# Effective STL

## 50 Specific Ways to Improve Your Use of the Standard Template Library

Scott Meyers

ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

# 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

我第一次写关于标准模板库的东西是在1995年，那时我决定把《More Effective C++》的最后一个条款写成一个STL的简要概览。我早该更好地了解STL。不久以后，我开始收到一些邮件，问我什么时候写《Effective STL》。

我把这个想法忍了几年。一开始，我对STL不够熟悉，所以不能给出关于它的建议。但随着时间的推移，我STL的经验丰富了，而主要问题出现在了其他方面。当一个程序库的在效率和可扩展性设计上表现出突破性的时候从来没有出过什么问题，但当开始使用STL时，这成了我无法预见的实际问题。迁移到一个几乎最简单的STL程序都成了一个挑战，不光是因为库的实现变化多端，而且因为现有编译器对模板支持有好有坏。STL的教材很难得到，所以学习“ STL的编程方式 ”很难；但即使跨越了这个障碍，找到正确易学的参考文档同样很困难。可能最令人畏惧的是，即使最小的STL使用错误也往往会导致一个编译器诊断的风暴——每一个错误都有上千个字长，而且大多涉及的类，函数或模板在令人厌恶的源代码中并没有被提及——几乎都是难以理解的。虽然我很钦佩STL和它背后的英雄们，但我还是觉得把STL推荐给实践中的程序员并不合适。我不能肯定*有可能*有效地使用STL。

然后我开始注意到一些让我感到惊奇的事情。尽管有很多小问题，尽管只有令人消沉的文档，尽管编译器的诊断信息像无线电信号杂音，但仍然有很多我的咨询客户在使用STL。而且，他们不只是玩玩而已，他们竟然把STL用到了产品的代码中！这是一个革命。我知道STL表现出的是一流的设计，但任何让程序员必须忍受移植性的麻烦、贫乏的文档和天书般的错误信息，却设计得很好的库也是不会被拥护的。我了解到越来越多的专业程序员都认为即使一个实现得很不好的STL也比什么都没有要好得多。

此外，我知道STL的境遇只会越来越好。程序库和编译器对（它们的）标准兼容性会越来越好，更好的文档将会出现（它已经存在了——请见从297页开始的“ 参考书目 ”），而且编译器的诊断会渐渐改进（在极大程度上，我们仍然在等待，但条款49提供了怎样在其间应付的建议）。因此我决定插嘴，尽一份力量来支持STL运动的萌芽。这本书就是结果：改善使用C++ STL的50个有效做法。

一开始，我计划在1999年下半年写这本书。带着这个想法，我组织了一个大纲。但我暂停和改变了进程。我

停止了写书的工作，开发了一个介绍性的STL训练课程，把它教给几拨不同的程序员。大约一年后，我回到写书的工作中，根据我在训练课程中得到的经验意味深长地修改了大纲。和我的《Effective C++》成功的方法一样，它们都是以真正的程序员所面对的问题为基础的。我希望《Effective STL》同样从事于STL编程的实践方面——这是对专业开发人员最重要的方面。

我总是在寻找能让我加深对C++理解的方法。如果你对STL编程有新的建议或者如果你对这本书有什么评论的话，请让我知道。另外，让本书尽可能地正确是我继续的目标，所以如果谁挑出了本书的任何一个错误请务必告诉我——不论是技术、语法、错别字或任何其他东西——我将在本书再次印刷的时候，把第一位挑出错误的读者大名加到致谢名单中。请将你的建议、见解、批评发至 estl@aristeia.com。

我维护有本书第一次印刷以来的修订记录，其中包括错误更正、文字修润、以及技术更新。这份记录可以从《Effective STL》的勘误表网站http://www.aristeia.com/BookErrata/estl1e-errata.html得到。

如果你希望在我对此书作出修改时得到通知，我想你应该加入我的邮件列表。我用这个列表来通知对我的C++工作感兴趣的人。详情请见http://www.aristeia.com/MailingList/。

SCOTT DOUGLAS MEYERS STAFFORD, OREGON
http://www.aristeia.com APRIL 2001

# 致谢

在我理解STL、建立关于它的培训课程和写这本书的大约两年时间里，我得到了大量帮助。在我所有帮助的来源中，有两个特别重要。 第一个是Mark Rodgers。当我建立它们时，Mark慷慨地自愿检查我的培训材料，而且我从他那里比从其他人那里学到更多关于STL的东西。他也作为本书的技术评论家，再次提供改进了几乎每个条款的观点和见解。

另一个突出的信息源是几个有关C++的Usenet新闻组，特别是comp.lang.c++.moderated（“ clcm ”），comp.std.c++和microsoft.public.vc.stl。十多年来，我已经依赖于像这样的新闻组中的参加者来回答我的问题和挑战我的思想，而且很难想象如果没有他们我该怎么办。我非常感谢Usenet社区对本书和我早先关于C++的出版物的帮助。

多种出版物形成了我对STL的理解，其中最重要的列在了参考书目里。我对Josuttis的《The C++ Standard Library》[3]倾向特别大。

本书基本上是其他人做的一些见解和观点的摘要，虽然有一些想法是我自己的。我已经努力跟踪我在哪里学到了什么，但这个任务是没希望的，因为一个典型的条款包含在很长时间内从很多来源收集的信息。所以跟踪是不完全的，但是我已经尽力了。请注意在这里我的目标是总结一个想法或技术我第一次是在哪里学到的，而不是一个想法或技术最先在哪里开发或谁先想到了它。

在条款1，我的基于节点的容器对事务性语义提供了更好的支持的观点是基于Josuttis的《The C++ Standard Library》第5.11.2节[3]。条款2包括一个来自Mark Rodgers的关于当分配器类型改变之后typedef起了什么作用的例子。条款5由Reeves的《C++ Report》专栏《STL Gotchas》[17]激发而来。条款8源自Sutter的《Exceptional C++》[8]的条款37，而Kevlin Henney提供了关于auto_ptr的容器在实际中为什么失效的重要细节。在Usenet的帖子里，Matt Austern提供了分配器什么时候很有用的例子，而我把他的例子包含在条款11。条款12基于SGI STL网站[21]上关于线程安全性的讨论。条款13关于引用计数在多线程环境下的性能影响的材料来自Sutter关于这个主题的文章[20]。条款15的主意来自Reeves的《C++ Report》专栏《Using Standard string in the Real World, Part 2》[18]。在条款16，Mark Rodgers提出了我演示的让C API直接把数据写入一个vector的技术。条款17包含的信息来自Siemel Naran和Cart Barron在Usenet的帖子。条款18是我从Sutter的《C++ Roport》专栏《When Is a Container Not a Container?》[12]偷来的。在条款20，Mark Rodgers捐献了把一个指针通过一个解引用仿函数转换成一个对象的主意，而Scott Lewandowski提出了我呈现的DereferenceLess版本。条款21源自Doug Harrison在microsoft.public.vc.stl的帖子，但我决定把那个条款的重点放在相等。条款22基于Sutter的《C++ Report》专栏《Standard Library News: sets and maps》[13]；Matt Austern帮我了解了标准委员会的库问题#103的情况。条款23是从Austern的《C++ Report》文章《Why You Shouldn't Use set -- and What to Use Instead》[15]得到的灵感；David Smallberg为我的DataCompare实现做出了一个优雅的改进。我对Dinkumware的散列容器的描述基于

Plauger的《C/C++ Users Journal》专栏《Hash Tables》[16]。Mark Rodgers并不同意条款26的全部建议，但那个条款的早先动机是他的有些容器的成员函数只接受iterator类型的实参的观点。我对条款29的处理来自和扩充自Matt Austern和James Kanze发起的Usenet讨论；我也受到Kreft和Langer的《C++ Report》文章《A Sophisticated Implementation of User-Defined Inserters and Extractors》[25]的影响。条款30是因为Josuttis的《The C++ Standard Library》[3]第5.4.2节的讨论。在条款31，Marco Dalla Gasperina捐献了使用nth_element计算中值的例子，而使用那个算法来寻找百分点直接来自Stroustrup的《The C++ Programming Language》[7]第18.7.1节。条款32受到Josuttis的《The C++ Standard Library》[3]第5.6.1节的材料影响。条款35源自Austern的《C++ Report》专栏《How to Do Case-Insensitive String Comparison》[11]，而James Kanze和John Potter的clcm帖子帮我精炼了我对出现这个问题的理解。Stroustrup的copy_if实现，也就是我在条款36演示的，来自他的《The C++ Programming Language》[7]第18.6.1节。条款39受到Josuttis的出版物很大的推动，他在《The C++ Standard Library》[3]里写了“带状态的判定式”，在标准库问题#92，和他的《C++ Report》文章《Predicates vs. Function Objects》[14]。在我这里，我使用了他的例子并接受他提出的一个解决方案，虽然术语“纯函数”是我发明的。Matt Austern证实了我在条款41的关于属于mem_fun和mem_fun_ref的历史的猜想。条款42可以追溯到当我考虑违反那个方针时Mark Rodgers给我的一个演讲。Mark Rodgers也对条款44的观点负责，也就是搜索map和multimap时，非成员函数检查每个pair的两个组件，而成员函数的搜索只检查第一个（键）组件。条款45包含来自多个clcm贡献者的信息，包括John Potter、Marcin Kasperski、Pete Becker、Dennis Yelle和David Abrahams。David Smallberg提醒我在进行基于等价的搜索和对有序序列容器的计数上equal_range的效用。Andrei Alexandreseu帮我了解了我在条款50提到的“到引用的引用问题”的情况，而我从Mark Rodgers提供的Boost网站[22]上类似的例子模仿出这个问题的例子。

当然，附录A的材料归功于Matt Austern。我感谢他不仅允许我在本书里包含它，而且还把它改得比原来更好。

好的技术书需要彻底的出版前检查，而很幸运我受益于一群非同寻常的天才技术评论家的见解。Brian Kernighan和Cliff Green为部分草稿提供了早期意见，而手稿的完整版本由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 Becket、Victor Von，当然还有Mark Rodgers细察。Katrina Avery做了审稿。

准备一本书最富挑战性的部分之一是找到好的技术评论家。我感谢John Putter把我介绍给Jared Manning和Aaron Campbell。

Herb Sutter好心地同意作为我在微软的Visual Studio .NET beta版上有些STL测试程序编译、运行和报告方面的行为的代理，而Leor Zolman担任测试在本书里的所有代码的极为艰巨的工作。当然，剩余的任何错误都是我的过错，并非Herb或者Leor的。

Angelika Langer让我看到了STL函数对象一些方面的不明状态。有关函数对象，本书说的比可能的要少，但它

所说的基本上保证是真的。至少我希望它是。

本书的这次印刷比早期印刷的好，因为我指出了下面这些眼尖的读者发现的问题：Jon Webb、Michael Hawkins、Derek Price、Jim Seheller、Carl Manaster、Herb Sutter、Albert Franklin、George King、Dave Miller、Harold Howe、John Fuller、Tim McCarthy、John Hershberger、Igor Mikolic-Torreira、Stephan Bergmann、Robert Allan Schwartz、John Potter、David Grigsby、Sanjay Pattni、Jesper Andersen、Jing Tao Wang、Andr é Blavier、Dan Schmidt、Bradley White、Adam Petersen、Wayne Goertel和Gabriel Netterdag。我感谢他们的在改进《Effective STL》方面所提供的帮助。

我在Addison-Wesley的合作者包括John Wait（我的编辑，现在是高级VP），Alicia Carey和Susannah Buzard（他的助手n和n+1），John Fuller（产品协调员），Karin Hansen（封面设计者），Jason Jones（都是技术领袖，特别是关于Adobe喷薄而出的有魔力的软件），Marty Rabinowitz（他们的老板，但他也工作），以及Curt Johnson，Chanda Leary-Coutu和Robin Bruce（都是销售员，但仍然非常好）。

Abbi Staley总能把星期日的午餐变成令人愉快的体验。

虽然在这之前有六本书和一张CD，我的妻子，Nancy，用她常有的自制力容忍了我研究和写作的要求，而且当我最需要的时候能给我鼓励和支持。她从未忘记提醒我生活比C++和软件重要。

然后是我们的狗，Persephone。当我写这个的时候，正是她的第六个生日。今晚，她和Nancy和我将去Baskin-Robbins吃冰淇淋。 Persephone将享受香草。一铲，在一个杯子里。走咯。

# 导读

你已经熟悉了STL。你知道怎么建立容器，迭代它们的内容，添加删除元素和应用常见算法，比如find和sort。但你并不满足，你不能摆脱STL所提供的超过它们能带来的好处的感觉。应该简单的任务并非那样。应该直截了当的操作确有资源泄漏或错误行为。应该高效的过程却需要比你希望给它们的更多的时间和内存。是的，你知道怎么使用STL，但你不确定你在*有效地*使用它。

我为你写了这本书。

在《Effective STL》中，我解释了怎样结合STL组件来在库的设计得到最大的好处。这样的信息允许你对简单、直接的问题开发简单、直接的解决方案，也帮你对更复杂的问题设计优雅的方法。我描述了常见的STL使用错误，而且向你演示怎么避开它们。那帮助你躲开闪资源漏、不可移植的代码和未定义的行为。我讨论了优化代码的方法，所以你能使STL表现得像它应该的那样快速、光滑。

本书里的信息将使你成为一个更好的STL程序员，它将让你成为一个更高产的程序员。而且它将让你成为一个更愉快的程序员，使用STL很有趣，但是有效地使用它更为有趣，这种有趣是它们必须把你拽离键盘，因为你不能相信你争拥有的好时光。即使对STL的匆匆一瞥也能发现它是一个非常酷的库，但这份酷比你可能想象的更宽更深。我在本书的一个主要目标是传达给你这个库有多神奇，因为在我编程的差不多30年里，我从未见过任何像STL的东西。你或许也没有。

## 定义、使用和扩展STL

没有“STL”的官方定义，而且当人们使用这个术语时，不同的人表示的是不同的东西。在本书中，“STL”的意思是与迭代器合作的C++标准库的一部分。那包括标准容器（包括string），iostream库的一部分，函数对象和算法。它不包括标准容器适配器（stack、queue和priority_queue）以及bitset和valarray容器，因为它们缺乏迭代器支持。它也不包括数组。真的，数组以指针的形式支持迭代器，但数组是C++语言的一部分，并非库。

技术上，我的STL的定义排除了标准C++库的扩展，特别是散列容器，单链表，rope和多种非标准函数对象。虽然如此，一个有效的STL程序员需要知道这样的扩展，因此我在合适的地方提到了它们。的确，条款25致力于非标准散列容器的概述。它们现在不在STL里，但类似它们的东西几乎肯定将要进入标准C++库的下一个版本，而在窥见未来是有价值的。

存在STL扩展的原因之一是STL是被设计为可扩展的库。不过，在本书里，我关注于使用STL，而不是给它添加新的组件。例如，如果你发现，我没有说多少关于写你自己的算法的东西，而且我根本没有在写新容器和迭代器上提供指导。我相信在你着手增加它的能力之前，掌握STL已经提供的东西很重要，所以那是我在《Effective STL》里关注的。当你决定建立你自己STLesque组件时，你将在像Josuttis的《The C++ Standard Library》[3]和Austern的《Generic Programming and the STL》[4]这样的书里找到建议。我确实在这本书里讨论的STL扩展的一个方面是写你自己的函数对象。你不可能在不知道怎么写自己的函数对象的情况下有效地使用STL，所以我为这个主题投入了整整一章（第6章）。

# 引用

先前段落中对Josuttis和Austern的书的引用演示了我怎么处理书目引用的。通常，我试图提及引用的足够信息以便已经熟悉它的人能够鉴别出来。例如，如果你已经知道这些作者的书，就不必转向参考书目那里查明[3]和[4]提及的你已经知道的书。当然，如果你不熟悉一个出版物，参考书目（从第225页开始）会给你全面的引用。

我经常引用三个一般没有使用引用编号的东西。第一个是C++国际标准[5]，我通常把它简称为“标准”。其他两个是我早先关于C++的书，《Effective C++》[1]和《More Effective C++》[2]。

# STL和标准

我经常提及C++标准，因为《Effective STL》专注于可移植的，与标准一致的C++。理论上，我在这本书里演示的一切都可以用于每个C++实现。实际上，那不是真的。编译器的缺陷和STL实现凑合成防止一些有效的代码编译或表现出它们应该有的行为。那是很常见的情况，我描述了这些问题，而且解释了你应该怎么变通地解决他们。

有时候，最容易的变通办法是使用另一个STL实现。附录B给一个这种情况的例子。实际上，STL用得越多，*编译器*和*库实现*的区别就越重要。程序员在设法让合法的代码编译时遇到困难，他们通常责备他们的编译器，但对于STL，编译器可能是好的，而STL实现是不良的。为了强调你得依赖编译器和库实现的事实，我使用你的*STL平台*。一个STL平台是一个特定编译器和一个标准模板库特定实现的组合。在本书里，如果我提及一个编译器问题，你能确信我意思是编译器有问题。但是，如果我说你的STL平台有问题，你应该理解为“可能是编译器缺陷，可能是库缺陷，或许都有”。

我一般提及你的“编译器们”——*复数*。那是我长期相信你通过确保代码可以在多于一个的编译器上工作的方法来改进你的代码质量（特别是移植性）的产物。此外，使用多个编译器一般可以简化拆解由STL的使用不当造成的错误信息难题。（条款49致力于解码此类消息的方法。）

关于与标准一致的代码，我强调的另一个方面是你应该避免构造未定义行为。这样的构造可能在运行期做任何事情。不幸的是，这意味着它们可能正好做了你想要的，而那会导致一种错误的安全感。太多程序员以为未定义行为总会导致一个明显的问题，例如，一个分段错误或其他灾难性的错误。未定义行为的结果实际上更为狡猾，例如，破坏极少引用的数据。它们也可以通过程序运行。我发现一个未定义行为的好定义是“为我工作，为你工作，在开发和QA期间工作，但在你最重要的用户面前爆炸了”。避免未定义行为很重要，所以我指出了它会出现的通常情况。你应该训练你自己警惕这样的情况。

# 引用计数

讨论STL而没有提及引用计数是几乎不可能的。正如你将在条款7和33看见的，基于指针的容器的设计几乎总要导致引用计数。另外，很多string实现内部是引用计数的，而且，正如条款15解释的，这可能是一个你不能忽视的实现细节。在本书里，我认为你已经熟悉引用的基础。如果你不是，大多数中级和高级C++教材都覆盖了这个主题。例如，在《More Effective C++》里，相关材料是在条款28和29。如果你不知道引用计数是什么而且你不喜欢学习，不要担心。你也可以读完本书，虽然这里那里可能有一些句子不像它们应该的那么有意义。

# string和wstring

我关于string说的任何东西都可以相等地应用到给它的宽字符兄弟，wstring。类似地，任何时候我提及string和char或char*之间的关系，对于wstring和wchar_t或wchar_t*之间的关序也是正确的。换句话说，只是因为我在本书里不明确地提及宽字符字符串，不要以为STL不能支持它们。它也像基于char的字符串一样支持它们。它必须。string和wstring都是同一个模板的实例化，basic_string。

# 术语，术语，术语

这不是STL的入门书，所以我认为你知道基本的东西。仍然，下面的术语十分重要，我感到需要强迫复习它们：

- vector、string、deque和list被称为*标准序列容器*。*标准关联容器*被是set、multiset、map和multimap。
- 迭代器被分成五个种类，基于它们支持的操作。简要地说，*输入迭代器*是每个迭代位置只能被读一次的只读迭代器。*输出迭代器*是每个迭代位置只能被写一次的只写迭代器。输入和输出迭代器被塑造为读和写输入和输出流（例如，文件）。因此并不奇怪输入和输出迭代器最通常的表现分别是istream_iterator和ostream_iterator。

  *前向迭代器*有输入和输出迭代器的能力，但是它们可以反复读或写一个位置。它们不支持operator--，所以它们可以高效地向前移动任意次数。所有标准STL容器都支持比前向迭代器更强大的迭代器，但是，正如你可以在条款25看到的，散列容器的一种设计可以产生前向迭代器。单链表容器（在条款50提到）也提供前向迭代器。

  *双向迭代器*就像前向迭代器，除了它们的后退可以像前进一样容易。标准关联容器都提供双向迭代器。list也有。

  *随机访问迭代器*可以做双向迭代器做的一切事情，但它们也提供“迭代器算术”，即，有一步向前或向后跳的能力。vector、string和deque都提供随机访问迭代器。指进数组的指针可以作为数组的随机访问迭代器。

- 重载了函数调用操作符（即，operator()）的任何类叫做*仿函数类*。从这样的类建立的对象称为*函数对象*或*仿函数*。STL中大部分可以使用函数对象的地方也都可以用真函数，所以我经常使用术语“函数对象”来表示C++函数和真的函数对象。
- 函数bind1st和bind2nd称为*绑定器*。

STL的一个革命性方面是它的复杂度保证。这些保证约束了任何STL操作允许表现的工作量。这极好，因为它可以帮你确定同一问题不同方法的相对效率，不论你使用的是什么STL平台。不幸的是，如果你不了解计算机科学的行话，在复杂度保证后面的专有名词可能会把你弄糊涂。这里有一个关于我在本书里使用的复杂度术语的快速入门。每个都引用了它作为*n*的函数做一件事情要多久，n是容器或区间的元素数目。

- 以*常数时间*执行的操作的性能不受*n*的改变而影响，例如，向list中插入一个元素是一个常数时间操作。不管list有一个还是一百万个元素，插入都花费同样数量的时间。

  不要太照字面理解占用“常数时间”。它不意味着做某些事情所花费时间的数量是字面上的常数，它只表明不

被n影响。例如，两个STL平台可能花费非常不同数量的时间执行相同的“常数时间”操作。如果一个库比另一个有更复杂的实现，或如果一个编译器执行了充分的更积极优化的时候，这就会发生。

常数时间复杂度的一个变体是*分摊常数时间*。以分摊常数时间运行的操作通常是常数时间的操作，但偶尔它们花的时间也取决于$n$。分摊常数时间操作*通常*以常数时间运行。

- 当$n$变大时，以*对数时间*运行的操作需要更多的时间运行，但它需要的时间以与$n$的对数成正比的比率增长。例如，在一百万个项上的一次操作预计花费大约在一百个项上三倍的时间，因为$\log n^3 = 3 \log n$。在关联容器上的大多数搜寻操作（例如，set::find）是对数时间操作。
- 以线性时间运行的操作需要的时间以与$n$成正比的比率增长。标准算法count以线性时间运行，因为它必须查看给定区间中的每个元素。如果区间的大小扩大了三倍，它也必须做三倍的工作，而且我们期望它大约花费三倍时间来完成。

通常，常数时间操作运行得比要求对数时间的快，而对数时间操作运行得比线性的快。当$n$变得足够大时，这总是真的，但对于$n$相对小的值，有时候更差理论复杂度的操作可能或胜过更好理论复杂度的操作。如果你想对知道更多STL复杂度保证的东西，转向Josuttis的《The C++ Standard Library》[3]。

术语的最后一个要注意的东西是，记住map或multimap里的每个元素都有两个组件。我一般叫第一个组件键，叫第二个组件值。以

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

为例，string是键，double是值。

# 代码例子

这本书充满了例子代码，当我引入每个例子时我都作了解释。不过仍然值得提前知道一些事情。

你可以从上面map的例子看到我通常忽略#include，而且忽视STL组件在namespace std里的事实。当定义map m，我可以这么写，

```
#include <map>
#include <string>

using std::map;
using std::string;

map<string, double> m;
```

但我喜欢让我们省掉这些噪音。

导读

当我为一个模板声明一个形式类型参数时，我使用typename而不是class。即，不这么写，

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

我这么写：

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

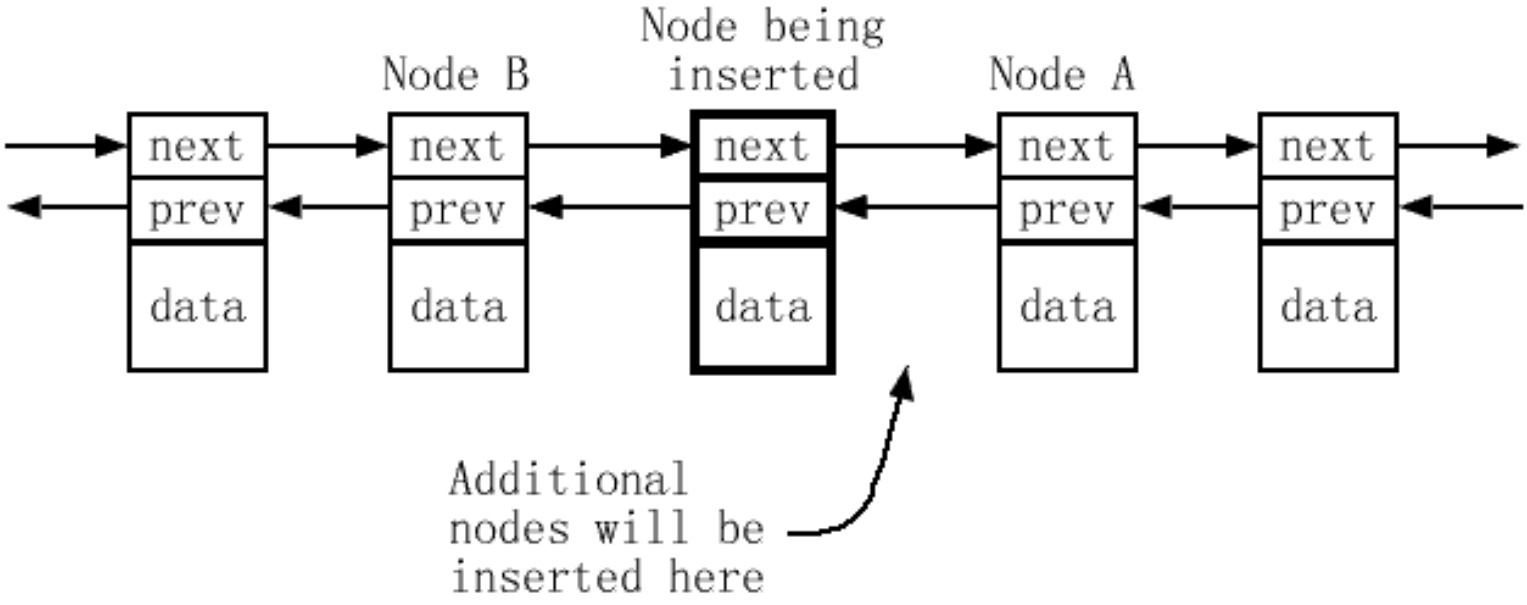
用这个场景里，class和typename表示完全相同的东西，但我发现typename能更清楚地表示我通常想要说的：T可以是*任何*类型；不必是一个类。如果你喜欢使用class来声明类型参数，那也可以。在这个场景里是用typename或class完全是风格的问题。

在另一个场景里，这不再是风格问题。为了避免潜在的解析含糊（我将提供给你细节），你被要求在依赖形式类型参数的类型名字之前使用typename。这样的类型被称为*依赖类型*，一个例子将帮助阐明我所说的。假设你想为函数写一个模板，给定一个STL容器，返回容器中的最后一个元素是否大于第一个元素。这是一种方法：

