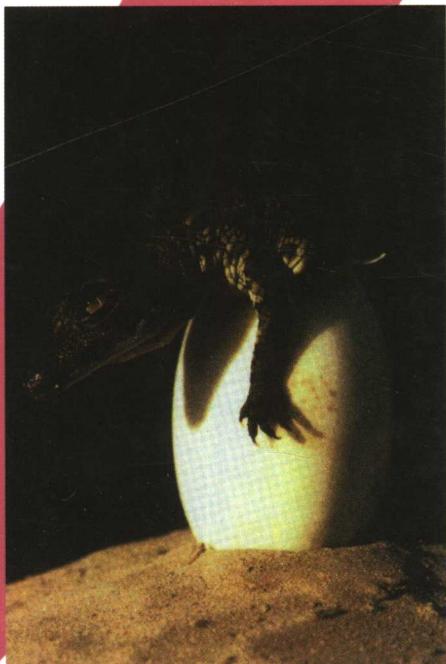


# Effective STL 中文版

50条有效使用STL  
的经验

[美] Scott Meyers 著  
潘爱民 陈铭 邹开红 译



清华大学出版社

ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

# Effective STL 中文版

“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,但是要想学会用好STL却不容易。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,STL程序员可以最大限度地使用STL。

其他的书只是描述了STL中有些什么内容,而本书则讲述了如何使用STL。本书共有50条指导原则,在讲述每一条指导原则的时候,Scott Meyers都提供了透彻的分析和深刻的实例,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,其中涉及到的容器有:标准STL容器(例如vector和list)、非标准的STL容器(例如hash\_set和hash\_map),以及非STL容器(例如bitset)。
- 一些改进效率的技术,通过它们可以最大程度地提高STL(以及使用STL的程序)的效率。
- 一些深层次的知识,其中涉及到迭代器、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,根据它们行为方式上的微妙差异,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。



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。

ISBN 7-302-12695-X



9 787302 126959 >

定价: 30.00 元



# Effective STL 中文版

50 条有效使用 STL 的经验

[美] Scott Meyers 著

潘爱民 陈 铭 邹开红 译

清华大学出版社  
北京

Simplified Chinese edition copyright © 2005 by **PEARSON EDUCATION ASIA LIMITED and TSINGHUA UNIVERSITY PRESS.**

Original English language title from Proprietor's edition of the Work.

Original English language title: Effective STL by Scott Meyers, Copyright © 2001

EISBN: 0-201-74962-9

All Rights Reserved.

Published by arrangement with the original publisher, Pearson Education, Inc., publishing as **Addison-Wesley**.  
This edition is authorized for sale only in the People's Republic of China (excluding the Special Administrative Region of Hong Kong and Macao).

本书中文简体翻译版由 Pearson Education 授权给清华大学出版社在中国境内(不包括中国香港、澳门特别行政区)出版发行。

北京市版权局著作权合同登记号 图字: 01-2004-5452

版权所有，翻印必究。举报电话: 010-62782989 13501256678 13801310933

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

#### 图书在版编目(CIP)数据

Effective STL 中文版：50 条有效使用 STL 的经验 / (美) 迈耶斯 (Meyers,S.) 著；潘爱民，陈铭，邹开红译. —北京：清华大学出版社，2006.4

书名原文：Effective STL

ISBN 7-302-12695-X

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

中国版本图书馆 CIP 数据核字 (2006) 第 020616 号

出 版 者：清华大学出版社 地 址：北京清华大学学研大厦  
<http://www.tup.com.cn> 邮 编：100084  
社 总 机：010-62770175 客户服务：010-62776969

责任编辑：汤斌浩

印 刷 者：北京中科印刷有限公司

装 订 者：北京市密云县京文制本装订厂

发 行 者：新华书店总店北京发行所

开 本：185×230 印张：13.75 字数：309 千字

版 次：2006 年 4 月第 1 版 2006 年 4 月第 1 次印刷

书 号：ISBN 7-302-12695-X/TP · 8109

印 数：1 ~ 4000

定 价：30.00 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的这三本姊妹作分别从各个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，需要仔细地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的源码规模并不大，但是它蕴含的思想非常深刻。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，但是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，应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，通过对这些用法的讲解，读者可以更加清楚地了解应该如何来看待这些与实现相关的知识。

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于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。在现有的STL书籍中，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` 两种容器有关的一些注意事项。

- 涉及到关联容器有更多的陷阱，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——哈希容器。
- 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完成了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，它使得 STL 中每一个算法都具有极强的扩展性，本书也特别讨论了涉及到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作了修订。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

北京大学燕北园

# 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, 却发现了许多我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, 而且也是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, 但有的却非常差。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仍然是一大困难。可能最沮丧的是, 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, 并且大多数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 背后的人们更是钦佩无比, 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, 我自己并不确定要有效地使用 STL 是否是可能的。

然后, 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, 尽管它的文档并不完整, 尽管编译器的诊断信息有如传输线上的噪音一样, 但是, 我的许多咨询客户正在使用 STL。而且, 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(请参考本书后面所附的参考资料目录),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,尽我的一份绵薄之力。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。但后来我改变了路线。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于 C++ 的理解。如果你有新的关于 STL 编程学习方法的建议,或者你对于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,或者其他原因的错误,我都会很高兴地在将来的印刷中向第一位报告错误的人表示感谢。请把你的指导建议、你的看法,以及你的批评意见发送到 [estl@aristeia.com](mailto:estl@aristeia.com)。

我为本书维护了一份自第一次印刷以来的修订纪录,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纪录: <http://www.aristeia.com/BookErrata/estl1/e-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。我通过该邮件列表来向那些对我的 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。在所有的帮助之中，有两个尤其重要。第一是 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都要多得多。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` (“clem”)、`comp.std.c++` 和 `microsoft.public_vc.stl`。差不多有十多年时间，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资料列表中。尤其让我受益良多的是 Josuttis 的 *The C++ Standard Library* [3]。

本书基本上是其他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 小节的基础之上。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++* [8] 中的第 37 条，以及 Kevlin 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 条中有关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，即，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。我窃取了 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“通过一个解除指针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 *DereferenceLess* 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 条则是建立在 Sutter 在 *C++ Report* 上的专栏“*Standard Library News: sets and maps*”[13]的基础之上的; 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 Use 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 哈希容器建立在 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受 *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 小节中的讨论。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 小节。第 31 条受到了 Josuttis 在 *The C++ Standard Library*[3] 的 5.6.1 小节中的材料的影响。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 在 clcm 上的帖子帮助我加深了对于所涉及到的各个问题的理解。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]、在 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,我使用了我的术语“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,而成员搜索操作则只检查每个元素的第一个组件(键)。第 45 条包含了众多 clcm 发贴者贡献的信息,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资料包含到本书中，而且他亲自对这些资料做了调整，使之更适合于本书。

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 在很早时候就根据本书的部分草稿提出了他们的建议，而下列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 为本书做了文字审查。

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。为此我要感谢 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，而不是 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。本书并没有太多地介绍函数对象，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、Jim Scheller、Carl Manaster、Herb Sutter、Albert Franklin、George King、Dave Miller、Harold Howe、John Fuller、Tim McCarthy、John Hershberger、Igor Mikolic-Torreira、Stephen Bergmann、Robert Allan Schwartz、John Potter、David Grigsby、Sanjay Pattni、Jesper Andersen、Jing Tao Wang、André Blavier、Dan Schmidt、Bradley White、Adam Petersen、Wayne Goertel、Gabriel Netterdag、Jason Kenny、Scott Blachowicz、Seyed H. Haeri、Gareth McCaughan、Giulio Agostini、Fraser Ross、Wolfram Burkhardt、Keith Stanley、Leor Zolman、Chan Ki Lok、Motti Abramsky、Kevlin Henney、Stefan Kuhlins、Phillip Ngan、Jim Phillips、Ruediger Dreier、Guru Chander、Charles Brockman，以及 Day Barr。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，现在已经是一位高级副总裁了)、Alicia Carey 和 Susannah Buzard(他的第 n 位和第 n+1 位助手)、John Fuller(产品协调人)、Karin Hansen(封面设计者)、Jason Jones(全才的技术高手，尤其是在 Adobe 开发的恐怖的排版软件方面)、Marty Rabinowitz(他们的老板，但是他自己也工作)，以及 Curt Johnson、Chanda Leary-Coutu 和 Robin Bruce(都是市场人才，但都非常友善)。

而 Abbi Staley 则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，她给了我鼓励。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 要吃香草味的。盛上一勺，放在杯子里，打包带走。

# 目 录

---

引言 .....	1
<b>第 1 章 容器 .....</b>	<b>9</b>
第 1 条：慎重选择容器类型。 .....	9
第 2 条：不要试图编写独立于容器类型的代码。 .....	12
第 3 条：确保容器中的对象拷贝正确而高效。 .....	16
第 4 条：调用 <code>empty</code> 而不是检查 <code>size()</code> 是否为 0。 .....	18
第 5 条：区间成员函数优先于与之对应的单元素成员函数。 .....	19
第 6 条：当心 C++ 编译器最烦人的分析机制。 .....	26
第 7 条：如果容器中包含了通过 <code>new</code> 操作创建的指针，切记在容器对象析构前将指针 <code>delete</code> 掉。 .....	28
第 8 条：切勿创建包含 <code>auto_ptr</code> 的容器对象。 .....	32
第 9 条：慎重选择删除元素的方法。 .....	34
第 10 条：了解分配子( <code>allocator</code> )的约定和限制。 .....	38
第 11 条：理解自定义分配子的合理用法。 .....	43
第 12 条：切勿对 STL 容器的线程安全性有不切实际的依赖。 .....	46
<b>第 2 章 <code>vector</code> 和 <code>string</code> .....</b>	<b>51</b>
第 13 条： <code>vector</code> 和 <code>string</code> 优先于动态分配的数组。 .....	51
第 14 条：使用 <code>reserve</code> 来避免不必要的重新分配。 .....	53
第 15 条：注意 <code>string</code> 实现的多样性。 .....	55
第 16 条：了解如何把 <code>vector</code> 和 <code>string</code> 数据传给旧的 API。 .....	59
第 17 条：使用“ <code>swap</code> 技巧”除去多余的容量。 .....	62
第 18 条：避免使用 <code>vector&lt;bool&gt;</code> 。 .....	64
<b>第 3 章 关联容器 .....</b>	<b>67</b>
第 19 条：理解相等( <code>equality</code> )和等价( <code>equivalence</code> )的区别。 .....	67
第 20 条：为包含指针的关联容器指定比较类型。 .....	71
第 21 条：总是让比较函数在等值情况下返回 <code>false</code> 。 .....	74
第 22 条：切勿直接修改 <code>set</code> 或 <code>multiset</code> 中的键。 .....	77
第 23 条：考虑用排序的 <code>vector</code> 替代关联容器。 .....	82
第 24 条：当效率至关重要时，请在 <code>map::operator[]</code> 与 <code>map::insert</code> 之间谨慎做出选择。 .....	87

第 25 条：熟悉非标准的哈希容器。 .....	91
<b>第 4 章 迭代器 .....</b>	<b>95</b>
第 26 条：iterator 优先于 const_iterator、reverse_iterator 以及 const_reverse_iterator。 .....	95
第 27 条：使用 distance 和 advance 将容器的 const_iterator 转换成 iterator。 .....	98
第 28 条：正确理解由 reverse_iterator 的 base() 成员函数所产生的 iterator 的用法。 .....	101
第 29 条：对于逐个字符的输入请考虑使用 istreambuf_iterator。 .....	103
<b>第 5 章 算法 .....</b>	<b>106</b>
第 30 条：确保目标区间足够大。 .....	106
第 31 条：了解各种与排序有关的选择。 .....	110
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。 ..	115
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。 .....	118
第 34 条：了解哪些算法要求使用排序的区间作为参数。 .....	121
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。 .....	124
第 36 条：理解 copy_if 算法的正确实现。 .....	128
第 37 条：使用 accumulate 或者 for_each 进行区间统计。 .....	130
<b>第 6 章 函数子、函数子类、函数及其他 .....</b>	<b>135</b>
第 38 条：遵循按值传递的原则来设计函数子类。 .....	135
第 39 条：确保判别式是“纯函数”。 .....	139
第 40 条：若一个类是函数子，则应使它可配接。 .....	142
第 41 条：理解 ptr_fun、mem_fun 和 mem_fun_ref 的来由。 .....	145
第 42 条：确保 less<T>与 operator< 具有相同的语义。 .....	149
<b>第 7 章 在程序中使用 STL .....</b>	<b>153</b>
第 43 条：算法调用优先于手写的循环。 .....	153
第 44 条：容器的成员函数优先于同名的算法。 .....	160
第 45 条：正确区分 count、find、binary_search、lower_bound、upper_bound 和 equal_range。 .....	162
第 46 条：考虑使用函数对象而不是函数作为 STL 算法的参数。 .....	170
第 47 条：避免产生“直写型”(write-only)的代码。 .....	174
第 48 条：总是包含(#include)正确的头文件。 .....	177
第 49 条：学会分析与 STL 相关的编译器诊断信息。 .....	178
第 50 条：熟悉与 STL 相关的 Web 站点。 .....	185
<b>参考书目 .....</b>	<b>191</b>
<b>附录 A：地域性与忽略大小写的字符串比较 .....</b>	<b>195</b>
<b>附录 B：对 Microsoft 的 STL 平台的说明 .....</b>	<b>204</b>

# 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 用法错误，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 粗粗浏览一遍，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，尤其是哈希容器、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哈希容器的一般性介绍。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资料的引用方式。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]，提到的时候我往往会简单地称做“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

如果不提到引用计数技术而来讨论 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++ 书籍都涉及到了这一话题。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，哈希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”，即在一步内向前或向后跳跃的能力。`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對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做键(key)，而把第二部分称做值(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。

