102
getline( ),562
preprocessor # to turn a variable name into a string（预处理器#把变量名转换成字符串），196
preprocessor string concatenation（预处理器字符串拼接），395
stringizing, preprocessor（字符串长度，预处理器），395
macros（宏），192
operator #（运算符#），196
stringstream（字符串流），520
strong typing, C++ is a more strongly typed language（强类型，C++是一种类型定义比较严格的语言），450
Stroustrup, Bjarne, 4, 433, 696, 748, 776, 779
struct（结构），175, 219, 238, 269
aggregate initialization（集合初始化），302
array of（...的数组），183
hiding function names inside（内部隐藏函数名），230
minimum size（最小尺寸），241
pointer selection of member with ->（指针成员用->选择），178
size of（...的大小），240
structure（结构）
aggregate initialization and structures（集合初始化和结构），302
declaration（声明），245, 265
definition in a header file（在头文件中定义），234
friend（友元），264
nested（嵌套），248
redeclaring（重声明），245
subobject（子对象），585, 587, 588, 604
substitutability, in OOP（可替代性在OOP中），24
substitution（替代）
principle（原则），35
value（值），334
subtraction (-)（减），156
subtyping（子类型），606
suffix rules, makefile（后缀规则, makefile），205
SUFFIXES, makefile, 206
sugar, syntactic（糖，语法），485
switch, 123, 293
defining variables inside the selector statement（在选择器语句中定义变量），145
syntax（语法）
function declaration syntax（函数声明语法），82
operator overloading（运算符重载），487
sugar, with operator overloading（糖，运算符重载），485
variable declaration syntax（变量声明语法），83
synthesized（综合）
default constructor, behavior of（默认构造函数，...的行为），305
member functions that are automatically created by the compiler（由编译器自动创建的成员函数），600, 619
system specification（系统说明），48
system(), 98

T

tab, 95
table-driven code（表驱动的代码），201
tag name（标签名），220
tag, comment for linking（标记，用于连接的注释），148
template（模板），689, 696
argument list（参数表），700
basic usage（基本应用），104
class（类），742
constants and default values in templates（模板中的常数和默认值），703
container class templates and virtual functions（容器类模板和虚函数），743
function（函数），742
generated classes（生成类），699
header file（头文件），700, 707
implies an interface（隐含接口），701

inline（内联），707
instantiation（实例），699
multiple definitions（多次定义），700
non-inline template member function definitions（非内联模板成员函数定义），699
preprocessor macros for parameterized types, instead of templates（参数化类型的预处理器宏，不用模板），696
Standard Template Library (STL)（标准模板库），103
Stash and Stack examples as templates（Stash 和Stack例子模板），705
weak typing（弱类型），701
temporary object（临时对象），347, 368, 535
bugs, 348
function references（函数引用），453
passing a temporary object to a function（向函数传递临时对象），351
return value（返回值），508
ternary operator（三元运算符），164
testing（测试）
automated（自动），62
eXtreme Programming (XP)（极限编程），61
Thinking in C++ Volume 2, what's in it and how to get it, 3
Thinking in C: Foundations for Java and C++ CD ROM, 2, 112, 776
this, 286, 363, 380, 429, 468, 552, 642
address of current object（当前对象的地址），234
throw（抛出），572
time, Standard C library（时间，标准C库），384
time_t, 384
token pasting, preprocessor（标志粘贴，预处理），395
toupper( ), unexpected results（toupper( )，非预期结果），376
trailing arguments only can be defaults（限制参数只能为默认值），322
training（培训），69
and mentoring（指导），71, 73
seminars from MindView（MindView公司的课堂讨论），16
translation unit（翻译单元/翻译单位），228, 432
true（真），158, 163, 166, 246
and false, in conditionals（false,在条件中），17
bool, true and false, 131
try block（try块），572
type（类型）
abstract data type（抽象数据类型），239
automatic type conversion（自动类型转换），533
preventing with the keyword explicit（关键字显式防止），534
with operator overloading（运算符重载），535
base（基本），32
basic built-in（基本内部），129
cast（类型转换），135
checking（检查），80, 83, 153, 167
stricter in C++（C++中更严格），227
conversion（转换），228
implicit（隐式），154
creation, composite（创建,组合），174
data type equivalence to class（数据类型等同于类），26
derived（派生），32
function type（函数类型），390
improved type checking（改进的类型检查），236
incomplete type specification（不完全的类型说明），265, 277
inheritance, is-a（继承, is-a），615
initialization of built-in types with 'constructors'（用“构造函数”初始化内部类型），354
run-time type identification (RTTI)（运行时类型识别），655, 680
storing type information（强类型信息），637
type checking（类型检查）
for enumerations（对枚举型），180
for unions（对联合），181
type-safe linkage（类型安全连接），313
user-defined（用户定义），76, 239
weak typing（弱类型），38, 702
C++ via templates（通过模板C++），701
typedef, 174, 177, 220, 231, 414
typefaces, book（typefaces, 书），18
typeid, 680

typeinfo standard header file（typeinfo标准头文件），680
type-safe downcast（类型安全向下转换），678