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

在这个例子里，局部变量begin和end的类型是C::const_iterator。const_iterator是依赖形式类型参数C的一种类型。因为C::const_iterator是一种依赖类型，你被要求在它之前放上typename这个词。（一些编译器错误地接受没有typename的代码，但这样的代码不可移植。）

我希望你注意到了在上面例子里我对颜色的使用。那是为了让你的注意力集中于特别重要的部分代码。通常，我加亮相关例子之间的差别，正如我在Widget例子里演示两种可能的声明参数T的方法。用于唤起例子中特别值得注意的部分的颜色也延伸到图表。例如，来自条款5的这张图使用颜色来区分当新元素被插入list时受影响的两个指针：




我也为章号使用颜色，但这样的使用完全没有理由。这作为我的第一本两色的书，我希望你能原谅我的一点色彩丰富。

我最喜爱的参数名中的有两个是lhs和rhs。它们分别代表“左手边”和“右手边”，而且当声明操作符时，我发现它们特别有用。这是来自条款19的一个例子：

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

当这个函数在这样的场景下被调用时，

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

x，在“==”的左边，在operator==里面被称为lhs，而y被称为rhs。

至于类名Widget，与GUI或者工具无关。这指示我为“做某件事的某个类”使用的名字。有时，正如第7页上的，Widget是一个类模板而不是一个类。在这样的情况里，你可能发现我仍然称Widget为一个类，即使这真的是一个模板。只要对讨论的东西不会产生歧义，忽略类和类模板、结构体和结构体模板、函数和函数模板之间的差别就不会伤害任何人。在可能混淆的情况下，我会分清模板和它们产生的类、结构体和函数。

# 涉及效率的条款

我考虑过在《Effective STL》中包含一章关于效率的，但我最后决定当前的组织是更好的。仍然，许多项目关注与减少空间和运行期需要。为了方便你的性能提高，这里有一张包含实际上关于效率的章节的表：

# 《Effective STL》的方针

构成本书50个条款的方针是基于世界上最有经验的STL程序员的见解和建议。这些方针总结了你应该总是做的——或总是避免做的——以发挥标准模板库的最大功效。同时，它们只是方针。在一些条件下，违反它们是有意义的。例如，条款7的标题告诉你在容器销毁前应该删除容器内new得的指针，但那个条款的正文说明只适用于当容器销毁时被那些指针指向的对象也得跟随而去的情况。情况经常如此，但不永远为真。类似的，条款35的标题恳求你使用STL算法进行简单的忽略大小写字符串比较，但条款的正文指出在一些情况里，你最好使用不仅在STL之外，而且甚至不是标准C++一部分的一个函数！

只有你足够了解你正写的软件，它运行的环境，以及建立它的场景，才能确定违反我提出的方针是否合理。多数时候，不是，而且伴随每个条款的讨论解释了为什么。在一些情况里，它是。对指南奴隶般的盲从是不对的，但骑士般的漠视也不对。在一个人冒险之前，你应该保证你有一个好原因。

# 容器

没错，STL有迭代器、算法和函数对象，但对于大多数C++程序员，容器是最突出的。它们比数组更强大更灵活，可以动态增长（也常是缩减），可以管理属于它们自己的内存，可以跟踪它们拥有的对象数目，可以限制它们支持操作的算法复杂度等等。它们的普及是很容易理解的。它们比竞争对手更好，不管竞争对手是来自其他库或你自己写的容器类型。STL容器不只是好，而是*相当好*。

本章关注的是可以适用于所有STL容器的指导方针。后面的章节则专注于特殊的容器类型。把这个专题放在这里还因为让你知道在选择适当的容器时应该面对的约束；避免产生为一个容器类型写的代码也可以用于其它容器类型的错觉；容器里对象拷贝操作的重要性；当指针或auto_ptr存放在容器中时出现的难点；删除的输入和输出；你可以或不可以使用自定义分配器；达到最高效率的技巧和考虑在多线程环境下容器的使用。

有很多需要覆盖的领域，但没关系。这些条款会把它分解为小块，按照这种方式，你差不多*现在*就能发现几个可以应用于你的代码的主意。

# 条款1：仔细选择你的容器

你知道C++中有很多你可以支配的容器，但是你意识到有多少吗？要确定你没有忽略你的选项，这里有一个快速回顾。

- 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本质上是一个重型字符串。（“绳子（rope）”是重型的“线（string）”。明白了吗？）你可以找到一个关于这些非标准（但常见的）容器的概览在条款50。
- 非标准关联容器hash_set、hash_multiset、hash_map和hash_multimap。我在条款25检验了这些可以广泛获得的基于散列表的容器和标准关联容器的不同点。
- vector<char>可以作为string的替代品。条款13描述了这个替代品可能会有意义的情况。
- vector作为标准关联容器的替代品。就像条款23所说的，有时候vector可以在时间和空间上都表现得比标准关联容器好。
- 几种标准非STL容器，包括数组、bitset、valarray、stack、queue和priority_queue。因为它们是非STL容器，所以在本书中关于它们我说得很少，虽然条款16提到了数组比STL容器更有优势的一种情况，而条款18揭示了为什么bitset可能比vector<bool>要好。值得注意的是，数组可以和STL算法配合，因为指针可以当作数组的迭代器使用。

这是所有的选项，而且可以考虑的范围和可以在它们之间的选择一样丰富。不走运的是，STL的大多数讨论只限于容器世界的一个很窄的视野，忽略了很多关于选择适当容器的问题。就连标准都介入了这个行动，提供了以下的在vector、deque和list之间作选择的指导方案：

> vector、list和deque提供给程序员不同的复杂度，因此应该这么用：vector是一种可以默认使用的序列类型，当很频繁地对序列中部进行插入和删除时应该用list，当大部分插入和删除发生在序列的头或尾时可以选择deque这种数据结构。

如果你主要关心的是算法复杂度，我想这个方案是有理由的建议，但需要关心更多东西。

现在，我们要检查一些可以补充算法复杂度的重要的容器相关问题，但首先我需要介绍一种STL容器的分类方法，它被讨论的次数并不像它应该的那样多。那是连续内存容器和基于节点的容器的区别。

*连续内存容器*（也叫做*基于数组的容器*）在一个或多个（动态分配）的内存块中保存它们的元素。如果一个

新元素被查入或者已存元素被删除，其他在同一个内存块的元素就必须向上或者向下移动来为新元素提供空间或者填充原来被删除的元素所占的空间。这种移动影响了效率（参见条款5和14）和异常安全（就像我们将会看到的）。标准的连续内存容器是vector、string和deque。非标准的rope也是连续内存容器。

*基于节点的容器*在每个内存块（动态分配）中只保存一个元素。容器元素的插入或删除只影响指向节点的指针，而不是节点自己的内容。所以当有东西插入或删除时，元素值不需要移动。表现为链表的容器——比如list和slist——是基于节点的，所有的标准关联容器也是（它们的典型实现是平衡树）。非标准的散列容器使用不同的基于节点的实现，就像我们将会在条款25中看到的。

利用这个不恰当的术语，我们已经准备好描述一些大多数关于在容器间选择的问题。在这个讨论中，我略过考虑非STL类容器（比如，数组、bitset等），因为毕竟这是本关于STL的书。

- *你需要“可以在容器的任意位置插入一个新元素”的能力吗？*如果是，你需要序列容器，关联容器做不到。
- *你关心元素在容器中的顺序吗？*如果不，散列容器就是可行的选择。否则，你要避免使用散列容器。
- *必须使用标准C++中的容器吗？*如果是，就可以除去散列容器、slist和rope。
- *你需要哪一类迭代器？*如果必须是随机访问迭代器，在技术上你就只能限于vector、deque和string，但你也可能会考虑rope（关于rope的更多信息在条款50）。如果需要双向迭代器，你就用不了slist（参见条款50）和散列容器的一般实现（参见条款25）。
- *当插入或者删除数据时，是否非常在意容器内现有元素的移动？*如果是，你就必须放弃连续内存容器（参见条款5）。
- *容器中的数据的内存布局需要兼容C吗？*如果是，你就只能用vector（参见条款16）。
- *查找速度很重要吗？*如果是，你就应该看看散列容器（参见条款25），排序的vector（参见条款23）和标准的关联容器——大概是这个顺序。
- *你介意如果容器的底层使用了引用计数吗？*如果是，你就得避开string，因为很多string的实现是用引用计数（参见条款13）。你也不能用rope，因为权威的rope实现是基于引用计数的（参见条款50）。于是你得重新审核你的string，你可以考虑使用vector<char>。
- *你需要插入和删除的事务性语义吗？*也就是说，你需要有可靠地回退插入和删除的能力吗？如果是，你就需要使用基于节点的容器。如果你需要多元素插入（比如，以范围的方式——参见条款5）的事务性语义，你就应该选择list，因为list是唯一提供多元素插入事务性语义的标准容器。事务性语义对于有兴趣写异常安全代码的程序员来说非常重要。（事务性语义也可以在连续内存容器上实现，但会有一个性能开销，而且代码不那么直观。要了解这方面的知识，请参考Sutter的《Exceptional C++》的条款17[8]。）
- *你要把迭代器、指针和引用的失效次数减到最少吗？*如果是，你就应该使用基于节点的容器，因为在这些容器上进行插入和删除不会使迭代器、指针和引用失效（除非它们指向你删除的元素）。一般来说，在连续内存容器上插入和删除会使所有指向容器的迭代器、指针和引用失效。
- *你需要具有有以下特性的序列容器吗：1）可以使用随机访问迭代器；2）只要没有删除而且插入只发*

> *生在容器结尾，指针和引用的数据就不会失效？*这个一个非常特殊的情况，但如果你遇到这种情况，deque就是你梦想的容器。（有趣的是，当插入只在容器结尾时，deque的迭代器也可能会失效，deque是唯一一个“在迭代器失效时不会使它的指针和引用失效”的标准STL容器。）

这些问题几乎不是事情的完结。比如，它们没有关注不同的容器类型使用不同的内存配置策略（条款10和14讨论了这些策略的一些方面）。但是，它们已经足够是你信服了，除非你对元素顺序、标准的一致性、迭代器能力、内存布局和C的兼容性、查找速度、因为引用计数造成的行为不规则、事务性语义的轻松实现和迭代器失效的条件没兴趣，你得在容器操作的算法复杂度上花更多的考虑时间。当然这样的复杂度是重要的，但这离整个故事很远。

当面对容器时，STL给了你很多选项。如果你的视线超越了STL的范围，那就会有更多的选项。在选择一个容器前，要保证考虑了所有你的选项。一个“默认容器”？我不这么认为。

# 条款2：小心对“ 容器无关代码 ”的幻想

STL是建立在泛化之上的。数组泛化为容器，参数化了所包含的对象的类型。函数泛化为算法，参数化了所用的迭代器的类型。指针泛化为迭代器，参数化了所指向的对象的类型。

这只是个开始。独立的容器类型泛化为序列或关联容器，而且类似的容器拥有类似的功能。标准的内存相邻容器（参见条款1）都提供随机访问迭代器，标准的基于节点的容器（再参见条款1）都提供双向迭代器。序列容器支持push_front或push_back，但关联容器不支持。关联容器提供对数时间复杂度的lower_bound、upper_bound和equal_range成员函数，但序列容器却没有。

随着泛化的继续，你会自然而然地想加入这个运动。这种做法值得赞扬，而且当你写你自己的容器、迭代器和算法时，你会自然而然地推行它。唉，很多程序员试图把它推行到不同的样式。他们试图在他们的软件中泛化容器的不同，而不是针对容器的特殊性编程，以至于他们可以用，可以说，现在是一个vector，但以后仍然可以用比如deque或者list等东西来代替——都可以在不用改变代码的情况下来使用。也就是说，他们努力去写“ 容器无关代码 ”。这种可能是出于善意的泛化，却几乎总会造成麻烦。

最热心的“ 容器无关代码 ”的鼓吹者很快发现，写既要和序列容器又要和关联容器一起工作的代码并没有什么意义。很多成员函数只存在于其中一类容器中，比如，只有序列容器支持push_front或push_back，只有关联容器支持count和lower_bound，等等。在不同种类中，甚至连一些如insert和erase这样简单的操作在名称和语义上也是天差地别的。举个例子，当你把一个对象插入一个序列容器中，它保留在你放置的位置。但如果你把一个对象插入到一个关联容器中，容器会按照的排列顺序把这个对象移到它应该在的位置。举另一个例子，在一个序列容器上用一个迭代器作为参数调用erase，会返回一个新迭代器，但在关联容器上什么都不返回。（条款9给了一个例子来演示这点对你所写的代码的影响。）

假设，然后，你希望写一段可以用在所有常用的序列容器上——vector, deque和list——的代码。很显然，你必须使用它们能力的交集来编写，这意味着不能使用reserve或capacity（参见条款14），因为deque和list不支持它们。由于list的存在意味着你得放弃operator[]，而且你必须受限于双向迭代器的性能。这意味着你不能使用需要随机访问迭代器的算法，包括sort，stable_sort，partial_sort和nth_element（参见条款31）。

另一方面，你渴望支持vector的规则，不使用push_front和pop_front，而且用vector和deque都会使splice和成员函数方式的sort失败。在上面约束的联合下，后者意味着你不能在你的“ 泛化的序列容器 ”上调用任何一种sort。

这是显而易见的。如果你冒犯里其中任何一条限制，你的代码会在至少一个你想要使用的容器配合时发生编

译错误。可见这种代码有多阴险。

这里的罪魁祸首是不同的序列容器所对应的不同的迭代器、指针和引用的失效规则。要写能正确地和vector, deque和list配合的代码，你必须假设任何使那些容器的迭代器，指针或引用失效的操作符真的在你用的容器上起作用了。因此，你必须假设每次调用insert都使所有东西失效了，因为deque::insert会使所有迭代器失效，而且因为缺少capacity，vector::insert也必须假设使所有指针和引用失效。（条款1解释了deque是唯一一个在迭代器失效的情况下指针和引用仍然有效的东西）类似的理由可以推出一个结论，所有对erase的调用必须假设使所有东西失效。

想要知道更多？你不能把容器里的数据传递给C风格的界面，因为只有vector支持这么做（参见条款16）。你不能用bool作为保存的对象来实例化你的容器，因为——正如条款18所阐述的——vector 并非总表现为一个vector，实际上它并没有真正保存bool值。你不能期望享受到list的常数时间复杂度的插入和删除，因为vector和deque的插入和删除操作是线性时间复杂度的。

当这些都说到做到了，你只剩下一个“泛化的序列容器”，你不能调用reserve、capacity、operator[]、push_front、pop_front、splice或任何需要随机访问迭代器的算法；调用insert和erase会有线性时间复杂度而且会使所有迭代器、指针和引用失效；而且不能兼容C风格的界面，不能存储bool。难道这真的是你想要在你的程序里用的那种容器？我想不是吧。

如果你控制住了你的野心，决定愿意放弃对list的支持，你仍然放弃了reserve、capacity、push_front和pop_front；你仍然必须假设所有对insert和erase的调用有线性时间复杂度而且会使所有东西失效；你仍然不能兼容C风格的布局；而且你仍然不能储存bool。

如果你放弃了序列容器，把代码改为只能和不同的关联容器配合，这情况并没有什么改善。要同时兼容set和map几乎是不可能的，因为set保存单个对象，而map保存对象对。甚至要同时兼容set和multiset（或map和multimap）也是很难的。set/map的insert成员函数只返回一个值，和他们的multi兄弟的返回类型不同，而且你必须避免对一个保存在容器中的值的拷贝份数作出任何假设。对于map和multimap，你必须避免使用operator[]，因为这个成员函数只存在于map中。

面对事实吧：这根本没有必要。不同的容器是不同的，而且它们的优点和缺点有重大不同。它们并不被设计成可互换的，而且你做不了什么包装的工作。如果你想试试看，你只不过是在考验命运，但命运并不想被考验。

接着，当天黑以后你认识到你决定使用的容器，嗯，不是最理想的，而且你需要使用一个不同的容器类型。你现在知道当你改变容器类型的时候，不光要修正编译器诊断出来的问题，而且要检查所有使用容器的代码，根据新容器的性能特征和迭代器，指针和引用的失效规则来看看那些需要修改。如果你从vector切换到其他东西，你也需要确认你不再依靠vector的C兼容的内存布局；如果你是切换到一个vector，你需要保证你不用它来保存bool。

既然有了要一次次的改变容器类型的必然性，你可以用这个常用的方法让改变得以简化：使用封装，封装，再封装。其中一种最简单的方法是通过自由地对容器和迭代器类型使用typedef。因此，不要这么写:

```
class Widget {...};
vector<Widget> vw;
Widget bestWidget;
...                          // 给bestWidget一个值
vector<Widget>::iterator i =         // 寻找和bestWidget相等的Widget
    find(vw.begin(), vw.end(), bestWidget);
```

要这么写：

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

这是改变容器类型变得容易得多，如果问题的改变是简单的加上用户的allocator时特别方便。（一个不影响对迭代器/指针/参考的失效规则的改变）

```
class Widget { ... };
template<typename T>                        // 关于为什么这里需要一个template
SpecialAllocator { ... };                   // 请参见条款10
typedef vector<Widget, SpecialAllocator<Widget> > WidgetContainer;
typedef WidgetContainer::iterator WCIterator;
WidgetContainer cw;                         // 仍然能用
Widget bestWidget;
...
WCIterator i = find(cw.begin(), cw.end(), bestWidget);  // 仍然能用
```

如果typedef带来的代码封装作用对你来说没有任何意义的话，你仍然会称赞它们可以节省许多工作。比如，你有一个如下类型的对象

```
map<string,
    vectorWidget>::iterator,
    CIStringCompare>          // CIStringCompare是“ 忽略大小写的字符串比较 ”
                    // 参见条款19
```

而且你要用const_iterator遍历这个map，你真的想不止一次地写下

```
map<string, vectorWidget>::iterator, CIStringCompare>::const_iterator
```

？当你使用STL一段时间以后，你会认识到typedef是你的好朋友。

typedef只是其它类型的同义字，所以它提供的的封装是纯的词法（译注：不像#define是在预编译阶段替换的）。typedef并不能阻止用户使用（或依赖）任何他们不应该用的（或依赖的）。如果你不想暴露出用户对你所决定使用的容器的类型，你需要更大的火力，那就是class。

要限制如果用一个容器类型替换了另一个容器可能需要修改的代码，就需要在类中隐藏那个容器，而且要通过类的接口限制容器特殊信息可见性的数量。比如，如果你需要建立一个客户列表，请不要直接用list。取而代之的是，建立一个CustomerList类，把list隐藏在它的private区域：

```
class CustomerList {
private:
    typedef list<Customer> CustomerContainer;
    typedef CustomerContainer::iterator CCIterator;
    CustomerContainer customers;
public:             // 通过这个接口
    ...              // 限制list特殊信息的可见性
};
```

一开始，这样做可能有些无聊。毕竟一个customer list是一个list，对吗？哦，可能是。稍后你可能发现从列表的中部插入和删除客户并不像你想象的那么频繁，但你真的需要快速确定客户列表顶部的20%——一个为nth_element算法量身定做的任务（参见条款31）。但nth_element需要随机访问迭代器，不能兼容list。在这种情况下，你的客户“ list ”可能更应该用vector或deque来实现。

当你决定作这种更改的时候，你仍然必须检查每个CustomerList的成员函数和每个友元，看看他们受影响的程度（根据性能和迭代器/指针/引用失效的情况等等），但如果你做好了对CustomerList地实现细节做好封装的话，那对CustomerList的客户的影响将会很小。你写不出容器无关性代码，但他们可能可以。

# 条款3：使容器里对象的拷贝操作轻量而正确

容器容纳了对象，但不是你给它们的那个对象。此外，当你从容器中获取一个对象时，你所得到的对象不是容器里的那个对象。取而代之的是，当你向容器中添加一个对象（比如通过insert或push_back等），进入容器的是你指定的对象的*拷贝*。拷进去，拷出来。这就是STL的方式。

一旦一个对象进入一个容器，以后对它的拷贝并不少见。如果你从vector、string或deque中插入或删除了什么，现有的容器元素会移动（拷贝）（参见条款5和14）。如果你使用了任何排序算法（参见条款31）：next_permutation或者previous_permutation；remove、unique或它们的同类（参见条款32）；rotate或reverse等，对象会移动（拷贝）。是的，拷贝对象是STL的方式。

你可能会对所有这些拷贝是怎么完成的感兴趣。这很简单，一个对象通过使用它的拷贝成员函数来拷贝，特别是它的拷贝构造函数和它的拷贝赋值操作符（很好的名字，不是吗？）。对于用户自定义类，比如Widget，这些函数传统上是这么声明的：

```
class Widget {
public:
    ...
    Widget(const Widget&);        // 拷贝构造函数
    Widget& operator=(const Widget&);     // 拷贝赋值操作符
    ...
};
```

如果你自己没有声明这些函数，你的编译器始终会为你声明它们。拷贝内建类型（比如int、指针等）也始终是通过简单地拷贝他们的内在比特来完成的。（有关拷贝构造函数和赋值操作符的详细情况，请参考任何C++的介绍性书籍。在《Effective C++》中，条款11和27专注于这些函数的行为。）

因为会发生所有这些拷贝，本条款的动机现在就很清楚了。如果你用一个拷贝过程很昂贵对象填充一个容器，那么一个简单的操作——把对象放进容器也会被证明为是一个性能瓶颈。容器中移动越多的东西，你就会在拷贝上浪费越多的内存和时钟周期。此外，如果你有一个非传统意义的“拷贝”的对象，把这样的对象放进容器总是会导致不幸。（会导致的不幸之一的例子请看条款8。）

当然由于继承的存在，拷贝会导致分割。那就是说，如果你以基类对象建立一个容器，而你试图插入派生类

对象，那么当对象（通过基类的拷贝构造函数）拷入容器的时候对象的派生部分会被删除：

```
vector<Widget> vw;
class SpecialWidget:   // SpecialWidget从上面的Widget派生
public Widget {...};
SpecialWidget sw;
vw.push_back(sw);          // sw被当作基类对象拷入vw
                  // 当拷贝时它的特殊部分丢失了
```

分割问题暗示了把一个派生类对象插入基类对象的容器几乎总是错的。如果你希望结果对象表现为派生类对象，比如，调用派生类的虚函数等，总是错的。（关于分割问题更多的背景知识，请参考《Effective C++》条款22。它会在STL中发生的另一个例子，参见条款38。）

一个使拷贝更高效、正确而且对分割问题免疫的简单的方式是建立指针的容器而不是对象的容器。也就是说，不是建立一个Widget的容器，建立一个Widget*的容器。拷贝指针很快，它总是严密地做你希望的（指针拷贝比特），而且当指针拷贝时没有分割。不幸的是，指针的容器有它们自己STL相关的头疼问题。你可以从条款7和33了解它们。如果你想避免这些头疼并且仍要避开效率、正确性和分割问题，你可能会发现智能指针的容器是一个吸引人的选择，请转到条款7。

如果所有这些使STL的拷贝机制听起来很疯狂，就请重新想想。是，STL进行了大量拷贝，但它通常设计为避免不必要的对象拷贝，实际上，它也被实现为避免不必要的对象拷贝。和C和C++内建容器的行为做个对比，下面的数组：

```
Widget w[maxNumWidgets];          // 建立一个大小为maxNumWidgets的Widgets数组
                  // 默认构造每个元素
```

即使我们一般只使用其中的一些或者我们立刻使用从某个地方获取（比如，一个文件）的值覆盖每个默认构造的值，这也得构造maxNumWidgets个Widget对象。使用STL来代替数组，你可以使用一个可以在需要的时候增长的vector：

```
vector<Widget> vw;                // 建立一个0个Widget对象的vector
                  // 需要的时候可以扩展
```

我们也可以建立一个可以足够包含maxNumWidgets个Widget的空vector，但没有构造Widget：

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

```
vw.reserve(maxNumWidgets);          // reserve的详细信息请参见条款14
```

和数组对比，STL容器更文明。它们只建立（通过拷贝）你需要的个数的对象，而且它们只在你指定的时候做。是的，我们需要知道STL容器使用了拷贝，但是别忘了一个事实：比起数组它们仍然是一个进步。

# 条款4：用empty来代替检查size()是否为0

对于任意容器c，写下

```
if (c.size() == 0)...
```

本质上等价于写下

```
if (c.empty())...
```

这就是例子。你可能会奇怪为什么一个构造会比另一个好，特别是事实上empty的典型实现是一个返回size是否返回0的内联函数。

你应该首选empty的构造，而且理由很简单：对于所有的标准容器，empty是一个常数时间的操作，但对于一些list实现，size花费线性时间。

但是什么造成list这么麻烦？为什么不能也提供一个常数时间的size？答案是对于list特有的splice有很多要处理的东西。考虑这段代码：

```
list<int> list1;
list<int> list2;
...
list1.splice(                          // 把list2中
      list1.end(), list2,                 // 从第一次出现5到
      find(list2.begin(), list2.end(), 5),        // 最后一次出现10
      find(list2.rbegin(), list2.rend(), 10).base()   // 的所有节点移到list1的结尾。
);                                     // 关于调用的
                                       // "base()"的信息，请参见条款28
```

除非list2在5的后面有一个10，否则这段代码无法工作，但是咱们假设这不是问题。取而代之的是，咱们关注于这个问题：接合后list1有多少元素？很明显，接合后list1的元素个数等于接合之前list1的元素个数加上接合进去的元素个数。但是有多少元素接合进去了？那等于由find(list2.begin(), list2.end(), 5)和find(list2.rbegin(),

list2.rend(), 10).base()所定义的区间的元素个数。OK，那有多少？在没有遍历这个区间并计数之前无法知道。那就是问题所在。

假设你现在要负责实现list。list不只是一个普通的容器，它是一个*标准*容器，所以你知道你的类会被广泛使用。你自然希望你的实现越高效越好。你指出客户常常会想知道list中有多少元素，所以你把size()做成常数时间的操作。因此你要把list设计为它总是知道包含有多少元素。

与此同时，你知道对于所有标准容器，只有list提供了不用拷贝数据就能把元素从一个地方接合到另一个地方的能力。你的推论是,很多list用户会特别选择list，因为它提供了高效的接合。他们知道从一个list接合一个区域到另一个list可以在常数时间内完成，而你知道他们了解这点，所以你确定需要符合他们的期望，那就是splice是一个常数时间的成员函数。

这让你进退两难。如果size是一个常数时间操作，当操作时每个list成员函数必须更新list的大小。也包括了splice。但让区间版本的splice更新它所更改的list大小的唯一的方法是算出接合进来的元素的个数，但那么做就会使它不可能有你所希望的常数时间的性能。如果你去掉了splice的区间形式要更新它所修改的list的大小的需求，splice就可以是常数时间，但size就变成线性时间的操作。一般来说，它必须遍历它的整个数据结构来才知道它包含多少元素。不管你如何看待它，有的东西——size或splice的区间形式——必须让步。一个或者另一个可以是常数时间操作，但不能都是。