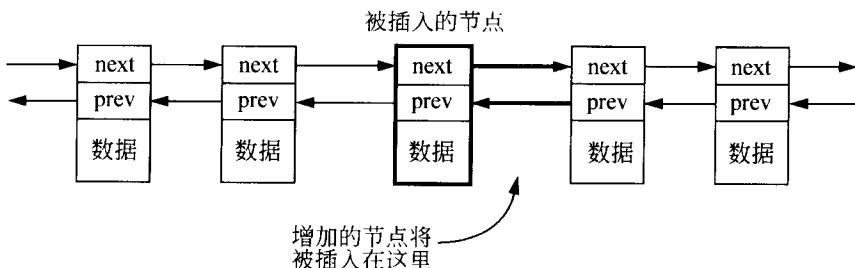
但是在另一种情况下，它就不是一个风格问题了。为了避免潜在的语法解析二义性(细节我将省去)，你需要在从属于形式类型参数的类型名前面使用 **typename**。这样的类型被称为从属类型(dependent type)，用一个例子可以阐明这一点。假设你要写一个函数模板，给它一个 STL 容器，它将返回容器中的最后一个元素是否大于第一个元素。下面是一种实现方式：

```
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词 **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<sup>1</sup>：



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 12 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。

<sup>1</sup> 出版者注：原著使用了不同颜色（红色）来突出要强调的内容，中文版在出版时考虑相关因素，将要强调的内容以粗体形式表示。

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，请在 `map::operator[]` 与 `map::insert` 之间谨慎做出选择。
- 第 25 条：熟悉非标准的哈希容器。
- 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这么做。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 容器不是简单地好，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，拷贝操作的重要性；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。

涉及到的方方面面很多。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。
- **非标准的关联容器** `hash_set`、`hash_multiset`、`hash_map` 和 `hash_multimap`。在第 25 条中，我分析了这些基于哈希表的、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）。非标准的哈希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如何排序的？如果不关心，则哈希容器是一个可行的选择方案；否则，你要避免哈希容器。
- 你选择的容器必须是标准 C++ 的一部分吗？如果必须是，就排除了哈希容器、`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）以及哈希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，就要考虑哈希容器（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条所示,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拷贝做任何假定。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得的。

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;
...
vector<Widget>::iterator i =           //为 bestWidget 赋一个值
    find(vw.begin(), vw.end(), bestWidget); //找到一个与 bestWidget 具
                                                //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;
...
WCIterator i = 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。比如你有如下类型的对象

```
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 CustomerContainer::iterator CCIIterator;

    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: 确保容器中的对象拷贝正确而高效。

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(通过如 `insert` 或 `push_back` 之类操作)向容器中加入对象时, 存入容器的是你所指定的对象的拷贝。当(通过如 `front` 或 `back` 之类操作)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拷贝。进去的是拷贝, 出来的也是拷贝(`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, 它经常会进一步被拷贝。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(拷贝)(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(拷贝)。没错, 拷贝对象是 STL 的工作方式。

可能你想知道这种拷贝动作是怎样进行的。这很简单。利用一个对象的拷贝成员函数就可以很方便地拷贝该对象, 特别是对象的拷贝构造函数(`copy constructor`)和拷贝赋值操作符(`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&);           //拷贝构造函数
    Widget& operator=(const Widget&); //拷贝赋值操作符
    ...
};
```

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拷贝。(有关拷贝构造函数和拷贝赋值操作符的细节, 可参考任何一本 C++ 的入门书。在 *Effective C++* 中, 第 11 条和第 27 条是

专门针对这两个函数的行为的。)

考虑到这些拷贝过程，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，而对象的拷贝操作又很费时，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，拷贝所需要的内存和时间就越多。而且，如果这些对象的“拷贝”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，拷贝动作会导致剥离(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(通过基类的拷贝构造函数)被拷贝进容器时，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;                                    //sw 作为基类对象被拷贝进 vw 中
vw.push_back(sw);                                  //它的派生类特有部分在拷贝时被丢掉了
```

“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背景知识，请参阅 *Effective C++* 的第 22 条。关于 STL 中剥离问题的另一个例子，请参见第 38 条。)

使拷贝动作高效、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。拷贝指针的速度非常快，并且总是会按你期望的方式进行(它拷贝构成指针的每一位)，而且当它被拷贝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 好像是在疯狂拷贝，那就让我们再想一想。没错，STL 做了很多拷贝，但它总的设计思想是为了避免不必要的拷贝。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 maxNumWidgets 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(通过拷贝)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 容器是在创建拷贝; 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(
    list1.end(), list2,           //把 list2 中从第一个含 5 的节点
    find(list2.begin(), list2.end(), 5), //到最后一个含 10 的所有节点
    find(list2.rbegin(), list2 rend(), 10).base() //移动到 list1 的末尾。
); //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拷贝任何数据的能力。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惟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。

设计这个小测验是为了两个目的。首先，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拷贝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 `operator=` 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>::const_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拷贝,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。数据被拷贝这一事实没有任何意义,因为 STL 就是建立在拷贝数据这一基础上的。对于 STL 来说,拷贝是基础,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,更频繁地拷贝对象,而且/或者做冗余的操作。

比如,假定你要把一个`int`数组拷贝到一个`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拷贝到 `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+1`, 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拷贝构造函数。(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, 将会调用该元素的拷贝构造函数。)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` 的拷贝构造函数。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他们最终的位置上，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拷贝构造函数，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。非标准哈希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（例如 `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。）

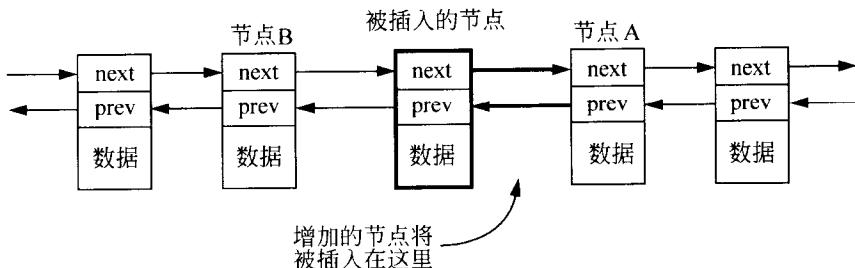
不明智地使用重复的单元素插入操作而不是一次区间插入操作，这样所带来的最后一个性能问题跟内存分配有关，尽管它同时还伴有讨厌的拷贝问题。在第 14 条将会指出，如果试图把一个元素插入到 `vector` 中，而它的内存已满，那么 `vector` 将分配具有更大容量（capacity）的新内存，把它的元素从旧内存拷贝到新内存中，销毁旧内存中的元素，并释放旧内存。然后它把要插入的元素加入进来。第 14 条还解释了多数 `vector` 实现每次在内存耗尽时，会把容量加倍，因此，插入 `numValues` 个新元素最多可导致  $\log_2 numValues$  次新的内存分配。第 14 条指出，表现出这种行为的 `vector` 实现是存在的，因此，把 1 000 个元素逐个插入可能会导致 10 次新的内存分配（包括低效的元素拷贝）。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，拷贝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。

啰啰嗦嗦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` 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，你想把这些整数拷贝到一个 `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拷贝到 `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);
```

下面这行做了同样的事情。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。惟一的区别是, `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>()); //第一个参数两边的括号
```

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;  
    ...  
    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(ada-

ptable) 的能力(见第 40 条)。考虑到 DeleteObject 的设计初衷(用于 for\_each)，很难想像这是一个问题。

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++* 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++* 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：

```
void doSomething()
{
    typedef boost::shared_ptr<Widget> SPW; //SPW=“指向 Widget 的
                                                //      shared_ptr”
    vector<SPW> vwp;
    for (int i = 0; i < SOME_MAGIC_NUMBER; ++i)
        vwp.push_back(SPW(new Widget));       //从 Widget* 创建 SPW，然后
                                                //对它进行一次 push_back
                                                //使用 vwp
    }                                       //这里不会有 Widget 泄漏，即使在上面的代码中有异常被抛出
```

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(*Effective STL*)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。<sup>1</sup>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，那我就告诉你拷贝 `auto_ptr` 意味着什么，这会很特别——有些人会说很古怪。

当你拷贝一个 `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：拷贝一个 `auto_ptr` 意味着改变它的值：

---

<sup>1</sup> 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]。

```

auto_ptr<Widget> pw1(new Widget); //pw1 指向一个 Widget
auto_ptr<Widget> pw2(pw1);           //pw2 指向 pw1 的 Widget; pw1 被置为 NULL。
                                     // (Widget 的所有权从 pw1 转移到 pw2 上。)
pw1 = pw2;                         //现在 pw1 又指向 Widget 了; pw2 被置为 NULL。

```

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：

```

bool widgetAPCompare(const auto_ptr<Widget>& lhs,
                      const auto_ptr<Widget>& rhs)
{
    return *lhs < *rhs;                //对这个例子，假定对 Widget
                                      //存在 operator<操作符。
}
vector<auto_ptr<Widget>> widgets;   //创建一个 vector 并用指向 Widget 的
                                      //auto_ptr 填满它。记住，这不应该通过编译！
...
sort(widgets.begin(), widgets.end(), //对 vector 排序
     widgetAPCompare);

```

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：

```

template<class RandomAccessIterator,      //对 sort 的这一声明是直接从标准
         class Compare>            //中抄过来的
void sort(RandomAccessIterator first,
          RandomAccessIterator last,
          Compare comp)
{
    //这一 typedef 将在后面解释
    typedef typename iterator_traits<RandomAccessIterator>::value_type
        ElementType;
    RandomAccessIterator i;
    ...
                                      //使 i 指向基准元素

```

```

ElementType pivotValue(*i);           // 把基准元素拷贝到局部临时变量中;
...                                     // 见随后的讨论
                                         // 做其余的排序工作
}

```

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。惟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：

```
ElementType pivotValue(*i);
```

因为它把一个元素从被排序的区间中拷贝到一个临时对象中。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拷贝的 `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拷贝了一个基准元素。

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：

```
Container<int> c;
```

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):

```
c.erase(remove(c.begin(), c.end(), 1963), //当 c 是 vector、string 或 deque
      c.end()); //时, erase-remove 习惯用法是删除
               //特定值的元素的最好办法
```

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:

```
c.remove(1963); //当 c 是 list 时, remove 成员函数
                  //是删除特定值的元素的最好办法。
```

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`:

```
c.erase(1963); //当 c 是标准关联容器时, erase 成员
                  //函数是删除特定值元素的最好办法。
```

这样做不仅是正确的, 而且是高效的, 只需要对数时间的开销。(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:

```
bool badValue(int ); //返回 x 是否为“坏值”
```

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:

```
c.erase(remove_if(c.begin(), c.end(),
                  badValue), c.end()); //当 c 是 vector、string 或 deque
                           //时, 这是删除使 badValue 返回 true
                           //的对象的最好办法。
c.remove_if(badValue); //当 c 是 list 时, 这是删除使
                        //badValue 返回 true 的对象
                        //的最好办法。
```

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` 把我们需要的值拷贝到一个新容器中, 然后把原来容器的内容和新容器的内容相互交换:

```

AssocContainer<int> c;                                //c 现在是一个标准关联容器
...
AssocContainer<int> goodValues;                      //保存不被删除的值的临时容器
remove_copy_if(c.begin(), c.end(),           //把不被删除的值从 c 拷贝到
                inserter(goodValues,          //goodValues 中
                           goodValues.end()),
                           badValue);
c.swap(goodValues);                            //交换 c 和 goodValues 的内容

```

这种办法的缺点是需要拷贝所有不被删除的元素，而我们可能并不希望付出这么多的拷贝代价。

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：

```

AssocContainer<int> c;
...
for (AssocContainer<int>::iterator i = c.begin(); //清晰、直接，但有 bug 的
     i != c.end();                               //代码，用来删除使
     ++i){                                     //badValue 返回 true 的
    if (badValue(*i)) c.erase(i);               //元素。别这么做！
}

```

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：

```

AssocContainer<int> c;
...
for (AssocContainer<int>::iterator i = c.begin(); //for 循环的第三部分是
     i != c.end();                               //空的，i 在下面递增。
     /*什么也不做*/){
    if (badValue(*i)) c.erase(i++);             //对坏值，把当前的 i 传给
    else ++i;                                  //erase，递增 i 是副作用；
}                                                 //对好值，则简单地递增 i

```

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：

```
ofstream logFile; //要写入的日志文件
AssocContainer<int> c;
...
for (AssocContainer<int>::iterator i = c.begin(); //循环条件同上
     i != c.end();){
    if (badValue(*i)){
        logFile << "Erasing " << *i << '\n'; //写日志文件
        c.erase(i++); //删除元素
    }
    else ++i;
}
```

现在给我们带来麻烦的是 *vector*、*string* 和 *deque*。我们不能再使用 *erase-remove* 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：

```
for (SeqContainer<int>::iterator i = c.begin();
     i != c.end();){
    if (badValue(*i)){
        logFile << "Erasing " << *i << '\n';
        i = c.erase(i); //把 erase 的返回值赋给 i,
    } //使 i 的值保持有效
    else ++i;
}
```

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`<sup>1</sup>。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

---

<sup>1</sup> 这一点仅对以迭代器作为参数的 `erase` 有效。关联容器还提供了一种以值为参数的 `erase` 形式, 该形式返回被删除的元素的个数。不过这里, 我们只关心通过迭代器的 `erase`。

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,嗯,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,单单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:

```
template<typename T>
class SpecialAllocator{...}; // 用户定义的分配子模板
```

```

typedef SpecialAllocator<Widget> SAW;           //SAW=“针对 Widget 的
//          SpecialAllocator”
list<Widget, SAW> L1;
list<Widget, SAW> L2;
...
L1.splice(L1.begin(), L2);                      //把 L2 的节点移动到 L1 前面

```

请回忆一下，当 `list` 的元素从一个 `list` 链接到另一个时，并没拷贝任何东西。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:

```
void* operator new(size_t bytes);
pointer allocator<T>::allocate(size_type numObjects);
//请记住 “pointer” 是个类型定义, 它实际上总是 T*
```

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 的值则应是 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:

```
list<int> L;                                //等同于 list<int, allocator<int>>;
                                                //从未要求 allocator<int>分配内存!
set<Widget, SAW> s;                          //记住 SAW 是对 SpecialAllocator<Widget>的
                                                //类型定义; 并没有通过 SAW 分配内存!
```

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：

```
template<typename T,
         typename Allocator = allocator<T>> //list 的可能实现
class list{
private:
    Allocator alloc; //类型为 T 的对象的分配子
    struct ListNode{ //链表中的节点
        T data;
        ListNode *prev;
        ListNode *next;
    };
    ...
};
```

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：

```
template <typename T> //标准的分配子是这样声明的,
class allocator{ //但这同样也可以是一个用户所写的分配子模板
public:
    template<typename U>
    struct rebind{
        typedef allocator<U> other;
    };
    ...
};
```

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：

```
Allocator::rebind<ListNode>::other
```

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` 带有惟一的类型参数 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:

```
void* mallocShared(size_t bytesNeeded);
void* freeShared(void* ptr);
```

而你想把 STL 容器的内容放到这块共享内存中去。没问题:

```
template<typename T>
class SharedMemoryAllocator{
public:
    ...
    pointer allocate<size_type numObjects, const void *localityHint = 0>
    {
        return static_cast<pointer>(mallocShared(numObjects * sizeof(T)));
    }
    void deallocate(pointer ptrToMemory, size_type numObjects)
    {
        freeShared(ptrToMemory);
    }
    ...
};
```

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`:

```
//方便的 typedef
typedef
    vector<double, SharedMemoryAllocator<double>> SharedDoubleVec;
...
{
    //开始某个代码块
    ...
    SharedDoubleVec v;           //创建一个 vector, 其元素位于共享内存中
    ...
```

```
}
```

