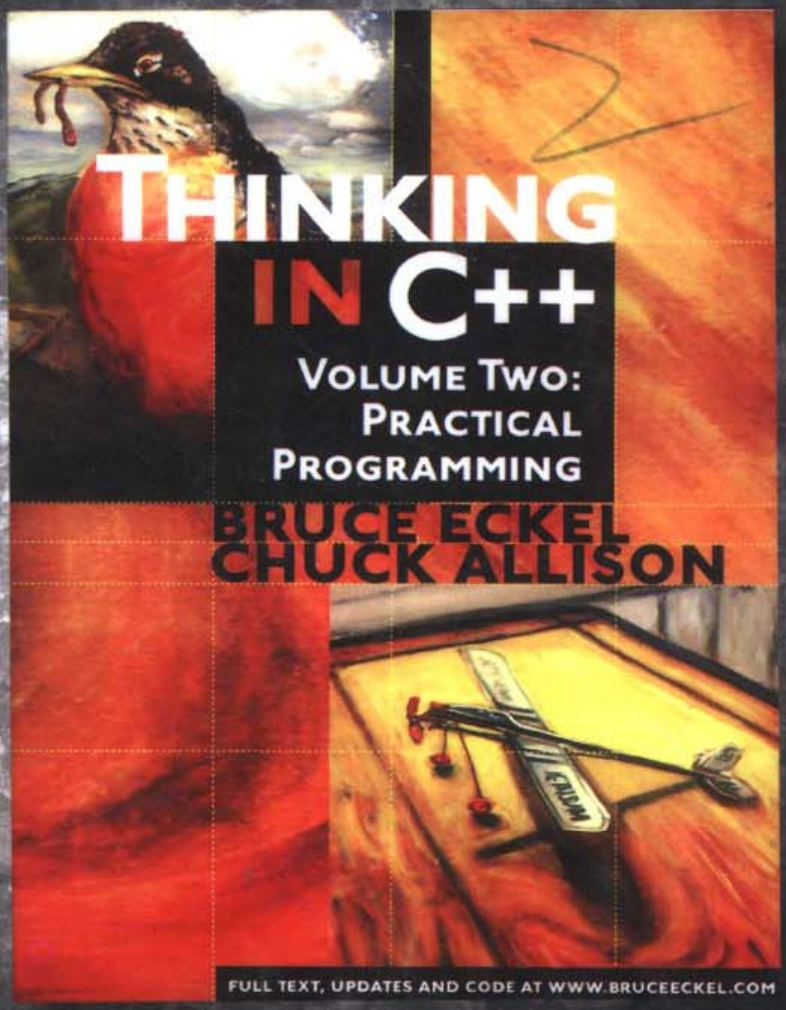
PEARSON Prentice Hall

计 算 机 科 学 丛 书

C++编程思想

第2卷 实用编程技术

(美) Bruce Eckel Chuck Allison 著 刁成嘉 等译



Thinking in C++

Volume Two: Practical Programming

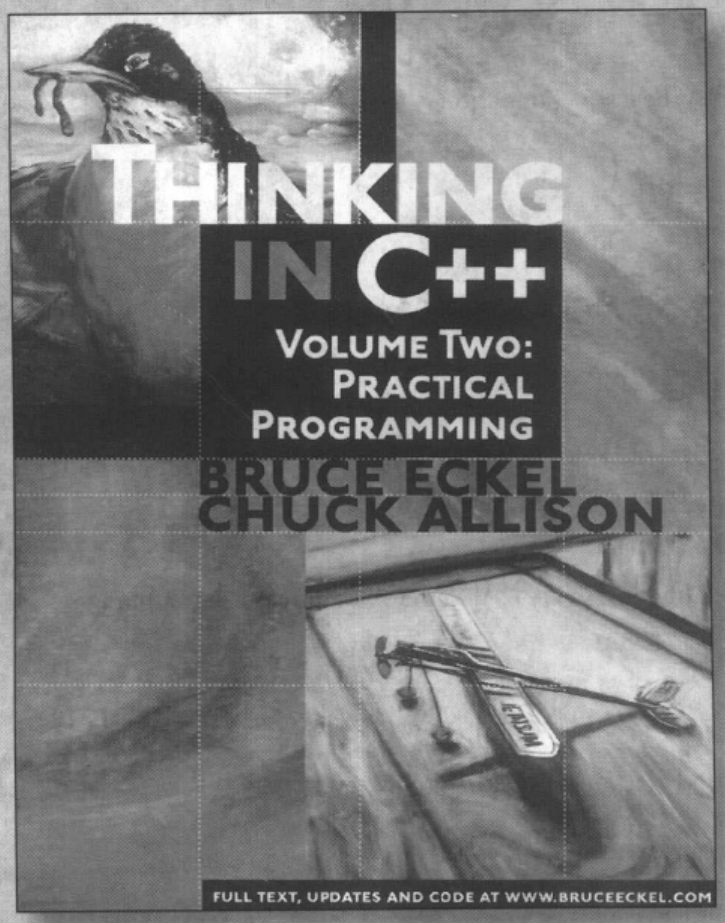
机械工业出版社
China Machine Press

计 算 机 科 学 丛 书

C++编程思想

第2卷 实用编程技术

(美) Bruce Eckel Chuck Allison 著 刁成嘉 等译



Thinking in C++

Volume Two: Practical Programming

本书介绍C++实用的编程技术和最佳的实践方法，深入探究了异常处理方法和异常安全设计；介绍C++的字符串、输入输出流、STL算法、容器和模板的现代用法，包括模板元编程；解释多重继承问题的难点，展示RTTI的实际使用，描述了典型的设计模式及其实现，特别介绍被认为是标准C++下一版特征之一的多线程处理编程技术，并提供了最新的研究成果。本书适合作为高等院校计算机及相关专业的本科生、研究生的教材，也可供从事软件开发的研究人员和科技工作者参考。

Simplified Chinese edition copyright © 2005 by Pearson Education Asia Limited and China Machine Press.

Original English language title: Thinking in C++, Volume 2: Practical Programming , (ISBN: 0-13-035313-2) by Bruce Eckel and Chuck Allison, Copyright © 2004.

All rights reserved.

Published by arrangement with the original publisher, Pearson Education, Inc., publishing as Prentice Hall, Inc.

本书封面贴有Pearson Education（培生教育出版集团）激光防伪标签，无标签者不得销售。

版权所有，侵权必究。

本书法律顾问　北京市展达律师事务所

本书版权登记号：图字：01-2004-0557

图书在版编目（CIP）数据

C++编程思想，第2卷：实用编程技术 /（美）埃克尔（Eckel, B.）等著；刁成嘉等译．－北京：机械工业出版社，2006.1

（计算机科学丛书）

书名原文：Thinking in C++, Volume 2: Practical Programming

ISBN 7-111-17115-2

Ⅰ.C…　Ⅱ.①埃…　②刁…　Ⅲ.C语言－程序设计　Ⅳ.TP312

中国版本图书馆CIP数据核字（2005）第092239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：隋　曦

北京京北制版厂印刷·新华书店北京发行所发行

2006年1月第1版第1次印刷

787mm×1092mm 1/16·33.25印张

印数：0 001-6 000册

定价：59.00元

凡购本书，如有倒页、脱页、缺页，由本社发行部调换

本社购书热线：（010）68326294

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年开始，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，有的还专程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：除“计算机科学丛书”之外，对影印版的教材，则单独开辟出“经典原版书库”；同时，引进全美通行的教学辅导书“Schaum's Outlines”系列组成“全美经典学习指导系列”。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。

这三套丛书是响应教育部提出的使用外版教材的号召，为国内高校的计算机及相关专业

的教学度身订造的。其中许多教材均已为M. I. T.，Stanford，U.C. Berkeley，C. M. U. 等世界名牌大学所采用。不仅涵盖了程序设计、数据结构、操作系统、计算机体系结构、数据库、编译原理、软件工程、图形学、通信与网络、离散数学等国内大学计算机专业普遍开设的核心课程，而且各具特色——有的出自语言设计者之手、有的历经三十年而不衰、有的已被全世界的几百所高校采用。在这些圆熟通博的名师大作的指引之下，读者必将在计算机科学的宫殿中由登堂而入室。

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：hzjsj@hzbook.com
联系电话：（010）68995264
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			

“恭喜两位完成了这部经典之作！这部精品既妙趣横生，又不乏深度……所用专业知识的精确和语言应用的缜密真是让我大为震撼……我相信你们已经达到了大师级水平，简直太出色了！”

——《C/C++ Users Journal》杂志专栏主编 Bjorn Karlsson

“此书是一项巨大的成就，你的书架上早就该有这本书了。”

——《Doctor Dobbs Journal》杂志特约编辑 Al Stevens

“Eckel的作品是惟一一本如此清晰地阐述如何重新思考以面向对象方法构造程序的书籍。这本书也是一本讲授C++来龙去脉的优秀指南。”

——《Unix Review》杂志的编辑 Andrew Binstock

“Bruce 在C++方面的洞察力一次次令我惊叹，而这本《C++编程思想》则是他思想的精萃。如果你想获得C++中难题的清晰解答，就请购买这部杰作吧。”

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

“《C++编程思想》不仅系统而详细地探讨了何时和如何使用内联、引用、运算符重载、继承和动态对象等方面的重要问题，而且还讨论了一些深入的技术，如怎样正确使用模板、异常及多重继承等。Eckel本人的面向对象和程序设计的思想也完全融入这部著作中。《C++编程思想》是每个C++开发人员案头必备之书，即每一位用C++开发重要软件的开发人员必须拥有的一本书。”

——《PC Magazine》杂志特约编辑 Richard Hale Shaw

谨以此书

献给那些一直为C++的发展而不知疲倦地工作的人们。

译 者 序

C++语言是一种使用广泛的程序设计语言，掌握了C++基础知识和基本编程技巧的人们，如果还想对C++有深入的了解，并且掌握更高级的C++编程技术的话，我们愿意向广大读者推荐《C++编程思想　第2卷：实用编程技术》的中译本。作者Bruce Eckel是C++标准委员会拥有表决权的成员之一，本书第1版荣获《软件开发》杂志评选的1996年度图书震撼大奖（Jolt Award），成为该年度最佳图书，在美国非常畅销。本书内容十分丰富，结构设计循序渐进，案例翔实而深入浅出，有一定的深度和广度。

二位作者致力于计算机教学数十年，经验十分丰富。在本书的讲授方法、例子和每章后面的练习的选用上都别具特色。通过一些非常简单的例子和简练的叙述，准确地阐明了C++编程实践中最困难的一些问题和概念，给人以拨云见日、耳目一新的感觉。读者在学习那些原本难于理解的内容时，常常会有豁然开朗的奇特效果，从而在不知不觉中接受并掌握了实用的编程技术。

本书介绍了实用的编程技术和最佳的实践方法，解决了C++开发中最困难的课题。内容上分为3部分：第二部分深入探究异常处理方法，清晰解释了异常安全设计；第三部分研究了C++的字符串、输入输出流、STL算法和容器；详细阐述了模板的现代用法，包括模板元编程；第三部分解释多重继承的难点，展示RTTI的实际使用，描述典型的设计模式及其实现，介绍被认为是标准C++下一版特征之一的多线程处理编程技术，并提供了最新的研究成果。书中所举的程序例子都经过多个软件平台和编译器的测试，稳定可靠。本书不仅适合C++的初学者，对有经验的C++程序员来说，每次阅读也总能有新的体会，这正是本书的魅力所在。也正因为如此，本书不仅适合作为高等院校计算机、信息技术及相关专业本科生、研究生的教材，也可供广大从事软件开发的研究人员和科技工作者参考。

作为译者，我早已耳闻《C++编程思想》是一本别具特色的畅销书，并拜读了本书第1卷的中译本，其内容、讲授方法和特色让我受益匪浅。受机械工业出版社华章公司的委托，我有幸承担《C++编程思想》第2卷的翻译工作。翻译这样的成功之作，既是机遇，又是挑战。在翻译的过程中惟恐因水平有限而不能将原著中精彩内容如实转达，所以在翻译本书的过程中力求忠于原著，对书中出现的大量专业术语力求遵循标准译法，并在有可能引起歧义的地方注上英文原文。

本书在翻译过程中受到南开大学信息学院计算机系刘璟教授的关心和支持，特此表示感谢。邢恩军、刘胜斐、罗仕波、郑莹莹、肖鹏、程玉鹏、黄硕、金士英、杨鹏飞、赵建树、田新、漆芳敏、费志泉、邰业军、申芳、杨志真、刁奕、高建国、旷昊、蓝炳伟、王叙、李平参加了本书部分章节的初译。由于水平有限，翻译不妥或错误之处在所难免，敬请广大读者批评指正。

刁成嘉

2005年9月

于南开园

前　言

1

通过对本套教材第1卷的学习，读者已经掌握了C与C++的基础知识。这一卷将涉及其更为高级的特性，使读者领悟C++编程的方法与思想，从而编写出健壮的C++程序。

现在假定读者已经熟悉了第1卷的内容。

目标

编写这套教材的目标是:

1. 每节只介绍适当的学习内容，使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前消化每个已学过的概念。

2. 讲授实用编程技巧，以便读者在日常的学习和工作中使用这些技巧。

3. 只把对于理解这门语言比较重要的内容介绍给读者，而不是将我们所知的一切都罗列出来。我们相信，不同信息的重要性是不同的。有些内容对于95%的程序员来说肯定没有必要知道，这些信息只会迷惑人们，加深人们对这门语言复杂性的恐惧。举一个关于C语言的例子，如果记住运算符优先级表（我们从未做到这一点），就能够写出漂亮的代码。但如果对其进行深究，它会让代码的读者或维护者感到迷茫。所以可以摒弃优先级，而在优先级不很清楚的情况下使用括号。同样，C++语言中的某些信息对于写编译程序的人员来说更为重要，而对程序员来说却没那么重要。

4. 尽可能将每一节内容充分集中，使得授课时间及两个练习之间的间隔时间不长。这样不仅能使读者的思维在每次课堂研讨会期间更加活跃与投入，还可使他们有更大的成就感。

5. 尽力不用任何特定厂商的C++版本。我们已在所有能见到的C++实现版本中测试了本教材中的代码（前言中稍后将有介绍），有的实现版本无法工作，那是因为它没有遵循C++标准，我们已经在示例中标注这些事实（读者会在源代码中看到这些标注），以便将其从构建过程中摒弃。 2

6. 教材中代码的自动编译和测试。由于已经发现未经编译和测试的代码很可能有问题，所以在这一卷中，本教材所提供的例子全是测试过的代码。此外，读者可从http://www.MindView.net下载这些代码，它们是直接从本教材的文本中摘录的，这些程序能够用自动生成的测试文件进行编译和运行测试。读者可以通过这种方式知道教材中的代码都是正确的。

各章简介

下面是本教材各章内容的简要介绍。

第一部分　建立稳定的系统

第1章　异常处理。出错处理在程序设计中一直是一个问题。即便你返回了错误信息或设置了一个标志，函数调用者还会对此视而不见。异常处理是C++的主要特征之一，该机制解

决这类问题的方法如下：在致命错误发生时，允许该函数“抛出”一个对象。对应于不同的错误抛掷不同类型的对象，那么该函数的调用者就可以在独立的出错处理子程序中“捕获”这些对象。如果程序中抛出了一个异常，该异常就不能被忽略，这样就可以保证会触发一些事件来响应这一错误。决定采用异常处理机制是影响代码设计向良性方向发展的重要方法。

第2章　防御性编程。许多软件故障都是可以预防的。防御性编程是一种编写代码的方式，采用此种方式能够较早地发现并更正错误，从而避免了这些错误对相关工作区域造成的危害。在开发过程中使用断言（assertion）是一种很重要的方法，该方法能够在程序员编写代码的过程中进行合法性检验，与此同时在代码中留下了一个可执行文档，该文档可用来揭示程序员开始编写代码时的思路。在向用户交出程序前应严格地测试编写的代码。对于成功地进行常规软件开发的人员来说，自动单元测试框架是一个不可缺少的工具。

第二部分　标准C++库

第3章　深入理解字符串。最为常见的编程工作是对文本进行处理。C++字符串类将程序员从内存管理事务中解脱出来，使其有足够的时间和精力增强文本处理能力。此外，为适应国际化应用的需求，C++也支持对宽字符和区域字符的操作。

第4章　输入输出流。输入输出流类是最早的C++库之一，它提供必不可少的输入输出功能。使用输入输出流类就是用I/O库来代替C语言中的**stdio.h**。这种I/O库用起来更容易、更灵活并且更易于扩展——可对其做适当的调整使之能够与新定义的类一起工作。该章告诉读者怎样充分利用现有的输入输出流类库来实现标准I/O、文件I/O以及内存中的格式化操作。

第5章　深入理解模板。现代C++的显著特征是模板的强大功能。模板的作用不仅仅在于生成容器。借助于模板，还可开发出具有健壮性、通用性和高性能的类库。关于模板的内容，需要了解的还有很多，它们构成了C++语言内的一个子语言，使得程序员能在更大程度上控制编译过程。模板的引入对C++程序设计来说是一场革命，可以毫不夸张地说，自从有了模板，C++程序设计焕然一新了。

第6章　通用算法。算法处于计算的核心。C++借助其模板功能提供了一大批功能强大、高效且易用的通用算法。标准算法也可以通过函数对象进行自定义。该章研究了模板库中的所有算法。（第6章和第7章讲的是标准C++模板库，也就是通常所说的标准模板库（Standard Template Library，STL）。）

第7章　通用容器。C++以一种类型安全的方式提供对所有常见数据结构的支持。用户不必为容器中的内容而感到忧虑，其对象的同一性得到了保证。可通过迭代器将容器的遍历与容器自身相分离，这是模板的又一杰作。这种巧妙的安排能够将算法灵活应用于容器，而容器则采用了最简单的设计。

第三部分　专题

第8章　运行时类型识别。当你只用一个对象指针或引用指向基类型时，运行时类型识别（RunTime Type Identification，RTTI）就会找到该对象的确切类型。一般情况下，有时会有意忽略掉一个对象的确切类型，而利用虚函数机制实现对应于那个类型的正确操作。但有时（比如当编写像调试器这样的软件工具时）借助于此信息知道一个对象的确切类型是非常有用的，常常可以非常有效地进行某些特殊操作。这一章解释RTTI的用途及其使用方法。

第9章　多重继承。一个新类可以从多个现存类中继承，这话乍听起来很简单。但是，由

此而产生的二义性和对基类对象的多次复制将很难避免。这些问题可通过建立虚基类来解决，但更大的问题仍然存在：什么时候用多重继承？只有当你需要通过多于一个的公共基类来操作一个对象时，多重继承才是必需的。这一章对多重继承的语法做了解释，也提出了可选方案——特别针对使用模板怎样解决一个典型问题进行了深入讨论。运用多重继承来修复一个“被损坏了的”类接口是关于C++这一特性的经典案例。

第10章　设计模式。自从对象产生以来，在程序设计领域最具革命性的飞跃是设计模式的引进。设计模式是对应于公认的编程问题的经典解决方案，它独立于语言之外，其表述方式的特殊性使它能应用于许多情况之下。因此，像单件（Singleton）、工厂方法（Factory Method）和访问者（Visitor）这样的模式现都已被一般的程序员接受和使用了。这一章介绍如何通过C++来实现和使用一些较为有用的设计模式。

第11章　并发。人们越来越期待有响应功能的用户接口，而这种接口能（看起来像）同时处理多任务。现代操作系统允许进程拥有共享进程地址空间的多线程。多线程程序设计要求编程人员有与众不同的思维方式，然而，在进行多线程程序设计时也会遇到一些困难。这一章通过一个可免费获得的类库（由IBM的 Eric Crahen 提供的ZThread库）介绍怎样使用C++来有效地管理多线程应用。

练习

5

我们发现，在课堂讨论期间使用简单的练习特别有助于学生对相关概念的理解。所以，在每一章后面都附有一定量的练习题。

这些练习题十分简单，可当堂完成；但有一点，需要有老师在场观察证实，以确保所有的学生都掌握了相关内容。有些练习题有一定的挑战性，是为激发优秀学生的学习兴趣准备的。所有练习被设计为可以在短时间内完成，只是用来测试和完善学生所掌握的知识，而不是为了提出挑战（很可能读者自己会找到这些难题——或者更可能的是难题会自己找上门来）。

源代码

本教材的源代码是免费版权软件，通过网站http://www.MindView.net发布。该版权是为了防止在未经许可的情况下在印刷媒体上再度出版这些代码。

在解压缩代码的起始目录中你会发现下列的版权声明：

```
//:! :CopyRight.txt
(c) 1995-2004 MindView, Inc.  All rights reserved.
Source code file from the book
"Thinking in C++, 2nd Edition, Volume 2."

The following permissions are granted respecting the
computer source code, which is contained in this file:

Permission is granted to classroom educators to use this
file as part of instructional materials prepared for
classes personally taught or supervised by the educator who
uses this permission, provided that (a) the book "Thinking
in C++" is cited as the origin on each page or slide that
contains any part of this file, and (b) that you may not
remove the above copyright legend nor this notice. This
permission extends to handouts, slides and other
presentation materials.
```

6

For purposes that do not include the publication or presentation of educational or instructional materials, permission also is granted to computer program designers and programmers, and to their employers and customers, (a) to use and modify this file for the purpose of creating executable computer software, and (b) to distribute resulting computer programs in binary form only, provided that (c) you may not remove the above copyright legend nor this notice from retained source code copies of this file, and (d) each copy distributed in binary form has embedded within it the above copyright notice.

Apart from the permissions granted above, the sole authorized distribution point for additional copies of this file is http://www.MindView.net (and official mirror sites) where it is available, subject to the permissions and restrictions set forth herein.

The following are clarifications of the limited permissions granted above:

1. You may not publish or distribute originals or modified versions of the source code to the software other than in classroom situations described above.

2. You may not use the software file or portions thereof in printed media without the express permission of the copyright owner.

The copyright owner and author or authors make no representation about the suitability of this software for any purpose. It is provided "as is," and all express, implied, and statutory warranties and conditions of any kind including any warranties and conditions of merchantability, satisfactory quality, security, fitness for a particular purpose and non-infringement, are disclaimed. The entire risk as to the quality and performance of the software is with you.
In no event will the authors or the publisher be liable for any lost revenue, savings, or data, or for direct, indirect, special, consequential, incidental, exemplary or punitive damages, however caused and regardless of any related theory of liability, arising out of this license and/or the use of or inability to use this software, even if the vendors and/or 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 have a correction for an error in the software, please submit the correction to www.MindView.net. (Please use the same process for non-code errors found in the book.)

If you have a need for permissions not granted above, please inquire of MindView, Inc., at www.MindView.net or send a request by email to Bruce@EckelObjects.com.
///:~

7

只要遵守以上的版权声明，读者就可以在自己的项目里和课堂上使用这些代码。

编译器

读者使用的编译器可能不支持本教材所论及的C++的所有特征，尤其是当该编译器并非是其最新版本的时候，这种情况就显得尤为突出。所以实现像C++这样的语言绝非易事；同时读者会希望C++的特征一点点展现，而非一下子全部出现。但是，如果读者试做了教材中的一个例子，结果编译器报告了一大堆错误，这就不一定仅仅是代码或编译器中的一个故障那么简单了——很可能在读者选用的编译器上根本就运行不了那个代码。

教材中的代码已经用很多编译器进行过测试，目的是确保这些代码符合C++标准，并且在尽可能多的编译器上运行。遗憾的是，并非所有的编译器都符合C++标准，因此在使用这些编译器构造可执行文件时，去掉了某些文件。这些被去除的文件在makefiles里都有体现，而makefiles是为这本教材的代码包自动生成的，并且可从 http://www.MindView.net 下载。在makefiles中，从每个程序代码清单开头的注释中都可以看到这些嵌入的排除标记符，这样读者将会知道是否应让某个特定的编译器来运行那个代码（少数情况下，编译器确实会编译代码，但执行动作却是错的，本教材将这些代码也排除在外）。 8

下面就是有关的排除标记符和相应的编译器：

- **{-dmc}**Walter Bright 的 Digital Mars编译器，专为Windows设计，可从www.DigitalMars.com免费下载。这种编译器兼容性超强，整本教材中几乎都看不到该排除标记符。
- **{-g++}**免费的Gnu C++ 3.3.1，在大多数Linux软件包和Macintosh OSX中都预装了该编译器。该编译器也是专为Windows设计的Cygwin的一部分（见下文）。从gcc.gnu.org可以得到其为大多数其他操作平台而设计的版本。
- **{-msc}**Visual C++.NET是微软(Microsoft)推出的第7版编译器（使用前必须先安装Visual Studio.NET，不能免费下载）。
- **{-bor}**Borland C++第6版（不可免费下载；这是最新的版本）。
- **{-edg}**Edison Design Group (EDG) C++。该编译器可用来检测代码是否符合标准C++。只是由于类库的原因才出现了这个标记符，因为本教材采用了Dinkumware有限公司赠送的带有兼容类库的EDG前端免费副本。单独使用编译器不会出现任何编译错误。
- **{-mwcc}**为Macintosh OSX设计的Metrowerks Code Warrior。注意，使用OSX也必须先安装Gnu C++编译器。

如果从http://www.MindView.net下载并解压了本教材的代码包，读者会发现用来为上述编译器建立代码的构造文件（makefiles）。本教材使用免费的GNU-**make**，它在Linux、Cygwin（一个可在Windows上运行的免费Unix shell，详见www.Cygwin.com）环境下运行，也可安装在读者自己的计算机平台上——详见www.gnu.org/software/make。（这些文件在其他**make**上也可能运行，但得不到支持。）一旦安装了**make**，如果在命令行运行方式下键入**make**，就会得到有关如何为上述编译器建立教材中代码的操作步骤。 9

注意，本教材程序示例文件中的这些标记符指出了当时调试用的编译器的版本。很可能在这本教材出版后相应的编译器版本已经升级。也有可能我们在使用很多编译器对本教材中的代码进行编译时，错误地配置了某个编译器；如果没有配错编译器，则相应的代码应该早已经被正确地编译。因此，在自己的编译器上重新调试这些代码，并检查从http://www.MindView.net网站下载的代码是否为最新版本就显得尤为重要。

语言标准

在本教材中，当提到ANSI/ISO C标准时，指的是1989标准，而且一般情况下只是说“C”。只有当有必要区分标准1989 C和较早的版本，如制定标准前的C语言版本时，才会做出区分。在本教材中并不涉及C99。

ANSI/ISO C++委员会很早以前就制定出了第一个C++标准，通常称为C++98。本教材用“标准C++”来指这个标准化语言。如果只说C++，那就意味着是“标准 C++”。C++标准委员会还在继续发布对使用C++的公众群体很重要的信息，这些会促使另一C++标准C++Ox的形成，但它的产生在近几年内不太可能实现。

研讨班和咨询

Bruce Eckel的公司——MindView公司，提供基于本教材中的材料和高级主题的公共实习培训研讨班。每课所讲的都是从各章中精选的内容，每次讲授完毕，后面有一个检测练习阶段，每个学生都能够受到个别指导。我们还提供现场培训、咨询、辅导、设计和代码演练的服务。从**http://www.MindView.net**网站上可获得有关即将开办的研讨班信息、相关报名表和其他联系信息。

10

错误

无论我们怎样挖空心思地去检查错误，总会漏掉一些错误，但这些错误却常常能被热心的读者发现。如果读者发现了任何认为是错误的地方，请使用本教材电子版中的反馈系统与我们联系。读者可在**http://www.MindView.net**网站上找到该系统。非常感谢您的帮助。

关于封面

封面上的艺术作品由Larry O'Brien的妻子Tina Jensen绘制（是的，就是那个在《软件开发》杂志中任多年编辑的Larry O'Brien）。封面不但图画很美，而且也很好地体现了多态性。采用这些图画的灵感来源于Daniel Will-Harris，他是一位封面设计者（www.Will-Harris.com），Bruce 的同事。

致谢

本教材的第2卷曾长时间停留在完成一半的状态，这是因为Bruce还要去做其他事情，即忙于Java、设计模式，尤其是Python (详见www.Python.org)等方面的工作。如果Chuck那时没有心甘情愿地（Chuck 有时认为自己当时的做法很愚蠢）去完成本书的另一半工作，这本教材几乎就不会问世。他是Bruce愿把这份未完的工作托付给的极少数人之一。Chuck 做什么事都追求精确无误，他清晰的解释促成了本教材今天的辉煌。

Jamie King是在本教材完成过程中由Chuck指导的实习生。他是保证这本教材顺利完成必不可少的一分子；他不但为Chuck提供反馈信息，更可贵的是他不懈地发现与置疑他不完全理解的教材中的每一个细节。如果本教材回答了读者的问题，那么很可能就是因为Jamie先问过这个问题了。Jamie还改进了许多示例程序并在每一章的末尾设计了很多练习题。Chuck的另一位由MindView公司资助的实习生Scott Baker帮助完成了第3章的练习题。

11

IBM的Eric Crahen在第11章（并发）的完成过程中发挥了很大作用。在寻找合适的线程

软件包时，我们找到了一个直观、易用的软件包，而且该软件包在完成工作的过程中体现了足够的健壮性。我们从Eric那里得到了这个软件包以及其他一些帮助——他非常乐于合作，而且他还根据使用中的反馈意见来增强他的类库，我们也从他的见解中获益很多。

非常感谢Pete Becker担任本教材的技术编辑。极少有人能像Pete那样出众而且善于表达自己的想法。像Pete那样能够精通C++和软件开发的人也不多。我们也非常感谢Bjorn Karlsson的慷慨和他及时的技术支持，因为他审阅了整个文稿，并做了精辟的批注。

Walter Bright为确保他的Digital Mars C++编译器可以编译教材中的所有实例，曾做过不懈的努力。他还将其编译器作为http://www.DigitalMars.com上可免费下载的软件。谢谢了，Walter!

本教材中的思想和见解来源于许多方面。包括：我们的朋友们，如Andrea Provaglio、Dan Saks、Scott Meyers、Charles Petzold和Michael Wilk；语言开拓者，如Bjarne Stroustrup、Andrew Koenig和Rob Murray；C++标准委员会成员，如Nathan Myers（他给我们提供了特别的帮助和毫无保留的见解）、Herb Sutter、PJ Plauger、Kevlin Henney、David Abrahams、Tom Plum、Reg Charney、Tom Penello、Sam Druker、Uwe Steinmueller、John Spicer、Steve Adamczyk和Daveed Vandevoorde；在软件开发会议上就C++领域发展进程发过言的人员（该会议由Bruce创建和促进，Chuck在会上发表演说）；Michael Seaver、Huston Franklin、David Wagstaff等Chuck 的同事；还有培训班上的学生们，我们需要认真听取他们的问题从而使这本教材更加清晰易懂。

本教材的规划、字体的选择、封面设计以及封面照片由Bruce的朋友Daniel Will-Harris来完成，他是著名作家和设计师。他在初中时就常常研究字母，那时就期待着计算机的发明和桌面排版系统的问世。此外，我们自己做了排版文件，所以排版错误是我们的错误。本教材是使用Microsoft® Word XP来编写的。排版文件也是使用该软件生成的。正文的字体选用Georgia，标题采用的是Verdana，代码采用的字体是Andale Mono（以上指本书英文版）[⊖]。

在这里也感谢Edison Design Group和Dinkumware有限公司慷慨的专业技术人员，感谢他们让我们免费拷贝了他们的编译器和类库（分别拷贝的）。没有他们的专业帮助及无私的给予，本教材中的一些程序示例根本无法得到调试。还要感谢Howard Hinnant和Metrowerks的工作人员，感谢让我们拷贝他们的编译器；还要感谢Sandy Smith和SlickEdit的人员，正是他们这么多年来一直为Chuck提供世界一流的编写环境。此外，Greg Comeau也贡献出了他的基于EDG的编译器的拷贝（Comeau C++），这里一并表示感谢。 12

特别要感谢我们的所有老师及所有学生（他们也是我们的老师）。

Evan Cofsky(Evan@TheUnixMan.com) 提供了服务器方面的多种帮助。他也曾用其擅长的语言——Python，在程序研发工作中为本教材尽心尽力。Sharlynn Cobaugh 和Paula Steuer也帮了不少忙，他们的帮助使Bruce 能够从繁忙的项目中脱身出来。

Bruce 的妻子Dawn McGee在本教材编写期间给予了很多灵感和巨大热情。以下是对本教材的出版给予过支持的朋友们的部分名单，但不是全部：Mark Western、Gen Kiyooka、Kraig Brockschmidt、Zack Urlocker、Andrew Binstock、Neil Rubenking、Steve Sinofsky、JD Hildebrandt、Brian McElhinney、Brinkley Barr、*Midnight Engineering* 杂志的Bill Gates、Larry Constantine和Lucy Lockwood、Tom Keffer、Greg Perry、Dan Putterman、Christi

⊖ 本书的英文影印版已由机械工业出版社出版。——编辑注

Westphal、Gene Wang、Dave Mayer、David Intersimone、Claire Sawyers、几位意大利籍朋友（Andrea Provaglio、Laura Fallai、Marco Cantu、Corrado、Ilsa 和 Christina Giustozzi）、Chris 和 Laura Strand、Almquist一家、Brad Jerbic, John Kruth & Marilyn Cvitanic, Holly Payne（就是那个著名的小说家！）、Mark Mabry、Robbins一家、Moelter 一家（还有McMillan一家）、Wilk一家、Dave stoner、Laurie Adams、Cranston一家、Larry Fogg、Mike 和 Karen Sequeira、Gary Entsminger 和Allison Brody、Chester Andersen、Joe Lordi、Dave和Brenda Bartlett、Rentschler一家、Sudek一家、Lynn和Todd，还有他们的家人。当然，还有我们的父母、Sandy、James和Natalie、Kim和Jared、Isaac和Abbi。

目　录

出版者的话
专家指导委员会
译者序
前言

第一部分　建立稳定的系统

第1章　异常处理 ……2
1.1　传统的错误处理 ……2
1.2　抛出异常 ……4
1.3　捕获异常 ……5
1.3.1　try块 ……5
1.3.2　异常处理器 ……5
1.3.3　终止和恢复 ……6
1.4　异常匹配 ……7
1.4.1　捕获所有异常 ……8
1.4.2　重新抛出异常 ……8
1.4.3　不捕获异常 ……9
1.5　清理 ……10
1.5.1　资源管理 ……11
1.5.2　使所有事物都成为对象 ……12
1.5.3　auto_ptr ……14
1.5.4　函数级的try块 ……15
1.6　标准异常 ……16
1.7　异常规格说明 ……18
1.7.1　更好的异常规格说明 ……21
1.7.2　异常规格说明和继承 ……21
1.7.3　什么时候不使用异常规格说明 ……22
1.8　异常安全 ……22
1.9　在编程中使用异常 ……25
1.9.1　什么时候避免异常 ……25
1.9.2　异常的典型应用 ……26
1.10　使用异常造成的开销 ……28
1.11　小结 ……30
1.12　练习 ……30
第2章　防御性编程 ……32
2.1　断言 ……34
2.2　一个简单的单元测试框架 ……36
2.2.1　自动测试 ……37
2.2.2　TestSuite框架 ……39
2.2.3　测试套件 ……42
2.2.4　测试框架的源代码 ……43
2.3　调试技术 ……47
2.3.1　用于代码跟踪的宏 ……48
2.3.2　跟踪文件 ……48
2.3.3　发现内存泄漏 ……49
2.4　小结 ……53
2.5　练习 ……54

第二部分　标准C++库

第3章　深入理解字符串 ……58
3.1　字符串的内部是什么 ……58
3.2　创建并初始化C++字符串 ……59
3.3　对字符串进行操作 ……62
3.3.1　追加、插入和连接字符串 ……62
3.3.2　替换字符串中的字符 ……63
3.3.3　使用非成员重载运算符连接 ……66
3.4　字符串的查找 ……67
3.4.1　反向查找 ……70
3.4.2　查找一组字符第1次或最后一次出现的位置 ……71
3.4.3　从字符串中删除字符 ……73
3.4.4　字符串的比较 ……74
3.4.5　字符串和字符的特性 ……77
3.5　字符串的应用 ……81
3.6　小结 ……85
3.7　练习 ……85
第4章　输入输出流 ……88
4.1　为什么引入输入输出流 ……88

4.2 救助输入输出流 ……91
4.2.1 插入符和提取符 ……91
4.2.2 通常用法 ……94
4.2.3 按行输入 ……95
4.3 处理流错误 ……96
4.4 文件输入输出流 ……98
4.4.1 一个文件处理的例子 ……98
4.4.2 打开模式 ……100
4.5 输入输出流缓冲 ……100
4.6 在输入输出流中定位 ……102
4.7 字符串输入输出流 ……104
4.7.1 输入字符串流 ……105
4.7.2 输出字符串流 ……106
4.8 输出流的格式化 ……109
4.8.1 格式化标志 ……109
4.8.2 格式化域 ……110
4.8.3 宽度、填充和精度设置 ……111
4.8.4 一个完整的例子 ……111
4.9 操纵算子 ……114
4.9.1 带参数的操纵算子 ……114
4.9.2 创建操纵算子 ……116
4.9.3 效用算子 ……117
4.10 输入输出流程序举例 ……119
4.10.1 维护类库的源代码 ……119
4.10.2 检测编译器错误 ……122
4.10.3 一个简单的数据记录器 ……124
4.11 国际化 ……127
4.11.1 宽字符流 ……128
4.11.2 区域性字符流 ……129
4.12 小结 ……131
4.13 练习 ……131
第5章 深入理解模板 ……134
5.1 模板参数 ……134
5.1.1 无类型模板参数 ……134
5.1.2 默认模板参数 ……136
5.1.3 模板类型的模板参数 ……137
5.1.4 typename关键字 ……141
5.1.5 以template关键字作为提示 ……142
5.1.6 成员模板 ……143
5.2 有关函数模板的几个问题 ……145
5.2.1 函数模板参数的类型推断 ……145
5.2.2 函数模板重载 ……148
5.2.3 以一个已生成的函数模板地址作为参数 ……149
5.2.4 将函数应用到STL序列容器中 ……152
5.2.5 函数模板的半有序 ……154
5.3 模板特化 ……155
5.3.1 显式特化 ……155
5.3.2 半特化 ……156
5.3.3 一个实例 ……158
5.3.4 防止模板代码膨胀 ……160
5.4 名称查找问题 ……163
5.4.1 模板中的名称 ……163
5.4.2 模板和友元 ……167
5.5 模板编程中的习语 ……171
5.5.1 特征 ……171
5.5.2 策略 ……175
5.5.3 奇特的递归模板模式 ……177
5.6 模板元编程 ……178
5.6.1 编译时编程 ……179
5.6.2 表达式模板 ……185
5.7 模板编译模型 ……190
5.7.1 包含模型 ……190
5.7.2 显式实例化 ……191
5.7.3 分离模型 ……192
5.8 小结 ……193
5.9 练习 ……194
第6章 通用算法 ……196
6.1 概述 ……196
6.1.1 判定函数 ……198
6.1.2 流迭代器 ……200
6.1.3 算法复杂性 ……201
6.2 函数对象 ……202
6.2.1 函数对象的分类 ……203
6.2.2 自动创建函数对象 ……203
6.2.3 可调整的函数对象 ……206
6.2.4 更多的函数对象例子 ……207
6.2.5 函数指针适配器 ……212
6.2.6 编写自己的函数对象适配器 ……216
6.3 STL算法目录 ……219

6.3.1 实例创建的支持工具 ……220
6.3.2 填充和生成 ……223
6.3.3 计数 ……224
6.3.4 操作序列 ……225
6.3.5 查找和替换 ……228
6.3.6 比较范围 ……233
6.3.7 删除元素 ……235
6.3.8 对已排序的序列进行排序和运算 …238
6.3.9 堆运算 ……245
6.3.10 对某一范围内的所有元素进行运算 ……245
6.3.11 数值算法 ……251
6.3.12 通用实用程序 ……253
6.4 创建自己的STL风格算法 ……254
6.5 小结 ……255
6.6 练习 ……256
第7章 通用容器 ……260
7.1 容器和迭代器 ……260
7.2 概述 ……261
7.2.1 字符串容器 ……265
7.2.2 从STL容器继承 ……266
7.3 更多迭代器 ……268
7.3.1 可逆容器中的迭代器 ……269
7.3.2 迭代器的种类 ……270
7.3.3 预定义迭代器 ……271
7.4 基本序列容器：vector、list和deque …275
7.4.1 基本序列容器的操作 ……275
7.4.2 向量 ……277
7.4.3 双端队列 ……282
7.4.4 序列容器间的转换 ……284
7.4.5 被检查的随机访问 ……285
7.4.6 链表 ……286
7.4.7 交换序列 ……290
7.5 集合 ……291
7.6 堆栈 ……297
7.7 队列 ……299
7.8 优先队列 ……302
7.9 持有二进制位 ……309
7.9.1 bitset<n> ……310
7.9.2 vector<bool> ……312
7.10 关联式容器 ……314
7.10.1 用于关联式容器的发生器和填充器 ……317
7.10.2 不可思议的映像 ……319
7.10.3 多重映像和重复的关键字 ……320
7.10.4 多重集合 ……322
7.11 将STL容器联合使用 ……325
7.12 清除容器的指针 ……327
7.13 创建自己的容器 ……328
7.14 对STL的扩充 ……330
7.15 非STL容器 ……331
7.16 小结 ……335
7.17 练习 ……335

第三部分 专 题

第8章 运行时类型识别 ……340
8.1 运行时类型转换 ……340
8.2 typeid 操作符 ……344
8.2.1 类型转换到中间层次类型 ……345
8.2.2 void型指针 ……346
8.2.3 运用带模板的RTTI ……347
8.3 多重继承 ……348
8.4 合理使用RTTI ……348
8.5 RTTI的机制和开销 ……352
8.6 小结 ……352
8.7 练习 ……353
第9章 多重继承 ……355
9.1 概论 ……355
9.2 接口继承 ……356
9.3 实现继承 ……358
9.4 重复子对象 ……362
9.5 虚基类 ……365
9.6 名字查找问题 ……372
9.7 避免使用多重继承 ……374
9.8 扩充一个接口 ……375
9.9 小结 ……378
9.10 练习 ……378
第10章 设计模式 ……380
10.1 模式的概念 ……380
10.2 模式分类 ……381

10.3 简化习语 ……382
10.3.1 信使 ……382
10.3.2 收集参数 ……383
10.4 单件 ……384
10.5 命令：选择操作 ……388
10.6 消除对象耦合 ……391
10.6.1 代理模式：作为其他对象的前端…392
10.6.2 状态模式：改变对象的行为 ……393
10.7 适配器模式 ……395
10.8 模板方法模式 ……396
10.9 策略模式：运行时选择算法 ……397
10.10 职责链模式：尝试采用一系列策略模式 ……398
10.11 工厂模式：封装对象的创建 ……400
10.11.1 多态工厂 ……402
10.11.2 抽象工厂 ……404
10.11.3 虚构造函数 ……406
10.12 构建器模式：创建复杂对象 ……410
10.13 观察者模式 ……415
10.13.1 “内部类”方法 ……417
10.13.2 观察者模式举例 ……419
10.14 多重派遣 ……422
10.15 小结 ……428
10.16 练习 ……428
第11章 并发 ……430
11.1 动机 ……430
11.2 C++中的并发 ……431
11.3 定义任务 ……433
11.4 使用线程 ……434
11.4.1 创建有响应的用户界面 ……435
11.4.2 使用执行器简化工作 ……437
11.4.3 让步 ……439
11.4.4 休眠 ……440
11.4.5 优先权 ……441
11.5 共享有限资源 ……442
11.5.1 保证对象的存在 ……442
11.5.2 不恰当地访问资源 ……445
11.5.3 访问控制 ……447
11.5.4 使用保护简化编码 ……448
11.5.5 线程本地存储 ……451
11.6 终止任务 ……452
11.6.1 防止输入/输出流冲突 ……452
11.6.2 举例观赏植物园 ……453
11.6.3 阻塞时终止 ……456
11.6.4 中断 ……457
11.7 线程间协作 ……461
11.7.1 等待和信号 ……461
11.7.2 生产者-消费者关系 ……464
11.7.3 用队列解决线程处理的问题 ……467
11.7.4 广播 ……471
11.8 死锁 ……476
11.9 小结 ……480
11.10 练习 ……481

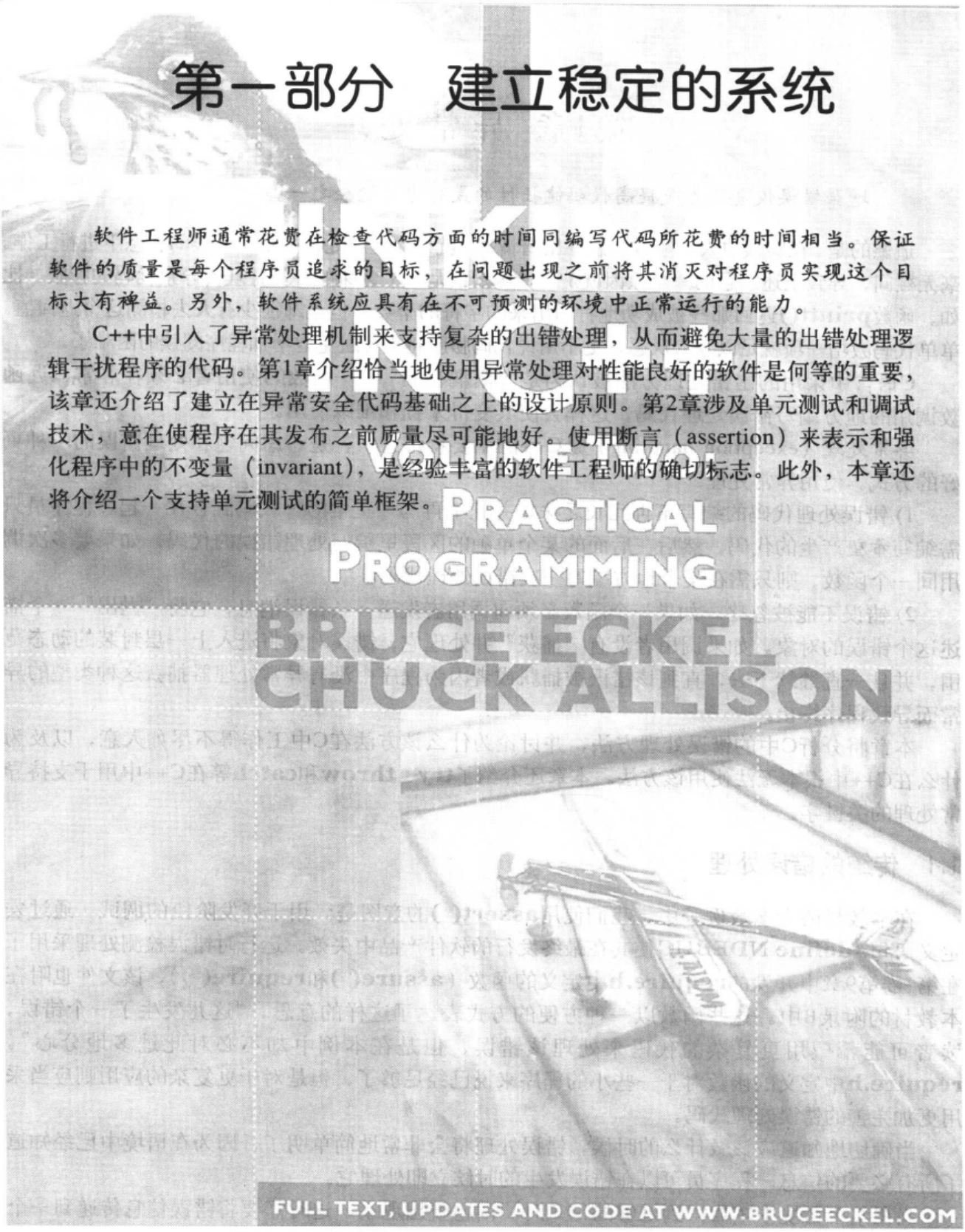
附　录

附录A 推荐读物 ……484
附录B 其他 ……488
索引 ……493

第一部分　建立稳定的系统

软件工程师通常花费在检查代码方面的时间同编写代码所花费的时间相当。保证软件的质量是每个程序员追求的目标，在问题出现之前将其消灭对程序员实现这个目标大有裨益。另外，软件系统应具有在不可预测的环境中正常运行的能力。

C++中引入了异常处理机制来支持复杂的出错处理，从而避免大量的出错处理逻辑干扰程序的代码。第1章介绍恰当地使用异常处理对性能良好的软件是何等的重要，该章还介绍了建立在异常安全代码基础之上的设计原则。第2章涉及单元测试和调试技术，意在使程序在其发布之前质量尽可能地好。使用断言（assertion）来表示和强化程序中的不变量（invariant），是经验丰富的软件工程师的确切标志。此外，本章还将介绍一个支持单元测试的简单框架。



第1章 异常处理

增强错误恢复能力是提高代码健壮性的最有力的途径之一。

遗憾的是，在实践中人们通常会忽略出错情况，就好像程序处在一个无错误的状态下进行工作。毫无疑问，导致上述问题的一个原因就是，检测错误是一个乏味的工作并且会导致代码的膨胀。比如，函数**printf()**返回那些被成功地打印出来的字符的个数，但是却很少有人去检测这个返回值。单单代码激增一项就足以令人厌恶，更不用说代码膨胀将不可避免地增加程序阅读的困难了。

C语言中采用的出错处理方法被认为是“紧耦合的”——函数的使用者必须在非常靠近函数调用的地方编写错误处理代码，这样会使其变得笨拙和难以使用。

异常处理（exception handling）是C++的主要特征之一，是考虑问题和处理错误的一种更好的方式。使用异常处理：

1) 错误处理代码的编写不再冗长乏味，并且不再与“正常的”代码混合在一起。程序员只需编写希望产生的代码，然后在后面的某个单独的区段里编写处理错误的代码。如果要多次调用同一个函数，则只需在某个地方编写一次错误处理代码。

2) 错误不能被忽略。如果一个函数必须向调用者发送一条错误消息，它将“抛出”一个描述这个错误的对象。如果调用者没有“捕获”并处理它，错误对象将进入上一层封装的动态范围，并且一直继续下去，直到该错误被捕获或者因为程序中没有异常处理器捕获这种类型的异常而导致程序终止。

本章将分析C中的错误处理方法，并讨论为什么该方法在C中工作得不尽如人意，以及为什么在C++中根本无法使用该方法。本章还介绍了**try**、**throw**和**catch**等在C++中用于支持异常处理的关键字。

1.1 传统的错误处理

在本教材的大多数例子中，我们使用**assert( )**的意图是：用于开发阶段的调试，通过宏定义语句**#define NDEBUG**使其在最终发行的软件产品中失效。运行时错误检测处理采用了在第1卷第9章中开发的**require.h**中定义的函数（**assure( )**和**require( )**），该文件也附在本教材的附录B中。这些函数以一种方便的方式表达了这样的意思，“这儿发生了一个错误，读者可能希望用更复杂的代码来处理该错误，但是在本例中却不必对此过多地分心”。**require.h**中定义的函数对于一些小的程序来说已经足够了，但是对于更复杂的应用则应当采用更加完善的错误处理代码。

当确切地知道应该做什么的时候，错误处理将会非常地简单明了，因为在语境中已经知道了所有必要的信息。程序员可以在错误发生的时候立即处理它。

当在某个语境中不能得到足够的信息时，问题就出现了，这时需要将错误信息传递到一个包含上述信息的不同的语境中去。在C语言中，有三种方法处理这种情况：

1) 在函数中返回错误信息，如果无法将返回值用于这个方面，则设置一个全局的错误状态标志（标准C中提供了**errno**和**perror( )**来支持这种方法）。就像前面提到的一样，程序员会因为冗长乏味的错误检查而倾向于忽略错误信息。另外，从一个出现异常情况的函数中返回可

能根本没有意义。

2) 使用鲜为人知的标准C库中的信号处理系统，由函数**signal( )**（用以推断事件发生时出现了什么情况）和函数**raise( )**（产生一个事件）实现。同样，该方式的耦合度非常高，因为如果要使用可能产生信号的库函数，库的使用者必须了解和设置适当的信号处理机制。在大型项目中，不同的库产生的信号值可能会发生冲突。

3) 使用标准C库中的非局部跳转函数：**setjmp( )**和**longjmp( )**。使用**setjmp( )**可以在程序中保存一个已知的无错误的状态，一旦发生错误，就可以通过调用**longjmp( )**返回到该状态。同样，这使得错误发生的位置与保存状态的位置之间产生高度耦合。 17

当考虑C++的错误处理方案时，会涉及到另一个关键问题：C中的信号处理方法和函数**setjmp( )/longjmp( )**并不调用析构函数，所以对象不会被正确地清理。（实际上，如果**longjmp( )**跳转到超出析构函数的作用范围以外的地方，则程序的行为是不可预料的）。在这种情况下不可能有效地从错误状态中恢复，因为程序中留下了未被清理但又不能被再次访问的对象。下面的代码演示了函数**setjmp( )/longjmp( )**的使用方法：

```
//: C01:Nonlocal.cpp
// setjmp() & longjmp().
#include <iostream>
#include <csetjmp>
using namespace std;

class Rainbow {
public:
  Rainbow() { cout << "Rainbow()" << endl; }
  ~Rainbow() { cout << "~Rainbow()" << endl; }
};

jmp_buf kansas;

void oz() {
  Rainbow rb;
  for(int i = 0; i < 3; i++)
    cout << "there's no place like home" << endl;
  longjmp(kansas, 47);
}

int main() {
  if(setjmp(kansas) == 0) {
    cout << "tornado, witch, munchkins..." << endl;
    oz();
  } else {
    cout << "Auntie Em! "
         << "I had the strangest dream..."
         << endl;
  }
} ///:~
```

18

函数**setjmp( )**的行为很特别，因为如果直接调用它，它便会将所有与当前处理器相关的状态信息（比如指令指针的内容、运行时栈指针等）保存到**jmp_buf**中去并返回0。在这种情况下，它的表现与一个普通的函数一样。然而，如果使用同一个**jmp_buf**调用**longjmp( )**，则函数返回时就好像刚从**setjmp( )**中返回时一样——又回到了刚刚从**setjmp( )**返回的地方。这一次，返回值是调用**longjmp( )**时所使用的第二个参数(argument)，因此可以通过这个值断定程序实际是从**longjmp( )**返回的。读者可以想像有很多不同的**jmp_buf**的情况，程序可以跳到多个不同的位置。局部**goto**（使用标号）和这种非局部跳转的区别在于，通过**setjmp( ) /**

longjmp( )程序可以返回到运行栈中任何预先确定的位置（即任何调用**setjmp( )**的位置）。

在C++中使用**longjmp( )**的问题是它不能识别对象。特别是，当跳出某个作用域的时候，它不会调用对象的析构函数[⊖]。而析构函数的调用在C++中是必需的操作，所以这种方法不能用于C++。实际上，C++标准已经说明，使用**goto**跳入某个作用域（有效地跳过构造函数的调用），或使用**longjmp( )**跳出某个作用域而且这个作用域的栈中有某个对象需要析构时，程序的行为是不确定的。

1.2 抛出异常

如果在程序的代码中出现了异常的情况——也就是说，通过当前语境无法获得足够的信息以决定应该采取什么样的措施——程序员可以创建一个包含错误信息的对象并把它抛出当前语境，通过这种方式将错误信息发送到更大范围的语境中。这种方式被称为“*抛出一个异常*”。如下所示：

```
//: C01:MyError.cpp {RunByHand}

class MyError {
  const char* const data;
public:
  MyError(const char* const msg = 0) : data(msg) {}
};

void f() {
  // Here we "throw" an exception object:
  throw MyError("something bad happened");
}

int main() {
  // As you'll see shortly, we'll want a "try block" here:
  f();
} ///:~
```

19

在上面的代码中，**MyError**是一个普通的类，它的构造函数接受一个**char***类型的变量作为参数。在抛出一个异常时，可以使用任意的类型（包括内置类型），但通常应当为抛出的异常创建特定的类。

关键字**throw**将导致一系列事情的发生。首先，它将创建程序所抛出的对象的一个拷贝，然后，实际上，包含**throw**表达式的函数返回了这个对象，即使该函数原先并未设计为返回这种对象类型。一种简单的考虑异常处理的方式是将其看作是一种*交错返回机制*（alternate return mechanism）（如果仔细分析这个问题，就会发现自己陷入了困境）。当然也可以通过抛出一个异常而离开正常的作用域。在任何一种情况下都会返回一个值，并且退出函数或作用域。

由于异常会造成程序返回到某个地方，而这个地方与正常函数调用时遇到**return**语句后程序返回到的地方完全不同，所以异常与**return**语句的相似性仅止于此。（可以在程序中恰当的部分编写异常处理器代码，这段代码可能与异常抛出的位置相差很远。）另外，异常发生之前创建的局部对象被销毁。这种对局部对象的自动清理通常被称为“*栈反解*（stack unwinding）”。

而且，在程序中可以抛出许多程序员希望的不同类型的对象。典型的做法是，程序员要为

⊖ 当读者运行这个例子的时候，可能会感到奇怪——某些C++编译器包含了扩展的**longjmp( )**，能够清除栈中的对象。但这种行为是不可移植的。

每一种不同的异常情况抛出不同类型的对象。这么做的目的是为了将错误信息保存在相应的对象和对象类名中，这样，在调用者的语境中根据这些信息就可以决定应该如何处理这些异常了。

1.3 捕获异常

20

就像前面提到的一样，C++异常处理机制的一个好处是，可以使程序员在一个地方专注于所要解决的问题，而在另一个地方对这段代码所产生的错误进行处理。

1.3.1 try块

如果在一个函数内部抛出了异常（或者被这个函数所调用的其他函数抛出了异常），这个函数将会因为抛出异常而退出。如果不想因为一个**throw**而退出函数，可以在函数中试图解决实际产生程序设计问题的地方（和可能产生异常的地方）设置一个**try**块。这个块被称做**try**块的原因是程序需要在这里尝试着调用各种函数。**try**块只是一个普通的程序块，由关键字**try**引导：

```
try {
  // Code that may generate exceptions
}
```

如果希望通过仔细地检查每一个被调用函数的返回值来发现错误，程序员必须围绕每一次函数调用编写初始化和检测代码，即使多次调用同一个函数也是如此。使用异常处理时，可以将所有工作放入**try**块中，然后在**try**块的后面处理可能产生的异常。这样一来，代码将更容易编写和阅读，因为代码的设计目标不会被错误处理所干扰。

1.3.2 异常处理器

当然，被抛出的异常肯定会在某个地方终止。这个地方就是*异常处理器*（exception handler），程序员需要为每一种想捕获的异常类型设置一个异常处理器。然而，多态对于异常同样有效，因此一个异常处理器可以处理某种异常类型和这种类型的派生类。

异常处理器紧接着**try**块，并且由关键字**catch**标识：

```
try {
  // Code that may generate exceptions
} catch(type1 id1) {
  // Handle exceptions of type1
} catch(type2 id2) {
  // Handle exceptions of type2
} catch(type3 id3)
  // Etc...
} catch(typeN idN)
  // Handle exceptions of typeN
}
// Normal execution resumes here...
```

21

catch子句的语法类似于带有单一参数的函数。可以在异常处理器内部使用标识符（**id1**、**id2**等），就像使用函数的参数一样。如果不需要在异常处理器中使用标识符，这些标识符可以省略。异常类型通常提供了对其进行处理的足够的信息。

异常处理器都必须紧跟在**try**块之后。一旦某个异常被抛出，异常处理机制将会依次寻找参数类型与异常类型相匹配的异常处理器。找到第一个这样的异常处理器后，程序的执行流程进入这个**catch**子句，于是系统就可以认为该异常已经处理了。（查找异常处理器的过程在找到第一个匹配的**catch**子句之后就终止了。）只有匹配的**catch**子句才会执行；在执行完最后一个与该try块相关的异常处理器后，程序又恢复到正常的控制流程。

注意，在**try**块中，不同的函数调用可能产生相同类型的异常，这时，只需要一个异常处理器就可以了。

为了举例说明**try**和**catch**，我们在这里修改了**Nonlocal.cpp**，将其中的**setjmp( )**用一个**try**块代替，将**longjmp( )**用一个**throw**语句代替：

```
//: C01:Nonlocal2.cpp
// Illustrates exceptions.
#include <iostream>
using namespace std;

class Rainbow {
public:
  Rainbow() { cout << "Rainbow()" << endl; }
  ~Rainbow() { cout << "~Rainbow()" << endl; }
};

void oz() {
  Rainbow rb;
  for(int i = 0; i < 3; i++)
    cout << "there's no place like home" << endl;
  throw 47;
}

int main() {
  try {
    cout << "tornado, witch, munchkins..." << endl;
    oz();
  } catch(int) {
    cout << "Auntie Em! I had the strangest dream..."
         << endl;
  }
} ///:~
```

22

当执行函数**oz( )**中的**throw**语句时，程序的控制流程开始回溯，直到找到某个带有**int**型参数的**catch**子句为止。程序在这个**catch**子句的主体中恢复运行。这个程序与**Nonlocal.cpp**的最重要的区别在于，当**throw**语句造成程序的执行过程从**oz( )**函数返回时，对象**rb**的析构函数被调用。

1.3.3 终止和恢复

在异常处理理论中有两个基本的模型：终止和恢复。在终止（termination）（C++支持这种模型）模型中，假定错误非常严重，以至于不可能在异常发生的地点自动恢复程序的执行。也就是说，无论谁抛出一个异常，都表明程序已经陷入了无法挽救的困境，并且不需要再返回发生异常的地方。

另一个异常处理模型被称为恢复（resumption）模型，在20世纪60年代，PL/I语言首先引入该模型[⊖]。使用恢复模型意味着异常处理器希望能够校正这种情况，然后自动地重新执行发生错误的代码，并希望第二次执行能够成功。如果希望在C++中重新恢复程序的执行，则必须显式地将程序的执行流程转移到错误发生的地方，通常的做法是重新调用发生错误的函数。将**try**块放到**while**循环中，不断地重新执行**try**块中的程序，直到产生满意的结果，这种做法并不罕见。

23

历史上，使用支持恢复性异常处理模型的操作系统的程序员们，最终使用的是类似终止模

⊖ 通过ON ERROR功能，BASIC语言长期以来支持一种有限形式的恢复性异常处理模型。

型的代码并跳过了异常恢复。虽然恢复模型非常吸引人，但是在实践中并没有多大用处。其中一个原因或许就是发生异常的位置和异常处理器之间的距离。对于远处的异常处理器来说，终止执行是一个问题；而另一个问题是，对于一个大型系统来说，在很多位置都可能发生异常，从异常发生的位置跳转到远处的异常处理器然后再返回，这在概念上也十分困难。

1.4 异常匹配

一个异常被抛出以后，异常处理系统将按照在源代码中出现的顺序查找最近的异常处理器。一旦找到匹配的异常处理器，就认为该异常已经被处理了而不再继续查找下去。

匹配一个异常并不要求异常与其处理器之间完全相关。一个对象或者是指向派生类对象的引用都会与其基类处理器匹配。（然而，如果异常处理器是针对对象而不是针对引用的，这个异常对象将会被“切割”——被截取成基类对象——就好像一个基类对象被传递给了异常处理器。除了丢失派生类包含的所有附加信息之外，这并没有什么危害。）由于这种原因，并且为了避免再次拷贝异常对象，最好是通过引用而不是通过值来捕获异常[⊖]。如果一个指针被抛出，将使用通常的标准指针转换来匹配异常。但是，在匹配过程中，不会将一种异常类型自动转换成另一种异常类型。比如：

```
//: C01:Autoexcp.cpp
// No matching conversions.
#include <iostream>
using namespace std;
class Except1 {};

class Except2 {
public:
  Except2(const Except1&) {}
};

void f() { throw Except1(); }

int main() {
  try { f();
  } catch(Except2&) {
    cout << "inside catch(Except2)" << endl;
  } catch(Except1&) {
    cout << "inside catch(Except1)" << endl;
  }
} ///:~
```

24

尽管读者可能会认为，通过使用*转换构造函数*（converting constructor）将一个**Except1**对象转换成一个**Except2**对象，可以使得第一个异常处理器被匹配。但是，异常处理系统在处理异常的过程中并不做这种转换，结果是，程序在**Except1**异常处理器那里结束。

下面的例子显示了基类的异常处理器怎样就能够捕获派生类异常：

```
//: C01:Basexcpt.cpp
// Exception hierarchies.
#include <iostream>
using namespace std;

class X {
```

⊖ 读者可能总是希望在异常处理器中通过**const**引用来指定异常对象。（极少有程序在异常处理器中修改异常和重新抛出异常。）我们不对这种做法武断地评价。

```
public:
  class Trouble {};
  class Small : public Trouble {};
  class Big : public Trouble {};
  void f() { throw Big(); }
};

int main() {
  X x;
  try {
    x.f();
  } catch(X::Trouble&) {
    cout << "caught Trouble" << endl;
  // Hidden by previous handler:
  } catch(X::Small&) {
    cout << "caught Small Trouble" << endl;
  } catch(X::Big&) {
    cout << "caught Big Trouble" << endl;
  }
} ///:~
```

25

在这里，异常处理机制总是将**Trouble**对象，或派生自**Trouble**的任何对象（通过公有继承）[⊖]，匹配到第一个异常处理器。由于第一个异常处理器截获了所有的异常，所以第二和第三个异常处理器永远不会被调用。比较有意义的做法是，首先捕获派生类异常，并且将基类放到最后用于捕获其他不太具体的异常。

需要注意的是，这些例子都是通过引用来捕获异常的，尽管对这些类而言这一点并不重要，因为派生类中没有附加的成员，而且异常处理器中也没有参数标识符。通常情况下，应该在异常处理器中使用引用参数而不是值参数，以防异常对象所包含的信息被切割掉。

1.4.1 捕获所有异常

有时候，程序员可能希望创建一个异常处理器，使其能够捕获所有类型的异常。用省略号代替异常处理器的参数列表就可以实现这一点：

```
catch(...) {
  cout << "an exception was thrown" << endl;
}
```

由于省略号异常处理器能够捕获任何类型的异常，所以最好将它放在异常处理器列表的最后，从而避免架空它后面的异常处理器。

26

省略号异常处理器不允许接受任何参数，所以无法得到任何有关异常的信息，也无法知道异常的类型。它是一个“全能捕获者”。这种**catch**子句经常用于清理资源并重新抛出所捕获的异常。

1.4.2 重新抛出异常

当需要释放某些资源时，例如网络连接或位于堆上的内存需要释放时，通常希望重新抛出一个异常。（详见本章后面的“资源管理”一节。）如果发生了异常，读者不必关心到底是什么错误导致了异常的发生——只需要关闭以前打开的一个连接。此后，读者希望在某些更接近用户的语境（也就是说，在调用链中的更高层次）中对异常进行处理。在这种情况下，省略号异常处理器正符合这种的要求。这种处理方法，可以捕获所有异常，清理相关资源，然后重新抛

⊖ 只有明确的、可访问的基类才能够捕获派生类异常。这种规则将验证异常所需的运行代价减到最小。请记住，异常是在运行时而不是在编译时被检测的，因此，编译时存在的大量信息在处理异常的时候是不存在的。

出该异常，以使得其他地方的异常处理器能够处理该异常。在一个异常处理器内部，使用不带参数的**throw**语句可以重新抛出异常：

```
catch(...) {
  cout << "an exception was thrown" << endl;
  // Deallocate your resource here, and then rethrow
  throw;
}
```

与同一个**try**块相关的随后的**catch**子句仍然会被忽略——**throw**子句把这个异常传递给位于更高一层语境中的异常处理器。另外，这个异常对象的所有信息都会保留，所以位于更高层语境中的捕获特定类型异常的异常处理器能够获取这个对象包含的所有信息。

1.4.3 不捕获异常

就像在这一章的开始所说的，因为异常不可以忽略，而且将错误处理逻辑从问题发生地附近分开，所以异常处理被认为比传统的返回错误代码的技术要好。如果**try**块之后的异常处理器不能匹配所抛出的异常，那么这个异常就会被传递给位于更高一层语境中的异常处理器，也就是说函数或**try**块周围的**try**块不捕获这个异常。（由于**try**块的位置处在函数调用链的较高层次，所以乍看起来并不明显。）这个过程持续进行，直到某一层存在一个异常处理器能够匹配这个异常。这时，异常处理系统认为已经“捕获”了这个异常，不再继续查找其他的异常处理器。

27

1. terminate( ) 函数

如果没有任何一个层次的异常处理器能够捕获某种异常，一个特殊的库函数**terminate( )**（在头文件**<exception>**中定义）会被自动调用。默认情况下，**terminate( )**调用标准C库函数**abort( )**使程序执行异常终止而退出。在Unix系统中，**abort( )**还会导致主存储器信息转储（core dump）。当**abort( )**被调用时，程序不会调用正常的终止函数，也就是说，全局对象和静态对象的析构函数不会执行。在下列两种情况下**terminate( )**函数也会执行：局部对象的析构函数抛出异常时，栈正在进行清理工作（也称栈反解，即异常的抛出过程被打断）；或者是全局对象或静态对象的构造函数或析构函数抛出一个异常。（一般来说，不允许析构函数抛出异常。）

2. set_terminate( )函数

通过使用标准的**set_terminate( )**函数，可以设置读者自己的**terminate( )**函数，**set_terminate( )**返回被替换的指向**terminate( )**函数的指针（第一次调用**set_terminate( )**函数时，返回函数库中默认的**terminate( )**函数的指针），这样就可以在需要的时候恢复原来的**terminate( )**。自定义的**terminate( )**函数不能有参数，而且其返回值的类型必须是**void**。另外，这里所设置的**terminate( )**函数不能返回（return）也不能抛出异常，而且它必须执行某种方式的程序终止逻辑。如果**terminate( )**函数被调用，就意味着问题已经无法解决了。

下面的例子显示了如何使用**set_terminate( )**函数。在这个例子中，**set_terminate( )**函数的返回值被保存下来并且被还原，使得**terminate( )**函数可以用来帮助隔离产生不可捕获的异常的代码块。

```
//: C01:Terminator.cpp
// Use of set_terminate(). Also shows uncaught exceptions.
#include <exception>
#include <iostream>
using namespace std;
```

```
void terminator() {
  cout << "I'll be back!" << endl;
  exit(0);
}
void (*old_terminate)() = set_terminate(terminator);

class Botch {
public:
  class Fruit {};
  void f() {
    cout << "Botch::f()" << endl;
    throw Fruit();
  }
  ~Botch() { throw 'c'; }
};

int main() {
  try {
    Botch b;
    b.f();
  } catch(...) {
    cout << "inside catch(...)" << endl;
  }
} ///:~
```

28

old_terminate的定义乍看起来有点让人迷惑：它不但创建了一个指向函数的指针，而且用**set_terminate( )**函数的返回值初始化这个指针。尽管读者可能熟悉在指向函数的指针的声明之后紧跟一个分号的定义形式，但是，在这段代码中使用的恰好是另一种可以在定义时初始化的变量。

类**Botch**不仅在函数**f( )**中抛出异常，而且在析构函数中也抛出异常。在**main( )**函数中可以看出，正是析构函数中抛出的异常造成了程序调用**terminate( )**。尽管异常处理器被声明成**catch(...)**，看起来应该能够捕获所有异常，不会导致对**terminate( )** 的调用，但是，实际上**terminate( )**总会被调用。程序在处理一个异常的时候会释放在栈上分配的对象，这时，**Botch**的析构函数被调用，从而产生了第二个异常，这个异常迫使程序调用**terminate( )**。因此，抛出异常或由于某种原因导致一个异常被抛出的析构函数通常被认为象征着拙劣的设计或糟糕的编码。

1.5 清理

异常处理的魅力之一在于程序能够从正常的处理流程中跳转到恰当的异常处理器中。如果异常抛出时，程序不做恰当的清理工作，那么异常处理本身并没有什么用处。C++的异常处理必须确保当程序的执行流程离开一个作用域的时候，对于属于这个作用域的所有由构造函数建立起来的对象，它们的析构函数一定会被调用。

29

这里有一个例子，演示了当构造函数没有正常结束时不会调用相关联的析构函数。这个例子还显示了当在创建对象数组的过程中抛出异常时会发生什么情况：

```
//: C01:Cleanup.cpp
// Exceptions clean up complete objects only.
#include <iostream>
using namespace std;

class Trace {
  static int counter;
  int objid;
```

```
public:
  Trace() {
    objid = counter++;
    cout << "constructing Trace #" << objid << endl;
    if(objid == 3) throw 3;
  }
  ~Trace() {
    cout << "destructing Trace #" << objid << endl;
  }
};

int Trace::counter = 0;

int main() {
  try {
    Trace n1;
    // Throws exception:
    Trace array[5];
    Trace n2;  // Won't get here.
  } catch(int i) {
    cout << "caught " << i << endl;
  }
} ///:~
```

读者可以通过跟踪程序的执行过程来了解类**Trace**的对象踪迹。它用一个静态数据成员**counter**来统计已经创建的对象的个数，而用普通数据成员**objid**来追踪特定对象的编号。

main函数首先创建一个单独的对象**n1**（**objid** 0），然后试图创建一个具有五个**Trace**对象的数组，但是，在第四个对象（#3）被完整创建之前抛出了一个异常。对象**n2**根本就没有被创建。在这里可以看到程序的输出结果为：

30

```
constructing Trace #0
constructing Trace #1
constructing Trace #2
constructing Trace #3
destructing Trace #2
destructing Trace #1
destructing Trace #0
caught 3
```

对象数组的三个元素成功创建了，但是在创建第四个对象数组元素的过程中构造函数抛出了一个异常。在**main( )**函数中，由于第四个构造函数（用于创建**array[2]**对象）没有完成，所以只有**array[1]** 对象和**array[0]**对象的析构函数被调用。最后，对象**n1**销毁。因为对象**n2**根本就没有创建，所以也没有销毁。

1.5.1 资源管理

当在编写的代码中用到异常时，非常重要的一点是，读者应该问一下，“如果异常发生，程序占用的资源都被正确地清理了吗？”大多数情况下不用担心，但是在构造函数里有一个特殊的问题：如果一个对象的构造函数在执行过程中抛出异常，那么这个对象的析构函数就不会被调用。因此，编写构造函数时，程序员必须特别的仔细。

困难的事情是在构造函数中分配资源。如果在构造函数中发生异常，析构函数将没有机会释放这些资源。这个问题经常伴随着“悬挂”指针（“naked” pointer）出现。例如：

```
//: C01:Rawp.cpp
// Naked pointers.
#include <iostream>
#include <cstddef>
using namespace std;
```

```
class Cat {
public:
  Cat() { cout << "Cat()" << endl; }
  ~Cat() { cout << "~Cat()" << endl; }
};

class Dog {
public:
  void* operator new(size_t sz) {
    cout << "allocating a Dog" << endl;
    throw 47;
  }
  void operator delete(void* p) {
    cout << "deallocating a Dog" << endl;
    ::operator delete(p);
  }
};

class UseResources {
  Cat* bp;
  Dog* op;
public:
  UseResources(int count = 1) {
    cout << "UseResources()" << endl;
    bp = new Cat[count];
    op = new Dog;
  }
  ~UseResources() {
    cout << "~UseResources()" << endl;
    delete [] bp; // Array delete
    delete op;
  }
};

int main() {
  try {
    UseResources ur(3);
  } catch(int) {
    cout << "inside handler" << endl;
  }
} ///:~
```

31

程序的输出为:

```
UseResources()
Cat()
Cat()
Cat()
allocating a Dog
inside handler
```

32

程序的执行流程进入了**UseResources**的构造函数，**Cat**的构造函数成功地完成了创建对象数组中的三个对象。然而，在**Dog::operator new( )**函数中抛出了一个异常（用于模拟内存不足错误（out-of-memory error））。程序在执行异常处理器之时突然终止，**UseResources**的析构函数没有被调用。这是正确的，因为**UseResources**的构造函数没有完成，但是，这也意味着，在堆上成功创建的**Cat**对象不会被销毁。

1.5.2 使所有事物都成为对象

为了防止资源泄漏，读者必须使用下列两种方式之一来防止“不成熟的”的资源分配方式:

- 在构造函数中捕获异常，用于释放资源。
- 在对象的构造函数中分配资源，并且在对象的析构函数中释放资源。

使用下述方法可以使对象的每一次资源分配都具有原子性，由于资源分配成为局部对象生命周期的一部分，如果某次分配失败了，那么在栈反解的时候，其他已经获得所需资源的对象能够被恰当地清理。这种技术称为资源获得式初始化（Resource Acquisition Is Initialization，RAII），因为它使得对象对资源控制的时间与对象的生命周期相等。为了达到上述目标，利用模板修改前一个例子是一个好方法：

```
//: C01:Wrapped.cpp
// Safe, atomic pointers.
#include <iostream>
#include <cstddef>
using namespace std;

// Simplified. Yours may have other arguments.
template<class T, int sz = 1> class PWrap {
  T* ptr;
public:
  class RangeError {}; // Exception class
  PWrap() {
    ptr = new T[sz];
    cout << "PWrap constructor" << endl;
  }
  ~PWrap() {
    delete[] ptr;
    cout << "PWrap destructor" << endl;
  }
  T& operator[](int i) throw(RangeError) {
    if(i >= 0 && i < sz) return ptr[i];
    throw RangeError();
  }
};

class Cat {
public:
  Cat() { cout << "Cat()" << endl; }
  ~Cat() { cout << "~Cat()" << endl; }
  void g() {}
};

class Dog {
public:
  void* operator new[](size_t) {
    cout << "Allocating a Dog" << endl;
    throw 47;
  }
  void operator delete[](void* p) {
    cout << "Deallocating a Dog" << endl;
    ::operator delete[](p);
  }
};

class UseResources {
  PWrap<Cat, 3> cats;
  PWrap<Dog> dog;
public:
  UseResources() { cout << "UseResources()" << endl; }
  ~UseResources() { cout << "~UseResources()" << endl; }
  void f() { cats[1].g(); }
```

33

```
};

int main() {
  try {
    UseResources ur;
  } catch(int) {
    cout << "inside handler" << endl;
  } catch(...) {
    cout << "inside catch(...)" << endl;
  }
} ///:~
```

34

这种使用模板来封装指针的方法与第一种方法的区别在于，这种方法使得每个指针都被嵌入到对象中。在调用**UseResources**类的构造函数之前这些对象的构造函数首先被调用，并且如果它们之中的任何一个构造函数在抛出异常之前完成，那么这些对象的析构函数也会在栈反解的时候被调用。

PWrap模板是迄今为止读者见到的最典型的使用异常的例子：在**operator[ ]**中使用了一个称做**RangeError**的嵌套类（nested class），如果参数越界，则创建一个**RangeError**类型的异常对象。因为**operator[ ]**的返回值类型是一个引用，所以它不能返回0。（程序中不能有空引用。）这是一个真正的异常情况——在当前语境中，程序不知道该做什么；而且不能返回一个不可能的值。在这个例子中，**RangeError**[⊖]是非常简单的，它假设类的名字能够表达所有必需的信息。如果认为出错对象的索引也很重要的话，可以在**RangeError**类中添加一个数据成员来容纳这个索引值。

这时，程序的输出为：

```
Cat()
Cat()
Cat()
PWrap constructor
allocating a Dog
~Cat()
~Cat()
~Cat()
PWrap destructor
inside handler
```

程序为**Dog**分配存储空间的时候再一次抛出了异常，但是这一次**Cat**数组中的对象被恰当的清理了，没有出现内存泄漏。

35

1.5.3 auto_ptr

由于在一个典型的C++程序中动态分配内存是频繁使用的资源，所以C++标准中提供了一个**RAII**封装类，用于封装指向分配的堆内存（heap memory）的指针，这就使得程序能够自动释放这些内存。**auto_ptr**类模板是在头文件**<memory>**中定义的，它的构造函数接受一个指向类属类型（generic type）的指针（无论在代码中使用什么类）作为参数。**auto_ptr**类模板还重载了指针运算符*和->，以便对持有的**auto_ptr**对象的原始指针进行前面介绍的那些运算。这样，读者就可以像使用原始指针一样使用**auto_ptr**对象。下面的代码演示了如何使用**auto_ptr**：

⊖ 注意，在这种情况下最好使用C++标准库中定义的异常类——**std::out_of_range**。

```
//: C01:Auto_ptr.cpp
// Illustrates the RAII nature of auto_ptr.
#include <memory>
#include <iostream>
#include <cstddef>
using namespace std;

class TraceHeap {
  int i;
public:
  static void* operator new(size_t siz) {
    void* p = ::operator new(siz);
    cout << "Allocating TraceHeap object on the heap "
         << "at address " << p << endl;
    return p;
  }
  static void operator delete(void* p) {
    cout << "Deleting TraceHeap object at address "
         << p << endl;
    ::operator delete(p);
  }
  TraceHeap(int i) : i(i) {}
  int getVal() const { return i; }
};

int main() {
  auto_ptr<TraceHeap> pMyObject(new TraceHeap(5));
  cout << pMyObject->getVal() << endl;  // Prints 5
} ///:~
```

TraceHeap类重载了**new**运算符和**delete**运算符，这样，就可以准确地看到在程序运行过程中发生了什么事情。注意，像其他类模板一样，**main( )**函数里必须在模板参数中指定所要使用的数据类型。但是这里不能使用**TraceHeap***——**auto_ptr**已经知道了要存储指定类型的指针。**main( )**函数的第二行证实了**auto_ptr**的**operator->( )**函数间接使用了基本的原始指针。最重要的一点是，尽管程序没有显式地删除该原始指针，但是在栈反解的时候，**pMyObject**对象的析构函数会删除该原始指针，下面程序的输出证实了这一点：

36

```
Allocating TraceHeap object on the heap at address 8930040
5
Deleting TraceHeap object at address 8930040
```

auto_ptr类模板可以很容易地用于指针数据成员。由于通过值引用的类对象总会被析构，所以当对象被析构时，这个对象的**auto_ptr**成员总是能释放它所封装的原始指针。[⊖]

1.5.4 函数级的try块

由于构造函数能够抛出异常，读者可能希望处理在对象的成员或其基类子对象被初始化的时候抛出的异常。为了做到这一点，可以把这些子对象的初始化过程放到函数级**try**块中。与通常的语法不同，作为构造函数初始化部分的**try**块是构造函数的函数体，而相关的**catch**块紧跟着构造函数的函数体，就像下面这个例子中所写的一样：

```
//: C01:InitExcept.cpp {-bor}
// Handles exceptions from subobjects.
#include <iostream>
using namespace std;
```

⊖ 有关**auto_ptr**的详细信息，请参考Herb Sutter在1999年10月发表的文章“Using auto_ptr Effectively”——《C/C++ Users Journal》，第63~67页。

```
class Base {
  int i;
public:
  class BaseExcept {};
  Base(int i) : i(i) { throw BaseExcept(); }
};
class Derived : public Base {
public:
  class DerivedExcept {
    const char* msg;
  public:
    DerivedExcept(const char* msg) : msg(msg) {}
    const char* what() const { return msg; }
  };
  Derived(int j) try : Base(j) {
    // Constructor body
    cout << "This won't print" << endl;
  } catch(BaseExcept&) {
    throw DerivedExcept("Base subobject threw");;
  }
};

int main() {
  try {
    Derived d(3);
  } catch(Derived::DerivedExcept& d) {
    cout << d.what() << endl;  // "Base subobject threw"
  }
} ///:~
```

37

注意，在**Derived**类的构造函数中，初始化列表处在关键字**try**和构造函数的函数体之间。如果在构造函数中发生异常，**Derived**类所包含的对象也就没有构造完成，因此程序返回到创建该对象代码的地方（构造函数的调用者）是没有意义的。由于这个原因，惟一合理的做法就是在函数级的**catch**子句中抛出异常。

尽管不是非常有用，C++还是允许在所有函数中使用函数级**try**块，下面的例子说明了这种用法：

```
//: C01:FunctionTryBlock.cpp {-bor}
// Function-level try blocks.
// {RunByHand} (Don't run automatically by the makefile)
#include <iostream>
using namespace std;

int main() try {
  throw "main";
} catch(const char* msg) {
  cout << msg << endl;
  return 1;
} ///:~
```

38

在这种情况下，**catch**块中的代码可以像函数体中的代码一样正常返回。这种形式的函数级**try**块与在函数中添加**try-catch**来环绕所有代码没有什么区别。

1.6 标准异常

读者也可以使用标准C++库中定义的异常。一般来说，使用标准异常类比用户自己定义异常类要方便快捷得多。如果标准类不能满足要求，也可以把它们作为基类来派生出自己的异常类。

所有的标准异常类归根结底都是从**exception**类派生的，**exception**类的定义在头文件**<exception>**中。**exception**类的两个主要派生类为**logic_error**和**runtime_error**，这两个类的定义在头文件**<stdexcept>**中（这个头文件包含 **<exception>**）。**logic_error**类用于描述程序中出现的逻辑错误，例如传递无效的参数。运行时错误（runtime error）是指那些无法预料的事件所造成的错误，例如硬件故障或内存耗尽。**logic_error**和**runtime_error**都提供了一个参数类型为**std::string**的构造函数，这样就可以将消息保存在这两种类型的异常对象中，通过**exception::what( )**函数，读者可以从对象中得到它所保存的消息，如下面程序所示：

```
//: C01:StdExcept.cpp
// Derives an exception class from std::runtime_error.
#include <stdexcept>
#include <iostream>
using namespace std;

class MyError : public runtime_error {
public:
  MyError(const string& msg = "") : runtime_error(msg) {}
};

int main() {
  try {
    throw MyError("my message");
  } catch(MyError& x) {
    cout << x.what() << endl;
  }
} ///:~
```

39

尽管**runtime_error**的构造函数把消息保存在它的**std::exception**子对象中，但是**std::exception**并没有提供一个参数类型为**std::string**的构造函数。用户最好从**runtime_error** 类或**logic_error**类（或这两个类中某个类的派生类）来派生自己的异常类，而不要直接从**std::exception**类派生。

下面的几个表格描述了标准异常类：

exception	这个类是由C++标准库为所有抛出异常的类提供的基类。读者可以调用**what( )**函数并取得exception对象初始化时被设置的可选字符串
logic_error	从**exception**类派生。报告程序逻辑错误，通过检查代码，能够发现这类错误
runtime_error	从**exception**类派生。报告运行时错误，只有在程序运行时，这类错误才可能被检测到

输入输出流异常类**ios::failure**也是从**exception**派生的，但是它没有子类。

读者可以直接使用下面两个表中所列的异常类，或者把它们作为基类来派生自己的更加具体的异常类。

从logic_error派生的异常类	
domain_error	报告违反了前置条件
invalid_argument	表明抛出这个异常的函数接收到了一个无效的参数
length_error	表明程序试图产生一个长度大于等于**npos**的对象（语境长度的最大可能值的类型通常为**std::size_t**）
out_of_range	报告一个参数越界错误

40

bad_cast	抛出这个异常的原因是在运行时类型识别（runtime type identification）中发现程序执行了一个的无效的动态类型转换（**dynamic_cast**）表达式（见第8章）
bad_typeid	当表达式**typeid(*p)**中的参数**p**是一个空指针时抛出这个异常。（这也是运行时类型识别的特性，见第8章）

从runtime_error派生的异常类	
range_error	报告违反了后置条件
overflow_error	报告一个算术溢出错误
bad_alloc	报告一个失败的存储分配

1.7 异常规格说明

有时不要求程序提供资料告诉函数的使用者在函数使用时会抛出什么异常。但是，如果这样做，函数的使用者就无法确定如何编码来捕获所有可能的异常，所以这种做法通常被认为是不友好的。如果函数的使用者可以得到源代码，他们就可以通过查找**throw**语句来找到函数所抛出的异常，但是，以库的形式提供的函数通常是不包含源代码的。好的文档能够弥补这一缺陷，但是有多少软件项目能够提供编写良好的文档呢？C++提供一种语法来告诉使用者函数所抛出的异常，这样他们就能正确处理这些异常了。这就是可选的*异常规格说明*（exception specification），它是函数声明的修饰符，写在参数列表的后面。

41

异常规格说明再次使用了关键字**throw**，函数可能抛出的所有可能异常的类型应该被写在**throw**之后的括号中。这里的函数声明如下所示：

```
void f() throw(toobig, toosmall, divzero);
```

在涉及异常的情况下，传统的函数声明：

```
void f();
```

意味着函数可能抛出任何类型的异常。下面的函数声明

```
void f() throw();
```

意味着函数不会抛出任何异常（最好确认一下，这个函数所调用的所有函数也不会抛出异常！）。

从好的编码策略、好的文档和便于函数调用这几个方面来说，当读者编写可能抛出异常的函数时，最好考虑使用异常规格说明。（在这一章的后面，将会讨论这一方针的变化。）

1. **unexpected( )**函数

如果函数所抛出的异常没有列在异常规格说明的异常集中，那将会出现什么情况呢？在这种情况下，一个特殊的函数**unexpected( )**将会被调用。默认的**unexpected( )**函数会调用本章前面所讲到的**terminate( )**函数。

2. **set_unexpected( )**函数

像**terminate( )**函数一样，**unexpected( )**可以提供一种机制设置自己的函数来响应意外的异常（unexpected exception）。读者可以调用函数**set_unexpected( )**来完成这件事，类似于**set_terminate( )**，**set_unexpected( )** 函数使用一个函数指针作为参数，这个指针所指向的函数没有参数，而且其返回值类型为**void**。因为**set_ unexpected ( )**函数返回了**unexpected ( )**函数指针先前的值，所以可以保存这个值，并且在以后恢复它。为了要使用**set_unexpected( )**函数，编程人员必须在代码中包含头文件**<exception>**。下面这个例

42

子用于显示迄今为止这一部分所讨论内容的简单应用:

```
//: C01:Unexpected.cpp
// Exception specifications & unexpected(),
//{-msc} (Doesn't terminate properly)
#include <exception>
#include <iostream>
using namespace std;

class Up {};
class Fit {};
void g();

void f(int i) throw(Up, Fit) {
  switch(i) {
    case 1: throw Up();
    case 2: throw Fit();
  }
  g();
}

// void g() {}          // Version 1
void g() { throw 47; } // Version 2

void my_unexpected() {
  cout << "unexpected exception thrown" << endl;
  exit(0);
}

int main() {
  set_unexpected(my_unexpected); // (Ignores return value)
  for(int i = 1; i <=3; i++)
    try {
      f(i);
    } catch(Up) {
      cout << "Up caught" << endl;
    } catch(Fit) {
      cout << "Fit caught" << endl;
    }
} ///:~
```

创建**Up**类和**Fit**类作为异常类。虽然异常类通常都很小，但是可以用它们来保存附加信息提供给异常处理器作为参考。 43

函数**f( )**在其异常规格说明中声明仅会抛出**Up**和**Fit**类型的异常，但是从函数的定义来看却不是这样的。函数**g( )**的第1个版本（Version 1）被函数**f( )**调用时不会抛出任何异常。但是如果有人修改了函数**g( )**，使它抛出一个不同类型的异常（就像这个例子中的函数**g( )**的第2个版本（Version 2）抛出一个**int**型异常），那么函数**f( )**的异常规格说明就违反了规则。

按照自定义**unexpected( )**函数的格式要求，**my_unexpected( )**函数没有参数和返回值。这个函数只是显示一条消息，表明它被调用了，然后退出程序（在这里使用**exit(0)**），这样编写本书时所使用的**make**程序就不会失败了）。新的**unexpected( )**函数中不能有**return**语句。

在**main( )**函数中，**try**块位于**for**循环的内部，因此，所有的可能情况都被执行了。使用这种方式，程序可以实现类似异常恢复的功能。把**try**块嵌套在**for**、**while**、**do**或**if**块中，并且触发异常来试图解决问题；然后重新测试**try**块中的代码。

仅**Up**和**Fit**异常能够被捕获，因为函数**f( )**的编写者称只有这两种异常会被触发。函数**g( )**

的第2个版本使得**my_unexpected( )**被调用，因为**f( )**抛出了一个**int**型的异常。

在调用**set_unexpected( )**的时候，函数的返回值被忽略了，如果希望在某个时刻恢复先前的**unexpected( )**，读者可以参考本章前面所讲的**set_terminate( )**例子，将**set_unexpected( )**的返回值保存在一个指向函数的指针中。

典型的**unexpected**处理器会将错误记入日志，然后调用**exit( )**终止程序。它也可以抛出另外一个异常（或重新抛出相同的异常）或调用**abort( )**。如果它抛出的异常类型不再违反触发**unexpected**的函数的异常规格说明，那么程序将恢复到这个函数被调用的位置重新开始异常匹配。（这是**unexpected( )**函数特有的行为。）

如果**unexpected**处理器所抛出的异常还是不符合函数的异常规格说明，下列两种情况之一将会发生：

44 1) 如果函数的异常规格说明中包括**std::bad_exception**（在**<exception>**中定义），**unexpected**处理器所抛出的异常会被替换成**std::bad_exception**对象，然后，程序恢复到这个函数被调用的位置重新开始异常匹配。

2) 如果函数的异常规格说明中不包括**std::bad_exception**，程序会调用**terminate( )**函数。

下面的程序演示了这种行为：

```
//: C01:BadException.cpp {-bor}
#include <exception>    // For std::bad_exception
#include <iostream>
#include <cstdio>
using namespace std;

// Exception classes:
class A {};
class B {};

// terminate() handler
void my_thandler() {
  cout << "terminate called" << endl;
  exit(0);
}

// unexpected() handlers
void my_uhandler1() { throw A(); }
void my_uhandler2() { throw; }

// If we embed this throw statement in f or g,
// the compiler detects the violation and reports
// an error, so we put it in its own function.
void t() { throw B(); }

void f() throw(A) { t(); }
void g() throw(A, bad_exception) { t(); }

int main() {
  set_terminate(my_thandler);
  set_unexpected(my_uhandler1);
  try {
    f();
  } catch(A&) {
    cout << "caught an A from f" << endl;
  }
  set_unexpected(my_uhandler2);
  try {
```

45

```
    g();
  } catch(bad_exception&) {
    cout << "caught a bad_exception from g" << endl;
  }
  try {
    f();
  } catch(...) {
    cout << "This will never print" << endl;
  }
} ///:~
```

处理器**my_uhandler1( )**抛出一个可以接受的异常**(A)**，所以程序的执行流程成功地恢复到了第1个**catch**块中。处理器**my_uhandler2( )**抛出的异常（**B**）不合法，但是**g**的异常规格说明中包括**bad_exception**，所以类型为**B**的异常被替换成类型为**bad_exception**对象，所以第2个**catch**也成功了。由于**f**的异常规格说明中不包括**bad_exception**，所以程序终止处理器（terminate handler）**my_thandler( )**被调用了。程序的输出为:

```
caught an A from f
caught a bad_exception from g
terminate called
```

1.7.1 更好的异常规格说明

读者可能会觉得现行的异常规格说明规范不太好，而

```
void f();
```

应该表示函数不会抛出异常。如果程序员想抛出任意类型的异常，他应该写成如下形式:

```
void f() throw(...); // Not in C++
```

这确实是一种改进，因为函数的声明会变得更加明确。遗憾的是，通过阅读代码，读者不一定能够准确地知道函数是否会抛出异常——例如，内存分配失败会触发异常。更坏的情况是: 在异常处理机制出现之前编写的函数会发觉，由于它们所调用的函数抛出了异常，所以它们也不经意地抛出了异常（它们可能会链接到新的可抛出异常的版本）。因此，这种不明确的描述被保存了下来:

46

```
void f();
```

意味着“我可能会抛出异常，也可能不会抛出异常。”为了避免干扰代码的演化，这种不确定性是必须的。如果读者想明确表示函数**f**不会抛出任何异常，可以使用空的异常类型列表，如下所示:

```
void f() throw();
```

1.7.2 异常规格说明和继承

类中的每个公有函数本质上来说都是类与用户的一种约定。用户传给函数特定的参数，它执行某种处理并且/或者返回结果。同样的约定必须在派生类中保持有效; 否则，派生类和基类之间“是一个（is-a)”的关系就会被违背。由于异常规格说明在逻辑上也是函数声明的一部分，所以在继承层次结构中也必须保持一致。例如，如果基类的一个成员函数声明它只抛出一种类型的异常**A**，那么派生类中覆盖这个函数的函数不能在异常规格说明列表中添加其他异常。因为如果添加其他异常，就会造成依赖于基类接口的任何程序崩溃。读者可以在派生类函数的异常规格说明中指定较少的异常或指定为不抛出异常，因为这样不需要用户修改任何代码。读者也可以在派生类函数的异常规格说明中指定任何“是一个（is-a)”**A**来代替**A**。举例如下:

```
//: C01:Covariance.cpp {-xo}
// Should cause compile error. {-mwcc}{-msc}
#include <iostream>
using namespace std;

class Base {
public:
  class BaseException {};
  class DerivedException : public BaseException {};
  virtual void f() throw(DerivedException) {
    throw DerivedException();
  }
  virtual void g() throw(BaseException) {
    throw BaseException();
  }
};

class Derived : public Base {
public:
  void f() throw(BaseException) {
    throw BaseException();
  }
  virtual void g() throw(DerivedException) {
    throw DerivedException();
  }
}; ///:~
```

47

由于**Derived::f( )**违反了**Base::f( )**的异常规格说明，所以编译器将认为**Derived::f( )**是错误的（或者至少给出一个警告）。**Derived::g( )**的异常规格说明可以被编译器接受，因为**DerivedException**“是一个（is-a）”**BaseException**（没有其他可能性）。读者可以认为**Base/Derived**和**BaseException/DerivedException**是并行的类层次结构；在派生类中，可以用**DerivedException**的返回值来代替指向异常规格说明中的**BaseException**对象的引用。这种行为被称为协变（covariance）（因为两套类同时在各自的继承层次结构上向下变化）。（回顾在第1卷中曾说过：参数类型不能协变——在覆盖虚函数的时候不允许修改函数的签名。）

1.7.3 什么时候不使用异常规格说明

如果阅读标准C++库中定义的函数声明，读者会发现没有一个函数使用了异常规格说明。尽管这看起来很奇怪，但是这种看似奇怪的做法是有原因的：标准C++库主要是由模板组成的，无法知道普通的类或函数会做些什么。例如，读者正在开发一个普通的栈模板，并且在**pop**函数中使用异常规格说明，如下所示：

```
T pop() throw(logic_error);
```

48

由于读者所能预见到的错误只有栈下溢，读者可能认为在异常规格说明中指定一个**logic_error**或某种恰当的异常类型是安全的。但是类型**T**的拷贝构造函数可能会抛出异常。那么，**unexpected( )**会被调用，程序终止。应用系统无法提供可支持异常处理的保证。当无法知道会触发什么异常时，不要使用异常规格说明。这就是为什么模板类，也就是标准C++库的主要组成部分，不使用异常规格说明的原因——它们将其所知道的异常写在文档中，把剩下的事情交给用户来做。异常规格说明主要是为非模板类准备的。

1.8 异常安全

在第7章中，将要深入讨论标准C++库中的容器，包括栈容器。读者将会注意到，**pop( )**

成员函数的声明如下:

```
void pop();
```

读者可能感觉很奇怪，**pop( )**的返回值类型是void。它仅仅删除了栈顶元素。为了获得栈顶元素的值，程序员得在调用**pop( )**之前调用**top( )**。这种做法有一个很重要的原因就是**stack**必须保证异常安全，在标准C++库设计过程中异常安全是一个至关重要的考虑因素。当面对异常的时候有不同级别的异常安全，但是，顾名思义，异常安全中最重要的一点是正确的语义。

假设读者正在使用动态数组实现一个栈（使用**data**作为数组变量名，使用**count**作为整数计数器的变量名），并且试图编写一个带有返回值的**pop( )**函数。这个**pop( )**函数的代码如下:

```
template<class T> T stack<T>::pop() {
  if(count == 0)
    throw logic_error("stack underflow");
  else
    return data[--count];
}
```

如果为了得到返回值而调用拷贝构造函数，函数却在最后一行抛出一个异常，当该值返回时会发生什么情况呢？因为发生了异常，函数并没有将应该退栈的元素返回，但是**count**已经减1了，所以函数希望得到的栈顶元素丢失了！问题产生的原因是这个函数试图一次做两件事情:（1）返回值，并且（2）改变栈的状态。最好将这两个独立的动作放到两个独立的函数中，这就是标准的**stack**类的做法。（换句话说，遵守内聚设计原则——每个函数只做一件事情。）异常安全代码能够使对象保持状态的一致性而且能够避免资源泄漏。 49

读者需要仔细编写自定义的赋值操作符。在第1卷的第12章，读者已经看到**operator=**应该遵守下面的模式:

1) 确保程序不是给自己赋值。如果是的话，跳到步骤6。（这是一种严格的最优化。）
2) 给指针数据成员分配所需的新内存。
3) 从原有的内存区向新分配的内存区拷贝数据。
4) 释放原有的内存。
5) 更新对象的状态，也就是把指向分配新堆内存地址的指针赋值给指针数据成员。
6) 返回 ***this**。

重要的是，直到所有的新增部件都被安全地分配到内存并初始化之前不要修改对象的状态。一个好的技巧是将步骤2和步骤3放到单独的函数中，这个函数常被叫做**clone( )**。下面的例子演示了如何在一个拥有两个指针成员（**theString**和**theInts**）的类中使用这一技术:

```
//: C01:SafeAssign.cpp
// An Exception-safe operator=.
#include <iostream>
#include <new>       // For std::bad_alloc
#include <cstring>
#include <cstddef>
using namespace std;

// A class that has two pointer members using the heap
class HasPointers {
  // A Handle class to hold the data
  struct MyData {
    const char* theString;
    const int* theInts;
```

50

```
    size_t numInts;
    MyData(const char* pString, const int* pInts,
      size_t nInts)
    : theString(pString), theInts(pInts), numInts(nInts) {}
  } *theData;  // The handle
  // Clone and cleanup functions:
  static MyData* clone(const char* otherString,
      const int* otherInts, size_t nInts) {
    char* newChars = new char[strlen(otherString)+1];
    int* newInts;
    try {
      newInts = new int[nInts];
    } catch(bad_alloc&) {
      delete [] newChars;
      throw;
    }
    try {
      // This example uses built-in types, so it won't
      // throw, but for class types it could throw, so we
      // use a try block for illustration. (This is the
      // point of the example!)
      strcpy(newChars, otherString);
      for(size_t i = 0; i < nInts; ++i)
        newInts[i] = otherInts[i];
    } catch(...) {
      delete [] newInts;
      delete [] newChars;
      throw;
    }
    return new MyData(newChars, newInts, nInts);
  }
  static MyData* clone(const MyData* otherData) {
    return clone(otherData->theString, otherData->theInts,
                 otherData->numInts);
  }
  static void cleanup(const MyData* theData) {
    delete [] theData->theString;
    delete [] theData->theInts;
    delete theData;
  }
public:
  HasPointers(const char* someString, const int* someInts,
              size_t numInts) {
    theData = clone(someString, someInts, numInts);
  }
  HasPointers(const HasPointers& source) {
    theData = clone(source.theData);
  }
  HasPointers& operator=(const HasPointers& rhs) {
    if(this != &rhs) {
      MyData* newData = clone(rhs.theData->theString,
        rhs.theData->theInts, rhs.theData->numInts);
      cleanup(theData);
      theData = newData;
    }
    return *this;
  }
  ~HasPointers() { cleanup(theData); }
  friend ostream&
  operator<<(ostream& os, const HasPointers& obj) {
    os << obj.theData->theString << ": ";
    for(size_t i = 0; i < obj.theData->numInts; ++i)
      os << obj.theData->theInts[i] << ' ';
```

```
    return os;
  }
};

int main() {
  int someNums[] = { 1, 2, 3, 4 };
  size_t someCount = sizeof someNums / sizeof someNums[0];
  int someMoreNums[] = { 5, 6, 7 };
  size_t someMoreCount =
  sizeof someMoreNums / sizeof someMoreNums[0];
  HasPointers h1("Hello", someNums, someCount);
  HasPointers h2("Goodbye", someMoreNums, someMoreCount);
  cout << h1 << endl;  // Hello: 1 2 3 4
  h1 = h2;
  cout << h1 << endl;  // Goodbye: 5 6 7
} ///:~
```

为了方便起见，类**HasPointers**使用**MyData**类作为两个指针的句柄。一旦需要分配更多的内存，不论是在构造函数中还是在赋值操作中，程序最终都会调用第一个**clone**函数来完成任务。如果第一条使用**new**运算符分配内存的语句失败，则会自动抛出一个**bad_alloc**异常。如果第二条分配内存的语句（为**theInts**分配内存）失败，系统必须清理为**theString**分配的内存——第一个**try**块捕获了**bad_alloc**异常。第二个**try**块不是至关重要的，因为只是拷贝**int**和指针（不会触发异常），但是当拷贝对象的时候，它们的赋值操作符可能会触发异常，所以应该清理它们。请注意，在这两个异常处理器中，*重新抛出*了异常。这是因为在这里系统只是进行资源管理工作，函数的使用者仍然需要知道发生了什么错误，所以系统的异常处理机制让异常沿着函数调用动态链向上传播。不会默默地吞没异常的软件库被称做*异常中立的*(exception neutral)。对于读者来说，始终需要努力写出异常安全且异常中立的软件库。[⊖] 52

如果仔细检查上面的代码，就会发现没有一个**delete**操作会抛出异常。这段代码正是基于这一事实。回忆一下，当程序中用**delete**删除一个对象的时候，这个对象的析构函数会被调用。结果是：事实上，只能假设析构函数不抛出异常，否则无法设计异常安全的代码。不要让析构函数抛出异常。（在本章结束之前将对此进行多次提醒。）[⊜]

1.9 在编程中使用异常

对大多数程序员，尤其是C程序员来说，他们目前使用的程序设计语言不支持异常，所以需要做一些调整。下面是一些在程序设计中使用异常的指导原则。

1.9.1 什么时候避免异常

异常并不能解决所有问题；过度使用会造成麻烦。本文下面的部分指出了在哪种情况下不应该使用异常。有关何时应该使用异常的最好建议是：只有当函数不符合它的规格说明时才抛出异常。 53

1. 不要在异步事件中使用异常

标准C的**signal( )**系统及类似系统负责处理异步事件：这些事件是发生在程序流程之外的，

⊖ 如果读者对更深入地分析异常安全问题感兴趣，权威的参考书是Herb Sutter的《Exceptional C++》，Addison-Wesley, 2000。

⊜ 在栈反解过程中，库函数**uncaught_exception( )**返回**true**。因此从技术上来讲，用户可以使用**uncaught_exception( )**来测试当前状态，如果反回**false**，那么就可以让析构函数抛出异常。我们从未见过使用这种技术实现优秀设计的先例，所以只是在脚注中提及。

而且这些事件的发生是程序无法预料的。由于异常和它的处理器必须处在相同的函数调用栈上，所以无法使用C++中的异常机制来处理异步事件。也就是说，异常依赖于程序运行栈上的动态函数调用链（他们有“动态作用域（dynamic scope）”），然而异步事件必须由完全独立的代码来处理，这些代码不是正常程序流程的一部分（典型的例子是：中断服务例程和事件循环）。不要在中断处理程序中抛出异常。

这并不是说异步事件不能与异常发生联系。但是，中断处理程序应该尽快完成工作并返回。处理这种情况的典型方式是，中断处理程序设置一个标记，程序的主干代码同步地检查这个标记。

2. 不要在处理简单错误的时候使用异常

如果能得到足够的信息来处理错误，那么就不要使用异常。程序员应该在当前语境中处理这个错误，而不是将一个异常抛出到更大的（上一层）语境中。

此外，C++在遇到机器层事件如除零错误[⊖]时不会抛出异常。读者可以认为其他一些机制，如操作系统或硬件会处理这种事件。这样，C++的异常机制可以相当有效，它们被隔离起来只用于处理程序级的异常状况。

3. 不要将异常用于程序的流程控制

异常看起来有点像函数返回机制的代替品，也有点像**switch**语句，因此，读者可能觉得用异常来代替这些普通的语言机制很有吸引力。这是一种错误的想法，部分原因是异常处理系统的效率比普通的程序流控制差很多。异常仅仅是一个非常事件，使用异常要付出一定的代价。同样，如果把异常用于处理错误之外的其他地方，也会令类或函数的使用者带来混乱。

54

4. 不要强迫自己使用异常

某些程序是相当简单的（例如一些小型的实用程序）。程序中可能只需要接收输入数据，进行某些处理。在这些程序中，可能在分配内存时失败，打开文件时失败等等。遇到这类情况，显示一个消息然后退出程序就可以了，最好把清理工作交给操作系统来处理，而不必费劲地捕获所有异常并释放资源。简单地说，如果读者不需要异常，就不要强迫自己使用它们。

5. 新异常，老代码

另一个问题出现在需要对现有的没有使用异常的程序进行修改的情况下。在程序设计中，可能引入了一个使用异常机制的库，并且想知道是否应该修改程序中所有的代码。假设在程序中已经拥有了一个令人满意的错误处理模式，最直接的方法是把使用新库的覆盖范围最大的代码段（可能是**main( )**函数中的所有语句）放到**try**块中，追加一个**catch(...)**，然后是基本的错误信息。可以进一步精练它们，根据需要的程度，添加更明确的异常处理器来改进这种做法，但是无论如何，新添加的代码应该尽可能的少。更好的方法是把产生异常的代码隔离在**try**块中，并且编写异常处理器把异常转换成与现有错误处理模式兼容的形式。

当一个编程人员正在编写一个供其他人使用的库时，特别是当无法知道他们如何响应致命性错误条件的时候，慎重地考虑异常非常重要（回忆一下之前对异常安全的讨论，为什么标准C++库中没有使用异常规格说明）。

1.9.2 异常的典型应用

在下列情况下请使用异常：

- 修正错误并且重新调试产生异常的函数。
- 在重新调试中的函数外面补偿一些行为以便使程序得以继续执行。

⊖ 某些编译器在这种情况下会抛出异常，但是它们通常提供编译器选项来禁止这种（不常见的）行为。

- 在当前语境中做尽可能多的事情，并把同样类型的异常重新抛出到更高层的语境中。
- 在当前语境中做尽可能多的事情，并将一个不同类型的异常抛出到更高层的语境中。 55
- 终止程序。
- 将使用普通错误处理模式的函数(尤其是C库函数)封装起来，以便用异常来代替原有的错误处理模式。
- 简化。如果建立的错误处理模式使事情变得更复杂并且难以使用，那么异常可以使错误处理更加简单有效得多。
- 使建立的库和程序更安全。使用异常既是一种短期投资（为了调试方便）也是一种长期投资（为了应用系统的健壮性）。

1. 什么时候使用异常规格说明

异常规格说明就像函数原型：它提醒使用者来编写异常处理代码以及处理什么异常。它提醒编译器这个函数可能抛出异常，让编译器能够在运行时检测违反该异常规格说明的情况。

一个程序设计人员不能总是通过检查代码来预测某个特定的函数会抛出什么异常。有时候函数会产生无法预料的异常，有时候一个不抛出异常的旧函数会被一个抛出异常的新函数替换掉，并且迫使程序调用**unexpected( )**。任何时候如果要使用异常规格说明，或调用使用异常规格说明的函数，最好编写自己的**unexpected( )**函数，在这个**unexpected( )**函数中将消息记入日志，然后抛出异常或终止程序。

如前所述，应该避免在模板类中使用异常规格说明，因为无法预料模板参数类（template parameter classes）所抛出的异常的类型。

2. 从标准异常开始

在编写自己的异常类之前检查标准C++库中所定义的异常类。如果标准异常类符合系统设计的要求，则可能会使用户更容易理解和处理。

如果标准库中没有定义用户所需的异常类，应尽量从现有的标准异常类中继承出一个。如果用户能够使用**exception( )**类中定义的接口函数**what( )**，那么用户的异常类将显得非常友好。

3. 嵌套用户自己的异常 56

如果为用户自己的特定类创建异常类，最好在这个特定类中或包含这个特定类的名字空间中嵌套异常类，这就为读者提供了一个明确的信息——这个异常类仅在用户自己的特定类中使用。另外，这也避免了污染全局名字空间。

即使用户自己的异常类是从C++标准异常类中派生的，用户也可以嵌套它们。

4. 使用异常层次结构

异常层次结构为用户的类或库可能遇到的不同类型的重要错误提供了一个有价值的分类方法。这种方法给用户提供了有价值的信息，帮助他们组织代码，使他们能够有选择地忽略所有异常的附加的类型而仅仅捕获基类类型。另外，后来添加到异常类层次结构中的从相同基类继承的任何异常不会迫使用户重写现有的代码——针对基类的异常处理器将会捕获到这个新异常。

标准C++异常类是异常层次结构的一个好的范例。如果可以的话，应基于这些异常类来创建用户自己的异常类。

5. 多重继承（MI）

读者在研读第9章的时候就会发现，惟一必须用到多重继承的情况是：当需要将一个对象指针向上类型转换成两个不同的基类类型时——也就是说，读者同时需要这两个基类的多态行为。异常层次结构在这种情况下也是有用的，因为多重继承异常类的任何一个基类的异常处理器都能够处理这个异常。

6. 通过引用而不是通过值来捕获异常

正如读者在“异常匹配”那节内容所见到的，应该通过引用来捕获异常，这么做有两个原因：

- 当异常对象被传递到异常处理器中的时候，避免进行不必要的对象拷贝。
- 当派生类对象被当作基类对象捕获时，避免对象切割。

57 尽管可以抛出并且捕获指针类型的异常，但是如果这么做的话，将会在代码中引入紧耦合——抛出异常的代码和捕获异常的代码，必须就如何为异常对象分配内存和如何清理异常对象达成一致。由于在堆耗尽的时候也可能会触发异常，所以这也造成了一个问题。如果程序抛出异常对象，异常处理系统负责处理所有与存储有关的问题。

7. 在构造函数中抛出异常

由于构造函数没有返回值，有两种方法来报告在构造对象期间发生的错误。

- 设置一个非局部的标记，并且希望用户检查它。
- 返回一个未完成的创建对象，并且希望用户检查它。

这个问题至关重要，因为C程序员希望所有的对象创建工作总是成功的，这一点在C语言中并不是不合理的，因为C语言中的类型非常简单。但是在C++程序中，不理会构造函数中出现的故障而继续运行，肯定会导致灾难性的后果，所以构造函数是抛出异常最重要的位置之一——现在用户有了一种安全有效的方式来处理构造函数异常。然而，当构造函数抛出异常时，用户必须注意对象内部的指针和它的清理方式。

8. 不要在析构函数内部触发异常

因为析构函数会在抛出其他异常的过程中被调用，所以绝不要在析构函数中抛出异常或在析构函数中执行其他可能触发抛出异常的操作。如果在析构函数中抛出异常，这个新的异常可能会在现存的异常（其他异常）到达**catch**子句之前被抛出，这会导致程序调用**terminate( )**函数。

如果在析构函数中调用的函数可能会抛出异常，应该在这个析构函数中编写一个**try**块，并把这些函数调用放到**try**块中，析构函数必须自己处理所有这些异常。绝对不能有任何一个异常从析构函数中抛出。

9. 避免悬挂指针

请看这一章前面的**Wrapped.cpp**程序。如果需要给指针分配资源，那么悬挂指针通常意
58 味着构造函数的弱点。如果在构造函数中抛出异常，因为指针没有析构函数，那么这些资源将无法释放。请使用**auto_ptr**或其他智能指针（smart pointer）类型[⊖]来处理指向堆内存的指针。

1.10 使用异常造成的开销

当异常被抛出时，将造成相当多的运行时开销（但是，这是有益的开销，因为对象被自动清理了！）。由于这种原因，不要将异常作为正常控制流的一部分使用，无论这种想法看起来多么精巧诱人。异常应该很少发生，所以开销主要是由异常造成的而不是由正常执行的代码造成的。异常处理机制的重要设计目标之一是，当异常没有发生时，它不应该影响系统的运行速度；也就是说，如果不抛出异常，那么代码的运行速度就像没有使用异常处理机制时一样快。这么说是否正确，依赖于用户所使用的特定编译器的实现方式。（参考这一节的后面部分对“零代价模型（zero-cost model）”的描述。）

⊖ 在网址http://www.boost.org/libs/smart_ptr/index.htm可以找到增强的智能指针类型。下一版的标准C++正在考虑包含这些智能指针类型中的一部分。

可以这样认为，一个**throw**表达式就像是一个特殊的系统函数调用，它接收异常对象作为参数并且沿着执行调用链向上回溯。为了完成这项工作，编译器需要在栈上放置额外的信息，来辅助栈反解过程。为了理解这些内容，用户需要了解有关运行栈（runtime stack）的知识。

每当函数被调用的时候，有关这个函数的信息被压到运行栈顶部的*活动记录实例*（activation record instance，ARI）中，也叫*栈结构*（stack frame）。典型的栈结构包含调用函数的指令所在的地址（这样，程序的执行流程可以返回到这个地址），指向这个函数静态父对象的ARI（某个作用域，它在词法上包含被调用函数，这样这个函数就可以访问全局变量了）的指针，和指向调用函数的指针（它的*动态父函数*）。沿着动态父函数链反复追踪所得到的逻辑结果路径就是*动态链*，或称其为*调用链*，读者在这一章的前面见到过它。这就是为什么当异常抛出时执行流程能够回溯，这种机制使得在彼此缺乏了解的情况下开发出来的程序的各个部分，能够在运行时互相传递出错信息。 59

对于异常处理机制系统允许栈反解，每个函数额外的异常相关信息，必须对每一个栈结构来说都是可用的。这些信息描述了哪个析构函数应该被调用（因此，局部对象可以被清理），这些信息显示了当前函数是否有**try**块，而且这些信息列出了与**try**块相关的**catch**子句能够捕获哪些异常。这些额外信息会造成存储空间的消耗，所以支持异常处理机制的程序要比不支持异常处理机制的程序大[⊖]。因为在运行期间生成扩展栈结构的逻辑必须由编译器生成，所以使用异常处理的程序在编译时也较大。

为了演示这一点，在这里使用Borland C++ Builder和Microsoft Visual C++[⊖]：在支持异常处理机制和不支持异常处理机制的模式下分别编译下面的程序：

```
//: C01:HasDestructor.cpp {O}
class HasDestructor {
public:
  ~HasDestructor() {}
};

void g(); // For all we know, g may throw.

void f() {
  HasDestructor h;
  g();
} ///:~
```

如果允许异常处理，编译器必须为**f( )**保存有关析构函数**~HasDestructor( )**在运行时的大量信息到**ARI**（活动记录实例）中（这样即使**g( )**抛出异常，**f( )**也能正确地销毁对象**h**）。 60
下表总结了编译结果文件（.obj）的大小（单位：字节）。

编译器\模式	支持异常处理	不支持异常处理
Borland	616	234
Microsoft	1162	680

不要把两种模式之间文件大小的百分比看得太重。请记住，典型情况下异常（应该）只构成程序的很小一部分，其空间开销是相当小的（通常只占总开销的5%~15%）。

⊖ 这取决于在不使用异常的情况下用户必须插入多少代码来检查返回值。

⊖ Borland在默认情况下允许异常处理；使用－**x**编译器选项来禁止异常处理。Microsoft在默认情况下不允许异常处理；使用－**GX**选项开启异常处理。两种编译器都使用－**c**选项作为只执行编译过程的选项。

额外的管理工作会降低执行速度，但是聪明的编译器会避免这种情况。由于与异常处理代码和局部对象偏移量有关的信息只在编译时刻计算一次，这些信息可以保存在与每个函数相关的单独位置中，而不是保存在每个ARI中。我们基本上已经从每个活动记录实例中消除了异常的空间开销，因此也避免了压栈操作造成的附加时间开销。这种方法称为异常处理的零代价（zero-cost）模型[⊖]，早先提及的优化存储被认为是影子栈（shadow stack）。[⊜]

1.11 小结

错误恢复是程序员在编写每个程序时最关心的内容。当使用C++创建程序的组件供他人使用时，错误恢复是非常重要的。要创建一个健壮的系统，那么它的每一个组件都必须足够健壮。

61 C++中异常处理的目标是简化创建庞大、可靠程序所需的工作，用更少的代码使软件开发者对程序中是否仍然包含未处理的错误拥有更多信心。在不损失或很少损失性能的情况下，在很少干扰现存代码的情况下，现在实现了这一目标。

基本的异常不难掌握；一旦能够掌握，就可以在程序中开始使用它们。异常是能够为用户要开发的项目提供直接和重大利益的特性之一。

1.12 练习

1-1 编写三个函数：一个通过返回错误值来指出错误情况，一个设置**errno**标志，最后一个使用**signal( )**。编写代码调用这些函数并响应产生的错误。编写第4个函数，这个函数抛出异常。调用这个函数并捕获异常。描述这4种方法的区别，为什么说异常处理机制是一种更好的方法。

1-2 创建一个类，这个类含有抛出异常的成员函数。在这个类中嵌套一个类作为异常对象的类型。这个异常类使用一个**const char***作为参数；这个参数代表一个描述字符串。创建一个抛出这种异常的成员函数。（在函数的异常规格说明中描述这种异常。）编写一个**try**块调用这个成员函数，写一个**catch**子句通过显示描述字符串的方式处理这个异常。

1-3 重新编写第1卷第13章中的**Stash**类，为**operator[ ]**抛出**out_of_range**异常。

1-4 编写一个普通的**main( )**函数，捕获所有的异常并报告错误。

1-5 创建一个类，这个类带有自己的**new**运算符。这个运算符为十个对象分配内存，并且在为第11个对象分配内存时抛出“run out of memory”（内存用完）异常。并且添加一个静态成员函数用于回收这个内存。创建**main( )**函数其中包含**try**块和**catch**子句，在**catch**子句中调用回收内存的子例程。把这些代码放在一个**while**循环中，用于演示从异常恢复并继续执行的过程。

62 1-6 创建一个抛出异常的析构函数，编写代码来为自己证明这是一个坏主意。（在能够捕获现有异常的异常处理器被调用之前，如果一个新的异常被抛出，会调用**terminate( )**。）

1-7 证明所有异常对象（被抛出的异常对象）都会被正确销毁。

1-8 证明如果我们在堆上创建一个异常对象，并且抛出指向这个对象的指针，那么这个对象不会被清理。

⊖ GNU C++编译器默认使用零代价模型。Metrowerks Code Warrior for C++也有一个选项，能够选择使用零代价模型。

⊜ 感谢Scott Meyers和Josee Lajoie在零代价模型上的洞察力。读者可以在Josee的精彩文章“Exception Handling: Behind the Scenes,” C++ Gems, SIGS, 1996中找到有关异常如何工作的更多信息。

1-9 编写一个带有异常规格说明的函数，这个函数能够抛出4种异常：一个**char**、一个**int**、一个**bool**和一个自己的异常类。在**main( )**函数中捕获每种异常，并且验证这些捕获的异常。从标准异常类派生自己的异常类。使用下述方法编写**main( )**函数：系统从异常中恢复并尝试重新执行抛出异常的函数。

1-10 修改上一个练习，让函数抛出一个违反异常规格说明的**double**类型的异常。在自己的unexpected处理函数中捕获这个违反异常规格说明的错误，显示一个消息然后优雅地退出程序（也就是说不要调用**abort( )**）。

1-11 编写一个**Garage**类，这个类包含一个**Car**，这个**Car**的**Motor**出了故障。在**Garage**类的构造函数中使用函数级**try**块用于捕获**Car**对象初始化时抛出的异常（从**Motor**类抛出的异常）。从**Garage**类构造函数的异常处理器中抛出一个不同的异常，并在**main( )**函数中捕获这个异常。

第2章 防御性编程

编写“完美的软件”对开发者来说可能是一个难以达到的目标，但是应用一些常规的防御性技术，对于提高代码的质量将会大有帮助。

尽管典型的软件产品的复杂性保证了测试人员总有做不完的工作，然而，程序设计人员仍然渴望创造零缺陷的软件。面向对象设计技术为开发大型项目解决了很多困难，但是最终用户还得自己编写循环和函数。这些“细微处编程”(programming in the small)的详细内容成为用户设计的较大组件所需的构建块。如果循环偶尔突然退出，或者函数只是在“大多数”时候能够计算出正确的结果，那么，不管系统的整体结构（方法论）是多么的优秀，最终还是会陷入麻烦之中。本章中读者将会学习到一些经验，不管项目是什么规模的，这些经验都能够帮助程序员创建出健壮的代码。

代码的其中一个内涵是对问题解决方法的描述。在设计循环的时候，程序员应该能够清楚地告诉读者（包括程序员自己）其准确的想法是什么。在程序中某个特定的地方，应该能够大胆地声明某些条件或其他一些控制方法。（如果不能做出这种声明，只能说明实际上还未解决这个问题。）这种声明称为不变量（invariant），因为在代码中它们出现的那一点上它们应该恒为真；否则，要么是设计有缺陷，要么是代码没有正确地反映程序设计人员的设计意图。

考虑这样一个实现Hi-Lo猜谜游戏的程序。某个人在1到100之间任选一个数，另一个人猜这个数。（现在我们让计算机猜这个数。）选数的人告诉猜数的人他猜的数比正确的数大，还是比正确的数小或是正好相等。对猜数的人来说，最好的策略是进行二分查找，选择待查找数字范围的中间点。根据选数的人回答的“大”或“小”，猜数的人能够知道这个数到底在该范围的哪一半中。重复这个过程，每次重复都能够把范围缩小一半。那么怎么样编写这个循环来正确模拟猜数过程呢？像下面这样写是不够的：


```
bool guessed = false;
while(!guessed) {
  ...
}
```

因为恶意的用户可能会欺骗编程者，使他们花整天的时间进行猜测。然而每次猜测时能做什么样的假设呢？换句话说，在每个循环中设计什么样的条件来控制循环的迭代次数呢？

现假定：秘密的数字是在当前有效的未猜过的数的范围之内：[1，100]。假设用**low**和**high**两个变量标记数字范围的端点。每次进入循环，在循环开始的时候，需要确定该秘密的数字在范围**[low，high]**之内，每次迭代结束之后重新计算数的范围，使其在进行下一次循环迭代时仍然含有该秘密数字。

这样做的目标是在编写代码的时候表示出循环的不变量条件，使得程序可以在运行的时候检测到违背条件的情况。遗憾的是，由于计算机不知道秘密的数字，程序员不能在代码中直接表达这个条件，但是至少能够写一段这种效果的注释：

```
while(!guessed) {
  // INVARIANT: the number is in the range [low, high]
  ...
}
```

当用户回答猜测结果太大或太小（即超出秘密数字的可能范围）时，会出现什么情况呢？这样的欺骗会造成新计算出来的子范围不包括秘密数字。因为一个谎言总会导致另一个谎言，最终会使猜数范围缩减到不包含任何数字（由于每次将猜数范围缩减一半，而且秘密数字并不在范围内）。下面的程序可以表示这种情况。

```
//: C02:HiLo.cpp {RunByHand}
// Plays the game of Hi-Lo to illustrate a loop invariant.
#include <cstdlib>
#include <iostream>
#include <string>
using namespace std;

int main() {
  cout << "Think of a number between 1 and 100" << endl
       << "I will make a guess: "
       << "tell me if I'm (H)igh or (L)ow" << endl;
   int low = 1, high = 100;
   bool guessed = false;
   while(!guessed) {
     // Invariant: the number is in the range [low, high]
     if(low > high) {  // Invariant violation
       cout << "You cheated! I quit" << endl;
       return EXIT_FAILURE;
     }
     int guess = (low + high) / 2;
     cout << "My guess is " << guess << ". ";
     cout << "(H)igh, (L)ow, or (E)qual? ";
     string response;
     cin >> response;
     switch(toupper(response[0])) {
       case 'H':
         high = guess - 1;
         break;
       case 'L':
         low = guess + 1;
         break;
       case 'E':
         guessed = true;
         break;
       default:
         cout << "Invalid response" << endl;
         continue;
     }
   }
   cout << "I got it!" << endl;
   return EXIT_SUCCESS;
} ///:~
```

65

条件表达式**if(low > high)**可以发现违反不变量条件的情况，因为如果用户总是说实话，那么在用完这些猜测前总能找到这个秘密数字。

也可以使用标准C的技术通过从**main( )**函数中返回不同的值来向调用者报告程序的状态。用语句**return 0**的可携带值来表示程序执行成功，但是没有可携带的值可作为表示程序执行失败的返回值。因此，可以在这种情况下使用**<cstdlib>**中声明的宏：**EXIT_FAILURE**表示程序执行失败的返回值。为了代码的一致性，无论何时使用**EXIT_FAILURE**，我们也使用**EXIT_SUCCESS**来表示程序执行成功，虽然**EXIT_SUCCESS**总是被定义成0。

66

2.1 断言

在Hi-Lo程序执行依赖于用户输入的情况下，无法防止在程序运行过程中出现违反不变量条件的事件发生。然而，不变量条件通常仅仅依赖于编写的代码，所以这些不变量条件始终持有程序设计是否已经正确实现的证据。在这种情况下，可以明确地使用断言（assertion），断言是一个肯定的语句，（只要能证明在程序的执行过程中断言恒真，就证明了程序的正确性。）用来肯定显示设计意图。

假设现在正在实现一个整数向量（vector）：一个可以按照需求扩展的数组。添加一个元素到向量中的函数必须首先检查在数组的下面的位置是否有空闲的单元；如果没有空闲单元，函数必须请求更多的堆空间，而将现有的元素拷贝到新分配的内存空间中，最后把这个新元素添加到数组中（并且释放旧的数组）。如下所示：

```
void MyVector::push_back(int x) {
  if(nextSlot == capacity)
    grow();
  assert(nextSlot < capacity);
  data[nextSlot++] = x;
}
```

在这个例子中，**data**是一个整型的动态数组，有**capacity**个单元，前**nextSlot**个单元已经被使用了。**grow( )**函数的作用是扩大**data**数组，使新数组的**capacity**值比**nextSlot**大。**MyVector**的行为是否正确依赖于设计决定，如果其他支持代码正确，**MyVector**就不会出错。可以使用定义在头文件**<cassert>**中的**assert( )**宏断言这种情况。

标准C库中的**assert( )**宏是简明扼要的并且也可携带返回信息。如果参数赋值得到的条件为非零值，程序将不受干扰地继续运行；否则，引发断言错误的表达式和其所在的源文件的文件名、这个断言所在行的行号都会被一起送到标准错误输出信道打印出来，然后程序异常终止。这种反应方式太过激烈吗？实际上，当基本设计中的假定已经失败时，让程序继续执行会造成更加剧烈的反应。如果出现了这种情况，就应该修改程序。

如果所有的事情都进行得很好，则应该在配置最终产品之前完整地测试代码，在测试过程中不应该出现触发断言错误的情况。（本章随后会讲更多有关测试的问题。）视应用程序的性质而定，运行时检测所有断言所耗费的机器周期可能会大大降低程序的执行效率，以至严重影响这个系统在该领域的应用。如果是这样的话，定义**NDEBUG**宏并重新编译程序，会自动去掉所有断言代码。

67

现在来看看断言是如何工作的吧，注意，典型的**assert( )**实现如下所示：

```
#ifdef NDEBUG
  #define assert(cond) ((void)0)
#else
  void assertImpl(const char*, const char*, long);
  #define assert(cond) \
    ((cond) ? (void)0 : assertImpl(???))
#endif
```

当定义了**NDEBUG**宏的时候，这段代码蜕化成表达式**(void) 0**，再加上写在每个**assert( )**后面的分号，所以最终留在编译器流中的内容仅仅是一条无意义的语句。如果没有定义**NDEBUG**宏，**assert(cond)**被扩展成条件语句，当**cond**的值为零时，调用与编译器相关的函数（称为**assertImpl( )**），调用这个函数时需要三个参数，这三个参数分别是：断言语句所在文件的文件名、断言语句所在行的行号和一个字符串，这个字符串表示**cond**的文本

形式。(在这个例子中，使用“???”来代替这些参数，上面提到的字符串文本是在这里确定的，断言语句所在文件的文件名和断言语句所在行的行号也是在文件中宏出现的位置确定的。如何得到这些值对于这个问题的讨论来说并不重要。)如果想开启或关闭程序中某些位置的断言，不但必须**#define**或**#undef NDEBUG**，而且必须重新包含**<cassert>**。当预处理器遇见它们时对宏进行赋值，并且无论**NDEBUG**是什么状态都将其应用在包含的位置上。最常用的定义**NDEBUG**的方式是作为编译器选项给整个程序定义，不管是通过可视化开发环境的项目设置还是通过命令行，如下所示：

```
mycc -DNDEBUG myfile.cpp
```

大多数编译器使用**-D**标记来定义宏的名字。(为上面的编译器**mycc**替换要编译的可执行文件的文件名。)这种方法的好处是，可以把断言留在源文件中作为不可多得的珍贵文档使用，而不会在运行时造成性能损失。因为当定义了**NDEBUG**，断言中的代码就会消失，所以确保不在断言中做额外操作是至关重要的。断言中只能包含不会修改程序状态的测试条件。

是否应该在发行版中使用**NDEBUG**仍然有争论。Tony Hoare是最有影响的计算机科学家之一，[⊖]他比喻说，关掉类似断言的运行时检查，就像一个热衷于航海的人，当他在陆地上训练的时候穿着救生衣，然而当他下海的时候却脱掉了救生衣。[⊜]断言在软件产品中失灵所造成的问题远比效率降低要严重得多，因此要做出明智的选择。 68

不是所有情况下都应该使用断言。如第1章所示，用户造成的错误和运行时资源故障应该用抛出异常以信号的方式来通知系统。读者可能希望当粗略描述代码的时候，在大多数错误情况下使用断言，并决心在随后的编码过程中用健壮的异常处理来代替它们。这是一种很诱人想法。由于在随后的修改过程中，可能会漏掉某些断言，所以，像对待其他的诱惑一样，必须要十分小心。记住：断言的意图是验证设计决定，造成它失败的惟一原因应该是程序逻辑有缺陷。理想的结果是在开发阶段就解决掉所有违背断言的情况。如果某个条件不完全在程序的控制之下，那么不要对这个条件使用断言（例如，依赖于用户输入的条件）。特别是不应该使用断言来验证函数的参数；在参数错误的情况下，应该抛出**logic_error**异常。

用断言作为工具来确保程序的正确性是Bertrand Meyer在其所著的《Design by Contract methodology》书中正式提出来的。[⊜]每一个函数都有一个隐含的与客户程序的约定，给定某一个前置条件，保证会出现某一个后置条件。换句话说，前置条件是使用该函数的必要条件，例如提供某一范围内的参数，后置条件是该函数提供的结果，通过返回值或通过副作用给出。

当客户程序给出了一个无效的输入，必须告诉这些客户程序，它们违反了约定。这并不是终止程序的最好时机（尽管这样做是正当的，因为它们违反了约定），在这种情况下，应该抛出异常。这就是为什么标准C++库有从**logic_error**类派生的异常类，例如**out_of_range**异常类，用以抛出异常。[⊜]如果这些函数只被程序设计人员自己调用，例如自己设计的类中的私有函数，因为能够控制整体情况并且希望在发行代码之前进行调试，所以使用**assert( )**宏是适当的。 69

后置条件的失败表明程序中有错误，在任何时间使用断言来测试任何不变量都是适当的，

⊖ 他发明了快速排序算法和其他一些东西。

⊜ 引用自“Programming Language Pragmatics”，Michael L. Scott, Morgan-Kaufmann, 2000。

⊜ 参考他所著的书籍《Object-Oriented Software Construction》，Prentice-Hall, 1994。

⊜ 这在概念上仍然是一种断言，但是由于不想终止程序的运行，使用**assert( )**宏并不恰当。例如，Java 1.4在断言失败的时候抛出异常。

包括在一个函数结束的时候测试后置条件。将这种方法应用于维护对象状态的类成员函数中特别合适。在先前提到的**MyVector**例子中，对于所有的公有成员函数来说合理的不变量应该是：

```
assert(0 <= nextSlot && nextSlot <= capacity);
```

或者，如果**nextSlot**是一个无符号整数，简化的结果是

```
assert(nextSlot <= capacity);
```

这样的不变量称为*类不变量*（class invariant），可以使用断言对它进行适度的强制。对于基类来说，它们的子类扮演了一个分包人的角色，因为它们必须维持基类与其客户之间最初的约定。因此，派生类的前置条件不能超过基类与其客户的约定而再强加额外的要求，并且派生类的后置条件必须至少与基类的后置条件一样多。[⊖]

确认返回给客户的结果是否正确与测试没有什么不同，所以在这种情况下使用后置条件断言（post-condition assertions）就与测试工作重复了。当然，这是一种好的文档，但是不止一个软件开发者被这种用法愚弄了，他们错误地把后置条件断言当成单元测试的一种替代品。

70

2.2 一个简单的单元测试框架

为编写软件而做的所有工作都是为了满足客户需求。[⊜]确定这些需求非常困难，它们每天都可能在变化；软件开发人员可能在每周召开的例行项目会议中发现，自己花费一周时间所做的工作并不是用户真正想要的。

如果得不到一个可供参照的系统，然后在它的基础上提出改进意见，人们无法清晰地说明软件需求。最好应该确定一个小的系统，设计、编写、测试这个小系统。在提出改进意见之后，再重新完成它。以迭代方式开发程序的能力是面向对象方法的最大优势之一，但是这需要有才干的程序员，这些程序员应该能够精心制作扩展力非常强的代码。修改现有程序是困难的。

另外一个修改程序的动力来自程序设计人员自己。技艺超群的程序员频繁地改进代码的设计。其实，那些软件行业的旗舰公司推出的被改得乱七八糟、错综复杂的产品，有哪件不被产品维护程序员一再诅咒为费解而又难以修改的拼凑物呢？由于经营成本方面的考虑，迫使编程人员损害系统的功能性，放弃了代码的可扩展性，而这正是保证代码持久性所需要的。“如果程序没有坏掉，就不要修改它”，最后的方法是“没法修改它——重写算了。”这种状况必须改变。

幸运的是，现在软件行业越来越倾向于代码重构了，重构是通过改造系统内部的代码从而改进程序的设计，并且不改变程序的行为。[⊝]这种改善包括从某个函数中摘录出一个新函数，或者相反，组合几个成员函数为一个成员函数；用某个对象替换一个成员函数；参数化一个成员函数或类；有条件地替换多态性。代码重构有助于代码的进化。

71

不管修改程序的动力来自于用户还是来自于程序员，今天的修改可能破坏昨天的工作。需要有一种方式来构造代码，随着时间的流逝，这些代码应该能够经得起变化和改进所带来的负面效应。

⊖ 有一个比较好的短语能帮忙记住这种现象：“不要只索取不付出（Require no more; promise no less）”，这个短语首先在《C++ FAQs》中被Marshall Cline和Greg Lomow (Addison-Wesley, 1994)创造出来。由于前置条件在派生类中被弱化了，所以称其为*逆变的*（contravariant），相反，后置条件是*协变的*（covariant）（这就解释了为什么我们在第1章中提到了异常规格说明的协变）。

⊜ 这一部分基于Chuck的文章，“The Simplest Automated Unit Test Framework That Could Possibly Work” C/C++ Users Journal, Sept. 2000。

⊝ 关于这个主题的一本好书是Martin Fowler的《Refactoring: Improving the Design of Existing Code》(Addison-Wesley, 2000)。参见http://www.refactoring.com。重构在极限编程中是一种至关重要的实践。

极限编程（extreme programming，XP）[⊖] 仅仅是众多支持快速开发实践方法中的一种。在这一节中，要探究一种便于使用的自动单元测试工具框架，它能够使我们成功地开发出灵活的可扩展的程序。（注意：测试工程师是软件专业人员，以测试其他人编写的代码为生，在软件开发活动中，这些人更是必不可少的。在这里只不过想描述一种方法，这种方法能够帮助软件开发人员开发出更好的代码。）

通过编写单元测试程序，开发者能够对下面的两点关键内容获得足够的信心：

1) 我理解需求。

2) 我的代码符合需求（以我所学的知识来说，它们是最好的）。

先编写单元测试程序是一种能够确保将要编写的代码能够正确工作的最好方法。这种简单的工作能够帮助程序员将精力集中于所要完成的任务上，先编写单元测案例试然后编写代码或许能够导致更快地完成工作，比直接编写代码更快。用极限编程的术语来说就是：

测试程序 + 编码 比直接编码更快。

先编写测试程序同样能够防止边界条件破坏程序，使程序的代码更加健壮。

如果系统无法正常工作，而代码却通过了所有的测试程序，那么问题多半不是出在代码中。“代码已经通过了所有测试程序”就是一个很好的理由。

2.2.1 自动测试

单元测试是什么样子的呢？有太多的开发人员常常只希望他们的代码在获得符合要求的输入时能够产生预期的输出，他们只是用眼睛检查程序的输出。这种方法存在两个危险。首先，程序的输入不可能总是符合要求。大家都知道应该检查程序输入数据的边界，但是当程序员竭尽全力来使程序能够工作时，很难顾及考虑这些事情。如果在编写程序代码之前首先编写测试程序，就可以以测试工程师的角度来问自己，“什么情况会造成程序的破坏呢？”编写测试程序能够证明所写的函数不会被破坏掉，然后程序员再以开发者的身份来完成这些函数。首先编写测试程序能够使程序员写出更好的代码。 72

第二个危险是用眼睛观察来检查程序的输出，这是很乏味的而且容易出错。这样的事情计算机也能做，并且不会出错。最好用布尔表达式的集合来表示测试问题，让测试程序来报告编码的任何错误。

例如，假设想要构造一个**Date**类，这个类有以下的特性：

- 可以用一个字符串(YYYYMMDD)、三个整数(Y, M, D)或者什么也不用（获得当前日期）来初始化日期值。
- **Date**对象能够生成年、月、日的值或“YYYYMMDD”形式的字符串。
- 所有相关量能够进行有效的比较、能够计算两个日期的差（在年、月、日中）。
- 日期的比较能够跨越任意个世纪（例如，1600和2200）。

这个类能够用三个整数分别表示年、月、日。（确保表示年的整数至少是16位（bit）的，以满足上面所说的最后一个特性。）**Date**类的接口可能如下所示：

```
//: C02:Date1.h
// A first pass at Date.h.
#ifndef DATE1_H
#define DATE1_H
#include <string>
```

⊖ 参考Kent Beck所著《Extreme Programming Explained: Embrace Change》，Addison Wesley 1999。轻量级方法，例如XP已经使the Agile Alliance得到了加强（参见http://www.agilealliance.org/home）。

```
class Date {
public:
  // A struct to hold elapsed time:
  struct Duration {
    int years;
    int months;
    int days;
    Duration(int y, int m, int d)
    : years(y), months(m), days(d) {}
  };
  Date();
  Date(int year, int month, int day);
  Date(const std::string&);
  int getYear() const;
  int getMonth() const;
  int getDay() const;
  std::string toString() const;
  friend bool operator<(const Date&, const Date&);
  friend bool operator>(const Date&, const Date&);
  friend bool operator<=(const Date&, const Date&);
  friend bool operator>=(const Date&, const Date&);
  friend bool operator==(const Date&, const Date&);
  friend bool operator!=(const Date&, const Date&);
  friend Duration duration(const Date&, const Date&);
};
#endif // DATE1_H ///:~
```

73

在实现这个类之前，读者可以先编写测试程序，使读者牢固地掌握需求。读者可能提出如下代码:

```
//: C02:SimpleDateTest.cpp
//{L} Date
#include <iostream>
#include "Date.h" // From Appendix B
using namespace std;

// Test machinery
int nPass = 0, nFail = 0;
void test(bool t) { if(t) nPass++; else nFail++; }

int main() {
  Date mybday(1951, 10, 1);
  test(mybday.getYear() == 1951);
  test(mybday.getMonth() == 10);
  test(mybday.getDay() == 1);
  cout << "Passed: " << nPass << ", Failed: "
       << nFail << endl;
}
/* Expected output:
Passed: 3, Failed: 0
*/ ///:~
```

74

在这个普通的测试案例中，函数**test( )**维护着两个全局变量**nPass**和**nFail**。惟一需要程序员用眼睛检查的是最终的得分结果。如果测试失败，更复杂的**test( )**函数能够显示适当的消息。在这一章的后面描述的测试框架不但包括这样一个测试函数，而且还包括其他一些东西。

现在，可以逐步实现**Date**类，使其通过测试，然后可以继续进行反复测试直到满足所有需求。由于是首先编写测试程序，程序员会更多地注意考虑其他边边角角的情况，而这些情况可能破坏即将实现的程序，会使程序员在第一时间就更加注意编写出正确的代码。这样的练习

可能会使读者写出下面的用于测试**Date**类的一种描述:

```
//: C02:SimpleDateTest2.cpp
//{L} Date
#include <iostream>
#include "Date.h"
using namespace std;

// Test machinery
int nPass = 0, nFail = 0;
void test(bool t) { if(t) ++nPass; else ++nFail; }

int main() {
  Date mybday(1951, 10, 1);
  Date today;
  Date myevebday("19510930");

  // Test the operators
  test(mybday < today);
  test(mybday <= today);
  test(mybday != today);
  test(mybday == mybday);
  test(mybday >= mybday);
  test(mybday <= mybday);
  test(myevebday < mybday);
  test(mybday > myevebday);
  test(mybday >= myevebday);
  test(mybday != myevebday);

  // Test the functions
  test(mybday.getYear() == 1951);
  test(mybday.getMonth() == 10);
  test(mybday.getDay() == 1);
  test(myevebday.getYear() == 1951);
  test(myevebday.getMonth() == 9);
  test(myevebday.getDay() == 30);
  test(mybday.toString() == "19511001");
  test(myevebday.toString() == "19510930");

  // Test duration
  Date d2(2003, 7, 4);
  Date::Duration dur = duration(mybday, d2);
  test(dur.years == 51);
  test(dur.months == 9);
  test(dur.days == 3);

  // Report results:
  cout << "Passed: " << nPass << ", Failed: "
       << nFail << endl;
} ///:~
```

这个测试案例可以开发得更加完整。例如，在这里还没有测试程序的可持续性。至此作者所要表达的意思应该已经很清楚了。**Date**类的完整实现在附录中的**Date.h**和**Date.cpp**文件中[⊖]。

2.2.2 TestSuite框架

读者可以从万维网（World Wide Web）下载某些C++自动单元测试工具，例如

⊖ 本书所写的Date类是能够国际化的，也就是说它支持宽字符集（wide character set）。我们将在下一章的结尾介绍宽字符集。

CppUnit。[⊖]在这里，本节的目的不仅仅为了介绍一种易于使用的测试结构，而且要使读者容易理解它，甚至在必要的时候修改它。因此，怀着“只要能用，做最简单的”的信念，[⊜]作者开发了测试套件框架（TestSuite Framework），从其名字的命名就可以看出**TestSuite**中包含两个主要的类：**Test**和**Suite**。

76

Test类是一个抽象基类，可以从这个类派生用户自己的测试对象。**Test**类保存着测试时成功和失败的次数，测试失败时能够显示相关测试条件等信息。只需重写成员函数**run( )**就行了，在这个函数中应该定义一些布尔型的测试条件，并且依次调用**test_( )**宏来测试它们。

为了使用这个框架来定义测试**Date**类的案例，可以继承**Test**类，下面的程序就是这个测试案例：

```
//: C02:DateTest.h
#ifndef DATETEST_H
#define DATETEST_H
#include "Date.h"
#include "../TestSuite/Test.h"

class DateTest : public TestSuite::Test {
  Date mybday;
  Date today;
  Date myevebday;
public:
  DateTest(): mybday(1951, 10, 1), myevebday("19510930") {}
  void run() {
    testOps();
    testFunctions();
    testDuration();
  }
  void testOps() {
    test_(mybday < today);
    test_(mybday <= today);
    test_(mybday != today);
    test_(mybday == mybday);
    test_(mybday >= mybday);
    test_(mybday <= mybday);
    test_(myevebday < mybday);
    test_(mybday > myevebday);
    test_(mybday >= myevebday);
    test_(mybday != myevebday);
  }
  void testFunctions() {
    test_(mybday.getYear() == 1951);
    test_(mybday.getMonth() == 10);
    test_(mybday.getDay() == 1);
    test_(myevebday.getYear() == 1951);
    test_(myevebday.getMonth() == 9);
    test_(myevebday.getDay() == 30);
    test_(mybday.toString() == "19511001");
    test_(myevebday.toString() == "19510930");
  }
  void testDuration() {
    Date d2(2003, 7, 4);
    Date::Duration dur = duration(mybday, d2);
    test_(dur.years == 51);
    test_(dur.months == 9);
```

77

⊖ 参考http://sourceforge.net/projects/cppunit可以得到更多信息。

⊜ 这是极限编程的主要原则之一。

```
    test_(dur.days == 3);
  }
};
#endif // DATETEST_H ///:~
```

运行这个测试案例很简单，只需实例化一个**DateTest**对象并调用它的成员函数**run( )**就可以了。

```
//: C02:DateTest.cpp
// Automated testing (with a framework).
//{L} Date ../TestSuite/Test
#include <iostream>
#include "DateTest.h"
using namespace std;

int main() {
  DateTest test;
  test.run();
  return test.report();
}
/* Output:
Test "DateTest":
        Passed: 21,      Failed: 0
*/ ///:~
```

Test::report( )函数显示前面的输出，并且把测试失败的次数作为返回值，这个值也适合作为**main( )**函数的返回值。

Test类使用运行时类型识别（**RTTI**）[⊖]来取得测试类的类名（例如，**DateTest**），并将这个类名用于测试结果报告。默认情况下**Test**类将测试结果送到标准输出，如果想要把测试结果写到文件中可以使用**setStream( )**成员函数。在本章的后面，读者将会看到**Test**类的实现。 78

test_( )宏能够将失败的布尔条件摘录成文本形式，并且使这段文本包含**test_( )**宏所在文件的文件名和**test_( )**宏所在行的行号。[⊜]为了观察测试失败时会发生什么情况，可以在代码中故意引入错误，例如：可以颠倒上一个例子代码中**DateTest::testOps( )**函数在第一次调用**test_( )**时所用的测试条件。程序的输出准确地显示了在哪里、哪个测试出现了错误。

```
DateTest failure: (mybday > today) , DateTest.h (line 31)
Test "DateTest":
        Passed: 20      Failed: 1
```

除了**test_( )**之外，框架中还包括函数**succeed_( )**和**fail_( )**，这两个函数用于无法使用布尔测试的情况。当测试类的时候可能抛出异常的，这时候应该使用这两个函数。在测试的时候创建一个会触发异常的输入集。如果异常没有发生，就表明程序中出现了错误，这时候应该调用**fail_( )**，就可以清楚地显示一段消息并且修改测试失败次数的值。如果期望的异常发生了，就应该调用**succeed_( )**修改测试成功次数的值。

为了举例说明这两种情况，假设现在已经修改了**Date**类两个非默认构造函数的异常规格说明。如果输入参数不能表示一个合法的日期，这两个构造函数会抛出**DateError**异常（嵌套在**Date**类内的一个类型，派生自**std::logic_error**）：

⊖ “运行时类型识别”将在第9章中讨论。使用**typeinfo**类的**name( )**成员函数。如果读者使用Microsoft Visual C++，必须指定一个编译器选项**/GR**。如果没有指定这个参数，在运行时将会出现非法访问错误。

⊜ 使用字符串化运算（stringizing，通过#预处理运算符）以及预定义的宏**__FILE__**和**__LINE__**。参考本章随后部分的代码。

```
Date(const string& s) throw(DateError);
Date(int year, int month, int day) throw(DateError);
```

79

现在，成员函数**DateTest::run( )**可以调用下面的函数来测试异常处理了：

```
void testExceptions() {
  try {
    Date d(0,0,0);  // Invalid
    fail_("Invalid date undetected in Date int ctor");
  } catch(Date::DateError&) {
    succeed_();
  }
  try {
    Date d("");  // Invalid
    fail_("Invalid date undetected in Date string ctor");
  } catch(Date::DateError&) {
    succeed_();
  }
}
```

在这两种情况下，如果函数中不抛出异常，就表明程序出错了。注意，由于这种测试不计算布尔表达式的值，所以必须用手工方式向**fail_( )**中传递一个消息作为参数，

2.2.3 测试套件

实际的软件项目通常包含很多类，需要一种组织测试用例的方式，使程序设计人员能够通过按一个按钮来测试整个项目。[⊖]**Suite**类可以将测试案例集中到一个函数单元中。程序设计人员可以使用**addTest( )**成员函数添加一个**Test**对象到**Suite**中，也可以使用**addSuite( )**将现有的一个测试套件添加到**Suite**中。为了演示**Suite**的使用，将第3章中用到**Test**类的程序集中到一个单独的测试套件中。注意，这些文件在第3章的文件子目录中。

```
//: C03:StringSuite.cpp
//{L} ../TestSuite/Test ../TestSuite/Suite
//{L} TrimTest
// Illustrates a test suite for code from Chapter 3
#include <iostream>
#include "../TestSuite/Suite.h"
#include "StringStorage.h"
#include "Sieve.h"
#include "Find.h"
#include "Rparse.h"
#include "TrimTest.h"
#include "CompStr.h"
using namespace std;
using namespace TestSuite;

int main() {
    Suite suite("String Tests");
    suite.addTest(new StringStorageTest);
    suite.addTest(new SieveTest);
    suite.addTest(new FindTest);
    suite.addTest(new RparseTest);
    suite.addTest(new TrimTest);
    suite.addTest(new CompStrTest);
    suite.run();
    long nFail = suite.report();
    suite.free();
```

80

⊖ 也可以使用批处理文件和shell脚本文件。**Suite**类是一种基于C++的组织相关测试用例的方法。

```
    return nFail;
}
/* Output:
s1 = 62345
s2 = 12345
Suite "String Tests"
====================
Test "StringStorageTest":
   Passed: 2   Failed: 0
Test "SieveTest":
   Passed: 50  Failed: 0
Test "FindTest":
   Passed: 9   Failed: 0
Test "RparseTest":
   Passed: 8   Failed: 0
Test "TrimTest":
   Passed: 11  Failed: 0
Test "CompStrTest":
   Passed: 8   Failed: 0
*/ ///:~
```

上述的5个测试案例完全包含在头文件中。因为**TrimTest**包含一个静态数据，而静态数据必须定义在实现文件中，所以**TrimTest**不但需要一个头文件，而且还需要实现文件。程序输出的头两行是**StringStorage**测试的结果。程序员必须向测试套件的构造函数传递一个参数，这个参数就是测试套件的名字。成员函数**Suite::run( )**调用它所包含的每一个测试案例的**Test::run( )** 函数。**Suite::report( )** 函数所做的工作与此差不多，也可以将每个测试案例的测试报告输出到不同的流中，而不使用那个属于测试套件的报告。如果用**addSuite( )**添加的测试案例已经被指定了流指针，那么这个测试案例将使用这个流。否则，测试案例使用**Suite**对象指定的输出流。（就像**Test**一样，测试套件的构造函数有1个可选的第2个参数，这个参数的默认值为**std::cout**。）**Suite**的析构函数并不能自动删除它包含的指向**Test**对象的指针，因为这些**Test**对象并不需要保留在堆上；而这些工作由**Suite::free( )**来完成。

81

2.2.4 测试框架的源代码

在代码解压包中，测试框架的源代码包含在一个叫做**TestSuite**的文件子目录中，可以从www.MindView.net网站上得到这个代码的解压包。为了使用这个测试框架，读者必须在头文件的查找路径中包含**TestSuite**子目录，在库文件的查找路径中包含**TestSuite**子目录，这样才能链接相关的目标文件。下面是**Test.h**头文件：

```
//: TestSuite:Test.h
#ifndef TEST_H
#define TEST_H
#include <string>
#include <iostream>
#include <cassert>
using std::string;
using std::ostream;
using std::cout;

// fail_() has an underscore to prevent collision with
// ios::fail(). For consistency, test_() and succeed_()
// also have underscores.

#define test_(cond) \
  do_test(cond, #cond, __FILE__, __LINE__)
#define fail_(str) \
  do_fail(str, __FILE__, __LINE__)
```

```
namespace TestSuite {

class Test {
  ostream* osptr;
  long nPass;
  long nFail;
  // Disallowed:
  Test(const Test&);
  Test& operator=(const Test&);
protected:
  void do_test(bool cond, const string& lbl,
    const char* fname, long lineno);
  void do_fail(const string& lbl,
    const char* fname, long lineno);
public:
  Test(ostream* osptr = &cout) {
    this->osptr = osptr;
    nPass = nFail = 0;
  }
  virtual ~Test() {}
  virtual void run() = 0;
  long getNumPassed() const { return nPass; }
  long getNumFailed() const { return nFail; }
  const ostream* getStream() const { return osptr; }
  void setStream(ostream* osptr) { this->osptr = osptr; }
  void succeed_() { ++nPass; }
  long report() const;
  virtual void reset() { nPass = nFail = 0; }
};

} // namespace TestSuite
#endif // TEST_H ///:~
```

82

Test类有3个虚函数:

- 虚析构函数
- **reset( )**函数
- 纯虚函数**run( )**

我们在第1卷中曾说过，通过基类指针释放在堆上分配的派生类对象是错误的，除非基类有1个虚析构函数。任何想成为基类的类（如果类中出现了至少1个其他虚函数，就说明这个类想成为基类）应该有1个虚的析构函数。**Test::reset( )**的默认实现只是将成功和失败计数器的值重置为零。读者可以重写这个函数，让它重置派生测试对象中数据的状态；确保在重写函数中明确调用**Test::reset( )**，使其重置计数器的状态。因为需要在派生类中重写**Test::run( )**函数，所以它是一个纯虚成员函数。

83

在预处理的时候，**test_( )**和**fail_( )**宏能够取得其所在文件的文件名和其所在行的行号。刚开始的时候并没有在这两个宏的名字后面加下划线，但是**fail( )**宏与**ios::fail( )**产生冲突，造成了编译器错误。

下面是**Test**类其余函数的实现:

```
//: TestSuite:Test.cpp {O}
#include "Test.h"
#include <iostream>
#include <typeinfo>
using namespace std;
using namespace TestSuite;

void Test::do_test(bool cond, const std::string& lbl,
```

```
  const char* fname, long lineno) {
  if(!cond)
    do_fail(lbl, fname, lineno);
  else
    succeed_();
}

void Test::do_fail(const std::string& lbl,
  const char* fname, long lineno) {
  ++nFail;
  if(osptr) {
    *osptr << typeid(*this).name()
           << "failure: (" << lbl << ") , " << fname
           << " (line " << lineno << ")" << endl;
  }
}

long Test::report() const {
  if(osptr) {
    *osptr << "Test \"" << typeid(*this).name()
           << "\":\n\tPassed: " << nPass
           << "\tFailed: " << nFail
           << endl;
  }
  return nFail;
} ///:~
```

Test类不但保存着成功测试的次数和失败测试的次数，而且保存着**Test::report( )**显示测试结果所需的流。**test_( )**和**fail_( )**宏在预处理的时候取得当前文件的文件名和当前行的行号信息，并把文件名传递给**do_test( )**，把行号传递给**do_fail( )**，这两个函数则显示一个消息并修改相关的计数器。我们认为测试对象没有理由使用拷贝和赋值操作，所以通过将这两个函数的原型声明为私有并且忽略他们各自的函数体来禁止这两种操作。 84

下面是**Suite**类的头文件：

```
//: TestSuite:Suite.h
#ifndef SUITE_H
#define SUITE_H
#include <vector>
#include <stdexcept>
#include "../TestSuite/Test.h"
using std::vector;
using std::logic_error;

namespace TestSuite {

class TestSuiteError : public logic_error {
public:
  TestSuiteError(const string& s = "")
  : logic_error(s) {}
};

class Suite {
  string name;
  ostream* osptr;
  vector<Test*> tests;
  void reset();
  // Disallowed ops:
  Suite(const Suite&);
  Suite& operator=(const Suite&);
public:
```

```
  Suite(const string& name, ostream* osptr = &cout)
  : name(name) { this->osptr = osptr; }
  string getName() const { return name; }
  long getNumPassed() const;
  long getNumFailed() const;
  const ostream* getStream() const { return osptr; }
  void setStream(ostream* osptr) { this->osptr = osptr; }
  void addTest(Test* t) throw(TestSuiteError);
  void addSuite(const Suite&);
  void run();  // Calls Test::run() repeatedly
  long report() const;
  void free();  // Deletes tests
};

} // namespace TestSuite
#endif // SUITE_H ///:~
```

85

Suite类在**vector**中保存指向**Test**对象的指针。请注意**addTest( )**成员函数上的异常规格说明。当读者向测试套件中添加一个测试案例的时候，**Suite::addTest( )**检查传递到这个函数的指针是否为空；如果为空指针，则抛出**TestSuiteError**异常。由于在这种情况下不可能把一个空指针传递给测试套件，所以**addSuite( )**用断言检查测试套件所包含的每一个测试案例，就像其他函数遍历测试案例的**vector**一样（请参考下面的实现）。像**Test**类一样，**Suite**类禁止拷贝构造函数和赋值操作。

```
//: TestSuite:Suite.cpp {O}
#include "Suite.h"
#include <iostream>
#include <cassert>
#include <cstddef>
using namespace std;
using namespace TestSuite;

void Suite::addTest(Test* t) throw(TestSuiteError) {
  // Verify test is valid and has a stream:
  if(t == 0)
    throw TestSuiteError("Null test in Suite::addTest");
  else if(osptr && !t->getStream())
    t->setStream(osptr);
  tests.push_back(t);
  t->reset();
}

void Suite::addSuite(const Suite& s) {
  for(size_t i = 0; i < s.tests.size(); ++i) {
    assert(tests[i]);
    addTest(s.tests[i]);
  }
}

void Suite::free() {
  for(size_t i = 0; i < tests.size(); ++i) {
    delete tests[i];
    tests[i] = 0;
  }
}

void Suite::run() {
  reset();
  for(size_t i = 0; i < tests.size(); ++i) {
    assert(tests[i]);
```

86

```
    tests[i]->run();
  }
}

long Suite::report() const {
  if(osptr) {
    long totFail = 0;
    *osptr << "Suite \"" << name
             << "\"\n=======";
    size_t i;
    for(i = 0; i < name.size(); ++i)
      *osptr << '=';
    *osptr << "=" << endl;
    for(i = 0; i < tests.size(); ++i) {
      assert(tests[i]);
      totFail += tests[i]->report();
    }
    *osptr << "=======";
    for(i = 0; i < name.size(); ++i)
      *osptr << '=';
    *osptr << "=" << endl;
    return totFail;
  }
  else
    return getNumFailed();
}

long Suite::getNumPassed() const {
  long totPass = 0;
  for(size_t i = 0; i < tests.size(); ++i) {
    assert(tests[i]);
    totPass += tests[i]->getNumPassed();
  }
  return totPass;
}

long Suite::getNumFailed() const {
  long totFail = 0;
  for(size_t i = 0; i < tests.size(); ++i) {
    assert(tests[i]);
    totFail += tests[i]->getNumFailed();
  }
  return totFail;
}

void Suite::reset() {
  for(size_t i = 0; i < tests.size(); ++i) {
    assert(tests[i]);
    tests[i]->reset();
  }
} ///:~
```