不同的list实现用不同的方式解决这个矛盾，依赖于他们的作者选择的是让size或splice的区间形式达到最高效率。如果你碰巧使用了一个常数时间的splice的区间形式比常数时间的size优先级更高的list实现，调用empty比调用size更好，因为empty总是常数时间操作。即使你现在用的不是这样的实现，你可能发现自己会在未来会使用一个这样实现。比如，你可能把你的代码移植到一个使用不同的STL实现的不同的平台，你也可能只是决定在你现在的平台上切换到一个不同的STL实现。

不管发生了什么，如果你用empty来代替检查是否size() == 0，你都不会出错。所以在想知道容器是否包含0个元素的时候都应该调用empty。

# 条款5：尽量使用区间成员函数代替它们的单元素兄弟

快！给定两个vector，v1和v2，使v1的内容和v2的后半部分一样的最简单方式是什么？不要为“当v2有偶数个元素时才有一半”而烦恼，只要做一些合理的东西。

时间到！如果你的答案是

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

或者其他很相似的东西，你就答对了，可以获得金质奖章。如果你的答案涉及到多于一条语句，但没有使用任何形式的循环，你接近了正确答案，但没有金质奖章。如果你的答案涉及到一个循环，你就需要花些时间来改进。如果你的答案涉及到多个循环，那好，我们只能说你真的需要这本书。

顺便说说，如果你对这个问题的答案的回答包含了“嗯？”，请高度注意，因为你将会学到一些真的有用的东西。

这个测验设计为做两件事。第一，它提供给我一个机会来提醒你assign成员函数的存在，太多的程序员没注意到这是一个很方便的方法。它对于所有标准序列容器（vector，string，deque和list）都有效。无论何时你必须完全代替一个容器的内容，你就应该想到赋值。如果你只是拷贝一个容器到另一个同类型的容器，operator=就是选择的赋值函数，但对于示范的那个例子，当你想要给一个容器完全的新数据集时，assign就可以利用，但operator=做不了。

这个测验的第二个理由是演示为什么区间成员函数优先于它们的单元素替代品。*区间成员函数*是一个像STL算法的成员函数，使用两个迭代器参数来指定元素的一个区间来进行某个操作。不用区间成员函数来解决这个条款开头的问题，你就必须写一个显式循环，可能就像这样：

```
vector<Widget> v1, v2;        // 假设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);
```

条款43详细验证了为什么你应该尽量避免手写显式循环，但你不必读那个条款就能知道写这段代码比写assign的调用要做多得多的工作。就像我们将马上会看到的，循环也会造成一个效率的损失，但我们一会儿会处理的。

避免循环的一个方法是按照条款43的建议，使用一个算法来代替：

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

写这些仍然比写assign的调用要做更多的工作。此外，虽然在这段代码中没有表现出循环，在copy中的确存在一个循环(参见条款43）。结果，效率损失仍然存在。我也会在下面讨论。在这里，我要离题一下来指出几乎所有目标区间是通过插入迭代器（比如，通过inserter，back_inserter或front_inserter）指定的copy的使用都可以——*应该*——通过调用区间成员函数来代替。比如这里，这个copy的调用可以用一个insert的区间版本代替：

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

这个输入量稍微比调用copy少，但它发生的也比说的要直接：数据插入v1。调用copy也表达了那个意思，但没那么直接。这把重点放到了错误的地方。对于发生了什么的关注点不应该是元素被拷贝，而是有新的数据加入v1。insert成员函数使这变得清晰了。copy的使用把它变得晦涩。关于东西被拷贝这个事实并没有什么好关注的，因为STL构建在东西会被拷贝的假定上。拷贝对STL来说很基本，它是本书条款3的主题！

太多的STL程序员过度使用copy，所以我重复一遍我的建议：几乎所有目标区间被插入迭代器指定的copy的使用都可以用调用的区间成员函数的来代替。

返回我们的assign的例子，我们已经确定两个尽量使用区间成员函数代替它们的单元素兄弟的理由。

- 一般来说使用区间成员函数可以输入更少的代码。
- 区间成员函数会导致代码更清晰更直接了当。

简而言之，区间成员函数所产生的代码更容易写，更容易理解。不是这样吗？

唉，有些人会把这个论点看作编程风格的问题，而开发者喜欢争论风格问题几乎和喜欢争论什么是真正的编辑器一样。（虽然有很多疑问，但它的确是Emacs。）如果有一个确定区间成员函数优先于它们的单元素兄弟的更普遍赞成的标准将会很有好处。对于标准序列容器，我们有一个：效率。当处理标准序列容器时，应用单元素成员函数比完成同样目的的区间成员函数需要更多地内存分配，更频繁地拷贝对象，而且/或者造成多余操作。

比如，假设你要把一个int数组拷贝到vector前端。（数据可能首先存放在一个数组而不是vector中，因为数据来自于遗留的C API。对于混合使用STL容器和C API的问题的讨论，参见条款16。）使用vector区间insert函数，真的微不足道：

```
int data[numValues];              // 假设numValues在
                                  // 其他地方定义

vector<int> v;
...
v.insert(v.begin(), data, data + numValues);   // 把data中的int
                                  // 插入v前部
```

在一个显式循环中使用迭代调用来插入，它可能看起来多多少少像这样：

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

注意我们必须小心保存insert返回值以用于下次循环迭代。如果我们在每次插入后没有更新insertLoc，我们就会有两个问题。首先，所有第一次以后的循环迭代会导致未定义行为，因为每次调用insert会使insertLoc无效。第二，即使insertLoc保持有效，我们总是在vector的前部插入（也就是，在v.begin()），这样的结果就是整数以反序拷贝到v中。

如果我们按照条款43的指引，用调用copy来代替循环，我们会得出像这样的东西：

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

这次演示了copy模板，这段代码基于copy，这和使用显式循环的代码几乎一样，所以处于效率分析的目的，我们会关注于显示循环，要牢记分析也是一样有效于使用copy的代码。着眼于显式循环可以更容易地了解效率冲击都在哪里。是的，“冲击”是复数，因为使用insert单元素版本的代码对你征收了三种不同的性能税，而如果你用区间版本的insert，则一种都没有。

第一种税在于没有必要的函数调用。把numValues个元素插入v，每次一个，自然会花费你numValues次调用insert。使用insert的区间形式，你只要花费一次调用，节省了numValues-1次调用。当然，可能的内联会使你节省这种税，但再次说明，它也可能不会。只有一件事情是确定的，使用insert的区间形式，你明确地不必为此花费。

内联也节省不了你的第二种税——无效率地把v中的现有元素移动到它们最终插入后的位置的开销。每次调用insert来增加一个新元素到v，插入点以上的每个元素都必须向上移动一次来为新元素腾出空间。所以在位置p的元素必须向上移动到位置p+1等。在我们的例子中，我们在v的前部插入了numValues个元素。那意味着在v中每个插入之前的元素都得向上移动一共numValues个位置。但每次insert调用时每个只能向上移动一个位置，所以每个元素一共会被移动numValues次。如果v在插入前有n个元素，则一共会发生n*numValues次移动。在这个例子里，v容纳int，所以每次移动可能会归结为一次memmove调用，但如果v容纳了用户自定义类型比如Widget，每次移动会导致调用那个类型的赋值操作符或者拷贝构造函数。（大部分是调用赋值操作符，但每次vector的最后一个元素被移动，那个移动会通过调用元素的拷贝构造函数来完成。）于是在一般情况下，把numValues个新对象每次一个地插入容纳了n个元素的vector<Widget>的前部需要花费n*numValues次函数调用：(n-1)*numValues调用Widget赋值操作符和numValues调用Widget拷贝构造函数。即使这些调用内联了，你仍然做了移动numValues次v中的元素的工作。

相反的是，标准要求区间insert函数直接把现有元素移动到它们最后的位置，也就是，开销是每个元素一次移动。总共开销是n次移动，numValues次容器中的对象类型的拷贝构造函数，剩下的是类型的赋值操作符。相比单元素插入策略，区间insert少执行了n*(numValues-1)次移动。花一分钟想想。这意味着如果numValues是100，insert的区间形式会比重复调用insert的单元素形式的代码少花费99%的移动！

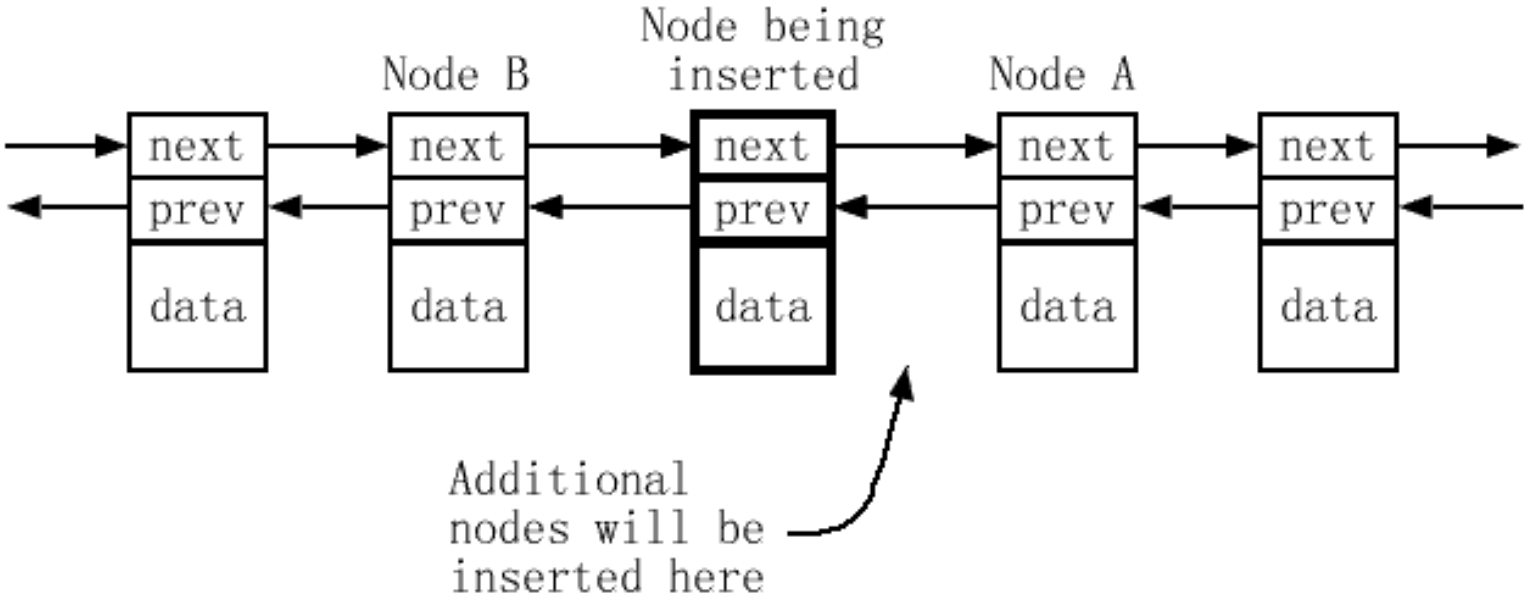
在我转向单元素成员函数和它们的区间兄弟的第三个效率开销前，我有一个小修正。我在前面写的段落都是真理，而且除了真理没别的了，但并不是真理的全部。仅当可以不用失去两个迭代器的位置就能决定它们之间的距离时，一个区间insert函数才能在一次移动中把一个元素移动到它的最终位置。这几乎总是可能的，因为所有前向迭代器提供了这个功能，而且前向迭代器几乎到处都是。所有用于标准容器的迭代器都提供了前向迭代器的功能。非标准的散列容器（参见条款25）的迭代器也是。在数组中表现为迭代器的指针也提供了这样的功能。事实上，唯一不提供前向迭代器能力的标准迭代器是输入和输出迭代器。因此，除了当传给insert区间形式的迭代器是输入迭代器（比如istream_iterator——参见条款6）外，我在上面写的东西都是真的。在那个唯一的情况下，区间插入必须每次一位地把元素移动到它们的最终位置，期望中的优点就消失了。（对于输出迭代器，这个问题不会发生，因为输出迭代器不能用于为insert指定一个区间。）

留下的最后一种性能税很愚蠢，重复使用单元素插入而不是一个区间插入就必须处理内存分配，虽然在它里面也有一个令人讨厌的拷贝。就像条款14解释的，当你试图去把一个元素插入内存已经满了的vector时，这个vector会分配具有更多容量的新内存，从旧内存把它的元素拷贝到新内存，销毁旧内存里的元素，回收旧内存。然后它添加插入的元素。条款14也解释了每当用完内存时，大部分vector实现都使它们的容量翻倍，所以插入numValues个新元素会导致最多$\log_2$numValues次新内存的分配。条款14也关注了展示该行为的现有实现，所以每次一个地插入1000个元素会导致10次新的分配（包括它们负责的元素拷贝）。与之对比的是（而且，就目前来看，是可预测的），一个区间插入可以在开始插入东西前计算出需要多少新内存（假设给的是前向迭代器），所以它不用多于一次地重新分配vector的内在内存。就像你可以想象到的，这个节省相当可观。

我刚才进行分析是用于vector的，但同样的理由也作用于string。对于deque，理由也很相似，但deque管理它们内存的方式和vector和string不同，所以重复内存分配的论点不能应用。但是，关于很多次不必要的元素移动的论点通常通过对函数调用次数的观察也应用到了（虽然细节不同）。

在标准序列容器中，就剩下list了，在这里使用insert区间形式代替单元素形式也有一个性能优势。关于重复函数调用的论点当然继续有效，但因为链表的工作方式，拷贝和内存分配问题没有发生。取而代之的是，这里有一个新问题：过多重复地对list中的一些节点的next和prev指针赋值。

每当一个元素添加到一个链表时，持有元素的链表节点必须有它的next和prev指针集，而且当然新节点前面的节点（我们叫它B，就是“before”）必须设置它的next指针，新节点后面的节点（我们叫它A，就是“after”）必须设置它的prev指针：




当一系列新节点通过调用list的单元素insert一个接一个添加时，除了最后一个以外的其他新节点都会设置它的next指针*两次*，第一次指向A，第二次指向在它后面插入的元素。每次在A前面插入时，它都会设置它的prev指针指向一个新节点。如果numValues个节点插入A前面，插入节点的next指针会发生numValues-1次多余的赋值，而且A的prev指针会发生numValues-1次多余的赋值。合计2*(numValues-1)次没有必要的指针赋值。当然，指针赋值很轻量，但如果不是必须，为什么要为它们花费呢？

现在已经很清楚你可以不必，避免开销的关键是使用list的insert区间形式。因为那个函数知道最后有多少节点会被插入，它可以避免多余的指针赋值，对每个指针只使用一次赋值就能设置它正确的插入后的值。

对于标准序列容器，当在单元素插入和区间插入之间选择时，除编程风格之外还有很多东西都浮出水面。对于关联容器，效率问题几乎没有，但是附加的重复调用单元素insert函数的开销问题仍然存在。此外，区间插入的特别类型在关联容器中也可能导致优化，但据我所知，这样的优化目前只存在于理论中。当然，在你看到这点的时候，理论可能已经变成实践了，所以关联容器的区间插入可能变得比单元素插入更有效。毫无疑问它们不会降低效率，所以你选择它们没有任何损失。

即使没有效率的论点，当你写代码时使用区间成员函数需要更少的输入这个事实仍然存在，它产生的代码也更容易懂，从而增强你软件的长期维护。只要两个特性就足以使你尽量选择区间成员函数。效率优势真的只是一个红利。

经历了关于区间成员函数的奇迹的长篇大论后，只需要我为你总结一下就可以了。知道那个成员函数支持区间可以使你更容易去发现使用它们的时机。在下面的，参数类型iterator意思是容器的迭代器类型，也就是container::iterator。另一方面，参数类型InputIterator意思是可以接受任何输入迭代器。

- 区间构造。所有标准容器都提供这种形式的构造函数：

```
container::container(InputIterator begin,          // 区间的起点
                     InputIterator end);   // 区间的终点
```

如果传给这个构造函数的迭代器是istream_iterators或istreambuf_iterators（参见条款29），你可能会遇到C++的最惊异的解析，原因之一是你的编译器可能会因为把这个构造看作一个函数声明而不是一个新容器对象的定义而中断。条款6告诉你需要知道所有关于解析的东西，包括怎么对付它。

- 区间插入。所有标准序列容器都提供这种形式的insert：

```
void container::insert(iterator position,          // 区间插入的位置
                       InputIterator begin,   // 插入区间的起点
                       InputIterator end);    // 插入区间的终点
```

关联容器使用它们的比较函数来决定元素要放在哪里，所以它们了省略position参数。

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

当寻找用区间版本代替单元素插入的方法时，不要忘记有些单元素变量用采用不同的函数名伪装它们自己。比如，push_front和push_back都把单元素插入容器，即使它们不叫insert。如果你看见一个循环调用push_front或push_back，或如果你看见一个算法——比如copy——的参数是front_inserter或者back_inserter，你就发现了一个insert的区间形式应该作为优先策略的地方。

- 区间删除。每个标准容器都提供了一个区间形式的erase，但是序列和关联容器的返回类型不同。序列容器提供了这个：

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

而关联容器提供这个：

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

为什么不同？解释是如果erase的关联容器版本返回一个迭代器（被删除的那个元素的下一个）会招致一个无法接受的性能下降。我是众多发现这个徒有其表的解释的人之一，但标准说的就是标准说的，标准说erase的序列和关联容器版本有不同的返回类型。

这个条款的对insert的性能分析大部分也同样可以用于erase。单元素删除的函数调用次数仍然大于一次调用区间删除。当使用单元素删除时，每一次元素值仍然必须向它们的目的地移动一位，而区间删除可以在一个单独的移动中把它们移动到目标位置。

关于vector和string的插入和删除的一个论点是必须做很多重复的分配。（当然对于删除，会发生重复的回收。）那是因为用于vector和string的内存自动增长来适应于新元素，但当元素的数目减少时它不自动收缩。（条款17描述了你怎么减少被vector或string持有的不必要的内存。）

一个非常重要的区间erase的表现是erase-remove惯用法。你可以在条款32了解到所有关于它的信息。

- 区间赋值。就像我在这个条款的一开始提到的，所有标准列容器都提供了区间形式的assign：

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

所以现在我们明白了，尽量使用区间成员函数来代替单元素兄弟的三个可靠的论点。区间成员函数更容易写，它们更清楚地表达你的意图，而且它们提供了更高的性能。那是很难打败的三驾马车。

# 条款6：警惕C++最令人恼怒的解析

假设你有一个int的文件，你想要把那些int拷贝到一个list中。这看起来像是一个合理的方式：

```
ifstream dataFile("ints.dat");
list<int> data(istream_iterator<int>(dataFile), // 警告！这完成的并不
               istream_iterator<int>());        // 是像你想象的那样
```

这里的想法是传一对istream_iterator给list的区间构造函数（参见条款5），因此把int从文件拷贝到list中。

这段代码可以编译，但在运行时，它什么都没做。它不会从文件中读出任何数据。它甚至不会建立一个list。那是因为第二句并不声明list，而且它也不调用构造函数。其实它做的是……，它做得很奇怪。我不敢直接告诉你，因为你可能不相信我。取而代之的是，我得一点一点展开这个解释。你坐下了吗？如果没有，你可能要找找附近有没有椅子……

我们会从最基本的开始。这行声明了一个函数f带有一个double而且返回一个int：

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

第二行作了同样的事情。名为d的参数左右的括号是多余的，被忽略：

```
int f(double (d));    // 同上；d左右的括号被忽略
```

下面这行声明了同样的函数。它只是省略了参数名：

```
int f(double);        // 同上；参数名被省略
```

你应该很熟悉这三种声明形式了吧，虽然可以把括号放在参数名左右这一点可能比较新。（在不久以前我也觉得它新。）

现在让我们再看看三个函数声明。第一个声明了一个函数g，它带有一个参数，那个参数是指向一个没有参

数、返回double的函数的指针：

```
int g(double (*pf)()); // g带有一个指向函数的指针作为参数
```

这是完成同一件事的另一种方式。唯一的不同是pf使用非指针语法来声明（一个在C和C++中都有效的语法）：

```
int g(double pf());   // 同上；pf其实是一个指针
```

照常，参数名可以省略，所以这是g的第三种声明，去掉了pf这个名字：

```
int g(double ());          // 同上；参数名省略
```

注意*参数名左右*的括号（就像f的第二种声明中的d）和*单独的*括号（正如本例）之间的区别。参数名左右的括号被忽略，但单独的括号指出存在一个参数列表：它们声明了存在指向函数的指针的参数。

用这些f和g的声明做了热身，我们准备检查本条款开头的代码。这里再写一遍：

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

打起精神，这声明了一个函数data，它的返回类型是list<int>。这个函数data带有两个参数：

- 第一个参数叫做dataFile。它的类型是istream_iterator<int>。dataFile左右的括号是多余的而且被忽略。
- 第二个参数没有名字。它的类型是指向一个没有参数而且返回istream_iterator<int>的函数的指针。

奇怪吗？但这符合C++里的一条通用规则——几乎任何东西都可能被分析成函数声明。如果你用C++编程有一段时间了，你应该会遇到另一个这条规则的表象。有多少次你会看见这个错误？

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

这并没有声明一个叫做w的Widget，它声明了一个叫作w的没有参数且返回Widget的函数。学会识别这个*失言（faux pas）*是成为C++程序员的一个真正的通过仪式。

所有这些都很有趣（以它自己的扭曲方式），但它没有帮我们说出我们想要说的，也就是应该用一个文件的

内容来初始化一个list<int>对象。现在我们知道了我们必须战胜的解析，那就很容易表示了。用括号包围一个实参的声明是不合法的，但用括号包围一个函数调用的观点是合法的，所以通过增加一对括号，我们强迫编译器以我们的方式看事情：

```
list<int> data((istream_iterator<int>(dataFile)),     // 注意在list构造函数
               istream_iterator<int>());     // 的第一个实参左右的
                                             // 新括号
```

这是可能的声明数据方法，给予istream_iterators的实用性和区间构造函数（再次，参见条款5），值得知道它是怎样完成的。

不幸的是，目前并非所有编译器都知道它。在我测试的几种中，几乎一半拒绝接受数据的声明，除非它错误地接受没有附加括号形式的声明！为了安慰这样的编译器，你可以滚你眼睛并且使用我辛辛苦苦解释的错误的数据声明，但那将不可移植而且目光短浅。毕竟，目前编译器存在的分析错误肯定会在将来被修正，是吧？（肯定的！）

一个更好的解决办法是在数据声明中从时髦地使用匿名istream_iterator对象后退一步，仅仅给那些迭代器名字。以下代码到哪里都能工作：

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

命名迭代器对象的使用和普通的STL编程风格相反，但是你得判断这种方法对编译器和必须使用编译器的人都模棱两可的代码是一个值得付出的代价。

# 条款7：当使用new得指针的容器时，记得在销毁容器前delete那些指针

STL中的容器非常优秀。它们提供了前向和逆向遍历的迭代器（通过begin、end、rbegin等）；它们能告诉你所容纳的对象类型（通过value_type的typedef）；在插入和删除中，它们负责任何需要的内存管理；它们报告容纳了多少对象和最多可能容纳的数量（分别通过size和max_size）；而且当然当容器自己被销毁时会自动销毁容纳的每个对象。

给了这样聪明的容器，很多程序员不再担心用完以后的清除工作。呵呵，他们说，他们的容器会帮他们解决那个麻烦。在很多情况下，他们是对的，但当容器容纳的是指向通过new分配的对象的指针时，他们就错了。的确，当一个指针的容器被销毁时，会销毁它（那个容器）包含的每个元素，但指针的“析构函数”是无操作！它肯定不会调用delete。

结果，下面代码直接导致一个内存泄漏：

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

当vwp除了生存域后，vwp的每个元素都被销毁，但那并不改变从没有把delete作用于new得到的对象这个事实。那样的删除是你的职责，而不是vector的。这是一个特性。只有你知道一个指针是否*应该*被删除。

通常，你需要它们被删除。当情况如此时，可以很简单地实现：

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

```
            ++i) {
        delete *i;
    }
}
```

这可以工作，除非你不是对你“工作”的意思很吹毛求疵。一个问题是新的for循环代码比for_each多得多，但没有使用for_each来的清楚（参见条款43）。另一个问题是这段代码不是异常安全的。如果在用指针填充了vwp和你要删除它们之间抛出了一个异常，你会再次资源泄漏。幸运的是，两个问题都可以克服。

要把你的类似for_each的循环转化为真正使用for_each，你需要把delete转入一个函数对象中。这像儿戏般简单，假设你有一个喜欢和STL一起玩的孩子：

```
template<typename T>
struct DeleteObject :                    // 条款40描述了为什么
    public unary_function<const T*, void> { // 这里有这个继承
    void operator()(const T* ptr) const
    {
        delete ptr;
    }
};
```

现在你可以这么做：

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

不幸的是，这让你指定了DeleteObject将会删除的对象的类型（在本例中是Widget）。那是很讨厌的，vwp是一个vector<Widget*>，所以当然DeleteObject会删除Widget*指针！咄！这种冗余就不光是讨厌了，因为它会导致很难跟踪到的bug。假设，比如，有的人恶意地故意从string继承：

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

这是很危险的行为，因为string，就像所有的标准STL容器，缺少虚析构函数，而从没有虚析构函数的类公有

继承是一个大的C++禁忌。（详细信息参考任意好的C++书。在《Effective C++》中，要看的地方是条款14。）但是，仍有一些人做这种事，所以让我们考虑一下下面代码会有什么行为：

```
void doSomething()
{
    deque<SpecialString*> dssp;
    ...
    for_each(dssp.begin(), dssp.end(),          // 行为未定义！通过没有
        DeleteObject<string>());            // 虚析构函数的基类
}                                   // 指针来删除派生对象
```

注意dssp被声明为容纳SpecialString*指针，但for_each循环的作者告诉DeleteObject它将删除string*指针。很容易知道会出现什么样的错误。SpecialString的行为当然很像string，所以如果它的用户偶尔忘了他们用的是SpecialStrings而不是string是可以原谅的。

你可以通过编译器推断传给DeleteObject::operator()的指针的类型来消除这个错误（也减少DeleteObject的用户需要的击键次数）。我们需要做的所有的事就是把模板化从DeleteObject移到它的operator()：

```
struct DeleteObject {                   // 删除这里的
                                    // 模板化和基类
    template<typename T>                // 模板化加在这里
    void operator()(const T* ptr) const
    {
        delete ptr;
    }
}
```

编译器知道传给DeleteObject::operator()的指针的类型，所以我们可以让它通过指针的类型自动实例化一个operator()。这种类型演绎下降让我们放弃使DeleteObject可适配的能力（参见条款40）。想想DeleteObject的设计目的，会很难想象那会是一个问题。

使用新版DeleteObject，用于SpecialString的客户代码看起来像这样：

```
void doSomething()
{
    deque<SpecialString*> dssp;
    ...
```

```
    for_each(dssp.begin(), dssp.end(),
        DeleteObject());                // 啊！良好定义的行为！
}
```