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：

```
void *pVectorMemory =                               //为 SharedDoubleVec 对象分配
    mallocShared(sizeof(SharedDoubleVec));          //足够的内存
SharedDoubleVec *pv =                            //使用“placement new”在内存
    new (pVectorMemory) SharedDoubleVec;           //中创建一个 SharedDoubleVec
                                                //对象：见下文
...
pv->~SharedDoubleVec();                         //使用对象(通过 pv)
freeShared(pVectorMemory);                      //析构共享内存中的对象
                                                //释放最初分配的那一块共享内存
```

但愿上面的注释能解释清楚这是如何工作的。大致是这样：首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：

```
class Heap1{
public:
    ...
    static void *alloc(size_t numBytes, const void *memoryBlockToBeNear);
    static void dealloc(void *ptr);
    ...
};

class Heap2{...};                                //有同样的 alloc/dealloc 接口
```

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：

```

template<typename T, typename Heap>
class SpecificHeapAllocator{
public:
    ...
    pointer allocate(size_type numObjects, const void *localityHint = 0)
    {
        return static_cast<pointer>(Heap::alloc(numObjects * sizeof(T),
                                                localityHint));
    }
    void deallocate(pointer ptrToMemory, size_type numObjects)
    {
        Heap::dealloc(ptrToMemory);
    }
    ...
};

};


```

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

```

vector<int, SpecificHeapAllocator<int, Heap1> > v; //v 和 s 的元素都放
set<int, SpecificHeapAllocator<int, Heap1> > s;      //在 Heap1 中
list<Widget,
     SpecificHeapAllocator<Widget, Heap2> > L;          //L 和 m 的元素都放
map<int , string, less<int>,                           //在 Heap2 中
     SpecificHeapAllocator<pair<const int, string>,
     Heap2> > m;

```

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。理解为什么會这样是很重要的。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。

```
vector<int> v;
...
vector<int>::iterator first5(find(v.begin(), v.end(), 5));      //第 1 行
if (first5 != v.end()){                                         //第 2 行
    *first5 = 0;                                                 //第 3 行
}
```

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，很难想像一个 STL 实现能自动推断出这一点。考虑到同步原语(例如信号量、互斥体等)通常会有较高的开销，这就更难想像一个 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：

```
template<typename Container>           //一个为容器获取和释放互斥体的模板
class Lock{                           //框架；其中的很多细节被省略了
public:
    Lock(const Container& container)
    :c(container)
    {
        getMutexFor(c);           //在构造函数中获取互斥体
    }
```

```

~Lock()
{
    releaseMutexFor(c); //在析构函数中释放它
}
private:
    const Container& c;
};

```

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：

```

vector<int> v;
...
{
    Lock<vector<int> > lock(v); //创建新的代码块
    vector<int>::iterator first5(find(v.begin(), v.end(), 5));
    if (first5 != v.end()) {
        *first5 = 0;
    }
} //代码块结束，自动释放互斥体

```

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<sup>1</sup>。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

<sup>1</sup> 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。

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。当涉及到 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。

# 第 2 章 vector 和 string

所有的 STL 容器都是有用的，但大多数 C++ 程序员会发现使用 `vector` 和 `string` 的时候更多一些。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：

1. 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。
2. 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。
3. 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拷贝，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拷贝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重指出。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拷贝构造函数。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）。第三，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 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，它们的容量即加倍。
2. 把容器的所有元素从旧的内存拷贝到新的内存中。
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、拷贝和析构步骤，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，对象的拷贝和析构，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 1 000 之间的值的 `vector<int>`。如果不使用 `reserve`，你可能会这样做：

```
vector<int> v;
for (int i = 1; i <= 1000; ++i) v.push_back(i);
```

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 1 000 大致等于  $2^{10}$ 。）

把这段代码改为使用 `reserve`，如下所示：

```
vector<int> v;
v.reserve(1000);
for (int i = 1; i <= 1000; ++i) v.push_back(i);
```

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：

```
string s;
...
if (s.size() < s.capacity()) {
    s.push_back('x');
}
```

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，“把猫放入栈中的十六种方法”(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：

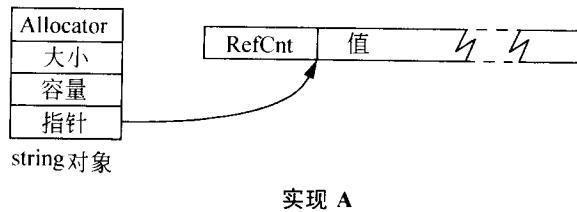
- 它的分配子的一份拷贝。这个字段是可选的，原因可参阅第 10 条，那里介绍了关于分配子的怪异规则。

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。

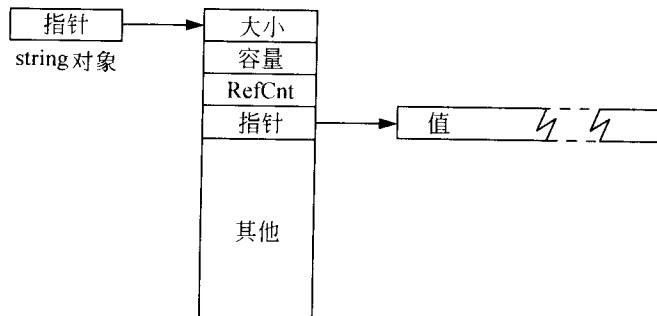
在实现 A 中，每个 `string` 对象包含其分配子的一份拷贝、该字符串的大小、它的容量，以及一个指针，该指针指向一块动态分配的内存，其中包含了引用计数(即 `RefCnt`)和字符串的值。在该实现中，使用默认分配子的 `string` 对象其大小是一个指针的 4 倍。若使用了自定义的分配子，则 `string` 对象会更大一些，多出的部分取决于分配子对象的大小。



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。

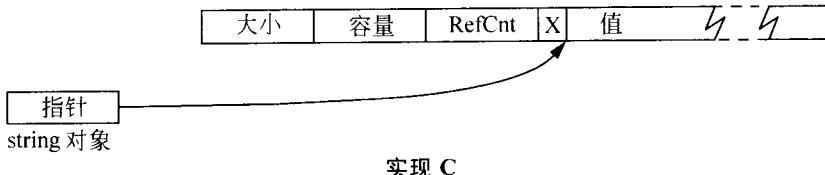
B 的 `string` 所指向的对象中包含了该字符串的大小、容量和引用计数，以及一个指向一块动态分配的内存的指针，该内存中存放了字符串的值。该对象还包含了一些与多线程环境下的同步控制相关的额外数据。这些数据不在我们的讨论范围之内，所以我把这部分数据标记为“其他”。



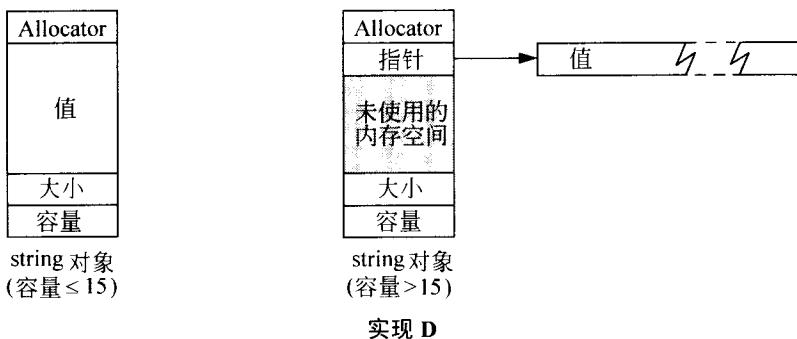
实现 B

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 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。



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`：

```
string s("Perse"); //我们的狗叫 Persephone，但我们通常叫它 Perse。请访问  
//它的网站：http://www.aristeia.com/Persephone/
```

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：

```
string s("Perse"); //s 是大小为 5 的字符串
```

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。总结这些区别如下：

- `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拷贝时，引用计数才有用，而有些应用并不经常拷贝内存，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（如参阅第 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

```
vector<int> v;
```

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：

```
void doSomething(const int* pInts, size_t numInts);
```

则我们可以这样做：

```
doSomething(&v[0], v.size());
```

这样或许能行。可能不会出错。惟一麻烦的地方在于，`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：

```
if (!v.empty()) {
    doSomething(&v[0], v.size());
}
```

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：

```
void doSomething(const char* pString);
```

如下所示：

```
doSomething(s.c_str());
```

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：

```
void doSomething(const int* pInts, size_t numInts);
void doSomething(const char* pString);
```

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拷贝，该拷贝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。我甚至不愿想像会发生什么情况。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:

```
//C API: 该函数以一个指向最多有 arraySize 个 double 类型数据的数组的指针为参数，并  
//向该数组中写入数据。它返回已被写入的 double 数据的个数，这个数不会超过 arraySize。  
size_t fillArray(double* pArray, size_t arraySize);  
vector<double> vd(maxNumDoubles);           //创建大小为 maxNumDoubles  
                                         //的 vector  
vd.resize(fillArray(&vd[0], vd.size()));    //使用 fillArray 向 vd 中写入数据，  
                                         //然后把 vd 的大小改为 fillArray  
                                         //所写入的元素的个数
```

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拷贝到相应字符串中：

```
//C API: 该函数以一个指向最多有 arraySize 个 char 类型数据的数组的指针为参数，并  
//向该数组中写入数据。它返回已被写入的 char 数据的个数，这个数不会超过 arraySize。  
size_t fillString(char* pArray, size_t arraySize);  
vector<char> vc(maxNumChars); //创建大小为 maxNumChars  
//的 vector
```

```

size_t charsWritten = fillString(&vc[0], vc.size(0));
                           //使用 fillString 向 vc 中写入数据
string s(vc.begin(), vc.begin() + charsWritten);
                           //通过区间构造函数，把数据从 vc
                           //拷贝到 s 中(见第 5 条)

```

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

```

size_t fillArray(double *pArray, size_t arraySize);
vector<double> vd(maxNumDoubles);           //同上
vd.resize(fillArray(&vd[0], vd.size()));

deque<double> d(vd.begin(), vd.end());      //把数据拷贝到 deque 中
list<double> l(vd.begin(), vd.end());        //把数据拷贝到 list 中
set<double> s(vd.begin(), vd.end());         //把数据拷贝到 set 中

```

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拷贝到一个 `vector` 中，然后传给该 API：

```

void doSomething(const int* pInts, size_t numInts);    //C API(见上)
set<int> intSet;                                     //存储要传给 API 的数据的 set
...
vector<int> v(intSet.begin(), intSet.end());   //把 set 的数据拷贝到 vector
if (!v.empty()) doSomething(&v[0], v.size()); //把数据传给 API

```

你也可以把数据拷贝到一个数组中，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：

```

class Contestant{...};
vector<Contestant> contestants;

```

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）100 000，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：

```
vector<Contestant>(contestants).swap(contestants);
```

表达式 `vector<Contestant>(contestants)` 创建一个临时的矢量，它是 `contestants` 的拷贝：这是由 `vector` 的拷贝构造函数来完成的。然而，`vector` 的拷贝构造函数只为所拷贝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` 肿胀的容量。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：

```
string s;
...
string(s).swap(s);
```

//让 s 变大，然后删除它的大部分字符  
//对 s 做 shrink-to-fit

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：

```
vector<Contestant> v;
```

```

string s;
...
vector<Contestant>().swap(v);           //使用 v 和 s
                                         //清除 v 并把它的容量变为最小
string().swap(s);                      //清除 s 并把它的容量变为最小

```

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：

```

T *p = &c[0];                                //用 operator[] 返回的变量的地址
                                              //初始化一个 T* 变量

```

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：

```

vector<bool> v;
bool *pb = &v[0];                            //用 vector<bool>::operator[] 返回的
                                              //变量的地址初始化一个 bool* 变量

```

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 Patterns* [6] 中的“proxy”那一章。）抛开细节，`vector<bool>` 看起来像是这样：

```
template<typename Allocator>
vector<bool, Allocator> {
public:
    class reference {...};           // 用来为指向单个位的引用而产生代理的类
    reference operator[](size_type n); // operator[] 返回一个代理
    ...
};
```

现在，下面的代码为何不能通过编译应该是很明了了：

```
vector<bool> v;
bool *pb = &v[0];           // 错了！表达式的右边是 vector<bool>:::
                           // reference* 类型，而不是 bool* 类型
```

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<sup>1</sup>(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。

---

<sup>1</sup> 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。不错，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。不错，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 中没有基于哈希表的容器，并举出两个常见的（非标准的）实现。尽管 STL 本身没有提供哈希表，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：

```

class Widget {
public:
    ...
private:
    TimeStamp lastAccessed;
    ...
};

```

而我们可能有一个针对 `Widget` 的 `operator==`, 它忽略了这个域:

```

bool operator==(const Widget& lhs, const Widget& rhs)
{
    // 忽略了 lastAccessed 域的代码
}

```

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:

```

!(w1 < w2);           // w1 < w2 不为真
&&                      // 而且
!(w2 < w1);           // w2 < w1 不为真

```

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:

```

!c.key_comp()(x, y) && !c.key_comp()(y, x)      // 在 c 的排列顺序中, x 在 y 之前
// 不为真, y 在 x 之前也不为真

```

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`:

```
struct CIStringCompare:                                //不区分大小写的字符串比较类
public
binary_function<string, string, bool> { //该基类的信息请参阅第 40 条
bool operator()(const string& lhs,
                 const string& rhs) const
{
    return ciStringCompare(lhs, rhs);      //关于 ciStringCompare 是如何
}                                              //实现的, 请参阅第 35 条
};
```

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

```
set<string, CIStringCompare> ciSS;                  //ciSS="不区分大小写的字符串集"
```

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:

```
ciSS.insert("Persephone");                         //一个新元素被插入到集合中
ciSS.insert("persephone");                          //没有新元素被插入到集合中
```

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:

```
if (ciSS.find("persephone") != ciSS.end())...        //该检查将成功
```

但如果我们使用非成员的 `find` 算法, 则查找将失败:

```
if (find(ciSS.begin(), ciSS.end(),
         "persephone") != ciSS.end())...                //该检查将失败
```

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`：

```
set2CF<string, CISTringCompare, equal_to<string> > s;
```

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，泊尔塞福涅<sup>1</sup>）的两种拼写插入到 `s` 中：

```
s.insert("Persephone");
s.insert("persephone");
```

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？

```
if (s.find("persephone") != s.end()) ... //这个检查会成功还是失败？
```

`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，

<sup>1</sup> 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。

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）基于哈希表的关联容器，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：

```
set<string*> ssp //ssp="set of string ptrs"  
ssp.insert(new string("Anteater"));  
ssp.insert(new string("Wombat"));  
ssp.insert(new string("Lemur"));  
ssp.insert(new string("Penguin"));
```