87

在本教材的剩余部分中将使用**TestSuite**框架。

2.3 调试技术

最好的调试习惯是使用本章开始的时候所述的断言；使用断言可以在程序代码真正出现问题之前，帮助程序设计人员找到其中的逻辑错误。这部分内容介绍的一些技巧和技术可以在程序调试过程中给予一定的帮助。

2.3.1 用于代码跟踪的宏

某些情况下，在程序执行过程中将执行到的每一条语句行代码打印到**cout**或一个跟踪文件中是很有用处的。下面是一个能够完成这种功能的预处理宏：

```
#define TRACE(ARG) cout << #ARG << endl; ARG
```

现在可以用这个宏来处理跟踪语句代码了，然而这可能会导致一些问题。例如，如果采用下面的语句：

88

```
for(int i = 0; i < 100; i++)
  cout << i << endl;
```

将这两行程序都放在**TRACE( )**宏中，就会得到如下代码：

```
TRACE(for(int i = 0; i < 100; i++))
TRACE(  cout << i << endl;)
```

预处理之后，代码会变成这样：

```
cout << "for(int i = 0; i < 100; i++)" << endl;
for(int i = 0; i < 100; i++)
  cout << "cout << i << endl;" << endl;
cout << i << endl;
```

这并不是我们想要的。因此，使用这种技术时必须格外细心。

下面是**TRACE( )**宏的一个变种：

```
#define D(a) cout << #a "=[" << a << "]" << endl;
```

如果要想显示一个表达式，只需用它作为参数调用**D( )**。程序执行时会显示这个表达式，接着显示它的值（假设这个表达式的结果类型重载了运算符**<<**）。例如，可以这样写**D(a + b)**。并可以在任何时间使用这个宏来检查中间结果的值。

这两个宏能够完成调试器所能实现的两个最基本功能：跟踪代码的执行过程并显示表达式的值。好的调试器是个杰出且高效的工具，但是在某些时候，可能找不到可以使用的调试器，或者找到的调试器不好用。但是不管在任何情况下，读者都能使用上述两种技术。

2.3.2 跟踪文件

免责声明：这部分和下面部分内容所包含的代码已经被C++标准正式拒绝了。特别是，这里使用宏重定义了**cout**和**new**，如果不仔细的话可能会造成奇怪的结果。这里提供的例子可以在作者使用的所有编译器中正常工作，并提供有用的信息。这是本教材惟一一处偏离编码实践兼容性标准的地方。是否使用在于读者自己！注意，为了使这个例子能够工作，必须使用**using**声明，这样可以去掉**cout**前面的名字空间前缀，例如，在这段代码中不能使用**std::cout**。

89

下面的代码简单地创建了一个跟踪文件，并把所有本来应被送到**cout**的输出送到了这个跟踪文件。现在必须做的所有事情就是**#define TRACEON**并且包含相关的头文件（当然，仅仅将两行关键代码正确地写到文件中是相当的容易）。

```
//: C03:Trace.h
// Creating a trace file.
#ifndef TRACE_H
#define TRACE_H
#include <fstream>

#ifdef TRACEON
std::ofstream TRACEFILE__("TRACE.OUT");
#define cout TRACEFILE__
#endif
```

```
#endif // TRACE_H ///:~
```

下面这段代码是对上述头文件的简单测试:

```
//: C03:Tracetst.cpp {-bor}
#include <iostream>
#include <fstream>
#include "../require.h"
using namespace std;

#define TRACEON
#include "Trace.h"

int main() {
  ifstream f("Tracetst.cpp");
  assure(f, "Tracetst.cpp");
  cout << f.rdbuf(); // Dumps file contents to file
} ///:~
```

因为**cout**已经被**Trace.h**中的宏修改成了其他东西，所以程序中所有的**cout**语句现在都把信息送到了跟踪文件。当读者所使用的操作系统不能简便的进行输出重定向时，这种方式能够方便的将输出保存到文件中。

2.3.3 发现内存泄漏

90

第1卷中讲解过下列直观的调试技术:

1) 为了对数组边界进行检查，可以使用第1卷**C16:Array3.cpp** 中实现的**Array**模板来定义所有数组。当准备发行代码的时候，可以关闭边界检查以提高性能。(尽管这种方法对于指针数组不管用。)

2) 检查基类中的非虚析构函数。

跟踪new/delete和malloc/free语句

通常的内存分配问题包括: 对不是在动态存储区（free store）上分配的内存误使用**delete**，多次重复释放在动态存储区上分配的一个内存，最常见的情况是忘记删除一个指针。这一节讨论了一种能够帮助跟踪这类问题的系统。

上一小节所述免责声明的附加条款: 因为这种方法重载了运算符**new**，所以下述技术在某些平台上可能无法使用，而且只能用于不直接调用**operator new( )**函数的程序。在这本教材中，我们一直非常小心，希望只介绍完全符合C++标准的代码，但是这是一个特例，主要基于如下原因:

1) 尽管这种技术是不合标准的，但是它能用于很多编译器。[⊖]

2) 我们希望利用这种方法来阐述某些有用的思想。

为了使用这种内存检查系统，在这里只需包含头文件**MemCheck.h**，并链接**MemCheck.obj**到应用程序中，这个系统能够截获所有对**new**和**delete**的调用，并且通过在程序中调用宏**MEM_ON( )**（在本章的后面解释）来初始化内存跟踪。所有有关内存分配和释放的踪迹都被打印在标准输出上（通过**stdout**）。当使用这种系统的时候，**new**运算符所在文件的文件名和**new**运算符所在行的行号被保存了下来。这是用**operator new**的定位语法（placement syntax）来完成的。[⊖]虽然在典型的情况下，只有当需要将一个对象放到内存中

91

⊖ 本书主要的技术审阅者，Dinkumware公司的Pete Becker，提醒读者使用宏来替换C++关键字是不符合规定的。他对这种技术的评价是: "这是一种旁门左道（dirty trick）。有时候必须利用旁门左道来找出代码不能正确运行的原因，所以可以把它放到我们的工具箱中，但是不要把它留在发行版本中。"这是对程序员的告诫。

⊖ 感谢C++标准委员会的成员Reg Charney提出这种诀窍。

的指定位置时才使用定位语法。这种内存检查方法也可以创建带有多个参数的**operator new( )**来达到目的。下面的例子就是用有多个参数的**operator new( )**来实现的，当**new**被调用的时候，用**_FILE_**和**_LINE_**宏来获得其所在的文件名和行号并存储：

```
//: C02:MemCheck.h
#ifndef MEMCHECK_H
#define MEMCHECK_H
#include <cstddef>  // For size_t

// Usurp the new operator (both scalar and array versions)
void* operator new(std::size_t, const char*, long);
void* operator new[](std::size_t, const char*, long);
#define new new (__FILE__, __LINE__)

extern bool traceFlag;
#define TRACE_ON() traceFlag = true
#define TRACE_OFF() traceFlag = false

extern bool activeFlag;
#define MEM_ON() activeFlag = true
#define MEM_OFF() activeFlag = false

#endif // MEMCHECK_H ///:~
```

重要的是，当读者想跟踪动态存储区的活动时，可以在任何源文件中包含这个文件，但是它必须是所有被包含文件的最后一个（在其他**#include**之后）。标准库中大部分头文件定义的是模板类，并且大多数编译器使用模板编译的包含模型（inclusion model）（这句话的意思是说，所有源代码都包含在头文件中），**MemCheck.h**中替换**new**的宏将会篡改库中源代码所使用的所有**new**运算符的实例（并且可能造成编译错误）。另外，读者大概只想跟踪存在于自己编写的代码中的内存错误，而不会理会库中的代码是不是有错。

92

下面的文件包含内存跟踪的实现，所有的输出都是通过C的标准输入/输出来完成的，而没有使用C++的输入输出流。它们之间没有什么差别，虽然在编程时没有反对对动态存储区使用输入输出流，但是当尝试使用输入输出流时，有些编译器会报错。而所有编译器都能接受**<cstdio>**版本的输入输出。

```
//: C02:MemCheck.cpp {O}
#include <cstdio>
#include <cstdlib>
#include <cassert>
#include <cstddef>
using namespace std;
#undef new

// Global flags set by macros in MemCheck.h
bool traceFlag = true;
bool activeFlag = false;

namespace {

// Memory map entry type
struct Info {
  void* ptr;
  const char* file;
  long line;
};

// Memory map data
```

```
const size_t MAXPTRS = 10000u;
Info memMap[MAXPTRS];
size_t nptrs = 0;

// Searches the map for an address
int findPtr(void* p) {
  for(size_t i = 0; i < nptrs; ++i)
    if(memMap[i].ptr == p)
      return i;
  return -1;
}

void delPtr(void* p) {
  int pos = findPtr(p);
  assert(pos >= 0);
  // Remove pointer from map
  for(size_t i = pos; i < nptrs-1; ++i)
    memMap[i] = memMap[i+1];
  --nptrs;
}

// Dummy type for static destructor
struct Sentinel {
  ~Sentinel() {
    if(nptrs > 0) {
      printf("Leaked memory at:\n");
      for(size_t i = 0; i < nptrs; ++i)
        printf("\t%p (file: %s, line %ld)\n",
          memMap[i].ptr, memMap[i].file, memMap[i].line);
    }
    else
      printf("No user memory leaks!\n");
  }
};

// Static dummy object
Sentinel s;

} // End anonymous namespace

// Overload scalar new
void*
operator new(size_t siz, const char* file, long line) {
  void* p = malloc(siz);
  if(activeFlag) {
    if(nptrs == MAXPTRS) {
      printf("memory map too small (increase MAXPTRS)\n");
      exit(1);
    }
    memMap[nptrs].ptr = p;
    memMap[nptrs].file = file;
    memMap[nptrs].line = line;
    ++nptrs;
  }
  if(traceFlag) {
    printf("Allocated %u bytes at address %p ", siz, p);
    printf("(file: %s, line: %ld)\n", file, line);
  }
  return p;
}
// Overload array new
void*
```

93

94

```
operator new[](size_t siz, const char* file, long line) {
  return operator new(siz, file, line);
}

// Override scalar delete
void operator delete(void* p) {
  if(findPtr(p) >= 0) {
    free(p);
    assert(nptrs > 0);
    delPtr(p);
    if(traceFlag)
      printf("Deleted memory at address %p\n", p);
  }
  else if(!p && activeFlag)
    printf("Attempt to delete unknown pointer: %p\n", p);
}

// Override array delete
void operator delete[](void* p) {
  operator delete(p);
} ///:~
```

布尔型标志**traceFlag**和**activeFlag**是全局变量，可以在代码中用宏**TRACE_ON( )**、**TRACE_OFF( )**、**MEM_ON( )**和**MEM_OFF( )**来修改它们。一般来说，可以用**MEM_ON( )**和**MEM_OFF( )**这对宏将**main( )**函数中的所有代码包围起来，这样内存的分配和释放就会一直被跟踪。内存跟踪显示了函数**operator new( )**和**operator delete( )**的活动。这种跟踪在默认情况下是打开的，可以用**TRACE_OFF( )**来关闭它。任何情况下，最终结果都会打印出来（参考本章后面的测试运行）。

MemCheck工具在**Info**结构类型的数组中保存全部内存地址、文件名和行号：内存地址是使用**operator new( )**分配内存时得到的，文件名是**new**运算符所在文件的文件名，而行号是**new**运算符所在行的行号。为了避免与放入全局名字空间中的其他名字冲突，应该把尽可能多的内容放在匿名名字空间中。当程序停止的时候，单独存在的**Sentinel**类调用一个静态对象的析构函数。这个析构函数检查**memMap**，看看是否有等待删除的指针（表明程序中存在内存泄漏）。

95

在程序中，**operator new( )**使用**malloc( )**来获取内存，然后把指针和相关的文件信息保存到**memMap**中。**operator delete( )**函数做相反的工作，它调用**free( )**释放内存并对**nptrs**的值减**1**，但是它首先会检查传送过来的指针参数是否在映射表（map）中。如果这个指针不在映射表中，就说明程序员正在试图释放的不是在动态存储区上分配的内存，或者已经释放了这段内存，并把这段内存的地址从映射表中删除了。**activeFlag**变量在这里非常重要，因为不想对系统关闭过程中所做的内存释放活动进行处置。通过在程序代码的最后调用**MEM_OFF( )**可以将**activeFlag**设为**false**，这样，随后的**delete**调用将会被忽略。（在实际的程序中，这样做是不好的，但是，在这里这样做的目的是发现内存泄漏，而不是调试库。）简单地说，现在做的所有工作就是排列**new**和**delete**，将它们进行匹配。

下面是一个使用**MemCheck**工具进行测试的简单例子：

```
//: C02:MemTest.cpp
//{L} MemCheck
// Test of MemCheck system.
#include <iostream>
#include <vector>
#include <cstring>
#include "MemCheck.h"   // Must appear last!
```

```
using namespace std;

class Foo {
  char* s;
public:
  Foo(const char*s ) {
    this->s = new char[strlen(s) + 1];
    strcpy(this->s, s);
  }
  ~Foo() { delete [] s; }
};

int main() {
  MEM_ON();
  cout << "hello" << endl;
  int* p = new int;
  delete p;
  int* q = new int[3];
  delete [] q;
  int* r;
  delete r;
  vector<int> v;
  v.push_back(1);
  Foo s("goodbye");
  MEM_OFF();
} ///:~
```

96

这个例子证实了，可以在如下场合中使用**MemCheck**：代码中使用了流，代码中使用了标准容器（standard containers），以及代码中某个类的构造函数分配了内存。指针**p**和**q**的内存分配和释放没有问题，但是指针**r**不是指向在堆上分配的内存的指针，所以程序的输出显示了一个错误，报告程序试图删除一个未知的指针。

```
hello
Allocated 4 bytes at address 0xa010778 (file: memtest.cpp,
line: 25)
Deleted memory at address 0xa010778
Allocated 12 bytes at address 0xa010778 (file: memtest.cpp,
line: 27)
Deleted memory at address 0xa010778
Attempt to delete unknown pointer: 0x1
Allocated 8 bytes at address 0xa0108c0 (file: memtest.cpp,
line: 14)
Deleted memory at address 0xa0108c0
No user memory leaks!
```

因为调用了**MEM_OFF( )**，所以后面**vector**和**ostream** 对**operator delete( )** 的调用过程并没有进行。读者仍然可能会见到容器重新分配内存时调用**delete**所产生的输出结果。

如果在程序的开始就调用**TRACE_OFF( )**，那么输出结果将是：

```
hello
Attempt to delete unknown pointer: 0x1
No user memory leaks!
```

2.4 小结

令软件工程师头痛的大多数问题都可以通过仔细考虑正在进行的工作来避免。即使没有按常规在代码中使用**assert( )**宏，在编写循环和函数的时候，程序设计人员还是会在心里使用断言。如果使用**assert( )**，将会很快发现程序中存在的逻辑错误，并且最终能够写出可读性

97

更好的程序。记住，断言只能用于不变量条件检查，而不能将其用于运行时错误处理。

没有什么能够比彻底测试完代码能够给软件开发者的心灵带来如此的安宁了。如果在过去的时候（程序中的错误使）人心生烦恼，那么就使用自动测试框架——像教材中介绍的这种——把常规的测试集成到自己的日常工作中吧。软件开发者（和他们的用户）将会为其所做的工作而感到高兴。

2.5 练习

2-1 使用**TestSuite**框架编写一个测试程序来测试标准**vector**类，彻底地测试整型**vector**类的下列成员函数：**push_back( )**（在**vector**的末端添加一个元素）、**front( )**（返回**vector**中的第一个元素）、**back( )**（返回**vector**中的最后一个元素）、**pop_back( )**（删除最后一个元素，不返回它）、**at( )**（返回指定索引位置中的元素）和**size( )**（返回元素的个数）。验证：如果给出的索引产生越界情况，**vector::at( )**会抛出**std::out_of_range**异常。

2-2 假设有人要求开发一个名为**Rational**的类，这个类支持有理数（分数）。在**Rational**对象中的分数始终保存最低项（默认值为0），并且分母为0的情况是一个错误。下面是**Rational**类接口的例子：

```
//: C02:Rational.h {-xo}
#ifndef RATIONAL_H
#define RATIONAL_H
#include <iosfwd>

class Rational {
public:
  Rational(int numerator = 0, int denominator = 1);
  Rational operator-() const;
  friend Rational operator+(const Rational&,
                            const Rational&);
  friend Rational operator-(const Rational&,
                            const Rational&);
  friend Rational operator*(const Rational&,
                            const Rational&);
  friend Rational operator/(const Rational&,
                            const Rational&);
  friend std::ostream&
  operator<<(std::ostream&, const Rational&);
  friend std::istream&
  operator>>(std::istream&, Rational&);
  Rational& operator+=(const Rational&);
  Rational& operator-=(const Rational&);
  Rational& operator*=(const Rational&);
  Rational& operator/=(const Rational&);
  friend bool operator<(const Rational&,
                        const Rational&);
  friend bool operator>(const Rational&,
                        const Rational&);
  friend bool operator<=(const Rational&,
                         const Rational&);
  friend bool operator>=(const Rational&,
                         const Rational&);
  friend bool operator==(const Rational&,
                         const Rational&);
  friend bool operator!=(const Rational&,
                         const Rational&);
```

98

```
};
#endif // RATIONAL_H ///:~
```

为这个类编写一个完整的规格说明，包括前置条件、后置条件和异常说明。

2-3 使用**TestSuite**框架编写一个测试案例，为上一个练习写出的所有规格说明做彻底地测试，包括测试异常。

2-4 实现**Rational**类，使其通过上一个练习时写出的所有测试案例。仅对不变量使用断言。

2-5 下面的**BuggedSearch.cpp**文件包含一个二分查找函数，这个函数在区间**[beg, end)**中查找整型数**what**。算法中有些错误。使用本章介绍的跟踪技术调试这个查找函数。

```
//: C02:BuggedSearch.cpp {-xo}
//{L} ../TestSuite/Test
#include <cstdlib>
#include <ctime>
#include <cassert>
#include <fstream>
#include "../TestSuite/Test.h"
using namespace std;

// This function is only one with bugs
int* binarySearch(int* beg, int* end, int what) {
  while(end - beg != 1) {
    if(*beg == what) return beg;
    int mid = (end - beg) / 2;
    if(what <= beg[mid]) end = beg + mid;
    else beg = beg + mid;
  }
  return 0;
}

class BinarySearchTest : public TestSuite::Test {
  enum { SZ = 10 };
  int* data;
  int max; // Track largest number
  int current; // Current non-contained number
               // Used in notContained()
  // Find the next number not contained in the array
  int notContained() {
    while(data[current] + 1 == data[current + 1])
      ++current;
    if(current >= SZ) return max + 1;
    int retValue = data[current++] + 1;
    return retValue;
  }
  void setData() {
    data = new int[SZ];
    assert(!max);
    // Input values with increments of one.  Leave
    // out some values on both odd and even indexes.
    for(int i = 0; i < SZ;
        rand() % 2 == 0 ? max += 1 : max += 2)
      data[i++] = max;
  }
  void testInBound() {
    // Test locations both odd and even
    // not contained and contained
    for(int i = SZ; --i >=0;)
      test_(binarySearch(data, data + SZ, data[i]));
    for(int i = notContained(); i < max;
        i = notContained())
      test_(!binarySearch(data, data + SZ, i));
  }
```

99

100

```
  void testOutBounds() {
    // Test lower values
    for(int i = data[0]; --i > data[0] - 100;)
      test_(!binarySearch(data, data + SZ, i));
    // Test higher values
    for(int i = data[SZ - 1];
        ++i < data[SZ -1] + 100;)
      test_(!binarySearch(data, data + SZ, i));
  }
public:
  BinarySearchTest() { max = current = 0; }
  void run() {
    setData();
    testInBound();
    testOutBounds();
    delete [] data;
  }
};

int main() {
  srand(time(0));
  BinarySearchTest t;
  t.run();
  return t.report();
} ///:~
```

第二部分　标准C++库

101

标准C++除了包含所有的标准C库外（其中有为支持类型安全而进行的少许增加与更改），还加进了自己特有的库。这些库比起标准C中的库，功能更加强大。可以说，它们的影响力几乎就等同于由C到C++的转变。

本教材的这一部分将对标准C++库的重点部分进行深入的介绍。

有关整个C++库的最全面、也是最令人费解的参考读物就是C++标准本身。Bjarne Stroustrup编写的《The C++ Programming Language》[⊖]（第3版，Addison Wesley, 2000）对C++语言和库来说仍然是值得信赖的参考读物。最著名的专门论述C++库的参考读物是Nicolai Josuttis的《The C++ Standard Library: A Tutorial and Reference》（Addison Wesley，1999）。本部分中各章的编写目的是：给读者提供丰富的描述说明与范例，使读者在解决与标准库的使用相关的任何问题时都有一个好的起点。但是，这里没有涉及某些不常用的技术和主题。如果在这些章里没有找到所需的内容，请参考上述两本书。编写这本教材的目的不是替代上述两本书，而是提供一些必要的补充。希望读者学习完接下来的内容后能够更轻松地理解那两本书。

读者会发现，这些章里并不包含标准C++库中的所有函数和类的详细文档，本教材把这些描述工作留给了别人，特别是P. J. Plauger的《Dinkumware C/C++ Library Reference》，在http://www.dinkumware.com可以得到这些文档。它是用超文本置标语言（Hypertext Markup Language, HTML）格式写成的，是标准库文档联机资源中的精品。读者可守在电脑旁，当需要查找某些内容时，使用一下网络浏览器即可查到。可以联机查阅，也可以购买这些文档放在本机上，以便随时查阅。它包括C和C++库的全部参考页（reference page）。（所以，它能够很好地帮助读者解决有关标准C/C++编程的问题。）电子文档是十分有效的，因为它不但随时可用，而且还能进行电子查找。

102

以上这些资料可满足程序员在奋力编程时的参考需要（如果本书对某些内容讲述不清，也可以参考上述这些资料）。附录A也列举了一些其他的参考资料。

这部分的首章介绍了标准C++ **string**类。它是一个功能非常强大的工具，可以简化在文本处理中可能遇到的大部分“琐事”。很可能在C语言中需要使用多行代码才能完成的字符串处理操作，用**string**类中的一个成员函数调用就可以完成。

第4章介绍的内容是**iostreams**库，它的内容包含与文件、字符串对象（string target）和系统控制台（system console）输入输出操作相关的类。

虽然第5章“深入理解模板”不是完全针对库的一章，但它对后面两章的内容介绍做了必要的准备工作。第6章将研究标准C++库提供的通用算法。由于这些算法都是用模板实现的，因此可以将它们应用于任意对象序列（sequence）。第7章介绍了标准容器及它们关联的迭代器（associated iterator）。首先介绍算法的原因是，只使用数组和**vector**容器（从第1卷开始就一直在使用）就可对其进行全面地研究。很自然地，本教材会在与容器相关的部分使用标准算法，因此在研究容器前熟悉算法是很有好处的。

FULL TEXT ... BRUCEECKEL.COM

⊖ 本书已由机械工业出版社引进出版。——编辑注

第3章 深入理解字符串

*在C语言中，对字符型数组进行字符串处理是最费时的工作之一。字符型数组要求程序员了解静态引用串（static quoted string）与在堆和堆栈中生成的数组之间的差别，实际上有时用类型“**char***”就能达到要求，而有时则必须拷贝整个数组。*

尤其是，由于字符串操作的普遍性，字符型数组可能造成许多混淆与错误。尽管如此，多年来创建字符串类仍是初级C++程序员通常的练习题。标准C++库中的**string**类一劳永逸地解决了字符型数组的处理问题，它监控内存在空间分配和拷贝构造时的情况，程序设计人员根本就不用为此劳神。

本章[⊖]研究标准C++中的**string**类，先简要介绍C++字符串的构成要素，然后阐释C++版本的字符串类与传统C语言字符型数组有哪些不同。读者将会了解使用**string**对象时的各种操作方法，还会看到C++ **string**类在处理不同字符集和字符串数据转换时的神来之笔。

文本处理是编程语言最古老的应用之一，因此C++ **string**类吸取了大量曾经被C及其他编程语言长时间使用的编程思想和术语。当开始介绍C++ **string**类时，应该再次明确这个事实。不论采用哪一种编程方法，有3个操作是我们希望**string**类能够做到的：

- 创建或修改**string**中存放的字符序列。
- 检测**string**中元素的存在性。
- 能够在多种描述**string**字符的方案之间进行转换。

读者将会看到C++ **string**对象是怎样完成这些工作的。

3.1 字符串的内部是什么

在C语言中，字符串基本上就是字符型数组，并且总是以二进制零（通常被称为*空结束符*（null terminator））作为其最末元素。C++ **string**与它们在C语言中的前身截然不同。首先，也是最重要的不同点，C++ **string**隐藏了它所包含的字符序列的物理表示。程序设计人员不必关心数组的维数或空结束符方面的问题。**string**也包含关于其数据容量及存储地址的“内务处理”信息。具体地说，C++ **string**对象知道自己在内存中的开始位置、包含的内容、包含的字符长度（length in characters）以及在必需重新调整内部数据缓冲区的大小之前自己可以增长到的最大字符长度。C++字符串极大地减少了C语言编程中3种最常见且最具破坏性的错误：超越数组边界，通过未被初始化或被赋以错误值的指针来访问数组元素，以及在释放了某一数组原先所分配的存储单元后仍保留了“悬挂”指针。

C++标准没有定义字符串类内存布局（memory layout）的确切实现。采用这种体系结构是为了获得足够的灵活性，从而允许不同的编译器厂商能够提供不同的实现，并且向用户保证提供可预测的行为。特别是，C++标准没有定义在哪种确切的情况下应该为字符串对象分配存储单元来保存数据。字符串分配规则明确规定：允许但不要求引用计数实现（reference-counted implementation），但无论其实现是否采用引用计数（reference counting），其语义都必须一致。

⊖ 本章的某些材料来源于Nancy Nicolaisen。

这种表示稍有不同，在C语言中，每个**char**型数组都占据各自的物理存储区。在C++中，独立的几个**string**对象可以占据也可以不占据各自特定的物理存储区，但是，如果采用引用计数避免了保存同一数据的拷贝副本，那么各个独立的对象（在处理上）必须看起来并表现得就像独占地拥有各自的存储区一样。例如:

```
//: C03:StringStorage.h
#ifndef STRINGSTORAGE_H
#define STRINGSTORAGE_H
#include <iostream>
#include <string>
#include "../TestSuite/Test.h"
using std::cout;
using std::endl;
using std::string;

class StringStorageTest : public TestSuite::Test {
public:
  void run() {
    string s1("12345");
    // This may copy the first to the second or
    // use reference counting to simulate a copy:
    string s2 = s1;
    test_(s1 == s2);
    // Either way, this statement must ONLY modify s1:
    s1[0] = '6';
    cout << "s1 = " << s1 << endl;  // 62345
    cout << "s2 = " << s2 << endl;  // 12345
    test_(s1 != s2);
  }
};
#endif // STRINGSTORAGE_H ///:~

//: C03:StringStorage.cpp
//{L} ../TestSuite/Test
#include "StringStorage.h"

int main() {
  StringStorageTest t;
  t.run();
  return t.report();
} ///:~
```

105

只有当字符串被修改的时候才创建各自的拷贝，这种实现方式称为*写时复制*（copy-on-write）策略。当字符串只是作为值参数（value parameter）或在其他只读情形下使用，这种方法能够节省时间和空间。

不论一个库的实现是不是采用引用计数，它对**string**类的使用者来说都应该是透明的。遗憾的是，情况并不总是这样。在多线程[⊖]程序中，几乎不可能安全地使用引用计数来实现。 106

3.2 创建并初始化C++字符串

创建和初始化字符串对象是一件简单的事情并且相当灵活。在下面的**Smallstring.cpp**例子中，第1个**string**对象**imBlank**虽然被声明了，但并不包含初始值。C语言中的**char**型数组在初始化前都包含随机的无意义的位模式（bit pattern），而与此不同**imBlank**确实包含

⊖ 很难在保证线程安全的前提下实现引用计数（参阅Herb Sutter的《More Exceptional C++》第104~114页）。详见第11章关于多线程编程的部分。

了有意义的信息。这个**string**对象被初始化成包含“没有字符（no character）”，通过类的成员函数能够正确地报告其长度为零并且没有数据元素。

第2个串是**heyMom**，它被文字参数“Where are my socks ?”初始化，这种形式的初始化使用一个引用字符数组（quoted character array）作为**string**构造函数的参数。相比之下，对象**standardReply**只使用一个赋值操作来完成初始化。这一组中的最后一个字符串是**useThisOneAgain**，它的初始化采用的是一个现有的C++**string**对象来完成。换句话说，这个例子阐述了可以对新创建的**string**对象做以下几件事：

- 创建空**string**对象，且并不立即用字符数据对其初始化。
- 将一个文字的引用字符数组作为参数传递给构造函数，以此来对一个**string**对象进行初始化。
- 用等号（=）来初始化一个**string**对象。
- 用一个**string**对象初始化另一个**string**对象。

```
//: C03:SmallString.cpp
#include <string>
using namespace std;

int main() {
  string imBlank;
  string heyMom("Where are my socks?");
  string standardReply = "Beamed into deep "
    "space on wide angle dispersion?";
  string useThisOneAgain(standardReply);
} ///:~
```

107

这些都是**string**对象初始化最简单的形式，但若对此做少许改动，便可更灵活地进行初始化，并对其进行更好地控制。可以这样做：

- 使用C语言的**char**型数组或C++ **string**类两者任一个的一部分。
- 用**operator+**来将不同的初始化数据源结合在一起。
- 用**string**对象的成员函数**substr( )**来创建一个子串。

下面的程序解释了这些特征：

```
//: C03:SmallString2.cpp
#include <string>
#include <iostream>
using namespace std;

int main() {
  string s1("What is the sound of one clam napping?");
  string s2("Anything worth doing is worth overdoing.");
  string s3("I saw Elvis in a UFO");
  // Copy the first 8 chars:
  string s4(s1, 0, 8);
  cout << s4 << endl;
  // Copy 6 chars from the middle of the source:
  string s5(s2, 15, 6);
  cout << s5 << endl;
  // Copy from middle to end:
  string s6(s3, 6, 15);
  cout << s6 << endl;
  // Copy many different things:
  string quoteMe = s4 + "that" +
  // substr() copies 10 chars at element 20
  s1.substr(20, 10) + s5 +
```

```
  // substr() copies up to either 100 char
  // or eos starting at element 5
  "with" + s3.substr(5, 100) +
  // OK to copy a single char this way
  s1.substr(37, 1);
  cout << quoteMe << endl;
} ///:~
```

108

string类对象的成员函数**substr( )**将开始位置作为其第1个参数，而将待选字符的个数作为其第2个参数。两个参数都有默认值。如果使用空的参数列表来调用**substr( )**，那么将会构造出整个**string**对象的一个拷贝，所以这是复制**string**对象的一种简便方法。

下面是程序的输出:

```
What is
doing
Elvis in a UFO
What is that one clam doing with Elvis in a UFO?
```

注意上例的最后一行。C++允许不同的**string**类对象初始化技术在单个语句中的混合使用，这是一种灵活方便的特征。还有，最后一个初始化操作从源**string**对象中复制的仅仅是一个字符。

另一个稍微精巧些的初始化方法利用了**string**类的迭代器**string::begin( )**和**string::end( )**。这种技术将**string**看作容器对象（迄今为止读者所见到的容器主要是**vector**——在第7章将会看到更多的容器），它用迭代器来指示字符序列的开始与结尾。借助这种方法，就可以给**string**类的构造函数传递两个迭代器，构造函数从一个迭代器开始直到另一个迭代器结束，将它们之间的数据拷贝到新的**string**对象中:

```
//: C03:StringIterators.cpp
#include <string>
#include <iostream>
#include <cassert>
using namespace std;

int main() {
  string source("xxx");
  string s(source.begin(), source.end());
  assert(s == source);
} ///:~
```

109

迭代器并不局限于**begin( )**和**end( )**；可以对一个对象使用的迭代器的运算包括增1、减1以及加上整数偏移量，这些运算允许程序员从源**string**对象中提取字符的子集。

不可以使用单个的字符、ASCII码或其他整数值来初始化C++字符串。但是，可用单个字符的多个拷贝来初始化字符串:

```
//: C03:UhOh.cpp
#include <string>
#include <cassert>
using namespace std;

int main() {
  // Error: no single char inits
  //! string nothingDoing1('a');
  // Error: no integer inits
  //! string nothingDoing2(0x37);
  // The following is legal:
  string okay(5, 'a');
  assert(okay == string("aaaaa"));
} ///:~
```

第1个参数表示放入字符串中的第2个参数的拷贝的个数。第2个参数只能是单个字符的**char**型数据，而不能是**char**型数组。

3.3 对字符串进行操作

有用C语言编程经验的人，都习惯用函数族对**char**型数组进行写入、查找、修改和复制等操作。对于**char**型数组的处理，标准C语言库函数中有两个方面不太尽如人意。首先，这些函数分为两族（family），组织得十分松散：无格式（plain）族，以及那些在随后的操作中需要提供计算字符个数的函数族。C语言提供的用于处理**char**型字符数组的那些库函数的函数名列表不但冗长，而且充满了模糊不清、晦涩难懂的名字，其中大部分的名字叫人读不出来，这些都让虔诚的用户很吃惊。虽然这些函数的参数类型及个数颇为一致，但想要用好这些函数，程序员必须对函数命名和参数传递的细节等慎之又慎。

110 标准C语言的**char**型数组工具中存在着其固有的第2个误区，那就是它们都显式地依赖一种假设：字符数组包括一个空结束符。若由于疏忽或是其他差错，这个空结束符被忽略或重写，这个小小的差错就会使C语言的**char**型数组处理函数几乎不可避免地操作其已分配空间之外的内存，有时会带来灾难性的后果。

C++提供的**string**类对象，在使用的便利性和安全性上都有很大的提高。为了实际的字符串处理操作，在**string**类中，不同名的成员函数的数量几乎跟C语言库中的函数一样多，但是由于有重载，使**string**类的功能更加强大。这些特征再加上C++命名机制理性化以及明智地使用了默认参数，使**string**类比起C语言库的**char**型数组函数更便于使用。

3.3.1 追加、插入和连接字符串

C++字符串有几个颇具价值而且最便于使用的特色，其中之一就是：无需程序员干预，它们可根据需要自行扩充规模。这不仅使得字符串处理代码更加可靠，同时也几乎完全消除了令人生厌的“内务处理”琐事——跟踪字符串的存储边界。比方说，创建一个字符串对象并且将其初始化成一个由50个‘X’组成的字符串，然后再存进50个“Zowie”，这个字符串对象自己会自动重新分配足够的存储空间来适应数据的增长。如果代码处理的字符串改变了长度，但程序员并不知道改变的幅度，也许只有这时读者才能最真切地感受到C++字符串的优越性。此外，当字符串增长时，字符串成员函数**append( )**和**insert( )**很明显地重新分配了存储空间：

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

int main() {
  string bigNews("I saw Elvis in a UFO. ");
  cout << bigNews << endl;
  // How much data have we actually got?
  cout << "Size = " << bigNews.size() << endl;
  // How much can we store without reallocating?
  cout << "Capacity = " << bigNews.capacity() << endl;
  // Insert this string in bigNews immediately
  // before bigNews[1]:
  bigNews.insert(1, " thought I");
  cout << bigNews << endl;
  cout << "Size = " << bigNews.size() << endl;
  cout << "Capacity = " << bigNews.capacity() << endl;
  // Make sure that there will be this much space
  bigNews.reserve(500);
```

111

```
  // Add this to the end of the string:
  bigNews.append("I've been working too hard.");
  cout << bigNews << endl;
  cout << "Size = " << bigNews.size() << endl;
  cout << "Capacity = " << bigNews.capacity() << endl;
} ///:~
```

下面是来自特定编译器的输出:

```
I saw Elvis in a UFO.
Size = 22
Capacity = 31
I thought I saw Elvis in a UFO.
Size = 32
Capacity = 47
I thought I saw Elvis in a UFO. I've been
working too hard.
Size = 59
Capacity = 511
```

这个例子证实了，即使可以安全地避免分配及管理**string**对象所占用的存储空间的工作，**C++string**类也提供了几个工具以便监视和管理它们的存储规模。注意到改变分配给字符串的存储空间的规模是多么轻松了吧。**size( )**函数返回当前在字符串存储的字符数，它跟**length( )**成员函数的作用是一样的。**capacity( )**函数返回当前分配的存储空间的规模，也即在没有要求更多存储空间时，字符串所能容纳的最大字符数。**reserve( )**函数提供一种优化机制，它按照程序员的意图，预留一定数量的存储空间，以便将来使用；**capacity( )**返回的值不小于最近一次调用**reserve( )**所使用的值。如果要生成的新字符串的规模比当前的字符串大或者说是需要截短原字符串，**resize( )**函数就会在字符串的末尾追加空格。(**resize( )**的一个重载可以指定一个不同的填充字符。) 112

string类的成员函数为数据分配存储空间的确切方式取决于C++类库的实现。在使用C++类库的某种实现来测试上述例子时，读者会发现，当系统进行存储空间再分配遇到偶数字(word)(即，全整数(full-integer))的边界时，会隐含增加一个字节。为什么会这样呢？**string**类的设计者曾作过不懈的努力让**char**型数组和C++字符串对象可以混合使用，为此，在这种特定的实现中，**StrSize.cpp**报告的存储容量数字，意味着预留出一个字节以便很容易地容纳空结束符(用**char**型数组表示一个字符串时，该字符串的最后一个表示串结束的字符)的插入。

3.3.2 替换字符串中的字符

insert( )函数使程序员放心地向字符串中插入字符，而不必担心会使存储空间越界，或者会改写插入点之后紧跟的字符。存储空间增大了，原有的字符会很“礼貌地”改变其存储位置，以便安置新元素。但有时这并不是程序员所希望的。如果希望字符串的大小保持不变，就应该使用**replace( )**函数来改写字符。**replace( )**有很多的重载版本，最简单的版本用了3个参数：一个参数用于指示从字符串的什么位置开始改写；第二个参数用于指示从原字符串中剔除多少个字符；另外一个是替换字符串(它所包含的字符数可以与被剔除的字符数不同)。举例如下:

```
//: C03:StringReplace.cpp
// Simple find-and-replace in strings.
#include <cassert>
#include <string>
using namespace std;

int main() {
```

```
  string s("A piece of text");
  string tag("$tag$");
  s.insert(8, tag + ' ');
  assert(s == "A piece $tag$ of text");
  int start = s.find(tag);
  assert(start == 8);
  assert(tag.size() == 5);
  s.replace(start, tag.size(), "hello there");
  assert(s == "A piece hello there of text");
} ///:~
```

113

tag串首先插入到**s**串中（注意：在函数调用中的第1个参数值指示的插入点之前进行插入，并且在**tag**串后添加一个额外的空字符），接着进行查找和替换。

在调用**replace( )**前程序员应检查是否会找到什么。前面的例子用一个**char***来进行替换操作，**replace( )**还有一个重载版本，用一个**string**来进行替换操作。下面的例子更完整地演示了**replace( )** 函数：

```
//: C03:Replace.cpp
#include <cassert>
#include <cstddef>  // For size_t
#include <string>
using namespace std;

void replaceChars(string& modifyMe,
  const string& findMe, const string& newChars) {
  // Look in modifyMe for the "find string"
  // starting at position 0:
  size_t i = modifyMe.find(findMe, 0);
  // Did we find the string to replace?
  if(i != string::npos)
    // Replace the find string with newChars:
    modifyMe.replace(i, findMe.size(), newChars);
}

int main() {
  string bigNews = "I thought I saw Elvis in a UFO. "
                   "I have been working too hard.";
  string replacement("wig");
  string findMe("UFO");
  // Find "UFO" in bigNews and overwrite it:
  replaceChars(bigNews, findMe, replacement);
  assert(bigNews == "I thought I saw Elvis in a "
         "wig. I have been working too hard.");
} ///:~
```

114

如果**replace**找不到要查找的字符串，它返回 **string::npos**。数据成员**npos**是**string**类的一个静态常量成员，它表示一个不存在的字符位置。[⊖]

当有新字符复制到现存的一串序列的元素中间时，**replace( )**并不增加**string**的存储空间规模，这一点与**insert( )**不同。但是，**replace( )**必要时也会增加存储空间，例如当所做的“替换”会使原字符串扩充超越到当前分配的存储边界时。举例如下：

```
//: C03:ReplaceAndGrow.cpp
#include <cassert>
#include <string>
```

⊖ 它是“无位置”（no position）的缩写，并且是字符串分配算符**size_type**（默认是**std::size_t**）所能表示的最大值。

```
using namespace std;

int main() {
  string bigNews("I have been working the grave.");
  string replacement("yard shift.");
  // The first argument says "replace chars
  // beyond the end of the existing string":
  bigNews.replace(bigNews.size() - 1,
    replacement.size(), replacement);
  assert(bigNews == "I have been working the "
         "graveyard shift.");
} ///:~
```

对**replace( )**的调用使“替换”超出了原有序列的边界，这与追加操作是等价的。注意，此例中**replace( )**扩展了相应的串序列的规模。

读者可能会不辞劳苦地研读本章，试图找到相对简单的题目，如用一个字符替换字符串中各处出现的另一不同字符。一旦找到前面这些关于替换的材料，读者就会认为找到了答案，然后就开始学习貌似很复杂的材料，如替换字符组和计数等等。难道**string**类就没有一种方法用一个字符替换字符串中各处出现的另一个字符吗？

借助如下的**find( )**和 **replace( )**成员函数，可很容易地实现上述函数：

115

```
//: C03:ReplaceAll.h
#ifndef REPLACEALL_H
#define REPLACEALL_H
#include <string>

std::string& replaceAll(std::string& context,
  const std::string& from, const std::string& to);
#endif // REPLACEALL_H ///:~
```

```
//: C03:ReplaceAll.cpp {O}
#include <cstddef>
#include "ReplaceAll.h"
using namespace std;

string& replaceAll(string& context, const string& from,
  const string& to) {
  size_t lookHere = 0;
  size_t foundHere;
  while((foundHere = context.find(from, lookHere))
    != string::npos) {
    context.replace(foundHere, from.size(), to);
    lookHere = foundHere + to.size();
  }
  return context;
} ///:~
```

此处使用的**find( )**版本将开始查找的位置作为第2参数，如果找不到则返回**string::npos**。将变量**LookHere**表示的位置传送到替换串，这是很重要的，以防字符串**from**是字符串**to**的子串。下面的程序测试了**replaceAll**函数：

```
//: C03:ReplaceAllTest.cpp
//{L} ReplaceAll
#include <cassert>
#include <iostream>
#include <string>
#include "ReplaceAll.h"
using namespace std;
```

```
int main() {
  string text = "a man, a plan, a canal, Panama";
  replaceAll(text, "an", "XXX");
  assert(text == "a mXXX, a plXXX, a cXXXal, PXXXama");
} ///:~
```

116

大家知道，**string**类自身并不能解决所有可能出现的问题。许多解决方案都是由标准库[⊖]中的算法完成的，因为**string**类几乎可与**STL**序列等价（借助于前面所说的迭代器）。所有通用算法的工作对象都是容器中某个“范围”内的元素。通常这个范围指的是“从容器前端到末尾”。**string**对象看上去就像是字符的容器：可用**string::begin( )**得到容器范围的前端，用**string::end( )**得到其末尾。下面的例子显示了如何使用**replace( )**算法将所有单个的字符‘X’替换为‘Y’：

```
//: C03:StringCharReplace.cpp
#include <algorithm>
#include <cassert>
#include <string>
using namespace std;

int main() {
  string s("aaaXaaaXXaaXXXaXXXXaaa");
  replace(s.begin(), s.end(), 'X', 'Y');
  assert(s == "aaaYaaaYYaaYYYaYYYYaaa");
} ///:~
```

注意，这里调用的**replace( )**并不是**string**的成员函数。另外，**replace( )**算法将字符串中出现的某个字符全部用另一个字符替换掉，这一点与**string::replace( )**函数不同，因为后者只进行一次替换。

replace( )算法的工作对象只是单一的对象（本例中是**char**对象），它不会替换引用**char**型数组或**string**对象。由于**string**很像一个STL序列，很多其他算法对它也适用，这些算法可以解决**string**类的成员函数没能直接解决的问题。

117

3.3.3 使用非成员重载运算符连接

对于一个学习C++**string**处理的C程序员来说，等待他的最令人欣喜的发现之一就是，借助**operator+**和**operator+=**可以如此轻而易举地实现**string**的合并与追加。这些运算符使合并串的操作在语法上类似于数值型数据的加法运算：

```
//: C03:AddStrings.cpp
#include <string>
#include <cassert>
using namespace std;

int main() {
  string s1("This ");
  string s2("That ");
  string s3("The other ");
  // operator+ concatenates strings
  s1 = s1 + s2;
  assert(s1 == "This That ");
  // Another way to concatenates strings
  s1 += s3;
  assert(s1 == "This That The other ");
```

⊖ 将在第6章详述。

```
  // You can index the string on the right
  s1 += s3 + s3[4] + "ooh lala";
  assert(s1 == "This That The other The other oooh lala");
} ///:~
```

使用**operator+**和**operator+=**运算符是合并**string**数据的一种既灵活又方便的方法。在语句的右边，程序员几乎可以采用任意一种样式对由单字符或多字符构成的分组进行赋值。

3.4 字符串的查找

string成员函数中的 **find**族是用来在给定字符串中定位某个或某组字符的。下面是 **find** 族成员及其一般用法:

字符串查找成员函数	函数功能及实现
find( )	在一个字符串中查找一个指定的单个字符或字符组。如果找到，就返回首次匹配的开始位置；如果没有查找到匹配的内容，则返回**npos**
find_first_of( )	在一个目标串中进行查找，返回值是第1个与指定字符组中任何字符匹配的字符位置。如果没有查找到匹配的内容，则返回**npos**
find_last_of( )	在一个目标串中进行查找，返回最后一个与指定字符组中任何字符匹配的字符位置。如果没有查找到匹配的内容，则返回**npos**
find_first_not_of( )	在一个目标串中进行查找，返回第一个与指定字符组中任何字符都不匹配的元素的位置。如果找不到那样的元素则返回**npos**
find_last_not_of( )	在一个目标串中进行查找，返回下标值最大的与指定字符组中任何字符都不匹配的元素的位置。若找不到那样的元素则返回**npos**
rfind( )	对一个串从尾至头查找一个指定的单个字符或字符组。如果找到，就返回首次匹配的开始位置。如果没有查找到匹配的内容，则返回**npos**

118

find( )的最简单应用就是在**string**对象中查找一个或多个字符。这个重载的**find( )**函数使用一个参数用来指示要查找的字符（子串），还有另一可选的参数用来表示从字符串的何处开始查找子串。（默认的开始查找位置是0。）把**find**放在循环体内，可以很容易地从头至尾遍历一个字符串，重复查找字符串中所有可能出现的与指定字符或字符组匹配的子串。

119

下面的程序使用Eratosthenes筛选法查找小于50的素数。这种方法从数字2开始，标记所有2（3，5，…）的倍数为非素数，对其他后选素数重复该过程。**SieveTest**的构造函数对**sieveChars**进行初始化，设置其字符序列（array）的初始大小，并且用‘P’来填充每个成员。

```
//: C03:Sieve.h
#ifndef SIEVE_H
#define SIEVE_H
#include <cmath>
#include <cstddef>
#include <string>
#include "../TestSuite/Test.h"
using std::size_t;
using std::sqrt;
using std::string;

class SieveTest : public TestSuite::Test {
  string sieveChars;
```

```
public:
  // Create a 50 char string and set each
  // element to 'P' for Prime:
  SieveTest() : sieveChars(50, 'P') {}
  void run() {
    findPrimes();
    testPrimes();
  }
  bool isPrime(int p) {
    if(p == 0 || p == 1) return false;
    int root = int(sqrt(double(p)));
    for(int i = 2; i <= root; ++i)
      if(p % i == 0) return false;
    return true;
  }
  void findPrimes() {
    // By definition neither 0 nor 1 is prime.
    // Change these elements to "N" for Not Prime:
    sieveChars.replace(0, 2, "NN");
    // Walk through the array:
    size_t sieveSize = sieveChars.size();
    int root = int(sqrt(double(sieveSize)));
    for(int i = 2; i <= root; ++i)
      // Find all the multiples:
      for(size_t factor = 2; factor * i < sieveSize;
           ++factor)
        sieveChars[factor * i] = 'N';
  }
  void testPrimes() {
    size_t i = sieveChars.find('P');
    while(i != string::npos) {
      test_(isPrime(i++));
      i = sieveChars.find('P', i);
    }
    i = sieveChars.find_first_not_of('P');
    while(i != string::npos) {
      test_(!isPrime(i++));
      i = sieveChars.find_first_not_of('P', i);
    }
  }
};
#endif // SIEVE_H ///:~
```

120

```
//: C03:Sieve.cpp
//{L} ../TestSuite/Test
#include "Sieve.h"

int main() {
  SieveTest t;
  t.run();
  return t.report();
} ///:~
```

find( )函数在**string**内部进行搜索，检测多次出现的一个字符或字符组，**find_first_not_of( )**查找其他的字符或子串。

string类中没有改变字符串大小写的函数，但借助于标准C语言的库函数**toupper( )**和**tolower( )**（这两个函数一次只改变一个字符的大小写），可很容易地创建这类函数。下面的例子演示了忽略了大小写的查找：

121

```
//: C03:Find.h
#ifndef FIND_H
#define FIND_H
#include <cctype>
```

```
#include <cstddef>
#include <string>
#include "../TestSuite/Test.h"
using std::size_t;
using std::string;
using std::tolower;
using std::toupper;

// Make an uppercase copy of s
inline string upperCase(const string& s) {
  string upper(s);
  for(size_t i = 0; i < s.length(); ++i)
    upper[i] = toupper(upper[i]);
  return upper;
}

// Make a lowercase copy of s
inline string lowerCase(const string& s) {
  string lower(s);
  for(size_t i = 0; i < s.length(); ++i)
    lower[i] = tolower(lower[i]);
  return lower;
}

class FindTest : public TestSuite::Test {
  string chooseOne;
public:
  FindTest() : chooseOne("Eenie, Meenie, Miney, Mo") {}
  void testUpper() {
    string upper = upperCase(chooseOne);
    const string LOWER = "abcdefghijklmnopqrstuvwxyz";
    test_(upper.find_first_of(LOWER) == string::npos);
  }
  void testLower() {
    string lower = lowerCase(chooseOne);
    const string UPPER = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
    test_(lower.find_first_of(UPPER) == string::npos);
  }
  void testSearch() {
    // Case sensitive search
    size_t i = chooseOne.find("een");
    test_(i == 8);
    // Search lowercase:
    string test = lowerCase(chooseOne);
    i = test.find("een");
    test_(i == 0);
    i = test.find("een", ++i);
    test_(i == 8);
    i = test.find("een", ++i);
    test_(i == string::npos);
    // Search uppercase:
    test = upperCase(chooseOne);
    i = test.find("EEN");
    test_(i == 0);
    i = test.find("EEN", ++i);
    test_(i == 8);
    i = test.find("EEN", ++i);
    test_(i == string::npos);
  }
  void run() {
    testUpper();
    testLower();
    testSearch();
```

122

```
  }
};
#endif // FIND_H ///:~

//: C03:Find.cpp
//{L} ../TestSuite/Test
#include "Find.h"
#include "../TestSuite/Test.h"

int main() {
  FindTest t;
  t.run();
  return t.report();
} ///:~
```

upperCase( )和**lowerCase( )**两个函数的流程形式相同：它们先复制参数**string**对象，接着改变其大小写。程序**Find.cpp**并不是解决大小写敏感问题的最佳方案，所以在讲到**string**的比较时将会再次讨论它。

123 ### 3.4.1 反向查找

如果需要在一个**string**对象中从后往前进行查找（用“后进／先出”的顺序查找数据），可以使用字符串成员函数**rfind( )**：

```
//: C03:Rparse.h
#ifndef RPARSE_H
#define RPARSE_H
#include <cstddef>
#include <string>
#include <vector>
#include "../TestSuite/Test.h"
using std::size_t;
using std::string;
using std::vector;

class RparseTest : public TestSuite::Test {
  // To store the words:
  vector<string> strings;
public:
  void parseForData() {
    // The ';' characters will be delimiters
    string s("now.;sense;make;to;going;is;This");
    // The last element of the string:
    int last = s.size();
    // The beginning of the current word:
    size_t current = s.rfind(';');
    // Walk backward through the string:
    while(current != string::npos) {
      // Push each word into the vector.
      // Current is incremented before copying
      // to avoid copying the delimiter:
      ++current;
      strings.push_back(s.substr(current, last - current));
      // Back over the delimiter we just found,
      // and set last to the end of the next word:
      current -= 2;
      last = current + 1;
      // Find the next delimiter:
      current = s.rfind(';', current);
    }
    // Pick up the first word -- it's not
    // preceded by a delimiter:
```

```
    strings.push_back(s.substr(0, last));
  }
  void testData() {
    // Test them in the new order:
    test_(strings[0] == "This");
    test_(strings[1] == "is");
    test_(strings[2] == "going");
    test_(strings[3] == "to");
    test_(strings[4] == "make");
    test_(strings[5] == "sense");
    test_(strings[6] == "now.");
    string sentence;
    for(size_t i = 0; i < strings.size() - 1; i++)
      sentence += strings[i] += " ";
    // Manually put last word in to avoid an extra space:
    sentence += strings[strings.size() - 1];
    test_(sentence == "This is going to make sense now.");
  }
  void run() {
    parseForData();
    testData();
  }
};
#endif // RPARSE_H ///:~
```

124

```
//: C03:Rparse.cpp
//{L} ../TestSuite/Test
#include "Rparse.h"

int main() {
  RparseTest t;
  t.run();
  return t.report();
} ///:~
```

字符串成员函数**rfind( )**从后往前遍历字符串，查找并且报告与其匹配字符（组）所在的序列排列（array）下标，若不成功则报告**string::npos**。

3.4.2 查找一组字符第1次或最后一次出现的位置

使用**find_first_of( )**和**find_last_of( )**成员函数可以很方便地实现一些小的功能，比如从字符串的头尾两端删除空白字符。注意，它并不触动原字符串，而是返回一个新字符串：

```
//: C03:Trim.h
// General tool to strip spaces from both ends.
#ifndef TRIM_H
#define TRIM_H
#include <string>
#include <cstddef>

inline std::string trim(const std::string& s) {
  if(s.length() == 0)
    return s;
  std::size_t beg = s.find_first_not_of(" \a\b\f\n\r\t\v");
  std::size_t end = s.find_last_not_of(" \a\b\f\n\r\t\v");
  if(beg == std::string::npos) // No non-spaces
    return "";
  return std::string(s, beg, end - beg + 1);
}
#endif // TRIM_H ///:~
```

125

第1次条件判断是为了检查**string**是否为空；如果为空，则直接返回原字符串的1个拷贝，

不再进行其他判断。注意，一旦找到结束点，函数就会使用开始点的位置和计算出来的子串长度作为参数调用**string**类的构造函数，用来创建1个基于原字符串的新的**string**对象。

对这样一个通用工具进行的测试需要十分彻底:

```
//: C03:TrimTest.h
#ifndef TRIMTEST_H
#define TRIMTEST_H
#include "Trim.h"
#include "../TestSuite/Test.h"

class TrimTest : public TestSuite::Test {
  enum {NTESTS = 11};
  static std::string s[NTESTS];
public:
  void testTrim() {
    test_(trim(s[0]) == "abcdefghijklmnop");
    test_(trim(s[1]) == "abcdefghijklmnop");
    test_(trim(s[2]) == "abcdefghijklmnop");
    test_(trim(s[3]) == "a");
    test_(trim(s[4]) == "ab");
    test_(trim(s[5]) == "abc");
    test_(trim(s[6]) == "a b c");
    test_(trim(s[7]) == "a b c");
    test_(trim(s[8]) == "a \t b \t c");
    test_(trim(s[9]) == "");
    test_(trim(s[10]) == "");
  }
  void run() {
    testTrim();
  }
};
#endif // TRIMTEST_H ///:~
```

126

```
//: C03:TrimTest.cpp {O}
#include "TrimTest.h"

// Initialize static data
std::string TrimTest::s[TrimTest::NTESTS] = {
  " \t abcdefghijklmnop \t ",
  "abcdefghijklmnop \t ",
  " \t abcdefghijklmnop",
  "a", "ab", "abc", "a b c",
  " \t a b c \t ", " \t a \t b \t c \t ",
  "\t \n \r \v \f",
  "" // Must also test the empty string
}; ///:~
```

```
//: C03:TrimTestMain.cpp
//{L} ../TestSuite/Test TrimTest
#include "TrimTest.h"

int main() {
  TrimTest t;
  t.run();
  return t.report();
} ///:~
```

读者可以看到，在**strings**型数组中字符型数组自动转换成了**string**对象。读者可以使用这个数组提供测试案例，检查**string**两端的空格和制表符是否删除了，以及确定**string**中间的空格和制表符是否保留了下来。

3.4.3 从字符串中删除字符

使用**erase( )**成员函数删除字符串中的字符是简单而有效的。这个函数有两个参数：一个参数表示开始删除字符的位置（默认值是0）；另一个参数表示要删除多少个字符（默认值是**string::npos**）。如果指定删除的字符个数比字符串中剩余的字符还多，那么剩余的字符将全部被删除（所以调用不含参数的**erase( )**函数将删除字符串中的所有字符）。有时，删除一个HTML文件中的标记（tag）与特殊字符是很有用的，这样就可以得到类似于浏览器中所显示的文本文件，仅仅作为纯文本文件。下面这个例子用**erase( )**来完成这个工作：

127

```
//: C03:HTMLStripper.cpp {RunByHand}
//{L} ReplaceAll
// Filter to remove html tags and markers.
#include <cassert>
#include <cmath>
#include <cstddef>
#include <fstream>
#include <iostream>
#include <string>
#include "ReplaceAll.h"
#include "../require.h"
using namespace std;

string& stripHTMLTags(string& s) {
  static bool inTag = false;
  bool done = false;
  while(!done) {
    if(inTag) {
      // The previous line started an HTML tag
      // but didn't finish. Must search for '>'.
      size_t rightPos = s.find('>');
      if(rightPos != string::npos) {
        inTag = false;
        s.erase(0, rightPos + 1);
      }
      else {
        done = true;
        s.erase();
      }
    }
    else {
      // Look for start of tag:
      size_t leftPos = s.find('<');
      if(leftPos != string::npos) {
        // See if tag close is in this line:
        size_t rightPos = s.find('>');
        if(rightPos == string::npos) {
          inTag = done = true;
          s.erase(leftPos);
        }
        else
          s.erase(leftPos, rightPos - leftPos + 1);
      }
      else
        done = true;
    }
  }
  // Remove all special HTML characters
  replaceAll(s, "&lt;", "<");
  replaceAll(s, "&gt;", ">");
  replaceAll(s, "&amp;", "&");
```

128

```
  replaceAll(s, " ", " ");
  // Etc...
  return s;
}

int main(int argc, char* argv[]) {
  requireArgs(argc, 1,
    "usage: HTMLStripper InputFile");
  ifstream in(argv[1]);
  assure(in, argv[1]);
  string s;
  while(getline(in, s))
    if(!stripHTMLTags(s).empty())
      cout << s << endl;
} ///:~
```

这个例子甚至能够删除跨越多行[⊖]的HTML标记。这归功于静态标志**inTag**，一旦发现了开始标记，此逻辑标志就会被设为**true**，无论相应的结束标记是否与这个开始标记在同一行。所有形式的**erase( )**都包括在**stripHTMLFlags( )**函数中。[⊖]在这里所用的**getline( )**版本是在**<string>**头文件中声明的（全局）函数，这个函数使用起来很方便，因为它可以在其**string**参数中存储任意长度的一行文本。在使用**istream::getline( )**时不必考虑所用字符序列（array）维数的大小。注意，此程序使用了本章开始时介绍的**replaceAll( )**函数。下一章将采用字符串流来构造一个更加优秀的解法。

129

3.4.4 字符串的比较

字符串的比较与数字的比较有其固有的不同。数字有恒定的永远有意义的值。为了评定两个只符串的大小关系，必须进行*字典比较*（lexical comparison）。字典比较的意思是，当测试一个字符看它是“大于”还是“小于”另一个字符时，实际比较的是它们的数值表示，而这些数值表示是由当前所使用的字符集的校对序列来决定的。通常，这种校对序列是ASCII校对序列，它给英语的可打印字符分配的数值为从32到127范围内的连续十进制数字。在ASCII校对序列中，序列表中第一个“字符”是空格，然后是几个常用标点符号，再往后是大小写字母。遵照字母表的编排，比较靠前的字符的ASCII码值都低于比较靠后的字符。知道了这些细节，了解和记忆以下事实就更容易了：当字典比较报告字符串**s1**“大于”字符串**s2**时，也即两者相比较时遇到第1对不同的字符时，字符串**s1**中第1个不同的字符比字符串**s2**中同样位置的字符在ASCII表中的位置更靠后。

C++提供了多种字符串比较方法，它们各具特色。其中最简单的就是使用非成员的重载运算符函数：**operator ==**、**operator !=**、**operator >**、**operator <**、**operator >=**和**operator <=**。

```
//: C03:CompStr.h
#ifndef COMPSTR_H
#define COMPSTR_H
#include <string>
#include "../TestSuite/Test.h"
using std::string;

class CompStrTest : public TestSuite::Test {
public:
```

⊖ 为了简化说明，这一版本不考虑嵌套标记，例如注释。

⊖ 使用数学方法来引发一些对**erase( )**的调用，在此是很有吸引力的。由于某些情况下其操作数之一是**string::npos**（可能得到的最大无符号整型变量），整型溢出就可能发生，进而会搞垮整个算法。

130

```
  void run() {
    // Strings to compare
    string s1("This");
    string s2("That");
    test_(s1 == s1);
    test_(s1 != s2);
    test_(s1 > s2);
    test_(s1 >= s2);
    test_(s1 >= s1);
    test_(s2 < s1);
    test_(s2 <= s1);
    test_(s1 <= s1);
  }
};
#endif // COMPSTR_H ///:~
```

```
//: C03:CompStr.cpp
//{L} ../TestSuite/Test
#include "CompStr.h"

int main() {
  CompStrTest t;
  t.run();
  return t.report();
} ///:~
```

重载的比较运算符不但能进行字符串全串比较还能进行字符串的个别字符元素的比较。

在下面的例子中，注意在比较运算符左右两边的自变量类型的灵活性。为了高效率地运行，对于字符串对象、引用文字和指向C语言风格的字符串的指针等的直接比较，**string**类不创建临时**string**对象，而是采用重载运算符进行。

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

int main() {
  string s2("That"), s1("This");
  // The lvalue is a quoted literal
  // and the rvalue is a string:
  if("That" == s2)
    cout << "A match" << endl;
  // The left operand is a string and the right is
  // a pointer to a C-style null terminated string:
  if(s1 != s2.c_str())
    cout << "No match" << endl;
} ///:~
```

131

c_str( )函数返回一个**const char***，它指向一个C语言风格的具有“空结束符”的字符串，此字符串与**string**对象的内容等价。当想将一个字符串传送给一个标准C语言函数时，比如**atoi( )**或**<cstring>**头文件中定义的任一函数，此时，**const char***可派得上用场。不过，将**c_str( )**的返回值作为非 **const**参数应用于任一函数都是错误的。

在字符串的运算符中，不会找到逻辑非（!）或逻辑比较运算符（**&&**和**||**）。（也不会找到重载版的C语言逐位（二进制数位）运算符**&**、**|**、**^** 或 **~**。）重载字符串类的非成员比较运算符被限定在一个可以清晰地、无二义性地应用于多个字符或字符组的子集中。

compare( )成员函数能够提供远比非成员运算符集更复杂精密的比较手段。它提供的那些重载版本，可以比较:

- 两个完整的字符串。
- 一个字符串的某一部分与另一字符串的全部。
- 两个字符串的子集。

下面的例子用来比较两个完整的字符串:

```
//: C03:Compare.cpp
// Demonstrates compare() and swap().
#include <cassert>
#include <string>
using namespace std;

int main() {
  string first("This");
  string second("That");
  assert(first.compare(first) == 0);
  assert(second.compare(second) == 0);
  // Which is lexically greater?
  assert(first.compare(second) > 0);
  assert(second.compare(first) < 0);
  first.swap(second);
  assert(first.compare(second) < 0);
  assert(second.compare(first) > 0);
} ///:~
```

132

本例中**swap( )**函数所做的工作，顾名思义是：交换其自身对象和参数的内容。为了对一个字符串或两个字符串中的字符子集进行比较，可加上两个参数，一个参数定义开始比较的位置，另一个参数定义字符子集要考虑的字符个数。例如，可以使用下面这个**compare( )**函数的重载版:

s1.compare(s1StartPos, s1NumberChars, s2, s2StartPos, s2NumberChars);

举例如下:

```
//: C03:Compare2.cpp
// Illustrate overloaded compare().
#include <cassert>
#include <string>
using namespace std;

int main() {
  string first("This is a day that will live in infamy");
  string second("I don't believe that this is what "
                "I signed up for");
  // Compare "his is" in both strings:
  assert(first.compare(1, 7, second, 22, 7) == 0);
  // Compare "his is a" to "his is w":
  assert(first.compare(1, 9, second, 22, 9) < 0);
} ///:~
```

在以往的例子中，如果涉及字符串中的个别字符，教材中都使用C语言风格的数组索引语法。C++中的字符串类提供一种**s[n]**表示法的替代方法：**at( )**成员函数。如果不出现意外事件，在C++中这两种索引机制产生的结果是一样的:

```
//: C03:StringIndexing.cpp
#include <cassert>
#include <string>
using namespace std;
```

133

```
int main() {
  string s("1234");
  assert(s[1] == '2');
  assert(s.at(1) == '2');
} ///:~
```

然而，数组索引下标表示[ ]与at( )之间有一个重要的不同点。如果程序员想引用一个超过边界的数组元素，at( )将会友好地抛出一个异常，而普通的[ ]下标语法将让程序员自行决策：

```
//: C03:BadStringIndexing.cpp
#include <exception>
#include <iostream>
#include <string>
using namespace std;

int main() {
  string s("1234");
  // at() saves you by throwing an exception:
  try {
    s.at(5);
  } catch(exception& e) {
    cerr << e.what() << endl;
  }
} ///:~
```

有责任心的程序员不会去用有冒险性的索引，程序员希望能够从自动边界检查中受益。使用at( )代替[ ]，就有机会从容地修复由于引用了不存在的数组元素而产生的错误。在一个测试编译器上执行这个程序，得到的输出结果是：

```
invalid string position
```

at( )成员抛出的是一个out_of_range类对象，它（最终）派生于std::exception。程序可在一个异常处理器中捕获该对象，并采取适当的补救措施，比如重新计算越界下标或扩充数组。采用string::operator[ ]( )不会有那样的保护性，它的危险性等同于C语言中对char型数组的处理。[⊖]

134

3.4.5 字符串和字符的特性

本章前面的程序Find.cpp可能导致读者提出下面这个显而易见的问题：为什么对大小写不敏感的比较没有成为标准string类的一部分？对此问题的回答揭示了关于C++字符串对象真实性质的有趣背景。

读者可以考虑一下，字符有“大小写”到底意味着什么。希伯来语、波斯语和日本汉字并不使用大小写的概念，即对这些语言来说大小写没有什么意义。这似乎是说，如果有方法将一些语言指定为“全大写”或“全小写”，就能够设计出通用的解决方案。但是，某些采用“大小写”概念的语言，同时也用可区别的标记改变了特殊字符的意义，如：西班牙语中的变音符号，法语中的抑扬符号，还有德语中的元音变音。因此，任何试图全面解决此问题的大小写敏感的分类整理方案，最终都会变得非常复杂直至不能再进行下去。

虽然通常将C++ string看成一个类，但事实并非如此。需要说明一下，basic_string< >模板是一种更通用的工具，而string类型只是其更专门化的版本。请看string在标准C++头文

⊖ 鉴于上述安全原因，C++标准制定委员会正考虑一个议案来对string::operator[]进行重新定义，以便在 C++0x 中使其与string::at( )等价。

件里的声明：[⊖]

```
typedef basic_string<char> string;
```

要了解字符串类的本质，请看**basic_string< >**模板：

```
template<class charT, class traits = char_traits<charT>,
  class allocator = allocator<charT> > class basic_string;
```

135 本教材将在第5章中详细讨论模板（比第1卷第16章要详细得多）。但现在，只需注意一下**string**类型是通过使用**char**实例化**basic_string**模板而创建的。在**basic_string< >**模板声明内部，下面的一行：

```
class traits = char_traits<charT>,
```

告诉读者基于**basic_string< >** 模板的类的行为，是由基于**char_traits< >**模板的某个类指定的。因此，**basic_string< >**模板产生的是面向字符串的类，此类的操作对象是除了**char**以外的类型（比如宽字符（wide character））。为了达到这一目的，**char_traits< >**模板控制多种字符集的内容和校对行为，而这些字符集用的是字符比较函数**eq( )**（相等），**ne( )**（不等）和**lt( )**（小于）。**basic_string< >**字符串的比较函数就依赖于这些函数。

这就是为什么字符串类不包含对大小写不敏感的成员函数的原因：因为那不属于它的本职工作。为了改变字符串类比较字符的方式，必须提供不同的**char_traits< >**模板，因为它定义了对个别字符进行比较的成员函数的行为。

可以用此信息构造一种忽略大小写的新类型的**string**类。首先，定义一个从现存模板中继承的一种对大小写不敏感的新的**char_traits< >**模板。其次，仅重写需要更改的成员，使其能逐个字符进行大小写不敏感比较（除了之前提及的3个对字符进行词典比较的成员函数之外，还会为**char_traits**提供函数**find( )**和**compare( )**的新的实现）。最后，我们将用**typedef**定义一个基于**basic_string**的新类，但使用对大小写不敏感的**ichar_traits**模板作为第2个参数：

```
//: C03:ichar_traits.h
// Creating your own character traits.
#ifndef ICHAR_TRAITS_H
#define ICHAR_TRAITS_H
#include <cassert>
#include <cctype>
#include <cmath>
#include <cstddef>
#include <ostream>
#include <string>
using std::allocator;
using std::basic_string;
using std::char_traits;
using std::ostream;
using std::size_t;
using std::string;
using std::toupper;
using std::tolower;

struct ichar_traits : char_traits<char> {
  // We'll only change character-by-
```

136

⊖ 读者实现时可定义这里的所有3个模板参数。由于最后两个模板参数有默认值，那样一个声明与在此写的内容是等价的。

```
  // character comparison functions
  static bool eq(char c1st, char c2nd) {
    return toupper(c1st) == toupper(c2nd);
  }
  static bool ne(char c1st, char c2nd) {
    return !eq(c1st, c2nd);
  }
  static bool lt(char c1st, char c2nd) {
    return toupper(c1st) < toupper(c2nd);
  }
  static int
  compare(const char* str1, const char* str2, size_t n) {
    for(size_t i = 0; i < n; ++i) {
      if(str1 == 0)
        return -1;
      else if(str2 == 0)
        return 1;
      else if(tolower(*str1) < tolower(*str2))
        return -1;
      else if(tolower(*str1) > tolower(*str2))
        return 1;
      assert(tolower(*str1) == tolower(*str2));
      ++str1; ++str2; // Compare the other chars
    }
    return 0;
  }
  static const char*
  find(const char* s1, size_t n, char c) {
    while(n-- > 0)
      if(toupper(*s1) == toupper(c))
        return s1;
      else
        ++s1;
    return 0;
  }
};

typedef basic_string<char, ichar_traits> istring;

inline ostream& operator<<(ostream& os, const istring& s) {
  return os << string(s.c_str(), s.length());
}
#endif // ICHAR_TRAITS_H ///:~
```

137

该程序提供了一个**typedef**命名的**istring**类，这样该类就能在各方面像普通的**string**类一样工作，除了在进行比较的时候不考虑大小写。为了方便起见，程序也提供了一种重载的**operator <<( )**，以便打印**istring**。举例如下：

```
//: C03:ICompare.cpp
#include <cassert>
#include <iostream>
#include "ichar_traits.h"
using namespace std;

int main() {
  // The same letters except for case:
  istring first = "tHis";
  istring second = "ThIS";
  cout << first << endl;
  cout << second << endl;
  assert(first.compare(second) == 0);
```

```
  assert(first.find('h') == 1);
  assert(first.find('I') == 2);
  assert(first.find('x') == string::npos);
} ///:~
```

它只是一个很小的也没有什么实用价值的例子。为使**istring**完全等价于**string**，还得创建其他必要的函数以便支持新的**istring**类型。

通过下面的**typedef**，**<string>** 头文件提供宽字符串类：

```
typedef basic_string<wchar_t> wstring;
```

在宽字符流（wide stream）（代替**ostream**的**wostream**，也在 **<iostream>** 中定义）和头文件 **<cwctype>**（**<cctype>** 的宽字符版本）中，也体现出对宽字符串的支持。运用这些，再加上标准库里**char_traits**中的**wchar_t**说明，就可以完成**ichar_traits**的宽字符版本：

138

```
//: C03:iwchar_traits.h {-g++}
// Creating your own wide-character traits.
#ifndef IWCHAR_TRAITS_H
#define IWCHAR_TRAITS_H
#include <cassert>
#include <cmath>
#include <cstddef>
#include <cwctype>
#include <ostream>
#include <string>

using std::allocator;
using std::basic_string;
using std::char_traits;
using std::size_t;
using std::towlower;
using std::towupper;
using std::wostream;
using std::wstring;

struct iwchar_traits : char_traits<wchar_t> {
  // We'll only change character-by-
  // character comparison functions
  static bool eq(wchar_t c1st, wchar_t c2nd) {
    return towupper(c1st) == towupper(c2nd);
  }
  static bool ne(wchar_t c1st, wchar_t c2nd) {
    return towupper(c1st) != towupper(c2nd);
  }
  static bool lt(wchar_t c1st, wchar_t c2nd) {
    return towupper(c1st) < towupper(c2nd);
  }
  static int compare(
    const wchar_t* str1, const wchar_t* str2, size_t n) {
    for(size_t i = 0; i < n; i++) {
      if(str1 == 0)
        return -1;
      else if(str2 == 0)
        return 1;
      else if(towlower(*str1) < towlower(*str2))
        return -1;
      else if(towlower(*str1) > towlower(*str2))
        return 1;
      assert(towlower(*str1) == towlower(*str2));
      ++str1; ++str2; // Compare the other wchar_ts
    }
```

139

```
    return 0;
  }
  static const wchar_t*
  find(const wchar_t* s1, size_t n, wchar_t c) {
    while(n-- > 0)
      if(towupper(*s1) == towupper(c))
        return s1;
      else
        ++s1;
    return 0;
  }
};

typedef basic_string<wchar_t, iwchar_traits> iwstring;

inline wostream& operator<<(wostream& os,
  const iwstring& s) {
  return os << wstring(s.c_str(), s.length());
}
#endif // IWCHAR_TRAITS_H  ///:~
```

如同读者所见，这基本上是一个要求在源代码中的适当位置放置一个‘w’的练习。测试程序如下所示:

```
//: C03:IWCompare.cpp {-g++}
#include <cassert>
#include <iostream>
#include "iwchar_traits.h"
using namespace std;

int main() {
  // The same letters except for case:
  iwstring wfirst = L"tHis";
  iwstring wsecond = L"ThIS";
  wcout << wfirst << endl;
  wcout << wsecond << endl;
  assert(wfirst.compare(wsecond) == 0);
  assert(wfirst.find('h') == 1);
  assert(wfirst.find('I') == 2);
  assert(wfirst.find('x') == wstring::npos);
} ///:~
```

140

遗憾的是，某些编译器对宽字符仍然没有提供足够的支持。

3.5 字符串的应用

如果仔细查看本教材的程序举例代码，读者会注意到注释中的一些标记。这些都是供Python程序使用的，Python是Bruce所编写的用于提取代码并生成makefile的程序。例如，以双斜线加冒号开始的一行表示源文件的第1行。其余行所描述的信息包括文件名、文件所在的位置以及是否应该只编译文件，而不是生成可执行文件，例如，在上面的程序中，第1行包含字符串**C03:IWCompare.cpp**，它表示文件 **IWCompare.cpp**应该被提取到目录**C03**下。

源文件的最后一行包括3条斜线，其后是一个冒号和一个波形号。如果第1行有一个惊叹号，并且其后紧接着冒号，源代码的第1行和最后一行就不会输出到文件上（这只适用于数据文件）。（为什么在代码中隐藏这些标记呢，那是因为当将代码提取器应用于本教材的代码正文时，我们不想破坏提取器。）

Bruce的Python程序所做的远不止提取代码。如果文件名后有标记“**{O}**”，它在makefile中的条目将被设置为只编译文件，而不将其链接到可执行文件中。（第2章的测试框架就是这么构建的。）为了将这样一个文件与另一个源代码例子链接起来，目标执行文件的源文件会包括一个“**{L}**”指令，就像：

```
//{L} ../TestSuite/Test
```

本部分将介绍一个程序，该程序仅用来提取所有的代码，以便程序员进行手工编译和检查。程序员可以用这个程序来提取本教材中的所有代码，并将文档保存为文本文件[⊖]（称其为TICV2.txt），在shell命令行中执行类似下面的命令：

141

```
C:> extractCode TICV2.txt /TheCode
```

这个命令读取文本文件**TICV2.txt**，然后在根目录**/ TheCode**下的子目录里写出所有源代码文件。目录树如下所示：

```
TheCode/
   C0B/
   C01/
   C02/
   C03/
   C04/
   C05/
   C06/
   C07/
   C08/
   C09/
   C10/
   C11/
   TestSuite/
```

各章中例子的源文件包括在相应的目录里。

下面是程序：

```
//: C03:ExtractCode.cpp {-edg} {RunByHand}
// Extracts code from text.
#include <cassert>
#include <cstddef>
#include <cstdio>
#include <cstdlib>
#include <fstream>
#include <iostream>
#include <string>
using namespace std;
// Legacy non-standard C header for mkdir()
#if defined(__GNUC__) || defined(__MWERKS__)
#include <sys/stat.h>
#elif defined(__BORLANDC__) || defined(_MSC_VER) \
  || defined(__DMC__)
#include <direct.h>
#else
#error Compiler not supported
#endif

// Check to see if directory exists
```

142

⊖ 注意，当文件被存成文本时，Microsoft Word的某些版本将会错误地将单个引述字符替换成扩展了的ASCII码字符，这会造成编译错误，我们不知道错误产生的原因。读者只需用单引号手工替换那个字符就行了。

```
// by attempting to open a new file
// for output within it.
bool exists(string fname) {
  size_t len = fname.length();
  if(fname[len-1] != '/' && fname[len-1] != '\\')
    fname.append("/");
  fname.append("000.tmp");
  ofstream outf(fname.c_str());
  bool existFlag = outf;
  if(outf) {
    outf.close();
    remove(fname.c_str());
  }
  return existFlag;
}

int main(int argc, char* argv[]) {
  // See if input file name provided
  if(argc == 1) {
    cerr << "usage: extractCode file [dir]" << endl;
    exit(EXIT_FAILURE);
  }
  // See if input file exists
  ifstream inf(argv[1]);
  if(!inf) {
    cerr << "error opening file: " << argv[1] << endl;
    exit(EXIT_FAILURE);
  }
  // Check for optional output directory
  string root("./");  // current is default
  if(argc == 3) {
    // See if output directory exists
    root = argv[2];
    if(!exists(root)) {
      cerr << "no such directory: " << root << endl;
      exit(EXIT_FAILURE);
    }
    size_t rootLen = root.length();
    if(root[rootLen-1] != '/' && root[rootLen-1] != '\\')
      root.append("/");
  }
  // Read input file line by line
  // checking for code delimiters
  string line;
  bool inCode = false;
  bool printDelims = true;
  ofstream outf;
  while(getline(inf, line)) {
    size_t findDelim = line.find("//" "/:~");
    if(findDelim != string::npos) {
      // Output last line and close file
      if(!inCode) {
        cerr << "Lines out of order" << endl;
        exit(EXIT_FAILURE);
      }
      assert(outf);
      if(printDelims)
        outf << line << endl;
      outf.close();
      inCode = false;
      printDelims = true;
    } else {
      findDelim = line.find("//" ":");
```

143

```
    if(findDelim == 0) {
      // Check for '!' directive
      if(line[3] == '!') {
        printDelims = false;
        ++findDelim;  // To skip '!' for next search
      }
      // Extract subdirectory name, if any
      size_t startOfSubdir =
        line.find_first_not_of(" \t", findDelim+3);
      findDelim = line.find(':', startOfSubdir);
      if(findDelim == string::npos) {
        cerr << "missing filename information\n" << endl;
        exit(EXIT_FAILURE);
      }
      string subdir;
      if(findDelim > startOfSubdir)
144       subdir = line.substr(startOfSubdir,
                               findDelim - startOfSubdir);
        // Extract file name (better be one!)
        size_t startOfFile = findDelim + 1;
        size_t endOfFile =
          line.find_first_of(" \t", startOfFile);
        if(endOfFile == startOfFile) {
          cerr << "missing filename" << endl;
          exit(EXIT_FAILURE);
        }
        // We have all the pieces; build fullPath name
        string fullPath(root);
        if(subdir.length() > 0)
          fullPath.append(subdir).append("/");
        assert(fullPath[fullPath.length()-1] == '/');
        if(!exists(fullPath))
#if defined(__GNUC__) || defined(__MWERKS__)
          mkdir(fullPath.c_str(), 0);  // Create subdir
#else
          mkdir(fullPath.c_str());  // Create subdir
#endif
        fullPath.append(line.substr(startOfFile,
                        endOfFile - startOfFile));
        outf.open(fullPath.c_str());
        if(!outf) {
          cerr << "error opening " << fullPath
               << " for output" << endl;
          exit(EXIT_FAILURE);
        }
        inCode = true;
        cout << "Processing " << fullPath << endl;
        if(printDelims)
          outf << line << endl;
      }
      else if(inCode) {
        assert(outf);
        outf << line << endl;  // Output middle code line
      }
    }
  }
  exit(EXIT_SUCCESS);
} ///:~
```

首先，读者应注意某些条件编译指令。用于在文件系统中创建目录的**mkdir( )**函数，是

由 POSIX标准[⊖]在头文件**<sys/stat.h>** 中定义的。遗憾的是，很多编译器仍然在使用不同的另一个头文件(**<direct.h>**)。对于不同的头文件，各自的**mkdir( )**识别标志也有所不同：POSIX 指定了两个参数，而旧版本只有一个。因此，在以后的程序中就有了更多的条件编译，以便选择正确的**mkdir( )**进行调用。在本教材的例子中通常不使用条件编译，但是这个特别的程序太有用了，以至于不能放置哪怕一点额外的工作进去，因为读者能用它提取教材中所有的代码。 145

ExtractCode.cpp中的**exists( )**函数通过打开目录中一个临时文件的方式来判断这个目录是否存在。如果打开文件失败，目录就不存在。要删除一个文件，可以将其**char***型名字传送到 **std::remove( )**中。

主程序首先判定命令行参数的合法性，然后一次一行地读取输入文件，同时查找特殊的源代码定界符。布尔标志符**inCode**表示程序在源文件的中间，所以这些代码行应当输出。如果源代码的开始标记不是跟随惊叹号，**printDelims**标志符将为真；否则第1行与最后一行将不会写出。首先查看结束定界符的存在性，这一点很重要，因为开始标记是结束定界符的一个子集。如果先查找开始标记，则程序在找到开始标记和结束定界符的时候都会返回成功。如果遇到结束标记，程序就知道源文件正在处理过程中；否则，文本文件中定界符的摆放方式就有错误了。如果**inCode**为真，那就没什么问题，程序（可选地）写下最后一行然后关闭文件。当找到开始标记时，系统就从语法上分析目录和文件名的组成，然后打开文件。下面几个与**string**有关的函数在此例中都用到了：**length( )**、**append( )**、**getline( )**、**find( )**（两个版本）、**find_first_not_of( )**、**substr( )**、**find_first_of( )**、**c_str( )**，当然还有**operator <<( )**。

3.6 小结

C++ **string**对象的优越性是C语言中相关功能难以望其项背的，这给程序研发者带来了极大的便利。在很大程度上，**string**类使得通过字符型指针来引用字符串已经不再必要了。这就从根本上消除了由于使用未经初始化的指针或具有不正确值的指针造成的一系列软件缺陷。 146

为了适应字符串中数据长度增长变化的需要，C++字符串动态且透明地扩充其内部的数据存储空间。当字符串中存储的数据增长超过最初分配给它的内存空间边界时，字符串对象就会进行存储管理调用，从堆中提取和归还存储空间。稳定的存储分配方案避免了内存泄漏，并且有可能比“依靠（编程人员）自己转来转去”的内存管理方式更加有效。

string类成员函数为字符串的创建、修改和查找提供了相当广泛的工具集。字符串的比较总是大小写敏感的，但也可对字符串进行大小写不敏感的比较。方法是先将字符串数据复制到具有C语言风格的带有空结束符的字符串中，然后调用大小写不敏感字符串比较函数，暂时将字符串对象中存放的数据转换成单一的大写或小写字母；也可以创建大小写不敏感的字符串类，重载用来创建**basic_string**对象的字符特性。

3.7 练习

3-1 编写并测试一个函数，逆转字符串中字符的顺序。

3-2 回文是一个单词或词组，不管从前还是从后开始读，结果都是一样的。例如“madam”或“wow”。编写一个程序，接受来自命令行的一个字符串参数，使用在上一个练习

⊖ POSIX是一个 IEEE标准，支持“便携式操作系统接口（Portable Operating System Interface）”在 Unix系统中的许多低级系统调用，POSIX 就是这些调用的一体化产物。

(3-1)中编写的函数，打印出这个字符串是否为回文。

3-3 修改在练习3-2中编写的程序，如果位置对称的两个字母大小写不同，仍然使其返回**true**。例如“Civic”仍会返回**true**，虽然第一个字母是大写字母。

3-4 修改练习3-3中的程序，使其能够忽略标点符号与空格。例如“Able was I, ere I saw Elba.”也报告**true**。

3-5 使用下面字符串声明并且只能用**char**（不能用印刷错误的串或不可思议的数字）：

147

```
string one("I walked down the canyon with the moving
mountain bikers.");
string two("The bikers passed by me too close for
comfort.");
string three("I went hiking instead.");
```

生成下列句子：

```
I moved down the canyon with the mountain bikers. The
mountain bikers passed by me too close for comfort. So
I went hiking instead.
```

3-6 编写一个名为**replace**的程序，接受3个命令行参数，其中一个参数表示输入的文本文件；一个参数表示被替换掉的字符串（称为**from**）；还有一个表示替换后的字符串（称为**to**）。此程序应该将一个新文件写到标准输出，并将所有的**from**被**to**代替的事件显示出来。

3-7 重写练习3-6，忽略大小写，替换所有**from**。

3-8 使练习3-3中的程序获得一个来自命令行的文件名，然后显示此文件中所有是回文的单词（忽略大小写）。不要重复显示（即使它们的大小写不同）。所找的回文仅限于单词。（与练习3-4不同。）

3-9 修改**HTMLStripper.cpp**，使其在遇到一个标记时就显示这个标记的名字。然后还显示在这个标记与相应的结束标记之间的内容。假设无标记的嵌套，并且所有的标记都有结束标记（表示为<TAGNAME>）。

3-10 编写一个程序，采用3个命令行参数（一个文件名和两个字符串）。按照程序开头处的用户输入（用户会选择使用那一种匹配模式），将文件中那些含有两个字符串的行，两个字符串中任意一个字符串的行、只有一个字符串的行、或两个字符串都不含的行，全部显示到屏幕。除了“两个字符串都不含”的情况外，为了突出强调输入的字符串，在每一个显示的字符串的开头与结尾全部标上星号（*）。

3-11 编写一个程序，采用两个命令行参数（一个文件名和一个字符串），计算字符串在文件中出现的次数，包括其作为子串出现的情况（但不计重叠）。例如，输入字符串“ba”将在单词“basketball”中匹配两次，但输入字符串“ana”在单词“banana”中只匹配一次。将字符串在文件中匹配的次数，还有出现字符串的单词的平均长度显示到屏幕。（如果字符串在单词中出现的次数大于1，当计算该单词的平均长度时，只将该单词计算一次。）

148

3-12 编写一个程序，使用来自命令行的一个文件名，并对字符使用情况进行统计，包括标点符与空格（所有的字符值是从Ox21[33]到Ox7E[126]，还包括空格字符）。也就是说，计算每个字符在文件中出现的次数，然后将它们按ASCII排列顺序（空格，然后！，"，#，等等），或按用户在程序开始时输入的字符使用频率的升序或降序来显示其结果。对于空格，显示单词“Space”而非单个空字符' '。程序运行结果如下所示：

```
Format sequentially, ascending, or descending
(S/A/D): D
t:  526
r:  490
etc.
```

3-13 使用**find( )**和**rfind( )**，编写一个程序。它采用两个命令行参数（一个文件名和一个字符串），显示与该字符串不匹配的第1个和最后一个单词（包括它们的索引值），还有该字符串出现的第一个与最后一个的索引值。当上述任一查找都失败时，显示“Not Found”。

3-14 使用**find_first_of**函数“族”（但不是惟一的）。编写一个程序，删掉文件中所有非字母数字型的字符（空格与句号除外），然后将句号后的第1个字母大写。

3-15 再次使用**find_first_of**函数“族”。编写一个程序，将一个文件名用作命令行参数，然后将文件中所有的数字格式化为货币值。忽略第1个十进制小数点与其后第1个非数值型字符之间的所有小数点，将所得数值四舍五入到百分位。例如，字符串 12.399abc 29.00.6a（美式转换）将被格式化为 $12.40 abc$ 29.01a。

3-16 编写一个程序，采用两个命令行参数（一个文件名和一个数字），搅乱文件中的每一个单词：随机交换每个单词中的两个字母，交换次数由第2个参数提供。（即，如果从命令行传送到程序中的是0，就不能搅乱单词；如果传送进来的是1，一对随机选择的字母应被交换；如果输入的是2，两对随机选择字母将被交换；以此类推。）

3-17 编写一个程序，从命令行获得一个文件名，显示其中句子的个数（定义为文件中句号的个数）、每个句子中字符的平均个数，还有文件中字符的总个数。 149

3-18 自行证明，当有越界情况发生时，**at( )**成员函数确实会抛出一个异常，但是索引运算符（**[ ]**）则不会这么做。

第4章 输入输出流

处理一般的I/O问题，比仅仅使用标准I/O库函数并把它变成一个类需要做更多的工作。

如果能把所有平常的“容器（receptacle）”——标准I/O函数、文件以及内存块一看作相同的对象，都使用相同的接口进行操作，这不是很好吗？这种思想是建立在输入输出流之上的。与C语言**stdio**（标准输入/输出）库中各式各样的函数相比，输入输出流使用起来更容易、更安全，有时甚至更高效。

C++类库中的输入输出流类通常是C++初学者最先学习使用的部分。本章讨论输入输出流中比C语言中**stdio**更强大的功能，阐述了文件流、字符串流和标准控制台流。

4.1 为什么引入输入输出流

读者可能想知道以前的C库到底有什么不好。为什么不把C库封装成新的类呢？有时这是一种好的解决办法。例如，**stdio**中定义的**FILE**为指向文件的指针，假定现在需要安全地打开文件并且不依赖用户调用**close( )**来关闭它，下面的程序可以实现这一目标：

```
//: C04:FileClass.h
// stdio files wrapped.
#ifndef FILECLASS_H
#define FILECLASS_H
#include <cstdio>
#include <stdexcept>

class FileClass {
  std::FILE* f;
public:
  struct FileClassError : std::runtime_error {
    FileClassError(const char* msg)
    : std::runtime_error(msg) {}
  };
  FileClass(const char* fname, const char* mode = "r");
  ~FileClass();
  std::FILE* fp();
};
#endif // FILECLASS_H ///:~
```


当在C语言中进行文件I/O时，是使用无保护的指向FILE **struct**的指针来完成有关操作，但这个类封装了文件结构指针，并且用构造函数和析构函数来确保指针被正确地初始化和清理。构造函数的第2个参数是文件打开模式，默认值为“**r**”即“只读模式”。

为了在文件I/O函数中使用这个指针的值，可以用存取访问函数（access function）**fp( )**取得它。下面是这个成员函数的定义：

```
//: C04:FileClass.cpp {O}
// FileClass Implementation.
#include "FileClass.h"
#include <cstdlib>
#include <cstdio>
using namespace std;

FileClass::FileClass(const char* fname, const char* mode) {
```

```
  if((f = fopen(fname, mode)) == 0)
    throw FileClassError("Error opening file");
}

FileClass::~FileClass() { fclose(f); }

FILE* FileClass::fp() { return f; } ///:~
```

就像平常所做的一样，构造函数调用**fopen( )**，而且要确保返回结果不为零，结果为零说明打开文件失败。如果文件不能正常打开，则抛出异常。

析构函数用来关闭文件，而存取访问函数**fp( )**则返回指针**f**。下面是使用**FileClass**的一个简单例子：

```
//: C04:FileClassTest.cpp
//{L} FileClass
#include <cstdlib>
#include <iostream>
#include "FileClass.h"
using namespace std;
int main() {
  try {
    FileClass f("FileClassTest.cpp");
    const int BSIZE = 100;
    char buf[BSIZE];
    while(fgets(buf, BSIZE, f.fp()))
      fputs(buf, stdout);
  } catch(FileClass::FileClassError& e) {
    cout << e.what() << endl;
    return EXIT_FAILURE;
  }
  return EXIT_SUCCESS;
} // File automatically closed by destructor
///:~
```

153

现在，创建一个**FileClass**对象并在普通的C文件I/O函数中通过调用**fp( )**使用它。当用完这个对象之后就不需要再理会它了；当文件对象超出其作用域后，析构函数会关闭该文件。

虽然**FILE**指针是私有的，但它并不是特别安全，因为成员函数**fp( )**可以检索它。既然惟一的作用似乎只是为了确保指针能被正确初始化和清除，那么为什么不把它设计成公有的或使用**struct** 来代替呢？注意，当能够用函数**fp( )**取得指针**f**的一个拷贝的时候，不能同时给**f**赋值——这项操作完全由类来控制。得到由**fp( )**返回的指针后，客户程序员仍然能给结构元素赋值或对其进行进一步处理，所以从安全的角度对于**FILE**指针，与其确保其合法性还不如将其作为结构的固有成员。

如果需要得到完全的安全，就必须防止客户直接存取**FILE**指针。所有的常用文件I/O函数都必须作为成员函数封装在类中，使得借助于C语言能做到的每一件事，在C++类中均可做到：

```
//: C04:Fullwrap.h
// Completely hidden file IO.
#ifndef FULLWRAP_H
#define FULLWRAP_H
#include <cstddef>
#include <cstdio>
#undef getc
#undef putc
#undef ungetc
using std::size_t;
using std::fpos_t;
```

154

```
class File {
  std::FILE* f;
  std::FILE* F(); // Produces checked pointer to f
public:
  File(); // Create object but don't open file
  File(const char* path, const char* mode = "r");
  ~File();
  int open(const char* path, const char* mode = "r");
  int reopen(const char* path, const char* mode);
  int getc();
  int ungetc(int c);
  int putc(int c);
  int puts(const char* s);
  char* gets(char* s, int n);
  int printf(const char* format, ...);
  size_t read(void* ptr, size_t size, size_t n);
  size_t write(const void* ptr, size_t size, size_t n);
  int eof();
  int close();
  int flush();
  int seek(long offset, int whence);
  int getpos(fpos_t* pos);
  int setpos(const fpos_t* pos);
  long tell();
  void rewind();
  void setbuf(char* buf);
  int setvbuf(char* buf, int type, size_t sz);
  int error();
  void clearErr();
};
#endif // FULLWRAP_H ///:~
```

这个类几乎包含了**<cstdio>**中所有的文件**I/O**函数。（不包含**vfprintf( )**，它只是用来实现**printf( )**成员函数。）

类**File**的构造函数和前面的例子相同，并且这个类还有一个默认的构造函数。如果想创建**File**对象数组，或把**File**对象作为另一个类的数据成员来使用，这时类的初始化操作不在构造函数中完成，而是发生在其所属的对象已经创建之后，在这些情况下默认构造函数是很重要的。

155 默认构造函数将私有**FILE**指针**f**设为0。但是在对**f**进行任何引用之前，必须对其进行检查以确保指针不为空。这项操作由成员函数**F( )**完成，这个函数为私有成员函数，这样做的目的是只允许类中的其他成员函数调用它。（不想让用户直接访问类的**FILE**结构。）

无论如何这并不是一种糟糕的解决方法。这种方法能起很好的作用，甚至能设想为标准（控制台）I/O和内核格式化（in-core formatting）（读/写一个内存块，而不是文件或控制台）构造相似的类。

在这里遇到的绊脚石是用于可变参数列表函数（variable argument list function）的运行时解释程序（runtime interpreter）。运行时解释程序是一段代码，它的作用是在运行时解析格式串（format string），以及提取并解释从可变参数列表中得到的参数。产生这个问题有4个原因：

1）即使仅仅需要使用解释程序的一小部分功能，该解释程序的所有内容也都会被加载到可执行程序中。所以，如果在程序中仅仅使用**printf("%c",'x');**，那么程序包中所有的函数也都会被加载进来，包括打印浮点数和字符串的函数。没有标准选项可以减少程序使用的空间。

2）因为解释是发生在运行时的，所以无法免除运行开销。这是很令人沮丧的，因为编译时所有的信息都存在格式串中，但是直到运行时刻才能对其进行求值。然而，如果能在编译时解析格式串中的变量，就可以产生直接的函数调用，速度比运行时解释程序更快（尽管**printf( )**及同类函数已经很好地优化了）。

3) 因为格式串直到运行时才能求值，所以可以没有编译时错误检查。如果读者曾经为找出函数**printf( )**中的错误而对其使用错误的数字或者参数类型进行测试，也许就对这个问题比较熟悉了。C++为尽早发现错误，就进行编译时错误检查做了许多工作，这使得代码的编写更加容易。把类型安全检查交给I/O库来完成似乎是欠妥的，尤其是进行大量I/O操作时。

4) 对于C++来说，最关键的问题是**printf( )**函数族不具备可扩展性。设计它们的目的仅仅是用来处理C语言中的基本数据类型（**char**、**int**、**float**、**double**、**wchar_t**、**char***、**wchar_t***和**void***）以及这些数据类型的变体。读者也许会认为每次添加一个新类时，可以重载函数**printf( )**和**scanf( )**（以及它们的用于处理文件和字符串的变体），但是请记住，重载函数的参数列表中参数的类型必须不同，然而**printf( )**函数族把类型信息隐藏在可变参数列表和格式串中。对于一种语言如C++来说，如果设计它的目的是为了很容易地添加新的数据类型，那么这个限制是无法接受的。 156

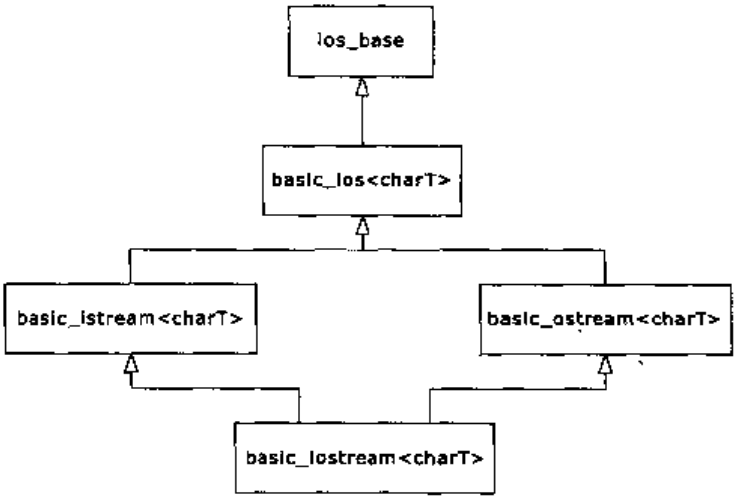
4.2 救助输入输出流

这些问题清楚地表明I/O是标准C++类库最重要的内容之一。由于“hello，world”几乎是每个程序员学习一门新语言时所编写的第1个程序，并且实际上每个程序都会用到I/O，所以C++中的I/O类库必须非常易于使用。更大的挑战在于I/O类库必须适用于任何新的类。如此一来，这个基础类库在设计时就需要一些技巧。除了能够学习到处理I/O和格式化操作用到的多种方法并提高其使用的准确性外之外，在本章中读者还会看到一个真正功能强大的C++类库是如何工作的。

4.2.1 插入符和提取符

流是一个传送和格式化固定宽度（fixed width）字符的对象。读者可以获得一个输入流（通过**istream**类的子类）、一个输出流（使用**ostream**对象）、或者同时实现两种功能的流（使用从**iostream**派生的对象）。输入输出流类库提供了下面几种不同的类：用于文件输入输出的**ifstream**、**ofstream**和**fstream**，用于标准C++中**string**类输入输出的**istringstream**、**ostringstream**和**stringstream**。所有的这些流类拥有几乎相同的接口，所以能够以统一的方式使用这些流类，不管操作对象是文件、标准I/O、内存区，还是**string**对象。这样单一的接口同样支持扩充和增加一些新定义的类。某些函数实现格式化命令，而某些函数以非格式化方式读写字符。

前面提到的流类实际上是模板的特化（template specialization），就像标准**string**类是**basic_string**模板的特化[⊖]。下图描述了输入输出流类继承体系中的基本类： 157



⊖ 在第5章中深入讨论。

类**ios_base**声明了所有流类共有的内容，不依赖于流所处理的字符类型。这些声明大部分是常量以及处理这些常量的函数，它们会在本章反复出现。其他类是以基础字符类型为参数的模板。例如类**istream**，定义如下：

```
typedef basic_istream<char> istream;
```

定义本章前面提及的类时，都使用了相似的类型定义（type definition）。另外，C++中也用**wchar_t**（第3章中介绍的宽字符类型）来替换**char**定义了所有的输入输出流类。这在本章末尾可以看到。模板**basic_ios**定义了输入和输出通用的函数，但是这依赖于基础字符类型(几乎不使用它们)。模板**basic_istream**定义了一般的输入函数，**basic_ostream**定义了一般的输出函数。后面介绍的文件流类和字符串流类增加了特殊的流处理功能。

158

在输入输出流类库中，重载了两种运算符以简化输入输出流的使用。运算符 << 常用作输入输出流的*插入符*（inserter），运算符 >> 常用作*提取符*（extractor）。

提取符按照目标对象的类型解析输入信息，举例说明，可以使用**cin**对象，它是输入流，相当于C中的**stdin**，即可重定向标准输入（redirectable standard input）。在代码中包含头文件**<iostream>**时，就会预定义这个对象。

```
int i;
cin >> i;

float f;
cin >> f;

char c;
cin >> c;

char buf[100];
cin >> buf;
```

所有的内置数据类型都重载了**operator >>**。读者自己也可以重载**operator >>**，这将在后面看到。

为了显示不同变量中的内容，读者可以与插入符 << 一起使用**cout**对象（相当于标准输出（standard output）；同样地，**cerr**对象相当于标准错误输出（standard error））：

```
cout << "i = ";
cout << i;
cout << "\n";
cout << "f = ";
cout << f;
cout << "\n";
cout << "c = ";
cout << c;
cout << "\n";
cout << "buf = ";
cout << buf;
cout << "\n";
```

159

尽管增强了类型检查功能，但是这样写出来的代码很乏味，而且似乎比用**printf( )**写出来的代码没有多大的提高。幸运的是，重载的插入符和提取符可以连续使用，构成复杂的表达式，使得写（和读）更容易：

```
cout << "i = " << i << endl;
cout << "f = " << f << endl;
cout << "c = " << c << endl;
cout << "buf = " << buf << endl;
```

为自己的类定义插入符和提取符，就是重载相关的运算符以完成正确的操作，即：

- 第1个参数定义成流（输入为**istream**，输出为**ostream**）的非**const**引用。
- 执行向/从流中插入/提取数据的操作（通过处理对象的组成元素）。
- 返回流的引用。

输入输出流应该是非常量，因为处理流数据将改变流的状态。通过返回流，如前所述，可以将这些流操作链接成单一的语句。

举个例子，考虑如何输出一个MM-DD-YYYY格式的**Date**类对象。下面的代码使用了插入符：

```
ostream& operator<<(ostream& os, const Date& d) {
  char fillc = os.fill('0');
  os << setw(2) << d.getMonth() << '-'
     << setw(2) << d.getDay() << '-'
     << setw(4) << setfill(fillc) << d.getYear();
  return os;
}
```

这个函数不能设为**Date**类的成员函数，因为运算符 << 左边的操作数必须是输出流。**ostream**的成员函数**fill( )**用于更换填充字符（padding character），当输出域（field）的宽度大于输出数据长度时，使用填充字符填充超出部分，域宽由操纵算子（manipulator）**setw( )**决定。使用“0”作为前导填充字符，所以显示10月之前的月份时，如显示9月份为“09”。函数**fill( )**返回原有的填充字符（默认为一个空格符），以便在后面使用操纵算子**setfill( )**恢复这个填充字符，本章后面将深入讨论操纵算子。 160

使用提取符需要注意输入数据错误。通过设置流的失败标志位（fail bit）可以表明产生了流错误，如下所示：

```
istream& operator>>(istream& is, Date& d) {
  is >> d.month;
  char dash;
  is >> dash;
  if(dash != '-')
    is.setstate(ios::failbit);
  is >> d.day;
  is >> dash;
  if(dash != '-')
    is.setstate(ios::failbit);
  is >> d.year;
  return is;
}
```

一旦流的失败标志位被设置，则在流恢复到有效状态之前，此外所有的流操作都会被忽略（简要说明一下）。这就是为什么即使设置了**ios::failbit**，上述代码也继续进行提取操作。这种实现有些宽松，因为它允许在日期字符串的数字和短线之间插入空格（因为在默认情况下 >> 运算符在读取内置数据类型时会跳过空格）。对提取符来说，下面是合法的日期字符串：

```
"08-10-2003"
"8-10-2003"
"08 - 10 - 2003"
```

下面是非法的日期字符串：

```
"A-10-2003" // No alpha characters allowed
"08%10/2003" // Only dashes allowed as a delimiter
```

将会在4.3节处理流错误部分深入讨论流状态。 161

4.2.2 通常用法

正如**Date**提取符所展示的，必须防止错误的输入。如果输入一个非法的值，处理过程就会出现错误，并且很难恢复。另外，默认情况下的格式化输入使用空格作为界定符。把前面的代码片断合成一个单独的程序，看看会发生什么情况：

```
//: C04:Iosexamp.cpp {RunByHand}
// Iostream examples.
#include <iostream>
using namespace std;

int main() {
  int i;
  cin >> i;

  float f;
  cin >> f;

  char c;
  cin >> c;

  char buf[100];
  cin >> buf;

  cout << "i = " << i << endl;
  cout << "f = " << f << endl;
  cout << "c = " << c << endl;
  cout << "buf = " << buf << endl;

  cout << flush;
  cout << hex << "0x" << i << endl;
} ///:~
```

给出如下输入：

```
12 1.4 c this is a test
```

预期的输出：

162

```
12
1.4
c
this is a test
```

但是输出和预期的有些不同：

```
i = 12
f = 1.4
c = c
buf = this
0xc
```

注意，**buf**仅得到了第1个单词，因为输入机制是通过寻找空格来分割输入的，而空格出现在 **"this"**的后面。另外，如果连续输入的字符串长度超过**buf**的存储空间，就会发生**buf**溢出现象。

实际上在交互程序中，经常需要一次输入一行字符序列，当这些字符安全地存储到缓冲区后再进行扫描和转换工作。使用这种方法，读者不必担心输入程序的执行因非法数据的出现而阻塞。

另一个需要考虑的内容是在命令行界面的输入输出。这仅在过去控制台比一台打字机强不

了多少的时候才有意义，但是现在图形用户界面（graphical user interface，GUI）迅速占据了统治地位。在这样的环境下使用控制台I/O是否还有意义呢？除了很简单的例子或测试外，可以完全忽略**cin**并采用下面的方法：

1) 如果程序需要输入，则从文件中读入数据——读者很快就会看到通过输入输出流来使用文件非常容易。文件输入输出流在图形用户界面下也能很好地工作。

2) 正如我们建议的那样，读取输入但不试图对其进行转换。当输入数据被保存到某处，并且在转换时不会造成错误时，才可以安全地扫描它。

3) 输出的情况有所不同。如果使用图形用户界面，就不需要用**cout**，必须把数据输出到文件（和输出到**cout**是一样的），或者使用图形用户界面应用程序实现数据显示。否则，把数据输出到**cout**便很有意义。在这两种情况下，输入输出流的输出格式化功能十分有用。 163

在大型项目中，另一个常用的方法可以节省编译时间。例如，看看本章前面是如何在头文件中声明**Date**流操作符的。仅需要包含函数的原型，不需要在**Date.h**中包含整个**<iostream>**头文件。标准的方法是像下面这样仅声明类：

```
class ostream;
```

这是一种将接口从实现中分离的早就在频繁使用的技巧，称做是*前置声明*（(forward declaration)，在其出现的位置上，**ostream**应当被视为*未完成的类型*，因为这时编译器还没有看到类的定义）。

然而，这样的声明不能正常工作，有两个原因：

1) 流类是在名字空间**std**中定义的。

2) 这些流类是模板。

正确的声明应该是：

```
namespace std {
  template<class charT, class traits = char_traits<charT> >
    class basic_ostream;
  typedef basic_ostream<char> ostream;
}
```

（正如读者所看到的，就像**string**类，流类使用了第3章中提到的字符特征类。）由于为所有需要引用的流类编写代码是十分枯燥乏味的，C++标准提供了头文件**<iosfwd>**来完成这些工作。**Date**头文件如下所示：

```
// Date.h
#include <iosfwd>

class Date {
  friend std::ostream& operator<<(std::ostream&,
                                  const Date&);
  friend std::istream& operator>>(std::istream&, Date&);
// Etc.
```

164

4.2.3 按行输入

有3种可选的方法来实现按行输入：

- 成员函数**get( )**
- 成员函数**getline( )**
- 定义在头文件**<string>**中的全局函数**getline( )**

前两个函数有3个参数：

1) 指向字符缓冲区的指针，用于保存结果。

2) 缓冲区的大小（为了保证缓冲区不会溢出）。

3) 结束字符，根据结束字符判断何时停止读入操作。

结束字符（terminating character）默认情况下为'\n'，这是常用的结束字符。当在输入过程中遇到结束字符时，这两个函数都会在结果缓冲区末尾存储一个零。

那么，**get( )**和**getline( )**两个函数有什么不同呢？细微而重要的区别在于：当遇到输入流中的界定符（delimiter，即结束字符）时**get( )**停止执行，但是并不从输入流中提取界定符。这时，如果再次调用**get( )**会遇到同一个界定符，函数将立即返回而不会提取输入。（为了解决这个问题，据推测，需要在下一个**get( )**函数中使用不同的界定符或使用不同的输入函数。）函数**getline( )**则相反，它将从输入流中提取界定符，但仍然不会把它存储到结果缓冲区中。

<string>中定义的函数**getline( )**使用起来很方便。它不是成员函数，而是在名字空间**std**中声明的独立函数。这个函数仅有两个非默认参数，输入流和**string**对象。从函数名可以看出，它从输入流中读取字符直到遇到第1个界定符（默认为'\n'）并且丢弃这个界定符。这个函数的优点在于它把数据读入一个**string**对象中，所以不用担心缓冲区的大小。

一般来说，当采用按行输入的方式处理文本文件时，需要使用其中的一个**getline( )**函数。

165

1. **get( )**函数的重载版本

函数**get( )**也有另外3个重载版本：其中一个版本没有参数，使用**int**作为返回值类型，返回下一个字符；另一个版本使用**char**类型的引用作为参数，函数从流中读取一个字符放到这个参数中；还有一个版本则把流类对象直接存储到基础的缓冲区结构。本章后面将对最后一个版本进行深入研究。

2. 读原始字节

如果准确知道正在处理的数据并需要把字节直接移动到内存中一个变量、数组或结构中，可以使用非格式化的I/O函数**read( )**。这个函数的第1个参数是指向目标内存的指针，第2个参数是需要读入的字节数。如果预先把信息保存在文件中，例如使用输出流的**write( )**成员函数将信息保存为二进制形式（当然，使用相同的编译器），那么这个函数就十分有用。在后面读者会看到这些函数的例子。

4.3 处理流错误

前面涉及的**Date**提取符在某种情况下将设置流的失败位。那么，用户如何知道这样的失败是何时发生的呢？可以通过调用流的成员函数来测试流错误，或者如果不关心到底发生了什么错误，而仅仅用来评估流中这个布尔量的来龙去脉。这两种技术都依赖于流的错误位的状态。

1. 流状态

类**ios_base**从类**ios**派生而来[⊖]，定义了4个标志位来测试流的状态：

标志位	意　义
badbit	发生了（或许是物理上的）致命性错误。流将不能继续使用
eofbit	输入结束（文件流的物理结束或用户结束了控制台流输入，例如用户按下了Ctrl-Z 或 Ctrl-D）
failbit	I/O操作失败，主要原因是非法数据（例如，试图读取数字时遇到字母）。流可以继续使用。输入结束时也将设置failbit标志
goodbit	一切正常；没有错误发生。也没有发生输入结束

166

⊖ 由于这个原因，可以使用**ios::failbit** 取代**ios_base::failbit**以节省输入。

可以通过调用相应成员函数，根据其返回的布尔值来测试发生了什么情况，返回的布尔值指出设置了哪个标志位。当除了**goodbit**之外的其他3个标志位没有被设置时，流类的成员函数**good( )**返回真（**true**）。如果**eofbit**被设置则函数**eof( )**返回真，表明程序试图从已经到达末尾的流（通常是文件流）中读取数据。因为输入结束（end-of-input）发生在C++的读操作试图越过物理介质末尾的时候，所以**failbit**标志位也会被设置，表示期望获取的数据没有被成功读取。如果设置了**failbit**或**badbit**标志位中的任意一个，则函数**fail( )**返回真，只有设置了**badbit**标志位，函数**bad( )**才返回真。

一旦设置了流状态中的任何一个标志位，这些标志位将保持不变，但程序员并不希望总保持这种状况。读文件时，也许想在文件结束之前将文件指针重定位到前面的位置。仅靠移动文件指针不会自动重置**eofbit**或**failbit**标志位。需要程序员自己在程序中调用**clear( )**函数来清空标志位，如下所示：

```
myStream.clear(); // Clears all error bits
```

调用**clear( )**后，立即调用函数**good( )**则返回**true**。正如较早时提及的**Date**提取符一样，函数**setstate( )**用来设置标志位，而标志位的值由我们传递给它。函数**setstate( )**不会影响其他标志位，如果已经设置了其他的标志位，则这些标志位将保持不变。如果想设置某个特定的标志位而重置其他所有的标志位，可以调用重载的**clear( )** 函数，将需要设置的标志位的表达式作为参数传递给它，如下所示：

```
myStream.clear(ios::failbit | ios::eofbit);
```

在大多数情况下，程序员不想逐个检查流状态位，只想知道所有操作是否成功完成。当从头到尾读文件时就是这种情况；此时只想知道什么时候读文件完成。可以使用返回值为**void***的转换函数，当流出现在布尔表达式中时，这个函数被自动调用。使用下面的语句完成流的读入，直到输入结束：

167

```
int i;
while(myStream >> i)
  cout << i << endl;
```

记住，**operator>>( )**返回它的流参数，所以上面的**while**语句用布尔表达式对流进行测试。这个特殊的例子假定输入流**myStream**包含由空格符分隔的整数。函数**ios_base::operator void *( )**仅在流上调用函数**good( )** 并返回调用结果[⊖]。因为大部分流操作返回流对象，所以使用这个语句是比较方便的。

2. 流类和异常

在出现异常概念之前的很长时间里，输入输出流是作为C++的一部分存在的，所以一直采用手工方式检查流状态。为了保持向后兼容性，这种手工检查流状态的方法仍能在程序中使用，但是现代的输入输出流采用抛出异常的方法来代替它。流的成员函数**exceptions( )**接受一个参数，这个参数用于表示程序员希望在哪个状态位出现时抛出异常。当遇到这样的状态时，就抛出一个**std::ios_base::failure**类型的异常，**std::ios_base::failure**继承自**std::exception**。

尽管可以为4种流状态中的任何一个状态触发失败异常，但是这样做并不是好办法。如第1章所述，要在真正发生异常的情况下使用异常，但是“已至文件尾”不仅不是异常，而且是在

⊖ 习惯上优先使用函数**operator void*( )**而不是**operator bool( )**，因为从**bool**型隐式转换到**int**型会引起错误，在用整形表达式时，不应该错误地应用流。函数**operator void*( )**在布尔表达式中应该隐式调用。

预料之中的正常情况。因此读者也许只想对**badbit**所代表的错误使用异常，使用方法如下：

168

```
myStream.exceptions(ios::badbit);
```

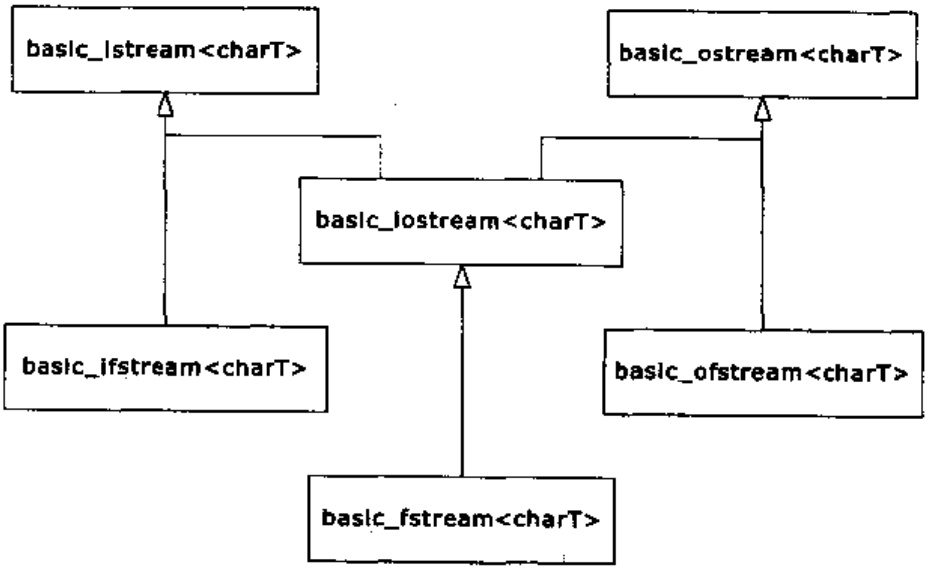
在流接流（stream-by-stream）的基础上能使用异常，因为**exceptions( )**是流类的成员函数。函数**exceptions( )**返回位掩码（bitmask）[⊖]（**iostate**类型，它依赖于编译器，可转成**int**型），表示哪种流状态会触发异常。如果这些状态位已经设置，就会立即抛出异常。当然，如果与流一起使用异常，则最好捕获异常，这意味着需要把流处理过程封装到含有**ios::failure**处理器的**try**块中。许多程序员认为这很乏味，他们只在发生错误的地方手工检查错误状态（因此假如，大部分情况下他们不希望**bad( )**返回**true**）。这是让流抛出异常成为可选项而不是默认项的另一原因。在任何情况下都可以选择处理流错误的方式。基于相同的原因，我们推荐使用异常进行错误处理，这里也采用这种方法。

4.4 文件输入输出流

使用输入输出流类操纵文件比使用C语言中的**stdio**更容易、更安全。打开一个文件要做的全部工作就是创建一个对象——这是构造函数所做的工作。不需要显式地关闭文件（尽管能使用成员函数**close( )**来关闭文件），因为当对象超出作用域时析构函数会关闭文件。构造一个**ifstream**对象用于创建默认的输入文件。构造一个**ofstream**对象用于创建默认的输出文件。一个**fstream**对象即可以用于输入文件，也可以用于输出文件。

169

下图说明了适用于输入输出流类的文件流类：



和以前一样，这里实际使用的类都是由类型定义的模板的特化。例如，**ifstream**用来处理**char**文件，定义如下：

```
typedef basic_ifstream<char> ifstream;
```

4.4.1 一个文件处理的例子

下面这个例子展示了迄今为止所讨论到的所有特性。注意，对**<fstream>**的包含声明了文件输入输出类。尽管在许多平台上，这样的声明将自动包含**<iostream>**，但编译器不一定

⊖ 用作单个标志位的一个整数类型。

都这样做。如果想写出可移植的代码，则这两个头文件都要包含进来：

```
//: C04:Strfile.cpp
// Stream I/O with files;
// The difference between get() & getline().
#include <fstream>
#include <iostream>
#include "../require.h"
using namespace std;

int main() {
  const int SZ = 100; // Buffer size;
  char buf[SZ];
  {
    ifstream in("Strfile.cpp"); // Read
    assure(in, "Strfile.cpp"); // Verify open
    ofstream out("Strfile.out"); // Write
    assure(out, "Strfile.out");
    int i = 1; // Line counter

    // A less-convenient approach for line input:
    while(in.get(buf, SZ)) { // Leaves \n in input
      in.get(); // Throw away next character (\n)
      cout << buf << endl; // Must add \n
      // File output just like standard I/O:
      out << i++ << ": " << buf << endl;
    }
  } // Destructors close in & out

  ifstream in("Strfile.out");
  assure(in, "Strfile.out");
  // More convenient line input:
  while(in.getline(buf, SZ)) { // Removes \n
    char* cp = buf;
    while(*cp != ':')
      ++cp;
    cp += 2; // Past ": "
    cout << cp << endl; // Must still add \n
  }
} ///:~
```

170

创建**ifstream**对象和**ofstream**对象后都调用了函数**assure( )**，以确保文件被成功打开。在编译器希望产生结果为布尔值的地方，流对象产生了表示成功或失败的值。

第1个**while**循环演示了两个**get( )**函数的使用。第1个**get( )**读字符到缓冲区，当已经读取了**SZ-1**个字符或者读取字符过程中遇到了第3个参数（默认为**'\n'**）所规定的字符时，在缓冲区中写入零结束符。函数**get( )**跳过输入流中的结束字符，所以需使用没有参数的**get( )**，通过调用**in.get( )**丢掉结束字符，这个函数读取一个字节，并返回一个**int**型的值。也可以使用具有两个默认参数的成员函数**ignore( )**。它的第1个参数表示要跳过的字符的数目，默认值为1。第2个参数指明遇到哪个字符时，函数**ignore( )**退出（在提取这个字符之后），默认值为**EOF**。

171

紧接着是两个相似的输出语句：一个输出到**cout**，一个输出到文件**out**。注意，使用这个方法很方便，不需要担心对象类型，因为格式化语句对所有**ostream**对象都能很好地处理。前一条语句把一行显示到标准输出上，第2条语句将一行以及其行号写入一个新文件中。

为演示函数**getline( )**如何工作，现在打开刚刚创建的文件，跳过行号。在以读方式再次打开文件之前，为了确保正确地关闭这个文件，这里有两种方法可供选择。可以使用花括号把程序的第1部分括起来，从而强制对象**out**超出作用域，这会使系统调用析构函数并关闭该文

件，这个程序就是这样做的。也可以调用成员函数**close( )**关闭这些文件；如果采用这种方法，甚至可以通过调用成员函数**open( )**重用对象**in**。

第2个**while**循环语句说明了函数**getline( )**遇到输入流中的结束字符（函数的第3个参数，默认为**'\n'**）时是如何将其除去的。尽管**getline( )**像**get( )**一样在缓冲区中写入字符零，但它不在缓冲区插入结束字符。

这个例子，连同本章的大部分例子，都假定对任何重载函数**getline( )**的调用都会遇到新行界定字符。如果不是这种情况，流的状态位**eofbit**就会被设置，并且对**getline( )**的调用返回**false**，造成程序丢失该输入的最后一行字符。

4.4.2 打开模式

通过覆盖构造函数的默认参数，可以控制文件的打开模式。下表说明了控制文件打开模式的标志:

标 志	功 能
ios::in	打开输入文件。将这种打开模式应用于**ofstream**可以使得现存文件不被截断
ios::out	打开输出文件。将这种模式应用于**ofstream**，而且没有使用**ios::app**、**ios::ate**或**ios::in**时，意味着使用的是**ios::trunc**模式
ios::app	打开一个仅用于追加的输出文件
ios::ate	打开一个已存在文件（输入或输出），并把文件指针指向文件末尾
ios::trunc	如果文件存在，则截断旧文件
ios::binary	以二进制方式打开文件。默认打开为文本方式

172

位掩码（bitwise）或运算符可以使用这些标记的组合。

二进制标记只在一些非UNIX系统上起作用，例如从MS-DOS发展而来的操作系统，这些系统对行结束界定符有特殊约定。例如，在MS-DOS系统的文本模式（默认模式）下，每输出一个换行符(**'\n'**)，文件系统实际上输出了两个字符，一个回车符/换行符（CRLF）对，ASCII码为**0x0D**和**0x0A**。相反，当在文本模式下读这样的文件到内存，遇到这样的字节对时，就在相应的位置写入一个**'\n'**来替代。如果想绕过这个特殊的处理过程，可以采用二进制模式打开文件。在二进制模式下，不论是否写生僻的字节到文件，都没有需要特殊处理的事情，总是能进行操作（通过调用**write( )**）。然而，应该在使用**read( )**或**write( )**的时候以二进制模式打开文件，因为这些函数有一个每次读/写字节个数的参数。在这些情况下，如果流中包含额外字符**'\r'**，字节个数参数将不再起作用。如果想使用本章后面将要讨论的流指针定位命令，则也应该使用二进制模式打开文件。

可以通过声明**fstream**对象打开一个文件，同时用作输入和输出。声明**fstream**对象的时候，必须使用足够的打开模式标记，告诉文件系统现在想进行输入、输出或既输入又输出。从输出切换到输入的时候，需要刷新流或者重定位文件指针。从输入切换到输出，也需要重定位文件指针。通过**fstream**对象创建一个文件，在构造函数中使用**ios::trunc**打开模式标志，可以调用输入和输出文件。

173

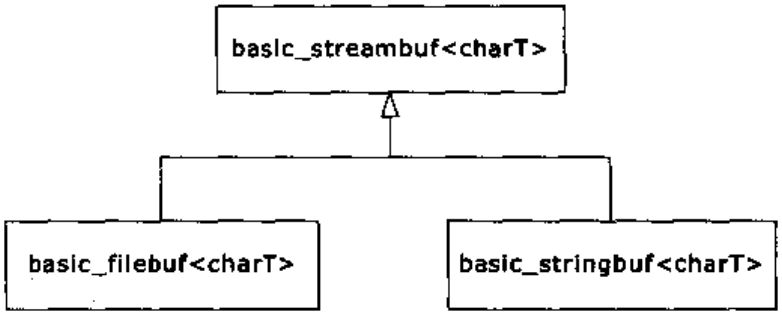
4.5 输入输出流缓冲

好的设计原则指出，无论何时创建一个新类，都应该尽最大努力隐藏实现细节。只让用户看到他们需要知道的东西，将其他东西都设为**private**以避免引起混乱。使用插入符和提取符

时，一般程序员不知道或不必关心数据在哪里产生和消亡，不管处理的对象是标准I/O、文件、内存还是新创建的类或设备。

然而，重要的是与产生和消耗数据的输入输出流部分进行通信。为给这部分提供统一的接口并隐藏底层实现，标准库把它抽象成一个类，称为**streambuf**。每个输入输出流对象包含一个指向**streambuf**的指针。（对象类型依赖于被处理的内容是标准I/O、文件、还是内存等等。）用户可以直接访问**streambuf**；例如，可以把原始字节移入或移出**streambuf**，而不用通过封装它的输入输出流对其进行格式化。这可以通过调用**streambuf**对象的成员函数来完成。

目前，读者需要知道的最重要的事情，就是每个输出流对象包含一个指向**streambuf**对象的指针，并且**streambuf**对象拥有一些可供调用的成员函数。对于文件流类和字符串流类，他们有特殊的流缓冲区类型，如下图所示：



为了能够访问**streambuf**，每个输入输出流对象有一个成员函数**rdbuf( )**，它返回一个指向对象**streambuf**的指针，通过这个指针可以对**streambuf**对象进行存取。这样就可以对基础的**streambuf**调用任何成员函数。然而，更有趣的是可以把**streambuf**指针和另一个输入输出流对象用运算符 << 连接到一起。这将把右侧对象中的字符都输出到运算符 << 左侧的对象中去。这样一来，如果想把一个输入输出流中的字符全部移动到另一个输入输出流中，就不需要通过令人厌烦的方法一次读一个字符或一行字符（会引起潜在的错误）。这是一种很优雅的方法。 174

下面是一个简单的示例程序，该程序打开一个文件并把文件中的内容送到到标准输出（和前面的例子相似）：

```
//: C04:Stype.cpp
// Type a file to standard output.
#include <fstream>
#include <iostream>
#include "../require.h"
using namespace std;

int main() {
  ifstream in("Stype.cpp");
  assure(in, "Stype.cpp");
  cout << in.rdbuf(); // Outputs entire file
} ///:~
```

在这里使用这个程序的源代码文件作为参数创建了一个**ifstream**对象。如果文件不能打开，则函数**assure( )**报告打开文件失败。所有的工作实际上都发生在如下语句中

```
cout << in.rdbuf();
```

这条语句把文件中的全部内容输出到**cout**。这样的语句不仅使得代码更简洁，而且常常比一

次移动若干字节效率更高。

一种形式的**get( )**函数直接把数据写入另一个对象的**streambuf**中。这个函数的第1个参数是目标**streambuf**的引用，第2个参数是使函数**get( )**停止执行的结束字符（默认情况下为'**\n**'）。如下所示，还有另外一种将文件发送到标准输出的方法：

175

```
//: C04:Sbufget.cpp
// Copies a file to standard output.
#include <fstream>
#include <iostream>
#include "../require.h"
using namespace std;

int main() {
  ifstream in("Sbufget.cpp");
  assure(in);
  streambuf& sb = *cout.rdbuf();
  while(!in.get(sb).eof()) {
    if(in.fail())          // Found blank line
      in.clear();
    cout << char(in.get()); // Process '\n'
  }
} ///:~
```

函数**rdbuf( )**返回一个指针，它必须解除引用（dereference），以满足函数查看对象的需要。流缓冲区不会被复制（它们没有拷贝构造函数），所以将**sb**定义为**cout**的流缓冲区的引用。并调用函数**fail( )**和**clear( )**以处理输入文件中的空行（在这个例子中就是如此）。当这个特殊的重载函数**get( )** 看到在一行中有两个新界定符（一个空行的证据），就设置输入流的失败位，所以必须调用**clear( )**来重置失败位以继续从流中读取数据。对**get( )**的第2次调用提取并回应每个新行的界定字符。（记住，**get( )**和**getline( )**提取界定字符的方法不同。）

也许并不需要经常使用这种技术，但是知道这种方法总是有益处的[⊖]。

4.6 在输入输出流中定位

176

每种类型的输入输出流都有“下一个”字符从哪里来（如果是**istream**）或到哪里去（如果是**ostream**）的概念。在某些情况下，需要移动流的位置（通过设置“流指针”给流重新定位）。可以使用两种方式进行流定位：一种是使用称为**streampos**的流指针进行绝对流位置定位；另一种方式和标准C中用于处理文件的库函数**fseek( )**相似，实现从文件头、文件尾或当前位置移动某个给定的字节数进行相对流定位。

用**streampos**进行绝对流位置定位，需要先调用一个“告知”函数，以便知道流指针在流中的确切位置：对于**ostream**调用**tellp( )**，对于**istream**调用**tellg( )**。（“p”表示“写指针”，“g”表示“读指针”。）这个函数返回一个**streampos**，稍后当要回到流中定位流指针时要用到它，对**ostream**对象调用**seekp( )**，对**istream**对象调用**seekg( )**。

第2种方法是相对定位，使用重载版本的**seekp( )**和**seekg( )**函数。函数的第1个参数是要移动的字符数目：这个数目可正可负。第2个参数是移动方向：

⊖ 关于流缓冲区和流的更深入的讨论可参考Addison-Wesley出版社1999年出版的Langer & Kreft的《Standard C++ Iostreams and Locales》。

ios::beg	流的开始位置
ios::cur	流的当前位置
ios::end	流的末端位置

下面是在文件中进行定位的一个例子，但是这里没有C语言中**stdio**对于在文件中进行定位所做的那些限制。在C++中，可以在任何类型的输入输出流中进行定位（尽管在标准流对象如**cin**和**cout**中不允许这样做）：

```
//: C04:Seeking.cpp
// Seeking in iostreams.
#include <cassert>
#include <cstddef>
#include <cstring>
#include <fstream>
#include "../require.h"
using namespace std;

int main() {
  const int STR_NUM = 5, STR_LEN = 30;
  char origData[STR_NUM][STR_LEN] = {
    "Hickory dickory dus. . .",
    "Are you tired of C++?",
    "Well, if you have,",
    "That's just too bad.",
    "There's plenty more for us!"
  };
  char readData[STR_NUM][STR_LEN] = {{ 0 }};
  ofstream out("Poem.bin", ios::out | ios::binary);
  assure(out, "Poem.bin");
  for(int i = 0; i < STR_NUM; i++)
    out.write(origData[i], STR_LEN);
  out.close();
  ifstream in("Poem.bin", ios::in | ios::binary);
  assure(in, "Poem.bin");
  in.read(readData[0], STR_LEN);
  assert(strcmp(readData[0], "Hickory dickory dus. . .")
    == 0);
  // Seek -STR_LEN bytes from the end of file
  in.seekg(-STR_LEN, ios::end);
  in.read(readData[1], STR_LEN);
  assert(strcmp(readData[1], "There's plenty more for us!")
    == 0);
  // Absolute seek (like using operator[] with a file)
  in.seekg(3 * STR_LEN);
  in.read(readData[2], STR_LEN);
  assert(strcmp(readData[2], "That's just too bad.") == 0);
  // Seek backwards from current position
  in.seekg(-STR_LEN * 2, ios::cur);
  in.read(readData[3], STR_LEN);
  assert(strcmp(readData[3], "Well, if you have,") == 0);
  // Seek from the begining of the file
  in.seekg(1 * STR_LEN, ios::beg);
  in.read(readData[4], STR_LEN);
  assert(strcmp(readData[4], "Are you tired of C++?")
    == 0);
} ///:~
```

这个程序使用二进制输出流写一首诗到文件中。重新打开**ifstream**文件后，使用**seekg( )**“获取指针”位置。正如读者所看到的，文件指针可以从文件头、文件尾、或从文件当前位置

177

进行搜索。显然，从文件头开始移动指针需要提供一个正数做参数，而从文件尾移动指针需要提供一个负数做参数。

178 既然已经熟悉了**streambuf**类以及如何在流中定位，读者就能理解创建一个既能够读文件又能够写文件的流对象的另一种方法了（不使用**fstream**对象）。下面的代码首先使用某些标记创建一个**ifstream**，这些标记表明它既能用于文件输入又能用于文件输出。因为不能对**ifstream**对象进行写入操作，所以需要创建一个具有内置流缓冲区的**ostream**对象：

```
ifstream in("filename", ios::in | ios::out);
ostream out(in.rdbuf());
```

读者也许想知道向这两个对象之一写入数据时会发生什么情况。举例如下：

```
//: C04:Iofile.cpp
// Reading & writing one file.
#include <fstream>
#include <iostream>
#include "../require.h"
using namespace std;

int main() {
  ifstream in("Iofile.cpp");
  assure(in, "Iofile.cpp");
  ofstream out("Iofile.out");
  assure(out, "Iofile.out");
  out << in.rdbuf(); // Copy file
  in.close();
  out.close();
  // Open for reading and writing:
  ifstream in2("Iofile.out", ios::in | ios::out);
  assure(in2, "Iofile.out");
  ostream out2(in2.rdbuf());
  cout << in2.rdbuf();  // Print whole file
  out2 << "Where does this end up?";
  out2.seekp(0, ios::beg);
  out2 << "And what about this?";
  in2.seekg(0, ios::beg);
  cout << in2.rdbuf();
} ///:~
```

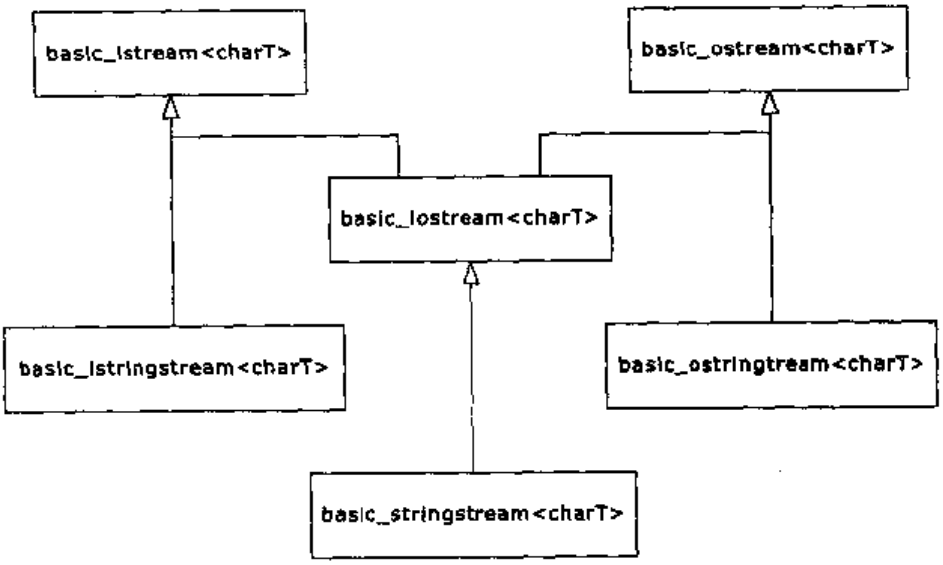
前5行把这个程序的源代码拷贝到一个叫做**iofile.out**的文件，接着关闭该文件，这样一来，就为读者提供一个安全的文本文件。然后，使用前面介绍的技术创建两个对象，以便对同一个
179 文件进行读和写操作。在语句**cout<<in2.rdbuf( )**中，可以看到“读”指针被初始化为指向文件头。“写”指针被设置为指向文件尾，因为文本信息“Where does this end up?”是追加到文件中的。然而，如果写指针通过调用函数**seekp( )**被移动到指向文件头，新写入的文本将覆盖原来的文本。调用函数**seekg( )**把读指针移回到文件头，就可以分别看到两次写操作后所显示的文件中的内容。当**out2**超出作用域范围后，系统调用析构函数，使文件自动保存和关闭。

4.7 字符串输入输出流

字符串输入输出流类直接对内存而不是对文件和标准输出（设备）进行操作。它使用与**cin**及**cout**相同的读取和格式化函数来操纵内存中的数据。在早期的计算机中，内存储器是计算机的核心，所以这种功能的类型常常称为内核格式化（in-core formatting）。

字符串流的类名模仿文件流的类名。如果需要创建一个字符串流，并从这个流读取字符，可以创建**istringstream**。如果需要写字符到字符串流，可以创建**ostringstream**。所有字

符串流类的声明都包含在标准头文件**<sstream>**中。照例，有一些类模板被添加到输入输出流类层次结构中，如下图所示:



180

4.7.1 输入字符串流

为了使用流操作从一个字符串中读取数据，需要创建**istringstream**对象并用字符串初始化这个对象。下面的程序说明了如何使用**istringstream**对象:

```
//: C04:Istring.cpp
// Input string streams.
#include <cassert>
#include <cmath>  // For fabs()
#include <iostream>
#include <limits> // For epsilon()
#include <sstream>
#include <string>
using namespace std;

int main() {
  istringstream s("47 1.414 This is a test");
  int i;
  double f;
  s >> i >> f; // Whitespace-delimited input
  assert(i == 47);
  double relerr = (fabs(f) - 1.414) / 1.414;
  assert(relerr <= numeric_limits<double>::epsilon());
  string buf2;
  s >> buf2;
  assert(buf2 == "This");
  cout << s.rdbuf(); // " is a test"
} ///:~
```

181

读者可以看到，这是一种将字符串转换为特定类型值的方法，比标准C中的库函数如**atof( )**和**atoi( )**更灵活、更通用，尽管后者对单个字符串的转换效率更高。

在表达式**s >> i >> f**中，字符串中的第1个数字摘录到**i**，第2个数字输出到**f**。因为它依赖于数据的类型进行摘录，所以不是“以空格作为首选界定符的字符集”。例如，当字符串为“**1.414 47 This is a test**”时，则**i**提取的值为**1**，因为输入过程会在小数点处停止。**f**提取的值为**0.414**。如果想把一个浮点数分成整数部分和小数部分，则这种方法就很有效。而在其他

情况下这似乎是错误的做法。第2个**assert( )**函数判断读出的数据是否发生了错误；这样做比简单地比较两个浮点数是否相等更好。函数**epsilon( )**返回一个在**<limits>**中定义的常量，代表双精度数字的机器误差，这是在比较两个**double**型数据时所能得到的最小误差[⊖]。

或许读者已经猜到**buf2**中的内容不等于剩下的串，只是等于下一个空格字符之前的单词。一般来说，当知道输入流中数据的顺序并把它转换成除字符串之外的数据类型时，最好使用输入输出流提取符。然而，如果想一次性提取一个串中剩余的部分并把它输出到另一个输入输出流中，可以使用程序中所示的函数**rdbuf( )**。

为测试本章早些时候提到的**Date**提取符，在下面的测试程序中使用输入字符串流：

182

```
//: C04:DateIOTest.cpp
//{L} ../C02/Date
#include <iostream>
#include <sstream>
#include "../C02/Date.h"
using namespace std;

void testDate(const string& s) {
  istringstream os(s);
  Date d;
  os >> d;
  if(os)
    cout << d << endl;
  else
    cout << "input error with \"" << s << "\"" << endl;
}

int main() {
  testDate("08-10-2003");
  testDate("8-10-2003");
  testDate("08 - 10 - 2003");
  testDate("A-10-2003");
  testDate("08%10/2003");
} ///:~
```

在**main( )**函数中，将每个字符串作为引用参数依次调用函数**testDate( )**，函数**testDate( )**把参数封装到**istringstream**对象中，这样就可以测试为**Date**对象所编写的流提取符了。函数**testDate( )**也对插入符**operator<<( )**进行了测试。

4.7.2 输出字符串流

为了创建输出字符串流，可以构造**ostringstream**对象，这个类对象可以管理动态字符缓冲区，缓冲区存放要插入的任何字符串。为了将输出结果格式化为**string**对象，可以调用成员函数**str( )**，举例如下：

```
//: C04:Ostring.cpp {RunByHand}
// Illustrates ostringstream.
#include <iostream>
#include <sstream>
#include <string>
using namespace std;
int main() {
  cout << "type an int, a float and a string: ";
```

183

⊖ 关于机器双精度数字和浮点数的一般性计算，请参考“标准C库，第三部分”，C/C++用户周刊，1995年3月。也可查阅网页：www.freshsources.com/1995006a.htm。

```
  int i;
  float f;
  cin >> i >> f;
  cin >> ws; // Throw away white space
  string stuff;
  getline(cin, stuff); // Get rest of the line
  ostringstream os;
  os << "integer = " << i << endl;
  os << "float = " << f << endl;
  os << "string = " << stuff << endl;
  string result = os.str();
  cout << result << endl;
} ///:~
```

这个例子中读取**int**和**float**的语句与**Istring.cpp**中的语句相似。下面是一个输出结果的例子（黑体部分为键盘输入）。

```
type an int, a float and a string: 10 20.5 the end
integer = 10
float = 20.5
string = the end
```

读者可以看到，就像其他的输出流，输出字节到**ostringstream**对象可以使用一般的格式化工具，如运算符**<<**和**endl**。每次调用**str( )**函数都会返回一个新的**string**对象，所以字符串流内置的**stringbuf**对象不会被破坏。

在第3章中，给出了一个程序**HTMLStripper.cpp**，它的作用是删除文本文件中的所有HTML标记及特殊字符。这里给出一种使用字符串流的更优美的版本：

```
//: C04:HTMLStripper2.cpp {RunByHand}
//{L} ../C03/ReplaceAll
// Filter to remove html tags and markers.
#include <cstddef>
#include <cstdlib>
#include <fstream>
#include <iostream>
#include <sstream>
#include <stdexcept>
#include <string>
#include "../C03/ReplaceAll.h"
#include "../require.h"
using namespace std;

string& stripHTMLTags(string& s) throw(runtime_error) {
  size_t leftPos;
  while((leftPos = s.find('<')) != string::npos) {
    size_t rightPos = s.find('>', leftPos+1);
    if(rightPos == string::npos) {
      ostringstream msg;
      msg << "Incomplete HTML tag starting in position "
          << leftPos;
      throw runtime_error(msg.str());
    }
    s.erase(leftPos, rightPos - leftPos + 1);
  }
  // Remove all special HTML characters
  replaceAll(s, "&lt;", "<");
  replaceAll(s, "&gt;", ">");
  replaceAll(s, "&amp;", "&");
  replaceAll(s, " ", " ");
  // Etc...
  return s;
```

184

```
}

int main(int argc, char* argv[]) {
  requireArgs(argc, 1,
    "usage: HTMLStripper2 InputFile");
  ifstream in(argv[1]);
  assure(in, argv[1]);
  // Read entire file into string; then strip
  ostringstream ss;
  ss << in.rdbuf();
  try {
    string s = ss.str();
    cout << stripHTMLTags(s) << endl;
    return EXIT_SUCCESS;
  } catch(runtime_error& x) {
    cout << x.what() << endl;
    return EXIT_FAILURE;
  }
} ///:~
```

在这个程序中，通过把文件流的**rdbuf( )** 调用放到一个**ostringstream**对象中，最终将整个文件读到一个字符串中。这样搜索HTML文件中的界定符对并将其删除就很容易了，也不必像第3章中的前一版本那样考虑跨越多行的标记。

185

下面的例子说明了如何使用双向（即，读/写）字符串流：

```
//: C04:StringSeeking.cpp {-bor}{-dmc}
// Reads and writes a string stream.
#include <cassert>
#include <sstream>
#include <string>
using namespace std;

int main() {
  string text = "We will hook no fish";
  stringstream ss(text);
  ss.seekp(0, ios::end);
  ss << " before its time.";
  assert(ss.str() ==
    "We will hook no fish before its time.");
  // Change "hook" to "ship"
  ss.seekg(8, ios::beg);
  string word;
  ss >> word;
  assert(word == "hook");
  ss.seekp(8, ios::beg);
  ss << "ship";
  // Change "fish" to "code"
  ss.seekg(16, ios::beg);
  ss >> word;
  assert(word == "fish");
  ss.seekp(16, ios::beg);
  ss << "code";
  assert(ss.str() ==
    "We will ship no code before its time.");
  ss.str("A horse of a different color.");
  assert(ss.str() == "A horse of a different color.");
} ///:~
```

同前面一样，通过调用**seekp( )** 移动写指针，调用**seekg( )**重定位读指针。字符串流比文件流使用起来更容易，因为任何时候都可以从流的读状态切换到写状态或从写状态切换到读

状态，但是在这个例子中没有引入这方面的内容。在这个过程中，不需要重定位读指针或写指针、或刷新流。在这个程序中也说明了重载的**str( )** 函数，它用一个新字符串替换流中内置的**stringbuf**。 186

4.8 输出流的格式化

输入输出流的设计目标是使用户易于移动和/或格式化字符。如果不能用C语言中提供的**printf( )**及相关函数完成大部分格式化操作，输入输出流将不会有多大用处。在这部分，读者可以学习输入输出流类所有的格式化输出函数，之后可以按照自己的意愿格式化输出数据。

开始学习输入输出流的格式化输出函数的时候可能会引起混淆，因为存在不只一种控制格式化输出的方法：通过成员函数和运算符。之后可能遇到的会引起混淆的地方有：设置状态标志来控制格式化的一般成员函数，如左对齐或右对齐；在十六进制表示法中使用大写字母；始终使用小数点表示浮点数的值等。另一方面，个别的成员函数用来设置和读取填充字符、域宽和精度的值。

为对这些函数进行分类，首先应检查输入输出流的内部格式化数据，以及能够修改这些数据的成员函数。（如果想这样做的话，用成员函数就可以进行所有的操作）。操作符将单独讨论。

4.8.1 格式化标志

类**ios**包含一些数据成员，用于存储与流有关的所有格式化信息。其中一些数据存储在变量中，其值有一个变动范围：浮点数精度、输出域宽和用于填充输出的字符（一般为空格）。有关格式化的其他信息由格式化标志决定，为了节省空间，通常多个标志组合在一起使用。成员函数**ios::flags( )**可以得到格式化标志所代表的值，这个函数没有参数，函数的返回值为包含当前格式化标志的**fmtflags**（一般被认为是**long**的同义词）对象。其他函数用于改变格式化标志并返回格式化标志的原有值。

```
fmtflags ios::flags(fmtflags newflags);
fmtflags ios::setf(fmtflags ored_flag);
fmtflags ios::unsetf(fmtflags clear_flag);
fmtflags ios::setf(fmtflags bits, fmtflags field);
```

187

第1个函数强制改变程序需要的所有标志。但更多的时候，是使用其他3个函数，每次改变一个标志。

函数**setf( )**的使用似乎会引起一些混淆。要想知道该使用那个版本的重载函数，就需要知道将要改变的格式化标志的类型。有两种类型的格式化标志：一类是仅被设置为开或关的开/关标志；另一类标志和其他的标志联合使用。开/关标志最简单且容易理解，使用**setf(fmtflags)** 函数打开标志，使用**unsetf(fmtflags)** 函数关闭标志。这些标志在下面的表中说明：

开/关标志	作　用
ios::skipws	跳过空格（输入流的默认情况。）
ios::showbase	打印整型值时指出数字的基数（比如，为**dec**、**oct**或**hex**） **showbase**标志处于开状态时，输入流也能识别前缀基数
ios::showpoint	显示浮点值的小数点并截断数字末尾的零
ios::uppercase	显示十六进制值时使用大写**A~F**，显示科学计数中的**E**
ios::showpos	显示正数前的加号（+）
ios::unitbuf	“单元缓冲区”（unit buffering）。每次插入后刷新流

188 例如，为了在**cout**中显示正号，可以使用语句**cout.setf(ios::showpos)**。如果使用了**cout.unsetf(ios::showpos)**，则不再显示正号。

unitbuf标志控制着*单元缓冲区*（unit buffering），这意味着每次在输出流中插入数据后都会立即刷新该输出流。这对于错误跟踪很方便，当程序崩溃时，数据仍然会保存到日志文件中。下面的程序演示了单元缓冲区：

```
//: C04:Unitbuf.cpp {RunByHand}
#include <cstdlib>  // For abort()
#include <fstream>
using namespace std;

int main() {
  ofstream out("log.txt");
  out.setf(ios::unitbuf);
  out << "one" << endl;
  out << "two" << endl;
  abort();
} ///:~
```

在插入任何数据到流对象之前都需要打开单元缓冲区。当把**setf( )**注释掉（comment out）时，某个特殊的编译器仅写入了一个字母**'o'**到文件**log.txt**。而使用单元缓冲区则不会丢失任何数据。

默认情况下，标准错误输出流**cerr**的单元缓冲区处于打开状态。使用单元缓冲会导致大量的系统开销，所以如果程序中频繁使用输出流，则不要使用单元缓冲区，除非不需要考虑程序的执行效率。

4.8.2 格式化域

第2类格式化标志是分组使用的。一次只能设置这些标志中的一个，就像老式汽车收音机上的按钮一样——当按下其中一个按钮时，其他按钮会弹起来。遗憾的是，标志的这种互斥设置不能自动地完成，编程人员必须注意将要设置什么标志，以防错误地使用了**setf( )**函数。例如，对于每一种基数（number base）都有相应的标志：十六进制（hexadecimal）、十进制（decimal）和八进制（octal）。这些标志统称为**ios::basefield**。如果已经设置了**ios::dec**标志这时又调用**setf(ios::hex)**，会设置**ios::hex**标志却不会清除**ios::dec**标志位，从而导致
189 不确定的行为。作为前一函数的替代，可以调用第2种形式的**setf( )**函数**setf(ios::hex, ios::basefield)**。这个函数首先清除**ios::basefield**域中的所有位，然后设置**ios::hex**标志。这种形式的**setf( )**在设置一种标志的时候会确保同组的其他标志被“清除”。**ios::hex**操纵算子自动地完成所有工作，因此不需要关心类的内部实现细节，甚至不需要关心**ios::basefield**是一个二进制标志的集合。接下来，读者将会看到多种操纵算子，在所有使用**setf( )**的地方提供相同的功能。

下面是这些标志及其作用：

ios::basefield	作　用
ios::dec	使整型数值的基数为10（十进制形式）（默认的基数——没有前缀）
ios::hex	使整型数值的基数为16（十六进制形式）
ios::oct	使整形数值的基数为8（八进制形式）

ios::floatfield	作 用
ios::scientific	以科学计数形式显示浮点数。精度域指示小数点后数字的位数
ios::fixed	以固定格式显示浮点数。精度域指示小数点后数字的位数
"automatic"（不设置任何标志位）	精度域指示所有有效数字的位数

ios::adjustfield	作 用
ios::left	使数值左对齐，使用填充字符填充右边空位
ios::right	使数值右对齐。使用填充字符填充左边空位。此为默认对齐方式
ios::internal	把填充字符放到前导正负号或基数指示符之后，数值之前（换句话说，如果输出正负号，则正负号左对齐，而数值右对齐。）

190

4.8.3 宽度、填充和精度设置

用来控制输出域（字段）的宽度、填补输出域的填充字符和显示浮点数精度的内部变量，由与其同名的成员函数进行读写。

函 数	作 用
int ios::width( )	返回当前宽度。默认为0。用于插入符和提取符
int ios::width(int n)	设定宽度，并返回先前的宽度
int ios::fill( )	返回当前填充字符。默认为空格符
int ios::fill(int n)	设定填充字符，并返回先前的填充字符
int ios::precision( )	返回当前浮点数精度。默认情况下，精确到小数点后6位
int ios::precision(int n)	设定浮点数精度，返回先前的精度。"precision（精度）"的含义参看**ios::floatfield**表

191

fill和**precision**的值是相当直观的，但**width**的值就需要解释一下。当宽度为0时，在流中插入一个值的结果是，生成表示这个值所需的最少字符数。宽度为正的意思是，在流中插入一个数，至少会产生宽度所规定数量的字符；如果插入字符的个数小于宽度值，则用填充字符填充空余位置。然而，输出的值永远都不会被截断，所以如果试图在宽度为2时打印123，仍会得到123。宽度只能指定输出字符的最小数目；没有设定输出字符最大数目的方法。

宽度还有一个明显的不同，因为每个插入符或提取符都可能受它的值的影响，所以它被每个插入符或提取符隐含自动重置为0。它不是一个真正的状态变量，而是插入符和提取符的隐含参数。如果想得到固定不变的宽度，需要在每次使用插入符或提取符之后调用**width( )**。

4.8.4 一个完整的例子

为了使读者明白如何使用前面讨论过的所有函数，这里给出一个调用了所有这些函数的例子：

```
//: C04:Format.cpp
// Formatting Functions.
#include <fstream>
#include <iostream>
#include "../require.h"
using namespace std;
#define D(A) T << #A << endl; A

int main() {
  ofstream T("format.out");
  assure(T);
  D(int i = 47;)
```

```
D(float f = 2300114.414159;)
const char* s = "Is there any more?";
D(T.setf(ios::unitbuf);)
D(T.setf(ios::showbase);)
D(T.setf(ios::uppercase | ios::showpos);)
D(T << i << endl;) // Default is dec
D(T.setf(ios::hex, ios::basefield);)
D(T << i << endl;)
D(T.setf(ios::oct, ios::basefield);)
D(T << i << endl;)
D(T.unsetf(ios::showbase);)
D(T.setf(ios::dec, ios::basefield);)
D(T.setf(ios::left, ios::adjustfield);)
D(T.fill('0');)
D(T << "fill char: " << T.fill() << endl;)
D(T.width(10);)
T << i << endl;
D(T.setf(ios::right, ios::adjustfield);)
D(T.width(10);)
T << i << endl;
D(T.setf(ios::internal, ios::adjustfield);)
D(T.width(10);)
T << i << endl;
D(T << i << endl;) // Without width(10)

D(T.unsetf(ios::showpos);)
D(T.setf(ios::showpoint);)
D(T << "prec = " << T.precision() << endl;)
D(T.setf(ios::scientific, ios::floatfield);)
D(T << endl << f << endl;)
D(T.unsetf(ios::uppercase);)
D(T << endl << f << endl;)
D(T.setf(ios::fixed, ios::floatfield);)
D(T << f << endl;)
D(T.precision(20);)
D(T << "prec = " << T.precision() << endl;)
D(T << endl << f << endl;)
D(T.setf(ios::scientific, ios::floatfield);)
D(T << endl << f << endl;)
D(T.setf(ios::fixed, ios::floatfield);)
D(T << f << endl;)

D(T.width(10);)
T << s << endl;
D(T.width(40);)
T << s << endl;
D(T.setf(ios::left, ios::adjustfield);)
D(T.width(40);)
T << s << endl;
} ///:~
```

这个例子中用了一种技巧来创建一个跟踪文件，以监视程序执行时发生了什么事情。宏**D(a)**用预处理器（程序）把**a**转换成串并显示。然后对**a**进行重复迭代，所以语句顺次执行。宏把所有信息输出到跟踪文件**T**。程序的输出为:

```
int i = 47;
float f = 2300114.414159;
T.setf(ios::unitbuf);
T.setf(ios::showbase);
T.setf(ios::uppercase | ios::showpos);
T << i << endl;
+47
```

```
T.setf(ios::hex, ios::basefield);
T << i << endl;
0X2F
T.setf(ios::oct, ios::basefield);
T << i << endl;
057
T.unsetf(ios::showbase);
T.setf(ios::dec, ios::basefield);
T.setf(ios::left, ios::adjustfield);
T.fill('0');
T << "fill char: " << T.fill() << endl;
fill char: 0
T.width(10);
+470000000
T.setf(ios::right, ios::adjustfield);
T.width(10);
0000000+47
T.setf(ios::internal, ios::adjustfield);
T.width(10);
+000000047
T << i << endl;
+47
T.unsetf(ios::showpos);
T.setf(ios::showpoint);
T << "prec = " << T.precision() << endl;
prec = 6
T.setf(ios::scientific, ios::floatfield);
T << endl << f << endl;

2.300114E+06
T.unsetf(ios::uppercase);
T << endl << f << endl;

2.300114e+06
T.setf(ios::fixed, ios::floatfield);
T << f << endl;
2300114.500000
T.precision(20);
T << "prec = " << T.precision() << endl;
prec = 20
T << endl << f << endl;

2300114.50000000000000000000
T.setf(ios::scientific, ios::floatfield);
T << endl << f << endl;

2.30011450000000000000e+06
T.setf(ios::fixed, ios::floatfield);
T << f << endl;
2300114.50000000000000000000
T.width(10);
Is there any more?
T.width(40);
0000000000000000000000Is there any more?
T.setf(ios::left, ios::adjustfield);
T.width(40);
Is there any more?0000000000000000000000
```

194

研究这个输出文件可以使读者对输入输出流的格式化成员函数理解得更清晰、明确。

4.9 操纵算子

从前面的程序可以看出，调用成员函数进行流的格式化操作有些冗长乏味。为使读操作和写操作更容易，C++语言提供了操纵算子的集合，这些操纵算子可以实现与相应的成员函数相同的功能。操纵算子使用起来更方便，因为可以在一个表达式中插入它们；不需要单独的函数调用语句。

195 操纵算子改变流的状态而不是（或同时）处理数据。例如，当在一个输出表达式中插入**endl**时，不但在流中插入了一个换行符，而且刷新了流（即，将存储在流内部缓冲区中但还未真正输出的所有字符输出）。也可像这样刷新流：

```
cout << flush;
```

这引起调用成员函数**flush( )**的副作用，即

```
cout.flush();
```

（没在流中插入任何东西。）其他的基本操纵算子改变数的基数为**oct**（八进制）、**dec**（十进制）或**hex**（十六进制）：

```
cout << hex << "0x" << i << endl;
```

既然这样，以后的数字输出将继续保持为十六进制模式，直至修改它。通过在输出流中插入**dec**或**oct**来改变这种模式。

也存在一种针对提取操作的操纵算子，它可以“吃掉”空格：

```
cin >> ws;
```

不带参数的操纵算子在头文件**<iostream>**中定义。包括**dec**、**oct**和**hex**，分别对应于**setf(ios::dec, ios::basefield)**、**setf(ios::oct, ios::basefield)** 和**setf(ios::hex, ios::basefield)**，但前者更简洁。在头文件**<iostream>**中也包含**ws**、**endl**和**flush**以及在这里说明的其他操纵算子：

操纵算子	作　用
showbase	输出整型数时显示数字的基数（**dec**、**oct**或**hex**）
noshowbase	
showpos	显示正数前面的正号（+）
noshowpos	
uppercase	用大写的A~F显示十六进制数，在科学计数型数字中使用大写的E
nouppercase	
showpoint	显示浮点数的十进制小数点和尾部的0
noshowpoint	
skipws	跳过输入中的空格
noskipws	
left	左对齐，向右边填充字符
right	右对齐，向左边填充字符
internal	把填充字符放到引导符或基数指示符和数值之间
scientific	指出浮点数的优先输出格式（科学计数法vs.定点小数）
fixed	

196

4.9.1 带参数的操纵算子

有6个标准的带参数的操纵算子，如**setw( )**。这些操纵算子在头文件**<iomanip>**中定义，

下表对这些操纵算子做了总结:

操纵算子	作 用
setiosflags(fmtflags n)	其作用相当于调用函数**setf(n)**。设置后会一直起作用，直到下一次设置将其改变，如调用**ios::setf( )**
resetiosflags(fmtflags n)	清除由**n**代表的格式化标志。本次设置一直有效，直到进行下一次设置，如调用**ios::unsetf( )**
setbase(base n)	将数的基数设为**n**，**n**的取值为10、8或16。(如传递给**n**的值为其他值则自动置为0) 如果**n**的值为0，则输出数的基数为10，但输入使用C语言惯用法：10为10，010为8，0xf为15。对输出流也可使用**dec**、**oct**和**hex**
setfill(char n)	将填充字符设为**n**，就像**ios::fill( )**
setprecision(int n)	将数字精度设为**n**，就像**ios::precision( )**
setw(int n)	将宽度设为**n**，就像**ios::width( )**

197

如果程序中大量使用流格式化操作，则可以发现使用操纵算子代替调用流类成员函数可以简化代码。这里的例子用操纵算子重写了前面的程序。(程序中删除了**D( )**宏，使得代码更容易阅读。)

```
//: C04:Manips.cpp
// Format.cpp using manipulators.
#include <fstream>
#include <iomanip>
#include <iostream>
using namespace std;

int main() {
  ofstream trc("trace.out");
  int i = 47;
  float f = 2300114.414159;
  char* s = "Is there any more?";

  trc << setiosflags(ios::unitbuf
           | ios::showbase | ios::uppercase
           | ios::showpos);
  trc << i << endl;
  trc << hex << i << endl
      << oct << i << endl;
  trc.setf(ios::left, ios::adjustfield);
  trc << resetiosflags(ios::showbase)
      << dec << setfill('0');
  trc << "fill char: " << trc.fill() << endl;
  trc << setw(10) << i << endl;
  trc.setf(ios::right, ios::adjustfield);
  trc << setw(10) << i << endl;
  trc.setf(ios::internal, ios::adjustfield);
  trc << setw(10) << i << endl;
  trc << i << endl; // Without setw(10)

  trc << resetiosflags(ios::showpos)
      << setiosflags(ios::showpoint)
      << "prec = " << trc.precision() << endl;
  trc.setf(ios::scientific, ios::floatfield);
  trc << f << resetiosflags(ios::uppercase) << endl;
  trc.setf(ios::fixed, ios::floatfield);
  trc << f << endl;
  trc << f << endl;
```

198

```
  trc << setprecision(20);
  trc << "prec = " << trc.precision() << endl;
  trc << f << endl;
  trc.setf(ios::scientific, ios::floatfield);
  trc << f << endl;
  trc.setf(ios::fixed, ios::floatfield);
  trc << f << endl;
  trc << f << endl;

  trc << setw(10) << s << endl;
  trc << setw(40) << s << endl;
  trc.setf(ios::left, ios::adjustfield);
  trc << setw(40) << s << endl;
} ///:~
```

读者可以看到，在这个程序中有多处地方，将多条语句合并到一条链式表达的插入语句中。注意对函数**setiosflags( )**的调用，其参数为几个格式化标志的按位或。前一例子中相同的功能由**setf( )**和**unsetf( )**实现。

对输出流使用函数**setw( )**时，输出表达式被格式化输出到一个临时串，格式化串的长度与传递给**setw( )**的参数相比较，根据比较结果决定是否需要用当前填充字符填补空余位置。换言之，**setw( )**影响格式化输出操作的结果字符串。同样，对输入流使用**setw( )**函数进行设置，只在读字符串时有意义，下面的例子很清楚地说明了这一点：

199

```
//: C04:InputWidth.cpp
// Shows limitations of setw with input.
#include <cassert>
#include <cmath>
#include <iomanip>
#include <limits>
#include <sstream>
#include <string>
using namespace std;

int main() {
  istringstream is("one 2.34 five");
  string temp;
  is >> setw(2) >> temp;
  assert(temp == "on");
  is >> setw(2) >> temp;
  assert(temp == "e");
  double x;
  is >> setw(2) >> x;
  double relerr = fabs(x - 2.34) / x;
  assert(relerr <= numeric_limits<double>::epsilon());
} ///:~
```

如果试图读一个字符串，函数**setw( )**准确地控制着提取字符的数目，读取过程遇到小数点时结束。第1次提取获得了两个字符，而第2次提取仅获得了一个字符，尽管将读取数目设置为两个。这是因为**operator>>( )**使用空格作为界定符（除非关闭**skipws**标志）。然而，当试图读取一个数字时，例如读取**x**，不能用**setw( )**来限定读取字符的个数。对于输入流，**setw( )**只能用于字符串的提取。

4.9.2 创建操纵算子

有时，读者喜欢创建自己的操纵算子，而且创建过程也相当简单。不带参数（零参数，zero-argument）的操纵算子是仅一个函数，例如**endl**，它只是以**ostream**对象的引用为参数，

返回值为一个**ostream**对象的引用的函数。**endl**的声明为:

```
ostream& endl(ostream&);
```

现在，当执行语句:

200

```
cout << "howdy" << endl;
```

时，**endl**将产生该函数的地址。编译器会问，“是否存在一个函数，它以一个函数的地址作为参数？”头文件**<iostream>**中的预定义函数负责这项工作；这些函数称为应用算子(applicator)（因为它们将一个函数应用到流类）。应用算子调用它的函数参数，并传递**ostream**对象作为自己的参数。在这里，不需要知道应用算子是如何创建操纵算子的；仅需知道操纵算子的存在。这里有一个（简化的）**ostream**应用算子的代码:

```
ostream& ostream::operator<<(ostream& (*pf)(ostream&)) {
  return pf(*this);
}
```

实际的定义因涉及模板会更复杂一些，这行代码说明了这项技术。当一个函数，如 ***pf**(以流作为参数，返回流的引用)，被插入到一个流中时，调用上面的应用算子函数，之后执行**pf**指针指向的函数。**ios_base**、**basic_ios**、**basic_ostream**和**basic_istream**的应用算子在标准C++库中预定义。

这里有一个比较简明的例子解释了上面所描述的过程，例子中创建了一个叫做**nl**的操纵算子，它的作用是在流中插入换行符（也就是说，这个操纵算子不刷新流，不像**endl**那样):

```
//: C04:nl.cpp
// Creating a manipulator.
#include <iostream>
using namespace std;

ostream& nl(ostream& os) {
  return os << '\n';
}

int main() {
  cout << "newlines" << nl << "between" << nl
       << "each" << nl << "word" << nl;
} ///:~
```

当插入**nl**到一个输出流如**cout**时，调用顺序为:

201

```
cout.operator<<(nl) → nl(cout)
```

表达式

```
os << '\n';
```

在函数**nl( )**内部调用**ostream::operator(char)**，它返回一个流对象，这个流对象最终从**nl( )**返回。[⊖]

4.9.3 效用算子

读者已经看到，零参数的操纵算子很容易创建。但是如何创建带参数的操纵算子呢？如果研究头文件**<iomanip>**，就会发现一种称做**smanip**的类型，它返回带参数的操纵算子。读者也许试图仿照**smanip**定义自己的带参数的操纵算子，但是请不要这样做。因为类型

⊖ 在把**nl**定义到头文件之前，使之成为**inline**(内联)函数。

smanip是依赖于系统实现的，所以不具备可移植性。幸运的是，可以使用由Jerry Schwarz提出的叫做*效用算子*（effector）的技术直接定义独立于机器实现的操纵算子。[⊖]一个效用算子是一个简单的类，该类的构造函数可以格式化一个字符串，这个字符串描述了读者希望的操作，并将这个字符串连同重载的**operator<<** 一起插入到流中。这里有一个含有两个效用算子的程序例子。第1个效用算子输出一个截断的字符串，第2个效用算子以二进制方式输出一个数。

```
//: C04:Effector.cpp
// Jerry Schwarz's "effectors."
#include <cassert>
#include <limits>  // For max()
#include <sstream>
#include <string>
using namespace std;

// Put out a prefix of a string:
class Fixw {
  string str;
public:
  Fixw(const string& s, int width) : str(s, 0, width) {}
  friend ostream& operator<<(ostream& os, const Fixw& fw) {
    return os << fw.str;
  }
};

// Print a number in binary:
typedef unsigned long ulong;

class Bin {
  ulong n;
public:
  Bin(ulong nn) { n = nn; }
  friend ostream& operator<<(ostream& os, const Bin& b) {
    const ulong ULMAX = numeric_limits<ulong>::max();
    ulong bit = ~(ULMAX >> 1); // Top bit set
    while(bit) {
      os << (b.n & bit ? '1' : '0');
      bit >>= 1;
    }
    return os;
  }
};

int main() {
  string words = "Things that make us happy, make us wise";
  for(int i = words.size(); --i >= 0;) {
    ostringstream s;
    s << Fixw(words, i);
    assert(s.str() == words.substr(0, i));
  }
  ostringstream xs, ys;
  xs << Bin(0xCAFEBABEUL);
  assert(xs.str() ==
    "1100""1010""1111""1110""1011""1010""1011""1110");
  ys << Bin(0x76543210UL);
  assert(ys.str() ==
    "0111""0110""0101""0100""0011""0010""0001""0000");
} ///:~
```

202

⊖ Jerry Schwarz是输入输出流的设计者。

类**Fixw**的构造函数创建**char***参数的一个被截短的拷贝，由析构函数释放创建拷贝时分配的内存。重载运算符**operator<<**把第2个参数的内容即**Fixw**对象插入到第1个参数即**ostream**对象中，然后返回**ostream**对象，所以它能够在一个链式表达式中使用。当在一个表达式中使用**Fixw**时，如下所示: 203

```
cout << Fixw(string, i) << endl;
```

该语句调用类**Fixw**的构造函数创建一个**Fixw**临时对象，这个临时对象被传给**operator<<**。它的作用相当于带参数的操纵算子。临时**Fixw**对象在这条语句结束前将一直存在。

Bin效用算子依赖这样一个事实: 右移无符号数字时在二进制数的高位补零。可以使用**numeric_limits<unsigned long>::max( )**（产生**unsigned long**数的最大值，在标准头文件**<limits>**中定义）利用高位集产生一个值，并且这个值从头至尾进行位移用来询问被测试的数字（通过右移），依次屏蔽每一位。为了具有可读性，现在已经将代码中的字符串文字并列；编译器会将分开的这些字符串合并到一个字符串中。

这项技术历来存在的问题是: 一旦为**char***创建了**Fixw**对象或为**unsigned long**创建了**Bin**对象，就不允许再为**Fixw**类或**Bin**类创建不同的类对象。然而，使用名字空间后这个问题就不存在了。效用算子和操纵算子并不等同，尽管它们可以用来解决相同的问题。如果发现某个问题使用效用算子不能解决，就需要克服操纵算子的复杂性。

4.10 输入输出流程序举例

这部分将介绍几个例子，这些例子使用了本章中讲述的知识。尽管存在很多处理字节的工具（UNIX下的流编辑器，如**sed**和**awk**或许是最常用的，而一个文本编辑器也属于此类），但一般来说这些工具有一些局限性。**sed**和**awk**可能比较慢，而且只能处理前向序列里的行，并且文本编辑器通常需要人机交互，或至少学习一门专用的宏语言。使用输入输出流编写的程序没有这些限制: 具有快速性、可移植性和灵活性。

4.10.1 维护类库的源代码 204

一般来说，当创建一个类时，读者往往会想到有关库的术语: 类的声明在头文件**Name.h**中定义，而类的成员函数的实现在文件**Name.cpp**中建立。这些文件有特殊的需求: 一个特殊的编码标准（这里的程序使用本教材中的代码格式），而且头文件中的预处理语句能避免类的重复声明。（重复声明使编译器不知道哪个类是程序真正需要的。这些类可能不同，所以编译器会输出一个错误信息。）

这个例子创建一个新的头文件/实现文件对，或修改已存在的一个头文件/实现文件对。如果文件已经存在，则对文件进行检查并修改，如果文件不存在则使用合适的格式创建该文件。

```
//: C04:Cppcheck.cpp
// Configures .h & .cpp files to conform to style
// standard. Tests existing files for conformance.
#include <fstream>
#include <sstream>
#include <string>
#include <cstddef>
#include "../require.h"
using namespace std;

bool startsWith(const string& base, const string& key) {
  return base.compare(0, key.size(), key) == 0;
}
```

```
void cppCheck(string fileName) {
  enum bufs { BASE, HEADER, IMPLEMENT, HLINE1, GUARD1,
    GUARD2, GUARD3, CPPLINE1, INCLUDE, BUFNUM };
  string part[BUFNUM];
  part[BASE] = fileName;
  // Find any '.' in the string:
  size_t loc = part[BASE].find('.');
  if(loc != string::npos)
    part[BASE].erase(loc); // Strip extension
  // Force to upper case:
  for(size_t i = 0; i < part[BASE].size(); i++)
    part[BASE][i] = toupper(part[BASE][i]);
  // Create file names and internal lines:
  part[HEADER] = part[BASE] + ".h";
  part[IMPLEMENT] = part[BASE] + ".cpp";
  part[HLINE1] = "//" ": " + part[HEADER];
  part[GUARD1] = "#ifndef " + part[BASE] + "_H";
  part[GUARD2] = "#define " + part[BASE] + "_H";
  part[GUARD3] = "#endif // " + part[BASE] +"_H";
  part[CPPLINE1] = string("//") + ": " + part[IMPLEMENT];
  part[INCLUDE] = "#include \"" + part[HEADER] + "\"";
  // First, try to open existing files:
  ifstream existh(part[HEADER].c_str()),
           existcpp(part[IMPLEMENT].c_str());
  if(!existh) { // Doesn't exist; create it
    ofstream newheader(part[HEADER].c_str());
    assure(newheader, part[HEADER].c_str());
    newheader << part[HLINE1] << endl
              << part[GUARD1] << endl
              << part[GUARD2] << endl << endl
              << part[GUARD3] << endl;
  } else { // Already exists; verify it
    stringstream hfile; // Write & read
    ostringstream newheader; // Write
    hfile << existh.rdbuf();
    // Check that first three lines conform:
    bool changed = false;
    string s;
    hfile.seekg(0);
    getline(hfile, s);
    bool lineUsed = false;
    // The call to good() is for Microsoft (later too):
    for(int line = HLINE1; hfile.good() && line <= GUARD2;
        ++line) {
      if(startsWith(s, part[line])) {
        newheader << s << endl;
        lineUsed = true;
        if(getline(hfile, s))
          lineUsed = false;
      } else {
        newheader << part[line] << endl;
        changed = true;
        lineUsed = false;
      }
    }
    // Copy rest of file
    if(!lineUsed)
      newheader << s << endl;
      newheader << hfile.rdbuf();
      // Check for GUARD3
      string head = hfile.str();
      if(head.find(part[GUARD3]) == string::npos) {
        newheader << part[GUARD3] << endl;
```

```
        changed = true;
      }
      // If there were changes, overwrite file:
      if(changed) {
        existh.close();
        ofstream newH(part[HEADER].c_str());
        assure(newH, part[HEADER].c_str());
        newH << "//@//\n"  // Change marker
             << newheader.str();
      }
    }
    if(!existcpp) { // Create cpp file
      ofstream newcpp(part[IMPLEMENT].c_str());
      assure(newcpp, part[IMPLEMENT].c_str());
      newcpp << part[CPPLINE1] << endl
             << part[INCLUDE] << endl;
    } else { // Already exists; verify it
      stringstream cppfile;
      ostringstream newcpp;
      cppfile << existcpp.rdbuf();
      // Check that first two lines conform:
      bool changed = false;
      string s;
      cppfile.seekg(0);
      getline(cppfile, s);
      bool lineUsed = false;
      for(int line = CPPLINE1;
          cppfile.good() && line <= INCLUDE; ++line) {
        if(startsWith(s, part[line])) {
          newcpp << s << endl;
          lineUsed = true;
          if(getline(cppfile, s))
            lineUsed = false;
        } else {
          newcpp << part[line] << endl;
          changed = true;
          lineUsed = false;
        }
      }
      // Copy rest of file
      if(!lineUsed)
        newcpp << s << endl;
      newcpp << cppfile.rdbuf();
      // If there were changes, overwrite file:
      if(changed) {
        existcpp.close();
        ofstream newCPP(part[IMPLEMENT].c_str());
        assure(newCPP, part[IMPLEMENT].c_str());
        newCPP << "//@//\n"  // Change marker
               << newcpp.str();
      }
    }
}

int main(int argc, char* argv[]) {
  if(argc > 1)
    cppCheck(argv[1]);
  else
    cppCheck("cppCheckTest.h");
} ///:~
```

207

首先注意一个有用的函数**startsWith( )**，这个函数的名字说明了它的功能——如果函数

的第1个字符串参数的内容以第2个字符串参数的内容开始（即第2个参数为第1个参数的前缀）时，它返回**true**。在查找期待的注释及相关的包含语句时使用这个函数。定义了字符串数组**part**之后，就可使用循环从头至尾对源代码文件中所期待查找的语句序列进行操作。如果源代码文件不存在，则仅将语句写到用已经给出的文件名命名的新文件中。如果文件存在，则每次搜索文件中的一行，并进行校验该行的出现。如果期待查找的语句不存在，则将其插入到源码文件中。需要特别注意的是，要确保不要遗漏已经存在的行（参看代码中使用布尔变量**lineUsed**的地方）。注意，现在是在对一个已经存在的文件使用**stringstream**对象，所以能够先写文件的内容至该对象，然后再从该对象中读取和搜索信息。

枚举类型**bufs**中的有名枚举常量分别是：**BASE**，用大写字母表示不带扩展名的基本文件名；**HEADER**，头文件名；**IMPLEMENT**，实现文件（扩展名为**cpp**）名；**HLINE1**，头文件中的第1行基本代码；**GUARD1**、**GUARD2**和**GUARD3**，头文件中的“警戒（guard）”行（防止多重包含）；**CPPLINE1**，**cpp**文件中的第1行；**INCLUDE**，**cpp**文件中包含头文件的语句。

208 如果运行这个程序但不带任何参数，则会创建下面两个文件：

```
// CPPCHECKTEST.h
#ifndef CPPCHECKTEST_H
#define CPPCHECKTEST_H

#endif // CPPCHECKTEST_H

// CPPCHECKTEST.cpp
#include "CPPCHECKTEST.h"
```

（这里省略了双斜线后面第1行注释中的冒号，以免混淆本教材中的代码提取符。在由执行**cppCheck**产生的真正输出中会包含在此省略的冒号。）

通过从文件中删去某些行然后重新执行程序，可以对程序完成的功能进行测试。可以看到每次执行程序后被删除的行会被写回文件。文件被修改后，在文件的第1行会加入字符串“//@//”以使读者注意到文件的变化。再次对文件进行处理前需要去掉这行（否则程序**cppcheck**执行时会假定原文件的第1行注释丢失）。

4.10.2 检测编译器错误

本教材中设计的所有代码在编译时都不会有错误发生。代码中会引起编译时错误的行，将用特殊的注释符号“//!”进行注释。下面的程序删去了这些特殊的注释，并添加了带有文件编号的注释行。当读者在自己的编译器上编译这些程序时，可能会产生错误信息，对所有文件进行编译时会看到所有文件的文件编号。这个程序在一个特殊文件中附加修改过的行，从而可以很容易地找出任何一个没有产生错误的行。

```
//: C04:Showerr.cpp {RunByHand}
// Un-comment error generators.
#include <cstddef>
#include <cstdlib>
#include <cstdio>
#include <fstream>
#include <iostream>
#include <sstream>
#include <string>
#include "../require.h"
using namespace std;

const string USAGE =
```

209

```
  "usage: showerr filename chapnum\n"
  "where filename is a C++ source file\n"
  "and chapnum is the chapter name it's in.\n"
  "Finds lines commented with //! and removes\n"
  "the comment, appending //(#) where # is unique\n"
  "across all files, so you can determine\n"
  "if your compiler finds the error.\n"
  "showerr /r\n"
  "resets the unique counter.";

class Showerr {
  const int CHAP;
  const string MARKER, FNAME;
  // File containing error number counter:
  const string ERRNUM;
  // File containing error lines:
  const string ERRFILE;
  stringstream edited; // Edited file
  int counter;
public:
  Showerr(const string& f, const string& en,
    const string& ef, int c)
  : CHAP(c), MARKER("//!"), FNAME(f), ERRNUM(en),
    ERRFILE(ef), counter(0) {}
  void replaceErrors() {
    ifstream infile(FNAME.c_str());
    assure(infile, FNAME.c_str());
    ifstream count(ERRNUM.c_str());
    if(count) count >> counter;
    int linecount = 1;
    string buf;
    ofstream errlines(ERRFILE.c_str(), ios::app);
    assure(errlines, ERRFILE.c_str());
    while(getline(infile, buf)) {
      // Find marker at start of line:
      size_t pos = buf.find(MARKER);
      if(pos != string::npos) {
        // Erase marker:
        buf.erase(pos, MARKER.size() + 1);
        // Append counter & error info:
        ostringstream out;
        out << buf << " // (" << ++counter << ") "
            << "Chapter " << CHAP
            << " File: " << FNAME
            << " Line " << linecount << endl;
        edited << out.str();
        errlines << out.str(); // Append error file
      }
      else
        edited << buf << "\n"; // Just copy
      ++linecount;
    }
  }
  void saveFiles() {
    ofstream outfile(FNAME.c_str()); // Overwrites
    assure(outfile, FNAME.c_str());
    outfile << edited.rdbuf();
    ofstream count(ERRNUM.c_str()); // Overwrites
    assure(count, ERRNUM.c_str());
    count << counter; // Save new counter
  }
};
```

210

```
int main(int argc, char* argv[]) {
  const string ERRCOUNT("../errnum.txt"),
    ERRFILE("../errlines.txt");
  requireMinArgs(argc, 1, USAGE.c_str());
  if(argv[1][0] == '/' || argv[1][0] == '-') {
    // Allow for other switches:
    switch(argv[1][1]) {
      case 'r': case 'R':
        cout << "reset counter" << endl;
        remove(ERRCOUNT.c_str()); // Delete files
        remove(ERRFILE.c_str());
        return EXIT_SUCCESS;
      default:
        cerr << USAGE << endl;
        return EXIT_FAILURE;
    }
  }
  if(argc == 3) {
    Showerr s(argv[1], ERRCOUNT, ERRFILE, atoi(argv[2]));
    s.replaceErrors();
    s.saveFiles();
  }
} ///:~
```

211 读者可以用自己选择的标记替换文件中的标记。

程序从每个文件中每次读入一行，然后从这行的开头逐个字符搜索指定的标记；修改这一行并把它放入错误行列表和字符串流对象**edited**中。当所有的文件处理结束后，关闭文件（到达文件范围末尾），作为输出文件重新打开它，将**edited**中的内容输出到文件中。注意，计数器也被保存到一个外部文件中。在下一次程序执行时，计数器的计数在上次计数值的基础上增加。

4.10.3 一个简单的数据记录器

这个例子说明了一种可以将数据记录到磁盘，然后检索它以便进行处理的方法。程序想要产生一个多点的海洋温度——深度轮廓图。类**DataPoint**用来保存数据：

```
//: C04:DataLogger.h
// Datalogger record layout.
#ifndef DATALOG_H
#define DATALOG_H
#include <ctime>
#include <iosfwd>
#include <string>
using std::ostream;

struct Coord {
  int deg, min, sec;
  Coord(int d = 0, int m = 0, int s = 0)
  : deg(d), min(m), sec(s) {}
  std::string toString() const;
};

ostream& operator<<(ostream&, const Coord&);

class DataPoint {
  std::time_t timestamp; // Time & day
  Coord latitude, longitude;
  double depth, temperature;
public:
  DataPoint(std::time_t ts, const Coord& lat,
```

```
          const Coord& lon, double dep, double temp)
  : timestamp(ts), latitude(lat), longitude(lon),
      depth(dep), temperature(temp) {}
  DataPoint() : timestamp(0), depth(0), temperature(0) {}
  friend ostream& operator<<(ostream&, const DataPoint&);
};
#endif // DATALOG_H ///:~
```

212

类**DataPoint**包含一个时间标志，时间标志为头文件**<ctime>**中定义的**time_t**类型的值，还有经度和纬度坐标，以及深度和温度值。在这里使用插入符进行格式化操作。下面是文件的实现：

```
//: C04:DataLogger.cpp {O}
// Datapoint implementations.
#include "DataLogger.h"
#include <iomanip>
#include <iostream>
#include <sstream>
#include <string>
using namespace std;

ostream& operator<<(ostream& os, const Coord& c) {
  return os << c.deg << '*' << c.min << '\''
            << c.sec << '"';
}

string Coord::toString() const {
  ostringstream os;
  os << *this;
  return os.str();
}

ostream& operator<<(ostream& os, const DataPoint& d) {
  os.setf(ios::fixed, ios::floatfield);
  char fillc = os.fill('0'); // Pad on left with '0'
  tm* tdata = localtime(&d.timestamp);
  os << setw(2) << tdata->tm_mon + 1 << '\\'
     << setw(2) << tdata->tm_mday << '\\'
     << setw(2) << tdata->tm_year+1900 << ' '
     << setw(2) << tdata->tm_hour << ':'
     << setw(2) << tdata->tm_min << ':'
     << setw(2) << tdata->tm_sec;
  os.fill(' '); // Pad on left with ' '
  streamsize prec = os.precision(4);
  os << " Lat:"    << setw(9) << d.latitude.toString()
     << ", Long:"  << setw(9) << d.longitude.toString()
     << ", depth:" << setw(9) << d.depth
     << ", temp:"  << setw(9) << d.temperature;
  os.fill(fillc);
  os.precision(prec);
  return os;
} ///:~
```

213

使用函数**Coord::toString( )**是必要的，因为类**DataPoint**的插入符在输出经度和纬度之前调用了**setw( )**。无论何时对**Coord**对象使用流插入符，宽度只对第1次插入（即插入数据到**Coord::deg**）有效，因为宽度改变后总是立即重置。调用函数**setf( )**使得输出浮点数时精度是固定的，函数**precision( )**设置精度为小数点后四位十进制数。请注意，程序中是如何恢复在调用插入符之前设置填充字符和数据精度的。

为得到存储在**DataPoint::timestamp**中的各个测试点的测试时间，可以调用函数**std::localtime( )**，该函数返回指向**tm**对象的静态指针。结构**tm**布局（定义）如下：

```
struct tm {
  int tm_sec; // 0-59 seconds
  int tm_min; // 0-59 minutes
  int tm_hour; // 0-23 hours
  int tm_mday; // Day of month
  int tm_mon; // 0-11 months
  int tm_year; // Years since 1900
  int tm_wday; // Sunday == 0, etc.
  int tm_yday; // 0-365 day of year
  int tm_isdst; // Daylight savings?
};
```

1. 产生测试数据

这里的程序用来建立一个二进制格式的测试数据文件（使用**write( )**函数），使用**DataPoint**插入符建立ASCII格式的第2个文件。也可以把这些数据显示到屏幕上，但以文件形式查看更方便。

```
//: C04:Datagen.cpp
// Test data generator.
//{L} DataLogger
#include <cstdlib>
#include <ctime>
#include <cstring>
#include <fstream>
#include "DataLogger.h"
#include "../require.h"
using namespace std;

int main() {
  time_t timer;
  srand(time(&timer)); // Seed the random number generator
  ofstream data("data.txt");
  assure(data, "data.txt");
  ofstream bindata("data.bin", ios::binary);
  assure(bindata, "data.bin");
  for(int i = 0; i < 100; i++, timer += 55) {
    // Zero to 199 meters:
    double newdepth  = rand() % 200;
    double fraction = rand() % 100 + 1;
    newdepth += 1.0 / fraction;
    double newtemp = 150 + rand() % 200; // Kelvin
    fraction = rand() % 100 + 1;
    newtemp += 1.0 / fraction;
    const DataPoint d(timer, Coord(45,20,31),
                      Coord(22,34,18), newdepth,
                      newtemp);
    data << d << endl;
    bindata.write(reinterpret_cast<const char*>(&d),
                  sizeof(d));
  }
} ///:~
```

214

文件**data.txt**为ASCII格式、采用顺序方法创建的顺序文件，而文件**data.bin**为二进制格式文件，构造函数根据标志**ios::binary**建立此文件。为了说明文本文件采用的格式化形式，这里给出文件**data.txt**的第1行内容（因为行的长度大于本教材页的宽度，所以进行了换行）：

```
07\28\2003 12:54:40 Lat:45*20'31", Long:22*34'18", depth:
16.0164, temp: 242.0122
```

标准C库函数**time( )**用执行该语句的当前时间来更新由函数参数指向的**time_t**的值，在

大多数操作平台上，这个时间是从1970年1月1日00:00:00 GMT（Aquarius（水瓶星座）时代的黎明?）开始的秒的计数值。利用标准C中的库函数**srand( )**作为随机数产生器来设置当前时间也是很方便的方法，这里就是如此。

之后，把**timer**定时器增加55秒，在各个模拟读操作之间产生有趣的间隔。 215

各采集点的经度和纬度值采用固定值，表示所采集的数据集是在某个特定的区域。深度和温度值由标准C库函数**rand( )**产生，该函数返回一个零到依赖于操作平台的常量**RAND_MAX**之间的伪随机数，**RAND_MAX**常量（一般为所在操作平台的无符号整形最大值）在文件**<cstdlib>**中定义。为把得到的伪随机数限制在一个期望的合理范围内，使用取模运算符**%**（从整数相除得到余数）和范围的上限来限定。这些伪随机数都是整数，为了添加小数部分，第2次调用**rand( )**以产生小数，将得到的值加一后取倒数（防止除数为零的错误）。

本程序中，文件**data.bin**被用作数据容器，尽管这个数据容器存在于磁盘而不是RAM中。函数**write( )**把数据以二进制方式输出到磁盘上。函数的第1个参数是源数据块的起始地址——注意必须将参数设置为**char***类型，因为函数**write( )**使用专用流（narrow stream）。第2个参数是要写出的字符数目，在这个例子中就是**DataPoint**类对象的大小（再一次指明，因为使用的是窄字符流）。因为类**DataPoint**不含指针，所以输出这个类的对象到磁盘上不会产生问题。如果类对象很复杂，则必须实现串行化（serialization）设计，把指针指向的数据写入磁盘，在以后读回数据时再定义新的指针。（不在本卷中讨论串行化——大部分商业化销售的类库都有一些串行化结构来构造它们。）

2. 校验和查看数据

为校验以二进制格式存储的数据的正确性，可以用输入流的成员函数**read( )**将数据读到内存，然后和最初由**Datagen.cpp**生成的文本文件进行比较。下面的例子仅把格式化的结果输出到**cout**，但读者可以把这些输出重新送到一个文件中，然后用文件比较“实用程序”来进行校验，校验这个文件与最初的文件是否完全相同：

```
//: C04:Datascan.cpp
//{L} DataLogger
#include <fstream>
#include <iostream>
#include "DataLogger.h"
#include "../require.h"
using namespace std;
int main() {
  ifstream bindata("data.bin", ios::binary);
  assure(bindata, "data.bin");
  DataPoint d;
  while(bindata.read(reinterpret_cast<char*>(&d),
         sizeof d))
    cout << d << endl;
} ///:~
```

216

4.11 国际化

现在，软件工业是一个新兴的、健康的、具有世界范围的经济市场，这需要应用程序能在多种语言环境下运行。早在20世纪80年代，C标准委员会就加入了对非美国表达方式习惯的区域性（locale）机制的支持。所谓区域性是在显示一些实体，如时间和货币数量时当地人们习惯使用的方式。在20世纪90年代，C标准委员会同意补充处理宽字符（wide character）（由类型**wchar_t**表示）的特殊函数进入标准C，容许这些函数支持非ASCII码字符集，一般用于西

欧诸国范围。尽管宽字符所占空间的大小并不特殊，但在一些操作平台上把它按32位字长进行实现，可以满足统一代码协会（Unicode Consortium）对编码的特殊需要，同时也适用于亚洲标准化组织定义的多字节字符集。C++支持宽字符和区域（locale）字符，把两者整合到了输入输出流类库中。

4.11.1 宽字符流

宽字符流（wide stream）是一个处理那些宽字符的流类。目前引入的所有例子（除了第3章中那些带有宽字符特征的例子外）都使用专门处理**char**类型的窄（narrow）字符流。因为流操作的本质都是一样的，与基础字符类型没有关系，所以一般将其封装成模板。例如，可以将所有输入流类都与类模板**basic_istream**连接来定义：

```
template<class charT, class traits = char_traits<charT> >
class basic_istream {...};
```

事实上，根据下面的类型定义，所有输入流类都是该模板的特化：

217

```
typedef basic_istream<char> istream;
typedef basic_istream<wchar_t> wistream;
typedef basic_ifstream<char> ifstream;
typedef basic_ifstream<wchar_t> wifstream;
typedef basic_istringstream<char> istringstream;
typedef basic_istringstream<wchar_t> wistringstream;
```

其他流类型的定义与此类似。

总而言之，读者用这些方法可以创建不同字符类型的流。但事情也不是那么简单。原因是提供给**char**类型和**wchar_t**类型的字符处理函数的名称不相同。比较两个窄字符串，比如使用函数**strcmp( )**。而用于两个宽字符的比较函数为**wcscmp( )**。（请记住这些函数在C语言中的原始声明，这些函数没有重载版本，所有的函数名需要具有惟一性。）正因为如此，一般情况下流类对象的比较运算符不能仅仅调用**strcmp( )**。这就需要引入一种方法，使用这种方法可以在进行流对象的比较操作时自动调用正确的底层函数。

解决的办法是找出它们因子的差异成为一个新的抽象。对字符的操作被抽象成为一个**char_traits**模板，正如在第3章结尾处讨论的，这个模板中预定义了**char**（字符型）和**wchar_t**（宽字符型）类型。比较两个字符串时，**basic_string**先调用**traits::compare( )**（记住特征参数**traits**是模板的第2个参数），**traits::compare( )**再根据所用的字符类型调用**strcmp( )**或**wcscmp( )**。（对于**basic_string**来说，这一点是必须清晰的。）

如果访问底层字符处理函数，只需要关注**char_traits**，但大多数情况下不需要特别注意。然而，为了增强程序的健壮性，需要将插入符和提取符定义为模板，以适应用户要在宽字符流上对它们的使用。

为解释清楚，回忆一下本章开始时引入的类**Date**中的插入符。它的原始定义如下：

```
ostream& operator<<(ostream&, const Date&);
```

这个插入符只能用于窄字符流。为了使其具有通用性，现在把它定义成基于**basic_ostream**的模板：

218

```
template<class charT, class traits>
std::basic_ostream<charT, traits>&
operator<<(std::basic_ostream<charT, traits>& os,
           const Date& d) {
  charT fillc = os.fill(os.widen('0'));
  charT dash = os.widen('-');
```

```
  os << setw(2) << d.month << dash
     << setw(2) << d.day << dash
     << setw(4) << d.year;
  os.fill(fillc);
  return os;
}
```

注意，也需要用模板参数**charT**替换**char**来声明**fillc**，因为**fillc**的声明依赖于模板实例化时的参数是**char**还是**wchar_t**。

因为在定义模板时不知道所使用的流的类型，所以需要有一种自动将字符文字的长度转换成对于该流来说大小合适的方法。成员函数**widen( )**负责处理这项工作。例如，对表达式**widen('-')**来说就是将其参数转变成**L'-'**（文字语法相当于**wchar_t('-')**转变），如果为宽字符流则不进行转换。反之亦然。如果需要，函数**narrow( )**将字符转换成**char**类型。

可以使用**widen( )**为本章前面较早提供的程序例子编写一个名为**nl**操纵算子的通用版本：

```
template<class charT, class traits>
basic_ostream<charT,traits>&
nl(basic_ostream<charT,traits>& os) {
  return os << charT(os.widen('\n'));
}
```

4.11.2 区域性字符流

不同国家的计算机输出之间最显著的不同，在于分割整数和实数的小数部分所使用的标点符号。在美国，一个句号表示一个小数点，但是世界上大多数国家用逗号表示小数点。如果为各个区域的国家的输出显示分别定义不同的格式，这是十分不方便的。这里再一次使用抽象来解决这个问题

这次的抽象是区域（locale）。每一流类都有相联系的区域对象，这些对象用来指出如何显示不同的文化背景下的确定的数量。一个区域对象管理一系列视文化不同而定的数量的显示规则，定义如下：

219

种 类	作 用
collate	允许按照不同的比较顺序比较字符串
ctype	对字符类型和 <cctype> 中的惯用程序进行抽象
monetary	支持货币数量的不同格式显示
numeric	支持实数的不同格式的显示，包括数的基（小数点）和分组（每千位）符号
time	支持多种不同格式的时间和日期的显示
messages	其实现依赖于不同内容的消息目录（如不同语言下的错误消息）

下面的程序说明了基本区域性字符流的行为：

```
//: C04:Locale.cpp {-g++}{-bor}{-edg} {RunByHand}
// Illustrates effects of locales.
#include <iostream>
#include <locale>
using namespace std;

int main() {
  locale def;
  cout << def.name() << endl;
  locale current = cout.getloc();
  cout << current.name() << endl;
  float val = 1234.56;
  cout << val << endl;
```

```
  // Change to French/France
220 cout.imbue(locale("french"));
  current = cout.getloc();
  cout << current.name() << endl;
  cout << val << endl;

  cout << "Enter the literal 7890.12: ";
  cin.imbue(cout.getloc());
  cin >> val;
  cout << val << endl;
  cout.imbue(def);
  cout << val << endl;
} ///:~
```

输出结果如下:

```
C
C
1234.56
French_France.1252
1234,56
Enter the literal 7890.12: 7890,12
7890,12
7890.12
```

默认的区域为“C”区域，是C和C++程序员多年来一直使用的（基本上是英语和美语文化）。所有的流最初都完全“浸透（imbue)”在“C”区域环境下。成员函数**imbue( )**改变了一个流使用的区域。注意程序输出了“法语”区域在ISO（国际标准化组织）中的全称（即在法国表达的“法语”相对于其他国家表达的“法语”)。这个例子说明在这种区域下，数字中的小数点用逗号表示。如果想在这种区域的规则下进行输入，则需要把**cin**改变到相同的区域下工作。

每个区域目录被分成几个领域，每个领域都是一些对应于相应目录封装了特定功能的类。例如，目录**time**包含的领域有**time_put**和**time_get**，它们分别含有进行时间、日期的输入（**input**）和输出（**output**）的函数。而目录**monetary**包含的领域有**money_get**、**money_put**和**moneypunct**。（**moneypunct**决定了流通中的货币符号。）下面的程序说明了**moneypunct**领域。（**time**领域需要用到一种复杂的迭代器，它超出了本章的讨论范围。）

```
221 //: C04:Facets.cpp {-bor}{-g++}{-mwcc}{-edg}
#include <iostream>
#include <locale>
#include <string>
using namespace std;

int main() {
  // Change to French/France
  locale loc("french");
  cout.imbue(loc);
  string currency =
    use_facet<moneypunct<char> >(loc).curr_symbol();
  char point =
    use_facet<moneypunct<char> >(loc).decimal_point();
  cout << "I made " << currency << 12.34 << " today!"
       << endl;
} ///:~
```

输出了法国流通货币符号和小数点分隔符:

```
I made Ç12,34 today!
```

读者也可定义自己的领域以构建个人化的区域。[⊖]但要当心，区域的开销是相当可观的。事实上，一些供销商提供了区别于标准C++库的不同风格的库，以便满足对使用标准库有限制条件的环境。[⊜]

4.12 小结

本章详细地介绍了输入输出流类库。从本章中学到的内容可以满足读者使用输入输出流创建程序的需要。注意，输入输出流的一些附加的特性并不常用，读者可以查阅输入输出流头文件和阅读编译器文档或本章及附录中提到的参考文献。

4.13 练习

222

4-1 有一个由创建的**ifstream**对象打开的文件。创建一个**ostringstream**对象，使用其成员函数**rdbuf( )**读该文件全部内容到**ostringstream**对象中。提取文件基础缓冲区的**string**拷贝，使用标准C语言头文件**<cctype>**中定义的宏**toupper( )**将每个字符转换为大写。将结果输出到一个新文件。

4-2 编写程序：打开一个文件（文件名作为命令行的第1个参数），并搜索文件中单词集合中的任意一个单词（作为参数出现在命令行上）。每次读入一行并将匹配的行（连同行号一起）写到一个新文件中。

4-3 编写一个程序：添加“版权声明”到所有源代码文件的开始位置，这些信息通过程序命令行参数指明。

4-4 使用自己喜欢的文本搜索程序（如**grep**），输出包含一种特殊模式的所有文件的文件名（仅输出文件名）。重新发送输出到一个新文件。编写一个程序，用这个新文件里的内容来产生一个批处理文件，这个批处理文件对每个由文本搜索程序找到的文件，调用自编的编辑器进行编辑。

4-5 我们知道使用**setw( )**可以设置读入字符的最小个数，但是如何设置读入字符的最大数量？编写一个效用算子，使得用户可以指定提取字符数目的最大值。该效用算子也可以进行输出。输出时，这个效用算子可以截短输出域的宽度，如果需要可以保持域宽限制的设置。

4-6 编程演示如下过程：如果失败或致命错误标志位设置后，随后突然产生流异常，流将立即抛出异常。

4-7 由字符串流提供转换很容易实现，不过要付出一定的代价。编写一个程序，实现**stringstream**转换系统，把这个程序和**atoi( )**比较，以便观察**stringstream**转换系统最终花费的代价。

4-8 编写一个结构**Person**，结构的数据域包含名字，年龄，地址等等。其中的字符串类型数据成员为固定大小的数组。每条记录的关键字为身份证号码（社会保险编号）。实现下面的类**Database**： 223

⊖ 详情请参看 Langer & Kreft的著作。

⊜ 例如，参看 Dinkumware的Abridged 库的网址 http://www.dinkumware.com。这个库不支持locale，对异常的支持为可选项。

```
class DataBase {
public:
  // Find where a record is on disk
  size_t query(size_t ssn);
  // Return the person at rn (record number)
  Person retrieve(size_t rn);
  // Record a record on disk
  void add(const Person& p);
};
```

写一些**Person**记录到磁盘（不要把这些记录都放在内存中）。当用户需要时，从磁盘中将这些记录读回到内存。**DataBase**类中的I/O操作使用**read( )**和**write( )**处理所有**Person**记录。

4-9 为结构**Person**编写一个插入符**operator<<**，实现对读入的记录用格式化形式显示。通过将数据输出到文件来演示该插入符的功能。

4-10 假定存储结构**Person**的数据库丢失了，但前一个练习中产生的输出文件还存在。使用这个文件重新建立数据库。要确保程序中使用错误检查。

4-11 写1 000 000次**size_t(-1)**（操作平台规定的最大的无符号整数**unsigned int**）到一个文本文件。再以二进制格式写1 000 000次**size_t(-1)**到一个二进制文件。比较两个文件的大小，看二进制格式的文件能节省多少空间。（读者或许首先想要计算出在自己的操作平台上能节省多少空间。）

4-12 打印一个无理数如**sqrt(2.0)**时，通过重复增加函数**precision( )**的参数值，观察输入输出流实现的显示该数精度位数的最大个数。

4-13 编写一个程序，从文件中读入一些实数，并且显示（打印）这些实数的和、平均值、最小值和最大值。

4-14 在执行前，猜测下面程序的输出结果：

```
//: C04:Exercise14.cpp
#include <fstream>
#include <iostream>
#include <sstream>
#include "../require.h"
using namespace std;

#define d(a) cout << #a " ==\t" << a << endl;

void tellPointers(fstream& s) {
  d(s.tellp());
  d(s.tellg());
  cout << endl;
}
void tellPointers(stringstream& s) {
  d(s.tellp());
  d(s.tellg());
  cout << endl;
}
int main() {
  fstream in("Exercise14.cpp");
  assure(in, "Exercise14.cpp");
  in.seekg(10);
  tellPointers(in);
  in.seekp(20);
  tellPointers(in);
  stringstream memStream("Here is a sentence.");
  memStream.seekg(10);
```

224

```
  tellPointers(memStream);
  memStream.seekp(5);
  tellPointers(memStream);
} ///:~
```

4-15 假定要从按如下格式存储数据的文件中按行提取数据:

```
//: C04:Exercise15.txt
Australia
5E56,7667230284,Langler,Tyson,31.2147,0.00042117361
2B97,7586701,Oneill,Zeke,553.429,0.0074673053156065
4D75,7907252710,Nickerson,Kelly,761.612,0.010276276
9F2,6882945012,Hartenbach,Neil,47.9637,0.0006471644
Austria
480F,7187262472,Oneill,Dee,264.012,0.00356226040013
1B65,4754732628,Haney,Kim,7.33843,0.000099015948475
DA1,1954960784,Pascente,Lester,56.5452,0.0007629529
3F18,1839715659,Elsea,Chelsy,801.901,0.010819887645
Belgium
BDF,5993489554,Oneill,Meredith,283.404,0.0038239127
5AC6,6612945602,Parisienne,Biff,557.74,0.0075254727
6AD,6477082,Pennington,Lizanne,31.0807,0.0004193544
4D0E,7861652688,Sisca,Francis,704.751,0.00950906238
Bahamas
37D8,6837424208,Parisienne,Samson,396.104,0.0053445
5E98,6384069,Willis,Pam,90.4257,0.00122009564059246
1462,1288616408,Stover,Hazal,583.939,0.007878970561
5FF3,8028775718,Stromstedt,Bunk,39.8712,0.000537974
1095,3737212,Stover,Denny,3.05387,0.000041205248883
7428,2019381883,Parisienne,Shane,363.272,0.00490155
///:~
```

225

这些数据按地区划分成若干部分，每部分的开头是一个地区名称，下面的每行都是该地区的每一个销售人员的信息。由逗号分隔开的域（字段）代表每个销售人员的相关数据。每行的第1个域是SELLER_ID，遗憾的是，这个域是按十六进制数的格式写的。第2个域是PHONE_NUMBER（注意，有一些域缺少地区编码）。接下来是LAST_NAME和FIRST_NAME。TOTAL_SALES是倒数第2栏。最后一栏是这个销售人员的售货量占公司总售货量的百分比的小数表示。编写程序，在终端窗口用格式化方式显示这些数据，执行结果可以很容易地显示各个销售人员业绩的趋势。输出的样本如下所示:

```
                    Australia
              -------------------------------

*Last Name*  *First Name*  *ID*   *Phone*        *Sales*   *Percent*

Langler      Tyson         24150  766-723-0284    31.24    4.21E-02
Oneill       Zeke          11159  XXX-758-6701   553.43    7.47E-01
(etc.)
```

227

第5章 深入理解模板

C++模板应用的便利性远远超出了它只是一种“T类型容器”（containers of T）的范畴。尽管其最初的设计动机是为了能产生类型安全的通用容器，但在现代C++中，模板也用来生成自定义代码，这些代码通过编译时的程序设计构造来优化程序的执行。

本章将从实用的角度来看看现代C++中利用模板编程的强大能力（以及缺陷）。对于与模板相关的C++语言的优点和缺陷的更完备的分析，推荐读者阅读由Daveed Vandevoorde和Nico Josuttis[⊖]所著的那本极棒的书。

5.1 模板参数

正如在第1卷中描述的那样，模板有两类：函数模板和类模板。二者都是由它们的参数来完全地描绘模板的特性。每个模板参数描述了下述内容之一：

1) 类型（或者是系统固有类型或者是用户自定义类型）。

2) 编译时常数值（例如，整数、指针和某些静态实体的引用，通常是作为无类型参数的引用）。

3) 其他模板。

第1卷中所举的例子都属于第1种情况，也是最常用的。现在作为简单的类似容器模板的典型示例似乎就是**Stack**类。作为容器，**Stack**对象与容器中存储的对象的类型毫无关联；持有对象的逻辑独立于所持有的对象的类型。基于这个原因，可以用一个类型参数来代表所包含的类型：

228

```
template<class T> class Stack {
  T* data;
  size_t count;
public:
  void push(const T& t);
  // Etc.
};
```

某个特定的**Stack**实例所使用的实际类型，由参数**T**的实参类型来决定：

```
Stack<int> myStack;  // A Stack of ints
```

编译器通过用**int**替代**T**生成相应的代码，从而提供了一个**Stack**的**int**版。在这个例子中，由模板生成类的实例的名字是**Stack<int>**。

5.1.1 无类型模板参数

一个无类型模板参数必须是一个编译时所知的整数值。举个例子，可以创建一个固定长度的**Stack**，指定一个无类型参数作为其中基础数组的大小，如下所示：

```
template<class T, size_t N> class Stack {
  T data[N];  // Fixed capacity is N
  size_t count;
```

⊖ Vandevoorde 和Josuttis 所著的《C++ Templates: The complete Guide》Addison Wesley，2003。注意“Daveed”有时会写作“David”。

```
public:
  void push(const T& t);
  // Etc.
};
```

当需要这个模板的一个实例时，必须为参数N提供一个编译时常数值，例如：

```
Stack<int, 100> myFixedStack;
```

由于N的值在编译时是已知的，内含的数组（**data**）可以被置于运行时堆栈而不是动态存储空间。这种方式避免了与动态内存分配的高层关联，从而提高了运行性能。根据之前提过的模式，上述模板的实例化类名字是**Stack<int, 100>**。这意味着任何一个N的不同取值都会产生一个惟一的类类型。例如，**Stack<int, 99>**与**Stack<int, 100>**就是两个不同的类。 229

将在第7章详细讨论的**bitset**类模板，是标准C++库中惟一使用了无类型模板参数（它指定了**bitset**对象所持有的位的数目）的类。下面的随机数生成器的例子使用了**bitset**来跟踪这些数，这样在随机数生成器下一次工作周期重新开始之前，所有在允许范围内的数都将无重复地按照随机顺序返回。这个例子也重载了运算符**operator( )**，用来产生一个熟悉的功能调用语法。

```
//: C05:Urand.h {-bor}
// Unique randomizer.
#ifndef URAND_H
#define URAND_H
#include <bitset>
#include <cstddef>
#include <cstdlib>
#include <ctime>
using std::size_t;
using std::bitset;

template<size_t UpperBound> class Urand {
  bitset<UpperBound> used;
public:
  Urand() { srand(time(0)); } // Randomize
  size_t operator()(); // The "generator" function
};

template<size_t UpperBound>
inline size_t Urand<UpperBound>::operator()() {
  if(used.count() == UpperBound)
    used.reset();  // Start over (clear bitset)
  size_t newval;
  while(used[newval = rand() % UpperBound])
    ; // Until unique value is found
  used[newval] = true;
  return newval;
}
#endif // URAND_H ///:~
```

由**Urand**生成的数全是独一无二的，这是因为**bitset used**跟踪了随机空间中（上限设置成模板参数）所有可能产生的数，并且设置相应的状态位来记录每一个使用过的数。当这些数全都用完了之后，**bitset**被清空以便为下一次工作重新开始做准备。下面是一个描述如何使用**Urand**对象的简单的驱动程序： 230

```
//: C05:UrandTest.cpp {-bor}
#include <iostream>
#include "Urand.h"
using namespace std;
```

```
int main() {
  Urand<10> u;
  for(int i = 0; i < 20; ++i)
    cout << u() << ' ';
} ///:~
```

正像将在本章后面解释的那样，无类型模板参数在优化数值的计算方面也是很重要的。

5.1.2 默认模板参数

在类模板中，可以为模板参数提供默认（缺省）参数，但是在函数模板中却不行。作为默认的模板参数，它们只能被定义一次，编译器会知道第1次的模板声明或定义。一旦引入了一个默认参数，所有它之后的模板参数也必须具有默认值。例如，为了使前面介绍的固定大小的**Stack**模板更友好一些，可以加入一个默认参数，如下所示：

```
template<class T, size_t N = 100> class Stack {
  T data[N];  // Fixed capacity is N
  size_t count;
public:
  void push(const T& t);
  // Etc.
};
```

现在，如果在声明一个**Stack**对象时省略了第2个模板参数，**N**的值将默认为100。

231

也可以为所有参数提供默认值，但当声明一个实例时必须使用一对空的尖括号，这样编译器就知道说明了一个类模板。

```
template<class T = int, size_t N = 100>  // Both defaulted
class Stack {
  T data[N];  // Fixed capacity is N
  size_t count;
public:
  void push(const T& t);
  // Etc.
};

Stack<> myStack;  // Same as Stack<int, 100>
```

默认参数大量用于标准C++库中。比如**vector**类模板声明如下：

```
template<class T, class Allocator = allocator<T> >
class vector;
```

注意，在最后两个右尖括号字符之间有空格。这就避免了编译器将那两个字符（> >）解释为右移运算符。

这个声明说明了**vector**有两个参数：一个参数表示它持有的包含对象的类型，另一个参数代表**vector**所使用的分配器。任何时候只要省略了第2个参数，就会使用标准**allocator**模板，它的参数由第1个模板参数来确定。这个声明也说明，可以在随后的次一级模板的参数中使用该模板参数，就像在这里使用**T**一样。

尽管不能在函数模板中使用默认的模板参数，却能够用模板参数作为普通函数的默认参数。下面的函数模板在参数列表中加入了一个元素：

```
//: C05:FuncDef.cpp
#include <iostream>
using namespace std;

template<class T> T sum(T* b, T* e, T init = T()) {
```

```
  while(b != e)
    init += *b++;
    return init;
  }

  int main() {
    int a[] = { 1, 2, 3 };
    cout << sum(a, a + sizeof a / sizeof a[0]) << endl; // 6
  } ///:~
```

232

sum( )的第3个参数是作为对这些元素进行累积运算的初始值。由于省略了它，这个参数就默认为是**T( )**，在这里是**int**或其他系统固有的类型，它调用了一个伪构造函数执行零初始化操作。

5.1.3 模板类型的模板参数

模板可以接受的第3种模板参数类型是另一个类模板。如果想在代码中将一个模板类参数用作另一个模板，编译器首先需要知道这个参数是一个模板。下面的例子说明了一个模板类型的模板参数：

```
//: C05:TempTemp.cpp
// Illustrates a template template parameter.
#include <cstddef>
#include <iostream>
using namespace std;

template<class T>
class Array { // A simple, expandable sequence
  enum { INIT = 10 };
  T* data;
  size_t capacity;
  size_t count;
public:
  Array() {
    count = 0;
    data = new T[capacity = INIT];
  }
  ~Array() { delete [] data; }
  void push_back(const T& t) {
    if(count == capacity) {
      // Grow underlying array
      size_t newCap = 2 * capacity;
      T* newData = new T[newCap];
      for(size_t i = 0; i < count; ++i)
        newData[i] = data[i];
      delete [] data;
      data = newData;
      capacity = newCap;
    }
    data[count++] = t;
  }
  void pop_back() {
    if(count > 0)
      --count;
  }
  T* begin() { return data; }
  T* end() { return data + count; }
};

template<class T, template<class> class Seq>
```

233

```
class Container {
  Seq<T> seq;
public:
  void append(const T& t) { seq.push_back(t); }
  T* begin() { return seq.begin(); }
  T* end() { return seq.end(); }
};

int main() {
  Container<int, Array> container;
  container.append(1);
  container.append(2);
  int* p = container.begin();
  while(p != container.end())
    cout << *p++ << endl;
} ///:~
```

Array类模板是个很平常的序列容器。**Container**模板包含两个参数：一个参数是它持有的类对象类型，还有一个参数是它持有的类对象类型的序列数据结构。在**Container**类的实现中下面一行语句通知编译器，**Seq**是一个模板：

```
Seq<T> seq;
```

如果还没有声明**Seq**是一个模板类型的模板参数，编译器就不会在这里将**Seq**解释为一个模板，尽管已经如此使用了它。在**main( )**中使用了一个持有整数的**Array**将一个**Container**实例化，因此本例中的**Seq**代表**Array**。

234 注意，在本例**Container**的声明中对**Seq**的参数进行命名不是必需的。所讨论的这一行是：

```
template<class T, template<class> class Seq>
```

尽管可以这样写：

```
template<class T, template<class U> class Seq>
```

无论什么地方参数**U**都不是必需的。加上这个参数仅仅是为了说明**Seq**是一个持有单一类型参数的类模板。这种情况类似于某些时候省略函数参数的名称，当不需要它们的时候就可以省略掉。例如当重载自增（增1）运算符++时：

```
T operator++(int);
```

这里的**int**仅仅是一个占位符，并不需要有变量名称。

下面的程序使用了一个固定大小的数组，它有一个特别的模板参数表示数组的长度：

```
//: C05:TempTemp2.cpp
// A multi-variate template template parameter.
#include <cstddef>
#include <iostream>
using namespace std;

template<class T, size_t N> class Array {
  T data[N];
  size_t count;
public:
  Array() { count = 0; }
  void push_back(const T& t) {
    if(count < N)
      data[count++] = t;
  }
  void pop_back() {
    if(count > 0)
      --count;
  }
  T* begin() { return data; }
```

```
  T* end() { return data + count; }
};
template<class T,size_t N,template<class,size_t> class Seq>
class Container {
  Seq<T,N> seq;
public:
  void append(const T& t) { seq.push_back(t); }
  T* begin() { return seq.begin(); }
  T* end() { return seq.end(); }
};

int main() {
  const size_t N = 10;
  Container<int, N, Array> container;
  container.append(1);
  container.append(2);
  int* p = container.begin();
  while(p != container.end())
    cout << *p++ << endl;
} ///:~
```

235

再说明一次，在**Container**的声明内部，**Seq**的声明中参数名称不是必需的，但是需要有两个参数来声明数据成员**seq**，所以无类型参数**N**出现在模板型参数前面。

结合一下默认参数和模板型的模板参数就会发现一些细微的深一层问题。当编译器看到模板型模板参数的内部参数时，无法顾及到默认参数，因此为了得到一个确切的匹配，必须重复声明默认参数。下面的例子中，在固定大小的**Array**模板中使用了一个默认参数，这个例子也显示了如何在C++语言中适应这个古怪的举动。

```
//: C05:TempTemp3.cpp {-bor}{-msc}
// Template template parameters and default arguments.
#include <cstddef>
#include <iostream>
using namespace std;

template<class T, size_t N = 10>  // A default argument
class Array {
  T data[N];
  size_t count;
public:
  Array() { count = 0; }
  void push_back(const T& t) {
    if(count < N)
      data[count++] = t;
  }
  void pop_back() {
    if(count > 0)
      --count;
  }
  T* begin() { return data; }
  T* end() { return data + count; }
};

template<class T, template<class, size_t = 10> class Seq>
class Container {
  Seq<T> seq;  // Default used
public:
  void append(const T& t) { seq.push_back(t); }
  T* begin() { return seq.begin(); }
  T* end() { return seq.end(); }
};
```

236

```
int main() {
  Container<int, Array> container;
  container.append(1);
  container.append(2);
  int* p = container.begin();
  while(p != container.end())
    cout << *p++ << endl;
} ///:~
```

在下面这行语句中默认值的大小为10是必须的:

```
template<class T, template<class, size_t = 10> class Seq>
```

不管是在**Container**中**seq**的定义，还是在**main( )**中**container**的定义都使用了默认值。本例与**TempTemp2.cpp**惟一的不同点，就是使用了默认值。这也是与前面所陈述的规则——即默认参数在一个编辑单元内仅能出现一次——惟一的例外。

由于标准序列容器（**vector**、**list**和**deque**，它们将在第7章中深入讨论）都有一个默认的分配器参数，上面讲解到的技术能帮助实现我们曾经有过的一个想法：传递这些序列容器中的一个作为模板参数。下面的程序分别传递**vector**模板类型参数和**list**模板类型参数创建了**Container**的两个实例:

237

```
//: C05:TempTemp4.cpp {-bor}{-msc}
// Passes standard sequences as template arguments.
#include <iostream>
#include <list>
#include <memory>  // Declares allocator<T>
#include <vector>
using namespace std;

template<class T, template<class U, class = allocator<U> >
         class Seq>
class Container {
  Seq<T> seq; // Default of allocator<T> applied implicitly
public:
  void push_back(const T& t) { seq.push_back(t); }
  typename Seq<T>::iterator begin() { return seq.begin(); }
  typename Seq<T>::iterator end() { return seq.end(); }
};

int main() {
  // Use a vector
  Container<int, vector> vContainer;
  vContainer.push_back(1);
  vContainer.push_back(2);
  for(vector<int>::iterator p = vContainer.begin();
      p != vContainer.end(); ++p) {
    cout << *p << endl;
  }
  // Use a list
  Container<int, list> lContainer;
  lContainer.push_back(3);
  lContainer.push_back(4);
  for(list<int>::iterator p2 = lContainer.begin();
      p2 != lContainer.end(); ++p2) {
    cout << *p2 << endl;
  }
} ///:~
```

这里命名了内部模板**Seq**的第1个参数（使用名字**U**），这是因为标准序列容器的分配器必须使用与序列容器中所包含对象的类型相同的类型对自己进行参数化。同时，还由于默认的

allocator参数是已知的，就可以像在前述程序中一样，在随后引用的**Seq<T>**中省略掉它。然而，要想彻底地解释清楚这个例子，还必须讨论一下**typename**这个关键字的语义：

5.1.4 typename关键字

238

考虑下面的程序：

```
//: C05:TypenamedID.cpp {-bor}
// Uses 'typename' as a prefix for nested types.

template<class T> class X {
  // Without typename, you should get an error:
  typename T::id i;
public:
  void f() { i.g(); }
};

class Y {
public:
  class id {
  public:
    void g() {}
  };
};

int main() {
  X<Y> xy;
  xy.f();
} ///:~
```

这个模板定义假定，处理的类**T**必须拥有某种称为**id**的嵌套标识符。**id**也可以是一个**T**的静态数据成员，这样就可以直接对**id**进行操作，但却不能“创建类型**id**”的“某个对象”，在这个例子中，标识符**id**被当作**T**内的一个嵌套类型处理。至于类**Y**，**id**本来就是它的一个嵌套类型（没有**typename**关键字），但编译器在编译类**X**的时候却根本不知道这些。

当模板中出现一个标识符时，若编译器可以在把这个标识符当作一个类型，或把它当作一个除类型之外的其他元素之间进行选择的话，则编译器将不会认为这个标识符是一个类型。也就是说，它会认为这个标识符指的是一个对象（其中包括那些基本类型的变量），或者是一个枚举，或者是其他什么。但是它绝不会——也不可能——认为它是一个类型。

由于在上述两种情况下，编译器默认的行为不会认为一个标识符名称是一个类型，因此必须对嵌套的名称使用**typename**关键字进行说明（除了在构造函数的初始化列表中，这时它的出现既不是必要的也不是允许的）。在上例中，当编译器看到**typename T::id**，它就会明白（由于关键字**typename**）**id**指的是一个嵌套类型，之后它就可以创建一个这个类型的对象了。 239

这个规则的简化叙述就是：若一个模板代码内部的某个类型被模板类型参数所限定，则必须使用关键字**typename**作为前缀进行声明，除非它已经出现在基类的规格说明中，或者它出现在同一作用域范围内的初始化列表中（这种情况下一定不要使用**typename**关键字）。

上面解释了关键字**typename**在程序**TempTemp4.cpp**中的使用。没有它，编译器就不会认为**Seq<T>::iterator**表达式是一个类型，而在程序中却要用它来定义成员函数**begin( )**和**end( )**的返回类型。

下面的例子定义了一个函数模板，它能够打印任意标准C++序列容器中的数据，这个例子使用了与**typename**类似的一种用法：

```
//: C05:PrintSeq.cpp {-msc}{-mwcc}
// A print function for Standard C++ sequences.
#include <iostream>
#include <list>
#include <memory>
#include <vector>
using namespace std;

template<class T, template<class U, class = allocator<U> >
         class Seq>
void printSeq(Seq<T>& seq) {
  for(typename Seq<T>::iterator b = seq.begin();
      b != seq.end();)
    cout << *b++ << endl;
}

int main() {
  // Process a vector
  vector<int> v;
  v.push_back(1);
  v.push_back(2);
  printSeq(v);
  // Process a list
  list<int> lst;
  lst.push_back(3);
  lst.push_back(4);
  printSeq(lst);
} ///:~
```

240

同前面一样，若没有**typename**关键字，编译器就会把**iterator**看作是**Seq<T>**的一个静态数据成员，这是一个语法错误，因为这里要求它是一个类型。

1. 创建一个新类型

有一点很重要：一定不能认为关键字**typename**创建了一个新类型名。它确实没有。它的目的就是要通知编译器，被限定的那个标识符应该被解释为一个类型。请看下面这行语句：

```
typename Seq<T>::iterator It;
```

它产生一个名为**It**的变量，该变量被声明为**Seq<T>::iterator**类型。若想创建一个新类型名，通常应该使用关键字**typedef**，如下所示：

```
typedef typename Seq<It>::iterator It;
```

2. 用typename代替class

关键字**typename**的另一个作用是，可以在模板定义的模板参数列表中选择使用**typename**代替**class**：

```
//: C05:UsingTypename.cpp
// Using 'typename' in the template argument list.

template<typename T> class X {};

int main() {
  X<int> x;
} ///:~
```

对大多数程序而言，这种描述方式使得代码更加一目了然。

5.1.5 以template关键字作为提示

当一个类型标识符不是预期的标识符时，正好**typename**关键字可以帮助编译器识别它们，

但编译器却还存在一些潜在的困难，比如'<'字符和'>'字符，它们不是标识符而是标记号(token)。有时它们代表小于号或大于号，而有时它们又作为模板参数列表的界定符。在这里，再次用**bitset**类作为例子来说明这个问题: 241

```
//: C05:DotTemplate.cpp
// Illustrate the .template construct.
#include <bitset>
#include <cstddef>
#include <iostream>
#include <string>
using namespace std;

template<class charT, size_t N>
basic_string<charT> bitsetToString(const bitset<N>& bs) {
  return bs. template to_string<charT, char_traits<charT>,
                                allocator<charT> >();
}

int main() {
  bitset<10> bs;
  bs.set(1);
  bs.set(5);
  cout << bs << endl; // 0000100010
  string s = bitsetToString<char>(bs);
  cout << s << endl;  // 0000100010
} ///:~
```

类**bitset**通过它的**to_string**成员函数支持向字符串对象的转换。为了支持向多种字符串类的转换，**to_string**本身就做成了一个模板，它是根据第3章讨论的**basic_string**模板模式创建的。**bitset**中的**to_string**的声明如下所示:

```
template<class charT, class traits, class Allocator>
basic_string<charT, traits, Allocator> to_string() const;
```

上面的**bitsetToString( )**函数模板可以要求用不同类型的字符串表示**bitset**。例如，若想获得一个宽字符串，可以改写成如下的调用形式:

```
wstring s = bitsetToString<wchar_t>(bs);
```

注意，**basic_string**使用了默认的模板参数，这样在返回值中就不必重复**char_traits**和**allocator**参数。遗憾的是，**bitset::to_string**没有使用默认参数。使用**bitsetToString<char>(bs)**比每次都写出长长的完全地限制调用**bs.template to_string<char, char_traits, allocator<char> >( )**要方便得多。 242

bitsetToString( )的返回语句中包含了**template**关键字，有趣的是，它位于作用在**bitset**对象**bs**上的点运算符之后的右边位置。使用这个关键字的原因是，如果解析这个模板，**to_string**标记之后的 < 字符就会被解释为一个小于号而不是一个模板参数列表的开始标记。这里的**template**关键字的使用会告诉编译器，紧接着的是一个模板名称，这样 < 字符就会被正确地解释出来。基于同样的原因，这种用法也会用在运用于模板中的->和::的运算符上。与**typename**关键字一样，这种模板解析技术仅仅能用于模板代码[⊖]中。

5.1.6 成员模板

bitset::to_string( )函数模板是一个成员模板的例子: 在另一个类或者类模板中声明的

⊖ C++标准协会正在考虑解除这些解析提示仅仅适用于模板中的规则的限制，有一些编译器已经允许将它们用于非模板代码中。

一个模板。它允许一些独立的模板参数结合，以便组合使用。标准C++库中的**complex**类模板就是一个有用的例子。**complex**模板有一个类型参数，它代表一个拥有复数的实部和虚部的基础浮点类型。下面的代码片断是从标准库中摘录出来的，它说明了在**complex**类模板中的成员模板构造函数：

```
template<typename T> class complex {
public:
  template<class X> complex(const complex<X>&);
```

标准的**complex**模板使用已有的类型如**float**、**double**和**long double**等对参数**T**进行特化。上面的成员模板构造函数创建了一个新复数，这个复数使用了另外一个浮点类型作为它的基类型，如下所示：

243

```
complex<float> z(1, 2);
complex<double> w(z);
```

在**w**的声明中，**complex**模板参数**T**是**double**类型，**X**是**float**类型。成员模板使得这种灵活的变换更加容易。

在模板中定义另一个模板是一种嵌套操作，如果想在外部类的定义之外定义成员模板，那么作为引入模板的前缀必须能够反映这种嵌套。例如，如果要实现**complex**类的模板，还想在**complex**模板类定义之外定义成员模板构造函数，可以如下定义：

```
template<typename T>
template<typename X>
complex<T>::complex(const complex<X>& c) {/* Body here… */}
```

标准库中成员函数模板的另一个应用是在容器的初始化中。假设有一个**int**型的**vector**，要想用它初始化一个新的**double**型的**vector**，如下所示：

```
int data[5] = { 1, 2, 3, 4, 5 };
vector<int> v1(data, data+5);
vector<double> v2(v1.begin(), v1.end());
```

只要**v1**中的元素与**v2**中的元素类型兼容（这里就是**double**型和**int**型）即可。**vector**类模板有如下成员模板构造函数：

```
template<class InputIterator>
vector(InputIterator first, InputIterator last,
       const Allocator& = Allocator());
```

这个构造函数在**vector**声明中使用了两次。**v1**用**int**型数组进行初始化时，**InputIterator**的类型是**int***。**v2**使用**v1**进行初始化时，使用了成员模板构造函数的一个实例，用**InputIterator**表示**vector<int>::iterator**。

成员模板也可以是类（不一定必须是函数）。下面的例子说明了一个外部类模板内的成员类模板：

244

```
//: C05:MemberClass.cpp
// A member class template.
#include <iostream>
#include <typeinfo>
using namespace std;

template<class T> class Outer {
public:
  template<class R> class Inner {
  public:
    void f();
  };
```

```
};

template<class T> template<class R>
void Outer<T>::Inner<R>::f() {
  cout << "Outer == " << typeid(T).name() << endl;
  cout << "Inner == " << typeid(R).name() << endl;
  cout << "Full Inner == " << typeid(*this).name() << endl;
}

int main() {
  Outer<int>::Inner<bool> inner;
  inner.f();
} ///:~
```

在第8章中将会详细阐述**typeid**运算符，它只有一个参数并返回一个**type_info**对象，这个对象的**name( )**函数生成一个表示参数类型的字符串。例如，**typeid(int).name( )**返回字符串“**int**”（实际的返回值与具体的操作平台有关）。**typeid**运算符也可以用一个表达式作参数，返回一个代表这个表达式类型的**type_info**对象，例如，对于**int i**，**typeid(i).name( )**返回的内容类似“**int**,”而**typeid(& i).name( )**返回的内容类似“**int***”。

上述程序的输出应该如下所示：

```
Outer == int
Inner == bool
Full Inner == Outer<int>::Inner<bool>
```

主程序中变量**inner**的声明同时实例化了**Inner<bool>**和**Outer<int>**。

成员模板函数不能被声明为**virtual**类型。当今的编译器技术在解析一个类时，希望知道这个类的虚函数表的大小。如果允许虚成员模板函数的存在，则需要提前知道程序中所有这些成员函数的调用在什么地方。这是很不灵活的，尤其是在多文件项目中。 245

5.2 有关函数模板的几个问题

正如一个类模板描述了一族类，一个函数模板描述了一族函数。产生每种模板类型的语法本质上是相同的，只是在如何使用上有点区别。当在实例化类模板时总是需要使用尖括号并且提供所有的非默认模板参数。然而，对于函数模板，经常可以省略掉模板参数，甚至根本不允许使用默认模板参数。仔细看一下在**<algorithm>**头文件中声明的**min( )**函数模板的实现，如下所示：

```
template<typename T> const T& min(const T& a, const T& b) {
  return (a < b) ? a : b;
}
```

可以通过提供尖括号里面的参数类型来调用这个模板，正如对类模板的操作一样，如下所示：

```
int z = min<int>(i, j);
```

这个语法告诉编译器，**min( )**模板需要在参数**T**的位置使用**int**来进行特化，这样编译器就会产生相应的代码。依据从类模板中产生类的命名模式，可以认为这个实例化的函数名称是**min<int>( )**。

5.2.1 函数模板参数的类型推断

可以像上面的例子一样，一直使用这样明确的函数模板特化方式，但是如果可以不明确指定模板参数类型，而让编译器从函数的参数中推断出它们的类型将会更方便，如下所示：

```
int z = min(i, j);
```

246 如果i和j都是int类型，编译器会知道程序需要的是min<int>( )，之后它会自动进行实例化。由于在模板定义时指定了惟一的模板类型参数用于函数的两个参数，因此这两个参数的类型必须一致。对于由一个模板参数来限定类型的函数参数，C++系统不能提供标准转换。例如，若想在两个不同类型的参数（一个int型和一个double型）中找出其中的最小值，下面的这种min( )调用将会出错：

```
int z = min(x, j); // x is a double
```

由于x和j是不同的类型，没有单一的参数与min( )定义中的模板参数T匹配，因此这个调用与模板声明也不匹配。要解决这个问题，可以将一个参数的类型转换为另一个参数的类型，或者恢复到完全说明调用语法，如下所示：

```
int z = min<double>(x, j);
```

该语句告诉编译器产生double版本的min( )，之后j通过标准类型转换规则向上转型成double型数据（因为函数min<double>(const double&, const double&)会自动产生转换）。

也可以要求min( )的两个参数类型完全独立，如下所示：

```
template<typename T, typename U>
const T& min(const T& a, const U& b) {
  return (a < b) ? a : b;
}
```

这通常会是一个好办法，但由于min( )必须返回一个值，却没有一个理想的方式来决定这个返回值的类型到底是T还是U，因此这里的这个“好办法”还是有问题的。

若一个函数模板的返回类型是一个独立的模板参数，当调用它的时候就一定要明确指定它的类型，因为这时已经无法从函数参数中推断出它的类型了。下面的fromString模板就是这样的例子。

```
//: C05:StringConv.h
// Function templates to convert to and from strings.
#ifndef STRINGCONV_H
#define STRINGCONV_H
#include <string>
#include <sstream>

template<typename T> T fromString(const std::string& s) {
  std::istringstream is(s);
  T t;
  is >> t;
  return t;
}

template<typename T> std::string toString(const T& t) {
  std::ostringstream s;
  s << t;
  return s.str();
}
#endif // STRINGCONV_H ///:~
```

247

这些函数模板提供了std::string与任意类型之间的转换，二者分别给出了一个流类插入符和提取符。下面是一个使用了包含在标准库中复数（complex）类的测试程序：

```
//: C05:StringConvTest.cpp
#include <complex>
#include <iostream>
#include "StringConv.h"
using namespace std;
```

```
int main() {
  int i = 1234;
  cout << "i == \"" << toString(i) << "\"" << endl;
  float x = 567.89;
  cout << "x == \"" << toString(x) << "\"" << endl;
  complex<float> c(1.0, 2.0);
  cout << "c == \"" << toString(c) << "\"" << endl;
  cout << endl;

  i = fromString<int>(string("1234"));
  cout << "i == " << i << endl;
  x = fromString<float>(string("567.89"));
  cout << "x == " << x << endl;
  c = fromString<complex<float> >(string("(1.0,2.0)"));
    cout << "c == " << c << endl;
  } ///:~
```

248

输出和预期的结果相同:

```
i == "1234"
x == "567.89"
c == "(1,2)"

i == 1234
x == 567.89
c == (1,2)
```

注意，在每一个**fromString( )**的实例化调用中，都指定了模板参数。如果有一个函数模板，它的模板参数既作为参数类型又作为返回类型，那么一定要首先声明函数的返回类型参数，否则就不能省略掉函数参数表中的任何类型参数。作为一个示例，看看下面这个著名的函数模板: [⊖]

```
//: C05:ImplicitCast.cpp

template<typename R, typename P>
R implicit_cast(const P& p) {
  return p;
}

int main() {
  int i = 1;
  float x = implicit_cast<float>(i);
  int j = implicit_cast<int>(x);
  //! char* p = implicit_cast<char*>(i);
} ///:~
```

如果将程序中靠近文件顶部的模板参数列表中的**R**和**P**交换一下，这个程序将不能通过编译，这是因为没有指定函数的返回类型（第1个模板参数将作为函数的参数类型）。最后一行（被注解掉的）也是不合法的用法，原因是没有从**int**到**char***的标准类型转换。**implicit_cast**显示了代码中允许的类型转换。 249

稍加注意，甚至可以用这种办法推断出数组的维数。下面的例子中有一个数组初始化函数模板（**init2**），它进行了这样的推断:

```
//: C05:ArraySize.cpp
#include <cstddef>
using std::size_t;

template<size_t R, size_t C, typename T>
```

⊖ 参见Stroustrup，《The C++ Programming Language》，第3版，Addison Wesley，第335~336页。

```
void init1(T a[R][C]) {
  for(size_t i = 0; i < R; ++i)
    for(size_t j = 0; j < C; ++j)
      a[i][j] = T();
}

template<size_t R, size_t C, class T>
void init2(T (&a)[R][C]) {  // Reference parameter
  for(size_t i = 0; i < R; ++i)
    for(size_t j = 0; j < C; ++j)
      a[i][j] = T();
}

int main() {
  int a[10][20];
  init1<10,20>(a);  // Must specify
  init2(a);         // Sizes deduced
} ///:~
```

数组维数没有被作为函数参数类型的一部分进行传递，除非这个参数是指针或引用。函数模板**init2**声明了**a**是一个二维数组的引用，因此它的维数**R**和**C**可由模板很容易地推断出来，这样就使得**init2**可以非常方便地初始化一个任意大小的二维数组。模板**init1**不是通过引用来传递数组，因此数组的大小必须被明确指定，尽管也可以推断出它的类型参数。

5.2.2 函数模板重载

像普通函数一样，也可以用相同的函数名重载函数模板。编译器在处理程序中的函数调用时，它必须能够知道哪一个模板或普通函数是最适合调用的函数。

250 接着前面介绍的**min( )**函数模板，在这里再添加几个普通函数：

```
//: C05:MinTest.cpp
#include <cstring>
#include <iostream>
using std::strcmp;
using std::cout;
using std::endl;

template<typename T> const T& min(const T& a, const T& b) {
  return (a < b) ? a : b;
}

const char* min(const char* a, const char* b) {
  return (strcmp(a, b) < 0) ? a : b;
}

double min(double x, double y) {
  return (x < y) ? x : y;
}

int main() {
  const char *s2 = "say \"Ni-!\"", *s1 = "knights who";
  cout << min(1, 2) << endl;       // 1: 1 (template)
  cout << min(1.0, 2.0) << endl;   // 2: 1 (double)
  cout << min(1, 2.0) << endl;     // 3: 1 (double)
  cout << min(s1, s2) << endl;     // 4: knights who (const
                                   //                 char*)
  cout << min<>(s1, s2) << endl;   // 5: say "Ni-!"
                                   //    (template)
} ///:~
```

除了函数模板，这个程序还定义了两个非模板函数：一个C语言风格的字符串版本和一个**double**版本的**min( )**函数。若这个程序中不存在函数模板，上面主函数第1行的函数调用将会调用**double**版本的**min( )**函数，这是由于**int**型可以经标准转换为**double**型。由于模板能够产生一个**int**版本的**min( )**函数，这肯定是最佳的匹配，因此事实上就是这样进行的。第2行中的调用是一个**double**版本的**min( )**函数的准确匹配，第3行也调用了同一个函数，只是在内部将1转变成1.0。第4行中，直接调用了**min( )**函数的**const char***版本。第5行在函数名后加一对空的尖括号来强迫编译器使用模板，因此编译器从模板中生成它的一个**const char***版本来使用（从它的错误输出可以证实——这个函数比较了两个字符串的地址！[⊖]）。如果想知道为什么在应该用**using namespace std**的地方使用了几个**using**声明，这是因为有些编译器在其中包含了**std::min( )**的头文件，这将会与在程序中命名的**min( )**声明发生冲突。 251

如上所述，只要编译器能够区分开，就可以重载同名的模板。例如可以声明一个包含3个参数的**min( )**函数模板：

```
template<typename T>
const T& min(const T& a, const T& b, const T& c);
```

这个模板版本仅仅是为了调用带有3个同类型参数的**min( )**函数而生成的。

5.2.3 以一个已生成的函数模板地址作为参数

在很多情况下需要获得一个函数的地址。例如，可以生成一个函数，它的参数是一个指向另一个函数的指针。此处的另一个函数有可能就是由一个函数模板生成的，因此需要以某种方式来处理这种以函数模板的地址做参数的情况：[⊖]

```
//: C05:TemplateFunctionAddress.cpp {-mwcc}
// Taking the address of a function generated
// from a template.

template<typename T> void f(T*) {}

void h(void (*pf)(int*)) {}

template<typename T> void g(void (*pf)(T*)) {}

int main() {
  h(&f<int>); // Full type specification
  h(&f); // Type deduction
  g<int>(&f<int>); // Full type specification
  g(&f<int>); // Type deduction
  g<int>(&f); // Partial (but sufficient) specification
} ///:~
```

252

这个例子说明了几个问题。首先，既然使用模板，所有的标识就必须匹配。函数**h( )**有一个指针参数，这个指针指向一个函数——它有一个**int***型参数，返回值类型为**void**。这个函数就是模板**f( )**生成的函数。其次，拥有一个函数指针作参数的函数本身可以是一个模板，如本例中的函数模板**g( )**。

也可以在**main( )**中看到类型推断。第1个对**h( )**的调用明确地给出了**f( )**的模板参数，但

⊖ 从技术上说，对不在同一个数组中的两个指针的比较是一种不明确的行为，但如今的编译器不再对此做出解释。所有要这样做的理由都认为是正确的。

⊖ 感谢Nathan Myers提供了这个例子。

由于**h( )**规定只接收具有**int***参数的函数地址作参数，因此第2个调用由编译器来推断类型。至于**g( )**，它的情况就更加有趣了，因为它在其中引用了两个模板。如果什么都不给，编译器就推断不出类型；但若说明了一个**int**，或者赋予**f( )**或者赋予**g( )**，余下的类型编译器自己就能够推断出来。

当想把在**<cctype>**中声明的**tolower**或**toupper**传递给函数做参数时，就会出现一个模糊的问题。例如，在编程中，有可能使用它们和**transform**算法（将在下一章详细阐述）将一个字符串转变成小写或者大写。必须小心使用，因为存在多个有关这些函数的声明。一个初学者的使用方法可能会像下面这样：

```
// The variable s is a std::string
transform(s.begin(), s.end(), s.begin(), tolower);
```

transform算法的第4个参数（在这个例子中就是**tolower( )**）作用到字符串**s**中的每一个字符上，这个算法还把结果写回**s**中，也就是将**s**中的每一个字符都用它的小写形式进行重写。作为一条语句它写在了那里，但这个语句可能执行了也可能根本就没有执行！在下面的情况下它就执行失败了：

```
//: C05:FailedTransform.cpp {-xo}
#include <algorithm>
#include <cctype>
#include <iostream>
#include <string>
using namespace std;
int main() {
  string s("LOWER");
  transform(s.begin(), s.end(), s.begin(), tolower);
  cout << s << endl;
} ///:~
```

253

即使编译器让这个程序侥幸通过，它也是不合法的。原因是**<iostream>**头文件中也建造了可利用的具有两个参数的**tolower( )**和**toupper( )**版本：

```
template<class charT> charT toupper(charT c,
                                    const locale& loc);
template<class charT> charT tolower(charT c,
                                    const locale& loc);
```

这两个函数模板的第2个参数是**locale**类型参数。在上面的程序中，编译器无法得知它应该使用**<cctype>**中定义的具有一个参数的**tolower( )**版本还是上述的版本。可以（几乎可以！）用**transform**调用中的类型转换来解决这个问题，如下所示：

```
transform(s.begin(),s.end(),s.begin()
          static_cast<int (*)(int)>(tolower));
```

（用**int**代替**char**后重新调用**tolower( )**和**toupper( )**函数执行。）上面的类型转换很清楚地表达了想要使用具有一个参数的**tolower( )**函数版本的期望。这种做法对某些编译器可能会成功，但并不是所有的编译器都是如此。其原因有点晦涩难懂：在C语言中允许一个库的实现连接“C连接”（意味着函数名称不包含所有的辅助信息[⊖]，而正常的C++函数却包含）到从C语言继承过来的函数。如果是这样话，类型转换则失败：因为**transform**是一个C++函数模板，它期待它的第4个参数进行C++连接——并且类型转换也不允许改变这种连接。我们又陷入了困境！

⊖　例如在一个修饰的名称中被编码的类型信息。

解决的办法是在一个语义明确的语境中调用**tolower( )**。例如，可以编写一个名叫**strTolower( )**的函数，将它放在一个不包含**<iostream>**的独立的文件中，如下所示：

```
//: C05:StrTolower.cpp (0) {-mwcc}
#include <algorithm>
#include <cctype>
#include <string>
using namespace std;

string strTolower(string s) {
  transform(s.begin(), s.end(), s.begin(), tolower);
  return s;
} ///:~
```

254

该程序没有包含头文件**<iostream>**，在这种语境中，程序使用的编译器就不会引入带有两个参数的**tolower( )**版本[⊖]，当然也就不会产生任何问题。经过这样处理之后，就可以正常使用这个函数了：

```
//: C05:Tolower.cpp {-mwcc}
//{L} StrTolower
#include <algorithm>
#include <cctype>
#include <iostream>
#include <string>
using namespace std;
string strTolower(string);

int main() {
  string s("LOWER");
  cout << strTolower(s) << endl;
} ///:~
```

另一个解决办法是写一个经过封装的函数模板，用来清楚地调用正确的**tolower( )**版本：

```
//: C05:ToLower2.cpp {-mwcc}
#include <algorithm>
#include <cctype>
#include <iostream>
#include <string>
using namespace std;

template<class charT> charT strTolower(charT c) {
  return tolower(c);  // One-arg version called
}

int main() {
  string s("LOWER");
  transform(s.begin(),s.end(),s.begin(),&strTolower<char>);
  cout << s << endl;
} ///:~
```

255

这种版本有一个好处，即由于基础字符类型是一个模板参数，因而该模板既可以处理宽字符串，也可以处理窄字符串。C++标准委员会正致力于修订语言，使得第1个例子（没有使用类型转换的）能够执行，也许过不了多久这些外围工作就可以被忽略了（由C++标准来完成）。[⊜]

⊖ C++编译器能够引入它们想要的任何地方的名称，然而幸运的是，大多数编译器对自己不需要的名称不会进行声明。

⊜ 若对关于C++语言修改的建议感兴趣，可以参看Core Issue 352。

5.2.4 将函数应用到STL序列容器中

假设要想获得一个STL序列容器（更多的内容将在后面的章节中学习；现在只用到STL序列容器家族中的**vector**），并且想将一个成员函数应用到这个容器包含的所有对象中。因为一个**vector**可以包含任意类型的对象，这就需要一个可以应用到任意类型的**vector**对象的函数：

```
//: C05:ApplySequence.h
// Apply a function to an STL sequence container.

// const, 0 arguments, any type of return value:
template<class Seq, class T, class R>
void apply(Seq& sq, R (T::*f)() const) {
  typename Seq::iterator it = sq.begin();
  while(it != sq.end())
    ((*it++)->*f)();
}

// const, 1 argument, any type of return value:
template<class Seq, class T, class R, class A>
void apply(Seq& sq, R(T::*f)(A) const, A a) {
  typename Seq::iterator it = sq.begin();
  while(it != sq.end())
    ((*it++)->*f)(a);
}
256
// const, 2 arguments, any type of return value:
template<class Seq, class T, class R,
         class A1, class A2>
void apply(Seq& sq, R(T::*f)(A1, A2) const,
    A1 a1, A2 a2) {
  typename Seq::iterator it = sq.begin();
  while(it != sq.end())
    ((*it++)->*f)(a1, a2);
}

// Non-const, 0 arguments, any type of return value:
template<class Seq, class T, class R>
void apply(Seq& sq, R (T::*f)()) {
  typename Seq::iterator it = sq.begin();
  while(it != sq.end())
    ((*it++)->*f)();
}

// Non-const, 1 argument, any type of return value:
template<class Seq, class T, class R, class A>
void apply(Seq& sq, R(T::*f)(A), A a) {
  typename Seq::iterator it = sq.begin();
  while(it != sq.end())
    ((*it++)->*f)(a);
}

// Non-const, 2 arguments, any type of return value:
template<class Seq, class T, class R,
         class A1, class A2>
void apply(Seq& sq, R(T::*f)(A1, A2),
    A1 a1, A2 a2) {
  typename Seq::iterator it = sq.begin();
  while(it != sq.end())
    ((*it++)->*f)(a1, a2);
}
// Etc., to handle maximum likely arguments ///:~
```

上面的**apply( )**函数模板有一个对容器类进行引用的引用参数，还有一个指针参数，它指向容器类中包含的对象的一个成员函数。这个模板使用一个迭代器遍历该序列，并且在每个对象上调用这个成员函数。现在已经重载了**const**版本的函数模板，这样一来，在**const**和非**const**函数中就都可以调用这个成员函数了。

注意，在**applySequence.h**中没有包含STL头文件（也没有包含其他的头文件），因此它并不局限于只能用于STL容器。然而，它的假设（主要是由于**iterator**的名称及行为）确实是用于STL序列容器中，而且它也假设容器的元素都是指针类型。 257

读者可以看到有多个**apply( )**版本，更进一步地说明了函数模板的重载。尽管这些模板允许返回任意类型的值（这点被忽略了，但类型信息要求匹配指向成员函数的指针），每一个版本都带有不同个数的参数，并且由于这是模板，因此它们的参数可以是任意类型。在此惟一的缺陷，就是没有提供一个可以产生模板的“超模板”；读者必须亲自决定究竟需要几个参数来选用合适的模板定义。

为了测试一下**apply( )** 的这几个重载版本，这里创建了一个类**Gromit**[⊖]，它包含几个带有不同个数参数的函数，以及由**const**和非**const**的成员函数：

```
//: C05:Gromit.h
// The techno-dog. Has member functions
// with various numbers of arguments.
#include <iostream>

class Gromit {
  int arf;
  int totalBarks;
public:
  Gromit(int arf = 1) : arf(arf + 1), totalBarks(0) {}
  void speak(int) {
    for(int i = 0; i < arf; i++) {
      std::cout << "arf! ";
      ++totalBarks;
    }
    std::cout << std::endl;
  }
  char eat(float) const {
    std::cout << "chomp!" << std::endl;
    return 'z';
  }
  int sleep(char, double) const {
    std::cout << "zzz..." << std::endl;
    return 0;
  }
  void sit() const {
    std::cout << "Sitting..." << std::endl;
  }
}; ///:~
```

258

现在，可以用**apply( )**模板函数来调用**vector<Gromit*>**对象中包含的**Gromit**的成员函数了，如下所示：

```
//: C05:ApplyGromit.cpp
// Test ApplySequence.h.
#include <cstddef>
#include <iostream>
```

⊖ 参考由Nick Park 出品的描写Wallace和Gromit英国动画短片。

```
#include <vector>
#include "ApplySequence.h"
#include "Gromit.h"
#include "../purge.h"
using namespace std;

int main() {
  vector<Gromit*> dogs;
  for(size_t i = 0; i < 5; i++)
    dogs.push_back(new Gromit(i));
  apply(dogs, &Gromit::speak, 1);
  apply(dogs, &Gromit::eat, 2.0f);
  apply(dogs, &Gromit::sleep, 'z', 3.0);
  apply(dogs, &Gromit::sit);
  purge(dogs);
} ///:~
```

purge( )函数是一个小型的实用程序，该函数调用**delete**来清除序列上的所有元素。读者将会在第7章找到它的定义，并且本教材在许多的地方都会用到它。

尽管**apply( )**的定义有点复杂，而且不像读者所希望的那样使初学者都可以理解，但是它使用起来却非常简单明了，初学者也可以在使用中知道它企图完成什么功能，而不必知道它是如何完成的。这也是所有程序组件应该追求的目标。复杂的细节由程序组件的设计者来完成。而用户只关心完成他们的目标，他们不必看、不必了解、也不依赖那些底层实现的细节。在下一章中要探索将函数应用到序列容器中的更灵活的方式。

259

5.2.5 函数模板的半有序

前面曾经提到过，使用像**min( )**函数这样的普通函数重载，比使用函数模板更可取。如果一个函数可以匹配某个函数调用，为什么还要再生成另外一个函数呢？在缺少普通函数时，对函数模板进行重载有可能引起二义性（ambiguity）的情况，即不知选择哪个模板。为了将发生这种情况的几率减到最低，系统为这些函数模板定义了*次序*（ordering），在生成模板函数的时候，编译器将从这些函数模板中选择*特化程度最高*（most specialized）的模板（如果有这种模板的话）。一个函数模板要考虑多种特化，在这些特化的模板中对于某个特定的函数模板来说，如果每一种可能的参数列表的选择都能够匹配该模板的参数列表，那么，这些可能的参数列表选择也都能够匹配另一个函数模板的参数列表，但反过来却不成立。请看下面的函数模板声明，取自C++标准文档：

```
template<class T> void f(T);
template<class T> void f(T*);
template<class T> void f(const T*);
```

任何类型都可以匹配第1个模板。第2个模板比第1个模板的特化程度更高，因为只有指针类型才能够匹配它。换句话说，可以把匹配第2个模板的一组可能的函数调用看作是匹配第1个模板的子集。上面的第2个模板和第3个模板的声明也存在类似的关系：第3个仅仅能被指向**const**的指针匹配调用，但第2个模板包含了任意的指针类型。下面的程序说明了这些规则：

```
//: C05:PartialOrder.cpp
// Reveals ordering of function templates.
#include <iostream>
using namespace std;

template<class T> void f(T) {
  cout << "T" << endl;
}
```

```
  template<class T> void f(T*) {
    cout << "T*" << endl;
  }
template<class T> void f(const T*) {
  cout << "const T*" << endl;
}

int main() {
  f(0);             // T
  int i = 0;
  f(&i);            // T*
  const int j = 0;
  f(&j);            // const T*
} ///:~
```

260

f(&i)调用和第1个模板匹配，但由于第2个模板的特化程度更高，因此这里调用了第2个模板。在此处第3个模板不能被调用，这是因为该指针不是指向**const**的指针。**f(&j)**调用匹配了所有这3个模板（比如，在第2个模板中的**T**就是**const int**），但是由于同样的原因，第3个模板特化程度更高，因此实际上调用了它。

如果在一组重载的函数模板中没有“特化程度最高”的模板，则会出现二义性，编译器将会报错。这就是为什么把这种特征叫做“半有序（partial ordering）”的缘故——它不可能完全解决所有可能出现的情况。类似的规则同样也存在于类模板中（参见5.3.2节）。

5.3 模板特化

术语特化（specialization）在C++中有一个特别的与模板相关的含义。从本质上说，一个模板定义就是一个实体一般化（generalization）的过程，因为它在一般条件下描述了某个范围内的一族函数或类。给定模板参数时，这些模板参数决定了这一族函数或类的许多可能的实例中的一个独一无二的实例，因此这样的结果就被称做模板的一个特化。本章开始时介绍的**min( )**函数模板是一个寻找最小值函数的一般化，因为没有指定它的参数类型。若为这个模板参数提供了类型，不管它是明确给定的还是通过参数推断获得的，由编译器生成的结果代码（例如，**min <int> ( )**）都是这个模板的一个特化。生成的代码也被认为是这个模板的一个实例化（instantiation），就像是由模板工具完全生成它的整个代码体一样。

5.3.1 显式特化

261

编程人员也可以自己为一个模板提供代码来使其特化，采用这种方法进行编码的程序设计人员越来越多。类模板经常需要程序员为它提供模板特化，在本节，将从**min( )**函数模板开始介绍这个语法。

回忆一下本章前面讨论过的**MinTest.cpp**中下面的这个普通函数：

```
const char* min(const char* a, const char* b) {
  return (strcmp(a, b) < 0) ? a : b;
}
```

这是一个比较字符串而不是比较地址的**min( )**调用。尽管这个例子在这里没有什么用处，但可以作为替代为**min( )**定义一个**const char***的特化，如下面的程序所示：

```
//: C05:MinTest2.cpp
#include <cstring>
#include <iostream>
using std::strcmp;
using std::cout;
```

```
using std::endl;

template<class T> const T& min(const T& a, const T& b) {
  return (a < b) ? a : b;
}

// An explicit specialization of the min template
template<>
const char* const& min<const char*>(const char* const& a,
                                    const char* const& b) {
  return (strcmp(a, b) < 0) ? a : b;
}

int main() {
  const char *s2 = "say \"Ni-!\"", *s1 = "knights who";
  cout << min(s1, s2) << endl;
  cout << min<>(s1, s2) << endl;
} ///:~
```

前缀“**template< >**”告诉编译器接下来的是一个模板的特化。模板特化的类型必须出现在函数名后紧跟的尖括号中，就像通常在一个明确指定参数类型的函数调用中一样。注意，程序在这个显式特化（explicit specialization）中仔细地（carefully）用**const char***代替了**T**。一旦在最初的模板定义时使用了**const T**，则这个**const**就会修正所有的**T**类型。这里的**const**常量是一个指向**const char***的指针。因此在模板特化时必须用**const char* const**代替**const T**。当编译器在程序中看到一个带有**const char***参数的**min( )**调用时，它就会去实例化**min( )**的**const char***版本，这样该版本就可以被调用了。这个程序中的两次**min( )**调用都是调用同一个**min( )**的特化版本。

262

类模板显式特化往往比函数模板显式特化更有用。当程序员为一个类模板提供了一份完整的特化时，依然需要实现其中的所有成员函数。这是由于所提供的是一个单独的类，客户代码常常希望能提供完整的接口实现。

标准库中有一个**vector**的显式特化，该特化在它持有**bool**类型的对象时使用。**Vector <bool>**实现的功能是让库将位（bit）封装成整数（integer）来节省存储空间。[⊖]

如前所述，基本**vector**类模板的声明如下：

```
template<class T, class Allocator = allocator<T> >
class vector {...};
```

要为**bool**类型的对象进行特化，如下显式声明该特化：

```
template<> class vector<bool, allocator<bool> > {...};
```

这再一次很快地被认为是一个完整的显式特化，因为有**template< >**前缀，还因为所有基本的模板参数都由令人感到满意的一个参数列表附加到类名中。

结果证明，事实上**vector <bool>** 比前面所描述的模板特化方法具有更好的灵活性，具体内容请参看下节。

263

5.3.2 半特化

类模板也可以半特化（partial specialization），这意味着在模板特化的某些方法中至少还有一个方法，其模板参数是“开放的”。**Vector <bool>** 限定了对象类型（**bool**类型），但并没有指定参数**allocator**的类型。下面是一个实际的**vector <bool>** 声明：

⊖ 第7章将会详细讨论**vector<bool>**。

```
template<class Allocator> class vector<bool, Allocator>;
```

读者可以把它理解成半特化，因为在**template**关键字后面（还没有被指定的参数）和**class**关键字后面（已经指定了参数）的尖括号里都是非空的参数列表。由于**vector <bool>** 以这种方式定义，用户就可以提供一个自定义的**allocator**类型，即使参数列表中包含的类型**bool**是不变的。换句话说，类模板的特化和这种特别的半特化，构成了一种“重载”的类模板。

类模板的半有序

选用哪个类模板来进行实例化的规则类似于函数模板的半有序规则——应该选择“特化程度最高”的模板。在下面的程序中，各个**f( )**成员函数里面的字符串解释了每个模板定义的职责：

```
//: C05:PartialOrder2.cpp
// Reveals partial ordering of class templates.
#include <iostream>
using namespace std;

template<class T, class U> class C {
public:
  void f() { cout << "Primary Template\n"; }
};

template<class U> class C<int, U> {
public:
  void f() { cout << "T == int\n"; }
};

template<class T> class C<T, double> {
public:
  void f() { cout << "U == double\n"; }
};
template<class T, class U> class C<T*, U> {
public:
  void f() { cout << "T* used\n"; }
};

template<class T, class U> class C<T, U*> {
public:
  void f() { cout << "U* used\n"; }
};

template<class T, class U> class C<T*, U*> {
public:
  void f() { cout << "T* and U* used\n"; }
};

template<class T> class C<T, T> {
public:
  void f() { cout << "T == U\n"; }
};

int main() {
  C<float, int>().f();    // 1: Primary template
  C<int, float>().f();    // 2: T == int
  C<float, double>().f(); // 3: U == double
  C<float, float>().f();  // 4: T == U
  C<float*, float>().f(); // 5: T* used [T is float]
  C<float, float*>().f(); // 6: U* used [U is float]
  C<float*, int*>().f();  // 7: T* and U* used [float,int]
  // The following are ambiguous:
//   8: C<int, int>().f();
//   9: C<double, double>().f();
```

264

```
//  10: C<float*, float*>().f();
//  11: C<int, int*>().f();
//  12: C<int*, int*>().f();
} ///:~
```

如同读者看到的，可以根据模板参数是否是指针类型，或者它们是否和特化参数类型相同来部分地指定模板参数。当用**T***作为模板参数来进行模板特化时，如上例的主程序中的第5行，**T**本身不是最高级的被传递的指针类型——它是一个指针指向的类型（本例中是**float**）。**T***特化是一种允许用指针类型来匹配的模式。如果用**int****作为第1个模板参数，**T**就是**int***。主程序中的第8行具有二义性，因为程序中既有把**int**作为第1个参数的模板，也有带有两个类型相同参数的独立模板——在这两个模板中无法进一步进行取舍。同样的逻辑错误存在于第9行到第12行。

265

5.3.3 一个实例

可以很容易地从一个类模板中派生出一个模板，也可以通过继承和实例化一个现存的模板来创建一个新的模板。例如，若**vector**模板已经完成了程序员想要做的绝大多数事情，但在某种应用中，还希望有一个能够自己进行排序的版本，则可以很容易地复用**vector**代码。下面的例子是建立一个**vector < T >** 的派生类，而且添加了排序（sorting）功能。注意，该类派生自**vector**，由于其没有虚析构函数，当需要在析构函数中执行清除操作的时候，这个派生类就会很危险。

```
//: C05:Sortable.h
// Template specialization.
#ifndef SORTABLE_H
#define SORTABLE_H
#include <cstring>
#include <cstddef>
#include <string>
#include <vector>
using std::size_t;

template<class T>
class Sortable : public std::vector<T> {
public:
  void sort();
};

template<class T>
void Sortable<T>::sort() { // A simple sort
  for(size_t i = this->size(); i > 0; --i)
    for(size_t j = 1; j < i; ++j)
      if(this->at(j-1) > this->at(j)) {
        T t = this->at(j-1);
        this->at(j-1) = this->at(j);
        this->at(j) = t;
      }
}
// Partial specialization for pointers:
template<class T>
class Sortable<T*> : public std::vector<T*> {
public:
  void sort();
};

template<class T>
void Sortable<T*>::sort() {
  for(size_t i = this->size(); i > 0; --i)
```

266

```
    for(size_t j = 1; j < i; ++j)
      if(*this->at(j-1) > *this->at(j)) {
        T* t = this->at(j-1);
        this->at(j-1) = this->at(j);
        this->at(j) = t;
      }
}

// Full specialization for char*
// (Made inline here for convenience -- normally you would
// place the function body in a separate file and only
// leave the declaration here).
template<> inline void Sortable<char*>::sort() {
  for(size_t i = this->size(); i > 0; --i)
    for(size_t j = 1; j < i; ++j)
      if(std::strcmp(this->at(j-1), this->at(j)) > 0) {
        char* t = this->at(j-1);
        this->at(j-1) = this->at(j);
        this->at(j) = t;
      }
}
#endif // SORTABLE_H ///:~
```

除了实例化类这个功能之外，**Sortable**模板对所有其他功能都有一个限制：它们必须包含一个>运算符。它只可以正确地处理非指针对象（包括系统内在类型的对象）。完全特化使用**strcmp( )**对元素进行比较，并根据以空结束符来界定字符串的规则来对**char***型的**vector**进行排序。上例中的“**this->**”是一个强制用法[⊖]，它将会在本章后面的“名字查找问题”一节中介绍[⊖]。 267

这里有一个**Sortable.h**的驱动程序，它使用了本章前面介绍过的随机数生成器：

```
//: C05:Sortable.cpp
//{-bor} (Because of bitset in Urand.h)
// Testing template specialization.
#include <cstddef>
#include <iostream>
#include "Sortable.h"
#include "Urand.h"
using namespace std;

#define asz(a) (sizeof a / sizeof a[0])

char* words[] = { "is", "running", "big", "dog", "a", };
char* words2[] = { "this", "that", "theother", };

int main() {
  Sortable<int> is;
  Urand<47> rnd;
  for(size_t i = 0; i < 15; ++i)
    is.push_back(rnd());
  for(size_t i = 0; i < is.size(); ++i)
    cout << is[i] << ' ';
  cout << endl;
  is.sort();
  for(size_t i = 0; i < is.size(); ++i)
    cout << is[i] << ' ';
```

⊖ 可以使用任何其他合法的限定用法来代替**this->**，比如**Sortable::at( )**或者**vector<T>::at( )**。上要是它必须被限定。

⊖ 也可参看第7章中**PriorityQueue6.cpp**的解释。

```
  cout << endl;

  // Uses the template partial specialization:
  Sortable<string*> ss;
  for(size_t i = 0; i < asz(words); ++i)
    ss.push_back(new string(words[i]));
  for(size_t i = 0; i < ss.size(); ++i)
    cout << *ss[i] << ' ';
  cout << endl;
  ss.sort();
  for(size_t i = 0; i < ss.size(); ++i) {
    cout << *ss[i] << ' ';
    delete ss[i];
  }
  cout << endl;

  // Uses the full char* specialization:
  Sortable<char*> scp;
  for(size_t i = 0; i < asz(words2); ++i)
    scp.push_back(words2[i]);
  for(size_t i = 0; i < scp.size(); ++i)
    cout << scp[i] << ' ';
  cout << endl;
  scp.sort();
  for(size_t i = 0; i < scp.size(); ++i)
    cout << scp[i] << ' ';
  cout << endl;
} ///:~
```

268

上面的每一个模板的实例化都使用了该模板的不同版本。**Sortable <int>** 使用的是基本的模板。**Sortable <string*>** 使用了指针类型的半特化版本。最后，**Sortable <char*>** 使用的是**char***类型的完全特化模板。若没有这个完全特化版本，读者也许会误认为一切还会照原样正确无误地执行，因为**words**数组仍能排序出“a big dog is running”，这是由于半特化版本也可以完成每个数组的第1个字符的比较。然而，对于**words2**数组却不能够正确地排序。

5.3.4 防止模板代码膨胀

无论何时，一旦对某个类模板进行了实例化，伴随着所有在程序中调用的该模板的成员函数，类定义中用于对其进行详尽描述的特化代码也就会生成。只有被调用的成员函数才生成代码。这是不错的，读者可以在下面的程序中看到这一点：

```
//: C05:DelayedInstantiation.cpp
// Member functions of class templates are not
// instantiated until they're needed.

class X {
public:
  void f() {}
};

class Y {
public:
  void g() {}
};

template<typename T> class Z {
  T t;
public:
  void a() { t.f(); }
  void b() { t.g(); }
```

269

```
};

int main() {
  Z<X> zx;
  zx.a(); // Doesn't create Z<X>::b()
  Z<Y> zy;
  zy.b(); // Doesn't create Z<Y>::a()
} ///:~
```

在这里，尽管模板Z打算使用T的两个成员函数f( )和g( )，但实际上，当在程序中明确地为zx调用Z<X>::a( )时候，程序在编译时就只能够生成Z<X>::a( )的代码。（若同时也想生成Z<X>::b( ) 的代码，则会产生一个编译时错误信息，因为它试图调用一个并不存在的X::g( ) 。）同样的道理，对zy.b( )的调用也不会生成Z<Y>::a( )。结果是，Z模板可以跟类X和类Y在一起使用，但是，当类在进行第1次实例化时，如果所有的成员函数都生成了，就会使许多模板的使用明显地受到限制。

假设有一个模板化的Stack容器，现在想用int、int*和char*对它进行特化。这样将会生成3个版本的Stack代码，并链接为程序的一部分。使用模板的原因之一，首先就是不必手工复制代码；但是代码仍然被复制了——只不过是编译器代替程序员完成了这个工作而已。可以结合使用完全特化和半特化模板，将指针类型存储到某个独立的类中，这样可以减少程序实现的体积。其关键是用void*进行完全特化，然后从void*实现中派生出所有其他的指针类型，这样共同的代码就可以共享了。下面的程序说明了这个技术：

270

```
//: C05:Nobloat.h
// Shares code for storing pointers in a Stack.
#ifndef NOBLOAT_H
#define NOBLOAT_H
#include <cassert>
#include <cstddef>
#include <cstring>

// The primary template
template<class T> class Stack {
  T* data;
  std::size_t count;
  std::size_t capacity;
  enum { INIT = 5 };
public:
  Stack() {
    count = 0;
    capacity = INIT;
    data = new T[INIT];
  }
  void push(const T& t) {
    if(count == capacity) {
      // Grow array store
      std::size_t newCapacity = 2 * capacity;
      T* newData = new T[newCapacity];
      for(size_t i = 0; i < count; ++i)
        newData[i] = data[i];
      delete [] data;
      data = newData;
      capacity = newCapacity;
    }
    assert(count < capacity);
    data[count++] = t;
  }
  void pop() {
    assert(count > 0);
```

```
    --count;
  }
  T top() const {
    assert(count > 0);
    return data[count-1];
  }
  std::size_t size() const { return count; }
};

// Full specialization for void*
template<> class Stack<void *> {
  void** data;
  std::size_t count;
  std::size_t capacity;
  enum { INIT = 5 };
public:
  Stack() {
    count = 0;
    capacity = INIT;
    data = new void*[INIT];
  }
  void push(void* const & t) {
    if(count == capacity) {
      std::size_t newCapacity = 2*capacity;
      void** newData = new void*[newCapacity];
      std::memcpy(newData, data, count*sizeof(void*));
      delete [] data;
      data = newData;
      capacity = newCapacity;
    }
    assert(count < capacity);
    data[count++] = t;
  }
  void pop() {
    assert(count > 0);
    --count;
  }
  void* top() const {
    assert(count > 0);
    return data[count-1];
  }
  std::size_t size() const { return count; }
};

// Partial specialization for other pointer types
template<class T> class Stack<T*> : private Stack<void *> {
  typedef Stack<void *> Base;
public:
  void push(T* const & t) { Base::push(t); }
  void pop() {Base::pop();}
  T* top() const { return static_cast<T*>(Base::top()); }
  std::size_t size() { return Base::size(); }
};
#endif // NOBLOAT_H ///:~
```

这个简单的栈能根据需要进行容量的扩充。**void***特化通过**template < >** 前缀的功效（就是说，模板参数列表为空）做成了一个优秀的完全特化版本。如前所述，在一个类模板的特化中必然要实现所有的成员函数。这个特征同样存在于所有其他指针类型的类模板的特化中。由于仅仅想用**Stack <void*>** 作为实现目标，而且也不希望将它的任何接口直接暴露给用户，因此半特化只用于从**Stack <void*>** 私有派生出来的其他指针类型。每个指针实例化后的成员函数都是

Stack <void*> 中相应函数的一个有细微改进的函数。因而，无论何时对一个非**void***类型的指针类型进行实例化，它产生的代码只是单独使用基本（primary）模板所产生的代码的一小部分[⊖]。下面是一个驱动程序:

```
//: C05:NobloatTest.cpp
#include <iostream>
#include <string>
#include "Nobloat.h"
using namespace std;

template<class StackType>
void emptyTheStack(StackType& stk) {
  while(stk.size() > 0) {
    cout << stk.top() << endl;
    stk.pop();
  }
}

// An overload for emptyTheStack (not a specialization!)
template<class T>
void emptyTheStack(Stack<T*>& stk) {
  while(stk.size() > 0) {
    cout << *stk.top() << endl;
    stk.pop();
  }
}
int main() {
  Stack<int> s1;
  s1.push(1);
  s1.push(2);
  emptyTheStack(s1);
  Stack<int *> s2;
  int i = 3;
  int j = 4;
  s2.push(&i);
  s2.push(&j);
  emptyTheStack(s2);
} ///:~
```

273

为方便起见，在这个程序中包含了两个**emptyStack**函数模板。由于函数模板不支持半特化，所以程序中提供了重载的函数模板。**emptyStack**的第2个版本比第1个版本的特化程度更高一些，所以当用到指针类型特化函数模板的时候它总是被选用。在这个程序中实例化了3个类模板：**Stack<int>**、**Stack<void*>** 和**Stack<int*>**。由于**Stack<int*>** 派生于**Stack<void*>**，因此**Stack<void*>** 是一种隐式实例化。如果一个程序要为多个指针类型进行实例化就可能产生大量的代码，可以通过一个**Stack**模板来节省大量的代码空间。

5.4 名称查找问题

当编译器碰到一个标识符时，它必须能够确定这个标识符所代表的实体的类型和作用域（如果它是一个变量，就是生存期）。模板的引入增加了这个问题的复杂度。当编译器首次看到一个模板定义时它不知道有关这个模板的任何信息，只有当它看到模板的实例化时，它才能判断这个模板是否被正确地使用了。这种状况导致了模板编译需要分两个阶段进行。

5.4.1 模板中的名称

在第1阶段，编译器解析模板定义，寻找明显的语法错误，还要对它所能解析的所有名称

⊖ 由于这个改进（forwarding）函数是内联的，因此不产生**Stack<void*>**的代码!

进行解析。对于不依赖于模板参数的名称，编译器使用普通名称查找的方法解析它们，如果有必要，编译器也会依赖模板参数进行查找（在后面讨论）。它不能够解析的名称就是所谓的*关联名*（dependent name），这些名称以某种方式依赖于模板参数。只有等到用实际参数来实例化模板的时候，这些名称才能被解析。因此模板编译的第2个阶段就是模板实例化。在这里，由编译器来决定是否使用模板的一个显式特化来取代基本的模板。

274

在看下面的例子之前，必须至少理解两个术语。*限定名*（qualified name）是指具有类名前缀，或者是被一个对象名加上点运算符修饰，或者是被一个指向某一对象的指针加一个箭头运算符所限定的名称修饰。限定名举例如下：

```
MyClass::f();
x.f();
p->f();
```

本教材中多次使用限定名，最近的用法是把它与**typename**关键字相联系。之所以称为限定名是因为这些目标名（如上面例子中的**f**）被明确的与一个类或者与一个名字空间相联系，它们会告诉编译器应该去哪儿寻找这些名称的声明。

另一个术语是*关联参数查找*（argument-dependent lookup[⊖]，ADL），这个机制起初是设计用来简化在名字空间中声明的非成员函数调用（包含运算符）。看下面的代码：

```
#include <iostream>
#include <string>
// ...
  std::string s("hello");
  std::cout << s << std::endl;
```

请注意，在头文件中的典型习惯用法中，没有使用**using namespace std**指令。没有了这条指令，就必须使用“**std::**”来限定**std**名字空间中的每项内容。但是，在这里并没有用它来限定**std**中的所有内容。你能知道哪一个是不合格的吗？

275

在程序中没有指定使用哪一个运算符函数。程序员希望下述事情发生，但却不想键入这些代码：

```
std::operator<<(std::operator<<(std::cout,s),std::endl);
```

为了使最初的输出语句能够按照预先的设计执行，ADL规定：当出现了对某个非限定函数的调用，而该非限定函数却没有在一个（标准）作用域内进行声明时，编译器为了匹配这个函数声明，就会寻找它的每一个参数的名字空间来进行匹配。在最初的语句中，第1个函数调用是：

```
operator<<(std::cout, s);
```

由于在初始引用的名字空间作用域中没有这个函数声明，编译器注意到这个函数的第1个参数（**std::cout**）在名字空间**std**中；因此它就把这个名字空间添加到作用域列表中，以此来寻找一个能完美匹配**operator<<（std::ostream&,std::string）**的独一无二的函数。它通过**<string>**头文件发现这个函数是在**std**名字空间中声明的。

没有ADL，名字空间的使用将会非常的不方便。注意，ADL通常从所有合格的名字空间中引入存有质疑的名称的所有声明——若没有一个最好的匹配，将会产生二义性。

为了避开ADL不用，可以将函数名称置于一对圆括号中：

```
(f)(x, y);  // ADL suppressed
```

⊖ 也称为Koenig查找，因为Andrew Koenig首先向C++标准委员会建议了这种查找技术。ADL用一般概念阐述了模板是否应该包括于其中。

现在来看看下面的程序：[⊖]

```
//: C05:Lookup.cpp
// Only produces correct behavior with EDG,
// and Metrowerks using a special option.
#include <iostream>
using std::cout;
using std::endl;

void f(double) { cout << "f(double)" << endl; }

template<class T> class X {
public:
  void g() { f(1); }
};

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

int main() {
  X<int>().g();
} ///:~
```

276

本程序使用的编译器是支持前端兼容用法的Edison Design Group（EDG）[⊜]，上面的代码在这个编译器上不用修改就能正确执行。某些编译器，例如Metrowerks，可以通过配置选项实现正确的查找。输出结果应该是：

```
f(double)
```

这是因为**f**是一个非关联的名称，它早在定义模板的过程中就已经解析了，那时只有**f（double）**在模板的作用域之中。遗憾的是，在实际的应用系统中存在着大量的依赖非标准行为的不规范代码，即将**g( )**中**f(1)**的调用与其后的**f(int)**绑定在了一起，因此编译器的编写者也就不情愿的去做改动了。

下面是一个更详细的例子：[⊜]

```
//: C05:Lookup2.cpp {-bor}{-g++}{-dmc}
// Microsoft: use option -Za (ANSI mode)
#include <algorithm>
#include <iostream>
#include <typeinfo>
using std::cout;
using std::endl;

void g() { cout << "global g()" << endl; }

template<class T> class Y {
public:
  void g() {
    cout << "Y<" << typeid(T).name() << ">::g()" << endl;
  }
  void h() {
    cout << "Y<" << typeid(T).name() << ">::h()" << endl;
  }
  typedef int E;
```

277

⊖ 这是Herb Sutter奉献的一个程序。

⊜ 很多编译器使用这种前端兼容用法，包括Comeau C++。

⊜ 这也是基于Herb Sutter的一个例子。

```
};

typedef double E;

template<class T> void swap(T& t1, T& t2) {
  cout << "global swap" << endl;
  T temp = t1;
  t1 = t2;
  t2 = temp;
}

template<class T> class X : public Y<T> {
public:
  E f() {
    g();
    this->h();
    T t1 = T(), t2 = T(1);
    cout << t1 << endl;
    swap(t1, t2);
    std::swap(t1, t2);
    cout << typeid(E).name() << endl;
    return E(t2);
  }
};

int main() {
  X<int> x;
  cout << x.f() << endl;
} ///:~
```

这个程序的输出应该是:

```
global g()
Y<int>::h()
0
global swap
double
1
```

看看**X::f( )**中的声明:

278

- **E**，是**X::f( )**的返回类型，它不是一个关联名称，因此在解析模板的时候它就被找到了。编译器找到了**E**后，用**typedef**将**E**命名成一个**double**类型。这种情况可能看起来有点奇怪，因为在非模板类中，**E**在基类中的声明应该先被找到，但那是非模板类的规则。(基类**Y**，是一个关联基类（dependent base class)，因此在模板定义期间不能够找到它)。
- **g( )**的调用也是不依赖参数类型的，因为它没有用到**T**。若**g**带有某些定义在另一个名字空间中的类类型的参数，则ADL就会将这个名字空间接管过来，因为在它的作用域内没有带有这种参数的**g**定义。因此，这个调用匹配了**g( )**的全局声明。
- **this->h( )**调用是一个限定名称调用，限定它的对象（**this**）指的是当前对象，即该当前对象是**X**类型的，**X**通过继承机制又依赖于名称**Y<T>**。**X**中没有函数**h( )**，因此查找将去**X**的基类**Y<T>** 作用域内寻找。由于这是一个关联名称，它在实例化期间进行查找，**Y<T>** 此时已经完全准确地知道（包括可能在**X**的定义之后编写的任何可能的特化)。因此它调用的是**Y<int>::h( )**。
- **t1**和**t2**的声明是关联的。
- 对**operator<<(cout,t1)**的调用也是关联的，因为**t1**的类型为**T**。它是在**T**已经确定为**int**后才进行查找，并且在**std**中找到后添加了**int**。

- **swap( )**的非限定调用也是关联的，因为它的参数是类型**T**。这从根本上引起了全局**swap(int&, int&)**的实例化。
- **std::swap( )**的限定调用不是关联的，因为**std**是一个固定的名字空间。编译器知道去**std**中寻找合适的声明。（“::”左边的限定词必须为关联的限定名称提及一个模板参数）。之后**std::swap( )**函数模板在实例化期间生成**std::swap(int&, int&)**。**X<T>::f( )**中再也没有关联名称了。

综上所述：若名称是关联的，则它的查找是在实例化时进行，非限定的关联名称除外，它是一个普通名称查找，它的查找进行的比较早是在定义时进行。所有模板中的非关联名称被较早地查找，这种查找是在模板定义被解析的时候进行。（若有必要，这种名称还有另一种实例化期间的查找，此时实际参数类型是已知的。） 279

如果读者已经理解了我们讨论过的例子，请准备好学习下一节有关**friend**声明的内容，它也许会带给读者另一个惊奇。

5.4.2 模板和友元

在类中声明一个友元函数，就允许一个类的非成员函数访问这个类的非公有成员。若友元函数的名称是被限定的，则将会在限定它的名字空间或类中找到它。但是，如果它是非限定的，编译器必须假定在某处能找到这个友元函数的定义，因为所有的标识符必须有一个惟一的作用域。编程人员希望把这个函数定义在最近的封装名字空间（而非类）作用域内，这个作用域内也包括与那个函数有友元关系的类。通常这个作用域就是全局作用域。下面的非模板例子清晰地阐明了这一点：

```
//: C05:FriendScope.cpp
#include <iostream>
using namespace std;

class Friendly {
  int i;
public:
  Friendly(int theInt) { i = theInt; }
  friend void f(const Friendly&); // Needs global def.
  void g() { f(*this); }
};

void h() {
  f(Friendly(1));  // Uses ADL
}

void f(const Friendly& fo) {  // Definition of friend
  cout << fo.i << endl;
}
int main() {
  h(); // Prints 1
  Friendly(2).g(); // Prints 2
} ///:~
```

280

Friendly类中**f( )**的声明是非限定的，因此编译器希望能最终将这个声明链接到位于文件作用域（在本例中就是包含**Friendly**的名字空间作用域）中的它的定义上。它的定义出现在函数**h( )**的定义之后。然而，**h( )**中对链接到同一函数的**f( )**的调用却是一件单独的事情。这个过程由ADL进行解析。由于**h( )**中的**f( )**的参数是一个**Friendly**对象，为了匹配**f( )**的声明，

编译器将会去寻找**Friendly**类，并将成功找到。如果调用的是**f(1)**（这是很有意义的，因为1能被隐含地转变为**Friendly(1)**），这个调用将会失败，因为没有任何提示来通知编译器应该去哪儿寻找这个**f( )**的声明。在这种情况下，EDG编译器会正确地解释**f**没有定义。

现在假定**Friendly**和**f**都是模板，程序如下所示：

```
//: C05:FriendScope2.cpp
#include <iostream>
using namespace std;

// Necessary forward declarations:
template<class T> class Friendly;
template<class T> void f(const Friendly<T>&);

template<class T> class Friendly {
  T t;
public:
  Friendly(const T& theT) : t(theT) {}
  friend void f<>(const Friendly<T>&);
  void g() { f(*this); }
};

void h() {
  f(Friendly<int>(1));
}

template<class T> void f(const Friendly<T>& fo) {
  cout << fo.t << endl;
}

int main() {
  h();
  Friendly<int>(2).g();
} ///:~
```

281

首先要注意到**Friendly**中**f**的声明里的尖括号。这是必要的，它告诉编译器**f**是一个模板。否则，编译器就会去寻找一个名为**f**的普通函数而不会找到它。读者可能会在尖括号里加上模板参数（**<T>**），但它其实可以很容易地从声明中推断出来。

在类定义之前，提前声明函数模板**f**是很有必要的，虽然在前面的例子中并没有这个声明，因为当时**f**不是模板；这句话的意思清楚表明：友元函数模板必须提前声明。为了恰当地声明**f**，**Friendly**也必须在它之前进行声明，因为**f**具有一个**Friendly**参数，因此**Friendly**的声明在程序开始的最前面。也可以将**f**的完全定义放在**Friendly**的初始声明之后，这样就避免了将它的定义和声明分离开，但选择这样做是为了更接近上一个例子的格式。

为了在模板中使用友元，最后还可以选择这样做：在主类模板定义中完全地定义友元函数。下面来看看上例在这种情况下是如何修改的：

```
//: C05:FriendScope3.cpp {-bor}
// Microsoft: use the -Za (ANSI-compliant) option
#include <iostream>
using namespace std;

template<class T> class Friendly {
  T t;
public:
  Friendly(const T& theT) : t(theT) {}
  friend void f(const Friendly<T>& fo) {
    cout << fo.t << endl;
  }
```

```
  void g() { f(*this); }
};

void h() {
  f(Friendly<int>(1));
}

int main() {
  h();
  Friendly<int>(2).g();
} ///:~
```

282

本例和前面的例子有一个重要的区别：在这里**f**不再是一个模板，而是一个普通函数。（请记住，要表明**f( )**是一个模板，尖括号是必不可少的。）**Friendly**类模板每次实例化的时候，就会生成一个新的带有当前**Friendly**特化参数的普通重载函数。这就是为什么Dan Saks称之为“产生新友元”[⊖]的原因。这是为模板定义友元函数的最方便的方式。

为了说明得更清楚一些，假设现在要想向一个类模板中加入非成员友元运算符。下面只是个仅持有一个普通值的类模板：

```
template<class T> class Box {
  T t;
public:
  Box(const T& theT) : t(theT) {}
};
```

有些初学者没有理解本节前面的例子，他们可能会灰心丧气。因为程序中没有一个简单的流输出插入符来验证程序所做的工作。如果不在**Box**的定义中定义自己的运算符，就必须提供早些时候讨论过的前置声明：

```
//: C05:Box1.cpp
// Defines template operators.
#include <iostream>
using namespace std;

// Forward declarations
template<class T> class Box;

template<class T>
Box<T> operator+(const Box<T>&, const Box<T>&);
template<class T>
ostream& operator<<(ostream&, const Box<T>&);

template<class T> class Box {
  T t;
public:
  Box(const T& theT) : t(theT) {}
  friend Box operator+<>(const Box<T>&, const Box<T>&);
  friend ostream& operator<< <>(ostream&, const Box<T>&);
};

template<class T>
Box<T> operator+(const Box<T>& b1, const Box<T>& b2) {
  return Box<T>(b1.t + b2.t);
}

template<class T>
```

283

⊖ 来源于2001年9月份在波特兰的一次“C++研讨会”。

```
ostream& operator<<(ostream& os, const Box<T>& b) {
  return os << '[' << b.t << ']';
}

int main() {
  Box<int> b1(1), b2(2);
  cout << b1 + b2 << endl;  // [3]
//  cout << b1 + 2 << endl; // No implicit conversions!
} ///:~
```

程序在这里定义了一个加运算符和一个输出流操作符。主程序揭示了这个方法的一个缺陷：无法使用隐式转换（如表达式**b1+2**），因为模板没有提供这些转换。使用内部类，非模板方法将会使程序变得短小、更强健：

```
//: C05:Box2.cpp
// Defines non-template operators.
#include <iostream>
using namespace std;

template<class T> class Box {
  T t;
public:
  Box(const T& theT) : t(theT) {}
  friend Box<T> operator+(const Box<T>& b1,
                          const Box<T>& b2) {
    return Box<T>(b1.t + b2.t);
  }
  friend ostream&
  operator<<(ostream& os, const Box<T>& b) {
    return os << '[' << b.t << ']';
  }
};

int main() {
  Box<int> b1(1), b2(2);
  cout << b1 + b2 << endl; // [3]
  cout << b1 + 2 << endl; // [3]
} ///:~
```

284

由于运算符成员函数是普通函数（为**Box**的每一个特化进行的重载——本例中就是**int**），像平常一样，隐式转换也可以使用；因此表达式**b1+2**是合法的。

注意，有一个特殊的类型不能成为**Box**或其他任意类模板的友元，这个类型是**T**——或由**T**参数化的类模板类型。无论如何，找不到一个很合理的原因解释为什么不能这样用，但的确如此，**friend class T** 这个声明是不合法的，也不能被编译。

友元模板

在程序中可以更精确地说明一个模板的哪些特化是类的友元。在上节的例子中，只有函数模板**f**是一个友元，它与特化**Friendly**的类型相同。举例来说，只有特化**f<int>(const Friendly<int>&)**才是类**Friendly<int>**的一个友元。这个例子是通过**Friendly**的模板参数来特化在其友元声明中的**f**的方式来实现的。若愿意，可以产生一个特别的、固定的**f**特化作为所有**Friendly**实例的一个友元，如下所示：

```
// Inside Friendly:
friend void f<>(const Friendly<double>&);
```

通过用**double**代替**T**，**f**的**double**特化可以访问任意**Friendly**特化的非公有成员。而**f<double>()**特化直到被明确调用时才会被实例化。

同样，若声明了一个参数不依赖于T的非模板函数，这个函数就是所有**Friendly**实例的一个友元：

285

```
// Inside Friendly:
friend void g(int);  // g(int) befriends all Friendlys
```

同前面一样，由于**g(int)**是非限定的，他必须在文件作用域（包含**Friendly**的名字空间作用域）内定义。

也可以让f的所有特化成为所有**Friendly**特化的友元。只需要通过一个所谓的*友元模板*（friend template）来实现，如下所示：

```
template<class T> class Friendly {
  template<class U> friend void f<>(const Friendly<U>&);
```

由于友元声明的模板参数独立于T，因此任意的T和U的组合都允许使用，形成友元关系。像成员模板一样，友元模板也可以出现在非模板类中。

5.5 模板编程中的习语

语言是表达思想的一种工具，新的编程语言特征总是会产生新的程序设计技术。本节讨论一些经常使用的模板编程用语，自模板被引入C++语言，这些用语就已经出现并应用了好多年了。[⊖]

5.5.1 特征

特征模板技术，最先由Nathan Myers倡导，它是一种将与某种类型相关联的所有声明绑定在一起的实现方式。本质上说，使用特征技术，可以以一种灵活的方法从它们的语境中将这些类型和值进行“混合与匹配”，同时又使得程序的代码灵活易读并且易于维护。

一个最简单的特征模板的例子是定义在**<limits>**中的**numeric_limits**类模板。这个基本模板的定义如下：

286

```
template<class T> class numeric_limits {
public:
  static const bool is_specialized = false;
  static T min() throw();
  static T max() throw();
  static const int digits = 0;
  static const int digits10 = 0;
  static const bool is_signed = false;
  static const bool is_integer = false;
  static const bool is_exact = false;
  static const int radix = 0;
  static T epsilon() throw();
  static T round_error() throw();
  static const int min_exponent = 0;
  static const int min_exponent10 = 0;
  static const int max_exponent = 0;
  static const int max_exponent10 = 0;
  static const bool has_infinity = false;
  static const bool has_quiet_NaN = false;
  static const bool has_signaling_NaN = false;
  static const float_denorm_style has_denorm =
                                  denorm_absent;
  static const bool has_denorm_loss = false;
```

⊖ 另一个模板用语，混入继承，将在第9章讨论。

```
    static T infinity() throw();
    static T quiet_NaN() throw();
    static T signaling_NaN() throw();
    static T denorm_min() throw();
    static const bool is_iec559 = false;
    static const bool is_bounded = false;
    static const bool is_modulo = false;
    static const bool traps = false;
    static const bool tinyness_before = false;
    static const float_round_style round_style =
                                  round_toward_zero;
};
```

<limits>头文件为所有基本数字类型定义了特化（当**is_specialized**成员被设为**true**时）。例如，若想得到浮点数字系统的**double**版本的基类型，可以使用表达式**numeric_limits<double>::radix**。为了得到有用的最小整数值，可以使用**numeric_ limits<int>::min( )**。在程序中，并非向所有的**numeric_limits**成员都提供了基本类型。（例如，**epsilon( )**只对浮点数类型有意义。）

287 有些值总是整数，它们是**numeric_limits**的静态数据成员。有些可能不总是整数值，例如**float**的最小值，它们作为静态内联成员函数实现。这是因为C++只允许在类定义中初始化整数（integral）静态数据成员常量。

在第3章中读者看到了字符串类如何使用特征技术控制字符处理函数。**std::string**类和**std::wstring**类是**std::basic_string**模板的特化，它的定义如下所示：

```
template<class charT,
  class traits = char_traits<charT>,
  class allocator = allocator<charT> >
  class basic_string;
```

模板参数**charT**代表了基础字符类型，它通常是**char**类型或**wchar_t**类型。基本的**char_traits**模板是典型的空模板，标准库提供了对**char**和**wchar_t**进行的特化。下面是根据C++标准提供的一个**char_traits<char>**特化：

```
template<> struct char_traits<char> {
  typedef char char_type;
  typedef int int_type;
  typedef streamoff off_type;
  typedef streampos pos_type;
  typedef mbstate_t state_type;
  static void assign(char_type& c1, const char_type& c2);
  static bool eq(const char_type& c1, const char_type& c2);
  static bool lt(const char_type& c1, const char_type& c2);
  static int compare(const char_type* s1,
                     const char_type* s2, size_t n);
  static size_t length(const char_type* s);
  static const char_type* find(const char_type* s,
                               size_t n,
                               const char_type& a);
  static char_type* move(char_type* s1,
                         const char_type* s2, size_t n);
  static char_type* copy(char_type* s1,
                         const char_type* s2, size_t n);
  static char_type* assign(char_type* s, size_t n,
                           char_type a);
```

288

```
  static int_type not_eof(const int_type& c);
  static char_type to_char_type(const int_type& c);
  static int_type to_int_type(const char_type& c);
  static bool eq_int_type(const int_type& c1,
```

```
                                const int_type& c2);
  static int_type eof();
};
```

basic_string类模板使用这些函数，用于基于字符操作的通用的字符串处理。当声明一个**string**变量时，例如:

```
std::string s;
```

事实上，正在声明的**s**格式如下所示（由于在**basic_string**特化中有默认的模板参数）:

```
std::basic_string<char, std::char_traits<char>,
                  std::allocator<char> > s;
```

由于字符特征已经从**basic_string**类模板中分离出来，可以使用一个惯用的特征类来取代**std::char_traits**。下面的例子显示了这种灵活性:

```
//: C05:BearCorner.h
#ifndef BEARCORNER_H
#define BEARCORNER_H
#include <iostream>
using std::ostream;

// Item classes (traits of guests):
class Milk {
public:
  friend ostream& operator<<(ostream& os, const Milk&) {
    return os << "Milk";
  }
};

class CondensedMilk {
public:
  friend ostream&
  operator<<(ostream& os, const CondensedMilk &) {
    return os << "Condensed Milk";
  }
};
class Honey {
public:
  friend ostream& operator<<(ostream& os, const Honey&) {
    return os << "Honey";
  }
};

class Cookies {
public:
  friend ostream& operator<<(ostream& os, const Cookies&) {
    return os << "Cookies";
  }
};

// Guest classes:
class Bear {
public:
  friend ostream& operator<<(ostream& os, const Bear&) {
    return os << "Theodore";
  }
};

class Boy {
public:
  friend ostream& operator<<(ostream& os, const Boy&) {
```

289

```
    return os << "Patrick";
  }
};

// Primary traits template (empty-could hold common types)
template<class Guest> class GuestTraits;

// Traits specializations for Guest types
template<> class GuestTraits<Bear> {
public:
  typedef CondensedMilk beverage_type;
  typedef Honey snack_type;
};

template<> class GuestTraits<Boy> {
public:
  typedef Milk beverage_type;
  typedef Cookies snack_type;
};
```

290

```
#endif // BEARCORNER_H ///:~
```

```
//: C05:BearCorner.cpp
// Illustrates traits classes.
#include <iostream>
#include "BearCorner.h"
using namespace std;

// A custom traits class
class MixedUpTraits {
public:
  typedef Milk beverage_type;
  typedef Honey snack_type;
};

// The Guest template (uses a traits class)
template<class Guest, class traits = GuestTraits<Guest> >
class BearCorner {
  Guest theGuest;
  typedef typename traits::beverage_type beverage_type;
  typedef typename traits::snack_type snack_type;
  beverage_type bev;
  snack_type snack;
public:
  BearCorner(const Guest& g) : theGuest(g) {}
  void entertain() {
    cout << "Entertaining " << theGuest
         << " serving " << bev
         << " and " << snack << endl;
  }
};

int main() {
  Boy cr;
  BearCorner<Boy> pc1(cr);
  pc1.entertain();
  Bear pb;
  BearCorner<Bear> pc2(pb);
  pc2.entertain();
  BearCorner<Bear, MixedUpTraits> pc3(pb);
  pc3.entertain();
} ///:~
```

在这个程序中，为招待作为客人的类**Boy**和类**Bear**的实例，提供了适合他们口味的食物。

Boy喜欢牛奶和小甜点，**Bear**喜欢浓缩的牛奶和蜂蜜。客人与食物之间的关联是通过一个基本的（空的）特征类模板的特化完成的。**BearCorner**的默认参数保证了客人能够获得恰当的食物，但也可以用一个简单的符合特征类需求的类来代替它，就像上面用到的**MixedUpTraits**类。这个程序的输出是: 291

```
Entertaining Patrick serving Milk and Cookies
Entertaining Theodore serving Condensed Milk and Honey
Entertaining Theodore serving Milk and Honey
```

特征类的使用提供了两个关键的优点: (1) 在将对象与其关联的属性或函数配对方面提供了灵活性和可扩充性，(2) 它保持了模板参数列表的短小易读。如果一个客人与30个类型相关，那么，将所有30个参数直接在每一个**BearCorner**声明中指定，这将是非常不方便的。而将这些类型放在一个独立的特征类中就会大大简化这项工作。

如第4章所述，特征技术也可用于实现流和区域化。在第6章有一个名为**PrintSequence.h**头文件，其中可以找到一个迭代器特征类的例子。

5.5.2 策略

如果检查一下用**wchar_t** 特化的**char_traits**，就会发现实际上它相当于**char**特化的副本:

```
template<> struct char_traits<wchar_t> {
  typedef wchar_t char_type;
  typedef wint_t int_type;
  typedef streamoff off_type;
  typedef wstreampos pos_type;
  typedef mbstate_t state_type;
  static void assign(char_type& c1, const char_type& c2);
  static bool eq(const char_type& c1, const char_type& c2);
  static bool lt(const char_type& c1, const char_type& c2);
  static int compare(const char_type* s1,
                     const char_type*  s2, size_t n);
  static size_t length(const char_type* s);
  static const char_type* find(const char_type* s,
                               size_t n,
                               const char_type& a);
  static char_type* move(char_type* s1,
                         const char_type* s2, size_t n);
  static char_type* copy(char_type* s1,
                         const char_type* s2, size_t n);
  static char_type* assign(char_type* s, size_t n,
                           char_type a);
  static int_type not_eof(const int_type& c);
  static char_type to_char_type(const int_type& c);
  static int_type to_int_type(const char_type& c);
  static bool eq_int_type(const int_type& c1,
                          const int_type& c2);
  static int_type eof();
};
```

292

两个版本惟一的真正的区别是，所包含的类型集不同（**char**和**int**分别相对于**wchar_t**和**wint_t**）。两者所提供的函数是相同的。[⊖]这更突出了一个事实: 特征类是为特征（trait）而设计的，在相关的特征类之间的改变通常就是类型和常量值，或者是使用了相关类型的模板参数的固定算法。通常特征类本身就是模板，因为它们包含的类型和常量通常被看作是基本模

⊖ 实际上（这不是实质上的，例如，）**char_traits<>::compare( )**在一个实例中可能调用函数**strcmp( )**，而在另一个实例中就有可能调用**wcscmp( )**。而在这里所说的是**compare( )**函数执行的功能是相同的。

板的特征参数（例如，**char**和**wchar_t**）。

将函数（functionality）与模板参数关联起来也是有用的，因而客户端程序员在他们编码的时候能够轻松地定制代码行为。举例来说，下面的这个**BearCorner**程序版本，支持不同的招待类型：

```
//: C05:BearCorner2.cpp
// Illustrates policy classes.
#include <iostream>
#include "BearCorner.h"
using namespace std;

// Policy classes (require a static doAction() function):
class Feed {
public:
  static const char* doAction() { return "Feeding"; }
};
class Stuff {
public:
  static const char* doAction() { return "Stuffing"; }
};

// The Guest template (uses a policy and a traits class)
template<class Guest, class Action,
         class traits = GuestTraits<Guest> >
class BearCorner {
  Guest theGuest;
  typedef typename traits::beverage_type beverage_type;
  typedef typename traits::snack_type snack_type;
  beverage_type bev;
  snack_type snack;
public:
  BearCorner(const Guest& g) : theGuest(g) {}
  void entertain() {
    cout << Action::doAction() << " " << theGuest
         << " with " << bev
         << " and " << snack << endl;
  }
};

int main() {
  Boy cr;
  BearCorner<Boy, Feed> pc1(cr);
  pc1.entertain();
  Bear pb;
  BearCorner<Bear, Stuff> pc2(pb);
  pc2.entertain();
} ///:~
```

293

BearCorner类中的**Action**模板参数希望有一个名为**doAction( )**的静态成员函数，它用在**BearCorner< >::entertain( )**中。用户按照意愿可以选择**Feed**或**Stuff**，二者都提供了所需的函数。用这种方式来封装函数的类称为策略类（policy class）。在上例中，招待“策略”是通过**Feed::doAction( )**和**Stuff::doAction( )**提供的。这些策略类可能是普通类，也可能是模板，还有可能是结合了使用继承机制全部优点的类。关于基于策略的更深入的设计技术，请参看Andrei Alexandrescu的书[⊖]。关于这个主题，这本书具有权威性。

294

⊖ 《Modern C++ Design: Generic Programming and Design Patterns Applied》，Addison Wesley，2001。

5.5.3 奇特的递归模板模式

任何一个初学C++的程序设计者都知道如何修改一个类，使它跟踪一个类当前实际存在的对象个数。必须做的所有工作就是添加静态成员、修改构造函数和析构函数的逻辑，如下所示：

```
//: C05:CountedClass.cpp
// Object counting via static members.
#include <iostream>
using namespace std;

class CountedClass {
  static int count;
public:
  CountedClass() { ++count; }
  CountedClass(const CountedClass&) { ++count; }
  ~CountedClass() { --count; }
  static int getCount() { return count; }
};

int CountedClass::count = 0;

int main() {
  CountedClass a;
  cout << CountedClass::getCount() << endl;   // 1
  CountedClass b;
  cout << CountedClass::getCount() << endl;   // 2
  { // An arbitrary scope:
    CountedClass c(b);
    cout << CountedClass::getCount() << endl; // 3
    a = c;
    cout << CountedClass::getCount() << endl; // 3
  }
  cout << CountedClass::getCount() << endl;   // 2
} ///:~
```

CountedClass的所有构造函数都对静态数据成员**count**进行增1计数，而析构函数进行减1计数。静态成员函数**getCount( )**获取当前对象的个数。

295

每次想为新添加的一个类的对象进行计数的时候，手工添加这些成员实在是太枯燥了。在面向对象程序设计中，过去常常对代码进行重用或共享采用的是继承方式，在本例中这也只是半个解决方案。请观察，当在基类中使用计数逻辑时会有什么情况发生：

```
//: C05:CountedClass2.cpp
// Erroneous attempt to count objects.
#include <iostream>
using namespace std;

class Counted {
  static int count;
public:
  Counted() { ++count; }
  Counted(const Counted&) { ++count; }
  ~Counted() { --count; }
  static int getCount() { return count; }
};

int Counted::count = 0;

class CountedClass : public Counted {};
class CountedClass2 : public Counted {};

int main() {
```

```
  CountedClass a;
  cout << CountedClass::getCount() << endl;    // 1
  CountedClass b;
  cout << CountedClass::getCount() << endl;    // 2
  CountedClass2 c;
  cout << CountedClass2::getCount() << endl;   // 3 (Error)
} ///:~
```

派生自**Counted**的所有类都共享了相同的、惟一的静态数据成员，因此通过跨越**Counted**层次结构中所有的类，它们的对象个数全部被跟踪。现在所需的是，有一种能自动为每个派生类生成一个不同基类的方式。一种奇特的模板构造实现了这种方式，如下所示：

296

```
//: C05:CountedClass3.cpp
#include <iostream>
using namespace std;

template<class T> class Counted {
  static int count;
public:
  Counted() { ++count; }
  Counted(const Counted<T>&) { ++count; }
  ~Counted() { --count; }
  static int getCount() { return count; }
};

template<class T> int Counted<T>::count = 0;

// Curious class definitions
class CountedClass : public Counted<CountedClass> {};
class CountedClass2 : public Counted<CountedClass2> {};

int main() {
  CountedClass a;
  cout << CountedClass::getCount() << endl;    // 1
  CountedClass b;
  cout << CountedClass::getCount() << endl;    // 2
  CountedClass2 c;
  cout << CountedClass2::getCount() << endl;   // 1 (!)
} ///:~
```

每个派生类都派生于一个惟一的基类，这个基类将它本身（派生类）作为模板参数！它看起来像是陷入了一个递归（循环）的定义，而且还有可能在某次计算中将某个任意的基类成员作为模板参数。由于**Counted**的数据成员不依赖于**T**，当模板被解析的时候，**Counted**的大小（为零！）就可以知道。因此究竟使用什么样的参数来实例化**Counted**无关紧要，因为它的大小总是相同的。当它被解析时，用任意一个**Counted**实例的派生类当然也可以完成，而且不会产生递归。由于每个基类都是惟一的，它有属于自己的静态数据，因而无论如何，这都是一个实现了向任意类中添加计数的便捷方法。Jim Coplien是第1个在刊物上提出这种有趣的派生方法的人；他在一篇名为“奇特的递归模板模式（curiously recurring template pattern）”[⊖]的文章中提出了这个方法。

297

5.6 模板元编程

1993年，编译器开始支持简单的模板构造，因此用户可以定义通用的容器和函数。同一时期，C++标准委员会也正在考虑将STL纳入标准C++，当时在C++标准委员会的成员们周围，

⊖ 《C++Gems》，由Stan Lippman编辑，SIGS，1996。

到处都是那些已经通过验证的精巧的和令人惊讶的程序例子[⊖]，其中一个简单的例子如下所示:

```
//: C05:Factorial.cpp
// Compile-time computation using templates.
#include <iostream>
using namespace std;

template<int n> struct Factorial {
  enum { val = Factorial<n-1>::val * n };
};

template<> struct Factorial<0> {
  enum { val = 1 };
};

int main() {
  cout << Factorial<12>::val << endl; // 479001600
} ///:~
```

程序输出了由参数**12!**实例化后的正确值！并没有警告发出。那么警告是什么呢？警告就是：在程序开始运行前就已经完成了计算！

当编译器试图对**Factorial<12>**进行实例化时，编译器发现必须先实例化**Factorial<11>**，而后者又要求实例化**Factorial<10>**，以此类推。这个递归最终在特化**Factorial<1>**时结束，此时计算展开，**Factorial<12>::val**由整数常量479 001 600代替，至此编译结束。由于所有的计算都由编译器来做，其包含的值必须是编译时常量，因此使用了**enum**。程序运行时，惟一要做的工作就是跟随一个换行符来打印这个常量的值。为了说服自己，使读者相信是**Factorial**的一个特化导致了产生正确的编译时结果值，可以用它作为一个数组的维数来验证一下，如下所示:

298

```
double nums[Factorial<5>::val];
assert(sizeof nums == sizeof(double)*120);
```

5.6.1 编译时编程

正如将进行类型参数代替作为一种方便的方法，这意味着产生了一种支持编译时编程的机制。这样的程序称为**模板元程序**（template metaprogram）（因为正在“为一个程序进行编程”），事实证明可以用它做很多的事情。实际上，模板元编程就是完全的图灵机（Turing complete），因为它支持选择（if-else）和循环（通过递归）。从理论上讲，可以用它执行任何的计算[⊖]。上面的**factorial**程序例子显示了如何实现循环：编写一个递归模板，并且通过一个特化来提供一个终止递归的规则。下面的例子显示了如何利用相同的技术在编译时计算斐波那契（Fibonacci）数:

```
//: C05:Fibonacci.cpp
#include <iostream>
using namespace std;

template<int n> struct Fib {
  enum { val = Fib<n-1>::val + Fib<n-2>::val };
};
```

⊖ 技术上讲，这些都是编译时常值，因此可能会说其中的标识符按照习惯用法应该全是大写字母，之所以坚持用小写是因为在这里它们模拟了变量。

⊖ 1966年，Böhm和Jacopini证明了任意具有以下特征的语言都相当于一个图灵机：支持选择和循环，并具有使用变量随机数的能力。图灵机被认为具有表达任意算法的能力。

```
template<> struct Fib<1> { enum { val = 1 }; };

template<> struct Fib<0> { enum { val = 0 }; };

int main() {
  cout << Fib<5>::val << endl;   // 6
  cout << Fib<20>::val << endl;  // 6765
} ///:~
```

299

斐波那契数的数学定义如下:

$$f_n = \begin{cases} 0, n = 0 \\ 1, n = 1 \\ f_{n-2} + f_{n-1}, n > 1 \end{cases}$$

前两种情况导致了上面的模板特化，第3行的规则就是基本的模板。

1. 编译时循环

在一个模板元程序中要计算任意的循环，首先必须再用公式表示递归。例如，若想计算整数**n**的**p**次方，下面几行程序使用了一个循环就可以完成:

```
int val = 1;
while(p--)
  val *= n;
```

也可以将它写成一个递归过程:

```
int power(int n, int p) {
  return (p == 0) ? 1 : n*power(n, p - 1);
}
```

现在它就可以很容易地用一个模板元程序来实现:

```
//: C05:Power.cpp
#include <iostream>
using namespace std;

template<int N, int P> struct Power {
  enum { val = N * Power<N, P-1>::val };
};

template<int N> struct Power<N, 0> {
  enum { val = 1 };
};

int main() {
  cout << Power<2, 5>::val << endl;  // 32
} ///:~
```

300

由于**N**仍是一个自由模板参数，因此需要用一个半特化来作为终止条件。注意，这个程序仅当指数为非负时才运行。

由Czarnecki和Eisenecker[⊖]改编的下述元程序是很有趣的，因为它使用一个模板作为模板参数，模拟传递一个函数作为另一个函数的参数。其中的“循环是通过”**0..n**这些数字来实现的:

⊖ Czarnecki 和Eisenecker，《Generative Programming: Methods, Tools, and Applications》，Addison Wesley，2000，第417页。

```
//: C05:Accumulate.cpp
// Passes a "function" as a parameter at compile time.
#include <iostream>
using namespace std;

// Accumulates the results of F(0)..F(n)
template<int n, template<int> class F> struct Accumulate {
  enum { val = Accumulate<n-1, F>::val + F<n>::val };
};

// The stopping criterion (returns the value F(0))
template<template<int> class F> struct Accumulate<0, F> {
  enum { val = F<0>::val };
};

// Various "functions":
template<int n> struct Identity {
  enum { val = n };
};

template<int n> struct Square {
  enum { val = n*n };
};

template<int n> struct Cube {
  enum { val = n*n*n };
};
int main() {
  cout << Accumulate<4, Identity>::val << endl; // 10
  cout << Accumulate<4, Square>::val << endl;   // 30
  cout << Accumulate<4, Cube>::val << endl;     // 100
} ///:~
```

301

基本的**Accumulate**模板试图计算**F(n)+F(n-1)...F(0)**的和。终止递归是通过一个“返回”**F(0)**的半特化来实现的。参数**F**本身就是一个模板，在本节以前的例子中它通常是一个函数。模板**Identity**、**Square**和**Cube**，用它们的模板参数计算自己名字表明的相应函数。在**main( )**函数中，**Accumulate**的第1个实例化计算是求和：4+3+2+1+0，因此**Identity**函数仅仅“返回”它的模板参数。**main( )**中第2行是计算这些数字的平方和：（16+9+4+1+0）。最后计算立方和：（64+27+8+1+0）。

2. 循环分解

算法设计者们总是尽力优化他们的程序。其中一个是在时间方面的优化——特别是在数字计算编程中——采用的是循环分解（loop unrolling），这是一项将顶层循环的次数减到最小的技术。典型的循环分解的例子是矩阵相乘。下面的函数将一个矩阵与一个向量相乘（假设常量**ROWS**和**COLS**已经定义过了）：

```
void mult(int a[ROWS][COLS], int x[COLS], int y[COLS]) {
  for(int i = 0; i < ROWS; ++i) {
      y[i] = 0;
      for(int j = 0; j < COLS; ++j)
        y[i] += a[i][j]*x[j];
  }
}
```

如果**COLS**是偶数，则进行增1和比较循环控制变量**j**的那一层循环体，就能用将该计算“分解”的方法切成两部分，使其变成在内部循环中成对出现的两部分计算：

```
void mult(int a[ROWS][COLS], int x[COLS], int y[COLS]) {
  for(int i = 0; i < ROWS; ++i) {
```

```
        y[i] = 0;
        for(int j = 0; j < COLS; j += 2)
          y[i] += a[i][j]*x[j] + a[i][j+1]*x[j+1];
    }
  }
```

302

通常，若**COLS**是**k**的一个因子，每次内部循环迭代都可以执行**k**操作，大量减少顶层的循环。这种节省只有在大数组上操作才是明显的，但这正好是一个产业的强度数学计算的严谨的例子。

内联函数也构成了循环分解的一种格式。请看下面计算整数幂的方法:

```
//: C05:Unroll.cpp
// Unrolls an implicit loop via inlining.
#include <iostream>
using namespace std;

template<int n> inline int power(int m) {
  return power<n-1>(m) * m;
}

template<> inline int power<1>(int m) {
  return m;
}

template<> inline int power<0>(int m) {
  return 1;
}

int main() {
  int m = 4;
  cout << power<3>(m) << endl;
} ///:~
```

从概念上来讲，编译器必须生成模板参数分别为3、2、1的3个**power< >**特化。因为每个函数的代码可以内联，实际插入**main( )**函数的代码就是一个单一的表达式**m*m*m**。这样一来，一个存在内联的简单模板特化就提供了一种方法，该方法可以完全避免循环控制顶层的出现[⊖]。这种循环分解的方法受使用的编译器的内联深度的限制。

303

3. 编译时选择

为了模拟在编译时的条件，可以在一个**enum**声明中使用3目条件运算符。下面的程序就使用了这个技术，在编译时计算两个整数之中的最大值:

```
//: C05:Max.cpp
#include <iostream>
using namespace std;

template<int n1, int n2> struct Max {
  enum { val = n1 > n2 ? n1 : n2 };
};

int main() {
  cout << Max<10, 20>::val << endl;  // 20
} ///:~
```

如果想使用编译时条件来控制自定义代码的生成，可以利用**true**和**false**值的特化:

⊖ 有一种更好的计算整数幂的方法: Russian Peasant算法。

```
//: C05:Conditionals.cpp
// Uses compile-time conditions to choose code.
#include <iostream>
using namespace std;

template<bool cond> struct Select {};

template<> class Select<true> {
  static void statement1() {
    cout << "This is statement1 executing\n";
  }
public:
  static void f() { statement1(); }
};

template<> class Select<false> {
  static void statement2() {
    cout << "This is statement2 executing\n";
  }
public:
  static void f() { statement2(); }
};

template<bool cond> void execute() {
  Select<cond>::f();
}

int main() {
  execute<sizeof(int) == 4>();
} ///:~
```

304

这个程序相当于下面的表达式:

```
if(cond)
  statement1();
else
  statement2();
```

除了条件**cond**在编译时确定之外，**execute< >( )**和**Select< >**的适当版本都由编译器进行实例化。函数**Select< >::f( )**在运行时执行。可以用类似的方式仿效一个**switch**语句，但不是用值**true**和**false**，而是特化每个case值。

4. 编译时断言

在第2章中讨论了将断言（assertion）作为整个防御性编程策略的一部分的优点。断言基本上就是一个其后有一个适当动作的布尔表达式的判断：若条件为真则什么都不做，否则就停止并附带一个诊断消息。最好能尽快发现断言失败。若可以在编译时对一个表达式求值，就使用编译时断言。下面的例子使用了这个技术，它将一个布尔表达式映射到一个数组声明中:

```
//: C05:StaticAssert1.cpp {-xo}
// A simple, compile-time assertion facility

#define STATIC_ASSERT(x) \
  do { typedef int a[(x) ? 1 : -1]; } while(0)

int main() {
  STATIC_ASSERT(sizeof(int) <= sizeof(long)); // Passes
  STATIC_ASSERT(sizeof(double) <= sizeof(int)); // Fails
} ///:~
```

do循环为一个数组**a**的定义产生了一个临时空间，这个数组的大小由一个不确定的条件所决定。定义一个大小为－1的数组是不合法的，因此当条件为假时这条语句将失败。

305 前面的内容说明了如何对编译时布尔表达式求值。在效仿编译时断言方面剩下的问题就是打印一个有意义的错误消息并且停止编译。所有的编译错误都要求编译器停止编译；解决这个问题的一个技巧是在错误消息中插入有用的文本。从Alexandrescu[⊖]处获得的下面的例子使用了模板特化，一个局部类和一个小巧且奇妙的宏来完成这项工作：

```
//: C05:StaticAssert2.cpp {-g++}
#include <iostream>
using namespace std;

// A template and a specialization
template<bool> struct StaticCheck {
  StaticCheck(...);
};

template<> struct StaticCheck<false> {};

// The macro (generates a local class)
#define STATIC_CHECK(expr, msg) {              \
  class Error_##msg {};                        \
  sizeof((StaticCheck<expr>(Error_##msg()))); \
}

// Detects narrowing conversions
template<class To, class From> To safe_cast(From from) {
  STATIC_CHECK(sizeof(From) <= sizeof(To),
               NarrowingConversion);
  return reinterpret_cast<To>(from);
}

int main() {
  void* p = 0;
  int i = safe_cast<int>(p);
  cout << "int cast okay" << endl;
  //! char c = safe_cast<char>(p);
} ///:~
```

这个例子定义了一个函数模板**safe_cast< >( )**用来进行两个对象长度的检查。它检查源对象类型长度是否不大于目标对象类型的长度。如果目标对象类型的长度较小，则用户将会在
306 编译时得到通知：现正试图进行一个窄类型转换。注意，**StaticCheck**类模板有一个奇特的特性：任何模板参数的特化都可以被转换成**StaticCheck<true>**的实例（由于它的构造函数中的省略号[⊖]），并且没有任何模板参数的特化可以被转换成**StaticCheck<false>**的实例，因为没有为这种特化提供转换。它的思想是：在编译时如果相关条件在测试时为真，就创建一个新类的实例并且将它转换成为**StaticCheck<true>**对象；或者当条件在测试时为假，将它转换成一个**StaticCheck<false>**对象。由于**sizeof**运算符在编译时完成它的工作，因而用它来执行转换任务。当条件为假时，编译器将做出解释：它不知道如何将这个新类类型转换成**StaticCheck<false>**对象。（在**STATIC_CHECK( )**中的**sizeof**调用里面的特殊圆括号，是为了防止编译器认为程序正在试图将**sizeof**作为函数调用，这是不合法的）。为了在错误消息中插入一些有意义的信息，新的类名在它的名字中携带了关键且有意义的文字信息。

理解这项技术的最好的方式就是使其融入程序，请看上面例子**main( )**中的这一行：

⊖ 《Modern C++ Design》，第23~26页。

⊖ 不允许将对象类型（除了内建的）传递给一个省略号参数特化，但是由于只是计算它的大小（一个编译时操作），实际上表达式在运行时没有被判断。

```
int i = safe_cast<int>(p);
```

safe_cast<int>(p)的调用包含了下面的宏扩充代码，它代替了第1行代码：

```
{                                                  \
   class Error_NarrowingConversion {};             \
   sizeof(StaticCheck<sizeof(void*) <= sizeof(int)> \
          (Error_NarrowingConversion()));          \
}
```

（回忆一下标记传递预处理操作符：##，它将它的操作数连接为一个单一标记，因此经过预处理后**Error_##NarrowingConversion**变成了标记**Error_NarrowingConversion**。）**Error_NarrowingConversion**类是一个*局部类*（意味着它在一个非名字空间作用域内声明），因为它无需在这个程序的其他地方使用。这里**sizeof**运算符的使用试图决定**StaticCheck<true>**的实例的大小（因为在本程序使用的系统平台上**sizeof(void*)< = sizeof(int)**为真），由**Error_NarrowingConversion( )**调用返回的临时对象隐式产生。编译器知道新类**Error_NarrowingConversion**的大小（它为空），因此在编译时**sizeof**在**STATIC_CHECK( )**中外层的使用是合法的。由于从**Error_NarrowingConversion**临时对象转换到**StaticCheck<true>**实例取得成功，因而这个**sizeof**的外层应用和执行都可以继续。 307

现在来看看，如果**main( )**函数中最后一行的注解被去掉将会发生什么事情：

```
char c = safe_cast<char>(p);
```

在这里，把**safe_cast<char>(p)**中的**STATIC_CHECK( )**宏扩充为：

```
{                                                  \
   class Error_NarrowingConversion {};             \
   sizeof(StaticCheck<sizeof(void*) <= sizeof(char)> \
          (Error_NarrowingConversion()));          \
}
```

由于表达式**sizeof(void*)<=sizeof(char)**为假，此时程序将尝试进行从**Error_NarrowingConversion**临时对象到**StaticCheck<false>**实例的转换，如下所示：

```
sizeof(StaticCheck<false>(Error_NarrowingConversion()));
```

它失败了，所以编译器将发出一个如下的消息并停止工作：

```
Cannot cast from 'Error_NarrowingConversion' to
'StaticCheck<0>' in function
char safe_cast<char,void *>(void *)
```

类名**Error_NarrowingConversion**是由编码人员巧妙设计的有意义的消息。通常为了用这种技术来执行一个静态断言，应该调用**STATIC_CHECK**宏去进行编译时条件检查，并且用一个有意义的名称（函数名称、参数名称、模板名称等）来描述这个错误。

5.6.2 表达式模板 308

模板最强大的应用大概是在1994年由Todd Veldhuizen[⊖]和Daveed Vandevoorde[⊖]分别提出的一类模板技术：*表达式模板*（expression template）。表达式模板能够使某些计算得到的全方位