直截了当而且类型安全，正如我们喜欢的一样。

但仍不是异常安全的。如果在SpecialString被new但在调用for_each之前抛出一个异常，就会发生泄漏。那个问题可以以多种方式被解决，但最简单的可能是用智能指针的容器来代替指针的容器，典型的是引用计数指针。（如果你不熟悉智能指针的概念，你应该可以在任何中级或高级C++书上找到描述。在《More Effective C++》中，这段材料在条款28。）

STL本身没有包含引用计数指针，而且写一个好的——一个总是可以正确工作的——非常需要技巧，除非必须否则你一定不想做。我在1996年的《More Effective C++》中发布了用于引用计数智能指针的代码，尽管是基于已制定的智能指针实现并提交给由有经验的开发人员做了很多发布前检查，小bug报告还是持续了很多年。智能指针出错的不同方式的数量很值得注意。（详细信息请参考《More Effective C++》错误列表[28]。）

幸运的是，基本上不需要你自己写，因为经过检验的实现不难找到。一个这样的智能指针是Boost库（参见条款50）中的shared_ptr。利用Boost的shared_ptr，本条款的原始例子可以重写为这样：

```
void doSomething()
{
    typedef boost::shared_ ptr<Widget> SPW; //SPW = "shared_ptr
                                // to Widget"
    vector<SPW> vwp;
    for (int i = 0; i < SOME_MAGIC_NUMBER; ++i)
        vwp.push_back(SPW(new Widget)); // 从一个Widget建立SPW,
                                // 然后进行一次push_back
        ...                 // 使用vwp
}                           // 这里没有Widget泄漏，甚至
                            // 在上面代码中抛出异常
```

你不能有的愚蠢思想是你可以通过建立auto_ptr的容器来形成可以自动删除的指针。那是很可怕的想法，非常危险。我在条款8讨论了为什么你应该避免它。

我们需要记住的所有事情就是STL容器很智能，但它们没有智能到知道是否应该删除它们所包含的指针。当你要删除指针的容器时要避免资源泄漏，你必须用智能引用计数指针对象（比如Boost的shared_ptr）来代替

指针，或者你必须在容器销毁前手动删除容器中的每个指针。

最后，你可能会想到既然一个类似DeleteObject的结构体可以简化避免容纳对象指针的容器资源泄漏，那么是否可能建立一个类似的DeleteArray结构体来简化避免容纳指向数组的指针的容器资源泄漏呢？这当然*可能*，但是否明智则不是同一回事了。条款13解释了为什么动态分配数组几乎总是劣于vector和string对象，所以在你要写DeleteArray之前，请先看看条款13。运气好的话，你会知道DeleteArray是永远不会出现的结构体。

# 条款8：永不建立auto_ptr的容器

坦白地说，本条款不需要出现在《Effective STL》里。auto_ptr的容器（COAPs）是禁止的。试图使用它们的代码都不能编译。C++ 标准委员会花费了无数努力来安排这种情况[1]。我本来不需要说有关COAPs的任何东西，因为你的编译器对这样的容器应该有很多抱怨，而且所有那些都是不能编译的。

唉，很多程序员使用STL平台不会拒绝COAPs。更糟的是，很多程序员妄想地把COAPs看作简单、直接、高效地解决经常伴随指针容器（参见条款7和33）资源泄漏的方案。结果，很多程序员被引诱去使用COAPs，即使建立它们不应该成功。

我会马上解释COAPs的幽灵有多令人担心，以至于标准化委员会采取特殊措施来保证它们不合法。现在，我要专注于一个不需要auto_ptr甚至容器知识的缺点：COAPs不可移植。它们能是怎么样的？C++标准禁止他们，比较好的STL平台已经实现了。可以有足够理由推断随着时间的推移，目前不能实现标准的这个方面的STL平台将变得更适应，并且当那发生时，使用COAPs的代码将更比现在更不可移植。如果你重视移植性（并且你应该是），你将仅仅因为它们的移植测试失败而拒绝COAPs。

但可能你没有移植性思想。如果是这样，请允许我提醒你拷贝auto_ptr的独特——有的人说是奇异——的定义。

当你拷贝一个auto_ptr时，auto_ptr所指向对象的所有权被转移到拷贝的auto_ptr，而被拷贝的auto_ptr被设为NULL。你正确地说一遍：*拷贝一个auto_ptr将改变它的值*：

```
auto_ptr<Widget> pw1(new Widget);          // pw1指向一个Widget
auto_ptr<Widget> pw2(pw1);                 // pw2指向pw1的Widget;
                                           // pw1被设为NULL。（Widget的
                                           // 所有权从pw1转移到pw2。）
pw1 = pw2;                                 // pw1现在再次指向Widget；
                                           // pw2被设为NULL
```

这非常不寻常，也许它很有趣，但你（作为STL的用户）关心的原因是它导致一些非常令人惊讶的行为。例如，考虑这段看起来很正确的代码，它建立一个auto_ptr<Widget>的vector，然后使用一个比较指向的Widget的值的函数对它进行排序。

```
bool widgetAPCompare(const auto_ptr<Widget>& lhs,
                     const auto_ptr<Widget>& rhs) {
    return *lhs < *rhs;          // 对于这个例子，假设Widget
}                                // 存在operator<

vector<auto_ptr<Widget> > widgets;          // 建立一个vector，然后
                                 // 用Widget的auto_ptr填充它；
                                 // 记住这将不能编译！
sort(widgets.begin(), widgets.end(),        // 排序这个vector
                     widgetAPCompare);
```

这里的所有东西看起来都很合理，而且从概念上看所有东西也都很合理，但结果却完全不合理。例如，在排序过程中widgets中的一个或多个auto_ptr可能已经被设为NULL。排序这个vector的行为可能已经改变了它的内容！值得去了解这是怎么发生的。

它会这样是因为实现sort的方法——一个常见的方法，正如它呈现的——是使用了快速排序算法的某种变体。我们不关心快速排序的妙处，但排序一个容器的基本思想是，选择容器的某个元素作为“主元”，然后对大于和小于或等于主元的值进行递归排序。在sort内部，这样的方法多多少少看起来像这样：

```
template<class RandomAccessIterator,        // 这个sort的声明
         class Compare>       // 直接来自于标准
void sort(RandomAccessIterator first,
          RandomAccessIterator last,
          Compare comp)
{
    // 这个typedef在下面解释
    typedef typename iterator_traits<RandomAccessIterator>::value_type
        ElementType;
    RandomAccessIterator i;
    ...                  // 让i指向主元
    ElementType pivotValue(*);          // 把主元拷贝到一个
                         // 局部临时变量中；参见
                         // 下面的讨论
    ...                  // 做剩下的排序工作
}
```

除非你是在阅读STL源代码方面很有经验，否则这看起来可能有些麻烦，但其实并不坏。唯一的难点是引用了iterator_traits<RandomAccessIterator>::value_type，而只不过是传给sort的迭代器所指向的对象类型的怪异的

STL方式。（当我们涉及iterator_traits<RandomAccessIterator>::value_type时，我们必须在它前面写上typename，因为它是一个依赖于模板参数类型的名字，在这里是RandomAccessIterator。更多关于typename用法的信息，翻到第7页。）

上面代码中棘手的是这一行，

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

因为它把一个元素从保存的区间拷贝到局部临时对象中。在我们的例子里，这个元素是一个auto_ptr<Widget>，所以这个拷贝操作默默地把被拷贝的auto_ptr——vector中的那个——设为NULL。另外，当pivotValue出了生存期，它会自动删除指向的Widget。这时sort调用返回了，vector的内容已经改变了，而且至少一个Widget已经被删除了。也可能有几个vector元素已经被设为NULL，而且几个widget已经被删除，因为快速排序是一种递归算法，递归的每一层都会拷贝一个主元。

这落入了一个很讨厌的陷阱，这也是为什么标准委员会那么努力地确保你不会掉进去。通过永不建立auto_ptr的容器来尊重它对你的利益的工作，即使你的STL平台允许那么做。

如果你的目标是智能指针的容器，这不意味着你不走运了。智能指针的容器是很好的，条款50描述了你在哪里可以找到和STL容器咬合良好的智能指针。只不过auto_ptr不是那样的智能指针。完全不是。

---

[1] 如果你对auto_ptr标准化的痛苦历程感兴趣，把你的web浏览器指向More Effective C++网站的auto_ptr更新页[29]。

# 条款9：在删除选项中仔细选择

假定你有一个标准STL容器，c，容纳int，

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

而你想把c中所有值为1963的对象都去掉。令人吃惊的是，完成这项任务的方法因不同的容器类型而不同：没有一种方法是通用的。

如果你有一个连续内存容器（vector、deque或string——参见条款1），最好的方法是erase-remove惯用法（参见条款32）：

```
c.erase(remove(c.begin(), c.end(), 1963),          // 当c是vector、string
        c.end());                    // 或deque时，
                                     // erase-remove惯用法
                                     // 是去除特定值的元素
                                     // 的最佳方法
```

这方法也适合于list，但是，正如条款44解释的，list的成员函数remove更高效：

```
c.remove(1963);      // 当c是list时，
                     // remove成员函数是去除
                     // 特定值的元素的最佳方法
```

当c是标准关联容器（即，set、multiset、map或multimap）时，使用任何叫做remove的东西都是完全错误的。这样的容器没有叫做remove的成员函数，而且使用remove算法可能覆盖容器值（参见条款32），潜在地破坏容器。（关于这样的破坏的细节，参考条款22，那个条款也解释了为什么试图在map和multimap上使用remove肯定不能编译，而试图在set和multiset上使用可能不能编译。）

不，对于关联容器，解决问题的适当方法是调用erase：

```
```

```
c.erase(1963);      // 当c是标准关联容器时
                    // erase成员函数是去除
                    // 特定值的元素的最佳方法
```

这不仅是正确的，而且很高效，只花费对数时间。（序列容器的基于删除的技术需要线性时间。）并且，关联容器的erase成员函数有基于等价而不是相等的优势，条款19解释了这一区别的重要性。

让我们现在稍微修改一下这个问题。不是从c中除去每个有特定值的物体，让我们消除下面判断式（参见条款39）返回真的每个对象：

```
bool badValue(int x);  // 返回x是否是“ bad ”
```

对于序列容器（vector、string、deque和list），我们要做的只是把每个remove替换为remove_if，然后就完成了：

```
c.erase(remove_if(c.begin(), c.end(), badValue),     // 当c是vector、string
        c.end());                 // 或deque时这是去掉
                                  // badValue返回真
                                  // 的对象的最佳方法
c.remove_if(badValue);                    // 当c是list时这是去掉
                                  // badValue返回真
                                  // 的对象的最佳方法
```

对于标准关联容器，它不是很直截了当。有两种方法处理该问题，一个更容易编码，另一个更高效。“更容易但效率较低”的解决方案用remove_copy_if把我们需要的值拷贝到一个新容器中，然后把原容器的内容和新的交换：

```
AssocContainer<int> c;                    // c现在是一种
...                               // 标准关联容器
AssocContainer<int> goodValues;           // 用于容纳不删除
                                  // 的值的临时容器
remove_copy_if(c.begin(), c.end(),                // 从c拷贝不删除
          inserter(goodValues,        // 的值到
                   goodValues.end()),         // goodValues
                   badValue);
c.swap(goodValues);                       // 交换c和goodValues
```

```
                                    // 的内容
```

对这种方法的缺点是它拷贝了所有不删除的元素，而这样的拷贝开销可能大于我们感兴趣支付的。

我们可以通过直接从原容器删除元素来避开那笔帐单。不过，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迭代c中的元素，和原来一样删除元素。

看起来，这个任务很简单，而且实际上，代码也很简单。不幸的是，那些正确工作的代码很少是跃出脑海的代码。例如，这是很多程序员首先想到的：

```
AssocContainer<int> c;
...
for (AssocContainer<int>::iterator i = c.begin();     // 清晰，直截了当
          i!= c.end();                 // 而漏洞百出的用于
          ++i) {                     // 删除c中badValue返回真
     if (badValue(*i)) c.erase(i);          // 的每个元素的代码
}                                    // 不要这么做！
```

唉，这有未定义的行为。当容器的一个元素被删时，指向那个元素的所有迭代器都失效了。当c.erase(i)返回时，i已经失效。那对于这个循环是个坏消息，因为在erase返回后，i通过for循环的++i部分自增。

为了避免这个问题，我们必须保证在调用erase之前就得到了c中下一元素的迭代器。最容易的方法是当我们调用时在i上使用后置递增：

```
AssocContainer<int> c;
...
for (AssocContainer<int>::iterator i = c.begin();     // for循环的第三部分
     i != c.end();                   // 是空的；i现在在下面
     /*nothing*/ ){                    // 自增
     if (badValue(*i)) c.erase(i++);       // 对于坏的值，把当前的
     else ++i;                             // i传给erase，然后
}                                    // 作为副作用增加i；
                                     // 对于好的值，
                                     // 只增加i
```

这种调用erase的解决方法可以工作，因为表达式i++的值是i的旧值，但作为副作用，i增加了。因此，我们把i的旧值（没增加的）传给erase，但在erase开始执行前i已经自增了。那正好是我们想要的。正如我所说的，代

码很简单，只不过不是大多数程序员在第一次尝试时想到的。

现在让我们进一步修改该问题。不仅删除badValue返回真的每个元素，而且每当一个元素被删掉时，我们也想把一条消息写到日志文件中。

对于关联容器，这说多容易就有多容易，因为只需要对我们刚才开发的循环做一个微不足道的修改就行了：

```
ofstream logFile;                       // 要写入的日志文件
AssocContainer<int> c;
...
for (AssocContainer<int>::iterator i = c.begin();     // 循环条件和前面一样
     i !=c.end();){
     if (badValue(*i)){
          logFile << "Erasing " << *i <<'\n';    // 写日志文件
          c.erase(i++);               // 删除元素
     }
     else ++i;
}
```

现在是vector、string和deque给我们带来麻烦。我们不能再使用erase-remove惯用法，因为没有办法让erase或remove写日志文件。而且，我们不能使用刚刚为关联容器开发的循环，因为它为vector、string和deque产生未定义的行为！要记得对于那样的容器，调用erase不仅使所有指向被删元素的迭代器失效，也使被删元素之后的所有迭代器失效。在我们的情况里，那包括所有i之后的迭代器。我们写i++，++i或你能想起的其它任何东西都没有用，因为没有能导致迭代器有效的。

我们必须对vector、string和deque采用不同的战略。特别是，我们必须利用erase的返回值。那个返回值正是我们需要的：一旦删除完成，它就是指向紧接在被删元素之后的元素的有效迭代器。换句话说，我们这么写：

```
for (SeqContainer<int>::iterator i = c.begin();
     i != c.end();){
     if (badValue(*i)){
          logFile << "Erasing " << *i << '\n';
          i = c.erase(i);              // 通过把erase的返回值
     }                          // 赋给i来保持i有效
     else
          ++i;
}
```

这可以很好地工作，但只用于标准序列容器。由于论证一个可能的问题（条款5做了），标准关联容器的erase的返回类型是void[1]。对于那些容器，你必须使用“后置递增你要传给erase的迭代器”技术。（顺便说说，在为序列容器编码和为关联容器编码之间的这种差别是为什么写容器无关代码一般缺乏考虑的一个例子——参见条款2。）

为了避免你奇怪list的适当方法是什么，事实表明对于迭代和删除，你可以像vector/string/deque一样或像关联容器一样对待list；两种方法都可以为list工作。

如果我们观察在本条款中提到的所有东西，我们得出下列结论：

- 去除一个容器中有特定值的所有对象：

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

  如果容器是list，使用list::remove。

  如果容器是标准关联容器，使用它的erase成员函数。

- 去除一个容器中满足一个特定判定式的所有对象：

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

  如果容器是list，使用list::remove_if。

  如果容器是标准关联容器，使用remove_copy_if和swap，或写一个循环来遍历容器元素，当你把迭代器传给erase时记得后置递增它。

- 在循环内做某些事情（除了删除对象之外）：

  如果容器是标准序列容器，写一个循环来遍历容器元素，每当调用erase时记得都用它的返回值更新你的迭代器。

  如果容器是标准关联容器，写一个循环来遍历容器元素，当你把迭代器传给erase时记得后置递增它。

如你所见，与仅仅调用erase相比，有效地删除容器元素有更多的东西。解决问题的最好方法取决于你是怎样鉴别出哪个对象是要被去掉的，储存它们的容器的类型，和当你删除它们的时候你还想要做什么（如果有的话）。只要你小心而且注意了本条款的建议，你将毫不费力。如果你不小心，你将冒着产生不必要低效的代码或未定义行为的危险。

---

[1] 这仅对带有迭代器实参的erase形式是正确的。关联容器也提供一个带有一个值的实参的erase形式，而那种形式返回被删掉的元素个数。但这里，我们只关心通过迭代器删除东西。

# 条款10：注意分配器的协定和约束

分配器是怪异的。它们最初是为抽象内存模型而开发的，允许库开发者忽略在某些16位操作系统上near和far指针的区别（即，DOS和它的有害产物），但努力失败了。分配器也被设计成促进全功能内存管理器的发展，但事实表明那种方法在STL的一些部分会导致效率损失。为了避免效率冲击，C++标准委员会向标准中添加了词语，把分配器弱化为对象，同时也表达了他们不会让操作损失能力的希望。

还有更多。正如operator new和operator new[]，STL分配器负责分配（和回收）原始内存，但分配器的客户接口与operator new、operator new[]甚至malloc几乎没有相似之处。最后（而且可能非常惊人），大多数标准容器*从未*向它们相关的分配器索要内存。从没有。结果分配器是，嗯，分配器是怪异的。

当然，那不是它们的错，而且无论如何，这不意味着它们没用。但是，在我解释分配器好在哪里之前（那是条款11的主题），我需要解释它们哪里不好。有许多事情分配器*好像*能做，但不能，而且在你试图开始使用之前，知道领域的边界很重要。如果不，你将肯定会受伤。此外，关于分配器的事实如此独特，总结它的行为既有启发性又有趣。至少我希望是。

分配器的约束的列表从用于指针和引用的残留typedef开始。正如我提到的，分配器最初被设想为抽象内存模型，在那种情况下，分配器在它们定义的内存模型中提供指针和引用的typedef才有意义。在C++标准里，类型T的对象的默认分配器（巧妙地称为allocator<T>）提供typedef allocator<T>::pointer和allocator<T>::reference，而且也希望用户定义的分配器也提供这些typedef。

C++老手立即发现这有问题，因为在C++里没有办法捏造引用。这样做要求有能力重载operator.（“点操作符”），而那是不允许的。另外，建立行为像引用的对象是使用*代理对象*的例子，而代理对象会导致很多问题。（一个这样的问题产生了条款18。对代理对象的综合讨论，转向《More Effective C++》的条款30，你能知道什么时候它们工作什么时候不。）

就STL里的分配器而言，没有任何代理对象的技术缺点会导致指针和引用typedef失效，实际上标准明确地允许库实现假设每个分配器的pointer typedef是T*的同义词，每个分配器的reference typedef与T&相同。对，库实现可以忽视typedef并直接使用原始指针和引用！所以即使你可以设法写出成功地提供新指针和引用类型的分配器的方法，也好不到哪里去，因为你使用的STL实现将自由地忽视你的typedef。很优雅，不是吗？

当你钦佩标准化的怪癖时，我将再介绍一个。分配器是对象，那表明它们可能有成员功能，内嵌的类型和typedef（例如pointer和reference）等等，但标准允许STL实现认为所有相同类型的分配器对象都是等价的而且比较起来总是相等。很唐突，听起来并不可怕，而且对它当然有好的动机。考虑这段代码：

```
template<typename T>                    // 一个用户定义的分配器
class SpecialAllocator {...};           // 模板
typedef SpecialAllocator<Widget> SAW;   // SAW = “ SpecialAllocator
                                        // for Widgets ”
list<Widget, SAW> L1;
list<Widget, SAW> L2;
...
L1.splice(L1.begin(), L2);              // 把L2的节点移到
                                        // L1前端
```

记住当list元素从一个list被接合到另一个时，没有拷贝什么。取而代之的是，调整了一些指针，曾经在一个list中的节点发现他们自己现在在另一个list中。这使接合操作既迅速又异常安全。在上面的例子里，接合前在L2里的节点接合后出现在L1中。

当L1被销毁时，当然，它必须销毁它的所有节点（以及回收它们的内存），而因为它现在包含最初是L2一部分的节点，L1的分配器必须回收最初由L2的分配器分配的节点。现在清楚为什么标准允许STL实现认为相同类型的分配器等价。所以由一个分配器对象（比如L2）分配的内存可以安全地被另一个分配器对象（比如L1）回收。如果没有这样的认为，接合操作将更难实现。显然它们不能像现在一样高效。（接合操作的存在也影响了STL的其他部分。另一个例子参见条款4。）

那当然好，但你想得越多，越会意识到STL实现可以认为相同类型的分配器等价是多严厉的约束。那意味着可移植的分配器对象——在不同的STL实现下都功能正确的分配器——不能有状态。让我们明确这一点：它意味着*可移植的分配器不能有任何非静态数据成员*，至少没有会影响它们行为的。一个都没有。没有。那表示，例如，你不能有从一个堆分配的SpecialAllocator<int>和从另一个堆分配的另一个SpecialAllocator<int>。这样的分配器不等价，而试图使用那两个分配器的现存STL实现可能导致错误的运行期数据结构。

注意这是一个*运行期*问题。有状态的分配器可以很好地编译。它们只是不按你期待的方式运行。确保一个给定类型的所有分配器都等价是你的责任。如果你违反这个限制，不要期待编译器发出警告。

为了对标准委员会公平，我应该指出，在“ 允许STL实现认为相同类型的分配器等价 ”的文字之后，紧接着有下列陈述：

> 鼓励实现提供...支持非相等实例的库。在那样的实现中，...当分配器实例非相等时，容器和算法的语义是由实现定义的。

这是个可爱的句子，但是作为一个考虑开发带状态自定义分配器的STL用户，它几乎没向你提供什么。你可以利用这句话除非（1）你知道你使用的STL实现支持不等价的分配器，（2）你愿意钻研它们的文档来确定

你是否可以接受“非相等”分配器的实现定义行为，（3）你不关心把你的代码移植到那些可能从标准给予的自由中获得好处的STL实现。简而言之，这个段落——第20.1.5节第5段，为坚持要求知道的那些人——是标准为分配器的“I have a dream”演讲。在梦想成为现实之前，关心移植性的程序员应该把他们自己限制在没有状态的自定义分配器。

我早先提及了分配器在分配原始内存方面类似operator new，但它们的接口不同。如果你看看operator new和allocator<T>::allocate最普通形式的声明，就会很清楚：

```
void* operator new(size_t bytes);
pointer allocator<T>::allocate(size_type numObjects);
                              // 记住事实上“pointer”总是
                              // T*的typedef
```

两者都带有一个指定要分配多少内存的参数，但对于operator new，这个参数指定的是字节数，而对于allocator<T>::allocate，它指定的是内存里要能容纳多少个T对象。例如，在sizeof(int) == 4的平台上，如果你要足够容纳一个int的内存，你得把4传给operator new，但你得把1传给allocator<int>::allocate。（在operator new情况下这个参数的类型是size_t，而在allocate的情况下它是allocator<T>::size_type。在两种情况里，它都是无符号整数类型，通常allocator<T>::size_type是一个size_t的typedef。）关于这个差异没有什么“错误”，但是operator new和allocator<T>::allocate之间的不同协定使应用自定义operator new的经验到开发自定义分配器的过程变得复杂。

operator new和allocator<T>::allocate的返回类型也不同。operator new返回void*，那是C++传统的表示一个到未初始化内存的指针的方式。allocator<T>::allocate返回一个T*（通过pointer typedef），不仅不传统，而且是有预谋的欺诈。从allocator<T>::allocate返回的指针并不指向一个T对象，因为T还没有被构造！在STL里暗示的是希望allocator<T>::allocate的调用者将最后在它返回的内存里构造一个或多个T对象（也许通过allocator<T>::construct，通过uninitialized_fill或通过raw_storage_iterator的一些应用），虽然在这里没有发生vector::reserve或string::reserve（参见条款14）。在operator new和allocator<T>::allocate之间返回类型的不同使未初始化内存的概念模型发生了变化，而它再次使把关于实现operator new的知识应用到开发自定义分配器变得困难。

那也带来了我们对STL分配器最后的好奇——大多数标准容器从未调用它们例示的分配器。这是两个例子：

```
list<int> L;                  // 和list<int, allocator<int> >一样；
                              // allocator<int>从未用来
                              // 分配内存！
set<Widget, SAW> s;           // 记住SAW是一个
                              // SpecialAllocator<Widget>的typedef；
                              // SAW从未分配内存！
```

这个怪癖对list和所有标准关联容器都是真的（set、multiset、map和multimap）。那是因为这些是*基于节点*的容器，即，这些容器所基于的数据结构是每当值被储存就动态分配一个新节点。对于list，节点是列表节点。对于标准关联容器，节点通常是树节点，因为标准关联容器通常用平衡二叉搜索树实现。

想一会儿可能怎么实现list<T>。list本身由节点组成，每个节点容纳一个T对象和到list中后一个和前一个节点的指针：

```
template<typename T,                // list的可能
typename Allocator = allocator<T> >    // 实现
class list{
private:
    Allocator alloc;            // 用于T类型对象的分配器

    struct ListNode{            // 链表里的节点
        T data:
        ListNode *prev;
        ListNode *next;
    };
    ...
};
```

当添加一个新节点到list时，我们需要从分配器为它获取内存，我们要的不是T的内存，我们要的是包含了一个T的ListNode的内存。那使我们的Allocator对象没用了，因为它不为ListNode分配内存，它为T分配内存。现在你理解list为什么从未让它的Allocator做任何分配了：分配器不能提供list需要的。

list需要的是从它的分配器类型那里获得用于ListNode的对应分配器的方法。按照协定，分配器得提供完成那个工作的typedef，否则将会很难办。那个typedef叫做other，但它不那么简单，因为other是嵌入一个叫做rebind的结构体的typedef，rebind自己是一个嵌入分配器的模板——分配器本身也是模板！

请不要试图考虑最后那句话。取而代之的是，看看下段代码，然后直接阅读后面的解释。

```
template<typename T>                // 标准分配器像这样声明，
class allocator {                   // 但也可以是用户写的
public:                         // 分配器模板
    template<typename U>
    struct rebind{
        typedef allocator<U> other;
```

```
    }
    ...
};
```

在list<T>的实现代码里，需要确定我们持有的T的分配器所对应的ListNode的分配器类型。我们持有的T的分配器类型是模板参数Allocator。在本例中，ListNodes的对应分配器类型是：

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

和我保持一致。每个分配器模板A（例如，std::allocator，SpecialAllocator，等）都被认为有一个叫做rebind的内嵌结构体模板。rebind带有一个类型参数，U，并且只定义一个typedef，other。other是A<U>的一个简单名字。结果，list<T>可以通过Allocator::rebind<ListNode>::other从它用于T对象的分配器（叫做Allocator）获取对应的ListNode对象分配器。