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有序。

```
for (set<string*>::const_iterator i = ssp.begin(); //你期望看到:  
      i != ssp.end(); // "Anteater",  
      ++i) // "Lemur",  
         cout << *i << endl; // "Penguin",  
                           // "Wombat"
```

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：

```
copy(ssp.begin(), ssp.end(), //把 ssp 中的字符串拷贝到 cout  
      ostream_iterator<string>(cout, "\n")); // (但这不能通过编译)
```

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。<sup>1</sup>

为了解决这个问题，请记住

```
set<string*> ssp;
```

是下面代码的缩写

```
set<string*, less<string*> > ssp;
```

最精确地说，它是下面代码的缩写

```
set<string*, less<string*>, allocator<string*> > ssp;
```

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：

```
struct StringPtrLess:
    public binary_function<const string*,           //采用该基类的原因见第 40 条
                           const string*,
                           bool> {
    bool operator()(const string *ps1, const string *ps2) const
    {
        return *ps1 < *ps2;
    }
};
```

然后可以用 `StringPtrLess` 作为 `ssp` 的比较类型：

```
typedef set<string*, StringPtrLess> StringPtrSet;
StringPtrSet ssp;                                //创建一个包含字符串的集合，并把它
                                                //用 StringPtrLess 定义的规则排序
...                                                 //同先前一样插入 4 个字符串
```

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）：

<sup>1</sup> 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。

```

for (StringPtrSet::const_iterator i = ssp.begin();
     i != ssp.end();           //打印出的是"Anteater", "Lemur",
     ++i)                     // "Penguin"和"Wombat"
cout << **i << endl;

```

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：

```

void print(const string *ps)           //把 ps 指向的对象打印到 cout
{
    cout << *ps << endl;
}
for_each(ssp.begin(), ssp.end(), print); //对 ssp 中的每个对象调用 print

```

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：

```

//当向该类型的函数子传入 T* 时，它们返回 const T&
struct Dereference {
    template<typename T>
    const T& operator()(const T *ptr) const
    {
        return *ptr;
    }
};
transform(ssp.begin(), ssp.end(),           //通过解除指针引用，“转
         ostream_iterator<string>(cout, "\n"), //换”ssp 中的每个元素，
         Dereference());                   //并把结果写到 cout

```

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：

```

bool stringPtrLess(const string* ps1,      //将作为一个比较函数，用于按照
                   const string* ps2)      //字符串的值对 string* 指针进行排序
{
    return *ps1 < *ps2;
}

```

```
set<string, stringPtrLess> ssp;           //试图使用 stringPtrLess 作为 ssp 的
                                            //比较函数; 这不能通过编译
```

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:

```
struct DereferenceLess {
    template<typename PtrType>
    bool operator()(PtrType pT1,
                     PtrType pT2) const           //参数是按值传入的, 因为我们期望
                                            //它们是(或表现得像)指针
    {
        return *pT1 < *pT2
    }
}
```

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:

```
set<string*, DereferenceLess> ssp;      //与 set<string*, StringPtrLess>
                                            //的行为相同
```

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:

```
set<int, less_equal<int> > s; //s 用"<="来排序
s.insert(10);                  //插入 10
```

现在让我们再插入 10:

```
s.insert(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:

```
!(10_A <= 10_B) && !(10_B <= 10_A) // 检查 10_A 和 10_B 的等价性
```

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

```
!(true) && (!true)
```

而这又简化为

```
false && false
```

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 的两份拷贝, 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:

```
struct StringPtrGreater:  
public binary_function<const string*, // 加粗的部分显示了这段代码是怎样
```

```

        const string*,           //在第 20 条 StringPtrLess 的基础
        bool> {
    bool operator()(const string *ps1, const string *ps2) const
    {
        return !(ps1 < ps2);           //把旧的测试简单取反！这是不对的！
    }
};

```

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：

```

struct StringPtrGreater:
    public binary_function<const string*,      //这是对关联容器的合法的比较类型
                           const string*,
                           bool> {
    bool operator()(const string *ps1, const string *ps2) const
    {
        return *ps2 < *ps1;                  //返回*ps2 是否在*ps1 之前
    }                                         //(即改变操作数的顺序)
};

```

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`。

唉。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`：

```

multiset<int, less_equal<int> > s;           //s 仍然用"≤"来排序
s.insert(10);                                //插入 10A
s.insert(10);                                //插入 10B

```

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<sub>A</sub> 和 10<sub>B</sub> 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拷贝返回 `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 1 000)，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：

```
map<int, string> m;  
...  
m.begin()->first = 10;                      // 错！map 的键不能修改  
multimap<int, string> mm;  
...  
mm.begin()->first = 20;                      // 错！multimap 的键同样不能修改
```

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，而无需任何强制类型转换。(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：

```
class Employee {
public:
    ...
    const string& name() const;           // 获取雇员的名字
    void setName(const string& name);    // 设置雇员的名字
    const string& title() const;          // 获取雇员的头衔
    void setTitle(const string& title);   // 设置雇员的头衔
    int idNumber() const;                // 获取雇员的 ID 号
    ...
};
```

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惟一的 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：

```
struct IDNumberLess:
    public binary_function<Employee, Employee, bool> {           // 见第 40 条
    bool operator()(const Employee& lhs,
                     const Employee& rhs) const
    {
        return lhs.idNumber() < rhs.idNumber();
    }
};

typedef set<Employee, IDNumberLess> EmpIDSet;
EmpIDSet se;                                         // se 是按照 ID 号进行排序的雇员集合
```

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：

```
Employee selectedID;                                // 存储有特定 ID 号的 Employee 变量
...
EmpIDSet::iterator i = se.find(selectedID);
if (i != se.end()) {
    i->setTitle("Corporate Deity");               // 给雇员赋以新的头衔
}
```

因为我们在这里所做的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<sup>1</sup>，不同的实现者以不同的方式做了诠释。结果是，很可能发现在有的 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：

```
EmpIDSet se;                                //如前，se 是按照 ID 号排序的雇员集合
Employee selectedID;                         //如前，selectedID 是存储有特定 ID 号
                                              //的 Employee 变量
...
EmpIDSet::iterator i = se.find(selectedID);
if (i != se.end()) {
    i->setTitle("Corporate Deity");        //有些 STL 实现将认为这一行不合法
}
```

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。

<sup>1</sup> 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，则必定有人会作出这种选择。

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：

```
EmpIDSet::iterator i = se.find(selectedID);
if (i != se.end()) {
    i->setTitle("Corporate Deity");           //有些 STL 实现将认为这一行不合法,
}                                              //因为*i 为 const
```

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：

```
if (i != se.end()) {
    const_cast<Employee*>(*i).setTitle("Corporate Deity");
}                                              //转换掉*i 的 const 性质
```

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：

```
if (i != se.end()) {
    static_cast<Employee*>(*i).setTitle("Corporate Deity");
}                                              //把*i 转换成 Employee
```

它与下面的方式是等价的：

```
if (i != se.end()) {
    ((Employee)(*i)).setTitle("Corporate Deity");
}                                              //同上。只是使用了 C 方式的类型转换
```

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

个拷贝，`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，它是在该对象的拷贝上被调用的。这两种语法形式都等价于：

```
if (i != se.end()) {  
    Employee tempCopy(*i); // 把 *i 拷贝到 tempCopy  
    tempCopy.setTitle("Corporate Deity"); // 修改 tempCopy  
}
```

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拷贝。在 `map` 或 `multimap` 的情况下，请记住，不要把该拷贝的第一个部分声明为 `const`。毕竟，你想要改变它。
3. 修改该拷贝，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：

```

EmpIDSet se;                                //如前, se 是按照 ID 号排序的雇员集合
Employee selectedID;                         //如前, selectedID 是存储有特定
                                              //ID 号的 Employee 变量

...
EmpIDSet::iterator i = se.find(selectedID);   //第 1 步: 找到待修改的元素
if (i != se.end()) {
    Employee e(*i);                          //第 2 步: 拷贝该元素
    e.setTitle("Corporate Deity");           //第 3 步: 修改拷贝
    se.erase(i++);                           //第 4 步: 删除该元素;
                                              //递增迭代器以保持它的有效性(见第 9 条)
    se.insert(i, e);                          //第 5 步: 插入新元素; 提示它的位置
                                              //和原来的相同
}

```

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, 考虑非标准的哈希容器几乎总是值得的(见第 25 条)。通过适当的哈希函数, 哈希容器几乎能提供常数时间的查找能力。(如果哈希函数选择得不合适, 或者表太小, 则哈希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, 哈希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首先做一些插入操作, 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三个阶段，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阶段。

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部性。

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4 096（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，和使用 `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部性，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），因为这时通常需要拷贝 `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：

```
vector<Widget> vw;                                //取代 set<Widget>
...
sort(vw.begin(), vw.end());                         //设置阶段：大量的插入，查找却很少
                                                       //设置阶段结束。（当模拟一个 multiset
                                                       //时，可能会选用 stable_sort;
                                                       //见第 31 条。）
Widget w;                                         //用于存放待查找值的对象
...
if (binary_search(vw.begin(), vw.end(), w)) ... //通过 binary_search 查找

vector<Widget>::iterator i =
    lower_bound(vw.begin(), vw.end(), w);           //通过 lower_bound 查找;
if (i != vw.end() && !(w < *i)) ...               //对"!(w < *i)"测试的解释，见第 19 条
```

```
pair<vector<Widget>::iterator,
      vector<Widget::iterator> range =
      equal_range(vw.begin(), vw.end(), w); //通过 equal_range 查找
if (range.first != range.second) ...
                                         //结束查找阶段，开始重组阶段
...
sort(vw.begin(), vw.end());           //开始新的查找阶段...
```

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：

```
typedef pair<string, int> Data;           //本例子中存储在"map"中的类型

class DataCompare {                      //比较函数的类
public:
    bool operator()(const Data& lhs,          //用于排序的比较函数
                     const Data& rhs) const
    {
        return keyLess(lhs.first, rhs.first);   //keyLess 是小于
    }
    bool operator()(const Data& lhs,          //用于查找的比较函数
                     const string& rhs) const
    {
        return lhs.first < rhs;
    }
}
```

```

        const Data::first_type& k) const
    {
        return keyLess(lhs.first, k);                                // (第 1 种形式)
    }
    bool operator()(const Data::first_type& k,                  // 用于查找的比较函数
                     const Data& rhs) const
    {
        return keyLess(k, rhs.first);                                // (第 2 种形式)
    }
private:
    bool keyLess(const Data::first_type & k1,                  // "实际"的比较函数
                 const Data::first_type & k2) const
    {
        return k1 < k2;
    }
};

```

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。惟一的区别是，需要用 `DataCompare` 对象作为比较函数：

```

vector<Data> vd;                                // 取代 map<string, int>
...                                                 // 设置阶段：大量的插入，查找却很少
sort(vd.begin(), vd.end());                      // 设置阶段结束。(当模拟 multiset 时，
                                                // 你可能应选用 stable_sort;
                                                // 见第 31 条。)
string s;                                         // 用于存放待查找值的对象
...
if (binary_search(vd.begin(),                // 开始查找阶段
                  vd.end(), s,
                  DataCompare())) ...                  // 通过 binary_search 查找
vector<Data>::iterator i =                    // 通过 lower_bound 查找，同样，对
lower_bound(vd.begin(), vd.end(),           // "!DataCompare()(s, *i)" 测试
            s, DataCompare());                // 的解释请见第 19 条

```

```

if (i != vd.end() && !DataCompare()(s, *i)) ...
pair<vector<Data>::iterator,
    vector<Data>::iterator> range =
equal_range(vd.begin(), vd.end(),           //通过 equal_range 查找
            s, DataCompare());
if (range.first != range.second) ...
...
//结束查找阶段，开始重组阶段

sort(vd.begin(), vd.end(), DataCompare()); //开始新的查找阶段...

```

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：

```

class Widget {
public:
    Widget();
    Widget(double weight);
    Widget& operator=(double weight);
    ...
};

```

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：

```

map<int, Widget> m;
m[1] = 1.50;
m[2] = 3.67;
m[3] = 10.5;
m[4] = 45.8;
m[5] = 0.0003;

```

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

```
map<K, V> m;
```

表达式

```
m[k] = v;
```

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：

```
map<int, Widget> m;
m[1] = 1.50;
```

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

```
m[1] = 1.50;
```

在功能上等同于：

```
typedef map<int, Widget> IntWidgetMap; //typedef 是为了
                                         //方便而使用的
pair<IntWidgetMap::iterator, bool> result =
    m.insert(IntWidgetMap::value_type(1, Widget())); //用键值 1 和默认构造
                                                       //的值对象创建一个新
                                                       //的 map 条目；对
                                                       //value_type 的解释
                                                       //见下文
result.first->second = 1.50; //为新构造的值对象
                             //赋值
```

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：

```
m.insert(IntWidgetMap::value_type(1, 1.50));
```

这里的最终效果和前面的代码相同，只是它通常会节省三个函数调用：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。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：

```
m[k] = v; //使用 operator[] 把  
           //k 的值更新为 v  
  
m.insert(IntWidgetMap::value_type(k, v)).first->second = v;  
           //使用 insert 把 k 的值  
           //更新为 v
```

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：

```
iterator affectedPair =  
    efficientAddOrUpdate(m, k, v);  
    //如果 k 不在 map m 中，则高效地把  
    //pair(k, v) 添加到 m 中；不然就高效地
```

```
// 把与 k 相关联的值更新为 v。返回一个迭代器，指向刚刚添加的或更新的 pair
```

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。

```
template<typename MapType,> //map 的类型
         typename KeyArgType,> //KeyArgType 和 ValueArgType
         typename ValueArgType> //为什么是类型参数，请参见下文
typename MapType::iterator
efficientAddOrUpdate(MapType& m,
                      const KeyArgType& k,
                      const ValueArgType& v)
{
    typename MapType::iterator lb = //确定 k 在什么位置或应在什么位置;
        m.lower_bound(k); //此处为何需要"typename"，请参见
                           //本书"引言"的"代码例子"部分
    if (lb != m.end() && //如果 lb 指向的 pair 的键与 k 等价，
        !(m.key_comp()(k, lb->first))) {
        lb->second = v; //则更新 pair 的值并返回指向该
        return lb; //pair 的迭代器
    }
    else {
        typedef typename MapType::value_type MVT;
        return m.insert(lb, MVT(k, v)); //把 pair<k, v>添加到 m 中，并
    } //返回一个指向该新元素的迭代器
}
```

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:

```
map<int, Widget> m; //同前
```

前面说过, `Widget` 接受来自 `double` 的赋值:

```
class Widget { //同前
public:
...
Widget& operator=(double weight);
...
};
```

现在考虑对 `efficientAddOrUpdate` 的调用:

```
efficientAddOrUpdate(m, 10, 1.5);
```

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: 熟悉非标准的哈希容器。

STL 程序员用了 STL 一段时间后就会想, “`vector`、`list`、`map`, 很好, 但是怎么没有哈希容器呢? ”。是的, 标准 C++ 库中没有任何哈希容器。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哈希容器, 但到现在为止, STL 中没有哈希容器。

然而, 如果你喜欢哈希表, 你也不必灰心。你不必将就着过没有哈希表的日子, 也不

必自己写一个哈希表。与 STL 兼容的哈希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，编写出使用哈希表的具有相当移植性的代码仍然是可能的，当然，如果哈希容器是标准化的，则事情要容易得多。（现在你明白了为什么标准化很重要。）

在已有的几种哈希容器的实现中，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哈希容器实现。STLport（同样，见第 50 条）也提供了哈希容器，但是 STLport 的哈希容器是以 SGI 的哈希容器为基础的。所以，在本条款中，你可以假设我所写的关于 SGI 哈希容器的内容对于 STLport 哈希容器也同样适用。

哈希容器是关联容器，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，哈希容器也要求指定一个哈希函数。这意味着，哈希容器的声明应该如下所示：

```
template<typename T,
         typename HashFunction,
         typename CompareFunction,
         typename Allocator = allocator<T> >
class hash_container;
```

这与 SGI 的哈希容器声明很接近，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）：

```
template<typename T,
         typename HashFunction = hash<T>,
         typename CompareFunction = equal_to<T>,
         typename Allocator = allocator<T> >
class hash_set;
```

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 的哈希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。对于哈希容器，这个决定不无道理，因为哈希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 哈希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、哈希函数类型、比较函数类型和分配子类型，但是它把默认的哈希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）：

```
template<typename T, typename CompareFunction>
class hash_compare;

template<typename T,
         typename HashingInfo = hash_compare<T, less<T> >,
         typename Allocator = allocator<T> >
class hash_set;
```

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哈希函数和比较函数，但同时还含有一些枚举值，用于控制哈希表中桶的最小数目，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，哈希表的桶数目将增加，表中的某些元素要被重新做哈希计算。（SGI 提供了一些成员函数来实现类似的控制功能，通过它们可以控制哈希桶的个数，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：

```
template<typename T, typename CompareFunction = less<T> >
class hash_compare {
public:
    enum {
        bucket_size = 4;                      //元素个数与桶个数的最大比率
        min_buckets = 8;                      //最小的桶数目
    };
    size_t operator()(const T&) const;      //哈希函数
    bool operator()(const T&,                //比较函数
                    const T&) const;
    ...
};                                         //省略了其他细节，包括对
                                            //CompareFunction 的使用
```

重载 operator() 的做法（在这种情况下，同时实现了哈希函数和比较函数）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，你都可以让哈希实现来决定所有的策略。你可以简单地这样写：

```
hash_set<int> intTable; //创建一个 int 类型的哈希集合
```

为了能够通过编译，哈希表必须包含整数类型（比如 `int`），因为默认的哈希函数通常局限于整数类型。（SGI 的默认哈希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哈希策略（open hashing scheme），由指向元素的单向链表的指针数组（桶）构成。Dinkumware 同样使用了开放式哈希策略，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 的哈希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 的哈希表实现中没有 `rbegin` 和 `rend` 成员函数。Dinkumware 的哈希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 本身没有哈希容器，但是与 STL 兼容的哈希容器（只是其接口、能力和行为各有不同）却并不难得到。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。

# 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。

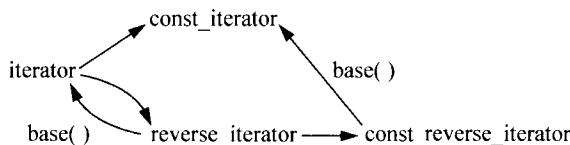
让我们来看两点。首先看一下 `vector<T>` 容器中 `insert` 和 `erase` 函数的原型：

```
iterator insert(iterator position, const T& x);
iterator erase(iterator position);
iterator erase(iterator rangeBegin, iterator rangeEnd);
```

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到了 `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：

```
typedef deque<int> IntDeque;           //typedef 可以极大地简化 STL 容器类
typedef IntDeque::iterator Iter;        //和 iterator 类型的用法
typedef IntDeque::const_iterator ConstIter;
Iter i;
ConstIter ci;                         //使 ci 和 i 指向同一容器
...                                     //比较一个 iterator 和
if (i == ci) ...                      //一个 const_iterator
```

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。惟一不寻常的地方是，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：

```
if (ci == i)...                         //当上面的比较不能编译的时候，这是解决方法
```

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：

```
if (i - ci >= 3) ...                   //如果 i 与 ci 之间至少有三个元素
```

如果迭代器的类型不同，你（完全正确）的代码也可能被（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：

```
if ( ci + 3 <= i ) ...           //当上面的 if 语句不能编译的时候，这是解决方法
```

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杀手锏。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`：

```
typedef deque<int> IntDeque;           //类型定义，简化代码
typedef IntDeque::iterator Iter;
typedef IntDeque::const_iterator ConstIter;

ConstIter ci;                         //ci 是一个 const_iterator
...
Iter i(ci);                          //编译错误！从 const_iterator 到
                                     //iterator 没有隐式转换途径
Iter i(const_cast<Iter>(ci));        //仍然是编译错误！不能将
                                     //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 条介绍的哈希容器)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`<sup>1</sup>，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。

```
typedef deque<int> IntDeque;           // 同前
typedef IntDeque::iterator Iter;
typedef IntDeque::const_iterator ConstIter;

IntDeque d;
ConstIter ci;
...
Iter i(d.begin());           // 使 i 指向 d 的起始位置
advance(i, distance(i, ci)); // 移动 i，使它指向 ci 所指的位置
// (下面说明了为什么需要修改才能编译)
```

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

<sup>1</sup> 我曾经发现这里介绍的这种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/estl1e-errata.html> 上对本书英文原版第 121 页的 8/22/01 注释。

<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：

```
template<typename InputIterator>
typename iterator_traits<InputIterator>::difference_type
distance(InputIterator first, InputIterator last);
```

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：

```
template<typename InputIterator>
typename iterator_traits<InputIterator>::difference_type
distance(InputIterator first, InputIterator last);
```

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：

```
advance(i, distance(i, ci)); //移动 i, 使它指向 ci 所指的位置
```

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。

```
advance(i, distance<ConstIter>(i, ci)); //将 i 和 ci 都当作 const_iterator,
//计算出它们之间的距离, 然后将 i 移动
//这段距离
```

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，以及某些哈希容器实现(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。

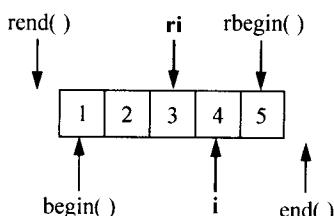
调用 `reverse_iterator` 的 `base()` 成员函数可以得到“与之相对应的” `iterator`，但是这句话实际上并没有说明其真正的含义。作为一个例子，先来看一下下面这段代码，它把数值 1 到 5 放进一个 `vector` 中，然后将一个 `reverse_iterator` 指向数值 3，并且通过其 `base()` 函数初始化一个 `iterator`：

```
vector<int> v;
v.reserve(5); //见第 14 条

for(int i = 1; i <= 5; ++ i) { //插入 1 到 5
    v.push_back(i);
}

vector<int>::reverse_iterator ri = //使 ri 指向 3
    find(v.rbegin(), v.rend(), 3);
vector<int>::iterator i(ri.base()); //使 i 与 ri 的基相同
```

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：

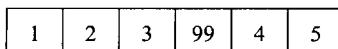


如图所示，在 `reverse_iterator` 与对应的由 `base()` 产生的 `iterator` 之间存在偏移，这段偏移也正好勾画出了 `rbegin()` 和 `rend()` 与对应的 `begin()` 和 `end()` 之间的偏移。但仅知道这些还远远不够。特别是，对于希望在 `ri` 上所要执行的操作，你如何通过 `i` 来执行呢？该图并没有体现出这一点。

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。

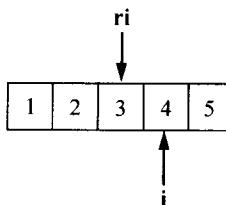
假设你要在 `ri` 所指定的位置上插入一个新的元素到 `v` 中，并且假设你要插入的值是 99。记住，在上图中 `ri` 遍历 `vector` 的顺序是自右向左，而 `insert` 操作会将新元素插入到其参数所指定位置的元素的前面，我们期望 99（按照逆向的遍历顺序）将会出现在 3 的前面。因此，插入操作之后，`v` 的布局应该如下图所示：



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,  $r_i$  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 `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，其中暗藏着惊悚之外：

```
vector<int> v;
...
vector<int>::reverse_iterator ri =
    find(v.rbegin(), v.rend(), 3); //同上，使 ri 指向 3
```

```
v.erase(--ri.base()); //试图删除 ri.base()前面位置上的元素;  
//对于 vector, 往往编译通不过
```

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，如果你在你的 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！

```
... //同上  
v.erase((++ri).base()); //删除 ri 所指的元素；这下编译没问题了！
```

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拷贝到一个`string`对象中，以下的代码看上去是一种合理的解决方案：

```
ifstream inputFile("interestingData.txt");  
string fileData((istream_iterator<char>(inputFile)), //将 inputFile 读入  
                istream_iterator<char>()); //fileData; 为什么  
//这是不正确的，请看  
//下文解释；关于该语  
//句的一条警告，请参  
//阅第 6 条
```

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拷贝到 `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：

```
ifstream inputFile("interestingData.txt");

```

现在，`inputFile` 中的所有字符都会被拷贝到 `fileData` 中。

然而，你可能会发现整个拷贝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：

```
ifstream inputFile("interestingData.txt");
string fileData((istreambuf_iterator<char>(inputFile)),
    istreambuf_iterator<char>());
```

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` 只是多输入了三个字符，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。

# 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;
```

```

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

```

...
list<int> results;                                //同前
transform(values.begin(), values.end(),           //现在 results 是一个 list
          front_inserter(results),                 //将 transform 的结果以逆向顺序
          transmogrify);                        //插入到 result 的头部

```

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：

```
list<int> results; //同前
transform(values.rbegin(), values.rend(), //将 transform 的结果插入到
         front_inserter(results), //results 的头部，并保持对象
         transmogrify); //的相对顺序
```

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：

```
vector<int> values; //同前
...
vector<int> results; //同前，只是在调用 transform 之前，
... //先在 results 中加入了一些数据
transform(values.begin(), values.end(),
         inserter(results, results.begin() + results.size()/2),
         transmogrify); //将 transmogrify 的结果插入到
                     //results 容器的中间位置上
```

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：

```
vector<int> values; //同上
vector<int> results;
...
results.reserve(results.size() + values.size()); //在 results 中为 values
                                                 //预留存储空间
transform(values.begin(), values.end(),
         inserter(results, results.begin() + results.size()/2),
         transmogrify); //同上，但这次 results 不需要
                      //重新分配内存空间
```

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):

```
vector<int> values;                                //同上
vector<int> results;
...
Results.reserve(results.size() + values.size());    //同上
transform(values.begin(), values.end(),
          results.end(),                                //变换的结果会被写入到
          transmogrify);                               //尚未初始化的内存,
                                                //结果将是不确定的
```

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:

```
vector<int> values;                                //同上
vector<int> results;
...
results.reserve(results.size() + values.size());    //同上
transform(values.begin(), values.end(),
          back_inserter(results),                      //将变化的结果加入到
          transmogrify);                            //results 的尾部, 同时避
                                                //免内存的重新分配
```

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。

```
vector<int> values;
vector<int> results;
...
```

```

if (results.size() < values.size()) {           //确保 results 至少
    results.resize(values.size());             //和 values 一样大
}
transform(values.begin(), values.end(),
         results.begin(),
         transmogrify);                         //覆盖 results 中的前
                                                //values.size() 个元素

```

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：

```

...
results.clear();                                //清除 results 中的所有元素
results.reserve(values.size());                 //预留足够的空间
transform(values.begin(), values.end(),
         back_inserter(results),               //将 transform 的结果放
         transmogrify);                      //到 results 中

```

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下都是完美无缺的。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：

```

bool qualityCompare(const Widget& lhs, const Widget& rhs)
{
    //返回 lhs 的质量是否好于 rhs 的质量的结果
}
...

```

```

partial_sort(widgets.begin(),
               widgets.begin() + 20,
               widgets.end(),
               qualityCompare);
...