⊖ 在Lippman的《C++ Gems》，SIGS，1996中可以找到Todd的原文的再版。它也表明了除了保留数学符号和优化的代码，表达式模板也允许在C++库中结合使用其他编程语言中的范例和机制，如Lambda 表达式。另一个例子是奇特的类库Spirit，它是一个大量使用表达式模板的语法剖析器，它允许在C++中直接使用（一个近似的）EBNF符号，且产生了非常有效的语法剖析器。参看http://spirit.sourceforge.net/。

⊖ 参看他和Nico的《C++ Templates》，这是一本早期的权威著作。

的编译时优化，这些优化产生这样一些代码，其执行起来至少像支持优化的Fortran（专门用于科学计算的编程语言）代码一样快速，并且通过运算符重载仍旧保持了数学的原始符号。尽管可能在日常编程中并不使用这种技术，但它是由C++编写的许多复杂的高性能数学库的基础[⊖]。

为了引发读者学习表达式模板的兴趣，请看一个典型的数值线性代数的运算，将两个矩阵或向量相加[⊖]，如下所示：

```
D = A + B + C;
```

按照一般初学者的实现方式，这个表达式将会导致一些临时变量的产生——一个是**A+B**，一个是**(A+B)+C**。当这些变量代表极大的矩阵或向量时，这种方法将会耗尽系统的资源的确让人无法接受。表达式模板允许在没有临时变量的情况下使用同一个表达式。

下面的程序定义了一个**MyVector**类来模拟任意大小的数学向量。在程序中使用一个无类型的模板参数来表示向量的长度。程序还定义了一个**MyVectorSum**类来担当一个中间代理类，用其计算**MyVector**对象之和。这将允许使用惰性计算，因而向量的各个组成部分的相加不需要临时变量就可以执行。

309

```
//: C05:MyVector.cpp
// Optimizes away temporaries via templates.
#include <cstddef>
#include <cstdlib>
#include <ctime>
#include <iostream>
using namespace std;

// A proxy class for sums of vectors
template<class, size_t> class MyVectorSum;

template<class T, size_t N> class MyVector {
  T data[N];
public:
  MyVector<T,N>& operator=(const MyVector<T,N>& right) {
    for(size_t i = 0; i < N; ++i)
      data[i] = right.data[i];
    return *this;
  }
  MyVector<T,N>& operator=(const MyVectorSum<T,N>& right);
  const T& operator[](size_t i) const { return data[i]; }
  T& operator[](size_t i) { return data[i]; }
};

// Proxy class hold references; uses lazy addition
template<class T, size_t N> class MyVectorSum {
  const MyVector<T,N>& left;
  const MyVector<T,N>& right;
public:
  MyVectorSum(const MyVector<T,N>& lhs,
              const MyVector<T,N>& rhs)
  : left(lhs), right(rhs) {}
  T operator[](size_t i) const {
    return left[i] + right[i];
  }
};
```

⊖ 即《Blitz++》(http://www.oonumerics.org/blitz/)，《the Matrix Template Library》(http://www.osl.iu.edu/research/mtl/) 和《POOMA》(http://www.acl.lanl.gov/pooma/)。

⊖ 指的是在数学中的“向量”，它是一个固定长度、一维的数值数组。

```
// Operator to support v3 = v1 + v2
template<class T, size_t N> MyVector<T,N>&
MyVector<T,N>::operator=(const MyVectorSum<T,N>& right) {
  for(size_t i = 0; i < N; ++i)
    data[i] = right[i];
  return *this;
}

// operator+ just stores references
template<class T, size_t N> inline MyVectorSum<T,N>
operator+(const MyVector<T,N>& left,
          const MyVector<T,N>& right) {
  return MyVectorSum<T,N>(left, right);
}

// Convenience functions for the test program below
template<class T, size_t N> void init(MyVector<T,N>& v) {
  for(size_t i = 0; i < N; ++i)
    v[i] = rand() % 100;
}

template<class T, size_t N> void print(MyVector<T,N>& v) {
  for(size_t i = 0; i < N; ++i)
    cout << v[i] << ' ';
  cout << endl;
}

int main() {
  srand(time(0));
  MyVector<int, 5> v1;
  init(v1);
  print(v1);
  MyVector<int, 5> v2;
  init(v2);
  print(v2);
  MyVector<int, 5> v3;
  v3 = v1 + v2;
  print(v3);
  MyVector<int, 5> v4;
  // Not yet supported:
//!  v4 = v1 + v2 + v3;
} ///:~
```

310

当**MyVectorSum**类产生时，它并不进行计算；它只是持有两个待加向量的引用。仅当访问一个向量和的成员（即它的**operator[ ]( )**）时计算才会发生。为了对**MyVector**的赋值操作符进行重载，将**MyVectorSum**作为一个表达式的参数来使用，如下所示：

311

```
v1 = v2 + v3;  // Add two vectors
```

当对表达式**v1+v2**求值时，返回一个**MyVectorSum**对象（实际上，是一个插入的内联对象，因为**operator+( )**已经声明为**inline**）。这是一个很小的、固定大小的对象（它仅仅持有两个引用）。然后调用上面提到的赋值操作符：

```
v3.operator=<int,5>(MyVectorSum<int,5>(v2, v3));
```

这个运算采用实时运算的方式，将**v1**和**v2**的相应元素相加得到的和赋值给**v3**各自相应的元素。这就不会产生**MyVector**的临时对象。

然而这个程序不支持多于两个操作数的表达式运算，比如

```
v4 = v1 + v2 + v3;
```

原因是在第1次相加后，还会尝试进行第2次相加:

```
(v1 + v2) + v3;
```

这个表达式需要一个重载运算符**operator+( )**，它的第1个参数是**MyVectorSum**类型，第2个参数是**MyVector**类型。可以尝试提供多个重载来满足所有的情况，但最好的办法是让模板来做这项工作，如下面的程序所示:

```
//: C05:MyVector2.cpp
// Handles sums of any length with expression templates.
#include <cstddef>
#include <cstdlib>
#include <ctime>
#include <iostream>
using namespace std;

// A proxy class for sums of vectors
template<class, size_t, class, class> class MyVectorSum;

template<class T, size_t N> class MyVector {
  T data[N];
public:
  MyVector<T,N>& operator=(const MyVector<T,N>& right) {
    for(size_t i = 0; i < N; ++i)
      data[i] = right.data[i];
    return *this;
  }
  template<class Left, class Right> MyVector<T,N>&
  operator=(const MyVectorSum<T,N,Left,Right>& right);
  const T& operator[](size_t i) const {
    return data[i];
  }
  T& operator[](size_t i) {
    return data[i];
  }
};

// Allows mixing MyVector and MyVectorSum
template<class T, size_t N, class Left, class Right>
class MyVectorSum {
  const Left& left;
  const Right& right;
public:
  MyVectorSum(const Left& lhs, const Right& rhs)
  : left(lhs), right(rhs) {}
  T operator[](size_t i) const {
    return left[i] + right[i];
  }
};

template<class T, size_t N>
template<class Left, class Right>
MyVector<T,N>&
MyVector<T,N>::
operator=(const MyVectorSum<T,N,Left,Right>& right) {
  for(size_t i = 0; i < N; ++i)
    data[i] = right[i];
  return *this;
}
// operator+ just stores references
template<class T, size_t N>
inline MyVectorSum<T,N,MyVector<T,N>,MyVector<T,N> >
```

312

```
operator+(const MyVector<T,N>& left,
          const MyVector<T,N>& right) {
  return MyVectorSum<T,N,MyVector<T,N>,MyVector<T,N> >
      (left,right);
}

template<class T, size_t N, class Left, class Right>
inline MyVectorSum<T, N, MyVectorSum<T,N,Left,Right>,
            MyVector<T,N> >
operator+(const MyVectorSum<T,N,Left,Right>& left,
          const MyVector<T,N>& right) {
  return MyVectorSum<T,N,MyVectorSum<T,N,Left,Right>,
                         MyVector<T,N> >
    (left, right);
}
// Convenience functions for the test program below
template<class T, size_t N> void init(MyVector<T,N>& v) {
  for(size_t i = 0; i < N; ++i)
    v[i] = rand() % 100;
}

template<class T, size_t N> void print(MyVector<T,N>& v) {
  for(size_t i = 0; i < N; ++i)
    cout << v[i] << ' ';
  cout << endl;
}

int main() {
  srand(time(0));
  MyVector<int, 5> v1;
  init(v1);
  print(v1);
  MyVector<int, 5> v2;
  init(v2);
  print(v2);
  MyVector<int, 5> v3;
  v3 = v1 + v2;
  print(v3);
  // Now supported:
  MyVector<int, 5> v4;
  v4 = v1 + v2 + v3;
  print(v4);
  MyVector<int, 5> v5;
  v5 = v1 + v2 + v3 + v4;
  print(v5);
} ///:~
```

313

使用模板参数**Left**和**Right**，这个模板很容易地引出一个和的参数类型，来代替上例中指派的那些类型。由于**MyVectorSum**模板持有这额外的两个参数，因此它能表示由**MyVector**和**MyVectorSum**任意组成的一对参数的和。

314

赋值操作符现在是一个成员函数模板。这将允许任一对**<T,N>**与任一对**<Left,Right>**结合，因此一个**MyVector**对象能够得到来自一个**MyVectorSum**对象的赋值，该**MyVectorSum**对象持有**MyVector**和**MyVectorSum**类型的引用，这两个类型的引用可以组成任何可能的一对。

与前面一样，可以通过跟踪一个简单的赋值操作来准确地了解这个地方发生了些什么事情，从下述表达式开始

```
v4 = v1 + v2 + v3;
```

由于结果表达式变得笨拙冗长且难于处理，在下面的解释中，我们用**MVS**作为**MyVectorSum**的缩写，并且忽略模板的参数。

第1个操作是**v1+v2**，这将调用内联函数**operator+( )**，这个内联函数依次将**MVS(v1, v2)**插入到编译流中。然后它被相加到**v3**上，从而表达式**MVS(MVS(v1, v2), v3)**产生一个临时对象。这个完整语句的最后表达结果是:

```
v4.operator+(MVS(MVS(v1, v2), v3));
```

这种转换完全由编译器安排，它也解释了为什么把这种技术冠以“表达式模板”之名。**MyVectorSum**模板代表了一个表达式（上例中是加法表达式），上述的嵌套调用可能也使读者回忆起左关联表达式**v1+v2+v3**的语法分析树。

由Angelika Langer和Klaus Kreft写的一篇优秀文章解释了这项技术如何扩展到更复杂的计算。[⊖]

315

5.7 模板编译模型

读者可能已经注意到，所有例举的模板例子都是将完整定义的模板放在每个编译单元中。（例如，将它们完全放在单文件程序中，或者放在多文件程序的头文件中）。这种方法与传统的编程方法背道而驰，传统的编程方法通过将函数声明放在靠后的头文件中，而将函数实现放在独立的文件中（即，**.cpp**）的方法，使得普通函数的定义与它们的声明相分离。

与这种传统方法分离的理由如下:

- 头文件中的非内联函数体会导致多函数的定义，从而导致链接错误。
- 隐藏来自客户有益的函数实现，从而减少了编译时连接。
- 商家可能将预编译代码（为一个特定的编译器编写）分配到各个头文件中，从而使得用户看不到函数的具体实现。
- 头文件越小，编译时间就越短。

5.7.1 包含模型

另一方面，模板本质上不是代码，而是产生代码的指令。只有模板的实例化才是真正的代码。当一个编译器在编译期间已经看到了一个完整的模板定义，又在同一个翻译单元内碰到了这个模板实例化点的时候，它就必须涉及这样一个事实：一个相同的实例化点可能会呈现在另一个翻译单元内。处理这种情况最普遍的方法，是在每一个翻译单元内都为这个实例化生成代码，让连接器清除这些副本。另一种特殊的方法也可以很好地处理这种情况，就是用不能被内联的内联函数和虚函数表，这也是为什么虚函数表如此流行的原因之一。但是，有一些编译器宁愿依靠更复杂的机制也不愿意多次产生同一个特定的实例化。C++翻译系统也有责任避免这种由于多个相同的实例化点而产生的错误。

这种方法的一个缺点是，所有的模板源代码对客户都是可见的，因此对于销售库的商家而言，几乎没有机会隐藏他们的实现策略。包含模型的另一个缺点是，头文件比函数体分开编译时大多了。这种方式相比传统编译模型而言，大大增加了编译时间。

316

为了帮助减少包含模型所需要的大的头文件，C++提供了两个（不惟一的）可供选择的代码组织机制：可以使用*显式实例化*（explicit instantiation）来手工地实例化每一个模板特化，也可以使用*导出模板*（exported templates），它支持最大限度的独立的编译。

⊖ Langer和Kreft的文章“C++ Expression Templates”见2003年3月的“C/C++ Users Journal”。也可以参见2003年6月同一本杂志上的由Thomas Becker发表的有关表达式模板的文章（该文章是本节内容素材的来源）。

5.7.2 显式实例化

编程人员可以用手工指引编译器实例化他所选择的任何模板特化。当使用这个技术时，对于每个这样的特化，必须有且仅有一条相应的指令；否则可能会收到一个多定义的错误，就像在普通的非内联函数中使用了相同的标识符一样。为了进行示例说明，首先（错误地）将本章前面的例子中的**min( )**模板的声明与定义相分离，遵循普通的非内联函数的标准模式。下面的例子包含了5个文件：

- **OurMin.h**：包含**min( )**函数模板的声明。
- **OurMin.cpp**：包含**min( )**函数模板的定义。
- **UseMin1.cpp**：尝试使用**min( )**的一个**int**型实例化。
- **UseMin2.cpp**：尝试使用**min( )**的一个**double**型实例化。
- **MinMain.cpp**：调用**usemin1( )**和**usemin2( )**。

```
//: C05:OurMin.h
#ifndef OURMIN_H
#define OURMIN_H
// The declaration of min()
template<typename T> const T& min(const T&, const T&);
#endif // OURMIN_H ///:~
```

```
// OurMin.cpp
#include "OurMin.h"
// The definition of min()
template<typename T> const T& min(const T& a, const T& b) {
  return (a < b) ? a : b;
}
```

317

```
//: C05:UseMin1.cpp {O}
#include <iostream>
#include "OurMin.h"
void usemin1() {
  std::cout << min(1,2) << std::endl;
} ///:~
```

```
//: C05:UseMin2.cpp {O}
#include <iostream>
#include "OurMin.h"
void usemin2() {
  std::cout << min(3.1,4.2) << std::endl;
} ///:~
```

```
//: C05:MinMain.cpp
//{L} UseMin1 UseMin2 MinInstances
void usemin1();
void usemin2();

int main() {
  usemin1();
  usemin2();
} ///:~
```

当尝试建立这个程序时，连接器报告有未解析的**min<int>( )**和**min<double>( )**的外部引用。原因是当编译器在**UseMin1**和**UseMin2**中碰到对**min( )**特化的调用时，只有**min( )**的声明是可见的。由于它的定义不可用，编译器认为它可能来源于某些其他的翻译单元，这样一来在这一点上所需的特化就没有被实例化，从而将问题留给了连接器，连接器最终解释它无法找到它们。

为了解决程序中的这个问题，将引入一个新文件**MinInstances.cpp**，它显式地实例化了所需的**min( )**特化：

```
//: C05:MinInstances.cpp {O}
#include "OurMin.cpp"

// Explicit Instantiations for int and double
template const int& min<int>(const int&, const int&);
template const double& min<double>(const double&,
                                   const double&);
///:~
```

318

为了手工实例化一个特定的模板特化，可以在该特化的声明前使用**template**关键字。注意，在这里必须包含**OurMin.cpp**而不是**OurMin.h**，这是因为编译器需要用模板定义来进行实例化。然而，这里也是程序中惟一放置该模板定义的地方[⊖]，因为它提供了程序所需要的独一无二的**min( )**的实例化——在其他的文件中只要有其声明就足够了。由于使用了宏预处理器来包含**OurMin.cpp**，因而需要加入包含警告：

```
//: C05:OurMin.cpp {O}
#ifndef OURMIN_CPP
#define OURMIN_CPP
#include "OurMin.h"

template<typename T> const T& min(const T& a, const T& b) {
  return (a < b) ? a : b;
}
#endif // OURMIN_CPP ///:~
```

现在，当把所有的文件一起编译为一个完整的程序时，就会找到**min( )**的惟一的实例，程序就可以正确运行，输出结果如下：

```
1
3.1
```

编程人员也可以手工实例化类和静态数据成员。当显式实例化一个类时，除了一些之前可能已经显式实例化了的成员外，特化所需要的所有成员函数都要进行实例化。这一点很重要，因为使用这种机制时，它必须舍弃很多无用的模板——某些特定的模板将依据它们的参数类型实现不同的功能。隐式实例化在此处有优势：其中只有被调用的成员函数才进行实例化。

319

显式实例化多用于大型软件工程项目中，因为这样可以避免大量的编译时间。采用隐式实例化还是采用显式实例化完全独立于使用哪一个模板进行编译。可以通过使用包含模型或者分离模型（下节讨论）中的任何一种模型来进行手工实例化。

5.7.3 分离模型

模板编译的分离模型跨越了所有的翻译单元，将函数模板定义或者静态数据成员定义从它们的声明中分离出来，就像使用导出（exporting）模板机制下的普通函数和数据一样。在学习了前两节的内容后，这种说法似乎听起来有点儿奇怪。读者首先就可能会问：如果包含模型使用得很顺手，为什么还要怀疑它呢？原因有两个：有历史原因也有技术原因。

从历史上看，包含模型是第1个经历广泛的商品化使用的模型——所有的C++编译器都支持包含模型。其中的部分原因是，在进行标准化的过程中直到该过程后期也没有能够很好地说明

⊖ 正如前面解释的那样，在每一个程序中只能一次显式实例化一个模板。

分离模型，再一个原因就是由于包含模型本身更容易实现一些。在分离模型的语义定下来之前的很长一段时间里，就已经存在很多正在运行着的与之相关的代码了。

分离模型实现起来是如此的困难，以至于直到2003年夏天，仅有一个编译器前端（EDG）支持分离模型。那时，这个编译器如有请求，它仍旧要求模板源代码在编译时可以用来执行实例化操作。方法是在适当的位置使用一些中介代码，来取代总是要求最初的源代码随时准备好以备使用的形式。这样就可以在不需要传递源代码的情况下传递某些“预编译”模板。鉴于本章前面介绍过的查找的复杂性（就是有关在模板定义的语境中查找关联名称的内容），当编译一个实例化某个模板的程序时，仍然要以某种形式来使用一个完整模板的定义。

将一个模板定义的源代码与它的声明相分离的程序语法是很简单的。只要使用**export**关键字就可以了：

```
// C05:OurMin2.h
// Declares min as an exported template
// (Only works with EDG-based compilers)
#ifndef OURMIN2_H
#define OURMIN2_H
export template<typename T> const T& min(const T&,
                                         const T&);
#endif // OURMIN2_H ///:~
```

320

类似于**inline**或者**virtual**，关键字**export**在一个编译流中仅需出现一次。在这个编译流中，引入了一个导出模板。由于这个原因，我们在实现文件中无需重复它，但是再对它进行一下声明是一个好习惯。

```
// C05:OurMin2.cpp
// The definition of the exported min template
// (Only works with EDG-based compilers)
#include "OurMin2.h"
export
template<typename T> const T& min(const T& a, const T& b) {
  return (a < b) ? a : b;
} ///:~
```

之前用过的**UseMin**文件只需包含正确的头文件（**OurMin2.h**），主程序不用改动。尽管看起来这已经产生了正确的分离，但带有模板定义的文件（**OurMin2.cpp**）仍然必须传递给用户（因为**min( )**的每一个实例化都必须进行处理），直至遇到这样的情况：表示模板定义的某种中介代码形式得到支持。因此，当一个正确的分离模型标准提出来的时候，其中所有的好处并不会都在今天马上体现出来。当今只有一个编译器家族支持**export**（那些基于EDG前端的编译器），而且这些编译器当前并没有开发将模板定义分配到已编译的格式中的潜力。

5.8 小结

模板广泛使用的程度远远超过简单的类型参数化。当对模板结合使用参数类型推断、用户自定义特化和模板元编程的时候，C++模板作为一种强有力的代码产生机制已经形成了。

这里有一个我们没有提及的C++模板的缺陷，就是在解释编译时错误信息报告方面的困难。由于编译器产生总量难以预料的出错信息报告文本可能是完全无法避免的。现在，C++编译器已经改进了它们的模板错误信息报告方式，此外Leor Zolman也已开发了一种名为**STLFilt**的工具，这种工具采用提取有用信息和抛掉冗余信息[⊖]的方式，使得汇报的这些错误信息更具可读性。 321

⊖ 访问http://www.bdsoft.com/tools/stlfilt.html。

读者从本章得到的另一个重要的思想就是，一个模板意味着一个接口。也就是说，尽管关键字**template**意味着："我可以接受任何类型的参数"，但在模板定义中的代码也要求某些需要提供支持的运算符和成员函数——这些运算符和成员函数就是接口。因此，实际上一个模板定义意味着："我将接受任何支持这个接口的参数"。若编译器能够仅仅说："嗨，这种实例化模板的类型参数不支持这个接口——不能使用这个类型"，事情会变得更好一些。运用模板构成的带有"隐含的类型检查"的接口机制，比起那些要求所有的类型都必须从某些基类派生出来的纯面向对象的使用惯例来说更加灵活。

在第6章和第7章中，将深入探讨模板最著名的应用：标准C++库的子集，即广为人知的标准模板库（STL）。第9章和第10章中也用到了本章未提及的某些模板技术。

5.9 练习

5-1 编写一个具有单一类型模板参数的一元函数模板。用类型**int**生成它的一个完全特化。再为这个拥有单一的**int**参数的函数产生一个非模板重载。在主函数中调用这三个函数。

5-2 编写一个类模板，该类模板用**vector**实现一个栈数据结构。

5-3 对习题（5-2）的解答进行修改，使得用来实现栈的容器的类型是一个模板类型的模板参数。

322 5-4 在下面的代码中，类**NonComparable**中没有**operator=( )**。解释一下为什么类**HardLogic**的出现引起了一个编译错误，而**SoftLogic**却没有？

```
//: C05:Exercise4.cpp {-xo}
class Noncomparable {};

struct HardLogic {
  Noncomparable nc1, nc2;
  void compare() {
    return nc1 == nc2; // Compiler error
  }
};

template<class T> struct SoftLogic {
  Noncomparable nc1, nc2;
  void noOp() {}
  void compare() {
    nc1 == nc2;
  }
};

int main() {
  SoftLogic<Noncomparable> l;
  l.noOp();
} ///:~
```

5-5 编写一个持有单一类型参数（T）的函数模板，它接受了4个函数参数：一个T类数组、一个开始索引值、一个结束索引值（在允许范围之内的）和一个可选择的初始值。函数返回指定范围内所有数组元素值与初始值的和。用默认构造函数为T类型的数据用默认方式赋初值。

5-6 重做上面的习题，根据本章讨论过的技术，使用显式实例化来手工生成**int**和**double**的特化。

5-7 请指出为什么下列代码无法编译？（提示：看看类成员函数访问了什么？）

```
//: C05:Exercise7.cpp {-xo}
class Buddy {};

template<class T> class My {
  int i;
public:
  void play(My<Buddy>& s) {
    s.i = 3;
  }
};

int main() {
  My<int> h;
  My<Buddy> me, bud;
  h.play(bud);
  me.play(bud);
} ///:~
```

323

5-8 指出下面的代码为什么无法编译？

```
//: C05:Exercise8.cpp {-xo}
template<class T> double pythag(T a, T b, T c) {
  return (-b + sqrt(double(b*b - 4*a*c))) / 2*a;
}

int main() {
  pythag(1, 2, 3);
  pythag(1.0, 2.0, 3.0);
  pythag(1, 2.0, 3.0);
  pythag<double>(1, 2.0, 3.0);
} ///:~
```

5-9 编写一些持有下列多种无类型参数的模板：一个**int**、一个指向**int**的指针、一个指向**int**类型的静态类成员的指针和一个指向一个静态成员函数的指针。

5-10 编写一个持有两个类型参数的类模板。为第1个参数定义半特化，另一个半特化指定第2个参数。在每一个特化中，都采用没有出现在基本模板中的成员。

5-11 定义一个持有单一类型参数的类模板，将其命名为**Bob**。使**Bob**成为一个名为**Friendly**的模板类的所有实例的友元，并且成为一个名为**Picky**的类模板的友元——仅当**Bob**和**Picky**的类型参数完全相同的时候。提供一些能证明这些类的友元关系的**Bob**成员函数。

第6章 通用算法

算法是计算的核心。能够编写出在任何一种类型序列下工作的算法，就可以使程序更加简单和安全。算法在运行时的自定义的能力革新了软件开发方式。

众所周知，标准模板库（Standard Template Library，STL）作为标准C++库的子集，最初是用来设计通用算法（generic algorithm）的——以类型安全的方式生成处理任何一种类型值序列的代码。以前每当需要处理一个数据集合的时候，就得重复地手工编写代码。设计STL的目标就是对于几乎每个任务都使用预定义的算法，来代替这种手工编码的工作。然而，这种方法在提供方便的同时，也为初学者的学习带来了一些困难。在学习完本章后，读者就能自己做出判定，是特别喜欢采用这些算法来进行编程还是越学越困惑。大多数人起初抵制算法的使用，但随着时间的推移，越来越多的人逐渐喜欢使用它们了。

6.1 概述

除了一些别的东西，标准库中的通用算法还提供一个词汇表来描述各种解法。随着对算法的熟悉，读者就会获得一个新的词汇集合用来讨论现在正在做什么，这些词汇往往比以前所用的词汇具有更高层次的抽象。例如，没有必要说“这个循环在运行过程中从这赋值到那，……，噢，我知道了，是复制！”，而是只需简单扼要地用**copy( )**就可以了。这就是初学者在早期的计算机编程中所做的——创建高层次的抽象来解释正在做什么以及用更少的时间来说明怎样去做。一旦解决了怎样做的问题，并且将其转换成代码隐藏于算法代码中，就可以在需要时重复使用这些算法。

这里有一个怎样使用**copy**算法的程序例子：


```
//: C06:CopyInts.cpp
// Copies ints without an explicit loop.
#include <algorithm>
#include <cassert>
#include <cstddef>  // For size_t
using namespace std;

int main() {
  int a[] = { 10, 20, 30 };
  const size_t SIZE = sizeof a / sizeof a[0];
  int b[SIZE];
  copy(a, a + SIZE, b);
  for(size_t i = 0; i < SIZE; ++i)
    assert(a[i] == b[i]);
} ///:~
```

copy( )算法的前两个参数表示输入序列的范围——此处是数组**a**。范围用一对指针表示。第1个指针指向该序列的第1个元素，第2个指针指向数组的*超越末尾的*（past the end）位置（即数组的最后一个元素的后面）。刚开始看起来可能觉得比较陌生，但这是传统的C语言的习惯用法，可以带来很大的便利。例如，这两个指针的差值就是序列中元素的个数。更重要的是，在实现**copy**时，第2个指针可以作为在序列中停止迭代的标记符。第3个参数代表输出序列的开始位置，在本例中输出序列是数组**b**。这里假设**b**表示的数组有足够的空间来接收要复制的

元素。

如果**copy( )**算法仅限于处理整数，那就没什么新奇的地方了。它还可以用于任何一种类型的序列。下面的例子用来复制**string**对象：

```
//: C06:CopyStrings.cpp
// Copies strings.
#include <algorithm>
#include <cassert>
#include <cstddef>
#include <string>
using namespace std;

int main() {
  string a[] = {"read", "my", "lips"};
  const size_t SIZE = sizeof a / sizeof a[0];
  string b[SIZE];
  copy(a, a + SIZE, b);
  assert(equal(a, a + SIZE, b));
} ///:~
```

327

这个例子引入了另一个算法**equal( )**，仅当第1个序列的每一个元素与第2个序列的对应元素相等时（使用**operator==( )**）返回**true**。这个例子对每个序列遍历了两次，一次用来复制，一次用来比较，而不是采用单一的一次循环！

通用算法能够达到如此灵活性是由于采用了函数模板。如果读者能将**copy( )**的实现想像成下面形式，这样差不多就对了：

```
template<typename T> void copy(T* begin, T* end, T* dest) {
  while(begin != end)
    *dest++ = *begin++;
}
```

说“差不多”是因为**copy( )**能够处理这样一类的序列，该序列由类似指针的任意类型来限制，例如迭代器。以此方式，**copy( )**可以用来复制**vector**：

```
//: C06:CopyVector.cpp
// Copies the contents of a vector.
#include <algorithm>
#include <cassert>
#include <cstddef>
#include <vector>
using namespace std;

int main() {
  int a[] = { 10, 20, 30 };
  const size_t SIZE = sizeof a / sizeof a[0];
  vector<int> v1(a, a + SIZE);
  vector<int> v2(SIZE);
  copy(v1.begin(), v1.end(), v2.begin());
  assert(equal(v1.begin(), v1.end(), v2.begin()));
} ///:~
```

第1个**vector**对象**v1**由数组**a**中的整数序列来初始化。第2个**vector**对象 **v2**的定义，使用一个不同的能够为**SIZE**个元素分配空间的**vector**构造函数，并且将其初始化为0（整型的默认值）。

328

如同前面的数组例子，**v2**要有足够的空间来接收**v1**内容的复制。为了方便起见，使用一个特殊的库函数**back_inserter( )**，该函数返回一个特殊类型的迭代器。利用这个迭代器就可以用插入元素的方式来代替重写这些元素，因此内存的大小就可以根据容器的需要自动扩大。

下面的例子使用了**back_inserter( )**，因此无需像前面例子那样，在建立输出**vector**的对象**v2**时必须确定其大小。

```
//: C06:InsertVector.cpp
// Appends the contents of a vector to another.
#include <algorithm>
#include <cassert>
#include <cstddef>
#include <iterator>
#include <vector>
using namespace std;

int main() {
  int a[] = { 10, 20, 30 };
  const size_t SIZE = sizeof a / sizeof a[0];
  vector<int> v1(a, a + SIZE);
  vector<int> v2;  // v2 is empty here
  copy(v1.begin(), v1.end(), back_inserter(v2));
  assert(equal(v1.begin(), v1.end(), v2.begin()));
} ///:~
```

back_inserter( )函数在头文件**<iterator>**中定义。我们将在下一章详细解释插入迭代器是如何工作的。

迭代器在本质上与指针相同，所以可以在标准库中以一种能够接受迭代器和指针两种参数的方式来实现算法。因此，**copy( )**的实现如下所示：

```
template<typename Iterator>
void copy(Iterator begin, Iterator end, Iterator dest) {
  while(begin != end)
    *begin++ = *dest++;
}
```

调用时无论采用哪种类型的参数，**copy( )**都假定它正确地实现了间接引用和自增运算符。如果没有，就会得到一个编译时错误。

329

6.1.1 判定函数

有时只想复制定义好的某个序列中的一个子集到另一个序列中，这个子集由只满足某个特殊条件的那些元素组成。为了达到这种灵活性，很多算法的调用序列允许提供一个*判定函数*(predicate)，即一个基于某种标准返回布尔型值的函数。例如，只想提取整数序列中那些小于或等于15的数。**copy( )**的一种称为**remove_copy_if( )**的版本能够完成这一工作，如下所示：

```
//: C06:CopyInts2.cpp
// Ignores ints that satisfy a predicate.
#include <algorithm>
#include <cstddef>
#include <iostream>
using namespace std;

// You supply this predicate
bool gt15(int x) { return 15 < x; }

int main() {
  int a[] = { 10, 20, 30 };
  const size_t SIZE = sizeof a / sizeof a[0];
  int b[SIZE];
  int* endb = remove_copy_if(a, a+SIZE, b, gt15);
  int* beginb = b;
  while(beginb != endb)
```

```
    cout << *beginb++ << endl; // Prints 10 only
} ///:~
```

remove_copy_if( )函数模板需要一些通常用来限定范围的指针，还增加了一个用户自选的判定函数。判定函数必须是指向一个函数[⊖]的指针，这个指针有一个与序列中元素同类型的参数，并且必须返回一个布尔型的值。在这里，当参数大于15时，函数**gt15**返回**true**。**remove_copy_if( )**算法对输入序列的每个元素都应用**gt15( )**，并且在向输出序列写入时忽略掉那些使判定函数产生真值的元素。

下面的程序展示了**copy**算法的另外一个变种：

330

```
//: C06:CopyStrings2.cpp
// Replaces strings that satisfy a predicate.
#include <algorithm>
#include <cstddef>
#include <iostream>
#include <string>
using namespace std;

// The predicate
bool contains_e(const string& s) {
  return s.find('e') != string::npos;
}

int main() {
  string a[] = {"read", "my", "lips"};
  const size_t SIZE = sizeof a / sizeof a[0];
  string b[SIZE];
  string* endb = replace_copy_if(a, a + SIZE, b,
    contains_e, string("kiss"));
  string* beginb = b;
  while(beginb != endb)
    cout << *beginb++ << endl;
} ///:~
```

与刚才忽略不满足判定函数的元素不同，此例中的**replace_copy_if( )**在输出一个序列时用一个固定的值来替代这些元素。输出结果是：

```
kiss
my
lips
```

因为“read”是几个输入字符串中惟一一个含有字母e的字符串，所以用字符串“kiss”来取代该字符串，“kiss”是**replace_copy_if( )**调用中指定的最后一个参数。

replace_if( )算法改变原始序列相应位置中的内容，而不是向单独的输出序列中写数据，程序如下所示：

```
//: C06:ReplaceStrings.cpp
// Replaces strings in-place.
#include <algorithm>
#include <cstddef>
#include <iostream>
#include <string>
using namespace std;

bool contains_e(const string& s) {
```

331

⊖ 或者是和函数一样可被调用的东西，我们将很快能遇到。

```
  return s.find('e') != string::npos;
}

int main() {
  string a[] = {"read", "my", "lips"};
  const size_t SIZE = sizeof a / sizeof a[0];
  replace_if(a, a + SIZE, contains_e, string("kiss"));
  string* p = a;
  while(p != a + SIZE)
    cout << *p++ << endl;
} ///:~
```

6.1.2 流迭代器

像任何好的软件库一样，标准C++库试图提供便捷的方法以自动完成常见的任务。在本章开始部分曾经提到过，可以使用通用算法来取代循环结构的设想。但是到目前为止，在提出的例子中仍然直接使用循环来打印输出结果。因为打印输出结果是最常见的任务之一，也可以期望有一种方法能够自动实现它。

这里引入流迭代器（stream iterator）的概念。一个流迭代器使用流作为输入或输出序列。例如，为了去除程序**CopyInts2.cpp**中的输出循环，可以像下面这样做：

```
//: C06:CopyInts3.cpp
// Uses an output stream iterator.
#include <algorithm>
#include <cstddef>
#include <iostream>
#include <iterator>
using namespace std;

bool gt15(int x) { return 15 < x; }
int main() {
  int a[] = { 10, 20, 30 };
  const size_t SIZE = sizeof a / sizeof a[0];
  remove_copy_if(a, a + SIZE,
                 ostream_iterator<int>(cout, "\n"), gt15);
} ///:~
```

332

在本例中，**remove_copy_if( )**的第3个参数位置上的输出序列**b**用一个输出流迭代器来代替，这个迭代器是在头文件**<iterator>**中声明的**ostream_iterator**类模板的一个实例。输出流迭代器重载其拷贝-赋值操作符，该重载操作符向相应的流写数据。**ostream_iterator**的这个特殊实例应用于输出流**cout**。每次**remove_copy_if( )**通过迭代器对**cout**赋一个来自输入序列**a**的整数。即迭代器向**cout**写入这个整数，并且随后还自动写入一个单独的字符串的一个实例，该字符串位于它的第2个参数位置上，在本例中是一个换行符。

用一个输出文件流来代替**cout**，就使得写文件同样很容易实现：

```
//: C06:CopyIntsToFile.cpp
// Uses an output file stream iterator.
#include <algorithm>
#include <cstddef>
#include <fstream>
#include <iterator>
using namespace std;

bool gt15(int x) { return 15 < x; }

int main() {
  int a[] = { 10, 20, 30 };
```

```
  const size_t SIZE = sizeof a / sizeof a[0];
  ofstream outf("ints.out");
  remove_copy_if(a, a + SIZE,
                 ostream_iterator<int>(outf, "\n"), gt15);
} ///:~
```

一个输入流迭代器允许算法从输入流中获得它的输入序列。这是靠构造函数和**operator++( )**从基础的流中读入下一个元素，以及重载**operator*( )**产生先前读入的值来完成的。因为算法需要两个指针来限定输入序列，所以可以用两种方式来构造**istream_iterator**，请看下面的程序: 333

```
//: C06:CopyIntsFromFile.cpp
// Uses an input stream iterator.
#include <algorithm>
#include <fstream>
#include <iostream>
#include <iterator>
#include "../require.h"
using namespace std;

bool gt15(int x) { return 15 < x; }

int main() {
  ofstream ints("someInts.dat");
  ints << "1 3 47 5 84 9";
  ints.close();
  ifstream inf("someInts.dat");
  assure(inf, "someInts.dat");
  remove_copy_if(istream_iterator<int>(inf),
                 istream_iterator<int>(),
                 ostream_iterator<int>(cout, "\n"), gt15);
} ///:~
```

程序中**replace_copy_if( )**的第1个参数，把一个**istream_iterator**的对象应用于含有整数的输入文件流。第2个参数使用**istream_iterator**类的默认构造函数。这个调用构造了**istream_iterator**的一个特殊值，用以指示文件的结束。这样，当第1个迭代器最终遇到物理文件的结尾时，它与**istream_iterator<int>( )**的值进行是否相等的比较，以便算法正确结束。注意，本例中完全避免了直接使用数组。

6.1.3 算法复杂性

使用某个软件库是对它的一种信任。用户相信（软件库的）实现者不仅能提供正确的功能，并且希望这些功能能够尽可能有效地执行。与其使用性能低下的算法，还不如自己来编写循环代码。

为了保证库实现的质量，C++标准不仅说明了算法应该做什么，而且说明了包括做得有多快，有时还包括应该使用多少存储空间。不能满足性能需求的算法也就是不符合标准的算法。算法的执行效率的度量称为复杂性（complexity）。 334

如果可能的话，C++标准会指定一个算法应该耗费的操作的精确次数。例如，**count_if( )**算法返回一个序列中满足给定判定函数的元素的个数。下面对**count_if( )**的调用，如果应用于类似本章前面的例子中的整数序列，会产生大于15的整数元素的个数:

```
size_t n = count_if(a, a + SIZE, gt15);
```

因为**count_if( )**必须对每个元素仔细检查一次，也就是比较的次数与序列中元素的个数肯定相等。**copy( )**算法有相同的规格说明。

对于其他算法可以指定其最多执行的操作次数。**find( )**算法要搜索一个序列，直到遇到一个等于它的第3个参数的元素:

```
int* p = find(a, a + SIZE, 20);
```

只要找到这样的元素就停止查找，并且返回一个指针，这个指针指向该元素第1次出现的位置。如果一个也没有找到，也返回一个指针，该指针指向超越序列末尾的（本例中是**a+SIZE**）位置。所以，**find( )**比较的次数最多等于序列中元素的个数。

有时候不能精确衡量一个算法将耗费运算的次数。在这种情况下，C++标准给出算法的渐近复杂性(asymptotic complexity)，这是对算法在大的序列输入下执行性能与已知公式相比较的度量。一个好的例子是**sort( )**算法，C++标准称其花费“在平均情况下约**nlogn**次比较”（**n**是序列中元素的个数）[⊖]。这样的复杂性度量似乎给人们一种关于一个算法的开销的“感觉”，不管怎么样，这至少是一种用来比较算法性能的有意义的基本依据。在下一章中读者将会看到，对于容器**set**的成员函数**find( )**来说，它具有对数级的复杂性，这意味着在大的集合中查找所花费的时间与元素个数的对数成正比。这比元素个数**n**要小的多，因此查找一个**set**最好使用它自己的成员函数**find( )**而不用一般的**find( )**算法。

335

6.2 函数对象

学习本章前面的例子，读者可能会注意到函数**gt15( )**的使用限制。如果用其他数而不是15来作为比较的阈值该怎么办？可能会需要**gt20( )**或**gt25( )**等等。为它们再编写单独的函数会耗费时间而且不合理，因为当编写应用代码时程序员需要知道所有要求的值。

后者的限制意味着不能使用运行时的值[⊖]来控制查找，这是不能接受的。为了克服这个困难，需要有一种在运行时把信息传递给判定函数的方式。例如，程序员可能需要一个能用任意比较值来初始化一个判定大于的函数（greater-than function）。遗憾的是，不能把这个值作为一个函数参数进行传递，因为这是个一元判定函数，比如**gt15( )**。它单独地应用于序列中的每一个值，因此必须只能有一个参数。

和往常一样，跳出这个两难的局面的方法就是创建一个抽象。在这里，需要这样的一个抽象，它实现起来像函数同时保存状态，使用时却不用考虑函数参数的个数。这种抽象称为*函数对象*(function object)。[⊗]

函数对象是重载了**operator( )**的类的一个实例，**operator( )**是函数调用运算符。这个运算符允许用函数调用语法并使用对象。如同其他对象一样，可以通过该对象的构造函数来初始化它。下面程序中的函数对象可以取代**gt15( )**:

336

```
//: C06:GreaterThanN.cpp
#include <iostream>
using namespace std;

class gt_n {
  int value;
public:
  gt_n(int val) : value(val) {}
  bool operator()(int n) { return n > value; }
};
```

⊖ 这是**O(n log n)**的英文简单表述，其表示对于大的**n**比较的次数与函数**f(n)= n log n**成正比。

⊖ 除非使用全局变量来烦琐地实现。

⊗ 函数对象也称*函数子*（functor），函数子是一个具有类似行为的数学概念。

```
int main() {
  gt_n f(4);
  cout << f(3) << endl;  // Prints 0 (for false)
  cout << f(5) << endl;  // Prints 1 (for true)
} ///:~
```

当创建函数对象f时，传递作为比较对照的固定值（4）。编译器像下面的函数调用一样计算表达式**f(3)**:

```
f.operator()(3);
```

返回值是表达式**3>value**的值，在本例中当**value**是4时为假。

因为这样的比较还可以应用于除**int**外的其他类型，只要把**gt_n( )**定义为一个类模板就有意义了。无需用户亲自做——标准库已经帮你做了。下面对函数对象的描述不仅使这个主题更清晰，而且读者对通用算法如何工作有了一个更好的理解。

6.2.1 函数对象的分类

标准C++库根据函数对象的运算符**operator( )**使用参数的个数和返回值的类型对其进行分类。这种分类是基于函数对象的运算符**operator( )**使用参数的个数分别为零个、一个或两个的情况进行:

发生器(Generator): 一种没有参数且返回一个任意类型值的函数对象。随机数发生器就是发生器的一个例子。标准库提供一个发生器，就是在**<cstdlib>**中声明的函数**rand( )**以及一些算法，如**generate_n( )**，它将发生器应用于序列。 337

一元函数(Unary Function): 一种只有一个任意类型的参数，且返回一个可能不同类型（比如可能是**void**）值的函数对象。

二元函数(Binary Function): 一种有两个任意类型的（可能是不同类型）参数，且返回一个任意类型（包括**void**）值的函数对象。

一元判定函数(Unary Predicate): 返回**bool**型值的一元函数。

二元判定函数(Binary Predicate): 返回**bool**型值的二元函数。

严格弱序(Strict Weak Ordering): 一种更广义地理解“相等”概念的二元判定函数。一些标准容器认为在两个元素中，如果任何一个都不小于另外一个（使用**operator<( )**），则二者相等。这对于比较浮点型值及其他类型的对象很重要，因为**operator==( )**是不可靠的或者说是不可行的。同时这种概念也适用于想在**struct**的所有字段的一个子集上对数据记录（**struct**）的序列进行排序的情况。这种比较方案被认为是一种严格弱序，因为具有相等关键字集（equal key）的两个记录作为对象的整体而言，两个记录不是真正的“相等”，但对于正在使用的这个比较来说是相等的。这种观念的重要性在下一章中将会更加明显。

另外，某些算法假定对它们处理的对象类型的有关操作是有效的。现在用下面这些术语来介绍这些假定:

小于可比较(LessThanComparable): 含有小于运算符**operator<**的类。

可赋值(Assignable): 含有对于同类型指定赋值操作符**operator=**的类。

相等可比较(EqualityComparable): 含有对于同类型相等运算符**operator==**的类。

在本章后面将使用这些术语来描述标准库中的通用算法。

6.2.2 自动创建函数对象

头文件**<functional>**定义了大量有用的通用函数对象。毋庸置疑，它们是很简单的，但可以用它们来组成更加复杂的函数对象。因此，在大多数情况下，无需编写任何函数就可以构 338

造出复杂的判定函数。可以用函数对象适配器（function object adaptor）[⊖]来获得一个简单的函数对象，并且调整它们用来与操作链中的其他函数对象配合。

举例说明，现在仅使用标准函数对象来完成前面介绍的**gt15( )**的工作。标准函数对象**greater**是一个二元函数对象，当它的第1个参数大于第2个参数时返回**true**。不能通过一个算法如**remove_copy_if( )**直接把它应用到整数序列，因为**remove_copy_if( )**是一个一元判定函数。可以通过用**greater**的第1个参数与某个固定值进行比较的方式，来构造一个一元判定函数。用函数对象适配器**bind2nd**作为固定的第2个参数的值，其值为15，如下列程序所示：

```
//: C06:CopyInts4.cpp
// Uses a standard function object and adaptor.
#include <algorithm>
#include <cstddef>
#include <functional>
#include <iostream>
#include <iterator>
using namespace std;

int main() {
  int a[] = { 10, 20, 30 };
  const size_t SIZE = sizeof a / sizeof a[0];
  remove_copy_if(a, a + SIZE,
                 ostream_iterator<int>(cout, "\n"),
                 bind2nd(greater<int>(), 15));
} ///:~
```

339

这个程序没用前面用户自己定义的判定函数**gt15( )**，却产生了与**CopyInts3.cpp**相同的结果。函数对象适配器**bind2nd( )**是一个模板函数，它创建一个**binder2nd**类型的函数对象。仅存储和传递两个参数给**bind2nd( )**，其中第1个参数必须是一个二元函数或函数对象（即带有两个参数的可以被调用的任意对象）。**binder2nd**中的**operator( )**函数，它本身是一个一元函数，该函数调用存储的二元函数，并传递引入的参数及其存储的固定值。

为了更具体地解释这个例子，现在调用由**bind2nd( )**创建的**binder2nd**的一个名为**b**的实例。当创建**b**时，它接收两个参数（**greater<int>( )**和15）并且保存它们。调用名为**g**的**greater<int>**的实例和名为**o**的输出流迭代器的实例。这时在前面的程序中对**remove_copy_if( )**的调用在概念上可表示成下面这样：

```
remove_copy_if(a, a + SIZE, o, b(g, 15).operator());
```

伴随着**remove_copy_if( )**在序列中的迭代，对每个元素调用**b**来决定当复制到目的流时是否忽略该元素。如果用**e**来标记当前元素，**remove_copy_if( )**中的调用等价于：

```
if(b(e))
```

但是**binder2nd**的函数调用运算符还要回来调用**g(e,15)**，所以上面的调用与下面的调用一样：

```
if(greater<int>(e, 15))
```

这就是我们要寻求的比较。这里还有一个**bind1st( )**适配器，它创建一个**binder1st**对象，该对象是相关联的输入二元函数确定的第一个参数。

这里有另外一个例子，用来计算某个序列中不等于20的元素的个数，这次使用前面介绍过

⊖ 依照C++标准，在这里的写成adaptor。在关于设计模式章节中我们根据习惯使用adapter。这两种写法都是可接受的。

的算法**count_if( )**。程序中有一个标准二元函数对象**equal_to**，还有一个函数对象适配器**not1( )**，该函数对象适配器以一元函数对象作为参数并转化其实际值。下面的程序将会完成这个任务：

```
//: C06:CountNotEqual.cpp
// Count elements not equal to 20.
#include <algorithm>
#include <cstddef>
#include <functional>
#include <iostream>
using namespace std;

int main() {
  int a[] = { 10, 20, 30 };
  const size_t SIZE = sizeof a / sizeof a[0];
  cout << count_if(a, a + SIZE,
                   not1(bind1st(equal_to<int>(), 20))):// 2
} ///:~
```

340

如在前面的例子中的**remove_copy_if( )**一样，**count_if( )**调用由位于它的第3个参数（称其为**n**）位置的函数对序列中的每一个元素进行判定，并且在每次返回**true**时使其内部的计数器增1。如前所述，如果称序列中当前元素为**e**，则语句

```
if(n(e))
```

在**count_if**实现过程中，可以解释为

```
if(!bind1st(equal_to<int>, 20)(e))
```

结束时如下所示：

```
if(!equal_to<int>(20, e))
```

这是因为**not1( )**返回的是调用它的一元函数参数的结果的逻辑否定。**equal_to**的第1个参数是20，因为在这里用**bind1st( )**来代替**bind2nd( )**。由于在参数中相等性测试是对称的，在这个例子中可以使用**bind1st( )**或**bind2nd( )**。

下面的表格显示了产生标准函数对象的模板，还显示了模板应用的表达式的种类：

名 称	类 型	产生的结果
plus	BinaryFunction	arg1+arg2
minus	BinaryFunction	arg1-arg2
multiplies	BinaryFunction	arg1*arg2
divides	BinaryFunction	arg1/arg2
modulus	BinaryFunction	arg1%arg2
negate	UnaryFunction	-arg1
equal_to	BinaryPredicate	arg1==arg2
not_equal_to	BinaryPredicate	arg1=arg2
greater	BinaryPredicate	arg1>arg2
less	BinaryPredicate	arg1<arg2
greater_equal	BinaryPredicate	arg1>=arg2
less_equal	BinaryPredicate	arg1<=arg2
logical_and	BinaryPredicate	arg1&&arg2
Logical_or	BinaryPredicate	arg1||arg2
logical_not	UnaryPredicate	!arg1
unary_negate	Unary Logical	!(UnaryPredicate(arg1))
binary_negate	Binary Logical	!(BinaryPredicate(arg1,arg2))

341

6.2.3 可调整的函数对象

标准函数适配器例如**bind1st( )**和**bind2nd( )**，对它们处理的函数对象做一些相关的假设。考虑前面的**CountNotEqual.cpp**程序中最后一行的表达式：

```
not1(bind1st(equal_to<int>(), 20))
```

342 **bind1st( )**适配器创建了一个**binder1st**类型的一元函数对象，它仅存储**equal_to<int>**的一个实例及值20。函数**binder1st::operator( )**需要知道它的参数类型和它的返回值类型；否则，它就不是一个有效的声明。解决这一问题的简便方式是，期望所有的函数对象提供这些类型的嵌套类型定义。对于一元函数，是类型名为**argument_type**和**result_type**；对于二元函数对象，为**first_argument_type**、**second_argument_type**和**result_type**。看看头文件**<functional>**中**bind1st( )**和**binder1st**的实现就显示了这一期望。首先检查一下可能出现在典型的库实现中的**bind1st( )**：

```
template<class Op, class T>
binder1st<Op> bind1st(const Op& f, const T& val) {
  typedef typename Op::first_argument_type Arg1_t;
  return binder1st<Op>(f, Arg1_t(val));
}
```

注意，模板参数**Op**，代表正在由**bind1st( )**调整的二元函数的类型，它必须含有一个名为**first_argument_type**的嵌套类型。（注意，如同在第5章中解释的那样，使用**typename**来通知编译器它是一个成员类型名。）现在看看**binder1st**在它的函数调用运算符的声明中如何使用**Op**中的类型名：

```
// Inside the implementation for binder1st<Op>
typename Op::result_type
operator()(const typename Op::second_argument_type& x)
  const;
```

为这些类提供类型名的函数对象，称为*可调整的函数对象*（adaptable function object）。

因为所有的标准函数对象以及用户使用函数对象适配器自己创建的函数对象都期望这些类型名称，所以头文件**<functional>**提供了两种模板来定义这些类型：**unary_function**和**binary_function**。当为派生自这些类的简单函数对象填写参数类型时，可以由这些类型作为模板参数。例如，假设要想使本章前面定义的函数对象**gt_n**成为可调整的，我们需要做的工作如下所示：

```
class gt_n : public unary_function<int, bool> {
  int value;
public:
  gt_n(int val) : value(val) {}
  bool operator()(int n) {
    return n > value;
  }
};
```

343

所有的**unary_function**都提供合适的类型定义，这些正如读者在定义中所见到的，由模板参数推断而来：

```
template<class Arg, class Result> struct unary_function {
  typedef Arg argument_type;
  typedef Result result_type;
};
```

这些类型通过**gt_n**变成可使用的，因为它是从**unary_function**公有派生来的。

binary_function模板使用起来与其类似。

6.2.4 更多的函数对象例子

下面的**FunctionObjects.cpp**例子，对多数内建的基本函数对象模板提供了简单的测试。读者可以看到，用这种方式如何使用每个模板，并取得相应的操作结果。为简便起见，在这个例子中使用了下面这些发生器当中的一个：

```
//: C06:Generators.h
// Different ways to fill sequences.
#ifndef GENERATORS_H
#define GENERATORS_H
#include <cstring>
#include <set>
#include <cstdlib>

// A generator that can skip over numbers:
class SkipGen {
  int i;
  int skp;
public:
  SkipGen(int start = 0, int skip = 1)
  : i(start), skp(skip) {}
  int operator()() {
    int r = i;
    i += skp;
    return r;
  }
};

// Generate unique random numbers from 0 to mod:
class URandGen {
  std::set<int> used;
  int limit;
public:
  URandGen(int lim) : limit(lim) {}
  int operator()() {
    while(true) {
      int i = int(std::rand()) % limit;
      if(used.find(i) == used.end()) {
        used.insert(i);
        return i;
      }
    }
  }
};

// Produces random characters:
class CharGen {
  static const char* source;
  static const int len;
public:
  char operator()() {
    return source[std::rand() % len];
  }
};
#endif // GENERATORS_H ///:~

//: C06:Generators.cpp {O}
#include "Generators.h"
const char* CharGen::source = "ABCDEFGHIJK"
  "LMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz";
```

344

```
const int CharGen::len = std::strlen(source);
///:~
```

在本章中的后面部分，将在列举的各种各样的例子中使用这些生成函数。**SkipGen**函数对象，返回一个算术序列当前元素的下一个数，它们共同的差值保存在数据成员**skp**中。**URandGen**对象在指定范围内产生一个惟一的随机数。（它使用**set**容器，**set**容器将在下一章中介绍。）**CharGen**对象返回一个随机的字母表中的字符。下面是一个使用**UrandGen**的程序例子：

345

```
//: C06:FunctionObjects.cpp {-bor}
// Illustrates selected predefined function object
// templates from the Standard C++ library.
//{L} Generators
#include <algorithm>
#include <cstdlib>
#include <ctime>
#include <functional>
#include <iostream>
#include <iterator>
#include <vector>
#include "Generators.h"
#include "PrintSequence.h"
using namespace std;

template<typename Contain, typename UnaryFunc>
void testUnary(Contain& source, Contain& dest,
  UnaryFunc f) {
  transform(source.begin(), source.end(), dest.begin(), f);
}

template<typename Contain1, typename Contain2,
  typename BinaryFunc>
void testBinary(Contain1& src1, Contain1& src2,
  Contain2& dest, BinaryFunc f) {
  transform(src1.begin(), src1.end(),
    src2.begin(), dest.begin(), f);
}

// Executes the expression, then stringizes the
// expression into the print statement:
#define T(EXPR) EXPR; print(r.begin(), r.end(), \
  "After " #EXPR);
// For Boolean tests:
#define B(EXPR) EXPR; print(br.begin(), br.end(), \
  "After " #EXPR);

// Boolean random generator:
struct BRand {
  bool operator()() { return rand() % 2 == 0; }
};
```

346

```
int main() {
  const int SZ = 10;
  const int MAX = 50;
  vector<int> x(SZ), y(SZ), r(SZ);
  // An integer random number generator:
  URandGen urg(MAX);
  srand(time(0));  // Randomize
  generate_n(x.begin(), SZ, urg);
  generate_n(y.begin(), SZ, urg);
  // Add one to each to guarantee nonzero divide:
  transform(y.begin(), y.end(), y.begin(),
```

```
  bind2nd(plus<int>(), 1));
// Guarantee one pair of elements is ==:
x[0] = y[0];
print(x.begin(), x.end(), "x");
print(y.begin(), y.end(), "y");
// Operate on each element pair of x & y,
// putting the result into r:
T(testBinary(x, y, r, plus<int>()));
T(testBinary(x, y, r, minus<int>()));
T(testBinary(x, y, r, multiplies<int>()));
T(testBinary(x, y, r, divides<int>()));
T(testBinary(x, y, r, modulus<int>()));
T(testUnary(x, r, negate<int>()));
vector<bool> br(SZ); // For Boolean results
B(testBinary(x, y, br, equal_to<int>()));
B(testBinary(x, y, br, not_equal_to<int>()));
B(testBinary(x, y, br, greater<int>()));
B(testBinary(x, y, br, less<int>()));
B(testBinary(x, y, br, greater_equal<int>()));
B(testBinary(x, y, br, less_equal<int>()));
B(testBinary(x, y, br, not2(greater_equal<int>())));
B(testBinary(x,y,br,not2(less_equal<int>())));
vector<bool> b1(SZ), b2(SZ);
generate_n(b1.begin(), SZ, BRand());
generate_n(b2.begin(), SZ, BRand());
print(b1.begin(), b1.end(), "b1");
print(b2.begin(), b2.end(), "b2");
B(testBinary(b1, b2, br, logical_and<int>()));
B(testBinary(b1, b2, br, logical_or<int>()));
B(testUnary(b1, br, logical_not<int>()));
B(testUnary(b1, br, not1(logical_not<int>())));
} ///:~
```

这个例子使用了一个简单的函数模板**print( )**，它能够打印任意类型的序列并且可以附加可选择的信息。这个模板包含在头文件**PrintSequence.h**中，详细内容将在本章后面介绍。 347

这两个模板函数用来自动处理测试各种函数对象模板的过程。有两个模板函数，是因为函数对象可能是一元的也可能是二元的。**testUnary( )**函数有一个源**vector**、一个目的**vector**和一个用在源**vector**上来产生目的**vector**的一元函数对象。在**testBinary( )**中，将两个源**vector**传送给一个二元函数来产生目的**vector**。在这两种情况下，模板函数仅回转并调用**transform( )**算法，**transform( )**算法将位于其第4个参数位置的一元函数或函数对象应用于序列中的每一元素上，并将结果输出到第3个参数所指示的序列中，在本例中与输入序列相同。

对于每个测试，用户都想看到描述测试的字符串和附加测试结果的字符串。为了自动完成这些工作，可以方便地使用预处理器；宏**T( )**和**B( )**分别含有用户想要执行的表达式。对表达式求值后，它们将相应范围的序列传递给**print( )**。为了产生这一信息，通过预处理器将表达式“字符串化”(stringized)。用这种方法，用户就可以看到相应的表达式代码，这些代码在程序执行后存储在结果**vector**中。

最后一个小工具**BRand**是一个创建随机**bool**型值的发生器对象。为了完成这一工作，它从**rand( )**得到一个随机数并且检查它是否大于 **(RAND_MAX+1)/2**。如果随机数均匀地分布，则值大于 **(RAND_MAX+1)/2**的情况将以50%的概率出现。

在**main( )**中，创建了3个**int**型的**vector**：**x**和**y**是源数值，**r**是结果值。为了用不大于50的随机数初始化**x**和**y**，使用**Generators.h**中的**URandGen**类型的一个发生器来完成这个任务。标准**generate_n( )**算法通过调用第3个参数（必须是一个发生器）、一个给定的次数

（在第2个参数中指定）来建立由第1个参数指定的一个序列。因为有一个**x**被**y**除的操作，所以必须保证**y**的值不为0，以防计算结果溢出。这是靠再次使用**transform( )**算法完成的，它从**y**中获得源值并把结果写回**y**。用下面的表达式创建了一个函数对象：

348
```
bind2nd(plus<int>(), 1)
```

表达式用**plus**函数对象来使第1个参数增1。如同本章前面所做的一样，在这里使用绑定程序适配器来构造一个一元函数，这样仅调用**transform( )**就能将其应用到一个序列上。

程序中的另外一个测试是比较两个**vector**中的元素是否相等，因此值得注意的是要保证至少有一对元素是相等的；这里包含0元素。

一旦打印了这两个**vector**，**T( )**测试产生数字型值的每一个函数对象，**B( )**测试产生布尔型结果的每一个函数对象。在打印**vector**时，将结果放入**vector<bool>**，它对于真值产生**'1'**，对假值产生**'0'**。下面是执行**FunctionObjects.cpp**的输出结果：

```
x:
4 8 18 36 22 6 29 19 25 47
y:
4 14 23 9 11 32 13 15 44 30
After testBinary(x, y, r, plus<int>()):
8 22 41 45 33 38 42 34 69 77
After testBinary(x, y, r, minus<int>()):
0 -6 -5 27 11 -26 16 4 -19 17
After testBinary(x, y, r, multiplies<int>()):
16 112 414 324 242 192 377 285 1100 1410
After testBinary(x, y, r, divides<int>()):
1 0 0 4 2 0 2 1 0 1
After testBinary(x, y, r, limit<int>()):
0 8 18 0 0 6 3 4 25 17
After testUnary(x, r, negate<int>()):
-4 -8 -18 -36 -22 -6 -29 -19 -25 -47
After testBinary(x, y, br, equal_to<int>()):
1 0 0 0 0 0 0 0 0 0
After testBinary(x, y, br, not_equal_to<int>()):
0 1 1 1 1 1 1 1 1 1
After testBinary(x, y, br, greater<int>()):
0 0 0 1 1 0 1 1 0 1
After testBinary(x, y, br, less<int>()):
0 1 1 0 0 1 0 0 1 0
After testBinary(x, y, br, greater_equal<int>()):
1 0 0 1 1 0 1 1 0 1
After testBinary(x, y, br, less_equal<int>()):
```
349
```
1 1 1 0 0 1 0 0 1 0
After testBinary(x, y, br, not2(greater_equal<int>())):
0 1 1 0 0 1 0 0 1 0
After testBinary(x,y,br,not2(less_equal<int>())):
0 0 0 1 1 0 1 1 0 1
b1:
0 1 1 0 0 0 1 0 1 1
b2:
0 1 1 0 0 0 1 0 1 1
After testBinary(b1, b2, br, logical_and<int>()):
0 1 1 0 0 0 1 0 1 1
After testBinary(b1, b2, br, logical_or<int>()):
0 1 1 0 0 0 1 0 1 1
After testUnary(b1, br, logical_not<int>()):
1 0 0 1 1 1 0 1 0 0
After testUnary(b1, br, not1(logical_not<int>())):
0 1 1 0 0 0 1 0 1 1
```

如果在输出结果中想使布尔型值显示为“真”和“假”而不是1和0，则要调用**cout.setf(ios::boolalpha)**。

一个绑定程序无需产生一元判定函数；它能创建任意的一元函数（即返回除**bool**型以外类型值的函数）。例如，可以用10乘以**vector**中的每个元素来使用带有绑定程序的**transform( )**算法：

```
//: C06:FBinder.cpp
// Binders aren't limited to producing predicates.
//{L} Generators
#include <algorithm>
#include <cstdlib>
#include <ctime>
#include <functional>
#include <iostream>
#include <iterator>
#include <vector>
#include "Generators.h"
using namespace std;

int main() {
  ostream_iterator<int> out(cout," ");
  vector<int> v(15);
  srand(time(0));  // Randomize
  generate(v.begin(), v.end(), URandGen(20));
  copy(v.begin(), v.end(), out);
  transform(v.begin(), v.end(), v.begin(),
            bind2nd(multiplies<int>(), 10));
  copy(v.begin(), v.end(), out);
} ///:~
```

350

因为**transform( )**的第3个参数与第1个参数一样，所以结果元素又被复制回源**vector**。本例中**bind2nd( )**创建的函数对象产生一个**int**型结果。

由绑定程序“绑定”的参数不能是一个函数对象，但也无需是一个编译时常量。例如：

```
//: C06:BinderValue.cpp
// The bound argument can vary.
#include <algorithm>
#include <functional>
#include <iostream>
#include <iterator>
#include <cstdlib>
using namespace std;

int boundedRand() { return rand() % 100; }

int main() {
  const int SZ = 20;
  int a[SZ], b[SZ] = {0};
  generate(a, a + SZ, boundedRand);
  int val = boundedRand();
  int* end = remove_copy_if(a, a + SZ, b,
                            bind2nd(greater<int>(), val));
  // Sort for easier viewing:
  sort(a, a + SZ);
  sort(b, end);
  ostream_iterator<int> out(cout, " ");
  cout << "Original Sequence:" << endl;
  copy(a, a + SZ, out); cout << endl;
  cout << "Values <= " << val << endl;
  copy(b, end, out); cout << endl;
} ///:~
```

这里用20个在0到100之间的随机数填充一个数组，当然用户可以在命令行提供一个值，用来限制产生随机数的范围。在**remove_copy_if( )**调用中，可以看到对于**bind2nd( )**的绑定参数是在相同范围内的顺序序列的随机数。这是一次运行的输出结果:

351

```
Original Sequence:
4 12 15 17 19 21 26 30 47 48 56 58 60 63 71 79 82 90 92 95
Values <= 41
4 12 15 17 19 21 26 30
```

6.2.5 函数指针适配器

算法无论在什么地方都要求有一个类似函数的实体，系统可以提供一个指向普通函数或是一个函数对象的指针。当算法通过函数指针调用时，就启用了本地的函数调用机制。如果是通过函数对象调用，则执行对象的**operator( )**成员。在**CopyInts2.cpp**中，把原始的函数**gt15( )**作为一个判定函数传递给**remove_copy_if( )**。同时也把指向返回随机数的函数的指针传递给**generate( )**和**generate_n( )**。

不能通过诸如**bind2nd( )**函数对象适配器来使用原始函数，因为这些函数对象适配器要求具有参数及结果类型的类型定义。不需要采用手工方式将原始的函数转化为函数对象，标准库为用户提供了一系列适配器来完成这一工作。**ptr_fun( )**适配器把指向一个函数的指针转化成为一个函数对象。所有这些并不是为无参数函数设计的——也就是说，它们必须是一元或二元函数。

下面的程序用**ptr_fun( )**来封装一个一元函数。

```
//: C06:PtrFun1.cpp
// Using ptr_fun() with a unary function.
#include <algorithm>
#include <cmath>
#include <functional>
#include <iostream>
#include <iterator>
#include <vector>
using namespace std;

int d[] = { 123, 94, 10, 314, 315 };
const int DSZ = sizeof d / sizeof *d;
bool isEven(int x) { return x % 2 == 0; }

int main() {
  vector<bool> vb;
  transform(d, d + DSZ, back_inserter(vb),
    not1(ptr_fun(isEven)));
  copy(vb.begin(), vb.end(),
    ostream_iterator<bool>(cout, " "));
  cout << endl;
  // Output: 1 0 0 0 1
} ///:~
```

352

不能仅把**isEven**传递给**not1**，因为**not1**需要知道它使用的实际参数的类型和返回值的类型。**ptr_fun( )**适配器可以通过模板参数推断出这些类型。**ptr_fun( )**的一元版本的定义如下所示:

```
template<class Arg, class Result>
pointer_to_unary_function<Arg, Result>
ptr_fun(Result (*fptr)(Arg)) {
  return pointer_to_unary_function<Arg, Result>(fptr);
}
```

正如读者看到的，**ptr_fun( )**的这种版本从**fptr**中推断出参数和结果的类型，并用它们来初始化一个存储**fptr**的**pointer_to_unary_function**对象。如代码的最后一行，对**pointer_to_unary_function**的函数调用操作符仅调用**fptr**:

```
template<class Arg, class Result>
class pointer_to_unary_function
: public unary_function<Arg, Result> {
  Result (*fptr)(Arg); // Stores the f-ptr
public:
  pointer_to_unary_function(Result (*x)(Arg)) : fptr(x) {}
  Result operator()(Arg x) const { return fptr(x); }
};
```

因为**pointer_to_unary_function**派生于**unary_function**，产生合适的类型定义对**not1**是很有用的。

同时也有**ptr_fun( )**的二元版本，它返回一个在执行上与一元情况类似的**pointer_to_binary_function**对象（派生于**binary_function**）。下面的程序使用**ptr_ fun( )** 的二元版本来增加序列中的乘方个数。同时，在向**ptr_fun( )**传递重载函数时也暴露出一个缺陷。 353

```
//: C06:PtrFun2.cpp {-edg}
// Using ptr_fun() for a binary function.
#include <algorithm>
#include <cmath>
#include <functional>
#include <iostream>
#include <iterator>
#include <vector>
using namespace std;

double d[] = { 01.23, 91.370, 56.661,
  023.230, 19.959, 1.0, 3.14159 };
const int DSZ = sizeof d / sizeof *d;

int main() {
  vector<double> vd;
  transform(d, d + DSZ, back_inserter(vd),
    bind2nd(ptr_fun<double, double, double>(pow), 2.0));
  copy(vd.begin(), vd.end(),
    ostream_iterator<double>(cout, " "));
  cout << endl;
} ///:~
```

pow( )函数在标准C++头文件**<cmath>**中对每个浮点数据类型进行重载，如下面程序所示:

```
float pow(float, int);  // Efficient int power versions ...
double pow(double, int);
long double pow(long double, int);
float pow(float, float);
double pow(double, double);
long double pow(long double, long double);
```

因为有多种**pow( )**的版本，编译器不知道选择哪一个。在这里，需要借助前面章节介绍的显式的函数模板特化来帮助编译器。[⊖] 354

⊖ 对于不同的库实现情况有所不同。如果**pow()**使用C 链接，这就意味着读者没有将其理解为C++函数，那么这个例子将不能通过编译。**ptr_fun** 要求是一指向普通重载的C++函数指针。

用通用算法将一个成员函数转化为适于使用的函数对象更是巧妙。例如，假定这里有一个经典的“图形（shape）”问题，并且想对**Shape**容器内的每个指针都应用**draw( )**成员函数：

```
//: C06:MemFun1.cpp
// Applying pointers to member functions.
#include <algorithm>
#include <functional>
#include <iostream>
#include <vector>
#include "../purge.h"
using namespace std;

class Shape {
public:
  virtual void draw() = 0;
  virtual ~Shape() {}
};

class Circle : public Shape {
public:
  virtual void draw() { cout << "Circle::Draw()" << endl; }
  ~Circle() { cout << "Circle::~Circle()" << endl; }
};

class Square : public Shape {
public:
  virtual void draw() { cout << "Square::Draw()" << endl; }
  ~Square() { cout << "Square::~Square()" << endl; }
};

int main() {
  vector<Shape*> vs;
  vs.push_back(new Circle);
  vs.push_back(new Square);
  for_each(vs.begin(), vs.end(), mem_fun(&Shape::draw));
  purge(vs);
} ///:~
```

355

for_each( )算法将序列中每一个元素依次传递给由第3个参数指示的函数对象。在这里，希望函数对象封装成它自身类的一个成员函数，所以对于成员函数调用来说，函数对象“参数”成了对象指针。为了产生这样的函数对象，**mem_fun( )**模板使用了指向成员的一个指针来作为它的参数。

当**mem_fun_ref( )**为一个对象直接调用成员函数时，**mem_fun( )**函数为了生成函数对象，该函数用一个指向被调用的成员函数的对象的指针进行调用。**mem_fun( )**和**mem_fun_ref( )**重载的一个版本设置，就是为有零个和一个参数的成员函数而建立的，并且用乘2来处理**const** 成员函数和非**const**成员函数。然而，模板和重载都完成了全部排序——在这里，读者只需要记住什么时候使用**mem_fun( )**和**mem_fun_ref( )**。

假设有一个对象（不是指针）的容器，现在想调用有一个参数的成员函数。传递的参数应该来自对象的第2个容器。为了完成这个调用，使用**transform( )**算法的第2种重载版本：

```
//: C06:MemFun2.cpp
// Calling member functions through an object reference.
#include <algorithm>
#include <functional>
#include <iostream>
#include <iterator>
#include <vector>
```

```
using namespace std;

class Angle {
  int degrees;
public:
  Angle(int deg) : degrees(deg) {}
  int mul(int times) { return degrees *= times; }
};
int main() {
  vector<Angle> va;
  for(int i = 0; i < 50; i += 10)
    va.push_back(Angle(i));
  int x[] = { 1, 2, 3, 4, 5 };
  transform(va.begin(), va.end(), x,
    ostream_iterator<int>(cout, " "),
    mem_fun_ref(&Angle::mul));
  cout << endl;
  // Output: 0 20 60 120 200
} ///:~
```

356

因为容器持有对象，所以必须通过成员函数指针来使用**mem_fun_ref( )**。这种版本的**transform( )**使用第1个范围（对象生存的范围）的开始和结束点；第2个范围的开始点，就是持有的那个表示成员函数的参数；目的迭代器，在本例中就是标准输出；且为每个对象调用函数对象。用**mem_fun_ref( )**和想要得到的成员指针来创建函数对象。注意，**transform( )**和**for_each( )**模板函数都是不完全的；**transform( )**要求它调用的函数返回一个值，**for_each( )**没有向它调用的成员函数传递所需的两个参数。因此，不能使用**transform( )**调用返回值为**void**的成员函数，也不能调用只有一个参数的**for_each( )**成员函数。

大多数任意类型的成员函数与**mem_fun_ref( )**一起工作。如果用户使用的编译器没有增加任何一个默认参数而超过标准库中指定的正规参数[⊖]，也可以使用标准库成员函数。例如，假设用户希望读一个文件并且查找其中的空白行。编译器可以允许像下面的程序一样使用**string::empty( )**成员函数：

```
//: C06:FindBlanks.cpp
// Demonstrates mem_fun_ref() with string::empty().
#include <algorithm>
#include <cassert>
#include <cstddef>
#include <fstream>
#include <functional>
#include <string>
#include <vector>
#include "../require.h"
using namespace std;

typedef vector<string>::iterator LSI;

int main(int argc, char* argv[]) {
  char* fname = "FindBlanks.cpp";
  if(argc > 1) fname = argv[1];
  ifstream in(fname);
  assure(in, fname);
  vector<string> vs;
```

357

⊖ 如果编译器能够使用默认参数（这些参数是合法的）来定义**string::empty**，那么表达式**&string::empty**就能定义一指向成员函数的指针，这个成员函数包含所有参数。因为无法让编译器提供额外参数，在将算法通过**mem_fun_ref**应用于**string::empty**时将出现“缺少参数”错误。

```
  string s;
  while(getline(in, s))
    vs.push_back(s);
  vector<string> cpy = vs; // For testing
  LSI lsi = find_if(vs.begin(), vs.end(),
     mem_fun_ref(&string::empty));
  while(lsi != vs.end()) {
    *lsi = "A BLANK LINE";
    lsi = find_if(vs.begin(), vs.end(),
      mem_fun_ref(&string::empty));
  }
  for(size_t i = 0; i < cpy.size(); i++)
    if(cpy[i].size() == 0)
      assert(vs[i] == "A BLANK LINE");
    else
      assert(vs[i] != "A BLANK LINE");
} ///:~
```

这个例子使用**find_if( )**、**mem_fun_ref( )**和**string::empty( )**一起，在指定范围的序列中查找第1个空白行的位置。打开文件并将其读入到**vector**对象后，重复这个处理，在文件中查找每一个空白行。每次找到一个空白行时，就用字符串“A BLANK LINE”来取代该空白行。所有这些工作，是在不引用迭代器来选择当前字符串的情况下完成的。

358

6.2.6 编写自己的函数对象适配器

考虑如何编写一个把表示浮点数的字符串转化为相应实际数字值的程序。作为对该问题进行编程的开始，这里有个创建字符串的发生器：

```
//: C06:NumStringGen.h
// A random number generator that produces
// strings representing floating-point numbers.
#ifndef NUMSTRINGGEN_H
#define NUMSTRINGGEN_H
#include <cstdlib>
#include <string>

class NumStringGen {
  const int sz; // Number of digits to make
public:
  NumStringGen(int ssz = 5) : sz(ssz) {}
  std::string operator()() {
    std::string digits("0123456789");
    const int ndigits = digits.size();
    std::string r(sz, ' ');
    // Don't want a zero as the first digit
    r[0] = digits[std::rand() % (ndigits - 1)] + 1;
    // Now assign the rest
    for(int i = 1; i < sz; ++i)
      if(sz >= 3 && i == sz/2)
        r[i] = '.'; // Insert a decimal point
      else
        r[i] = digits[std::rand() % ndigits];
    return r;
  }
};
#endif // NUMSTRINGGEN_H ///:~
```

当创建**NumStringGen**对象时，要告诉它字符串应该有多大。字符串由随机数发生器挑选出来的数字组成，并在中间插入一个小数点。

下面的程序使用**NumStringGen**来填写一个**vector<string>**。但是，要使用标准C库

函数**atof( )**来把字符串转化为浮点型数，**string**类型对象必须首先转化为**char**类型指针，这是因为从**string**到**char***没有自动类型转换。可以使用拥有**mem_fun_ref( )**和**string::c_str( )**的**transform( )**算法，先把所有的**string**都转化为**char***，然后再使用**atof**进行转换。 359

```
//: C06:MemFun3.cpp
// Using mem_fun().
#include <algorithm>
#include <cstdlib>
#include <ctime>
#include <functional>
#include <iostream>
#include <iterator>
#include <string>
#include <vector>
#include "NumStringGen.h"
using namespace std;

int main() {
  const int SZ = 9;
  vector<string> vs(SZ);
  // Fill it with random number strings:
  srand(time(0)); // Randomize
  generate(vs.begin(), vs.end(), NumStringGen());
  copy(vs.begin(), vs.end(),
    ostream_iterator<string>(cout, "\t"));
  cout << endl;
  const char* vcp[SZ];
  transform(vs.begin(), vs.end(), vcp,
    mem_fun_ref(&string::c_str));
  vector<double> vd;
  transform(vcp, vcp + SZ, back_inserter(vd),
    std::atof);
  cout.precision(4);
  cout.setf(ios::showpoint);
  copy(vd.begin(), vd.end(),
    ostream_iterator<double>(cout, "\t"));
  cout << endl;
} ///:~
```

这个程序做了两个转换：一是将C++字符串转换成C风格的字符串（字符数组），另一个是通过**atof( )**将C风格的字符串转换成数值。把这两个运算组合成一个将会更好。毕竟，在数学上能组合函数，那么在C++中为什么不能呢？

简明的方法就是用两个函数作为参数并按合适的顺序应用它们： 360

```
//: C06:ComposeTry.cpp
// A first attempt at implementing function composition.
#include <cassert>
#include <cstdlib>
#include <functional>
#include <iostream>
#include <string>
using namespace std;

template<typename R, typename E, typename F1, typename F2>
class unary_composer {
  F1 f1;
  F2 f2;
public:
  unary_composer(F1 fone, F2 ftwo) : f1(fone), f2(ftwo) {}
```

```
  R operator()(E x) { return f1(f2(x)); }
};

template<typename R, typename E, typename F1, typename F2>
unary_composer<R, E, F1, F2> compose(F1 f1, F2 f2) {
  return unary_composer<R, E, F1, F2>(f1, f2);
}

int main() {
  double x = compose<double, const string&>(
    atof, mem_fun_ref(&string::c_str))("12.34");
  assert(x == 12.34);
} ///:~
```

本例中的**unary_composer**对象存储函数指针**atof**和**string::c_str**，这样一来，在调用该对象的**operator( )**时首先要应用后面的函数。**compose( )**函数适配器是个便利的设置，因此用户无需明确提供全部的四个模板参数——**F1**和**F2**从调用中推断出来。

如果无需提供任何模板参数就更好了。这是靠对合适的函数对象坚持类型定义转换来完成的。换句话说，就是假定这些函数的组合是合适的。这就要求为**atof( )**使用**ptr_fun( )**。为了使其具有最大的灵活性，一旦把**unary_composer**传递到一个函数适配器，也能够使其适用。下面的程序这样做了并且很容易地解决了原来的问题：

361

```
//: C06:ComposeFinal.cpp {-edg}
// An adaptable composer.
#include <algorithm>
#include <cassert>
#include <cstdlib>
#include <functional>
#include <iostream>
#include <iterator>
#include <string>
#include <vector>
#include "NumStringGen.h"
using namespace std;

template<typename F1, typename F2> class unary_composer
: public unary_function<typename F2::argument_type,
                        typename F1::result_type> {
  F1 f1;
  F2 f2;
public:
  unary_composer(F1 f1, F2 f2) : f1(f1), f2(f2) {}
  typename F1::result_type
  operator()(typename F2::argument_type x) {
    return f1(f2(x));
  }
};

template<typename F1, typename F2>
unary_composer<F1, F2> compose(F1 f1, F2 f2) {
  return unary_composer<F1, F2>(f1, f2);
}

int main() {
  const int SZ = 9;
  vector<string> vs(SZ);
  // Fill it with random number strings:
  generate(vs.begin(), vs.end(), NumStringGen());
  copy(vs.begin(), vs.end(),
    ostream_iterator<string>(cout, "\t"));
```

```
  cout << endl;
  vector<double> vd;
  transform(vs.begin(), vs.end(), back_inserter(vd),
    compose(ptr_fun(atof), mem_fun_ref(&string::c_str)));
  copy(vd.begin(), vd.end(),
    ostream_iterator<double>(cout, "\t"));
  cout << endl;
} ///:~
```

362

本例中，必须再次使用**typename**来使编译器知道涉及的成员是一个嵌套类型。

一些实现[⊖]将函数对象的组合作为一个扩展来支持，C++标准委员会可能在标准C++的下一版本中增加这些功能。

6.3 STL算法目录

这一部分为读者查找适当的算法提供快捷参考。而把所有的STL算法的完整探究以及问题更深层的细节如性能等一起作为其他参考材料（见本章结尾及附录A）。本节的目标是使读者能够快速熟悉这些算法，并且假定如果需要更多的细节将查找到更多特别指明的参考资料。

尽管读者经常见到用完整的模板声明语法来描述算法，但我们在这里不这样做，因为已经知道它们是模板，并且很容易知道函数声明中的模板参数是什么。参数的类型名为需要的迭代器类型提供描述。读者会发现，这种形式更容易读懂，如果需要，可以很快地在模板头文件中找到所有的声明。

所有涉及迭代器而令人心烦的原因，都是为了（使算法）适用于符合标准库要求的任意类型的容器。到目前为止，本章仅用数组和**vector**作为序列阐述了通用算法，但是在下一章中，读者将会看到一个范围更广的数据结构，这些数据结构支持只用较少力气进行迭代的工作。由于这个原因，将对算法进行部分地分类，这种分类按它们所需要的迭代类型很容易完成。

363

迭代器的类名描述必须与该迭代器的类型相符合。这些迭代器没有用接口基类来强化这些迭代运算——仅是期望它们在这里出现而已。如有没有接口基类，接收这样程序的编译器可能就会抱怨。下面简要地描述迭代器的各种形式：

InputIterator。一个只允许单个向序列读入元素的输入迭代器，前向传递使用**operator++**和**operator***。也可以通过**operator==**和**operator!=**检测输入迭代器。这是约束的范围。

OutputIterator。一个只允许单个向序列写入元素的输出迭代器，前向传递使用**operator++**和**operator***。但是，这类**OutputIterator**不能用**operator==**和**operator!=**来进行测试，因为假定仅持续不断地向目的文件发送元素，而无需判定是否到达了目的文件的结束标志。也就是说，**OutputIterator**涉及的容器可以持有无限个数的对象，而不需要结尾检查。这一点非常重要，因此**OutputIterator**可以与**ostream**（通过**ostream_iterator**）一起使用，同时也普遍使用“插入”迭代器（它是**back_inserter( )**返回的迭代器类型）。

在相同范围的序列内，没有方法同时确定多个**InputIterators**或**OutputIterators**点，因此也就没有办法一起使用这样的迭代器。仅用迭代器来支持**istream**和**ostream**，使用**InputIterator**和**OutputIterator**就会产生理想的效果。同时也要注意，使用**InputIterators**或**OutputIterators**的算法对可接受的迭代器类型做最弱的限制，这意味着

⊖ 比如Borland C++第6版和Digital Mars编译器都提供的STLPort，而且STLPort基于SGI STL。

当遇到**InputIterator**或**OutputIterator**作为STL算法模板参数时，可以使用任意“更复杂”的迭代器类型。

ForwardIterator。因为仅仅可以从**InputIterator**中读和向**OutputIterator**中写，不能用它们中的任何一个来同时读和修改某个范围的数据元素，对这样的迭代器只能解析一次。使用**ForwardIterator**，这些限制就放松了；仍然仅用**operator++**前向移动，但是可以同时进行读和写，并且可以在相同的范围内比较这些迭代器是否相等。因为前向迭代器可以同时读和写，可以用它来取代**InputIterator**或**OutputIterator**。

364 **BidirectionalIterator**。实际上，这是一个也可以进行后向移动的**ForwardIterator**。也就是说，**BidirectionalIterator**支持**ForwardIterator**所做的全部操作，而另外还增加了**operator--**运算。

RandomAccessIterator。这种迭代器类型支持一个常规指针所做的全部运算：可以通过增加和减少某个整数值，来向前和向后跳跃移动（不是每次只移动一个元素），还可以用**operator[ ]**作为下标索引，可以从一个迭代器中减去另一个迭代器，也可以用**operator<**，**operator>** 来比较迭代器看哪个更大等等。如果要实现一个排序程序或其他类似的工作，随机存取迭代器是创建一个有效率的算法所必需的。

本章后面的算法描述中，使用的模板参数类型名由列出的迭代器类型（有时附加‘1’或‘2’来区分不同的模板参数）组成，同时也包括其他的参数，通常是函数对象。

当描述传递给运算的元素组时，经常使用数学上“范围”记号。即方括号表示“包括边界点”，圆括号表示“不包括边界点”。当使用迭代器时，要靠指向开始元素的迭代器和指向超越最后一个元素的“超越末尾的”迭代器来决定一个范围。由于根本就没有使用超越末尾的元素，决定这样一对迭代器的范围可以表示成**[first,last)**，这里**first**是指向开始元素的迭代器，**last**是超越末尾的迭代器。

大多数教材和对STL算法的讨论都根据它们副作用的大小来组织算法的先后顺序：非变异（non-mutating）算法在作用域范围内不对元素进行改变，变异（mutating）算法改变元素，等等。这些描述基于主要的基础行为或算法的实现——也就是基于设计者的观点。在实际使用中，用户会发现这种分类没用，因此应该根据要解决的问题来组织算法：当你查找某个元素或元素集合时，是不是对每个元素都执行一个运算、计算元素个数并且更新元素等等？这应该有助于更容易发现要求的算法。

365 如果在函数的声明前面没看到一个如**<utility>**或**<numeric>**的头文件，那么它就应该出现在 **<algorithm>** 中。同样地，所有的算法都在名字空间**std**中。

6.3.1 实例创建的支持工具

创建一些基本的工具来测试算法是很有用的。在这些例子中，将使用前面在**Generators.h**中涉及的发生器以及下面出现的这些内容。

显示一个序列是经常要做的工作，这里有一个函数模板用来打印任意一个序列，它不考虑序列中包含的数据类型：

```
//: C06:PrintSequence.h
// Prints the contents of any sequence.
#ifndef PRINTSEQUENCE_H
#define PRINTSEQUENCE_H
#include <algorithm>
#include <iostream>
#include <iterator>
```

```
template<typename Iter>
void print(Iter first, Iter last, const char* nm = "",
           const char* sep = "\n",
           std::ostream& os = std::cout) {
  if(nm != 0 && *nm != '\0')
    os << nm << ": " << sep;
  typedef typename
    std::iterator_traits<Iter>::value_type T;
  std::copy(first, last,
            std::ostream_iterator<T>(std::cout, sep));
  os << std::endl;
}
#endif // PRINTSEQUENCE_H ///:~
```

在默认的情况下，以一个换行符（“**n**”）作为分隔符，这个函数模板向**cout**输出，但可以通过修改默认参数来改变它。同时也可以在输出的开头打印一个信息。因为**print( )**使用**copy( )**算法经由**ostream_iterator**向**cout**发送对象，**ostream_iterator**必须知道它正在打印的对象的类型，该对象的类型由传递过来的迭代器的**value_type**成员推断而来。

std::iterator_traits模板能够使**print( )**函数模板处理由任意迭代器类型限定的序列。由标准容器如**vector**返回的迭代器类型定义了一个嵌套类型**value_type**，它代表元素的类型。但是当使用数组时，迭代器仅仅只是指针类型，因而不能有嵌套类型。为了支持标准库中与迭代器有关联的使用便利的类型，**std::iterator_traits**为指针类型提供了下面的半特化： 366

```
template<class T>
  struct iterator_traits<T*> {
    typedef random_access_iterator_tag iterator_category;
    typedef T value_type;
    typedef ptrdiff_t difference_type;
    typedef T* pointer;
    typedef T& reference;
  };
```

这样就使该模板可获得经由类型名**value_type**指明的元素类型（即**T**）。

稳定排序和不稳定排序

对于很多经常移动序列中元素的STL算法而言，有序列的*稳定再排序*和*不稳定再排序*之分。就比较函数而言，一个稳定的排序保持相等元素的原始相对顺序。例如，考虑序列**{c(1), b(1), c(2), a(1), b(2), a(2)}**。在算法中是根据字母来检查这些元素的相等性，但是它们的数字显示怎样在序列中出现（谁在前，谁在后？）。如果排序（例如），对这个序列使用不稳定的排序，就不能保证相同字母间的特定顺序，所以可能以**{a(2), a(1), b(1), b(2), c(2), c(1)}**结束。然而，如果使用稳定的排序，就会得到**{a(1), a(2), b(1), b(2), c(1), c(2)}**。STL的**sort( )**算法使用的是*快速排序*的一个变种，因此是不稳定的，但是STL也提供稳定的排序算法**stable_sort( )**。[⊖]

为了证明对一个序列进行重新排序的算法是稳定性算法还是不稳定性算法，我们需要一些方法来保持对元素原始位置的跟踪。下面是一种保持跟踪特殊对象原始出现顺序的**string**对象，它用**static map**对从**NString**到**Counters**进行映射。这样每个**NString**包含一个**occurrence**字段，用来表示在**NString**中发现的顺序。 367

⊖ **stable_sort()**使用*归并排序*，归并排序是稳定排序，但是在平均情况下比快速排序慢。

```
//: C06:NString.h
// A "numbered string" that keeps track of the
// number of occurrences of the word it contains.
#ifndef NSTRING_H
#define NSTRING_H
#include <algorithm>
#include <iostream>
#include <string>
#include <utility>
#include <vector>

typedef std::pair<std::string, int> psi;

// Only compare on the first element
bool operator==(const psi& l, const psi& r) {
  return l.first == r.first;
}

class NString {
  std::string s;
  int thisOccurrence;
  // Keep track of the number of occurrences:
  typedef std::vector<psi> vp;
  typedef vp::iterator vpit;
  static vp words;
  void addString(const std::string& x) {
    psi p(x, 0);
    vpit it = std::find(words.begin(), words.end(), p);
    if(it != words.end())
      thisOccurrence = ++it->second;
    else {
      thisOccurrence = 0;
      words.push_back(p);
    }
  }
public:
  NString() : thisOccurrence(0) {}
  NString(const std::string& x) : s(x) { addString(x); }
  NString(const char* x) : s(x) { addString(x); }
  // Implicit operator= and copy-constructor are OK here.
  friend std::ostream& operator<<(
    std::ostream& os, const NString& ns) {
    return os << ns.s << " [" << ns.thisOccurrence << "]";
  }
  // Need this for sorting. Notice it only
  // compares strings, not occurrences:
  friend bool
  operator<(const NString& l, const NString& r) {
    return l.s < r.s;
  }
  friend
  bool operator==(const NString& l, const NString& r) {
    return l.s == r.s;
  }
  // For sorting with greater<NString>:
  friend bool
  operator>(const NString& l, const NString& r) {
    return l.s > r.s;
  }
  // To get at the string directly:
  operator const std::string&() const { return s; }
};
```

368

```
// Because NString::vp is a template and we are using the
// inclusion model, it must be defined in this header file:
NString::vp NString::words;
#endif // NSTRING_H ///:~
```

通常使用**map**容器来将一个与字符串一起出现的数字关联起来，但直到第7章才会讨论映射的问题，因此在这里用成对的**vector**来代替映射。在第7章读者将会看到大量类似的例子。

执行有秩序的升序排序必需的运算符只有**NString::operator<( )**。同时还提供降序的排序操作符**operator>( )**，这样**greater**模板就能调用它了。

6.3.2 填充和生成

这些算法能够自动用一个特定值来填充（容器中的）某个范围的数据，或为（容器中的）某个特定范围生成一组值。“填充（fill）”函数向容器中多次插入一个值。“生成（generate）”函数使用如前面提到过的发生器来产生插入到容器中的值。 369

```
void fill(ForwardIterator first, ForwardIterator last,
  const T& value);
void fill_n(OutputIterator first, Size n, const T& value);
```

fill( )对 **[first,last)** 范围内的每个元素赋值**value**。**fill_n( )**对由**first**开始的**n**个元素赋值**value**。

```
void generate(ForwardIterator first, ForwardIterator last,
  Generator gen);
void generate_n(OutputIterator first, Size n, Generator
  gen);
```

generate( )为 **[first,last)** 范围内的每个元素进行一个**gen( )**调用，可以假定为每个元素产生一个不同的值。**generate_n( )** 对**gen( )n**调用**n**次，并且将返回值赋给由**first**开始的**n**个元素。

程序举例

下面的例子对**vector**进行填充和生成。同时也显示了**print( )**的使用:

```
//: C06:FillGenerateTest.cpp
// Demonstrates "fill" and "generate."
//{L} Generators
#include <vector>
#include <algorithm>
#include <string>
#include "Generators.h"
#include "PrintSequence.h"
using namespace std;

int main() {
  vector<string> v1(5);
  fill(v1.begin(), v1.end(), "howdy");
  print(v1.begin(), v1.end(), "v1", " ");
  vector<string> v2;
  fill_n(back_inserter(v2), 7, "bye");
  print(v2.begin(), v2.end(), "v2");
  vector<int> v3(10);
  generate(v3.begin(), v3.end(), SkipGen(4,5));
  print(v3.begin(), v3.end(), "v3", " ");
  vector<int> v4;
  generate_n(back_inserter(v4),15, URandGen(30));
  print(v4.begin(), v4.end(), "v4", " ");
} ///:~
```

370

vector<string> 用预定义的大小来创建。因为已经为**vector**中所有**string**对象创建了存储空间，**fill( )**可以用它的赋值操作对**vector**中的每个空间赋“howdy”的一个拷贝。同时，用空格来取代默认的换行符分隔符。

没有给定第2个**vector<string> v2**的初始大小，因此必须使用**back_inserter( )**来添加新元素，而不是试图对现有位置赋值。

除了用一个发生器来代替常量值以外，**generate( )**和**generate_n( )**函数与“填充”函数有相同的形式。在这里，这两个发生器都演示了它们的功能。

6.3.3 计数

所有的容器都含有一个成员函数**size( )**，它可以告之该容器包含有多少个元素。**size( )**的返回类型是迭代器的**difference_type**[⊖]（通常是**ptrdiff_t**），下面我们用**IntegralValue**表示。下面的两个算法可以满足一定标准的对象计数。

```
IntegralValue count(InputIterator first, InputIterator
  last, const EqualityComparable& value);
```

在这个算法中，产生 **[first,last)** 范围内其值等于**value**（当用**operator==**测试时）的元素的个数。

```
IntegralValue count_if(InputIterator first, InputIterator
  last, Predicate pred);
```

371

这个算法产生 **[first,last)** 范围内能使**pred**返回**true**的元素的个数。

程序举例

这里，用随机字符（包括一些重复的字符）填充**vector<char> v**。**set<char>**由**v**来初始化，因此它仅持有**v**中代表的各个字母中的一个。这个**set**对显示的所有字符的实例进行计数：

```
//: C06:Counting.cpp
// The counting algorithms.
//{L} Generators
#include <algorithm>
#include <functional>
#include <iterator>
#include <set>
#include <vector>
#include "Generators.h"
#include "PrintSequence.h"
using namespace std;

int main() {
  vector<char> v;
  generate_n(back_inserter(v), 50, CharGen());
  print(v.begin(), v.end(), "v", "");
  // Create a set of the characters in v:
  set<char> cs(v.begin(), v.end());
  typedef set<char>::iterator sci;
  for(sci it = cs.begin(); it != cs.end(); it++) {
    int n = count(v.begin(), v.end(), *it);
    cout << *it << ": " << n << ", ";
  }
  int lc = count_if(v.begin(), v.end(),
    bind2nd(greater<char>(), 'a'));
  cout << "\nLowercase letters: " << lc << endl;
```

⊖ 在第7章中我们将详细讨论迭代器。

```
  sort(v.begin(), v.end());
  print(v.begin(), v.end(), "sorted", "");
} ///:~
```

count_if( )算法通过对所有的小写字母计数来进行演示；用**bind2nd( )**和**greater**函数对象模板创建判定函数。

6.3.4 操作序列

372

这些都是有关移动序列的算法。

```
OutputIterator copy(InputIterator first, InputIterator
  last, OutputIterator destination);
```

使用赋值，从范围 **[first,last)** 复制序列到**destination**，每次赋值后都增加**destination**。这本质上是一个“左混洗（shuffle-left)”运算，所以源序列不能包含目的序列。由于使用了赋值操作，因此不能直接向空容器或容器末尾插入元素，而必须把**destination**迭代器封装在**insert_iterator**里（在与容器发生联系的情况下，典型地使用**back_inserter( )**或**inserter( )**）。

```
BidirectionalIterator2 copy_backward(BidirectionalIterator1
  first, BidirectionalIterator1 last,
  BidirectionalIterator2 destinationEnd);
```

这个算法如同**copy( )**一样，但是以相反的顺序复制元素。这本质上是“右混洗（shuffle-right)”运算，而且如同**copy( )** 一样，源序列不能包含目的序列。将源范围 **[first, last)** 序列复制到目的序列，但第1个目的元素是**destinationEnd - 1**。这个迭代器在每次赋值后减少。目的序列范围的空间必须已经存在（允许赋值），而且目的序列范围不能在源序列范围之内。

```
void reverse(BidirectionalIterator first,
  BidirectionalIterator last);
OutputIterator reverse_copy(BidirectionalIterator first,
  BidirectionalIterator last, OutputIterator destination);
```

这个函数的两种形式都倒置了范围**[first,last)**。**reverse( )**倒置原序列范围的元素，**reverse_copy( )**保持原序列范围元素顺序不变，而将倒置的元素复制到**destination**，返回结果序列范围的超越末尾(past-the-end)的迭代器。

```
ForwardIterator2 swap_ranges(ForwardIterator1 first1,
  ForwardIterator1 last1, ForwardIterator2 first2);
```

通过交换对应的元素来交换相等大小两个范围的内容。

373

```
void rotate(ForwardIterator first, ForwardIterator middle,
  ForwardIterator last);
OutputIterator rotate_copy(ForwardIterator first,
  ForwardIterator middle, ForwardIterator last,
  OutputIterator destination);
```

该算法把**[first, middle)** 范围中的内容移到该序列的末尾，并且将**[middle,last)** 范围中的内容移到该序列的开始位置。使用**rotate( )** 在适当的位置执行交换；使用**rotate_copy( )**不改变原始序列范围，且将轮换后（rotated）版本的元素复制到**destination**，返回结果范围的超越末尾的迭代器。注意，需要使用**swap_ranges( )**时，两个范围的大小是完全相等的，但“轮换”函数不是这样。

```
bool next_permutation(BidirectionalIterator first,
  BidirectionalIterator last);
bool next_permutation(BidirectionalIterator first,
  BidirectionalIterator last, StrictWeakOrdering
  binary_pred);
bool prev_permutation(BidirectionalIterator first,
  BidirectionalIterator last);
bool prev_permutation(BidirectionalIterator first,
  BidirectionalIterator last, StrictWeakOrdering
  binary_pred);
```

在这些算法中，排列(permutation)是一组元素的一种独一无二的排序。如果有**n**个元素，就会有**n**！（**n**的阶乘）种不同的元素的组合。所有的这些组合都可以概念化地以词典编纂（像字典一样）的顺序对序列进行排序，这样就产生了一种“后继(next)”和“前驱(previous)”排列的概念。因此无论范围内当前元素的顺序是什么样，在排列的序列中都有一个不同的“后继”和“前驱”的排列。

next_permutation( )和**prev_permutation( )**函数对元素重新排列成后继的或前驱的排列，如果成功则返回**true**。如果没有多个“后继”排列，元素以升序排序，**next_permutation( )**返回**false**。如果没有多个“前驱”排列，元素以降序排序，**previous_permutation( )**返回**false**。

374

具有**StrictWeakOrdering**参数的函数形式用**binary_pred**来执行比较，而不是**operator<**。

```
void random_shuffle(RandomAccessIterator first,
  RandomAccessIterator last);
void random_shuffle(RandomAccessIterator first,
  RandomAccessIterator last RandomNumberGenerator& rand);
```

这个函数随机地重排范围内的元素。如果用随机数发生器，它会产生均匀的分布结果。第1种形式使用内部随机数发生器，第2种使用用户提供的随机数发生器。对于正数**n**发生器必须返回一个在**[0,n)**范围内的值。

```
BidirectionalIterator partition(BidirectionalIterator
  first, BidirectionalIterator last, Predicate pred);
BidirectionalIterator
stable_partition(BidirectionalIterator first,
  BidirectionalIterator last, Predicate pred);
```

在这些算法中，“划分”函数将满足**pred**的元素移到序列的开始位置。迭代器指向其返回元素位置，该元素是超越这些元素中的最后一个（对于以满足**pred**的元素为开始的子序列，“末尾”迭代器有效）。这个位置通常称为“划分点(partition point)”。

使用**partition( )**，在函数调用后，每个结果子序列的元素顺序并没有被指定，但是用**stable_partition( )**，划分点前后这些元素的相对顺序与划分处理前相同。

程序举例

这里给出了序列运算的演示:

```
//: C06:Manipulations.cpp
// Shows basic manipulations.
//{L} Generators
// NString
#include <vector>
```

375

```
#include <string>
#include <algorithm>
#include "PrintSequence.h"
#include "NString.h"
#include "Generators.h"
using namespace std;

int main() {
  vector<int> v1(10);
  // Simple counting:
  generate(v1.begin(), v1.end(), SkipGen());
  print(v1.begin(), v1.end(), "v1", " ");
  vector<int> v2(v1.size());
  copy_backward(v1.begin(), v1.end(), v2.end());
  print(v2.begin(), v2.end(), "copy_backward", " ");
  reverse_copy(v1.begin(), v1.end(), v2.begin());
  print(v2.begin(), v2.end(), "reverse_copy", " ");
  reverse(v1.begin(), v1.end());
  print(v1.begin(), v1.end(), "reverse", " ");
  int half = v1.size() / 2;
  // Ranges must be exactly the same size:
  swap_ranges(v1.begin(), v1.begin() + half,
    v1.begin() + half);
  print(v1.begin(), v1.end(), "swap_ranges", " ");
  // Start with a fresh sequence:
  generate(v1.begin(), v1.end(), SkipGen());
  print(v1.begin(), v1.end(), "v1", " ");
  int third = v1.size() / 3;
  for(int i = 0; i < 10; i++) {
    rotate(v1.begin(), v1.begin() + third, v1.end());
    print(v1.begin(), v1.end(), "rotate", " ");
  }
  cout << "Second rotate example:" << endl;
  char c[] = "aabbccddeeffgghhiijj";
  const char CSZ = strlen(c);
  for(int i = 0; i < 10; i++) {
    rotate(c, c + 2, c + CSZ);
    print(c, c + CSZ, "", "");
  }
  cout << "All n! permutations of abcd:" << endl;
  int nf = 4 * 3 * 2 * 1;
  char p[] = "abcd";
  for(int i = 0; i < nf; i++) {
    next_permutation(p, p + 4);
    print(p, p + 4, "", "");
  }
  cout << "Using prev_permutation:" << endl;
  for(int i = 0; i < nf; i++) {
    prev_permutation(p, p + 4);
    print(p, p + 4, "", "");
  }
  cout << "random_shuffling a word:" << endl;
  string s("hello");
  cout << s << endl;
  for(int i = 0; i < 5; i++) {
    random_shuffle(s.begin(), s.end());
    cout << s << endl;
  }
  NString sa[] = { "a", "b", "c", "d", "a", "b",
    "c", "d", "a", "b", "c", "d", "a", "b", "c"};
  const int SASZ = sizeof sa / sizeof *sa;
  vector<NString> ns(sa, sa + SASZ);
  print(ns.begin(), ns.end(), "ns", " ");
```

376

```
  vector<NString>::iterator it =
    partition(ns.begin(), ns.end(),
      bind2nd(greater<NString>(), "b"));
  cout << "Partition point: " << *it << endl;
  print(ns.begin(), ns.end(), "", " ");
  // Reload vector:
  copy(sa, sa + SASZ, ns.begin());
  it = stable_partition(ns.begin(), ns.end(),
    bind2nd(greater<NString>(), "b"));
  cout << "Stable partition" << endl;
  cout << "Partition point: " << *it << endl;
  print(ns.begin(), ns.end(), "", " ");
} ///:~
```

观察这个程序结果的最好方法是运行该程序。(也可以将结果重新输出到一个文件中。)

vector<int> v1初始化为一个简单的升序序列，并且打印这个序列。读者将会看到**copy_backward( )**的效果（复制到**v2**，**v2**与**v1**相同大小）与普通的复制相同。再一次强调，**copy_backward( )**与**copy( )**做相同的工作——只是以相反的顺序操作。

reverse_copy( )实际上创建一个相反顺序的复制，**reverse( )**在适当的位置执行颠倒操作。接下来，**swap_ranges( )**将颠倒序列的上半部分和下半部分进行交换。范围可以比整个**vector**的子集小，只要它们大小相等就可以。

377

rotate( )是重新创建一个升序序列的演示，它通过多次交换**v1**的三分之一来完成排序工作。第2个**rotate( )**例子使用了字符且每次仅交换两个字符。通过这个例子，也展示了STL算法和**print( )**模板的灵活性，因为与使用其他任意类型一样，可以很容易地使用**char**数组。

为了演示**next_permutation( )**和**prev_permutation( )**，用全部**n!**（**n**的阶乘）种组合来排列“abcd”四个字母的集合。从输出结果中可以看到，排列遵循严格的定义顺序（即排列是确定性的处理）。

random_shuffle( )的一个快速演示是将其应用到一个**string**，并且看结果是什么。因为**string**对象含有可以返回合适迭代器的**begin( )**和**end( )**成员函数，很多STL算法都可以很容易地使用它。同时这里也使用了**char**型数组。

最后，用**NString**数组演示了**partition( )**和**stable_partition( )**。读者将会注意到，总计的初始化表达式使用的是**char**型数组，但**NString**含有一个**char***的构造函数，它能自动调用。

从输出结果中可以看到使用不稳定的划分，对象能正确地“被划分”在划分点之上和之下，但不是以特定的顺序进行；反之用稳定的划分则保持原始的顺序。

6.3.5 查找和替换

所有这些算法都用来在某个范围内查找一个或多个对象，该范围由开始的两个迭代器参数定义。

```
InputIterator find(InputIterator first, InputIterator last,
  const EqualityComparable& value);
```

这个算法在某个范围内的序列元素中查找**value**。返回一个迭代器，该迭代器指向在范围**[first, last)** 内**value**第1次出现的位置。如果**value**不在范围内，**find( )**返回**last**。这是线性查找（linear search）；也就是说，从范围的起始点开始，对每个连续的元素依次进行检查，而不对元素的顺序路径做任何假设。相反，**binary_search( )**（在后面定义）是在一个已经有序的序列上工作，因此能够更快地进行查找。

378

```
InputIterator find_if(InputIterator first, InputIterator
  last, Predicate pred);
```

这个算法如同**find( )**一样，**find_if( )**在指定的序列范围内执行线性查找。然而，代替查找**value**，**find_if( )**寻找一个满足**pred**的元素，当查找到这样的元素时**Predicate pred**返回**true**。如果不能找到这样的元素则返回**last**。

```
ForwardIterator adjacent_find(ForwardIterator first,
  ForwardIterator last);
ForwardIterator adjacent_find(ForwardIterator first,
  ForwardIterator last, BinaryPredicate binary_pred);
```

如同**find( )**一样，这些算法在指定的序列范围内执行线性查找。但不是仅查找一个元素，而是查找两个邻近的相等元素。函数的第1种形式查找两个相等的元素（通过**operator==**）。第2种形式查找两个邻近的元素，当找到这两个元素并一起传递给**binary_pred**时，产生**true**结果。如果找到这样的一对元素，则返回指向两个元素中第1个元素的迭代器；否则返回**last**。

```
ForwardIterator1 find_first_of(ForwardIterator1 first1,
  ForwardIterator1 last1, ForwardIterator2 first2,
  ForwardIterator2 last2);
ForwardIterator1 find_first_of(ForwardIterator1 first1,
  ForwardIterator1 last1, ForwardIterator2 first2,
  ForwardIterator2 last2, BinaryPredicate binary_pred);
```

如同**find( )**一样，上面这两个算法也在指定的序列范围内执行线性查找。这两种形式都是在第2个范围内查找与第1个范围内的某个元素相等的元素。第1种形式使用**operator==**，第2种形式使用提供的判定函数。在第2种形式中，第1个范围序列的当前元素成为**binary_pred**的第1个参数，第2个范围序列内的元素成为**binary_pred**的第2个参数。

379

```
ForwardIterator1 search(ForwardIterator1 first1,
  ForwardIterator1 last1, ForwardIterator2 first2,
  ForwardIterator2 last2);
ForwardIterator1 search(ForwardIterator1 first1,
  ForwardIterator1 last1, ForwardIterator2 first2,
  ForwardIterator2 last2 BinaryPredicate binary_pred);
```

这些算法检查第2个序列范围是否出现在第1个序列的范围内（顺序也完全一致），如果是则返回一个迭代器，该迭代器指向在第1个范围序列中第2个范围序列出现的开始位置。如果没有找到就返回**last1**。第1种形式测试使用**operator==**，第2种形式检测被比较的每对元素是否能使**binary_pred**返回**true**。

```
ForwardIterator1 find_end(ForwardIterator1 first1,
  ForwardIterator1 last1, ForwardIterator2 first2,
  ForwardIterator2 last2);
ForwardIterator1 find_end(ForwardIterator1 first1,
  ForwardIterator1 last1, ForwardIterator2 first2,
  ForwardIterator2 last2, BinaryPredicate binary_pred);
```

这些算法的形式和参数如同**search( )**，查找第2个范围的序列是否在第1个范围内作为子集出现，但是**search( )**查找该子集首先出现的位置，而**find_end( )**则查找该子集最后出现的位置，并且返回指向该子集的第一个元素的迭代器。

```
ForwardIterator search_n(ForwardIterator first,
  ForwardIterator last, Size count, const T& value);
ForwardIterator search_n(ForwardIterator first,
```

```
ForwardIterator last, Size count, const T& value,
BinaryPredicate binary_pred);
```

这些算法在**[first,last)** 范围内查找一组共**count**个连续的值，这些值都与**value**相等（在第1种形式中），或是当将所有这些与**value**相同的值传递给**binary_pred**时返回**true**（在第2种形式中）。如果不能找到这样的一组数值就返回**last**。

```
ForwardIterator min_element(ForwardIterator first,
  ForwardIterator last);
ForwardIterator min_element(ForwardIterator first,
  ForwardIterator last, BinaryPredicate binary_pred);
```

380

这些算法返回一个迭代器，该迭代器指向范围内“最小的”值首次出现的位置（如下面的解释——范围内可能会多次出现这个值）。如果范围为空则返回**last**。第1种版本用**operator<**执行比较，且返回值为**r**，其意义是：对于范围 **[first,r)** 中每个元素**e**，***e < *r**都为假。第2种版本用**binary_pred**比较，且返回值为**r**，其意义是：对于范围 **[first,r)** 中每个元素**e**，**binary_pred(*e ,*r)**都为假。

```
ForwardIterator max_element(ForwardIterator first,
  ForwardIterator last);
ForwardIterator max_element(ForwardIterator first,
  ForwardIterator last, BinaryPredicate binary_pred);
```

这些算法返回一个迭代器，该迭代器指向范围内最大值首次出现的位置。（范围内可能会多次出现最大值。）如果范围为空返回**last**。第1种版本用**operator<**执行比较，且返回值为**r**，其意义是：对于范围 **[first,r)** 中每个元素**e**，***r < *e**都为假。第2种版本用**binary_pred**执行比较，且返回值为**r**，其意义是：对于范围 **[first,r)** 中每个元素**e**，**binary_pred(*r,*e)**都为假。

```
void replace(ForwardIterator first, ForwardIterator last,
  const T& old_value, const T& new_value);
void replace_if(ForwardIterator first, ForwardIterator
  last, Predicate pred, const T& new_value);
OutputIterator replace_copy(InputIterator first,
  InputIterator last, OutputIterator result, const T&
  old_value, const T& new_value);
OutputIterator replace_copy_if(InputIterator first,
  InputIterator last, OutputIterator result, Predicate
  pred, const T& new_value);
```

在这些算法中，每一种“替换”形式都从头至尾在范围 **[first,last)** 内进行查找，找到与标准匹配的值并用**new_value**替换它们。**replace( )**和**replace_copy( )**都是仅仅查找**old_value**并对其进行替换；**replace_if( )**和**replace_copy_if( )**查找满足判定函数**pred**的值。函数的“复制”形式不修改原始范围，而是将作为替代的一个副本赋给**result**，更换它的值（每次赋值后增加**result**）。

381

程序举例

为了提供简单的可视结果，这个例子运算**int**型的**vector**。再强调一次，并不是将每一个算法的所有版本都展现出来。（一些意义很明显的算法被略去了。）

```
//: C06:SearchReplace.cpp
// The STL search and replace algorithms.
#include <algorithm>
#include <functional>
```

```
#include <vector>
#include "PrintSequence.h"
using namespace std;

struct PlusOne {
  bool operator()(int i, int j) { return j == i + 1; }
};

class MulMoreThan {
  int value;
public:
  MulMoreThan(int val) : value(val) {}
  bool operator()(int v, int m) { return v * m > value; }
};

int main() {
  int a[] = { 1, 2, 3, 4, 5, 6, 6, 7, 7, 7,
    8, 8, 8, 8, 11, 11, 11, 11, 11 };
  const int ASZ = sizeof a / sizeof *a;
  vector<int> v(a, a + ASZ);
  print(v.begin(), v.end(), "v", " ");
  vector<int>::iterator it = find(v.begin(), v.end(), 4);
  cout << "find: " << *it << endl;
  it = find_if(v.begin(), v.end(),
    bind2nd(greater<int>(), 8));
  cout << "find_if: " << *it << endl;
  it = adjacent_find(v.begin(), v.end());
  while(it != v.end()) {
    cout << "adjacent_find: " << *it
         << ", " << *(it + 1) << endl;
    it = adjacent_find(it + 1, v.end());
  }
  it = adjacent_find(v.begin(), v.end(), PlusOne());
  while(it != v.end()) {
    cout << "adjacent_find PlusOne: " << *it
         << "; " << *(it + 1) << endl;
    it = adjacent_find(it + 1, v.end(), PlusOne());
  }
  int b[] = { 8, 11 };
  const int BSZ = sizeof b / sizeof *b;
  print(b, b + BSZ, "b", " ");
  it = find_first_of(v.begin(), v.end(), b, b + BSZ);
  print(it, it + BSZ, "find_first_of", " ");
  it = find_first_of(v.begin(), v.end(),
    b, b + BSZ, PlusOne());
  print(it,it + BSZ,"find_first_of PlusOne"," ");
  it = search(v.begin(), v.end(), b, b + BSZ);
  print(it, it + BSZ, "search", " ");
  int c[] = { 5, 6, 7 };
  const int CSZ = sizeof c / sizeof *c;
  print(c, c + CSZ, "c", " ");
  it = search(v.begin(), v.end(), c, c + CSZ, PlusOne());
  print(it, it + CSZ,"search PlusOne", " ");
  int d[] = { 11, 11, 11 };
  const int DSZ = sizeof d / sizeof *d;
  print(d, d + DSZ, "d", " ");
  it = find_end(v.begin(), v.end(), d, d + DSZ);
  print(it, v.end(),"find_end", " ");
  int e[] = { 9, 9 };
  print(e, e + 2, "e", " ");
  it = find_end(v.begin(), v.end(), e, e + 2, PlusOne());
  print(it, v.end(),"find_end PlusOne"," ");
  it = search_n(v.begin(), v.end(), 3, 7);
```

382

```
  print(it, it + 3, "search_n 3, 7", " ");
  it = search_n(v.begin(), v.end(),
    6, 15, MulMoreThan(100));
  print(it, it + 6,
    "search_n 6, 15, MulMoreThan(100)", " ");
  cout << "min_element: "
       << *min_element(v.begin(), v.end()) << endl;
  cout << "max_element: "
       << *max_element(v.begin(), v.end()) << endl;
  vector<int> v2;
  replace_copy(v.begin(), v.end(),
    back_inserter(v2), 8, 47);
  print(v2.begin(), v2.end(), "replace_copy 8 -> 47", " ");
  replace_if(v.begin(), v.end(),
    bind2nd(greater_equal<int>(), 7), -1);
  print(v.begin(), v.end(), "replace_if >= 7 -> -1", " ");
} ///:~
```

383

这个例子以两个判定函数开始：**PlusOne**是一个二元判定函数，如果第2个参数等于第1个参数加1则返回**true**；**MulMoreThan**也是一个二元判定函数，如果第1个参数与第2个参数的乘积大于存储在对象中的值则返回**true**。这些二元判定函数在例子中用来进行测试。

在**main( )**中，创建一个数组**a**并将其提供给**vector<int> v**的构造函数。**vector**是查找和替换行动的目标，注意这里有很多重复元素——它们由一些查找/替换程序发现。

第1个测试演示**find( )**，在**v**中发现值4。返回值是指向4的第1个实例的迭代器，如果没有找到要查找的值，返回值指向输入范围的末尾（**v.end( )**）。

find_if( )算法使用了一个判定函数来决定是否找到了正确的元素。在本例中，用一个动态的**greater<int>**（即，“查看第1个**int**型参数是否大于第2个参数”）和固定的第2个参数8来创建一个执行中的判定函数**bind2nd( )**。因此，如果**v**中的值大于8则返回真。

因为在许多情况下**v**中会出现两个相同的相邻对象，所以设计**adjacent_find( )**测试来找到它们。查找从序列的开始位置出发，然后进入一个**while**循环，以便确定迭代器**it**没有到达该输入序列的末尾（这意味着不能再找到更多匹配的元素）。对于找到的每个匹配，循环打印这些匹配的元素并且执行下一个**adjacent_find( )**，这时就使用**it+1**作为第1个参数（用这种方式在三元组中能够找到两对匹配的元素）。

观察**while**循环，想一想如何能够使它的工作完成的更加精巧，如下所示：

```
while(it != v.end()) {
  cout << "adjacent_find: " << *it++
       << ", " << *it++ << endl;
  it = adjacent_find(it, v.end());
}
```

384

这个程序正是我们之前尝试的方式。但是该程序在任何编译器上都不可能得到期望的输出结果。这是因为对在循环内表达式中出现增1时的情况没有做出任何可靠的保证。

下一个测试使用了以**PlusOne**判定函数作为参数的**adjacent_find( )**，这个判定函数**PlusOne**能发现序列**v**中所有的下一个数比前一个数改变了1的元素位置。同样，采用**while**方法也能找到所有这样的情况。

算法**find_first_of( )**需要另外一个对象范围来辅助，这由数组**b**提供。由于**find_first_of( )**中的第1个和第2个范围由不同的模板参数控制，正如所见，这两个范围可以引用两个不同类型的容器。**find_first_of( )**的第2种形式也进行了测试，使用了**PlusOne**。

search( )算法精确地在第1个序列范围内找到了第2个范围序列，并且它们的元素具有相同

的顺序。**search( )**的第2种形式使用了一个判定函数，该形式的典型应用是查找那些定义相等的序列，但也有可能进行更加有趣的查找——在这里，**PlusOne**判定函数找到的范围是**{4,5,6}**。

find_end( )测试发现了在整个序列的最后出现的**{11,11,11}**。为了显示它实际上已经找到了最后出现的这个子集，从迭代器**it**指向的位置开始打印**v**串的剩余部分。

第1个**search_n( )**测试寻找7的3个副本，找到它们并且打印出来。当使用**search_n( )**的第2种版本时，判定函数的出现一般意味着使用它来判定两个元素间的相等性，但是也可以有一些选择的自由。并且使用一个函数对象，这个函数对象是用15（在本例中）去乘序列中的值，并且检查它们是否大于100。也就是说，**search_n( )**检测要做的是“找到6连续的那些值，当被15乘时，每个产生的数大于100。”这不能精确地描述读者平常期望做的那些工作，但是却可以为下次遇到不寻常的查找问题时提供一些办法。

min_element( )和**max_element( )**算法很直观，但是看上去有些怪异。函数似乎是用一个“*”来引用。实际上，返回的迭代器被释放掉以便产生打印的值。

为了测试替换，首先使用**replace_copy( )**（这样不会修改原始**vector**）以值47来替换所有值为8的元素。注意，用一个空的**vector v2**对**back_inserter( )**进行调用。为了演示**replace_if( )**，用标准模板**greater_equal**连同**bind2nd**创建一个函数对象，用－1替换所有值大于等于7的元素。 385

6.3.6 比较范围

下面这些算法提供比较两个范围的方法。乍看起来，这些算法执行的运算类似**search( )**函数。可是，**search( )**查找的是第2个序列出现在第1个序列中的位置，而**equal( )**和**lexicographical_compare( )**所做的只是进行两个序列的比较。另一方面，**mismatch( )**比较两个序列在哪里停止同步比较，这两个序列必须有完全相同的长度。

```
bool equal(InputIterator first1, InputIterator last1,
  InputIterator first2);
bool equal(InputIterator first1, InputIterator last1,
  InputIterator first2 BinaryPredicate binary_pred);
```

在这两个函数中，**[first1,last1)**表示的第1个范围是一个典型的表示方法。第2个范围开始于**first2**，但是没有“last2”因为第2个范围的长度由第1个范围的长度来决定。如果两个范围完全相同（有相同的元素和相同的顺序），**equal( )**函数返回真。在第1种情况中，由**operator==**执行比较，在第2种情况中，由**binary_pred**来决定两个元素是否相同。

```
bool lexicographical_compare(InputIterator1 first1,
  InputIterator1 last1, InputIterator2 first2,
  InputIterator2 last2);
bool lexicographical_compare(InputIterator1 first1,
  InputIterator1 last1, InputIterator2 first2,
  InputIterator2 last2, BinaryPredicate binary_pred);
```

这两个函数决定第1个范围是否“字典编纂顺序的小于（lexicographically less）”第2个范围。（如果范围1小于范围2返回**true**，否则返回**false**。）字典编纂顺序的比较(lexicographical comparison)或称为“字典(dictionary)”比较，意味着比较的顺序等同于按字典规则建立字符串的顺序：每次比较一个元素。如果第1个元素不同，第1个元素就决定了两个字符串比较的结果，但是如果相同，算法移到下一个元素继续检查它们，如此下去直到遇到不匹配的那对元素为止。在这个点上，检查这对元素，如果范围1序列中的这个元素小于范围2序列中的相应元素，**lexicographical_compare( )**返回**true**；否则返回**false**。如果用能得到的所有方法从头 386

至尾扫描一个范围或另一个范围（本算法中范围的长度可以不同），都没有发现不相等的地方，范围1不小于范围2，因此函数返回**false**。

如果两个范围的长度不同，按字典编纂顺序，一个范围内缺少的元素起到“领先于(precede)”另一个范围内存在的元素的作用，因此“abc”领先于“abcd”。如果算法执行到一个范围的结尾，还没有找到不匹配的元素对，这时短的范围领先（按字典编纂顺序领先，即小）。在这种情况下，如果短的范围是第1个范围，则结果是**true**，反之是**false**。

在函数的第1种版本中，由**operator<**执行比较，在第2种版本中，使用判定函数**binary_pred**。

```
pair<InputIterator1, InputIterator2>
mismatch(InputIterator1 first1, InputIterator1 last1,
  InputIterator2 first2);
pair<InputIterator1, InputIterator2>
mismatch(InputIterator1 first1, InputIterator1 last1,
  InputIterator2 first2, BinaryPredicate binary_pred);
```

这些算法如同在**equal( )**中一样，进行比较的两个范围的长度完全相同，因此仅需要第2个范围的**first**迭代器，第1个范围的长度可以用来做第2个范围的长度。这个函数的功能与**equal( )**正好相反，**equal( )**仅比较两个范围是否相同，**mismatch( )**告诉比较从哪里开始不同。为了完成这一工作，必须知道以下几点（1）第1个范围内出现不匹配的元素的位置；（2）第2个范围内出现不匹配的元素的位置。将两个迭代器一起装入一个**pair**对象并返回。如果没有出现不匹配，返回值是与第2个范围结合在一起的超越末尾的迭代器**last1**。**pair**模板类是一个**struct**，该**struct**含有两个用成员名**first**和**second**表示的元素，在**<utility>**头文件中定义。

如同在**equal( )**中一样，第1个函数测试相等性使用**operator==**，而第2个使用**binary_pred**。

387

程序举例

因为标准C++ **string**类构造得如同一个容器（它含产生类型**string::iterator**的对象成员函数**begin( )**和**end( )**），可以方便地用来创建字符范围序列来测试STL比较算法。然而，需要注意的是，**string**有一个相当完整的属于自己的运算集，因此在使用STL算法执行运算之前，需要查看一下**string**类。

```
//: C06:Comparison.cpp
// The STL range comparison algorithms.
#include <algorithm>
#include <functional>
#include <string>
#include <vector>
#include "PrintSequence.h"
using namespace std;

int main() {
  // Strings provide a convenient way to create
  // ranges of characters, but you should
  // normally look for native string operations:
  string s1("This is a test");
  string s2("This is a Test");
  cout << "s1: " << s1 << endl << "s2: " << s2 << endl;
  cout << "compare s1 & s1: "
       << equal(s1.begin(), s1.end(), s1.begin()) << endl;
  cout << "compare s1 & s2: "
       << equal(s1.begin(), s1.end(), s2.begin()) << endl;
  cout << "lexicographical_compare s1 & s1: "
```

```
        << lexicographical_compare(s1.begin(), s1.end(),
           s1.begin(), s1.end()) <<  endl;
  cout << "lexicographical_compare s1 & s2: "
        << lexicographical_compare(s1.begin(), s1.end(),
           s2.begin(), s2.end()) << endl;
  cout << "lexicographical_compare s2 & s1: "
        << lexicographical_compare(s2.begin(), s2.end(),
           s1.begin(), s1.end()) << endl;
  cout << "lexicographical_compare shortened "
           "s1 & full-length s2: " << endl;
  string s3(s1);
  while(s3.length() != 0) {
    bool result = lexicographical_compare(
      s3.begin(), s3.end(), s2.begin(),s2.end());
    cout << s3 << endl << s2 << ", result = "
          << result << endl;
     if(result == true) break;
     s3 = s3.substr(0, s3.length() - 1);
   }
   pair<string::iterator, string::iterator> p =
     mismatch(s1.begin(), s1.end(), s2.begin());
   print(p.first, s1.end(), "p.first", "");
   print(p.second, s2.end(), "p.second","");
} ///:~
```

388

注意，**s1**和**s2**的惟一不同点是：**s2**的“Test”中的大写字母“T”。比较**s1**和**s2**的相等性产生**true**。不出所料，因为大写字母“T”，**s1**和**s2**不相等。

为了理解**lexicographical_compare( )**测试的输出结果，要记住两件事：首先，比较是按一个字母接着一个字母的顺序执行的；第二，现在C++编译系统的操作平台上，大写字母字符“领先于”小写字母字符。在第1个测试中，是**s1**与**s1**进行比较。这当然是完全相等。以字典编纂顺序进行比较，不会产生一个序列小于另外一个序列的结果(这是比较要寻找的结果)，因此结果是**false**。第2个测试是问“**s1**领先于**s2**吗”？当比较进行到“test”中的第1个字符‘t’时，发现**s1**中的小写字母字符‘t’“大于”**s2**中的大写字母字符“T”，所以答案是**false**。但是，如果测试要看看**s2**是否领先于**s1**，答案是**true**。

为了更进一步地检测字典编纂顺序比较，本例中的下一个测试再次比较**s1**和**s2**（前面的比较返回**false**）。这次重复这个比较，每次通过循环减去**s1**（首先将**s1**复制到**s3**）末尾的一个字符，直到测试结果返回**true**。读者将会看到些什么？看到的是：只要从**s3**（**s1**的副本）中依次减到大写字母“T”，此时，则两个序列从开始一直到这一点都完全相等，不再进行计算。因为**s3**比**s2**短，**s3**字典编纂顺序领先于**s2**。

最后的测试使用**mismatch( )**。为了得到返回值，现在创建合适的**pair p**，用第1个范围的迭代器类型及第2个范围的迭代器类型（在本例中，都是**string::iterator**）构造模板。为了打印该函数产生的结果，函数中第1个范围的不匹配迭代器是**p.first**，第2个范围的迭代器是**p.second**。在这两种情况中，从函数不匹配的迭代器开始到范围的末尾来打印该范围序列，所以可以准确地看到哪里是迭代器指出的点。

6.3.7 删除元素

389

因为STL的通用性，这里对删除的概念有一点限制。既然在STL中仅能通过迭代器“删除”元素，而迭代器可以指向数组、**vector**、**list**等等，那么试图消毁正在被删除的元素和改变输入范围 **[first,last)** 的大小是不安全或是不合理的。（例如，一个已存在数组不能改变它的大小。）因此取而代之，STL“删除”函数重新排列该序列，就是将“已被删除的”元素排在序

列的末尾，“未删除的”元素排在序列的开头（与以前的顺序相同，只是减去被删除的元素——也就是说，这是一个稳定的操作）。然后函数返回一个指向序列的“新末尾”元素的迭代器，这个“新末尾”元素是不含被删除元素的序列的末尾，也是被删除元素序列的开头。换句话说，如果**new_last**是从“删除”函数返回的迭代器，则范围 **[first,new_last)** 是不包含任何被删除元素的序列，而范围 **[new_last，last)** 是被删除元素组成的序列。

如果想通过更多的STL算法来简单地使用序列，并把那些已被删除的元素包括在序列内，可以仅用**new_last**作为新的超越末尾的迭代器。但是，如果使用一个可以调整大小的容器**c**（不是一个数组），当想从容器中消除被删除的元素时，可以使用**erase( )**来完成，例如：

```
c.erase(remove(c.begin(), c.end(), value), c.end());
```

也可以使用属于所有标准序列容器的**resize( )**成员函数（更多关于此问题的内容将在第7章介绍）。

remove( )的返回值是称为**new_last**的迭代器，而**erase( )**则从**c**中真正删除掉所有的要被删除的元素。

[new_last,last) 中的迭代器是能解析的，但是那些元素值未被指定，应该不再使用。

```
ForwardIterator remove(ForwardIterator first,
  ForwardIterator last, const T& value);
ForwardIterator remove_if(ForwardIterator first,
  ForwardIterator last, Predicate pred);
OutputIterator remove_copy(InputIterator first,
  InputIterator last, OutputIterator result, const T&
  value);
OutputIterator remove_copy_if(InputIterator first,
  InputIterator last, OutputIterator result, Predicate
  pred);
```

390

这里介绍的每一种“删除”形式都从头至尾遍历范围 **[first, last)**，找到符合删除标准的值，并且复制未被删除的元素覆盖已被删除的元素（因此可有效地删除元素）。未被删除的元素的原始排列顺序仍然保持。返回值是指向超越范围末尾的迭代器，该范围不包含任何已被删除的元素。这个迭代器指向的元素的值未被指定。

“if”版本的删除把每一个元素传递给判定函数**pred( )**，来决定是否应该删除。（如果**pred( )**返回**true**，则删除该元素。）“copy”版本的删除不需要修改原始序列，而取而代之是复制未被删除的值到一个开始于**result**的新范围，并返回指向新范围的超越末尾的迭代器。

```
ForwardIterator unique(ForwardIterator first,
  ForwardIterator last);
ForwardIterator unique(ForwardIterator first,
  ForwardIterator last, BinaryPredicate binary_pred);
OutputIterator unique_copy(InputIterator first,
  InputIterator last, OutputIterator result);
OutputIterator unique_copy(InputIterator first,
  InputIterator last, OutputIterator result,
  BinaryPredicate binary_pred);
```

在这些算法中，“unique”函数的每一种版本都从头至尾遍历范围 **[first, last)**，找到相邻的相等值（即副本），并且通过复制覆盖它们来“删除”这些副本。未被删除的元素的原始顺序仍然保持不变。返回值是指向该范围的超越末尾的迭代器，该范围相邻副本已被删除。

因为要删除的只是相邻的副本，因此如果有可能的话，在调用“unique”算法之前，调用

sort( )，这样就能保证全部的副本都被删除掉。

对于输入范围内的每个迭代器的值**i**，包含在**binary_pred**调用版本中：

391

```
binary_pred(*i, *(i-1));
```

如果返回值是**true**，则认为***i**是一个副本。

"copy"版本不改变原始序列，取而代之复制未被删除的值到一个开始于**result**的新范围，并返回指向新范围的超越末尾的迭代器。

程序举例

这个例子给出了"remove"和"unique"函数工作的一个演示。

```
//: C06:Removing.cpp
// The removing algorithms.
//{L} Generators
#include <algorithm>
#include <cctype>
#include <string>
#include "Generators.h"
#include "PrintSequence.h"
using namespace std;

struct IsUpper {
  bool operator()(char c) { return isupper(c); }
};

int main() {
  string v;
  v.resize(25);
  generate(v.begin(), v.end(), CharGen());
  print(v.begin(), v.end(), "v original", "");
  // Create a set of the characters in v:
  string us(v.begin(), v.end());
  sort(us.begin(), us.end());
  string::iterator it = us.begin(), cit = v.end(),
    uend = unique(us.begin(), us.end());
  // Step through and remove everything:
  while(it != uend) {
    cit = remove(v.begin(), cit, *it);
    print(v.begin(), v.end(), "Complete v", "");
    print(v.begin(), cit, "Pseudo v ", " ");
    cout << "Removed element:\t" << *it
         << "\nPsuedo Last Element:\t"
         << *cit << endl << endl;
    ++it;
  }
  generate(v.begin(), v.end(), CharGen());
  print(v.begin(), v.end(), "v", "");
  cit = remove_if(v.begin(), v.end(), IsUpper());
  print(v.begin(), cit, "v after remove_if IsUpper", " ");
  // Copying versions are not shown for remove()
  // and remove_if().
  sort(v.begin(), cit);
  print(v.begin(), cit, "sorted", " ");
  string v2;
  v2.resize(cit - v.begin());
  unique_copy(v.begin(), cit, v2.begin());
  print(v2.begin(), v2.end(), "unique_copy", " ");
  // Same behavior:
  cit = unique(v.begin(), cit, equal_to<char>());
  print(v.begin(), cit, "unique equal_to<char>", " ");
} ///:~
```

392

字符串v是一个由随机产生的字符填满的字符容器。每个字符在remove语句中都被使用，但是每次都显示全部的字符串v，因此在得到结束点以后（存储在cit中），就可以看到该范围的剩余部分到底发生了什么变化。

为了演示remove_if( )，在函数对象类IsUpper中调用标准C库函数isupper( )（在<cctype>中），将一个对象作为一个判定函数传递给remove_if( )。仅当字符是大写的时候返回true，因此只保留小写字符。在这里，在print( )的调用中使用了范围的末尾作为参数，因此仅显示保留的元素。remove( )和remove_if( )的复制形式没有演示，因为它们是非复制版本的一个简单变种，无需例子就应该会使用。

先对小写字母的序列进行排序，为测试“unique”函数做准备。（如果该序列未排序，则“unique”函数就不能被定义，但这大概并不是读者想要的。）首先，unique_copy( )使用默认的元素比较将序列中独一无二的元素放入一个新的vector中，然后再使用含有判定函数的unique( )形式。判定函数嵌入到函数对象equal_to( )中，它与默认的元素比较产生相同的结果。

393

6.3.8 对已排序的序列进行排序和运算

STL算法的一个重要种类就是必须对已排好序的范围序列进行运算。STL提供了大量独立的排序算法，分别对应于稳定的、部分的或仅是规则的（不稳定的）排序。说也奇怪，只有部分排序有复制的版本。如果使用其他排序算法并且需要在一个副本上工作，那么就需要在排序前由用户自己来完成复制工作。

对于一个已经排好序的序列，可以在该序列上执行多种运算，包括从该序列中找出指定的某个元素或某组元素，到与另外的一个已排序的序列进行合并，或像数学集合一样来运算该序列等等。

对已排好序的序列进行包括排序或运算的每个算法都有两种版本。第1种版本使用对象自己的operator< 来执行比较，第2种版本用operator( )(a,b)来决定a和b的相对顺序。除此之外没有什么不同之处，所以不会在每个算法的描述中都指出这个不同点。

1. 排序

排序算法需要由随机存取的迭代器来限制序列的范围，比如vector或deque。list容器有自己的嵌入sort( )函数，因为它仅提供双向的迭代。

```
void sort(RandomAccessIterator first, RandomAccessIterator
  last);
void sort(RandomAccessIterator first, RandomAccessIterator
  last, StrictWeakOrdering binary_pred);
```

这些算法将 [first,last) 范围内的序列按升序顺序排序。第1种形式使用operator<，第2种形式使用提供的比较器对象来决定顺序。

```
void stable_sort(RandomAccessIterator first,
  RandomAccessIterator last);
void stable_sort(RandomAccessIterator first,
  RandomAccessIterator last, StrictWeakOrdering
  binary_pred);
```

394

这些算法将[first,last) 范围内的序列按升序顺序排序，保持相等元素的原始顺序。（假设元素可以是相等的但不是相同的，这一点很重要。）

```
void partial_sort(RandomAccessIterator first,
  RandomAccessIterator middle, RandomAccessIterator last);
```

```
void partial_sort(RandomAccessIterator first,
  RandomAccessIterator middle, RandomAccessIterator last,
  StrictWeakOrdering binary_pred);
```

这些算法对来自**[first,last)**范围中的一定数量的元素进行排序，这些元素可以放入范围**[first,middle)**中。排序结束，在范围**[middle,last)**中其余的那些元素并不保证它们的顺序。

```
RandomAccessIterator partial_sort_copy(InputIterator first,
  InputIterator last, RandomAccessIterator result_first,
  RandomAccessIterator result_last);
RandomAccessIterator partial_sort_copy(InputIterator first,
  InputIterator last, RandomAccessIterator result_first,
  RandomAccessIterator result_last, StrictWeakOrdering
  binary_pred);
```

这些算法对来自**[first,last)**范围中的一定数量的元素进行排序，这些元素可以放入范围**[result_first,result_last)**中，并且复制这些元素到 **[result_first,result_last)**。如果范围**[first,last)**比**[result_first,result_last)**小，则使用较少的元素。

```
void nth_element(RandomAccessIterator first,
  RandomAccessIterator nth, RandomAccessIterator last);
void nth_element(RandomAccessIterator first,
  RandomAccessIterator nth, RandomAccessIterator last,
  StrictWeakOrdering binary_pred);
```

这些算法如同**partial_sort( )**，**nth_element( )**部分地处理（排列）范围内的元素。但是，它比**partial_sort( )**要“少处理”得多。**nth_element( )**惟一保证的是无论选择什么位置，该位置都会成为一个分界点。范围**[first,nth)**内的所有元素都会成对地满足二元判定函数（通常默认的是**operator<**），而范围**(nth,last]**内的所有元素都不满足该判定。但是，任何一个子范围都不会是一个以特定的顺序排好序的序列，这不像**partial_sort( )**，它的第1个范围已排好序。 395

如果需要的是很弱的排序处理（例如，决定中值、百分点等等），这个算法要比**partial_sort( )**快得多。

2. 在已排序的范围中找出指定元素

一旦某个范围被排好序，就可以在范围内使用一系列运算来查找元素。在下面的函数中，总是存在有两种形式。一种是假定内在的**operator<** 来执行排序，第2种运算符是使用一些其他的比较函数对象来执行排序。必须使用与执行排序相同的比较方法来定位元素；否则，结果不确定。另外，如果试图在未排序的范围上使用这些函数，结果将不可预料。

```
bool binary_search(ForwardIterator first, ForwardIterator
  last, const T& value);
bool binary_search(ForwardIterator first, ForwardIterator
  last, const T& value, StrictWeakOrdering binary_pred);
```

这些算法告诉用户是否**value**出现在已排序的范围 **[first,last)** 中。

```
ForwardIterator lower_bound(ForwardIterator first,
  ForwardIterator last, const T& value);
ForwardIterator lower_bound(ForwardIterator first,
  ForwardIterator last, const T& value, StrictWeakOrdering
  binary_pred);
```

这些算法返回一个迭代器，该迭代器指出**value**在已排序的范围 **[first,last)** 中第1次出

现的位置。如果**value**没有出现，返回的迭代器则指出它在该序列中应该出现的位置。

```
ForwardIterator upper_bound(ForwardIterator first,
  ForwardIterator last, const T& value);
ForwardIterator upper_bound(ForwardIterator first,
  ForwardIterator last, const T& value, StrictWeakOrdering
  binary_pred);
```

396 这些算法返回一个迭代器，该迭代器指出在已排序的范围 **[first,last)** 中超越**value**最后出现的一个位置。如果**value**没有出现，返回的迭代器则指出它在该序列中应该出现的位置。

```
pair<ForwardIterator, ForwardIterator>
equal_range(ForwardIterator first, ForwardIterator last,
  const T& value);
pair<ForwardIterator, ForwardIterator>
equal_range(ForwardIterator first, ForwardIterator last,
  const T& value, StrictWeakOrdering binary_pred);
```

在这些算法中，本质上结合了**lower_bound( )**和**upper_bound( )**，返回一个指出**value**在已排序的范围 **[first,last)** 中的首次出现和超越最后出现的**pair**。如果没有找到，这两个迭代器都指出**value**在该序列中应该出现的位置。

读者可能会惊讶于一个发现，即二分查找（也称折半查找）算法使用一个前向顺序查找的迭代器而不是随机存取的迭代器。（绝大多数对二分查找的解释是使用索引。）记住随机存取迭代器“是（is-a）”向前顺序查找的迭代器，它可以用在后者（向前顺序查找的）指定的地方。如果传递给这些算法之一的迭代器实际上支持随机存取，则使用了有效率的对数时间查找，否则执行的是线性查找。[⊖]

3. 程序举例

下面的例子将输入的每一个单词转化成**NString**，并且将其加入到**vector<NString>**。然后使用**vector**来演示各种排序和查找算法。

```
//: C06:SortedSearchTest.cpp
// Test searching in sorted ranges.
// NString
#include <algorithm>
#include <cassert>
#include <ctime>
#include <cstdlib>
#include <cstddef>
#include <fstream>
#include <iostream>
#include <iterator>
#include <vector>
#include "NString.h"
#include "PrintSequence.h"
#include "../require.h"
using namespace std;

int main(int argc, char* argv[]) {
  typedef vector<NString>::iterator sit;
  char* fname = "Test.txt";
  if(argc > 1) fname = argv[1];
  ifstream in(fname);
  assure(in, fname);
```

397

⊖ 通过读取**tag**，算法能够决定迭代器的类型，这将在第7章讨论。

```
  srand(time(0));
  cout.setf(ios::boolalpha);
  vector<NString> original;
  copy(istream_iterator<string>(in),
    istream_iterator<string>(), back_inserter(original));
  require(original.size() >= 4, "Must have four elements");
  vector<NString> v(original.begin(), original.end()),
    w(original.size() / 2);
  sort(v.begin(), v.end());
  print(v.begin(), v.end(), "sort");
  v = original;
  stable_sort(v.begin(), v.end());
  print(v.begin(), v.end(), "stable_sort");
  v = original;
  sit it = v.begin(), it2;
  // Move iterator to middle
  for(size_t i = 0; i < v.size() / 2; i++)
    ++it;
  partial_sort(v.begin(), it, v.end());
  cout << "middle = " << *it << endl;
  print(v.begin(), v.end(), "partial_sort");
  v = original;
  // Move iterator to a quarter position
  it = v.begin();
  for(size_t i = 0; i < v.size() / 4; i++)
    ++it;
  // Less elements to copy from than to the destination
  partial_sort_copy(v.begin(), it, w.begin(), w.end());
  print(w.begin(), w.end(), "partial_sort_copy");
  // Not enough room in destination
  partial_sort_copy(v.begin(), v.end(), w.begin(),w.end());
  print(w.begin(), w.end(), "w partial_sort_copy");
  // v remains the same through all this process
  assert(v == original);
  nth_element(v.begin(), it, v.end());
  cout << "The nth_element = " << *it << endl;
  print(v.begin(), v.end(), "nth_element");
  string f = original[rand() % original.size()];
  cout << "binary search: "
       << binary_search(v.begin(), v.end(), f) << endl;
  sort(v.begin(), v.end());
  it = lower_bound(v.begin(), v.end(), f);
  it2 = upper_bound(v.begin(), v.end(), f);
  print(it, it2, "found range");
  pair<sit, sit> ip = equal_range(v.begin(), v.end(), f);
  print(ip.first, ip.second, "equal_range");
} ///:~
```

398

这个例子使用前面见过的**NString**类，它存储一个字符串的副本出现的次数。**stable_sort( )**的调用显示了含相等字符串的对象的原始顺序是如何保存的。同时也可以看到在“部分排序”期间到底发生什么事情（保留的未排序的元素处在非特定的顺序之中）。不存在“部分的稳定排序。”

注意，在**nth_element( )**的调用中，无论nth元素变成什么（因为**URandGen**发生器，它可以从一个元素变成另一个元素），其前面的元素总是小于它，后面的元素大于它，此外这些元素并没有特定的顺序。由于**URandGen**发生器不存在副本，但是如果使用允许副本的发生器，将会看到nth元素以前的元素小于等于该nth元素。

这个例子也演示了全部的3个二分查找算法。同介绍过的一样，**lower_bound( )**用来查找序列中第1个等于给定关键字值的元素，**upper_bound( )**指向个最后一个符合条件元素的

下一元素，而**equal_range( )**将两个结果作为一对数据返回。

4. 合并已排序的序列

如同前面一样，每个函数的第1种形式假定由内在的**operator<**执行排序。第2种形式必须使用一些其他比较函数对象执行排序。必须使用与执行排序相同的比较方法来定位元素；否则，结果不确定。另外，如果试图在未排序的序列范围上使用这些算法，结果也会不可预料。

399

```
OutputIterator merge(InputIterator1 first1, InputIterator1
  last1, InputIterator2 first2, InputIterator2 last2,
  OutputIterator result);
OutputIterator merge(InputIterator1 first1, InputIterator1
  last1, InputIterator2 first2, InputIterator2 last2,
  OutputIterator result, StrictWeakOrdering binary_pred);
```

在这些算法中，从**[first1,last1)** 和 **[first2,last2)** 中复制元素到**result**，这样在结果范围的序列以升序的顺序排序。这是一个稳定的运算。

```
void inplace_merge(BidirectionalIterator first,
  BidirectionalIterator middle, BidirectionalIterator
  last);
void inplace_merge(BidirectionalIterator first,
  BidirectionalIterator middle, BidirectionalIterator last,
  StrictWeakOrdering binary_pred);
```

这里假定**[first,middle)**和**[middle,last)** 是在相同的序列中已排好序的两个范围。合并这两个范围序列到一个结果序列，该结果序列范围**[first,last)** 包含将两个排好序的范围结合成一个有序的范围。

5. 程序举例

很容易看到，如果在合并中使用**int**类型将会发生什么事情。下面的例子同时也强调了算法（以及我们自己定义的**print**模板）是怎样与数组和容器一起工作的：

400

```
//: C06:MergeTest.cpp
// Test merging in sorted ranges.
//{L} Generators
#include <algorithm>
#include "PrintSequence.h"
#include "Generators.h"
using namespace std;

int main() {
  const int SZ = 15;
  int a[SZ*2] = {0};
  // Both ranges go in the same array:
  generate(a, a + SZ, SkipGen(0, 2));
  a[3] = 4;
  a[4] = 4;
  generate(a + SZ, a + SZ*2, SkipGen(1, 3));
  print(a, a + SZ, "range1", " ");
  print(a + SZ, a + SZ*2, "range2", " ");
  int b[SZ*2] = {0}; // Initialize all to zero
  merge(a, a + SZ, a + SZ, a + SZ*2, b);
  print(b, b + SZ*2, "merge", " ");
  // Reset b
  for(int i = 0; i < SZ*2; i++)
    b[i] = 0;
  inplace_merge(a, a + SZ, a + SZ*2);
  print(a, a + SZ*2, "inplace_merge", " ");
  int* end = set_union(a, a + SZ, a + SZ, a + SZ*2, b);
```

```
  print(b, end, "set_union", " ");
} ///:~
```

在**main( )**中，不是创建两个独立的数组，而是在数组**a**中创建两个首尾相连的范围。（这为**inplace_merge**带来方便。）第1个**merge( )**的调用把结果放入一个不同的数组**b**中。为了进行比较，同时也调用**set_union( )**，它与第1个**merge( )**的调用有相同的标识符及类似的行为，除了它从第2个集合中删除副本。最后，**inplace_merge( )**将**a**的两个部分结合到一起。

6. 在已排序的序列上进行集合运算

一旦范围已排好序，就可以在其上执行数学集合运算。

```
bool includes(InputIterator1 first1, InputIterator1 last1,
  InputIterator2 first2, InputIterator2 last2);
bool includes(InputIterator1 first1, InputIterator1 last1,
  InputIterator2 first2, InputIterator2 last2,
  StrictWeakOrdering binary_pred);
```

在这些算法中，如果**[first2,last2)** 是 **[first1,last1)** 的一个子集，返回**true**。没有任何一个范围要求只持有与另一个范围完全不同的元素，但是如果 **[first2,last2)** 持有**n**个特定值的元素，假如要想使返回结果为**true**，**[first1,last1)** 也必须同时至少持有**n**个元素。

401

```
OutputIterator set_union(InputIterator1 first1,
  InputIterator1 last1, InputIterator2 first2,
  InputIterator2 last2, OutputIterator result);
OutputIterator set_union(InputIterator1 first1,
  InputIterator1 last1, InputIterator2 first2,
  InputIterator2 last2, OutputIterator result,
  StrictWeakOrdering binary_pred);
```

这些算法在**result**范围中创建两个已排序范围的数学并集，返回值指向输出范围的末尾。没有任何一个输入范围要求只持有与另一个范围完全不同的元素，但是，如果在两个输入集合中多次出现某个特定值，结果集合中将包含完全相同的值出现的较大次数。

```
OutputIterator set_intersection(InputIterator1 first1,
  InputIterator1 last1, InputIterator2 first2,
  InputIterator2 last2, OutputIterator result);
OutputIterator set_intersection(InputIterator1 first1,
  InputIterator1 last1, InputIterator2 first2,
  InputIterator2 last2, OutputIterator result,
  StrictWeakOrdering binary_pred);
```

这些算法在**result**中产生两个输入集合的交集，返回值指向输出范围的末尾——即在两个输入集合中都出现的数值的集合。没有任何一个输入范围要求只持有与另一个范围完全不同的元素，但是如果某个特定值在两个输入集合中多次出现，结果集合中将包含完全相同的值出现的较小次数。

```
OutputIterator set_difference(InputIterator1 first1,
  InputIterator1 last1, InputIterator2 first2,
  InputIterator2 last2, OutputIterator result);
OutputIterator set_difference(InputIterator1 first1,
  InputIterator1 last1, InputIterator2 first2,
  InputIterator2 last2, OutputIterator result,
  StrictWeakOrdering binary_pred);
```

这些算法在**result**中产数学上集合的差，返回值指向输出结果范围的末尾。所有出现在

[frist1, last1) 中，但不在**[first2, last2)** 中出现的元素都放入结果集合。没有任何一个输入范围要求只持有独特的元素，但是如果某个特定值在两个输入集合中多次出现（在集合1中**n**次，集合2中**m**次），结果集合将包含这个值的**max(n-m,o)**个副本。

402

```
OutputIterator set_symmetric_difference(InputIterator1
  first1, InputIterator1 last1, InputIterator2 first2,
  InputIterator2 last2, OutputIterator result);
OutputIterator set_symmetric_difference(InputIterator1
  first1, InputIterator1 last1, InputIterator2 first2,
  InputIterator2 last2, OutputIterator result,
  StrictWeakOrdering binary_pred);
```

在**result**集合构成中，包括：

1) 所有在集合1中而不在集合2中的元素。

2) 所有在集合2中而不在集合1中的元素。

在这些算法中，没有任何一个输入范围要求只持有独特的元素，但是如果某个特定值在两个输入集合中多次出现（在集合1中**n**次，集合2中**m**次），结果集合将包含这个值的**abs(n-m)**个副本，其中**abs( )**是取绝对值函数。返回值指向输出结果范围的末尾。

7. 程序举例

观察仅使用字符的**vector**来演示集合运算将更加容易。这些字符是随机产生的，并被排序，但保留了副本，当有了副本时，现在就可以看到集合运算怎样执行。

```
//: C06:SetOperations.cpp
// Set operations on sorted ranges.
//{L} Generators
#include <algorithm>
#include <vector>
#include "Generators.h"
#include "PrintSequence.h"
using namespace std;

int main() {
  const int SZ = 30;
  char v[SZ + 1], v2[SZ + 1];
  CharGen g;
  generate(v, v + SZ, g);
  generate(v2, v2 + SZ, g);
  sort(v, v + SZ);
  sort(v2, v2 + SZ);
  print(v, v + SZ, "v", "");
  print(v2, v2 + SZ, "v2", "");
  bool b = includes(v, v + SZ, v + SZ/2, v + SZ);
  cout.setf(ios::boolalpha);
  cout << "includes: " << b << endl;
  char v3[SZ*2 + 1], *end;
  end = set_union(v, v + SZ, v2, v2 + SZ, v3);
  print(v3, end, "set_union", "");
  end = set_intersection(v, v + SZ, v2, v2 + SZ, v3);
  print(v3, end, "set_intersection", "");
  end = set_difference(v, v + SZ, v2, v2 + SZ, v3);
  print(v3, end, "set_difference", "");
  end = set_symmetric_difference(v, v + SZ,
    v2, v2 + SZ, v3);
  print(v3, end, "set_symmetric_difference","");
} ///:~
```

403

在**v**和**v2**产生、排序和打印之后，通过观察**v**的全部范围是否包含**v**的后半部分来测试

includes( )算法。如果包括，结果通常应该是真。数组**v3**保存**set_union( )**、**set_intersection( )**、**set_difference( )**和**set_symmetric_difference( )**的输出结果，每一个结果都显示出来，这样读者就可以分析、思考它们，并确信算法正如预想的那样执行。

6.3.9 堆运算

堆是一个像数组的数据结构，用来实现“优先队列”，“优先队列”是一个靠优先权调节检索元素的方式来组织的序列，其中优先权是依据某些比较函数决定的。标准库中的堆运算允许一个序列被视为是一个“堆”数据结构，这通常可以有效地返回最高优先权的元素，而无需全部排序整个序列。

如同“排序”运算一样，每个函数都有两种版本。第1种使用对象自己的**operator<**来执行比较；第2种使用另外的**StrictWeakOrdering**对象的**operator( )(a,b)**来比较两个对象：**a < b**。

```
void make_heap(RandomAccessIterator first,
  RandomAccessIterator last);
void make_heap(RandomAccessIterator first,
  RandomAccessIterator last,
  StrictWeakOrdering binary_pred);
```

404

这些算法将一个任意序列范围转化成堆。

```
void push_heap(RandomAccessIterator first,
  RandomAccessIterator last);
void push_heap(RandomAccessIterator first,
  RandomAccessIterator last,
  StrictWeakOrdering binary_pred);
```

这些算法向由范围**[first,last-1)** 决定的堆中增加元素***(last-1)**。换句话说，将最后一个元素放入堆中合适的位置。

```
void pop_heap(RandomAccessIterator first,
  RandomAccessIterator last);
void pop_heap(RandomAccessIterator first,
  RandomAccessIterator last,
  StrictWeakOrdering binary_pred);
```

在这些算法中，将最大的元素（在运算前实际上在***first**中，这是堆定义方式的缘故）放入位置***(last-1)**并且重新组织剩余的范围，使其仍然在堆的顺序中。如果只是抓取***first**，下一个元素就将不是下一个最大的元素；因此，如果想以完全优先队列的顺序保持堆，必须调用**pop_heap( )**来完成这个运算。

```
void sort_heap(RandomAccessIterator first,
  RandomAccessIterator last);
void sort_heap(RandomAccessIterator first,
  RandomAccessIterator last,
  StrictWeakOrdering binary_pred);
```

可以将这些算法完成的工作想像为**make_heap( )**的补充。它使一个以堆顺序排列的序列，转化成普通的排列顺序，这样它就不再是一个堆。这意味着如果调用**sort_heap( )**，将不能再在这个序列范围上使用**push_heap( )**或**pop_heap( )**。（当然，你可以使用这些函数，但不会完成任何有意义的工作。）这是个不稳定的排序。

405

6.3.10 对某一范围内的所有元素进行运算

这些算法遍历整个范围并对每个元素执行运算。它们在利用运算的结果方面有所不同：

for_each( )丢弃运算的返回值，而**transform( )**将每个运算的结果放入一个目的序列（也可以是原始序列）。

```
UnaryFunction for_each(InputIterator first, InputIterator
  last, UnaryFunction f);
```

在该算法中，对 **[first,last)** 中的每个元素应用函数对象**f**，丢弃每个个别的**f**应用的返回值。如果**f**仅是一个函数指针，说明这是典型的与返回值无关；但是，如果**f**是一个保留某些内部状态的对象，则该对象可以捕获一个返回值，它们结合到一起应用到该范围上。**for_each( )**的最终返回值是**f**。

```
OutputIterator transform(InputIterator first, InputIterator
  last, OutputIterator result, UnaryFunction f);
OutputIterator transform(InputIterator1 first,
  InputIterator1 last, InputIterator2 first2,
  OutputIterator result, BinaryFunction f);
```

这些算法，如同**for_each( )**一样，**transform( )**对范围 **[first,last)** 中的每个元素应用函数对象**f**。但是，不是丢弃每次函数调用的结果，**transform( )**而是将结果复制（使用**operator=**）到***result**，每次复制后增加**result**的内容。（**result**指向的序列必须有足够的存储空间；否则，用一个插入符强迫插入来代替赋值。）

transform( )的第1种形式仅调用了**f(*first)**，在这里第1个范围表示一个输入序列。类似地，第2种形式调用**f(*first1,*first2)**。（注意，第2个输入范围的长度由第1个输入范围的长度决定。）这两种情况的返回值都是超越末尾的迭代器，该迭代器指出结果输出范围。

406

程序举例

因为对容器中的对象做的大部分工作是对所有这些对象应用某个运算，这些都是相当重要的算法，值得为此给出一些例证。

首先，分析**for_each( )**。它扫描整个范围，依次提取每个元素并把它作为一个参数进行传递，如同调用的任何被授予的函数对象一样。因此，**for_each( )**执行那些由用户编写的规范的运算。如果想在编译器的头文件中查看**for_each( )**的模板定义，将会看到下述编码：

```
template<class InputIterator, class Function>
Function for_each(InputIterator first, InputIterator last,
                  Function f) {
    while(first != last)
      f(*first++);
    return f;
}
```

下面的例子显示了一些能够扩展这个模板的几种方法。首先，需要一个保持追踪它的对象的类，这样我们就可以知道这些对象被适当地销毁掉：

```
//: C06:Counted.h
// An object that keeps track of itself.
#ifndef COUNTED_H
#define COUNTED_H
#include <vector>
#include <iostream>

class Counted {
  static int count;
  char* ident;
public:
  Counted(char* id) : ident(id) { ++count; }
```

```
  ~Counted() {
    std::cout << ident << " count = "
              << --count << std::endl;
  }
};

class CountedVector : public std::vector<Counted*> {
public:
  CountedVector(char* id) {
    for(int i = 0; i < 5; i++)
      push_back(new Counted(id));
  }
};
#endif // COUNTED_H ///:~

//: C06:Counted.cpp {O}
#include "Counted.h"
int Counted::count = 0;
///:~
```

407

class Counted对已创建的**Counted**对象的个数保存一个静态的计数，并且当这些对象被销毁时通知用户[⊖]。另外，每个**Counted**对象保存一个**char***标识符以便追踪输出更加容易。

CountedVector由**vector<Counted*>**派生而来，并且在构造函数中创建一些**Counted**对象，处理每个想要的**char***。**CountedVector**使测试相当简单，如下所示：

```
//: C06:ForEach.cpp {-mwcc}
// Use of STL for_each() algorithm.
//{L} Counted
#include <algorithm>
#include <iostream>
#include "Counted.h"
using namespace std;

// Function object:
template<class T> class DeleteT {
public:
  void operator()(T* x) { delete x; }
};

// Template function:
template<class T> void wipe(T* x) { delete x; }

int main() {
  CountedVector B("two");
  for_each(B.begin(), B.end(), DeleteT<Counted>());
  CountedVector C("three");
  for_each(C.begin(), C.end(), wipe<Counted>);
} ///:~
```

408

显然，在这里有一些事情需要反复多次去做，既然这样，为什么不创建一个算法用**delete**来删除容器中所有的指针呢？可以使用**transform( )**来完成这项工作。**transform( )**优于**for_each( )**的地方在于**transform( )**将调用函数对象的结果赋给结果范围，该结果范围实际上是输入范围。这种情况意味着对输入范围的序列进行逐字的转换，因为每个元素是原先值的一个修改。在本例中这个方法尤其有用，因为在对每个指针调用**delete**后，为其赋安全的零值更加适合。**transform( )**可以很容易地做到这些：

⊖ 在这个例子中我们忽略了拷贝构造函数和赋值操作，因为没有使用它们。

```
//: C06:Transform.cpp {-mwcc}
// Use of STL transform() algorithm.
//{L} Counted
#include <iostream>
#include <vector>
#include <algorithm>
#include "Counted.h"
using namespace std;

template<class T> T* deleteP(T* x) { delete x; return 0; }

template<class T> struct Deleter {
  T* operator()(T* x) { delete x; return 0; }
};

int main() {
  CountedVector cv("one");
  transform(cv.begin(), cv.end(), cv.begin(),
    deleteP<Counted>);
  CountedVector cv2("two");
  transform(cv2.begin(), cv2.end(), cv2.begin(),
    Deleter<Counted>());
} ///:~
```