这或许对你有意义，或许不。（如果你注视它足够长时间了，它会，但你可能还必须注视一会儿。我知道我必须。）作为一个可能想要写自定义分配器的STL用户，你其实不需要知道它怎样工作。你需要知道的是如果你选择写分配器并让标准容器使用它们，你的分配器必须提供rebind模板，因为标准容器认为它在那里。（为了调试的目的，知道T对象的基于节点的容器为什么从未从T对象的分配器获取内存也是有帮助的。）

哈利路亚！我们最后完成了对分配器特质的检查。因此，如果你想要写自定义分配器，让我们总结你需要记得的事情。

- 把你的分配器做成一个模板，带有模板参数T，代表你要分配内存的对象类型。
- 提供pointer和reference的typedef，但是总是让pointer是T*，reference是T&。
- 决不要给你的分配器每对象状态。通常，分配器不能有非静态的数据成员。
- 记得应该传给分配器的allocate成员函数需要分配的*对象*个数而不是字节数。也应该记得这些函数返回T*指针（通过pointer typedef），即使还没有T对象被构造。
- 一定要提供标准容器依赖的内嵌rebind模板。

写你自己的分配器时你必须做的大部分事情是重现大量样板代码，然后修补一些成员函数，特别是allocate和deallocate。我建议你从Josuttis的样例allocator网页[23]或Austern的文章《What Are Allocators Good For?》[24]的代码开始，而不是从头开始写样板。

一旦你消化了本条款中的信息，你将知道很多关于分配器不能做的事情，但是那或许不是你想要知道的。相反，你或许想知道分配器能做什么。那有权成为一个丰富的主题，一个我称为“条款11”的主题。

# 条款11：理解自定义分配器的正确用法

你用了基准测试，性能剖析，而且实验了你的方法得到默认的STL内存管理器（即allocator <T>）在你的STL需求中太慢、浪费内存或造成过度的碎片的结论，并且你肯定你自己能做得比它好。或者你发现allocator<T>对线程安全采取了措拖，但是你只对单线程的程序感兴趣，你不想花费你不需要的同步开销。或者你知道在某些容器里的对象通常一同被使用，所以你想在一个特别的堆里把它们放得很近使引用的区域性最大化。或者你想建立一个相当共享内存的唯一的堆，然后把一个或多个容器放在那块内存里，因为这样它们可以被其他进程共享。 恭喜你！这些情况正好对应于一种适合于自定义分配器解决的方案。

例如，假定你有仿效malloc和free的特别程序，用于管理共享内存的堆，

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

并且你希望能把STL容器的内容放在共享内存中。没问题：

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

    void deallocate(pointer ptrToMemory, size_ type numObjects)
    {
        freeShared(ptrToMiemory);
    }
    ...
};
```

allocate里的pointer类型、映射和乘法的更多信息参见条款10。

你可以像这样使用SharedMemoryAllocator：

```
// 方便的typedef
typedef vector<double, SharedMemoryAllocator<double> >
                SharedDoubleVec;
...
{                                   // 开始一个块
    SharedDoubleVec v;                          // 建立一个元素在
                                    // 共享内存中的vector
    ...                             // 结束这个块
}
```

在紧挨着v定义的注释里的词语很重要。v使用SharedMemoryAllocator，所以v分配来容纳它元素的内存将来自共享内存，但v本身——包括它的全部数据成员——几乎将肯定不被放在共享内存里，v只是一个普通的基于堆的对象，所以它将被放在运行时系统为所有普通的基于堆的对象使用的任何内存。那几乎不会是共享内存。为了把v的内容和v本身放进共享内存，你必须做像这样的事情：

```
void *pVectorMemory =                       // 分配足够的共享内存
        mallocShared(sizeof(SharedDoubleVec)); // 来容纳一个
                                    // SharedDoubleVec对象
SharedDoubleVec *pv =                       // 使用“ placement new ”来
        new (pVectorMemory) SharedDoubleVec;        // 在那块内存中建立
                                    // 一个SharedDoubleVec对象；
                                    // 参见下面
                                    // 这个对象的使用（通过pv）
...
pv->~SharedDoubleVec();                     // 销毁共享内存
                                    // 中的对象
freeShared(pVectorMemory);                          // 销毁原来的
                                    // 共享内存块
```

我希望那些注释让你清楚是怎么工作的。基本上，你获得一些共享内存，然后在里面建立一个用共享内存为自己内部分配的vector。当你用完这个vector时，你调用它的析构函数，然后释放vector占用的内存。代码不很复杂，但我们在上面所做的比仅仅声明一个本地变量要苛刻得多。除非你真的要让一个容器（与它的元素相反）在共享内存里，否则我希望你能避免这个手工的四步分配/建造/销毁/回收的过程。

在这个例子里，无疑你已经注意到代码忽略了mallocShared可能返回一个null指针。显而易见，产品代码必须考虑这样一种可能性。 此外，共享内存中的vector的建立由“ placement new ”完成。如果你不熟悉placement new，你最喜欢C++课本应该可以告诉你。如果那个课本碰巧是《More Effective C++》，你将发现这个玩笑在条款8兑现。

作为分配器作用的第二个例子，假设你有两个堆，命名为Heap1和Heap2类。每个堆类有用于进行分配和回收的静态成员函数：

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

class Heap2 { ... };        // 有相同的alloc/dealloc接口
```

更进一步认为你想在不同的堆里联合定位一些STL容器的内容。同样没有问题。首先，你设计一个分配器，使用像Heap1和Heap2那样用于真实内存管理的类：

```
template<typenameT, 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来把容器的元素集合在一起：

```
vector<int, SpecificHeapAllocator<int, Heap1 > > v;              // 把v和s的元素
set<int, SpecificHeapAllocator<int Heap1 > > s;              // 放进Heap1

list<Widget,
        SpecificHeapAllocator<Widget, Heap2> > L;          // 把L和m的元素
map<int, string, less<int>,                                  // 放进Heap2
        SpecificHeapAllocator<pair<const int, string>,
                Heap2> > m;
```

在这个例子里，很重要的一点是Heap1和Heap2是*类型*而不是对象。STL为用不同的分配器对象初始化相同类型的不同STL容器提供了语法，但是我将不让你看它是什么。那是因为如果Heap1和Heap2是*对象*而不是类型，那么它们将是不等价的分配器，那就违反了分配器的等价约束，在条款10有详细说明。

因为这些例子演示的，分配器在许多情况里有用。只要你遵循相同类型的所有分配器都一定等价的限制条件，你将毫不费力地使用自定义分配器来控制一般内存管理策略，群集关系和使用共享内存以及其他特殊的堆。

# 条款12：对STL容器线程安全性的期待现实一些

标准C++的世界是相当保守和陈旧的。在这个纯洁的世界，所有可执行文件都是静态链接的。不存在内存映射文件和共享内存。没有窗口系统，没有网络，没有数据库，没有其他进程。在这种情况下，当发现标准没有提到任何关于线程的东西时你不该感到惊讶。你对STL的线程安全有的第一个想法应该是它将因实现而不同。

当然，多线程程序是很普遍的，所以大部分STL厂商努力使他们的实现在线程环境中可以正常工作。但是，即使他们做得很好，大部分负担仍在你肩上，而理解为什么会这样是很重要的。STL厂商只能为你做一些可以减少你多线程的痛苦的事情，你需要知道他们做了什么。

在STL容器（和大多数厂商的愿望）里对多线程支持的黄金规则已经由SGI定义，并且在它们的STL网站[21]上发布。大体上说，你能从实现里确定的最多是下列内容：

- 多个读取者是安全的。多线程可能同时读取一个容器的内容，这将正确地执行。当然，在读取时不能有任何写入者操作这个容器。
- 对不同容器的多个写入者是安全的。多线程可以同时写不同的容器。

就这些了，那么让我解释你可以*期望*的是什么，而不是你可以*确定*的。有些实现提供这些保证，但是有些不。

写多线程的代码很难，很多程序员希望STL实现是完全线程安全的。如果是那样，程序员可以不再需要自己做并行控制。毫无疑问这将带来很多方便，但这也非常难实现。一个库可能试图以下列方式实现这样完全线程安全的容器：

- 在每次调用容器的成员函数期间都要锁定该容器。
- 在每个容器返回的迭代器（例如通过调用begin或end）的生存期之内都要锁定该容器。
- 在每个在容器上调用的算法执行期间锁定该容器。（这事实上没有意义，因为，正如条款32所解释的，算法没有办法识别出它们正在操作的容器。不过，我们将在这里检验这个选项，因为它的教育意义在于看看为什么即使是可能的它也不能工作。）

现在考虑下列代码。它搜寻一个vector<int>中第一次出现5这个值的地方，而且，如果它找到了，就把这个值改为0。

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

在多线程环境里，另一个线程可能在行1完成之后立刻修改v中的数据。如果是那样，行2对first5和v.end的检测将是无意义的，因为v的值可能和它们在行1结束时的值不同。实际上，这样的检测会产生未定义的结果，因为另一线程可能插在行1和行2之间，使first5失效，或许通过进行一次插入操作造成vector重新分配它的内在内存。（那将使vector全部的迭代器失效。关于重新分配行为的细节，参见条款14。）类似的，行3中对*first5的赋值是不安全的，因为另一个线程可能在行2和行3之间执行，并以某种方式使first5失效，可能通过删除它指向（或至少曾经指向）的元素。

在上面列举的锁定方法都不能防止这些问题。行1中begin和end调用都返回得很快，以至于不能提供任何帮助，它们产生的迭代器只持续到这行的结束，而且find也在那行返回。

要让上面的代码成为线程安全的，v必须从行1到行3保持锁定。很难想象STL实现怎么能自动推断出这个。记住同步原语（例如，信号灯，互斥量，等等）通常开销很大，更难想象实现怎么在程序没有明显性能损失的情况下做到前面所说的——以这样的一种方式设计——让最多一个线程在1-3行的过程中能访问v。

这样的考虑解释了为什么你不能期望任何STL实现让你的线程悲痛消失。取而代之的是，你必须手工对付这些情况中的同步控制。 在这个例子里，你可以像这样做：

```
vector<int> v;
...
getMutexFor(v);
vector<int>::iterator first5(find(v.begin(), v.end(), 5));
if (first5 != v.end()) {                                      // 这里现在安全了
    *first5 = 0;                                              // 这里也是
}
releaseMutexFor(v);
```

一个更面向对象的解决方案是创建一个Lock类，在它的构造函数里获得互斥量并在它的析构函数里释放它，这样使getMutexFor和releaseMutexFor的调用不匹配的机会减到最小。这样的一个类（其实是一个类模板）基本是这样的：

```
template<typename Container>                                  // 获取和释放容器的互斥量
```

```
class Lock {                          // 的类的模板核心；
public:                               // 忽略了很多细节
    Lock(const Containers container)
            : c(container)
    {
        getMutexFor(c);               // 在构造函数获取互斥量
    }

    ~Lock()
    {
        releaseMutexFor(c);           // 在析构函数里释放它
    }

private:
    const Container& c;
};
```

使用一个类（像Lock）来管理资源的生存期（例如互斥量）的办法通常称为*资源获得即初始化*，你应该能在任何全面的C++教材里读到它。一个好的开端是Stroustrup的《The C++ Programming Language》[7]，因为Stroustrup普及了这个惯用法，但你也可以转到《More Effective C++》的条款9。不管你参考了什么来源，记住上述Lock是被剥离到最原始的本质的。一个工业强度的版本需要很多改进，但是那样的扩充与STL无关。而且这个最小化的Lock已经足够看出我们可以怎么把它用于我们一直考虑的例子：

```
vector<int> v;
...
{                                     // 建立新块；
    Lock<vector<int> > lock(v);                    // 获取互斥量
    vector<int>::iterator first5(find(v.begin(), v.end(), 5));
    if (first5 != v.end()) {
        *first5 = 0;
    }
}                                     // 关闭块，自动
                                      // 释放互斥量
```

因为Lock对象在Lock的析构函数里释放容器的的互斥量，所以在互斥量需要释放是就销毁Lock是很重要的。为了让这件事发生，我们建立一个里面定义了Lock的新块，而且当我们不再需要互斥量时就关闭那个块。这听上去像我们只是用关闭新块的需要换取了调用releaseMutexFor的需要，但是这是错误的评价。如果我们忘记为Lock建立一个新块，互斥量一样会释放，但是它可能发生得比它应该的更晚——当控制到达封闭块的末

端。如果我们忘记调用releaseMutexFor，我们将不会释放互斥量。

而且，这种基于Lock的方法在有异常的情况下是稳健的。C++保证如果抛出了异常，局部对象就会被销毁，所以即使当我们正在使用Lock对象时有异常抛出，Lock也将释放它的互斥量。如果我们依赖手工调用getMutexFor和releaseMutexFor，那么在调用getMutexFor之后releaseMutexFor之前如果有异常抛出，我们将不会释放互斥量。

异常和资源管理是重要的，但是它们不是本条款的主题。本条款是关于STL里的线程安全。当涉及到线程安全和STL容器时，你可以确定库实现允许在一个容器上的多读取者和不同容器上的多写入者。你*不能*希望库消除对手工并行控制的需要，而且你完全不能依赖于任何线程支持。

# vector和string

所有的STL容器都很有用，但如果你像大多数C++程序员，你会发现你自己接触vector和string比它们的同胞更经常。那是可以预料到的。vector和string被设计为代替大部分数组的应用，而数组很有用，它们被包含于从COBOL到Java的每个商业上成功的编程语言。

本章的条款从多个角度覆盖了vector和string。我们从一个为什么值得从数组转换过来的讨论开始，然后看看改进vector和string性能的方法，确定string的实现中重要的变种，检验怎么传递vector和string数据到只知道C的API，学习怎么除去过剩的内存分配。我们用一个有教育意义的不同寻常的东西，vector<bool>，这个不能使用的小vector的考察来作结尾。

本章的每一个条款都会帮你掌握这两个STL中最重要的容器，优化它们的应用。当我们完成的时候，你会知道怎么让它们更好地为你服务。

# 条款13：尽量使用vector和string来代替动态分配的数组

这一刻，你决定使用new来进行动态分配，你需要肩负下列职责：

1. 你必须确保有的人以后会delete这个分配。如果后面没有delete，你的new就会产生一个资源泄漏。
2. 你必须确保使用了delete的正确形式。对于分配一个单独的对象，必须使用“ delete ”。对于分配一个数组，必须使用“ delete [] ”。如果使用了delete的错误形式，结果会未定义。在一些平台上，程序在运行期会当掉。另一方面，它会默默地走向错误，有时候会造成资源泄漏，一些内存也随之而去。
3. 你必须确保只delete一次。如果一个分配被删除了不止一次，结果也会未定义。

职责真多，而且我不能理解为什么如果可以省心你却还要负责。感谢vector和string，用了它们就可以不像以前那么麻烦了。

无论何时，你发现你自己准备动态分配一个数组（也就是，企图写“ new T[...] ”），你应该首先考虑使用一个vector或一个string。（一般来说，当T是一个字符类型的时候使用string，否则使用vector，但我们在本条款的后面将遇到的情况中，vector<char>可能是一个合理的设计选择。）vector和string消除了上面的负担，因为它们管理自己的内存。当元素添加到那些容器中时它们的内存会增长，而且当一个vector或string销毁时，它的析构函数会自动销毁容器中的元素，回收存放那些元素的内存。

另外，vector和string是羽翼丰满的序列容器，所以它们让你支配可以作用于这样的容器的整个STL算法军火库。虽然数组也可以用于STL算法，但没有提供像begin、end和size这样的成员函数，也没有内嵌像iterator、reverse_iterator或value_type那样的typedef。而且char*指针当然不能和提供了专用成员函数的string竞争。STL用的越多，越会歧视内建的数组。

如果你关心你必须继续支持的遗留代码，它们都是基于数组的，放松点，无论如何都应该使用vector和string。条款16演示了把vector和string中的数据传给需要array的API有多简单，所以整合遗留代码一般都没有问题。

坦白地说，我想到了一个（也是唯一一个）用vector或string代替动态分配数组会出现的问题，而且它只关系到string。很多string实现在后台使用了引用计数（参见条款15），一个消除了不必要的内存分配和字符拷贝的策略，而且在很多应用中可以提高性能。事实上，一般认为通过引用计数优化字符串很重要，所以C++标准委员会特别设法保证了那是一个合法的实现。

唉，一个程序员的优化就是其他人的抱怨，而且如果你在多线程环境中使用了引用计数的字符串，你可能发现避免分配和拷贝所节省下的时间都花费在后台并发控制上了。（细节请参考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引用计数字符串，就应该注意线程安全性支持所带来的的性能下降问题。

要知道你正在使用的string实现是否是引用计数的，通常最简单的方式是参考库的文档。因为通常认为引用计数是一种优化，制作商一般把它作为一个特性来吹捧。另一种方法是看库的string实现的源代码。我一般不推荐尝试从库源代码中得到东西，但有时候这是唯一能找出你想知道的东西的方法。如果你选择了这个方法，就要记住string是一个basic_string<char>的typedef（而wstring是basic_string<wchar_t>的typedef），所以你真正需要看的是basic_string模板。最容易检查的地方是可能的类构造函数。看看它是否在某处增加了引用计数。如果是，string就是引用计数的。如果不是，要么就是string不是引用计数，要么就是你看错了代码。呵呵。

如果你用到的string实现是引用计数的，而你想在已经确定string的引用计数支持是一个性能问题的多线程环境中运行，你至少有三个合理的选择，而且没有一个放弃了STL。第一，看看你的库实现是否可以关闭引用计数，通常是通过改变预处理变量的值。当然那是不可移植的，但使工作变得可能，值得研究。第二，寻找或开发一个不使用引用计数的string实现（或部分实现）替代品。第三，考虑使用vector<char>来代替string，vector实现不允许使用引用计数，所以隐藏的多线程性能问题不会出现了。当然，如果你选择了vector<char>，你就放弃了string的专用成员函数，但大部分功能仍然可以通过STL算法得到，所以你从一种语法切换到另一种不会失去很多功能。

所有的结果都是简单的。如果你在使用动态分配数组，你可能比需要的做更多的工作。要减轻你的负担，就使用vector或string来代替。

# 条款14：使用reserve来避免不必要的重新分配

关于STL容器，最神奇的事情之一是只要不超过它们的最大大小，它们就可以自动增长到足以容纳你放进去的数据。（要知道这个最大值，只要调用名叫max_size的成员函数。）对于vector和string，只要需要更多空间，就以realloc等价的思想来增长。这个类似于realloc的操作有四个部分：

1. 分配新的内存块，它有容器目前容量的几倍。在大部分实现中，vector和string的容量每次以2为因数增长。也就是说，当容器必须扩展时，它们的容量每次翻倍。
2. 把所有元素从容器的旧内存拷贝到它的新内存。
3. 销毁旧内存中的对象。
4. 回收旧内存。

给了所有的分配，回收，拷贝和析构，你就应该知道那些步骤都很昂贵。当然，你不会想要比必须的更为频繁地执行它们。如果这没有给你打击，那么也许当你想到每次这些步骤发生时，所有指向vector或string中的迭代器、指针和引用都会失效时，它会给你打击的。这意味着简单地把一个元素插入vector或string的动作也可能因为需要更新其他使用了指向vector或string中的迭代器、指针或引用的数据结构而膨胀。

reserve成员函数允许你最小化必须进行的重新分配的次数，因而可以避免真分配的开销和迭代器/指针/引用失效。但在我解释reserve为什么可以那么做之前，让我简要介绍有时候令人困惑的四个相关成员函数。在标准容器中，只有vector和string提供了所有这些函数。

- size()告诉你容器中有多少元素。它*没有*告诉你容器为它容纳的元素分配了多少内存。
- capacity()告诉你容器在它已经分配的内存中可以容纳多少元素。那是容器在那块内存中*总共*可以容纳多少元素，而不是还可以容纳多少元素。如果你想知道一个vector或string中有多少没有被占用的内存，你必须从capacity()中减去size()。如果size和capacity返回同样的值，容器中就没有剩余空间了，而下一次插入（通过insert或push_back等）会引发上面的重新分配步骤。
- resize(*Container*::size_type n)强制把容器改为容纳n个元素。调用resize之后，size将会返回n。如果n小于当前大小，容器尾部的元素会被销毁。如果n大于当前大小，新默认构造的元素会添加到容器尾部。如果n大于当前容量，在元素加入之前会发生重新分配。
- reserve(*Container*::size_type n)强制容器把它的容量改为至少n，提供的n不小于当前大小。这一般强迫进行一次重新分配，因为容量需要增加。（如果n小于当前容量，vector忽略它，这个调用什么都不做，string可能把它的容量减少为size()和n中大的数，但string的大小没有改变。在我的经验中，使用reserve来从一个string中修整多余容量一般不如使用“交换技巧”，那是条款17的主题。）[1]

这个简介明确表示了只要有元素需要插入而且容器的容量不足时就会发生重新分配（包括它们维护的原始内存分配和回收，对象的拷贝和析构和迭代器、指针和引用的失效）。所以，避免重新分配的关键是使用reserve尽快把容器的容量设置为足够大，最好在容器被构造之后立刻进行。

例如，假定你想建立一个容纳1-1000值的vector<int>。没有使用reserve，你可以像这样来做：

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

在大多数STL实现中，这段代码在循环过程中将会导致2到10次重新分配。（10这个数没什么奇怪的。记住vector在重新分配发生时一般把容量翻倍，而1000约等于$2^{10}$。）

把代码改为使用reserve，我们得到这个：

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

这在循环中不会发生重新分配。

在大小和容量之间的关系让我们可以预言什么时候插入将引起vector或string执行重新分配，而且，可以预言什么时候插入会使指向容器中的迭代器、指针和引用失效。例如，给出这段代码，

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

push_back的调用不会使指向这个string中的迭代器、指针或引用失效，因为string的容量保证大于它的大小。如果不是执行push_back，代码在string的任意位置进行一个insert，我们仍然可以保证在插入期间没有发生重新分配，但是，与伴随string插入时迭代器失效的一般规则一致，所有从插入位置到string结尾的迭代器/指针/引用将失效。

回到本条款的主旨，通常有两情况使用reserve来避免不必要的重新分配。第一个可用的情况是当你确切或者大约知道有多少元素将最后出现在容器中。那样的话，就像上面的vector代码，你只是提前reserve适当数量的

空间。第二种情况是保留你可能需要的最大的空间，然后，一旦你添加完全部数据，修整掉任何多余的容量。修整部分不难，但是我将不这里显示它，因为对它有一个技巧。要学习这个技巧，请转向条款17。

---

[1] 根据勘误表，这里要加上一个注意点：调用reserve不改变容器中对象的个数。

# 条款15：小心string实现的多样性

Bjarne Stroustrup曾经用奇特的标题写一篇文章，《Sixteen Ways to Stack a Cat》[27]。事实表明实现string几乎有和那一样多的方法。当然，作为有经验而且老于世故的软件工程师，我们应该忽视“实现细节”，但是如果爱因斯坦是对的，上帝存在于细节里，现实要求我们有时皈依宗教。即使当细节不重要的时候，对它们有一些了解使我们能够确信它们不重要。

例如，一个string对象的大小是多少？换句话说，sizeof(string)返回什么值？如果你正密切注意内存消耗，这可能是一个重要的问题，或你正想用一个string对象代替一个原始的char*指针。

关于sizeof(string)的消息是“有趣”，如果你担心空间问题，这几乎肯定是你不想听到的。string和char*指针一样大的实现很常见，也很容易找到string是char*7倍大小的string实现。为什么会有差别？为了理解这一点，我们必须知道string可能存什么数据和它可能决定保存在哪里。

实际上每个string实现都容纳了下面的信息：

- 字符串的大小，也就是它包含的字符的数目。
- 容纳字符串字符的内存容量。（字符串大小和容量之间差别的回顾，参见条款14。）
- 这个字符串的值，也就是，构成这个字符串的字符。

另外，一个string可能容纳

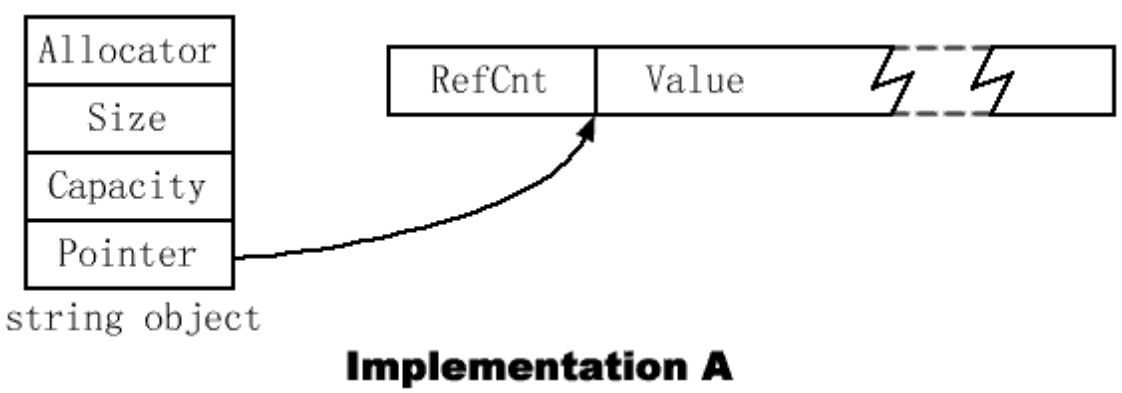
- 它的配置器的拷贝。对于为什么这个域是可选的解释，转向条款10并阅读关于这个古怪的管理分配器的规则。

依赖引用计数的string实现也包含了

- 这个值的引用计数。

不同的string实现以不同的方式把这些信息放在一起。为了证明我的意思，我将让你看四种不同的string实现使用的数据结构。并不是要特别选择这些实现，它们都来自于常用的STL实现，而正好是我检查的前四个库的string实现。

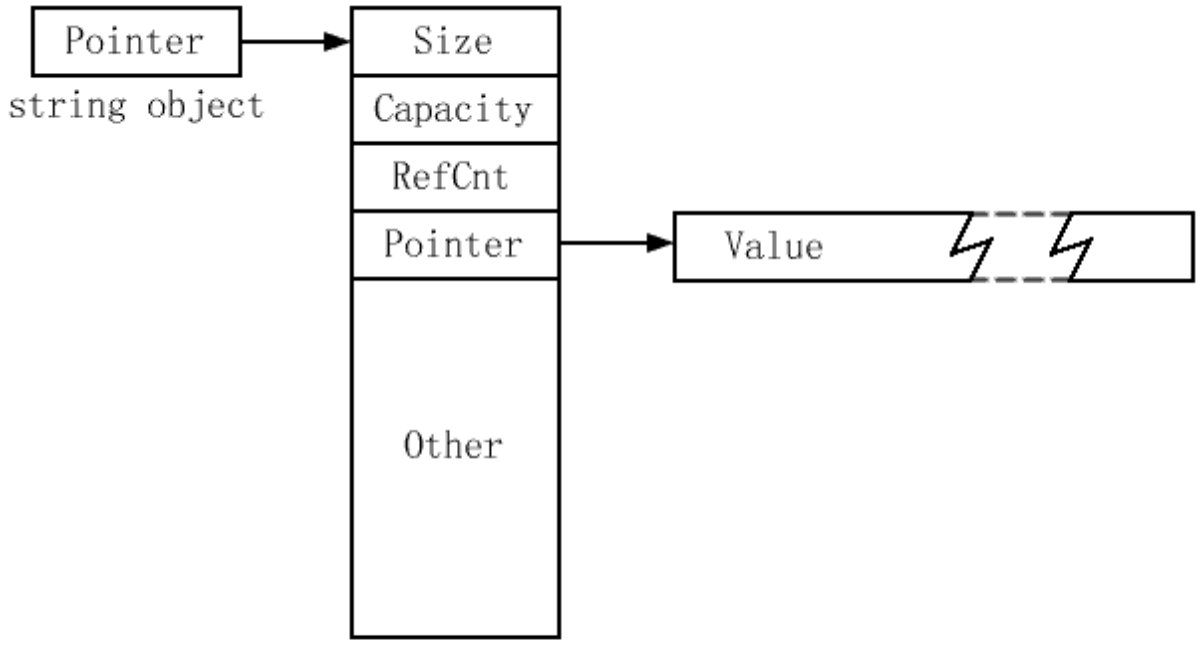
在实现A中，每个string对象包含一个它配置器的拷贝，字符串的大小，它的容量，和一个指向包含引用计数（“RefCnt”）和字符串值的动态分配的缓冲区的指针。在这实现中，一个使用默认配置器的字符串对象是指针大小的四倍。对于一个自定义的配置器，string对象会随配置器对象的增大而变大：




**Implementation A**

实现B的string对象和指针一样大，因为在结构体中只包含一个指针。再次，这里假设使用默认配置器。正如实现A，如果使用自定义配置器，这个string对象的大小会增加大约配置器对象的大小。在这个实现中，使用默认配置器不占用空间，这归功于这里用了一个在实现A中没有的使用优化。

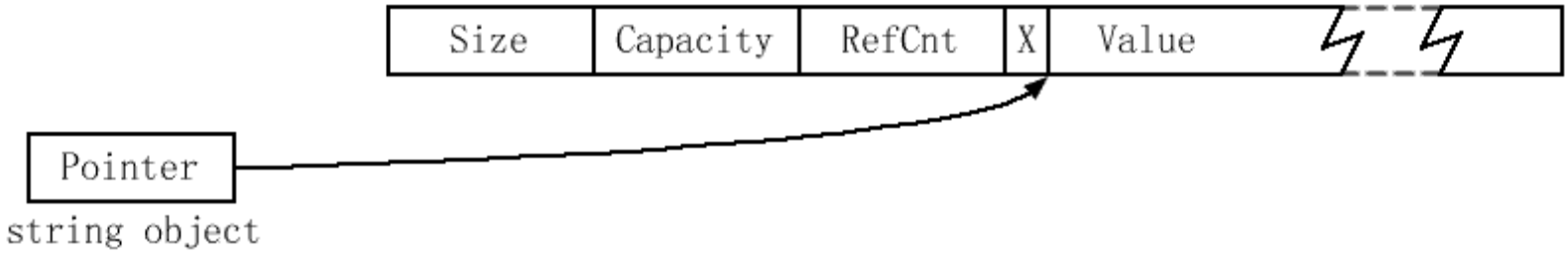
B的string指向的对象包含字符串的大小、容量和引用计数，以及容纳字符串值的动态分配缓冲区的指针。对象也包含在多线程系统中与并发控制有关的一些附加数据。这样数据在我们考虑之外，所以我只是把数据结构的那部分标记为“其他”：




**Implementation B**

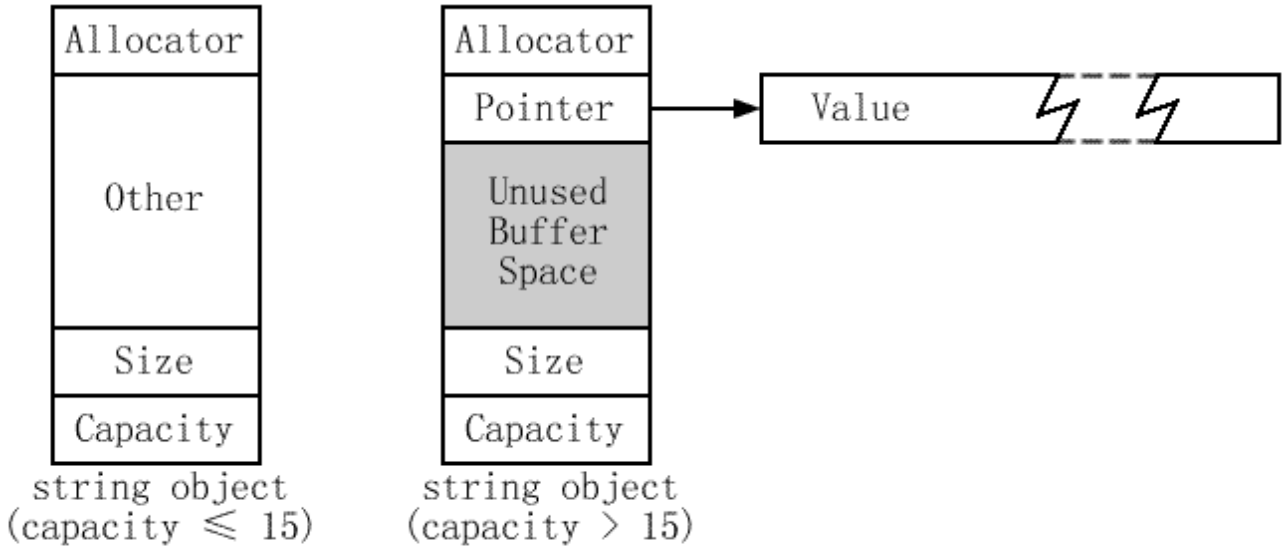
“其他”的框比其它框大，因为我按比例画框。如果一个框大小是另一个的两倍，大的框使用的字节数是小的两倍，在实现B中，用于并发控制的数据是一个指针大小的6倍。

实现C的string对象总是等于指针的大小，但是这个指针指向一个包含所有与string相关的东西的动态分配缓冲器：它的大小、容量、引用计数和值。没有每物体配置器（per-object allocator）的支持。缓冲区也容纳一些关于值可共享性的数据，我们在这里不考虑这个主题，所以我标记为“X”。（如果你首先对为什么一个引用计数值可能不可共享感兴趣，参考《More Effective C++》的条款29。）




**Implementation C**

实现D的string对象是一个指针大小的七倍（仍然假设使用了默认配置器）。这个实现没有使用引用计数，但每个string包含了一个足以表现最多15个字符的字符串值的内部缓冲区。因此小的字符串可以被整个保存在string对象中，一个有时被称为“小字符串优化”的特性。当一个string的容量超过15时，缓冲器的第一部分被用作指向动态分配内存的一个指针，而字符串的值存放在那块内存中：




**Implementation D**

这些图不仅证明了我能读源代码并能画漂亮的照片，而且它们也让你可以推断出在这样的语句中建立string，