```

//将质量最好的 20 个元素  
//顺序放在 widgets 的前  
//20 个位置上  
//使用 widgets

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，以此类推。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：

```

nth_element(widgets.begin(),
             widgets.begin() + 19,
             widgets.end(),
             qualityCompare);

```

//将最好的 20 个元素放在  
//widgets 的前部，但并不  
//关心它们的具体排列顺序

如你所见，对 `nth_element` 的调用与 `partial_sort` 基本上完全相同<sup>1</sup>。在效果上惟一不同之处在于：`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（质量其次），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？

<sup>1</sup> 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一个元素（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）。

`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：

```

vector<Widget>::iterator begin(widgets.begin()); //两个便捷变量，分别代表
vector<Widget>::iterator end(widgets.end());      //widgets 的 begin 和 end
                                                       //迭代器
vector<Widget>::iterator goalPosition;           //用于定位感兴趣的元素

//下面的代码找到具有中间质量级别的 Widget
goalPostion = begin + widgets.size() / 2;          //如果全排序的话，待查找
                                                       //的 Widget 应该位于中间
                                                       //找到 widgets 的中间质量值
nth_element(begin, goalPosition, end,
            qualityCompare);                         ...

                                                       //现在 goalPosition 所指
                                                       //的元素具有中间质量

//下面的代码找到区间中具有 75% 质量的元素
vector<Widget>::size_type goalOffset =
    0.25 * widgets.size();                      //找出如果全排序的话，待查找
                                                       //的 Widget 离起始处有多远

nth_element(begin, begin + goalOffset, end,
            qualityCompare);                      //找到 75% 处的质量值

```

```
... //现在 begin + goalOffset  
//所指的元素具有 75% 的质量
```

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:

```
bool hasAcceptableQuality(const Widget& w)  
{  
    //判断 w 的质量值是否为 2 或者更好  
}
```

然后,将这个函数传给 `partition` 算法:

```
vector<Widget>::iterator goodEnd = //将满足 hasAcceptableQuality  
partition(widgets.begin(), //的所有元素移到前部,然后返回一个  
        widgets.end(), //迭代器,指向第一个不满足条件  
        hasAcceptableQuality); //的 Widget
```

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` 中的元素拷贝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：

- |                                  |                              |
|----------------------------------|------------------------------|
| 1. <code>partition</code>        | 4. <code>partial_sort</code> |
| 2. <code>stable_partition</code> | 5. <code>sort</code>         |
| 3. <code>nth_element</code>      | 6. <code>stable_sort</code>  |

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：

```
template<class ForwardIterator, class T>
ForwardIterator remove(ForwardIterator first, ForwardIterator last,
                      const T& value);
```

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呢。惟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惟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：

```
vector<int> v;                                // 创建一个 vector<int>, 并
v.reserve(10);                                 // 填入 1-10。 (关于 reserve
for(int i = 1; i <= 10; ++i) {                  // 的解释见第 14 条。 }
    v.push_back(i);
}
cout << v.size();                               // 输出 10
v[3] = v[5] = v[9] = 99;                         // 使 3 个元素等于 99
remove(v.begin(), v.end(), 99);                 // 删除所有值等于 99 的元素
cout << v.size();                               // 仍然输出 10!
```

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：

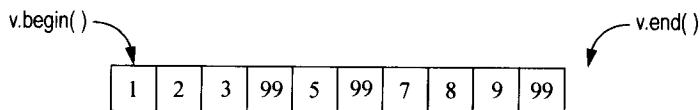
`remove` 不是真正意义上的删除，因为它做不到。

`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”。

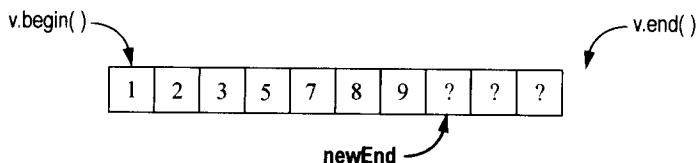
就我们所举的例子而言，调用 `remove` 之前 `v` 的布局如下：



如果我们将 `remove` 的返回值保存在一个新的迭代器对象 `newEnd` 中，

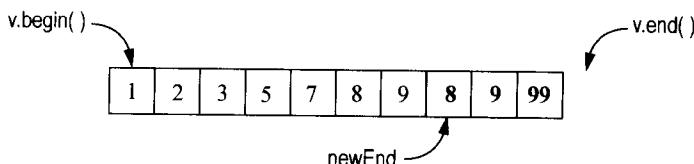
```
vector<int>::iterator newEnd(remove(v.begin(), v.end, 99));
```

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

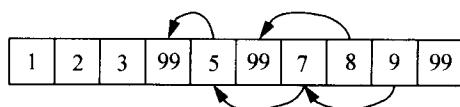
后你还可以找得到的特定地方，所以它没有这样做。这有什么问题吗？（如果你不想失去这些值，那么就应该使用 `partition` 而不是 `remove`，关于 `partition` 的解释请参阅第 31 条。）

`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：

```
vector<int> v; // 同前
...
```

```

v.erase(remove(v.begin(), v.end(), 99), v.end()); //真正删除所有值
//等于 99 的元素
cout << v.size(); //现在返回 7

```

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，以至它们被合并起来融入到了 `list` 的 `remove` 成员函数中。这是 STL 中惟一一个名为 `remove` 并且确实删除了容器中元素的函数：

```

list<int> li; //创建一个 list
...
li.remove(99); //加入一些值
//删除所有值为 99 的元素，因为是真正
//的删除，所以 li 的大小会改变

```

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` 并不是惟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，无需我多说了；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：

```

class Widget{
public:
...
bool isCertified() const; //该 Widget 是否已被验证过
...
};

vector<Widget*> v; //创建一个 vector，并用一些

```

```

...
v.push_back(new Widget);           //指向动态分配的 Widget 对象
...                                //的指针来填充该 vector

```

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。在本例子的情形中，当然应该使用 `remove_if`:

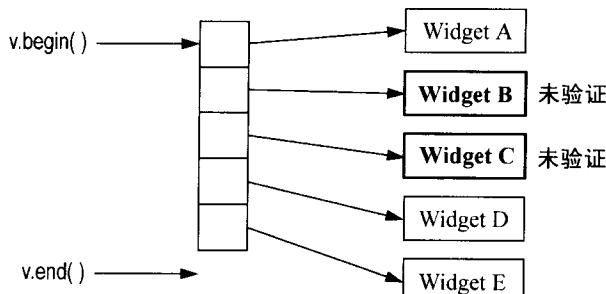
```

v.erase(remove_if(v.begin(), v.end(),
                  not1(mem_fun(&Widget::isCertified))), //未被验证过的
         v.end());                           //Widget 对象的
                                              //指针；关于
                                              //mem_fun 请参
                                              //阅第 41 条

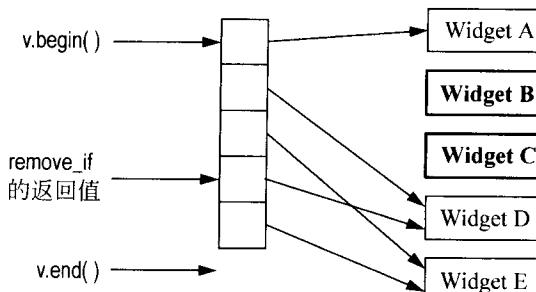
```

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。



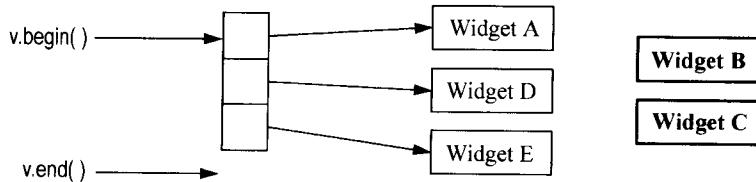
在调用了 `remove_if` 以后，v 往往应该如下图所示（图中也指明了 `remove_if` 返回的迭代器）：



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, 它们永远不会再被删除了, 它们所占用的内存和其他资源永远不会再被释放。

一旦 `remove_if` 和 `erase` 都返回之后, 则情形如下:



这使得资源泄漏的情况更加明显, 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。

```

void delAndNullifyUncertified(Widget*& pWidget) //如果*pWidget是一个
{
    If (!pWidget->isCertified()) {           //未被验证的Widget,
        delete pWidget;                      //则删除该指针,
        pWidget = 0;                          //并把它置成空
    }
}
for_each(v.begin(), v.end(),           //将所有指向未被验证的
         delAndNullifyUncertified);      //Widget 对象的指针删除
                                         //并置成空
v.erase(remove(v.begin(), v.end(),     //删除 v 中的空指针; 必须将
              static_cast<Widget*>(0)),   //0 转换成一个指针, 这样 C++
         v.end());                      //才能正确推断出 remove 的
                                         //第三个参数的类型

```

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：

```
template<typename T>
class RCSP{...};
typedef RCSP<Widget> RCSPW;
vector<RCSPW> v;
...
v.push_back(RCSPW(new Widget));           //RSCP = "Reference Counting"
                                         //          Smart Pointer"
                                         //RSCPW = "RSCP to Widget"
                                         //创建一个 vector，并将一些智能指针
                                         //加入到 vector 中；这些智能指针指向
                                         //动态分配的 Widget
...
v.erase(remove_if(v.begin(), v.end(),
                  not1(mem_fun(&Widget::isCertified))), //删除那些指向未
                                         //被验证的 Widget
v.end());                                //的指针；没有资源泄漏
```

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 `multiset` 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：

binary_search	lower_bound
upper_bound	equal_range
set_union	set_intersection
set_difference	set_symmetric_difference
merge	inplace_merge
includes	

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：

unique	unique_copy
--------	-------------

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。但让我们先看一下 C++ 标准是如何描述 `unique` 算法的行为的(斜体是我标的)：

Eliminates all but the first element from every *consecutive* group of equal elements.

(删除每一组连续相等的元素，仅保留其中的第一个。)

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“惟一”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` 类似的方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。

```
vector<int> v;                                // 创建一个 vector
...
sort(v.begin(), v.end(), greater<int>());      // 按降序进行排列
...                                              // 对数组进行操作(不改变值)

bool a5Exists =                                     // 在数组中查找 5，但假设
binary_search(v.begin(), v.end(), 5);            // 这个区间是升序排列的!
```

`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：

```
bool a5Exists = //在查找 5 的时
    binary_search(v.begin(), v.end(), 5, greater<int>()); //候也使用
                                                               //greater 作为
                                                               //比较函数
```

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，它涉及到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。

```

int ciCharCompare(char c1, char c2)           //忽略大小写地比较字符 c1 和 c2。
{
    //如果 c1<c2, 返回-1; 如果 c1>c2, 返回 1
    //返回 0; 如果 c1==c2, 返回 0

    int lc1 = tolower(static_cast<unsigned char>(c1));      //看下面关于这两
    int lc2 = tolower(static_cast<unsigned char>(c2));      //条语句的说明

    if (lc1 < lc2) return -1;
    if (lc1 > lc2) return 1;
    return 0;
}

```

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>`(因此`<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` 来表达的惟一办法是, 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。

```

int ciStringCompareImpl(const string &s1, //该函数的实现见下面
                       const string &s2);
int ciStringCompare(const string& s1, const string& s2)
{
    If (s1.size() <= s2.size()) return ciStringCompareImpl(s1, s2);
    else return -ciStringCompareImpl(s2, s1);
}

```

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:

```

int ciStringCompareImpl(const string &c1, const string &c2)
{
    typedef pair<string::const_iterator,
                 string::const_iterator>PSCI; //string::const_iterator"

    PSCI p = mismatch( //关于为什么用 not2 的
        s1.begin(), s1.end(), //解释请看下文, 关于
        s2.begin(), //ptr_fun 请参阅第 41 条
        not2(ptr_fun(ciCharCompare)));
}

if (p.first == s1.end()) { //如果为 true, 要么 s1 和 s2
    if(p.second == s2.end()) return 0; //相等, 或者 s1 比 s2 短
    else return -1;
}
return ciCharCompair(*p.first, *p.second); //字符串之间的关系和这两个
} //不匹配的字符之间的关系相同

```

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。惟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`:

```

bool ciCharLess(char c1, char c2) //返回在忽略大小写的情况下, c1
{
    return //是否在 c2 之前; 第 46 条将说
           //明为什么用函数对象更适合
    tolower(static_cast<unsigned char>(c1)) <
    tolower(static_cast<unsigned char>(c2));

```

```
}

bool ciStringCompare(const string &s1, const string &s2)
{
    return lexicographical_compare(s1.begin(), s1.end(),
                                  s2.begin(), s2.end(),
                                  ciCharLess);
}
```

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 `strcmpl` 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` 转换成 `const char*` 指针（见第 16 条），然后调用 `strcmp` 或 `strcmpl`：

有人可能会把这认为是一种取巧，但是，`strcmp/strcmpl` 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：

<code>copy</code>	<code>copy_backward</code>
<code>replace_copy</code>	<code>reverse_copy</code>
<code>replace_copy_if</code>	<code>unique_copy</code>
<code>remove_copy</code>	<code>rotate_copy</code>
<code>remove_copy_if</code>	<code>partial_sort_copy</code>
<code>uninitialized_copy</code>	

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，你却需要自己来实现。

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：

```
bool isDefective(const Widget& w)
```

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：

```
vector<Widget> widgets;
...
copy_if(widgets.begin(), widgets.end(),           //这不能通过编译
        ostream_iterator<Widget>(cerr, "\n"),       //STL 中没有 copy_if
        isDefective);
```

具有讽刺意味的是，`copy_if` 是最初的 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`:

```
template<typename InputIterator,
         typename OutputIterator,
         typename Predicate>
OutputIterator copy_if(InputIterator begin,
                      InputIterator end,
                      OutputIterator destBegin,
                      Predicate p)
{
    return remove_copy_if(begin, end, destBegin, not1(p));
}
```

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:

```
copy_if(widgets.begin(), widgets.end(),           //目的很好但
        ostream_iterator<Widget>(cerr, "\n"),      //通不过编译
        isDefective);
```

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, 函数指针必须首先用 `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:

```
template<typename InputIterator,           //copy_if 的一个正确实现
         typename OutputIterator,
         typename Predicate>
OutputIterator copy_if(InputIterator begin,
                      InputIterator end,
                      OutputIterator destBegin,
                      Predicate p)
{
    while (begin != end) {
```

```
    if (p(*begin)) *destBegin++ = *begin;
    ++begin;
}
return destBegin;
}
```

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，它和其他三个“数值算法”位于`<numeric>`中。（另外的三个算法分别是 `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：

```
list<double> ld; //创建一个 list，并加  
... //入一些 double 值  
double sum = accumulate( ld.begin(), ld.end(), 0.0); //计算它们的和，初始  
//值为 0.0
```

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：

```
double sum = accumulate(ld.begin(), ld.end(), 0); //计算它们的和,  
//初始值为0
```

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` 的值是不正确的。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）：

```
cout << "The sum of the ints on the standard input is"      // 打印出 cin
    << accumulate(istream_iterator<int>(cin),                  // 中 int 的和
                  istream_iterator<int>(),
                  0);
```

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：

```
string::size_type                                // 关于 string::size_type,
stringLengthSum(string::size_type sumSoFar,        // 请见下文
                const string& s)
{
    return sumSoFar + s.size();
}
```

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`:

```
set<string> ss;
...
string::size_type lengthSum =
    accumulate(ss.begin(), ss.end(),
               static_cast<string::size_type>(0),
               stringLengthSum);
```

//创建一个存放 string 的  
//容器，然后加入字符串  
//对 ss 中的每个元素调用 string-  
//LengthSum，然后把结果赋给  
//lengthSum。0 作为初始值  
//stringLengthSum；

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 `multiples` 函数子类(functor class)：

```
vector<float> vf;
...
float product =
    accumulate(vf.begin(), vf.end(),
               1.0f, multiples<float>());
```

//创建一个 float 容器并  
//加入一些数据  
//对 vf 中的每个元素调用 mul-  
//tiplies<float>，并把结果  
//赋给 product，初始值为 1.0f

这里惟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：

```
struct Point {
    Point(double initX, double initY): x(initX), y(initY) {}
    double x, y;
};
```

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：

```
list<Point> lp;
...
Point avg =
    accumulate(lp.begin(), lp.end(),
               Point(0,0), PointAverage());
```

//求出 lp 中的点的平均值

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` 的代码如下所示：

```
class PointAverage:  
    public binary_function<Point, Point, Point> { //请参阅第 40 条  
public :  
    PointAverage():xSum(0), ySum(0), numPoints(0){}  
  
    const Point operator()(const Point& avgSoFar, const Point& p)  
{  
        ++numPoints;  
        xSum += p.x;  
        ySum += p.y;  
        return Point(xSum/numPoints, ySum/numPoints);  
    }  
private:  
    size_t numPoints;  
    double xSum;  
    double ySum;  
}
```