这里显示了两种方法：使用模板函数或模板化的函数对象。在调用**transform( )**后，**vector**包含5个空指针，它更安全因为用它对任何副本**delete**都是无效的。

409 有一件事不能做，那就是在遍历中**delete**每个指针，而没有在函数或对象内部封装对**delete**的调用。即如下面所做：

```
for_each(a.begin(), a.end(), ptr_fun(operator delete));
```

这与前面的**destroy( )**调用有相同的问题：**operator delete( )**获取一个**void***，但是迭代器并不是指针。甚至如果要对它进行编译，得到的将是一系列用来释放存储空间的函数调用。不能得到对**a**中每个指针都调用**delete**的效果，然而不会调用析构函数。这显然不是想要的结果，所以需要封装对**delete**的调用。

在前面的**for_each( )**的例子中，忽略了算法的返回值。这个返回值是传递给**for_each( )**的函数。如果这个函数仅是指向一个函数的指针，该返回值并不是很有用，但如果它是一个函数对象那就完全不同了，这个函数对象可能含有内部的成员数据，可以使用这些成员数据来积累关于在**for_each( )**中见到的所有对象的信息。

例如，考虑一个简单的存货清单模型。每个**Inventory**对象包含它所代表的产品类型（在这里用单个的字符表示产品项目名称）、该产品的数量以及每种产品的价格。

```
//: C06:Inventory.h
#ifndef INVENTORY_H
#define INVENTORY_H
#include <iostream>
#include <cstdlib>
using std::rand;

class Inventory {
  char item;
  int quantity;
  int value;
public:
  Inventory(char it, int quant, int val)
  : item(it), cuantity(quant), value(val) {}
  // Synthesized operator= & copy-constructor OK
```

```
  char getItem() const { return item; }
  int getQuantity() const { return quantity; }
  void setQuantity(int q) { quantity = q; }
  int getValue() const { return value; }
  void setValue(int val) { value = val; }
  friend std::ostream& operator<<(
    std::ostream& os, const Inventory& inv) {
    return os << inv.item << ": "
      << "quantity " << inv.quantity
      << ", value " << inv.value;
  }
};

// A generator:
struct InvenGen {
  Inventory operator()() {
    static char c = 'a';
    int q = rand() % 100;
    int v = rand() % 500;
    return Inventory(c++, q, v);
  }
};
#endif // INVENTORY_H ///:~
```

410

成员函数取得产品项目的名称，取得并确定相应的数量和价格。**operator<<**向**ostream**打印出**Inventory**对象。用一个发生器来创建这些对象，这些对象含有顺序标记的产品项目名及随机的数量和价格。

为了找出产品项目的总数及全部价值，可以使用含有总计数据成员的**for_each( )**来创建一个函数对象：

```
//: C06:CalcInventory.cpp
// More use of for_each().
#include <algorithm>
#include <ctime>
#include <vector>
#include "Inventory.h"
#include "PrintSequence.h"
using namespace std;

// To calculate inventory totals:
class InvAccum {
  int quantity;
  int value;
public:
  InvAccum() : quantity(0), value(0) {}
  void operator()(const Inventory& inv) {
    quantity += inv.getQuantity();
    value += inv.getQuantity() * inv.getValue();
  }
  friend ostream&
  operator<<(ostream& os, const InvAccum& ia) {
    return os << "total quantity: " << ia.quantity
              << ", total value: " << ia.value;
  }
};

int main() {
  vector<Inventory> vi;
  srand(time(0));  // Randomize
  generate_n(back_inserter(vi), 15, InvenGen());
```

411

```
  print(vi.begin(), vi.end(), "vi");
  InvAccum ia = for_each(vi.begin(),vi.end(), InvAccum());
  cout << ia << endl;
} ///:~
```

InvAccum的**operator( )**有一个参数，这是**for_each( )**要求的。当**for_each( )**遍历某个范围时，获取该范围的每一个对象并将其传递给**InvAccum::operator( )**，它执行计算并保存结果。在这个处理的最后，**for_each( )**返回**InvAccum**对象，并打印该**InvAccum**对象。

使用**for_each( )**可以对**Inventory**对象做很多事。例如，**for_each( )**可以方便地将所有产品的价格增加10%。但是读者会注意到**Inventory**对象没有办法改变**item**的值。设计**Inventory**的程序员认为这是一个好的主意。毕竟，为什么想要改变一个商品的名称？但是在市场上的交易已经决定了要将所有的产品名称改为大写，使得它们看上去与“新的、改进的”产品一样。他们已经做了调研并且决定用新的产品名称来进行促销（好了，为了市场上的交易总需要做一些事情……）。所以这里不使用**for_each( )**，而是使用**transform( )**：

```
//: C06:TransformNames.cpp
// More use of transform().
#include <algorithm>
#include <cctype>
#include <ctime>
#include <vector>
#include "Inventory.h"
#include "PrintSequence.h"
using namespace std;

struct NewImproved {
  Inventory operator()(const Inventory& inv) {
    return Inventory(toupper(inv.getItem()),
      inv.getQuantity(), inv.getValue());
  }
};

int main() {
  vector<Inventory> vi;
  srand(time(0));  // Randomize
  generate_n(back_inserter(vi), 15, InvenGen());
  print(vi.begin(), vi.end(), "vi");
  transform(vi.begin(),vi.end(),vi.begin(),NewImproved());
  print(vi.begin(), vi.end(), "vi");
} ///:~
```

412

注意，结果范围与输入范围相同；即，在适当的位置执行转换。

现在假设销售部门需要产生一个特价清单，对每种商品有不同的折扣。原始的清单必须原样保留，并且需要产生任意数量的特价清单。销售部门将为每个新清单提供一个单独的折扣明细表。为了解决这个问题，这里使用**transform( )**的第2种版本：

```
//: C06:SpecialList.cpp
// Using the second version of transform().
#include <algorithm>
#include <ctime>
#include <vector>
#include "Inventory.h"
#include "PrintSequence.h"
using namespace std;

struct Discounter {
  Inventory operator()(const Inventory& inv,
```

```
    float discount) {
    return Inventory(inv.getItem(), inv.getQuantity(),
      int(inv.getValue() * (1 - discount)));
  }
};
struct DiscGen {
  float operator()() {
    float r = float(rand() % 10);
    return r / 100.0;
  }
};

int main() {
  vector<Inventory> vi;
  srand(time(0));  // Randomize
  generate_n(back_inserter(vi), 15, InvenGen());
  print(vi.begin(), vi.end(), "vi");
  vector<float> disc;
  generate_n(back_inserter(disc), 15, DiscGen());
  print(disc.begin(), disc.end(), "Discounts:");
  vector<Inventory> discounted;
  transform(vi.begin(),vi.end(), disc.begin(),
    back_inserter(discounted), Discounter());
  print(discounted.begin(), discounted.end(),"discounted");
} ///:~
```

413

给定一个**Inventory**对象和一个折扣比率，**Discounter**函数对象产生一个新的含折扣价格的**Inventory**对象。**DiscGen**函数对象仅产生随机的从1%到10%之间的折扣值用来进行测试。在**main( )**中创建两个**vector**，一个用于**Inventory**，一个用于折扣。将它们随同**Discounter**对象传递给**transform( )**、**transform( )**填充一个新的称为**discounted**的**vector<Inventory>**对象。

6.3.11 数值算法

这些算法都包含在头文件**<numeric>**中，因为它们主要用来执行数值计算。

```
T accumulate(InputIterator first, InputIterator last, T
  result);
T accumulate(InputIterator first, InputIterator last, T
  result, BinaryFunction f);
```

第1种形式是一般化的合计，对于由迭代器**i**指向的 **[first,last)** 中的每一个元素，执行运算**result=result+*i**，在这里**result**是**T**类型。但是，第2种形式更普遍；它对于范围中从头至尾的每一个元素***i**应用函数**f(result,*i)**。

414

注意**transform( )**的第2种形式和**accumulate( )**的第2种形式之间的相似之处。

```
T inner_product(InputIterator1 first1, InputIterator1
  last1, InputIterator2 first2, T init);
T inner_product(InputIterator1 first1, InputIterator1
  last1, InputIterator2 first2, T init, BinaryFunction1
  op1, BinaryFunction2 op2);
```

在这些算法中，计算两个范围 **[first1,last1)** 和 **[first2,first2+(last1-first1))** 的一个广义内积。用第1个序列中的元素乘以第2个序列中“平行的”元素并对其积进行累加来产生返回值。因此，如果有两个序列 **{1,1,2,2}** 和 **{1,2,3,4}**，内积是

```
(1*1) + (1*2) + (2*3) + (2*4)
```

返回结果为17。参数**init**是内积的初始值——可能是0也可能是任何值，这对于空的第1个序列

尤其重要，因为它是默认的返回值。第2个序列必须至少要含有与第1个序列一样多的元素。

第2种形式对它的序列应用一对函数。函数**op1**用来代替加法，而函数**op2**用来代替乘法。因此，如果对上面的序列应用**inner_product( )**的第2种版本，结果会是下面的这些运算：

```
init = op1(init, op2(1,1));
init = op1(init, op2(1,2));
init = op1(init, op2(2,3));
init = op1(init, op2(2,4));
```

所以，这与**transform( )**相似，但用执行两个运算来代替一个运算。

```
OutputIterator partial_sum(InputIterator first,
  InputIterator last, OutputIterator result);
OutputIterator partial_sum(InputIterator first,
InputIterator last, OutputIterator result,
BinaryFunction op);
```

415

这些算法计算一个广义部分和。创建一个新的在**result**开始的序列。新序列中每个元素都是**[first,last)** 范围中从第1个元素到当前选择的元素之间所有元素的累加和。例如，如果原始序列是 **{1,1,2,2,3}**，产生的结果序列是 **{1,1+1,1+1+2,1+1+2+2,1+1+2+2+3}**，即**{1,2,4,6,9}**。

在第2种版本中，使用二元函数**op**代替 + 运算符，取得累计到那个点的所有的“合计”，并且把它与新值结合起来。例如，对上面的序列使用**multiplies<int>( )**（一种乘法op）作为对象，输出结果是**{1,1,2,4,12}**。注意，在输入/输出两个序列中，第1个输出结果值始终与第1个输入值相同。

返回值指向输出范围**[result,result+(last-first))** 的末尾。

```
OutputIterator adjacent_difference(InputIterator first,
  InputIterator last, OutputIterator result);
OutputIterator adjacent_difference(InputIterator first,
  InputIterator last, OutputIterator result, BinaryFunction
  op);
```

这些算法计算全部范围 **[first,last)** 中的相邻元素的差。这意味着在新序列中，每个元素的值是原始序列中当前元素与前面的元素的差值（第1个值不变）。例如，如果原始序列是**{1,1,2,2,3}**，结果序列是 **{1,1-1,2-1,2-2,3-2}**，即**{1,0,1,0,1}**。

第2种形式使用二元函数**op**代替‘－’运算符执行“求差”。例如，如果对序列使用**multiplies<int>( )**作为函数对象（即用“乘法”代替“减法”），输出结果是 **{1,1,2,4,6}**。

返回值指向输出范围 **[result,result+(last-first))** 的末尾。

416

程序举例

这个程序在整型数组上测试**<numeric>**头文件中所有的算法的两种形式。读者将会注意到，在程序例子提供的函数或函数群的形式测试中，这些函数对象被使用的形式是一致的，而使用形式一致则产生相同的结果，因此结果是完全相同的。这里也演示了更加清晰的运算，该运算继续下去就是如何替换用户自己的运算。

```
//: C06:NumericTest.cpp
#include <algorithm>
#include <iostream>
#include <iterator>
#include <functional>
```

```
#include <numeric>
#include "PrintSequence.h"
using namespace std;

int main() {
  int a[] = { 1, 1, 2, 2, 3, 5, 7, 9, 11, 13 };
  const int ASZ = sizeof a / sizeof a[0];
  print(a, a + ASZ, "a", " ");
  int r = accumulate(a, a + ASZ, 0);
  cout << "accumulate 1: " << r << endl;
  // Should produce the same result:
  r = accumulate(a, a + ASZ, 0, plus<int>());
  cout << "accumulate 2: " << r << endl;
  int b[] = { 1, 2, 3, 4, 1, 2, 3, 4, 1, 2 };
  print(b, b + sizeof b / sizeof b[0], "b", " ");
  r = inner_product(a, a + ASZ, b, 0);
  cout << "inner_product 1: " << r << endl;
  // Should produce the same result:
  r = inner_product(a, a + ASZ, b, 0,
    plus<int>(), multiplies<int>());
  cout << "inner_product 2: " << r << endl;
  int* it = partial_sum(a, a + ASZ, b);
  print(b, it, "partial_sum 1", " ");
  // Should produce the same result:
  it = partial_sum(a, a + ASZ, b, plus<int>());
  print(b, it, "partial_sum 2", " ");
  it = adjacent_difference(a, a + ASZ, b);
  print(b, it, "adjacent_difference 1"," ");
  // Should produce the same result:
  it = adjacent_difference(a, a + ASZ, b, minus<int>());
  print(b, it, "adjacent_difference 2"," ");
} ///:~
```

417

注意，**inner_product( )**和**partial_sum( )**的返回值是结果序列的超越末尾的迭代器，因此作为**print( )**函数中的第2个迭代器。

因为每个函数的第2种形式允许用户提供自己的函数对象，所以仅函数的第1种形式是纯“数值的”。读者可以用**inner_product( )**做很多能想得到的非直观数值的事情。

6.3.12 通用实用程序

最后，这里还有与其他的算法一起使用的一些基本工具；用户自己可能会，也可能不会直接使用这些工具。

```
(Templates in the <utility> header)
template<class T1, class T2> struct pair;
template<class T1, class T2> pair<T1, T2>
  make_pair(const T1&, const T2&);
```

在本章前面描述并使用过这些工具。**pair**是一个简单的将两个对象（可能不同类型的对象）封装成一个对象的方法。当需要从一个函数返回多个对象时使用它是很典型的情况，但是也可以用来创建一个持有**pair**对象的容器，或将多个对象作为一个参数进行传递。通过**p.first**和**p.second**来访问指定的元素，这里**p**是**pair**对象。例如，本章中描述的**equal_range( )**函数，作为迭代器的**pair**来返回结果。可以直接**insert( )**一个**pair**到**map**或**multimap**中；对于这些容器来说**pair**是**value_type**。

如果想“在执行中”创建一个**pair**，典型的方法是使用模板函数**make_pair( )**，而不是显式地构造一个**pair**对象。**make_pair( )**会自动推断出它接收到的参数的类型，这样即减轻程序员打字的负担，也增加了程序的健壮性。

```
(From <iterator>)
difference_type distance(InputIterator first, InputIterator
last);
```

418 该算法计算**first**与**last**之间的元素个数。更准确地说，它返回一个整数值，这个整数表示在**frist**等于**last**之前它必须增加的次数。在这一处理过程中不会发生解析迭代器的现象。

(From<iterator>)

根据**n**的值前向移动迭代器**i**的位置。(如果迭代器是双向的，也可以根据**n**的负值向后移动。)这个算法意识到，对不同类型的迭代器应该采用不同的方法，而它使用的都是最有效的方法。例如，对随机迭代器可以用普通的算术（**i+=n**）直接增加，而双向迭代器必须增加**n**次。

```
(From <iterator>)
back_insert_iterator<Container>
  back_inserter(Container& x);
front_insert_iterator<Container>
  front_inserter(Container& x);
insert_iterator<Container>
  inserter(Container& x, Iterator i);
```

这些函数用来为给定的容器创建迭代器，以便向容器中插入元素，而不是用**operator=**覆盖容器中已存在的元素（这是默认的行为）。每种类型的迭代器对插入使用不同的运算：**back_insert_iterator**使用**push_back( )**，**front_insert_iterator**使用**push_front( )**，而**insert_iterator**使用**insert( )**（因此可以与关联式容器一起使用，而另外两种可以与顺序容器一起使用）。这些细节将在第7章中介绍。

```
const LessThanComparable& min(const LessThanComparable& a,
  const LessThanComparable& b);
const T& min(const T& a, const T& b,
  BinaryPredicate binary_pred);
```

在这些算法中，返回两个参数中较小的一个，或如果两个参数相等则返回第1个参数。第1种版本用**operator<**执行比较，而第2种版本将两个参数传递给**binary_pred**来执行比较。

419

```
const LessThanComparable& max(const LessThanComparable& a,
  const LessThanComparable& b);
const T& max(const T& a, const T& b,
  BinaryPredicate binary_pred);
```

这些算法与**min( )**很像，但是返回两个参数中较大的一个。

```
void swap(Assignable& a, Assignable& b);
void iter_swap(ForwardIterator1 a, ForwardIterator2 b);
```

使用赋值的方法来交换**a**和**b**的值。注意，所有的容器类都使用特化的**swap( )**版本，这比通用的版本要更有效得多。

iter_swap( )函数交换它涉及的两个参数的值。

6.4 创建自己的STL风格算法

只要适应了STL算法的风格，用户就可以开始创建自己的通用算法。因为这些算法符合STL中对所有其他算法的约定，使用自己编写的通用算法对熟悉STL的程序员来说更加容易，因此这也成为“扩展STL词汇表”的一种方式。

解决这个问题最容易的方法是在头文件**<algorithm>**中，找到那些与所需要的功能相似

的一些算法并将其作为样板，在其后模仿编写自己的代码。[⊖]（事实上，在头文件中所有STL实现都对模板直接提供代码。）

如果仔细观察标准C++库中算法的列表，就可能注意到一个明显的遗漏：没有**copy_if( )**算法。尽管用**remove_copy_if( )**可以完成相同的效果，但这样做相当不方便，因为必须要转化判定条件。（记住，**remove_copy_if( )**仅复制那些不满足判定条件的元素，并有效地删除那些满足判定条件的元素。）

420

读者可能对用编写一个函数对象适配器来完成这项工作感兴趣，在将函数对象适配器传递给**remove_copy_if( )**之前要取消掉那些不满足判定函数的元素，这意味着要通过如下的声明来完成：

```
// Assumes pred is the incoming condition
replace_copy_if(begin, end, not1(pred));
```

这看上去很合理，但是当读者想起使用判定函数时，而该判定函数是一个指向尚未完善的函数的指针，就会看到为什么它不能工作——**not1**期望的是一个能适应的函数对象，而现在不是这样。编写**copy_if( )**的惟一解决方法是从零开始做起。既然从查阅其他复制算法中了解到对输入和输出需要两个单独的迭代器，那么就可以用下面的例子完成这一工作：

```
//: C06:copy_if.h
// Create your own STL-style algorithm.
#ifndef COPY_IF_H
#define COPY_IF_H

template<typename ForwardIter,
  typename OutputIter, typename UnaryPred>
OutputIter copy_if(ForwardIter begin, ForwardIter end,
  OutputIter dest, UnaryPred f) {
  while(begin != end) {
    if(f(*begin))
      *dest++ = *begin;
    ++begin;
  }
  return dest;
}
#endif // COPY_IF_H ///:~
```

注意，**begin**的自增运算不能完整地进入到复制表达式之内。

6.5 小结

本章的目标是给读者一个关于标准模板库中算法实用性的理解。也就是说，使读者知道并能足够轻松地了解STL，这样就可以在符合C++规则的基础上开始使用它（或者至少考虑使用它，这样一来，读者就会回到这里并寻找合适的解决方法。）STL是强大的，不仅因为它是合理且完全的工具库，而且因为它提供了考虑问题解决方案的词汇表，它也是创建附加工具的框架。

421

尽管本章给出了一些创建用户自己的工具的例子，但还没进入到完全理解STL的所有细微之处所必需的理论深度。一旦进入这样的理论深度，读者就会创建出比已经介绍过的例子更加复杂的工具。遗漏这些内容的部分原因是本教材篇幅的限制，但大部分原因是因为它已经超出了本教材对该章的要求——在这里，我们的目标是给读者一个实用性的理解，以便使读者一天一天地逐步改进自己的编程技巧。

⊖ 当然，没有违反任何版权保护法律。

有大量的书籍专门讲解STL（在附录中列出了它们），但是作者在这里特别推荐Scott Meyers的《Effective STL》(Addison Wesley,2002)。

6.6 练习

6-1 创建一个返回**clock( )**（在**<ctime>**头文件中）当前值的发生器。创建一个**list<clock_t>**，并且通过该发生器用**generate_n( )**填充它。在列表中删除任意副本，并且使用**copy( )**把它打印到**cout**。

6-2 使用**transform( )**和**toupper( )**（在**<cctype>**头文件中），编写一个函数调用，将一个字符串全部转化成大写字母。

6-3 创建一个**Sum**函数对象模板，该函数对象模板在调用**for_each( )**时，累加范围内所有的值。

6-4 编写一个回文构词法发生器，以一个单词作为命令行参数，并且产生所有可能的字母排列。

6-5 编写一个“句子回文构词法发生器”，以一个句子作为命令行参数，并且产生所有可能的句子中单词的排列。（不要落下某些单词，仅是将它们前后左右移动。）

6-6 用基类**B**和派生类**D**创建一个类层次结构关系。在**B**中放入一个**virtual**成员函数**void f( )**，这样它将打印一个显示**B**中**f( )**被调用的消息，且为**D**重新定义这个函数来打印一个不同的信息。创建一个**vector<B*>**，并且用**B**和**D**的对象填充它。使用**for_each( )**来为**vector**中的每个对象调用**f( )**。

422

6-7 修改**FunctionObjects.cpp**，以便用**float**代替**int**。

6-8 修改**FunctionObjects.cpp**，模板化测试的主体，这样就能选择要测试的类型。（必须把**main( )**的大部分放入到一个单独的模板函数中。）

6-9 编写一个程序，以一个整数作为命令行参数，并找出它的所有因数。

6-10 编写一个程序，以一个文本文件的名称作为命令行参数。打开这个文件并且每次读入一个单词（提示：使用>>）。将每个词存储到一个**vector<string>**。将所有的词转化成小写，存储它们，删除全部的副本并打印结果。

6-11 编写一个程序，使用**set_intersection( )**找出两个输入文件中共有的所有单词。修改程序，使用**set_symmetric_difference( )**来显示两个输入文件中非共有的单词。

6-12 创建一个程序，在命令行给定一个整数，创建一个向上直到包括命令行数值在内的所有整数的阶乘的“阶乘表”。为了完成这个工作，编写一个发生器来填充**vector<int>**，然后与标准函数对象一起使用**partial_sum( )**。

6-13 修改**CalcInventory.cpp**，使它能找到所有数量小于某个总数的对象。提供这个总数作为命令行参数，并使用**copy_if( )**和**bind2nd( )**来创建小于目标值的数值的集合。

6-14 使用**UrandGen( )**产生100个数。（数的大小没有关系。）找到范围中模23的同余数（意思是当被23除时有相同的余数）。读者手工挑选一个随机数，确定这个数是否在该范围中，这是通过用这个数除以列表中的每一个数并检查结果是否是1来实现的，用手选的这个值替代使用**find( )**进行查找。

6-15 用在弧度制中表示角度的数填充**vector<double>**。使用函数对象组成，产生**vector**中的所有元素的正弦（见**<cmath>**头文件）。

6-16 测试读者所使用的计算机的速度。调用**srand(time(0))**，然后建一个随机数的数组。再次调用**srand(time(0))**，并在第2个数组中生成相同个数的随机数。用**equal( )**来

看两个数组是否相同。（如果你的计算机足够快的话，两次调用**time(o)**将返回相同的值。）如果两个数组内容不相同，对它们进行排序，并使用**mismatch( )**来看看它们到底哪里不相同。如果相同，增加数组的长度并再次测试。 423

6-17 创建一个STL风格的算法**transform_if( )**，它遵循**transform( )**的第1种形式，即仅在满足一元判定函数的对象上执行变换。将不满足判定函数的对象从结果中忽略掉。需要返回一个新的“末端”迭代器。

6-18 创建一个STL风格的**for_each( )**的重载形式算法，它遵循**transform( )**的第2种形式。它用两个输入范围，这样就可以将第2个输入范围的对象传递给一个二元函数，对第1个范围中的每个对象应用这个函数。

6-19 创建一个由**vector<vector<T>>**制造的**Matrix**类模板。用提供给它的一个友元**ostream & operator<<(ostream&,const Matrix&)**来显示矩阵。在可能的地方用STL函数对象创建以下二元运算：**operator+(const Matrix&,const Matrix&)**执行矩阵加法，**operator*(const Matrix&,const vector<int>&)**用一个**vector**乘一个矩阵，**operator*(const Matrix&,const Matrix&)**执行矩阵乘法。（如果忘记了它们的运算规则，可能需要查找一下矩阵运算的数学含义。）使用**int**和**float**测试建立的**Matrix**类模板。

6-20 使用以下的字符

```
"~`!@#$%^&*( )_-+={[]|\:;"'<.>,?/",
```

生成一个密码本，以命令行给定的输入文件作为单词字典文件。不考虑排除非字母的字符，也不考虑单词在字典文件中的语境意义等情况。使每一种字符串的排列映射为一个单词，例如：

```
"='}/%[}]|{*@?!"`,;>&^-~_:$+.#(<\"  apple
"|]\~>#.+%(/-_[`':;=}{*"$^!&?),@<"  carrot
"@=~[].\/<-`>#*)^%+,";&?!_{:|$}("  Carrot
```

等等。

确认在密码本中不存在副本（相同）的密码或单词。使用**lexicographical_compare( )**在密码上执行排序。用密码本把字典文件译成密码。再对编码的字典文件进行解码，并确认得到的解码文件是否与原文件有相同的内容。

6-21 用下面的名字

Jon Brittle
Jane Brittle
Mike Brittle
Sharon Brittle
George Jensen
Evelyn Jensen

424

找到一个为这些人安排婚礼照片的所有可能的方法。

6-22 在区分照片后，每对新娘和新郎都希望所有其他照片上的人们作为来宾一起参加他们的婚礼。例如，如果新娘和新郎（Jon Brittle和Jane Brittle）相邻，找出为照片上的这对新人安排来宾的所有的可能方法。

6-23 一家旅行社想要找出游客们从一个大陆的一端旅行到另一端（贯穿这个大陆）所花费的平均天数。问题是在调查中，一些游客不采用直接的路线，所用的时间往往要比需要的多（这样的例外数据点称为“局外点”）。使用下面的发生器，在一个**vector**上产生旅行

天数。使用**remove_if( )**删除**vector**中的所有的局外点。用**vector**中数据的平均值找出一般要花多长时间才能够完成旅行。

```
int travelTime() {
  // The "outlier"
  if(rand() % 10 == 0)
    return rand() % 100;
  // Regular route
  return rand() % 10 + 10;
}
```

6-24 在对一个已排序的序列范围进行查找时，确定采用**binary_search()**（二分查找）要比**find( )**（顺序查找）有多快。

6-25 某军队想在供选择的服役报名名单中征募新兵。他们已经决定征募那些在1997年报名注册应征的人们，按照出生日期，从年龄最大的开始依次征募直至最年轻的。在**vector**中产生任意数量的人（提供数据成员，如**age**和**yearEnrolled**）。划分**vector**，使那些在1997年登记注册的应征新兵在名单的开始位置，按照从最年轻的到年龄最大的顺序排序，名单中其余的部分按年龄从大到小排序。

6-26 用人口、（海拔）高度和天气等数据成员建一个名为**Town**的**class**。天气由一个**enum**用枚举常量表 **{RAINY,SNOWY,CLOUDY,CLEAR}** 建立。建一个产生**Town**对象的类。生成城镇的名称（采用读者自己起的有意义或者与地域无关的名称都行）或是从互联网上相关网站得来。保证全部的城镇名称是小写字母，并且没有重复。为了简便起见，我们建议保持城镇名称为一个词。为人口、海拔高度和天气字段，创建一个发生器，随机产生天气情况，在范围[100,1 000 000)内的人口及[0,8 000)英尺[⊖]内的海拔。用**Town**对象填充**vector**。把**vector**重新写入一个名为**Towns.txt**的新文件。

425

6-27 有一个生育高峰，导致了每个城镇人口按10%的速度增长。使用**transform( )**更新这些城镇数据，并将数据重新写回文件中。

6-28 用最高和最低人口来查找这些城镇。这个练习对**Town**类实施**operator<**操作。并尝试实现一个函数，该函数当第1个参数小于第2个参数时返回**true**。将它作为所使用算法的判定函数。

6-29 找出所有海拔在2500~3500英尺间的城镇。根据需要对**Town**类执行相关运算。

6-30 现在需要在某个海拔高度的地方建一个飞机场，而场地位置不是问题。整理现有的城镇名单使其没有副本（副本意味着：在相同的100英尺范围中不能有两个海拔。例如这样的类包括 [100,199)、[200,199)等等）。使用**<functional>**中的函数对象，至少用两种不同的方式将名单按升序排列。以降序完成相同的工作。根据需要对**Town**实现相关运算。

6-31 在基于栈的数组中产生一组任意数目的随机数。使用**max_element( )**找到数组中最大的数。将它与数组末尾的数进行交换。找到次最大的数并且放在先前的数之前。持续这样做直到所有的元素都被移动过。当算法结束时，就得到一个排好序的数组。（这就是“选择排序”。）

6-32 编写一个程序，从一个文件中提取电话号码（同时也包括名字以及其他需要的信息），并且将以222开始的电话号码改变为以863开始。同时要保存旧的号码。文件形式如下所示：

⊖ 1英尺=0.305m。

222 8945
756 3920
222 8432
等等。

426

6-33 编写一个程序，给定一个姓氏，找出每个有这个姓氏的人以及他或她的电话号码。使用处理序列范围的算法（例如**lower_bound**、**upper_bound**、**equal_range**等等）。以姓氏作为主关键字，名字作为次关键字进行排序。假定从形式如下的文件中读入姓名及电话号码。（对它们进行排序，先按姓氏排好序，在同姓氏的人们中再按名字排好序。）

John Doe	345 9483
Nick Bonham	349 2930
Jane Doe	283 2819

6-34 给定一个包含类似下面数据的文件，将所有州的首字母省略词提取出来并将其放入一个单独的文件中。（注意，不能为某个数据类型决定该数据所占的行数，数据所占的行数是随机的。）

ALABAMA
AL
AK
ALASKA
ARIZONA
AZ
ARKANSAS
AR
CA
CALIFORNIA
CO
COLORADO

等等。

当完成时，会得到一个含所有州的首字母省略词组成的文件，如下所示：

AL AK AZ AR CA CO CT DE FL GA HI ID IL IN IA KS KY LA ME MD
MA MI MN MS MO MT NE NV NH NJ NM NY NC ND OH OK OR PA
RI SC SD TN TX UT VT VA WA WV WI WY

6-35 创建一个**Employee**类，该类含有两个数据成员：**hours**和**hourlyPay**。**Employee**还含有一个返回雇员薪水的函数**calcSalary( )**。对任意数量的雇员产生随机的小时薪水及工作小时数。用一个**vector<Employee*>** 来存放这些数据。查看一下公司将为这段付薪时期花多少钱。 427

6-36 再次相互比较**sort( )**、**partial_sort( )**和**nth_element( )**函数，请查明如果都需要使用它们，那么使用其中的那一个进行弱排序可以更节省时间。

第7章 通用容器

容器类是一种特定代码重用问题的解决方案。它们是用于创建面向对象程序的构件，使程序内部模块的构建变得非常容易。

一个容器类描述了一个持有其他对象的对象。容器类如此重要以至于它们被认为是早期面向对象语言的基础。比如在Smalltalk中，程序员将编程语言看作一种与类库结合起来的翻译程序，而类库的关键就是容器类的集合。因此，C++编译器的供应商们很自然地也把容器类库包含在编译器中。读者将会注意到，本教材第1卷中以最简单的形式介绍过的**vector**是多么的有用。

就像很多其他早期的C++库一样，早期的容器类库遵循了Smalltalk的*基于对象的层次结构*（object-based hierarchy），这种结构在Smalltalk中工作得很好，但是它在C++中却变得如此笨拙而难以使用。这就需要另外的解决方法。

C++中处理容器是采用基于模板的方式。标准C++库中的容器提供了多种数据结构。这些数据结构可以与标准算法一起很好地工作，来满足常见的软件开发需求。

7.1 容器和迭代器

在解决一个特定的问题时，如果不知道到底需要多少个对象，或这些对象将要维持多长时间，也就不能预先知道怎样存储这些对象。而在程序实际运行前你并不知道要创建多大的存储空间。

在面向对象程序设计中大多数这样的问题解决起来似乎很简单；只须创建对象的另一种类型就可以了。对于存储问题，这种新的对象类型持有其他对象或者是指向这些对象的指针。这种新的对象类型，通常在C++中称为*容器*（在一些语言中也称为*收集器*（collection），每当必需适应放置在它内部的所有对象的需要的时候，容器都会自行扩展。所以不必预先知道容器中将要放入多少个对象；仅需要创建一个容器对象，然后由容器来处理全部细节。

幸运的是，一个好的面向对象编程语言都伴随着一个容器集。在C++中，它就是标准模板库（STL）。在某些库中，人们认为一个好的通用容器应该能够满足所有的需要，而在其他库中（特别是C++中）则针对不同的需要有不同类型的容器：一个**vector**用于高效地访问其中的所有元素，而一个链表**list**则用于高效地在其中的所有位置上进行插入操作，还有更多其他类型的容器，所以人们可以根据自己的需要来选择特定类型的容器。

所有的容器都有某种存入对象和取出对象的方法。将某一对象放进一个容器的方法是十分明显的；可用一个名为“压入”或“增加”或者类似名字的函数。而从容器中检索对象的方法却并不总是明确的。如果这是一个类似数组的实体，比如一个**vector**，可以使用一个索引检索操作符或函数来完成。但是，在很多情况下这样做并没有意义。而且，单一选择函数也有其局限性。如果需要在容器中操纵或者比较一组元素时该怎么办呢？

对于灵活的元素访问的解决方案就是使用*迭代器*，迭代器是一个对象，它的工作就是在容器中挑选元素并将其呈献给迭代器的使用者。作为一个类，迭代器同时也提供了一个抽象层，因此可以将容器的内部实现细节与用来访问容器的代码分隔开来。通过迭代器，容器可以被看作一个序列。迭代器允许遍历一个序列而无需考虑基本结构——即不管它是一个**vector**、一个**list**、一个**set**还是其他结构。如此一来，就提供了这样的灵活性：即使在轻易地改变了底层的数据结构以后，也不会扰乱遍历容器的程序代码。将迭代操作从容器的控制下分隔开来，也

允许同时存在的多重迭代器。

从设计的观点来说，人们实际上想要做的事情不过就是需要一个序列，并能够操纵该序列以解决自己的问题。如果序列的一个类型可以满足所有要求的话，就没有必要使用不同的类型。基于两种原因需要在容器中进行选择。首先，各种容器提供了不同的接口类型和外部行为。**stack**具有与**queue**不同的接口和行为，对于一个**set**或一个**list**而言这也是不同的。其中的某一个容器可能比其他容器能够为问题提供更加灵活的解决方案，或者它能提供传达人们设计意图的更清晰的抽象。其次，不同的容器对于某些操作可能具有不同的效率。比如，在**vector**和**list**之间进行比较，效率就会有所不同。它们都是具有几乎相同的接口和外部行为的简单序列。但是某些操作却可能具有完全不同的代价。在**vector**中对元素的随机访问只是一个时间恒定的操作；无论选择哪一个元素它的时间代价都是一样的。然而，通过遍历的方式对一个**list**中的元素进行随机访问却是一个代价巨大的操作，元素在**list**中的位置越靠后，所需要的时间就越长。另一方面，如果要想向一个序列的中间插入一个元素，使用**list**的代价却比**vector**低。这些操作以及其他操作的效率依赖序列的底层结构。在设计阶段，可能开始时使用一个**list**，后来又在调整性能时转而使用**vector**，或者反过来。使用了迭代器，就使那些只遍历序列的代码与底层序列实现的改变隔离开来。

431

要记住的是，容器仅仅是一个存储对象的储存柜。如果那个储存柜满足了人们所有的要求，或许确实没有必要了解它是如何实现的。如果读者是在那种内在开销来自于其他因素的编程环境中工作的话，一个**vector**和一个**list**之间代价的差别也许就没那么重要了。可能的需要只是序列的一种类型。你甚至可以想像一种“完美”的容器抽象，它可以根据其使用方法自动地调整底层的实现。[⊖]

STL参考文档

如前一章所述，读者也将注意到在本章中并没有包含用于描述每个STL容器的成员函数详尽的文档。虽然本章将描述我们使用到的成员函数，是我们没有给出其他成员函数的完整描述。我们推荐一些关于Dinkumware、Silicon Graphics以及STLPort STL实现的可利用的在线资源。[⊖]

7.2 概述

432

这里是一个使用**set**类模板的例子，一个模拟传统数学集合的容器，该容器不接受重复值。下面创建的**set**与**int**整型数据一起工作：

```
//: C07:Intset.cpp
// Simple use of STL set.
#include <cassert>
#include <set>
using namespace std;

int main() {
  set<int> intset;
  for(int i = 0; i < 25; i++)
    for(int j = 0; j < 10; j++)
      // Try to insert duplicates:
      intset.insert(j);
  assert(intset.size() == 10);
} ///:~
```

⊖ 这是一个State模式的例子，将在第10章介绍。

⊖ 请访问 http://www.dinkumware.com、http://www.sgi.com/tech/stl或 http://www.stlport.org。

成员函数**insert( )**完成所有的工作：它试图插入一个元素并且如果容器中已经存在相同的元素则不予插入。在使用一个集合中涉及的操作通常只限于插入元素和检测集合是否包含要插入的元素。也可以形成一个并集、一个交集或者一个差集，并测试一个集合是否是另一个集合的子集。在这个例子中，值0~9被插入集合25次，但是只有10个惟一的实例被接受。

现在考虑使用**Intset.cpp**的形式并修改它，用以显示包含在一个文档中的单词清单。该解决方案变得非常简单。

```
//: C07:WordSet.cpp
#include <fstream>
#include <iostream>
#include <iterator>
#include <set>
#include <string>
#include "../require.h"
using namespace std;

void wordSet(const char* fileName) {
  ifstream source(fileName);
  assure(source, fileName);
  string word;
  set<string> words;
  while(source >> word)
    words.insert(word);
  copy(words.begin(), words.end(),
    ostream_iterator<string>(cout, "\n"));
  cout << "Number of unique words:"
       << words.size() << endl;
}

int main(int argc, char* argv[]) {
  if(argc > 1)
    wordSet(argv[1]);
  else
    wordSet("WordSet.cpp");
} ///:~
```

433

这里惟一的实质区别在于，集合保存字符串而不是整数。这些单词被从一个文件中取出来，但其他操作与**Intset.cpp**中的类似。该输出不仅显示出所有重复的单词都已经被忽略掉，而且由于**set**的实现方式，这些单词都被自动地排过序。

set是*关联式容器*（associative container）的一个例子，它是标准C++库提供的3种容器之一。下表列出了容器及其分类总结：

分 类	容 器
序列容器	**vector**、**list**、**deque**
容器适配器	**queue**、**stack**、**priority_queue**
关联式容器	**set**、**map**、**multiset**、**multimap**

这些分类表示，针对不同的需要使用不同的模型。序列容器仅将它们的元素线性地组织起来，是最基本的容器类型。对于某些问题，这些序列需要附上某些特殊的属性，这正好是容器适配器要做的事情——它们对诸如队列或者栈的抽象建立模型。关联式容器则基于关键字来组织它们的数据，并允许快速地检索那些数据。

434

标准库中所有的容器都持有存入的对象的*拷贝*，并且根据需要扩展它们的资源，所以这些对象都必须是*可构造拷贝*（copy-constructible）（具有一个可访问的拷贝构造函数）和*可赋值*（assignable）拷贝（具有一个可访问的赋值操作符）的。一个容器与其他容器之间的关键不同

之处在于它们在内存中存储对象的方式和向用户提供什么样的操作。

如读者已经知道的那样，**vector**是一种允许快速随机访问其中元素的线性序列。然而，向类似于**vector**这样排在一起的序列的中间插入一个元素的操作的开销却是很大的，就像对一个数组进行这种操作一样。一个**deque**（双端队列（double-ended-queue），读作“deck”）也允许几乎与**vector**一样快的随机访问，但是当需要分配新的存储空间时速度明显更快，而且很容易在序列的前端和后端加进新的元素。**list**是一个双向链表，所以在其上随机地移动某个元素的代价很高，但却可以用很低的代价向其中任何地方插入元素。因此，**list**、**deque**和**vector**在基本功能上很相似（它们都是线性序列），只是在各种操作的代价上有所不同。在一个程序的开始阶段可以选择它们中的任何一种使用，只在为了调整效率的时候尝试更换为其他容器。

很多问题的解决其实只需要一个像**list**、**deque**或**vector**这样简单的线性序列。所有这3个容器都含有用于向序列尾部插入一个元素的成员函数**push_back( )**（**list**和**deque**还有一个**push_front( )**成员，用于将一个元素插入序列前端）。

但是，如何在一个序列容器中检索存储的元素呢？对于**vector**和**deque**可以使用索引检索操作符**operator[ ]**，但对于**list**这是行不通的。这3种容器都可以使用迭代器来访问元素。每种容器都提供了相应类型的迭代器来访问它的元素。

虽然容器由值来保存对象（也就是说，它们持有对象的全部拷贝），而在某些时候希望容器存储一些指针，这些指针可以指向某一层次结构的对象，这样一来就可以利用类表现出的多态行为。考虑经典的“图形（shape）”例子，在这里所有的图形都有一个共同的操作集，而且拥有不同类型的图形。这里有一个程序例子，它看起来像使用STL **vector**来持有指向在堆中创建的不同类型**Shape**对象的指针：

435

```
//: C07:Stlshape.cpp
// Simple shapes using the STL.
#include <vector>
#include <iostream>
using namespace std;

class Shape {
public:
  virtual void draw() = 0;
  virtual ~Shape() {};
};

class Circle : public Shape {
public:
  void draw() { cout << "Circle::draw" << endl; }
  ~Circle() { cout << "~Circle" << endl; }
};

class Triangle : public Shape {
public:
  void draw() { cout << "Triangle::draw" << endl; }
  ~Triangle() { cout << "~Triangle" << endl; }
};

class Square : public Shape {
public:
  void draw() { cout << "Square::draw" << endl; }
  ~Square() { cout << "~Square" << endl; }
};

int main() {
  typedef std::vector<Shape*> Container;
```

```
  typedef Container::iterator Iter;
  Container shapes;
  shapes.push_back(new Circle);
  shapes.push_back(new Square);
  shapes.push_back(new Triangle);
  for(Iter i = shapes.begin(); i != shapes.end(); i++)
    (*i)->draw();
  // ... Sometime later:
  for(Iter j = shapes.begin(); j != shapes.end(); j++)
    delete *j;
} ///:~
```

436 类**Shape**、**Circle**、**Square**和**Triangle**的创建非常相似。**Shape**是一个抽象基类（因为有纯虚函数指明标记 =**0**），它定义了所有**Shape**类型的接口。派生类覆盖虚函数**virtual draw( )**以实现相应的操作。现在要创建一串不同类型的**Shape**对象，并将它们原封不动存储在一个STL容器内。为方便起见，用类型定义：

```
typedef std::vector<Shape*> Container;
```

为**Shape***的**vector**创建一个别名，而用类型定义：

```
typedef Container::iterator Iter;
```

使用前面定义的别名为**vector<Shape*>::iterator**创建另一个别名。注意，容器类型名必须用于产生合适的迭代器，它被定义为一个嵌套类。虽然存在不同类型的迭代器（前向、双向、随机等等），但它们都拥有同一个基本接口：可以使用++对它们进行增1操作，可以对迭代器解析以便产生它们当前选中的对象，而且可以测试它们以查看是否已经到了序列的末尾。这就是在90%的时间里要做的事情。这也正是前面例子中所做的事情：一个容器被创建以后，它被填入不同类型的**Shape**指针。注意，向上类型转换发生在当**Circle**、**Square**或者**Rectangle**指针被加入**Shapes**容器中去的时候，容器并不知道加入的指针的具体类型，作为替代它只持有**Shape***。一旦指针被装入容器，它就失去了明确的特性而成为了一个匿名的**Shape***。这正是我们想要的：将它们全都投掷进容器，然后再利用多态性把它们挑选出来。

第1个**for**循环创建一个迭代器，并且通过调用容器的**begin( )** 成员函数将其设置为指向序列的开始端。所有的容器都有**begin( )**和**end( )**成员函数，分别用来产生选择序列开始端和超越末尾的迭代器。可以通过确认迭代器不等于通过调用**end( )**函数产生的迭代器的办法来测试操作是否已经完成；不要使用 < 或者 <=。只有 != 和 == 测试方式起作用，所以通常将循环写成如下形式：

```
for(Iter i = shapes.begin(); i != shapes.end(); i++)
```

这条语句的意思是“遍历序列中的每一个元素。”

437 对迭代器做什么才可以产生它所选择的元素呢？可以通过‘*’（这实际上是一个重载了的操作符）解析其引用（请读者思考其他方法）来实现。返回的是容器持有的任何东西。这个容器持有**Shape***，所以这就是***i**所产生的结果。如果希望调用**Shape**的成员函数，必须使用操作符 ->，因此写出下面的一行：

```
(*i)->draw();
```

这将会调用迭代器当前选择的**Shape***的**draw( )**成员函数。这里的括号虽然难看，但却是产生运算符优先级所必需的。

当这些对象已经被销毁或者在其他情况下这些指针被删除时，STL容器并不会自动地为它们包含的指针调用**delete**。如果用**new**在堆中创建了一个对象，并将其指针存放到某个容器中，这个容器不会提示该指针是否同时也存入到了另一个容器，也不会提示它是否指向堆内存

中的开始位置。用户必须始终负责管理自己的堆内存的分配。程序中的最后一行遍历并且删除容器中所有的对象，以便彻底地实施清理工作。处理容器中指针最容易和最安全的办法就是使用智能（smart）指针。要注意的是**auto_ptr**并不能用于这种目的，所以必须在C++标准库以外寻找适当的智能指针。[⊖]

可以修改这个程序中的两行以便改变本例中使用的容器类型。用包含**<list>**来替代包含**<vector>**，并且将第1个**typedef**改写如下：

```
typedef std::list<Shape*> Container;
```

以替代正在使用**vector**。对其他任何地方不做修改。可以这样做不是因为由继承强加了一个接口（在STL只有很少一点继承），而是因为按STL的设计者采用的惯例已经强加了接口，所以可以非常准确地进行这类交换。现在就可以很容易地在**vector**与**list**或是任何其他支持相同接口（语法和语义上均相同）的容器之间进行变换，并且看看对于需求来说哪种容器工作起来最快。 438

7.2.1 字符串容器

在前面的例子中，在**main( )**的最后需要遍历整个的链表，并且用**delete**删除所有的**Shape**指针：

```
for(Iter j = shapes.begin(); j != shapes.end(); j++)
  delete *j;
```

STL容器确保在其自身被销毁时将调用其包含的每个对象的析构函数。然而，指针并没有析构函数，因此用户必须自己用**delete**删除它们。

这里明显看到STL中的一个疏漏：在任何STL容器中都没有自动用**delete**删除它们包含的指针的设施，所以必须人工地自行解决。这表明STL的设计者们似乎认为指针的容器并不是一个有趣的问题，但事实并不是这样的。

由于存在多重成员资格（multiple membership）问题，使得自动删除一个指针成为问题。如果一个容器持有一个指向某个对象的指针，并不表明那个指针就不会在另一个容器中出现。**Trash**指针链表中的一个指向**Aluminum**对象的指针，也可能存在于一个**Aluminum** 指针链表中。如果发生了这种情况，哪个链表负责清理这个对象——即哪个链表“拥有”这个对象呢？

这个问题事实上可以通过在链表中存储对象而不是指针来解决。当链表被销毁的时候似乎它包含的对象也必须被销毁。在这里，当看到创建一个包含**string**型对象的容器时，STL表现出了它的闪光点。下面的例子将每一输入行作为一个**string**型字符串对象存入一个**vector<string>**中：

```
//: C07:StringVector.cpp
// A vector of strings.
#include <fstream>
#include <iostream>
#include <iterator>
#include <sstream>
#include <string>
#include <vector>
#include "../require.h"
using namespace std;

int main(int argc, char* argv[]) {
  const char* fname = "StringVector.cpp";
```

439

⊖ 随着更多的smart指针类型将加入下一个版本的标准，情况将会发生变化。如果想要先了解它们，可以在www.boost.org看到这些智能指针。

```
  if(argc > 1) fname = argv[1];
  ifstream in(fname);
  assure(in, fname);
  vector<string> strings;
  string line;
  while(getline(in, line))
    strings.push_back(line);
  // Do something to the strings...
  int i = 1;
  vector<string>::iterator w;
  for(w = strings.begin(); w != strings.end(); w++) {
    ostringstream ss;
    ss << i++;
    *w = ss.str() + ": " + *w;
  }
  // Now send them out:
  copy(strings.begin(), strings.end(),
    ostream_iterator<string>(cout, "\n"));
  // Since they aren't pointers, string
  // objects clean themselves up!
} ///:~
```

一旦名为**strings**的**vector<string>**被创建，文件中的每一行都作为一个**string**对象被读入并且放入**vector**中：

```
while(getline(in, line))
  strings.push_back(line);
```

对该文件的操作是向其中添加行号。一个**stringstream**对象提供了简便的方法将一个**int**型数字转换为一个用字符表示那个整数的**string**。

因为操作符**operator+**已被重载，所以组合**string**对象是相当容易的。非常合乎情理，迭代器**w**可以解析以便产生一个既可作为右值又可作为左值的字符串。

```
*w = ss.str() + ": " + *w;
```

440 通过迭代器可以反向给容器中的元素进行赋值，这可能会让人感到惊讶，但这就是对STL的精心设计做出的贡献。

因为**vector<string>**包含对象，这里有两件事值得注意。第一，就像前面解释过的那样，不必明确地清理**string**对象。即便将指向这些**string**对象的地址作为指针放入其他容器也一样，显而易见**strings**就是“主链表”，它拥有对那些对象的所有权。

第二，有效地使用对象的动态创建方法，并且还绝不使用**new**或者**delete**！因为已经保存了给予它的对象拷贝，所有这些问题交由**vector**进行处理。因此能够有效地清理编码。

7.2.2 从STL容器继承

那种即刻就能创建出一个元素序列的能力是令人惊异的，这使人们回想起以前为解决这个特殊的问题时花费了多少时间。比如，很多实用的程序都包括这样的功能，将一个文件读进内存，修改该文件，然后再写回到磁盘上。读者也可以用**StringVector.cpp**中的那些功能并将其打包成一个类，以便以后使用。

现在的问题是：在程序设计中，是创建一个**vector**类型的成员对象，还是采用继承的方式派生出一个类似的对象？一般情况下，面向对象设计准则更倾向于使用组合（成员对象）而不是继承，但是有些标准算法盼望有这样一些序列，它们实现某个特殊的接口，因此继承的使用常常是必需的。

```
//: C07:FileEditor.h
// A file editor tool.
#ifndef FILEEDITOR_H
#define FILEEDITOR_H
#include <iostream>
#include <string>
#include <vector>

class FileEditor : public std::vector<std::string> {
public:
  void open(const char* filename);
  FileEditor(const char* filename) { open(filename); }
  FileEditor() {};
  void write(std::ostream& out = std::cout);
};
#endif // FILEEDITOR_H ///:~
```

441

构造函数打开文件，将其读入到**FileEditor**，再用成员函数**write( )**将包含**string**的**vector**写入任何一个输出流**ostream**。注意，在**write( )**中可以使用一个默认的参数作为引用。

其实现相当简单：

```
//: C07:FileEditor.cpp {O}
#include "FileEditor.h"
#include <fstream>
#include "../require.h"
using namespace std;

void FileEditor::open(const char* filename) {
  ifstream in(filename);
  assure(in, filename);
  string line;
  while(getline(in, line))
    push_back(line);
}

// Could also use copy() here:
void FileEditor::write(ostream& out) {
  for(iterator w = begin(); w != end(); w++)
    out << *w << endl;
} ///:~
```

来自**StringVector.cpp**的函数在这里仅被重新打包。这是类进化的常用方法——程序员在开始时创建一个程序来解决某一特殊的应用，然后发现其中有些通用的功能，就可以把它们变成一个类。

现在，那个行号产生程序可以用**FileEditor**重新编写如下：

```
//: C07:FEditTest.cpp
//{L} FileEditor
// Test the FileEditor tool.
#include <sstream>
#include "FileEditor.h"
#include "../require.h"
using namespace std;
int main(int argc, char* argv[]) {
  FileEditor file;
  if(argc > 1) {
    file.open(argv[1]);
  } else {
    file.open("FEditTest.cpp");
  }
  // Do something to the lines...
```

442

```
  int i = 1;
  FileEditor::iterator w = file.begin();
  while(w != file.end()) {
    ostringstream ss;
    ss << i++;
    *w = ss.str() + ": " + *w;
    ++w;
  }
  // Now send them to cout:
  file.write();
} ///:~
```

现在，用于读取文件的操作都在构造函数中了：

```
FileEditor file(argv[1]);
```

(或者在成员函数**open( )**中)，而写操作发生在单独的一行中（默认将数据发送输出到**cout**)：

```
file.write();
```

该程序块被包含进来用以对内存中的文件进行修改。

7.3 更多迭代器

迭代器是为实现通用而做的抽象。它与不同类型的容器一起工作而不必了解那些容器的底层结构。绝大多数容器都支持迭代器，[⊖]所以可以像下面这样：

443

```
<ContainerType>::iterator
<ContainerType>::const_iterator
```

为一个容器创建迭代器类型。每一个容器都有一个起始成员函数**begin( )**以产生指向容器中起始元素的迭代器，和一个末尾成员函数**end( )**用以产生容器的超越末尾的标记迭代器。如果容器是一个**const**（常）容器，则**begin( )**和**end( )**产生**const**（常）迭代器，即不允许更换这些迭代器所指向的元素（因为相应的运算符都是**const**的）。

所有的迭代器都可以在它们的序列中向前移动（通过运算符**operator++**），并且允许使用 == 和 != 进行比较。因此，为在不超出范围的前提下前移一个名为**it**的迭代器，可以进行如下处理：

```
while(it != pastEnd) {
  // Do something
  ++it;
}
```

这里**pastEnd**是由容器的成员函数**end( )**产生的超越末尾的标记。

通过使用解析运算符（**operator***），一个迭代器可用于产生其当前所指的容器元素。这可以有两种形式。如果**it**是一个可以遍历容器的迭代器，并且**f( )**是容器持有的对象类型的一个成员函数，就可以使用两种形式中的任一种形式：

```
(*it).f();
```

或者

```
it->f();
```

了解了这些以后，就可以创建一个可以与任何容器一起工作的模板了。在这里，函数模板**apply( )**为容器中的每个对象调用一个成员函数，它使用一个指向成员函数的指针作为参数进行传递：

⊖ 容器适配器、栈、队列和优先队列不支持迭代器，因为在用户看来它们的行为与序列的行为并不相同。

```
//: C07:Apply.cpp
// Using simple iteration.
#include <iostream>
#include <vector>
#include <iterator>
using namespace std;

template<class Cont, class PtrMemFun>
void apply(Cont& c, PtrMemFun f) {
  typename Cont::iterator it = c.begin();
  while(it != c.end()) {
    ((*it).*f)(); // Alternate form
    ++it;
  }
}

class Z {
  int i;
public:
  Z(int ii) : i(ii) {}
  void g() { ++i; }
  friend ostream& operator<<(ostream& os, const Z& z) {
    return os << z.i;
  }
};

int main() {
  ostream_iterator<Z> out(cout, " ");
  vector<Z> vz;
  for(int i = 0; i < 10; i++)
    vz.push_back(Z(i));
  copy(vz.begin(), vz.end(), out);
  cout << endl;
  apply(vz, &Z::g);
  copy(vz.begin(), vz.end(), out);
} ///:~
```

444

在这里不能使用**operator->**，因为这将导致语句成为:

```
(it->*f)();
```

它将尝试使用迭代器的**operator->***，而该操作符在迭代器类中并未提供。[⊖]

就像在第6章中所看到的那样，可以更容易地使用**for_each( )**或者**transform( )**两者之中任一个函数应用到序列。 445

7.3.1 可逆容器中的迭代器

一个容器也可以是*可逆的*（reversible），这意味着容器可以产生一个从末尾反向移动的迭代器，这些迭代器也可以从容器的起始元素前向移动。所有标准的容器都支持这种双向迭代。

可逆容器拥有成员函数**rbegin( )**（用于产生一个选择了容器末尾的迭代器**reverse_iterator**）和**rend( )**（用于产生一个指向“超越起始”的迭代器**reverse_iterator**）。如果容器为**const**容器，则**rbegin( )**和**rend( )**将会产生**const_ reverse_iterator**。

下面的例子使用**vector**，但该例适用于所有支持迭代操作的容器:

⊖ 它仅仅适用于那些使用了一个（a **T***）指针作为迭代器类型的**vector**的实现，就像STLPort所做的那样。

```
//: C07:Reversible.cpp
// Using reversible containers.
#include <fstream>
#include <iostream>
#include <string>
#include <vector>
#include "../require.h"
using namespace std;

int main() {
  ifstream in("Reversible.cpp");
  assure(in, "Reversible.cpp");
  string line;
  vector<string> lines;
  while(getline(in, line))
    lines.push_back(line);
  for(vector<string>::reverse_iterator r = lines.rbegin();
      r != lines.rend(); r++)
    cout << *r << endl;
} ///:~
```

反向移动遍历一个容器，使用与一个前向移动遍历容器的普通的迭代器时相同的语法。

446 ### 7.3.2 迭代器的种类

标准C++库中的迭代器被归类为若干“种类”以便描述它们的性能。通常对它们的描述顺序是从行为约束最严格的种类到行为功能最强大的种类。

1. 输入迭代器：只读，一次传递

为输入迭代器的预定义实现只有**istream_iterator**和**istreambuf_iterator**，用于从一个输入流**istream**中读取。就像想像的那样，一个输入迭代器仅能对它所选择的每个元素进行一次解析，正如只能对一个输入流的特殊部分读取一次一样。它们只能前向移动。一个专门的构造函数定义了超越末尾的值。总之，输入迭代器可以对读操作的结果进行解析（对每一个值仅解析一次），然后前向移动。

2. 输出迭代器：只写，一次传递

这是对输入迭代器的补充，不过是对写操作而不是读操作。为输出迭代器的预定义实现只有**ostream_iterator**和**ostreambuf_iterator**，用于向一个输出流**ostream**写数据，还有一个一般较少使用的**raw_storage_iterator**。再次强调，它们只能对每个写出的值进行一次解析，并且只能前向移动。对于输出迭代器来说，没有使用超越末尾的值来结束的概念。总之，输出迭代器可以对写操作的值进行解析（对每一个值仅解析一次），然后前向移动。

3. 前向迭代器：多次读／写

前向迭代器包含了输入和输出迭代器两者的所有功能，加上还可以多次解析一个迭代器指定的位置，因此可以对一个值进行多次读/写。顾名思义，前向迭代器只能向前移动。没有专为前向迭代器预定义的迭代器。

4. 双向迭代器：operator--

双向迭代器具有前向迭代器的全部功能，另外它还可以利用自减操作符**operator--**向后一次移动一个位置。由**list**容器中返回的迭代器都是双向的。

5. 随机访问迭代器：类似于一个指针

最后，随机访问迭代器具有双向迭代器的所有功能，再加上一个指针所有的功能（一个指针就是一个随机访问迭代器），除了没有一种“空（null）”迭代器和空指针相对应。基本上可以这样说，一个随机访问迭代器就像一个指针那样可以进行任何操作，包括使用操作符

447

operator[ ]进行索引，加某个整数值到一个指针就可以向前或向后移动若干个位置，或者使用比较运算符在迭代器之间进行比较。

6. 重要性

人们为什么要关心这些分类法？当只需要以一种明确的方式（比如，仅仅是手工编码想要对某个容器中的对象进行所有的操作）来使用容器时，这些分类法通常并不重要。那么，对迭代器进行分类到底有什么用处。迭代器种类在下列场合中就很重要：

1) 使用某些不久就要演示的C++爱好者内置的迭代器类型，或者用户已经“学成毕业”，能够胜任创建自己的迭代器的工作（在本章稍后演示）。

2) 使用STL算法（第6章的主题）。每种算法对其迭代器都有使用场合的要求。在创建用户自己的可重用的算法模板的时候，迭代器种类的知识变得尤其重要，因为自定义算法需要的迭代器种类决定了该算法的灵活性。如果仅要求最基本的迭代器种类（输入或者输出迭代器），这种算法则适合于任何场合（**copy( )**就是这样的一个例子）。

一个迭代器的种类由一个迭代器的层次结构标记类进行标识。类名和迭代器的种类相符合，并且它们之间的派生层次结构反映了它们之间的关系：

```
struct input_iterator_tag {};
struct output_iterator_tag {};
struct forward_iterator_tag :
  public input_iterator_tag {};
struct bidirectional_iterator_tag :
  public forward_iterator_tag {};
struct random_access_iterator_tag :
  public bidirectional_iterator_tag {};
```

类**forward_iterator_tag**仅从**input_iterator_tag**派生，而不是从**output_iterator_ tag**派生，因为在使用前向迭代器的算法中需要超越末尾的迭代器值，但是使用输出迭代器的那些算法总是假定运算符**operator***是可以解析的。为了这个原因，保证一个超越末尾的值绝不会传递到那些希望使用输出迭代器的算法中是很重要的。 448

为提高效率，某些算法为不同种类的迭代器提供不同的实现，这些迭代器是从由其定义的迭代器标记中推出来的。在本章稍后部分当我们创建自己的迭代器类型时，将会用到这些标记类。

7.3.3 预定义迭代器

STL拥有一个用起来相当便利的预定义迭代器的集合。正如所见到的，对所有基本容器调用**rbegin( )**和**rend( )**可得到**reverse_iterator**对象。

因为某些STL算法需要插入迭代器（insertion iterator）——例如，**copy( )**算法——使用赋值操作符**operator=** 将对象放进目的容器中去。在向容器中填充（fill）而不是覆盖已经存在于目的容器中的那些元素时，当在那里已经没有空间可填充的时候，就会产生问题。插入迭代器所做的事情就是改变运算符**operator=**的实现来替代赋值操作，称为该容器的“压入”或“插入”函数，因此该函数就引起容器分配新的存储空间。**back_insert_iterator**和**front_insert_iterator**两者的构造函数都使用一个基本序列容器对象（**vector**、**deque**或**list**）作为其参数，并产生一个分别调用**push_back( )**或**push_front( )**以进行赋值的迭代器。有益的函数**back_inserter( )**和**front_inserter( )**让程序员在产生这些插入迭代器对象的时候少写一些代码。因为所有的基本序列容器都支持**push_back( )**，读者可能会发现，使用**back_inserter( )**已经成为某种经常性的工作。

insert_iterator能够向一个序列的中间插入元素，再一次代替了**operator=** 的含义，

但是这一次则是自动调用插入函数**insert( )**而不是某个“压入”函数。**insert( )**成员函数需要一个迭代器在插入前指定插入的位置，所以除了容器对象以外，**insert_iterator**还需要这个迭代器。插入器函数**inserter( )**产生相同的对象。

下面的例子演示了不同类型的插入器的使用:

449

```
//: C07:Inserters.cpp
// Different types of iterator inserters.
#include <iostream>
#include <vector>
#include <deque>
#include <list>
#include <iterator>
using namespace std;

int a[] = { 1, 3, 5, 7, 11, 13, 17, 19, 23 };

template<class Cont> void frontInsertion(Cont& ci) {
  copy(a, a + sizeof(a)/sizeof(Cont::value_type),
    front_inserter(ci));
  copy(ci.begin(), ci.end(),
    ostream_iterator<typename Cont::value_type>(
    cout, " "));
  cout << endl;
}

template<class Cont> void backInsertion(Cont& ci) {
  copy(a, a + sizeof(a)/sizeof(Cont::value_type),
    back_inserter(ci));
  copy(ci.begin(), ci.end(),
    ostream_iterator<typename Cont::value_type>(
    cout, " "));
  cout << endl;
}

template<class Cont> void midInsertion(Cont& ci) {
  typename Cont::iterator it = ci.begin();
  ++it; ++it; ++it;
  copy(a, a + sizeof(a)/(sizeof(Cont::value_type) * 2),
    inserter(ci, it));
  copy(ci.begin(), ci.end(),
    ostream_iterator<typename Cont::value_type>(
    cout, " "));
  cout << endl;
}

int main() {
  deque<int> di;
  list<int>  li;
  vector<int> vi;
  // Can't use a front_inserter() with vector
```

450

```
  frontInsertion(di);
  frontInsertion(li);
  di.clear();
  li.clear();
  backInsertion(vi);
  backInsertion(di);
  backInsertion(li);
  midInsertion(vi);
  midInsertion(di);
  midInsertion(li);
} ///:~
```

因为**vector**不支持**push_front( )**，所以它不能产生一个**front_insert_iterator**。然而，可以看到**vector**支持另外两种插入的类型（即便如此，稍后也将看到对于**vector**来说**insert( )**并不是一个高效的操作）。注意，这里用嵌套类型**Cont::value_type**而不是硬编码的**int**类型。

1. 更多的流迭代器

在第6章中，结合**copy( )**函数介绍了流迭代器**ostream_iterator**（输出迭代器）和**istream_iterator**（输入迭代器）的用法。要记住，输出流是没有“结束”这个概念的，因为用户总是在持续地写出更多的元素。然而，输入流却最终会结束（比如，达到了文件末尾），所以需要有一种方法来表现这一点。**istream_iterator**有两个构造函数，一个获得输入流**istream**并且产生一个实际读取的迭代器，另一个是默认构造函数，用于产生一个作为超越末尾的标记的对象。在下面的例子中这个对象被命名为**end**:

```
//: C07:StreamIt.cpp
// Iterators for istreams and ostreams.
#include <fstream>
#include <iostream>
#include <iterator>
#include <string>
#include <vector>
#include "../require.h"
using namespace std;

int main() {
  ifstream in("StreamIt.cpp");
  assure(in, "StreamIt.cpp");
  istream_iterator<string> begin(in), end;
  ostream_iterator<string> out(cout, "\n");
  vector<string> vs;
  copy(begin, end, back_inserter(vs));
  copy(vs.begin(), vs.end(), out);
  *out++ = vs[0];
  *out++ = "That's all, folks!";
} ///:~
```

451

当**in**用完输入时（在这个例子中，是指到达了文件的末尾），**init**与**end**相等，于是**copy( )**终止。

因为**out**是一个**ostream_iterator<string>**，使用运算符**operator=**可以分配任何**string**对象给解析后的迭代器，并且将那个**string**放入输出流中，就像在两个给**out**赋值的操作所做的那样。因为**out**在定义时以一个新行作为其第2个参数，所以这个赋值操作也在每次赋值时插入一个新行。

虽然可能创建一个**istream_iterator<char>**和**ostream_iterator<char>**，但实际上这样做会从语法上分析（parse）输入并且导致诸如自动地吃掉空白字符（空格、制表符和换行符），如果希望用一个输入流的精确地表现这样的动作，是不可取的。另一种方法可以使用特殊的迭代器**istreambuf_iterator**和**ostreambuf_iterator**，它们被设计用来严格地移动字符。[⊖]虽然这些都是模板，但它们都想要使用**char**或者**wchar_t**作为模板参数。[⊜]在下面的例子中，让我们来比较流迭代器和流缓冲迭代器的行为:

⊖ 创建这些迭代器实际上是为了将现场面从输入输出流中抽象出来，从而使现场面能够处理任何字符序列，不仅仅是输入输出流。这样现场允许输入输出流可以轻易地处理一些不同的格式（比如货币符号的表示方式）。

⊜ 对于其他参数类型，用户需要提供一个用于特化的**char_traits**。

```
//: C07:StreambufIterator.cpp
// istreambuf_iterator & ostreambuf_iterator.
#include <algorithm>
#include <fstream>
#include <iostream>
#include <iterator>
#include "../require.h"
using namespace std;

int main() {
  ifstream in("StreambufIterator.cpp");
  assure(in, "StreambufIterator.cpp");
  // Exact representation of stream:
  istreambuf_iterator<char> isb(in), end;
  ostreambuf_iterator<char> osb(cout);
  while(isb != end)
    *osb++ = *isb++; // Copy 'in' to cout
  cout << endl;
  ifstream in2("StreambufIterator.cpp");
  // Strips white space:
  istream_iterator<char> is(in2), end2;
  ostream_iterator<char> os(cout);
  while(is != end2)
    *os++ = *is++;
  cout << endl;
} ///:~
```

452

从语法上来分析，流迭代器使用由**istream::operator>>**来定义，如果读者正在直接从语法上分析字符的话这也许不是你所希望的——要从字符流中将所有的空白字符都去掉的做法相当罕见。事实上读者总希望在使用字符和流的时候使用流缓冲迭代器，而不是使用流迭代器。另外，**istream::operator>>**为每次操作增加了不小的开销，所以它只适合于较高级的操作，比如从语法上分析数字。[⊖]

2. 操纵未初始化的存储区

raw_storage_iterator在**<memory>**中定义，它是一个输出迭代器。它提供了使算法能够将其结果存储到未经初始化的内存的能力。其接口相当简单：构造函数持有一个指向某原始（未初始化）内存储区的迭代器（典型的指针），并且运算符**operator=** 将一个对象分配给那个原始内存。模板参数是输出迭代器的类型和将要被存储的对象类型，输出迭代器指向该原始存储区。这里的例子创建了**Noisy**对象，它们打印出这些对象的构造、赋值以及析构时的跟踪语句（将在稍后介绍**Noisy**类的定义）：

453

```
//: C07:RawStorageIterator.cpp {-bor}
// Demonstrate the raw_storage_iterator.
//{L} Noisy
#include <iostream>
#include <iterator>
#include <algorithm>
#include "Noisy.h"
using namespace std;

int main() {
  const int QUANTITY = 10;
  // Create raw storage and cast to desired type:
  Noisy* np = reinterpret_cast<Noisy*>(
    new char[QUANTITY * sizeof(Noisy)]);
```

⊖ 我们应该感谢Nathan Myers对此的解释。

```
  raw_storage_iterator<Noisy*, Noisy> rsi(np);
  for(int i = 0; i < QUANTITY; i++)
    *rsi++ = Noisy(); // Place objects in storage
  cout << endl;
  copy(np, np + QUANTITY,
    ostream_iterator<Noisy>(cout, " "));
  cout << endl;
  // Explicit destructor call for cleanup:
  for(int j = 0; j < QUANTITY; j++)
    (&np[j])->~Noisy();
  // Release raw storage:
  delete reinterpret_cast<char*>(np);
} ///:~
```

为了能够正确地使用**raw_storage_iterator**模板，原始存储区类型必须与所创建的对象类型相同。这就是为什么来自新**char**数组的指针被类型转换为**Noisy***的原因。赋值操作符使用拷贝构造函数将对象强制存入原始存储区。注意，必须显式地调用析构函数以便进行适当的清理工作，这也允许在操纵容器期间每次删除一个对象。但表达式**delete np**无论如何是无效的，因为在**delete**表达式中的一个静态指针类型，必须与**new**表达式中分配的类型相同。

7.4 基本序列容器：vector、list和deque

454

所有基本序列容器完全按照存进去时的顺序持有对象。然而，对于不同的基本序列容器，它们的操作效率是不同的，因此如果想要操纵具有某种特点的序列，则应当针对不同的操作类型选择合适的容器。到现在为止，本教材中已经使用了**vector**作为精选的容器。并经常在一些示例中应用它。然而，当开始用容器做更复杂的工作时，更多地了解容器的底层实现和行为就变得很重要了，这样就使得程序员能够根据需要做出正确的选择。

7.4.1 基本序列容器的操作

下面的例子用一个模板演示了所有基本序列容器：**vector**、**deque**和**list**所支持的操作：

```
//: C07:BasicSequenceOperations.cpp
// The operations available for all the
// basic sequence Containers.
#include <deque>
#include <iostream>
#include <list>
#include <vector>
using namespace std;

template<typename Container>
void print(Container& c, char* title = "") {
  cout << title << ':' << endl;
  if(c.empty()) {
    cout << "(empty)" << endl;
    return;
  }
  typename Container::iterator it;
  for(it = c.begin(); it != c.end(); it++)
    cout << *it << " ";
  cout << endl;
  cout << "size() "      << c.size()
       << " max_size() " << c.max_size()
       << " front() "    << c.front()
       << " back() "     << c.back()
       << endl;
}
```

455

```
template<typename ContainerOfInt> void basicOps(char* s) {
  cout << "------- " << s << " -------" << endl;
  typedef ContainerOfInt Ci;
  Ci c;
  print(c, "c after default constructor");
  Ci c2(10, 1); // 10 elements, values all 1
  print(c2, "c2 after constructor(10,1)");
  int ia[] = { 1, 3, 5, 7, 9 };
  const int IASZ = sizeof(ia)/sizeof(*ia);
  // Initialize with begin & end iterators:
  Ci c3(ia, ia + IASZ);
  print(c3, "c3 after constructor(iter,iter)");
  Ci c4(c2); // Copy-constructor
  print(c4, "c4 after copy-constructor(c2)");
  c = c2; // Assignment operator
  print(c, "c after operator=c2");
  c.assign(10, 2); // 10 elements, values all 2
  print(c, "c after assign(10, 2)");
  // Assign with begin & end iterators:
  c.assign(ia, ia + IASZ);
  print(c, "c after assign(iter, iter)");
  cout << "c using reverse iterators:" << endl;
  typename Ci::reverse_iterator rit = c.rbegin();
  while(rit != c.rend())
    cout << *rit++ << " ";
  cout << endl;
  c.resize(4);
  print(c, "c after resize(4)");
  c.push_back(47);
  print(c, "c after push_back(47)");
  c.pop_back();
  print(c, "c after pop_back()");
  typename Ci::iterator it = c.begin();
  ++it; ++it;
  c.insert(it, 74);
  print(c, "c after insert(it, 74)");
  it = c.begin();
  ++it;
  c.insert(it, 3, 96);
  print(c, "c after insert(it, 3, 96)");
  it = c.begin();
  ++it;
  c.insert(it, c3.begin(), c3.end());
  print(c, "c after insert("
    "it, c3.begin(), c3.end())");
  it = c.begin();
  ++it;
  c.erase(it);
  print(c, "c after erase(it)");
  typename Ci::iterator it2 = it = c.begin();
  ++it;
  ++it2; ++it2; ++it2; ++it2; ++it2;
  c.erase(it, it2);
  print(c, "c after erase(it, it2)");
  c.swap(c2);
  print(c, "c after swap(c2)");
  c.clear();
  print(c, "c after clear()");
}

int main() {
  basicOps<vector<int> >("vector");
```

456

```
  basicOps<deque<int> >("deque");
  basicOps<list<int> >("list");
} ///:~
```

第1个函数模板，**print( )**，演示了能够从任何序列容器中得到的基本信息：容器是否为空、容器当前大小、容器的最大可能尺寸、起始元素和终止元素等。也可以看到，每一个容器都有成员函数**begin( )**和**end( )**用以返回迭代器。

函数**basicOps( )**检测包括多种构造函数在内的所有其他信息（并且依次调用**print( )**）：默认构造函数、拷贝构造函数、数量和初值、起始和终止迭代器。还有一个用于赋值的**operator=**和两种类型的**assign( )**成员函数。这两个函数其中之一用来获取数量和初始值，而另一个则用来获取起始迭代器和终止迭代器。

所有的基本容器都是可逆容器，就像使用成员函数**rbegin( )**和**rend( )**所演示的一样。一个序列容器可以重置它的大小，而且可以用**clear( )**删除容器中的全部内容（所有元素）。当调用**resize( )**扩展一个序列时，新的元素使用序列内元素类型的默认构造函数，如果它们是内置类型，则使用0作为初始值。

用一个迭代器来指定在任何一个序列容器中想要插入元素的起始位置，可以用**insert( )**插入单个元素，或插入具有相同值的一组元素，或者由一组起始和终止迭代器标识的来自其他容器的一组元素。 457

要用**erase( )**清除序列中间的一个元素，使用一个迭代器；要用**erase( )**清除序列中间的一组元素，使用一对迭代器。注意，因为仅支持双向迭代器，**list**中所有迭代器都只能通过增1或减1来进行移动。（如果容器为可以产生随机访问迭代器的**vector**或者**deque**，**operator+**和**operator-**可以使迭代器移动更大的距离。）

尽管**list**和**deque**支持**push_front( )**和**pop_front( )**，**vector**却不支持，但3者都支持**push_back( )**和**pop_back( )**。

成员函数**swap( )**的命名令人有点疑惑，因为还存在另外一个非成员函数**swap( )**算法用以交换两个相同类型的对象的值。成员函数**swap( )**在两个容器间交换所有东西（如果这两个容器持有相同类型的对象的话），实际上高效地交换了容器本身。它通过交换各个容器的内容来高效地实现交换，这些容器通常存储的是指针。而非成员函数**swap( )**算法通常采用赋值的方式来交换其参数（对于整个容器来说是代价比较高的操作），但是对于标准容器来说，它已经通过模板的特化定制为调用成员函数**swap( )**了。还有一个**iter_swap**算法，使用迭代器来交换同一个容器中的两个元素。

以下部分的内容讨论各种类型的序列容器的特点。

7.4.2 向量

vector类模板被有意地设计成看起来像一个快速的数组，因为它具有数组风格的索引方式，而且它还可以动态地进行扩展。**vector**类模板具有非常基本的用途，以至于早在本教材的前面就用一种很基本的方法介绍过，并在前面的例子中经常使用。这一节将更深入地介绍**vector**。

为了达到最高效地进行索引和迭代，**vector**将其存储内容作为一个连续的对象数组来维护。在理解**vector**的行为时有一个关键点，那就是索引和迭代操作非常快，基本上和在一个对象数组上进行索引和迭代一样快。但是，这也意味着除了在最后一个元素之后插入新元素（即增补新元素）外，向**vector**中插入一个对象是不可以接受的操作。另外，当一个**vector**预分配的存储空间用完以后，为维护其连续的对象数组，它必须在另一个地方重新分配大块新的（更大的）存储空间，并把以前已有的对象拷贝到新的存储空间中去。这种方法造成了一些令 458

人不快的副作用。

1. 已分配存储区溢出的代价

vector从获取到某块存储区开始，就好像一直在进行猜测：程序员将计划放多少对象进去。在放入的对象还没有超出初始存储块所能装载的对象数目的时候，所有的操作都进行得飞快。(如果程序员预先知道有多少个对象的话，就可以使用**reserve( )**预先分配存储区)。但是，在程序运行过程中，最终将会放入过多的对象（超出该存储区的存储范围，即溢出），这时**vector**会做出如下响应：

1) 分配一块新的、更大的存储区。

2) 将旧存储区中的对象拷贝到新开辟的存储区中去（使用拷贝构造函数）。

3) 销毁旧存储区中所有的对象（为每一个对象调用析构函数）。

4) 释放旧存储区的内存。

对于复杂对象，如果经常把**vector**装填得过满的话，系统将会为这些拷贝构造和析构操作的完成付出高昂的代价，这就是为什么**vector**（以及一般的STL容器）被设计成值类型（比如那些容易拷贝的类型）容器的原因。其中也包括指针。

为了观察在填充一个**vector**时会发生什么事情，这儿有一个前面已经提到过的**Noisy**类。它打印出其有关创建、析构、赋值以及拷贝构造的信息：

```
//: C07:Noisy.h
// A class to track various object activities.
#ifndef NOISY_H
#define NOISY_H
#include <iostream>
using std::endl;
using std::cout;
using std::ostream;

class Noisy {
  static long create, assign, copycons, destroy;
  long id;
public:
  Noisy() : id(create++) {
    cout << "d[" << id << "]" << endl;
  }
  Noisy(const Noisy& rv) : id(rv.id) {
    cout << "c[" << id << "]" << endl;
    ++copycons;
  }
  Noisy& operator=(const Noisy& rv) {
    cout << "(" << id << ")=[" << rv.id << "]" << endl;
    id = rv.id;
    ++assign;
    return *this;
  }
  friend bool operator<(const Noisy& lv, const Noisy& rv) {
    return lv.id < rv.id;
  }
  friend bool operator==(const Noisy& lv,const Noisy& rv) {
    return lv.id == rv.id;
  }
  ~Noisy() {
    cout << "~[" << id << "]" << endl;
    ++destroy;
  }
  friend ostream& operator<<(ostream& os, const Noisy& n) {
    return os << n.id;
```

459

```
  }
  friend class NoisyReport;
};

struct NoisyGen {
  Noisy operator()() { return Noisy(); }
};

// A Singleton. Will automatically report the
// statistics as the program terminates:
class NoisyReport {
  static NoisyReport nr;
  NoisyReport() {} // Private constructor
  NoisyReport & operator=(NoisyReport &);  // Disallowed
  NoisyReport(const NoisyReport&);         // Disallowed
public:
  ~NoisyReport() {
    cout << "\n-------------------\n"
         << "Noisy creations: " << Noisy::create
         << "\nCopy-Constructions: " << Noisy::copycons
         << "\nAssignments: " << Noisy::assign
         << "\nDestructions: " << Noisy::destroy << endl;
  }
};
#endif // NOISY_H ///:~
```

460

```
//: C07:Noisy.cpp {O}
#include "Noisy.h"
long Noisy::create = 0, Noisy::assign = 0,
  Noisy::copycons = 0, Noisy::destroy = 0;
NoisyReport NoisyReport::nr;
///:~
```

每个**Noisy**对象都有其标识符，并且设置了一些静态**static**变量用来跟踪所有的创建、赋值（使用运算符**operator=**）、拷贝构造和析构操作。使用计数器**create**中的默认构造函数来初始化**id**；而拷贝构造函数和赋值操作符则通过右值取得它们的**id**值。与运算符**operator=**在一起的左值是一个已经初始化了的对象，所以**id**在被右值覆盖重写以前打印出其原来的**id**值。

为了支持诸如排序和查找（它们被某些容器隐含地使用）操作，**Noisy**必须有运算符**operator<**和**operator==**。这仅仅是比较**id**值。输出流**ostream**插入符遵循常用的形式并且仅打印出**id**值。

类型**NoisyGen**的对象是在检测期间产生**Noisy**对象的函数对象（因为有一个**operator()**）。

NoisyReport是一个单件对象，[⊖]因为我们仅仅要在程序结束时打印一个报告。它有一个私有的**private**构造函数，故不可能创建另外的**NoisyReport**对象，它不允许赋值和拷贝构造，而且有一个静态的名为**nr**的**NoisyReport**实例。在析构函数中只有可执行语句，它们在程序退出和调用静态的析构函数时执行。该析构函数打印出**Noisy**中的所有**static**静态变量所收集的一些统计信息。

461

使用**Noisy.h**，下列程序演示了一个**vector**分配的存储区溢出的情形：

```
//: C07:VectorOverflow.cpp {-bor}
// Shows the copy-construction and destruction
// that occurs when a vector must reallocate.
//{L} Noisy
#include <cstdlib>
#include <iostream>
```

⊖ 单件是一种著名的设计模式，将在第10章中深入介绍。

```
#include <string>
#include <vector>
#include "Noisy.h"
using namespace std;

int main(int argc, char* argv[]) {
  int size = 1000;
  if(argc >= 2) size = atoi(argv[1]);
  vector<Noisy> vn;
  Noisy n;
  for(int i = 0; i < size; i++)
    vn.push_back(n);
  cout << "\n cleaning up " << endl;
} ///:~
```

可以使用默认值1000，也可以通过命令行键入读者自己设定的值。

当运行该程序时，将会看到一个默认构造函数调用（为**n**），然后是很多的拷贝构造函数的调用，接下来是一些析构函数的调用，然后又是很多的拷贝构造函数的调用，等等。当**vector**用光了（超出）为线性数组分配的空间字节时，它必须（维护线性数组中的所有对象，这是它的工作中最重要的部分）获得一块更大的存储空间并且将原来的内容全部移过去，先拷贝然后销毁原来的对象。可以想到，如果存储了大量巨大而复杂的对象，该过程将会很快地变得令人望而却步。

这个问题有两种解决方案。最好的解决方案是要求程序员事先知道到底需要创建多少个对象。在这种情况下，可以使用**reserve( )**来告诉**vector**预分配多大的存储区，这样就避免了所有的拷贝和析构操作，而使得任何事情都可以很快地完成（特别是使用操作符**operator[ ]**来对对象进行随机访问）。注意，使用**reserve( )**预分配存储区与通过给出一个整数作为**vector**构造函数的第1个参数是有区别的；后者将使用元素类型的默认构造函数来初始化被规定的元素个数。

462

通常情况下，程序员并不知道将会需要多少个对象。如果**vector**的重分配操作使程序执行变得迟缓，可以改用其他序列容器。可以使用链表**list**，但是读者将会看到，**deque**允许在序列的两端快速地插入元素，并且在其扩展存储区的时候不需要拷贝和销毁对象的操作。**deque**也允许使用操作符**operator[ ]**进行随机访问，但是没有**vector**的操作符**operator[ ]**执行得那么快。因此，如果在程序的某一处创建所有的对象，并且在另一处随机访问它们，可以先填充一个**deque**，再从该**deque**的基础上创建一个**vector**，用以达到快速索引的目的。不应该按照这种习惯去编程——只要知道有这些问题就行了（也就是说，避免过早地进行性能优化）。

然而，**vector**的内存重分配还会带来更糟的问题。因为**vector**在一个优美简洁的数组中保存它的对象，被**vector**使用的迭代器可以是简单的指针。这很好——在所有的序列容器中，这些指针允许以最快的速度选择和操纵容器内的元素。不论它们是简单的指针，还是一个持有指向其容器内部指针的迭代器对象，考虑当添加了一个额外的对象时，为什么会发生导致**vector**进行重新分配存储区并且将其内容移动到别处去的事情。现在那个迭代器的指针指向了一个未知的地方：

```
//: C07:VectorCoreDump.cpp
// Invalidating an iterator.
#include <iterator>
#include <iostream>
#include <vector>
using namespace std;

int main() {
```

```
  vector<int> vi(10, 0);
  ostream_iterator<int> out(cout, " ");
  vector<int>::iterator i = vi.begin();
  *i = 47;
  copy(vi.begin(), vi.end(), out);
  cout << endl;
  // Force it to move memory (could also just add
  // enough objects):
  vi.resize(vi.capacity() + 1);
  // Now i points to wrong memory:
  *i = 48;  // Access violation
  copy(vi.begin(), vi.end(), out); // No change to vi[0]
} ///:~
```

463

这里举例说明了一个称为*迭代器无效*（iterator invalidation）的概念。某种操作引发了涉及容器底层数据的内部变化，因此在变化之前有效的那些迭代器可能后来都不再有效。如果这个程序正试图打破神秘感，那么在向一个**vector**加入多个对象时，请查看一下持有的迭代器的位置。在向**vector**中加入元素或者使用操作符**operator[ ]**来代替元素选择以后，需要得到一个新迭代器。如果将这种观察结果与向一个**vector**加入新对象所知道的潜在开销结合起来看的话，可以得出下述结论。使用一个**vector**的最安全的方法，就是一次性地填入所有的元素（在理想的情况下，首先应该知道到底需要多少个对象），然后在程序的另一处仅仅使用它（不再加入更多的元素）。这也是本教材到目前为止使用**vector**的方法。标准C++库文档给出了迭代器无效的容器操作。

在前面各章中那些使用**vector**作为“基本”容器的内容中，读者可能已经观察到，也许在所有的情况下不是最佳的选择。这是容器和数据结构理论中的一个基本问题，一般而言——“最佳”选择的改变取决于容器的使用方法。到目前为止，**vector**作为“最佳”选择的理由是它看起来跟一个数组非常相似，因此采用它使读者感到更熟悉和更容易。但是从现在开始，在选择容器时也应该考虑一下有关的其他问题。

2. 插入和删除元素

使用**vector**最有效的条件是：

1) 在开始时用**reserve( )**分配了正确数量的存储区，因此**vector**绝不再重新分配存储区。

2) 仅仅在序列的后端添加或者删除元素。

利用一个迭代器向**vector**中间插入或者删除元素是可能的，但是下面的程序却演示了一个糟糕的想法：

```
//: C07:VectorInsertAndErase.cpp {-bor}
// Erasing an element from a vector.
//{L} Noisy
#include <algorithm>
#include <iostream>
#include <iterator>
#include <vector>
#include "Noisy.h"
using namespace std;

int main() {
  vector<Noisy> v;
  v.reserve(11);
  cout << "11 spaces have been reserved" << endl;
  generate_n(back_inserter(v), 10, NoisyGen());
  ostream_iterator<Noisy> out(cout, " ");
  cout << endl;
```

464

```
  copy(v.begin(), v.end(), out);
  cout << "Inserting an element:" << endl;
  vector<Noisy>::iterator it =
    v.begin() + v.size() / 2; // Middle
  v.insert(it, Noisy());
  cout << endl;
  copy(v.begin(), v.end(), out);
  cout << "\nErasing an element:" << endl;
  // Cannot use the previous value of it:
  it = v.begin() + v.size() / 2;
  v.erase(it);
  cout << endl;
  copy(v.begin(), v.end(), out);
  cout << endl;
} ///:~
```

在运行该程序的时候，可以看到，对预分配函数**reserve( )**的调用实际上仅仅是分配存储区——并没有调用构造函数。对**generate_n( )**的调用非常频繁：每次对**NoisyGen::operator( )**的调用都导致一次构造、一次拷贝构造（加入**vector**）和一个临时对象的析构操作。但是，当一个对象被插入到**vector**的中间位置时，必须向后移动该插入位置后面所有的对象以便维持这个线性数组，而且由于有足够的空间，它通过赋值操作符来实现。（如果**reserve( )**的参数是10而不是11，它就必须重新分配存储区。）当从**vector**中删除一个对象时，再次使用赋值操作符将该删除位置后面所有的元素向前移动以覆盖被删除元素的位置。（注意，这就要求赋值操作符可以正确地清理左值。）最后，数组末端那个对象被删除。

465

7.4.3 双端队列

双端队列**deque**容器是一种优化了的、在序列两端对元素进行添加和删除操作的基本序列容器。它也允许适度快速地进行随机访问——就像**vector**一样，它也有一个**operator[ ]**操作符。然而，它没有**vector**的那种把所有的东西都保存在一块连续的内存块中的约束。**deque**的典型实现是利用多个连续的存储块（同时在一个映射结构中保持对这些块及其顺序的跟踪）。因此，向**deque**的两端添加或删除元素所带来的开销很小。另外，它从不需要在分配新的存储区时复制并销毁所包含的对象（就像**vector**所做的那样），所以在向序列两端添加未知数量的对象时，**deque**远比**vector**更有效率。这意味着，只有在确切知道到底需要多少个对象的时候，**vector**才是最优的选择。另外，在本教材前面的章节所例举的许多使用**vector**和**push_back( )**的程序示例，如果改用**deque**替代的话，可能会更有效率。**deque**的接口和**vector**的接口仅仅有很小的不一致（比如，**deque**拥有**push_front( )**和**pop_front( )**，当使用**vector**的时候就没有），所以将使用**vector**的代码转变为使用**deque**要做的工作是微不足道的。考虑程序**StringVector.cpp**，仅仅需要将程序中所有地方的单词“**vector**”改为“**deque**”，就可以使用**deque**了。下列程序将在**StringVector.cpp**中增加与**vector**操作平行的**deque**操作，并且比较它们的执行时间：

```
//: C07:StringDeque.cpp
// Converted from StringVector.cpp.
#include <cstddef>
#include <ctime>
#include <deque>
#include <fstream>
#include <iostream>
#include <iterator>
#include <sstream>
#include <string>
#include <vector>
```

```
#include "../require.h"
using namespace std;

int main(int argc, char* argv[]) {
  char* fname = "StringDeque.cpp";
  if(argc > 1) fname = argv[1];
  ifstream in(fname);
  assure(in, fname);
  vector<string> vstrings;
  deque<string> dstrings;
  string line;
  // Time reading into vector:
  clock_t ticks = clock();
  while(getline(in, line))
    vstrings.push_back(line);
  ticks = clock() - ticks;
  cout << "Read into vector: " << ticks << endl;
  // Repeat for deque:
  ifstream in2(fname);
  assure(in2, fname);
  ticks = clock();
  while(getline(in2, line))
    dstrings.push_back(line);
  ticks = clock() - ticks;
  cout << "Read into deque: " << ticks << endl;
  // Now compare indexing:
  ticks = clock();
  for(size_t i = 0; i < vstrings.size(); i++) {
    ostringstream ss;
    ss << i;
    vstrings[i] = ss.str() + ": " + vstrings[i];
  }
  ticks = clock() - ticks;
  cout << "Indexing vector: " << ticks << endl;
  ticks = clock();
  for(size_t j = 0; j < dstrings.size(); j++) {
    ostringstream ss;
    ss << j;
    dstrings[j] = ss.str() + ": " + dstrings[j];
  }
  ticks = clock() - ticks;
  cout << "Indexing deque: " << ticks << endl;
  // Compare iteration
  ofstream tmp1("tmp1.tmp"), tmp2("tmp2.tmp");
  ticks = clock();
  copy(vstrings.begin(), vstrings.end(),
    ostream_iterator<string>(tmp1, "\n"));
  ticks = clock() - ticks;
  cout << "Iterating vector: " << ticks << endl;
  ticks = clock();
  copy(dstrings.begin(), dstrings.end(),
    ostream_iterator<string>(tmp2, "\n"));
  ticks = clock() - ticks;
  cout << "Iterating deque: " << ticks << endl;
} ///:~
```

466

467

之所以向**vector**中增加对象时执行效率低下，就是因为存储区的重分配，读者可能期待两者之间产生戏剧性的差别。然而，对于一个有1.7MB的文本文件，由一个编译器编译的程序产生的输出如下（测试结果是被测量操作平台／编译器中特殊的时钟滴答声，不是秒数）：

```
Read into vector: 8350
Read into deque: 7690
```

```
Indexing vector: 2360
Indexing deque: 2480
Iterating vector: 2470
Iterating deque: 2410
```

不同的编译器和操作平台几乎都同意这个结果。得到的结果并没有产生什么戏剧性的变化，难道不是吗？这指出了一些重要的问题：

1) 我们（程序员和作者）对此进行典型的最坏猜测，就是在程序中的某些地方有低效的事件发生。

2) 效率来自各种效果的组合。在这里，读进每一行并将其转换为字符串就可以控制上面**vector**对**deque**的代价对照比较。

3) **string**类在效率方面的设计相当好。

这并不意味着在不确定的对象数将被存入容器末端的时候不用**deque**而用**vector**。正相反，应该用**deque**——特别是在调整程序的性能的时候。同时也要注意到，引起程序性能方面的问题通常并不是在你认为有问题的地方，了解性能瓶颈确切地点的惟一方法就是进行测试。在本章稍后的内容中，读者将会看到**vector**、**deque**和**list**之间更“纯的”性能比较。

7.4.4 序列容器间的转换

有时在程序的某一部分需要某一种容器的行为或效率，而在程序的另一部分则需要不同容器的行为或效率。比如，在向容器中添加对象时需要**deque**的效率，但是在对这些对象进行索引时又需要**vector**的效率。每一个基本序列容器（**vector**、**deque**和**list**）都有一个双迭代器的构造函数（指明了在创建一个新的对象时在序列中读取的起始和终止位置），和一个用于将数据读入一个现存的容器中的成员函数**assign( )**，所以可以很容易地将对象从一个序列容器移到另一个序列容器。

468

下面的例子将对象读入**deque**内，然后将其转换到一个**vector**：

```
//: C07:DequeConversion.cpp {-bor}
// Reading into a Deque, converting to a vector.
//{L} Noisy
#include <algorithm>
#include <cstdlib>
#include <deque>
#include <iostream>
#include <iterator>
#include <vector>
#include "Noisy.h"
using namespace std;

int main(int argc, char* argv[]) {
  int size = 25;
  if(argc >= 2) size = atoi(argv[1]);
  deque<Noisy> d;
  generate_n(back_inserter(d), size, NoisyGen());
  cout << "\n Converting to a vector(1)" << endl;
  vector<Noisy> v1(d.begin(), d.end());
  cout << "\n Converting to a vector(2)" << endl;
  vector<Noisy> v2;
  v2.reserve(d.size());
  v2.assign(d.begin(), d.end());
  cout << "\n Cleanup" << endl;
} ///:~
```

读者可以尝试各种尺寸大小的容器，但是请注意，这其实并没有什么差别——这些对象仅被拷贝构造到新的**vector**中去。有趣的是，在构建**vector**的时候**v1**并不会导致多次内存分配，

不管使用多少元素都是这样。读者可能最初会认为，必须遵循用在**v2**上的过程，预分配内存空间以避免零乱的重新分配，但这没有必要，因为**v1**使用的构造函数早就决定了需要的内存空间。

469

已配置存储区溢出的代价

与**VectorOverflow.cpp**不同，可以更清楚地看到，在使用**deque**的情况下，当一个存储块发生溢出时会发生什么事情。

```
//: C07:DequeOverflow.cpp {-bor}
// A deque is much more efficient than a vector when
// pushing back a lot of elements, since it doesn't
// require copying and destroying.
//{L} Noisy
#include <cstdlib>
#include <deque>
#include "Noisy.h"
using namespace std;

int main(int argc, char* argv[]) {
  int size = 1000;
  if(argc >= 2) size = atoi(argv[1]);
  deque<Noisy> dn;
  Noisy n;
  for(int i = 0; i < size; i++)
    dn.push_back(n);
  cout << "\n cleaning up " << endl;
} ///:~
```

在“cleaning up”输出出现之前，这里有相对较少的（如果有的话）析构函数调用。因为**deque**在多个块中分配所有的存储区，而不是像**vector**一样使用一个类数组的存储区，它从来不需要移动现存的各个数据块的存储区。（因此，就不会有额外的拷贝构造和析构发生。）**deque**仅仅分配一块新存储区。出于相同的原因，**deque**可以高效率地向序列开始端添加元素，因为如果它用完了存储区，它只需（再一次）为序列的开始端分配一个新的存储块。（然而，用于保存数据块索引信息的存储块却有可能需要重新分配。）可是，在一个**deque**的中间插入元素，可能甚至比**vector**更麻烦（但代价不大）。

因为**deque**聪明的存储管理方式，在向**deque**的两端添加元素以后，现存的迭代器都不会失效，就像在演示中对**vector**所做的那样（见**VectorCoreDump.cpp**）。如果坚持**deque**在以下情况下是最好的——从序列的两端插入或删除元素，合理地快速遍历，以及使用**operator[ ]**进行相当快速的随机访问——读者将会形成良好的编程习惯。

7.4.5 被检查的随机访问

470

vector和**deque**都提供了两个随机访问函数：进行索引操作符（**operator[ ]**），这是读者已经看到过的，以及**at( )**，它检测正被索引的容器的边界，如果超出了边界则抛掷出一个异常。使用**at( )**时代价更高一些：

```
//: C07:IndexingVsAt.cpp
// Comparing "at()" to operator[].
#include <ctime>
#include <deque>
#include <iostream>
#include <vector>
#include "../require.h"
using namespace std;

int main(int argc, char* argv[]) {
  long count = 1000;
```

```
  int sz = 1000;
  if(argc >= 2) count = atoi(argv[1]);
  if(argc >= 3) sz = atoi(argv[2]);
  vector<int> vi(sz);
  clock_t ticks = clock();
  for(int i1 = 0; i1 < count; i1++)
    for(int j = 0; j < sz; j++)
      vi[j];
  cout << "vector[] " << clock() - ticks << endl;
  ticks = clock();
  for(int i2 = 0; i2 < count; i2++)
    for(int j = 0; j < sz; j++)
      vi.at(j);
  cout << "vector::at() " << clock()-ticks <<endl;
  deque<int> di(sz);
  ticks = clock();
  for(int i3 = 0; i3 < count; i3++)
    for(int j = 0; j < sz; j++)
      di[j];
  cout << "deque[] " << clock() - ticks << endl;
  ticks = clock();
  for(int i4 = 0; i4 < count; i4++)
    for(int j = 0; j < sz; j++)
      di.at(j);
  cout << "deque::at() " << clock()-ticks <<endl;
  // Demonstrate at() when you go out of bounds:
  try {
    di.at(vi.size() + 1);
  } catch(...) {
    cerr << "Exception thrown" << endl;
  }
} ///:~
```

471

就像在第1章中看到的那样，不同的系统采用不同的方法来处理未捕获的异常，但是在使用**at( )**的时候，你可以通过多种途径知道程序的某一部分发生了错误，可是在使用**operator[ ]**时却可能对此一无所知。

7.4.6 链表

list以一个双向链表数据结构来实现，如此设计是为了在一个序列的任何地方快速地插入或删除元素，对于**vector**和**deque**而言这是一个代价高得多的操作。**list**没有操作符**operator[ ]**，所以当对一个**list**进行随机访问时速度非常之慢。其最适用的场合就是在按顺序从头到尾（反之亦然）遍历一个序列的时候，而不是随机地从序列中间选择某一个元素。尽管那样，其遍历速度与**vector**相比仍然较慢，但如果不做那么多遍历操作的话，那就不会成为影响程序性能的瓶颈。

在**list**中每个链接的内存开销需要为实际对象所在的存储区顶部设置一个前向和反向指针。因此，在有较大的对象需要从**list**中间进行插入或者删除时，**list**是一个较好的选择。

如果想查找对象要频繁地遍历序列，最好不使用**list**，因为遍历是从**list**的开始端——这是惟一能够开始的地方，除非已经得到一个迭代器，它指向已知道的离目标最近的位置——直到找到感兴趣的那个对象，所耗费的时间与该对象到**list**开始端之间的对象数目成比例。

list中的对象在创建以后绝不会移动。“移动”一个**list**的元素意味着改变其链接关系，但绝不会进行拷贝或者对某个实际的对象赋值。这就意味着，在元素被添加到**list**中时，迭代器不会失效，就像较早示例中利用**vector**演示的那样。这里有一个使用**Noisy**对象的**list**的例子：

472

```
//: C07:ListStability.cpp {-bor}
// Things don't move around in lists.
//{L} Noisy
#include <algorithm>
#include <iostream>
#include <iterator>
#include <list>
#include "Noisy.h"
using namespace std;

int main() {
  list<Noisy> l;
  ostream_iterator<Noisy> out(cout, " ");
  generate_n(back_inserter(l), 25, NoisyGen());
  cout << "\n Printing the list:" << endl;
  copy(l.begin(), l.end(), out);
  cout << "\n Reversing the list:" << endl;
  l.reverse();
  copy(l.begin(), l.end(), out);
  cout << "\n Sorting the list:" << endl;
  l.sort();
  copy(l.begin(), l.end(), out);
  cout << "\n Swapping two elements:" << endl;
  list<Noisy>::iterator it1, it2;
  it1 = it2 = l.begin();
  ++it2;
  swap(*it1, *it2);
  cout << endl;
  copy(l.begin(), l.end(), out);
  cout << "\n Using generic reverse(): " << endl;
  reverse(l.begin(), l.end());
  cout << endl;
  copy(l.begin(), l.end(), out);
  cout << "\n Cleanup" << endl;
} ///:~
```

对于**list**，例如像进行逆转和排序这些看起来激进的操作都不需要拷贝对象，那是因为，仅需要改变链接而不是移动对象。然而，要注意**sort( )**和**reverse( )**都是**list**的成员函数，所以它们有**list**内在的特殊知识，能够以再排列元素来代替拷贝它们。另一方面，函数**swap( )**则是一个通用算法，它并不了解有关**list**的特别之处，所以它利用拷贝的方法来进行两个元素的交换。一般情况下，如果系统提供了某个算法的成员版本就使用这个成员版本而不使用其等价的通用算法。特别应当指出，通用的**sort( )**和**reverse( )**算法仅适用于数组、**vector**和**deque**。 473

如果有较大、复杂的对象，就可能要首先选择**list**，特别是如果构造、析构、拷贝构造以及赋值操作的代价巨大，如果要进行大量的像对对象进行排序或以别的方式对它们进行重新排列操作的时候更是这样。

1. 特殊的list操作

list有一些特殊的内置操作使其以最好的方式利用**list**的结构。读者已经看到了**reverse( )**和**sort( )**，这里还有另外一些操作：

```
//: C07:ListSpecialFunctions.cpp
//{L} Noisy
#include <algorithm>
#include <iostream>
#include <iterator>
#include <list>
#include "Noisy.h"
#include "PrintContainer.h"
using namespace std;
```

```
int main() {
  typedef list<Noisy> LN;
  LN l1, l2, l3, l4;
  generate_n(back_inserter(l1), 6, NoisyGen());
  generate_n(back_inserter(l2), 6, NoisyGen());
  generate_n(back_inserter(l3), 6, NoisyGen());
  generate_n(back_inserter(l4), 6, NoisyGen());
  print(l1, "l1", " "); print(l2, "l2", " ");
  print(l3, "l3", " "); print(l4, "l4", " ");
  LN::iterator it1 = l1.begin();
  ++it1; ++it1; ++it1;
  l1.splice(it1, l2);
  print(l1, "l1 after splice(it1, l2)", " ");
  print(l2, "l2 after splice(it1, l2)", " ");
  LN::iterator it2 = l3.begin();
  ++it2; ++it2; ++it2;
  l1.splice(it1, l3, it2);
  print(l1, "l1 after splice(it1, l3, it2)", " ");
  LN::iterator it3 = l4.begin(), it4 = l4.end();
  ++it3; --it4;
  l1.splice(it1, l4, it3, it4);
  print(l1, "l1 after splice(it1,l4,it3,it4)", " ");
  Noisy n;
  LN l5(3, n);
  generate_n(back_inserter(l5), 4, NoisyGen());
  l5.push_back(n);
  print(l5, "l5 before remove()", " ");
  l5.remove(l5.front());
  print(l5, "l5 after remove()", " ");
  l1.sort(); l5.sort();
  l5.merge(l1);
  print(l5, "l5 after l5.merge(l1)", " ");
  cout << "\n Cleanup" << endl;
} ///:~
```

474

在用**Noisy**对象填充了4个**list**之后，一个**list**通过3种方式接合成另一个**list**。首先，整个链表**l2**在迭代器**it1**处被接合为链表**l1**。注意，在接合之后**l2**是空的——该接合意味着从源链表中删除所有对象。第2次接合从链表**l3**中在迭代器**it2**位置开始，将那些元素插入到链表**l1**中从迭代器**it1**处开始的位置。第3次接合从链表**l1**在迭代器**it1**处开始，并且使用了链表**l4**中始于迭代器**it3**终于迭代器**it4**的那些元素。从表面上看对源链表的提及是多余的，这是因为必须将那些将被传输到目的链表的元素从源链表中删除。

演示删除函数**remove( )**的代码输出表明，删除具有特定值的所有元素，链表不必排序。

最后，如果要用**merge( )**合并两个链表，只有确定这些链表是否都已经排过序，合并才有意义。在这种情况下最终得到的是一个包含了两个链表中所有元素并排过序的新链表（源链表已经被删除 —即其所有的元素已经被移动到目的链表中去了）。

一个**unique( )**成员函数将从**list**中删除所有重复的对象，惟一的条件就是首先对**list**进行排序：

```
//: C07:UniqueList.cpp
// Testing list's unique() function.
#include <iostream>
#include <iterator>
#include <list>
using namespace std;
int a[] = { 1, 3, 1, 4, 1, 5, 1, 6, 1 };
const int ASZ = sizeof a / sizeof *a;
int main() {
```

475

```
  // For output:
  ostream_iterator<int> out(cout, " ");
  list<int> li(a, a + ASZ);
  li.unique();
  // Oops! No duplicates removed:
  copy(li.begin(), li.end(), out);
  cout << endl;
  // Must sort it first:
  li.sort();
  copy(li.begin(), li.end(), out);
  cout << endl;
  // Now unique() will have an effect:
  li.unique();
  copy(li.begin(), li.end(), out);
  cout << endl;
} ///:~
```

在这里使用的**list**构造函数采用来自另外一个容器的起始和超越末尾的迭代器，并将那个容器中的所有元素复制到其自己的**list**中。这里，“容器”仅仅是一个数组，而“迭代器”是指向那个数组的指针，但是因为STL的设计，**list**的构造函数可以像使用任何其他容器一样很容易地使用数组。

函数**unique( )**仅仅将毗连的重复元素删除，因此在调用**unique( )**之前需要对序列进行排序。当试图解决的问题为根据当前排列的顺序除去毗连的重复元素的时候是个例外。

在这里还有4个附加的**list**成员函数未被演示：**remove_if( )**获得一个预报，该预报决定了某个元素是否能被删除；**unique( )**获得一个二元的预报以进行惟一性比较；**merge( )**获得一个附加的用于进行比较的参数；**sort( )**获得一个比较器（以提供比较或者覆盖当前存在的那个元素）。

2. 链表与集合

476

看一看前面的那个例子，读者可能已经注意到，如果需要一个没有重复元素并且已排过序的序列，得到结果可以是一个集合**set**。对这两个容器的性能进行比较将会很有帮助：

```
//: C07:ListVsSet.cpp
// Comparing list and set performance.
#include <algorithm>
#include <cstdlib>
#include <ctime>
#include <iostream>
#include <iterator>
#include <list>
#include <set>
#include "PrintContainer.h"
using namespace std;

class Obj {
  int a[20]; // To take up extra space
  int val;
public:
  Obj() : val(rand() % 500) {}
  friend bool
  operator<(const Obj& a, const Obj& b) {
    return a.val < b.val;
  }
  friend bool
  operator==(const Obj& a, const Obj& b) {
    return a.val == b.val;
  }
  friend ostream&
  operator<<(ostream& os, const Obj& a) {
```

```
    return os << a.val;
  }
};

struct ObjGen {
  Obj operator()() { return Obj(); }
};

int main() {
  const int SZ = 5000;
  srand(time(0));
  list<Obj> lo;
  clock_t ticks = clock();
  generate_n(back_inserter(lo), SZ, ObjGen());
  lo.sort();
  lo.unique();
  cout << "list:" << clock() - ticks << endl;
  set<Obj> so;
  ticks = clock();
  generate_n(inserter(so, so.begin()),
    SZ, ObjGen());
  cout << "set:" << clock() - ticks << endl;
  print(lo);
  print(so);
} ///:~
```

477

当运行程序的时候，读者会发现**set**比**list**快得多。这是可靠的——毕竟，**set**的主要工作就是在排过序的序列中只保存独一无二的元素。

这个程序使用了头文件**PrintContainer.h**，其中包含一个函数模板，该函数模板用于将任何序列容器打印到一个输出流。**PrintContainer.h**定义如下：

```
//: C07:PrintContainer.h
// Prints a sequence container
#ifndef PRINT_CONTAINER_H
#define PRINT_CONTAINER_H
#include "../C06/PrintSequence.h"

template<class Cont>
void print(Cont& c, const char* nm = "",
           const char* sep = "\n",
           std::ostream& os = std::cout) {
  print(c.begin(), c.end(), nm, sep, os);
}
#endif ///:~
```

这里定义的模板**print( )**仅调用了在前一章里的**PrintSequence.h**中定义的函数模板**print( )**。

7.4.7 交换序列

我们在前面已经提到过，所有的基本序列容器都有一个成员函数**swap( )**，该函数被设计用来将一个序列转换为另一个序列（但只能用于相同类型的序列）。成员函数**swap( )**利用了它对特定容器内部结构的知识，从而提高了操作的效率。

478

```
//: C07:Swapping.cpp {-bor}
// All basic sequence containers can be swapped.
//{L} Noisy
#include <algorithm>
#include <deque>
#include <iostream>
#include <iterator>
#include <list>
```

```
#include <vector>
#include "Noisy.h"
#include "PrintContainer.h"
using namespace std;
ostream_iterator<Noisy> out(cout, " ");

template<class Cont> void testSwap(char* cname) {
  Cont c1, c2;
  generate_n(back_inserter(c1), 10, NoisyGen());
  generate_n(back_inserter(c2), 5, NoisyGen());
  cout << endl << cname << ":" << endl;
  print(c1, "c1"); print(c2, "c2");
  cout << "\n Swapping the " << cname << ":" << endl;
  c1.swap(c2);
  print(c1, "c1"); print(c2, "c2");
}

int main() {
  testSwap<vector<Noisy> >("vector");
  testSwap<deque<Noisy> >("deque");
  testSwap<list<Noisy> >("list");
} ///:~
```

在运行这个程序时，读者将会发现，每一种类型的序列容器都能在不需要复制或者赋值的情况下将一种序列变换为另一种序列，即使这些序列的尺寸不同。实际上，其所做的是完全地将一个对象的资源交换为另一个对象的资源。

STL算法也包含一个**swap( )**，当该函数应用于两个相同类型的容器时，它使用成员函数**swap( )**来达到快速的性能。所以，如果对容器的一个容器应用**sort( )**算法，读者会发现其执行速度非常快——这表明对容器的一个容器进行快速排序也是设计STL的一个目的。 479

7.5 集合

集合**(set)**容器仅接受每个元素的一个副本。它也对元素排序。（进行排序并不是**set**的概念定义所固有的，但是STL **set**用一棵平衡树数据结构来存储其元素以提供快速的查找，因此在遍历**set**的时候就产生了排序的结果。）在本章的前两个例子中用到了**set**。

考虑为一本书创建索引的问题。有人可能喜欢从书中的所有单词开始创建，但是每个单词只需要一个实例，并且希望它们排过序。对于这个问题，容器**set**是理想的选择，它可以毫不费力地解决这个问题。然而，还存在标点符号和非字母字符的问题，它们必须被去掉以便产生正确的单词。该问题的一个解决方案就是用标准C库函数**isalpha( )**和**isspace( )**提取只需要的字符。可以用空白字符来替换所有不需要的字符，这样就可以很容易地从读入的每一行中提取出合法的单词：

```
//: C07:WordList.cpp
// Display a list of words used in a document.
#include <algorithm>
#include <cctype>
#include <cstring>
#include <fstream>
#include <iostream>
#include <iterator>
#include <set>
#include <sstream>
#include <string>
#include "../require.h"
using namespace std;
```

```
char replaceJunk(char c) {
  // Only keep alphas, space (as a delimiter), and '
  return (isalpha(c) || c == '\'') ? c : ' ';
}
int main(int argc, char* argv[]) {
  char* fname = "WordList.cpp";
  if(argc > 1) fname = argv[1];
  ifstream in(fname);
  assure(in, fname);
  set<string> wordlist;
  string line;
  while(getline(in, line)) {
    transform(line.begin(), line.end(), line.begin(),
              replaceJunk);
    istringstream is(line);
    string word;
    while(is >> word)
      wordlist.insert(word);
  }
  // Output results:
  copy(wordlist.begin(), wordlist.end(),
       ostream_iterator<string>(cout, "\n"));
} ///:~
```

480

调用**transform( )**，以空白字符替换每个要被忽略掉的字符。容器**set**不但忽略重复的单词，而且根据函数对象**less<string>**（**set**容器默认的第2个模板参数）比较它保存的那些单词，该函数依次使用**string::operator<( )**进行比较操作，因此这些单词按字母顺序出现。

仅仅为了得到一个经过排序的序列，就没有必要使用**set**。可以在不同的STL容器上使用函数**sort( )**（与STL众多的其他函数）来达到这个目的。然而，在这里或许**set**将会更快地完成该操作。当只想做查找操作时，使用**set**将会特别便利，因为其**find( )**成员函数有对数级的复杂性，因此它比通用的**find( )**算法要快得多。如前所述，通用的**find( )**算法需要遍历全部范围直到找到要搜寻的元素（这将导致最坏情况下算法复杂性为N，平均复杂性为$N/2$）。然而，如果有一个已经排过序的序列容器，在查找元素时使用**equal_range( )**，就可以得到对数级的算法复杂性。

下面这个程序显示了如何使用迭代器**istreambuf_iterator**来构建单词表，迭代器将字符从一个地方（输入流）移到另一个地方（一个**string**对象），该操作依赖标准C库函数**isalpha( )**的返回值是否为真：

481

```
//: C07:WordList2.cpp
// Illustrates istreambuf_iterator and insert iterators.
#include <cstring>
#include <fstream>
#include <iostream>
#include <iterator>
#include <set>
#include <string>
#include "../require.h"
using namespace std;

int main(int argc, char* argv[]) {
  char* fname = "WordList2.cpp";
  if(argc > 1) fname = argv[1];
  ifstream in(fname);
  assure(in, fname);
  istreambuf_iterator<char> p(in), end;
  set<string> wordlist;
  while(p != end) {
```

```
    string word;
    insert_iterator<string> ii(word, word.begin());
    // Find the first alpha character:
    while(p != end && !isalpha(*p))
      ++p;
    // Copy until the first non-alpha character:
    while(p != end && isalpha(*p))
      *ii++ = *p++;
    if(word.size() != 0)
      wordlist.insert(word);
  }
  // Output results:
  copy(wordlist.begin(), wordlist.end(),
    ostream_iterator<string>(cout, "\n"));
} ///:~
```

这个例子是由**Nathan Myers**提议的，他发明了**istreambuf_iterator**及其家族成员。这个迭代器从一个流中逐字符地提取信息。虽然**istreambuf_iterator**的模板参数可能暗示读者提取例如以**int**型数据而非**char**型数据，但事实却不是这样。该参数必须是某些字符类型——常规的**char**类型或宽字符类型。

文件打开以后，称为**p**的**istreambuf_iterator**被附到该**istream**上，这样就可以从中提取字符了。名为**wordlist**的**set<string>**将保存作为结果的单词。 482

采用**while**循环从输入流中读取单词直到发现输入流结束。这是用**istreambuf_iterator**的默认构造函数进行检测的，它产生超越末尾的迭代器对象**end**。因此，如果要进行测试以确信不在该流的末尾，只需用**p!=end**。

在这里使用的第2个迭代器类型是在前面已经看到的**insert_iterator**。利用它将对象插入一个容器。在这里，“容器”是一个称为**word**的**string**，它对于**insert_iterator**的目的来说，其行为像一个容器。对于**insert_iterator**的构造函数，则需要该容器和一个指明应在何处开始插入这些字符的迭代器。也可以使用一个**back_insert_iterator**，它需要容器拥有一个**push_back( )**（**string**产生）。

while循环在做好各种准备后，就开始查找第1个字母字符，对**start**进行增1操作直到那个字符被找到。然后将查找到的字符从一个迭代器指向的位置复制到另一个迭代器指向的位置，当发现一个非字母字符时停止复制。假定其不为空，则每一个**word**都被添加到**wordlist**中去。

可完全重用的标识符识别器

单词表例子使用不同的方法从流中提取标识符，但它们都不特别灵活。因为STL容器和算法都是围绕着迭代器来展开的，所以最灵活的解决方法是它自己使用迭代器。可以把**TokenIterator**想像成一个迭代器，该迭代器封装其任何能够产生字符的其他迭代器。因为它确实是一个输入迭代器类型（最原始的迭代器类型），可以向任何**STL**算法提供输入。不仅其本身就是一个很有用的工具，下列的**TokenIterator**也是如何设计用户自己的迭代器的很好的例子。[⊖]

TokenIterator类具有双重灵活性。首先，可以选择产生**char**输入的迭代器类型。其次，**TokenIterator**不是仅说明什么字符表示分界符，而是使用一个函数对象判定函数，其**operator( )**接受一个**char**型参数并决定它是否将计入标识符。虽然这两个例子在这里给出了什么字符属于标识符的静态概念，但是用户可以很容易地设计自己的函数对象，以便在读入字符的时候改变其状态，创建一个更复杂的解析器。 483

⊖ 这是Nathan Myers讲授的另一个例子。

和**TokenIterator**模板一起，下列的头文件还包含了两个基本判定函数，**Isalpha**和**Delimiters**:

```
//: C07:TokenIterator.h
#ifndef TOKENITERATOR_H
#define TOKENITERATOR_H
#include <algorithm>
#include <cctype>
#include <functional>
#include <iterator>
#include <string>

struct Isalpha : std::unary_function<char, bool> {
  bool operator()(char c) { return std::isalpha(c); }
};

class Delimiters : std::unary_function<char, bool> {
  std::string exclude;
public:
  Delimiters() {}
  Delimiters(const std::string& excl) : exclude(excl) {}
  bool operator()(char c) {
    return exclude.find(c) == std::string::npos;
  }
};

template<class InputIter, class Pred = Isalpha>
class TokenIterator : public std::iterator<
    std::input_iterator_tag, std::string, std::ptrdiff_t> {
  InputIter first;
  InputIter last;
  std::string word;
  Pred predicate;
public:
  TokenIterator(InputIter begin, InputIter end,
    Pred pred = Pred())
    : first(begin), last(end), predicate(pred) {
      ++*this;
  }
  TokenIterator() {} // End sentinel
  // Prefix increment:
  TokenIterator& operator++() {
    word.resize(0);
    first = std::find_if(first, last, predicate);
    while(first != last && predicate(*first))
      word += *first++;
    return *this;
  }
  // Postfix increment
  class CaptureState {
    std::string word;
  public:
    CaptureState(const std::string& w) : word(w) {}
    std::string operator*() { return word; }
  };
  CaptureState operator++(int) {
    CaptureState d(word);
    ++*this;
    return d;
  }
  // Produce the actual value:
  std::string operator*() const { return word; }
  const std::string* operator->() const { return &word; }
  // Compare iterators:
```

484

```
  bool operator==(const TokenIterator&) {
    return word.size() == 0 && first == last;
  }
  bool operator!=(const TokenIterator& rv) {
    return !(*this == rv);
  }
};
#endif // TOKENITERATOR_H ///:~
```

TokenIterator类派生自**std::iterator**模板。它可能表现出某些来自**std::iterator**的功能，但它是纯粹对迭代器进行标记的一种方式，用来告诉使用它的容器可以做些什么。在这里，可以看到作为**iterator_category**模板参数的**input_iterator_tag**——这告诉那些询问者，**TokenIterator**只有一个输入迭代器的能力，不能与那些需要更复杂的迭代器的算法一起使用。除了进行标记以外，**std::iterator**不能做超出提供几个有用的类型定义的任何事情。用户必须自行实现所有其他的功能。 485

起初读者可能会认为**TokenIterator**类有点奇怪，因为其第1个构造函数需要“开始”和“终止”两个迭代器作为参数，和它们在一起的还有一个判定函数。记住，这是一个“封装器”迭代器，在输入结束时它没有办法如何告知何时其输入处于末尾，所以在第1个构造函数中这个“终止”迭代器是必需的。第2个（默认）构造函数存在的理由是，STL算法（以及所有用户自己编写的算法）需要一个**TokenIterator**标记作为超越末尾的值。因为判断一个**TokenIterator**是否已经到达其输入的末尾的所有信息都已经由第1个构造函数收集，这第2个构造函数创建**TokenIterator**对象，在算法中它只作为占位符使用。

行为的核心发生在运算符**operator++**中。它使用**string::resize( )**擦除当前**word**的值，然后使用**find_if( )**寻找第1个满足判定函数的字符（如此来发现一个新的标识符的起始位置）。结果迭代器被分配给**first**，因此将**first**向前移动至标识符的起始位置。然后，一旦判定函数被满足而又没有到达输入的末尾，输入字符就被复制到**word**中。最后，**TokenIterator**对象返回，并且必须被解析以便访问新的标识符。

这里的前缀增1要求有一个**CaptureState**类型的对象在增1前持有值，因此它是可以返回的。产生的实际值是一个直接的**operator***操作。为输出迭代器定义的其余的函数仅仅是**operator==**和**operator!=**，用以指明**TokenIterator**是否已经到达了其输入的末尾。读者可以看到**operator==**的参数被忽略——它仅仅关心是否已经到达其内部的**last**迭代器。注意，**operator!=**是通过**operator==**定义的。

一个好的**TokenIterator**测试包括许多不同来源的输入字符，包括一个**streambuf_iterator**、一个**char***和一个**deque<char>::iterator**。最后，最初的单词表的问题解决如下：

```
//: C07:TokenIteratorTest.cpp {-g++}
#include <fstream>
#include <iostream>
#include <vector>
#include <deque>
#include <set>
#include "TokenIterator.h"
#include "../require.h"
using namespace std;

int main(int argc, char* argv[]) {
  char* fname = "TokenIteratorTest.cpp";
  if(argc > 1) fname = argv[1];
  ifstream in(fname);
```

486

```
  assure(in, fname);
  ostream_iterator<string> out(cout, "\n");
  typedef istreambuf_iterator<char> IsbIt;
  IsbIt begin(in), isbEnd;
  Delimiters delimiters(" \t\n~;()\"<>:{}[]+-=&*#.,/\\");
  TokenIterator<IsbIt, Delimiters>
    wordIter(begin, isbEnd, delimiters), end;
  vector<string> wordlist;
  copy(wordIter, end, back_inserter(wordlist));
  // Output results:
  copy(wordlist.begin(), wordlist.end(), out);
  *out++ = "----------------------------------";
  // Use a char array as the source:
  char* cp = "typedef std::istreambuf_iterator<char> It";
  TokenIterator<char*, Delimiters>
    charIter(cp, cp + strlen(cp), delimiters), end2;
  vector<string> wordlist2;
  copy(charIter, end2, back_inserter(wordlist2));
  copy(wordlist2.begin(), wordlist2.end(), out);
  *out++ = "----------------------------------";
  // Use a deque<char> as the source:
  ifstream in2("TokenIteratorTest.cpp");
  deque<char> dc;
  copy(IsbIt(in2), IsbIt(), back_inserter(dc));
  TokenIterator<deque<char>::iterator,Delimiters>
    dcIter(dc.begin(), dc.end(), delimiters), end3;
  vector<string> wordlist3;
  copy(dcIter, end3, back_inserter(wordlist3));
  copy(wordlist3.begin(), wordlist3.end(), out);
  *out++ = "----------------------------------";
  // Reproduce the Wordlist.cpp example:
  ifstream in3("TokenIteratorTest.cpp");
  TokenIterator<IsbIt, Delimiters>
    wordIter2(IsbIt(in3), isbEnd, delimiters);
  set<string> wordlist4;
  while(wordIter2 != end)
    wordlist4.insert(*wordIter2++);
  copy(wordlist4.begin(), wordlist4.end(), out);
} ///:~
```

487

在使用**istreambuf_iterator**时，创建一个附属于**istream**的对象，并与默认构造函数一起作为超越末尾标记。这两者用于创建将产生标识符的**TokenIterator**；而默认构造函数创建一个伪**TokenIterator**超越末尾的标记。（这仅仅是个占位符并且被忽略。）**TokenIterator**产生那些要被插入**string**容器的**string**——在这里，除了最后一个以外，在所有的情况下都使用**vector<string>**。（也可以将所有的结果连接成一个**string**。）除此之外，**TokenIterator**就像任何其他输入迭代器一样工作。

在定义一个双向（并且因此也成为一个随机访问）迭代器时，可以使用**std::reverse_iterator**适配器“免费地”得到反向迭代器。如果已经为一个具有双向能力的容器定义了一个迭代器的话，可以从如下在容器类里的前向遍历迭代器那里得到一个反向迭代器：

```
// Assume "iterator" is your nested iterator type
typedef std::reverse_iterator<iterator> reverse_iterator;
reverse_iterator rbegin() {return reverse_iterator(end());
reverse_iterator rend() {return reverse_iterator(begin());
```

std::reverse_iterator适配器可以做所有这些工作。比如，如果使用运算符*来解析反向迭代器，它自动地对它持有的前向迭代器的一个临时拷贝减1，以便返回正确的元素，因为反向迭代器在逻辑上指向它们引用的元素的下一个位置。

7.6 堆栈

堆栈**stack**容器，与**queue**和**priority_queue**一起被归类为适配器，这意味着它们将通过调整某一个基本序列容器以存储自己的数据。这是一个遗憾的令人困惑的情况，为什么某些事情一定要与它的底层实现的细节联系在一起呢——这些容器被称为“适配器”的真相只对库的创建者才有基本的价值。当用户使用它们时，通常并不关心它们是否是适配器，仅需知道它们能够解决自己的问题就行了。诚然，有时知道可以选择不同的实现或者在现存的容器对象之上建立一个适配器是很有用的，但是，通常那一层次的功能已经在适配器的行为中被删除了。因此，如果看到在别处某个容器被强调是一个适配器，一般只能指出实际上什么时候它是有用的。注意，每一种类型的适配器都有一个该适配器构建在其上的默认的容器，而且这种默认是最明智的实现方式。在大多数情况下，用户不必关心容器的底层的具体实现。 488

下面的例子显示实现**stack<string>**的3种方式：默认方式（使用**deque**），然后采用**vector**的方式，最后一个采用**list**的方式：

```
//: C07:Stack1.cpp
// Demonstrates the STL stack.
#include <fstream>
#include <iostream>
#include <list>
#include <stack>
#include <string>
#include <vector>
using namespace std;

// Rearrange comments below to use different versions.
typedef stack<string> Stack1; // Default: deque<string>
// typedef stack<string, vector<string> > Stack2;
// typedef stack<string, list<string> > Stack3;

int main() {
  ifstream in("Stack1.cpp");
  Stack1 textlines; // Try the different versions
  // Read file and store lines in the stack:
  string line;
  while(getline(in, line))
    textlines.push(line + "\n");
  // Print lines from the stack and pop them:
  while(!textlines.empty()) {
    cout << textlines.top();
    textlines.pop();
  }
} ///:~
```

如果读者使用过其他**stack**类的话，这里的**top( )**和**pop( )**操作似乎并不直观。当调用**pop( )**时，它返回一个**void**值而不是所预期的栈顶元素。如果想要栈顶元素，可以通过**top( )**取得指向它的一个引用。这样做的结果效率更高，因为传统的**pop( )**函数必须返回一个值而不是一个引用，因此调用拷贝构造函数。更重要的是，这是*异常安全的*（**exception safe**），就像我们在第1章中讨论的那样。如果**pop( )**在改变栈状态的同时试图返回栈顶元素，那么在元素的拷贝构造函数中产生的某个异常就会导致元素的丢失。在使用**stack**（或者一个**priority_queue**，将在稍后描述）时，可以高效地查阅**top( )**，就像你希望得那么快，然后明确使用**pop( )**将栈顶元素丢弃。（也许，如果使用一些不同于大家熟悉的“**pop**”这样的术语来定义函数，解释起来可能会更清楚一点儿。） 489

stack模板有一个简单的接口——本质上就是在较早前看到的那些成员函数。因为对于一个**stack**来说，只有访问其栈顶元素才有意义，没有提供能够遍历它的迭代器。也没有复杂的初始化形式，但是如果需要这样做的话，可以使用**stack**的底层容器。比如，假定有一个函数，期望**stack**的接口，但是在程序的其余部分需要将对象存储在**list**中。下面的程序存储文件中的每一行，与该行中的前导空白字符的个数一起存储。(可以想像把它作为对源代码执行某种重新格式化操作的出发点。)

```
//: C07:Stack2.cpp
// Converting a list to a stack.
#include <iostream>
#include <fstream>
#include <stack>
#include <list>
#include <string>
#include <cstddef>
using namespace std;

// Expects a stack:
template<class Stk>
void stackOut(Stk& s, ostream& os = cout) {
  while(!s.empty()) {
    os << s.top() << "\n";
    s.pop();
  }
}
class Line {
  string line; // Without leading spaces
  size_t lspaces; // Number of leading spaces
public:
  Line(string s) : line(s) {
    lspaces = line.find_first_not_of(' ');
    if(lspaces == string::npos)
      lspaces = 0;
    line = line.substr(lspaces);
  }
  friend ostream& operator<<(ostream& os, const Line& l) {
    for(size_t i = 0; i < l.lspaces; i++)
      os << ' ';
    return os << l.line;
  }
  // Other functions here...
};

int main() {
  ifstream in("Stack2.cpp");
  list<Line> lines;
  // Read file and store lines in the list:
  string s;
  while(getline(in, s))
    lines.push_front(s);
  // Turn the list into a stack for printing:
  stack<Line, list<Line> > stk(lines);
  stackOut(stk);
} ///:~
```

490

需要**stack**接口的函数仅仅发送每个**top( )**对象到一个**ostream**，然后通过调用**pop( )**将其删除。**Line**类判断前导空白字符的个数，然后存储没有这些前导空白字符的行的内容。**ostream operator<<**重新插入前导空白字符，因此该行能够被正确地打印出来，但是能很容易地通过改变**lspaces**的值来改变空白字符的个数。(做这件事的成员函数没有在这里显示。)

在**main( )** 函数中，输入文件被读入到**list<Line>**，然后链表中的每一行都被复制到一个**stack**，该**stack**被送到**stackOut( )**函数中。

不能从头至尾对一个**stack**进行迭代；这就强调了，当创建一个**stack**时，只能希望对其进行**stack**操作。可以使用一个**vector**及其**back( )**、**push_back( )**和**pop_back( )**成员函数获得等价的“堆栈”功能，还拥有**vector**的所有附加的功能。程序**Stack1.cpp**可以重写成如下形式： 491

```
//: C07:Stack3.cpp
// Using a vector as a stack; modified Stack1.cpp.
#include <fstream>
#include <iostream>
#include <string>
#include <vector>
using namespace std;

int main() {
  ifstream in("Stack3.cpp");
  vector<string> textlines;
  string line;
  while(getline(in, line))
    textlines.push_back(line + "\n");
  while(!textlines.empty()) {
    cout << textlines.back();
    textlines.pop_back();
  }
} ///:~
```

这个程序产生像**Stack1.cpp**一样输出，但现在还可以进行与**vector**一样的操作。**list**也可以将元素压入前端，但是它通常比与**vector**一起使用**push_back( )**的效率低。（另外，对于将元素压入前端的操作，**deque**通常比**list**的效率更高。）

7.7 队列

queue容器是一个受到限制的**deque**形式——只可以在队列一端放入元素，而在另一端删除它们。在功能上，可以在需要使用**queue**的任何地方使用**deque**，那时也能够使用**deque**的附加功能。需要使用**queue**而不是**deque**的惟一理由就是，当读者希望强调仅仅执行与**queue**相似的行为的时候。

queue类是一个如同**stack**的适配器，因为它也建立在另一个序列容器的基础之上。就像读者猜测的那样，对**queue**的理想的实现是**deque**，而其对**queue**来说是默认的模板参数；很少需要不同的实现。 492

如果想建立这样一个系统模型，即系统中的某些元素正在等待另一些元素的服务时，时常使用队列。“银行出纳员问题” 就是一个经典的例子。顾客们在随机的时间间隔到达银行，进入某一行队列排队，然后由一组出纳员服务。因为顾客们到达是随机的，并且每一个顾客得到的服务时间总数也是随机的，所以没有一种方法能决定性地知道在任何时间点某行队列有多长。然而，模拟这种情形并且看看到底会发生什么事情却是可能的。

在对现实的模拟中，每个顾客和每个出纳员都在独立的线程中运行。这多么像是一个多线程的环境，因此每个顾客和出纳员都有他自己的线程。然而，标准C++不支持多线程处理。另一方面，通过对代码做一些小的调整，模拟足够的多线程处理以提供一个满意的解决方案是可能的。[⊖]

⊖ 我们将在第11章再次讨论多线程处理问题。

在多线程处理中，多个受控制的线程同时运行，共享同一个地址空间。通常用户拥有的CPU数量都会比运行的线程数量少（常常只有一个CPU）。为得到虚拟的环境，应使每一个线程都拥有其自己的CPU，一种时间分片（**time-slicing**）机制说“OK，当前线程你已经占用了足够多的时间，我马上就要让你停止从而给其他线程一些时间了”。这种自动的线程停止和启动被称为抢占（**preemptive**），它意味着（程序员）不需要去管理线程处理的过程。

另一种方法就是每个线程自动地将CPU让给线程调度器，该线程调度器然后寻找另一个需要运行的线程。另外，建立“时间分片”进入到系统中的各个类。在这里，将那些出纳员表示为“线程”（顾客将是被动的）。每个出纳员都有一个进行无限循环处理的成员函数**run( )**，该成员函数将在执行确定数量的“时间单元”后返回。通过使用平常的返回机制，排除了任何需要进行的交换处理。虽然产生的程序很小，但是它提供了一个不平常的合理的模拟场景：

493

```
//: C07:BankTeller.cpp {RunByHand}
// Using a queue and simulated multithreading
// to model a bank teller system.
#include <cstdlib>
#include <ctime>
#include <iostream>
#include <iterator>
#include <list>
#include <queue>
using namespace std;

class Customer {
  int serviceTime;
public:
  Customer() : serviceTime(0) {}
  Customer(int tm) : serviceTime(tm) {}
  int getTime() { return serviceTime; }
  void setTime(int newtime) { serviceTime = newtime; }
  friend ostream&
  operator<<(ostream& os, const Customer& c) {
    return os << '[' << c.serviceTime << ']';
  }
};

class Teller {
  queue<Customer>& customers;
  Customer current;
  enum { SLICE = 5 };
  int ttime; // Time left in slice
  bool busy; // Is teller serving a customer?
public:
  Teller(queue<Customer>& cq)
  : customers(cq), ttime(0), busy(false) {}
  Teller& operator=(const Teller& rv) {
    customers = rv.customers;
    current = rv.current;
    ttime = rv.ttime;
    busy = rv.busy;
    return *this;
  }
  bool isBusy() { return busy; }
  void run(bool recursion = false) {
    if(!recursion)
      ttime = SLICE;
    int servtime = current.getTime();
    if(servtime > ttime) {
      servtime -= ttime;
```

494

```
      current.setTime(servtime);
      busy = true; // Still working on current
      return;
    }
    if(servtime < ttime) {
      ttime -= servtime;
      if(!customers.empty()) {
        current = customers.front();
        customers.pop(); // Remove it
        busy = true;
        run(true); // Recurse
      }
      return;
    }
    if(servtime == ttime) {
      // Done with current, set to empty:
      current = Customer(0);
      busy = false;
      return; // No more time in this slice
    }
  }
};

// Inherit to access protected implementation:
class CustomerQ : public queue<Customer> {
public:
  friend ostream&
  operator<<(ostream& os, const CustomerQ& cd) {
    copy(cd.c.begin(), cd.c.end(),
      ostream_iterator<Customer>(os, ""));
    return os;
  }
};

int main() {
  CustomerQ customers;
  list<Teller> tellers;
  typedef list<Teller>::iterator TellIt;
  tellers.push_back(Teller(customers));
  srand(time(0)); // Seed the random number generator
  clock_t ticks = clock();
  // Run simulation for at least 5 seconds:
  while(clock() < ticks + 5 * CLOCKS_PER_SEC) {
    // Add a random number of customers to the
    // queue, with random service times:
    for(int i = 0; i < rand() % 5; i++)
      customers.push(Customer(rand() % 15 + 1));
    cout << '{' << tellers.size() << '}'
         << customers << endl;
    // Have the tellers service the queue:
    for(TellIt i = tellers.begin();
      i != tellers.end(); i++)
      (*i).run();
    cout << '{' << tellers.size() << '}'
         << customers << endl;
    // If line is too long, add another teller:
    if(customers.size() / tellers.size() > 2)
      tellers.push_back(Teller(customers));
    // If line is short enough, remove a teller:
    if(tellers.size() > 1 &&
      customers.size() / tellers.size() < 2)
      for(TellIt i = tellers.begin();
        i != tellers.end(); i++)
```

495

```
        if(!(*i).isBusy()) {
          tellers.erase(i);
          break; // Out of for loop
        }
    }
} ///:~
```

每个顾客都需要一个确定的服务时间总额，这就是一个出纳员必须在为某个顾客提供其所需服务上花费的时间单元数。为每个顾客提供的服务时间总额都不同，并且这个时间总额肯定是随机的。另外，也不会知道在每个时间间隔内究竟会有多少个顾客到达，因此这也肯定是随机的。

这些顾客**Customer**对象被保存在一个**queue<Customer>**中，并且每个出纳员**Teller**对象都持有那个队列的一个引用。在一个**Teller**对象完成对当前**Customer**对象的服务以后，这个**Teller**将会从队列中得到另一个**Customer**，开始继续为这个新**Customer**提供服务，系统从该**Teller**分配到的时间片里面减少**Customer**的服务时间。所有这些逻辑都包含在成员函数**run( )**中，它只是一个具有3个分支的**if**语句，该语句基于以下事件建立，即当前顾客所必需的服务时间总额是小于、大于、或是等于出纳员在当前时间片中剩余的时间总额。注意，如果该出纳员**Teller**在完成对一个**Customer**的服务后还有多余的时间，它再获得一个新的**Customer**，然后实施自身的递归处理。就像使用**stack**一样，在使用**queue**时，它只是一个**queue**，不具有基础序列容器的任何其他功能。这包括获得一个迭代器以遍历**stack**的能力。然而，在**queue**内部将底层序列容器（构建**queue**的基础）作为一个**protected**成员来保存，在C++标准中该成员被指定以**'c'**来做标识符，这意味着可以通过派生自**queue**的类来访问底层实现。在这里类**CustomerQ**正是这样做的，其惟一目的就是定义一个**ostream operator<<**，以便在**queue**上实施迭代并显示其成员。

496

这个模拟系统的驱动器就是**main( )**函数中的**while**循环，它使用处理器的时钟滴答（定义于**<ctime>**中）来决定该模拟系统是否已经至少运行了5秒钟。在每次经过从头到尾的循环的开始，都要加入随机的顾客数，和随机的服务时间。为了看到系统当前的状态，出纳员的数量和队列的内容都将显示出来。每个出纳员处理完后，重复地显示这些信息。在这一点上，系统通过比较顾客和出纳员的数量来进行调整。如果某行队列太长，就加入其他的出纳员，而如果队列足够短，则删除一个出纳员。在程序中的这个调整区段中，可以用实验的策略得到关于添加和删除出纳员的最佳数据。如果这是惟一想要修改的区段，也许要将策略封装到不同的对象中去。

本教材将在第11章中的多线程练习中再次涉及这个例子。

7.8 优先队列

当向一个优先队列**priority_queue**用**push( )**压入一个对象时，那个对象根据一个比较函数或函数对象在队列中排序。（可以允许用默认的**less**模板来代替这个函数或函数对象，或者可以提供一个用户自己定义的函数或函数对象。）**priority_queue**确定在用**top( )**查看顶部元素时，该元素将是具有最高优先级的一个元素。当处理完该元素以后，调用**pop( )**删除它，并且促使下一个元素进入该位置。因此，**priority_queue**拥有与**stack**几乎相同的接口，但它的表现不同。

就像**stack**和**queue**一样，**priority_queue**是一个基于某个基本序列容器进行构建的适配器——默认的序列容器是**vector**。

497

创造一个用来处理**int**型数据的**priority_queue**是个很平常的工作：

```
//: C07:PriorityQueue1.cpp
#include <cstdlib>
#include <ctime>
#include <iostream>
#include <queue>
using namespace std;

int main() {
  priority_queue<int> pqi;
  srand(time(0)); // Seed the random number generator
  for(int i = 0; i < 100; i++)
    pqi.push(rand() % 25);
  while(!pqi.empty()) {
    cout << pqi.top() << ' ';
    pqi.pop();
  }
} ///:~
```

该程序向**priority_queue**压入100个数值介于0到24之间的随机数。在运行这个程序时，会看到它允许出现重复的值，而且最高值先出现。为了演示如何通过提供用户自己的函数或函数对象以改变其元素的排列顺序，下面的程序给予较低值的数以最高的优先级：

```
//: C07:PriorityQueue2.cpp
// Changing the priority.
#include <cstdlib>
#include <ctime>
#include <functional>
#include <iostream>
#include <queue>
using namespace std;

int main() {
  priority_queue<int, vector<int>, greater<int> > pqi;
  srand(time(0));
  for(int i = 0; i < 100; i++)
    pqi.push(rand() % 25);
  while(!pqi.empty()) {
    cout << pqi.top() << ' ';
    pqi.pop();
  }
} ///:~
```

498

一个更有趣的问题是to-do列表，这里每个对象都包含一个**string**，和一个主优先级及一个次优先级的值：

```
//: C07:PriorityQueue3.cpp
// A more complex use of priority_queue.
#include <iostream>
#include <queue>
#include <string>
using namespace std;

class ToDoItem {
  char primary;
  int secondary;
  string item;
public:
  ToDoItem(string td, char pri = 'A', int sec = 1)
  : primary(pri), secondary(sec), item(td) {}
  friend bool operator<(
    const ToDoItem& x, const ToDoItem& y) {
    if(x.primary > y.primary)
```

```
      return true;
    if(x.primary == y.primary)
      if(x.secondary > y.secondary)
        return true;
    return false;
  }
  friend ostream&
  operator<<(ostream& os, const ToDoItem& td) {
    return os << td.primary << td.secondary
      << ": " << td.item;
  }
};

int main() {
  priority_queue<ToDoItem> toDoList;
  toDoList.push(ToDoItem("Empty trash", 'C', 4));
  toDoList.push(ToDoItem("Feed dog", 'A', 2));
  toDoList.push(ToDoItem("Feed bird", 'B', 7));
  toDoList.push(ToDoItem("Mow lawn", 'C', 3));
  toDoList.push(ToDoItem("Water lawn", 'A', 1));
  toDoList.push(ToDoItem("Feed cat", 'B', 1));
  while(!toDoList.empty()) {
    cout << toDoList.top() << endl;
    toDoList.pop();
  }
} ///:~
```

499

由于是与**less< >**一同工作，所以**ToDoItem**的**operator<**必须是一个非成员函数。除此之外，每一件事情都是自动发生的。输出结果如下：

```
A1: Water lawn
A2: Feed dog
B1: Feed cat
B7: Feed bird
C3: Mow lawn
C4: Empty trash
```

由于设计上的原因，不能在一个**priority_queue**上从头到尾进行迭代，但是可以用一个**vector**来模拟**priority_queue**的行为，因此允许访问那个**vector**。可以通过观察**priority_queue**的实现来这样做，它使用的函数有**make_heap( )**、**push_heap( )**以及**pop_heap( )**。（这些函数是**priority_queue**的灵魂——事实上，可以说堆就是一个优先队列，**priority_queue**只是对它的一个封装。）结果相当简单，但是读者可能会想，可能还存在一个捷径。因为**priority_queue**使用的容器是**protected**的（并且有标识符，根据标准C++规格说明，该标识符被命名为**c**），所以可以继承一个新类，该新类提供了访问底层实现的途径：

```
//: C07:PriorityQueue4.cpp
// Manipulating the underlying implementation.
#include <algorithm>
#include <cstdlib>
#include <ctime>
#include <iostream>
#include <iterator>
#include <queue>
using namespace std;

class PQI : public priority_queue<int> {
public:
  vector<int>& impl() { return c; }
};

int main() {
```

500

```
  PQI pqi;
  srand(time(0));
  for(int i = 0; i < 100; i++)
    pqi.push(rand() % 25);
  copy(pqi.impl().begin(), pqi.impl().end(),
    ostream_iterator<int>(cout, " "));
  cout << endl;
  while(!pqi.empty()) {
    cout << pqi.top() << ' ';
    pqi.pop();
  }
} ///:~
```

然而，如果运行这个程序，就会发现当调用**pop( )**时，得到的这个**vector**包含的元素并不是按降序排列，这就是想要从优先队列得到的顺序。似乎如果想要创建一个作为优先队列的**vector**，必须手工完成它。就像下面这样：

```
//: C07:PriorityQueue5.cpp
// Building your own priority queue.
#include <algorithm>
#include <cstdlib>
#include <ctime>
#include <iostream>
#include <iterator>
#include <queue>
using namespace std;

template<class T, class Compare>
class PQV : public vector<T> {
  Compare comp;
public:
  PQV(Compare cmp = Compare()) : comp(cmp) {
    make_heap(this->begin(),this->end(), comp);
  }
  const T& top() { return this->front(); }
  void push(const T& x) {
    this->push_back(x);
    push_heap(this->begin(),this->end(), comp);
  }
  void pop() {
    pop_heap(this->begin(),this->end(), comp);
    this->pop_back();
  }
};

int main() {
  PQV< int, less<int> > pqi;
  srand(time(0));
  for(int i = 0; i < 100; i++)
    pqi.push(rand() % 25);
  copy(pqi.begin(), pqi.end(),
    ostream_iterator<int>(cout, " "));
  cout << endl;
  while(!pqi.empty()) {
    cout << pqi.top() << ' ';
    pqi.pop();
  }
} ///:~
```

501

但是这个程序表现得跟前面那个程序一样！读者在**vector**底层中的一个被称为堆（heap）的存储区观察到什么。这个堆数据结构表现为一个优先队列的树的结构（被存储在**vector**的

线性结构中），但是在对其从头到尾进行迭代的时候，并不会得到一个线性的优先队列顺序。你可能认为可以仅调用**sort_heap( )**进行排序，但是那只能起一次作用，然后你将不再拥有一个堆，而只剩一个被排过序的列表。这意味着，要返回将其作为一个堆来使用，用户必须记住首先调用**make_heap( )**。这些都可以被封装到自定义的优先队列中去：

```
//: C07:PriorityQueue6.cpp
#include <algorithm>
#include <cstdlib>
#include <ctime>
#include <iostream>
#include <iterator>
#include <queue>
using namespace std;

template<class T, class Compare>
class PQV : public vector<T> {
  Compare comp;
  bool sorted;
  void assureHeap() {
    if(sorted) {
      // Turn it back into a heap:
      make_heap(this->begin(),this->end(), comp);
        sorted = false;
      }
    }
  public:
    PQV(Compare cmp = Compare()) : comp(cmp) {
      make_heap(this->begin(),this->end(), comp);
      sorted = false;
    }
    const T& top() {
      assureHeap();
      return this->front();
    }
    void push(const T& x) {
      assureHeap();
      this->push_back(x); // Put it at the end
      // Re-adjust the heap:
      push_heap(this->begin(),this->end(), comp);
    }
    void pop() {
      assureHeap();
      // Move the top element to the last position:
      pop_heap(this->begin(),this->end(), comp);
      this->pop_back();// Remove that element
    }
    void sort() {
      if(!sorted) {
        sort_heap(this->begin(),this->end(), comp);
        reverse(this->begin(),this->end());
        sorted = true;
      }
    }
  };

  int main() {
    PQV< int, less<int> > pqi;
    srand(time(0));
    for(int i = 0; i < 100; i++) {
      pqi.push(rand() % 25);
      copy(pqi.begin(), pqi.end(),
```

502

```
        ostream_iterator<int>(cout, " "));
      cout << "\n-----" << endl;
    }
    pqi.sort();
    copy(pqi.begin(), pqi.end(),
      ostream_iterator<int>(cout, " "));
  cout << "\n-----" << endl;
  while(!pqi.empty()) {
    cout << pqi.top() << ' ';
    pqi.pop();
  }
} ///:~
```

503

如果**sorted**为真，**vector**就不是作为一个堆来进行组织的，而仅仅是个排过序的序列。函数**assureHeap( )**保证在对其进行任何堆操作之前使其倒退成为一个堆的形式。**main( )**中的第1个**for**循环引入了新的额外特性，它显示一个正在被构建的堆。

在前面的两个程序中采用了“**this->**”前缀的这种表面上并非必要（extraneous）的用法。虽然某些编译器不需要这种用法，但标准C++的定义有这种用法。注意，类**PQV**派生自**vector<T>**，因此继承自**vector<T>**的**begin( )**和**end( )**都是依赖的名字。[⊖]在模板的定义中，编译器不能查寻这些来自于依赖的基类的名字（在这种情况下为**vector**），因为对于某个给定的实例，一个明确特化的模板版本可能使用的并不是一个给定的成员。特别的命名需求保证在某些情况下用户不会结束正在调用的一个基类成员，在另外的情况下函数可能来自一个外围空间（比如一个全局的）的函数。编译器无法知道调用的**begin( )**是依赖的，因此必须使用“**this->**”限定给它提供一个线索。[⊖]这就告诉了编译器，这个**begin( )**是在**PQV**的范围之内的，因此它就会等待直到**PQV**的一个实例完全地被实例化。如果去掉这个限定前缀，编译器就会对名字**begin**和**end**尝试进行早期查找（在模板定义期间查找，并且会查找失败，因为在这个例子中包含的外围字典空间中并不包括这些名字声明）。然而在上面的程序代码中，编译器一直在**pqi**实例化的那一点等待，然后在**vector<int>**中寻找**begin( )**和**end( )**的正确的特化。

这个解决方案的惟一的缺点就是，用户必须记住在将其作为一个排过序的序列进行查看之前必须先调用**sort( )**（一个可以想得到的方法，就是重新定义所有能够产生迭代器的成员函数，以便保证排序的进行）。另一个解决方案就是创建一个非**vector**的优先队列，但是，每当需要时就构建一个使用**vector**的优先队列： 504

```
//: C07:PriorityQueue7.cpp
// A priority queue that will hand you a vector.
#include <algorithm>
#include <cstdlib>
#include <ctime>
#include <iostream>
#include <iterator>
#include <queue>
#include <vector>
using namespace std;

template<class T, class Compare> class PQV {
  vector<T> v;
  Compare comp;
public:
```

⊖ 这意味着它们在以某种方式依赖于一个模板参数。参看第5章中的“名字查找问题”一节。

⊖ 如第5章所述，即任何有效限定，比如**PQV::**，所做的那样。

```
  // Don't need to call make_heap(); it's empty:
  PQV(Compare cmp = Compare()) : comp(cmp) {}
  void push(const T& x) {
    v.push_back(x); // Put it at the end
    // Re-adjust the heap:
    push_heap(v.begin(), v.end(), comp);
  }
  void pop() {
    // Move the top element to the last position:
    pop_heap(v.begin(), v.end(), comp);
    v.pop_back(); // Remove that element
  }
  const T& top() { return v.front(); }
  bool empty() const { return v.empty(); }
  int size() const { return v.size(); }
  typedef vector<T> TVec;
  TVec getVector() {
    TVec r(v.begin(), v.end());
    // It's already a heap
    sort_heap(r.begin(), r.end(), comp);
    // Put it into priority-queue order:
    reverse(r.begin(), r.end());
    return r;
  }
};

int main() {
  PQV<int, less<int> > pqi;
  srand(time(0));
  for(int i = 0; i < 100; i++)
    pqi.push(rand() % 25);
  const vector<int>& v = pqi.getVector();
  copy(v.begin(), v.end(),
    ostream_iterator<int>(cout, " "));
  cout << "\n-----------" << endl;
  while(!pqi.empty()) {
    cout << pqi.top() << ' ';
    pqi.pop();
  }
} ///:~
```

505

PQV类模板随后采用了与STL的**priority_queue**相同的形式，但是拥有一个附加的成员函数**getVector( )**，该函数创建了一个从**PQV**中（这意味着它已经是一个堆）复制来的新的**vector**。然后它对那些副本进行排序（而**PQV**的**vector**并没有受到影响），并且将序列的顺序逆转，所以在遍历新的**vector**时产生了与从一个优先队列中弹出元素的操作等效的结果。

可以观察到，如果采用在**PriorityQueue4.cpp**中使用的从**priority_queue**中派生的方法来实现上述技术的话，可以得到更简洁的代码：

```
//: C07:PriorityQueue8.cpp
// A more compact version of PriorityQueue7.cpp.
#include <algorithm>
#include <cstdlib>
#include <ctime>
#include <iostream>
#include <iterator>
#include <queue>
using namespace std;

template<class T> class PQV : public priority_queue<T> {
public:
  typedef vector<T> TVec;
```

```
  TVec getVector() {
    TVec r(this->c.begin(),this->c.end());
    // c is already a heap
    sort_heap(r.begin(), r.end(), this->comp);
    // Put it into priority-queue order:
    reverse(r.begin(), r.end());
    return r;
  }
};

int main() {
  PQV<int> pqi;
  srand(time(0));
  for(int i = 0; i < 100; i++)
    pqi.push(rand() % 25);
  const vector<int>& v = pqi.getVector();
  copy(v.begin(), v.end(),
    ostream_iterator<int>(cout, " "));
  cout << "\n-----------" << endl;
  while(!pqi.empty()) {
    cout << pqi.top() << ' ';
    pqi.pop();
  }
} ///:~
```

506

以上简洁的解决方案，加上它保证用户不会得到一个处在未经排序状态的**vector**，使它变得最简单且最令人期待。惟一潜在的问题就是成员函数**getVector( )**采用传值的方式返回**vector<T>**，这可能在参数类型**T**的值比较复杂时会引发某些经常性的开销问题。

7.9 持有二进制位

因为C是一种旨在“接近硬件”的语言，但很多人都发现一个令人沮丧的现象，那就是对于数字没有一种固有的二进制的表示方法。当然，有十进制和十六进制（还可以容忍，仅仅因为它们能较容易地在你的头脑中形成一组二进制位），但是八进制呢？哎呀！每当你阅读正在尝试对其进行编程的芯片的说明书时，这些说明书不会使用八进制甚至十六进制来描述芯片的寄存器——他们使用二进制。然而，C不让用**0b01011101**这样的表示方法，很明显，对于接近硬件的语言来说，这才是最好的解决方案。

虽然在C++中仍然没有固有的表示二进制的方法，由于两个类的增加：二进制位集合**bitset**和逻辑向量**vector<bool>**而使得情况有所好转，它们都被设计用来操纵一组开/关值。[⊖]这些类型之间主要的不同是：

507

- 每个**bitset**持有一个固定位数的二进制位（**bit**）。用户在**bitset**的模板参数中建立二进制位的位数。像正常**vector**一样，**vector<bool>**可以动态地扩展为持有任意数目的**bool**值。
- **bitset**模板是为了在操纵二进制位时提高性能的目的而设计，因此并不是一个“正常的”STL容器。因此，它没有迭代器。作为一个模板参数，二进制位的位数目在编译时就已经知道了，并且允许将底层的整型数组存储在运行时的栈上。另一方面，**vector<bool>**容器是**vector**的一个特化，所以有一个普通**vector**的所有操作——该特化只是被设计用来提高**bool**数据的空间使用率。

在**bitset**和**vector<bool>**之间没有琐碎的转换，这意味着它们就是为了完全不同的目的

⊖ Chuck设计并提供了最初的关于**bitset**或**bitstring**、以及**vector<bool>**最早的参考实现，当时，即20世纪90年代早期他就是C++标准委员会中的一个活跃的成员。

而设计的。此外，它们都不是传统的“STL容器”。**bitset**模板类拥有一个面向二进制位层次的操作接口，绝不与到目前为止本教材中所讨论的STL容器类似。**vector**的特化**vector<bool>**类似于类-STL容器，但与将要在下面讨论的内容也是不同的。

7.9.1 bitset<n>

bitset作为模板接受一个无符号整型模板参数，该参数用来表示二进制位的位数。因此，**bitset<10>**与**bitset<20>**相比就是两种不同的类型，不能在它们两个之间进行比较、赋值等操作。

一个**bitset**以有效的形式提供了最一般的用于二进制位操作的方式。然而，每个**bitset**通过合理地将一组二进制位封装到一个整型数组中来实现（典型的如**unsigned long**，它至少包含32个二进制位）。另外，从一个**bitset**到一个数的惟一转化就是将其转化为一个**unsigned long**（通过函数**to_ulong()**）。

508

下面的例子测试了几乎所有**bitset**的功能（这里未介绍那些多余的或不重要的操作）。读者可以在每个打印输出的右边看到对**bitset**输出的描述，因此，可以将这些输出描述与它们原来的值进行比较。如果读者到现在为止还不了解二进制位操作方式的话，运行这个程序将会很有帮助。

```
//: C07:BitSet.cpp {-bor}
// Exercising the bitset class.
#include <bitset>
#include <climits>
#include <cstdlib>
#include <ctime>
#include <cstddef>
#include <iostream>
#include <string>
using namespace std;

const int SZ = 32;
typedef bitset<SZ> BS;

template<int bits> bitset<bits> randBitset() {
  bitset<bits> r(rand());
  for(int i = 0; i < bits/16 - 1; i++) {
    r <<= 16;
    // "OR" together with a new lower 16 bits:
    r |= bitset<bits>(rand());
  }
  return r;
}

int main() {
  srand(time(0));
  cout << "sizeof(bitset<16>) = "
       << sizeof(bitset<16>) << endl;
  cout << "sizeof(bitset<32>) = "
       << sizeof(bitset<32>) << endl;
  cout << "sizeof(bitset<48>) = "
       << sizeof(bitset<48>) << endl;
  cout << "sizeof(bitset<64>) = "
       << sizeof(bitset<64>) << endl;
  cout << "sizeof(bitset<65>) = "
       << sizeof(bitset<65>) << endl;
  BS a(randBitset<SZ>()), b(randBitset<SZ>());
  // Converting from a bitset:
  unsigned long ul = a.to_ulong();
  cout << a << endl;
  // Converting a string to a bitset:
  string cbits("111011010110111");
```

509

```
cout << "as a string = " << cbits <<endl;
cout << BS(cbits) << " [BS(cbits)]" << endl;
cout << BS(cbits, 2) << " [BS(cbits, 2)]" << endl;
cout << BS(cbits, 2, 11) << " [BS(cbits, 2, 11)]"<< endl;
cout << a << " [a]" << endl;
cout << b << " [b]" << endl;
// Bitwise AND:
cout << (a & b) << " [a & b]" << endl;
cout << (BS(a) &= b) << " [a &= b]" << endl;
// Bitwise OR:
cout << (a | b) << " [a | b]" << endl;
cout << (BS(a) |= b) << " [a |= b]" << endl;
// Exclusive OR:
cout << (a ^ b) << " [a ^ b]" << endl;
cout << (BS(a) ^= b) << " [a ^= b]" << endl;
cout << a << " [a]" << endl; // For reference
// Logical left shift (fill with zeros):
cout << (BS(a) <<= SZ/2) << " [a <<= (SZ/2)]" << endl;
cout << (a << SZ/2) << endl;
cout << a << " [a]" << endl; // For reference
// Logical right shift (fill with zeros):
cout << (BS(a) >>= SZ/2) << " [a >>= (SZ/2)]" << endl;
cout << (a >> SZ/2) << endl;
cout << a << " [a]" << endl; // For reference
cout << BS(a).set() << " [a.set()]" << endl;
for(int i = 0; i < SZ; i++)
  if(!a.test(i)) {
    cout << BS(a).set(i)
         << " [a.set(" << i <<")]" << endl;
    break; // Just do one example of this
  }
cout << BS(a).reset() << " [a.reset()]"<< endl;
for(int j = 0; j < SZ; j++)
  if(a.test(j)) {
    cout << BS(a).reset(j)
         << " [a.reset(" << j <<")]" << endl;
    break; // Just do one example of this
  }
cout << BS(a).flip() << " [a.flip()]" << endl;
cout << ~a << " [~a]" << endl;
cout << a << " [a]" << endl; // For reference
cout << BS(a).flip(1) << " [a.flip(1)]"<< endl;
BS c;
cout << c << " [c]" << endl;
cout << "c.count() = " << c.count() << endl;
cout << "c.any() = "
     << (c.any() ? "true" : "false") << endl;
cout << "c.none() = "
     << (c.none() ? "true" : "false") << endl;
c[1].flip(); c[2].flip();
cout << c << " [c]" << endl;
cout << "c.count() = " << c.count() << endl;
cout << "c.any() = "
     << (c.any() ? "true" : "false") << endl;
cout << "c.none() = "
     << (c.none() ? "true" : "false") << endl;
// Array indexing operations:
c.reset();
for(size_t k = 0; k < c.size(); k++)
  if(k % 2 == 0)
    c[k].flip();
cout << c << " [c]" << endl;
c.reset();
```