U

UML（统一建模语言），54
  indicating composition（表示组合），30
  Unified Modeling Language（统一建模语言，27，779
unary（一元）
  examples of all overloaded unary operators（所有重载的一元运算符例子），489
  minus -（减），163
  operators（运算符），159，163
  overloaded（重载），487
  plus +（加），163
underscore，leading，on identifiers (reserved)（下划线,前导，标志符(保留)），381
Unified Modeling Language (UML)（统一建模语言），27，779
union（联合）
  additional type checking（附加类型检查），181
  anonymous（匿名），320
  file scope（文件作用域），321
  difference between a union and a class（类和联合之间的区别），391
  member functions and access control（成员函数和访问控制），318
  safe（安全），319
  saving memory with（用...节省内存），181
unit，translation（单元，翻译），228
unnamed（未命名的）：
  arguments（参数），114
  namespace（名字空间），416
unresolved references，during linking（未解决的引用，在连接期间），88
unsigned（无符号），132
untagged enum（无标志枚举），320，358
unusual operator overloading（不常见的运算符重载），508
upcasting（向上类型转换），40，615，629，636，678，738
  by value（通过值），644
  copy-constructor（拷贝构造函数），617
  explicit cast for upcasting（向上转换的显式转换），681
  pointer（指针），631
  and reference upcasting（和引用向上类型转换），622
  reference（引用），630
  type information，lost（类型信息，丢失），622
use case（用例），49
  iteration（迭代），57
  scope（范围/作用域），57
user interface（用户界面），51
user-defined data type（用户定义的数据类型），76，129，239
using keyword，for namespaces（using关键字，名字空间），92，417
  declaration（声明），421，757
  directive（指令），92，418，757
  header files（头文件），247
  namespace std（名字空间std），247

V

value（值）
  constant（常数），154
  minimum and maximum for built-in types（内部类型的最小值和最大值），129
  pass-by-value（传值），137
  preprocessor value substitution（预处理器值替代），334
  return（返回），81
  returning by value（以值返回），352
varargs（变量参数），243
  variable argument list（变量参数表），243
variable（变量）
  argument list（参数表），243
  varargs（变量参数），243
  automatic（自动），42，149，153
  declaration syntax（声明语法），83
  defining（定义），145
  file scope（文件作用域），150
  global（全局），147

going out of scope（作用域外），143
hide from the enclosing scope（封闭作用域隐藏），292
initializer for a static variable of a built-in type（内部类型的静态变量的初始化），408
lifetime, in for loops（生命周期，在for循环中），292
local（局部），138, 149
point of definition（指针定义），289
register（寄存器），149
scoped（限定作用域的），42
stack（栈），225
turning name into a string（将名字转换成字符串），196
vector（矢量），740
assignment（赋值），108
of change（改变），59
push_back( ), 104
Standard C++ Library（标准C++库），102
virtual destructor（虚析构函数），665, 707, 736, 740
pure virtual destructor（纯虚析构函数），668
virtual function（虚函数），595, 627, 629, 646, 741
adding new virtual functions in the derived class（在派生类中增加新的虚函数），652
and dynamic_cast（动态类型转换），679
assembly-language code generated by a virtual function（由虚函数生成的汇编语言代码），642
constructors, behavior of virtual functions inside（构造函数，内部虚函数行为），662, 664
destructors, behavior of virtual functions inside（析构函数，内部虚函数行为），670
efficiency（效率），645
late binding（晚捆绑），637
operator overloading and virtual functions（运算符重载和虚函数），675
overriding（重写），632
picturing virtual functions（绘制虚函数），639
pure virtual function（纯虚函数）
and abstract base classes（和抽象基类），646
definitions（定义），651
size overhead of virtual functions（虚函数的大小开销），637
virtual keyword（virtual关键字），391, 632
in base-class declarations（在基类声明中），632
in derived-class declarations（在派生类声明中），632
virtual memory（虚内存），552
visibility（可见性），406
void
argument list（参数表），114
casting void pointers（空指针类型转换），235
keyword（关键字），114
pointer（指针），220, 450, 555, 559, 562
reference (illegal)（引用(不合法)），143
void*, 142, 170, 220
bugs（错误），235
containers and ownership（容器和所有权），671
delete, a bug（删除，一个错误），555
volatile（易变的），155, 365
casting with const_cast（用const_cast类型转换），170
Volume 2, Thinking in C++, 3
vpointer, abbreviated as VPTR（vpointer，缩写为VPTR），637
VPTR, 637, 640, 642, 662, 665
installation by the constructor（由构造函数安装），643
VTABLE, 637, 640, 642, 648, 653, 662, 665
inheritance and the VTABLE（继承和VTABLE），652

W

Waldrop, M. Mitchell, 781
weak（弱）
typing（类型定义），702
in C++ via templates（在C++中通过模板），701
weakly typed language（弱类型定义语言），38
while loop（while循环），101, 119
defining variables inside the control expression（在控制表达式中定义变量），145
width( ), iostreams（width( )，输入输出流），466
wild-card（不定，不定型），46
Will-Harris, Daniel, 17, 18
word size（字长），133
writing files（写文件），100

X

xor ^ bitwise exclusive-or（按位异或xor ^），159, 173
xor_eq, ^= bitwise exclusive-or-assignment（xor_eq, 按位异或赋值^=），173
XP, eXtreme Programming（极限编程），61

Z

zero indexing（零索引），183