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拷贝，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。

# 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拷贝到 `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（即被拷贝）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,
         DoSomething&>(di.begin(),
                         di.end(),
                         d);                      //用类型参数 DequeIntIter
                                            //和 DoSomething&来调用
                                            //for_each, 这将强制 d 按
                                            //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<sup>1</sup>。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，否则拷贝的开销会非常昂贵；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

<sup>1</sup> 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。

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*）：在对象拷贝过程中，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贅重，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>
class BPFCImpl:                                //针对修改后的 BPFC
    public unary_function<T, void> {           //的新的实现类
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拷贝动作。如果你是上述 `BPFC` 类的作者，那么你必须确保 `BPFC` 的拷贝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，惟一需要谨慎处理的就是 `BPFC` 的拷贝构造函数，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拷贝的。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，也就是说，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拷贝的函数对象。除此以外，对于用作判别式的函数对象，当它们被拷贝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拷贝，然后存放起来待以后再使用这些拷贝，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，并在第三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>` 中的第三个 `Widget`:

```

vector<Widget> vw;                      //创建容器，并添加
...                                         //一些 Widget
vw.erase(remove_if(vw.begin(),
                  vw.end(),
                  BadPredicate()),
         vw.end());                         //删除第三个元素，关于
                                            //如何使用 erase 和
                                            //remove_if，请参
                                            //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` 容器中的第三个元素，而且同时还删除了第六个！

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，是被拷贝到这两个算法中。（从技术上讲，并不一定这样做，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` 中第三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`）首先被拷贝到 `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三次。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` 的另一份拷贝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` 的一份拷贝。因此，当 `remove_copy_if` 第三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`（第三个和第六个），而不仅仅是你所期望的第三个元素。

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"
                                         // 的 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，惟一有点怪异的是，`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 WidgetNameCompare:
    public std::binary_function<Widget, 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），看看他们在调用特性（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++ 提供了三种不同的语法：

```
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
p->f();                                //是一个对象或对象的引用。
                                         //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。因此我们需要三个不同版本的 `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，拷贝构造函数总是完成拷贝的任务。（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，`operator==` 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。

# 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`，而什么时候又该使用 `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：

```
class Widget{
public:
    ...
    void redraw() const;
    ...
};
```

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：

```
list<Widget> lw;
...
for(list<Widget>::iterator i = lw.begin(); i != lw.end(); ++i){
    i->redraw();
}
```

你也可以选择使用 `for_each` 算法来完成：

```
for_each(lw.begin(), lw.end(),
        mem_fun_ref(&Widget::redraw)); //有关 mem_fun_ref 的
//信息，请参阅第 41 条
```

对于许多 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：

```
for(list<Widget>::iterator i = lw.begin(); i != lw.end(); ++i){
    i->redraw();
}
```

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()`：

```
for_each(lw.begin(), lw.end(),
        mem_fun_ref(&Widget::redraw)); //这个调用中只计算
//了一次 lw.end()
```

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（a）都是有效的；（b）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）：

```
//C API: 这个函数的输入参数是一个指向 double 数组的指针和该数组的大小,
//在函数中, 将向这个数组写入数据。它的返回值是实际被写入的 double 的个数。
size_t fillArray(double *pArray, size_t arraySize);
double data[maxNumDoubles]; //创建一个足够大的局部数组
deque<double> d; //创建 deque 并加入数据
...
size_t numDoubles =
    fillArray(data, maxNumDoubles); //从 API 取得数组
for(size_t i=0; i < numDoubles; ++i) { //把 data 中的每一个数据
    d.insert(d.begin(), data[i] + 41); //加上 41, 然后插入 d 的前
} //面。这段代码有问题!
```

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：

```
deque<double>::iterator insertLocation = d.begin();      //记住 d 的开始迭代器
for (size_t i = 0, i < numDoubles; ++i) {                //在 insertLocation 位置上
    d.insert(insertLocation++, data[i] + 41);   //插入 data[i]+41, 然后递
}                                                        //增 insertLocation。
                                                       //这段代码还有问题!
```

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：

```
deque<double>::iterator insertLocation =
    d.begin();                                         //同前
for (size_t i = 0; i < numDoubles; ++i) {
    insertLocation =                                //每次 insert 被调用之后
        d.insert(insertLocation, data[i] + 41);     //都更新 insertLocation,
    ++insertLocation;                                //以便保持迭代器有效,
}                                                    //然后递增迭代器
```

这段代码能够实现你真正想要的，但是想想你花了多少功夫才做到这一点！不妨和下面使用 `transform` 算法的代码做一下比较：

```
//从 data 中复制数据到 d 的前面，每个元素加上 41。
transform(data, data + numDoubles,           //将 data 中的元素拷贝
         inserter(d, d.begin()),            //到 d 的前部，每个元素
         bind2nd(plus<double>(), 41));   //加上 41
```

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，惟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初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的循环必须涉及到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，当你看到了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：

```
vector<int> v;
int x, y;
...
vector<int>::iterator i = v.begin();           //从 v.begin() 开始遍
for(; i != v.end(); ++i) {                     //历，直到找到合适的
    if (*i > x && *i < y) break;               //值，或者到达 v.end()
}
...
//现在 i 指向找到的值
//或者等于 v.end()
```

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，例如 SGI 的 `compose2`（见第 50 条）：

```
vector<int> ::iterator i =
    find_if(v.begin(), v.end(),           //找到第一个满足以下
            compose2(logical_and<bool>(),
                      bind2nd(greater<int>(), x),           //两个条件的 val:
                      bind2nd(less<int>(), y)));           //val>x 并且
                                                       //val<y 为真
```

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。

```
template<typename T>
class BetweenValues:
    public unary_function<T, bool> {           //见第 40 条
public:
    BetweenValues(const T& lowValue,           //用构造函数保存
                  const T& highValue)           //lowValue 和 highValue
        : lowVal(lowValue), highVal(highValue)
    {}
    bool operator()(const T& val) const       //检查 val 是否位于
    {                                         //lowValue 和 highValue
        return val > lowVal && val < highVal;   //之间
    }
private:
    T lowVal;
    T highVal;
```

```
};

...
vector<int>::iterator = find_if(v.begin(), v.end(),
    BetweenValues<int>(x, y));
```

但这也有它的局限性。第一，创建 `BetweenValues` 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`：

```
{
    ...
    template<typename T>
    class BetweenValues : public unary_function<T, bool> {...};
    vector<int>::iterator i = find_if(v.begin(), v.end(),
        BetweenValues<int>(x, y));
    ...
}
```

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：

```
{
    ...
    class BetweenValues : public unary_function<int, bool> {...};
    vector<int>::iterator i = find_if(v.begin(), v.end(),
        BetweenValues(x, y));
    ...
}
```

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事情很多，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<int> s;                                // 创建 set，并放入
...                                         // 一百万个值
set<int>::iterator i = s.find(727);          // 使用 find 成员函数
if (i != s.end()) ...
set<int>::iterator i = find(s.begin(), s.end(), 727); // 使用 find 算法
if (i != s.end()) ...
```

`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 1 000 000 次比较操作。即使 `set` 中的确包含了 727，`find` 算法仍然需要平均 500 000 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： 1 000 000 次（最坏情况） 500 000 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 1 000 000 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` 算法之间的惟一差别。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 条中介绍的非标准哈希容器，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拷贝 `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需任何对象拷贝，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拷贝对象要低得多，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，所以你无需再调用 `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。在接下去的讨论中，我将忽略 `count` 和 `find` 的 `_if` 形式

的算法，同样也会忽略 `binary_search`、`lower_bound`、`upper_bound` 以及 `equal_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，有多少个拷贝？”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`，则代码如下所示：

```
list<Widget> lw;                                //list 容器
Widget w;                                         //特定的 Widget 值
...
if (count(lw.begin(), lw.end(), w)) {           //w 在 lw 中
    ...
} else {                                         //w 不在 lw 中
    ...
}
```

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：

```
if (count(lw.begin(), lw.end(), w) != 0) ...
```

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：

```
if (find(lw.begin(), lw.end(), w) != lw.end()) {
    ...
} else {
    ...
}
```

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`：

```
list<Widget>::iterator i = find(lw.begin(), lw.end(), w);
if (i != lw.end()) {
    ...
} else {
    ...
}
//找到了, i 指向第一个满足条件的对象
//没有找到
```

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。）

```
vector<Widget> vw;                                //创建 vector，并加入数据
...
sort(vw.begin(), vw.end());                      //对 vector 进行排序
Widget w;                                         //待查找的值
...
if (binary_search(vw.begin(), vw.end(), w)) {
    ...
} else{
    ...
}
//w 在 vw 中
//w 不在 vw 中
```

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拷贝 (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, 它的第一份拷贝在哪里? 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`:

```
vector<Widget>::iterator i = lower_bound(vw.begin(), vw.end(), w);
if (i != vw.end() && *i == w) {           //确保 i 指向一个对象; 并且确保该
                                         //对象有正确的值; 这里有一个错误!
...
                                         //找到了该值, i 指向第一个具有
                                         //该值的对象
} else {
    ...
}
//没有找到
```

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:

```
if (i != vw.end() && *i == w) ...
```

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。首先, 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`:

```
vector<Widget> vw;
...
sort(vw.begin(), vw.end());
typedef vector<Widget>::iterator VWIter; //用类型定义提供方便
typedef pair<VWIter, VWIter> VWIterPair;
VWIterPair p = equal_range(vw.begin(), vw.end(), w);
if (p.first != p.second) { //如果 equal_range 返回非空区间...
    ...
    //找到了特定值, p.first 指向第一个
    //与 w 等价的对象, p.second 指向最
    //后一个与 w 等价的对象的下一个位置
} else{
    ...
    //没有找到特定值, p.first 和
    //p.second 都指向 w 的插入位置
}
```

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:

```
VWIterPair p = equal_range(vw.begin(), vw.end(), w);
cout << "There are" << distance(p.first, p.second)
    << "elements in vw equivalent to w.";
```

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:

```
class Timestamp {...};
bool operator<(const Timestamp& lhs, //判断 lhs 是否在 rhs 之前
                 const Timestamp& rhs);
vector<Timestamp> vt; //创建 vector
```

```
...                                //填充数据  
sort(vt.begin(), vt.end());        //按时间先后排序
```

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` 之前的时间戳对象。在这种情况下，我们并不想找到该区间中与 `ageLimit` 等价的时间戳对象，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。这是非常非常容易做到的，因为 `lower_bound` 会给我们一个精确的答案：

```
Timestamp ageLimit;  
...  
vt.erase(vt.begin(), lower_bound(vt.begin(),           //从 vt 中删除所有  
                                 vt.end(),             //在 ageLimit 之前  
                                 ageLimit));         //的对象
```

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：

```
vt.erase(vt.begin(), upper_bound(vt.begin(),           //从 vt 中删除所有  
                                 vt.end(),             //在 ageLimit 之前或者  
                                 ageLimit));         //与 ageLimit 等价的对象
```

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：

```
class Person{  
public:  
    ...  
    const string& name() const;  
    ...  
};  
  
struct PersonNameLess:  
public binary_function<Person, Person, bool> {      //见第 40 条  
bool operator()(const Person& lhs, const Person& rhs) const  
{  
    return lhs.name() < rhs.name();  
}  
};
```

```

list<Person> lp;
...
lp.sort(PersonNameLess());           //用 PersonNameLess 对 lp 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:

```

Person newPerson;
...
lp.insert(upper_bound(lp.begin(),           //把 newPerson 插入到 lp 中,
                     lp.end(),            //排在最后一个与 newPerson 等价
                     newPerson,           //之前或者与 newPerson 等价
                     PersonNameLess()),   //的对象之后
          newPerson);

```

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`:

```

set<Widget> s;                      //创建一个 set, 并加入数据
...
Widget w;                          //w 是待查找的值
...
if (s.count(w)) {                  //存在与 w 等价的值
    ...
} else {                           //不存在这样的值
    ...
}

```

而如果要在 `multiset` 和 `mymap` 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` 而言，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。首先，它的用法显得非常清晰，`count` 的含义就是“计数”。其次，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。

想知道什么？	使用算法		使用成员函数	
	对未排序的区间	对排序的区间	· 对 <code>set</code> 或 <code>map</code>	对 <code>multiset</code> 或 <code>mymap</code>
特定的值存在吗？	<code>find</code>	<code>binary_search</code>	<code>count</code>	<code>find</code>
特定的值存在吗？如果存在，那第一个有该值的对象在哪里？	<code>find</code>	<code>equal_range</code>	<code>find</code>	<code>find</code> 或 <code>lower_bound</code> (见下文)
第一个不超过特定值的对象在哪里？	<code>find_if</code>	<code>lower_bound</code>	<code>lower_bound</code>	<code>lower_bound</code>
第一个在某个特定值之后的对象在哪里？	<code>find_if</code>	<code>upper_bound</code>	<code>upper_bound</code>	<code>upper_bound</code>
具有特定值的对象有多少个？	<code>count</code>	<code>equal_range</code> (然后 <code>distance</code> )	<code>count</code>	<code>count</code>
具有特定值的对象都在哪里？	<code>find</code> (反复调用)	<code>equal_range</code>	<code>equal_range</code>	<code>equal_range</code>

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 `mymap`，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_search`、`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）。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：

```
vector<double> v;  
...  
sort(v.begin(), v.end(), greater<double>());
```

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：

```
inline  
bool doubleGreater(double d1, double d2)  
{  
    return d1 > d2;  
}  
...  
sort(v.begin(), v.end(), doubleGreater);
```

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，针对一个包含一百万个 `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，我们必须要明白，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：

```
sort(v.begin(), v.end(), doubleGreater);
```

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：

```
void sort(vector<double>::iterator first,           // 区间的开始
          vector<double>::iterator last,            // 区间的末尾
          bool (*comp)(double, double));           // 比较函数
```

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，但是在广泛使用的 STL 平台上却无法通过编译：

```
set<string> s;
...
transform(s.begin(), s.end(),
         ostream_iterator<string::size_type>(cout, "\n"),
         mem_fun_ref(&string::size));
```

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：

```
struct StringSize:
    public unary_function<string, string::size_type> {           // 见第 40 条
    string::size_type operator()(const string& s) const
    {
        return s.size();
    }
};

transform(s.begin(), s.end(),
         ostream_iterator<string::size_type>(cout, "\n"),
         StringSize());
```

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：

```

template<typename FPType> //返回两个浮点数的
FPType average(FPType val1, FPType val2) //平均值
{
    return (val1 + val2) / 2;
}

template<typename InputIter1,
         typename InputIter2>
void writeAverages(InputIter1 begin1, //将两个序列的值按顺序
                   InputIter1 end1, //对应取平均，然后写到
                   InputIter2 begin2, //一个流中
                   ostream& s)
{
    transform(
        begin1, end1, begin2,
        ostream_iterator<typename iterator_traits<InputIter1>::value_type(s,
            "\n")>,
        average<typename iterator_traits<InputIter1>::value_type> //错误?
    );
}