510

```
  // Assignment to bool:
  for(size_t ii = 0; ii < c.size(); ii++)
    c[ii] = (rand() % 100) < 25;
  cout << c << " [c]" << endl;
  // bool test:
  if(c[1])
    cout << "c[1] == true";
  else
    cout << "c[1] == false" << endl;
} ///:~
```

为产生有趣的随机**bitset**，在程序中创建了函数**randBitset( )**。该函数将每16个随机二进制位向左移动，直到**bitset**（其尺寸大小在函数中已经被模板化了）被填满为止，以此来演示**operator<<=**的使用。用**operator|=**将产生的数字和每组新的16位二进制数结合起来。

main( )函数首先显示了一个**bitset**单元的大小。如果它小于32位，**sizeof**就产生4（4字节 = 32位，其最大实现是一个**long**型的大小。如果它在32到64之间，则需要两个**long**型数，大于64需要3个**long**型，等等。因此，为了最有效地利用空间，所使用的二进制位数量应在适宜个数的固有**long**型数表示的范围中。然而，要注意的是，对该对象不存在额外的开销——就像是在为一个**long**型数进行手工译码一样。

511

虽然除了**to_ulong( )**之外没有其他的从**bitset**进行数字转换的函数，但是有一个流插入器**stream inserter**，它产生一个包含1和0的**string**，这个字符串可以和实际的**bitset**一样长。

虽然仍然没有用于表示二进制数的基本的格式，但是**bitset**支持最贴近的二进制表示形式：由1和0与在右边的最低有效位（least-significant bit，lsb）一起组成的一个**string**。3个构造函数演示创造一个完整的**string**、在第2个字符开始的**string**以及从第2个字符开始到第11个字符结束的**string**、可以使用**operator<<** 从一个**bitset**写到一个输出流**ostream**，它以1和0的方式出现。也可以使用**operator>>**从一个输入流**istream**中读入到**bitset**（在这里没有显示）。

必须注意，**bitset**仅有3个非成员运算符：与（&）、或（|）和异或（^）。其中的每个都创建一个新的**bitset**作为其返回值。在没有创建暂时值的地方，全部选择更有效率的**&=**、**|=**等形式的成员运算符。然而，这些形式改变了**bitset**的值（这个值在上面例子的大多数检测中即**a**）。为避免发生这种情况，通过调用**a**的拷贝构造函数创建一个作为左值的临时对象；这就是为什么**BS(a)**的形式如此。每次测试的结果都显示出来，有时候**a**被重新打印出来从而更容易以它进行参照。

在程序运行的时候，例子的其余部分有自我解释；如果没有，读者可以在自己使用的编译器的文档中或者在本章较早提到的其他文档中查找有关细节。

7.9.2 vector<bool>

容器**vector<bool>**是**vector**模板的一个特化。一个标准的**bool**变量至少需要一个字节，但是因为一个**bool**型只有两个状态，所以理想的**vector<bool>**实现是这样的，每一个**bool**值仅需一个二进制位来表示。因为典型的库实现将一组二进制位封装进整型数组之内，所以迭代器必须特殊定义并且不能是一个指向**bool**型的指针。

512

用于**vector<bool>**的位操纵函数比**bitset**的那些函数受到更多的限制。在**vector**中已有的这些成员函数基础上添加的惟一成员函数就是**flip( )**，用于使所有的位取反。它没有**bitset**中的**set( )**或**reset()**。当使用**operator[ ]**时，就会送回一个**vector<bool>::reference**类型的对象，该对象也有一个用于对个别的位取反的成员函数**flip( )**。

```
//: C07:VectorOfBool.cpp
// Demonstrate the vector<bool> specialization.
#include <bitset>
#include <cstddef>
#include <iostream>
#include <iterator>
#include <sstream>
#include <vector>
using namespace std;

int main() {
  vector<bool> vb(10, true);
  vector<bool>::iterator it;
  for(it = vb.begin(); it != vb.end(); it++)
    cout << *it;
  cout << endl;
  vb.push_back(false);
  ostream_iterator<bool> out(cout, "");
  copy(vb.begin(), vb.end(), out);
  cout << endl;
  bool ab[] = { true, false, false, true, true,
    true, true, false, false, true };
  // There's a similar constructor:
  vb.assign(ab, ab + sizeof(ab)/sizeof(bool));
  copy(vb.begin(), vb.end(), out);
  cout << endl;
  vb.flip(); // Flip all bits
  copy(vb.begin(), vb.end(), out);
  cout << endl;
  for(size_t i = 0; i < vb.size(); i++)
    vb[i] = 0; // (Equivalent to "false")
  vb[4] = true;
  vb[5] = 1;
  vb[7].flip(); // Invert one bit
  copy(vb.begin(), vb.end(), out);
  cout << endl;
  // Convert to a bitset:
  ostringstream os;
  copy(vb.begin(), vb.end(),
    ostream_iterator<bool>(os, ""));
  bitset<10> bs(os.str());
  cout << "Bitset:" << endl << bs << endl;
} ///:~
```

513

这个例子中的最后一部分创造了一个**vector<bool>**，通过先将它转换成一个仅包含0和1的**string**，再转换成为一个**bitset**。这里必须在编译时就知道**bitset**的大小。可以看出来，这个转换并不是基于常规的那种操作。

某些其他容器保证提供的功能不见了，**vector<bool>**特化给人的感觉是一种“有缺陷的”STL容器。比如，在其他的容器持有如下关系：

```
// Let c be an STL container other than vector<bool>:
T& r = c.front();
T* p = &*c.begin();
```

对于所有其他的容器，**front( )**函数产生一个左值（某个对象能获得一个指向它的非常量引用），函数**begin( )**必须产生某个对象的解析，并且得到其地址。因为二进制位是不可寻址的，所以上面的两个函数不可能用于处理持有二进制位的容器。**vector<bool>**和**bitset**两者都使用一个代理类（嵌套的**reference**引用类，之前提到过）在必要的时候读取和设置二进制位。

7.10 关联式容器

set、**map**、**multiset**和**multimap**被称为关联式容器（associative container），因为它们将关键字与值关联起来。至少**map**和**multimap**将关键字与值关联在一起，读者可以将一个**set**看成是没有值的**map**，它只有关键字（事实上，它们可以以这样的方式实现），**multiset**和**multimap**之间也有同样的关系。因此，由于结构的相似性，**set**和**multiset**都被归类为关联式容器。

关联式容器最重要的基本操作就是将对象放进容器。并且在**set**的情况下，要查看该对象是否已经在集合中；在**map**的情况下，需要先查看关键字是否已经在**map**中，如果存在，就需要为那个关键字设置关联的值。在这个主题上有很多变化，但是那是基本的概念。下面的例子显示了这些基本操作：

514

```
//: C07:AssociativeBasics.cpp {-bor}
// Basic operations with sets and maps.
//{L} Noisy
#include <cstddef>
#include <iostream>
#include <iterator>
#include <map>
#include <set>
#include "Noisy.h"
using namespace std;

int main() {
  Noisy na[7];
  // Add elements via constructor:
  set<Noisy> ns(na, na + sizeof na/sizeof(Noisy));
  Noisy n;
  ns.insert(n); // Ordinary insertion
  cout << endl;
  // Check for set membership:
  cout << "ns.count(n)= " << ns.count(n) << endl;
  if(ns.find(n) != ns.end())
    cout << "n(" << n << ") found in ns" << endl;
  // Print elements:
  copy(ns.begin(), ns.end(),
    ostream_iterator<Noisy>(cout, " "));
  cout << endl;
  cout << "\n-----------" << endl;
  map<int, Noisy> nm;
  for(int i = 0; i < 10; i++)
    nm[i]; // Automatically makes pairs
  cout << "\n-----------" << endl;
  for(size_t j = 0; j < nm.size(); j++)
    cout << "nm[" << j <<"] = " << nm[j] << endl;
  cout << "\n-----------" << endl;
  nm[10] = n;
  cout << "\n-----------" << endl;
  nm.insert(make_pair(47, n));
  cout << "\n-----------" << endl;
  cout << "\n nm.count(10)= " << nm.count(10) << endl;
  cout << "nm.count(11)= " << nm.count(11) << endl;
  map<int, Noisy>::iterator it = nm.find(6);
  if(it != nm.end())
    cout << "value:" << (*it).second
         << " found in nm at location 6" << endl;
  for(it = nm.begin(); it != nm.end(); it++)
    cout << (*it).first << ":" << (*it).second << ", ";
```

515

```
  cout << "\n-----------" << endl;
} ///:~
```

这里使用两个迭代器来创建**set<Noisy>**对象**ns**，使其进入一个**Noisy**对象的数组之内。但是也有一个默认的构造函数和一个拷贝构造函数，可以传入一个对象以便提供另一种做比较的方案。**set**和**map**两者都有一个成员函数**insert( )**用于向其中放入对象，可以用两种方式检查来看看对象是否已经存在于相应的关联式容器中。当给定一个关键字时，成员函数**count( )**会告之那个关键字在容器中存在多少次。（在**set**或者**map**中只能是0或者1，但是在**multiset**和**multimap**中则可能多于一个）。成员函数**find( )**将会产生一个指向首次出现（在**set**和**map**中则是惟一出现）给定关键字的元素的迭代器，或者如果找不到该关键字，将产生指向超越末尾的迭代器。所有的关联式容器都有**count( )**和**find( )**成员函数，这是很有意义的。这些关联式容器也都有成员函数**lower_bound( )**、**upper_bound( )**和**equal_range( )**，它们仅仅对**multiset**和**multimap**有意义，正如读者所见。（但是不要试图去搞清楚它们对**set**和**map**到底有什么用，因为它们被设计用来处理某个范围的重复关键字，这在**set**和**map**容器中是不允许的。）

设计一个**operator[ ]**总是多少有点进退两难。因为它被有意地作为一个数组索引操作来对待，人们在使用前一般不会想到对其进行测试。但是，假如决定将索引设置为超出数组范围以外的位置时会发生什么事情？一种选择是抛出一个异常，但是对于一个**map**，“在数组范围以外的位置进行索引”意味着希望在那个位置创建一个新条目，这就是STL **map**的处理方式。在创建**map<int,Noisy> nm**之后的第1个**for**循环使用**operator[ ]**来“查找”对象，但实际上这是在创建新的**Noisy**对象！如果使用**operator[ ]**查寻一个值而它又不存在的话，这个**map**就会创建一个新的关键字-值对（为这个值使用默认的构造函数）。这意味着，如果实际上仅想要查寻某个对象并不想创建一个新条目，就必须使用成员函数**count( )**（看这个对象是否在那里）或者**find( )**（得到指向它的迭代器）。

与**for**循环一起使用运算符**operator[ ]**来打印容器中的值会有许多问题。首先，它需要整数关键字（在这里恰好是这样）。其次且更糟的是，如果所有的关键字都不是连续的，那么将会完成从0到整个容器的大小全部都进行计算，如果某些点没有关键字-值对存在，系统将会自动创建它们并会错过一些较高值的关键字。最后，如果观察**for**循环的输出，将会看到其工作非常繁忙。起先读者会相当迷惑，一个简单的查寻怎么会出现如此多的构造与析构呢？只有当看到**map**模板中关于**operator[ ]**的代码时答案才清楚为什么，这段代码如下所示： 516

```
mapped_type& operator[] (const key_type& k) {
  value_type tmp(k,T());
  return (*((insert(tmp)).first)).second;
}
```

函数**map::insert( )**接受一个关键字-值对，如果在映像中已有与给定的关键字在一起的条目，就什么也不做——否则它为该关键字插入一个条目。在两者之中任一情况下，它返回一个新的关键字-值对，该关键字-值对的第1个元素持有指向被插入对的迭代器，如果发生了插入操作，该对的第2个元素持有值为真。成员**first**和**second**分别给出了关键字和值，因为**map::value_type**实际上只是一个为**std::pair**进行类型定义的**typedef**：

```
typedef pair<const Key, T> value_type;
```

读者已经在前面看到了**std::pair**模板。它是两个独立类型值的简单持有者，就像在其定义中所看到的那样：

```
template<class T1, class T2> struct pair {
  typedef T1 first_type;
  typedef T2 second_type;
  T1 first;
  T2 second;
  pair();
  pair(const T1& x, const T2& y) : first(x), second(y) {}
  // Templatized copy-constructor:
  template<class U, class V> pair(const pair<U, V> &p);
};
```

pair模板类非常有用，特别是想要一个函数返回两个对象的时候（因为一个**return**语句只能返回一个对象）。为了创建一个关键字-值对**pair**，甚至还有一个快捷的称为**make_pair( )**的函数，它用在**AssociativeBasics.cpp**中。

517

追溯上面执行的各个步骤，**map::value_type**是**map**的一个关键字-值对**pair**——实际上，它是**map**的一个条目。但是要注意，**pair**由值封装它的对象，这意味着将对象装入**pair**之内，拷贝构造是必须的。因此，在**map::operator[ ]**的**tmp**创建过程中，对于每个**pair**中的对象将包括至少一个拷贝构造函数调用和一个析构函数调用。在这里，可以很容易地完成这些操作，因为关键字是**int**型的。但是，如果想要看看根据**map::operator[ ]**的活动方式到底能产生什么样的结果，请运行下面这个程序：

```
//: C07:NoisyMap.cpp
// Mapping Noisy to Noisy.
//{L} Noisy
#include <map>
#include "Noisy.h"
using namespace std;

int main() {
  map<Noisy, Noisy> mnn;
  Noisy n1, n2;
  cout << "\n--------" << endl;
  mnn[n1] = n2;
  cout << "\n--------" << endl;
  cout << mnn[n1] << endl;
  cout << "\n--------" << endl;
} ///:~
```

读者将会看到，插入和查寻两者都会产生很多额外的对象，这是因为**tmp**对象的创建。如果回过来看**map::operator[ ]**，就会看到第2行调用了**insert( )**，并向其传递**tmp**——即**operator[ ]**每次都进行了插入操作。函数**insert( )**的返回值是一种不同的**pair**类型，其**first**是一个指向刚刚插入的关键字-值对的迭代器，而**second**则是一个表示在该处是否发生了插入操作的**bool**值。可以看到，**operator[ ]**抓取了**first**（迭代器），对其进行解析以产生**pair**，然后返回**second**，即该位置上的值。

因此，从上面的描述看来，**map**具有“如果在那里没有条目的话就创建一个”的奇妙行为，但是从下面（具体操作）来看，就是在使用**map::operator[ ]**时总是得到很多额外的对象创建和析构操作。幸运的是，**AssociativeBasics.cpp**也演示了如何减少插入和删除操作的开销。如果不需要它，尽量避免使用**operator[ ]**。成员函数**insert( )**比**operator[ ]**稍微更有效些。对于一个**set**，仅仅持有一个对象，而对于**map**来说，持有的是关键字-值对；所以**insert( )**需要一个**pair**作为其参数。这里就是**make_pair( )**派得上用场的地方，就像在程序中所能看到的那样。

518

为了在一个**map**中查寻对象，可以使用**count( )**来查看这个关键字是否在**map**中，或者

可以用**find( )**产生一个直接指向关键字-值对的迭代器。再次强调，因为**map**包含**pair**，这就是在解析它的时候为什么会产生迭代器的原因，所以选择**first**和**second**作为其参数。在运行**AssociativeBasics.cpp**的时候，读者将会注意到，使用迭代器的方法不会产生额外的对象的构造和析构操作。然而，就易编写或者易阅读的面向对象编码要求而言，这是不可取的。

7.10.1 用于关联式容器的发生器和填充器

在使用**<algorithm>**中的**fill( )**、**fill_n( )**、**generate( )**和**generate_n( )**函数模板向序列容器（**vector**、**list**和**deque**）中填充数据时，已经看到了它们是多么的有用。然而，它们的实现都使用**operator=** 赋值的方式将值放进序列容器，而向关联式容器中添加对象的方式是使用它们各自的成员函数**insert( )**。因此，在尝试与关联式容器一起使用“填充（fill）”和“产生（generate）”函数的时候，默认的“赋值”行为将会产生问题。

一个解决方案就是复制“填充”和“产生”函数，创建新的一种能用于关联式容器的函数。结果是，只有**fill_n( )**和**generate_n( )**函数能被复制（**fill( )**和**generate( )**复制序列，这对于关联式容器来说没有什么意义），但是这个工作是相当简单的，因为可以利用头文件**<algorithm>**作为工作的根据：

```
//: C07:assocGen.h
// The fill_n() and generate_n() equivalents
// for associative containers.
#ifndef ASSOCGEN_H
#define ASSOCGEN_H

template<class Assoc, class Count, class T>
void assocFill_n(Assoc& a, Count n, const T& val) {
  while(n-- > 0)
    a.insert(val);
}

template<class Assoc, class Count, class Gen>
void assocGen_n(Assoc& a, Count n, Gen g) {
  while(n-- > 0)
    a.insert(g());
}
#endif // ASSOCGEN_H ///:~
```

519

读者可以看到，没有使用迭代器，容器类自身被传递了（当然，通过使用引用）。

这段代码演示了两条有价值的经验教训。第1条就是，如果有什么需要的工作某个算法不能做，可以复制与其最接近的算法，并且修改它以满足需要。在STL头文件中有很多手到擒来例子，从这一点来说，大多数工作实际上已经完成了。

第2条经验教训进一步指出：如果观察的时间足够长，就会发现在STL中有一种方法来做这个工作，而不必再发明任何新的东西。当前的问题可以用**insert_iterator**（调用**inserter( )**而产生）来解决，它调用**insert( )**而非**operator=**以便在容器中放入对象。这不是仅仅对**front_insert_iterator**或者**back_insert_iterator**的变更，因为那些迭代器使用各自的**push_front( )**和**push_back( )**。每个插入迭代器都因为其用于插入操作的成员函数各具各的优点而不尽相同，**insert( )**正是我们所需要的一个函数。这里有一个演示显示进行填充和产生**map**和**set**两个容器的例子。（它也可以用于**multiset**和**multimap**。）首先，创建一些模板化的发生器。（这似乎像是有点过分，但在需要它们的时候，用户绝不会知道。为此，它们被放置在一个头文件中。）

```
//: C07:SimpleGenerators.h
// Generic generators, including one that creates pairs.
#include <iostream>
#include <utility>

// A generator that increments its value:
template<typename T> class IncrGen {
  T i;
public:
  IncrGen(T ii) : i(ii) {}
  T operator()() { return i++; }
};

// A generator that produces an STL pair<>:
template<typename T1, typename T2> class PairGen {
  T1 i;
  T2 j;
public:
  PairGen(T1 ii, T2 jj) : i(ii), j(jj) {}
  std::pair<T1,T2> operator()() {
    return std::pair<T1,T2>(i++, j++);
  }
};

namespace std {
// A generic global operator<< for printing any STL pair<>:
template<typename F, typename S> ostream&
operator<<(ostream& os, const pair<F,S>& p) {
  return os << p.first << "\t" << p.second << endl;
}
} ///:~
```

520

两个发生器都希望T可以进行增1操作，无论用什么来进行初始化，它们都仅使用**operator++**来产生新的值。**PairGen**创建一个STL **pair**对象作为其返回值，这也就是为什么可以使用**insert( )**向一个**map**或者**multimap**中放入对象的原因。

最后的函数是个一般用于输出流**ostream**的操作符**operator<<**，假定**pair**的每一个元素都支持流操作符**operator<<**，因此任何**pair**都能被打印。（这是在第5章中讨论过的名字空间**std**中奇怪的名字查寻的推论，在本章**Thesaurus.cpp**之后将再一次解释。）如下所示，这允许用**copy( )**来输出**map**：

```
//: C07:AssocInserter.cpp
// Using an insert_iterator so fill_n() and generate_n()
// can be used with associative containers.
#include <iterator>
#include <iostream>
#include <algorithm>
#include <set>
#include <map>
#include "SimpleGenerators.h"
using namespace std;

int main() {
  set<int> s;
  fill_n(inserter(s, s.begin()), 10, 47);
  generate_n(inserter(s, s.begin()), 10,
    IncrGen<int>(12));
  copy(s.begin(), s.end(),
    ostream_iterator<int>(cout, "\n"));
  map<int, int> m;
  fill_n(inserter(m, m.begin()), 10, make_pair(90,120));
```

521

```
  generate_n(inserter(m, m.begin()), 10,
    PairGen<int, int>(3, 9));
  copy(m.begin(), m.end(),
    ostream_iterator<pair<int,int> >(cout,"\n"));
} ///:~
```

传递给**inserter**的第2个参数是一个迭代器，它是最佳化的，暗示可以帮助较快地进行插入（而不总是从底层树形结构的根开始进行搜索）。因为**insert_iterator**可以用于很多不同类型的容器，对于非-关联式容器来说，它还有更多的意义——它是必需的。

注意**ostream_iterator**是如何被创建来输出一个**pair**的。如果未创建**operator<<**，它不会起什么作用。因为它是一个模板，它将自动地为**pair<int,int>**进行实例化。

7.10.2 不可思议的映像

一个普通的数组使用一个整数值来对连续排列的某种类型的元素集进行索引。**map**是一个关联式数组（associative array），这意味着，按照类数组的方式将一个对象与另一个对象关联到一起。而不是像处理普通数组的方式一样使用一个数字来选择某个数组元素，在这里利用一个对象来进行查寻！下面的例子对一个文本文件中的单词进行计数，因此索引是一个代表单词的**string**对象，被查寻的值就是保存字串（单词）总数的对象。

在一个类似于**vector**或**list**的单项容器中，仅保存着一样东西。但是在一个**map**中，将会得到两样东西：关键字（key）（用它来进行查寻，就像在**mapname[key]**中）以及作为对关键字进行查寻得到的结果值。如果只希望遍历整个**map**并列出每一个关键字-值对的话，可以使用一个迭代器，它在解析时产生一个包含了关键字及其值的**pair**对象。可以通过选择**first**和**second**访问**pair**对象中的成员。

522

这种将两项一起进行打包的相同思想也用于将元素插入**map**的操作，但是包含了关键字及值的**pair**是作为**map**实例化的一部分来进行创建的，该**pair**称为**value_type**。所以插入新元素操作的一个选择就是创建一个**value_type**对象，以适当的对象装载它，然后为**map**调用**insert( )**成员函数进行插入操作。下面的例子使用了上述的**map**的特性：如果尝试向**operator[ ]**传递一个关键字来查找某个对象，当那个对象不存在时，**operator[ ]**将会自动使用值对象的默认构造函数插入一个新的关键字-值对。以这种思想为基础，现在考虑一个单词计数程序的实现：

```
//: C07:WordCount.cpp
// Count occurrences of words using a map.
#include <iostream>
#include <fstream>
#include <map>
#include <string>
#include "../require.h"
using namespace std;

int main(int argc, char* argv[]) {
  typedef map<string, int> WordMap;
  typedef WordMap::iterator WMIter;
  const char* fname = "WordCount.cpp";
  if(argc > 1) fname = argv[1];
  ifstream in(fname);
  assure(in, fname);
  WordMap wordmap;
  string word;
  while(in >> word)
    wordmap[word]++;
  for(WMIter w = wordmap.begin(); w != wordmap.end(); w++)
```

```
    cout << w->first << ": " << w->second << endl;
} ///:~
```

这个例子显示了零初始化（zero-initialization）的能力。考虑程序代码中的这行：

523
```
wordmap[word]++;
```

这个将**int**与**word**关联在一起的表达式进行增1操作。如果map映像中没有这样的一个单词，则作为该单词的关键字-值对就会自动地插入到**map**映像中，并调用返回值为0的伪构造函数**int( )**并将其值初始化为0。

打印整个列表需要一个能够遍历该列表的迭代器。（这里不存在用于**map**的快捷方式的**copy( )**，除非需要再为**map**中的**pair**编写一个**operator<<**。）如前所述，解析该迭代器会产生一个**pair**对象，其中**first**成员为关键字，**second**成员为其值。

如果希望找到为某个特定单词的计数，可以使用数组的索引操作符，如下所示：

```
cout << "the: " << wordmap["the"] << endl;
```

可以看到，**map**的主要优点之一就是其清楚的语法；一个关联式数组对于读者来说是直观的。（然而要注意的是，如果单词“the”已不在**wordmap**中，一个新的条目就会被创建！）

7.10.3 多重映像和重复的关键字

多重映像**multimap**是一个能包含重复的关键字的**map**。起初这可能似乎是一个奇怪的想法，但令人惊讶的这种情况却经常发生。比如电话号码簿，同一个名字可以有很多个条目。

假定读者正在监视野生动植物，需要跟踪每一种有斑点的动物出现的时间和地点。因此，你就可能看到很多同一种类的动物，它们都在不同的时间和不同的地点出现。因此，如果将动物的类型作为关键字，就需要一个**multimap**。如下所示：

```
//: C07:WildLifeMonitor.cpp
#include <algorithm>
#include <cstdlib>
#include <cstddef>
#include <ctime>
#include <iostream>
#include <iterator>
#include <map>
#include <sstream>
#include <string>
#include <vector>
using namespace std;

class DataPoint {
  int x, y; // Location coordinates
  time_t time; // Time of Sighting
public:
  DataPoint() : x(0), y(0), time(0) {}
  DataPoint(int xx, int yy, time_t tm) :
    x(xx), y(yy), time(tm) {}
  // Synthesized operator=, copy-constructor OK
  int getX() const { return x; }
  int getY() const { return y; }
  const time_t* getTime() const { return &time; }
};

string animal[] = {
  "chipmunk", "beaver", "marmot", "weasel",
  "squirrel", "ptarmigan", "bear", "eagle",
  "hawk", "vole", "deer", "otter", "hummingbird",
};
const int ASZ = sizeof animal/sizeof *animal;
```

```
vector<string> animals(animal, animal + ASZ);

// All the information is contained in a
// "Sighting," which can be sent to an ostream:
typedef pair<string, DataPoint> Sighting;

ostream&
operator<<(ostream& os, const Sighting& s) {
  return os << s.first << " sighted at x= "
    << s.second.getX() << ", y= " << s.second.getY()
    << ", time = " << ctime(s.second.getTime());
}

// A generator for Sightings:
class SightingGen {
  vector<string>& animals;
  enum { D = 100 };
public:
  SightingGen(vector<string>& an) : animals(an) {}
  Sighting operator()() {
    Sighting result;
    int select = rand() % animals.size();
    result.first = animals[select];
    result.second = DataPoint(
      rand() % D, rand() % D, time(0));
    return result;
  }
};

// Display a menu of animals, allow the user to
// select one, return the index value:
int menu() {
  cout << "select an animal or 'q' to quit: ";
  for(size_t i = 0; i < animals.size(); i++)
    cout <<'['<< i <<']'<< animals[i] << ' ';
  cout << endl;
  string reply;
  cin >> reply;
  if(reply.at(0) == 'q') return 0;
  istringstream r(reply);
  int i;
  r >> i; // Converts to int
  i %= animals.size();
  return i;
}

int main() {
  typedef multimap<string, DataPoint> DataMap;
  typedef DataMap::iterator DMIter;
  DataMap sightings;
  srand(time(0));  // Randomize
  generate_n(inserter(sightings, sightings.begin()),
    50, SightingGen(animals));
  // Print everything:
  copy(sightings.begin(), sightings.end(),
    ostream_iterator<Sighting>(cout, ""));
  // Print sightings for selected animal:
  for(int count = 1; count < 10; count++) {
    // Use menu to get selection:
    // int i = menu();
    // Generate randomly (for automated testing):
    int i = rand() % animals.size();
    // Iterators in "range" denote begin, one
```

525

```
    // past end of matching range:
    pair<DMIter, DMIter> range =
      sightings.equal_range(animals[i]);
    copy(range.first, range.second,
      ostream_iterator<Sighting>(cout, ""));
  }
} ///:~
```

526

将观察到的所有数据都封装到一个**DataPoint**类中，该类非常简单足以使用综合赋值和拷贝构造函数来对其进行操作。它用标准C库的时间函数来记录观察的时间。

在**string**数组**animal**中，要注意的是在初始化期间，**char***构造函数将会自动地调用，这就使得对**string**数组进行初始化变得相当方便。因为在一个**vector**中较易使用动物的名字，所以在计算好数组的长度后，使用构造函数**vector(iterator,iterator)**来初始化一个**vector<string>**。

用于构建**Sighting**的关键字-值对是表示动物类型名称的**string**。**DataPoint**表示观察到该动物的时间和地点。标准的**pair**模板将这两个类型关联起来，并且使用类型定义产生**Sighting**类型。然后为**Sighting**创建一个**ostream operator<<**；这将允许对一个存储了**Sighting**的**map**或**multimap**进行迭代并显示它。

SightingGen产生在随机数据点随机观察到的数据用于测试。它有一个普通的**operator( )**，这对于函数对象来说是必需的，但是它还有一个构造函数，用于获得和存储一个引用到**vector<string>**，这就是前面提到的存储动物名称的地方。

DataMap是一个包含了**string-DataPoint**对的**multimap**，这意味着它存储**Sighting**对象。用**generate_n( )**产生的50个**Sighting**对象来填充该**DataMap**，并且显示它们。（注意，因为存在一个接受**Sighting**的**operator<<**，所以可以创建一个输出流迭代器**ostream_iterator**。）此时就可以请用户选择他们想要查看所有观察记录中的哪一种动物的情况。如果键入**q**程序就会退出，但是如果选择一个动物的编号，就会调用**equal_range( )**成员函数。这将会返回一个指向匹配对**pair**集的起始元素的迭代器（**DMIter**）和一个指向该匹配对**pair**集的超越末尾的迭代器。因为从一个函数中只能返回一个对象，因此**equal_range( )**使用了**pair**。因为**range**对拥有匹配集的起始和终止迭代器，所以这些迭代器可以在**copy( )**函数中用来打印对某种特定类型动物的所有观察记录。

527

7.10.4 多重集合

读者已经知道**set**仅允许插入每个值的惟一一个对象。而**multiset**看起来则比较古怪，因为它允许插入每个值的多个对象。这似乎违反了“集合”的完整的思想，读者可能会问，“‘它’在这个集合中吗？”如果集合中存在着多个“它”，将意味着什么呢？

想一想就会明白，如果这些重复的对象确实完全相同，在一个集合中有多个相同值的对象意义并不大（对那些出现的对象进行计数的情况可能是一个例外，但是，就像在本章较早时看到的，这个问题可以使用另一种更优雅的方法来处理）。因此，每个重复的对象都应该有什么地方“不同于”其他的重复对象——最有可能是在比较期间那些未被用作关键字计算的不同的状态信息。也就是说，通过比较操作这些对象看起来相同，但是它们却包括一些不同的内部状态。

像任何STL容器都必须对其元素进行排序一样，**multiset**模板在默认情况下使用**less**函数对象来决定元素的顺序。它使用了被包含类的比较运算符**operator<**，但可以允许用户用自己的比较函数来代替它。

考虑一个简单的包含一个用于进行比较的元素与另一个不用于进行比较的元素的类：

```
//: C07:MultiSet1.cpp
// Demonstration of multiset behavior.
#include <algorithm>
#include <cstdlib>
#include <ctime>
#include <iostream>
#include <iterator>
#include <set>
using namespace std;

class X {
  char c; // Used in comparison
  int i; // Not used in comparison
  // Don't need default constructor and operator=
  X();
  X& operator=(const X&);
  // Usually need a copy-constructor (but the
  // synthesized version works here)
public:
  X(char cc, int ii) : c(cc), i(ii) {}
  // Notice no operator== is required
  friend bool operator<(const X& x, const X& y) {
    return x.c < y.c;
  }
  friend ostream& operator<<(ostream& os, X x) {
    return os << x.c << ":" << x.i;
  }
};

class Xgen {
  static int i;
  // Number of characters to select from:
  enum { SPAN = 6 };
public:
  X operator()() {
    char c = 'A' + rand() % SPAN;
    return X(c, i++);
  }
};

int Xgen::i = 0;

typedef multiset<X> Xmset;
typedef Xmset::const_iterator Xmit;

int main() {
  Xmset mset;
  // Fill it with X's:
  srand(time(0));  // Randomize
  generate_n(inserter(mset, mset.begin()), 25, Xgen());
  // Initialize a regular set from mset:
  set<X> unique(mset.begin(), mset.end());
  copy(unique.begin(), unique.end(),
    ostream_iterator<X>(cout, " "));
  cout << "\n----" << endl;
  // Iterate over the unique values:
  for(set<X>::iterator i = unique.begin();
      i != unique.end(); i++) {
    pair<Xmit, Xmit> p = mset.equal_range(*i);
    copy(p.first,p.second, ostream_iterator<X>(cout, " "));
    cout << endl;
  }
} ///:~
```

528

529

在**X**中，所有的比较都产生**char c**。因为在这个例子中使用了默认的**less**比较对象，比较由**operator<**进行，这就是**multiset**所必需的全部工作。类**Xgen**随机产生**X**对象，但是用于比较的值被限制在**'A'**到**'E'**之间的范围内。在**main( )**函数中，创建一个**multiset<X>**并用**Xgen**向其中填入25个**X**对象，这就保证了那里存在重复的关键字。所以为了了解存在哪些惟一的关键字值，根据**multiset**创建了一个常规的**set<X>**（使用**iterator**、**iterator**构造函数）。这些值被显示出来，然后对在**multiset**中的每个关键字值都产生**equal_range( )**（**equal_range( )**在这里和在**multimap**中有着相同的意义：所有的元素都与进行匹配的关键字相匹配）。然后打印每个匹配的关键字集。

作为第2个例子，用**multiset**创建的一个（可能）版本更优雅的**WordCount.cpp**:

```
//: C07:MultiSetWordCount.cpp
// Count occurrences of words using a multiset.
#include <fstream>
#include <iostream>
#include <iterator>
#include <set>
#include <string>
#include "../require.h"
using namespace std;

int main(int argc, char* argv[]) {
  const char* fname = "MultiSetWordCount.cpp";
  if(argc > 1) fname = argv[1];
  ifstream in(fname);
  assure(in, fname);
  multiset<string> wordmset;
  string word;
  while(in >> word)
    wordmset.insert(word);
  typedef multiset<string>::iterator MSit;
  MSit it = wordmset.begin();
  while(it != wordmset.end()) {
    pair<MSit, MSit> p = wordmset.equal_range(*it);
    int count = distance(p.first, p.second);
    cout << *it << ": " << count << endl;
    it = p.second; // Move to the next word
  }
} ///:~
```

530

main( )函数中的设置与**WordCount.cpp**中完全相同，然后每个单词都只是被插入到**multiset<string>**中。一个迭代器被创建出来并且初始化为指向**multiset**的起始处；解析该迭代器就可以产生它所指向的当前单词。成员函数**equal_range( )**（并非通用算法）产生当前选中的单词的起始和终止迭代器，算法**distance( )**（在**<iterator>**中定义的）对该范围内的元素进行计数。然后，迭代器**it**向前移动到范围终止之处，并令其指向下一个单词。如果读者现在还不熟悉**multiset**的话，这里的代码就可能显得太复杂。但它的紧凑性和没有所需的诸如**Count**这样的支持类却具有很强的吸引力。

最后，这个容器到底是个真实地“集合”，或者还是应当使用别的名字来命名它呢？另一种选择是，可以将其命名为在某些容器库中定义的一般称为“袋子（bag）”的容器，因为一个袋子可以不加区别地保存任何东西——包括重复的对象。这样的命名比较接近实际情况，可是由于袋子没有对元素按怎样的顺序排列给出规范，所以它也不是完全适合。**multiset**（它要求所有重复的元素必须相互毗邻地存放）比起**set**的概念来其限制甚至更加严格。一个**set**实现可能使用散列函数（hashing function）来排列其元素，这样它将不会按排序的顺序存放这些元

素。另外，如果想要不受限制地（即没有任何指定标准）存放一个对象串，可以使用**vector**，**deque**或者**list**。

7.11 将STL容器联合使用

在使用一个同义语词汇编（thesaurus）时，读者可能想知道所有与某个特定单词相似的所有单词。在查寻一个单词的时候，读者希望结果由一个单词表给出。这里，那些“多重”容器（**multiset**或者**multimap**）都不适合。解决方案是将容器联合在一起使用，该方法用STL很容易实现。在这里，我们需要一个工具，其结果是形成一个功能强大的通用的概念，那就是能使字符串与一个**vector**关联成为**map**：

```
//: C07:Thesaurus.cpp
// A map of vectors.
#include <map>
#include <vector>
#include <string>
#include <iostream>
#include <iterator>
#include <algorithm>
#include <ctime>
#include <cstdlib>
using namespace std;

typedef map<string, vector<string> > Thesaurus;
typedef pair<string, vector<string> > TEntry;
typedef Thesaurus::iterator TIter;

// Name lookup work-around:
namespace std {
ostream& operator<<(ostream& os,const TEntry& t) {
  os << t.first << ": ";
  copy(t.second.begin(), t.second.end(),
    ostream_iterator<string>(os, " "));
  return os;
}
}

// A generator for thesaurus test entries:
class ThesaurusGen {
  static const string letters;
  static int count;
public:
  int maxSize() { return letters.size(); }
  TEntry operator()() {
    TEntry result;
    if(count >= maxSize()) count = 0;
    result.first = letters[count++];
    int entries = (rand() % 5) + 2;
    for(int i = 0; i < entries; i++) {
      int choice = rand() % maxSize();
      char cbuf[2] = { 0 };
      cbuf[0] = letters[choice];
      result.second.push_back(cbuf);
    }
    return result;
  }
};

int ThesaurusGen::count = 0;
```

531

532

```
const string ThesaurusGen::letters("ABCDEFGHIJKL"
  "MNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz");

// Ask for a "word" to look up:
string menu(Thesaurus& thesaurus) {
  while(true) {
    cout << "Select a \"word\", 0 to quit: ";
    for(TIter it = thesaurus.begin();
        it != thesaurus.end(); it++)
      cout << (*it).first << ' ';
    cout << endl;
    string reply;
    cin >> reply;
    if(reply.at(0) == '0') exit(0); // Quit
    if(thesaurus.find(reply) == thesaurus.end())
      continue; // Not in list, try again
    return reply;
  }
}

int main() {
  srand(time(0)); // Seed the random number generator
  Thesaurus thesaurus;
  // Fill with 10 entries:
  generate_n(inserter(thesaurus, thesaurus.begin()),
    10, ThesaurusGen());
  // Print everything:
  copy(thesaurus.begin(), thesaurus.end(),
    ostream_iterator<TEntry>(cout, "\n"));
  // Create a list of the keys:
  string keys[10];
  int i = 0;
  for(TIter it = thesaurus.begin();
    it != thesaurus.end(); it++)
    keys[i++] = (*it).first;
  for(int count = 0; count < 10; count++) {
    // Enter from the console:
    // string reply = menu(thesaurus);
    // Generate randomly
    string reply = keys[rand() % 10];
    vector<string>& v = thesaurus[reply];
    copy(v.begin(), v.end(),
      ostream_iterator<string>(cout, " "));
    cout << endl;
  }
} ///:~
```

533

Thesaurus将一个**string**（即单词）映射到一个**vector<string>**（同义词）。**TEntry**是**Thesaurus**中的一个条目。通过为**TEntry**创建一个输出流操作符**ostream operator<<**，可以很容易地打印来自**Thesaurus**中的这一条目（而整个**Thesaurus**可以容易地用**copy( )**进行打印）。注意，流插入符所处的非常奇怪的位置：它被放置在**std**名字空间中！[⊖]上面的这个**operator<<( )**函数将在**main( )**中的第1个**copy( )**调用中被**ostream_iterator**使用。在编译器实例化时，所需要的**ostream_iterator**特化根据参数关联查找（argument-dependent lookup，ADL）规则，它只查看**std**，因为函数**copy( )**所有的参数都在那儿声明。如果在全局名字空间中声明插入符（将其范围限定为迁移名字空间块），它就不会被发现。将

⊖ 从技术上讲，用户向标准名字空间中添加东西是不合法的，但这是防止出现隐藏的名字查找问题的最简单的方法，并且被使用的所有编译器支持。

其放入**std**中，就可以通过ADL找到它。

ThesaurusGen创建“单词”（它们仅是单个字母）以及这些单词的“同义词”（这是另外一些随机选择的单个字母）以用作同义语词汇编的条目。它随机挑选制造同义词条目的个数，但必须至少为两个。所有被选择的字母都被编进一个静态字符串**static string**索引中，该**static string**是**ThesaurusGen**的一部分。

在**main( )**函数中，创建一个**Thesaurus**，并填入10个条目，并且调用**copy( )**算法将它们打印出来。函数**menu( )**要求用户通过键入代表单词的字母来选择一个“单词”来进行查寻。用成员函数**find( )**来查找以确定该条目是否在**map**中。（记住不要使用**operator[ ]**，它将会在未找到匹配条目的情况下自动创建一个新的条目！）如果存在，就用**operator[ ]**取出**vector<string>**进行显示。对于**reply**字符串的选择是随机产生的，允许进行自动测试。

模板的使用使得表达功能强大的概念变得很容易，甚至可以更进一步地扩展这个概念创建一个**vector**的**map**，而**vector**又包含有**map**等等。由于这个原因，可以用这种方法联合任何STL容器。

7.12 清除容器的指针

534

在**Stlshape.cpp**中，容器中的那些指针自己不会自动清除。有方便的方法能很容易地做这些事情，不必每一次都为此编写专用代码。这里有个能够清除任何序列容器中指针的函数模板。注意，它被放置在本教材的根目录下面以方便使用：

```
//: :purge.h
// Delete pointers in an STL sequence container.
#ifndef PURGE_H
#define PURGE_H
#include <algorithm>

template<class Seq> void purge(Seq& c) {
  typename Seq::iterator i;
  for(i = c.begin(); i != c.end(); ++i) {
    delete *i;
    *i = 0;
  }
}

// Iterator version:
template<class InpIt> void purge(InpIt begin, InpIt end) {
  while(begin != end) {
    delete *begin;
    *begin = 0;
    ++begin;
  }
}
#endif // PURGE_H ///:~
```

在**purge( )** 的第1版中，要注意关键字**typename**是绝对必需的。该关键字正是设计用来解决问题的：**Seq**是一个模板参数，而**iterator**则是嵌套在该模板中的某种东西。那么**Seq::iterator**做什么用呢？关键字**typename**说明，它提到的是个类型，而不是其他什么东西。

虽然**purge( )**的容器版本必须与一个STL风格的容器一起工作，但**purge( )**的迭代器版本的工作区域则涵盖了所有范围，包括数组。

这里有一个重写了的**Stlshape.cpp**，修改并使用了**purge( )**函数：

535

```
//: C07:Stlshape2.cpp
// Stlshape.cpp with the purge() function.
#include <iostream>
#include <vector>
#include "../purge.h"
using namespace std;

class Shape {
public:
  virtual void draw() = 0;
  virtual ~Shape() {};
};

class Circle : public Shape {
public:
  void draw() { cout << "Circle::draw" << endl; }
  ~Circle() { cout << "~Circle" << endl; }
};

class Triangle : public Shape {
public:
  void draw() { cout << "Triangle::draw" << endl; }
  ~Triangle() { cout << "~Triangle" << endl; }
};

class Square : public Shape {
public:
  void draw() { cout << "Square::draw" << endl; }
  ~Square() { cout << "~Square" << endl; }
};

int main() {
  typedef std::vector<Shape*> Container;
  typedef Container::iterator Iter;
  Container shapes;
  shapes.push_back(new Circle);
  shapes.push_back(new Square);
  shapes.push_back(new Triangle);
  for(Iter i = shapes.begin(); i != shapes.end(); i++)
    (*i)->draw();
  purge(shapes);
} ///:~
```

536 在使用**purge( )**时，要仔细考虑该函数的所有权问题。如果在多个容器中持有同一个对象的指针，要确信不对其进行两次删除操作。不希望在第2个容器结束对该对象的使用之前就在第1个容器中将其销毁。对一个容器进行两次清除操作**purge( )**不会产生问题，因为**purge( )**在删除一个指针后将其值置为零，对一个零指针调用删除操作**delete**是一个安全的操作。

7.13 创建自己的容器

有了STL作基础，用户就可以创建自己的容器了。假定读者根据提供的迭代器进行模仿，用户自己创建的新容器将会表现得就好像一个内置的STL容器。

考虑某个“环形”数据结构，它是一个循环的序列容器。如果到达了环的末尾端点，即此时它刚好是绕回到起始端点（末尾端点和起始端点是同一个点）。这可以在熟练掌握**list**的基础上实现，如下所示:

```
//: C07:Ring.cpp
// Making a "ring" data structure from the STL.
#include <iostream>
#include <iterator>
#include <list>
#include <string>
using namespace std;

template<class T> class Ring {
  list<T> lst;
public:
  // Declaration necessary so the following
  // 'friend' statement sees this 'iterator'
  // instead of std::iterator:
  class iterator;
  friend class iterator;
  class iterator : public std::iterator<
    std::bidirectional_iterator_tag,T,ptrdiff_t>{
    typename list<T>::iterator it;
    list<T>* r;
  public:
    iterator(list<T>& lst,
      const typename list<T>::iterator& i)
    : it(i), r(&lst) {}
    bool operator==(const iterator& x) const {
      return it == x.it;
    }
    bool operator!=(const iterator& x) const {
      return !(*this == x);
    }
    typename list<T>::reference operator*() const {
      return *it;
    }
    iterator& operator++() {
      ++it;
      if(it == r->end())
        it = r->begin();
      return *this;
    }
    iterator operator++(int) {
      iterator tmp = *this;
      ++*this;
      return tmp;
    }
    iterator& operator--() {
      if(it == r->begin())
        it = r->end();
      --it;
      return *this;
    }
    iterator operator--(int) {
      iterator tmp = *this;
      --*this;
      return tmp;
    }
    iterator insert(const T& x) {
      return iterator(*r, r->insert(it, x));
    }
    iterator erase() {
      return iterator(*r, r->erase(it));
    }
  };
  void push_back(const T& x) { lst.push_back(x); }
```

537

```
  iterator begin() { return iterator(lst, lst.begin()); }
  int size() { return lst.size(); }
};

int main() {
  Ring<string> rs;
  rs.push_back("one");
  rs.push_back("two");
  rs.push_back("three");
  rs.push_back("four");
  rs.push_back("five");
  Ring<string>::iterator it = rs.begin();
  ++it; ++it;
  it.insert("six");
  it = rs.begin();
  // Twice around the ring:
  for(int i = 0; i < rs.size() * 2; i++)
    cout << *it++ << endl;
} ///:~
```

538

读者可以看到，绝大多数编码都是针对迭代器进行的。这个**Ring iterator**必须知道如何循环回到起始端点，所以它必须持有一个指向作为其“双亲”**Ring**对象的**list**的引用，从而知道是否已经到了环的末尾端点，这样它才能知道如何回到起始端点。

必须注意，为**Ring**设置的接口相当有限；特别是，这里没有**end( )**函数，因为一个环仅仅保持进行循环的状态。这就意味着不能在需要使用超越末尾的迭代器的任何STL算法中使用**Ring**，STL中这样的算法有很多。（添加这个特征并不是无足轻重的练习。）尽管这似乎使其使用受到了限制，但是考虑一下**stack**、**queue**和**priority_queue**，它们甚至全都没有产生任何迭代器！

7.14 对STL的扩充

尽管STL容器可以提供用户曾经需要的全部功能，但它们不是十全十美的。比如标准的**set**和**map**的实现都使用树型数据结构，尽管其操作相当快速，但并没有快速到足以满足用户需要的程度。在C++标准委员会中，对将利用散列算法实现的**set**和**map**包括进C++标准中的想法已经达到共识。然而由于没有足够的时间加入这些组件，最终他们放弃了这样做。[⊖]

539

幸运的是，还有可利用的免费替代品。有关STL的美好之处之一，就是它为创建类-STL（STL-like）的类建立了基本的模型。因此如果用户已经熟悉了STL，那么使用同样的模型创建的任何东西就都很容易理解了。

来自于Silicon Graphics[⊖]的SGI STL是最健壮的STL的实现之一，如果有需要可以用这个SGI STL替代用户编译器所使用的STL。另外，SGI增加了很多扩充的容器，包括**hash_set**、**hash_multiset**、**hash_map**、**hash_multimap**、**slist**（单链表）和**rope**（它是一个**string**的变种，对非常大型的字符串、字符串的快速连结和取子串等操作进行了优化）。

现在考虑在基于树结构的**map**和SGI **hash_map**之间进行性能比较。为简单起见，这里将进行从**int**到**int**之间的映射：

```
//: C07:MapVsHashMap.cpp
// The hash_map header is not part of the Standard C++ STL.
// It is an extension that is only available as part of the
// SGI STL (Included with the dmc distribution).
```

⊖ 它们可能包括在标准C++的下一个发行版本中。

⊖ 参见 http://www.sgi.com/tech/stl。

```
// You can add the header by hand for all of these:
//{-bor}{-msc}{-g++}{-mwcc}
#include <hash_map>
#include <iostream>
#include <map>
#include <ctime>
using namespace std;

int main() {
  hash_map<int, int> hm;
  map<int, int> m;
  clock_t ticks = clock();
  for(int i = 0; i < 100; i++)
    for(int j = 0; j < 1000; j++)
      m.insert(make_pair(j,j));
  cout << "map insertions: " << clock() - ticks << endl;
  ticks = clock();
  for(int i = 0; i < 100; i++)
    for(int j = 0; j < 1000; j++)
      hm.insert(make_pair(j,j));
  cout << "hash_map insertions: "
       << clock() - ticks << endl;
  ticks = clock();
  for(int i = 0; i < 100; i++)
    for(int j = 0; j < 1000; j++)
      m[j];
  cout << "map::operator[] lookups: "
       << clock() - ticks << endl;
  ticks = clock();
  for(int i = 0; i < 100; i++)
    for(int j = 0; j < 1000; j++)
      hm[j];
  cout << "hash_map::operator[] lookups: "
       << clock() - ticks << endl;
  ticks = clock();
  for(int i = 0; i < 100; i++)
    for(int j = 0; j < 1000; j++)
      m.find(j);
  cout << "map::find() lookups: "
       << clock() - ticks << endl;
  ticks = clock();
  for(int i = 0; i < 100; i++)
    for(int j = 0; j < 1000; j++)
      hm.find(j);
  cout << "hash_map::find() lookups: "
       << clock() - ticks << endl;
} ///:~
```

540

通过运行这个演示性能测试的程序，在所有的操作中**hash_map**超越**map**其速度有大约4:1的改进（而且就像所预期的那样，对于两种类型的**map**进行查寻，**find( )**都比**operator[ ]**稍微快些）。如果profiler显示出用户**map**中的性能成为系统的瓶颈，可以考虑使用**hash_map**。

7.15 非STL容器

在标准库中有两种“非STL”容器：**bitset**和**valarray**。[⊖]之所以称之为“非STL”，是因为这两种容器中没有一种能够完全满足STL容器的要求。在本章前部包括了**bitset**容器，将二进制位打包成整数并且不允许对其成员进行直接寻址。**valarray**模板类是一个类-**vector**的容器，

⊖ 在前面已经提到过，在某种程度上讲**vector<bool>**特化也是一个非STL容器。

541 该容器对有效率的数值的计算进行了优化。这两个容器都不提供迭代器。虽然可以用非数值类型来实例化**valarray**，但是它拥有一些用于操作数值型数据的数学函数，比如**sin**、**cos**、**tan**等等。

这里有一个用来打印**valarray**中元素的工具：

```
//: C07:PrintValarray.h
#ifndef PRINTVALARRAY_H
#define PRINTVALARRAY_H
#include <valarray>
#include <iostream>
#include <cstddef>

template<class T>
void print(const char* lbl, const std::valarray<T>& a) {
  std::cout << lbl << ": ";
  for(std::size_t i = 0; i < a.size(); ++i)
    std::cout << a[i] << ' ';
  std::cout << std::endl;
}
#endif // PRINTVALARRAY_H ///:~
```

valarray的大多数函数和运算符都将**valarray**作为一个整体来进行操作，就像下面的例子阐明的一样：

```
//: C07:Valarray1.cpp {-bor}
// Illustrates basic valarray functionality.
#include "PrintValarray.h"
using namespace std;

double f(double x) { return 2.0 * x - 1.0; }

int main() {
  double n[] = { 1.0, 2.0, 3.0, 4.0 };
  valarray<double> v(n, sizeof n / sizeof n[0]);
  print("v", v);
  valarray<double> sh(v.shift(1));
  print("shift 1", sh);
  valarray<double> acc(v + sh);
  print("sum", acc);
  valarray<double> trig(sin(v) + cos(acc));
  print("trig", trig);
  valarray<double> p(pow(v, 3.0));
  print("3rd power", p);
  valarray<double> app(v.apply(f));
  print("f(v)", app);
  valarray<bool> eq(v == app);
  print("v == app?", eq);
  double x = v.min();
  double y = v.max();
  double z = v.sum();
  cout << "x = " << x << ", y = " << y
       << ", z = " << z  << endl;
} ///:~
```

542

valarray类提供了一个构造函数，该构造函数接受一个目标类型的数组和数组中的元素计数作为其参数来初始化一个新的**valarray**。成员函数**shift( )**将每个**valarray**元素向左移动一个位置（或者，如果它的参数是个负值则向右移动），并且向移走元素后的空位中填入该类型的默认值（在这种情况下是0）。还有一个成员函数**cshift( )**，它进行循环移动（或者称为“旋转”）。所有数学运算符和函数都进行了重载以便用来操作**valarray**，二进位运算符要

求**valarray**具有相同类型和大小的参数。像**transform( )**算法一样，成员函数**apply( )**对每一个元素应用一个函数，但是结果被收集到一个结果**valarray**中。关系运算符返回大小匹配的**valarray<bool>**实例，该实例显示了元素与元素逐个对比的结果，例如上面的**eq**。大多数操作返回一个新的结果数组，但是很少，由于显而易见的原因，其中的一些操作返回一个数值，比如**min( )**、**max( )**、和**sum( )**。

对**valarray**可以做的最有趣的事情就是引用其元素的一个子集，不仅可以提取信息，而且可以更新这些信息。**valarray**的一个子集被称为一个切片（slice），某些运算符使用切片来做它们的工作。下面的简单程序就使用了切片：

```
//: C07:Valarray2.cpp {-bor}{-dmc}
// Illustrates slices and masks.
#include "PrintValarray.h"
using namespace std;

int main() {
  int data[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12 };
  valarray<int> v(data, 12);
  valarray<int> r1(v[slice(0, 4, 3)]);
  print("slice(0,4,3)", r1);
  // Extract conditionally
  valarray<int> r2(v[v > 6]);
  print("elements > 6", r2);
  // Square first column
  v[slice(0, 4, 3)] *= valarray<int>(v[slice(0, 4, 3)]);
  print("after squaring first column", v);
  // Restore it
  int idx[] = { 1, 4, 7, 10 };
  valarray<int> save(idx, 4);
  v[slice(0, 4, 3)] = save;
  print("v restored", v);
  // Extract a 2-d subset: { { 1, 3, 5 }, { 7, 9, 11 } }
  valarray<size_t> siz(2);
  siz[0] = 2;
  siz[1] = 3;
  valarray<size_t> gap(2);
  gap[0] = 6;
  gap[1] = 2;
  valarray<int> r3(v[gslice(0, siz, gap)]);
  print("2-d slice", r3);
  // Extract a subset via a boolean mask (bool elements)
  valarray<bool> mask(false, 5);
  mask[1] = mask[2] = mask[4] = true;
  valarray<int> r4(v[mask]);
  print("v[mask]", r4);
  // Extract a subset via an index mask (size_t elements)
  size_t idx2[] = { 2, 2, 3, 6 };
  valarray<size_t> mask2(idx2, 4);
  valarray<int> r5(v[mask2]);
  print("v[mask2]", r5);
  // Use an index mask in assignment
  valarray<char> text("now is the time", 15);
  valarray<char> caps("NITT", 4);
  valarray<size_t> idx3(4);
  idx3[0] = 0;
  idx3[1] = 4;
  idx3[2] = 7;
  idx3[3] = 11;
  text[idx3] = caps;
  print("capitalized", text);
} ///:~
```

543

一个**slice**对象接受3个参数：起始索引、要提取的元素合计数以及“跨距”，即两个用户感兴趣的元素之间的间距。切片可以用来作为一个现有**valarray**的索引，并且返回一个包含了被提取元素的新的**valarray**。比如一个**bool**型的**valarray**，它是由表达式**v>6**返回的值，也可以作为另一个**valarray**的索引；那些符合**true**值所在位置的元素都被提取出来。就像看到的那样，也可以将切片和掩码作为索引用在一个赋值操作的左边。一个**gslice**对象（即“generalized slice”，通用切片）就像一个切片，除了合计数和跨距参数是它们自己的数组之外，这意味着可以将一个**valarray**解释为一个多维数组。上面的例子从**v**中提取了一个2乘3的数组，从**v**中下标为0的元素开始，到相距6个元素的位置建立第1维的元素数，所做的其他事情就是在各维中每相距两个元素的位置提取一个数，这样就有效地从**v**中提取出了一个矩阵：

544

```
1 3 5
7 9 11
```

以下是这个程序的完整输出：

```
slice(0,4,3): 1 4 7 10
elements > 6: 7 8 9 10
after squaring v: 1 2 3 16 5 6 49 8 9 100 11 12
v restored: 1 2 3 4 5 6 7 8 9 10 11 12
2-d slice: 1 3 5 7 9 11
v[mask]: 2 3 5
v[mask2]: 3 3 4 7
capitalized: N o w   I s   T h e   T i m e
```

在矩阵乘法中可以发现一个使用切片的实际例子。考虑如何使用数组来编写两个整数矩阵相乘的函数。

```
void matmult(const int a[][MAXCOLS], size_t m, size_t n,
             const int b[][MAXCOLS], size_t p, size_t q,
             int result[][MAXCOLS);
```

这个函数将一个**m**乘**n**的矩阵***a***和一个**p**乘**q**的矩阵***b***相乘，这里**n**和**p**应当相等。就像读者可以看到的，没有什么事情像**valarray**那样，必须为每个矩阵的第2维确定最大值，因为数组中的每个位置都是静态决定了的（固定的）。而且也很难通过值返回一个结果数组，因此调用者通常传递一个结果数组作为参数。

使用**valarray**，不仅可以传递任意大小的矩阵，而且可以容易地处理任意类型的矩阵，并且通过传值的方式返回结果。其实现方式如下所示：

545

```
//: C07:MatrixMultiply.cpp
// Uses valarray to multiply matrices
#include <cassert>
#include <cstddef>
#include <cmath>
#include <iostream>
#include <iomanip>
#include <valarray>
using namespace std;

// Prints a valarray as a square matrix
template<class T>
void printMatrix(const valarray<T>& a, size_t n) {
  size_t siz = n*n;
  assert(siz <= a.size());
  for(size_t i = 0; i < siz; ++i) {
    cout << setw(5) << a[i];
    cout << ((i+1)%n ? ' ' : '\n');
  }
```

```
  cout << endl;
}

// Multiplies compatible matrices in valarrays
template<class T>
valarray<T>
matmult(const valarray<T>& a, size_t arows, size_t acols,
        const valarray<T>& b, size_t brows, size_t bcols) {
  assert(acols == brows);
  valarray<T> result(arows * bcols);
  for(size_t i = 0; i < arows; ++i)
    for(size_t j = 0; j < bcols; ++j) {
      // Take dot product of row a[i] and col b[j]
      valarray<T> row = a[slice(acols*i, acols, 1)];
      valarray<T> col = b[slice(j, brows, bcols)];
      result[i*bcols + j] = (row * col).sum();
    }
  return result;
}

int main() {
  const int n = 3;
  int adata[n*n] = {1,0,-1,2,2,-3,3,4,0};
  int bdata[n*n] = {3,4,-1,1,-3,0,-1,1,2};
  valarray<int> a(adata, n*n);
  valarray<int> b(bdata, n*n);
  valarray<int> c(matmult(a, n, n, b, n, n));
  printMatrix(c, n);
} ///:~
```

546

在结果矩阵**c**中，每一个条目都是**a**中的某一行与**b**中的某一列的点积。通过使用切片，可以将这些行和列作为**valarray**提取出来，并使用全局的*运算符和**valarray**提供的**sum( )**函数进行简洁地计算。作为结果的**valarray**在运行时进行计算；没有必要担心数组维数的静态限制。在这里确实需要自行计算位置**[i][j]**的线性偏移量（参见上面的公式**i * bcols + j**），但是为了自由地确定**valarray**的大小和类型，这是值得的。

7.16 小结

本章的目的不仅仅是在某种程度上深入地介绍STL容器。尽管不可能在这里涵盖STL的所有细节，读者现在也了解了足够的线索，并能在其他的资源中学习更多的信息。我们希望通过这一章帮助读者理解STL中强大的可用功能，显示了在理解和使用STL的基础上，如何能够更快速和更高效率地编程。

7.17 练习

7-1 创建一个**set<char>**，打开一个文件（文件名在命令行中给出），每次从文件中读入一个**char**，将每个**char**放入该集合中。打印结果并观察其组织结构。在这个特定文件里的字母中有未被使用的字母吗？

7-2 创建3个**Noisy**对象序列，**vector**、**deque**和**list**。对它们进行排序。现在编写一个函数模板，接收**vector**和**deque**序列作为参数来对它们进行排序，并记录下排序的时间。编写一个特化的模板函数对**list**进行同样的操作（确保调用其成员函数**sort( )**而不是使用通用算法）。比较不同类型序列的性能。

7-3 编写一个程序用来比较分别使用**list::sort( )**以及**std::sort( )**（STL算法版本的**sort( )**） 547

对链表进行排序的速度。

7-4 创建一个发生器以产生0到20（包括20）之间的随机**int**型值，用它们填充一个**multiset<int>**。对每个值出现的次数进行计数，遵循例程**MultiSetWordCount.cpp**中给出的方法。

7-5 修改**StlShape.cpp**，让它用**deque**而不用**vector**。

7-6 修改**Reversible.cpp**，使其与**deque**和**list**一起工作而非**vector**。

7-7 使用一个**stack<int>**并将斐波那契（Fibonacci）数列存储其中。程序的命令行应该指明想要的斐波那契数列中元素的个数，还要有一个可以查看栈中是否剩下最后两个元素的循环，如果剩下最后两个元素，则在今后的每次循环中压入一个新的符合斐波那契数列的元素。

7-8 仅使用3个**stack**（源栈（source）、排序栈（sorted）和失败者栈（losers）），通过首先存放数字到源栈上，来对一个随机的数字序列排序。假定源栈上的栈顶元素是最大的，将其压入排序栈。持续地将源栈中的元素弹出并与排序栈中的栈顶元素比较。无论哪个栈数字最小，将最小的数字从其栈中弹出并压入失败者栈。一旦源栈为空，使用失败者栈作为源栈并重复该过程，并且使用源栈作为失败者栈。当所有的数字都已经被存入胜利者栈（排序栈）以后，算法结束。

7-9 打开一个文本文件，在命令行中提供其文件名。每一次从文件中读入一个单词，并使用**multiset<string>**为每个单词创建一个单词计数。

7-10 修改**WordCount.cpp**，使其使用**insert( )**而非**operator[ ]**向**map**中插入元素。

7-11 创建拥有一个**operator<** 和一个**ostream& operator<<** 的一个类，该类应该包含一个具有优先级的数。为该类创建一个发生器，用来产生随机的具有优先级的数。用该发生器产生的数填充一个**priority_queue**，然后取出元素并观察它们是否按照正确的顺序排列。

7-12 重写**Ring.cpp**，使其用一个**deque**而非**list**作为其底层实现。

548 7-13 修改**Ring.cpp**，使其底层实现可以通过模板参数来进行选择。（将那个模板参数的默认值设为**list**。）

7-14 创建一个名为**BitBucket**的迭代器类，它仅接收发送给它的无论什么任何东西，而不会将其写到任何地方。

7-15 创建一种“猜单词”的游戏程序。创建一个类，该类包含一个**char**型成员和一个指示该**char**型成员是否已经被猜中的**bool**型成员。从一个文件中随机地选择一个单词，并且将其读入用户的新类型的**vector**。重复地询问用户对一个字符的猜测，在每次猜测之后，显示该单词中已猜中的字符，对未猜中的字符显示下划线。允许给用户提供猜测全部单词的方法。在每一次猜测之后对某个值减1，在该值到达零之前如果猜中了全部单词，则用户胜出。

7-16 打开一个文件，并将其读入一个字符串。使该串翻转并送入一个字符串流**stringstream**。用标识符迭代器**TokenIterator**从该字符串流**stringstream**读入标识符并将其存入到**list<string>**中。

7-17 比较分别基于**vector**、**deque**或**list**实现的**stack**的性能。

7-18 创建一个模板用以实现一个名为**SList**的单链表。提供默认构造函数、**begin( )**和**end( )**函数（通过适当的嵌套迭代器），**insert( )**、**erase( )**和析构函数。

7-19 产生一个随机的整数序列，将它们存入一个**int**型数组。用其内容来初始化一个

valarray<int>。用**valarray**操作计算这些数字序列的和、最小值、最大值、平均值和在序列正中间元素的值。

7-20 用12个随机值创建一个**valarray<int>**，用20个随机值创建另一个**valarray<int>**。将第1个**valarray**理解为一个3 × 4的**int**型矩阵，第2个解释为4 × 5的**int**型矩阵，并且根据矩阵乘法的规则将它们相乘。将结果保存在大小为15的**valarray<int>**中，它表示一个3 × 5的结果矩阵。使用切片将第1个矩阵的行分次与第2个矩阵的列相乘。将结果以长方形的矩阵形式打印出来。

第三部分 专 题

专业人员的标志体现在他（或她）更加注重精益求精。在本教材的第三部分讨论C++的高级特性，以及那些被C++专业人员中的精英们所使用的开发技术。

在软件研发过程中，有时也许背离正常的面向对象设计的习语常识检查一个对象的运行时类型。大多数情况下需要用虚函数来做这项工作，但是当编写如调试器、数据库观察器、或类浏览器这些特殊用途的软件工具时，则需要在运行时来决定它们的类型信息。这就是运行时类型识别（runtime type identification，RTTI）机制发挥作用的地方。RTTI是第8章的主题。

多重继承的使用在过去的这些年已经达到了滥用的地步，但某些语言甚至不支持它。当适当地运用多重继承时，它对精心制作优雅、高效的程序代码依然是一件强有力的工具。许多涉及多重继承的标准的实际应用在过去的这些年得到了长足的发展，这些内容将在第9章中介绍。

大概自从面向对象技术产生以来，在软件开发中最著名的创新就是设计模式的运用。对于在软件设计中包括的许多共同的问题，设计模式为其描述了解决方案，并且这些解决方案可以应用在许多情形中，并可以用任意一种语言来实现。在第10章中将描述许多精心挑选出的设计模式，并且用C++来实现这些设计模式。

第11章说明多线程编程的优势和所遇到的挑战。虽然大多数操作系统提供了多线程处理的功能，但标准C++现在的版本并没有说明对线程的支持。本教材使用一个可移植的、可免费利用的线程处理库来说明C++程序员可以怎样利用线程的优势去构建更多有用的和应答式的应用程序。

FULL TEXT, UPDATES AND CODE AT WWW.BRUCEECKEL.COM

第8章 运行时类型识别

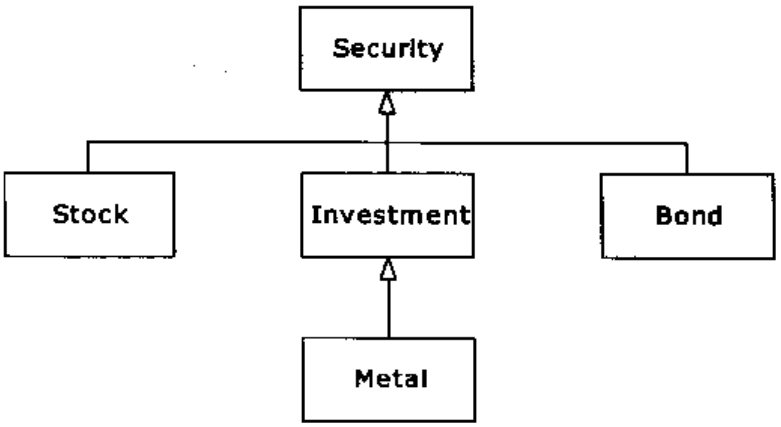
当仅有一个指针或引用指向基类型时，利用运行时类型识别（RTTI）可以找到一个对象的动态类型。

运行时类型识别可能被认为是C++中一个“次要”的特征，当程序员在编程过程中陷入非常困难的境地时，实用主义将会帮助他走出困境。正常情况下，程序员需要有意忽略对象的准确类型，而利用虚函数机制实现那个类型正确操作过程。然而，有时知道一个仅含有一个基类指针的对象的准确的运行时类型（即多半是派生的类型）是非常有用的。有了此信息，就可以更有效地进行某些特殊情况的操作，或者预防基类接口因无此信息而变得笨拙。大部分的类库都包含了虚函数，以便产生足够的运行时类型信息。当在C++中增加了异常处理时，这个特征需要对象的运行时类型的信息，因此，嵌入对这些信息的访问就使下一步工作变得很容易。本章将解释RTTI的用途和如何使用它。

8.1 运行时类型转换

通过指针或引用来决定对象运行时类型的一种方法是使用*运行时类型转换*（runtime cast），用这种方法可以查证所尝试进行的转换正确与否。当要把基类指针类型转换为派生类型时，这种方法非常有用。由于继承的层次结构的典型描述是基类在派生类之上，所以这种类型转换也称为*向下类型转换*（downcast）。

请看下面的类层次结构:



在下面的程序代码中，**Investment**类有一个其他类没有的额外操作，所以能够在运行时知道**Security**指针是否引用了**Investment**对象是很重要的。为了实现检查运行时的类型转换，每个类都持有一个整数标识符，以便可以与层次结构中其他的类区别开来。

```
//: C08:CheckedCast.cpp
// Checks casts at runtime.
#include <iostream>
#include <vector>
#include "../purge.h"
using namespace std;

class Security {
```

```
protected:
  enum { BASEID = 0 };
public:
  virtual ~Security() {}
  virtual bool isA(int id) { return (id == BASEID); }
};

class Stock : public Security {
  typedef Security Super;
protected:
  enum { OFFSET = 1, TYPEID = BASEID + OFFSET };
public:
  bool isA(int id) {
    return id == TYPEID || Super::isA(id);
  }
  static Stock* dynacast(Security* s) {
    return (s->isA(TYPEID)) ? static_cast<Stock*>(s) : 0;
  }
};

class Bond : public Security {
  typedef Security Super;
protected:
  enum { OFFSET = 2, TYPEID = BASEID + OFFSET };
public:
  bool isA(int id) {
    return id == TYPEID || Super::isA(id);
  }
  static Bond* dynacast(Security* s) {
    return (s->isA(TYPEID)) ? static_cast<Bond*>(s) : 0;
  }
};

class Investment : public Security {
  typedef Security Super;
protected:
  enum { OFFSET = 3, TYPEID = BASEID + OFFSET };
public:
  bool isA(int id) {
    return id == TYPEID || Super::isA(id);
  }
  static Investment* dynacast(Security* s) {
    return (s->isA(TYPEID)) ?
      static_cast<Investment*>(s) : 0;
  }
  void special() {
    cout << "special Investment function" << endl;
  }
};

class Metal : public Investment {
  typedef Investment Super;
protected:
  enum { OFFSET = 4, TYPEID = BASEID + OFFSET };
public:
  bool isA(int id) {
    return id == TYPEID || Super::isA(id);
  }
  static Metal* dynacast(Security* s) {
    return (s->isA(TYPEID)) ? static_cast<Metal*>(s) : 0;
  }
};
```

553

554

```
int main() {
  vector<Security*> portfolio;
  portfolio.push_back(new Metal);
  portfolio.push_back(new Investment);
  portfolio.push_back(new Bond);
  portfolio.push_back(new Stock);
  for(vector<Security*>::iterator it = portfolio.begin();
       it != portfolio.end(); ++it) {
    Investment* cm = Investment::dynacast(*it);
    if(cm)
      cm->special();
    else
      cout << "not an Investment" << endl;
  }
  cout << "cast from intermediate pointer:" << endl;
  Security* sp = new Metal;
  Investment* cp = Investment::dynacast(sp);
  if(cp) cout << "  it's an Investment" << endl;
  Metal* mp = Metal::dynacast(sp);
  if(mp) cout << "  it's a Metal too!" << endl;
  purge(portfolio);
} ///:~
```

多态的**isA( )**函数检查其参数是否与它的类型参数（**id**）相容，就意味着或者**id**与对象的**typeID**准确地匹配，或者与对象的祖先之一的类型匹配（因此在这种情况下调用**Super::isA( )**）。函数**dynacast( )**在每个类中都是静态的，**dynacast( )**为其指针参数调用**isA( )**来检查类型转换是否有效。如果**isA( )**返回**true**，则说明类型转换是有效的，并且返回匹配的类型转换指针。否则返回空指针，这告诉调用者类型转换无效，意味着最初的指针没有指向与想要的类型（可转换到的类型）相容的对象。对于能够检查中间类型的类型转换来说，这种机制完全是必须的，例如在前面的程序例子中，从一个指向一个**Metal**对象的**Security**类型指针，转换为**Investment**指针。[⊖]

555

在面向对象的应用程序中，因为平常的多态性方案解决了绝大部分问题，对大多数程序来说向下类型转换是不必要的，并且在实际的程序设计中并不提倡。然而，对于像调试器、类浏览器和数据库观察器这些工具程序来说，具有检查多派生类型转换的能力是非常重要的。借助**dynamic_cast**操作符，C++提供这样一个可检查的类型转换。使用**dynamic_cast**对前面的程序例子进行重写，就得到下面的程序：

```
//: C08:Security.h
#ifndef SECURITY_H
#define SECURITY_H
#include <iostream>

class Security {
public:
  virtual ~Security() {}
};

class Stock : public Security {};
class Bond : public Security {};

class Investment : public Security {
puolic:
  void special() {
```

⊖ 借助微软的编译器，我们必须启用RTTI；在默认情况下这是不能使用的。启用它的命令行选项是/**GR**。

```
    std::cout << "special Investment function" <<std::endl;
  }
};

class Metal : public Investment {};
#endif // SECURITY_H ///:~
```

```
//: C08:CheckedCast2.cpp
// Uses RTTI's dynamic_cast.
#include <vector>
#include "../purge.h"
#include "Security.h"
using namespace std;

int main() {
  vector<Security*> portfolio;
  portfolio.push_back(new Metal);
  portfolio.push_back(new Investment);
  portfolio.push_back(new Bond);
  portfolio.push_back(new Stock);
  for(vector<Security*>::iterator it =
       portfolio.begin();
       it != portfolio.end(); ++it) {
    Investment* cm = dynamic_cast<Investment*>(*it);
    if(cm)
      cm->special();
    else
      cout << "not a Investment" << endl;
  }
  cout << "cast from intermediate pointer:" << endl;
  Security* sp = new Metal;
  Investment* cp = dynamic_cast<Investment*>(sp);
  if(cp) cout << "  it's an Investment" << endl;
  Metal* mp = dynamic_cast<Metal*>(sp);
  if(mp) cout << "  it's a Metal too!" << endl;
  purge(portfolio);
} ///:~
```

556

由于原来例子中大部分的代码开销用在了类型转换检查上，所以这个例子就变得如此之短。如同其他新式风格的C++类型转换（**static_cast**等）一样，**dynamic_cast**的目标类型放在一对尖括号中，并且转换对象以操作数的方式出现。如果想要安全地进行向下类型转换，**dynamic_cast**要求使用的目标对象的类型是多态的（polymorphic）。[⊖]这就要求该类必须至少有一个虚函数。幸运的是，**Security**基类有一个虚析构函数，所以这里不需要再创建一个额外的函数去做这项工作。因为**dynamic_cast**在程序运行时使用了虚函数表，所以比起其他新式风格的类型转换操作来说它的代价更高。

用引用而非指针同样也可以使用**dynamic_cast**，但是由于没有诸如空引用这样的情况，这就需要采用其他方法来了解类型转换是否失败。这个“其他方法”就是捕获**bad_cast**异常，如下所示：

557

```
//: C08:CatchBadCast.cpp
#include <typeinfo>
#include "Security.h"
using namespace std;

int main() {
  Metal m;
```

⊖ 编译器典型地将一个指向一个类的RTTI表的指针插入到它的虚函数表中。

```
  Security& s = m;
  try {
    Investment& c = dynamic_cast<Investment&>(s);
    cout << "It's an Investment" << endl;
  } catch(bad_cast&) {
    cout << "s is not an Investment type" << endl;
  }
  try {
    Bond& b = dynamic_cast<Bond&>(s);
    cout << "It's a Bond" << endl;
  } catch(bad_cast&) {
    cout << "It's not a Bond type" << endl;
  }
} ///:~
```

bad_cast类在**<typeinfo>**头文件中定义，并且像标准库的大多数的类一样，在**std**名字空间中声明。

8.2 typeid 操作符

获得有关一个对象运行时信息的另一个方法，就是用**typeid**操作符来完成。这种操作符返回一个**type_info**类的对象，该对象给出与其应用有关的对象类型的信息。如果该对象的类型是多态的，它将给出那个应用（动态类型（dynamic type））的大部分派生类信息；否则，它将给出静态类型信息。**typeid**操作符的一个用途是获得一个对象的动态类型的名称，例如**const char***，就像在下面例子中可以看到的。

558

```
//: C08:TypeInfo.cpp
// Illustrates the typeid operator.
#include <iostream>
#include <typeinfo>
using namespace std;

struct PolyBase { virtual ~PolyBase() {} };
struct PolyDer : PolyBase { PolyDer() {} };
struct NonPolyBase {};
struct NonPolyDer : NonPolyBase { NonPolyDer(int) {} };

int main() {
  // Test polymorphic Types
  const PolyDer pd;
  const PolyBase* ppb = &pd;
  cout << typeid(ppb).name() << endl;
  cout << typeid(*ppb).name() << endl;
  cout << boolalpha << (typeid(*ppb) == typeid(pd))
       << endl;
  cout << (typeid(PolyDer) == typeid(const PolyDer))
       << endl;
  // Test non-polymorphic Types
  const NonPolyDer npd(1);
  const NonPolyBase* nppb = &npd;
  cout << typeid(nppb).name() << endl;
  cout << typeid(*nppb).name() << endl;
  cout << (typeid(*nppb) == typeid(npd)) << endl;
  // Test a built-in type
  int i;
  cout << typeid(i).name() << endl;
} ///:~
```

这个使用一个特定编译器的程序的输出是:

```
struct PolyBase const *
struct PolyDer
true
true
struct NonPolyBase const *
struct NonPolyBase
false
int
```

因为**ppb**是一个指针，所以输出的第1行是它的静态类型。为了在程序中得到RTTI的结果，需要检查指针或引用目标对象，这在第2行中说明。需要注意的是，RTTI忽略了顶层的**const**和**volatile**限定符。借助非多态类型，正好可以获得静态类型（该指针本身的类型）。正如读者所见，这里也支持内置类型。 559

结果是：因为没有可访问的构造函数并且禁止赋值操作，所以在**type_info**对象中不能存储**typeid**操作的结果。必须像在演示中描述的那样来使用它。另外，通过**type_info::name( )**返回的实际字符串依赖于编译器。例如，对于一个名为**C**的类，某些编译器返回的是字符串“class C”而不是字符串“C”。把**typeid**应用到解析一个空指针的一个表达式将会引起一个**bad_typeid**异常被抛出（该异常也定义在**<typeinfo>**中）。

下面的例子显示由**type_info::name( )**返回那个类名是完全限定的。

```
//: C08:RTTIandNesting.cpp
#include <iostream>
#include <typeinfo>
using namespace std;

class One {
  class Nested {};
  Nested* n;
public:
  One() : n(new Nested) {}
  ~One() { delete n; }
  Nested* nested() { return n; }
};

int main() {
  One o;
  cout << typeid(*o.nested()).name() << endl;
} ///:~
```

因为**Nested**是**One**类的一个成员类型，所以结果是**One::Nested**。

在实现定义的“整理顺序”（对文本的自然排序规则）中，也可以用**before(type_info&)**询问一个**type_info**对象是否在另一个**type_info**对象之前。其返回值为**true**或**false**。当编写代码

```
if(typeid(me).before(typeid(you))) // ...
```

时，就是询问在当前的整理顺序中，**me**是否在**you**之前。如果把**type_info**对象作为关键字会是很有用处的。 560

8.2.1 类型转换到中间层次类型

就像读者在前面使用了**Security**类层次结构的程序中所看到的，**dynamic_cast**不仅能发现准确的类型，并且能在多层的继承层次结构中将类型转换到中间层类型。下面是另一个例子。

```
//: C08:IntermediateCast.cpp
#include <cassert>
#include <typeinfo>
```

```
using namespace std;

class B1 {
public:
  virtual ~B1() {}
};

class B2 {
public:
  virtual ~B2() {}
};

class MI : public B1, public B2 {};
class Mi2 : public MI {};

int main() {
  B2* b2 = new Mi2;
  Mi2* mi2 = dynamic_cast<Mi2*>(b2);
  MI* mi = dynamic_cast<MI*>(b2);
  B1* b1 = dynamic_cast<B1*>(b2);
  assert(typeid(b2) != typeid(Mi2*));
  assert(typeid(b2) == typeid(B2*));
  delete b2;
} ///:~
```

这个例子有关于多重继承的很复杂的情况（在本章后面部分和第9章将会学习到更多有关多重继承的知识）。如果创建一个**Mi2**对象并将它向上类型转换到该继承层次结构的根（在这种情况下，选择两个可能的根中的一个），可以成功地使**dynamic_cast**回退至两个派生层**MI**或**Mi2**中的任何一个。

561 甚至可以从一个根到另一个根进行类型转换：

```
B1* b1 = dynamic_cast<B1*>(b2);
```

这也是成功的，因为**B2**实际上指向一个**Mi2**对象，该**Mi2**对象含有一个**B1**类型的子对象。

将类型转换到中间层类型，使**dynamic_cast**和**typeid**两者之间产生一个有趣的差异。**typeid**操作符始终产生指向静态的**type_info**型对象的引用，它描述该对象的动态类型。因此，**typeid**操作符不能给出中间层对象的类型信息。在下面的表达式中（结果是**true**），像**dynamic_cast**一样，**typeid**并没有把**b2**当作指向派生类的指针：

```
typeid(b2) != typeid(Mi2*)
```

b2的类型只不过是指针类型：

```
typeid(b2) == typeid(B2*)
```

8.2.2 void型指针

RTTI仅仅为完整的类型工作，这就意味着当使用**typeid**时，所有的类信息都必须是可利用的。特别是，它不能与**void**型指针一起工作：

```
//: C08:VoidRTTI.cpp
// RTTI & void pointers.
//!#include <iostream>
#include <typeinfo>
using namespace std;

class Stimpy {
public:
  virtual void happy() {}
```

```
  virtual void joy() {}
  virtual ~Stimpy() {}
};

int main() {
  void* v = new Stimpy;
  // Error:
//!  Stimpy* s = dynamic_cast<Stimpy*>(v);
  // Error:
//!  cout << typeid(*v).name() << endl;
} ///:~
```

562

一个**void***真实地意思是“无类型信息”。[⊖]

8.2.3 运用带模板的RTTI

因为所有的类模板所做的工作就是产生类，所以类模板可以很好地与RTTI一起工作。RTTI提供了一个方便的途径来获得对象所在类的名称。下面的示例打印出构造函数和析构函数的调用顺序：

```
//: C08:ConstructorOrder.cpp
// Order of constructor calls.
#include <iostream>
#include <typeinfo>
using namespace std;

template<int id> class Announce {
public:
  Announce() {
    cout << typeid(*this).name() << " constructor" << endl;
  }
  ~Announce() {
    cout << typeid(*this).name() << " destructor" << endl;
  }
};

class X : public Announce<0> {
  Announce<1> m1;
  Announce<2> m2;
public:
  X() { cout << "X::X()" << endl; }
  ~X() { cout << "X::~X()" << endl; }
};

int main() { X x; } ///:~
```

这个模板用一个**int**常量把一个类和其他类区分开，但是也可使用类型参数。在构造函数和析构函数内部，RTTI信息产生打印的类名。类**X**利用继承和组合两个方式创建一个类，这个类有一个有趣的构造函数和析构函数的调用顺序。输出如下：

563

```
Announce<0> constructor
Announce<1> constructor
Announce<2> constructor
X::X()
X::~X()
Announce<2> destructor
Announce<1> destructor
Announce<0> destructor
```

⊖ **dynamic_cast<void*>**总是给出完全的对象而不是一个子对象的地址。在第9章中更详细地解释这一点。

当然，可能会得到不同的输出结果，这取决于编译器如何表示它的**name( )**信息。

8.3 多重继承

RTTI机制必须正确地处理多重继承的所有复杂性，包括虚基类**virtual**（在下章深入地进行讨论——在读过第9章之后，读者也许需要再回过头来看本节的内容）：

```
//: C08:RTTIandMultipleInheritance.cpp
#include <iostream>
#include <typeinfo>
using namespace std;

class BB {
public:
  virtual void f() {}
  virtual ~BB() {}
};

class B1 : virtual public BB {};
class B2 : virtual public BB {};
class MI : public B1, public B2 {};

int main() {
  BB* bbp = new MI; // Upcast
  // Proper name detection:
  cout << typeid(*bbp).name() << endl;
  // Dynamic_cast works properly:
  MI* mip = dynamic_cast<MI*>(bbp);
  // Can't force old-style cast:
//! MI* mip2 = (MI*)bbp; // Compile error
} ///:~
```

564

typeid( )操作符正确地检测出实际对象的名字，即便它是采用**virtual**基类指针来完成这个任务的，**dynamic_cast**也正确地进行工作。但实际上，编译器不允许程序员用以前的方法尝试强制进行类型转换：

```
MI* mip = (MI*)bbp; // Compile-time error
```

编译器知道这样做绝不是正确的方法，因此需要程序员使用**dynamic_cast**。

8.4 合理使用RTTI

因为使用RTTI能从一个匿名基类的多态指针上发现类型信息。初学者很容易误用它，因为在学会使用虚函数进行多态调用方法之前，使用RTTI很有效。对于许多有过程化编程背景的人来说，不将程序组织成**switch**语句的集合是很困难的。借助RTTI他们可以实现这个愿望，但这样就损失了多态性在代码开发和维护过程中的重要价值。C++的目的就是希望用虚函数的多态机制贯穿代码的始终，只在必须的时候使用RTTI。

然而，使用虚函数多态机制的方法调用，要求我们拥有基类定义的控制权，因为在程序扩充的某些地方，可能会发现基类并没有包含我们所需要的虚函数。如果基类来自一个库或者由别人控制，这时RTTI就是一种解决该问题的方案；可以派生一个新类，并且添加我们需要的成员函数。在程序代码的其他地方，可以检查到我们这个特定的类，并且调用它的成员函数。这样做不会破坏多态性和程序的扩展能力，因为添加这样一个新类将不需要在程序中搜索**switch**语句。然而，当需要在程序主体中增加所需的新特征的代码时，则必须使用RTTI来检查该特定的类型。

如果只是为了某个特定类的利益而在基类中放进某种新特性，这意味着由那个基类派生出的所有其他子类都为一个纯虚函数而需要保留这些毫无意义的东西。这将使接口变的更不清晰，因为我们必须覆盖由基类继承来的所有纯虚函数，这是很令人烦恼的。 565

最后一点，RTTI有时可以解决效率问题。如果你的程序漂亮地运用了多态性，但是某个对象是以一种极低效的方式达到这个目的的，那么就将那个类挑出来，使用RTTI，并通过为其编写特别的代码来提高效率。

垃圾再生器

为了更进一步地举例说明RTTI的实际用途，下面的程序模拟了一个垃圾再生器。不同种类的“垃圾”被插入到一个容器中，然后根据它们的动态类型进行分类。

```
//: C08:Trash.h
// Describing trash.
#ifndef TRASH_H
#define TRASH_H
#include <iostream>

class Trash {
  float _weight;
public:
  Trash(float wt) : _weight(wt) {}
  virtual float value() const = 0;
  float weight() const { return _weight; }
  virtual ~Trash() {
    std::cout << "~Trash()" << std::endl;
  }
};

class Aluminum : public Trash {
  static float val;
public:
  Aluminum(float wt) : Trash(wt) {}
  float value() const { return val; }
  static void value(float newval) {
    val = newval;
  }
};

class Paper : public Trash {
  static float val;
public:
  Paper(float wt) : Trash(wt) {}
  float value() const { return val; }
  static void value(float newval) {
    val = newval;
  }
};

class Glass : public Trash {
  static float val;
public:
  Glass(float wt) : Trash(wt) {}
  float value() const { return val; }
  static void value(float newval) {
    val = newval;
  }
};
#endif // TRASH_H ///:~
```

566

用来表示垃圾类型单价的**static**值定义在实现文件中:

```
//: C08:Trash.cpp {O}
// A Trash Recycler.
#include "Trash.h"

float Aluminum::val = 1.67;
float Paper::val = 0.10;
float Glass::val = 0.23;
///:~
```

sumValue( )模板从头到尾对一个容器进行迭代，显示并计算结果:

```
//: C08:Recycle.cpp
//{L} Trash
// A Trash Recycler.
#include <cstdlib>
#include <ctime>
#include <iostream>
#include <typeinfo>
#include <vector>
#include "Trash.h"
#include "../purge.h"
using namespace std;
// Sums up the value of the Trash in a bin:
template<class Container>
void sumValue(Container& bin, ostream& os) {
  typename Container::iterator tally = bin.begin();
  float val = 0;
  while(tally != bin.end()) {
    val += (*tally)->weight() * (*tally)->value();
    os << "weight of " << typeid(**tally).name()
       << " = " << (*tally)->weight() << endl;
    ++tally;
  }
  os << "Total value = " << val << endl;
}

int main() {
  srand(time(0)); // Seed the random number generator
  vector<Trash*> bin;
  // Fill up the Trash bin:
  for(int i = 0; i < 30; i++)
    switch(rand() % 3) {
      case 0 :
        bin.push_back(new Aluminum((rand() % 1000)/10.0));
        break;
      case 1 :
        bin.push_back(new Paper((rand() % 1000)/10.0));
        break;
      case 2 :
        bin.push_back(new Glass((rand() % 1000)/10.0));
        break;
    }
  // Note: bins hold exact type of object, not base type:
  vector<Glass*> glassBin;
  vector<Paper*> paperBin;
  vector<Aluminum*> alumBin;
  vector<Trash*>::iterator sorter = bin.begin();
  // Sort the Trash:
  while(sorter != bin.end()) {
    Aluminum* ap = dynamic_cast<Aluminum*>(*sorter);
    Paper* pp = dynamic_cast<Paper*>(*sorter);
```

567

```
    Glass* gp = dynamic_cast<Glass*>(*sorter);
    if(ap) alumBin.push_back(ap);
    else if(pp) paperBin.push_back(pp);
    else if(gp) glassBin.push_back(gp);
    ++sorter;
  }
  sumValue(alumBin, cout);
  sumValue(paperBin, cout);
  sumValue(glassBin, cout);
  sumValue(bin, cout);
  purge(bin);
} ///:~
```

568

因为垃圾被不加分类地投入到一个容器中，这样一来，垃圾的所有具体类型信息就“丢失”了。但是，为了稍后适当地对废料进行分类，具体类型信息必须恢复，这将用到RTTI。

可以通过使用**map**来改进这种解决方案，该**map**将指向**type_info**对象的指针与一个包含**Trash**指针的**vector**关联起来。因为映像需要一个能识别排序的判定函数，这里提供了一个名为**TInfoLess**的结构，它调用**type_info::before( )**。当将**Trash**指针插入到映像中的时候，这些指针将与**type_info**关键字自动关联。注意，这里必须对**sumValue( )**进行不同的定义。

```
//: C08:Recycle2.cpp
//{L} Trash
// Recyling with a map.
#include <cstdlib>
#include <ctime>
#include <iostream>
#include <map>
#include <typeinfo>
#include <utility>
#include <vector>
#include "Trash.h"
#include "../purge.h"
using namespace std;

// Comparator for type_info pointers
struct TInfoLess {
  bool operator()(const type_info* t1, const type_info* t2)
  const { return t1->before(*t2); }
};

typedef map<const type_info*, vector<Trash*>, TInfoLess>
  TrashMap;

// Sums up the value of the Trash in a bin:
void sumValue(const TrashMap::value_type& p, ostream& os) {
  vector<Trash*>::const_iterator tally = p.second.begin();
  float val = 0;
  while(tally != p.second.end()) {
    val += (*tally)->weight() * (*tally)->value();
    os << "weight of "
       << p.first->name()  // type_info::name()
       << " = " << (*tally)->weight() << endl;
    ++tally;
  }
  os << "Total value = " << val << endl;
}

int main() {
  srand(time(0)); // Seed the random number generator
```

569

```
  TrashMap bin;
  // Fill up the Trash bin:
  for(int i = 0; i < 30; i++) {
    Trash* tp;
    switch(rand() % 3) {
      case 0 :
        tp = new Aluminum((rand() % 1000)/10.0);
        break;
      case 1 :
        tp = new Paper((rand() % 1000)/10.0);
        break;
      case 2 :
        tp = new Glass((rand() % 1000)/10.0);
        break;
    }
    bin[&typeid(*tp)].push_back(tp);
  }
  // Print sorted results
  for(TrashMap::iterator p = bin.begin();
      p != bin.end(); ++p) {
    sumValue(*p, cout);
    purge(p->second);
  }
} ///:~
```

为了直接调用**type_info::name( )**，我们在这里修改了**sumValue( )**，因为作为**TrashMap:: value_type**对的第1个成员，**type_info**对象现在是可获得的。这样就避免了为了获得正在处理的**Trash**的类型名而额外调用**typeid**，而这在该程序的以前版本中却是必须做的。

570

8.5 RTTI的机制和开销

实现RTTI典型的方法是，通过在类的虚函数表中放置一个附加的指针。这个指针指向那个特别类型的**type_info**结构。**typeid( )**表达式的结果非常简单：虚函数表指针取得**type_info**指针，并且产生一个对**type_info**结构的引用。因为这正好是一个双指针的解析操作，这是一个代价为常量时间的操作。

对于**dynamic_cast<destination*>(source_pointer)**来说，大部分的情况是相当直观的：检索**source_pointer**的RTTI信息，并且为**destination***类型取得RTTI信息。然后，库程序确定**source_pointer**类型是否属于类型**destination***或**destination***的一个基类。如果该基类型不是派生类的第1个基类，那么由于多重继承的原因返回的指针将是被调整过的。在继承层次结构和虚基类的使用中，因为一个基类型可以出现多次，所以对于多重继承来说情况将会更加复杂。

因为为了**dynamic_cast**而使用的库程序必须从头至尾对一个基类表进行检查，**dynamic_cast**开销可能高于**typeid( )**（但是分别获得了不同的信息，这些信息对于问题的解决来说是必要的），找到一个基类比找到一个派生类可能需要花更多的时间。另外，**dynamic_cast**将任何一个类型与任何其他类型相比较；在同一层次结构中可以不受限制地进行类型比较。这就增加了由**dynamic_cast**使用的库程序的额外开销。

8.6 小结

尽管通常情况下会为一个指向其基类的指针进行向上类型转换，然后再使用那个基类的通用接口（通过虚函数），但是如果知道一个由基类指针指向的对象的动态类型，有时候根据获

得的这些信息来进行相关处理可能会使事情变得更加有效，而这些正是RTTI所提供的。大部分通常的误用来自于一些程序员，这些误用是由于他们不理解虚函数而是采用RTTI来做类型检查的编码所造成的。C++的基本原理似乎提供了对违反类型的定义规则和完整性的情况进行监督和纠正的强有力的工具和保护，但是如果有谁想故意地误用或回避某一个语言的特征，那么将没有什么人可以阻止他这样做。有时候误用导致的小错却是取得经验的最快方法。

8.7 练习

571

8-1 创建一个类**Base**，它有一个**virtual**虚析构函数，同时创建一个派生类**Derived**，它派生于类**Base**。创建一个存储**Base**指针的**vector**，该指针指向随机生成的**Base**和**Derived**对象。使用该**vector**的内容来填充另外一个包含所有**Derived**指针的另一个**vector**。比较**typeid( )**和**dynamic_cast**的执行时间，看哪一个执行得更快。

8-2 修改本教材第1卷中的**C16:AutoCounter.h**，使它成为一个有用的调试工具。它将作为那些与追踪有关的各个类的嵌套成员来使用。将**AutoCounter**修改为一个模板，它将外围类的类名作为模板的参数，并且在所有的出错信息中利用RTTI来打印类名。

8-3 通过使用**typeid( )** 打印出模板的准确的名称，用RTTI作为辅助工具进行程序的调试。实例化各种类型的模板，并看看它们的结果是什么。

8-4 通过先将**Wind5.cpp**复制到一新位置，修改第1卷第14章中的**Instrument**的层次结构。现在，在**Wind**类中新增加一个虚函数**clearSpitValve( )**，并且在继承自**Wind**的所有派生类中重新定义它。实例化一个存储**Instrument**指针的**vector**，并用**new**操作符创建各种类型的**Instrument**对象来填充它。现在使用RTTI在这样一个容器中遍历查找类**Wind**或**Wind**的派生类的对象。并对这些对象调用**clearSpitValve( )**函数。注意，如果在工具（Instrument）基类中含有一个**clearSpitValve( )**函数，那么将会使该基类发生使人不愉快的混乱。

8-5 修改上一个练习，在该基类中放置一个**prepareInstrument( )**函数，它需要调用适当的函数（例如，在它适宜的时候调用**clearSpitValve( )**）。注意，**prepareInstrument( )**是放置在基类中的一个明智的函数，它的使用剔除了在上一个练习题中对RTTI的需要。

8-6 创建一个含有指针的**vector**，这些指针指向10个随机**Shape**对象（例如，至少是若干个**Square**和**Circle**）。重写每个具体的类中的**draw( )**成员函数，用于打印输出被画对象的尺寸（任何一个应用的长度或半径）。编写一个**main( )**程序，首先画出容器中所有的**Square**，并按其长度进行排序，然后再画出所有的**Circle**，并按其半径进行排序。 572

8-7 创建一个大的**vector**，它存储那些指向随机**Shape**对象的指针。在**Shape**中编写一个非虚（non-virtual）**draw( )**函数，使用RTTI来确定每个对象的动态类型，并且借助开关（switch）语句执行适当的代码来“画出”对象。然后使用虚函数，“用正确的方法”重新编写**Shape**的层次结构。比较两种方法的实现代码长度和执行时间。

8-8 创建一个关于**Pet**类的层次结构，其中包括**Dog**、**Cat**和**Horse**。再创建一个关于**Food**类的层次结构：其中包括**Beef**、**Fish**和**Oats**。**Dog**类有一个成员函数**eat( )**，其参数为**Beef**，同样**Cat::eat( )**将**Fish**对象作为其参数，而**Oats** 对象则作为参数传递给**Horse::eat( )**。创建这样一个**vector**，它含有指向随机生成的**Pet**对象的指针，并且访问每个**Pet**，并将正确的**Food**对象类型传递给对应的**eat( )**函数。

8-9 建立一个名为**drawQuad( )**的全局函数，它使用一个**Shape**对象的引用作为参数。如果它有4条边（也就是说，它是**Square**或**Rectangle**），那么它将调用其含有**Shape**参数的**draw( )**函数。否则将打印消息“不是一个四边形”。遍历这个包含指向随机生成的**Shape**对象指针的**vector**，在遍历时对每个被访问对象调用**drawQuad( )**。在**vector**中，放置那些指向**Square**、**Rectangle**、**Circle**和**Triangle**对象的指针。

8-10 根据类名对一个含有随机**Shape**对象的**vector**排序。用**type_info::before( )**作为排序的比较函数。

第9章 多重继承

多重继承（MI）的基本概念听起来相当简单：通过继承多个基类来创建一个新类。确切地说这种多重继承语法正是我们所期望的，并且只要继承层次结构图是简单的，那么多重继承也同样简单。

尽管MI可能引入一些二义性和奇怪的案例，在本章将对这些案例进行讨论。但是首先，这些案例将有助于读者对该主题获得一些基本认识。

9.1 概论

在C++之前，最成功的面向对象的语言是Smalltalk。Smalltalk是作为一种完全的面向对象语言而创造出来的。它被称作是纯粹的（pure）面向对象语言，而C++则被称作是一种混合的（hybrid）语言，这是因为C++支持多种形式的程序设计范例，而不仅仅只是面向对象的程序设计范例。一个由Smalltalk做出的设计本身就决定了所有类都是在一个单一的继承层次结构中派生的，都以一个基类作为根（称为**Object**——这就是基于对象的继承层次结构（object-based hierarchy）模型）。[⊖]在Smalltalk中，不可能创建这样一个新类：它不是派生自一个现存的类。这就是为什么在Smalltalk中实现多种形式的继承方式要花费大量的时间：在开始建立新类前，必须学习和掌握类库。因此Smalltalk的类继承层次结构是一棵单一的整体树。

Smalltalk中的类通常有很多的共同点，并且总是有某些共通的东西（**Object**的特征和行为），所以不会经常遇上需要从多个基类继承的情况。然而，在C++中却可以建立用户想要的多种不同的继承树。所以为了逻辑上的完整性，该语言必须有能力一次组合多个类——因而需要多重继承。

然而，程序员对多重继承的需求并不是显而易见的。关于在C++中多重继承是否是必要的这一问题存在着（现在仍然存在着）大量的争论。1989年在AT&T **cfront** 发布版（release） 2.0中加入了MI，这也是C++语言1.0版以来发生的首次重要的变化。[⊖]从那以后，许多其他的特征被加入到标准C++中（最著名的是模板），这些变化改变了编程的思想并且使MI的作用处于次要的地位。程序设计人员可以把MI看作是一个“次要”的语言特征，也就是说，在日常的程序设计决定中很少涉及它。

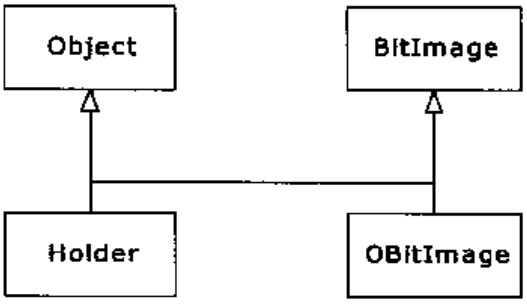
有关MI最激烈的争论之一涉及容器。假如要想建立这样一个容器，每个人都可以很容易地使用它。一种方法是将**void***作为该容器内部的类型。然而Smalltalk的方法是建立一个持有**Object**对象的容器，因为**Object**是Smalltalk继承层次结构的基类型。Smalltalk中的所有内容最终都派生自**Object**，所以持有**Object**的容器可以存储任何类型的对象。

现在考虑在C++中的情况。假设供应商**A**建立了一个基于对象的继承层次结构，该继承层次结构包括了一组有用的容器，这些容器中就包含想要使用的一种称为**Holder**的容器。接下来偶然遇到了供应商**B**提供的类继承层次结构，它包含了其他一些比较重要的类，例如**BitImage**类，它持有生动的图像。制造一个持有这些**BitImage**特征和行为的**Holder**容器的惟一方法，是创建一个派生自**Object**和**BitImage**两者的新类，这样，在**Holder**中就可以

⊖ 对Java和其他面向对象的语言来说这也是正确的。

⊖ 这些版本号是国际AT&T的编号方式。

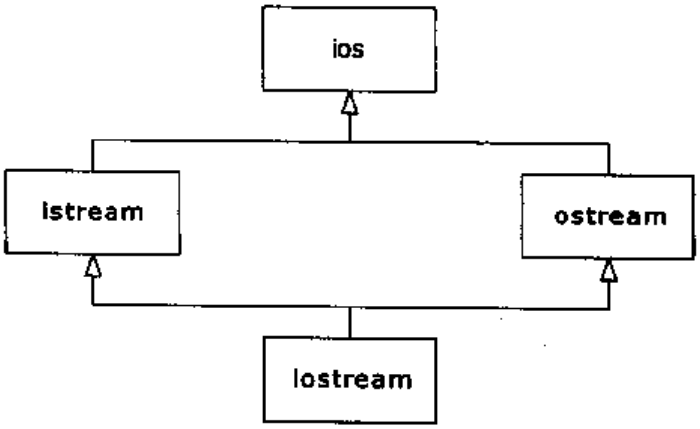
持有**BitImage**中的特征和行为:



这似乎是需要MI的一个重要理由，而许多类库就是建立在这种模型之上的。然而如第5章所述，模板的加入改变了创建容器的方法。所以这种情况不再是使用MI的动力。

575

而需要MI的另外一个原因跟设计有关。可以有意地用MI来使一个程序设计得更加灵活或更实用（至少表面上是这样）。在原始的**iostream**库设计中就有这样的一个例子（仍存在于目前的模板设计中，如第4章所述）:



istream和**ostream**就其自身来说都是有用的类，但是也可以通过从一个类同时派生出这两个类的方式产生它们，而该基类将这两个类的特征和行为结合在一起。类**ios**提供了所有这些流类的共同点，在这种情况下MI就是一种代码分解机制。

不管是什么原因激发我们使用MI，但是要真正使用它将比看上去要难得多。

9.2 接口继承

多重继承中毫无争议的一种运用属于*接口继承*（interface inheritance）。在C++中，所有的继承都是*实现继承*（implementation inheritance），因为在一个基类、接口和实现中的任何内容都将成为派生类的一部分。只继承一个类的某些部分（比如只继承接口）是不可能的。就像第1卷第14章说明的那样，当客户使用派生类的对象时，私有的和被保护的继承将可能限制派生类成员对基类成员的访问，但是这些并不会影响派生类；它仍然包含了所有的基类数据，并且可以访问所有的非私有的基类成员。

576

另一方面，接口继承仅仅是在一个派生类接口中加入了成员函数的*声明*（declaration），在C++中并不直接支持这种使用方法。C++中模拟接口继承常见的技术是从一个仅包含声明（没有数据和函数体）的*接口类*（interface class）派生一个类。除了析构函数以外，这些声明都是纯虚函数。举例如下:

```
//: C09:Interfaces.cpp
// Multiple interface inheritance.
#include <iostream>
#include <sstream>
#include <string>
using namespace std;

class Printable {
public:
  virtual ~Printable() {}
  virtual void print(ostream&) const = 0;
};

class Intable {
public:
  virtual ~Intable() {}
  virtual int toInt() const = 0;
};

class Stringable {
public:
  virtual ~Stringable() {}
  virtual string toString() const = 0;
};

class Able : public Printable, public Intable,
             public Stringable {
  int myData;
public:
  Able(int x) { myData = x; }
  void print(ostream& os) const { os << myData; }
  int toInt() const { return myData; }
  string toString() const {
    ostringstream os;
    os << myData;
    return os.str();
  }
};

void testPrintable(const Printable& p) {
  p.print(cout);
  cout << endl;
}

void testIntable(const Intable& n) {
  cout << n.toInt() + 1 << endl;
}

void testStringable(const Stringable& s) {
  cout << s.toString() + "th" << endl;
}

int main() {
  Able a(7);
  testPrintable(a);
  testIntable(a);
  testStringable(a);
} ///:~
```

577

类**Able**“实现”了接口**Printable**、**Intable**和**Stringable**，因为它提供了那些对它们进行声明的函数的实现。因为**Able**派生自所有这3个类，**Able**对象具有多“is-a”关系。例如，对象**a**的行为可能像一个**Printable**对象，因为它的类**Able**公有派生自**Printable**，并且提供

了对**print( )**的实现。测试函数并不需要知道作为参数使用的大多数的派生类对象的类型；它仅仅需要这样一个可以代替它们的参数类型的对象。

一般来说，采用模板来解决问题的方法将会使程序变得更加简洁：

```
//: C09:Interfaces2.cpp
// Implicit interface inheritance via templates.
#include <iostream>
#include <sstream>
#include <string>
using namespace std;
class Able {
  int myData;
public:
  Able(int x) { myData = x; }
  void print(ostream& os) const { os << myData; }
  int toInt() const { return myData; }
  string toString() const {
    ostringstream os;
    os << myData;
    return os.str();
  }
};

template<class Printable>
void testPrintable(const Printable& p) {
  p.print(cout);
  cout << endl;
}

template<class Intable>
void testIntable(const Intable& n) {
  cout << n.toInt() + 1 << endl;
}

template<class Stringable>
void testStringable(const Stringable& s) {
  cout << s.toString() + "th" << endl;
}

int main() {
  Able a(7);
  testPrintable(a);
  testIntable(a);
  testStringable(a);
} ///:~
```

578

Printable、**Intable**和**Stringable**这些名字现在仅是模板参数，这些参数假设在各自的语境中表示存在的操作。换句话说，测试函数可以接受任何一种类型的参数，这些参数类型与正确的识别标志和返回类型一起提供了一个成员函数的定义；这些参数并不必要派生自一个共同的基类。有些人更适应第1种版本的示例，因为类型名可以通过继承关系保证能够确定，该继承关系由预期的接口来实现。而其他人更满足于这样一个事实，如果提供的模板类型参数不能满足测试函数所需要的操作，该错误在编译时仍然会被捕获。对比前一个方法（继承），后面的方法是一种“较弱”的类型检查形式，但是对程序员（和程序）来说，效果是相同的。这就是被许多现代C++程序员所接受的弱输入检查的一种形式。

579

9.3 实现继承

如前所述，C++仅仅提供了实现继承，这就意味着所有的内容总是继承自基类。这样做有

很大的好处，因为它将使程序员从不得不在派生类中实现所有的细节（正如前面的例子中所采用接口继承所做的事情）中解放出来。多重继承的一个共同用途包括使用混入类（mixin）、这些混入类的存在是为了通过继承来增加其他类的功能。混入类不能刻意的由它本身进行实例化。

举个例子，假设一个客户使用了某个类，该类支持访问一个数据库。在这个情况下，仅仅有一个头文件可以使用——在这里指出，客户不能访问实现具体功能的这部分源代码。举例说明，假定**Database**类的实现如下所示:

```
//: C09:Database.h
// A prototypical resource class.
#ifndef DATABASE_H
#define DATABASE_H
#include <iostream>
#include <stdexcept>
#include <string>

struct DatabaseError : std::runtime_error {
  DatabaseError(const std::string& msg)
    : std::runtime_error(msg) {}
};

class Database {
  std::string dbid;
public:
  Database(const std::string& dbStr) : dbid(dbStr) {}
  virtual ~Database() {}
  void open() throw(DatabaseError) {
    std::cout << "Connected to " << dbid << std::endl;
  }
  void close() {
    std::cout << dbid << " closed" << std::endl;
  }
  // Other database functions...
};
#endif // DATABASE_H ///:~
```

580

这里已经省略了实际的数据库功能（存储操作、检索操作，等等），但是在这里那些功能并不重要。使用这个类需要一个数据库连接串，并调用**Database::open( )**来连接数据库，通过调用**Database::close( )**断开连接:

```
//: C09:UseDatabase.cpp
#include "Database.h"

int main() {
  Database db("MyDatabase");
  db.open();
  // Use other db functions...
  db.close();
}
/* Output:
connected to MyDatabase
MyDatabase closed
*/ ///:~
```

在一个典型的客户机-服务器模式的情况下，客户拥有多个对象，这些对象分享一个连接的数据库。尽管数据库的最后关闭是非常重要的，但数据库只能在不再需要访问它之后关闭。通常，将这种行为封装到一个类中，用来实现对使用数据库连接的客户实体的数目进行跟踪，并且在实体计数归为零时自动终止数据库的连接。为了给**Database**类加入引用计数，利用多重继

承将一个叫**Countable**的类混入**Database**类中，这样就创建了一个新类**DBConnection**。这就是**Countable** 混入类:

```
//: C09:Countable.h
// A "mixin" class.
#ifndef COUNTABLE_H
#define COUNTABLE_H
#include <cassert>

class Countable {
  long count;
protected:
  Countable() { count = 0; }
  virtual ~Countable() { assert(count == 0); }
public:
  long attach() { return ++count; }
  long detach() {
    return (--count > 0) ? count : (delete this, 0);
  }
  long refCount() const { return count; }
};
#endif // COUNTABLE_H ///:~
```

581

很明显，这不是一个独立类，因为它的构造函数是**protected**类型；它需要一个友元或派生类来使用它。析构函数是虚函数这一点非常重要，因为它只被**detach( )**中的**delete this**语句调用，并且需要将派生对象正确地销毁。[⊖]

DBConnection类继承了**Database** 和 **Countable**，并且提供了一个静态的**create( )**函数，这个函数用来初始化它的**Countable**子对象。这是将在第10章中讨论的工厂方法(Factory Method）设计模式的一个例子:

```
//: C09:DBConnection.h
// Uses a "mixin" class.
#ifndef DBCONNECTION_H
#define DBCONNECTION_H
#include <cassert>
#include <string>
#include "Countable.h"
#include "Database.h"
using std::string;

class DBConnection : public Database, public Countable {
  DBConnection(const DBConnection&); // Disallow copy
  DBConnection& operator=(const DBConnection&);
protected:
  DBConnection(const string& dbStr) throw(DatabaseError)
  : Database(dbStr) { open(); }
  ~DBConnection() { close(); }
public:
  static DBConnection*
  create(const string& dbStr) throw(DatabaseError) {
    DBConnection* con = new DBConnection(dbStr);
    con->attach();
    assert(con->refCount() == 1);
    return con;
  }
  // Other added functionality as desired...
```

582

⊖ 尽管这很重要，但是我们不需要未定义的行为。对一个基类来说没有一个虚析构函数将是一个错误。

```
};
#endif // DBCONNECTION_H ///:~
```

不用修改**Database**类，现在就有一个引用计数的数据库连接，并且可以确保数据库连接不会被偷偷地终止。通过**DBConnection**的构造函数和析构函数，使用第1章中提到的资源获取式初始化（the Resource Acquisition Is Initialization，RAII）方法来实现数据库的打开和关闭。这就使得**DBConnection**的使用变得很容易：

```
//: C09:UseDatabase2.cpp
// Tests the Countable "mixin" class.
#include <cassert>
#include "DBConnection.h"

class DBClient {
  DBConnection* db;
public:
  DBClient(DBConnection* dbCon) {
    db = dbCon;
    db->attach();
  }
  ~DBClient() { db->detach(); }
  // Other database requests using db…
};

int main() {
  DBConnection* db = DBConnection::create("MyDatabase");
  assert(db->refCount() == 1);
  DBClient c1(db);
  assert(db->refCount() == 2);
  DBClient c2(db);
  assert(db->refCount() == 3);
  // Use database, then release attach from original create
  db->detach();
  assert(db->refCount() == 2);
} ///:~
```

583

因为对**DBConnection::create( )**的调用又调用了**attach( )**，所以在结束时，必须显式调用**detach( )**来释放数据库的初始连接。注意，**DBClient**类也用RAII管理连接的使用。当程序结束时，这两个**DBClient**对象的析构函数将分别使引用计数减1（通过调用**detach( )**完成，这里的**DBConnection**继承自**Countable**），在对象**c1**被销毁后，当引用计数达到零时数据库连接将被关闭（因为调用了**Countable**的虚析构函数）。

模板方法一般用于混入继承，允许用户在编译时指定想要的混入类的类型。这样就可以使用不同的引用计数方法来完成这项工作，而不用显式地两次定义**DBConnection**。下面这个例子说明了这种方法是如何工作的：

```
//: C09:DBConnection2.h
// A parameterized mixin.
#ifndef DBCONNECTION2_H
#define DBCONNECTION2_H
#include <cassert>
#include <string>
#include "Database.h"
using std::string;

template<class Counter>
class DBConnection : public Database, public Counter {
  DBConnection(const DBConnection&); // Disallow copy
  DBConnection& operator=(const DBConnection&);
```

```
protected:
  DBConnection(const string& dbStr) throw(DatabaseError)
  : Database(dbStr) { open(); }
  ~DBConnection() { close(); }
public:
  static DBConnection* create(const string& dbStr)
  throw(DatabaseError) {
    DBConnection* con = new DBConnection(dbStr);
    con->attach();
    assert(con->refCount() == 1);
    return con;
  }
  // Other added functionality as desired...
};
#endif // DBCONNECTION2_H ///:~
```

584

这里惟一的变化是用于类定义的模板前缀（以及为了清楚起见而将**Countable**重新命名为**Counter**）。也可以把某个数据库类作为一个模板参数（可以从多个数据库访问类中进行选择），但它并不是一个混入类，因为它是一个独立类。尽管下面的例子将原始的**Countable**作为**Counter**混入类型使用，但是可以使用实现了适当的接口（**attach( )**、**detach( )**等等）的任何类型。

```
//: C09:UseDatabase3.cpp
// Tests a parameterized "mixin" class.
#include <cassert>
#include "Countable.h"
#include "DBConnection2.h"

class DBClient {
  DBConnection<Countable>* db;
public:
  DBClient(DBConnection<Countable>* dbCon) {
    db = dbCon;
    db->attach();
  }
  ~DBClient() { db->detach(); }
};

int main() {
  DBConnection<Countable>* db =
    DBConnection<Countable>::create("MyDatabase");
  assert(db->refCount() == 1);
  DBClient c1(db);
  assert(db->refCount() == 2);
  DBClient c2(db);
  assert(db->refCount() == 3);
  db->detach();
  assert(db->refCount() == 2);
} ///:~
```

多参数混入类型的一般模式很简单：

585

```
template<class Mixin1, class Mixin2, … , class MixinK>
class Subject : public Mixin1,
                public Mixin2,
                …
                public MixinK {…};
```

9.4 重复子对象

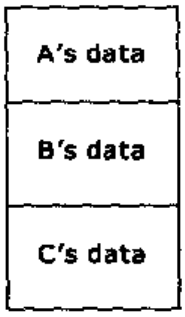
当从某个基类继承时，可以在其派生类中得到那个基类的所有数据成员的副本。下面的程

序说明了多个基类子对象在内存中的可能布局：[⊖]

```
//: C09:Offset.cpp
// Illustrates layout of subobjects with MI.
#include <iostream>
using namespace std;

class A { int x; };
class B { int y; };
class C : public A, public B { int z; };

int main() {
  cout << "sizeof(A) == " << sizeof(A) << endl;
  cout << "sizeof(B) == " << sizeof(B) << endl;
  cout << "sizeof(C) == " << sizeof(C) << endl;
  C c;
  cout << "&c == " << &c << endl;
  A* ap = &c;
  B* bp = &c;
  cout << "ap == " << static_cast<void*>(ap) << endl;
  cout << "bp == " << static_cast<void*>(bp) << endl;
  C* cp = static_cast<C*>(bp);
  cout << "cp == " << static_cast<void*>(cp) << endl;
  cout << "bp == cp? " << boolalpha << (bp == cp) << endl;
  cp = 0;
  bp = cp;
  cout << bp << endl;
}
/* Output:
sizeof(A) == 4
sizeof(B) == 4
sizeof(C) == 12
&c == 1245052
ap == 1245052
bp == 1245056
cp == 1245052
bp == cp? true
0
*/ ///:~
```

586

正如读者所见，对象**c**的**B**子对象部分从整个对象开始位置偏移了4个字节。其布局如下所示：



对象**c**以它的**A**子对象作为开头，然后是**B**子对象部分，最后的数据完全来自类型**C**本身。因为**C**“is-an”[⊜]**A**并且也“is-a”**B**，所以它可以向上类型转换为两者之中任一基类型。当向

⊖ 实际的布局在实现时确定。

⊜ “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”。见《C++编程思想第1卷：标准C++导引》第1章。——编辑注

上类型转换为A时，结果指针指向A子对象部分，这发生在C对象的开始位置，所以地址ap等同于表达式&c。然而，当向上类型转换为B子对象时，结果指针必须指向B子对象所在的实际位置，因为类B并不知道有关类C（或类A，就本例而言）的事情。换句话说，被bp指向的对象必须能够产生和独立的B对象相同的行为（除了任何需要多态的行为以外）。

当对bp进行类型转换倒退为一个C*时，由于原始对象是C，bp指向该对象开始的位置，因为它已经知道B子对象的位置，所以指针被调整指向了完整对象的起始地址。如果bp指向的是一个独立的B对象而不是C对象的开始位置，那么这种类型转换就是不合法的。[⊖]此外，在比较表达式bp == cp中，cp被隐式转换为B*，这是使该比较表达式变得有意义的惟一方法（这就是说，向上类型转换总是允许的），因此结果是true。因此，当在子对象和完整类型间来回转换时，要应用适当的偏移量。

587

空指针需要进行特别的处理。显然，如果在开始进行类型转换时指针为零，那么在转换到一个B子对象或从一个B子对象转换回来时，由于盲目地减去偏移量将会导致产生无效的地址。基于这种原因，当类型转换到B*或有来自B*的类型转换时，编译器产生逻辑检查，首先查看该指针是否为零。如果不为零，则可以应用偏移量；否则，当指针为零时就放弃使用偏移量。

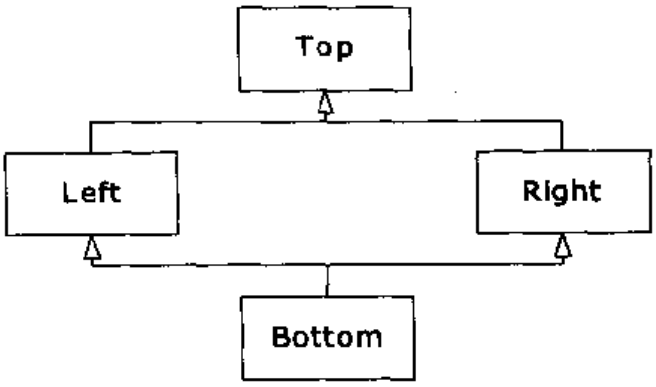
根据目前学习到的语法，如果现在有多个基类，并且如果这些基类依次有一个共同的基类，那么将得到顶层基类的两个副本。如下面的例子所示：

```
//: C09:Duplicate.cpp
// Shows duplicate subobjects.
#include <iostream>
using namespace std;

class Top {
  int x;
public:
  Top(int n) { x = n; }
};

class Left : public Top {
  int y;
public:
  Left(int m, int n) : Top(m) { y = n; }
};

class Right : public Top {
  int z;
public:
  Right(int m, int n) : Top(m) { z = n; }
};
class Bottom : public Left, public Right {
  int w;
public:
  Bottom(int i, int j, int k, int m)
  : Left(i, k), Right(j, k) { w = m; }
};

int main() {
  Bottom b(1, 2, 3, 4);
  cout << sizeof b << endl; // 20
} ///:~
```

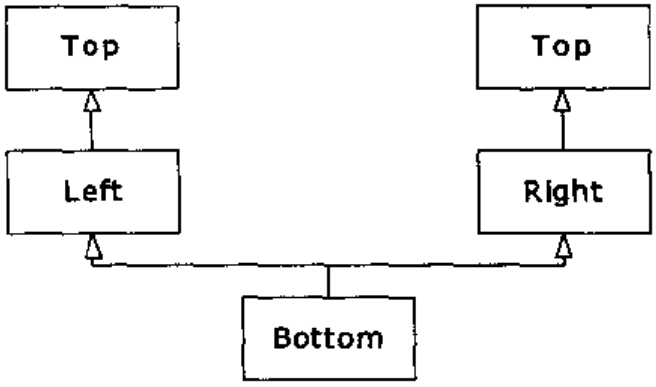
588

⊖ 但并不作为一个错误被检查。dynamic_cast可以解决这个问题。看前面章节的详细说明。

因为对象**b**的长度是20个字节，[⊖]所以在一个完整的**Bottom**对象中共有5个整型变量。这种情况的典型类图通常如下所示:



这就是所谓的“菱形继承”（也称“钻石继承”），但是在这个例子中可以较好地表示为如下的类图:



589

这种设计的不足之处表现在前面代码中**Bottom**类的构造函数上。用户认为只需要4个整型变量，但是哪些实际参数才是传递给**Left**和**Right**所需要的两个参数呢？尽管这一设计并不是固有的“错误”，但通常它并不是一个应用程序所需要的。在尝试将指向**Bottom**对象的指针转换成指向**Top**的指针时，同样也会出现问题。如前所述，可能需要调整对象指针的地址，这依赖于在完整的对象内部各子对象所处的位置，但是这里却有两个**Top**子对象供选择。因为编译器不知道选择那一个，所以这样一种向上类型转换是模棱两可的（二义性），也是不允许的。用同样的原因可以解释为什么一个**Bottom** 对象不能调用那个只定义在**Top**中的函数。如果存在这样一个函数**Top::f( )**，那么调用**b.f( )**需要涉及一个**Top**子对象作为执行语境，而这里却有两个**Top**可供选择。

9.5 虚基类

在这种情况下通常需要的是*真正的菱形继承*，**Left**和**Right**子对象在一个完整的**Bottom**对象内部共享着一个**Top**对象，这正是第1个类图描述的情况。这是通过使**Top**成为**Left**和**Right**的一个*虚基类*（virtual base class）来完成的:

⊖ 即**5*sizeof(int)**。因为编译器可以加入任意的数据类型，所以对象的长度至少是它各部分的总和，也可以更长。

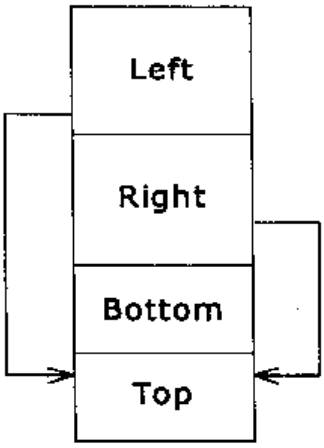
```
//: C09:VirtualBase.cpp
// Shows a shared subobject via a virtual base.
#include <iostream>
using namespace std;
class Top {
protected:
  int x;
public:
  Top(int n) { x = n; }
  virtual ~Top() {}
  friend ostream&
  operator<<(ostream& os, const Top& t) {
    return os << t.x;
  }
};

class Left : virtual public Top {
protected:
  int y;
public:
  Left(int m, int n) : Top(m) { y = n; }
};

class Right : virtual public Top {
protected:
  int z;
public:
  Right(int m, int n) : Top(m) { z = n; }
};

class Bottom : public Left, public Right {
  int w;
public:
  Bottom(int i, int j, int k, int m)
  : Top(i), Left(0, j), Right(0, k) { w = m; }
  friend ostream&
  operator<<(ostream& os, const Bottom& b) {
    return os << b.x << ',' << b.y << ',' << b.z
      << ',' << b.w;
  }
};

int main() {
  Bottom b(1, 2, 3, 4);
  cout << sizeof b << endl;
  cout << b << endl;
  cout << static_cast<void*>(&b) << endl;
  Top* p = static_cast<Top*>(&b);
  cout << *p << endl;
  cout << static_cast<void*>(p) << endl;
  cout << dynamic_cast<void*>(p) << endl;
} ///:~
```

590

591

给定类型的各个虚基类都涉及相同的对象，不论它在层次结构的哪个地方出现。[⊖]这意味着，当一个**Bottom**对象被实例化时，对象的布局看起来像下面的样子：

⊖ 使用术语层次结构（hierarchy）因为人人都在使用它，但是用来表示多重继承关系的图一般是一个有向无环图（DAG），也称为网格，其理由是显而易见的。



Left和**Right**子对象各有一个指向共享的**Top**子对象的指针（或某种概念上等价的对象），并且对**Left**和**Right**成员函数中那个子对象的所有引用都要通过这些指针来完成。[⊖]在这里，当从一个**Bottom**向上类型转换为一个**Top**对象时，不存在二义性的问题，因为这里只有一个**Top**对象可用来进行转换。

前面程序的输出结果如下：

```
36
1,2,3,4
1245032
1
1245060
1245032
```

592

打印出来的地址说明这种特殊的实现确实在完整的对象的结尾处储存**Top**子对象（尽管实际上它在那里并不重要）。**dynamic_cast**到**void***的结果总是确定指向完整对象的地址。

尽管在技术上这样做是不合法的，[⊖] 但是如果去掉了虚析构函数（和**dynamic_cast**语句，这样程序将可以通过编译），那么**Bottom**的长度将减少到24个字节。似乎**Bottom**的长度的减少量正好等于3个指针的大小。为什么呢？

重要的是不必太按照字面上的意思去推敲这些数字。当加入虚构造函数时，使用其他编译器处理仅使该类占用的空间增加4个字节。因为我们不是编译器的编写者，所以无法得知编译器的秘密。然而可以确定的是，一个带有多重继承的派生对象必须表现出它好像有多个VPTR，它的每个含有虚函数的直接基类都有一个。就是那么简单。编译器的作者不论发明什么样的最优化技术，但是这些编译器必须产生相同的行为。

在前面的代码中，最奇怪的事情是**Bottom**构造函数中对**Top**的初始化程序。正常的情况下，不必担心直接基类以外的子对象的初始化，因为所有的类只照料它们的直接基类的初始化。然而，由于从**Bottom**到**Top**有多个继承路径，因此依赖于中间类**Left**和**Right**将必需的初始化数据传递给基类导致了二义性——谁负责进行基类的初始化呢？基于这个原因，*最高层派生类*（most derived class）必须初始化一个虚基类。但是也对**Top**进行初始化的**Left**和**Right**构造函数中的表达式应该如何编写呢？当创建独立的**Left**或**Right**对象时，这些初始化表达式确实是必需的，但是当创建**Bottom**对象时，它们必须被忽略（因此，在**Bottom**构造函数的初

⊖ 这些指针的出现说明为什么**B**的长度远大于4个整型变量的长度。这是虚基类的部分开销。还有VPTR的开销，这归因于虚析构函数。

⊖ 再说明一次，基类必须有虚析构函数，但是大部分编译器都能使这个例子编译通过。

始化程序中，这些表达式都为零——当**Left**和**Right**的构造函数在**Bottom**对象的语境中执行时，这些位置上的任何值都将被忽略）。编译器为程序员处理所有这一切，但是了解责任所在还是很重要的。必须始终保证，多重继承层次结构中的所有具体的（非抽象的）类都知道任何虚基类并对它们进行相应的初始化。

593

这些责任规则不仅仅适用于类的初始化，而且适用于所有跨越类继承层次结构的操作。现在考虑前面代码中的流插入符。使数据成为保护的，这样就可以“骗取”和访问**operator<<(ostream&, const Bottom&)**中继承来的数据。通常将打印各子对象的工作分配到其各个相应的类来进行，并且在需要的时候让派生类调用它的基类函数，这样做更有意义。就像下面的代码的说明，如果在程序中尝试使用**operator<<( )**，将会出现什么情况？

```
//: C09:VirtualBase2.cpp
// How NOT to implement operator<<.
#include <iostream>
using namespace std;

class Top {
  int x;
public:
  Top(int n) { x = n; }
  virtual ~Top() {}
  friend ostream& operator<<(ostream& os, const Top& t) {
    return os << t.x;
  }
};

class Left : virtual public Top {
  int y;
public:
  Left(int m, int n) : Top(m) { y = n; }
  friend ostream& operator<<(ostream& os, const Left& l) {
    return os << static_cast<const Top&>(l) << ',' << l.y;
  }
};

class Right : virtual public Top {
  int z;
public:
  Right(int m, int n) : Top(m) { z = n; }
  friend ostream& operator<<(ostream& os, const Right& r) {
    return os << static_cast<const Top&>(r) << ',' << r.z;
  }
};

class Bottom : public Left, public Right {
  int w;
public:
  Bottom(int i, int j, int k, int m)
  : Top(i), Left(0, j), Right(0, k) { w = m; }
  friend ostream& operator<<(ostream& os, const Bottom& b){
    return os << static_cast<const Left&>(b)
      << ',' << static_cast<const Right&>(b)
      << ',' << b.w;
  }
};

int main() {
  Bottom b(1, 2, 3, 4);
  cout << b << endl;  // 1,2,1,3,4
} ///:~
```

594

在通常处理方式中不能盲目地向上分摊责任，因为**Left**和**Right**每个流插入程序都调用了**Top**流插入程序，并且再出现数据的副本。另外，这里需要模仿编译器的初始化办法。一种解决办法是在类中提供特殊的函数，这种函数知道有关虚基类的情况，在打印输出的时候忽略虚基类（而将工作留给最高层派生类）：

```
//: C09:VirtualBase3.cpp
// A correct stream inserter.
#include <iostream>
using namespace std;

class Top {
  int x;
public:
  Top(int n) { x = n; }
  virtual ~Top() {}
  friend ostream& operator<<(ostream& os, const Top& t) {
    return os << t.x;
  }
};

class Left : virtual public Top {
  int y;
protected:
  void specialPrint(ostream& os) const {
    // Only print Left's part
    os << ','<< y;
  }
public:
  Left(int m, int n) : Top(m) { y = n; }
  friend ostream& operator<<(ostream& os, const Left& l) {
    return os << static_cast<const Top&>(l) << ',' << l.y;
  }
};

class Right : virtual public Top {
  int z;
protected:
  void specialPrint(ostream& os) const {
    // Only print Right's part
    os << ','<< z;
  }
public:
  Right(int m, int n) : Top(m) { z = n; }
  friend ostream& operator<<(ostream& os, const Right& r) {
    return os << static_cast<const Top&>(r) << ',' << r.z;
  }
};

class Bottom : public Left, public Right {
  int w;
public:
  Bottom(int i, int j, int k, int m)
  : Top(i), Left(0, j), Right(0, k) { w = m; }
  friend ostream& operator<<(ostream& os, const Bottom& b){
    os << static_cast<const Top&>(b);
    b.Left::specialPrint(os);
    b.Right::specialPrint(os);
    return os << ',' << b.w;
  }
};
```

595

```
int main() {
  Bottom b(1, 2, 3, 4);
  cout << b << endl;  // 1,2,3,4
} ///:~
```

596 **specialPrint( )**函数是**protected**的，因为它们只能被**Bottom**调用。**specialPrint( )**函数只输出自己的数据并忽略**Top**子对象，因为当这些函数被调用时，**Bottom**插入程序将获得控制权。像**Bottom**的构造函数一样，**Bottom**插入程序也必须知道虚基类。同样的推理适用于具有虚基类的层次结构中的赋值操作符，也适用想要分担层次结构中所有类的工作的任何成员函数或非成员函数。

在讨论了虚基类后，现在可以举例说明对象初始化的“全部情节”。因为虚基类引起共享子对象，共享发生之前它们就应该存在才有意义。所以子对象的初始化顺序遵循如下的规则递归地进行:

1) 所有虚基类子对象，按照它们在类定义中出现的位置，从上到下、从左到右初始化。
2) 然后非虚基类按通常顺序初始化。
3) 所有的成员对象按声明的顺序初始化。
4) 完整的对象的构造函数执行。

下面的程序举例说明了这个过程:

```
//: C09:VirtInit.cpp
// Illustrates initialization order with virtual bases.
#include <iostream>
#include <string>
using namespace std;

class M {
public:
  M(const string& s) { cout << "M " << s << endl; }
};

class A {
  M m;
public:
  A(const string& s) : m("in A") {
    cout << "A " << s << endl;
  }
  virtual ~A() {}
};
```

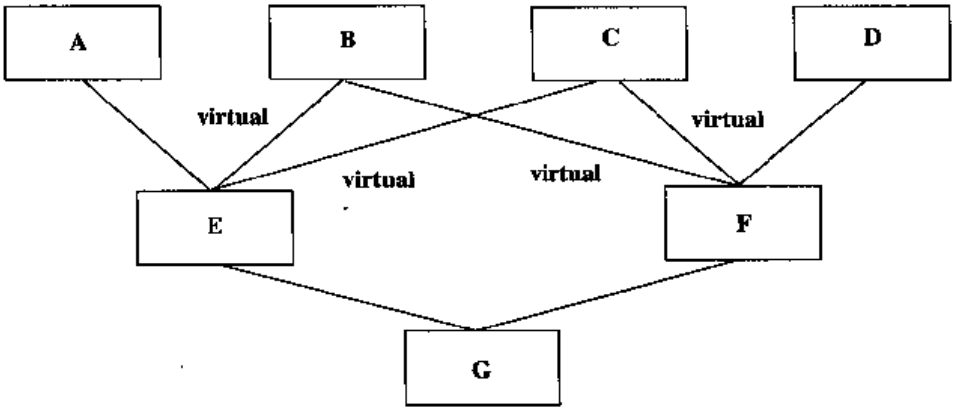
597

```
class B {
  M m;
public:
  B(const string& s) : m("in B")  {
    cout << "B " << s << endl;
  }
  virtual ~B() {}
};

class C {
  M m;
public:
  C(const string& s) : m("in C")  {
    cout << "C " << s << endl;
  }
  virtual ~C() {}
};

class D {
```

```
  M m;
public:
  D(const string& s) : m("in D") {
    cout << "D " << s << endl;
  }
  virtual ~D() {}
};

class E : public A, virtual public B, virtual public C {
  M m;
public:
  E(const string& s) : A("from E"), B("from E"),
  C("from E"), m("in E") {
    cout << "E " << s << endl;
  }
};

class F : virtual public B, virtual public C, public D {
  M m;
public:
  F(const string& s) : B("from F"), C("from F"),
  D("from F"), m("in F") {
    cout << "F " << s << endl;
  }
};
class G : public E, public F {
  M m;
public:
  G(const string& s) : B("from G"), C("from G"),
  E("from G"),  F("from G"), m("in G") {
    cout << "G " << s << endl;
  }
};

int main() {
  G g("from main");
} ///:~
```

598

这段代码中的类可以用下图表示:



每一个类都有一个嵌入的**M**类型的成员。注意，只有4个派生类是虚拟的: **E**派生自**B**和**C**、**F**派生自**B**和**C**。这个程序的输出结果是:

```
M in B
B from G
M in C
C from G
```

```
M in A
A from E
M in E
E from G
M in D
D from F
M in F
F from G
M in G
G from main
```

599

g的初始化需要首先初始化它的E和F部分，但是B和C子对象首先被初始化，因为它们是虚基类，并且二者的初始化在G的构造函数的初始化程序中进行，G是最高层派生类。类B没有基类，所以根据第3条规则，它的成员对象m 被初始化，然后它的构造函数打印输出“B from G”，对于E的C子对象处理相同。E子对象的初始化需要先对A、B和C子对象进行初始化。因为B和C已经被初始化，于是E子对象的A子对象接着被初始化，然后是E子对象自己初始化。相同的情况重复出现在g的 F子对象上，但是虚基类的初始化不重复进行。

9.6 名字查找问题

我们已经以子对象举例说明的二义性适用于任何名字，包括函数名。如果一个类有多个直接基类，就可以共享这些基类中那些同名的成员函数，如果要调用这些成员函数中的一个，那么编译器将不知道调用它们之中的哪一个。下面的程序举例将会报告这样一个错误:

```
//: C09:AmbiguousName.cpp {-xo}

class Top {
public:
  virtual ~Top() {}
};

class Left : virtual public Top {
public:
  void f() {}
};

class Right : virtual public Top {
public:
  void f() {}
};

class Bottom : public Left, public Right {};
int main() {
  Bottom b;
  b.f(); // Error here
} ///:~
```

600

类Bottom已经继承了两个同名的函数（因为名字查寻发生在重载解析之前，所以识别标志是不恰当的），并且没有方法在它们之间进行选择。通常消除二义性调用的方法，是以基类名来限定函数的调用:

```
//: C09:BreakTie.cpp

class Top {
public:
  virtual ~Top() {}
};
```

```
class Left : virtual public Top {
public:
  void f() {}
};

class Right : virtual public Top {
public:
  void f() {}
};

class Bottom : public Left, public Right {
public:
  using Left::f;
};

int main() {
  Bottom b;
  b.f(); // Calls Left::f()
} ///:~
```

现在在**Bottom**的作用域中可以找到名字**Left::f**，所以完全不用考虑名字**Right::f**的查找问题。为了介绍**Left::f( )**函数所能提供的更多额外功能，需要实现调用函数**Left::f( )**的**Bottom::f( )**函数。

在一个层次结构中的不同分支上存在的同名函数常常发生冲突。下面的继承层次结构不存在这样的问题：

601

```
//: C09:Dominance.cpp

class Top {
public:
  virtual ~Top() {}
  virtual void f() {}
};

class Left : virtual public Top {
public:
  void f() {}
};

class Right : virtual public Top {};

class Bottom : public Left, public Right {};

int main() {
  Bottom b;
  b.f(); // Calls Left::f()
} ///:~
```

程序在这里没有显式调用**Right::f( )**。因为**Left::f( )**是位于层次结构的最高层派生类，所以对**b.f( )**语句的执行将调用**Left::f( )**。为什么呢？现在假设**Right**不存在，这样就成为一个单一层次结构**Top <= Left <= Bottom**。在这里可以确定地预期由表达式**b.f( )**调用的函数是**Left::f( )**，因为一般的作用域规则是：一个派生类被认为嵌套在基类的作用域之内。一般情况下，如果类**A**直接或间接派生自类**B**，或换句话说，在继承层次结构中类**A**比类**B**处于“更高的派生层次”，[⊖]那么名字**A::f**就比名字**B::f**占优势（dominate）。因此，在同名的两个函数之间进

⊖ 注意，对这个例子来说虚拟继承是至关重要的。如果**Top**不是虚基类，将存在多个虚**Top**子对象，并且二义性还将存在。多重继承的优越性只与虚基类一同存在。

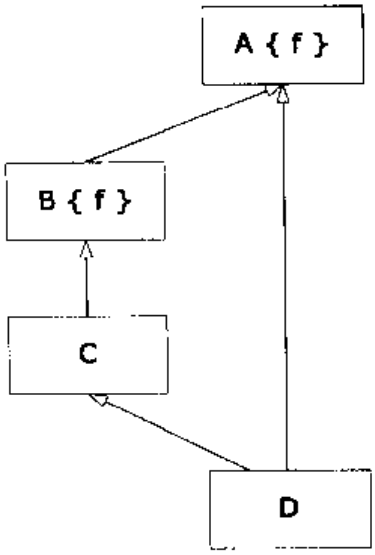
行选择时，编译器将选择占优势的那个函数。如果没有占优势的名字，就会产生二义性。

下面的程序更进一步地举例说明了占优势的原则:

```
//: C09:Dominance2.cpp
#include <iostream>
using namespace std;

class A {
public:
  virtual ~A() {}
  virtual void f() { cout << "A::f\n"; }
};

class B : virtual public A {
public:
  void f() { cout << "B::f\n"; }
};

class C : public B {};
class D : public C, virtual public A {};

int main() {
  B* p = new D;
  p->f(); // Calls B::f()
  delete p;
} ///:~
```

602

这个层次结构的类图如下:



603

类**A**是类**B**的基类（在这个例子中是直接基类），所以名字**B::f**比名字**A::f**占优势。

9.7 避免使用多重继承

当提到关于是否使用多重继承的问题时，至少要回答如下两个问题:

1) 是否需要通过新类来显示两个类的公共接口？（换句话说，如果一个类能够包含在另一个类中，那么仅有它的某些接口暴露在一个新类中。）

2) 需要向上类型转换成为两个基类类型吗？（当基类的数量多于两个时也适用。）

如果可以对上面任何一个问题回答“不是”，那么就可以避免使用MI，并且应该这样做。

请看这样的情况，一个类只是作为一个函数的参数需要向上进行类型转换。在这种情况

下，这个类就可以被嵌入，并且在新类中有一个自动类型转换函数提供产生一个指向嵌入的对象的引用。任何时候，如果要将新类的一个对象作为参数传递给某个期盼嵌入对象的函数，这就需要使用类型转换函数。[⊖]然而，类型转换不能用于普通的多态成员函数的选择；那需要使用继承机制来完成。推荐使用组合而不使用继承，从总体上来说这是个不错的设计指导原则。

9.8 扩充一个接口

多重继承最好的应用之一，涉及由第3方提供的脱离了程序员控制的代码。假设获得了这样一个库，它由一个头文件和一些编译好的成员函数组成，但没有这些成员函数的源代码。这个库是一个带有虚函数的类层次结构，并且包含一些能将指针指向类库中基类的全局函数；就是说，它使用了这些库对象的多态性。现在，假设使用这个库创建一个应用程序并利用基类的多态性编写了程序员自己的代码。 604

在软件项目开发的后期或在其维护期间，程序员可能发现由软件供应商提供的基类的接口并没有提供所需要的功能：提供的函数可能是非虚函数，但现在却需要它是个虚函数，或者接口中的虚函数完全地失效了，而该虚函数对于问题的解决却是至关重要的。使用多重继承可以解决这个问题。

例如，这里就是你得到的库的一个头文件：

```
//: C09:Vendor.h
// Vendor-supplied class header
// You only get this & the compiled Vendor.obj.
#ifndef VENDOR_H
#define VENDOR_H

class Vendor {
public:
  virtual void v() const;
  void f() const; // Might want this to be virtual...
  ~Vendor(); // Oops! Not virtual!
};

class Vendor1 : public Vendor {
public:
  void v() const;
  void f() const;
  ~Vendor1();
};

void A(const Vendor&);
void B(const Vendor&);
// Etc.
#endif // VENDOR_H ///:~
```

假设这个库很大，由多个派生类和一个较大的接口组成。注意，它还包括了函数**A( )**和**B( )**，这些函数都有一个基类对象的引用，并且都能利用多态性对其进行处理。这里是库的实现文件： 605

```
//: C09:Vendor.cpp {O}
// Assume this is compiled and unavailable to you.
#include "Vendor.h"
```

⊖ Jerry Schwarz，输入输出流（iostream）的作者，曾在个别场合表示如果他重新设计iostream的话，很可能从iostream的设计中去除多重继承，而采用多重流缓冲区和转换运算符。

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

void Vendor::v() const { cout << "Vendor::v()" << endl; }

void Vendor::f() const { cout << "Vendor::f()" << endl; }

Vendor::~Vendor() { cout << "~Vendor()" << endl; }

void Vendor1::v() const { cout << "Vendor1::v()" << endl; }

void Vendor1::f() const { cout << "Vendor1::f()" << endl; }

Vendor1::~Vendor1() { cout << "~Vendor1()" << endl; }

void A(const Vendor& v) {
  // ...
  v.v();
  v.f();
  // ...
}

void B(const Vendor& v) {
  // ...
  v.v();
  v.f();
  // ...
} ///:~
```

一般很难在用户自己的软件项目中获得这个源代码。而获得的只不过是像**Vendor.obj**或**Vendor.lib**（或与使用的系统相配的文件后缀）这样编译好的文件。

606 问题发生在对这个库的使用中。首先，析构函数不是虚函数。[⊖]另外，**f( )**没有被设计为虚函数；在这里，假定库的创造者决定它不需要是虚函数。用户还可能发现，作为基类的接口缺少解决问题所必要的函数。还可以假设用户已经编写了利用现有接口的一些代码（更不用说函数**A( )**和**B( )**，它们已经超出了用户的控制），并且不想修改它。

为了补救这个问题，用户可以创建一个自己的类接口，并且采用多重继承方法产生一组新的派生类，这些新派生类派生自用户创建的类接口和已存在的类：

```
//: C09:Paste.cpp
//{L} Vendor
// Fixing a mess with MI.
#include <iostream>
#include "Vendor.h"
using namespace std;

class MyBase { // Repair Vendor interface
public:
  virtual void v() const = 0;
  virtual void f() const = 0;
  // New interface function:
  virtual void g() const = 0;
  virtual ~MyBase() { cout << "~MyBase()" << endl; }
};

class Paste1 : public MyBase, public Vendor1 {
public:
```

⊖ 人们已经在商品化的C++库中看到了这点，至少在一些早期的库中是这样。

```
  void v() const {
    cout << "Paste1::v()" << endl;
    Vendor1::v();
  }
  void f() const {
    cout << "Paste1::f()" << endl;
    Vendor1::f();
  }
  void g() const { cout << "Paste1::g()" << endl; }
  ~Paste1() { cout << "~Paste1()" << endl; }
};

int main() {
  Paste1& p1p = *new Paste1;
  MyBase& mp = p1p; // Upcast
  cout << "calling f()" << endl;
  mp.f();  // Right behavior
  cout << "calling g()" << endl;
  mp.g(); // New behavior
  cout << "calling A(p1p)" << endl;
  A(p1p); // Same old behavior
  cout << "calling B(p1p)" << endl;
  B(p1p);  // Same old behavior
  cout << "delete mp" << endl;
  // Deleting a reference to a heap object:
  delete &mp; // Right behavior
} ///:~
```

607

在**MyBase**（它没有使用MI）中，**f( )**和析构函数现在都是虚函数，并且在接口中加入了一个新的虚函数**g( )**。现在，原始库中的每个派生类都必须重新创建，并在新的接口中利用MI混合。函数**Paste1::v( )**和**Paste1::f( )**只需要调用原始基类中的成员函数的版本。但是现在，如果将派生类对象向上类型转换为**MyBase**，就像在**main( )**中：

```
MyBase* mp = p1p; // Upcast
```

任何通过**mp**执行的函数调用都将是多态的，包括**delete**。同样地，新的接口函数**g( )**也可以通过**mp**来调用。下面是程序的输出结果：

```
calling f()
Paste1::f()
Vendor1::f()
calling g()
Paste1::g()
calling A(p1p)
Paste1::v()
Vendor1::v()
Vendor::f()
calling B(p1p)
Paste1::v()
Vendor1::v()
Vendor::f()
delete mp
~Paste1()
~Vendor1()
~Vendor()
~MyBase()
```

608

原始的库函数**A( )**和**B( )**仍然能够照常工作（假设新函数**v( )**调用了它的基类版本）。现在析构函数是**virtual**的，并且展现了正确的行为。

虽然这个例子有点混乱，但在实践中确实发生过，并且这个例子清晰地演示了哪里需要使用多重继承：必须能够将派生类向上类型转换为两个基类类型。

9.9 小结

C++中MI存在的一个原因是因为C++是一种混合语言，并且不能像Smalltalk和Java那样实现一个整体的类层次结构。而C++允许形成许多继承树，所以有时可能需要将来自两棵或多棵树的接口关联形成一个新类。

如果在类的层次结构中没有“菱形”的继承结构出现，MI将是相当简单的（虽然基类中完全相同的那些函数识别标志仍然必须解析）。如果有菱形继承结构出现，就需要通过引入虚基类来消除重复子对象。这不仅增加了混乱，而且使接下来的表达方式变得更加复杂和低效。

多重继承已经被称做“百分之90的goto语句”。[⊖]这种形容似乎是适当的，像避免使用goto语句那样在平常的编程中最好避免使用MI，但有时候它却很有用。它在C++中的地位是“次要的”，但却是C++的更高级特征，这一特征设计是用来解决特殊情况下出现的问题。如果读者发现自己经常使用了它，那么就需要检查一下使用它的原因。问一下自己，“是必需要向上类型转换成为所有的基类类型吗？”如果答案是否定的，假如嵌入的所有类的实例都不需要进行向上类型转换，那么编程工作将会变的更加简单。

609

9.10 练习

9-1 创建一个基类X，这个类包含具有一个int型参数的一个构造函数、一个返回类型为void的无参成员函数f( )。现在从基类X派生出类Y和Z，并为它们各自创建一个具有一个int型参数的构造函数。然后，再从类Y和Z派生出类A。创建类A的一个对象，并且为这个对象调用f( )。利用显式消除二义性的方法来解决问题。

9-2 以练习9-1的结果作为开始，创建一个指向基类X的名为px的指针，将前面创建的类A的对象地址赋值给px。利用虚基类解决这个问题。现在调整基类X，这样就不必再为X内部的A调用构造函数。

9-3 以练习9-2的结果作为开始，删除对f( )使用显式消除二义性的方法，并且看看是否可以通过px调用f( )。跟踪它看看哪个函数被调用了。解决这个问题，使得在类的继承层次结构中可以调用正确的函数。

9-4 与一个makeNoise( )函数声明一起，构造一个Animal接口类。与savePersonFromFire( )函数声明一起，构造一个SuperHero接口类。在这两个接口类中放置一个move( )函数声明。（记住构造接口的方法是使用纯虚函数。）现在定义3个单独的类：SuperlativeMan、Amoeba（一个性情无常的超级英雄）和TarantulaWoman；当Amoeba和Tarantula Woman实现Animal和SuperHero接口的时候，SuperlativeMan实现SuperHero的接口。定义两个全局函数animalSound(Animal*)和saveFromFire(SuperHero*)。寻求用这两个函数能够通过每个接口调用相应对象的所有方法。

9-5 重复上面的练习，但是利用模板而不是继承来实现接口，就像在Interfaces2.cpp中做的那样。

9-6 定义若干代表超级英雄（superhero）能力的具体的混入类（例如StopTrain、

⊖ Zack Urlocker创造的一个短语。

BendSteel和**ClimbBuilding**等等）。重新完成练习9-4，从这些混入类中派生出**SuperHero**派生类，并且调用它们的成员函数。

9-7 利用模板重复上面的练习，用强大的超级英雄（superhero）混入类作为模板参数。利用模板强大的功能更好地为其他类服务。 610

9-8 从练习9-4中撤销**Animal**接口，重新定义**Amoeba**使其仅实现**SuperHero**。现在定义一个从两个类**SuperlativeMan**和**Amoeba**继承来的**SuperlativeAmoeba**类。试着将**SuperlativeAmoeba**对象作为参数传递给**saveFromFire( )**。为了让上面的调用正确进行，需要做些什么？如何使用虚继承来改变对象的长度？

9-9 继续上面的练习，为练习9-4的**SuperHero**增加一个整型**strengthFactor**数据成员，并在构造函数中对其进行初始化。在3个派生类中也加入构造函数来初始化**strengthFactor**。在**SuperlativeAmoeba**中，必须要做哪些不同的工作？

9-10 继续上面的练习，分别给两个类**SuperlativeMan**和**Amoeba**（但不包括**SuperlativeAmoeba**）增加一个**eatFood( )**成员函数，这样两个版本的**eatFood( )**函数获得不同的食物对象的类型（所以这两个函数的识别标志不同）。在**SuperlativeAmoeba**中调用两个**eatFood( )**函数中的任一个时必须做哪些工作？为什么？

9-11 为**SuperlativeAmoeba**定义一个能够正确工作的输出流插入符和赋值操作符。

9-12 从层次结构中删除**SuperlativeAmoeba**，并且修改**Amoeba** 使它派生自两个类**SuperlativeMan**（它也是由**SuperHero**派生）和**SuperHero**。在**SuperHero**和**SuperlativeMan**（采用完全相同的识别标志）中实现一个虚函数**workout( )**，并且以**Amoeba**对象调用这个函数。哪个函数被调用了？

9-13 用组合而非继承重新定义**SuperlativeAmoeba**，使它的行为类似于**SuperlativeMan**或**Amoeba**。利用转换运算符提供隐式向上类型转换。将这种方法和继承方法进行比较。

9-14 假设有一个预先编译好的**Person**类（即只有头文件和编译好的目标文件）。假设**Person**还有一个非虚函数**work( )**。通过从**Person**派生和使用**Person::work( )**的一个实现，让**SuperHero**能够成为一个行为适度且遵守规矩的普通的**Person**，而让**SuperHero::work( )**成为虚函数。

9-15 定义一个引用计数错误的日志混入类**ErrorLog**，持有一个静态文件流，可以用这个流发送消息。当引用计数大于零时该类打开流，当引用计数归零时（始终附加在文件上）关闭流。让多个类的对象能够向静态日志流发送消息。通过**ErrorLog**中的跟踪语句，观察流的打开和关闭。 611

9-16 修改**BreakTie.cpp**，在其中加入派生（非虚派生）自**Bottom**的名为**VeryBottom**的类。除非在**using**中对**f**的声明将“左”变为“右”，**VeryBottom**应该看起来就和**Bottom**一样。修改**main( )**函数，实例化一个**VeryBottom**而非**Bottom**对象。请问，将调用哪个**f( )**？

613

第10章 设计模式

"……描述一个在我们周围一再出现的问题，然后描述解决这个问题的核心方法，这样就能够无数次地使用这个解决方法而不必重复劳动。"——**Christopher Alexander**

本章介绍程序设计的重要和非传统的"模式"方法。

"设计模式"（design pattern）运动或许是在面向对象设计方法学前进过程中的最新、最重要的一步，最初把设计模式概念载入编年史的，是Gamma、Helm、Johnson和Vlissides等4人合编的《Design Patterns》（Addison Wesley，1995）[⊖]一书，这本书一般也被称为"四人帮"（Gang of Four，GoF）书。"四人帮"针对问题的特定类型提出了23种解决方案。在本章中，讨论设计模式的基本概念并且给出一些代码示例，用以说明精选出来的设计模式。希望这样能够促使大家研读更多关于设计模式的资料，设计模式当今已经成为面向对象程序设计的几乎所有必须掌握的语汇的重要源泉[⊜]。

10.1 模式的概念

最初，可以将模式看做解决某一类特定问题的特别巧妙和具有洞察力的方法。它体现出一个开发团队从一个问题的所有角度出发做出全面的分析后，提出的最通用最灵活的对这类问题的解决方案。这类问题也许是读者以前曾经遇到和解决过的问题，但是读者那种解决方案大概没有即将看到的在模式中体现出的完整性。此外，模式的存在独立于任何特定的实现方法，我们可以用多种方法来实现它。

614

虽然称为"设计模式"，它们实际上与设计领域并无联系。模式与传统的关于分析、设计和实现的思想方法有所不同。模式体现了一个程序内部完整思想，因此它也能够跨越分析阶段和高层设计阶段。然而，因为模式常常有一个直接的代码实现，所以在底层设计或编码实现之前很难将其表示出来（在进入这些阶段之前，人们可能不会认识到需要某种特定的模式）。

可以把模式的基本概念看做一般情况下程序设计的基本概念：增加一些抽象层。当人们对某事物进行抽象的时候，隔离特定的细节，最直接的动机之一是为了使变化的事物与不变的事物分离开。做到这一点的另一个方法是，一旦发现程序中的某些部分可能被修改，那么就要阻止那些修改在代码中到处传播副作用。如果做到了这一点，代码不仅比较容易阅读和理解，而且也比较容易维护——这样做会带来一个注定的结果，那就是在软件开发的全过程中降低成本。

开发一种优雅和可维护的软件设计最困难的部分，常常是发现所谓"变化向量"（the vector of change）。（在这里，"向量"应理解为自然科学中的最大梯度，而不是一个容器类。）这就意味着寻找系统中变化的最重要的事物，换句话说，去寻找系统中开发成本最高的地方。一旦找到这个"变化向量"，就可以围绕这个焦点来构建系统的设计。

因此，设计模式的目标是封装变化（encapsulate change）。如果从这点来看，在本书中，

⊖ 为方便起见，书中的例子都是使用C++描述的；遗憾的是这种标准出现在C++前的方言缺乏一些诸如STL容器等现代语言特征。

⊜ 许多材料来源于"Thinking in Patterns:Problem-Solving Techniques using Java"，可从网站www.MindView.net上得到。

读者已经看到了一些设计模式。例如，可以把继承（inheritance）想像为一个设计模式（尽管是由编译器提供的一个实现）。它表示行为不同（这是变化的事物）的一些对象，它们具有相同的接口（这就是所谓不变的事物）。组合（composition）也可以被认为是一种模式，因为可以改变——动态或静态地改变——实现类的对象，因此而改变类的工作方式。然而正常情况下，由编程语言直接支持的特性不能被归类为设计模式。

读者也已经看到了“四人帮”书中出现的另外一个模式：迭代器（iterator）。在STL的设计中它被当作基本的工具来使用，这在本教材中较早时已经讨论过。当分步骤和依次挑选容器中的元素时，迭代器隐藏容器的特殊的实现。允许使用迭代器编写通用的代码，对某一范围内的所有元素进行操作，而不必关心保存这些元素的容器。因此，这些通用的代码可以和任何能够生成迭代器的容器一起使用。 615

组合优于继承

“四人帮”的最重要的贡献也许并不在于提出了模式的概念，而在于在该书的第1章中介绍的那句格言：“对象组合优于类继承”。理解继承和多态性如此具有挑战性，以至于人们可能对这些技术赋予了不适当的重要性。我们看到许多由于“继承嗜好”而导致的过于复杂的设计（包括我们自己的设计）——比如，由于坚持到处使用继承致使许多多重继承设计得到发展。

《极限编程》（Extreme programming）的指导原则之一是“只要能用，就做最简单的”。一个似乎需要继承的设计常常能够戏剧性地使用组合来代替而大大简化，从而使其更加灵活，在学习过本章中的一些设计模式之后读者将会理解这一点。因此，在考虑一个设计时，问问自己：“使用组合是不是更简单？这里真的需要继承吗？它能带来什么好处？”

10.2 模式分类

“四人帮”讨论了23个模式，按照下面3种目的分类（对所有模式都围绕可能变化的特定方面考虑）：

1) **创建型（Creational）**：用于怎样创建一个对象。通常包括隔离对象创建的细节，这样代码不依赖于对象是什么类型，因此在增加一种新的对象类型时不需要改变代码。本章将介绍单件（Singleton）模式、工厂（Factory）模式和构建器(Builder) 模式。

2) **结构型（Structural）**：影响对象之间的连接方式，确保系统的变化不需要改变对象间的连接。结构型模式常常由工程项目限制条件来支配。本章中将看到代理（Proxy）模式和适配器(Adapter) 模式。 616

3) **行为型（Behavioral）**：在程序中处理具有特定操作类型的对象。这些对象封装要执行的操作过程，比如解释一种语言、实践一个请求、遍历一个序列（如像在一个迭代器内）或者实现一个算法。本章包含命令（Command）模式、模板方法（Template Method）模式、状态（State）模式、策略（Strategy）模式、职责链（Chain of Responsibility）模式、观察者（Observer）模式、多派遣（Multiple Dispatching）模式和访问者（Visitor）模式的例子。

“四人帮”对23个模式的每个模式都用一节篇幅进行讨论，给出一个或多个例子，这些例子通常用C++描述，有时也用Smalltalk描述。本书不重复在“四人帮”书中阐述的那些模式的细节，既然那本书自成体系，就应该单独学习。在此提供的描述和例子旨在给读者一个关于模式的大致理解，以便对模式是什么和模式的重要性有一个感性的认识。

特征、习语和模式

接下来的内容已经超出“四人帮”书中的范围。自从“四人帮”的书出版以来，出现了更

多的模式，对于定义设计模式有了更细致的过程。[⊖]这是重要的，因为识别新的模式或适当地描述它们是不容易的。例如，对于什么是设计模式，在流行的文献中有一些混乱的定义和描述。模式不是琐碎的，它们通常也不是由某种编程语言中内部特征来表现。例如，构造函数和析构函数可以被称为“保证初始化和清除的设计模式”。对面向对象编程来说，这些是重要的和必要的结构，但它们是常规语言的特征，还没有丰富到足以被看成设计模式的地步。

另外一个非模式的例子就是来自各种形式的聚合。聚合是面向对象编程中的一个完全基本的原则：用一些对象制造另一些对象。然而，有时这种办法被错误地归类为一种模式。这是遗憾的，因为它打乱了设计模式的思想，暗示人们将第1次看到且感到惊奇的任何事物都归结为一种设计模式。

617 Java语言提供了另外一个使人误解的例子：JavaBeans 规格说明的设计者决定把简单的“get/set”命名约定称为一种设计模式（比如，**getInfo( )**返回一个**Info**属性，而**setInfo( )**改变这个属性）。这只是一个普通的命名约定，而不能构成设计模式。

10.3 简化习语

在讨论更复杂的技术之前，看一些能够保持代码简明的基本方法是有帮助的。

10.3.1 信使

信使（messenger）[⊜]是这些方法中最微不足道的一个，它将消息封装到一个对象中到处传递，而不是将消息的所有片段分开进行传递。注意，没有信使，下面例子中的**translate( )**的代码读起来将非常缺乏条理：