```
string s("Perse");          // 我们的狗叫做“ Persephone ”，但我们
                            // 一般只叫她“ Perse ”。访问她的网站
                            // http://www.aristeia.com/Persephone/
```

在实现D下将会没有动态分配，在实现A和C下一次，而在实现B下两次（一次是string对象指向的对象，一次是那个对象指向的字符缓冲区）。如果你关心动态分配和回收内存的次数，或如果你关心经常伴随这样分配的内存开销，你可能想要避开实现B。另一方面，实现B的数据结构包括了对多线程系统并发控制的特殊支持的事实意味着它比实现A或C更能满足你的需要，尽管动态分配次数较多。（实现D不需要对多线程的特殊支持，因为它不使用引用计数。条款13讲了更多线程和引用计数字符串之间的关系。更多关于你可能希望的STL容器中的线程支持方面的信息，参考条款12。）

在基于引用计数的设计中，字符串对象之外的每个东西都可以被多个字符串共享（如果它们有相同的值），所以我们可以从图中观察到的其他东西是实现A比B或C提供更少的共享性。特别是，实现B和C能共享一个字符串的大小和容量，因此潜在地减少了每物体分摊的的储存数据的开销。有趣的是，实现C不能支持每对象配置器的事实意味着它是唯一可以共享配置器的实现：所有字符串必须使用同一个！（再次，管理分配器规则的细节在条款10。）实现D在字符串对象间没有共享数据。

你不能完全从图中推断出的字符串行为的一个有趣方面是关于小字符串的内存管理策略。有些实现拒绝为小于一个适当字符数分配内存，实现A、C和D就是这样。再看看这条语句：

```
string s("Perse");          // s是一个大小为5的字符串
```

实现A有32个字符的最小分配大小，所以虽然在所有实现下s的大小是5，在实现A下它的容量是31。（第32个字符大概被保留作尾部的null，因此可以容易地实现c_str成员函数。）实现C也有一个最小量，但它是16，而且没有为尾部null保留空间。所以在实现C下，s的容量是16。实现D的最小缓冲区大小也是16，包括尾部null的空间。当然，在这里区别出实现D是因为容量小于16的字符串使用的内存包含在本身字符串对象中。实现B没有最小分配，在实现B下，s的容量是7。（为什么不是6或5。我不知道。我没有那么细致地读那些源代码，抱歉。）

如果你预计会有许多短字符串和两者中任何一个（1）你的释放环境内存非常小或（2）你关心引用的地点而且想要把字符串聚集在尽量少的页面中，我觉得对于最小分配的各种各样的实现策略可能对你很重要。

很显然，string实现的自由度比乍看之下多得多，也很显然，不同的实现以不同的方式从它们的设计灵活性中得到好处。让我们总结一下：

- 字符串值可能是或可能不是引用计数的。默认情况下，很多实现的确是用了引用计数，但它们通常提供了关闭的方法，一般是通过预处理

器宏。条款13给了一个你可能要关闭的特殊环境的例子，但你也可能因为其他原因而要那么做。比如，引用计数只对频繁拷贝的字符串有帮助，而有些程序不经常拷贝字符串，所以没有那个开销。

- string对象的大小可能从1到至少7倍char*指针的大小。
- 新字符串值的建立可能需要0、1或2次动态分配。
- string对象可能是或可能不共享字符串的大小和容量信息。
- string可能是或可能不支持每对象配置器。
- 不同实现对于最小化字符缓冲区的配置器有不同策略。

现在，不要误解我。我认为string是标准库中的最重要的组件之一，而且我鼓励你尽可能经常地使用它。例如条款13，专注于你为什么应该用string来取代动态分配的字符数组。同时，如果你要有效使用STL，你需要小心string实现的多样性，特别是如果你正在写必须在不同STL平台上运行的代码并且你面临严格的性能需求。

而且，string好像很简单。谁会想到它的实现可以如此有趣？

# 条款16: 如何将vector和string的数据传给遗留的API

因为C++语言已经在1998年标准化，C++的中坚分子在努力推动程序员从数组转到vector时就没什么顾虑了。同样明显的情况也发生于尝试使开发者从char*指针迁移到string对象的过程中。有很好的理由来做这些转变，包括可以消除常见的编程错误（参见条款13），而且有机会获得STL算法的全部强大能力（比如参见条款31）。

但是，障碍还是有的，最常见的一个就是已经存在的遗留的C风格API接受的是数组和char*指针，而不是vector和string对象。这样的API函数还将会存在很长时间，如果我们要有效使用STL的话，就必须和它们和平共处。

幸运的是，这很容易。如果你有一个vector对象v，而你需要得到一个指向v中数据的指针，以使得它可以被当作一个数组，只要使用&v[0]就可以了。对于string对象s，相应的咒语是简单的s.c_str()。但是继续读下去。如广告中难懂的条文时常指出的，必然会有几个限制。

给定一个

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

表达式v[0]生产一个指向vector中第一个元素的引用，所以，&v[0]是指向那个首元素的指针。vector中的元素被C++标准限定为存储在连续内存中，就像是一个数组，所以，如果我们想要传递v给这样的C风格的API：

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

我们可以这么做：

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

也许吧。可能吧。唯一的问题就是，如果v是空的。如果这样的话，v.size()是0，而&v[0]试图产生一个指向根本就不存在的东西的指针。这不是件好事。其结果未定义。一个较安全的方法是这样：

```
if (!v.empty()) {
```

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

如果你在一个不好的环境中，你可能会碰到一些半吊子的人物，他们会告诉你说可以用v.begin()代替&v[0]，因为（这些讨厌的家伙将会告诉你）begin返回指向vector内部的迭代器，而对于vector，其迭代器实际上是指针。那经常是正确的，但正如条款50所说，并不总是如此，你不该依赖于此。begin的返回类型是iterator，而不是一个指针，当你需要一个指向vector内部数据的指针时绝不该使用begin。如果你基于某些原因决定键入v.begin()，就应该键入&*v.begin()，因为这将会产生和&v[0]相同的指针，这样可以让你有更多的打字机会，而且让其他要弄懂你代码得人感觉到更晦涩。坦白地说，如果你正在和告诉你使用v.begin()代替&v[0]的人打交道的话，你该重新考虑一下你的社交圈了。（译注：在VC6中，如果用v.begin()代替&v[0]，编译器不会说什么，但在VC7和g++中这么做的话，就会引发一个编译错误）

类似从vector上获取指向内部数据的指针的方法，对string不是可靠的，因为（1）string中的数据并没有保证被存储在独立的一块连续内存中，（2）string的内部表示形式并没承诺以一个null字符结束。这解释了string的成员函数c_str存在的原因，它返回一个按C风格设计的指针，指向string的值。因此我们可以这样传递一个string对象s给这个函数，

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

像这样：

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

即使是字符串的长度为0，它都能工作。在那种情况下，c_str将返回一个指向null字符的指针。即使字符串内部自己内含null时，它同样能工作。但是，如果真的这样，doSomething很可能将第一个内含的null解释为字符串结束。string对象不在意是否容纳了结束符，但基于char*的C风格API在意。

再看一下doSomething的声明：

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

在两种形式下，指针都被传递为指向const的指针。vector和string的数据只能传给只读取而不修改它的API。这到目前为止都是最安全的事情。对于string，这也是唯一可做的，因为没有承诺说c_str产生的指针指在string数据的内部表示形式上；它可以返回一个指针指向数据的一个不可修改的拷贝，这个拷贝满足C风格API对格式的要求。（如果这个恐吓令你毛骨悚然的话，还请放心吧，因为它也许不成立。我没听说目前哪个库的实现

使用了这个自由权的。）

对于vector，有更多一点点灵活性。如果你将v传给一个修改其元素的C风格API的话，典型情况都是没问题，但被调用的函数绝不能试图改变vector中元素的个数。比如，它绝不能试图在vector还未使用的容量上“创建”新的元素。如果这么干了，v的内部状态将会变得不一致，因为它再也不知道自己的正确大小了。v.size()将会得到一个不正确的结果。并且，如果被调用的函数试图在一个大小和容量（参见条款14）相等的vector上追加数据的话，真的会发生灾难性事件。我甚至根本就不愿去想象它。实在太可怕了。

你注意到我在前面的“典型情况都是没问题”那句话用的是“典型地”一词吗？你当然注意到了。有些vector对其数据有些额外的限制，而如果你把一个vector传递给需要修改vector数据的API，你一定要确保这些额外限制继续被满足。举个例子，条款23解释了有序vector经常可以作为关联容器的替代品，但对这些vector而言，保持顺序非常重要。如果你将一个有序vector传给一个可能修改其数据的API函数，你需要重视vector在调用返回后不再保持顺序的情况。

如果你想用C风格API返回的元素初始化一个vector，你可以利用vector和数组潜在的内存分布兼容性将存储vecotr的元素的空间传给API函数：

```
// C API：此函数需要一个指向数组的指针，数组最多有arraySize个double
// 而且会对数组写入数据。它返回写入的double数，不会大于arraySize
size_t fillArray(double *pArray, size_t arraySize);
vector<double> vd(maxNumDoubles);                    // 建立一个vector，
                                                     // 它的大小是maxNumDoubles
vd.resize(fillArray(&vd[0], vd.size()));             // 让fillArray把数据
                                                     // 写入vd，然后调整vd的大小
                                                     // 为fillArray写入的元素个数
```

这个技巧只能工作于vector，因为只有vector承诺了与数组具有相同的潜在内存分布。但是，如果你想用来自C风格API的数据初始化string对象，也很简单。只要让API将数据放入一个vector<char>，然后从vector中将数据拷到string：

```
// C API：此函数需要一个指向数组的指针，数组最多有arraySize个char
// 而且会对数组写入数据。它返回写入的char数，不会大于arraySize
size_t fillString(char *pArray, size_t arraySize);
vector<char> vc(maxNumChars);                        // 建立一个vector，
                                                     // 它的大小是maxNumChars
size_t charsWritten = fillString(&vc[0], vc.size());  // 让fillString把数据写入vc
string s(vc.begin(), vc.begin()+charsWritten);       // 从vc通过范围构造函数
```

```
                                        // 拷贝数据到s（参见条款5）
```

事实上，让C风格API把数据放入一个vector，然后拷到你实际想要的STL容器中的主意总是有效的：

```
size_t fillArray(double *pArray, size_t arraySize);          // 同上

vector<double> vd(maxNumDoubles);                             // 一样同上
vd.resize(fillArray(&vd[0], vd.size()));

deque<double> d(vd.begin(), vd.end());                 // 拷贝数据到deque
list<double> l(vd.begin(), vd.end());                  // 拷贝数据到list
set<double> s(vd.begin(), vd.end());                   // 拷贝数据到set
```

此外，这也提示了vector和string以外的STL容器如何将它们的数据传给C风格API。只要将容器的每个数据拷到vector，然后将它们传给API：

```
void doSomething(const int* pints, size_t numInts);          // C API (同上)
set<int> intSet;                                  // 保存要传递给API数据的set
...
vector<int> v(intSet.begin(), intSet.end());                  // 拷贝set数据到vector
if (!v.empty()) doSomething(&v[0], v.size());                 // 传递数据到API
```

你也可以将数据拷进一个数组，然后将数组传给C风格的API，但你为什么想这样做？除非你在编译期就知道容器的大小，否则你不得不分配动态数组，而条款13解释了为什么你应该总是使用vector来取代动态分配的数组。

# 条款17：使用“ 交换技巧 ”来修整过剩容量

假设你正在为TV游戏秀《Give Me Lots Of Money — Now!》写支持软件，而且你要跟踪可能的竞争者，你把它们保存在一个vector中：

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

当这个秀需要一个新的竞争者时，它将被申请者淹没，你的vector很快获得很多元素。但是秀的制作人只要预期的游戏者，一个相对少数符合条件的候选人移到vector前端（可能通过partial_sort或partition——参见条款31），如果不是候选人的就从vector删除（典型的通过调用erase的区间形式——参见条款5）。这很好地减少了vector的大小，但没有减少它的容量。如果你的vector有时候容纳了10万个的可能的候选人，它的容量会继续保持在至少100,000，即使后来它只容纳10个。

要避免你的vector持有它不再需要的内存，你需要有一种方法来把它从曾经最大的容量减少到它现在需要的容量。这样减少容量的方法常常被称为“ 收缩到合适（shrink to fit） ”。收缩到合适很容易实现，但代码——我该怎么说？——比直觉的要少。让我演示给你看，然后我会解释它是怎么工作的。

这是你怎么修整你的竞争者vector过剩容量的方法：

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

表达式vector<Contestant>(contestants)建立一个临时vector，它是contestants的一份拷贝：vector的拷贝构造函数做了这个工作。但是，vector的拷贝构造函数只分配*拷贝的元素需要的内存*，所以这个临时vector没有多余的容量。然后我们让临时vector和contestants交换数据，这时我们完成了，contestants只有临时变量的修整过的容量，而这个临时变量则持有了曾经在contestants中的发胀的容量。在这里（这个语句结尾），临时vector被销毁，因此释放了以前contestants使用的内存。*瞧！*收缩到合适。

同样的技巧可以应用于string：

```
string s;
...                    // 使s变大，然后删除所有
```

```
                    // 它的字符
string(s).swap(s);              // 在s上进行“ 收缩到合适 ”
```

现在，语言警察要求我告诉你并没有保证这个技术会真的消除多余的空间。如果vector和string想要的话，实现可以自由地给予它们过剩的空间，而且有时候它们想要。比如，它们可能必须有一个最小容量限制，或者它们可能强制vector或string的容量是2的整数次方。（在我的经历中，这样不规则的string实现比vector实现更常见。例子参见条款15。）这近似于“ 收缩到合适 ”，然而，并不是真的意味着“ 使容量尽可能小 ”，它意味着“ 使容量和这个实现可以尽量给容器的当前大小一样小 ”。但是，只要没有切换不同的STL实现，这是你能做的最好的方法。所以当你想对vector和string进行“ 收缩到合适 ”时，就考虑“ 交换技巧 ”。

另外，交换技巧的变体可以用于清除容器和减少它的容量到你的实现提供的最小值。你可以简单地和一个默认构造的临时vector或string做个交换：

```
vector<Contestant> v;
string s;
...                         // 使用v和s
vector<Contestant>().swap(v);       // 清除v而且最小化它的容量
string().swap(s);                   // 清除s而且最小化它的容量
```

# 条款18：避免使用vector<bool>

做为一个STL容器，vector<bool>确实只有两个问题。第一，它不是一个STL容器。第二，它并不容纳bool。除此以外，就没有什么要反对的了。

一个东西不能成为STL容器只因为会有人会说它是。一个东西要成为STL容器就必须满足所有在C++标准23.1节中列出的容器必要条件。在这些要求中有这样一条：如果c是一个T类型对象的容器，且c支持operator[]，那么以下代码必须能够编译：

```
T *p = &c[0];           // 无论operator[]返回什么，
                        // 都可以用这个地址初始化一个T*
```

换句话说，如果你使用operator[]来得到Container<T>中的一个T对象，你可以通过取它的地址而获得指向那个对象的指针。（假设T没有倔强地重载一些操作符。）然而如果vector<bool>是一个容器，这段代码必须能够编译：

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

但它不能编译。因为vector<bool>是一个伪容器，并不保存真正的bool，而是打包bool以节省空间。在一个典型的实现中，每个保存在“ vector ”中的“ bool ”占用一个单独的比特，而一个8比特的字节将容纳8个“ bool ”。在内部，vector<bool>使用了与位域（bitfield）等价的思想来表示它假装容纳的bool。

正如bool，位域也只表现为两种可能的值，但真的bool和化装成bool的位域之间有一个重要的不同：你可以创建指向真的bool的指针，但却禁止有指向单个比特的指针。

引用单个比特也是禁止的，这为vector<bool>接口的设计摆出了难题。因为vector<T>::operator[]的返回值应该是T&。如果vector<bool>真正容纳bool，这不成问题，但因为它没有，vector<bool>::operator[]需要返回指向一个比特的引用，而并不存在这样的东西。

为了解决这个难题，vector<boo>::operator[]返回一个对象，其行为*类似于*比特的引用，也称为*代理对象*。（如果仅使用STL，你并不需要明白什么是代理对象，但它是一项值得了解的C++技术。关于代理对象的信

息，参考《More Effective C++》的条款30，还有Gamma等人的《设计模式》[6]中的“ Proxy ”章节。）深入本质来看，vector<bool>看起来像这样：

```
template <typename Allocator>
vector<bool, Allocator> {
public:
    class reference {...};              // 用于产生引用独立比特的代理类
    reference operator[](size_type n);          // operator[]返回一个代理
    ...
}
```

现在，这段代码不能编译的原因就很明显了：

```
vector<bool> v;
bool *pb = &v[0];                   // 错误！右边的表达式是
                    // vector<bool>::reference*类型，
                    // 不是bool*
```

因为它不能编译，所以vector<bool>不满足STL容器的必要条件。是的，vector<bool>是在标准中，是的，它*几乎*满足了所有STL容器的必要条件，但是几乎还不够好。你写的有关STL容器的模板越多，会越深刻地认识到这一点。那天会来的。我保证，当你会写出一个模板，它只在取容器元素的地址时会产生一个指向包含类型的指针时才能工作，到那时，你将突然明白容器和几乎是容器之间的区别。

也许你想知道为什么vector<bool>存在于标准中，而它并不是一个容器。答案是与一个失败的高贵实验有关，但让我们推迟一下那个讨论，我有一个更紧迫的问题。如果vector<bool>应避免，因为它不是一个容器，那当我*需要*一个vector<bool>时应该用什么？

标准库提供了两个替代品，它们能满足几乎所有需要。第一个是deque<bool>。deque提供了几乎所有vector所提供的（唯一值得注意的是reserve和capacity），而deque<bool>是一个STL容器，它保存真正的bool值。当然，deque内部内存不是连续的。所以不能传递deque<bool>中的数据给一个希望得到bool数组的C API[1]（参见条款16），但你也不能让vector<bool>做这一点，因为没有可移植的取得vector<bool>中数据的方法。（条款16中用于vector的技术不能在vector<bool>上通过编译，因为它们依赖于能够取得指向容器中包含的元素类型的指针。我提到过vector<bool>中不保存bool值吧？）

第二个vector<bool>的替代品是bitset。bitset不是一个STL容器，但它是C++标准库的一部分。与STL容器不同，它的大小（元素数量）在编译期固定，因此它不支持插入和删除元素。此外，因为它不是一个STL容器，它也不支持iterator。但就像vector<bool>，它使用一个压缩的表示法，使得它包含的每个值只占用一比

特。它提供vector<bool>特有的flip成员函数，还有一系列其他操作位集（collection of bits）所特有的成员函数。如果不在乎没有迭代器和动态改变大小，你也许会发现bitset正合你意。

现在我们来讨论那个失败的高贵实验，它遗留下的残渣就是STL非容器的vector<bool>。我前面提到代理对象在C++软件开发中十分有用。C++标准委员会的成员当然也明白，他们决定开发vector<bool>做为说明STL可以支持包含通过代理访问元素的容器的演示。它们的理由很从分，有了这个例子在标准中，开发者将有一个现成的参考来实现自己的基于代理的容器。

唉，他们发现的是不可能创建满足所有STL容器的需要的基于代理的容器。但因为某种原因，他们没有尝试在标准中再开发一个。你可以去推测为什么保留vector<bool>，但现实地说，这没关系。重要的是：vector<bool>不满足STL容器的必要条件，你最好不要使用它；而deque<bool>和bitset是基本能满足你对vector<bool>提供的性能的需要的替代数据结构。

---

[1] 这可能是C99 API，因为bool只在这个版本的C语言中才加入。

# 关联容器

有点像在电影《Oz国历险记（The Wizard of Oz）》（译注：国内又译为《绿野仙踪》。全世界最有名的童话片之一，荣获1939年奥斯卡最佳电影歌曲和最佳电影配乐。）中的多色马，关联容器是不同颜色的生物。真的，它们共享了序列容器的很多特性，但它们在很多基本方法上不同。比如，它们自动保持自己有序；它们通过等价而不是相等来查看它们的内容；set和map拒绝重复实体；map和multimap一般忽略它们包含的每个对象的一半。是的，关联容器也是容器，但如果你原谅我把vector和string比作Kansas，我们干脆不再在Kansas了。

在下面的条款里，我解释了等价的临界概念，描述了比较函数的一个重要约束，介绍了用于指针关联容器的自定义比较函数，从官方和实践方面讨论了键的常量性，在改进关联容器的效率上提供了建议。

在本章的结尾，我考察了STL缺乏的基于散列表的容器，我也审视了两个常见（非标准）的实现。虽然STL的确没有提供散列表，但你不必自己写或放弃它。高质量的实现很容易得到。

# 条款19：了解相等和等价的区别

STL充满了比较对象是否有同样的值。比如，当你用find来定位区间中第一个有特定值的对象的位置，find必须可以比较两个对象，看看一个的值是否与另一个相等。同样，当你尝试向set中插入一个新元素时，set::insert必须可以判断那个元素的值是否已经在set中了。

find算法和set的insert成员函数是很多必须判断两个值是否相同的函数的代表。但它们以不同的方式完成，find对“相同”的定义是*相等*，基于operator==。set::insert对“相同”的定义是*等价*，通常基于operator<。因为有定义不同，所以有可能一个定义规定了两个对象有相同的值而另一个定义判定它们没有。结果，如果你想有效使用STL，那么你必须明白相等和等价的区别。

操作上来说，相等的概念是基于operator==的。如果表达式“x == y”返回true，x和y有相等的值，否则它们没有。这很直截了当，但要牢牢记住，因为x和y有相等的值并不意味着所有它们的成员有相等的值。比如，我们可能有一个内部记录了最后一次访问的Widget类。