```

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：

```
template<typename FPType>
struct Average {
    public binary_function<FPType, FPType, FPType> { // 见第 40 条
        FPType operator()(FPType val1, FPType val2) const
    {
        return average(val1, val2);
    }
};

template<typename InputIter1, typename InputIter2>
void writeAverages(InputIter1 begin1, InputIter1 endl,
InputIter2 begin2, ostream& s)
{
```

```

    transform(
        begin1, endl, begin2,
        ostream_iterator<typename iterator_traits<InputIter1>::value_type>(s,
            "\n")>,
    Average<typename iterator_traits<InputIter1>::value_type>()
);
}

```

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？

```

vector<int> v;
int x, y;
...
v.erase(
    remove_if(find_if(v.rbegin(), v.rend(),
        bind2nd(greater_equal<int>(), y)).base(),
        v.end(),
        bind2nd(less<int>(), x)),
    v.end());

```

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。首先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：

```
v.f1(f2(f3(v.f4(), v.f5()), f6(f7(), y)).f8(), v.f9(), f6(f10(), x)), v.f9());
```

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。）

```
typedef vector<int>::iterator VecIntIter;

// 初始化 rangeBegin，使它指向 v 中大于等于 y 的最后一个元素之后的元素。
// 如果不存在这样的元素，则 rangeBegin 被初始化为 v.begin()。
// 如果最后这样的元素正好是 v 的最后一个元素，则 rangeBegin 被初始化为 v.end()。
VecIntIter rangeBegin=find_if(v.rbegin(), v.rend(),
                               bind2nd(greater_equal<int>(), y)).base();

// 在从 rangeBegin 到 v.end() 的区间中，删除所有小于 x 的值
v.erase(remove_if(rangeBegin, v.end(), bind2nd(less<int>(), x)), v.end());
```

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其值不小于 `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。

```
v.erase(remove_if(find_if(v.rbegin(), r.rend(), something).base(),
                  v.end(),
                  something),
        v.end());
```

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”的代码（`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“直写型的代码”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：

```
vector<int> v(10); //创建一个 size 为 10 的 vector
```

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：

```
string s(10); //试图创建一个 size 为 10 的 string
```

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。下面是我一个 STL 平台针对此错误给出的诊断消息：

```
example.cpp(20): error C2664: '__thiscall std::basic_string<char, struct std::char_traits<char>, class std::allocator<char>>::basic_string<char, struct std::char_traits<char>, class std::allocator<char> >(const class std::allocator<char> &)' : cannot convert parameter 1 from 'const int' to 'const class std::allocator<char> &'  
Reason: cannot convert from 'const int' to 'const class std::allocator<char>'  
No constructor could take the source type, or constructor overload resolution was ambiguous
```

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过的分配子；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：

```
basic_string<char, char_traits<char>, allocator<char> >
```

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：

```
std::basic_string<char, std::char_traits<char>, std::allocator<char> >
```

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`：

```
basic_string<char, string_char_traits<char>, __default_alloc_template<false, 0> >
```

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所示：

```
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，这是其三个构造函数中惟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<string, 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：

```
void NiftyEmailProgram::showEmailAddress(const string& nickname) const
{
    ...
    NicknameMap::iterator i = nicknames.find(nickname);
    if (i != nicknames.end()) ...
    ...
}
```

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：

```
example.cpp(17):error C2440:'initializing':cannot convert from 'class std::_Tree<class
```

```
std::basic_string<char, struct std::char_traits<char>, class std::allocator <char> >, struct  
std::pair<class std::basic_string<char, struct std::char_traits<char>, class  
std::allocator<char> > const , class std::basic_string<char, struct  
std::char_traits<char>, class std::allocator<char> >, struct std::map<class  
std::basic_string<char, struct std::char_traits<char>, class std::allocator <char> >, class  
std::basic_string<char, struct std::char_traits<char>, class std::allocator <char> >, struct  
std::less<class std::basic_string<char, struct std::char_traits<char>, class  
std::allocator<char> > >, class std::allocator<class std::basic_string<char, struct  
std::char_traits<char>, class std::allocator<char> > >::_Kfn, struct std:: less<class  
std::basic_string<char, struct std::char_traits<char>, class std::allocator <char> >, class  
std::allocator<class std::basic_string<char, struct std::char_traits<char>, class  
std::allocator<char> > > >::const_iterator' to 'class std::_Tree<class  
std::basic_string<char, struct std::char_traits<char>, class std::allocator <char> >, struct  
std::pair<class std::basic_string<char, struct std::char_traits<char>, class  
std::allocator<char> > const , class std::basic_string<char, struct  
std::char_traits<char>, class std::allocator<char> >, struct std::map<class  
std::basic_string<char, struct std::char_traits<char>, class std::allocator <char> >, class  
std::basic_string<char, struct std::char_traits<char>, class std::allocator <char> >, struct  
std::less<class std::basic_string<char, struct std::char_traits<char>, class  
std::allocator<char> > >, class std::allocator<class std::basic_string<char, struct  
std::char_traits<char>, class std::allocator<char> > >::_Kfn, struct std:: less<class  
std::basic_string<char, struct std::char_traits<char>, class std::allocator <char> >, class  
std::allocator<class std::basic_string<char, struct std::char_traits<char>, class  
std::allocator<char> > > >::iterator'
```

No constructor could take the source type, or constructor overload resolution was ambiguous

这条诊断信息足有 2 095 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 4 812 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：

```
example.cpp(17) : error C2440: 'initializing' : cannot convert from 'class  
std::_Tree<class string, struct std::pair<class string const , class string  
>, struct std::map<class string, class string, struct std::less<class string  
>, class std::allocator<class string > >::_Kfn, struct std::less<class string  
>, class std::allocator<class string > >::const_iterator' to 'class  
std::_Tree<class string, struct std::pair<class string const , class string  
>, struct std::map<class string, class string, struct std::less<class string  
>, class std::allocator<class string > >::_Kfn, struct std::less<class string
```

```
>, class std::allocator<class string > >::iterator'

No constructor could take the source type, or constructor overload
resolution was ambiguous
```

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：

```
example.cpp(17) : error C2440: 'initializing' : cannot convert from 'class
std::_Tree<class string,struct std::pair<class string const ,class string
>,struct NicknameMap::_Kfn,struct std::less<class string >,class
std::allocator<class string > >::const_iterator' to 'class std::_Tree<class
string,struct std::pair<class string const ,class string >,struct
NicknameMap::_Kfn,struct std::less<class string >,class std::allocator<
class string > >::iterator'

No constructor could take the source type, or constructor overload
resolution was ambiguous
```

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，惟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：

```
example.cpp(17) : error C2440: 'initializing' : cannot convert from 'class
```

```
std::Tree<SOMETHING>::const_iterator' to 'class
std::Tree<SOMETHING>::iterator'

No constructor could take the source type, or constructor overload
resolution was ambiguous
```

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：

```
class NiftyEmailProgram {
private:
    typedef map<string, string> NicknameMap;
    NicknameMap nicknames;
public:
    ...
    void showEmailAddress(const string& nickname) const;
};

void NiftyEmailProgram::showEmailAddress(const string& nickname) const
{
    ...
NicknameMap::iterator i = nicknames.find(nickname);
    if (i != nickname.end()) ...
    ...
}
```

惟一可能的解释是，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,struct 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：

```
vector<string*> v;           //试图将 string *指针的容器
copy(v.begin(), v.end(),    //按照 string 对象进行输出
     ostream_iterator<string>(cout, "\n"));
```

- 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：

```
list<int>::iterator i1, i2;      //将一个双向迭代器传递给
sort(i1, i2);                  //一个需要随机访问迭代器
                                //的算法
```

- 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：

- 哈希关联容器 `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所示：

```
map<int, string> m;  
...  
// 将 map 所有的键输出到 cout  
transform(m.begin(), m.end(),  
         ostream_iterator<int>(cout, "\n"),  
         select1st<map<int, string>::value_type>());  
// 创建一个 vector，并将 map 中所有的值拷贝到该 vector 中  
vector<string> v;  
transform(m.begin(), m.end(), back_inserter(v),  
         select2nd<map<int, string>::value_type>());
```

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 是目前惟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：

```
int transmogrify(int x);           //该函数根据 x 产生一个新的值
vector<int> values;
...
vector<int> results;               //把数据添加到 values 中
transform(values.begin(), values.end(),          //企图在超出 results 尾部的
         results.end(),                  //位置上写入元素!
         transmogrify);
```

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 目录下）：

```
C:\STLport\stlport\stl\debug\_iterator.h:265 STL assertion failure:
_Dereferenceable(*this)
```

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，在可以通过配置 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：

```
class Widget {  
public:  
    ...  
    int readStream(istream& stream);           //readStream 带一个  
    ...                                         //引用参数  
};  
vector<Widget*> vw;  
...  
for_each(                                         //大多数 STL 平台试图  
    vw.begin(), vw.end(),  
    bind2nd(mem_fun(&Widget::readstream), cin) //在这个调用中生成一  
);                                              //个指向引用的引用;  
                                                 //这样的代码不能通过编译
```

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 书 目

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

- [3] Nicolai M. Josuttis, *The C++ Standard Library: A Tutorial and Reference*, Addison-Wesley, 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/stl/>) 上各种资源的印刷版本。
- [5] ISO/IEC, *International Standard Programming Languages — C++*, Reference Number ISO/IEC 14882:1998 (E), 1998. 描述 C++ 的官方文档，通过 ANSI 的网址 <http://web>-

[store.ansi.org/ansidocstore/default.asp](http://store.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 0-201-61562-2. 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. 惟一一本专门论述 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 对哈希容器的处理办法（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://www.creport.com/html/from\\_pages/view\\_recent\\_articles\\_c.frm?AtricleID=402](http://www.creport.com/html/from_pages/view_recent_articles_c.frm?AtricleID=402) 看到此文。
- [20] Herb Sutter, “Optimizations That Aren’t (In a 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/thread\\_safety.html](http://www.sgi.com/tech/stl/thread_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++ Experts 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](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 online extension to the magazine), May 2000, <http://www.cuj.com/experts/1905/austern.htm>。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

- 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/cd1e-errata.html>.
- [29] The *More Effective C++ auto\_ptr* Update page,  
[http://www.awl.com/cseng/titles/0-201-63371-X/auto\\_ptr.html](http://www.awl.com/cseng/titles/0-201-63371-X/auto_ptr.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到的地域性问题。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”（101communications）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<sup>1</sup>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())
```

---

<sup>1</sup> 请参阅 *C++ Report*[19] 2000 年 4 月期上的 Andrei Alexandrescu 的专栏。

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，它决定两个字符串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。我也不可能脱俗；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“GEWÜR-ZTRAMINER”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答案，甚至在涉及到 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。<sup>1</sup> `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，那么下面的代码片段将按照适合于地域 `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：

<sup>1</sup> 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`。

```
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种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  $[f, l)$  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拷贝到缓冲区中，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 23 791 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。

# 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 拷贝到另一个 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）：

```
vector<Widget> vw1, vw2;
...
vw1.insert(vw1.end(), vw2.begin(), vw2.end());           // 把 vw2 中的 Widget
                                                          // 拷贝到 vw1 的尾部
```

如果是一个 vector 和一个 deque，则可以使用同样的做法：

```
vector<Widget> vw;
deque<Widget> dw;
...
vw.insert(vw.end(), dw.begin(), dw.end());           // 把 dw 中的 Widget
                                                          // 拷贝到 vw 的尾部
```

实际上，不管被拷贝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：

```
vector<Widget> vw;
...
...
```

```
list<Widget> lw;
...
vw.insert(vw.begin(), lw.begin(), lw.end());           //把 lw 中的 Widget
                                                       //拷贝到 vw 的前部

set<Widget> sw;
...
vw.insert(vw.begin(), sw.begin(), sw.end());           //把 sw 中的 Widget
                                                       //拷贝到 vw 的前部

template<typename T,
          typename Allocator = allocator<T> >           //与 STL 兼容的自定义
class SpecialContainer {...};                         //容器模板
SpecialContainer<Widget> scw;
...
vw.insert(vw.end(), scw.begin(), scw.end());           //把 scw 中的 Widget
                                                       //拷贝到 vw 的尾部
```

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：

```
template<class T, class Allocator = allocator<T> >
class vector {
public:
    ...
    template <class InputIterator>
    void insert(iterator position, InputIterator first, InputIterator last);
    ...
};
```

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

到这些模板，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：

```
void insert(iterator position,
            iterator first, iterator last);      // "iterator" 是容器的
                                                // 迭代器类型
```

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：

```
istream_iterator<Widget> begin(cin), end;           // 创建两个迭代器 begin
                                                       // 和 end, 用于从 cin 中
                                                       // 读入 Widget(见第 6 条)
vector<Widget> vw(begin, end);                      // 将 cin 的 Widget 读入
                                                       // vw 中(同样参见第 6 条),
                                                       // 在 MSVC4-6 上不能编译

list<Widget> lw;
...
lw.assign(vw.rbegin(), vw.rend());                  // 将 vw 的内容赋给 lw,
                                                       // 但顺序相反;
                                                       // 在 MSVC4-6 上不能编译

SpecialContainer<Widget> scw;
...
scw.insert(scw.end(), lw.begin(), lw.end());        // 将 lw 中的 Widget 插入
                                                       // 到 scw 的尾部;
                                                       // 在 MSVC4-6 上不能编译
```

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：

```

vector<Widget> vw(begin, end);           // 在 MSVC4-6 的 STL
                                         // 实现中，被拒绝
list<Widget> lw;
...
lw.assign(vw.rbegin(), vw.rend());       // 也被拒绝
SpecialContainer<Widget> scw;
...
scw.insert(scw.end(), lw.begin(), lw.end()); // 同样

```

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：

```

istream_iterator<Widget> begin(cin), end;           // 使用默认构造的 vw,
vector<Widget> vw;                                 // 然后将 cin 中的
copy(begin, end, back_inserter(vw));                // Widget 拷贝到 vw 中
list<Widget> lw;
...
lw.clear();                                         // 清空原来的 Widget;
copy(vw.rbegin(), vw.rend(), back_inserter(lw));   // 将 vw 的 Widget 按相
                                                       // 反的顺序拷贝到 lw 中
SpecialContainer<Widget> scw;
...
copy(lw.begin(), lw.end(),                         // 将 lw 的 Widget 拷贝
      inserter(scw, scw.end()));                   // 到 scw 的尾部

```

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 中使用上一节中介绍的针对 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），包括那些涉及到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 本身兼容的 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，包括哈希容器（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，无需我多说，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。