```
//: C10:MessengerDemo.cpp
#include <iostream>
#include <string>
using namespace std;

class Point { // A messenger
public:
  int x, y, z; // Since it's just a carrier
  Point(int xi, int yi, int zi) : x(xi), y(yi), z(zi) {}
  Point(const Point& p) :  x(p.x), y(p.y), z(p.z) {}
  Point& operator=(const Point& rhs) {
    x = rhs.x;
    y = rhs.y;
    z = rhs.z;
    return *this;
  }
  friend ostream&
  operator<<(ostream& os, const Point& p) {
    return os << "x=" << p.x << " y=" << p.y
              << " z=" << p.z;
  }
};

class Vector { // Mathematical vector
public:
  int magnitude, direction;
  Vector(int m, int d) : magnitude(m), direction(d) {}
```

618

⊖ 最新信息查询请登录http://hillside.net/patterns。

⊜ 这是Bill Venner 取的名字，在其他地方有别的名称。

```
};

class Space {
public:
  static Point translate(Point p, Vector v) {
    // Copy-constructor prevents modifying the original.
    // A dummy calculation:
    p.x += v.magnitude + v.direction;
    p.y += v.magnitude + v.direction;
    p.z += v.magnitude + v.direction;
    return p;
  }
};

int main() {
  Point p1(1, 2, 3);
  Point p2 = Space::translate(p1, Vector(11, 47));
  cout << "p1: " << p1 << " p2: " << p2 << endl;
} ///:-
```

代码在这里做了简单化处理以防混乱。

既然信使的目标只是为了携带数据，可将这些数据安排为公有成员以便访问。然而，也有理由将这些数据设为私有成员。

10.3.2 收集参数

信使的大兄弟是收集参数（Collecting Parameter），它的工作就是从传递给它的函数中获取信息。通常，当收集参数被传递给多个函数的时候使用它，就像蜜蜂在采集花粉一样。

容器对于收集参数特别有用，因为它已经设置为动态增加对象：

```
//: C10:CollectingParameterDemo.cpp
#include <iostream>
#include <string>
#include <vector>
using namespace std;

class CollectingParameter : public vector<string> {};

class Filler {
public:
  void f(CollectingParameter& cp) {
    cp.push_back("accumulating");
  }
  void g(CollectingParameter& cp) {
    cp.push_back("items");
  }
  void h(CollectingParameter& cp) {
    cp.push_back("as we go");
  }
};

int main() {
  Filler filler;
  CollectingParameter cp;
  filler.f(cp);
  filler.g(cp);
  filler.h(cp);
  vector<string>::iterator it = cp.begin();
  while(it != cp.end())
    cout << *it++ << " ";
  cout << endl;
} ///:~
```

619

收集参数必须有一些方法用来设置值或者插入值。注意，根据这个定义信使可以被当作收集参数来使用。问题的关键是收集参数通过接收它的函数进行传递和修改。

10.4 单件

单件（Singleton）也许是“四人帮”给出的最简单的设计模式，它是允许一个类有且仅有一个实例的方法。下面的程序显示在C++中如何实现一个单件模式:

620

```
//: C10:SingletonPattern.cpp
#include <iostream>
using namespace std;

class Singleton {
  static Singleton s;
  int i;
  Singleton(int x) : i(x) { }
  Singleton& operator=(Singleton&);  // Disallowed
  Singleton(const Singleton&);       // Disallowed
public:
  static Singleton& instance() { return s; }
  int getValue() { return i; }
  void setValue(int x) { i = x; }
};

Singleton Singleton::s(47);

int main() {
  Singleton& s = Singleton::instance();
  cout << s.getValue() << endl;
  Singleton& s2 = Singleton::instance();
  s2.setValue(9);
  cout << s.getValue() << endl;
} ///:~
```

创建一个单件模式的关键是防止客户程序员获得任何控制其对象生存期的权利。为了做到这一点，声明所有的构造函数为私有，并且防止编译器隐式生成任何构造函数。注意，拷贝构造函数和赋值操作符（这两个方法都故意没有实现，因为它们根本就不会被调用）被声明为私有，以便防止任何这类复制的动作产生。

还必须决定如何去创建这个对象。在这里，它是被静态创建的，但也可以等待，直到客户程序员提出要求再根据要求进行创建。这种方式称作*惰性初始化*（lazy initialization），这种做法，只在创建对象的代价不大，并且并不总是需要它的情况下才有意义。

如果返回的是一个指针而不是引用，用户可能会不小心删除此指针，因此上述实现被认为是最安全的（析构函数也可以声明为私有或者保护的，以便缓和此问题）。在任何情况下，对象应该私有保存。

621

通过公有成员函数来提供对其对象的访问。在这里，**instance( )**产生**Singleton**对象的引用。其余的接口（**getValue( )**和**setValue( )**）是常见的类接口。

注意，这种方法并没有限制只创建一个对象。这种技术也支持创建有限个对象的对象池。然而在这种情况下，可能遇到池中共享对象的问题。如果这是一个问题，可以采取创建一个对共享对象进出对象池登记的方法来解决。

单件的变体

一个类中的任何**static**静态成员对象都表示一个单件：有且仅有一个对象被创建。因此，

从某种意义上讲，编程语言对单件技术提供了直接支持；我们自然是在常规基础上使用它。然而，对于**static**对象（类成员或者非类成员）来说有个问题：就是初始化的顺序的确定，如本书第1卷所述。如果一个静态对象依赖于另一个对象，那么将这些对象按正确的顺序进行初始化是很重要的。

在第1卷中，已经指出了如何在一个函数中定义一个静态对象来控制初始化顺序。这种方法延迟对象的初始化，直到在该函数第1次被调用时才进行初始化。如果该函数返回一个静态对象的引用，就可以达到单件的效果，这样就消除了可能由静态初始化引起的许多烦恼。例如，假如想在第1次调用某个函数时创建一个日志文件，该函数返回了那个日志文件的引用。下面这个头文件将完成这个任务：

```
//: C10:LogFile.h
#ifndef LOGFILE_H
#define LOGFILE_H
#include <fstream>
std::ofstream& logfile();
#endif // LOGFILE_H ///:~
```

函数的实现必须不是内联的（must not be inlined），因为那将意味着整个函数包括在其中定义的静态对象，在任何包含它的翻译单元中都将被复制，这就违犯了C++的一次定义（one-definition）规则[⊖]。这肯定阻碍试图控制初始化顺序的努力（但可能以微妙的、很难发现的形式出现）。因此函数的实现必须分开：

622

```
//: C10:LogFile.cpp {O}
#include "LogFile.h"
std::ofstream& logfile() {
  static std::ofstream log("Logfile.log");
  return log;
} ///:~
```

现在**log**对象不被初始化，直至函数**logfile( )**第1次调用时才被初始化。因此，如果创建一个函数：

```
//: C10:UseLog1.h
#ifndef USELOG1_H
#define USELOG1_H
void f();
#endif // USELOG1_H ///:~
```

在函数的实现中使用**logfile( )**：

```
//: C10:UseLog1.cpp {O}
#include "UseLog1.h"
#include "LogFile.h"
void f() {
  logfile() << __FILE__ << std::endl;
} ///:~
```

并且在另一个文件中再次使用**logfile( )**：

```
//: C10:UseLog2.cpp
//{L} LogFile UseLog1
#include "UseLog1.h"
#include "LogFile.h"
using namespace std;
```

⊖ C++ 标准要求："任何翻译单元都不得对任何变量、函数、类类型、枚举类型或模板等多次定义。在程序中使用的非内联函数或对象只能定义一次。"

```
void g() {
  logfile() << __FILE__ << endl;
}

int main() {
  f();
  g();
} ///:~
```

623

直至首次调用函数**f( )**时，对象**log**才被创建。

可以很容易地将在一个成员函数内部的静态对象的创建与单件类结合在一起。**SingletonPattern.cpp**可用这个方法做如下修改：[⊖]

```
//: C10:SingletonPattern2.cpp
// Meyers' Singleton.
#include <iostream>
using namespace std;

class Singleton {
  int i;
  Singleton(int x) : i(x) { }
  void operator=(Singleton&);
  Singleton(const Singleton&);
public:
  static Singleton& instance() {
    static Singleton s(47);
    return s;
  }
  int getValue() { return i; }
  void setValue(int x) { i = x; }
};

int main() {
  Singleton& s = Singleton::instance();
  cout << s.getValue() << endl;
  Singleton& s2 = Singleton::instance();
  s2.setValue(9);
  cout << s.getValue() << endl;
} ///:~
```

624

如果两个单件彼此依赖，就会产生一个特别有趣的情况，如下所示：

```
//: C10:FunctionStaticSingleton.cpp

class Singleton1 {
  Singleton1() {}
public:
  static Singleton1& ref() {
    static Singleton1 single;
    return single;
  }
};

class Singleton2 {
  Singleton1& s1;
  Singleton2(Singleton1& s) : s1(s) {}
public:
  static Singleton2& ref() {
    static Singleton2 single(Singleton1::ref());
```

⊖ 这被称为Meyers单件，以它的创建者Scott Meyers命名。

```
    return single;
  }
  Singleton1& f() { return s1; }
};

int main() {
  Singleton1& s1 = Singleton2::ref().f();
} ///:~
```

当调用**Singleton2::ref( )**时，它导致其惟一的**Singleton2**对象被创建。在这个对象的创建过程中，**Singleton1::ref( )**被调用，这导致其惟一的**Singleton1**对象被创建。因为这种技术不依赖连接或装载的顺序，因此程序员能够很好地控制初始化的全过程，而导致较少的错误。

单件的另外一种变体采用将一个对象的“单件角（Singleton-ness）”从其实现中分离出来的方法。使用第5章提到的“奇特的递归模板模式（Curiously Recurring Template Pattern）”来实现：

```
//: C10:CuriousSingleton.cpp
// Separates a class from its Singleton-ness (almost).
#include <iostream>
using namespace std;
template<class T> class Singleton {
  Singleton(const Singleton&);
  Singleton& operator=(const Singleton&);
protected:
  Singleton() {}
  virtual ~Singleton() {}
public:
  static T& instance() {
    static T theInstance;
    return theInstance;
  }
};

// A sample class to be made into a Singleton
class MyClass : public Singleton<MyClass> {
  int x;
protected:
  friend class Singleton<MyClass>;
  MyClass() { x = 0; }
public:
  void setValue(int n) { x = n; }
  int getValue() const { return x; }
};

int main() {
  MyClass& m = MyClass::instance();
  cout << m.getValue() << endl;
  m.setValue(1);
  cout << m.getValue() << endl;
} ///:~
```

625

MyClass通过下面3个步骤产生一个单件：

1) 声明其构造函数为私有或保护的。

2) 声明类**Singleton<MyClass>**为友元。

3) 从**Singleton<MyClass>**派生出**MyClass**。

在第3步中的自引用可能令人难以置信，然而正如第5章所述，因为这只是对模板**Singleton**中模板参数的静态依赖。换句话说，类**Singleton<MyClass>**的代码之所以能够被编译器实例化，是因为它不依赖于类**MyClass**的大小。只是在后来，当函数**Singleton<**

626

MyClass>:: instance( )第1次被调用时，才需要类**MyClass**的大小，而此时编译器已经知道类**MyClass**的大小。[⊖]

有趣的是，像单件这样简单的设计模式能有多么复杂，这里实际上还没有涉及线程安全的问题。最后说明一点，单件应该少用。真正的单件对象很少出现，而最终单件应该用于代替全局变量。[⊖]

10.5 命令：选择操作

命令（command）模式的结构很简单，但是对于消除代码间的耦合（decoupling）——清理代码——却有着重要的影响。

在《Advanced C++: Programming Styles And Idioms》（Addison Wesley，1992）一书中，Jim Coplien 创造了术语*函子*（functor），它表示一个对象，该对象的惟一目的是封装一个函数（由于“函子”在数学上有其特定的意义，这里将用更加明确的术语*函数对象*(function object)来代替它）。其特点就是消除被调用函数的选择与那个函数被调用的位置之间的联系。

GoF书中也提到这个术语，但是没有使用。然而，函数对象的话题却在那本书的很多模式中被反复论及。

从最直观的角度来看，命令模式就是一个函数对象：一个作为对象的函数。通过将函数封装为对象，就能够以参数的形式将其传递给其他函数或者对象，告诉它们在履行请求的过程中执行特定的操作。可以说，*命令模式*是携带行为信息的信使。

627

```
//: C10:CommandPattern.cpp
#include <iostream>
#include <vector>
using namespace std;

class Command {
public:
  virtual void execute() = 0;
};

class Hello : public Command {
public:
  void execute() { cout << "Hello "; }
};

class World : public Command {
public:
  void execute() { cout << "World! "; }
};

class IAm : public Command {
public:
  void execute() { cout << "I'm the command pattern!"; }
};

// An object that holds commands:
class Macro {
  vector<Command*> commands;
```

⊖ 在《Modern C++ Design》一书中，Andrei Alexandrescu提出了一种优越的基于策略的解决方案实现单件模式。

⊖ 参看Hyslop 和Sutter 发表在2003 年3月的《issue of CUJ》上的文章“Once is Not Enough”可以了解更详细的信息。

```
public:
  void add(Command* c) { commands.push_back(c); }
  void run() {
    vector<Command*>::iterator it = commands.begin();
    while(it != commands.end())
      (*it++)->execute();
  }
};

int main() {
  Macro macro;
  macro.add(new Hello);
  macro.add(new World);
  macro.add(new IAm);
  macro.run();
} ///:~
```

命令模式的主要特点是允许向一个函数或者对象传递一个想要的动作。上述例子提供了将一系列需要一起执行的动作集进行排队的方法。在这里，可以动态创建新的行为，某些事情通常只能通过编写新的代码来完成，而在上述例子中可以通过解释一个脚本来实现（如果需要实现的东西很复杂请参考解释器模式）。 628

GoF认为“命令模式是回调（callback）的面向对象的替代物”[⊖]，然而这里的单词“back”是回调概念的重要的一部分——回调返回到回调的创建者所在的位置。另一方面，对于一个命令对象来说，典型的做法仅仅是创建它并且将之传递给一些函数或者对象，而不是从始至终以其他方式联系命令对象。

命令模式的一个常见的例子就是在应用程序中“撤销（undo）操作”功能的实现。每次在用户进行某项操作的时候，相应的“撤销操作”命令对象就被置入一个队列中。而每个命令对象被执行后，程序的状态就倒退一步。

利用命令模式消除与事件处理的耦合

正如读者将在下一章中要看到的，采用并发（concurrency）技术的原因之一是为了更容易地掌握事件驱动编程（event-driven programming），在事件驱动方式的编程中，这些事件出现的地方是不可预料的。例如，当程序正在执行一个操作时，用户按下“退出”按钮并且希望程序能够快速响应。

使用并发的论据是它能够防止程序中代码段间的耦合。也就是说，如果运行一个独立的线程用以监视退出按钮，程序的“正常”操作无需知道有关退出按钮或者其他需要监视的操作。

然而，一旦读者理解耦合是一个问题，就可以用命令模式来避免它。每个“正常”的操作必须周期性地调用一个函数来检查事件的状态，而通过命令模式，这些“正常”操作不需要知道有关它们所检查的事件的任何信息，也就是说它们已经与事件处理代码分离开来。

```
//: C10:MulticastCommand.cpp {RunByHand}
// Decoupling event management with the Command pattern.
#include <iostream>
#include <vector>
#include <string>
#include <ctime>
#include <cstdlib>
using namespace std;

// Framework for running tasks:
```

629

⊖ 见原书第235页。

```
class Task {
public:
  virtual void operation() = 0;
};

class TaskRunner {
  static vector<Task*> tasks;
  TaskRunner() {} // Make it a Singleton
  TaskRunner& operator=(TaskRunner&); // Disallowed
  TaskRunner(const TaskRunner&); // Disallowed
  static TaskRunner tr;
public:
  static void add(Task& t) { tasks.push_back(&t); }
  static void run() {
    vector<Task*>::iterator it = tasks.begin();
    while(it != tasks.end())
      (*it++)->operation();
  }
};

TaskRunner TaskRunner::tr;
vector<Task*> TaskRunner::tasks;

class EventSimulator {
  clock_t creation;
  clock_t delay;
public:
  EventSimulator() : creation(clock()) {
    delay = CLOCKS_PER_SEC/4 * (rand() % 20 + 1);
    cout << "delay = " << delay << endl;
  }
  bool fired() {
    return clock() > creation + delay;
  }
};

// Something that can produce asynchronous events:
class Button {
  bool pressed;
  string id;
  EventSimulator e; // For demonstration
public:
  Button(string name) : pressed(false), id(name) {}
  void press() { pressed = true; }
  bool isPressed() {
    if(e.fired()) press(); // Simulate the event
    return pressed;
  }
  friend ostream&
  operator<<(ostream& os, const Button& b) {
    return os << b.id;
  }
};

// The Command object
class CheckButton : public Task {
  Button& button;
  bool handled;
public:
  CheckButton(Button & b) : button(b), handled(false) {}
  void operation() {
    if(button.isPressed() && !handled) {
      cout << button << " pressed" << endl;
```

630

```
      handled = true;
    }
  }
};

// The procedures that perform the main processing. These
// need to be occasionally "interrupted" in order to
// check the state of the buttons or other events:
void procedure1() {
  // Perform procedure1 operations here.
  // ...
  TaskRunner::run(); // Check all events
}

void procedure2() {
  // Perform procedure2 operations here.
  // ...
  TaskRunner::run(); // Check all events
}
void procedure3() {
  // Perform procedure3 operations here.
  // ...
  TaskRunner::run(); // Check all events
}

int main() {
  srand(time(0)); // Randomize
  Button b1("Button 1"), b2("Button 2"), b3("Button 3");
  CheckButton cb1(b1), cb2(b2), cb3(b3);
  TaskRunner::add(cb1);
  TaskRunner::add(cb2);
  TaskRunner::add(cb3);
  cout << "Control-C to exit" << endl;
  while(true) {
    procedure1();
    procedure2();
    procedure3();
  }
} ///:~
```

631

在这里，命令对象由被单件**TaskRunner**执行的**Task**表示。**EventSimulator**创建一个随机延迟时间，所以当周期性的调用函数**fired( )**时，在某个随机时间段，其返回结果从**true**到**false**变化。**EventSimulator**对象在类**Button**中使用，模拟在某个不可预知的时间段用户事件发生的动作。**CheckButton**是**Task**的实现，在程序中通过所有“正常”代码对其进行周期性的检查——可以看到这些检查发生在函数**Procedure1( )**、**Procedure2( )**和**Procedure3( )**的末尾。

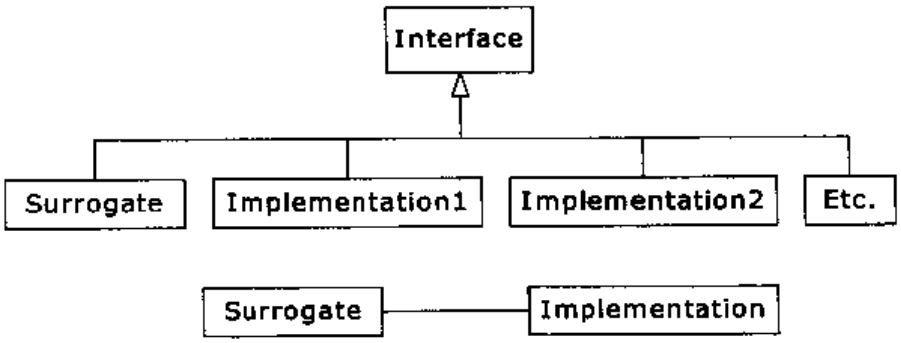
尽管这需要颇费点脑筋来设立命令对象，但是读者将在第11章中看到，如果采用线程处理方法则需要更多的考虑，小心预防并行编程中与生俱来的各种的困难问题，所以这种较简便的解决方法更可取。将**TaskRunner::run( )**调用植入一个多线程处理的“计时器”对象中，也可以创建一个很简单的线程处理方案。这样做，可以消除所有“正常操作”（上述例子中的过程）与事件代码间的耦合。

10.6 消除对象耦合

代理（Proxy）模式和状态（State）模式都提供一个代理（Surrogate）类。代码与代理类打交道，而做实际工作的类隐藏在代理类背后。当调用代理类中的一个函数时，代理类仅转而 632

去调用实现类中相应的函数。这两种模式是如此相似，从结构上看，可以认为代理模式只是状态模式的一个特例。设想将这两者合理地混合在一起组成一个称为代理（Surrogate）设计模式，这肯定是一个很具有诱惑力的想法，但是这两个模式的内涵（intent）是不一样的。这样做很容易陷入“如果结构相同模式就相同”的思想误区。必须始终关注模式的内涵，从而明确它的功能到底是什么。

基本思想很简单：代理（Surrogate）类派生自一个基类，由平行地派生自同一个基类的一个或多个类提供实际的实现：



当一个代理对象被创建的时候，一个实现对象就分配给了它，代理对象就将函数调用发给实现对象。

从结构上来看，代理模式和状态模式的区别很简单：代理模式只有一个实现类，而状态模式有多个（一个以上）实现。（在GoF中）认为这两种设计模式的应用也不同：代理模式控制对其实现类的访问，而状态模式动态地改变其实现类。然而，如果广义理解“控制对实现类的访问”，则这两个模式似乎是一个连续体的两部分。

10.6.1 代理模式：作为其他对象的前端

如果按照上面的图结构实现代理模式，其实现代码如下：

```
//: C10:ProxyDemo.cpp
// Simple demonstration of the Proxy pattern.
#include <iostream>
using namespace std;
class ProxyBase {
public:
  virtual void f() = 0;
  virtual void g() = 0;
  virtual void h() = 0;
  virtual ~ProxyBase() {}
};

class Implementation : public ProxyBase {
public:
  void f() { cout << "Implementation.f()" << endl; }
  void g() { cout << "Implementation.g()" << endl; }
  void h() { cout << "Implementation.h()" << endl; }
};

class Proxy : public ProxyBase {
  ProxyBase* implementation;
public:
  Proxy() { implementation = new Implementation(); }
  ~Proxy() { delete implementation; }
  // Forward calls to the implementation:
  void f() { implementation->f(); }
```

633

```
  void g() { implementation->g(); }
  void h() { implementation->h(); }
};

int main()  {
  Proxy p;
  p.f();
  p.g();
  p.h();
} ///:~
```

在某些情况下，类**Implementation**并不需要与类**Proxy**具有相同的接口——**Proxy**类可以任意“订购”（关联）**Implementation**类并且将函数调用提交给它，这就符合了代理的基本思想（值得注意的是，这种描述和GoF关于代理的定义不一致）。然而，使用共同的接口可以将代理的替代物插入到客户代码中——编写客户代码只用来与原对象进行通信，不需对其进行修改以接受代理（这大概是使用代理的关键问题）。此外，通过共同的接口，**Implementation**被迫实现**Proxy**需要调用的所有函数。

代理模式与状态模式之间的不同之处在于它们所解决的问题不同。GoF中给出了代理模式的一般用途，描述如下：

634

1) **远程代理**（**Remote proxy**）。为不同地址空间的对象提供代理。通过某些远程对象技术实现。

2) **虚拟代理**（**Virtual proxy**）。根据需要提供一种“惰性初始化”方式来创建高代价的对象。

3) **保护代理**（**Protection proxy**）。当不愿意客户程序员拥有被代理对象的全部访问权限时，使用保护代理。

4) **巧妙引用**（**Smart reference**）。当访问被代理的对象时，增加额外的活动。引用计数（reference counting）就是一个例子：它用来跟踪被代理的某个特定对象被引用的次数，以实现写入时复制（copy-on-write）并且防止对象起别名。[⊖]一个更简单的例子就是对特定函数的调用进行计数。

10.6.2 状态模式：改变对象的行为

状态模式产生一个可以改变其类的对象，当发现在大多数或者所有函数中都存在有条件的代码时，这种模式很有用。和代理模式一样，状态模式通过一个前端对象来使用后端实现对象履行其职责。然而，在前端对象生存期期间，状态模式从一个实现对象到另一个实现对象进行切换，以实现对于相同的函数调用产生不同的行为。如果在决定函数该做什么之前在每个函数内部做很多测试，那么这种方法是对实现代码的一种很好的改进。举个例子，在青蛙王子童话中，青蛙王子依照其所处的状态而有不同的行为。现在可以通过测试一个**bool**变量来实现：

```
//: C10:KissingPrincess.cpp
#include <iostream>
using namespace std;

class Creature {
  bool isFrog;
 public:
   Creature() : isFrog(true) {}
   void greet() {
     if(isFrog)
```

635

⊖ 参阅《C++编程思想》第1卷以获得关于引用计数更详细的知识。

```
      cout << "Ribbet!" << endl;
    else
      cout << "Darling!" << endl;
  }
  void kiss() { isFrog = false; }
};

int main() {
  Creature creature;
  creature.greet();
  creature.kiss();
  creature.greet();
} ///:~
```

然而，**greet( )**等任何其他所有函数在执行操作前都必须测试变量**isFrog**，这样就使代码变得笨拙至极，特别是在系统中加入额外的状态时情况会更加严重。通过将操作委派给状态对象，这种情况就可以改变，代码从而得到了简化。

```
//: C10:KissingPrincess2.cpp
// The State pattern.
#include <iostream>
#include <string>
using namespace std;

class Creature {
  class State {
  public:
    virtual string response() = 0;
  };
  class Frog : public State {
  public:
    string response() { return "Ribbet!"; }
  };
  class Prince : public State {
  public:
    string response() { return "Darling!"; }
  };
  State* state;
public:
  Creature() : state(new Frog()) {}
  void greet() {
    cout << state->response() << endl;
  }
  void kiss() {
    delete state;
    state = new Prince();
  }
};

int main() {
  Creature creature;
  creature.greet();
  creature.kiss();
  creature.greet();
} ///:~
```

636

在这里，将实现类设计为嵌套或者私有并不是必需的，但是如果能做到的话，就会创建出更加清晰的代码。

注意，对状态类的改变将会自动地在所有的代码中进行传播，而不需要编辑这些类来完成改变。

10.7 适配器模式

适配器（Adapter）模式接受一种类型并且提供一个对其他类型的接口。当给定一个库或者具有某一接口的一段代码，同时还给定另外一个库或者与前面那段代码的基本思想相同的一段代码而只是表达方式不一致时，适配器模式将十分有用。通过调整彼此的表达方式以适配彼此，将会迅速产生解决方法。

假设有个产生斐波那契数列的发生器类，如下所示：

```
//: C10:FibonacciGenerator.h
#ifndef FIBONACCIGENERATOR_H
#define FIBONACCIGENERATOR_H

class FibonacciGenerator {
  int n;
  int val[2];
public:
  FibonacciGenerator() : n(0) { val[0] = val[1] = 0; }
  int operator()() {
    int result = n > 2 ? val[0] + val[1] : n > 0 ? 1 : 0;
    ++n;
    val[0] = val[1];
    val[1] = result;
    return result;
  }
  int count() { return n; }
};
#endif // FIBONACCIGENERATOR_H ///:~
```

637

由于它是一个发生器，可以调用**operator( )**来使用它，如下所示：

```
//: C10:FibonacciGeneratorTest.cpp
#include <iostream>
#include "FibonacciGenerator.h"
using namespace std;

int main() {
  FibonacciGenerator f;
  for(int i =0; i < 20; i++)
    cout << f.count() << ": " << f() << endl;
} ///:~
```

也许读者希望利用这个发生器来执行STL数值算法操作。遗憾的是，STL算法只能使用迭代器才能工作，这就存在接口不匹配的问题。解决方法就是创建一个适配器，它将接受**FibonacciGenerator**并产生一个供STL算法使用的迭代器。由于数值算法只要求一个输入迭代器，该适配器模式相当地直观（为了某种目的，它产生了一个STL迭代器，如下所示）：

```
//: C10:FibonacciAdapter.cpp
// Adapting an interface to something you already have.
#include <iostream>
#include <numeric>
#include "FibonacciGenerator.h"
#include "../C06/PrintSequence.h"
using namespace std;

class FibonacciAdapter { // Produce an iterator
  FibonacciGenerator f;
  int length;
public:
```

638

```
  FibonacciAdapter(int size) : length(size) {}
  class iterator;
  friend class iterator;
  class iterator : public std::iterator<
    std::input_iterator_tag, FibonacciAdapter, ptrdiff_t> {
    FibonacciAdapter& ap;
  public:
    typedef int value_type;
    iterator(FibonacciAdapter& a) : ap(a) {}
    bool operator==(const iterator&) const {
      return ap.f.count() == ap.length;
    }
    bool operator!=(const iterator& x) const {
      return !(*this == x);
    }
    int operator*() const { return ap.f(); }
    iterator& operator++() { return *this; }
    iterator operator++(int) { return *this; }
  };
  iterator begin() { return iterator(*this); }
  iterator end() { return iterator(*this); }
};

int main() {
  const int SZ = 20;
  FibonacciAdapter a1(SZ);
  cout << "accumulate: "
    << accumulate(a1.begin(), a1.end(), 0) << endl;
  FibonacciAdapter a2(SZ), a3(SZ);
  cout << "inner product: "
    << inner_product(a2.begin(), a2.end(), a3.begin(), 0)
    << endl;
  FibonacciAdapter a4(SZ);
  int r1[SZ] = {0};
  int* end = partial_sum(a4.begin(), a4.end(), r1);
  print(r1, end, "partial_sum", " ");
  FibonacciAdapter a5(SZ);
  int r2[SZ] = {0};
  end = adjacent_difference(a5.begin(), a5.end(), r2);
  print(r2, end, "adjacent_difference", " ");
} ///:~
```

通过被告知斐波那契数列的长度来初始化**FibonacciAdapter**。当创建**iterator**时，它仅获得一个包含**FibonacciAdapter**的引用，这样它就能够访问**FibonacciGenerator**和**length**。注意，相等比较忽略了右边的值，因为惟一重要的问题是判断发生器是否达到其长度。此外，**operator++( )**没有修改迭代器；改变**FibonacciAdapter**状态的惟一操作是调用发生器**FibonacciGenerator**中的函数**operator( )**。我们在迭代器的这个极其简单的版本上是侥幸成功的，因为对输入迭代器的约束条件十分严格；特别是，在该序列中每个值只能读取一次。

639

在函数**main( )**中，可以看到所有4类不同的数值算法同**FibonacciAdapter**一起成功地通过了测试。

10.8 模板方法模式

应用程序结构框架允许从一个或一组类中继承以便创建一个新的应用程序，重用现存类中几乎所有的代码，并且覆盖其中一个或多个函数以便自定义所需要的应用程序。应用程序结构框架中的一个基本的概念是*模板方法*（Template Method）模式，它很典型地隐藏在覆盖的下

方，通过调用基类的不同函数（这里覆盖了其中一些函数以创建应用程序）来驱动程序运行。

模板方法模式的一个重要特征是它的定义在基类中（有时作为一个私有成员函数）并且不能改动——模板方法模式就是“坚持相同的代码”。它调用其他基类函数（就是那些被覆盖的函数）以便完成其工作，但是客户程序员不必直接调用这些函数，如下所示：

```
//: C10:TemplateMethod.cpp
// Simple demonstration of Template Method.
#include <iostream>
using namespace std;

class ApplicationFramework {
protected:
  virtual void customize1() = 0;
  virtual void customize2() = 0;
public:
  void templateMethod() {
    for(int i = 0; i < 5; i++) {
      customize1();
      customize2();
    }
  }
};

// Create a new "application":
class MyApp : public ApplicationFramework {
protected:
  void customize1() { cout << "Hello "; }
  void customize2() { cout << "World!" << endl; }
};

int main() {
  MyApp app;
  app.templateMethod();
} ///:~
```

640

驱动应用程序运行的“引擎”是模板方法模式。在GUI（图形用户界面）应用程序中，这个“引擎”就是主要的事件环。客户程序员只需提供**customize1( )**和**customize2( )**的定义，便可以令“应用程序”运行。

10.9 策略模式：运行时选择算法

注意，模板方法模式是“坚持相同的代码”，而被覆盖的函数是“变化的代码”。然而，这种变化在编译时通过继承被固定下来。按照“组合优于继承”的格言，可以利用组合来解决将变化的代码从“坚持相同的代码”中分开的问题，从而产生策略（Strategy）模式。这种方法有一个明显的好处：在程序运行时，可以插入变化的代码。策略模式也加入了“语境”，它可以是一个代理类，这个类控制着对特定策略对象的选择和使用——就像状态模式一样。

“策略”的意思就是：可以使用多种方法来解决某个问题——即“条条大路通罗马”。现在考虑一下忘记了某个人姓名时的情景。这里的程序可以用不同方法解决这个问题：

641

```
//: C10:Strategy.cpp
// The Strategy design pattern.
#include <iostream>
using namespace std;

class NameStrategy {
public:
```

```
  virtual void greet() = 0;
};

class SayHi : public NameStrategy {
public:
  void greet() {
    cout << "Hi! How's it going?" << endl;
  }
};

class Ignore : public NameStrategy {
public:
  void greet() {
    cout << "(Pretend I don't see you)" << endl;
  }
};

class Admission : public NameStrategy {
public:
  void greet() {
    cout << "I'm sorry. I forgot your name." << endl;
  }
};

// The "Context" controls the strategy:
class Context {
  NameStrategy& strategy;
public:
  Context(NameStrategy& strat) : strategy(strat) {}
  void greet() { strategy.greet(); }
};

int main() {
  SayHi sayhi;
  Ignore ignore;
  Admission admission;
  Context c1(sayhi), c2(ignore), c3(admission);
  c1.greet();
  c2.greet();
  c3.greet();
} ///:~
```

642

Context::greet( )可以正规地写得更复杂些；它类似模板方法模式，因为其中包含了不能改变的代码。但在函数**main( )**中可以看到，可以在运行时就策略进行选择。更进一步的做法，可以将状态模式与在**Context**对象的生存期期间变化的策略模式结合起来使用。

10.10 职责链模式：尝试采用一系列策略模式

职责链（Chain of Responsibility）模式也许被看做一个使用策略对象的“递归的动态一般化”。此时提出一个调用，在一个链序列中的每个策略都试图满足这个调用。这个过程直到有一个策略成功满足该调用或者到达链序列的末尾才结束。在递归方法中，有个函数反复调用其自身直至达到某个终止条件；在职责链中，一个函数调用自身，（通过遍历策略链）调用函数的一个不同实现，如此反复直至达到某个终止条件。这个终止条件或者是已到达策略链的底部（这样就会返回一个默认对象；不管能否提供这个默认结果，必须有个方法能够决定该职责链搜索是成功还是失败）或者是成功找到一个策略。

除了调用一个函数来满足某个请求以外，链中的多个函数都有此机会满足某个请求，因此它有点专家系统的意味。由于职责链实际上就是一个链表，它能够动态创建，因此它可以看做

是一个更一般的动态构建的**switch**语句。

在GoF中，有很多关于如何将职责链模式创建为一个链表的讨论。然而，如果审视这类模式，实在没必要考虑链是如何创建的；这是一个实现的细节。由于GoF是在大多数C++编译器中STL容器可利用之前编写的，它讨论创建链表最可能的原因有：（1）编译器中没有内置的链表，因此必须自己创建；（2）数据结构常常是作为学术界的一门基本的技术进行教授，GoF的作者们还没有数据结构应该是编程语言提供的有效标准工具的概念。讨论如何使用容器来实现职责链作为链的细节（在GoF中，它就是一个链表）对于这里的问题的解决没有什么意义，可以很方便地用STL容器来实现，如下所示。 643

在这里可以看到，使用一种自动递归搜索链中每个策略的机制，职责链模式自动找到一个解决方法：

```
//: C10:ChainOfReponsibility.cpp
// The approach of the five-year-old.
#include <iostream>
#include <vector>
#include "../purge.h"
using namespace std;

enum Answer { NO, YES };

class GimmeStrategy {
public:
  virtual Answer canIHave() = 0;
  virtual ~GimmeStrategy() {}
};

class AskMom : public GimmeStrategy {
public:
  Answer canIHave() {
    cout << "Mooom? Can I have this?" << endl;
    return NO;
  }
};

class AskDad : public GimmeStrategy {
public:
  Answer canIHave() {
    cout << "Dad, I really need this!" << endl;
    return NO;
  }
};

class AskGrandpa : public GimmeStrategy {
public:
  Answer canIHave() {
    cout << "Grandpa, is it my birthday yet?" << endl;
    return NO;
  }
};

class AskGrandma : public GimmeStrategy {
public:
  Answer canIHave() {
    cout << "Grandma, I really love you!" << endl;
    return YES;
  }
};
```

644

```
class Gimme : public GimmeStrategy {
  vector<GimmeStrategy*> chain;
public:
  Gimme() {
    chain.push_back(new AskMom());
    chain.push_back(new AskDad());
    chain.push_back(new AskGrandpa());
    chain.push_back(new AskGrandma());
  }
  Answer canIHave() {
    vector<GimmeStrategy*>::iterator it = chain.begin();
    while(it != chain.end())
      if((*it++)->canIHave() == YES)
        return YES;
    // Reached end without success...
    cout << "Whiiiiinnne!" << endl;
    return NO;
  }
  ~Gimme() { purge(chain); }
};

int main() {
  Gimme chain;
  chain.canIHave();
} ///:~
```

注意，“语境”类**Gimme**和所有策略类都派生自同一个基类**GimmeStrategy**。

如果读者研读GoF中关于职责链的那部分内容，将会发现其结构与上面介绍的内容有明显不一致地方，因为他们专注于创建自己的链表。然而，如果牢记职责链的本质是尝试多个解决方法直到找到一个起作用的方法，读者就会了解按顺序排好的实现机制并不是该模式的本质所在。

645

10.11 工厂模式：封装对象的创建

当发现需要添加新的类型到一个系统中时，最明智的首要步骤就是用多态机制为这些新类型创建一个共同的接口。用这种方法可以将系统中其余的代码与新添加的特定类型的代码分开。新类型的添加并不会扰乱已存在的代码…或者至少看上去如此。起初它似乎只需要在继承新类的地方修改代码，但这并非完全正确。仍须创建新类型的对象，在创建对象的地方必须指定要使用的准确的构造函数。因此，如果创建对象的代码遍布整个应用程序，在增加新类型时将会遇到同样的问题——仍然必须找出代码中所有与新类型相关的地方。这是由类的创建而不是类的使用（类型的使用问题已被多态机制解决了）而引起，但是效果是一样的：添加新类型将导致问题的出现。

这个问题的解决方法就是强制用一个通用的工厂（factory）来创建对象，而不允许将创建对象的代码散布于整个系统。如果程序中所有需要创建对象的代码都转到这个工厂执行，那么在增加新对象时所要做的全部工作就是只需修改工厂。这种设计是众所周知的工厂方法（Factory Method）模式的一种变体。由于每个面向对象应用程序都需要创建对象，并且由于人们可能通过添加新类型来扩展应用程序，工厂模式可能是所有设计模式中最有用的模式之一。

举一个例子，考虑常用的**Shape**例子。实现工厂模式的一种方法就是在基类中定义一个静态成员函数：

```
//: C10:ShapeFactory1.cpp
#include <iostream>
#include <stdexcept>
#include <cstddef>
```

```
#include <string>
#include <vector>
#include "../purge.h"
using namespace std;

class Shape {
public:
  virtual void draw() = 0;
  virtual void erase() = 0;
  virtual ~Shape() {}
  class BadShapeCreation : public logic_error {
  public:
    BadShapeCreation(string type)
    : logic_error("Cannot create type " + type) {}
  };
  static Shape* factory(const string& type)
    throw(BadShapeCreation);
};

class Circle : public Shape {
  Circle() {} // Private constructor
  friend class Shape;
public:
  void draw() { cout << "Circle::draw" << endl; }
  void erase() { cout << "Circle::erase" << endl; }
  ~Circle() { cout << "Circle::~Circle" << endl; }
};

class Square : public Shape {
  Square() {}
  friend class Shape;
public:
  void draw() { cout << "Square::draw" << endl; }
  void erase() { cout << "Square::erase" << endl; }
  ~Square() { cout << "Square::~Square" << endl; }
};

Shape* Shape::factory(const string& type)
  throw(Shape::BadShapeCreation) {
  if(type == "Circle") return new Circle;
  if(type == "Square") return new Square;
  throw BadShapeCreation(type);
}

char* sl[] = { "Circle", "Square", "Square",
  "Circle", "Circle", "Circle", "Square" };
int main() {
  vector<Shape*> shapes;
  try {
    for(size_t i = 0; i < sizeof sl / sizeof sl[0]; i++)
      shapes.push_back(Shape::factory(sl[i]));
  } catch(Shape::BadShapeCreation e) {
    cout << e.what() << endl;
    purge(shapes);
    return EXIT_FAILURE;
  }
  for(size_t i = 0; i < shapes.size(); i++) {
    shapes[i]->draw();
    shapes[i]->erase();
  }
  purge(shapes);
} ///:~
```

646

647

函数**factory( )**允许以一个参数来决定创建何种类型的**Shape**。在这里，参数类型为**string**，也可以是任何数据集。在添加新的**Shape**类型时，函数**factory( )**是当前系统中惟一需要修改的代码。（对象的初始化数据大概也可以由系统外获得，而不必像本例中那样来自硬编码数组。）

为了确保对象的创建只能发生在函数**factory( )**中，**Shape**的特定类型的构造函数被设为私有，同时**Shape**被声明为友元类，因此**factory( )**能够访问这些构造函数。（也可以只将**Shape::factory( )**声明为友元函数，但是似乎声明整个基类为友元类也没什么大碍。）这样的设计还有另外一个重要的含义——基类**Shape**现在必须了解每个派生类的细节——这是面向对象设计试图避免的一个性质。对于结构框架或者任何类库来说都应该支持扩充，但这样一来，系统很快就会变得笨拙，因为一旦新类型被加到这种层次结构中，基类就必须更新。可以使用下一小节将要讨论的多态工厂（polymorphic factory）来避免这种循环依赖。

10.11.1 多态工厂

在前面的例子中，静态成员函数**static factory( )**迫使所有创建对象的操作都集中在一个地方，因此这个地方就是惟一需要修改代码的地方。这确实是一个合理的解决方法，因为它完美地封装了对象的创建过程。然而，“四人帮”强调工厂方法模式的理由是，可以使不同类型的工厂派生自基本类型的工厂。工厂方法模式事实上是多态工厂模式的一个特例。这里修改了**ShapeFactory1.cpp**，所以工厂方法模式作为一个单独的类中的虚函数出现：

648

```
//: C10:ShapeFactory2.cpp
// Polymorphic Factory Methods.
#include <iostream>
#include <map>
#include <string>
#include <vector>
#include <stdexcept>
#include <cstddef>
#include "../purge.h"
using namespace std;

class Shape {
public:
  virtual void draw() = 0;
  virtual void erase() = 0;
  virtual ~Shape() {}
};

class ShapeFactory {
  virtual Shape* create() = 0;
  static map<string, ShapeFactory*> factories;
public:
  virtual ~ShapeFactory() {}
  friend class ShapeFactoryInitializer;
  class BadShapeCreation : public logic_error {
  public:
    BadShapeCreation(string type)
    : logic_error("Cannot create type " + type) {}
  };
  static Shape*
  createShape(const string& id) throw(BadShapeCreation) {
    if(factories.find(id) != factories.end())
      return factories[id]->create();
    else
      throw BadShapeCreation(id);
```

```
  }
};
// Define the static object:
map<string, ShapeFactory*> ShapeFactory::factories;

class Circle : public Shape {
  Circle() {} // Private constructor
  friend class ShapeFactoryInitializer;
  class Factory;
  friend class Factory;
  class Factory : public ShapeFactory {
  public:
    Shape* create() { return new Circle; }
    friend class ShapeFactoryInitializer;
  };
public:
  void draw() { cout << "Circle::draw" << endl; }
  void erase() { cout << "Circle::erase" << endl; }
  ~Circle() { cout << "Circle::~Circle" << endl; }
};

class Square : public Shape {
  Square() {}
  friend class ShapeFactoryInitializer;
  class Factory;
  friend class Factory;
  class Factory : public ShapeFactory {
  public:
    Shape* create() { return new Square; }
    friend class ShapeFactoryInitializer;
  };
public:
  void draw() { cout << "Square::draw" << endl; }
  void erase() { cout << "Square::erase" << endl; }
  ~Square() { cout << "Square::~Square" << endl; }
};

// Singleton to initialize the ShapeFactory:
class ShapeFactoryInitializer {
  static ShapeFactoryInitializer si;
  ShapeFactoryInitializer() {
    ShapeFactory::factories["Circle"]= new Circle::Factory;
    ShapeFactory::factories["Square"]= new Square::Factory;
  }
  ~ShapeFactoryInitializer() {
    map<string, ShapeFactory*>::iterator it =
      ShapeFactory::factories.begin();
    while(it != ShapeFactory::factories.end())
      delete it++->second;
  }
};

// Static member definition:
ShapeFactoryInitializer ShapeFactoryInitializer::si;

char* sl[] = { "Circle", "Square", "Square",
  "Circle", "Circle", "Circle", "Square" };

int main() {
  vector<Shape*> shapes;
  try {
    for(size_t i = 0; i < sizeof sl / sizeof sl[0]; i++)
```

649

650

```
      shapes.push_back(ShapeFactory::createShape(sl[i]));
    } catch(ShapeFactory::BadShapeCreation e) {
      cout << e.what() << endl;
      return EXIT_FAILURE;
    }
    for(size_t i = 0; i < shapes.size(); i++) {
      shapes[i]->draw();
      shapes[i]->erase();
    }
    purge(shapes);
  } ///:~
```

现在，工厂方法模式作为**virtual create( )**出现在它自己的**ShapeFactory**类中。这是一个私有成员函数，意味着不能直接调用它，但可以被覆盖。**Shape**的子类必须创建各自的**ShapeFactory**子类，并且覆盖成员函数**create( )**以创建其自身类型的对象。这些工厂是私有的，只能被主工厂方法模式访问。采用这种方法，所有客户代码都必须通过工厂方法模式创建对象。

Shape对象的实际创建是通过调用**ShapeFactory::createShape( )**完成的，这是一个静态成员函数，使用**ShapeFactory**中的**map**根据传递给它的标识符找到相应的工厂对象。工厂直接创建**Shape**对象，但是可以设想一个更为复杂的问题：在某个地方返回一个合适的工厂对象，然后该工厂对象被调用者用于以更复杂的方法创建一个对象。然而，似乎在大多数情况下不需要这么复杂地使用多态工厂方法模式，基类中的一个静态成员函数（正如**ShapeFactory1.cpp**中所示）就能很好地完成这项工作。

651

注意，**ShapeFactory**必须通过装载它的**map**与工厂对象进行初始化，这些操作发生在单件**ShapeFactoryInitializer**中。当增加一个新类型到这个设计时，必须定义该类型，创建一个工厂并修改**ShapeFactoryInitializer**，以便将工厂的一个实例插入到**map**中。这些额外的复杂操作再次暗示，如果不需要创建独立的工厂对象，尽可能使用静态（**static**）工厂方法模式。

10.11.2 抽象工厂

抽象工厂（Abstract Factory）模式看起来和前面看到的工厂方法很相似，只是它使用若干工厂方法（Factory Method）模式。每个工厂方法模式创建一个不同类型的对象。当创建一个工厂对象时，要决定将如何使用由那个工厂创建的所有对象。“四人帮”书中的例子实现各种图形用户界面（GUI）的可移植性：创建一个适合于正在使用的GUI的工厂对象，然后它将根据对它发出的对一个菜单、按钮或者滚动条等的请求自动创建适合该GUI的项目版本。这样就能够在一个地方隔离从一个GUI转变到另一个GUI的作用。

再举一个例子，假设要创建一个通用的游戏环境，并且希望它能支持不同类型的游戏。请看以下程序是如何使用抽象工厂模式的：

```
//: C10:AbstractFactory.cpp
// A gaming environment.
#include <iostream>
using namespace std;

class Obstacle {
public:
  virtual void action() = 0;
};

class Player {
public:
  virtual void interactWith(Obstacle*) = 0;
```

652

```
};

class Kitty: public Player {
  virtual void interactWith(Obstacle* ob) {
    cout << "Kitty has encountered a ";
    ob->action();
  }
};

class KungFuGuy: public Player {
  virtual void interactWith(Obstacle* ob) {
    cout << "KungFuGuy now battles against a ";
    ob->action();
  }
};

class Puzzle: public Obstacle {
public:
  void action() { cout << "Puzzle" << endl; }
};

class NastyWeapon: public Obstacle {
public:
  void action() { cout << "NastyWeapon" << endl; }
};

// The abstract factory:
class GameElementFactory {
public:
  virtual Player* makePlayer() = 0;
  virtual Obstacle* makeObstacle() = 0;
};

// Concrete factories:
class KittiesAndPuzzles : public GameElementFactory {
public:
  virtual Player* makePlayer() { return new Kitty; }
  virtual Obstacle* makeObstacle() { return new Puzzle; }
};

class KillAndDismember : public GameElementFactory {
public:
  virtual Player* makePlayer() { return new KungFuGuy; }
  virtual Obstacle* makeObstacle() {
    return new NastyWeapon;
  }
};

class GameEnvironment {
  GameElementFactory* gef;
  Player* p;
  Obstacle* ob;
public:
  GameEnvironment(GameElementFactory* factory)
  : gef(factory), p(factory->makePlayer()),
    ob(factory->makeObstacle()) {}
  void play() { p->interactWith(ob); }
  ~GameEnvironment() {
    delete p;
    delete ob;
    delete gef;
  }
};
```

653

```
int main() {
  GameEnvironment
    g1(new KittiesAndPuzzles),
    g2(new KillAndDismember);
  g1.play();
  g2.play();
}
/* Output:
Kitty has encountered a Puzzle
KungFuGuy now battles against a NastyWeapon */ ///:~
```

在此环境中，**Player**对象与**Obstacle**对象交互，但是**Player**和**Obstacle**类型依赖于具体的游戏。可以选择特定的**GameElementFactory**来决定游戏的类型，然后**GameEnvironment**控制游戏的设置和进行。在本例中，游戏的设置和进行很简单，但是那些动作（*初始条件*（initial condition）和*状态变化*（state change））在很大程度上决定了游戏的结果。在这里，**GameEnvironment**不是设计成继承的，即使这样做可能是有意义的。

这个例子也说明将在稍后讨论*双重派遣*（double dispatching）。

654 ### 10.11.3 虚构造函数

使用工厂方法模式的主要目标之一就是更好地组织代码，使得在创建对象时不需要选择准确的构造函数类型。也就是说，可以告诉工厂："现在还不能确切地知道需要什么类型的对象，但是这里有一些信息。请创建类型适当的对象"。

此外，在构造函数调用期间，虚拟机制并不起作用（发生早期绑定）。在某些情况下这是很棘手的事情。例如，在**Shape**程序中，在**Shape**对象的构造函数内部建立一切需要的东西然后由**draw( )**绘制**Shape**，这似乎是合理的。函数**draw( )**应该是一个虚函数，它将根据传递给**shape**的消息绘制相应的图形，消息表明图形本身是**Circle**、**Square**或者**Line**。然而，这些操作在构造函数内部不能采用这种方法，因为当在构造函数内部调用虚函数时，将由虚函数决定指向哪个"局部的"函数体。

如果想要在构造函数中调用虚函数，并使其完成正确的工作，必须使用某种技术来*模拟*虚构造函数。这是一个难题。请记住，虚函数的思想就是发送一个消息给对象，而让对象确定要做的正确事情。但是对象是由构造函数创建的。因此，虚构造函数好像是在对一个对象说："我不能准确知道你是什么类型的对象，但是无论如何要以正确的类型建造你。"对于普通的构造函数来说，编译器在编译时必须知道虚指针（VPTR）指向的虚函数表（VTABLE）的地址；而对于虚构造函数，即使存在这样的虚函数表，它也不可能做到这一点，因为它在编译时不知道任何类型信息。构造函数不能为虚函数是有道理的，因为它是这样一种函数，必须完全知道有关对象类型的所有信息。

可是，程序员有时还想要得到接近于虚构造函数的行为。

在**Shape**的例子中，在参数表中对**Shape**构造函数提交一些特定的信息，使构造函数创建特定类型的**Shape**对象（一个**Circle** 或是一个**Square**）而无需更多的干涉，这将是很好的。通常，程序员自己必需显式调用**Circle** 或是**Square**的构造函数。

655 Coplien[⊖]将他给出的解决此问题的方法取名为"信封和信件类"。"信封"类是基类，它是一个包含指向一个对象的指针的外壳，该对象也是一个基类类型。"信封"类的构造函数决定采用什么样的特定类型，在堆上创建一个该类型的对象，然后对它的指针分配对象（决定是在

⊖ James O. Coplien，《Advanced C++ Programming Styles and Idioms》，Addison Wesley，1992。

运行中调用构造函数时做出的，而不是在编译中做类型正常检查时做出的)。随后的所有函数调用都是由基类通过它的指针来进行处理。这实际上就是状态模式的小小变形，其中基类扮演派生类的代理的角色，而派生类提供行为中的变化:

```
//: C10:VirtualConstructor.cpp
#include <iostream>
#include <string>
#include <stdexcept>
#include <stdexcept>
#include <cstddef>
#include <vector>
#include "../purge.h"
using namespace std;

class Shape {
  Shape* s;
  // Prevent copy-construction & operator=
  Shape(Shape&);
  Shape operator=(Shape&);
protected:
  Shape() { s = 0; }
public:
  virtual void draw() { s->draw(); }
  virtual void erase() { s->erase(); }
  virtual void test() { s->test(); }
  virtual ~Shape() {
    cout << "~Shape" << endl;
    if(s) {
      cout << "Making virtual call: ";
      s->erase(); // Virtual call
    }
    cout << "delete s: ";
    delete s; // The polymorphic deletion
    // (delete 0 is legal; it produces a no-op)
  }
  class BadShapeCreation : public logic_error {
  public:
    BadShapeCreation(string type)
    : logic_error("Cannot create type " + type) {}
  };
  Shape(string type) throw(BadShapeCreation);
};

class Circle : public Shape {
  Circle(Circle&);
  Circle operator=(Circle&);
  Circle() {} // Private constructor
  friend class Shape;
public:
  void draw() { cout << "Circle::draw" << endl; }
  void erase() { cout << "Circle::erase" << endl; }
  void test() { draw(); }
  ~Circle() { cout << "Circle::~Circle" << endl; }
};

class Square : public Shape {
  Square(Square&);
  Square operator=(Square&);
  Square() {}
  friend class Shape;
public:
```

656

```
  void draw() { cout << "Square::draw" << endl; }
  void erase() { cout << "Square::erase" << endl; }
  void test() { draw(); }
  ~Square() { cout << "Square::~Square" << endl; }
};

Shape::Shape(string type) throw(Shape::BadShapeCreation) {
  if(type == "Circle")
    s = new Circle;
  else if(type == "Square")
    s = new Square;
  else throw BadShapeCreation(type);
  draw();  // Virtual call in the constructor
}

char* sl[] = { "Circle", "Square", "Square",
  "Circle", "Circle", "Circle", "Square" };

int main() {
  vector<Shape*> shapes;
  cout << "virtual constructor calls:" << endl;
  try {
    for(size_t i = 0; i < sizeof sl / sizeof sl[0]; i++)
      shapes.push_back(new Shape(sl[i]));
  } catch(Shape::BadShapeCreation e) {
    cout << e.what() << endl;
    purge(shapes);
    return EXIT_FAILURE;
  }
  for(size_t i = 0; i < shapes.size(); i++) {
    shapes[i]->draw();
    cout << "test" << endl;
    shapes[i]->test();
    cout << "end test" << endl;
    shapes[i]->erase();
  }
  Shape c("Circle"); // Create on the stack
  cout << "destructor calls:" << endl;
  purge(shapes);
} ///:~
```

657

基类**Shape**包含一个对象指针作为其惟一的数据成员，该指针指向**Shape**类型的对象。（在创建一个“虚构造函数”的模式时，务必确保这个指针总是被初始化成指向一个激活的对象。）这个基类实际上就是一个代理，因为这是客户程序惟一所能看到和与之进行交互的对象。

每次从**Shape**派生新的子类时，必须回到基类并且在基类**Shape**的“虚构造函数”内的一个位置增加那个类型的创建。这并不是件很繁重的任务，但缺点是在**Shape**类和其所有的派生类之间形成了依赖关系。

在这个例子中，交给虚构造函数的关于要创建对象的类型信息必须是显式说明的：它是一个用来命名类型的**string**。但是模式也可以用其他信息——比如说，在一个语法分析器中，可以把扫描器的输出结果给虚构造函数，而构造函数将利用这些信息来决定创建何种类型的对象。

658

虚构造函数**Shape(type)**在所有派生类未声明前不能定义。然而，默认的构造函数能够在**class Shape**中定义，但是它应该被声明为**protected**的，所以不能创建临时的**Shape**对象。这个默认的构造函数只能被派生类对象的构造函数调用。程序员被迫显式地创建一个默认构造函数，因为如果没有定义构造函数编译器将会自动创建一个。因为必须定义**Shape(type)**，所以也必须定义**Shape( )**。

在这种模式中，默认构造函数至少有一个重要的工作要做——它必须将指针s设置为零值。这在初听起来有点奇怪，但是应当记得，默认构造函数将作为实际对象的构造的一部分被调用——用Coplien的术语来说，它是“信件”而不是“信封”。然而，“信件”也是从“信封”派生出来的，它也继承数据成员s。在“信封”中s很重要，因为它指向实际的对象，但是在“信件”中，s只是一个超重行李。可是，即便是额外行李也应该被初始化。如果不调用默认构造函数为“信件”把s赋为零值，将会出现问题（这在后面将会看到）。

虚构造函数使用其参数提供的信息，这些信息完全能够决定对象的类型。注意，这些类型信息在运行时才能读取和使用，而在一般情况下，编译器在编译时必须知道确切的类型（这是本系统能够有效地模拟虚构造函数的另外一个原因）。

虚构造函数使用其参数来选择要构造的实际对象（“信件”），然后对“信封”内的指针赋值。至此，“信件”类对象创建完成，因此任何虚函数的调用得以正确地重定向。

作为一个例子，考虑在虚构造函数中调用函数**draw( )**。如果跟踪这个调用（手工或者使用调试器），将会看到它是从基类**Shape**中的函数**draw( )**开始调用的。这个函数调用“信封”的**draw( )**，“信封”指针s指向它的“信件”。所有从**Shape**中派生出来的类型共享同一个接口，所以这个虚调用能够正确执行，虽然它似乎在构造函数中。（实际上，“信件”类的构造函数已经执行完毕。）只要基类中所有的虚调用通过这个指向“信件”的指针仅调用同一个虚函数，系统就能正确地运作。

为了了解它是如何工作的，请思考**main( )**函数中的代码。为了填充**vector shapes**，调用“虚构造函数”以便产生**Shape**对象。通常像这样的情况，应该用调用实际类型的构造函数，这种类型的虚指针（VPTR）应该安置在该对象中。然而在这里，在每种情况下虚指针（VPTR）都是指向**Shape**的一个对象，而不是指向特定的一种类型如**Circle**、**Square**或是**Triangle**。 659

在**for**循环中，为每个**Shape**对象调用函数**draw( )**和**erase( )**，虚函数调用通过VPTR解析到相应类型。然而，在各种情况下它都是**Shape**。事实上，读者也许想知道为什么**draw( )**和**erase( )**要声明为虚函数。在下一步可以看到原因：**draw( )**的基类版本通过“信件”指针s调用“信件”的虚函数**draw( )**。这时，这个调用解析到对象的实际类型，而不是基类**Shape**。因此每次调用虚函数时，使用虚构造函数的运行时代价只是一个额外的虚间接引用（virtual indirection）。

为了创建如**draw( )**、**erase( )**和**test( )**等任何将被覆盖的函数，如前所述，必须全部前向调用基类实现中的指针s。这是因为当调用发生时，调用“信封”的成员函数将被解析指向**Shape**而不是**Shape**的派生类。只有当前向调用的时候，s才发生虚行为。在**main( )**函数中，可以看到所有工作都能正确执行，即使调用发生在构造函数和析构函数中。

析构函数操作

在这种模式中析构活动同样也是很复杂的。为了了解这点，让我们从头至尾说明，当对指向创建在堆上的一个**Shape**对象（尤其是一个Square）的指针调用**delete**时会发生什么情况。（这是比建立在栈上的对象复杂得多的对象。）这将是一个经过多态接口的**delete**，并通过调用**purge( )**来完成。

Shapes中任何指针的类型都是基类**Shape**的类型，所以编译器通过**Shape**产生调用。通常情况下，可以说这是一个虚调用，所以**Square**的析构函数将被调用。但是在用虚构造函数系统中，由编译器来创建实际的**Shape**对象，即使构造函数初始化“信件”的指针为指向一个特定的**Shape**类型。这里使用了虚机制，但是，在**Shape**对象中的VPTR是**Shape**对象的虚指针，而不是**Square**对象的虚指针。这样就解析到**Shape**的析构函数，该析构函数调用

660 **delete**，该指针实际指向一个**Square**对象。这还是个虚调用，不过这时它解析指向**Square**对象的析构函数。

C++通过编译器确保继承层次结构中的所有析构函数都被调用。**Square**的析构函数最先被调用，然后顺序调用任何中间类的析构函数，直至最后，基类的析构函数被调用。这个基类的析构函数中包含代码**delete s**。当这个析构函数最初被调用时，它针对的是“信封”的**s**，而现在它针对的是“信件”的**s**，这是因为“信件”从“信封”中继承，而不是因为它包含了什么东西。所以这个**delete**调用不应该有任何操作。

解决此问题的方法是使“信件”的指针**s**指向零。这样，当调用“信件”的基类析构函数时，实际上得到的就是**delete 0**，它的定义是不执行任何操作。因为默认构造函数被设为保护的，它只是在“信件”对象的构造过程中被调用。这是将**s**置为零值的惟一情况。

虽然这种描述很有趣，但是可以看到这是一个复杂的方法，所以隐藏构造的最常见的工具一般是普通的“工厂方法”而不是“虚构造函数”模式这样的方法。

10.12 构建器模式：创建复杂对象

构建器（Builder）（它和前面已经讨论过的工厂方法一样，属于创建型模式）模式的目标是将对象的创建与它的“表示法”（representation）分开。这就意味着，创建过程保持原状，但是产生对象的表示法可能不同。“四人帮”指出，构建器模式和抽象工厂模式主要的区别就是，构建器模式一步步创建对象，所以及时展开输出创建过程就似乎很重要。此外，“主管（director）”获得一个切片的流（stream），并且将这些切片传递给构建器，每个切片用来执行创建过程中的一步。

下面有一个例子，作为模型的一辆自行车按照其类型（山地车、旅行车或赛车）来选择零部件组装一辆自行车。一个构建器与每个自行车类都关联，每个构建器实现的接口由抽象类**BicycleBuilder**中指定。单独的类**BicycleTechnician**表示“四人帮”中描述的“导向器”对象，它使用具体的**BicycleBuilder**对象来构造**Bicycle**对象。

661

```
//: C10:Bicycle.h
// Defines classes to build bicycles;
// Illustrates the Builder design pattern.
#ifndef BICYCLE_H
#define BICYCLE_H
#include <iostream>
#include <string>
#include <vector>
#include <cstddef>
#include "../purge.h"
using std::size_t;

class BicyclePart {
public:
  enum BPart { FRAME, WHEEL, SEAT, DERAILLEUR,
    HANDLEBAR, SPROCKET, RACK, SHOCK, NPARTS };
private:
  BPart id;
  static std::string names[NPARTS];
public:
  BicyclePart(BPart bp) { id = bp; }
  friend std::ostream&
  operator<<(std::ostream& os, const BicyclePart& bp) {
    return os << bp.names[bp.id];
  }
```

```
};

class Bicycle {
  std::vector<BicyclePart*> parts;
public:
  ~Bicycle() { purge(parts); }
  void addPart(BicyclePart* bp) { parts.push_back(bp); }
  friend std::ostream&
  operator<<(std::ostream& os, const Bicycle& b) {
    os << "{ ";
    for(size_t i = 0; i < b.parts.size(); ++i)
      os << *b.parts[i] << ' ';
    return os << '}';
  }
};

class BicycleBuilder {
protected:
  Bicycle* product;
public:
  BicycleBuilder() { product = 0; }
  void createProduct() { product = new Bicycle; }
  virtual void buildFrame() = 0;
  virtual void buildWheel() = 0;
  virtual void buildSeat() = 0;
  virtual void buildDerailleur() = 0;
  virtual void buildHandlebar() = 0;
  virtual void buildSprocket() = 0;
  virtual void buildRack() = 0;
  virtual void buildShock() = 0;
  virtual std::string getBikeName() const = 0;
  Bicycle* getProduct() {
    Bicycle* temp = product;
    product = 0;  // Relinquish product
    return temp;
  }
};

class MountainBikeBuilder : public BicycleBuilder {
public:
  void buildFrame();
  void buildWheel();
  void buildSeat();
  void buildDerailleur();
  void buildHandlebar();
  void buildSprocket();
  void buildRack();
  void buildShock();
  std::string getBikeName() const { return "MountainBike";}
};

class TouringBikeBuilder : public BicycleBuilder {
public:
  void buildFrame();
  void buildWheel();
  void buildSeat();
  void buildDerailleur();
  void buildHandlebar();
  void buildSprocket();
  void buildRack();
  void buildShock();
  std::string getBikeName() const { return "TouringBike"; }
};
```

662

663

```
class RacingBikeBuilder : public BicycleBuilder {
public:
  void buildFrame();
  void buildWheel();
  void buildSeat();
  void buildDerailleur();
  void buildHandlebar();
  void buildSprocket();
  void buildRack();
  void buildShock();
  std::string getBikeName() const { return "RacingBike"; }
};

class BicycleTechnician {
  BicycleBuilder* builder;
public:
  BicycleTechnician() { builder = 0; }
  void setBuilder(BicycleBuilder* b) { builder = b; }
  void construct();
};
#endif // BICYCLE_H ///:~
```

Bicycle持有一个**vector**，用于保存指向**BicyclePart**对象的指针，这些对象表示用于构造自行车的部件。由一个**BicycleTechnician**（本例中的“主管”）调用派生的**BicycleBuilder**对象的函数**BicycleBuilder::createproduct( )**来初始化一辆自行车的创建。**BicycleTechnician::construct( )**函数调用**BicycleBuilder**接口中的所有函数（因为它不知道有什么具体的构建器类型）。具体的构建器类省略了（通过空函数体）那些与他们所构建的自行车的类型无关的动作，如下面的实现文件所示：

```
//: C10:Bicycle.cpp {O} {-mwcc}
#include "Bicycle.h"
#include <cassert>
#include <cstddef>
using namespace std;

std::string BicyclePart::names[NPARTS] = {
  "Frame", "Wheel", "Seat", "Derailleur",
  "Handlebar", "Sprocket", "Rack", "Shock" };

// MountainBikeBuilder implementation
void MountainBikeBuilder::buildFrame() {
  product->addPart(new BicyclePart(BicyclePart::FRAME));
}
void MountainBikeBuilder::buildWheel() {
  product->addPart(new BicyclePart(BicyclePart::WHEEL));
}
void MountainBikeBuilder::buildSeat() {
  product->addPart(new BicyclePart(BicyclePart::SEAT));
}
void MountainBikeBuilder::buildDerailleur() {
  product->addPart(
    new BicyclePart(BicyclePart::DERAILLEUR));
}
void MountainBikeBuilder::buildHandlebar() {
  product->addPart(
    new BicyclePart(BicyclePart::HANDLEBAR));
}
void MountainBikeBuilder::buildSprocket() {
  product->addPart(new BicyclePart(BicyclePart::SPROCKET));
}
```

664

```
void MountainBikeBuilder::buildRack() {}
void MountainBikeBuilder::buildShock() {
  product->addPart(new BicyclePart(BicyclePart::SHOCK));
}

// TouringBikeBuilder implementation
void TouringBikeBuilder::buildFrame() {
  product->addPart(new BicyclePart(BicyclePart::FRAME));
}
void TouringBikeBuilder::buildWheel() {
  product->addPart(new BicyclePart(BicyclePart::WHEEL));
}
void TouringBikeBuilder::buildSeat() {
  product->addPart(new BicyclePart(BicyclePart::SEAT));
}
void TouringBikeBuilder::buildDerailleur() {
  product->addPart(
    new BicyclePart(BicyclePart::DERAILLEUR));
}
void TouringBikeBuilder::buildHandlebar() {
  product->addPart(
    new BicyclePart(BicyclePart::HANDLEBAR));
}
void TouringBikeBuilder::buildSprocket() {
  product->addPart(new BicyclePart(BicyclePart::SPROCKET));
}
void TouringBikeBuilder::buildRack() {
  product->addPart(new BicyclePart(BicyclePart::RACK));
}
void TouringBikeBuilder::buildShock() {}

// RacingBikeBuilder implementation
void RacingBikeBuilder::buildFrame() {
  product->addPart(new BicyclePart(BicyclePart::FRAME));
}
void RacingBikeBuilder::buildWheel() {
  product->addPart(new BicyclePart(BicyclePart::WHEEL));
}
void RacingBikeBuilder::buildSeat() {
  product->addPart(new BicyclePart(BicyclePart::SEAT));
}
void RacingBikeBuilder::buildDerailleur() {}
void RacingBikeBuilder::buildHandlebar() {
  product->addPart(
    new BicyclePart(BicyclePart::HANDLEBAR));
}
void RacingBikeBuilder::buildSprocket() {
  product->addPart(new BicyclePart(BicyclePart::SPROCKET));
}
void RacingBikeBuilder::buildRack() {}
void RacingBikeBuilder::buildShock() {}

// BicycleTechnician implementation
void BicycleTechnician::construct() {
  assert(builder);
  builder->createProduct();
  builder->buildFrame();
  builder->buildWheel();
  builder->buildSeat();
  builder->buildDerailleur();
  builder->buildHandlebar();
  builder->buildSprocket();
  builder->buildRack();
```

665

```
  builder->buildShock();
} ///:~
```

Bicycle流插入符为各个**BicyclePart**调用相应的插入符，并且打印其类型名称以便知道**Bicycle**对象包含的内容。程序举例如下：

```
//: C10:BuildBicycles.cpp
//{L} Bicycle
// The Builder design pattern.
#include <cstddef>
#include <iostream>
#include <map>
#include <vector>
#include "Bicycle.h"
#include "../purge.h"
using namespace std;

// Constructs a bike via a concrete builder
Bicycle* buildMeABike(
  BicycleTechnician& t, BicycleBuilder* builder) {
  t.setBuilder(builder);
  t.construct();
  Bicycle* b = builder->getProduct();
  cout << "Built a " << builder->getBikeName() << endl;
  return b;
}

int main() {
  // Create an order for some bicycles
  map <string, size_t> order;
  order["mountain"] = 2;
  order["touring"] = 1;
  order["racing"] = 3;

  // Build bikes
  vector<Bicycle*> bikes;
  BicycleBuilder* m = new MountainBikeBuilder;
  BicycleBuilder* t = new TouringBikeBuilder;
  BicycleBuilder* r = new RacingBikeBuilder;
  BicycleTechnician tech;
  map<string, size_t>::iterator it = order.begin();
  while(it != order.end()) {
    BicycleBuilder* builder;
    if(it->first == "mountain")
      builder = m;
    else if(it->first == "touring")
      builder = t;
    else if(it->first == "racing")
      builder = r;
    for(size_t i = 0; i < it->second; --i)
      bikes.push_back(buildMeABike(tech, builder));
    ++it;
  }
  delete m;
  delete t;
  delete r;

  // Display inventory
  for(size_t i = 0; i < bikes.size(); ++i)
    cout << "Bicycle: " << *bikes[i] << endl;
  purge(bikes);
}
```

```
/* Output:
Built a MountainBike
Built a MountainBike
Built a RacingBike
Built a RacingBike
Built a RacingBike
Built a TouringBike
Bicycle: {
  Frame Wheel Seat Derailleur Handlebar Sprocket Shock }
Bicycle: {
  Frame Wheel Seat Derailleur Handlebar Sprocket Shock }
Bicycle: { Frame Wheel Seat Handlebar Sprocket }
Bicycle: { Frame Wheel Seat Handlebar Sprocket }
Bicycle: { Frame Wheel Seat Handlebar Sprocket }
Bicycle: {
  Frame Wheel Seat Derailleur Handlebar Sprocket Rack }
*/ ///:~
```

这种模式的功能就是它将部件组合成为一个完整产品的算法与部件本身分开，这样就允许通过一个共同接口的不同实现来为不同的产品提供不同的算法。

10.13 观察者模式

观察者（Observer）模式用于解决一个相当常见的问题：当某些其他对象改变状态时，如果一组对象需要进行相应的更新，那么应该如何处理呢？这可以在Smalltalk的MVC（model-view-controller，模型-视图-控制器）的“模型-视图”或是几乎完全等价的“文档-视图设计模式”中见到。假定有一些数据（即“文档”）和两个视图：一个图形视图和一个文本视图。在更改“文档”数据时，必须通知这些视图更新它们自身，这就是观察者模式所要完成的任务。 668

在下面的代码中使用两种对象的类型以实现观察者模式。类**Observable**跟踪那些当一类对象发生某种变化时需要被通知的对象。类**Observable**为列表上的每个观察者调用成员函数**notifyObservers( )**。成员函数**notifyObservers( )**是基类**Observable**的一部分。

在观察者模式中有两个“变化的事件”：正在进行观察的对象的数量和更新发生的方式。这就是说，观察者模式允许修改这二者而不影响周围的其他代码。

可以用很多方法来实现观察者模式，下面的代码将创建一个程序框架，读者可根据这个框架构建自己的观察者模式代码。首先，这个接口描述了什么是观察者模式，如下所示：

```
//: C10:Observer.h
// The Observer interface.
#ifndef OBSERVER_H
#define OBSERVER_H

class Observable;
class Argument {};

class Observer {
public:
  // Called by the observed object, whenever
  // the observed object is changed:
  virtual void update(Observable* o, Argument* arg) = 0;
  virtual ~Observer() {}
};
#endif // OBSERVER_H ///:~
```

因为在这种方法中**Observer**与**Observable**交互作用，所以必须首先声明**Observable**。另外，类**Argument**是空的，在更新过程中它只担任一个基类的角色，用于传递需要的任何

参数类型。如果需要，也可以仅传递额外的像**void***类型这样的参数。在这两种情况下无论哪种情况都有向下类型转换的操作。

669 类**Observer**是只有一个成员函数**update( )**的“接口”类。当正在被观察的对象认为到了更新其所有观察者的时机时，它将调用此函数。函数的参数是可选的；可以调用一个没有参数的**update( )**，这仍然符合观察者模式的要求。然而，更常见的是——它允许被观察的对象传递引起更新操作的对象（因为一个**Observer**可以注册到多个被观察对象）和任何额外的有用信息，而不必强迫**Observer**对象自己去搜寻正在被更新的对象并取得任何其他所需的信息。

“被观察对象”的类型是**Observable**的类型：

```
//: C10:Observable.h
// The Observable class.
#ifndef OBSERVABLE_H
#define OBSERVABLE_H
#include <set>
#include "Observer.h"

class Observable {
  bool changed;
  std::set<Observer*> observers;
protected:
  virtual void setChanged() { changed = true; }
  virtual void clearChanged() { changed = false; }
public:
  virtual void addObserver(Observer& o) {
    observers.insert(&o);
  }
  virtual void deleteObserver(Observer& o) {
    observers.erase(&o);
  }
  virtual void deleteObservers() {
    observers.clear();
  }
  virtual int countObservers() {
    return observers.size();
  }
  virtual bool hasChanged() { return changed; }
  // If this object has changed, notify all
  // of its observers:
  virtual void notifyObservers(Argument* arg = 0) {
    if(!hasChanged()) return;
    clearChanged(); // Not "changed" anymore
    std::set<Observer*>::iterator it;
    for(it = observers.begin();it != observers.end(); it++)
      (*it)->update(this, arg);
  }
  virtual ~Observable() {}
};
#endif // OBSERVABLE_H ///:~
```

670

再次说明，这里的设计比实际必需的更精细些。只要有方法用**Observable**注册一个**Observer**并有方法为**Observable**更新其**Observer**，不必太在意其成员函数的设定。然而，这个设计的意图是可重用的。（它是从用于Java标准库的设计中摘取出来的。）[⊖]

Observable对象有一个用于指示是否已被修改的标志。在一个简单的设计中可能没有

⊖ 它与Java不同，因为在通知所有观察者之前**java.util.Observable.notifyObservers( )**不会调用**clearChanged( )**。

标志；在出现变化时所有对象都将得到通知。然而需要注意的是，标志状态的控制是**protected**，所以只有继承者才能决定是什么造成了这个变化，而不是产生派生**Observer**类的末端用户。

收集到的**Observer**对象被保存在一个**set<Observer*>**中以防止复制；集合中的**setinsert( )**、**erase( )**、**clear( )**和**size( )**函数是开放的，允许在任何时候添加和删除**Observer**对象，因此提供了运行时的灵活性。

大部分工作是在函数**notifyObservers( )**中做的。如果标志**changed**没有设置，它不做任何动作。否则，它将首先清除标志**changed**以防重复调用**notifyObservers( )**所造成的时间浪费。这些是在通知观察者之前做的，以防被调用的**update( )**会执行某些能够引起变化的操作，进而把这个变化反馈给**Observable**对象。然后它遍历集合**set**并回调每个**Observer**对象的成员函数**update( )**。

671

起初似乎可以使用一个普通的**Observable**对象来管理更新操作。但这不会起作用；为了使之生效，必须从**Observable**派生出子类并且在派生类的代码中某处调用函数**setChanged( )**。这就是设置标志“changed”的成员函数，这就意味着当调用**notifyObservers( )**时，事实上所有观测者将得到通知。在哪里调用**setChanged( )**取决于程序的逻辑设计。

现在我们进入了一个进退两难的窘境。被观察的对象将有不止一个类似的可选择项。例如，假设有一个用于处理GUI的可选择项——比如按钮——类似的可选择项有鼠标击发按钮、鼠标在按钮上方移过以及（由于某些原因）改变按钮颜色等。所以我们希望能够向不同的观察者报告所有这些事件，而每个观察者只对一种不同类型的事件感兴趣。

问题是，在这种情况下要达到以上目的采用多重继承：“为了处理鼠标击发按钮从**Observable**中继承，为了处理鼠标在按钮上方移动从**Observable**中继承，如此等等，好啦，…哦！这不可能实现”。

10.13.1 “内部类”方法

在某些情况下，必须（有效地）向上类型转换（upcast）成为多个不同的类型，但是在这种情况下，需要为同一个基类型提供几个不同的实现。从Java中引进了这种解决方法，这种方法比C++的嵌套类更优越。Java有一个被称为内部类的内建特征，它很像C++中的嵌套类，但是它能够通过隐式使用内部类创建的对象的“this”指针来访问其包含（外围）类的非静态数据成员。[⊖]

为了在C++中实现内部类（inner class）方法，必须显式获得和使用指向包含对象的指针。举例如下：

```
//: C10:InnerClassIdiom.cpp
// Example of the "inner class" idiom.
#include <iostream>
#include <string>
using namespace std;
class Poingable {
public:
  virtual void poing() = 0;
};

void callPoing(Poingable& p) {
```

672

⊖ 内部类和子程序闭包（subroutine closure）有些相似，子程序闭包用于引用一个函数调用的环境以便稍后复制。

```
  p.poing();
}

class Bingable {
public:
  virtual void bing() = 0;
};

void callBing(Bingable& b) {
  b.bing();
}

class Outer {
  string name;
  // Define one inner class:
  class Inner1;
  friend class Outer::Inner1;
  class Inner1 : public Poingable {
    Outer* parent;
  public:
    Inner1(Outer* p) : parent(p) {}
    void poing() {
      cout << "poing called for "
        << parent->name << endl;
      // Accesses data in the outer class object
    }
  } inner1;
  // Define a second inner class:
  class Inner2;
  friend class Outer::Inner2;
  class Inner2 : public Bingable {
    Outer* parent;
  public:
    Inner2(Outer* p) : parent(p) {}
    void bing() {
      cout << "bing called for "
        << parent->name << endl;
    }
  } inner2;
public:
  Outer(const string& nm)
  : name(nm), inner1(this), inner2(this) {}
  // Return reference to interfaces
  // implemented by the inner classes:
  operator Poingable&() { return inner1; }
  operator Bingable&() { return inner2; }
};

int main() {
  Outer x("Ping Pong");
  // Like upcasting to multiple base types!:
  callPoing(x);
  callBing(x);
} ///:~
```

673

这个例子（有意以最简单的语法形式来说明这种方法；在后面很快就可以看到其实际的用法）以接口**Poingable**和**Bingable**开始，每个接口包含一个成员函数。由**callPoing( )**和**callBing( )**提供的服务要求它们接收的对象分别实现相应的**Poingable**和**Bingable**接口，除此之外它们对对象没有别的请求，这就使得使用**callPoing( )**和**callBing( )**具有最大限度的灵活性。注意，这两个接口中都缺少**virtual**析构函数——这就意味着不能通过接口来完成

析构对象。

类**Outer**的构造函数包含一些私有数据（如**name**），它希望同时提供**Poingable**和**Bingable**两个接口，这样它就能同**callPoing( )**和**callBing( )**一起使用。（在这种情形下也可以仅使用多重继承，但在这里为清晰起见而保持简单的程序结构。）为了能够在**Outer**不派生自**Poingable**的前提下提供**Poingable**对象，这里使用了内部类方法。首先，**class Inner**的声明说明这是一个名为**Inner**的嵌套类。这就允许在后面能够将其声明为外部类**Outer**的友元类。随后，现在嵌套类能够访问外部类**Outer**的所有私有成员，现在就可以定义嵌套类了。注意，嵌套类有一个指向用于创建**Outer**对象的指针，这个指针必须在嵌套类的构造函数中进行初始化。最后，来自**Poingable**的**poing( )**函数得以实现。另外一个用来实现**Bingable**的内部类采用同样的过程实现。每个内部类只有一个**private**实例被创建，该实例在**Outer**的构造函数中被初始化。通过创建成员对象并返回对它们的引用，排除了对象生存期可能产生的问题。

注意，两个内部类都是**private**的，事实上客户代码都不能访问其任何实现细节，因为两个访问函数**operator Poingable&( )**和**operator Bingable&( )** 只返回一个用来向上类型转换为接口的引用，而不是实现它的对象。事实上，因为两个内部类是**private**的，客户代码甚至不能向下类型转换为实现类，这样就在接口和实现之间提供了完全的隔离。 674

这里获得了定义自动类型转换函数**operator Poingable&( )**和**operator Bingable&( )**的额外特权。在**main( )**函数中，可以看到这些允许提供一种使得**Outer**看起来像是从**Poingable**和**Bingable**多重继承来的语法形式。不同之处在于这种“类型转换”在此情况下是单向的。只可以得到向上类型转换为**Poingable**或**Bingable**的效果，但是不能向下类型转换回**Outer**。在下面**observer**的例子中，可以看到更典型的方法：通过提供使用普通的成员函数而不是自动类型转换函数来访问内部类对象。

10.13.2 观察者模式举例

具备了**Observer**和**Observable**头文件和内部类方法的知识，现在请看一个观察者模式的程序例子：

```
//: C10:ObservedFlower.cpp
// Demonstration of "observer" pattern.
#include <algorithm>
#include <iostream>
#include <string>
#include <vector>
#include "Observable.h"
using namespace std;

class Flower {
  bool isOpen;
public:
  Flower() : isOpen(false),
    openNotifier(this), closeNotifier(this) {}
  void open() { // Opens its petals
    isOpen = true;
    openNotifier.notifyObservers();
    closeNotifier.open();
  }
  void close() { // Closes its petals
    isOpen = false;
    closeNotifier.notifyObservers();
    openNotifier.close();
```

675

```
  }
  // Using the "inner class" idiom:
  class OpenNotifier;
  friend class Flower::OpenNotifier;
  class OpenNotifier : public Observable {
    Flower* parent;
    bool alreadyOpen;
  public:
    OpenNotifier(Flower* f) : parent(f),
      alreadyOpen(false) {}
    void notifyObservers(Argument* arg = 0) {
      if(parent->isOpen && !alreadyOpen) {
        setChanged();
        Observable::notifyObservers();
        alreadyOpen = true;
      }
    }
    void close() { alreadyOpen = false; }
  } openNotifier;
  class CloseNotifier;
  friend class Flower::CloseNotifier;
  class CloseNotifier : public Observable {
    Flower* parent;
    bool alreadyClosed;
  public:
    CloseNotifier(Flower* f) : parent(f),
      alreadyClosed(false) {}
    void notifyObservers(Argument* arg = 0) {
      if(!parent->isOpen && !alreadyClosed) {
        setChanged();
        Observable::notifyObservers();
        alreadyClosed = true;
      }
    }
    void open() { alreadyClosed = false; }
  } closeNotifier;
};

class Bee {
  string name;
  // An "inner class" for observing openings:
  class OpenObserver;
  friend class Bee::OpenObserver;
  class OpenObserver : public Observer {
    Bee* parent;
  public:
    OpenObserver(Bee* b) : parent(b) {}
    void update(Observable*, Argument *) {
      cout << "Bee " << parent->name
        << "'s breakfast time!" << endl;
    }
  } openObsrv;
  // Another "inner class" for closings:
  class CloseObserver;
  friend class Bee::CloseObserver;
  class CloseObserver : public Observer {
    Bee* parent;
  public:
    CloseObserver(Bee* b) : parent(b) {}
    void update(Observable*, Argument *) {
      cout << "Bee " << parent->name
        << "'s bed time!" << endl;
    }
```

676

```
  } closeObsrv;
public:
  Bee(string nm) : name(nm),
    openObsrv(this), closeObsrv(this) {}
  Observer& openObserver() { return openObsrv; }
  Observer& closeObserver() { return closeObsrv;}
};

class Hummingbird {
  string name;
  class OpenObserver;
  friend class Hummingbird::OpenObserver;
  class OpenObserver : public Observer {
    Hummingbird* parent;
  public:
    OpenObserver(Hummingbird* h) : parent(h) {}
    void update(Observable*, Argument *) {
      cout << "Hummingbird " << parent->name
        << "'s breakfast time!" << endl;
    }
  } openObsrv;
  class CloseObserver;
  friend class Hummingbird::CloseObserver;
  class CloseObserver : public Observer {
    Hummingbird* parent;
  public:
    CloseObserver(Hummingbird* h) : parent(h) {}
    void update(Observable*, Argument *) {
      cout << "Hummingbird " << parent->name
        << "'s bed time!" << endl;
    }
  } closeObsrv;
public:
  Hummingbird(string nm) : name(nm),
    openObsrv(this), closeObsrv(this) {}
  Observer& openObserver() { return openObsrv; }
  Observer& closeObserver() { return closeObsrv;}
};

int main() {
  Flower f;
  Bee ba("A"), bb("B");
  Hummingbird ha("A"), hb("B");
  f.openNotifier.addObserver(ha.openObserver());
  f.openNotifier.addObserver(hb.openObserver());
  f.openNotifier.addObserver(ba.openObserver());
  f.openNotifier.addObserver(bb.openObserver());
  f.closeNotifier.addObserver(ha.closeObserver());
  f.closeNotifier.addObserver(hb.closeObserver());
  f.closeNotifier.addObserver(ba.closeObserver());
  f.closeNotifier.addObserver(bb.closeObserver());
  // Hummingbird B decides to sleep in:
  f.openNotifier.deleteObserver(hb.openObserver());
  // Something changes that interests observers:
  f.open();
  f.open(); // It's already open, no change.
  // Bee A doesn't want to go to bed:
  f.closeNotifier.deleteObserver(
    ba.closeObserver());
  f.close();
  f.close(); // It's already closed; no change
  f.openNotifier.deleteObservers();
  f.open();
  f.close();
} ///:~
```

677

在这里，令人感兴趣的事件是**Flower**的打开或关闭。由于内部类方法的使用，这两个事件成为可以独立进行观察的现象。类**OpenNotifier**和**CloseNotifier**都派生自**Observable**，因此它们能够访问**setChanged( )**，并且能够处理需要**Observable**的任何事件。请注意，与**InnerClassIdiom.cpp**相反，**Observable**的派生是**public**的。这是因为它们的一些成员函数要求必须能够被客户程序员访问。没有任何规定要求内部类必须为**private**；在**InnerClassIdiom.cpp**中只是遵从“尽可能声明为私有”的设计原则。可以将这些类声明为**private**，并在**Flower**中设定代理来开放那些成员函数，但这并不会有多大好处。

678

在**Bee**和**Hummingbird**中内部类方法也很便利地定义了多种**Observer**，因为这两个类都需要独立观察**Flower**的打开与关闭。请注意，内部类方法是如何提供了许多和继承一样有益的特性（例如，能够访问外部类中的私有数据）。

在**main( )**中，可以看到观察者模式的主要有益之处：以**Observable**动态地注册和注销**Observer**获得在程序运行时改变行为的能力。这个灵活性是以显著增加代码的代价而达到的——读者可能经常能看到在设计模式中的这种折中：增加某处的复杂性以换取另一处的灵活性的提升和（或）复杂性的降低。

如果仔细研究前面的例子，就会发现**OpenNotifier**和**CloseNotifier**使用了基本的**Observable**接口。这意味着，可以从其他完全不同的**Observer**类派生；**Observer**与**Flower**之间惟一的联系是**Observer**接口。

另外一种完成这种细微粒度的可观察现象的方法是对该现象使用某种形式的标记，例如空类、字符串或枚举等表示不同类型的可观察行为。这种方法可以使用聚合而不是继承来实现，不同之处主要在于时间与空间效率间的折中。而对于客户来说，这种差异是可以忽略的。

679

10.14 多重派遣

在处理多个类交互作用的情况时，程序会变得特别散乱。例如，考虑一个解析和执行数学表达式的系统。在系统中希望使用**Number + Number**、**Number * Number**等方式表达，其中**Number**是一族数值对象的基类。但是如果给出**a + b**，并且不知道**a**或**b**的准确的类型，那么怎样才能让这二者适当地进行交互作用呢？

刚开始回答时，有一些事情可能没有考虑：C++只执行单重派遣（single dispatching）。这就是说，如果在多个不知道类型的对象之间操作，C++只能在其中一个类型上激发动态绑定机制。这不能解决这里描述的问题，因此程序员只能手工发现一些类型并且有效地制造自己的动态绑定行为。

这种解决方法被称为*多重派遣*（Multiple dispatching）（GoF在访问者模式的语境下描述了这种方法，访问者模式将在下节介绍）。这里只有两个派遣，被称为*双重派遣*（double dispatching）。读者可能会记得，多态只能通过虚函数调用来实现，所以如果想要发生多重派遣，必须有一个虚函数调用以确定每个未知的类型。因此，如果处理的是不同层次结构的两个类型的交互作用，则每个层次结构都必须有一个虚函数调用。通常，将设立这样一种结构，使得一个成员函数的调用导致多个虚函数调用，并且因此在该过程中确定多个类型：对于每个派遣都需要一个虚函数调用。下面例子中被调用的虚函数是**compete( )**和**eval( )**，二者都是同一类型的成员函数（对于多重派遣这并不是必要条件）：[⊖]

⊖ 这个例子出现在其他作者的书籍中之前，用C++和Java两种语言描述的这个例子已在网站www.MindView.net存在了多年而没有归属。

```
//: C10:PaperScissorsRock.cpp
// Demonstration of multiple dispatching.
#include <algorithm>
#include <iostream>
#include <iterator>
#include <vector>
#include <ctime>
#include <cstdlib>
#include "../purge.h"
using namespace std;

class Paper;
class Scissors;
class Rock;

enum Outcome { WIN, LOSE, DRAW };

ostream& operator<<(ostream& os, const Outcome out) {
  switch(out) {
    default:
    case WIN: return os << "win";
    case LOSE: return os << "lose";
    case DRAW: return os << "draw";
  }
}

class Item {
public:
  virtual Outcome compete(const Item*) = 0;
  virtual Outcome eval(const Paper*) const = 0;
  virtual Outcome eval(const Scissors*) const= 0;
  virtual Outcome eval(const Rock*) const = 0;
  virtual ostream& print(ostream& os) const = 0;
  virtual ~Item() {}
  friend ostream& operator<<(ostream& os, const Item* it) {
    return it->print(os);
  }
};

class Paper : public Item {
public:
  Outcome compete(const Item* it) { return it->eval(this);}
  Outcome eval(const Paper*) const { return DRAW; }
  Outcome eval(const Scissors*) const { return WIN; }
  Outcome eval(const Rock*) const { return LOSE; }
  ostream& print(ostream& os) const {
    return os << "Paper   ";
  }
};

class Scissors : public Item {
public:
  Outcome compete(const Item* it) { return it->eval(this);}
  Outcome eval(const Paper*) const { return LOSE; }
  Outcome eval(const Scissors*) const { return DRAW; }
  Outcome eval(const Rock*) const { return WIN; }
  ostream& print(ostream& os) const {
    return os << "Scissors";
  }
};

class Rock : public Item {
public:
```

680

681

```
  Outcome compete(const Item* it) { return it->eval(this);}
  Outcome eval(const Paper*) const { return WIN; }
  Outcome eval(const Scissors*) const { return LOSE; }
  Outcome eval(const Rock*) const { return DRAW; }
  ostream& print(ostream& os) const {
    return os << "Rock    ";
  }
};

struct ItemGen {
  Item* operator()() {
    switch(rand() % 3) {
      default:
      case 0: return new Scissors;
      case 1: return new Paper;
      case 2: return new Rock;
    }
  }
};

struct Compete {
  Outcome operator()(Item* a, Item* b) {
    cout << a << "\t" << b << "\t";
    return a->compete(b);
  }
};

int main() {
  srand(time(0)); // Seed the random number generator
  const int sz = 20;
  vector<Item*> v(sz*2);
  generate(v.begin(), v.end(), ItemGen());
  transform(v.begin(), v.begin() + sz,
    v.begin() + sz,
    ostream_iterator<Outcome>(cout, "\n"),
    Compete());
  purge(v);
} ///:~
```

682

Outcome将函数**compete( )**返回的不同结果进行分类，**operator<<**简化了显示特定**Outcome**的过程。

Item是将被多重派遣的那些类型的基类。**Compete::operator( )**有两个**Item***类型的参数（并不知道两者的确切类型），并且调用**virtual Item::compete( )**函数开始双重派遣过程。虚拟机制决定了**a**的类型，因此它激发了在函数**compete( )**内部的**a**的具体类型的产生。在保留该类型的基础之上，函数**compete( )**调用**eval( )**执行第2次派遣。将其自身（**this**指针）作为一个参数传递给函数**eval( )**，从而产生一个对重载的**eval( )**函数的调用，因此保存了第1次派遣的类型信息。在完成第2次派遣时，两个**Item**对象的确切类型就都知道了。

在**main( )**函数中，STL算法**generate( )**生成**vector v**中的元素内容，然后**transform( )**在两个范围上应用**Compete::operator( )**。这个版本的**transform( )**产生第1个范围的起始和末尾点（包含双重派遣中使用的左边**Item**）；第2个范围的起始点，这个范围持有从双重派遣中所使用的右边的**Item**；目标迭代器在这个例子中是标准输出；以及用于为每个对象调用的函数对象（一个临时的**Compete**类型）。

建立多重派遣需要做许多工作，但是请记住这样做的好处是在调用的时候能够以简洁的句法表达方式达到预期的效果——而不是编写出笨拙的代码在调用的时候决定一个或多个对象的类型，可以说："你们两个！不管是什么类型，彼此之间可以适当地进行交互作用。"然而，在

编写多重派遣的程序代码之前，确保这种简洁性是非常重要的。

注意，利用表查找来进行多重派遣是有效的。在这里使用虚函数来进行查找，用来代替进行杂乱的表查找。如果有较多的派遣（并且有增加和修改的可能），表查找也许是更好的解决问题的方法。

用访问者模式进行多重派遣 683

访问者模式（Visitor，GoF中最后一个也是最复杂的一个模式）的目标是将类继承层次结构上的操作与这个层次结构本身分开。这是一个相当古怪的动机，因为在面向对象编程中所做的大部分工作是将数据和操作组合在一起来形成对象，并利用多态性根据对象的确切类型自动选择操作的正确变化。

利用访问者模式将操作从类的继承层次结构中提取出来置入一个独立的外部层次结构。“主层次结构”包含一个函数**visit( )**，该函数接受任何来自操作层次结构的对象。结果得到了两个类继承层次结构而不是一个。此外，可以看到，“主层次结构”变得很脆弱——如果要增加一个新类，也要强制改动第2个层次结构。因此，GoF认为主层次结构应该“很少地变化”。这个限制非常有限，从而更进一步降低了这种模式的可应用性。

为了便于讨论，假定主类层次结构是固定的；也许它是由其他供应商提供的，不能对该层次结构进行改动。如果有这个库的源代码就可以在基类中增加新的虚函数，但是，由于某些原因这是不可行的。一个更可能的方案就是增加新的虚函数，这样做很笨拙，或者说是难以维护的。GoF主张“在跨越不同的节点类上分配所有这些操作，将导致系统难以理解、维护和修改”。（读者将会看到，这样做将导致访问者模式更加难以理解、维护和修改。）GoF的另外一个主张是，要避免由于使用过多的操作而“玷污”了主层次结构的接口（但是，如果接口太“臃肿”了，应该问一下这个对象的要做的事情是否太多了）。

然而，库的创建者必定已预见到，用户将需要加入新的操作到层次结构中去，因此他们将函数**visit( )**包含了进去。

因此（假定实际上需要这么做）两难的窘境就是，用户需要向基类中添加新的成员函数，但是由于某种原因用户不能接触到基类。那么该如何处理这种情况呢？

访问者模式建立于前一节内容所示的双重派遣方案之上。访问者模式允许创建一个独立的类层次结构**Visitor**而有效地对主类的接口进行扩展，这个独立的类层次结构将主类上的各种 684
操作“虚化”。主类对象仅“接受”访问者，然后调用访问者的动态绑定的成员函数。因此，创建一个访问者，并将其传递给主层次结构，便可以获得和虚函数一样的效果。举例如下：

```
//: C10:BeeAndFlowers.cpp
// Demonstration of "visitor" pattern.
#include <algorithm>
#include <iostream>
#include <string>
#include <vector>
#include <ctime>
#include <cstdlib>
#include "../purge.h"
using namespace std;

class Gladiolus;
class Renuculus;
class Chrysanthemum;

class Visitor {
public:
```

```
  virtual void visit(Gladiolus* f) = 0;
  virtual void visit(Renuculus* f) = 0;
  virtual void visit(Chrysanthemum* f) = 0;
  virtual ~Visitor() {}
};

class Flower {
public:
  virtual void accept(Visitor&) = 0;
  virtual ~Flower() {}
};

class Gladiolus : public Flower {
public:
  virtual void accept(Visitor& v) {
    v.visit(this);
  }
};

class Renuculus : public Flower {
public:
  virtual void accept(Visitor& v) {
    v.visit(this);
  }
};

class Chrysanthemum : public Flower {
public:
  virtual void accept(Visitor& v) {
    v.visit(this);
  }
};

// Add the ability to produce a string:
class StringVal : public Visitor {
  string s;
public:
  operator const string&() { return s; }
  virtual void visit(Gladiolus*) {
    s = "Gladiolus";
  }
  virtual void visit(Renuculus*) {
    s = "Renuculus";
  }
  virtual void visit(Chrysanthemum*) {
    s = "Chrysanthemum";
  }
};

// Add the ability to do "Bee" activities:
class Bee : public Visitor {
public:
  virtual void visit(Gladiolus*) {
    cout << "Bee and Gladiolus" << endl;
  }
  virtual void visit(Renuculus*) {
    cout << "Bee and Renuculus" << endl;
  }
  virtual void visit(Chrysanthemum*) {
    cout << "Bee and Chrysanthemum" << endl;
  }
};
```

685

```
struct FlowerGen {
  Flower* operator()() {
    switch(rand() % 3) {
      default:
      case 0: return new Gladiolus;
      case 1: return new Renuculus;
      case 2: return new Chrysanthemum;
    }
  }
};

int main() {
  srand(time(0)); // Seed the random number generator
  vector<Flower*> v(10);
  generate(v.begin(), v.end(), FlowerGen());
  vector<Flower*>::iterator it;
  // It's almost as if I added a virtual function
  // to produce a Flower string representation:
  StringVal sval;
  for(it = v.begin(); it != v.end(); it++) {
    (*it)->accept(sval);
    cout << string(sval) << endl;
  }
  // Perform "Bee" operation on all Flowers:
  Bee bee;
  for(it = v.begin(); it != v.end(); it++)
    (*it)->accept(bee);
  purge(v);
} ///:~
```

686

Flower是主层次结构，**Flower**的各个子类通过函数**accept( )**得到一个**Visitor**。**Flower**主层次结构除了函数**accept( )**外没有别的操作，因此**Flower**层次结构的所有功能都将包含在**Visitor**层次结构中。注意，**Visitor**类必须要了解**Flower**的所有具体类型，如果添加一个**Flower**的新类型，整个**Visitor**层次结构必须重新工作。

每个**Flower**中的**accept( )**函数开始一个双重派遣，如上一节所述的双重派遣。第1次派遣决定了**Flower**的准确类型，第2次派遣决定了**Visitor**的准确类型。一旦知道了它们的准确类型，就可以对这两者执行恰当的操作。

因为其不寻常的动机以及显得愚笨的约束，使得人们极不可能使用访问者模式。GoF的例子是难以令人信服的——首先是编译器（编写编译器的人不是很多，似乎极少有人将访问者模式用于这些编译器中），他们也不适用于其他一些例子，认为用户实际上不可能像这样使用访问者模式来解决问题。为了使用访问者模式，用户将面临比在GoF中表现出来更大的压力从而抛弃普通的面向对象结构——这样做实际上获得了什么益处而值得换来如此多的复杂性和限制呢？当发现需要多个新的虚函数时为何不可以在基类中仅添加它们？或者，如果实际上需要添加新函数到现存的层次结构中而又不能修改那个层次结构，在这种情况下为什么不先考虑尝试使用多重继承呢？（尽管如此，用这种方法“挽救”现存的层次结构的可能性还是很小的。）出于同样的考虑，为了使用访问者模式，现存的层次结构必须一开始就将函数**visit( )**包括进来，因为如果它在后面添加进来的话就意味着可以修改这个层次结构，这样就能在该层次结构中添加需要的普通虚函数了。不！访问者模式从开始就必须是体系结构的一部分，为了使用它需要有比在GoF中提到的更伟大的动机。[⊖]

687

⊖ 将访问者模式包含在GoF中的动机可能是因为它非常灵巧。在一个专题讨论会上，GoF的一个作者对我们中的一个人这样说过：“访问者是我最喜欢的模式”。

之所以在这里介绍访问者模式，是因为看到它在不该使用的时候被使用了，正如多重继承和任何其他很多方法被不正确地使用一样。在使用访问者模式之前务必三思，多问几个为什么。比如，真的不能在基类中添加新的虚函数了吗？在主层次结构中真的需要限制添加新的类型吗？

10.15 小结

正如任何其他抽象的特点，设计模式的特点就是为了使工作更加容易。系统中总是有一些东西在变化——这可能是在软件项目生命周期中代码的变化，或许是在某个程序执行的生命周期期间某些对象的变化。找出变化的东西，利用设计模式封装这些变化，并使这些变化能够得到控制。

人们在进行程序设计时很容易迷恋于使用某个特定的设计模式，并且如果因为刚刚知道如何做就贸然去做也将给自己带来烦恼。最难做到的是什么？有点讽刺意味，是遵循《极限编程》(《*Extreme Programming*》) 中的那句格言："只要能用，就做最简单的"。仅仅做最简单的东西，不仅能够最快速的实现设计，而且其设计也很容易维护。如果这种最简单的东西不能完成工作，读者很快就会发现，除了花费时间编写复杂的实现方法之外，它们还是不起作用的。

688

10.16 练习

10-1 创建程序**SingletonPattern.cpp**的一个变体，使其所有函数成为静态函数。在这种情况下还需要**instance( )**函数吗？

10-2 基于程序**SingletonPattern.cpp**，创建一个类，此类提供一个与某个服务的连接，这个服务向（从）一个配置文件中存取数据。

10-3 基于程序**SingletonPattern.cpp**，创建一个管理其固定数目对象的类。假定这些对象是数据库连接，在任何一次读/写操作中只允许使用这些对象中的某个固定数目的对象。

10-4 通过向系统中增加另外一种状态来修改程序**KissingPrincess2.cpp**，这样每次亲吻之后将使**creature**青蛙王子进入下一状态。

10-5 在《C++编程思想》第2版第1卷和第2卷中（可以从www.BruceEckel.com下载）找到头文件**C16:TStack.h**。为这个类创建这样一个适配器，使该类能够使用此适配器将**STL** 算法**for_each( )**应用于**TStack**的元素。创建一个元素类型为**string**的**TStack**，用字符串元素填充它，并且使用**for_each( )**对**TStack**中的所有字符串中的所有字母字符进行计数。

10-6 创建一个能够从命令行中获取文件名列表的框架（即使用模板方法模式）。它把除了最后一个文件以外的所有文件作为读文件打开，最后一个文件作为写文件打开。该框架使用不确定的方法处理每个输入文件，并且将输出写到最后一个文件。利用继承于自定义的这个框架去创建两个独立的应用程序：

1）将每个文件中的所有字母转换为大写。

2）在这些文件中寻找由第1个文件给出的那些单词。

10-7 使用策略模式而不是模板方法模式来修改练习10-6。

10-8 修改程序**Strategy.cpp**使其包含状态行为，使其在对象**Context**的生存期期间能够改变策略。

10-9 修改程序**Strategy.cpp**，使用职责链方法，使其能尝试从不同方法中选取一种来显示出它们的名字并且不容许忘记它。

689

10-10 在程序**ShapeFactory1.cpp**中增加一个**Triangle**类。

10-11 在程序**ShapeFactory2.cpp**中增加一个**Triangle**类。

10-12 在程序**AbstractFactory.cpp**中增加一个名为**GnomesAndFairies**的**GameEnvironment**新类型。

10-13 修改程序**ShapeFactory2.cpp**让它用一个抽象工厂模式来创建不同的图形集合（比如说，一个特定的工厂类型对象创建“厚图形”，另一工厂类型对象创建“薄图形”，但是每个工厂对象都能够创建所有图形：圆、矩形、三角形等等）。

10-14 修改程序**VirtualConstructor.cpp**使其能够在**Shape::Shape(string type)**中使用**map**而不是**if-else** 语句。

10-15 将一个文本文件分解成单词的一个输入流（为简洁起见：输入流在空白字符处进行分解）。创建一个能将单词送入一个集合**set**里的构建器（builder），和另外一个能生成包含单词和统计单词出现次数的**map**的构建器（即对单词进行计数）。

10-16 在两个类中创建一个最小限度的**Observer-Observable**设计，在这个设计中没有基类，在文件**Observer.h**中没有额外的参数，并且在文件**Observable.h**中没有成员函数。仅仅用这两个类创建这个最小限度的设计，然后创建一个**Observable**和多个**Observer**，并使**Observable**更新**Observer**来演示你的设计。

10-17 修改程序**InnerClassIdiom.cpp**使**Outer**用多重继承而非内部类方法来实现。

10-18 修改程序**PaperScissorsRock.java**，使用表查找而非双重派遣。最容易的方法就是创建一个**map**的**map**，将每个对象的每个**map**的**typeid(obj).name( )**信息作为其关键字。然后就可以这样查找：**map[typeid(obj1).name( )] [typeid(obj2).name( )]**。注意如何能使系统配置更加简化。何时使用此方法比使用硬编码动态派遣更合适？能否创建一个不使用表查找而使用动态派遣的句法简洁的系统？

10-19 创建一个商业模型环境，其拥有3个**Inhabitant**的类型：**Dwarf**（用于工程师）、**Elf**（用于销售人员）以及**Troll**（用于经理）。现在创建一个名为**Project**的类，该类可以实例化不同的人，并且使用多重派遣使他们在相互之间使用**interact( )**。

10-20 修改练习10-19，以便使其交互作用更加细化。每个**Inhabitant** 能够利用**getWeapon( )**随机产生一个**Weapon**：**Dwarf** 使用**Jargon**或**Play**，**Elf** 使用**InventFeature** 或**SellImaginaryProduct**，**Troll**使用**Edict**和**Schedule**。在每次交互作用中决定哪些武器“赢”或“输”（就像在文件**PaperScissorsRock.cpp**中一样）。在**Project**中增加一个成员函数**battle( )**，这个函数获得两个**Inhabitant**并且使它们进行对抗比赛。现在为**Project**创建一个成员函数**meeting( )**，该成员函数用于创建**Dwarf**组、**Elf**组和**Manager**组，并且用函数**battle( )**让这几个小组相互之间对抗，直到只有一个小组的成员剩下。这个小组就是胜利者。 690

10-21 在程序**BeeAndFlowers.cpp**加入一个**Hummingbird Visitor**。

10-22 加入一个**Sunflower** 类型到程序**BeeAndFlowers.cpp**中，注意这样原程序需要怎样修改才能适应新增的类型？

10-23 修改程序**BeeAndFlowers.cpp**，不使用访问者模式，而“恢复”使用平常的类层次结构。并且将**Bee**变成一个收集参数方法。

691

第11章 并　发

对象提供了将一个程序分解为若干个独立部分的途径。在实际的工作中也经常需要把一个程序分割成若干个分开的、独立运行的子任务。

使用多线程处理（multithreading），每个独立的子任务都会被执行的线程（thread of execution）驱动，程序就好像每个线程都拥有自己的CPU。其底层实现机制实际上为线程划分出了CPU时间，但是在一般情况下，程序员在编程时并不需要去考虑它，这有助于简化多线程编程。

进程（process）是在其自己的地址空间运行的自含式（self-contained）程序。周期性地把CPU从一个任务切换到另一个任务，多任务处理（multitasking）操作系统在同一时刻可以运行多个进程（程序），使得它们看上去就好像都在独自运行。线程（thread）是一个进程内的单一连续的控制流。因此一个进程可以有多个并发执行的线程。由于这些线程运行在一个进程内，所以它们分享内存和其他资源。编写多线程处理程序中主要的困难就是在不同线程之间协调对这些资源的使用。

多线程处理有多种应用，而当程序的某些部分与一个特定事件或资源结合在一起的时候，最经常需要使用多线程。为了防止挂起程序的其余部分，需要创建一个与那个特定事件或资源关联在一起的线程，并使这个线程独立于主程序运行。

学习并发编程像是步入了一个崭新的世界，类似学习一门新的编程语言，或至少是学习一组新的语言概念。随着在大多数的微机操作系统中出现了支持线程的操作，在编程语言或者程序库中，也出现了用于线程的功能扩充。总而言之，线程编程：

1) 不仅看起来神秘，而且需要人们转换一下思考编程的方式。

692

2) 各种语言中对线程的支持看上去都是相似的。当理解了线程时，就会理解一个共同的表述方式。

理解并发编程与理解多态性有类似的难度。经过一番努力，就可以彻底了解其基本机制，但一般需要深入地学习和理解才能够真正掌握其实质。本章的目标是给读者打下有关并发编程基本原理的坚实基础，这样就会理解基本概念并且编写出合理的多线程处理程序。不过读者也要意识到，这也许会使你很容易变得太过自信。如果要编写任何复杂的程序，则需要研读关于这个主题的专著。

11.1 动机

使用并发的最能激发人们兴趣的理由之一，就是产生一个可做出响应的用户界面。考虑一个程序，其在执行某项强烈需要CPU的操作时，往往会忽略用户的输入并且无法做出响应。程序既需要继续执行其操作，又需要把控制权归还给用户界面，这样程序才可以响应用户的请求，这就是问题的关键。如果有一个“退出”按钮，我们不希望被迫在程序中的每个代码块中轮询检测它的状态。（这将会使数个退出按钮代码贯穿整个程序，对它的维护很让人头痛。）然而，却希望对这些退出按钮能够做出响应，就好像系统在定期地检测它一样。

传统的函数不可能在继续进行其操作的同时，又把控制权归还给程序的其余部分。事实上，这听起来像是一个不可能完成的任务，就好像一个CPU必须能同时出现在两个地方，但这正是

严谨的并发机制提供的错觉效果（在多处理器系统的情形中，这可不只是错觉）。

也可以使用并发机制来优化信息的吞吐量。比如，程序在等待信息输入到达I/O端口的时候可以做些其他重要的工作。要是没有线程处理，惟一可行的解决方法就是不断轮询I/O端口，但这个方法不仅笨拙而且实现起来比较困难。

如果有一台多处理器的计算机（multiprocessor machine），多个线程就可以分布在多个处理器上，用此方法可以极大地提高信息的吞吐量。这种情况通常出现在使用功能强大的多处理器的web服务器上，这样一来，就可以在程序中给每个请求分配一个线程，将大量的用户请求分配到多个CPU来进行处理。 693

在单CPU计算机上，一个使用多线程的程序仍然一次只能做一件事情，所以不使用任何线程编写出具有相同功能的程序在理论上是可能的。然而，多线程处理提供的重要好处是在程序的组织方面，可以使程序的设计极大的简化。某些类型的问题，比如模拟——例如，一个视频游戏——如果不支持并发是很难解决的。

线程处理模型为编程方式提供了方便，可以在同一时间内魔术般地简化一个程序中的多个操作：CPU将会轮流给每个线程分配一些CPU时间。[⊖]每个线程都觉得自己在一直占有CPU，但事实上CPU时间被切成片段分配给所有的线程。运行在多CPU计算机上的程序是个例外。但是，关于线程处理的一个重大好处是可以使人们从这一层次中抽出身来，所以代码不需要知道实际上是运行在单CPU计算机上还是多CPU计算机上。[⊖]因此，使用线程是创建透明可扩展程序的一条途径——如果一个程序运行得太慢，可以很容易地给所使用的计算机增加CPU来加速程序的运行。现在的趋向是，进行多任务处理和多线程处理是利用多处理器系统最合理的途径。

线程处理多少会降低进行计算的效率，但是从改善程序设计、资源平衡以及给用户提供方便等方面来说，还是相当值得的。一般情况下，使用线程能够创建一个更加松散耦合的设计（loosely coupled design）；否则，部分代码将被迫对这些通常由线程处理的工作花费更大的精力。

11.2 C++中的并发

694

在C++标准委员会创建最初的C++标准时，并发机制被明确排除在外，因为C没有并发，也还因为有许多极具竞争力的近似方法可以实现并发处理。似乎有太多的限制迫使程序员只能用这些方法中的一个。

然而，那些可供选择的方法，结果被证明是错的。为使用并发，就要找到和学习一个库，这就涉及库的特性和某个特定（软件）供应商的产品在工作时是否可靠的问题。另外，没有人能够保证这样的库能运行在不同的编译器上或者跨不同的平台运行。并且，既然并发不是标准语言的一部分，所以要找到懂得并发编程的C++程序员也会更困难。

另一种有影响的Java语言，把并发包含在核心语言中。尽管多线程处理仍然是复杂的，但Java程序员在学习的起始就注意学习多线程并从一开始就使用它。

C++标准委员会正在考虑把支持并发处理的功能加入到下一代C++语言中，但是在这一次正在编写的新版本中，还不清楚加入并发处理的库看起来像什么样子。因此，作者决定把ZThread库作为这一章的基础。首选该设计的原因，因为它是源代码开放的，并且可以免费在

⊖ 当系统使用时间分片机制时（比如Windows）这是正确的。Solaris 使用一个FIFO 并发模型：除非一个更高优先级的线程被唤醒，当前的线程会一直运行直到它被阻塞或终止。那意味着其他有相同优先级的线程直到当前线程放弃处理器后才会运行。

⊖ 假设我们为多CPU设计了它。否则在一个时间分片的单处理器系统上似乎运行良好的代码在移植到多CPU系统上时会失败，这是由于额外的CPU会引发问题而单CPU系统则不会。

http://zthread.sourceforge.net 网站上获取。ZThread的作者，IBM的Eric Crahen，为本章提供了很多有用的工具。[⊖]

本章仅使用ZThread库的一个子集，以传达线程处理的基本思想。值得注意的是，ZThread库还包含重要的比这里所展现的更复杂的对线程的支持，应该深入研究这个库才能更进一步地、完全理解它的性能。

695 **安装ZThread**

请注意，ZThread库是一个独立的项目，本教材的作者并不支持它；只是在本章中使用这个库，不能提供关于安装发行（installation issue）的技术支持。浏览ZThread的网站可以得到安装支持以及勘误报告。

ZThread库以源代码形式发布。从ZThread网站将其下载后（版本2.3或更高的版本），必须首先编译这个库，然后装配到项目中来使用这个库。

对大多UNIX风格的操作系统（Linux、SunOS、Cygwin等等），编译ZThread首选的方法是使用装配脚本（configure script）。文件解包（使用**tar**）后，仅执行：

```
./configure && make install
```

从ZThread档案文件的主目录开始编译，在/usr/local目录安装库的一份拷贝。使用这个脚本时可以自定义一些选项，包括文件的位置。若要了解详细内容，可以使用下面这个命令：

```
./configure -help
```

ZThread的代码也被组织成能对其他平台和编译器（比如Borland、Microsoft和Metrowerks）进行简化编译的形式。为了完成这个工作，创建一个新项目，把ZThread档案文件的src目录中的所有.cxx文件加入到文件列表中，然后进行编译。同时也要确保把档案文件的include目录包含在该项目的头文件搜索路径（header search path）下。具体的细节根据编译器的不同而有所不同，所以需要比较熟悉这些工具包后才能够使用这个选项。

一旦编译成功，下一步就是创建一个使用这个重新编译好的库的项目。首先，让编译器知道头文件放置的位置，以便程序中的**#include**语句能够正常工作。典型地，要将如下选项加入到工程：

```
-I/path/to/installation/include
```

如果使用装配脚本，可以选择任意安装路径作为前缀的路径（默认的情况是在/usr/local）。
696 如果使用build目录中的某个项目文件，只需将ZThread档案文件的主目录设置为安装路径。

接下来，需要在项目中加入一个选项，这会使连接器（linker）知道到哪里去寻找库。如果使用装配脚本，如下所示：

```
-L/path/to/installation/lib -lZThread
```

如果使用一个已提供的项目文件，那么应该这样做：

```
-L/path/to/installation/Debug ZThread.lib
```

再说一遍，如果使用装配脚本，可以选择任意安装路径作为前缀的路径。如果使用已提供的项目文件，路径就是ZThread档案文件的主目录。

注意，如果使用Linux或使用Windows下的Cygwin（www.cygwin.com），则不需要修改包含路径或库文件路径；安装进程及默认选项会做好全部工作。

在Linux下，也许需要把下面的东西添加到 **.bashrc**文件中，以便让运行时系统（runtime

⊖ 本章的大部分开始于《Thinking in Java, 3 third edition》（Prentice Hall 2003）的“并发”一章，而在处理上有非常显著的改变。

system）能够在执行本章中的程序时找到共享库文件**LibZThread-x.x.so.O**:

```
export LD_LIBRARY_PATH=/usr/local/lib:${LD_LIBRARY_PATH}
```

（假设使用的是默认安装进程和位于路径/user/local/lib下的共享库；否则，把路径改成用户所使用的位置。）

11.3 定义任务

一个线程执行一个任务（task），所以需要用某种方法来描述这个任务。**Runnable**类提供了一个公共接口来执行任何任意的任务。在这里，**Runnable**类是ZThread 库的核心，在安装完ZThread 库后，可以在include目录下的**Runnable.h**文件中找到它:

```
class Runnable {
public:
  virtual void run() = 0;
  virtual ~Runnable() {}
};
```

把**Runnable**类做成一个抽象基类，**Runnable**类就可以很容易地将一个基类与其他类结合起来。 697

为了定义一个任务，可以从**Runnable**类继承并且重载**run( )**函数，使任务去做命令它做的事情。

例如，下面的这个**LiftOff**任务显示了在火箭发射离地升空前的倒计时:

```
//: C11:LiftOff.h
// Demonstration of the Runnable interface.
#ifndef LIFTOFF_H
#define LIFTOFF_H
#include <iostream>
#include "zthread/Runnable.h"

class LiftOff : public ZThread::Runnable {
  int countDown;
  int id;
public:
  LiftOff(int count, int ident = 0) :
    countDown(count), id(ident) {}
  ~LiftOff() {
    std::cout << id << " completed" << std::endl;
  }
  void run() {
    while(countDown--)
      std::cout << id << ":" << countDown << std::endl;
    std::cout << "Liftoff!" << std::endl;
  }
};
#endif // LIFTOFF_H ///:~
```

标识符**id**能区别该任务的多个实例。如果只创建了单个实例，可以使用**ident**的默认值。析构函数允许读者看到一个任务已被正确地销毁。

在下面的例子中，任务的**run( )**函数不是被单独的线程驱动；它在**main( )**函数中仅被直接调用。

```
//: C11:NoThread.cpp
#include "LiftOff.h"

int main() {
```

```
  LiftOff launch(10);
    launch.run();
  } ///:~
```

698

当一个类从**Runnable**派生出来的时候，它必须有一个**run( )**函数，但它却没有什么特别的——没有产生任何天生的线程处理的能力。

为完成线程处理的行为，必须使用线程类**Thread**。

11.4 使用线程

为了使用线程驱动**Runnable**对象，就要创建独立的**Thread**对象，并且把一个**Runnable**指针传递给**Thread**的构造函数。这样就完成了线程的初始化，然后调用**Runnable**的 **run( )**函数将其作为一个可中断线程。使用一个**Thread**来驱动**LiftOff**，下面的例子显示了任何任务可以怎样在其他线程的语境中运行。

```
//: C11:BasicThreads.cpp
// The most basic use of the Thread class.
//{L} ZThread
#include <iostream>
#include "LiftOff.h"
#include "zthread/Thread.h"
using namespace ZThread;
using namespace std;

int main() {
  try {
    Thread t(new LiftOff(10));
    cout << "Waiting for LiftOff" << endl;
  } catch(Synchronization_Exception& e) {
    cerr << e.what() << endl;
  }
} ///:~
```

Synchronization_Exception是ZThread库的一部分，并且是所有ZThread异常的基类。如果在启动或正在使用线程的时候有错误发生，将会抛出这个异常。

Thread类构造函数仅需要一个指向**Runnable**对象的指针。创建一个**Thread**对象将为线程完成必要的初始化，然后调用**Runnable** 的**run( )**成员函数启动该任务。即使**Thread**的构造函数有效地调用了一个长时间运行的函数，这个构造函数也会快速返回。这里已经使一个成员函数有效地调用了**LiftOff::run( )**函数，并且那个成员函数还没有执行完，但是由于**LiftOff::run( )**函数正在被一个不同的线程执行，所以仍然可以继续在**main( )**线程中执行其他操作。（这种能力不仅限于**main( )** 线程——任何线程都可以启动另外的线程。）运行该程序就可以看到这一点。即使**LiftOff::run( )**已经被调用，“Waiting for LiftOff”消息也将会在倒数计数完成之前显现。因此，该程序同一时刻运行了两个函数——**LiftOff::run( )**和 **main( )**。

699

现在可以很容易地添加更多的线程来驱动更多的任务。在这里，可以看到所有的线程如何与其他的线程协调运行：

```
//: C11:MoreBasicThreads.cpp
// Adding more threads.
//{L} ZThread
#include <iostream>
#include "LiftOff.h"
#include "zthread/Thread.h"
using namespace ZThread;
using namespace std;
```

```
int main() {
  const int SZ = 5;
  try {
    for(int i = 0; i < SZ; i++)
      Thread t(new LiftOff(10, i));
    cout << "Waiting for LiftOff" << endl;
  } catch(Synchronization_Exception& e) {
    cerr << e.what() << endl;
  }
} ///:~
```

LiftOff构造函数的第2个参数用于标识每个任务。当运行该程序时将会看到，由于线程被不断的换入换出，以至于不同的任务被混合在一起执行。这种交换处理是由线程调度器（scheduler）自动控制的。如果运行的计算机上有多个处理器，线程调度器会在多个处理器间“安然地分配”（quietly distribute）线程。

for循环起先似乎有一点奇怪，因为**t**在**for**循环内作为局部变量被创建，然后立刻就跳出了作用域并被销毁。这就使它显得可能会立刻失去这个线程本身，但可以从输出看到：线程的确正在运行，直到结尾。这就说明了当创建了一个**Thread**对象的时候，相关联的线程就会在线程处理系统内注册，并保持其处于活动状态。即使基于栈的**Thread**对象被丢弃，线程本身也会继续处于活动状态直到其相关联的任务完成。虽然从C++的观点上来看这也许与直觉相悖，线程的概念偏离了准则：一个线程创建一个单独执行的线程，新创建的线程在函数调用结束后，仍然能够持续执行。这种偏离反映在对象消失之后底层线程的持续执行（the persistence of the underlying thread）上。 700

11.4.1 创建有响应的用户界面

如前所述，使用线程处理的动机之一就是创建有响应的用户界面。虽然在本教材中没有包括图形用户界面，读者还是可以看到基于控制台的用户界面的简单示例。

下面的例子从一个文件中按行读取数据并把它们打印到控制台上，每行显示完成之后会休眠（sleeping）（挂起（暂停执行）当前线程）一秒钟。（稍后，在本章中将会学习到有关休眠的更多知识。）在这个过程中程序不会检查用户输入，所以用户界面是无响应的。

```
//: C11:UnresponsiveUI.cpp {RunByHand}
// Lack of threading produces an unresponsive UI.
//{L} ZThread
#include <iostream>
#include <fstream>
#include <string>
#include "zthread/Thread.h"
using namespace std;
using namespace ZThread;

int main() {
  cout << "Press <Enter> to quit:" << endl;
  ifstream file("UnresponsiveUI.cpp");
  string line;
  while(getline(file, line)) {
    cout << line << endl;
    Thread::sleep(1000); // Time in milliseconds
  }
  // Read input from the console
  cin.get();
  cout << "Shutting down..." << endl;
} ///:~
```

701

为使程序能够做出响应，可以在一个单独的线程中执行一个显示文件的任务。然后，主线程可以监视用户输入，这样程序就变成有响应的了：

```
//: C11:ResponsiveUI.cpp {RunByHand}
// Threading for a responsive user interface.
//{L} ZThread
#include <iostream>
#include <fstream>
#include <string>
#include "zthread/Thread.h"
using namespace ZThread;
using namespace std;

class DisplayTask : public Runnable {
  ifstream in;
  string line;
  bool quitFlag;
public:
  DisplayTask(const string& file) : quitFlag(false) {
    in.open(file.c_str());
  }
  ~DisplayTask() { in.close(); }
  void run() {
    while(getline(in, line) && !quitFlag) {
      cout << line << endl;
      Thread::sleep(1000);
    }
  }
  void quit() { quitFlag = true; }
};

int main() {
  try {
    cout << "Press <Enter> to quit:" << endl;
    DisplayTask* dt = new DisplayTask("ResponsiveUI.cpp");
    Thread t(dt);
    cin.get();
    dt->quit();
  } catch(Synchronization_Exception& e) {
    cerr << e.what() << endl;
  }
  cout << "Shutting down..." << endl;
} ///:~
```

702

现在**main( )**函数线程可以在按下回车<Return>键时立刻做出响应，并调用**DisplayTask**类的**quit( )**函数。

这个例子也表明了各个任务之间需要通信——**main( )**函数线程中的任务需要通知**DisplayTask**关闭。由于有一个指向**DisplayTask**的指针，你也许会认为只对那个指针调用**delete**删除它就可以终止该任务，但这样会使程序变得不可靠。这样做的问题在于：当销毁任务时，这个任务可能正在做某些重要的处理，所以就有可能使程序处于不稳定的状态。在这里，由任务自己来决定什么时候关闭是安全的。做这件事最容易的一个办法是，仅需设置一个布尔标记，简单地变更这个标记来通知任务：现在希望该任务停止下来。当该任务到达一个稳定点时会检查那个标记，然后在从**run( )**返回之前做好清理现场所需的一切工作。当任务从**run( )**返回时，**Thread**对象知道该任务已经完成。

虽然这个程序足够简单，应该不会有任何问题，但是仍然还是有一些与任务间通信有关的小缺点。这将是本章稍后所要讨论的一个重要主题。

11.4.2 使用执行器简化工作

使用ZThread的执行器（Executor）可以减少编码的工作量。执行器在客户和任务的执行之间提供了一个间接层；客户不再直接执行任务，而是由一个中间的对象来执行该任务。

在**MoreBasicThreads.cpp**中使用一个**Executor**对象而非显式创建**Thread**对象，可以表示这些操作。一个**LiftOff**对象知道如何运行一个指定的任务；就像命令模式（Command Pattern），它给出一个函数以供调用执行。一个**Executor**对象知道如何建造合适的语境来执行**Runnable**对象。在下面的例子中，**ThreadedExecutor**为每个任务创建一个线程：

```
//: c11:ThreadedExecutor.cpp
//{L} ZThread
#include <iostream>
#include "zthread/ThreadedExecutor.h"
#include "LiftOff.h"
using namespace ZThread;
using namespace std;

int main() {
  try {
    ThreadedExecutor executor;
    for(int i = 0; i < 5; i++)
      executor.execute(new LiftOff(10, i));
  } catch(Synchronization_Exception& e) {
    cerr << e.what() << endl;
  }
} ///:~
```

703

注意，在某些情况下可以用单个的**Executor**对象来创建和管理系统中的所有线程。必须把线程处理代码放在一个**try**块中，因为如果出现错误的话**Executor**的**execute( )**函数可能会抛出**Synchronization_Exception**异常。对于任何包含同步对象状态转换（启动线程、获得互斥锁（mutex）、等待某些条件等等）的函数这都是正确的，读者稍后将会在本章中学到这些内容。

一旦**Executor**中的所有任务都完成了，程序就会退出。

在前面的例子中，**ThreadedExecutor**为需要运行的每个任务都创建了一个线程，但是用一个不同类型的**Executor**对象来代替**ThreadedExecutor**对象，就可以容易的改变任务的执行方式。在本章中，使用**ThreadedExecutor**就很好了，但在产生的代码中，由于创建了太多的线程，**ThreadedExecutor**将会导致过多的开销。在这种情况下，可以使用**PoolExecutor**对象来替换**ThreadedExecutor**对象，它使用一个有限的线程集以并行的方式执行提交的任务。

```
//: C11:PoolExecutor.cpp
//{L} ZThread
#include <iostream>
#include "zthread/PoolExecutor.h"
#include "LiftOff.h"
using namespace ZThread;
using namespace std;

int main() {
  try {
    // Constructor argument is minimum number of threads:
    PoolExecutor executor(5);
    for(int i = 0; i < 5; i++)
      executor.execute(new LiftOff(10, i));
  } catch(Synchronization_Exception& e) {
```

704

```
    cerr << e.what() << endl;
  }
} ///:~
```

使用**PoolExecutor**，可以预先将开销很大的线程分配工作一次做完，在可能的时候重用这些线程。这样做会节省时间，因为不会因不断地为了每个任务都创建一个线程而付出那些开销。并且在一个事件驱动的系统中，对于一些需要由线程来处理的事件，可以以很快地方式产生。而这些快速产生的线程可以仅从线程池中取出线程的方式来提供。因为**PoolExecutor**使用的**Thread**对象数量是有限的，所以不能滥用这些可用的资源。因此，尽管本教材将会使用**ThreadedExecutor**类，在产生的代码中还是要考虑使用**PoolExecutor**类。

ConcurrentExecutor类就像是一个**PoolExecutor**类，该类有大小固定的一个线程。对于需要在另一个线程中不断运行的任何任务（长期处于活动状态的任务）来说，这个类是很有用的，例如一个监听某个信道套接字连接的任务。对于需要在线程中运行的短任务它也是很方便的，比如，更新本地或远程日志的小任务，或者为事件分派线程等。

如果有多个任务被提交至一个**ConcurrentExecutor**，每个任务都会在下一个任务开始之前执行完成，所有的任务都使用同一个线程。在下面的例子中，将会看到，每个任务按其被提交的顺序执行，并且在下一个任务开始之前执行完成。因此，一个**ConcurrentExecutor**对象串行化（顺序执行）提交给它的任务。

```
//: C11:ConcurrentExecutor.cpp
//{L} ZThread
#include <iostream>
#include "zthread/ConcurrentExecutor.h"
#include "LiftOff.h"
using namespace ZThread;
using namespace std;

int main() {
  try {
    ConcurrentExecutor executor;
    for(int i = 0; i < 5; i++)
      executor.execute(new LiftOff(10, i));
  } catch(Synchronization_Exception& e) {
    cerr << e.what() << endl;
  }
} ///:~
```

705

就像**ConcurrentExecutor**，**SynchronousExecutor**用于需要同一时刻只运行一个任务的时候，串行代替了并发。不像**ConcurrentExecutor**、**SynchronousExecutor**自己不创建或管理线程。它使用提交任务的线程，因此只会作为同步的焦点（focal point for synchronization）来行动。如果有**n**个线程向**SynchronousExecutor**提交任务，永远不会一次（同一时刻）运行两个任务，另外，每个任务运行完成后，队列里的下一个任务才会开始执行。

例如，假设现在有许多线程运行着使用文件系统的任务，但正在编写的是可移植的代码，所以不想用**flock( )**或其他特定的操作系统调用来加锁一个文件。可以在任何线程中和一个**SynchronousExecutor**一起运行这些任务，来保证在同一时刻只有一个任务在运行。这种运行方式，不需要处理共享资源上的同步问题（而且其间不会冲击文件系统）。一个较好的解决方法，就是对资源的访问采用同步方式进行（将在本章稍后学到这些内容），但是，**SynchronousExecutor**可以跳过对适当合理的某些原型事件进行协调的麻烦。

```
//: C11:SynchronousExecutor.cpp
//{L} ZThread
#include <iostream>
#include "zthread/SynchronousExecutor.h"
#include "LiftOff.h"
using namespace ZThread;
using namespace std;

int main() {
  try {
    SynchronousExecutor executor;
    for(int i = 0; i < 5; i++)
      executor.execute(new LiftOff(10, i));
  } catch(Synchronization_Exception& e) {
    cerr << e.what() << endl;
  }
} ///:~
```

当运行该程序时，将会看到任务以其被提交的顺序执行，每个任务在下一个任务启动前完成。但是看不到有新线程创建——因为在本例中，**main( )**线程是提交所有任务的线程，所以每个任务中都用到了它。因为**SynchronousExecutor**主要用于原型处理，在产生的代码中不会大量用到它。 706

11.4.3 让步

如果知道在**run( )**函数中的一次遍历循环（大多数**run( )**函数包括一个长期运行的（long-running）循环）期间已经完成了所需要做的工作，就可以给线程调度机制一个暗示，现在已经做完了该做的工作，可以让其他线程使用CPU了。这个暗示（它仅仅是个暗示——不能保证所实现的系统会监听到它）以调用**yield( )**函数的形式来表示。

下面，在每次循环后使用让步操作，可以产生一个**LiftOff**示例的修改版本。

```
//: C11:YieldingTask.cpp
// Suggesting when to switch threads with yield().
//{L} ZThread
#include <iostream>
#include "zthread/Thread.h"
#include "zthread/ThreadedExecutor.h"
using namespace ZThread;
using namespace std;

class YieldingTask : public Runnable {
  int countDown;
  int id;
public:
  YieldingTask(int ident = 0) : countDown(5), id(ident) {}
  ~YieldingTask() {
    cout << id << " completed" << endl;
  }
  friend ostream&
  operator<<(ostream& os, const YieldingTask& yt) {
    return os << "#" << yt.id << ": " << yt.countDown;
  }
  void run() {
    while(true) {
      cout << *this << endl;
      if(--countDown == 0) return;
      Thread::yield();
    }
  }
};
```

707

```
int main() {
  try {
    ThreadedExecutor executor;
    for(int i = 0; i < 5; i++)
      executor.execute(new YieldingTask(i));
  } catch(Synchronization_Exception& e) {
    cerr << e.what() << endl;
  }
} ///:~
```

可以看到，任务的**run( )**成员函数完全由一个无限循环组成。使用**yield( )**比不使用它时，程序的输出均衡了许多。可以试着注释掉**Thread::yield( )**调用，看看有何不同。然而在一般情况下，**yield( )**只在极少的情形下有用处，别想依赖它来对应用程序做出任何严谨的调整。

11.4.4 休眠

可以控制线程行为的另一种办法，就是调用函数**sleep( )**，使线程根据给定的毫秒数停止执行一段时间。在前面的例子中，如果用调用**sleep( )**而非调用**yield ( )**，就会得到下面的程序：

```
//: C11:SleepingTask.cpp
// Calling sleep() to pause for awhile.
//{L} ZThread
#include <iostream>
#include "zthread/Thread.h"
#include "zthread/ThreadedExecutor.h"
using namespace ZThread;
using namespace std;

class SleepingTask : public Runnable {
  int countDown;
  int id;
public:
  SleepingTask(int ident = 0) : countDown(5), id(ident) {}
  ~SleepingTask() {
    cout << id << " completed" << endl;
  }
  friend ostream&
  operator<<(ostream& os, const SleepingTask& st) {
    return os << "#" << st.id << ": " << st.countDown;
  }
  void run() {
    while(true) {
      try {
        cout << *this << endl;
        if(--countDown == 0) return;
        Thread::sleep(100);
      } catch(Interrupted_Exception& e) {
        cerr << e.what() << endl;
      }
    }
  }
};

int main() {
  try {
    ThreadedExecutor executor;
    for(int i = 0; i < 5; i++)
      executor.execute(new SleepingTask(i));
  } catch(Synchronization_Exception& e) {
    cerr << e.what() << endl;
  }
} ///:~
```

708

Thread::sleep( )可以抛出一个**Interrupted_Exception**异常（将在稍后学到中断的概念），可以看到这个异常在**run( )**中被捕获。但是该任务是在**main( )**函数的**try**块中创建和执行的，这个**try**块捕获**Synchronization_Exception**（所有ZThread异常的基类）异常，因此在**run( )**中可能忽略异常并假设异常会传播到**main( )**函数中的异常处理器，这种情况有可能发生吗？这种情况不会发生，因为异常不会跨越线程传播倒退回**main( )**。因此，必须对可能在任务中出现的任何局部性异常进行处理。

读者会注意到，线程倾向于以任意顺序运行，这意味着**sleep( )**也不是一个控制线程执行顺序的办法。它仅会让线程的运行停止片刻。只能保证线程休眠最少100毫秒（在本例中），但线程恢复执行前可能要花更长时间，因为在休眠间歇过期后，线程调度器还需要时间来恢复它。 709

如果必须要控制线程的执行顺序，最好的办法是使用同步控制（稍后讲述），或者在某些情况下，根本不使用线程，而是自己编写以特定的顺序相互控制的协作子例程（cooperative routine）。

11.4.5 优先权

线程的优先权（priority），向线程调度器传达了一个线程的重要性。虽然CPU以不确定的顺序运行一个线程集，但是在这些等待的线程中，线程调度器将倾向于先运行有最高优先权的等待线程。然而，这并不意味着有较低优先权的线程就不会运行（也就是说，不会因为优先权的问题发生死锁）。有较低优先权的线程只不过趋向于运行较少而已。

这里有一个修改了的**MoreBasicThreads.cpp**，可以用来演示优先权的等级。线程的优先权通过使用**Thread**的**setPriority( )**函数来进行调整。

```
//: C11:SimplePriorities.cpp
// Shows the use of thread priorities.
//{L} ZThread
#include <iostream>
#include "zthread/Thread.h"
using namespace ZThread;
using namespace std;

const double pi = 3.14159265358979323846;
const double e = 2.7182818284590452354;

class SimplePriorities : public Runnable {
  int countDown;
  volatile double d; // No optimization
  int id;
public:
  SimplePriorities(int ident=0): countDown(5), id(ident) {}
  ~SimplePriorities() {
    cout << id << " completed" << endl;
  }
  friend ostream&
  operator<<(ostream& os, const SimplePriorities& sp) {
    return os << "#" << sp.id << " priority: "
      << Thread().getPriority()
      << " count: "<< sp.countDown;
  }
  void run() {
    while(true) {
      // An expensive, interruptable operation:
      for(int i = 1; i < 100000; i++)
        d = d + (pi + e) / double(i);
      cout << *this << endl;
      if(--countDown == 0) return;
    }
```

710

```
  }
};

int main() {
  try {
    Thread high(new SimplePriorities);
    high.setPriority(High);
    for(int i = 0; i < 5; i++) {
      Thread low(new SimplePriorities(i));
      low.setPriority(Low);
    }
  } catch(Synchronization_Exception& e) {
    cerr << e.what() << endl;
  }
} ///:~
```

在这里，插入符**operator<<( )**被重载，用来显示任务的标识符、优先权，以及**countDown**值。

可以看到，线程**high**的优先权处于最高级，其他所有线程被设置为最低优先级。在本例中没有使用**Executor**，这是因为需要直接访问线程以便设置它们的优先级。

在**SimplePriorities::run( )**内部，一个开销相当大的浮点计算被重复执行了100 000次，包括**double**类型的加法和除法。变量**d**是**volatile**（可变的）的，用来确保编译器不对其进行最优化。如果没有这个计算，就不会观察到设置优先级的效果。（可以试一下：注释掉包含**double**计算的**for**循环。）有了这个计算，就可以观察到**high**线程被线程调度器赋优先选择。（至少，这是装有Windows操作系统的机器的行为。）计算要持续足够长的时间，使得线程调度机制能够介入，来改变线程和注意它们的优先权，这样就使**high**线程得到优先选择。

711

还可以使用**getPriority( )**函数获得已有线程的优先权，并且可以在任何时候（不一定非得在线程运行之前，就像在**SimplePriorities.cpp**中一样）用**setPriority( )**函数改变它的优先权。

将优先权映射到操作系统的做法是有问题的。例如，Windows 2000有7个优先级别，而Sun的Solaris 系统有2^{31}个优先级别。只有将优先级别划分成非常大的粒度才是一个接近实用的方法，就像在ZThread库中使用的**Low**、**Medium**和**High**这样的3级优先级的划分。

11.5 共享有限资源

可以认为单线程处理程序就像围绕问题空间求解的一个实体，在某一时刻只做一件事情。因为只有一个实体，根本无需考虑在同一时刻两个实体试图使用同一资源的问题：比如两个人试图在同一车位停车，或两个人同时走过同一扇门，甚至两个人同时讲话这样的问题。

有了多线程处理，可以同时做很多事情，但是现在可能有两个或更多的线程试图在同一时刻使用同一个资源。这就可能引起两种不同的问题。首先，必需的资源可能不存在。在C++中，程序员在对象的生存期内对其有完全的控制权。创建线程来使用这些对象是很容易的，这些对象在线程完成之前被销毁。

第2个问题是，两个或更多的线程在其试图同时访问同一个资源时可能会发生冲突。如果不去防止这样的冲突，就会有两个线程试图同时访问同一银行账号、在同一打印机上打印、调整同一个变量的值等等问题。

本节介绍当任务仍然在使用某个对象时，而这个对象却突然消失了的问题，以及任务发生冲突时结束共享资源的问题。读者将会学到用来解决这些问题的有关工具。

11.5.1 保证对象的存在

在C++中，对内存和资源管理是主要的关注点。在创建任何C++程序时，可以选择在栈上

或者在堆（使用**new**）上创建对象。在一个单线程处理的程序中，通常很容易保持对对象生存期的跟踪，所以不要尝试使用已经销毁的对象。 712

本章中的示例显示在堆上使用**new**创建了**Runnable**对象，但请注意这些对象从来都不是被显式删除的。然而，当运行程序时，可以从输出中看到，线程库保持跟踪每个任务并最后删除它，这是因为调用了任务的析构函数。这是在**Runnable::run( )**成员函数完成时发生的——从**run( )**返回就显示任务已经完成。

让线程来负担删除任务是个问题。因为线程不用必须知道是否有另一个线程仍然需要获得对那个**Runnable**的引用，所以可能会提早销毁该**Runnable**。为了处理这个问题，ZThread中的任务被ZThread库机制自动地进行了引用计数（reference-counted）。任务一直维持到该任务的引用计数归零，此时才能够删除该任务。这就意味着，必须总是动态删除任务，所以它们不能在栈上创建。取而代之，任务必须总是用**new**来创建，就像在本章所有例子中看到的那样。

通常必须确保非任务对象在任务需要它们的时候长期保留在活动状态。否则，容易导致那些被任务使用的对象在任务完成之前离开作用域。如果这种情况发生，任务将尝试访问非法的存储单元，并将引起程序错误。这里有一个简单的例子:

```
//: C11:Incrementer.cpp {RunByHand}
// Destroying objects while threads are still
// running will cause serious problems.
//{L} ZThread
#include <iostream>
#include "zthread/Thread.h"
#include "zthread/ThreadedExecutor.h"
using namespace ZThread;
using namespace std;

class Count {
  enum { SZ = 100 };
  int n[SZ];
public:
  void increment() {
    for(int i = 0; i < SZ; i++)
      n[i]++;
  }
};

class Incrementer : public Runnable {
  Count* count;
public:
  Incrementer(Count* c) : count(c) {}
  void run() {
    for(int n = 100; n > 0; n--) {
      Thread::sleep(250);
      count->increment();
    }
  }
};

int main() {
  cout << "This will cause a segmentation fault!" << endl;
  Count count;
  try {
    Thread t0(new Incrementer(&count));
    Thread t1(new Incrementer(&count));
  } catch(Synchronization_Exception& e) {
    cerr << e.what() << endl;
  }
} ///:~
```

713

Count类初看上去似乎有点功能过强，但是如果**n**只是一个**int**型变量（不如是一个数组），编译器会把它放在寄存器中，那个存储单元在**Count**对象离开作用域后仍保持可用（虽然从技术上来说这是不合法的）。在这种情况下发现内存违例（violation）是困难的。最终结果依赖使用的编译器和操作系统的不同而有所不同，可以试着使**n**成为一个**int**型变量看看会发生什么。在任何事件中，如果**Count**包含如上的一个**int**数组，编译器被迫要将它放在栈上而非寄存器中。

Incrementer是一个使用**Count**对象的简单任务。在**main( )**函数中，可以看到**Incrementer**任务运行了足够长的时间，**Count**对象离开了作用域，所以该任务尝试访问一个非长期存在的对象。这就会产生一个程序错误。

为了解决这个问题，必须保证在这些任务之间任何被共享的对象要长期存在，只要这些任务需要它们。（如果对象没有被共享，可以把它们直接组成到任务类中，如此一来，使它们的生存期与那个任务捆绑在一起）。既然不希望静态的程序作用域控制对象的生存期，那么就可以把对象放置在堆上。并且确保直到没有其他对象（在此情况下指任务）使用它时才被销毁，这里使用了引用计数。

714

引用计数在本教材第1卷中有过透彻的讲解，本卷中更进一步地复习它。ZThread库包括一个名叫**CountedPtr**的模板，它自动执行引用计数并在引用计数归零时用**delete**删除一个对象。这里有一个使用**CountedPtr**对上面程序进行了修改的新程序，以防发生这类错误：

```
//: C11:ReferenceCounting.cpp
// A CountedPtr prevents too-early destruction.
//{L} ZThread
#include <iostream>
#include "zthread/Thread.h"
#include "zthread/CountedPtr.h"
using namespace ZThread;
using namespace std;

class Count {
  enum { SZ = 100 };
  int n[SZ];
public:
  void increment() {
    for(int i = 0; i < SZ; i++)
      n[i]++;
  }
};

class Incrementer : public Runnable {
  CountedPtr<Count> count;
public:
  Incrementer(const CountedPtr<Count>& c ) : count(c) {}
  void run() {
    for(int n = 100; n > 0; n--) {
      Thread::sleep(250);
      count->increment();
    }
  }
};
```

715

```
int main() {
  CountedPtr<Count> count(new Count);
  try {
    Thread t0(new Incrementer(count));
    Thread t1(new Incrementer(count));
  } catch(Synchronization_Exception& e) {
    cerr << e.what() << endl;
  }
} ///:~
```

Incrementer现在包含一个**CountedPtr**对象，由它管理**Count**。在**main( )**函数中，将**CountedPtr**对象以值传递方式传递给两个**Incrementer**对象，所以调用了拷贝构造函数，引用计数增1。只要任务仍然在运行，引用计数值就为非零，所以就不会销毁**CountedPtr**管理的**Count**对象。仅当所有使用**Count**的任务都完成时，**CountedPtr**才会调用（自动地）**Count**对象上的**delete**执行删除操作。

每当有对象被多于一个任务使用时，几乎总是需要使用**CountedPtr**模板来管理那些对象，以防由对象生存期争端而产生的问题。

11.5.2 不恰当地访问资源

考虑下面的例子，其中一个任务产生偶数，另外的任务来消费这些数。在这里，消费者线程的惟一工作就是检查偶数的有效性。

首先定义消费者线程**EvenChecker**，因为它在所有后续的例子中会被重复使用。为了使**EvenChecker**与进行试验的各种类型的发生器解除耦合，将创建一个名叫**Generator**的接口，它包含最少量且必需的函数，这些有关的函数是**EvenChecker**必须知道的：它有一个可以取消的**nextValue( )**函数。

```
//: C11:EvenChecker.h
#ifndef EVENCHECKER_H
#define EVENCHECKER_H
#include <iostream>
#include "zthread/CountedPtr.h"
#include "zthread/Thread.h"
#include "zthread/Cancelable.h"
#include "zthread/ThreadedExecutor.h"

class Generator : public ZThread::Cancelable {
  bool canceled;
public:
  Generator() : canceled(false) {}
  virtual int nextValue() = 0;
  void cancel() { canceled = true; }
  bool isCanceled() { return canceled; }
};

class EvenChecker : public ZThread::Runnable {
  ZThread::CountedPtr<Generator> generator;
  int id;
public:
  EvenChecker(ZThread::CountedPtr<Generator>& g, int ident)
  : generator(g), id(ident) {}
  ~EvenChecker() {
    std::cout << "~EvenChecker " << id << std::endl;
  }
  void run() {
    while(!generator->isCanceled()) {
      int val = generator->nextValue();
      if(val % 2 != 0) {
        std::cout << val << " not even!" << std::endl;
        generator->cancel(); // Cancels all EvenCheckers
      }
    }
  }
  // Test any type of generator:
  template<typename GenType> static void test(int n = 10) {
    std::cout << "Press Control-C to exit" << std::endl;
    try {
```

716

```
      ZThread::ThreadedExecutor executor;
      ZThread::CountedPtr<Generator> gp(new GenType);
      for(int i = 0; i < n; i++)
        executor.execute(new EvenChecker(gp, i));
    } catch(ZThread::Synchronization_Exception& e) {
      std::cerr << e.what() << std::endl;
    }
  }
};
#endif // EVENCHECKER_H ///:~
```

Generator类引入了抽象类**Cancelable**，它是ZThread库的一部分。**Cancelable**的目的是提供一个一致的接口，以便通过**cancel( )**函数来改变对象的状态，用**isCanceled( )**函数来检查对象是否已被取消。在这里，使用一个简单的**bool**型取消标志，类似于以前在**ResponsiveUI.cpp**中看到的**quitFlag**。注意，本例中的类是**Cancelable**而不是**Runnable**。另外，依赖于**Cancelable**对象（**Generator**）的所有**EvenChecker** 任务都要测试这个标志，看它是否已被取消，就像在**run( )**函数中所看到的那样。这种办法，由共享公共资源（**Cancelable Generator**）的任务监视着该资源，以便根据标志来结束监视。这就消除了所谓的*竞争条件*（race condition），即两个或更多的任务竞争着响应同一个条件，因此发生冲突，否则（没有发生冲突但却）产生不一致的结果。必须仔细考虑以防所有可能会使并发系统崩溃的情形发生。例如，一个任务不能依赖于其他任务，因为不能保证任务停止的顺序。在这里，使任务依赖于非任务对象（使用**CountedPtr**引用计数），消除了潜在的竞争条件。

717

在稍后各节中，将会看到ZThread库包含与线程结束有关的更通用的机制。

既然多个**EvenChecker**对象可以结束共享一个**Generator**，所以**CountedPtr**模板用于对**Generator**对象进行引用计数。

EvenChecker类中的最后一个成员函数是一个**static**静态成员模板。该模板在**CountedPtr**内部创建一个**Generator**，设置和进行对任何类型的**Generator**对象的测试，然后启动若干个使用那个**Generator**的**EvenChecker**。如果**Generator**失败了，**test( )**将会报告它并返回；否则，必须按Control-C键来结束它。

EvenChecker 任务不断地从与其发生联系的**Generator**中读取和测试值。注意，如果**generator->isCanceled( )**为真，**run( )**函数就返回，它告诉**EvenChecker::test( )**中的**Executor**，任务已经完成。任何**EvenChecker** 任务可以在与其发生联系的**Generator**上调用**cancel( )**函数，这会导致其他所有使用**Generator**的**EvenChecker**顺畅地关闭。

EvenGenerator很简单——由**nextValue( )**产生下一个偶数值：

718

```
//: C11:EvenGenerator.cpp
// When threads collide.
//{L} ZThread
#include <iostream>
#include "EvenChecker.h"
#include "zthread/ThreadedExecutor.h"
using namespace ZThread;
using namespace std;

class EvenGenerator : public Generator {
  unsigned int currentEvenValue; // Unsigned can't overflow
public:
  EvenGenerator() { currentEvenValue = 0; }
  ~EvenGenerator() { cout << "~EvenGenerator" << endl; }
  int nextValue() {
    ++currentEvenValue; // Danger point here!
    ++currentEvenValue;
```

```
    return currentEvenValue;
  }
};

int main() {
  EvenChecker::test<EvenGenerator>();
} ///:~
```

currentEvenValue的值在第1次增1之后与第2次增1之前的这段时间，可能会有一个线程调用**nextValue( )**（代码中注释着“Danger point here!”之处），其放进变量的值会处于一个“不正确的”状态。为了证明这种情况可能会发生，**EvenChecker::test( )**创建了一组**EvenChecker**对象，不断读取一个**EvenGenerator**的输出，并测试是否每个都为偶数。如果不是，会报告出错并关闭程序。

直到**EvenGenerator**完成多次循环，这个程序可能也不会发现问题，这依赖于你使用的操作系统的特性以及其他实现细节。如果要想尽快地看到它失败，可以尝试把一个**yield( )**调用放在第1次与第2次增1操作之间。在任何事件中，当**EvenGenerator**处于“不正确”状态时，因为**EvenChecker**线程仍能够访问**EvenGenerator**里的信息，所以**EvenChecker**最终将会失败。

11.5.3 访问控制

719

前面的例子显示了使用线程时会遇到的一个基本问题：你永远不会知道一个线程何时可能运行。想像一下，你坐在桌子前拿着一把叉子，打算叉盘中最后一块食物。当叉子碰到食物时，它却突然消失了（因为你的线程被挂起，另一个用餐者进来吃掉了食物）。这就是在编写并发程序时要处理的问题。

有时候，在试图使用某一资源时，并不关心它在同一时刻是否正在被访问。但是在大多数情况下还是要关心这个问题。对于多线程处理的工作，需要一些方法来防止两个线程同时访问同一个资源，至少要防止两个线程在临界期（critical period）内访问同一资源。

防止这种冲突的一个简单方法，就是在线程正在使用一个资源时，给该资源加一把锁。访问该资源的第1个线程给资源加上锁，然后其他线程在该资源未被解锁时不能访问它。解锁的同时，另一个要使用它的线程就可以对该资源加锁并且使用它，依此类推。假设汽车的前排座位是有限的资源，那个大喊“我要坐”的小孩就类似于声明获得该锁。

因此，在某个存储单元处于不适当的状态时，需要能够防止任何其他任务访问该存储单元。也就是说，需要有一个机制，当第1个任务已经在使用某个存储单元时，该机制用来排除（exclude）第2个任务对该存储单元的访问。这个想法对所有多线程处理系统来说是基本的，它被称为*相互排斥*（mutual exclusion）；该机制被简写为*互斥*（mutex）。ZThread库包含互斥机制，这在**Mutex.h**头文件中进行了声明。

在以上程序中解决这个问题，首先要能够识别*临界区*（critical section），在临界区中必须应用相互排斥机制；然后，在进入临界区之前获得互斥锁，并在临界区的终点*释放*（release）它。在任何时刻仅有一个线程可以获得该互斥锁，因此，相互排斥完成。

```
//: C11:MutexEvenGenerator.cpp {RunByHand}
// Preventing thread collisions with mutexes.
//{L} ZThread
#include <iostream>
#include "EvenChecker.h"
#include "zthread/ThreadedExecutor.h"
#include "zthread/Mutex.h"
using namespace ZThread;
```

720

```
using namespace std;

class MutexEvenGenerator : public Generator {
  unsigned int currentEvenValue;
  Mutex lock;
public:
  MutexEvenGenerator() { currentEvenValue = 0; }
  ~MutexEvenGenerator() {
    cout << "~MutexEvenGenerator" << endl;
  }
  int nextValue() {
    lock.acquire();
    ++currentEvenValue;
    Thread::yield(); // Cause failure faster
    ++currentEvenValue;
    int rval = currentEvenValue;
    lock.release();
    return rval;
  }
};

int main() {
  EvenChecker::test<MutexEvenGenerator>();
} ///:~
```

在**MutexEvenGenerator**中增加了一个叫做**lock**的**Mutex**型变量，并且在**nextValue( )**函数中使用**acquire( )**和**release( )**创建了临界区。另外，为了在**currentEvenValue**处于奇数状态时提高语境切换的可能性，一个**Thread::yield( )**调用被插入到两个增1语句之间。因为互斥机制防止了多个线程在同一时刻出现在同一个临界区中的情况，所以不会失败。但是如果有可能发生失败，调用**yield( )**是促使失败提早发生的很有用的方法。

注意，**nextValue( )**函数必须在临界区内部获得返回值，因为如果从临界区中返回，没有释放这个锁，因此将阻止其再次从临界区获得该锁。（这通常会导致死锁（deadlock），在本章的末尾将学到有关这方面的内容。）

第1个进入**nextValue( )**的线程获得了锁，那些试图获得该锁的其他任何线程都被阻塞在那里等待，直到第1个线程释放了该锁。这时候，系统的调度机制选择另一个正在等待得到该锁的线程进入**nextValue( )**。以这种方法，在同一时刻只有一个线程能通过被互斥锁保护的代码。

721

11.5.4 使用保护简化编码

当引入异常时，互斥锁的使用就迅速变得复杂起来。为确保互斥锁总能被释放，就必须保证每条可能的异常路径都包含一个对**release( )**函数的调用。另外，任何有多条返回路径的函数都必须小心，以保证在合适的地点调用**release( )**。

利用下述事实，可以很容易地解决这些问题：基于栈的（自动）对象有一个析构函数，不管是怎样从函数的作用域中退出的，该析构函数总会被调用。在ZThread库中，这个功能以**Guard**模板的方式实现。**Guard**模板创建对象，当这些对象被创建时用**acquire( )**函数获得一个**Lockable**对象；当这些**Guard**对象被销毁时，用**release( )**函数释放该锁。**Guard**对象创建于本地栈上，不管函数是如何退出的，它都将会被自动销毁，并且总能将**Lockable**对象解锁。在这里，把上面的例子用**Guard**重新实现：

```
//: C11:GuardedEvenGenerator.cpp {RunByHand}
// Simplifying mutexes with the Guard template.
//{L} ZThread
#include <iostream>
#include "EvenChecker.h"
```

```
#include "zthread/ThreadedExecutor.h"
#include "zthread/Mutex.h"
#include "zthread/Guard.h"
using namespace ZThreao;
using namespace std;

class GuardedEvenGenerator : public Generator {
  unsigned int currentEvenValue;
  Mutex lock;
public:
  GuardedEvenGenerator() { currentEvenValue = 0; }
  ~GuardedEvenGenerator() {
    cout << "~GuardedEvenGenerator" << endl;
  }
  int nextValue() {
    Guard<Mutex> g(lock);
    ++currentEvenValue;
    Thread::yield();
    ++currentEvenValue;
    return currentEvenValue;
  }
};

int main() {
  EvenChecker::test<GuardedEvenGenerator>();
} ///:~
```

722

注意，在**nextValue( )**函数中，临时返回值不是必须的。一般情况下，要编写的代码较少，因而用户出错的机会大大减少。

Guard模板的一个有意思的特征，就是它可以被安全地用于操纵其他保护（guard）。比如，下面程序中的第2个**Guard**可以用于临时解锁一个保护：

```
//: C11:TemporaryUnlocking.cpp
// Temporarily unlocking another guard.
//{L} ZThread
#include "zthread/Thread.h"
#include "zthread/Mutex.h"
#include "zthread/Guard.h"
using namespace ZThread;

class TemporaryUnlocking {
  Mutex lock;
public:
  void f() {
    Guard<Mutex> g(lock);
    // lock is acquired
    // ...
    {
      Guard<Mutex, UnlockedScope> h(g);
      // lock is released
      // ...
      // lock is acquired
    }
    // ...
    // lock is released
  }
};

int main() {
  TemporaryUnlocking t;
  t.f();
} ///:~
```

723

Guard也可以尝试在一个确定的时间内获得某个锁，然后放弃：