```
class Widget {
public:
    ...

private:
    TimeStamp lastAccessed;
    ...
};
```

我们可以有一个用于Widget的忽略这个域的operator==：

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

在这里，两个Widget即使它们的lastAccessed域不同也可以有相等的值。

等价是基于在一个有序区间中对象值的相对位置。等价一般在每种标准关联容器（比如，set、multiset、map

和multimap）的一部分——排序顺序方面有意义。两个对象x和y如果在关联容器c的排序顺序中没有哪个排在另一个之前，那么它们关于c使用的排序顺序有等价的值。这听起来很复杂，但实际上，它不。考虑一下，举一个例子，一个set<Widget> s。两个Widget w1和w2，如果在s的排序顺序中没有哪个在另一个之前，那么关于s它们有等价的值。set<Widget>的默认比较函数是less<Widget>，而默认的less<Widget>简单地对Widget调用operator<，所以w1和w2关于operator<有等价的值如果下面表达式为真：

```
!(w1 < w2)                    // w1 < w2时它非真
&&                            // 而且
!(w2<w1)                      // w2 < w1时它非真
```

这个有意义：两个值如果没有哪个在另一个之前（关于某个排序标准），那么它们等价（按照那个标准）。

在一般情况下，用于关联容器的比较函数不是operator<或甚至less，它是用户定义的判断式。（关于判断式的更多信息参见条款39。）每个标准关联容器通过它的key_comp成员函数来访问排序判断式，所以如果下式求值为真，两个对象x和y关于一个关联容器c的排序标准有等价的值：

```
!c.key_comp()(x, y) && !c.key_comp()(y, x)          // 在c的排序顺序中
                                                    // 如果x在y之前它非真，
                                                    // 同时在c的排序顺序中
                                                    // 如果y在x之前它非真
```

表达式!c.key_comp()(x, y)看起来很丑陋，但一旦你知道c.key_comp()返回一个函数（或一个函数对象），丑陋就消散了。!c.key_comp()(x, y)只不过是调用key_comp返回的函数（或函数对象），并把x和y作为实参。然后对结果取反，c.key_comp()(x, y)仅当在c的排序顺序中x在y之前时返回真，所以!c.key_comp()(x, y)仅当在c的排序顺序中x*不在*y之前时为真。

要完全领会相等和等价的含义，考虑一个忽略大小写的set<string>，也就是set的比较函数忽略字符串中字符大小写的set<string>。这样的比较函数会认为“ STL ”和“ stL ”是等价的。条款35演示了怎么实现一个函数，ciStringCompare，它进行了忽略大小写比较，但set要一个比较函数的*类型*，不是真的函数。要天平这个鸿沟，我们写一个operator()调用了ciStringCompare的仿函数类：

```
struct CIStringCompare:                         // 用于忽略大小写
    public                                      // 字符串比较的类；
    binary_function<string, string, bool> {     // 关于这个基类的信息
                                                // 参见条款40
    bool operator()(const string& lhs,
                    const string& rhs) const
```

```
    {
        return ciStringCompare(lhs, rhs);          // 关于ciStringCompare
    }                                              // 是怎么实现的参见条款35
}
```

给定CIStringCompare，要建立一个忽略大小写的set<string>就很简单了：

```
set<string, CIStringCompare> ciss;          // ciss = “ case-insensitive
                                            // string set ”
```

如果我们向这个set中插入“ Persephone ”和“ persephone ”，只有第一个字符串加入了，因为第二个等价于第一个：

```
ciss.insert("Persephone");          // 一个新元素添加到set中
ciss.insert("persephone");          // 没有新元素添加到set中
```

如果我们现在使用set的find成员函数搜索字符串“ persephone ”，搜索会成功，

```
if (ciss.find("persephone") != ciss.end())...        // 这个测试会成功
```

但如果我们用非成员的find算法，搜索会失败：

```
if (find(ciss.begin(), ciss.end(),
         "persephone") != ciss.end())...       // 这个测试会失败
```

那是因为“ persephone ”等价于“ Persephone ”（关于比较仿函数CIStringCompare），但不等于它（因为string ("persephone") != string("Persephone")）。这个例子演示了为什么你应该跟随条款44的建议优先选择成员函数（就像set::find）而不是非成员兄弟（就像find）的一个理由。

你可能会奇怪为什么标准关联容器是基于等价而不是相等。毕竟，大多数程序员对相等有感觉而缺乏等价的感觉。（如果不是这样，那就不需要本条款了。）答案乍看起来很简单，但你看得越近，就会发现越多问题。

标准关联容器保持有序，所以每个容器必须有一个定义了怎么保持东西有序的比较函数（默认是less）。等价是根据这个比较函数定义的，所以标准关联容器的用户只需要为他们要使用的任意容器指定一个比较函数

（决定排序顺序的那个）。如果关联容器使用相等来决定两个对象是否有相同的值，那么每个关联容器就需要，除了它用于排序的比较函数，还需要一个用于判断两个值是否相等的比较函数。（默认的，这个比较函数大概应该是equal_to，但有趣的是equal_to从没有在STL中用做默认比较函数。当在STL中需要相等时，习惯是简单地直接调用operator==。比如，这是非成员find算法所作的。）

让我们假设我们有一个类似set的STL容器叫做set2CF，“ set with two comparison functions ”。第一个比较函数用来决定set的排序顺序，第二个用来决定是否两个对象有相同的值。现在考虑这个set2CF：

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

在这里，s内部排序它的字符串时不考虑大小写，等价标准直觉上是这样：如果两个字符串中一个等于另一个，那么它们有相同的值。让我们向s中插入哈迪斯强娶的新娘（Persephone）的两个拼写：

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

着该怎么办？如果我们说"Persephone" != "persephone"然后两个都插入s，它们应该是什么顺序？记住排序函数不能分别告诉它们。我们可以以任意顺序插入，因此放弃以确定的顺序遍历set内容的能力吗？（不能已确定的顺序遍历关联容器元素已经折磨着multiset和multimap了，因为标准没有规定等价的值（对于multiset）或键（对于multimap）的相对顺序。）或者我们坚持s的内容的一个确定顺序并忽略第二次插入的尝试（“ persephone ”的那个）？如果我们那么做，这里会发生什么？

```
if (s.find("persephone") != s.end())... // 这个测试成功或失败？
```

大概find使用了等价检查，但如果我们为了维护s中元素的一个确定顺序而忽略了第二个insert的调用，这个find会失败，即使“ persephone ”的插入由于它是一个重复的值的原则而被忽略！

总之，通过只使用一个比较函数并使用等价作为两个值“ 相等 ”的意义的仲裁者，标准关联容器避开了很多会由允许两个比较函数而引发的困难。一开始行为可能看起来有些奇怪（特别是当你发现成员和非成员find可能返回不同结果），但最后，它避免了会由在标准关联容器中混用相等和等价造成的混乱。

有趣的是，一旦你离开*有序的*关联容器的领域，情况就变了，相等对等价的问题会——已经——重临了。有两个基于散列表的非标准（但很常见）关联容器的一般设计。一个设计是基于相等，而另一个是基于等价。我鼓励你转到条款25去学更多这样的容器和设计以决定该用哪个。

# 条款20：为指针的关联容器指定比较类型

假定你有一个string*指针的set，你把一些动物的名字插入进set：

```
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"));
```

然后你写了下列代码打印set的内容，希望字符串按字母顺序出现。毕竟，确定set保持它们的内容有序。

```
for (set<string*>::const_iterator i = ssp.begin();          // 你希望看到
     i != ssp.end();             // 这个：“ Anteater ”
     ++i)                    // “ Lemur ”，“ Penguin ”，
  cout << *i << endl;                  // “ Wombat ”
```

注释描述了你希望看见的，但你根本没看见。取而代之的是，你看见四个十六进制的数。它们是指针的值。因为set容纳指针，*i不是一个string，它是一个string的指针。让这成为提醒你坚持条款43的指导并避免自己写循环的一课。如果你已经改为调用copy算法，

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

你将不仅打更少的字符，而且你将很快会查明你的错误，因为这个copy的调用将不能编译，ostream_iterator需要知道被打印的对象的类型，所以当你告诉它是一个string时（通过作为模板参数传递），编译器检测到那和ssp中储存的对象类型（是string*）之间不匹配，它们会拒绝编译代码。获得了额外的类型安全。

如果你愤怒地把显式循环中的*i改为**i，你*可能*可以得到你想要的输出，但也可能不。是的，动物名字将被打印，但它们按字母顺序出现的机会只是24份之1。ssp保持它的内容有序，但是它容纳的是指针，所以它以*指针的值*排序，而不以string值。对于四个指针值可能有24种排列（译注：4! = 4 * 3 * 2 * 1 = 24），所以指针被

储存时有24种可能的顺序。因此你看见字符串按字母排序有24份之1的几率。[1]

为了克服这个问题，你应该回忆起

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

是这个的简写：

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

好，为了完全准确，它是

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

的简化，但是分配器在本条款里与我们无关，所以我们将忽略它们。

如果你想要string*指针以字符串值确定顺序被储存在set中，你不能使用默认比较仿函数类less<string*>。你必须改为写你自己的比较仿函数类，它的对象带有string*指针并按照指向的字符串值来进行排序。就像这样：

```
struct StringPtrLess:
    public binary_function<const string*,        // 使用这个基类
                           const string*,        // 的理由参见条款40
                           bool> {
    bool operator()(const string *ps1, const string *ps2) const
    {
        return *ps1 < *ps2;
    }
};
```

然后你可以使用StringPtrLess作为ssp的比较类型：

```
typedef set<string*, StringPtrLess> StringPtrSet;
StringPtrSet ssp;                          // 建立字符串的集合，
                                // 按照StringPtrLess定义的顺序排序
```

```
...                                 // 和前面一样插入
                                    // 同样四个字符串
```

现在你的循环最后将做你想要它做的（也就是前面你使用*i代替**i所修正的问题）：

```
for (StringPtrSet::const_iterator i = ssp.begin();    // 打印“ Anteater ”，
        i != ssp.end();             // “ Lemur ”
        ++i)                    // “ Penguin ”
    cout << **i << endl;            // “ Wombat ”
```

如果你想要改为使用算法，你可以写一个知道怎么在打印string*指针之前对它们解引用的函数，然后和for_each联用那个函数：

```
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);        // 在ssp中的每个
                                    // 元素上调用print
```

或者你想象并写出了泛型的解引用仿函数类，然后让它和transform与ostream_iterator连用：

```
// 当本类型的仿函数被传入一个T*时，它们返回一个const T&
struct Dereference {
    template <typename T>
    const T& operator()(const T *ptr) const
    {
        return *ptr;
    }
};

transform(ssp.begin(), ssp.end(),                   // 通过解引用“ 转换 ”
        ostream_iterator<string>(cout, "\n"),  // ssp中的每个元素，
        Dereference());                 // 把结果写入cout
```

但是，用算法代替循环不是要点，至少对于本条款来说是这样的。（它是条款43的要点。）要点是无论何时你建立一个指针的标准关联容器，你必须记住容器会以指针的值排序。这基本上不是你想要的，所以你几乎总是需要建立自己的仿函数类作为比较类型。

注意到我写的是“比较*类型*”。你可能奇怪为什么必须特意创造一个仿函数类而不是简单地为set写一个比较函数。例如，你可能想试试：

```
bool stringPtrLess(const string* ps1,        // 将成为用于
                   const string* ps2)        // 按字符串值
{                                            // 排序的string*指针
    return *ps1 < *ps2;                      // 的比较函数
}

set<string*, stringPtrLess> ssp;             // 假设使用stringPtrLess
                                             // 作为ssp的比较函数；
                                             // 这不能编译
```

这里的问题是每个set模板的第三个参数都是一种*类型*。令人遗憾的是，stringPtrLess不是一种类型，它是一个函数。这就是为什么尝试使用stringPtrLess作为set的比较函数不能编译的原因，set不要一个函数，它要的是能在内部用实例化*建立*函数的一种类型。

无论何时你建立指针的关联容器，注意你也得指定容器的比较类型。大多数时候，你的比较类型只是解引用指针并比较所指向的对象（就像上面的StringPtrLess做的那样）。鉴于这种情况，你手头最好也能有一个用于那种比较的仿函数模板。像这样：

```
struct DereferenceLess {
    template <typename PtrType>
        bool operator()(PtrType pT1,         // 参数是值传递的，
                        PtrType pT2) const   // 因为我们希望它们
        {                                    // 是（或行为像）指针
            return *pT1 < *pT2;
        }
};
```

这样的模板消除了写像StringPtrLess那样的类的需要，因为我们可以改为使用DereferenceLess：

```
set<string*, DereferenceLess> ssp;           // 行为就像
```

```
                    // set<string*, StringPtrLess>
```

噢，还有一件事。本条款是关于指针的关联容器，但它也可以应用于表现为指针的对象的容器，例如，智能指针和迭代器。如果你有一个智能指针或迭代器的关联容器，那也得为它指定比较类型。幸运的是，指针的这个解决方案也可以用于类似指针的对象。正如DereferenceLess适合作为T*的关联容器的比较类型一样，它也可以作为T对象的迭代器和智能指针容器的比较类型。

---

[1] 实际上，这24种排列很可能不是平等的，所以“24份之1”的陈述有点使人误解。确实，有24个不同的顺序，而且你可能得到它们中的任何一个。

# 条款21: 永远让比较函数对相等的值返回false

让我向你展示一些比较酷的东西。建立一个set，比较类型用less_equal，然后插入一个10：

```
set<int, less_equal<int> > s;                    // s以“ <= ”排序
s.insert(10);                                    // 插入10
```

现在尝试再插入一次10：

```
s.insert(10);
```

对于这个insert的调用，set必须先要判断出10是否已经位于其中了。 我们知道它是，但set可是木头木脑的，它必须执行检查。为了便于弄明白发生了什么，我们将一开始已经在set中的10称为$10_A$，而正试图插入的那个10叫$10_B$。

set遍历它的内部数据结构以查找哪儿适合插入$10_B$。最终，它总要检查$10_B$是否与$10_A$相同。关联容器对“ 相同 ”的定义是等价（参见条款19），因此set测试$10_B$是否等价于$10_A$。当执行这个测试时，它自然是使用set的比较函数。在这一例子里，是operator<=，因为我们指定set的比较函数为less_equal，而less_equal意思就是operator<=。于是，set将计算这个表达式是否为真：

```
!(10_A <= 10_B) && !(10_B <= 10_A)                // 测试10_A和10_B是否等价
```

哦，$10_A$和$10_B$都是10，因此，$10_A <= 10_B$ 肯定为真。同样清楚的是，$10_B <= 10_A$。于是上述的表达式简化为

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

再简化就是

```
false && false
```

结果当然是false。 也就是说，set得出的结论是$10_A$与$10_B$不等价，因此不一样，于是它将$10_B$插入容器中$10_A$的旁边。在技术上而言，这个做法导致未定义的行为，但是通常的结果是set以拥有了两个为10的值的拷贝而告终，也就是说它不再是一个set了。通过使用less_equal作为我们的比较类型，我们破坏了容器！此外，任何对相等的值返回true的比较函数都会做同样的事情。是不是很酷？

OK，也许你对酷的定义和我不一样。就算这样，你仍然需要确保你用在关联容器上的比较函数总是对相等的值返回false。但是，你需要保持警惕。对这条规则的违反容易达到令人吃惊的后果。

举个例子，条款20描述了该如何写一个比较函数以使得容纳string*指针的容器根据string的值排序，而不是对指针的值排序。那个比较函数是按升序排序的，但我们现在假设你需要string*指针的容器的降序排序的比较函数。自然是抓现成的代码来修改了。如果你不细心，可能会这么干，我已经加亮了对条款20中代码作了改变的部分：

```
struct StringPtrGreater:                    // 高亮显示
public binary_function<const string*,       // 这段代码和89页的改变
const string*,                              // 当心，这代码是有瑕疵的！
bool> {
    bool operator()(const string *ps1, const string *ps2) const
    {
        return !(*ps1 < *ps2);              // 只是相反了旧的测试；
    }                                       // 这是不对的！
};
```

这里的想法是通过将比较函数内部结果取反来达到反序的结果。很不幸，取反“ < ”不会给你（你所期望的）“ > ”，它给你的是“ >= ”。而你现在知道，因为它将对相等的值返回true，对关联容器来说，它是一个无效的比较函数。

你真正需要的比较类型是这个：

```
struct StringPtrGreater:                    // 对关联容器来说
public binary_function<const string*,       // 这是有效的比较类型
const string*,
bool> {
    bool operator()(const string *ps1, const string *ps2) const
    {
        return *ps2 < *ps1;                 // 返回*ps2是否
```

```
    }                           // 大于*ps1（也就是
};                              // 交换操作数的顺序）
```

要避免掉入这个陷阱，你所要记住的就是比较函数的返回值表明的是在此函数定义的排序方式下，一个值是否大于另一个。相等的值绝不该一个大于另一个，所以比较函数总应该对相等的值返回false。

唉。

我知道你在想什么。你正在想，“当然，这对set和map很有意义，因为这些容器不能容纳复本。但是multiset和multimap怎么样呢？那些容器可以容纳复本，那些容器可能包含副本，因此，如果容器认为两个值相等的对象不等价，我需要注意些什么？它将会把两个都存储进去的，这正是multi系列容器的所要支持的事情。没有问题，对吧？”

错。想知道为什么，让我们返回头去看最初的例子，但这次使用的是一个mulitset：

```
multiset<int, less_equal<int> > s;          // s仍然以“<=”排序
s.insert(10);                   // 插入10A
s.insert(10);                   // 插入10B
```

现在，s里有两个10的拷贝，因此我们期望如果我们在它上面做一个equal_range，我们将会得到一对指出包含这两个拷贝的范围的迭代器。但那是不可能的。equal_range，虽然叫这个名字，但不是指示出相等的值的范围，而是等价的值的范围。在这个例子中，s的比较函数说$10_A$和$10_B$是不等价的，所以不可能让它们同时出现在equal_range所指示的范围内。

你明白了吗？除非你的比较函数总是为相等的值返回false，你将会打破所有的标准关联型容器，不管它们是否允许存储复本。

从技术上说，用于排序关联容器的比较函数必须在它们所比较的对象上定义一个“严格的弱序化(strict weak ordering)”。（传给sort等算法（参见条款31）的比较函数也有同样的限制）。如果你对严格的弱序化含义的细节感兴趣，可在很多全面的STL参考书中找到，比如Josuttis的《The C++ Standard Library》[3]（译注：中译本《C++标准程序库》P176），Austern的《Generic Programming and the STL》[4]，和SGI STL的网站[21]。我从未发现这个细节如此重要，但一个对严格的弱序化的要求直接指向了这个条款。那个要求就是任何一个定义了严格的弱序化的函数都必须在传入相同的值的两个拷贝时返回false。

嗨! 这就是这个条款！

# 条款22：避免原地修改set和multiset的键

本条款的动机很容易理解。正如所有标准关联容器，set和multiset保持它们的元素有序，这些容器的正确行为依赖于它们保持有序。如果你改了关联容器里的一个元素的值（例如，把10变为1000），新值可能不在正确的位置，而且那将破坏容器的有序性。很简单，是吗？

这对于map和multimap特别简单，因为试图改变这些容器里的一个键值的程序将不能编译：

```
map<int, string> m;
...
m.begin()->first = 10;              // 错误！map键不能改变
multimap<int, string> mm;
...
mm.begin()->first = 20;             // 错误！multimap键也不能改变
```

那是因为map<K, V>或multimap<K, V>类型的对象中元素的类型是pair<const K, V>。因为键的类型const K，它不能改变。（嗯，如果你使用一个const_cast，正如我们将在下面看到的，你或许能改变它。不管相信与否，有时那是你*想要*做的。）

但注意本条款的标题没有提及map或multimap。那有一个原因。正如上面例子演示的，原地修改键对map和multimap来说是不可能的（除非你使用映射），但是它对set和multiset却是可能的。对于set<T>或multiset<T>类型的对象来说，储存在容器里的元素类型只不过是T，并非const T。因此，set或multiset里的元素可能在你想要的任何时候改变。不需要映射。（实际上，事情不完全那么简单，但我们不久将看到。没理由超过自己。我们先得会爬，然后才能在碎玻璃上爬。）

让我们从明白为什么set或multiset里的元素不是常数开始。假设我们有一个雇员的类：

```
class Employee {
public:
    ...
    const string& name() const;              // 获取雇员名
    void setName(const string& name);        // 设置雇员名
    const string& getTitle() const;      // 获取雇员头衔
```

```
    void setTitle(string& title);       // 设置雇员头衔
    int idNumber() const;               // 获取雇员ID号
    ...
}
```

如你所见，我们可以得到雇员各种各样的信息。不过，让我们做合理的假设，每个雇员有唯一的ID号，就是idNumber函数返回的数字。 然后，建立一个雇员的set，很显然应该只以ID号来排序set：

```
struct IDNumberLess{
    public binary_function<Employee, Employee, bool> {    // 参见条款40
        bool operator()(const Employees lhs,
                const Employee& rhs) const
        {
            return lhs.idNumber() < rhs.idNumber();
        }
};

typedef set<Employee, IDNumberLess> EmpIDSet;
EmpIDSet se;                                // se是雇员的set，
                                    // 按照ID号排序
```

实际上，雇员的ID号是set中元素的键。其余的雇员数据只是虚有其表。在这里，没有理由不能把一个特定雇员的头衔改成某个有趣的东西。像这样：

```
Employee selectedID;                        // 容纳被选择的雇员
...                                 // ID号的变量
EmpIDSet::iterator i = se.find(selectedID);
if (i != se.end()){
    i->setTitle("Corporate Deity");             // 给雇员新头衔
}
```

因为在这里我们只是改变雇员的一个与set排序的方式无关的方面（一个雇员的非键部分），所以这段代码不会破坏set。那是它合法的原因。但它的合法排除了set/multiset的元素是const的可能。而且那是它们为什么不是的原因。

你可能想知道，为什么相同的逻辑不能应用于map和multimap的键？难道建立一个从雇员到，比如说，他们居住的国家的map，map的比较函数是IDNumberLess，正如前一个例子，没有意义吗？而且如果给定了这样

一个map，难道没有理由在不影响雇员ID号的情况下改变一个雇员的职位，正如前一个例子?

坦率地说，我认为它可以。不过，由于同样的坦率，我怎么认为是不重要的。重要的是标准委员会怎么认为，而它认为的是map/multimap键应该是const而set/multiset的值不是。

因为set或multiset里的值不是const，所以试图改变它们可以编译。本条款的目的是提醒你如果你改变set或multiset里的元素， 你必须确保不改变一个*键部分*——影响容器有序性的元素部分。如果你做了，你会破坏容器，再使用那个容器将产生未定义的结果， 而且那是你的错误。另一方面，这个限制只应用于被包含对象的键部分。对被包含元素的所有其他部分来说，是开放的：随便改变！

除了碎玻璃以外。记得我早先提及的碎玻璃吗？我们现在在那里了。抓一些绷带跟我来。

即使set和multiset的元素不是const，实现仍然有很多方式可以阻止它们被修改。例如，实现可以让用于set<T>::iterator的operator*返回一个常数T&。即，它可以让set的迭代器解引用的结果是set元素的常量引用。在这样的实现下，将没有办法修改set或multiset的元素，因为所有访问那些元素的方法都将在让你访问之前加一个const。

这样的实现合法吗？可以证明是。也可以证明不是。标准在这里有矛盾，而根据Murphy定律（译注：Murphy定律就是“ If there are two or more ways to do something, and one of those ways can result in a catastrophe, then someone will do it. ”（当有两条或更多的路让你抉择，如果其中一条会导致失败，那么你一定会选到它。）随着时间流逝，这一“ 定律 ”逐渐进入习语范畴，其内涵被赋予无穷的创意，也出现了众多的变体，其中最著名的一条也被称为Finagle定律，具体内容为：“ If anything can go wrong, it will. ”（如果一个东西可能会出错，那它一定会出错。）这一定律被认为是对“ Murphy定律 ”最好的模仿和阐述。），不同实现会以不同的方式解释它们。结果是发现这些代码，我早先声明可以编译的，却经常在某些STL实现上不能编译：

```
EmpIDSet se;                        // 同前，se是一个以ID号
                                    // 排序的雇员set
Employee selectedID;                // 同前，selectedID是一个带有
                                    // 被选择ID号的雇员
...
EmpIDSet::iterator i = se.find(selectedID);
if (i != se.end()){
    i->setTitle("Corporate Deity");     // 有些STL实现会
}                                   // 拒绝这行
```

因为标准的模棱两可和它已经造成的解释区别，试图修改set或multiset中的元素的代码不可移植。

所以我们站在哪边？值得鼓舞的是，事情并不复杂：

- 如果不关心移植性，你想要改变set或multiset中元素的值，而且你的STL实现让你侥幸成功，继续做。只是要确定不要改变元素的键部分，即，会影响容器有序性的元素部分。
- 如果你在乎移植性，就认为set和multiset中的元素不能被修改，至少不能在没有映射的情况下。

啊，映射。我们已经看过有时候完全有理由改变set或multiset元素的非键部分，所以我觉得我得被迫向你演示怎样做。也就是怎样做才能既正确又可移植。它不难，但是它用到了太多程序员忽略的一个细节：你必须映射到一个*引用*。作为一个例子，再次看看我们刚看的不能在一些实现下编译的setTitle调用：