```
//: C11:TimedLocking.cpp
// Limited time locking.
//{L} ZThread
#include "zthread/Thread.h"
#include "zthread/Mutex.h"
#include "zthread/Guard.h"
using namespace ZThread;

class TimedLocking {
  Mutex lock;
public:
  void f() {
    Guard<Mutex, TimedLockedScope<500> > g(lock);
    // ...
  }
};

int main() {
  TimedLocking t;
  t.f();
} ///:~
```

在这个例子中，如果在500毫秒内不能获得锁，则抛出一个**Timeout_Exception**异常。

同步整个类

ZThread库还提供了一个**GuardedClass**模板来自动地为整个类创建同步封装器(wrapper)。这意味着该类中的每个成员函数都将自动被保护。

```
//: C11:SynchronizedClass.cpp {-dmc}
//{L} ZThread
#include "zthread/GuardedClass.h"
using namespace ZThread;

class MyClass {
public:
  void func1() {}
  void func2() {}
};

int main() {
  MyClass a;
  a.func1(); // Not synchronized
  a.func2(); // Not synchronized
  GuardedClass<MyClass> b(new MyClass);
  // Synchronized calls, only one thread at a time allowed:
  b->func1();
  b->func2();
} ///:~
```

724

对象**a**是非同步的，所以**func1( )**和**func2( )**能被任意个线程在任何时刻调用。对象**b**被**GuardedClass**封装器保护了起来，所以每个成员函数都被自动同步，在任意时刻每个对象仅有一个函数能被调用。

封装器在类一级的粒度上加锁，这也许会影响到它性能。[⊖]如果一个类包含某些互不相关的函数，也许用两种不同的锁在内部同步这些函数会更好一些。然而如果这样做了，则意味着

⊖ 这可能很重要。通常函数只有小部分需要被保护。把这些保护放在函数入口点常常可以使临界区比它实际需要的要长。

该类也许包含非强相关（strongly associated）的数据组。应该考虑把这个类分解成两个类。

用一个互斥锁保护一个类的所有成员函数并不能自动保证那个类是线程安全（thread-safe）的。必须小心考虑所有的线程处理问题，以便保证线程的安全性。

11.5.5 线程本地存储

消除任务在共享资源上发生冲突问题的第2种办法是消除共享变量，对使用同一对象的各个不同线程，可以为同一个变量创建不同的存储单元。因此，如果有5个线程使用一个含有变量**x**的对象，线程本地存储（thread local storage）会自动为变量**x**产生5个不同的存储片段（单元）。幸运的是，线程本地存储的创建和管理由ZThread库的**ThreadLocal**模板自动管理，如下所示：

725

```
//: C11:ThreadLocalVariables.cpp {RunByHand}
// Automatically giving each thread its own storage.
//{L} ZThread
#include <iostream>
#include "zthread/Thread.h"
#include "zthread/Mutex.h"
#include "zthread/Guard.h"
#include "zthread/ThreadedExecutor.h"
#include "zthread/Cancelable.h"
#include "zthread/ThreadLocal.h"
#include "zthread/CountedPtr.h"
using namespace ZThread;
using namespace std;

class ThreadLocalVariables : public Cancelable {
  ThreadLocal<int> value;
  bool canceled;
  Mutex lock;
public:
  ThreadLocalVariables() : canceled(false) {
    value.set(0);
  }
  void increment() { value.set(value.get() + 1); }
  int get() { return value.get(); }
  void cancel() {
    Guard<Mutex> g(lock);
    canceled = true;
  }
  bool isCanceled() {
    Guard<Mutex> g(lock);
    return canceled;
  }
};

class Accessor : public Runnable {
  int id;
  CountedPtr<ThreadLocalVariables> tlv;
public:
  Accessor(CountedPtr<ThreadLocalVariables>& tl, int idn)
  : id(idn), tlv(tl) {}
  void run() {
    while(!tlv->isCanceled()) {
      tlv->increment();
      cout << *this << endl;
    }
  }
  friend ostream&
    operator<<(ostream& os, Accessor& a) {
    return os << "#" << a.id << ": " << a.tlv->get();
```

726

```
  }
};

int main() {
  cout << "Press <Enter> to quit" << endl;
  try {
    CountedPtr<ThreadLocalVariables>
      tlv(new ThreadLocalVariables);
    const int SZ = 5;
    ThreadedExecutor executor;
    for(int i = 0; i < SZ; i++)
      executor.execute(new Accessor(tlv, i));
    cin.get();
    tlv->cancel(); // All Accessors will quit
  } catch(Synchronization_Exception& e) {
    cerr << e.what() << endl;
  }
} ///:~
```

当通过实例化该模板来创建**ThreadLocal**对象时，只能用**get( )**和**set( )**成员函数访问该对象的内容。**get( )**函数返回一份与那个线程相关联的对象的拷贝，而**set( )**则将其参数插入到与那个线程相关的对象中存储，并返回存储单元中原来所保存的对象。可以看到，这种方法用在了**ThreadLocalVariables**里的**increment( )**和**get( )**函数中。

由于**tlv**被多个**Accessor**对象共享，它被写成像**Cancelable**一样，以便在想要停止系统运行时，让**Accessor**可以收到信号。

在运行该程序时，将看到各个线程分配有自己的存储单元的证据。

727

11.6 终止任务

在前面的例子中，读者已经看到了使用“退出标志”或**Cancelable**接口以适当的方式来终止一个任务。这是解决该问题的合理的途径。然而，在某些情形下任务却必须要突然地结束。在本节中，读者将会学到有关这样终止任务所产生的后果和存在的问题。

首先，看一个示例，这个示例不仅示范了终止任务的问题，而且也是资源共享的另一个例子。为了介绍这个例子，首先需要解决输入输出流冲突的问题。

11.6.1 防止输入/输出流冲突

读者也许已经注意到了前面例子中的输出有时候会出现信息混淆的现象。当初创建C++ 输入/输出流时并没有考虑线程处理的事情，因此没有采取什么措施阻止一个线程的输出与其他线程输出之间的冲突。所以必须编写应用程序来处理输入/输出流同步的问题。

为了解决这个问题，首先需要创建全部的输出数据信息包，然后明确决定什么时候尝试将其发送到控制台。一个简单的解决办法是将信息写入一个**ostringstream**，然后用一个带有互斥锁的对象作为所有线程的输出点，以防多个线程同时写入数据：

```
//: C11:Display.h
// Prevents ostream collisions.
#ifndef DISPLAY_H
#define DISPLAY_H
#include <iostream>
#include <sstream>
#include "zthread/Mutex.h"
#include "zthread/Guard.h"

class Display { // Share one of these among all threads
  ZThread::Mutex iolock;
```

```
public:
  void output(std::ostringstream& os) {
    ZThread::Guard<ZThread::Mutex> g(iolock);
    std::cout << os.str();
  }
};
#endif // DISPLAY_H ///:~
```

728

通过这个办法，预先定义一个标准**operator<<( )**函数，可以使用熟悉的**ostream**运算符在内存中构建对象。当一个任务需要显示输出时，它创建一个临时的**ostringstream**对象，用于构建想要的输出消息。当它调用**output( )**时，互斥锁会阻止多个线程同时向该**Display**对象写入数据。(在程序中必须只使用一个**Display**对象，正如在下面例子中将看到的。)

这恰恰表现了其基本思想，但是如果必要的话，可以构建一个更精细的框架。例如，可以把它做成一个单件（Singleton）来强迫实现一个程序中仅有一个**Display**对象的要求。(ZThread库有一个**Singleton**模板用来支持单件。)

11.6.2 举例观赏植物园

在这个模拟程序中，公园委员会想要了解每天有多少人通过这个公园的多个入口进入。每个入口有一个十字转门或其他种类的计数器，当该十字转门的计数器增1之后，一个用来表示公园中游客总数的共享计数器也增1。

```
//: C11:OrnamentalGarden.cpp {RunByHand}
//{L} ZThread
#include <vector>
#include <cstdlib>
#include <ctime>
#include "Display.h"
#include "zthread/Thread.h"
#include "zthread/FastMutex.h"
#include "zthread/Guard.h"
#include "zthread/ThreadedExecutor.h"
#include "zthread/CountedPtr.h"
using namespace ZThread;
using namespace std;

class Count : public Cancelable {
  FastMutex lock;
  int count;
  bool paused, canceled;
public:
Count() : count(0), paused(false), canceled(false) {}
int increment() {
  // Comment the following line to see counting fail:
  Guard<FastMutex> g(lock);
  int temp = count ;
  if(rand() % 2 == 0) // Yield half the time
    Thread::yield();
  return (count  = ++temp);
}
int value() {
  Guard<FastMutex> g(lock);
  return count;
}
void cancel() {
  Guard<FastMutex> g(lock);
  canceled = true;
}
bool isCanceled() {
```

729

```
    Guard<FastMutex> g(lock);
    return canceled;
  }
  void pause() {
    Guard<FastMutex> g(lock);
    paused = true;
  }
  bool isPaused() {
    Guard<FastMutex> g(lock);
    return paused;
  }
};

class Entrance : public Runnable {
  CountedPtr<Count> count;
  CountedPtr<Display> display;
  int number;
  int id;
  bool waitingForCancel;
public:
  Entrance(CountedPtr<Count>& cnt,
    CountedPtr<Display>& disp, int idn)
  : count(cnt), display(disp), number(0), id(idn),
    waitingForCancel(false) {}
  void run() {
    while(!count->isPaused()) {
      ++number;
      {
        ostringstream os;
        os << *this << " Total: "
           << count->increment() << endl;
        display->output(os);
      }
      Thread::sleep(100);
    }
    waitingForCancel = true;
    while(!count->isCanceled()) // Hold here...
      Thread::sleep(100);
    ostringstream os;
    os << "Terminating " << *this << endl;
    display->output(os);
  }
  int getValue() {
    while(count->isPaused() && !waitingForCancel)
      Thread::sleep(100);
    return number;
  }
  friend ostream&
  operator<<(ostream& os, const Entrance& e) {
    return os << "Entrance " << e.id << ": " << e.number;
  }
};

int main() {
  srand(time(0)); // Seed the random number generator
  cout << "Press <ENTER> to quit" << endl;
  CountedPtr<Count> count(new Count);
  vector<Entrance*> v;
  CountedPtr<Display> display(new Display);
  const int SZ = 5;
  try {
    ThreadedExecutor executor;
    for(int i = 0; i < SZ; i++) {
```

730

```
      Entrance* task = new Entrance(count, display, i);
      executor.execute(task);
      // Save the pointer to the task:
      v.push_back(task);
    }
    cin.get(); // Wait for user to press <Enter>
    count->pause(); // Causes tasks to stop counting
    int sum = 0;
    vector<Entrance*>::iterator it = v.begin();
    while(it != v.end()) {
      sum += (*it)->getValue();
      ++it;
    }
    ostringstream os;
    os << "Total: " << count->value() << endl
       << "Sum of Entrances: " << sum << endl;
    display->output(os);
    count->cancel(); // Causes threads to quit
  } catch(Synchronization_Exception& e) {
    cerr << e.what() << endl;
  }
} ///:~
```

731

Count是一个类，它是用来保存公园游客数的主计数器。单个**Count**对象在**main( )**中定义为**count**，同时**count**被作为一个**CountedPtr**实例保存在**Entrance**中，因此被所有**Entrance**对象共享。本例中，使用一个叫**lock**的**FastMutex**模板实例而非普通的**Mutex**，因为**FastMutex**使用本地操作系统的互斥锁并因此产生许多有趣的结果。

在**increment( )**函数中，一个使用**lock**对象的**Guard**对象用来同步对**count**的访问。在大约一半时间，这个函数使用**rand( )**来引发**yield( )**，在这中间取来**count**的数据放入**temp**，并且使**temp**增1，再把**temp**存回到**count**之中。因为这个原因，如果注释掉**Guard**对象的定义，很快就会看到程序崩溃，因为多线程将会同时对**count**进行访问和修改。

Entrance类也持有一个本地的**number**，用来记录已通过这个特定入口的游客数。这里提供了对**count**对象的双重检验，以确保所记录的游客数正确。**Entrance::run( )**仅使**number**变量和**count**对象增1，并休眠100毫秒。

在主函数中，一个**vector<Entrance*>**用于装载已经创建的每个**Entrance**。用户按下**<Enter>**键之后，该**vector**用来迭代所有的个体**Entrance**值并计算其总和。

这个程序在运行时遇到相当少的额外麻烦时，就会以一种稳定的方式关闭所有的对象。编写这个程序的部分原因是为了说明在结束多线程处理程序的执行时需要多么谨慎，还有部分原因是为了示范**interrupt( )**函数的值，读者不久就会学到这些。

Entrance对象间发生的所有通信都要通过一个**Count**对象。当用户按下<Enter>键时，**main( )**函数用**pause( )**发送消息给**count**。由于每个**Entrance::run( )**都在监视着**count**对象是否暂停下来，这将引发每个**Entrance**对象迁移到**waitingForCancel**等待状态，在这种状态下它将不再计数，但仍然处于活动状态。这是必要的，因为**main( )**必须能安全迭代（遍历）在**vector<Entrance*>**中的每个对象。注意，因为在一个**Entrance**完成计数并迁移至**waitingForCancel**等待状态之前，发生迭代的可能性很小（可以忽略），所以函数**getValue( )**循环调用**sleep( )**直到对象迁移至**waitingForCancel**等待状态。（这是被称为忙等待（busy wait）的形式之一，是不受欢迎的。稍后会在本章中看到首选的解决办法，它使用了**wait( )**函数。）一旦**main( )**完成了对**vector<Entrance*>**的一次遍历迭代，**cancel( )**消息就会被送至**count**对象。再强调一次，所有**Entrance**对象都会监视这个状态变化。在这点

732

上，它们打印一条终止消息并从**run( )**中退出，这导致每个任务都会被线程处理机制销毁掉。

当程序运行时，将看到总的计数和当一个游客走过十字转门时每个入口的计数显示。如果注释掉**Count::increment( )**中的**Guard**对象，读者就会注意到游客总数不再是预期的值了。每个十字转门所统计的游客数都与**count**中的值不同。只要互斥锁**Mutex**在那里同步对**Counter**的访问，一切就会正常进行。切记：在这里，**Count::increment( )**使用**temp**和**yield( )**函数放大了失败的潜在可能。在实际的线程处理问题中，从统计学上来说失败的可能性很小，所以读者会很容易陷入相信不会有什么问题会发生的陷阱。就像在上面的例子中，可能会有一些隐藏的问题并没有在这个程序里发生，所以在对并发程序的代码进行复审时应格外仔细。

原子操作

注意，**Count::value( )**使用一个**Guard**对象进行同步并返回**count**的值。这就提出一个有趣的观点，因为这段代码不用同步大概也可以在大部分编译器和操作系统上良好运行。其理由就是，在一般情况下一个简单的操作比如返回一个**int**型变量就是一个*原子操作*（atomic operation），这意味着或许它在一个微处理器指令中完成而不会被中断。（多线程处理机制不能在一个微处理器指令中间停止一个线程。）也就是说，原子操作不能被线程处理机制中断，因此不需要被保护。[⊖] 实际上，如果删除取**count**的值送到**temp**的操作，并且删除**yield( )**函数，代之以仅直接**count**增1操作，这样或许不需要进行互斥加锁处理也会工作的很好，因为增1操作通常也是原子操作。[⊜]

733

问题在于C++标准并不能保证任何这类操作的原子性。虽然诸如像返回一个**int**型值的操作，和对一个**int**型的值进行增1的操作在大多数计算机上几乎确定地是原子的，但是这并没有保证。正因为没有保证，所以必须考虑最坏的情况。有时可能要调查特定计算机（经常要通过阅读汇编语言）上的原子行为并根据这种假设编写代码。那总归是危险的、欠谨慎的做法。以上相关的信息很容易丢失或者被隐藏，其他人可能会认为这段代码可以被移植到其他机器上，当移植后就会发疯般地追踪由线程冲突而引发的偶然的错误。

所以，虽然从**Count::value( )**上删除保护似乎可以照常工作，但并不是无懈可击的，因此可能会在某些机器上看到偏离常轨的行为。

11.6.3 阻塞时终止

前面例子中的**Entrance::run( )**在主循环中包含一个**sleep( )**调用。我们知道在那个例子中**sleep( )**休眠最后会被唤醒，在任务到达循环的顶部时检查**isPaused( )**的状态，便有机会跳出循环。然而，**sleep( )**仅是一个线程在其执行过程中被阻塞的一种情况，有时必须终止一个被阻塞的任务。

734

1. *线程状态*

一个线程可以处于以下4种状态之一：

1) *新建*（New）状态：一个线程只是在被创建的瞬间暂时地保持这个状态。它分配任何必需的系统资源并完成初始化。在这一点它有资格获得CPU时间。线程调度器随后将把该线程转

⊖ 这样说过于简单。有时甚至当它看上去好像是一个原子操作且会是安全的时候它却可能不是，所以当决定不使用同步时必须非常小心。删除用于同步的代码通常是过度优化的一个标志——它会导致我们陷入很多麻烦中而且不会得到更多好处，甚至得不到任何东西。

⊜ 原子性不是惟一的问题。在多处理器系统上，可见性问题比在单处理器上多得多。一个线程所做的改变，即使它们在不能被中断的意义上来说是原子的，对其他线程来说仍然有可能是不可见的（比如：这些改变会被暂时存储在本地处理器缓存中），所以不同的线程会看见应用程序的不同状态。同步机制迫使一个线程做出的改变在多处理器系统上是跨应用程序可见的，然而不使用同步，这些变化何时会变为可见是不确定的。

换到可运行或阻塞状态。

2) 可运行（Runnable）状态：这个状态意味着当时间分片机制为该线程分配可利用的CPU周期时，线程就可以运行。因此，在任何时刻，某个线程可能运行也可能不运行，但是如果线程调度器安排它，则没什么事情会阻止其运行；这时，它既不处于死亡状态，也不处于阻塞状态。

3) 阻塞（Blocked）状态：线程可以运行了，但有某种事件阻止了它的运行。（比如，它也许正在等待I/O操作完成。）当一个线程处于阻塞状态时，线程调度器会忽略该线程并且不分配给它任何CPU时间。直到线程重新进入可运行状态之前，它不执行任何操作。

4) 死亡（Dead）状态：一个处于死亡状态的线程，不能再被调度也不能获得任何CPU时间。它的任务已经完成，不再是可运行的。使一个线程消逝的正常的办法就是让它从**run( )**函数返回。

2. 变为阻塞状态

当一个线程不能继续运行时它就处在阻塞状态。一个线程变为阻塞状态的原因如下：

- 调用**sleep(milliseconds)**使线程进入休眠状态，在这种情况下该线程在指定时间内不会运行。
- 已经使用**wait( )**挂起了该线程的运行。在得到**signal( )**或**broadcast( )**消息之前它不会再一次变为可执行状态。我们在后面的小节里将检验这些问题。
- 线程正在等待某个I/O操作完成。
- 线程正在尝试进入一段被一个互斥锁保护的代码块，而那个互斥锁已经被其他线程获得。

735

现在需要注意的问题是：有时需要在某个线程处于阻塞状态时终止它。线程在执行到代码中的某一点上能自己检查状态值并决定结束运行，如果不能等待线程到达代码中的这一点，那么就必须强迫线程脱离阻塞状态。

11.6.4 中断

正如想像的那样，在一个**Runnable::run( )**函数的中间跳出，会比等待函数到达**isCanceled( )**函数的检查点（或者程序员准备离开函数的其他地方）时跳出显得更加混乱。当从被阻塞的任务中离开时，可能需要销毁与之相关的对象并清理有关的资源。正因为这样，在一个任务的**run( )**中间跳出更像是抛出一个异常，所以在ZThread库中，异常被用于此类退出。（这样处于不适当使用异常的边缘，因为这意味着经常把异常用于控制流。）[⊖] 为了在以此方式结束一个任务时能返回到一个已知的正确状态，要谨慎地考虑代码的执行路径，在**catch**子句中正确清除所有的东西。在本节，读者会看到就这些问题的介绍。

为了终止一个阻塞的线程，ZThread库提供了**Thread::interrupt( )**函数。这个函数用来为那类线程设置中断状态（interrupted status）。一个使用了中断状态设置的线程，如果已经被阻塞或尝试进行阻塞操作时将会抛出一个**Interrupted_Exception**异常。当异常被抛出或者假如任务调用了**Thread::interrupted( )**时，中断状态将重新设置。正如读者所见，**Thread::interrupted( )**提供了不用抛出异常而离开**run( )**函数中循环的第2条途径。

这里的例子显示了**interrupt( )**的基本功能：

736

```
//: C11:Interrupting.cpp
// Interrupting a blocked thread.
//{L} ZThread
#include <iostream>
```

⊖ 无论如何，在ZThread中异常绝不会被异步发送。因此退出中间指令或函数调用不会有危险。并且只要我们使用**Guard**模板获得互斥锁，那么如果抛出异常的话，互斥锁会被自动释放。

```
#include "zthread/Thread.h"
using namespace ZThread;
using namespace std;

class Blocked : public Runnable {
public:
  void run() {
    try {
      Thread::sleep(1000);
      cout << "Waiting for get() in run():";
      cin.get();
    } catch(Interrupted_Exception&) {
      cout << "Caught Interrupted_Exception" << endl;
      // Exit the task
    }
  }
};

int main(int argc, char* argv[]) {
  try {
    Thread t(new Blocked);
    if(argc > 1)
      Thread::sleep(1100);
    t.interrupt();
  } catch(Synchronization_Exception& e) {
    cerr << e.what() << endl;
  }
} ///:~
```

可以看到，除了将插入数据到**cout**之外，阻塞还能发生在**run( )**函数中包含的其他两个地点：即对**Thread::sleep(1000)**和**cin.get( )**的调用。可给程序传递任何命令行参数，可以通知**main( )**休眠足够长的时间，以便任务到时候能结束它的**sleep( )**和调用**cin.get( )**。[⊖] 如果不给程序传递参数，就会跳过**main( )**中的**sleep( )**。在这里，在任务休眠时发生了对函数**interrupt( )**的调用。读者将会看到，这将导致**Interrupted_Exception**异常被抛出。如果给程序一个命令行参数，就会发现如果一个任务被阻塞在IO操作上，它不能被中断。也就是说，除了IO操作，一个任务可以从任何阻塞操作中中断出来。[⊜]

737

如果正在创建一个执行IO操作的线程，这还是让人有点困惑，因为这意味着I/O有使多线程处理程序死锁的潜在可能性。问题在于，再次强调，在设计思想上C++没有被设计成使用线程处理；恰恰相反，它假装线程处理并不存在。因此，输入输出流库不是线程友好（thread-friendly）的。如果新的C++标准决定增加对线程的支持，输入输出流库也许需要重新考虑其处理方法。

1. 被一个互斥锁阻塞

如果试图调用一个函数，而该函数的互斥锁已经被别的线程获得了，那么这个调用该函数的任务就会被挂起，直到该互斥锁变成可获得时为止。下面的例子测试了这种阻塞是否可被中断。

```
//: C11:Interrupting2.cpp
// Interrupting a thread blocked
// with a synchronization guard.
//{L} ZThread
#include <iostream>
#include "zthread/Thread.h"
#include "zthread/Mutex.h"
```

⊖ 实际上，**sleep( )**只提供最小的延迟，不是保证延迟，所以可能（尽管不可思议）**sleep(1100)**会在**sleep(1000)**之前被唤醒。

⊜ C++标准中没有说明在IO操作期间中断不能出现。然而大多数实现不支持它。

```
#include "zthread/Guard.h"
using namespace ZThread;
using namespace std;

class BlockedMutex {
  Mutex lock;
public:
  BlockedMutex() {
    lock.acquire();
  }
  void f() {
    Guard<Mutex> g(lock);
    // This will never be available
  }
};

class Blocked2 : public Runnable {
  BlockedMutex blocked;
public:
  void run() {
    try {
      cout << "Waiting for f() in BlockedMutex" << endl;
      blocked.f();
    } catch(Interrupted_Exception& e) {
      cerr << e.what() << endl;
      // Exit the task
    }
  }
};

int main(int argc, char* argv[]) {
  try {
    Thread t(new Blocked2);
    t.interrupt();
  } catch(Synchronization_Exception& e) {
    cerr << e.what() << endl;
  }
} ///:~
```

738

BlockedMutex类有一个构造函数，它获得对象自己的互斥锁**Mutex**并且绝不释放它。由于这个原因，如果试图调用**f( )**，总会被阻塞，因为该互斥锁**Mutex**不能被获得。在**Blocked2**中，**run( )**函数将因此停止在对**blocked.f( )**的调用上。当运行程序时就会看到，和IO流的调用不同，**interrupt( )**能够跳出已被一个互斥锁阻塞的调用。[⊖]

2. 中断检查

注意，当在一个线程上调用**interrupt( )**时，中断仅发生在任务进入一个阻塞操作的那一时刻，或者已经在一个阻塞操作内（正如你所知道的，假如在IO的情况下，就会陷在里面）。但是，编写什么样的代码，才能使是否产生这样的阻塞调用依赖于它的运行条件呢？如果只能通过在一个被阻塞的调用上抛出异常来退出，也许始终不能离开**run( )**循环。因此，假如调用**interrupt( )**来停止一个任务，如果在**run( )**循环没有发生任何阻塞调用，该任务就需要另外的机会来退出。

739

中断状态（interrupted status）提供了这样的机会，它通过调用**interrupt( )**进行设置。而调用**interrupted( )**来检查中断状态，这不仅能告知**interrupt( )**是否已经被调用，它也会清除中断状态。清除中断状态可以确保整个架构不会两次通知正被中断的任务。它会用一个

⊖ 注意，尽管不太可能，对**t.interrupt( )**的调用实际可以发生在对**blocked.f( )**的调用之前。

Interrupted_Exception异常或者一个成功的**Thread:: interrupted( )**测试来通知使用者。如果想再次检查是否被中断了，在调用**Thread::interrupted( )**时可以把测试结果存储起来。

下面的例子显示了当设置了中断状态时，**run( )**函数中在处理阻塞和非阻塞两种可能性的情况下所要用到的典型的习语：

```
//: C11:Interrupting3.cpp {RunByHand}
// General idiom for interrupting a task.
//{L} ZThread
#include <iostream>
#include "zthread/Thread.h"
using namespace ZThread;
using namespace std;

const double PI = 3.14159265358979323846;
const double E = 2.7182818284590452354;

class NeedsCleanup {
  int id;
public:
  NeedsCleanup(int ident) : id(ident) {
    cout << "NeedsCleanup " << id << endl;
  }
  ~NeedsCleanup() {
    cout << "~NeedsCleanup " << id << endl;
  }
};
class Blocked3 : public Runnable {
  volatile double d;
public:
  Blocked3() : d(0.0) {}
  void run() {
    try {
      while(!Thread::interrupted()) {
        point1:
        NeedsCleanup n1(1);
        cout << "Sleeping" << endl;
        Thread::sleep(1000);
        point2:
        NeedsCleanup n2(2);
        cout << "Calculating" << endl;
        // A time-consuming, non-blocking operation:
        for(int i = 1; i < 100000; i++)
          d = d + (PI + E) / (double)i;
      }
      cout << "Exiting via while() test" << endl;
    } catch(Interrupted_Exception&) {
      cout << "Exiting via Interrupted_Exception" << endl;
    }
  }
};

int main(int argc, char* argv[]) {
  if(argc != 2) {
    cerr << "usage: " << argv[0]
      << " delay-in-milliseconds" << endl;
    exit(1);
  }
  int delay = atoi(argv[1]);
  try {
    Thread t(new Blocked3);
    Thread::sleep(delay);
```

```
    t.interrupt();
  } catch(Synchronization_Exception& e) {
    cerr << e.what() << endl;
  }
} ///:~
```

如果采用抛出异常来离开循环，**NeedsCleanup**类强调了对相关资源进行正确清理的必要性。注意，在**Blocked3::run( )**中没有定义指针，那是为了异常的安全，所有的资源必须封装在基于栈的对象中，以便异常处理器可以调用析构函数来自动清理它们。 741

必须在调用**interrupt( )**之前给程序传递一个命令行参数，此参数为用毫秒表示的延迟时间。使用不同的延迟，能从循环中不同的地点退出**Blocked3::run( )**函数：从正处于阻塞状态的**sleep( )**调用中退出，以及从非阻塞状态的数学计算中退出。可以看到，如果**interrupt( )**在标签**point2**后被调用（非阻塞操作期间）。首先循环已经完成，其次所有的本地对象被析构，最后循环经由**while**语句在顶部退出。然而，如果**interrupt( )**在**point1**和**point2**之间被调用（在**while**语句之后，但是在阻塞操作**sleep( )**之前或之中），任务通过**Interrupted_Exception**异常退出。在这种情况下，只有在异常被抛出的位置之前已经被创建完成的栈对象才会被清理，并且有机会在**catch**子句中完成其他的清理操作。

设计用来响应**interrupt( )**函数的类必须建立一种策略，以便保证其能保持一致的状态。这通常意味着，所有的资源获取都要封装在基于栈的对象中，以便无论**run( )**循环如何退出，对象的析构函数都会被调用。如果正确地做了，像这样的代码一定是优雅的。在没有向对象接口中加入任何特别的函数的情况下，可以创建出完全封装了其同步机制，但仍能对外部激励（通过**interrupt( )**）有响应的组件。

11.7 线程间协作

正如读者所看到的，当使用线程在同一时刻运行多个任务时，可以使用互斥锁来同步两个任务的行为的方法，来阻止一个任务干扰另一个任务的资源。也就是说，如果两个任务对一个共享资源（通常是内存）相互争夺，就要使用互斥锁来保证在同一时刻只允许一个任务访问那个资源。

在这个问题解决之后，可以继续考虑线程间协作的问题，以便多个线程能一起工作来共同解决某个问题。现在问题不在于线程之间的彼此干扰，而在于其和谐工作，由于问题的某一部分必须在另外一部分能被解决之前解决完毕。这更像是一个工程进度表：必须先挖房屋的地基，但是钢结构构件的铺设和混凝土构件可以并行建造，这些任务都必须在混凝土基础浇注之前完成。管道设备必须在混凝土平板浇注好之前放置好，而混凝土平板要在开始搭建框架结构之前安置，等等。这些任务中有些可以并行进行，但是某些步骤则要求在完成其他所有任务之后才能继续进行。 742

这些任务协作时的关键问题是这些任务间的“握手”。为完成这个握手过程，使用相同的基础：互斥机制，互斥机制在这种情况下可以保证只有一个任务响应信号。这就消除了任何可能的竞争条件。要熟练掌握互斥锁，这里为任务增加了一个方法，让它把自己挂起来，直到某些外部状态发生改变（例如“管道设备现在已经就位”），表明此时任务可以向前进行。在本节中，读者会看到任务间的握手问题，在握手期间会出现的问题，以及这些问题相应的解决方法。

11.7.1 等待和信号

在Zthread库中，使用互斥锁并允许任务挂起的基类是**Condition**，可以通过在条件**Condition**上调用**wait( )**挂起一个任务。当外部状态发生改变时，这种改变也许意味着某个

任务应该继续进行处理。调用信号函数**signal( )**可以通知该任务而唤醒它，或者调用**broadcast( )**，而唤醒所有在那个**Condition**对象上被挂起的任务。

wait( )有两种形式。第1种形式接受一个毫秒数作为参数，这个参数与**sleep( )**函数中的参数有相同的含义："在这段时间暂停"。**wait( )**的第2种形式不要参数；这种形式更常见。这两种形式的**wait( )**都会释放被**Condition**对象所控制的互斥锁 **Mutex**，并且会挂起线程直到**Condition**对象收到一个**signal( )**或者**broadcast( )**。如果超时，第1种形式在接收到**signal( )**或**broadcast( )**之前也可以结束。

因为**wait( )**会释放**Mutex**，这意味着该**Mutex**可以被其他线程获得。因此，当调用**wait( )**时，就相当于说："现在已经做完了该做的所有事情，我将在此等待，但是我希望如果其他同步操作可以执行的允许它们执行。"

743 典型的情况是，当在等待某个条件的改变时就使用**wait( )**，而这个条件的改变在当前函数之外的因素控制之下进行。(这些条件常常会被另一个线程改变。)在线程内检测条件时，你不希望进行空循环等待；这就是所谓的"忙等待"，而"忙等待"通常会大量占用CPU周期。因此，**wait( )**在等待外部条件变化时挂起线程，只在**signal( )**或**broadcast( )**被调用时(暗示某些相关事件已经发生)，唤醒线程并检测发生的变化。因此，**wait( )** 为同步线程之间的活动提供了一种方法。

下面看一个简单的例子。**WaxOMatic.cpp**有两个进程：一个进程给**Car**上蜡，另一个进程给**Car**抛光。抛光进程在上蜡进程完成前不能进行其工作，并且上蜡进程在汽车可以再穿上另一个蜡外套之前必须等待直到抛光进程完成。**WaxOn**和**WaxOff**都使用了**Car**对象，这个**Car**对象包含了一个用于挂起一个在**waitForWaxing( )**或**waitForBuffing( )**内的线程的**Condition**。

```
//: C11:WaxOMatic.cpp {RunByHand}
// Basic thread cooperation.
//{L} ZThread
#include <iostream>
#include <string>
#include "zthread/Thread.h"
#include "zthread/Mutex.h"
#include "zthread/Guard.h"
#include "zthread/Condition.h"
#include "zthread/ThreadedExecutor.h"
using namespace ZThread;
using namespace std;

class Car {
  Mutex lock;
  Condition condition;
  bool waxOn;
public:
  Car() : condition(lock), waxOn(false) {}
  void waxed() {
    Guard<Mutex> g(lock);
    waxOn = true; // Ready to buff
    condition.signal();
  }
  void buffed() {
    Guard<Mutex> g(lock);
    waxOn = false; // Ready for another coat of wax
    condition.signal();
  }
  void waitForWaxing() {
```

744

```
    Guard<Mutex> g(lock);
    while(waxOn == false)
      condition.wait();
  }
  void waitForBuffing() {
    Guard<Mutex> g(lock);
    while(waxOn == true)
      condition.wait();
  }
};

class WaxOn : public Runnable {
  CountedPtr<Car> car;
public:
  WaxOn(CountedPtr<Car>& c) : car(c) {}
  void run() {
    try {
      while(!Thread::interrupted()) {
        cout << "Wax On!" << endl;
        Thread::sleep(200);
        car->waxed();
        car->waitForBuffing();
      }
    } catch(Interrupted_Exception&) { /* Exit */ }
    cout << "Ending Wax On process" << endl;
  }
};

class WaxOff : public Runnable {
  CountedPtr<Car> car;
public:
  WaxOff(CountedPtr<Car>& c) : car(c) {}
  void run() {
    try {
      while(!Thread::interrupted()) {
        car->waitForWaxing();
        cout << "Wax Off!" << endl;
        Thread::sleep(200);
        car->buffed();
      }
    } catch(Interrupted_Exception&) { /* Exit */ }
    cout << "Ending Wax Off process" << endl;
  }
};

int main() {
  cout << "Press <Enter> to quit" << endl;
  try {
    CountedPtr<Car> car(new Car);
    ThreadedExecutor executor;
    executor.execute(new WaxOff(car));
    executor.execute(new WaxOn(car));
    cin.get();
    executor.interrupt();
  } catch(Synchronization_Exception& e) {
    cerr << e.what() << endl;
  }
} ///:~
```

745

在**Car**的构造函数中，一个互斥锁**Mutex**被封装于**Condition**对象中，以便**Mutex**可以用于管理任务间的通信。然而，**Condition**对象不包含有关进程状态的信息，所以还要管理另外的信息用来指出进程的状态。在这里，**Car**有一个**bool waxOn**，这个布尔变量指出上蜡、

抛光进程的状态。

在**waitForWaxing( )**中检查**waxOn**标志，如果它是**false**则调用中的线程通过调用**Condition**对象上的**wait( )**被挂起。重要的是，这发生在一个被保护的子句中，在这个子句中该线程获得了互斥锁（在这里，是通过创建一个**Guard**对象获得的）。当调用**wait( )**时，该线程被挂起并释放互斥锁。释放互斥锁是必要的，因为为了安全地改变对象的状态（比如，把**waxOn**的值变为**true**，为了使被挂起的线程继续进行下去，这是必须做的），互斥锁必须能够被一些其他任务获得。在本例中，当其他线程调用**waxed( )**来告知被挂起的线程该去做某个工作的时候，互斥锁必须能获得以便把**waxOn**的值变为**true**。然后，**waxed( )**向**Condition**对象发送一个信号**signal( )**，由它来唤醒那个在调用**wait( )**中被挂起的线程。虽然可以在一个被保护的子句中调用信号**signal( )**——就像这里一样——但并不要求这样做。[⊖]

746

为了从等待**wait( )**中唤醒一个线程，必须首先重新获得其在进入**wait( )**时释放的互斥锁。直到互斥锁变成可用之前，该线程不会被唤醒。

wait( )的调用被置于一个**while**循环内部，用这个循环来检查相关的条件。这很重要，基于以下两个原因：[⊖]

- 很可能当某个线程得到一个信号**signal( )**时，其他一些条件可能已经改变了，但这些条件在这里与调用**wait( )**的原因无关。如果有这种情况，该线程在其相关的条件改变之前将再一次被挂起。
- 在该线程从其**wait( )**函数中醒来之时，可能另外某个任务改变了一些条件，因此这个线程就不能或者没兴趣在此时执行其操作了。再次强调，它应再次调用**wait( )**而被重新挂起。

因为这两个原因在调用**wait( )**时总会出现，故总要编写在**while**循环内调用**wait( )**的一段程序来测试与线程相关的条件。

747

WaxOn::run( )代表给汽车上蜡进程中的第1步，所以它执行其操作（调用**sleep( )**来模拟上蜡所需要的时间）。然后它告知汽车上蜡完毕，并调用**waitForBuffing( )**，该函数使用**wait( )**挂起线程，直到**WaxOff**进程为汽车调用**buffed( )**，改变状态并调用**notify( )**。另一方面，**WaxOff::run( )**立即迁移到**waitForWaxing( )**，并因此被挂起直到由**WaxOn**将上蜡工作完成并且**waxed( )**被调用。当运行此程序时可以看到，将控制权在两个线程之间来回传递，从而使这两步进程交替重复执行。当按下回车（<Enter>）键时，**interrupt( )**停止这两个线程的运行——当为一个执行器对象**Executor**调用**interrupt( )**时，它为其控制下的所有线程调用**interrupt( )**。

11.7.2 生产者-消费者关系

线程处理问题中的一个常见的情形是生产者-消费者（producer-consumer）关系，一个任务创建对象而另一个任务消费这些对象。在这种情况下，要确定（在其他事件中）进行消费的任务不会意外遗漏掉已产生的任何对象。

为了说明该问题，考虑一个有3个任务的机器：一个任务是制作烤面包，一个任务是给烤

⊖ 这与Java相反，在Java中必须持有锁才能调用**notify( )**（Java 版的**signal( )**）。尽管Posix 线程不要求必须持有锁才能调用**signal( )**或**broadcast( )**，但是这种做法是推荐的做法。ZThread库松散基于Posix 线程。

⊖ 在一些平台上有第3种办法跳出**wait( )**，即所谓伪唤醒（spurious wakeup）。一个伪唤醒本质上意味着一个线程过早地停止了阻塞（当等待一个条件变量或信号量时）而没有被**signal( )**或**broadcast( )**激活。线程就像是自己醒过来一样。伪唤醒存在的原因是，在某些平台上实现POSIX线程或类似的东西，并不像它在某些平台上那样直截了当。对这些平台来说，允许伪唤醒能够简化建立类似pthread库的工作。ZThread中不存在伪唤醒，因为该库弥补并对用户隐藏了这些问题。

面包抹黄油，还有一个任务是往抹好黄油的烤面包上抹果酱。

```
//: C11:ToastOMatic.cpp {RunByHand}
// Problems with thread cooperation.
//{L} ZThread
#include <iostream>
#include <cstdlib>
#include <ctime>
#include "zthread/Thread.h"
#include "zthread/Mutex.h"
#include "zthread/Guard.h"
#include "zthread/Condition.h"
#include "zthread/ThreadedExecutor.h"
using namespace ZThread;
using namespace std;

// Apply jam to buttered toast:
class Jammer : public Runnable {
  Mutex lock;
  Condition butteredToastReady;
  bool gotButteredToast;
  int jammed;
public:
  Jammer() : butteredToastReady(lock) {
    gotButteredToast = false;
    jammed = 0;
  }
  void moreButteredToastReady() {
    Guard<Mutex> g(lock);
    gotButteredToast = true;
    butteredToastReady.signal();
  }
  void run() {
    try {
      while(!Thread::interrupted()) {
        {
          Guard<Mutex> g(lock);
          while(!gotButteredToast)
            butteredToastReady.wait();
          ++jammed;
        }
        cout << "Putting jam on toast " << jammed << endl;
        {
          Guard<Mutex> g(lock);
          gotButteredToast = false;
        }
      }
    } catch(Interrupted_Exception&) { /* Exit */ }
    cout << "Jammer off" << endl;
  }
};

// Apply butter to toast:
class Butterer : public Runnable {
  Mutex lock;
  Condition toastReady;
  CountedPtr<Jammer> jammer;
  bool gotToast;
  int buttered;
public:
  Butterer(CountedPtr<Jammer>& j)
  : toastReady(lock), jammer(j) {
    gotToast = false;
```

748

```
    buttered = 0;
  }
  void moreToastReady() {
749 Guard<Mutex> g(lock);
    gotToast = true;
    toastReady.signal();
  }
  void run() {
    try {
      while(!Thread::interrupted()) {
        {
          Guard<Mutex> g(lock);
          while(!gotToast)
            toastReady.wait();
          ++buttered;
        }
        cout << "Buttering toast " << buttered << endl;
        jammer->moreButteredToastReady();
        {
          Guard<Mutex> g(lock);
          gotToast = false;
        }
      }
    } catch(Interrupted_Exception&) { /* Exit */ }
    cout << "Butterer off" << endl;
  }
};

class Toaster : public Runnable {
  CountedPtr<Butterer> butterer;
  int toasted;
public:
  Toaster(CountedPtr<Butterer>& b) : butterer(b) {
    toasted = 0;
  }
  void run() {
    try {
      while(!Thread::interrupted()) {
        Thread::sleep(rand()/(RAND_MAX/5)*100);
        // ...
        // Create new toast
        // ...
        cout << "New toast " << ++toasted << endl;
        butterer->moreToastReady();
      }
    } catch(Interrupted_Exception&) { /* Exit */ }
    cout << "Toaster off" << endl;
  }
750 };

  int main() {
    srand(time(0)); // Seed the random number generator
    try {
      cout << "Press <Return> to quit" << endl;
      CountedPtr<Jammer> jammer(new Jammer);
      CountedPtr<Butterer> butterer(new Butterer(jammer));
      ThreadedExecutor executor;
      executor.execute(new Toaster(butterer));
      executor.execute(butterer);
      executor.execute(jammer);
      cin.get();
      executor.interrupt();
```

```
  } catch(Synchronization_Exception& e) {
    cerr << e.what() << endl;
  }
} ///:~
```

这些类以逆序定义，这样做的目的是简化前向引用（forward-referencing）的操作问题。

Jammer和**Butterer**都包含一个**Mutex**对象、一个**Condition**对象和一些内部状态信息。通过改变这些内部状态信息的状态，来指出进程要被挂起或恢复执行。（**Toaster**不需要这些，因为它是生产者，无需等待任何事情。）两个**run( )**函数都执行同一个操作，设置一个状态标志，然后调用**wait( )**来挂起任务。**moreToastReady( )**和**moreButteredToastReady( )**函数改变各自的状态标志，以指示某些事情已经发生了改变，进程现在要考虑恢复执行，然后调用信号**signal( )**唤醒该线程。

本例与前面例子的不同之处在于，至少从概念上讲，这里生产了一些东西：烤面包。烤面包的生产率有点随机化，这就增加了不确定性。读者将会看到，在运行程序时可能会出错，因为会有许多片烤面包掉在地板上——没抹黄油，也没抹果酱。

11.7.3 用队列解决线程处理的问题

线程处理问题常常基于需要对任务进行串行化上——也就是说，要使事情有序地进行处理。**ToastOMatic.cpp**不仅必要注意让事情有序，还必须能够加工好烤面包片，而且在此期间不用担心它会掉到地板上。使用队列可以采用同步的方式访问其内部元素，这样就可以解决很多线程处理问题：

751

```
//: C11:TQueue.h
#ifndef TQUEUE_H
#define TQUEUE_H
#include <deque>
#include "zthread/Thread.h"
#include "zthread/Condition.h"
#include "zthread/Mutex.h"
#include "zthread/Guard.h"

template<class T> class TQueue {
  ZThread::Mutex lock;
  ZThread::Condition cond;
  std::deque<T> data;
public:
  TQueue() : cond(lock) {}
  void put(T item) {
    ZThread::Guard<ZThread::Mutex> g(lock);
    data.push_back(item);
    cond.signal();
  }
  T get() {
    ZThread::Guard<ZThread::Mutex> g(lock);
    while(data.empty())
      cond.wait();
    T returnVal = data.front();
    data.pop_front();
    return returnVal;
  }
};
#endif // TQUEUE_H ///:~
```

这是通过在标准C++库的双端队列**deque**上添加下面的内容建立起来的：

1) 加入同步以确保在同一时刻不会有两个线程添加对象。

2) 加入**wait( )**和**signal( )**以便在队列为空时让消费者线程自动挂起，并在有多个元素可用时恢复执行。

752 这些相对量较少的代码能解决为数可观的问题。[⊖]

这里有个简单的测试程序，将对**LiftOff**对象的串行化执行进行测试。消费者是**LiftOffRunner**，它把每个**LiftOff**对象从**TQueue**中取出来并直接执行。（也就是说，它通过显式调用**run( )**来使用自己的线程，而不是为每个任务启动一个新线程。）

```
//: C11:TestTQueue.cpp {RunByHand}
//{L} ZThread
#include <string>
#include <iostream>
#include "TQueue.h"
#include "zthread/Thread.h"
#include "LiftOff.h"
using namespace ZThread;
using namespace std;

class LiftOffRunner : public Runnable {
  TQueue<LiftOff*> rockets;
public:
  void add(LiftOff* lo) { rockets.put(lo); }
  void run() {
    try {
      while(!Thread::interrupted()) {
        LiftOff* rocket = rockets.get();
        rocket->run();
      }
    } catch(Interrupted_Exception&) { /* Exit */ }
    cout << "Exiting LiftOffRunner" << endl;
  }
};

int main() {
  try {
    LiftOffRunner* lor = new LiftOffRunner;
    Thread t(lor);
    for(int i = 0; i < 5; i++)
      lor->add(new LiftOff(10, i));
    cin.get();
    lor->add(new LiftOff(10, 99));
    cin.get();
    t.interrupt();
  } catch(Synchronization_Exception& e) {
    cerr << e.what() << endl;
  }
} ///:~
```

753

任务被**main( )**函数置于**TQueue**队列上，被**LiftOffRunner**从**TQueue**队列上取走。注意，**LiftOffRunner**可以忽略同步问题，因为这些问题由**TQueue**来解决。

适当的进行烘烤

为解决**ToastOMatic.cpp**中存在的问题，我们可以在加工进程期间使用**TQueue**管理烤面包。为了做到这点，需要实际的烤面包对象，它们保持并显示了其状态：

⊖ 注意，如果读者由于某些原因停止了读，写者将继续写入直到系统内存用完。如果这是用户的程序存在的一个问题，用户可以添加所允许的最大元素计数，队列满时写者线程将被阻塞。

```
//: C11:ToastOMaticMarkII.cpp {RunByHand}
// Solving the problems using TQueues.
//{L} ZThread
#include <iostream>
#include <string>
#include <cstdlib>
#include <ctime>
#include "zthread/Thread.h"
#include "zthread/Mutex.h"
#include "zthread/Guard.h"
#include "zthread/Condition.h"
#include "zthread/ThreadedExecutor.h"
#include "TQueue.h"
using namespace ZThread;
using namespace std;

class Toast {
  enum Status { DRY, BUTTERED, JAMMED };
  Status status;
  int id;
public:
  Toast(int idn) : status(DRY), id(idn) {}
  #ifdef __DMC__ // Incorrectly requires default
  Toast() { assert(0); } // Should never be called
  #endif
  void butter() { status = BUTTERED; }
  void jam() { status = JAMMED; }
  string getStatus() const {
    switch(status) {
      case DRY: return "dry";
      case BUTTERED: return "buttered";
      case JAMMED: return "jammed";
      default: return "error";
    }
  }
  int getId() { return id; }
  friend ostream& operator<<(ostream& os, const Toast& t) {
    return os << "Toast " << t.id << ": " << t.getStatus();
  }
};

typedef CountedPtr< TQueue<Toast> > ToastQueue;

class Toaster : public Runnable {
  ToastQueue toastQueue;
  int count;
public:
  Toaster(ToastQueue& tq) : toastQueue(tq), count(0) {}
  void run() {
    try {
      while(!Thread::interrupted()) {
        int delay = rand()/(RAND_MAX/5)*100;
        Thread::sleep(delay);
        // Make toast
        Toast t(count++);
        cout << t << endl;
        // Insert into queue
        toastQueue->put(t);
      }
    } catch(Interrupted_Exception&) { /* Exit */ }
    cout << "Toaster off" << endl;
  }
};
```

754

```
// Apply butter to toast:
class Butterer : public Runnable {
  ToastQueue dryQueue, butteredQueue;
public:
  Butterer(ToastQueue& dry, ToastQueue& buttered)
  : dryQueue(dry), butteredQueue(buttered) {}
  void run() {
    try {
      while(!Thread::interrupted()) {
        // Blocks until next piece of toast is available:
        Toast t = dryQueue->get();
        t.butter();
        cout << t << endl;
        butteredQueue->put(t);
      }
    } catch(Interrupted_Exception&) { /* Exit */ }
    cout << "Butterer off" << endl;
  }
};

// Apply jam to buttered toast:
class Jammer : public Runnable {
  ToastQueue butteredQueue, finishedQueue;
public:
  Jammer(ToastQueue& buttered, ToastQueue& finished)
  : butteredQueue(buttered), finishedQueue(finished) {}
  void run() {
    try {
      while(!Thread::interrupted()) {
        // Blocks until next piece of toast is available:
        Toast t = butteredQueue->get();
        t.jam();
        cout << t << endl;
        finishedQueue->put(t);
      }
    } catch(Interrupted_Exception&) { /* Exit */ }
    cout << "Jammer off" << endl;
  }
};

// Consume the toast:
class Eater : public Runnable {
  ToastQueue finishedQueue;
  int counter;
public:
  Eater(ToastQueue& finished)
  : finishedQueue(finished), counter(0) {}
  void run() {
    try {
      while(!Thread::interrupted()) {
        // Blocks until next piece of toast is available:
        Toast t = finishedQueue->get();
        // Verify that the toast is coming in order,
        // and that all pieces are getting jammed:
        if(t.getId() != counter++ ||
           t.getStatus() != "jammed") {
          cout << ">>>> Error: " << t << endl;
          exit(1);
        } else
          cout << "Chomp! " << t << endl;
      }
    } catch(Interrupted_Exception&) { /* Exit */ }
    cout << "Eater off" << endl;
```

```
  }
};

int main() {
  srand(time(0)); // Seed the random number generator
  try {
    ToastQueue dryQueue(new TQueue<Toast>),
               butteredQueue(new TQueue<Toast>),
               finishedQueue(new TQueue<Toast>);
    cout << "Press <Return> to quit" << endl;
    ThreadedExecutor executor;
    executor.execute(new Toaster(dryQueue));
    executor.execute(new Butterer(dryQueue,butteredQueue));
    executor.execute(
      new Jammer(butteredQueue, finishedQueue));
    executor.execute(new Eater(finishedQueue));
    cin.get();
    executor.interrupt();
  } catch(Synchronization_Exception& e) {
    cerr << e.what() << endl;
  }
} ///:~
```

在这个解决方案中，两件事情会马上变得很明显：第一，在每个**Runnable**类中代码的数量和复杂性通过队列**TQueue**的使用会显著减少，因为进行保护、通信，以及**wait( ) / signal( )**操作现在都由**TQueue**来维护。**Runnable**类不再拥有任何**Mutex**或**Condition**对象。第二，类之间的耦合被消除了，因为每个类只与它的**TQueue**通信。注意，现在类的定义次序是独立的。较少的代码和较少的耦合总归是一件好事，这暗示着在这里**TQueue**的使用有积极作用，就像在大多数问题中它所做的那样。 757

11.7.4 广播

signal( )函数唤醒了一个正在等待**Condition**对象的线程。然而，也许会有多个线程在等待某个相同的条件对象，在这种情况下需要使用**broadcast( )**而不是**signal( )**把这些线程都唤醒。

现在考虑一个假想的制造汽车的机器人流水线，作为一个例子它集中体现了本章中的许多概念。每辆**Car**将在几个阶段内装配完成，在本例中将看到这样一个阶段：底盘制造好后，在这段时间里安装附属的发动机、驱动传动装置（drive train）和车轮。通过一个**CarQueue**将**Car**从一个地方传送到另一个地方，**CarQueue**是一个**TQueue**的类型。一个**Director**从进来的**CarQueue**队列中取出每辆**Car**（作为一个未加工的底盘）并放置在一个**Cradle**中，所有工作都在这里完成。在这个地方，主管**Director**通知所有正在等待的机器人（使用广播**broadcast( )**），**Car**已经在**Cradle**中准备就绪，机器人们可以进行装配工作了。三种类型的机器人开始进行工作，当它们完成任务时给**Cradle**发送一个消息。**Director**一直等到所有任务都完成后，把**Car**放到出去的**CarQueue**队列上传送到下一个工序。在这里，出去的**CarQueue**队列的消费者是个**Reporter**对象，它仅打印该**Car**，说明那个任务已正确地完成了。

```
//: C11:CarBuilder.cpp {RunByHand}
// How broadcast() works.
//{L} ZThread
#include <iostream>
#include <string>
#include "zthread/Thread.h"
#include "zthread/Mutex.h"
#include "zthread/Guard.h"
#include "zthread/Condition.h"
#include "zthread/ThreadedExecutor.h"
```

```
#include "TQueue.h"
using namespace ZThread;
using namespace std;

class Car {
  int id;
758  bool engine, driveTrain, wheels;
public:
  Car(int idn) : id(idn), engine(false),
  driveTrain(false), wheels(false) {}
  // Empty Car object:
  Car() : id(-1), engine(false),
  driveTrain(false), wheels(false) {}
  // Unsynchronized -- assumes atomic bool operations:
  int getId() { return id; }
  void addEngine() { engine = true; }
  bool engineInstalled() { return engine; }
  void addDriveTrain() { driveTrain = true; }
  bool driveTrainInstalled() { return driveTrain; }
  void addWheels() { wheels = true; }
  bool wheelsInstalled() { return wheels; }
  friend ostream& operator<<(ostream& os, const Car& c) {
    return os << "Car " << c.id << " ["
      << " engine: " << c.engine
      << " driveTrain: " << c.driveTrain
      << " wheels: " << c.wheels << " ]";
  }
};

typedef CountedPtr< TQueue<Car> > CarQueue;

class ChassisBuilder : public Runnable {
  CarQueue carQueue;
  int counter;
public:
  ChassisBuilder(CarQueue& cq) : carQueue(cq),counter(0) {}
  void run() {
    try {
      while(!Thread::interrupted()) {
        Thread::sleep(1000);
        // Make chassis:
        Car c(counter++);
        cout << c << endl;
        // Insert into queue
        carQueue->put(c);
      }
    } catch(Interrupted_Exception&) { /* Exit */ }
    cout << "ChassisBuilder off" << endl;
  }
};
759 class Cradle {
    Car c; // Holds current car being worked on
    bool occupied;
    Mutex workLock, readyLock;
    Condition workCondition, readyCondition;
    bool engineBotHired, wheelBotHired, driveTrainBotHired;
  public:
    Cradle()
    : workCondition(workLock), readyCondition(readyLock) {
      occupied = false;
      engineBotHired = true;
      wheelBotHired = true;
      driveTrainBotHired = true;
```

```
  }
  void insertCar(Car chassis) {
    c = chassis;
    occupied = true;
  }
  Car getCar() { // Can only extract car once
    if(!occupied) {
      cerr << "No Car in Cradle for getCar()" << endl;
      return Car(); // "Null" Car object
    }
    occupied = false;
    return c;
  }
  // Access car while in cradle:
  Car* operator->() { return &c; }
  // Allow robots to offer services to this cradle:
  void offerEngineBotServices() {
    Guard<Mutex> g(workLock);
    while(engineBotHired)
      workCondition.wait();
    engineBotHired = true; // Accept the job
  }
  void offerWheelBotServices() {
    Guard<Mutex> g(workLock);
    while(wheelBotHired)
      workCondition.wait();
    wheelBotHired = true; // Accept the job
  }
  void offerDriveTrainBotServices() {
    Guard<Mutex> g(workLock);
    while(driveTrainBotHired)
      workCondition.wait();
    driveTrainBotHired = true; // Accept the job
  }
  // Tell waiting robots that work is ready:
  void startWork() {
    Guard<Mutex> g(workLock);
    engineBotHired = false;
    wheelBotHired = false;
    driveTrainBotHired = false;
    workCondition.broadcast();
  }
  // Each robot reports when their job is done:
  void taskFinished() {
    Guard<Mutex> g(readyLock);
    readyCondition.signal();
  }
  // Director waits until all jobs are done:
  void waitUntilWorkFinished() {
    Guard<Mutex> g(readyLock);
    while(!(c.engineInstalled() && c.driveTrainInstalled()
            && c.wheelsInstalled()))
      readyCondition.wait();
  }
};

typedef CountedPtr<Cradle> CradlePtr;

class Director : public Runnable {
  CarQueue chassisQueue, finishingQueue;
  CradlePtr cradle;
public:
  Director(CarQueue& cq, CarQueue& fq, CradlePtr cr)
```

760

```
  : chassisQueue(cq), finishingQueue(fq), cradle(cr) {}
  void run() {
    try {
      while(!Thread::interrupted()) {
        // Blocks until chassis is available:
        cradle->insertCar(chassisQueue->get());
        // Notify robots car is ready for work
        cradle->startWork();
        // Wait until work completes
        cradle->waitUntilWorkFinished();
        // Put car into queue for further work
        finishingQueue->put(cradle->getCar());
      }
    } catch(Interrupted_Exception&) { /* Exit */ }
761   cout << "Director off" << endl;
    }
  };

  class EngineRobot : public Runnable {
    CradlePtr cradle;
  public:
    EngineRobot(CradlePtr cr) : cradle(cr) {}
    void run() {
      try {
        while(!Thread::interrupted()) {
          // Blocks until job is offered/accepted:
          cradle->offerEngineBotServices();
          cout << "Installing engine" << endl;
          (*cradle)->addEngine();
          cradle->taskFinished();
        }
      } catch(Interrupted_Exception&) { /* Exit */ }
      cout << "EngineRobot off" << endl;
    }
  };

  class DriveTrainRobot : public Runnable {
    CradlePtr cradle;
  public:
    DriveTrainRobot(CradlePtr cr) : cradle(cr) {}
    void run() {
      try {
        while(!Thread::interrupted()) {
          // Blocks until job is offered/accepted:
          cradle->offerDriveTrainBotServices();
          cout << "Installing DriveTrain" << endl;
          (*cradle)->addDriveTrain();
          cradle->taskFinished();
        }
      } catch(Interrupted_Exception&) { /* Exit */ }
      cout << "DriveTrainRobot off" << endl;
    }
  };

  class WheelRobot : public Runnable {
    CradlePtr cradle;
  public:
    WheelRobot(CradlePtr cr) : cradle(cr) {}
    void run() {
762   try {
        while(!Thread::interrupted()) {
          // Blocks until job is offered/accepted:
          cradle->offerWheelBotServices();
```

```
        cout << "Installing Wheels" << endl;
        (*cradle)->addWheels();
        cradle->taskFinished();
      }
    } catch(Interrupted_Exception&) { /* Exit */ }
    cout << "WheelRobot off" << endl;
  }
};

class Reporter : public Runnable {
  CarQueue carQueue;
public:
  Reporter(CarQueue& cq) : carQueue(cq) {}
  void run() {
    try {
      while(!Thread::interrupted()) {
        cout << carQueue->get() << endl;
      }
    } catch(Interrupted_Exception&) { /* Exit */ }
    cout << "Reporter off" << endl;
  }
};

int main() {
  cout << "Press <Enter> to quit" << endl;
  try {
    CarQueue chassisQueue(new TQueue<Car>),
             finishingQueue(new TQueue<Car>);
    CradlePtr cradle(new Cradle);
    ThreadedExecutor assemblyLine;
    assemblyLine.execute(new EngineRobot(cradle));
    assemblyLine.execute(new DriveTrainRobot(cradle));
    assemblyLine.execute(new WheelRobot(cradle));
    assemblyLine.execute(
      new Director(chassisQueue, finishingQueue, cradle));
    assemblyLine.execute(new Reporter(finishingQueue));
    // Start everything running by producing chassis:
    assemblyLine.execute(new ChassisBuilder(chassisQueue));
    cin.get();
    assemblyLine.interrupt();
  } catch(Synchronization_Exception& e) {
    cerr << e.what() << endl;
  }
} ///:~
```

763

注意，**Car**走了一个捷径：它假设布尔操作是原子的，就像以前讨论过的那样，有时候这是一个安全的假定，但需要周密考虑。[⊖]每个**Car**从一个未加工的底盘开始，不同的机器人给它装配上不同的部分，当它们去做这件事时要调用适当的“add”函数。

ChassisBuilder只是每秒钟创建一个新的**Car**，把它放进**chassisQueue**队列中。**Director**通过把下一个**Car**从**chassisQueue**队列中取出，把它放入**Cradle**，而通知所有机器人去**startWork( )**，并通过调用**waitUntilWorkFinished( )**挂起自己等一系列操作来管理装配进程。当工作完成时，**Director**把**Car**从**Cradle**中取出并放入**finishingQueue**。

Cradle是发送信号操作的关键。互斥锁**Mutex**和**Condition**条件对象控制着两件事情：机器人进行的工作和显示所有的操作是否已经完成。一个特定类型的机器人能够通过调用与其类型相适应的“提供”函数将其服务提供给**Cradle**。在这个地方，机器人线程被挂起，直到

⊖ 详细说明，请参考本章先前关于多处理器和可见性的注释。

Director调用开始工作函数**startWork( )**，它改变雇佣标志（hiring flag）并调用**broadcast( )**来通知所有机器人出来工作。虽然这个系统允许任意数量的机器人提供服务，但每个机器人为做这些工作需要挂起自己的线程。可以想像一个更复杂的系统，在该系统中各种机器人在不同的**Cradle**里面注册自己，并没有被注册进程挂起。然后将它们存在一个对象池中，等待第1个需要完成某种任务的**Cradle**。

每个机器人完成了它的任务（改变进程中**Car**的状态）后，它就调用**taskFinished( )**，此函数向**readyCondition**发送一个信号**signal( )**，而这正是**Director**在**waitUntilWorkFinished( )**函数中所等待的。每次主管（director）线程醒来，都会检查**Car**的状态。如果它仍然未完成，这个线程会被再次挂起。

764 当**Director**将一个**Car**插入到**Cradle**中时，可以通过运算符**operator->( )**在**Car**上执行操作。为了防止多次提取同一辆汽车，用一个标志引发产生一个错误报告。（在ZThread库中异常不能跨线程传播。）

在**main( )**函数中，随着**ChassisBuilder**开始持续启动进程，所有必需的对象都被创建并且所有的任务都被初始化。（然而，因为**TQueue**的行为，如果它先启动也没关系。）注意，这个程序遵循了本章给出的所有关于对象和任务生存周期的指南，故停止进程是安全的。

11.8 死锁

因为线程可以变为阻塞，且因为对象可以拥有互斥锁，这些锁能够阻止线程在锁被释放之前访问这个对象。所以就有可能出现这种情况，某个线程在等待另一个线程而第2个线程又在等待别的线程，以此类推，直到这个链上的最后一个线程回过头来等待第1个线程。这样就会得到一个由互相等待的线程构成的连续的循环，而使任何线程都不能运行。这称为*死锁*（deadlock）。

如果试图运行一个程序，它立刻就死锁了，人们马上就会知道程序出了问题，并且可以跟踪程序的执行过程来找到问题所在。真正的问题在于，这个程序似乎运行良好，但却隐藏着死锁的可能性。在这种情况下，死锁可能发生但事先却得没有任何征兆，所以它潜伏在程序里，直到客户发现它出其不意地发生了。（并且这个死锁的过程可能很难重复显现。）因此，仔细设计程序来预防死锁是开发并发程序的一个关键的要素。

现在看一个由Edsger Dijkstra虚构的有关死锁的经典问题：*哲学家聚餐*（dining philosopher）问题。该问题的基本描述指定了5个哲学家（不过这里的例子允许任意数目的哲学家）。这些哲学家将花费他们的部分时间用来进行思考，花费其余部分时间进餐。当他们进行思考时，不需要任何共享资源，但是在他们进餐时使用有限数量的餐具。在原始的问题描述中，餐具就是叉子，每个人需要用两个叉子从桌子中央的碗里取意大利面条，但似乎将餐具说成是筷子更合理。显然，只要每个哲学家进餐就需要两根筷子。

765 该问题引入了一个难点：作为哲学家，他们只有很少的钱，所以只买得起5根筷子。哲学家们围坐在桌子周围，哲学家与哲学家之间放一根筷子。当一个哲学家想进餐时，他必须同时得到他左边的那根筷子和右边的那根筷子。如果该哲学家的旁边（左边或右边）有人正在使用他所需要的筷子，则我们的这个哲学家就必须等待，直到所需要的筷子变成可用的。

```
//: C11:DiningPhilosophers.h
// Classes for Dining Philosophers.
#ifndef DININGPHILOSOPHERS_H
#define DININGPHILOSOPHERS_H
#include <string>
#include <iostream>
#include <cstdlib>
```

```
#include "zthread/Condition.h"
#include "zthread/Guard.h"
#include "zthread/Mutex.h"
#include "zthread/Thread.h"
#include "Display.h"

class Chopstick {
  ZThread::Mutex lock;
  ZThread::Condition notTaken;
  bool taken;
public:
  Chopstick() : notTaken(lock), taken(false) {}
  void take() {
    ZThread::Guard<ZThread::Mutex> g(lock);
    while(taken)
      notTaken.wait();
    taken = true;
  }
  void drop() {
    ZThread::Guard<ZThread::Mutex> g(lock);
    taken = false;
    notTaken.signal();
  }
};

class Philosopher : public ZThread::Runnable {
  Chopstick& left;
  Chopstick& right;
  int id;
  int ponderFactor;
  ZThread::CountedPtr<Display> display;
  int randSleepTime() {
    if(ponderFactor == 0) return 0;
    return rand()/(RAND_MAX/ponderFactor) * 250;
  }
  void output(std::string s) {
    std::ostringstream os;
    os << *this << " " << s << std::endl;
    display->output(os);
  }
public:
  Philosopher(Chopstick& l, Chopstick& r,
  ZThread::CountedPtr<Display>& disp, int ident,int ponder)
  : left(l), right(r), id(ident), ponderFactor(ponder),
    display(disp) {}
  virtual void run() {
    try {
      while(!ZThread::Thread::interrupted()) {
        output("thinking");
        ZThread::Thread::sleep(randSleepTime());
        // Hungry
        output("grabbing right");
        right.take();
        output("grabbing left");
        left.take();
        output("eating");
        ZThread::Thread::sleep(randSleepTime());
        right.drop();
        left.drop();
      }
    } catch(ZThread::Synchronization_Exception& e) {
      output(e.what());
    }
```

766

```
  }
  friend std::ostream&
  operator<<(std::ostream& os, const Philosopher& p) {
    return os << "Philosopher " << p.id;
  }
};
#endif // DININGPHILOSOPHERS_H ///:~
```

两个哲学家**Philosopher**不可以同时用**take( )**拿同一根筷子**Chopstick**，因为**take( )**用一个互斥锁**Mutex**进行同步。另外，如果筷子已经被一个**Philosopher**占用，另一个**Philosopher**可以在可用条件**available Condition**上用**wait( )**等待，直到**Chopstick**的当前持有者调用**drop( )**放下筷子（这也必须同步以防竞争条件，并且保证多处理器系统中的内存可见性）使**Chopstick**变为可用的。

767

每个**Philosopher**持有其左边和右边**Chopstick**对象的引用，所以可以尝试拿起它们。**Philosopher**的目的是用部分时间进行思考，用另一部分时间进餐，在**main( )**函数中就是这样表达的。然而读者会注意到，如果**Philosopher**花很少的时间进行思考，当他们试图进餐时都来对**Chopstick**进行竞争，死锁就会更快地发生。所以，可以这样试验一下，思考算子**ponderFactor**用于衡量一个**Philosopher**花费在思考和进餐上的时间长度趋势。一个较小的**ponderFactor**将增加死锁的可能性。

在**Philosopher::run( )**中，每个**Philosopher**仅仅不断地重复进行思考和进餐。可以看到**Philosopher**用于思考的时间量是随机的，然后试图用**take( )**拿右边的**Chopstick**，再用**take( )**拿左边的**Chopstick**，用于进餐的时间量亦是随机的，然后再次重复这样做。对输出到控制台的操作进行同步，就像在本章中较早时见到的一样。

这个问题很有趣，因为它显示了一个程序可能表面上看起来运行正确，但事实上有死锁的倾向。为了演示这一点，可以使用命令行参数调整因子来影响进餐哲学家的总数及每个哲学家花费思考的时间。如果有许多哲学家或者他们花费很多时间进行思考，那么，虽然有死锁的可能性，但也许永远也看不到死锁。值为0的命令行参数趋向于使死锁尽快发生。[⊖]

768

```
//: C11:DeadlockingDiningPhilosophers.cpp {RunByHand}
// Dining Philosophers with Deadlock.
//{L} ZThread
#include <ctime>
#include "DiningPhilosophers.h"
#include "zthread/ThreadedExecutor.h"
using namespace ZThread;
using namespace std;

int main(int argc, char* argv[]) {
  srand(time(0)); // Seed the random number generator
  int ponder = argc > 1 ? atoi(argv[1]) : 5;
  cout << "Press <ENTER> to quit" << endl;
  enum { SZ = 5 };
  try {
    CountedPtr<Display> d(new Display);
    ThreadedExecutor executor;
    Chopstick c[SZ];
    for(int i = 0; i < SZ; i++) {
```

⊖ 在写本章内容的时候，Cygwin（www.cygwin.com）正在进行调整变化，改善对它的线程处理的支持。利用Cygwin的这个可利用版本下的程序，我们仍然不能观察死锁行为。举个例子，该程序在Linux系统下很快地就会死锁。

```
      executor.execute(
        new Philosopher(c[i], c[(i+1) % SZ], d, i,ponder));
    }
    cin.get();
    executor.interrupt();
    executor.wait();
  } catch(Synchronization_Exception& e) {
    cerr << e.what() << endl;
  }
} ///:~
```

注意，**Chopstick**对象不需要内部标识符；而是通过其在数组**c**中的位置来识别它们。在构造**Chopstick**对象时，赋予每个**Philosopher**一个左边和一个右边的**Chopstick**对象的引用；这些是在**Philosopher**可以进餐之前必须获取的餐具。除最后一个**Philosopher**外，将**Philosopher**的座位安排在贴近的一双**Chopstick**对象之间来初始化每个**Philosopher**。最后一个**Philosopher**的右边**Chopstick**是顺序号为第0根**Chopstick**，这样就完成了环绕桌子的座位摆放。因为最后一个**Philosopher**就座的右边紧挨着第1个**Philosopher**，他们俩共享第0根筷子。这样的安排可能在某一时间点上所有的哲学家同时试图进餐，并且等待紧挨着他们的哲学家放下筷子，这样程序将会死锁。

如果这些线程（哲学家）花费在其他任务（思考）上的时间越多于花费在进餐上的时间，则他们需要共享资源（筷子）时发生冲突的可能性就会越低，因此可以使人相信，这个程序是没有死锁的（使用非0的**ponder**值），尽管事实上它可能会死锁。

为了修正这个问题，必须明白如果同时满足以下4种条件，死锁就会发生：

769

1) 相互排斥。线程使用的资源至少有一个必须是不可共享的。在这种情况下，一根筷子一次就只能被一个哲学家使用。

2) 至少有一个进程必须持有某一种资源，并且同时等待获得正在被另外的进程所持有的资源。也就是说，要发生死锁一个哲学家必须持有一根筷子并且等待另一根筷子。

3) 不能以抢占的方式剥夺一个进程的资源。所有进程只能把释放资源作为一个正常事件。我们的哲学家是有礼貌的，他们不会从别的哲学家手中抢夺筷子。

4) 出现一个循环等待，一个进程等待另外的进程所持有的资源，而这个被等待的进程又等待另一个进程所持有的资源，以此类推直到某个进程去等待被第1个进程所持有的资源。因此，头尾相接环环相扣，因此大家都被锁住了。在**DeadlockingDiningPhilosophers.cpp**中，是因为每个哲学家都试图先得到右边的筷子，而后再得到左边的筷子，所以发生了循环等待。

因为必须所有这些条件都满足才会引发死锁，那么只需阻止其中一个条件发生就可防止产生死锁。在这个程序中，防止死锁最容易的办法是破坏条件四。这个条件发生的原因是由于每个哲学家都试图以特定的顺序拿筷子：先右后左。正因为如此，才可能进入这样的情形：每个人都把持着其右边的筷子，而等待得到其左边的筷子，由此导致循环等待条件产生。然而，如果最后一个哲学家被初始化为先尝试拿左边的筷子，然后再拿右边的筷子，那么该哲学家将永远无法阻止右边紧挨着的哲学家拿到他自己左边的筷子。在这种情形下，就防止了循环等待。这只是问题的一种解决方法，读者也可以通过阻止其他条件发生来解决该问题（更具体的细节请参考论述高级的线程处理的书籍）：

```
//: C11:FixedDiningPhilosophers.cpp {RunByHand}
// Dining Philosophers without Deadlock.
//{L} ZThread
#include <ctime>
#include "DiningPhilosophers.h"
#include "zthread/ThreadedExecutor.h"
```

770

```
using namespace ZThread;
using namespace std;

int main(int argc, char* argv[]) {
  srand(time(0)); // Seed the random number generator
  int ponder = argc > 1 ? atoi(argv[1]) : 5;
  cout << "Press <ENTER> to quit" << endl;
  enum { SZ = 5 };
  try {
    CountedPtr<Display> d(new Display);
    ThreadedExecutor executor;
    Chopstick c[SZ];
    for(int i = 0; i < SZ; i++) {
      if(i < (SZ-1))
        executor.execute(
          new Philosopher(c[i], c[i + 1], d, i, ponder));
      else
        executor.execute(
          new Philosopher(c[0], c[i], d, i, ponder));
    }
    cin.get();
    executor.interrupt();
    executor.wait();
  } catch(Synchronization_Exception& e) {
    cerr << e.what() << endl;
  }
} ///:~
```

通过确保最后一个哲学家在拿起和放下其右边筷子之前先拿起和放下其左边筷子，就可以消除死锁，程序将会流畅地运行。

没有编程语言上的支持可以帮助防止死锁；这取决于你是否能通过谨慎的设计来避免死锁。这对于那些正试图调试一个发生死锁程序的人来说并不是什么值得安慰的消息。

11.9 小结

本章的目标是给读者提供一个采用线程进行并发编程的基础：

771

1) 可以运行多个独立的任务。

2) 当这些任务关闭时，必须全面地考虑所有可能的问题。在任务完成之前，它们使用的对象或其他任务可能会消失。

3) 任务在彼此争夺共享资源时会产生冲突。互斥锁是用来防止这些冲突的基本工具。

4) 如果不谨慎设计的话，任务可能死锁。

然而，有很多其他有关线程处理方面的工具，可以帮助来解决线程处理的问题。ZThread库就包含有很多这样的工具，比如信号量（semaphore）和在本章中所见到的与队列相似的特殊类型的队列。可以探究这个库以及其他有关线程处理的专题资源来得到更深入的知识。

至关重要的是要学会什么时候应该使用并发，以及什么时候应该避免使用并发。使用它的主要原因是：

- 为了处理许多任务，这些任务交织在一起，应用并发可以更有效地使用计算机（包括透明地分配任务到多CPU的能力）。
- 为了能够较好地组织代码。
- 为了用户使用更方便。

资源均衡的经典例子是在I/O等待期间使用CPU。给用户带来便利的经典例子是在长时间下载过程期间监视“停止”按钮是否按下。

线程额外的优点是它们提供“轻量级”的执行语境切换（约为100条指令）而非“重量级”进程语境切换（约上千条指令）。由于一个给定的进程中所有的线程共享同一内存空间，一个轻量级语境切换只改变了程序执行的先后顺序和局部变量。进程改变——重量级语境切换——必须调换所有内存空间。

多线程处理的主要缺陷是:

- 当等待共享资源时性能降低。
- 处理线程需要额外的CPU开销。
- 拙劣的程序设计决定会引发毫无益处的复杂性。
- 为诸如饥饿、竞争、死锁和活锁等病态行为的出现创造了机会。
- 跨平台操作造成的不一致性。在为本章开发原始材料（使用Java）时，作者就发现竞争条件在某些计算机上会很快出现，但在另外的计算机上则不会。本章中的C++例子在不同的操作系统下其行为是不同的（但通常是可接受的）。如果在某台计算机上开发一个程序，并且工作似乎一切正常，然而当发布它时你也许会因得到完全不受欢迎的结果而大吃一惊。

772

与线程一起存在的最大的困难之一是，因为多个线程也许在共享某个资源——比如一个对象中的内存——并且还要必须确定多个线程不会在同时读取和改变那个资源。这需要头脑精明地使用同步工具，必须要彻底理解这些同步工具，因为它们可以神不知鬼不觉地将程序引入到死锁的境遇。

另外，线程的应用有一定的技巧。C++被设计为允许创建足够多的对象来满足解决问题的需要——至少在理论上是这样。（比如：为进行工程上的有限元分析而创建上百万个对象，这可能是不现实的。）然而，想要创建的线程数目通常有一个上限，因为在达到某个数量时，线程的性能就会变得极差。这个临界点很难探测，且常常依赖于操作系统和线程库；它可以是少于一百个，也可能达到数千个线程。就我们常常只创建少量的线程以解决某个问题而言，这个限制没有多大作用；但是在更一般的设计中它就会变成一个约束。

用一种特定的语言或库来进行线程处理不管似乎有多么简单，都认为它是一种变幻无常的魔术。人们在编程时总会有些没有考虑周全的地方，因此就会在你最没有预料到的时候“咬你一口”。（比如，请注意因为哲学家进餐问题可以进行调整，所以死锁很少发生，人们就会得到一切都万事大吉的假象。）这里引用Python 编程语言的发明者Guido van Rossum的一个恰当的描述:

773

在任何多线程处理的程序设计中，大多数的错误来自于线程处理问题。这和编程语言无关——它是深层次的问题，即人们至今仍未完全理解的线程的性质。

有关线程处理更高级的讨论，请参看《Parallel and Distributed Programming Using C++》一书，Cameron Hughes和Tracey Hughes著，Addison-Wesley出版社2004年出版。

11.10 练习

11-1 从**Runnable**继承一个类，并重写**run( )**函数。在**run( )**中打印一个消息，然后调用**sleep( )**。重复这个过程3遍，然后从**run( )**返回。在构造函数中放进一条启动（start-up）消息，并且在任务结束时打印一个关闭（shut-down）消息。创建几个此类型的线程对象，运行它们并观察会发生什么结果。

11-2 修改**BasicThreads.cpp**，使**LiftOff**线程启动其他的**LiftOff**线程。

11-3 修改**ResponsiveUI.cpp**，消除任何可能的竞争条件。（假设**bool**操作是非原子的。）

11-4 在**Incrementer.cpp**中，修改**Count**类，使用一个**int**变量而不使用**int**数组。解释产生的行为。

11-5 在**EvenChecker.h**中，纠正**Generator**类中的潜在问题。（假设**bool**操作是非原子的。）

11-6 修改**EvenGenerator.cpp**，使用**interrupt( )**而不使用退出标志。

11-7 在**MutexEvenGenerator.cpp**中，改变**MutexEvenGenerator::nextValue( )**

中的代码，使返回表达式处在**release( )**语句之前，并解释会有什么情形发生。

11-8 修改**ResponsiveUI.cpp**，使用**interrupt( )**而非**quitFlag**方法。

774 11-9 查看ZThread库中**Singleton**的文档。修改**OrnamentalGarden.cpp**，以便**Display**对象被一个单件**Singleton**控制，防止多个**Display** 对象被意外地创建。

11-10 改变**OrnamentalGarden.cpp**中的**Count::increment( )**函数，使它直接对**count**增1（即使用**count++**）。现在删去保护，看看是否会引起失败。这种做法是安全可靠的吗？

11-11 修改**OrnamentalGarden.cpp**，使用**interrupt( )**而非**pause( )**机制。确保这种解决方案不会过早销毁对象。

11-12 修改**WaxOMatic.cpp**，添加**Process**类的更多实例，使它对汽车外壳上蜡进行3次上蜡和抛光，而不是只有一次。

11-13 创建**Runnable**的两个子类，一个子类在**run( )** 中启动然后调用**wait( )**。另一个子类的**run( )**获得第1个**Runnable**对象的引用。在若干秒后其**run( )**会为第1个线程调用**signal( )**，以使第1个线程可以打印一条消息。

11-14 创建一个“忙等待”的例子。一个线程休眠一段时间，然后把一个标志设置为**true**。第2个线程在一个**while**循环中监视这个标志（这就是“忙等待”），当标志变为**true**时，把它设置回为**false**，并把这个变化报告给控制台。注意程序在“忙等待”中耗费了多少时间，创建该程序的第2个版本，使用**wait( )**代替“忙等待”。特别提示：运行profiler，显示在每种情况下使用的CPU时间。

11-15 修改**TQueue.h**，添加可允许的最大元素计数值。如果元素数达到这个数量，后续的写操作应该被阻塞，直到元素计数小于最大元素计数值为止。编写代码检测这种行为。

11-16 修改**ToastOMaticMarkII.cpp**，使用两条独立的流水线，在烤面包三明治上涂抹花生酱和果冻，并为已完成的三明治使用一个输出队列**TQueue**。使用一个**Reporter**对象显示结果，就像在**CarBuilder.cpp**中那样。

11-17 使用真正的线程处理而非模拟线程处理重写**C07:BankTeller.cpp**。

11-18 修改**CarBuilder.cpp**，给机器人添加标识符，并且添加不同种类机器人的更多实例。注意是否所有机器人都得到利用。

775 11-19 修改**CarBuilder.cpp**，给汽车制造进程增加另一个阶段，添加排气系统、车身、挡泥板。如同在第1阶段，假设这些进程可同时被机器人执行。

11-20 修改**CarBuilder.cpp**，使**Car**可以对所有**bool**（布尔）变量进行同步访问。因为互斥锁**Mutex**不能被拷贝，这将需要对整个程序进行重大的修改。

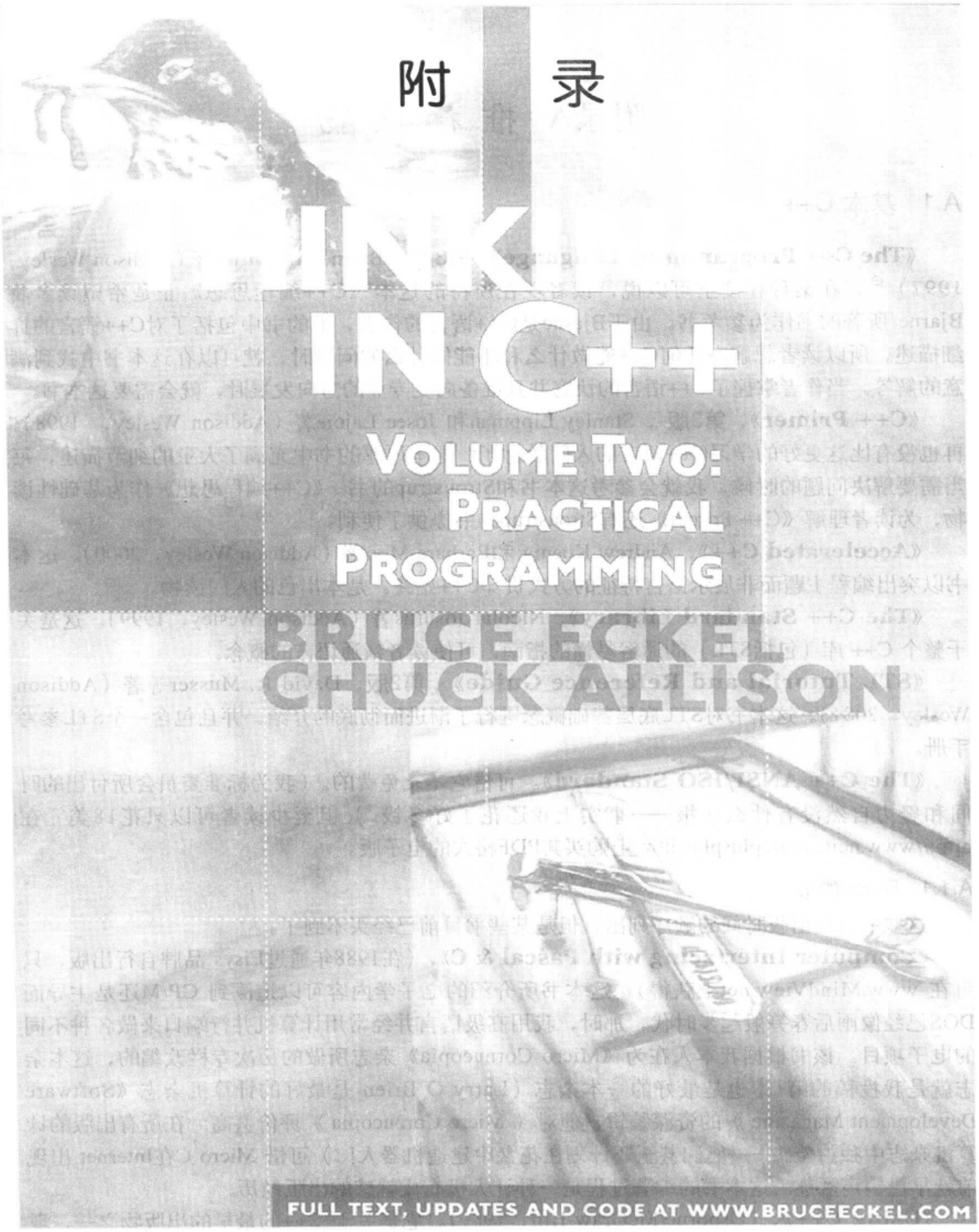
11-21 使用本章给出的**CarBuilder.cpp**中的方法，给房屋建筑的例程建模。

11-22 创建一个**Timer**类，这个类有两个选项：一个只发射一次的一次射击计时器，以及每隔一定的时间间隔定期发射的计时器。和**C10:MulticastCommand.cpp**一起使用这个类，把对**TaskRunner::run( )**的调用从程序中移到计时器类中。

11-23 改变哲学家进餐的例子中的两处内容，以便除了思考时间之外，还在命令行上控制**Philosopher**的数量。尝试输入不同的值并解释结果。

11-24 改变**DiningPhilosophers.cpp**，以便**Philosopher**正确拿起下一根可用的筷子。（当一个**Philosopher**取完其筷子后，他们把筷子放入一个筷笼中。当**Philosopher**需要进餐时，他们就从筷笼中取出下两根可用的筷子。）这种做法能够消除死锁的可能性吗？能仅通过减少可用筷子的数量而重新引入死锁吗？

附 录


THINK
IN C++
VOLUME TWO:
PRACTICAL
PROGRAMMING
BRUCE ECKEL
CHUCK ALLISON
FULL TEXT, UPDATES AND CODE AT WWW.BRUCEECKEL.COM

附录A 推荐读物

A.1 基本 C++

《The C++ Programming Language》，第3版，Bjarne Stroustrup著（Addison Wesley，1997）[⊖]。在某种程度上可以说，读者现在所持的这本《C++编程思想》正是希望读者将Bjarne 所著的书作为参考书。由于Bjarne是C++语言的作者，他的书中包括了对C++语言的详细描述，所以读者想解决任何C++能做什么和不能做什么的问题时，就可以在这本书中找到满意的解答。当作者掌握了C++语言的诀窍并且准备向更专业的方向发展时，就会需要这本书。

《C++ Primer》，第3版，Stanley Lippman和 Josee Lajoie著（Addison Wesley，1998）。再也没有比这更好的学习C++语言的入门教材了；这本厚厚的书中充满了大量的细节描述，每当需要解决问题的时候，我就会参考这本书和Stroustrup的书。《C++编程思想》作为基础性读物，为读者理解《C++ Primer》还有Stroustrup的书提供了便利。

《Accelerated C++》，Andrew Koenig 和Barbara Moo著（Addison Wesley，2000）。这本书以突出编程主题而非展示语言特征的方式贯穿C++始终。是本出色的入门读物。

《The C++ Standard Library》，Nicolai Josuttis著（Addison Wesley，1999）。这是关于整个 C++ 库（包括STL）的通俗易懂的指南。可使读者精通语言的概念。

《STL Tutorial and Reference Guide》，第2版，David R. Musser等著（Addison Wesley，2001）。这本书对STL底层基础概念进行了渐进而彻底的介绍。并且包含一个STL参考手册。

《The C++ ANSI/ISO Standard》。可惜它不是免费的。（我为标准委员会所付出的时间和努力自然没有什么回报——事实上我还花了好多钱。）但至少读者可以只花18美元在http://www.ncits.org/cplusplus.htm 上购买其PDF格式的电子版。

A.1.1 Bruce的书

这些书都按出版时间的顺序列出，但是某些书目前已经买不到了。

《Computer Interfacing with Pascal & C》，（在1988年通过Eisys 品牌自行出版，只可在 www.MindView.net上获得）。这本书所介绍的电子学内容可以追溯到 CP/M还是主导而DOS已经像雨后春笋般起步时代。那时，我用高级语言并经常用计算机并行端口来做各种不同的电子项目。该书根据我本人在为《Micro Cornucopia》杂志所做的历次专栏改编的，这本杂志就是我投稿的第1本也是最好的一本杂志（Larry O'Brien 是最好的计算机杂志《Software Development Magazine》的资深编辑，他对《Micro Cornucopia》评价甚高：在所有出版的计算机杂志中独占鳌头——他们甚至还计划在花盆中建造机器人！）可惜 Micro C在Internet 出现前就早已销声匿迹。这本书的编著过程是一种让人极有成就感的出版经历。

《Using C++》，（Osborne/McGraw-Hill，1989）。它是关于 C++的最早的出版物之一，现已绝版，并且为其第2版代替——重新命名该书为《C++ Inside & Out》。

⊖ 本书的中文版已由机械工业出版社出版。——编辑注

《C++ Inside & Out》，(Osborne/McGraw-Hill，1993)。如前所述，这本书是《Using C++》的第2版。该书对 C++的论述更为精确，但它毕竟是1992年前后的作品，后来就被《Thinking in C++》所代替。读者可以从 www.MindView.net 网站获知更多的关于此书的信息，并且可以下载其源代码。

《Thinking in C++》，第1版，(Prentice Hall，1995)[㊀]。它是1995年软件研发类的最佳书籍，获得该年度《Software Development Magazine》杂志的震撼奖（Jolt Award)。

《Thinking in C++》，第2版，第1卷，(Prentice Hall，2000)[㊁]。可从 www.MindView.net下载。

《Black Belt C++: the Master's Collection》，Bruce Eckel编著（M&T Books，1994)，现已绝版（但一般还可通过网上绝版书籍的相关服务获得)。这本书的各章是作者主持的软件研发研讨会（Software Development Conference）上很多C++专家对关于C++发展历程所做论述的集锦。这本书的封面促使了我在今后的时间里自行完成自己所有著作的封面设计。

《Thinking in Java》，第1版，(Prentice Hall，1998)。[㊂]这本书第1版荣获《Software Development Magazine》杂志的畅销奖、《Java Developer's Journal》杂志的编者推荐精品图书奖、《JavaWorld Reader's》杂志设置的最佳图书之精品奖。电子版附在书后的 CD ROM中，也可从 www.MindView.net 上下载。

《Thinking in Java》，第2版，(Prentice Hall，2000)[㊃]。这一版摘得《JavaWorld》杂志的编者推荐精品图书奖。电子版附在书后的 CD ROM中，也可从 www.MindView.net 上下载。

《Thinking in Java》，第3版，(Prentice Hall，2002)[㊄]。这一版夺得《Software Development Magazine》杂志的2002年度之震撼奖、《Java Developer's Journal》杂志的编者推荐精品图书奖。书后附加的新CD ROM现在包括《The Hands-On Java CD ROM》第2版的前7个学术报告。

《The Hands-On Java CD ROM》，第3版，(MindView，2004)。它基于《Thinking in Java》第3版书的内容，其中包括Bruce的超过15个小时的学术报告，还有相应的幻灯片，涵盖了Java 语言的整个基础。只能从www.MindView.net网站上获得。

A.1.2 Chuck 的书

《C & C++ Code Capsules》，由Chuck Allison著（Prentice-Hall，1998)。它对实践C与C++编程有实际而通用的指导意义，并且完全涵盖了1998 ISO C++标准，特别是库特征。在引导读者做更高级课题时，它会起到桥梁的作用。这本书是源于Chuck在《C/C++ Users Journal》杂志上的出色专栏编辑而成。

《Thinking in C: Foundations for Java & C++》，Chuck Allison著（它不是一本书，而是1999年MindView公司在学术研讨会上制作的CD ROM，与《Thinking in Java》和《Thinking in C++》Volume 1是绑定在一起的)。这其实是一个多媒体教程，它包括讲述C语言基础的讲稿和幻灯片，为读者学习Java或 C++做准备。它不是完全关于C语言的教程，只

779

㊀ 本书的中文版已由机械工业出版社出版。——编辑注

㊁ 本书的中文版和英文影印版已由机械工业出版社出版。——编辑注

㊂ 本书的中文版已由机械工业出版社出版。——编辑注

㊃~㊄ 本书的中文版和英文影印版已由机械工业出版社出版。——编辑注

附加那些对继续学习其他语言十分必要的材料，同时还额外包括一些为C++程序员提供的C++特性。学习本教程的先决条件：读者必须实践过高级编程语言，如Pascal、BASIC、FORTRAN或LISP。

A.2 深入理解 C++

780 下面介绍的这些都是更深层次探讨语言的书，它们可以帮助读者避免犯开发C++程序中一些固有且典型的错误。

《Large-scale C++ Software Design》，John Lakos著（Addison Wesley，1996）。可以激发读者的学习欲望，给读者介绍对于大型C++项目的实际而通用的编程技巧。

《Effective C++》，**第2版**，Scott Meyers著（Addison Wesley，1997）。它是改善C++设计的经典的技术性书籍。它系统地归纳了让许多程序员不得不煞费苦心学习的材料。

《More Effective C++》，Scott Meyers著（Addison Wesley，1995）。它是（上述）《Effective C++》的延续，是另一部C++经典之作。

《Effective STL》，Scott Meyers著（Addison Wesley，2001）。它广泛且深入地覆盖了实用的使用STL的方法。它还包含了独有的专家意见。

《Generic Programming and the STL》，Matt Austern著（Addison Wesley，1998）。它探究STL设计的概念的基础结构。它着重于理论，但在通用库的设计方面让读者眼界大开。

《Exceptional C++》，Herb Sutter著（Addison Wesley，2000）。它会引导读者从各种问题和相应解法中循序渐进地学习。为现代C++程序稳健的设计提供简单易记且可行性强的建议。

《More Exceptional C++》，Herb Sutter著（Addison Wesley，2001）。它是（上述）《Exceptional C++》的延续。

《C++FAQs》，**第2版**，Marshall Cline、Greg Lomow 和Mike Girou著（Addison Wesley，1998）。它是一本结构精心设计的书，提纲挈领地提出了C++的常见问题及其解答。涵盖了从初级到高级的广泛的主题。

《C++Gotchas》，Stephen Dewhurst著（Addison Wesley，2002）。该书汇集了现代的易于发现却难于纠正的C++行为怪异的程序和问题，这本书被C++专家广为认可。

《C++ Templates, The Complete Guide》，Daveed Vandevoorde和Nicolai M. Josuttis著（Addison Wesley，2002）。它是第1本也是惟一一本完全致力于模板王朝的书。被认为是模版的权威参考书。

《Standard C++ iostreams and Locales》，Angelika Langer和 Klaus Kreft著（Addison Wesley，2000）。它最深入地阐述了目前可获得的输入输出流。深入探究了流实现与习惯的用法。这是本使用便利的参考手册也是一本指南。

781 **《Design & Evolution of C++》**，Bjarne Stroustrup著（Addison Wesley，1994）[⊖]。它追溯了C++发展的全过程，为各个时期的设计决策提供历史借鉴。如果读者想知道C++为何会变成今天这样，这本书将会阐释一切，因为它是由语言的设计者编写的。

《Modern C++ Design》，Andrei Alexandrescu著（Addison Wesley，2001）。它是C++基于策略设计的标准文本。汇集了很多使用模版的高级实践方法。

《Parallel and Distributed Programming Using C++》，Cameron Hughes和Tracey

⊖ 本书的中文版和英文影印版已由机械工业出版社出版。——编辑注

Hughes著（Addison Wesley，2004）。它完全而深入浅出地涵盖了并发的各个方面，包括基本概念、线程和多进程。读者对象可以是初学者也可以是专家一级的人物。

《Generative Programming》，Krzysztof Czarnecki 和Ulrich Eisenecker著（Addison Wesley，2000）。它是高级C++技术划时代的书籍。促使软件自动化向更高阶段发展。

《Multi-Paradigm Design for C++》，James O. Coplien 著（Addison Wesley，1998）。它是本高级读物，阐释了为达到有效的C++设计，怎样协调过程编程、面向对象编程还有一般编程的用法。

A.3 设计模式

《Design Patterns》，Erich Gamma 等著（Addison Wesley，1995）[⊖]。这是一本具革命性的书籍，它将设计模式引入软件产业。汇集了一些精选的设计模式，这些模式都有其设计动机和举例（大部分例子采用C++语言编写，有一些用SmallTalk语言编写）。

《Pattern-Oriented Software Architecture，Volume 1: A System of Patterns》，Frank Buschmann等著（John Wiley & Son，1996）[⊜]。这是另一本从实践角度介绍设计模式的书。它引进了新的设计模式。

⊖ 本书的中文版和英文影印版已由机械工业出版社出版。——编辑注

⊜ 本书的中文版已由机械工业出版社出版。——编辑注

附录B 其 他

此附录包含的文件是构建第2卷中例子所必需的。

```
//: :require.h
// 测试程序中的错误条件
#ifndef REQUIRE_H
#define REQUIRE_H
#include <cstdio>
#include <cstdlib>
#include <fstream>

inline void require(bool requirement,
  const char* msg = "Requirement failed") {
  //为旧式编译器提供的局部语句"using namespace std":
  using namespace std;
  if(!requirement) {
    fputs(msg, stderr);
    fputs("\n", stderr);
    exit(EXIT_FAILURE);
  }
}

inline void requireArgs(int argc, int args,
  const char* msg = "Must use %d arguments") {
  using namespace std;
  if(argc != args + 1) {
    fprintf(stderr, msg, args);
    fputs("\n", stderr);
    exit(EXIT_FAILURE);
  }
}

inline void requireMinArgs(int argc, int minArgs,
  const char* msg = "Must use at least %d arguments") {
  using namespace std;
  if(argc < minArgs + 1) {
    fprintf(stderr, msg, minArgs);
    fputs("\n", stderr);
    exit(EXIT_FAILURE);
  }
}

inline void assure(std::ifstream& in,
  const char* filename = "") {
  using namespace std;
  if(!in) {
    fprintf(stderr, "Could not open file %s\n", filename);
    exit(EXIT_FAILURE);
  }
}

inline void assure(std::ofstream& in,
  const char* filename = "") {
  using namespace std;
  if(!in) {
    fprintf(stderr, "Could not open file %s\n", filename);
```

```
    exit(EXIT_FAILURE);
  }
}

inline void assure(std::fstream& in,
  const char* filename = "") {
  using namespace std;
  if(!in) {
    fprintf(stderr, "Could not open file %s\n", filename);
    exit(EXIT_FAILURE);
  }
}
#endif // REQUIRE_H ///:~

//: C0B:Dummy.cpp
// 至少给出编写文件
// 这个路径的一个目标
int main() {} ///:~
```

Date类文件:

```
//: C02:Date.h
#ifndef DATE_H
#define DATE_H
#include <string>
#include <stdexcept>
#include <iosfwd>

class Date {
  int year, month, day;
  int compare(const Date&) const;
  static int daysInPrevMonth(int year, int mon);
public:
  //用于日期计算的类
  struct Duration {
    int years, months, days;
    Duration(int y, int m, int d)
    : years(y), months(m) ,days(d) {}
  };
  // 一个异常类
  struct DateError : public std::logic_error {
    DateError(const std::string& msg = "")
    : std::logic_error(msg) {}
  };
  Date();
  Date(int, int, int) throw(DateError);
  Date(const std::string&) throw(DateError);
  int getYear() const;
  int getMonth() const;
  int getDay() const;
  std::string toString() const;
  friend Duration duration(const Date&, const Date&);
  friend bool operator<(const Date&, const Date&);
  friend bool operator<=(const Date&, const Date&);
  friend bool operator>(const Date&, const Date&);
  friend bool operator>=(const Date&, const Date&);
  friend bool operator==(const Date&, const Date&);
  friend bool operator!=(const Date&, const Date&);
  friend std::ostream& operator<<(std::ostream&,
                                  const Date&);
  friend std::istream& operator>>(std::istream&, Date&);
};
#endif // DATE_H ///:~
```

785

```
//: C02:Date.cpp {O}
#include "Date.h"
#include <iostream>
#include <sstream>
#include <cstdlib>
#include <string>
#include <algorithm> // 用于 swap( )函数
#include <ctime>
#include <cassert>
#include <iomanip>
using namespace std;

namespace {
  const int daysInMonth[][13] = {
    { 0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 },
    { 0, 31, 29, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 }
  };
  inline bool isleap(int y) {
    return y%4 == 0 && y%100 != 0 || y%400 == 0;
  }
}

Date::Date() {
  //获得当前日期
  time_t tval = time(0);
  struct tm *now = localtime(&tval);
  year = now->tm_year + 1900;
  month = now->tm_mon + 1;
  day = now->tm_mday;
}

Date::Date(int yr,int mon,int dy) throw(Date::DateError) {
  if(!(1 <= mon && mon <= 12))
    throw DateError("Bad month in Date ctor");
  if(!(1 <= dy && dy <= daysInMonth[isleap(year)][mon]))
    throw DateError("Bad day in Date ctor");
  year = yr;
  month = mon;
  day = dy;
}

Date::Date(const std::string& s) throw(Date::DateError) {
  // 假定格式为YYYYMMDD
  if(!(s.size() == 8))
    throw DateError("Bad string in Date ctor");
  for(int n = 8; --n >= 0;)
    if(!isdigit(s[n]))
      throw DateError("Bad string in Date ctor");
  string buf = s.substr(0, 4);
  year = atoi(buf.c_str());
  buf = s.substr(4, 2);
  month = atoi(buf.c_str());
  buf = s.substr(6, 2);
  day = atoi(buf.c_str());
  if(!(1 <= month && month <= 12))
    throw DateError("Bad month in Date ctor");
  if(!(1 <= day && day <=
    daysInMonth[isleap(year)][month]))
    throw DateError("Bad day in Date ctor");
}

int Date::getYear() const { return year; }
```

```
int Date::getMonth() const { return month; }

int Date::getDay() const { return day; }

string Date::toString() const {
  ostringstream os;
  os.fill('0');
  os << setw(4) << year
     << setw(2) << month
     << setw(2) << day;
  return os.str();
}

int Date::compare(const Date& d2) const {
  int result = year - d2.year;
  if(result == 0) {
    result = month - d2.month;
    if(result == 0)
      result = day - d2.day;
  }
  return result;
}

int Date::daysInPrevMonth(int year, int month) {
  if(month == 1) {
    --year;
    month = 12;
  }
  else
    --month;
  return daysInMonth[isleap(year)][month];
}

bool operator<(const Date& d1, const Date& d2) {
  return d1.compare(d2) < 0;
}
bool operator<=(const Date& d1, const Date& d2) {
  return d1 < d2 || d1 == d2;
}
bool operator>(const Date& d1, const Date& d2) {
  return !(d1 < d2) && !(d1 == d2);
}
bool operator>=(const Date& d1, const Date& d2) {
  return !(d1 < d2);
}
bool operator==(const Date& d1, const Date& d2) {
  return d1.compare(d2) == 0;
}
bool operator!=(const Date& d1, const Date& d2) {
  return !(d1 == d2);
}

Date::Duration
duration(const Date& date1, const Date& date2) {
  int y1 = date1.year;
  int y2 = date2.year;
  int m1 = date1.month;
  int m2 = date2.month;
  int d1 = date1.day;
  int d2 = date2.day;

  // 计算出比较值
  int order = date1.compare(date2);
```

788

```
  if(order == 0)
    return Date::Duration(0,0,0);
  else if(order > 0) {
    // 在局部使date1先于date2
    using std::swap;
    swap(y1, y2);
    swap(m1, m2);
    swap(d1, d2);
  }

  int years = y2 - y1;
  int months = m2 - m1;
  int days = d2 - d1;
  assert(years > 0 ||
     years == 0 && months > 0 ||
     years == 0 && months == 0 && days > 0);
  //做明显的更正(必须在调整月份前调整天数! )
  //这是一个循环，以防前一个月是二月还有天数小于-28
  int lastMonth = m2;
  int lastYear = y2;
  while(days < 0) {
    //从 month 中借天
    assert(months > 0);
    days += Date::daysInPrevMonth(
      lastYear, lastMonth--);
    --months;
  }

  if(months < 0) {
    //从 year 中借月
    assert(years > 0);
    months += 12;
    --years;
  }
  return Date::Duration(years, months, days);
}

ostream& operator<<(ostream& os, const Date& d) {
  char fillc = os.fill('0');
  os << setw(2) << d.getMonth() << '-'
     << setw(2) << d.getDay() << '-'
     << setw(4) << setfill(fillc) << d.getYear();
  return os;
}

istream& operator>>(istream& is, Date& d) {
  is >> d.month;
  char dash;
  is >> dash;
  if(dash != '-')
    is.setstate(ios::failbit);
  is >> d.day;
  is >> dash;
  if(dash != '-')
    is.setstate(ios::failbit);
  is >> d.year;
  return is;
} ///:~
```

789

下面是第6章用到的文本文件:

```
//: C06:Test.txt
f a f d A  G f d F a A F h f A d f f a a
///:~
```

790

索　引

注：索引中的页码为英文原书的页码，与本书中边栏的页码相同。

<
<cctype>，252
<cstdlib>，215
<ctime>，212
<exception>，38
<fstream>，169
<functional>，338
<iomanip>，196
<iosfwd>，163
<limits>，181，203，285
<memory>，35
<sstream>，179
<stdexcept>，38
<typeinfo>，557

A

abort( )，27
Abstract Factory design pattern（抽象工厂设计模式），651
abstraction, in program design（抽象，在程序设计中），614
accumulate algorithm（**accumulate** 算法），413
activation record instance（活动记录实例），58
adaptable function object（可调整的函数对象），341
Adapter design pattern（适配器设计模式），636
adaptor（适配器）:
  container（容器），433，487;
  function object（函数对象），338;
  function pointer（函数指针），351;
  iterator（迭代器），487
adjacent_difference algorithm（**adjacent_difference**算法），415
adjacent_find algorithm（**adjacent_find**算法），378
aggregation, design patterns（聚合，设计模式），616
Alexandrescu，Andrei，294，305
algorithm（算法）:
  accumulate，413;
  adjacent_difference，415;
  adjacent_find，378;
applicators（应用算子），405;
binary_search，395;
complexity（复杂性），333;
copy（复制）326，365;
copy_backward，372;
count（计数），370;
count_if，334，371;
counting（计数），370;
creating your own（创建自己的...），419;
equal（相等），327，385;
equal_range，396;
fill，369;
fill_n，369;
filling and generating（填充与生成），368;
find（查找），334，377;
find_end，379;
find_first_of，378;
find_if，378;
for_each，355，405;
general utilities（通用实用程序），417;
generate，369;
generate_n，369;
generic（通用的），325;
heap operations（堆运算），403;
includes，400;
inner_product，414;
inplace_merge，399;
iter_swap，419，457;
lexicographical_compare，385;
lower_bound，395;
make_heap，404;
manipulating sequences（操作序列），372;
max，419;
max_element，380;
merge，399;
merging（合并），398;
min，418;
min_element，379;
mismatch，386;
next_permutation，373;

nth_element, 394;
numeric（数值的）, 413;
ordering（排序）, 393;
partial_sort, 394;
partial_sort_copy, 394;
partial_sum, 414;
partition, 374;
pop_heap, 404;
predicate（判定函数）, 329;
prev_permutation, 373;
push_heap, 404;
random_shuffle, 374;
range of sequence in(…的序列范围), 326;
remove, 389;
remove_copy, 389;
remove_copy_if, 329, 339, 350, 390;
remove_if, 389;
removing elements（删除元素）, 389;
replace, 380;
replace_copy, 380;
replace_copy_if, 330, 380;
replace_if, 330, 380;
reverse, 372;
reverse_copy, 372;
rotate, 373;
rotate_copy, 373;
search, 379;
search_n, 379;
searching and replacing(查找和替换), 377;
set operations（**set** 操作）, 400;
set_difference, 401;
set_intersection, 401;
set_symmetric_difference, 402;
set_union, 401;
sort, 366, 393;
sort_heap, 404;
sorting（排序）, 393;
stable_partition, 374;
stable_sort, 366, 393;
swap, 419;
swap_ranges, 373;
transform, 347, 349, 355, 405;
unique, 390;
unique_copy, 390;
upper_bound, 395;
utilities（实用程序）, 417
ANSI/ISO C++ Committee（ANSI/ISO C++ 委员会）, 9
applicator algorithms（应用算子算法）, 405
applicator, iostreams manipulator（应用算子、输入输出流操纵算子）, 200
applying a function to a container（将一个函数用于容器）, 255
argument_type, 342
argument-dependent lookup（关联参数查找）, 274, 278;
disabling（失效的）, 275
assert macro, 66
assertion（断言）, 66;
side effects in an (…中的副作用), 67
Assignable, 337
associative container（关联式容器）, 433, 513
atof( ), 181
atoi( ), 181
atomic operation（原子操作）, 732
auto_ptr, 35;
not for containers（不是针对容器）, 437
automated testing（自动测试）, 71
automatic type conversion, and exception handling（自动类型转换、和异常处理）, 23

B

back_insert_iterator, 448, 482
back_inserter( ), 328, 370, 372, 418, 448
bad_cast exception class（**bad_cast**异常类）, 40, 557
bad_exception class (**bad_exception**类), 44
bad_typeid exception class（**bad_typeid**异常类）, 40,559
badbit（致命错误标志位）, 165
basic_istream, 158, 216
basic_ostream, 158, 217
basic_string, 134,217,241
Becker, Pete, 11
before( ), RTTI function（**before( )**,RTTI函数）, 559
behavioral design patterns（行为设计模式）, 616
bidirectional iterator（双向迭代器）, 446
BidirectionalIterator, 364
binary files（二进制文件）, 172, 214
binary function（二元函数）, 337
binary predicate（二元判定函数）, 337
binary search（折半查找）, 63
binary_function, 342, 353;
first_argument_type, 342;
result_type, 342;
second_argument_type, 342
binary_negate function object（**binary_negate**函

数对象)，341
binary_search algorithm (**binary_search**算法)，395
bind1st function object adaptor (**bind1st**函数对象适配器)，339
bind2nd function object adaptor (**bind2nd**函数对象适配器)，338，350，371
binder1st function object (**binder1st**函数对象)，339
binder2nd function object (**binder2nd**函数对象)，339
bitset，229，506，540;
  to_string( )，241
blocking, and threads (阻塞、和线程)，734
book errors, reporting (书中的错误，报告)，10
Bright, Walter，8，11
broadcast( ), threading (**broadcast( )**，线程) 734，742，757
buffering, stream (缓冲，流)，173
Builder design pattern (构建器设计模式)，660
busy wait, threading (忙等，线程)，732,743

C

cancel( ), ZThread library function (**cancel( )**，ZThread 库函数)，717
Cancelable, ZThread library class (**Cancelable**，ZThread 库类)，717
cast: downcast (类型转换：向下类型转换)，551;
  dynamic_cast，555;
  runtime (运行时)，551;
  runtime type identification, casting to intermediate levels (运行时类型识别，转换类型到中间层类型)，560
catch，20;
  catching any exception (捕获任意异常)，25
cerr，158
cfront，574
Chain of Responsibility design pattern (职责链设计模式)，642
chaining, in iostreams (链接，在输入输出流中)，159
change, vector of change (变更，变更向量)，614
char_traits，217，241，287
character traits (字符特征)，217;
  compare( )，217
cin，158
class: (类):
  hierarchies and exception handling (层次结构与异常处理)，24;
  invariant (不变量)，69;
  maintaining library source (维护库源代码)，204;
  wrapping (封装)，151
  class template: (类模板):
  partial ordering (半有序)，263;
  partial specialization (半特化)，263
cleaning up the stack during exception handling (在异常处理时清理堆栈)，28
clear( )，166，175
close( )，168
code bloat, of templates (代码膨胀，模板的)，268
code invariant (代码不变量)，63
cohesion (内聚)，49
Collecting Parameter design pattern (idiom) (收集参数设计模式 (习语))，618
command line, interface (命令行，接口)，162
Command pattern (命令模式)，626;
  decoupling (消除耦合)，628
Committee, ANSI/ISO C++ (委员会，ANSI/ISO C++)，9
compilation, of templates (编译，模板的)，315
compile time: assertions (编译时间：断言)，304;
  error checking (错误检测)，155;
  looping (循环)，299;
  selection (选择)，303
complexity of algorithms (算法复杂性)，333
compose non-standard function object adaptor (**compose**非标准函数对象适配器)，360
composition, and design patterns (组合，和设计模式)，614，615
concurrency (并发)，691;
  blocking (阻塞)，734;
  Command pattern (命令模式)，628;
  when to use it (什么时间用它)，771
ConcurrentExecutor (Concurrency) (**ConcurrentExecutor** (并发))，704
Condition class, threading (**Condition**类，线程)，742
console I/O (控制台输入/输出)，162
constructor: (构造函数):
  default constructor synthesized by the compiler (由编译器合成的默认构造函数)，620;
  design patterns (设计模式)，616;
  exception handling (异常处理)，29，30，57;
  failing (故障)，57;
  order of constructor and destructor calls (构造函数与析构函数调用的顺序)，562;
  private constructor (私有构造函数)，620;
  protected (保护的)，581;

simulating virtual constructors（模拟虚拟构造函数），654；
container（容器），429；
adaptor（适配器），433，487；
associative（关联式），433，513；
bitset，506，540；
cleaning up（清理），437，534；
combining STL containers（将STL容器联合使用），530；
creating custom（创建自定义选项），536；
deque，434，465；
duplicate keys（重复关键字），523；
extended STL containers（扩展型STL容器），440；
list，434，471；
map，513，521；
multimap，513，523；
multiple membership problem（多成员问题），438；
multiset，513，527；
of pointers（指针的），436；
priority_queue，496；
queue，491；
reversible（可逆的），445；
sequence（序列），433；
sequence operations（序列操作），454；
set，479，513；
stack，487；
valarray，540；
value-based（基于值的），434；
vector（向量），434，457；
vector<bool>，506，511
contract, design by,（书名：契约式设计）68
conversion, automatic type conversions and exception handling（转换，自动类型转换和异常处理），23
cooperation between threads（线程间的协作），741
Coplien, James，296，655
copy algorithm（**copy**算法），326，365
copy_backward algorithm（**copy_backward**算法），372
copy-on-write（写入时复制），634
count algorithm（**count**算法），370
count_if algorithm（**count_if**算法），334，371
CountedPtr, reference-counting template in ZThread library (Concurrency)（**CountedPtr**, ZThread库中的引用计数模板（并发）），714
counting algorithms（计数算法），370
cout，158
covariance, of exception specifications（协变，异常规格说明的），47
Crahen, Eric，11，694
creational design patterns（可创建的设计模式），615
critical section, in thread programming（临界区，在线程编程中），719
curiously recurring template pattern（奇特的递归模板模式），294，624
Cygwin, and ZThreads，696
Czarnecki, Krysztof，300

D

datalogger（数据记录器），211
dead thread（死亡线程），734
deadlock（死锁），720，764；
conditions for（为……的条件），769
debugging（调试），87
dec，187
declaration, forward（说明，前向），163
default constructor: synthesized by the compiler（默认构造函数：由编译器合成），620
dependent base class（关联基类），278
dependent name（关联名），274
deque，434，465
design:（设计）：
abstraction in program design（程序设计中的抽象），614；
cohesion（内聚），49；
decisions（意图），66；
exception-neutral（异常中立），52；
exception-safe（异常安全），48
design by contract（契约式设计），68
design patterns（设计模式），613；
Abstract Factory（抽象工厂），651；
Adapter（适配器），636；
aggregation（聚合），616；
behavioral（行为的），616；
Builder（构建器），660；
Chain of Responsibility（职责链），642；
Collecting Parameter idiom（收集参数习语），618；
Command（命令），626；
constructors（构造函数），616；
creational（可创建的），615；
destructors（析构函数），616；
Double Dispatching（双重派遣），679；
Factory Method（工厂方法），581，645；
Messenger idiom（信使习语），617；
Multiple Dispatching（多重派遣），679；
Observer（观察者），667；

Proxy（代理），632;
simulating virtual constructors（模拟虚构造函数），654;
Singleton（单件），460，619;
State（状态），634;
Strategy（策略），640;
structural（结构的），615;
Template Method（模板方法），639;
vector of change（变化的向量），614;
Visitor（访问者），683
destructor（析构函数），659;
design patterns（设计模式），616;
exception handling（异常处理），28，57;
explicit call（显式调用），453;
order of constructor and destructor calls（构造函数与析构函数调用的顺序），562;
virtual（虚的），581
diamond inheritance（菱形继承），588
difference_type，370
dining philosophers, threading（哲学家进餐，线程），764
dispatching:（派遣）:
Double Dispatching design pattern（双重派遣设计模式），679;
Multiple Dispatching design pattern（多重派遣设计模式），679
distance( )，417
divides function object（**divides**函数对象），341
documentation, library（文件，库），101
document-view architecture（文档可视体系结构），667
domain_error exception class（**domain_error**异常类），40
dominance（优先于），601
Double Dispatching design pattern（双重派遣设计模式），653，679
downcast（向下类型转换），551
dynamic type, of an object（动态类型，一个对象的），557
dynamic_cast，555;
casting to intermediate levels（类型转换到中间层），560;
difference between **dynamic_cast** and **typeid**, runtime type identification（**dynamic_cast**和**typeid**之间的差别，运行时类型识别），561;
for polymorphic types（对于多态类型），556

E

effectors（效用算子），201
efficiency:runtime type identification（效率：运行时类型识别），565;
threads and（线程和），693
Eisenecker, Ulrich，300
ellipses, with exception handling（省略号，由于异常处理），25
endl，195
envelope, and letter classes（信封，和信件类），655
eofbit（文件结束标志位），166
epsilon( )，181
equal algorithm（**equal**算法），327，385
equal_range algorithm（**equal_range**算法），396
equal_to function object（**equal_to**函数对象），339，341
EqualityComparable，337
errno，16
error:handling（错误：处理），15;
handling, in C(处理，使用C语言)，16;
recovery（恢复），15;
reporting errors in book（报告书中的错误），10
event-driven programming, and the Command pattern（事件驱动编程，和命令模式），628
exception class (**exception**类)，38;
what( )，38
exception handling（异常处理），15;
asynchronous events (异步事件)，53;
atomic allocations for safety(基于安全的原子分配)，32;
automatic type conversions (自动类型转换)，23;
bad_cast exception class（**bad_cast**异常类），40，557;
bad_exception class（**bad_exception**类），44;
bad_typeid exception class（**bad_typeid**异常类），40，559;
catching an exception（捕获一个异常），20;
catching any exception（捕获任一异常），25，26;
catching by reference（通过引用捕获），23;
catching via accessible base (通过可访问的基类捕获)，25;
class hierarchies（类层次结构），24;
cleaning up the stack during a throw（在一次抛掷期间清空栈），28;
constructors（构造函数），29，30，57;
destructors（析构函数），28，36，57;
domain_error exception class（**domain_error**异常类），40;
ellipses（省略号），25;
exception class（**exception**类），38;
exception class, **what( )**(**exception**类, **what( )**),

38;
exception handler（异常处理器），20;
exception hierarchies（异常层次结构），56;
exception matching（异常匹配），23;
exception neutral（异常中立），52;
exception safety（异常安全），48;
exception specifications（异常规格说明），40;
exception type（**exception**类型），39;
incomplete objects（不完全的对象），29;
inheritance（继承），24;
invalid_argument exception class（**invalid_argument**异常类），40;
length_error exception class（**length_error**异常类），40;
logic_error class（**logic_error**类），38;
memory leaks（内存泄漏），29;
multiple inheritance（多重继承），56;
naked pointers（悬挂指针），30;
object slicing and（对象切割和），23;
out_of_range exception class（**out_of_range**异常类），40;
overhead of（...的开销），58;
programming guidelines（编程指导原则），52;
references（引用），34，56;
resource management（资源管理），30;
rethrowing an exception（重新抛掷一个异常），26，52;
runtime_error class（**runtime_error**类），38;
set_terminate( )，27;
set_unexpected( )，41;
specifications, and inheritance（规格说明，和继承），46;
specifications, covariance of（规格说明，...的协变），47;
specifications, when not to use（规格说明，当不用的时候），47;
stack unwinding（栈反解），19;
Standard C++ library exceptions（标准C++库异常），38;
terminate( )，44;
termination vs. resumption（终止与恢复），22;
testing（测试），79;
throwing & catching pointers（抛掷和捕获指针），57;
throwing an exception（抛出一个异常），18，19;
typical uses of exceptions（异常的典型用法），54;
uncaught exceptions（不捕获异常），26，28;
unexpected( )，41;
when to avoid（什么时候避免），52;
zero-cost model（零代价模型），60;
ZThreads (Concurrency)（ZThreads，（并发）），708
exception specifications（异常规格说明），40;
covariance of（...的协变），47;
inheritance（继承），46;
when not to use（什么时候不用），47
exclusion, mutual, in threads（排斥，相互的，在线程中），719
Executors, ZThread (Concurrency)（执行器，ZThread（并发）），702
explicit instantiation, of templates（显式实例化，模板的），316
export keyword（**export**关键字），319
exported templates（导出模板），319
expression templates（表达式模板），308
extractor, stream（提取符，流），158
Extreme Programming (XP)（极限编程，XP），71，615

F

facet: locale（领域：区域），220
Factory Method design pattern（工厂方法设计模式），581，645
fail( )，175
failbit（失败标志位），160，166
Fibonacci（斐波那契），298，636
file streams（文件流），**close( )**，168
file, iostreams（文件，输入输出流），156，162
FILE, stdio（**FILE**，标准输入/输出），152
fill algorithm（**fill**算法），369
fill_n algorithm（**fill_n**算法），369
filling and generating algorithms（填充和生成算法），368
find algorithm（**find**算法），334，377
find_end algorithm（**find_end**算法），379
find_first_of algorithm（**find_first_of**算法），378
find_if algorithm（**find_if**算法），378
first_argument_type，342
flock( ), and SynchronousExecutor (Concurrency)（**flock( )**，同步执行器（并发）），705
flush, iostreams（**flush**，输入输出流），195
for_each algorithm（**for_each**算法），355，405
format fields（格式化域），188
format flags:（格式标识）:，187;
dec，187;
hex，187;
ios; **showbase**，187;

showpoint, 187;
showpos, 187;
skipws, 187;
unitbuf, 187;
uppercase, 187;
oct, 187
formatted I/O (格式化I/O), 186
formatting: (格式化:), 179;
in-core (内核), 179;
manipulators, iostreams (操纵算子, 输入输出流), 194;
output stream (输出流), 186
forward declaration (前置声明), 163
forward iterator (前向迭代器), 446
forward_iterator_tag, 447
ForwardIterator, 363
framework, unit test (框架, 单元测试), 75
friend template (友元模板), 284
friends, of templates (友元, 模板的), 279
front_insert_iterator, 448
front_inserter( ), 418, 448
fseek( ), 176
fstream, 168;
simultaneous input and output (同步输入输出), 172
function: applying a function to a container (函数: 将一个函数应用于一个容器), 255;
binary (二元的), 337;
unary (一元的), 337
function object (函数对象), 335, 626;
adaptable (可调整的), 341;
adaptor (适配器), 338;
binary_negate, 341;
bind1st adaptor (**bind1st**适配器), 339;
bind2nd adaptor (**bind2nd**适配器), 338, 350;
binder1st, 339;
binder2nd, 339;
classification (分类), 336;
divides, 341;
equal_to, 339, 341;
greater, 338, 341, 371;
greater_equal, 341;
less, 341;
less_equal, 341;
logical_and, 341;
logical_not, 341;
logical_or, 341;
minus, 340;
modulus, 341;
multiplies, 341;
negate, 341;
not_equal_to, 341;
not1 adaptor (**not1**适配器), 339;
plus, 340;
unary_negate, 341
function object adaptor (函数对象适配器), 338;
bind2nd, 371;
not1, 352
pointer_to_binary_function, 353;
pointer_to_unary_function, 352
function pointer adaptor (函数指针适配器), 351;
ptr_fun, 351
function template (函数模板), 245;
address of (...的地址), 251;
explicit qualification (显式说明), 246;
overloading (重载), 249;
partial ordering of (...的半有序), 259;
specialization (特化), 261;
type deduction parameters in (类型推断参数在...), 245
function-call operator (函数调用运算符), 335
function-level try blocks (函数级try块), 36
functor (函数), 626;
see *function object* (参见函数对象), 335

G

Gang of Four (GoF) (四人帮 (GoF)), 613
general utility algorithms (通用实用程序算法), 417
generate algorithm (**generate**算法), 369
generate_n algorithm (**generate_n**算法), 369
generator (发生器), 337, 369
generic algorithms (通用算法), 325
get pointer (获取指针), 177
get( ), 170;
overloaded versions (重载版), 165
getline( ), 164, 171
getline( ), for **strings** (getline( ), 针对**strings**), 129
getPriority( ), 711
GoF, Gang of Four (GoF, 四人帮), 613
goodbit (正常标志位), 166
greater function object (**greater**函数对象), 338, 341, 371
greater_equal function object (**greater_equal**函数对象), 341
Guard template, ZThread (concurrency) (Guard 模板, ZThread (并发)), 721

H

handler, exception（处理器，异常），20
handshaking, between concurrent tasks（握手，并发任务间的），742
hash_map non-standard container（**hash_map**非标准的容器），539
hash_multimap non-standard container（**hash_multimap**非标准的容器），539
hash_multiset non-standard container（**hash_multiset**非标准的容器），539
hash_set non-standard container（**hash_set**非标准的容器），539
heap operations（堆运算），403
hex，187
hierarchy, object-based（层次结构，基于对象的），573

I

I/O:（输入输出）:，162;
console（控制台），162;
interactive 交互式，162;
raw（原始的），165;
threads, blocking（线程，阻塞），737
i18n,see *internationalization*(i18n，参见国际化)，216
ifstream，156，168，174
ignore( )，170
imbue( )，220
implementation inheritance（实现继承），579
includes algorithm（**includes**算法），400
inclusion model, of template compilation（包含模型，模板编译的），315
incomplete type（不完全类型），163
in-core formatting（内核格式化），179
inheritance:（继承）:，614;
design patterns（设计模式），614;
diamond（菱形），588;
hierarchies（层次结构），573;
implementation（实现），579;
interface（界面/接口），575
inheritance, multiple（继承，多重），573，673;
avoiding（回避），603;
dominance（支配），601;
name lookup（名字查询），599;
runtime type identification（运行时类型识别），560，563，570
initialization:（初始化）:，621;
controlling initialization order（控制初始化顺序），621;
lazy（惰性），620;
object（对象），596;
Resource Acquisition Is Initialization (RAII)（资源分配式初始化（RAII）），32，36，582;
zero initialization（零初始化），522
inner class idiom, adapted from Java（内部类习语，由Java改编而来），671
inner_product algorithm（**inner_product**算法），414
inplace_merge algorithm（**inplace_merge**算法），399
input iterator（输入迭代器），446
input_iterator_tag，447
InputIterator，363
insert( )，448
insert_iterator，372，448，482
inserter( )，372，418，448
inserter, stream（插入符，流），158
instantiation, template（实例化，模板），260
interactive I/O（交互式I/O），162
interface:（界面/接口）:，576;
class（类），576;
command-line（命令行），162;
extending an（扩展一个），603;
inheritance（继承），575;
repairing an interface with multiple inheritance（修复一个拥有多重继承的接口），603;
responsive user（响应用户），700
internationalization（国际化），216
interrupt( ), threading（**interrupt( )**，线程），735
interrupted status, threading（中断状态，线程），739
Interrupted_Exception, threading（**Interrupted_Exception**，线程），739
invalid_argument exception class（**invalid_argument**异常类），40
invalidation, iterator（无效，迭代器），463
invariant:（不变量）:，69;
class（类），69;
code（代码），63;
loop（循环），64
ios:app，172;
ate，172;
basefield，188;
beg，176;
binary，172，214;
cur，176;
end，176;
failbit（失败标志位），160;
fill( )，190;
in，171;

out，172;
precision( )，190;
showbase，187;
showpoint，187;
showpos，187;
skipws，187;
trunc，172;
unitbuf，187;
uppercase，187;
width( )，190
ios_base，157
iostate，168
iostreams（输入输出流），156;
applicator（应用算子），200;
automatic（自动的）189;
badbit（致命错误标志位），165;
binary mode（二进制模式），172，214;
buffering（缓冲），173;
clear function（**clear**函数），166，175;
dec manipulator（**dec**操纵算子），195;
endl manipulator（**endl**操纵算子），195;
eofbit（文件结束位），166;
errors（错误）165;
exceptions（异常），167;
exceptions function（**exceptions**函数），167;
extractor（提取符），158;
fail function（**fail**函数），175;
failbit（失败标志位），166;
failure exception type（**failure**异常类型），167;
files（文件），162;
fill( )，190;
fixed，196;
flags( )，186;
flush，195;
fmtflags type（**fmtflags**类型），186;
format fields（格式化域），188;
format flags（格式化标志），186;
formatting（格式化），186;
fseek( )，176;
get( )，170;
getline( )，171;
goodbit（正常标志位），166;
hex manipulator（**hex**操纵算子），195;
ignore( )，170;
imbue( )，220;
inserter（插入器），158;
internal，196;
ios::basefield，188;
ios::dec，189;
ios::fixed，189;
ios::hex，189;
ios::internal，190;
ios::left，190;
ios::oct，189;
ios::right，190;
ios::scientific，189;
iostate type（**iostate**类型），168;
left，196;
locales（区域），216;
manipulators（操纵算子），194;
manipulators, creating（操纵算子，创建），199;
narrow（窄字符），216;
narrow function（**narrow**函数），218;
noshowbase，195;
noshowpoint，196;
noshowpos，195;
noskipws，196;
nouppercase，195;
oct manipulator（**oct**操纵算子），195;
open modes（打开模式），171;
operator<<，158;
operator>>，158;
positioning（定位），175;
precision( )，190，213;
resetiosflags manipulator（**resetiosflags**操纵算子），196;
right，196;
scientific，196;
seeking in（在...中查找），175;
setbase manipulator（**setbase**操纵算子），197;
setf( )，187，188，213;
setfill manipulator（**setfill**操纵算子），197;
setiosflags manipulator（**setiosflags**操纵算子），196;
setprecision manipulator（**setprecision**操纵算子），197;
setstate function（**setstate**函数），166;
setw manipulator（**setw**操纵算子），197，213;
showbase，195;
showpoint，196;
showpos，195;
skipws，196;
smanip type（**smanip**类型），201;
string I/O with（字符串输入/输出用于），179;
text mode（文本模式），172;
threads, colliding output（线程，冲突输出），727;

unsetf( ), 188;
uppercase, 195;
wide (宽的), 216;
widen function (**widen**函数), 218;
width( ), 190;
write( ), 213;
ws manipulator (**ws**操纵算子), 195
istream (输入输出流), 156;
get( ), 164;
getline( ), 164;
read( ), 165;
seekg( ), 176;
tellg( ), 176
istream_iterator, 333, 446, 450
istreambuf_iterator, 446, 451, 481
istringstream, 156, 179
iter_swap algorithm (**iter_swap**算法), 419, 457
iterator (迭代器), 429, 615;
adapting a class to produce (调整一个类来生成), 637;
adaptor (适配器), 487;
bidirectional (双向), 446;
categories (种类), 446;
forward (前向), 446;
input (输入), 446;
invalidation (无效), 463;
istream (输入流), 333;
ostream (输出流), 332;
output (输出), 446;
past-the-end (超越末尾的), 443;
random-access (随机存取的), 446;
reverse (反向), 445;
stream (流), 331;
stream iterator (流迭代器), 450;
tag (标记符), 447;
traits (特征), 366
iterator_traits, 366

J

Josuttis, Nico, 101

K

King, Jamie, 10
Koenig, Andrew, 274
Kreft, Klaus, 314, 780

L

Lajoie, Josee, 60
Langer, Angelika, 314, 780
lazy initialization (惰性初始化), 620, 634
length_error exception class (**length_error**异常类), 40
less function object (**less**函数对象), 341
less_equal function object (**less_equal**函数对象), 341
LessThanComparable, 337
letter, envelope and letter classes (信件、信封和信件类), 655
lexicographical_compare algorithm (**lexicographical_compare**算法), 385
library: (库):, 101;
documentation (文档), 101;
maintaining class source (维护类库的源代码), 204
line input (行输入), 162
linear search (线性查找), 377
Linux, and ZThreads (Linux和ZThreads), 696
list, 434, 471;
merge( ), 474;
remove( ), 474;
reverse( ), 472;
sort( ), 472;
unique( ), 474;
vs. **set** (跟**set**对比), 476
locale (区域), 216, 218;
collate category (**collate**类项), 219;
ctype category (**ctype**类项), 219;
facet (领域), 220;
iostreams (输入输出流), 216;
messages category (**messages**类项), 219;
monetary category (**monetary**类项), 219;
money_get facet (**money_get**领域), 220;
money_punct facet (**money_punct**领域), 220;
money_put facet (**money_put**领域), 220;
numeric category (**numeric**类项), 219;
time category (**time**类项), 219;
time_get facet (**time_get**领域), 220;
time_put facet (**time_put**领域), 220
localtime( ), 213
logic_error class (**logic_error**类), 38
logical_and function object (**logical_and**函数对象), 341
logical_not function object(**logical_not** 函数对象), 341
logical_or function object (**logical_or**函数对象), 341

longjmp( ), 16
loop:（循环）:, 64;
  invariant（不变量）, 64;
  unrolling（分解）, 301
lower_bound algorithm（**lower_bound**算法）, 395

M

machine epsilon（机器误差）, 181
maintaining class library source （维护类库的源代码）, 204
make_heap algorithm（**make_heap**算法）, 404, 499
make_pair( ), 417
manipulating sequences（操作序列）, 372
manipulators（操纵算子）, 160;
  creating（创建）, 199;
  iostreams formatting（输入输出流格式化）, 194;
  with arguments（带参数的）, 196
map（映射）, 521;
  keys and values（关键字与值）, 521
max algorithm（**max**算法）, 419
max_element algorithm（**max_element**算法）, 380
mem_fun member pointer adaptor（**mem_fun**成员指针适配器）, 355
mem_fun_ref member pointer adaptor（**mem_fun_ref**成员指针适配器）, 355
member templates（成员模板）, 242;
  vs. **virtual**（对**virtual**）, 245
memory leaks（内存泄漏）, 90
memory management, and threads（内存管理，和线程）, 711
merge algorithm（**merge**算法）, 399
merging algorithms（合并算法）, 398
Messenger design pattern (idiom)（信使设计模式（习语））, 617
metaprogramming（元编程）, 297;
  compile-time assertions（编译时断言）, 304;
  compile-time looping（编译时循环）, 299;
  compile-time selection（编译时选择）, 303;
  loop unrolling（循环分解）, 301;
  Turing completeness of（图灵机完全的）, 298
Meyer, Bertrand, 68
Meyers, Scott, 60, 623
min algorithm（**min**算法）, 418
min_element algorithm（**min_element**算法）, 379
minus function object（**minus**函数对象）, 340
mismatch algorithm（**mismatch**算法）, 386
mixin:（混入）:, 579;
  class（类）, 579;
  parameterized（参数化）, 583
model-view-controller (MVC)(模型视图控制器(MVC)), 667
modulus function object（**modulus**函数对象）, 341
money_get, 220
money_punct, 220
money_put, 220
multimap, 523
Multiple Dispatching design pattern（多重派遣设计模式）, 679
multiple inheritance（多重继承）, 573, 673;
  avoiding（避免）, 603;
  dominance（占优势的）, 601;
  duplicate subobjects（重复子对象）, 585;
  exception handling（异常处理）, 56;
  name lookup（名字查询）, 599;
  repairing an interface（修复一个接口）, 603;
  runtime type identification（运行时类型识别）, 560, 563, 570
multiplies function object（**multiplies**函数对象）, 341
multiprocessor machine, and threading （多处理机，和线程）, 692
multiset（多重集合）, 527;
  equal_range( ), 529
multitasking（多任务）, 691
multithreading（多线程）, 691;
  drawbacks（缺点）, 771;
  ZThread library for C++（C++的ZThread库）, 694
mutex:（互斥锁）, 721;
  simplifying with the Guard template（用Guard模板简化）, 721;
  threading（线程）, 742;
  ZThread **FastMutex**, 731
mutual exclusion, in threads（相斥，在线程中）, 719
Myers, Nathan, 11, 251, 285, 452, 481, 482

N

naked pointers, and exception handling（悬挂指针，和异常处理）, 30
name lookup, and multiple inheritance（名字查询，和多重继承）, 599
name( ), RTTI function（**name**, RTTI函数）, 559
narrow streams（窄字符流）, 216

narrow( ), 218
negate function object (**negate**函数对象), 341
new, placement (**new**, 定位), 91
newline, differences between DOS and Unix (换行符, DOS与Unix的差别), 172
next_permutation algorithm (**next_permutation**算法), 373
not_equal_to function object (**not_equal_to**函数对象), 341
not1 function object adaptor (**not1**函数对象适配器), 339, 352
nth_element algorithm (**nth_element**算法), 394
numeric algorithms (数值算法), 413
numeric_limits, 203, 285

O

object: (对象): , 596;
  initialization (初始化), 596;
  object-based hierarchy (基于对象的层次结构), 573;
  slicing, and exception handling (切割, 和异常处理), 23
Observable, 668
Observer design pattern (观察者设计模式), 667
oct, 187
ofstream, 156, 168
one-definition rule (一次定义规则), 622
open modes, iostreams (打开模式, 输入输出流), 171
operator new( ), 90
operator void*( ) , for streams (**operator void*( )** 对下流), 167
operator( ), 229, 335, 339
operator++( ), 234
optimization, throughput, with threading (优化, 信息的吞吐量, 采用线程处理), 692
order: (顺序): , 621;
  controlling initialization (控制初始化), 621;
  of constructor and destructor calls (构造函数与析构函数调用的), 562
ordering: (排序): , 393;
  algorithms (算法), 393;
  strict weak (严格弱的), 337
ostream, 156;
  fill( ), 160;
  manipulators (操纵算子), 160;
  seekp( ), 176;
  setfill( ), 160;
  setw( ), 160;
  tellp, 176;
  write( ), 165
ostream_iterator, 332, 365, 446, 451
ostreambuf_iterator, 446, 451
ostringstream, 156, 179;
  str( ), 182
out_of_range exception class (**out_of_range**异常类), 40
output: (输出): , 446;
  iterator (迭代器), 446;
  stream formatting (流格式化), 186
output_iterator_tag, 447;
OutputIterator, 363
overhead, exception handling (开销, 异常处理), 58
overloading, function template (重载, 函数模板), 249

P

parameter, template(参数, 模板), 227
parameterized mixin (参数化混入), 583
Park, Nick, 257
partial ordering: (半有序): , 263;
  class templates (类模板), 263;
  function templates (函数模板), 259
partial specialization, template (半特化, 模板), 263
partial_sort algorithm (**partial_sort**算法), 394
partial_sort_copy algorithm (**partial_sort_copy**算法), 394
partial_sum algorithm (**partial_sum**算法), 414
partition algorithm (**partition**算法), 374
past-the-end iterator (超越末尾的迭代器), 443
patterns, design patterns (模式, 设计模式), 613
perror( ), 16
philosophers, dining, and threading (哲学家, 进餐和线程), 764
placement **new** (定位**new**), 91
Plauger, P. J., 101
plus function object (**plus**函数对象), 340
pointer to member adaptor: (指向成员适配器的指针): , 355;
  mem_fun, 355;
  mem_fun_ref, 355
pointer, smart (指针, 智能的), 437
pointer_to_binary_function function object adaptor (**pointer_to_binary_function**函数对象适配器), 353
pointer_to_unary_function function object adaptor (**pointer_to_unary_function**函数对象适配器), 352

policies（策略），291
policy class（策略类），293
polymorphism（多态性），564
PoolExecutor (Concurrency)（对象池执行器（并发）），703
pop_heap algorithm（**pop_heap**算法），404，499
POSIX standard（POSIX标准），145
postcondition（后置条件），68
precision( )，213
precondition（前置条件），68
predicate（判定函数），329；
  binary（二元的），337；
  unary（一元的），337
prev_permutation algorithm（**prev_ permutation** 算法），373
printf( )，154；
  error code（错误代码），15
priority, thread（优先权/优先级，线程），709
priority_queue，496；
  as a heap（作为一个堆），499；
  pop( )，500；
private constructor（私有构造函数），620
process, threading and（进程，线程和），691
producer-consumer, threading（生产者消费者，线程），747
programming paradigms（编程范例），573
protected constructor（保护的构造函数），581
Proxy design pattern（代理设计模式），632
ptr_fun function pointer adaptor（**ptr_fun**函数指针适配器），351
pure virtual function（纯虚函数），576
push_back( )，434，448，482
push_front( )，434，448
push_heap algorithm（**push_heap**算法），404，499
put pointer（put指针），176

Q

qualified name（限定名），274，278
queue，491
queues, thread, for problem-solving（队列，线程，为了解决问题），750
quicksort（快速排序），366

R

race condition（竞争条件），717
RAII，32，36，582
raise( )，16
rand( )，215
RAND_MAX，215
random_shuffle algorithm（**random_shuffle**算法），374
random-access iterator（随机存取迭代器），446
RandomAccessIterator，364
range, of sequence（范围，序列的），326
raw byte I/O（原始字节I/O），165
raw_storage_iterator，446，452
rbegin( )，445，448
rdbuf( )，174
read( )，165
refactoring（重构），70
reference counting（引用计数），582，634；
  ZThreads (Concurrency)（ZThreads（并发）），712
references:（引用）:，557；
  bad_cast，557；
  exception handling（异常处理），34，56
remove algorithm（**remove**算法），389
remove_copy algorithm（**remove_copy**算法），389
remove_copy_if algorithm（**remove_copy_if**算法），329，339，350，390
remove_if algorithm（**remove_if**算法），389
removing elements, algorithm（删除元素，算法），389
rend( )，445，448
reordering, stable and unstable（再排序，稳定和不稳定），366
replace algorithm（**replace**算法），380
replace_copy algorithm（**replace_copy**算法），380
replace_copy_if algorithm（**replace_copy_if**算法），330，380
replace_if algorithm（**replace_if**算法），330，380
reporting errors in book（报告教材中的错误），10
requirements（要求），70
reserve( )，458
resize( )，456
Resource Acquisition Is Initialization (RAII)（资源获得式初始化（RAII）），32，36，582
responsive user interfaces（响应用户界面），700
result_type，342
resumption, vs.termination, exception handling（恢复，对比结束，异常处理），22
rethrow, exception（再抛掷，异常），26，52
reverse algorithm（**reverse**算法），372
reverse_copy algorithm（**reverse_copy**算法），372

reverse_iterator, 445, 448, 487
reversible container（可逆容器）, 445
rope non-standard string class（**rope**非标准字符串类）, 539
rotate algorithm（**rotate**算法）, 373
rotate_copy algorithm（**rotate_copy**算法）, 373
Runnable, 696
runtime cast（运行时类型转换）, 551
runtime stack（运行时的栈）, 228
runtime type identification（运行时类型识别）, 551;
  casting to intermediate levels（类型转换到中间层类型）, 560;
  const and **volatile** and（**const**和**volatile**和）, 558;
  difference between **dynamic_cast** and **typeid**（**dynamic_cast**和**typeid**之间的差别）, 561;
  efficiency（效率）, 565;
  mechanism & overhead（机制和开销）, 570;
  misuse（误用）, 564;
  multiple inheritance（多重继承）, 560, 563, 570;
  templates and（模板和）, 562;
  type_info, 570;
  type_info class（**type_info**类）, 557;
  type_info::before( ), 559;
  type_info::name( ), 559;
  typeid operator（**typeid**运算符）, 557;
  void pointers（空类型指针）, 561;
  VTABLE（虚函数表）, 570;
  when to use it（什么时候用它）, 564
runtime_error class（**runtime_error** 类）, 38

S

Saks, Dan, 282
Schwarz, Jerry, 201
search algorithm（**search**算法）, 379
search_n algorithm（**search_n**算法）, 379
searching and replacing algorithms（查找和替换算法）, 377
second_argument_type, 342
seekg( ), 176
seeking in iostreams（在输入输出流中查找）, 175
seekp( ), 176
separation model, of template compilation（分离模型，模板编译的）, 319
sequence:（序列）:, 470;
  at( ), 470;
  container（容器）, 433;
  converting between sequences（序列间的转换）, 467;
  deque, 465;
  erase( ), 457;
  expanding with **resize( )**（用**resize( )**进行扩展）, 456;
  insert( ), 457;
  list, 471;
  operations（运算）, 454;
  operator[ ], 471;
  random-access（随机存取）, 470;
  swap( ), 457;
  swapping sequences（交换序列）, 477;
  vector, 457
serialization:（串行化）:, 215;
  object（对象）, 215;
  thread（线程）, 750
set, 479, 513;
  find( ), 480;
  operations（运算）, 400;
  ordering of（...的排序）, 480;
  STL set class example（STL集合类例子）, 432;
  vs. **list**（对，**list**）, 476
set_difference algorithm（**set_difference**算法）, 401
set_intersection algorithm（**set_intersection**算法）, 401
set_symmetric_difference algorithm（**set_symmetric difference**算法）, 402
set_terminate( ), 27
set_unexpected( ), 41
set_union algorithm（**set_union**算法）, 401
setf( ), 187, 213
setjmp( ), 16
setPriority( ), 711
setw( ), 213
Sieve of Eratosthenes（Eratosthenes 筛选法）, 119
signal( ), 16, 53;
  threading（线程）, 734, 742
Singleton（单件）, 460, 619;
  implemented with curiously recurring template pattern（用奇特的递归模板模式实现）, 624;
  Meyers' Singleton, 623;
  ZThreads library (concurrency)（ZThreads库（并发））, 728
sleep( ), threading（**sleep( )**，线程）, 707, 734
slice ,**valarray**（切片，**valarray**）, 542
slicing, object slicing and exception handling（切割，对

象切割和异常处理)，23
slist non-standard container（**slist**非标准容器），539
Smalltalk（Smalltalk程序设计语言），573
smanip，201
smart pointer（智能指针），437
software quality（软件质量），63
sort algorithm（**sort**算法），366，393
sort_heap algorithm（**sort_heap**算法），404
sorting algorithms（排序算法），393
specialization:（实例化）:，261;
function template（函数模板），261;
template（模板），260
specification, exception（规格说明，异常），40
srand( )，214
stable reordering（稳定排序），366
stable_partition algorithm（**stable_partition**算法），374
stable_sort algorithm（**stable_sort**算法），366，393
stack，487;
exception safety of（...的异常安全），489;
pop( )，489;
push( )，489;
top( )，489
stack frame（栈结构），58
stack unwinding（栈反解），19
Standard C（标准C），9
Standard C++（标准C++），9;
concurrency（并发），694;
exception types（异常类型），38
State design pattern（状态设计模式），634
stdio，151
STL extensions（STL的扩展），538
Strategy design pattern（策略设计模式），640
strcmp( )，217
stream（流），156;
errors（错误），165;
iterator（迭代器），331，450;
output formatting（输出格式化），186;
state（状态），165
streambuf，173;
get( )，174;
rdbuf( )，174
streampos，176
strict weak ordering（严格弱序），337
StrictWeakOrdering，374，403
string，103;
append( )，110;
at( )，132;
c_str( )，131;
capacity( )，111;
case-insensitive search（忽略大小写的查找），120;
character traits（字符特征），134;
compare( )，131;
concatenation（连接），117;
empty( )，356;
erase( )，126;
find( )，115;
find_first_not_of( )，118;
find_first_of( )，118;
find_last_not_of( )，118;
find_last_of( )，118;
getline( )，129;
indexing operations（检索操作），133;
insert( )，110;
iterator（迭代器），108;
length( )，111;
memory management（内存管理），110，114;
npos member（**npos**成员），114;
operator!=，129;
operator[]，132;
operator+，117;
operator+=，117;
operator<，129;
operator<=，129;
operator==，129;
operator>，129;
operator>=，129;
reference-counted（引用计数），104
relational operators（关系运算符），129;
replace( )，112;
reserve( )，111;
resize( )，111;
rfind( )，118;
size( )，111;
stream I/O（输入输出流），156;
substr( )，107;
swap( )，132;
transforming strings to typed values（将字符串转化成有类型的值），181
string streams（字符串流），179
stringbuf，183
stringizing, preprocessor operator（字符串化，预处理器运算符），193
Stroustrup, Bjarne，101
struct tm，213
structural design patterns（结构化设计模式），615
subobject, duplicate subobjects in multiple inheritance

(子对象，多重继承中重复的子对象)，585
subtasks（子任务），691
suite, test（套件，测试），79
surrogate, in design patterns（代理，在设计模式中），631
swap algorithm（**swap**算法），419
swap_ranges algorithm（**swap_ranges**算法），373
synchronization:（同步）:，732;
(concurrency) example of problem from lack of synchronization（(并发）缺乏同步性的问题例子），732;
blocking（阻塞），734;
thread（线程），719
Synchronization_Exception, ZThread library（**Synchronization_Exception**，ZThread库），698，703
synchronized, threading, wrapper for an entire class（同步，线程，对整个类的封装器），723
SynchronousExecutor(Concurrency)（**Synchronous Executor**（并发）），705

T

tag, iterator tag classes（标识符，迭代器标识符类），447
task, defining for threading（任务，为线程处理定义），696
tellg( )，176
tellp( )，176
template:（模板）:，274;
argument-dependent lookup in（在...中的关联参数查找），274;
code bloat, preventing（代码膨胀，防止），268;
compilation（编译），274;
compilation models（编译模型），315;
compilation, two-phase（编译，二阶段），274;
curiously recurring template pattern（奇特的递归模板模式），294;
default arguments（默认参数），230;
dependent names in（...中的关联名称），274;
explicit instantiation（显式实例化），316;
export，319;
expression templates（表达式模板），308;
friend template（友元模板），284;
friends（友元），279;
function（函数），245;
idioms（习语），285;
inclusion compilation model（包含编译模型），315;
instantiation（实例化），260;
keyword（关键字），240;
member（成员），242;
member, and **virtual** keyword（成员，和virtual关键字），245;
metaprogramming（元编程），297;
name lookup issues（名字查找问题），273;
names in（...中的名字），273;
non-type parameters（无类型参数），228;
parameters（参数），227;
partial ordering of class templates（类模板的半有序），263;
partial ordering of function templates（函数模板的半有序），259;
partial specialization（半特化），263;
policy-based design（基于策略的设计），291;
qualified names in（...中的限定名称），274，278;
runtime type identification and（运行时类型识别和...），562;
separation compilation model（分离编译模型），319;
specialization（特化），260;
template template parameters（模板的模板参数），232;
traits（特征），285
Template Method design pattern（模板方法设计模式），639
terminate( )，27，44;
uncaught exceptions（不捕获异常），26
terminating threads（线程结束），735
termination problem, concurrency（结束问题，并发），727
termination, vs. resumption, exception handling（结束还是恢复，异常处理），22
test:（测试）:，71;
automated unit testing（自动单元测试），71;
Boolean expressions in testing（测试中的布尔表达式），72;
framework（框架），75;
suite（套件），79;
test-first programming（先测试编程），71;
unit（单元），70
Test class（**Test**类），76
TestSuite framework（**TestSuite**框架），75
text processing（文本处理），103
thread（线程），691;
atomic operation（原子操作），732;
blocked（被阻塞），734;
broadcast( )，734，742，757;
busy wait（忙等待），732，743;
Cancelable, ZThread library class（**Cancelable**，ZThread库类），717;

colliding over resources, improperly accessing shared resources（资源冲突，对共享资源的访问失当），715;
concurrency（并发），691;
Condition class for **wait( )** and **signal( )**（对于**wait( )**和**signal( )**的**Condition**类），742;
cooperation（协作），741;
dead state（死亡状态），734;
deadlock（死锁），720，764;
deadlock, and priorities（死锁，和优先权），709;
dining philosophers（进餐的哲学家），764;
drawbacks（缺点），771;
example of problem from lack of synchronization（缺乏同步性的问题例子），732;
getPriority( )，711;
handshaking between tasks（任务间的握手），742;
I/O and threads, blocking（输入输出和线程，阻塞），737;
interrupt( )，735;
interrupted status（中断状态），739;
Interrupted_Exception，739;
iostreams and colliding output（输入输出流和冲突输出），727;
memory management（内存管理），711;
multiple, for problem-solving（多，为了解决问题），741;
mutex, for handshaking（互斥锁，为了握手）742;
mutex, simplifying with the **Guard** template（互斥锁，用**Guard**模板简化操作），721;
new state（新状态），734;
order of task shutdown（任务停止执行的顺序），717;
order of thread execution（线程执行的顺序），708;
priority（优先权），709;
producer-consumer（生产者-消费者），747;
queues solve problems（用队列解决问题），750;
race condition（竞争条件），717;
reference counting（引用计数），712;
reference counting with **CountedPtr**（用**Countedptr**引用计数），714;
runnable state（可运行状态），734;
serialization（串行），750;
setPriority( )，711;
sharing resources（共享资源），711;
signal( )，734，742;
sleep( )，707，734;
states（状态），734;
synchronization（同步），719;
synchronization and blocking（同步和阻塞），734;
synchronized wrapper for an entire class（对整个类的同步封装器），723;
termination（终止），735;
termination problem（终止问题），727;
thread local storage（线程本地存储），724;
threads and efficiency（线程和效率），693;
TQueue, solving threading problems with（TQueue，用…解决线程问题），750;
wait( )，734，742;
when to use threads（什么时候使用线程），771;
yield( )，706;
ZThread **FastMutex**，731
ThreadedExecutor (Concurrency)（**Threaded Executor**（并发）），702
throughput, optimize（信息的存吐量，优化），692
throw，19
throwing an exception（抛出一个异常），18
time( )，214
time_get，220
time_put，220
tolower，252
toupper，252
TQueue, solving threading problems with（**TQueue**，用…解决线程问题），750
trace:（追踪）:，88;
file（文件），88;
macro（宏），87
traits（特征），285;
iterator（迭代器），366
transform algorithm（**transform**算法），252，347，349，355，405
transforming character strings to typed values（将字符串转换成有类型的值），181
try，20
try block（try块），20;
function-level（函数级），36
type:（类型）:，23;
automatic type conversions and exception handling（自动类型转换和异常处理），23;
deduction, of function template parameters（推断，函数模板参数的），245;
incomplete（不完全的），163;
runtime type identification (RTTI)（运行时类型识别（RTTI）），551
type_info:name function（**type_info:name**函数），244;
structure（结构），570
type_info class（**type_info**类），557
type_info::before( )，559

type_info::name( ), 559
typeid operator (**typeid**运算符), 244, 557;
difference between **dynamic_cast** and **typeid**, runtime type identification (**dynamic_cast**和**typeid**之间的差别, 运行时类型识别), 561
typename:, 237;
keyword (关键字), 237;
typedef, 240;
versus **class** (...对 class), 240
typing, weak (输入, 弱的), 579

U

unary function (一元函数), 337
unary predicate (一元判定函数), 337
unary_composer non-standard function object (**unary_composer**非标准函数对象), 360
unary_function, 342, 352;
argument_type, 342;
result_type, 342
unary_negate function object (**unary_negate**函数对象), 341
uncaught exceptions (不捕获异常), 26
uncaught_exception( ), 52
unexpected( ), 41
Unicode (统一代码), 216
unique algorithm (**unique**算法), 390
unique_copy algorithm (**unique_copy**算法), 390
unit buffering (单元缓冲), 188
unit test (单元测试), 70
unstable reordering (不稳定的再排序), 366
upcast (向上类型转换), 603
upper_bound algorithm (**upper_bound** 算法), 395
Urlocker, Zack, 608
user interface, responsive, with threading (用户界面, 响应, 用于线程处理), 692, 700
utility algorithms (实用程序算法), 417

V

valarray, 540;
slice (切片), 542
value_type, 450
van Rossum, Guido, 773
Vandevoorde, Daveed, 308
vector, 457;
reserve( ), 458
vector of change (变化的向量), 614
vector<bool>, 263, 506, 511
Veldhuizen, Todd, 308
virtual: (虚的):, 563, 589;
base class (基类), 563, 589;
base, initialization of (基类, …的初始化), 592;
destructor (析构函数), 581;
function table (函数表), 654;
pure virtual functions (纯虚函数), 576;
simulating virtual constructors (模拟虚构造函数), 654;
virtual functions inside constructors (构造函数内的虚函数), 654
Visitor design pattern (访问者设计模式), 683
void (空类型), 561
VPTR (虚指针), 654
VTABLE (虚函数表), 654;
runtime type identification (运行时类型识别), 570

W

wait( ), threading (**wait( )**, 线程), 734, 742
wchar_t, 216
wcscmp( ), 217
weak typing (弱输入), 579
web servers, multiprocessor (网络服务器, 多处理机), 692
wide: (宽):, 216;
character (字符), 216;
stream (流), 216;
stream function, **wcscmp( )** (流函数, **wcscmp( )**), 217
widen( ), 218
Will-Harris, Daniel, 11
wrapping, class (封装, 类), 151
write( ), 165, 213
ws, 195

X

XP, Extreme Programming (极限编程), 71, 615

Y

yield( ), threading (**yield( )**, 线程), 706

Z

zero initialization (零初始化), 522
Zolman, Leor, 320
ZThread:, 717;
Cancelable class (**Cancelable**类), 717;
Executors (执行器), 702;
installing the library (安装该库), 695;
multithreading library for C++ (C++的多线程库), 694

type_info::name( ), 559
typeid operator (**typeid**运算符), 244, 557;
difference between **dynamic_cast** and **typeid**, runtime type identification (**dynamic_cast**和**typeid**之间的差别, 运行时类型识别), 561
typename:, 237;
keyword (关键字), 237;
typedef, 240;
versus **class** (...对 class), 240
typing, weak (输入, 弱的), 579

U

unary function (一元函数), 337
unary predicate (一元判定函数), 337
unary_composer non-standard function object (**unary_composer**非标准函数对象), 360
unary_function, 342, 352;
argument_type, 342;
result_type, 342
unary_negate function object (**unary_negate**函数对象), 341
uncaught exceptions (不捕获异常), 26
uncaught_exception( ), 52
unexpected( ), 41
Unicode (统一代码), 216
unique algorithm (**unique**算法), 390
unique_copy algorithm (**unique_copy**算法), 390
unit buffering (单元缓冲), 188
unit test (单元测试), 70
unstable reordering (不稳定的再排序), 366
upcast (向上类型转换), 603
upper_bound algorithm (**upper_bound** 算法), 395
Urlocker, Zack, 608
user interface, responsive, with threading (用户界面, 响应, 用于线程处理), 692, 700
utility algorithms (实用程序算法), 417

V

valarray, 540;
slice (切片), 542
value_type, 450
van Rossum, Guido, 773
Vandevoorde, Daveed, 308
vector, 457;
reserve( ), 458
vector of change (变化的向量), 614
vector<bool>, 263, 506, 511
Veldhuizen, Todd, 308
virtual: (虚的):, 563, 589;
base class (基类), 563, 589;
base, initialization of (基类, …的初始化), 592;
destructor (析构函数), 581;
function table (函数表), 654;
pure virtual functions (纯虚函数), 576;
simulating virtual constructors (模拟虚构造函数), 654;
virtual functions inside constructors (构造函数内的虚函数), 654
Visitor design pattern (访问者设计模式), 683
void (空类型), 561
VPTR (虚指针), 654
VTABLE (虚函数表), 654;
runtime type identification (运行时类型识别), 570

W

wait( ), threading (**wait( )**, 线程), 734, 742
wchar_t, 216
wcscmp( ), 217
weak typing (弱输入), 579
web servers, multiprocessor (网络服务器, 多处理机), 692
wide: (宽):, 216;
character (字符), 216;
stream (流), 216;
stream function, **wcscmp( )** (流函数, **wcscmp( )**), 217
widen( ), 218
Will-Harris, Daniel, 11
wrapping, class (封装, 类), 151
write( ), 165, 213
ws, 195

X

XP, Extreme Programming (极限编程), 71, 615

Y

yield( ), threading (**yield( )**, 线程), 706

Z

zero initialization (零初始化), 522
Zolman, Leor, 320
ZThread:, 717;
Cancelable class (**Cancelable**类), 717;
Executors (执行器), 702;
installing the library (安装该库), 695;
multithreading library for C++ (C++的多线程库), 694