```
EmpIDSet::iterator i = se.find(selectedID);
if (i != se.end()) {
    i->setTitle("Corporate Deity");       // 有些STL实现会
}                                         // 拒绝这样，因为*i是const
```

为了让它可以编译并且行为正确，我们必须映射掉*i的常量性。这是那么做的正确方法：

```
if (i != se.end()) {                                  // 映射掉
    const_cast<Employee&>(*i).setTitle("Corporate Deity");       // *i的
}                                                     // 常量性
```

这可以得到i指向的对象，告诉你的编译器把映射的结果当作一个（非常数）Employee的引用，然后在那个引用上调用setTitle。除了解释为什么这可以工作，我将解释为什么一种可替代的方法不表现人们经常期望的样子。

很多人想到的是这段代码：

```
if (i != se.end()){                                   // 把*i
    static_cast<Employee>(*i).setTitle("Corporate Deity");       // 映射到
}                                                     // 一个Employee
```

它也等价于如下内容：

```
if (i != se.end()) {                                  // 同上，
    ((Employee)(*i)).setTitle("Corporate Deity");        // 但使用C
}                                                     // 映射语法
```

这两个都能编译，而且因为它们等价，所以它们错的原因也相同。在运行期，它们不能修改*i！在这两个情况里，映射的结果是一个*i副本的临时匿名对象，而setTitle是在匿名的物体上调用，不在*i上！*i没被修改，因为setTitle从未在那个对象上调用，它在那个对象的*副本*上调用。两个句法形式等价于这个：

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

现在应该清楚映射到引用的重要性了吧。通过映射到引用，我们避免了建立一个新对象。取而代之的是，映射的结果是一个现有对象的引用，i指向的对象。当我们在有这个引用指定的对象上调用setTitle时，我们是在*i上调用setTitle，而且那正是我们想要的。

我刚才写的适用于set和multiset，但当我们转向map和multimap时，细节变粗了。注意map<K, V>或multimap<K, V>包含pair<*const* K, V>类型的元素。那个const表明pair的第一个组件被*定义*为常量，而那意味着试图修改它是未定义的行为（即使映射掉它的常量性）。理论上，一个STL实现可能把这样的值写到一个只读的内存位置（比如，一旦写了就通过系统调用进行写保护的虚拟内存页），而且试图映射掉它的常量性，最多，没有效果。我从未听说一个实现那么做，但如果你是一个坚持遵循标准拟定的规则的人，你*绝不会*试图映射掉map或multimap键的常量性。

你肯定听过映射是危险的，而且我希望本书能清楚到让我相信你可以尽量避免它们的程度。进行映射将临时剥去类型系统的安全性，而且当你把安全网抛至脑后时，我们已经讨论的缺陷例证了所能发生的。

大多数映射可以避免，那包括我们刚刚考虑的。如果你要总是可以工作而且总是安全地改变set、multiset、map或multimap里的元素，按五个简单的步骤去做：

1. 定位你想要改变的容器元素。如果你不确定最好的方法，条款45提供了关于怎样进行适当搜寻的指导。
2. 拷贝一份要被修改的元素。对map或multimap而言，确定不要把副本的第一个元素声明为const。毕竟，你想要改变它！
3. 修改副本，使它有你想要在容器里的值。
4. 从容器里删除元素，通常通过调用erase（参见条款9）。
5. 把新值插入容器。如果新元素在容器的排序顺序中的位置正好相同或相邻于删除的元素，使用insert的“提示”形式把插入的效率从对数时间改进到分摊的常数时间。使用你从第一步获得的迭代器作为提示。

这是同一个累人的雇员例子，这次以安全、可移植的方式写：

```
EmpIDSet se;                        // 同前，se是一个以ID号
                                    // 排序的雇员set
Employee selectedID;                // 同前，selectedID是一个带有
                                    // 需要ID号的雇员
...
EmpIDSet::iterator i =
    se.find(selectedID);            // 第一步：找到要改变的元素
if (i!=se.end()){
    Employee e(*i);                 // 第二步：拷贝这个元素
    se.erase(i++);                  // 第三步：删除这个元素；
                                    // 自增这个迭代器以
                                    // 保持它有效（参见条款9）
    e.setTitle("Corporate Deity");  // 第四步：修改这个副本
    se.insert(i, e);                // 第五步：插入新值；提示它的位置
                                    // 和原先元素的一样
}
```

你将原谅我以这种方法放它，但要记得*关键*的事情是对于set和multiset，如果你进行任何容器元素的原地修改，你有责任确保容器保持有序。

# 条款23：考虑用有序vector代替关联容器

当需要一个提供快速查找的数据结构时，很多STL程序员立刻会想到标准关联容器：set、multiset、map和multimap。直到现在这很好，但不是永远都好。如果查找速度真得很重要，的确也值得考虑使用非标准的散列容器（参见条款25）。如果使用了合适的散列函数，则可以认为散列容器提供了常数时间的查找。（如果选择了不好的散列函数或表的太小，散列表的查找性能可能明显下降，但在实际中这相对少见。）对于多数应用，被认为是常数时间查找的散列容器要好于保证了对数时间查找的set、map和它们的multi同事。

即使你需要的就只是对数时间查找的保证，标准关联容器仍然可能不是你的最佳选择。和直觉相反，对于标准关联容器，所提供的性能也经常劣于本该比较次的vector。如果你要有效使用STL，你需要明白什么时候和怎么让一个vector可以提供比标准关联容器更快的查找。

标准关联容器的典型实现是平衡二叉查找树。一个平衡二叉查找树是一个对插入、删除和查找的混合操作优化的数据结构。换句话说，它被设计为应用于进行一些插入，然后一些查找，然后可能再进行一些插入，然后也许一些删除，然后再来一些查找，然后更多的插入或删除，然后更多的查找等。这个事件序列的关键特征是插入、删除和查找都是混合在一起的。一般来说，没有办法预测对树的下一个操作是什么。

在很多应用中，使用数据结构并没有那么混乱。它们对数据结构的使用可以总结为这样的三个截然不同的阶段：

1. 建立。通过插入很多元素建立一个新的数据结构。在这个阶段，几乎所有的操作都是插入和删除。几乎没有或根本没有查找。
2. 查找。在数据结构中查找指定的信息片。在这个阶段，几乎所有的操作都是查找。几乎没有或根本没有插入和删除。
3. 重组。修改数据结构的内容，也许通过删除所有现有数据和在原地插入新数据。从动作上说，这个阶段等价于阶段1。一旦这个阶段完成，应用程序返回阶段2。

对于这么使用它们的数据结构的应用来说，一个vector可能比一个关联容器能提供更高的性能（时间和空间上都是）。但不是任意的vector都会，只有有序vector。因为只有有序容器才能正确地使用查找算法——binary_search、lower_bound、equal_range等（参见条款34）。但为什么一个（有序的）vector的二分法查找比一个二叉树的二分法查找提供了更好的性能？因为有些东西是过时的但却是真的，其中的一个是大小问题。其他东西不那么过时也不那么真，其中的一个是引用局部性问题。

考虑第一个大小问题。假设我们需要一个容器来容纳Widget对象，而且，因为查找速度对我们很重要，我们

考虑一个Widget的关联容器和一个有序vector<Widget>。如果我们选择一个关联容器，我们几乎确定了要使用平衡二叉树。这样的树是由树节点组成，每个都不仅容纳了一个Widget，而且保存了一个该节点到左孩子的指针，一个到它右孩子的指针，和（典型的）一个到它父节点的指针。这意味着在关联容器中用于存储一个Widget的空间开销至少会是三个指针。

与之相对的是，当在vector中存储Widget并没有开销：我们简单地存储一个Widget。当然，vector本身有开销，在vector结尾也可能有空的（保留）空间（参见条款14），但是每个vector开销是可以忽略的（通常是三个机器字，比如，三个指针或两个指针和一个int），而且如果必要的话，末尾空的空间可以通过“交换技巧”去掉（看见条款17）。即使这个附加的空间没有去掉，也并不影响下面的分析，因为当查找时不会引用那段内存。

假设我们的数据结构足够大，它们可以分成多个内存页面，但是vector比关联容器需要的页面要少。那是因为vector不需要每个Widget的开销，而关联容器给每个Widget上附加了三个指针。要知道为什么这很重要，假设在你使用的系统上一个Widget的大小是12个字节，指针是4个字节，一个内存页面是4096（4K）字节。忽略每个容器的开销，当用vector保存时，你可以在一页面上放置341个Widget，但使用关联容器时你最多只能放170个。因此关联容器和vector比起来，你将会使用大约两倍的内存。如果你使用的环境可以用虚拟内存，就很可以容易地看出那会造成大量的页面错误，因此一个系统会因为大数据量而明显慢下来。

实际上我在这里还是对关联容器很乐观的，因为我们假设在二叉树中的节点都群集在一个相关的小内存页面集中。大部分STL实现使用自定义内存管理器（实现在容器的配置器上——参见条款10和11）来达到这样的群集，但是如果你的STL实现没有改进树节点中的引用局部性，这些节点会分散在所有你的内存空间。那会导致更多的页面错误。即使使用了自定义群集内存管理器，关联容器也会导致很多页面错误，因为，不像连续内存容器，比如vector，基于节点的容器更难保证在容器的遍历顺序中一个挨着一个的元素在物理内存上也是一个挨着一个。但当进行二分查找时那种内存组织方式（译注：遍历顺序中一个挨着一个的元素在物理内存上也是一个挨着一个）正好是页面错误最少的。

概要：在有序vector中存储数据很有可能比在标准关联容器中保存相同的数据消耗更少的内存；当页面错误值得重视的时候，在有序vector中通过二分法查找可能比在一个标准关联容器中查找更快。

当然，有序vector的大缺点是它必须保持有序！当一个新元素插入时，大于这个新元素的所有东西都必须向上移一位。它和听起来一样昂贵，如果vector必须重新分配它的内在内存（参见条款14），则会更昂贵，因为vector中所有的元素都必须拷贝。同样的，如果一个元素从vector中被删除，所有大于它的元素都要向下移动。vector的插入和删除都很昂贵，但是关联容器的插入和删除则很轻量。这就是为什么只有当你知道你的数据结构使用的时候查找几乎不和插入和删除混合时，使用有序vector代替关联容器才有意义。

本条款有很多文字，但不幸的是只有很少的例子，所以我们来看看一个使用有序vector代替set的代码骨架：

```
vector<Widget> vw;                    // 代替set<Widget>
```

```
...                                     // 建立阶段：很多插入，
                                        // 几乎没有查找
sort(vw.begin(), vw.end());                     // 结束建立阶段。（当
                                        // 模拟一个multiset时，你
                                        // 可能更喜欢用stable_sort
                                        // 来代替；参见条款31。）
Widget w;                                   // 用于查找的值的对象
...                                     // 开始查找阶段
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))...                  // 条款19解释了
                                        // “!(w < *i)”测试
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...
```

就像你可以看到的，这非常直截了当。里面最难的东西就是怎么在搜索算法中做出选择（比如，binary_search、lower_bound等），条款45可以帮你做出选择。

当你决定用vector代替map或multimap时，事情会变得更有趣，因为vector必须容纳pair对象。毕竟，那是map和multimap所容纳的。但是要注意，如果你声明一个map<K, V>的对象（或者等价的multimap），保存在map中的元素类型是pair<const K, V>。如果要用vector模拟map或者multimap，你必须去掉const，因为当你对vector排序时，它的元素的值将会通过赋值移动，那意味着pair的两个组件都必须是可赋值的。当使用vector来模拟map<K, V>时，保存在vector中数据的类型将是pair<K, V>，而不是pair<const K, V>。

map和multimap以顺序的方式保存他们的元素，但用于排序目的时它们只作用于元素的key部分（pair的第一个组件），所以当排序vector时你必须做一样的事情。你需要为你的pair写一个自定义比较函数，因为pair的operator<作用于pair的两个组件。

有趣的是，你会需要第二个比较函数来进行查找。用来排序的比较函数将作用于两个pair对象，但是查找只用到key值。必须传给用于查找的比较函数一个key类型的对象（要查找的值）和一个pair（存储在vector中的一个pair）——两个不同的类型。还有一个附加的麻烦，你不会知道key还是pair是作为第一个实参传递的，所以你真的需要两个用于查找的比较函数：一个key值先传递，一个pair先传递。

这个例子演示了怎么把这些东西合在一起：

```
typedef pair<string, int> Data;                    // 在这个例子里
                                                   // "map"容纳的类型
class DataCompare {                                // 用于比较的类
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在下面
    }

    bool operator()(const Data& lhs,               // 用于查找的比较函数
                    const Data::first_type& k) const   // （形式1）
    {
        return keyLess(lhs.first, k);
    }

    bool operator()(const Data::first_type& k,     // 用于查找的比较函数
                    const Data& rhs) const     // （形式2）
    {
        return keyLessfk, rhs.first);
    }

private:
    bool keyLess(const Data::first_type& k1,       // “ 真的 ”
                 const Data::first_type& k2) const  // 比较函数
    {
        return k1 < k2;
    }
};
```

在这个例子中，我们假设有序vector将模拟map<string, int>。这段代码几乎是上面讨论的字面转换，除了存在成员函数keyLess。那个函数的存在是用来保证几个不同的operator()函数之间的一致性。每个这样的函数只是简单地比较两个key的值，所以我们把这个测试放在keyLess中并让operator()函数返回keyLess所做的事情，这比复制那个逻辑要好。这个软件工程中绝妙的动作增强了DataCompare的可维护性，但有一个小缺点，它提供了有不同参数类型的operator()函数，这将导致函数对象无法适配（看见条款40）。噢，好了。

把有序vector用作map本质上和用作set一样。唯一大的区别是必须把DataCompare对象用作比较函数：

```
vector<Data> vd;                                  // 代替map<string, int>
...                                               // 建立阶段：很多插入，
                                                  // 几乎没有查找
sort(vd.begin(), vd.end(), DataCompare());        // 结束建立阶段。（当
                                                  // 模拟multimap时，你
                                                  // 可能更喜欢用stable_sort
                                                  // 来代替；参见条款31。）
string s;                                         // 用于查找的值的对象
...                                               // 开始查找阶段
if (binary_search(vd.begin(), vd.end(), s,
                  DataCompare()))...              // 通过binary_search查找
vector<Data>::iterator i =
    lower_bound(vd.begin(), vd.end(), s,
                  DataCompare());                 // 在次通过lower_bound查找，
if (i != vd.end() && !DataCompare()(s, *i))...    // 条款45解释了
                                                  // “ !DataCompare()(s, *i) ”测试
pair<vector<Data>::iterator,
          vector<Data>::iterator> range =
               equal_range(vd.begin(), vd.end(), s,
                           DataCompare()); // 通过equal_range查找
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...
```

正如你所见，一旦你写了DataCompare，东西都很好的依序排列了。而一旦位置合适了，只要你的程序按照101页描述的阶段方式使用数据结构，它们往往比相应的使用真的map的设计运行得更快而且使用更少内存。如果你的程序不是按照阶段的方式操作数据结构，那么使用有序vector代替标准关联容器几乎可以确定是在浪费时间。

# 条款24：当关乎效率时应该在map::operator[]和map-insert之间仔细选择

让我们假设有一个支持默认构造函数以及从一个double构造和赋值的Widget类：

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

现在让我们假设我们想建立一个从int到Widget的map，而且我们想有初始化有特定值的映射。这可以简化为：

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

实际上，简化掉的唯一事情是忘了实际上的进行了什么操作。那很糟糕，因为实际进行的会造成一个相当大的性能冲击。

map的operator[]函数是个奇怪的东西。它与vector、deque和string的operator[]函数无关，也和内建的数组operator[]无关。相反，map::operator[]被设计为简化“添加或更新”功能。即，给定

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

这个表达式

```
```

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

检查键k是否已经在map里。如果不，就添加上，以v作为它的对应值。如果k已经在map里，它的关联值被更新成v。

这项工作的原理是operator[]返回一个与k关联的值对象的引用。然后v赋值给所引用（从operator[]返回的）的对象。当要更新一个已存在的键的关联值时很直接，因为已经有operator[]可以用来返回引用的值对象。但是如果k还不在map里，operator[]就没有可以引用的值对象。那样的话，它使用值类型的默认构造函数从头开始建立一个，然后operator[]返回这个新建立对象的引用。

让我们再次地看看原先例子的第一部分：

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

表达式m[1]是m.operator[](1)的简化，所以这是一个map::operator[]的调用。那个函数必须返回一个Widget的引用，因为m 的映射类型是Widget。在这里，m里面还没有任何东西，所以键1在map里没有入口。因此operator[]默认构造一个Widget来作为关联到1的值，然后返回到那个Widget的引用。最后，Widget成为赋值目标：被赋值的值是1.50。

换句话说，这个语句

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

功能上等价于这个：

```
typedef map<int, Widget> IntWidgetMap;                  // 方便的
                                                        // typedef
pair<IntWidgetMap::iterator, bool> result =                     // 用键1建立新
        m.insert(IntWidgetMap::value_type(1, Widget()));    // 映射入口
                                                        // 和一个默认构造的
                                                        // 值对象；
                                                        // 看下面对于
                                                        // value_type的
                                                        // 注释
result.first->second = 1.50;                                    // 赋值给
                                                        // 新构造的
```

```
                                        // 值类型
```

现在已经很清楚为什么这种方法可能降低性能了。我们先默认构造一个Widget，然后我们立即赋给它新值。如果用想要的值构造Widget比默认构造Widget然后进行赋值显然更高效，我们就应该用直截了当的insert调用来替换operator[]的使用（包括它的构造加赋值）：

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

这与上面的那些代码有相同的最终效果，除了它通常节省了三次函数调用：一个建立临时的默认构造Widget对象，一个销毁那个临时的对象和一个对Widget的赋值操作。那些函数调用越昂贵，你通过使用map-insert代替map::operator[]就能节省越多。

上面的代码利用了每个标准容器都提供的value_type typedef。这typedef没有什么特别重要的，但对于map和multimap（以及非标准容器的hash_map和hash_multimap——参见条款25），记住它是很重要的，容器元素的类型总是某种pair。

我早先谈及的operator[]被设计为简化“添加或更新”功能，而且现在我们理解了当“增加”被执行时，insert比operator[]更高效。当我们做更新时，情形正好相反，也就是，当一个等价的键（参见条款19）这已经在map里时。为了看出为什么情况是这样，看看我们的更新选项：

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

语法本身也许会让你信服地支持operator[]，但在这里我们关注于效率，所以我们将忽略它。insert的调用需要IntWidgetMap::value_type类型的实参（即pair<int, Widget>），所以当我们调用insert时，我们必须构造和析构一个那种类型的对象。那耗费了一对构造函数和析构函数，也会造成一个Widget的构造和析构，因为pair<int, Widget>本身包含了一个Widget对象，operator[]没有使用pair对象，所以没有构造和析构pair和Widget。

因此出于对效率的考虑，当给map添加一个元素时，我们断定insert比operator[]好；而从效率和美学考虑，当更新已经在map里的元素值时operator[]更好。如果STL提供一个两全其美的函数，即，在句法上吸引人的包中的高效的“添加或更新”功能。例如，很容易可以想象一个像这样的调用接口：

```
iterator affectedPair =                    // 如果键k不再map m中；高效地
    efficientAddOrUpdate(m, k, v);         // 把pair(k, v)添加到m中；否则
                                           // 高效地把和k关联
                                           // 的值更新为v。返回一个
                                           // 指向添加或修改的
                                           // pair的迭代器
```

但是，在STL内没有像这样的函数，正如下面的代码所演示的，自己写一个并不难。那些注释总结了正在做什么，而且随后的段落也提供了一些附加的解释。

```
template<typename MapType,                 // map的类型
         typename KeyArgType,              // KeyArgType和ValueArgtype
         typename ValueArgtype>            // 是类型参数
                                           // 的原因请看下面
typename MapType::iterator
    efficientAddOrUpdate(MapType& m,
                         const KeyArgType& k,
                         const ValueArgtype& v)
{
    typename MapType::iterator Ib =        // 找到k在或应该在哪里；
        m.lower_bound(k);                  // 为什么这里
                                           // 需要“ typename ”
                                           // 参见第7页
    if(Ib != m.end() &&                    // 如果Ib指向一个pair
        !(m.key_comp()(k, Ib->first))) {   // 它的键等价于k...
            Ib->second = v;                // 更新这个pair的值
            return Ib;                     // 并返回指向pair的
        }                                  // 迭代器
        else{
            typedef typename MapType::value_type MVT;
            return m.insert(Ib, MVT(k, v)); // 把pair(k, v)添加到m并
        }                                  // 返回指向新map元素的
}                                          // 迭代器
```

执行一个高效的增加或更新，我们需要能找出k的值是否在map中；如果是这样，那它在哪里；如果不是，它该被插入哪里。这个工作是为low_bound（参见条款45）量身定做的，所以在这里我们调用那个函数。确定

lower_bound是否用我们要寻找的键找到了一个元素，我们对后半部分进行一个等价测试（参见条款19），一定要对map使用正确的比较函数：通过map::key_comp提供的比较函数。等价测试的结果告诉我们应该进行增加还是更新。

如果是更新，代码很直截了当。插入分支更有趣，因为它使用了insert的“提示”形式。结构m.insert(Ib, MVT(k，v))“提示”了Ib鉴别出了键等价于k的新元素正确的插入位置，而且标准保证如果提示正确，那么插入将在分摊的常数时间内发生，而不是对数时间。在efficientAddOrUpdate里，我们知道Ib鉴别出了适当的插入位置，因此insert的调用被保证为是一次常数时间操作。

这个实现的一个有趣方面是KeyArgType和ValueArgType不必是储存在map里的类型。它们只需要*可以转换*到储存在map里的类型。一个可选的方法是去掉类型参数KeyArgType和ValueArgType，改为使用MapType::key_type和MapType::mapped_type。但是，如果我们那么做，在调用时我们可能强迫发生不必要的类型转换。例如，再次看看我们在本条款的例子里使用的map定义：

```
map<int, Widget> m;                    // 同前
```

别忘了Widget接受从一个double赋值：

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

现在考虑efficientAddOrUpdate的调用：

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

假设是一次更新操作，即，m已经包含键是10的元素。那样的话，上面的模板推断出ValueArgType是double，函数体直接把1.5作为double赋给与10相关的那个Widget。那是通过调用Widget::operator(double)完成的。如果我们用了MapType::mapped_type作为efficientAddOrUpdate的第3个参数的类型，在调用时我们得把1.5转化成一个Widget，那样的话我们就得花费本来不需要的一次Widget构造（以及随后的析构）。

efficientAddOrUpdate实现中的细节可能很有趣，但它们没有本条款的要点重要，也就是当关乎效率时应该在map::operator[]和map-insert之间仔细选择。如果你要更新已存在的map元素，operator[]更好，但如果你要增加一个新元素，insert则有优势。

条款24：当关乎效率时应该在map::operator[]和map-insert之间仔细选择

# 条款25：熟悉非标准散列容器

STL程序员一般用不了多久就开始惊讶，“vector、list、map，很好，但是散列（hash）表在哪里"？唉，在标准C++库里没有任何散列表。每个人都同意这是个不幸，但是标准委员会觉得需要加给他们的工作可能会过渡地推迟标准的完成。可以肯定标准的下一个版本将包含散列表，但是目前，STL没有散列的东西。

但是如果你喜欢散列表，那么振作起来。你不需要放弃或自己做一个。兼容STL的散列关联容器可以从多个来源获得，而且它们甚至有*事实上的*标准名字：hash_set、hash_multiset、hash_map和hash_multimap。（译注：在C++标准委员会的议案中，散列容器的名字是unordered_set、unordered_multiset、unordered_map和unordered_multimap。恰好是为了避开现存的hash_*名字。）

在这些公用的名字后面，有不同的实现，呃，不同。它们在接口、能力、内在数据结构和支持操作的相关效率方面不同。写出使用散列表的适度可移植的代码仍然是可能的，但不像如果散列容器被标准化一样容易。（现在你知道为什么标准很重要了吧。）

对于几个可得到的散列容器的实现，最常见的两个来自SGI（参见条款50）和Dinkumware（参见附录B），所以在下文里，我把自己限制在来自这些厂商的散列容器的设计。STLport（再次参见条款50）也提供散列容器，但是STLport的散列容器是基于来自SGI的。为了本条款的目的，假设我写的任何关于SGI散列容器的东西也适用于STLport散列容器。

散列容器是关联容器，因此你不该惊讶，正如所有关联容器，它们需要知道储存在容器中的对象类型，用于这些对象的比较函数，以及用于这些对象的分配器。另外，散列容器需要散列函数的说明。下面是散列容器声明：

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

这非常接近于散列容器的SGI声明，主要差别是SGI为HashFunction和CompareFunction提供了默认类型。hash_set的SGI声明看起来基本上像这样（我为了演示的目的已经稍微调整了一下）：

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

SGI设计的一个值得注意的方面是使用equal_to作为默认比较函数。这违背标准关联容器的约定——默认比较函数是less。这个设计结果不仅仅表示简单地改变默认比较函数。SGI的散列容器确定在一个散列容器中的两个对象是否有相同的值是通过*相*等测试，而不是等价（参见条款19）。对于散列容器来说，这不是一个不合理的决定，因为散列关联容器，不像它们在标准中的（通常基于树）兄弟，不需要保持有序。

Dinkumware设计的散列容器采取一些不同的策略。它仍然允许你指定对象类型、散列函数类型、比较函数类型和分配器类型，但是它把默认的散列和比较函数移进一个单独的类似特性的叫做hash_compare的类，而且它把hash_compare作为容器模板的HashingInfo实参的默认值。（如果你不熟悉“特性”类的概念，打开一本好的STL参考，比如Josuttis的《C++ Standard Library》[3]并学习char_traits和iterator_traits模板的动机和实现。）

例如，这是Dinkumware的hash_set声明（再次为演示而调整过）：

```
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;
```

这种接口设计有趣的地方是HashingInfo的使用。容器的散列和比较函数储存在那里，但HashingInfo类型也容纳了控制表中桶（bucket）最小数量，以及容器元素对桶的最大允许比率的枚举。当这比率被超过时，表中桶的数量就增加，而表中的一些元素需要重新散列。（SGI提供具有类似控制表中桶和表中元素对桶比率的成员函数。）（译注：如果你不知道桶和散列表的原理，那么看看数据结构的书中关于散列表的部分。）

做了一些为了演示的调整之后，hash_compare（HashingInfo的默认值）看起来或多或少像这样：

```
template<typename T, typename CompareFunction = less<T> >
class hash_compare {
public:
```

```
    enum {
        bucket_size = 4,              // 元素对桶的最大比率
        min_buckets = 8           // 桶的最小数量
    };

size_t operator()(const T&) const;              // 散列函数
bool operator()(const T&,                    // 比较函数
                const T&) const;

...                                 // 忽略一些东西，包括
                                    // CompareFunction的使用
}
```

重载operator()（在这里是实现散列和比较函数）是比你可以想象的更经常出现的一个策略。对于这一想法的另一个应用，参见条款23。

Dinkumware设计允许你写你自己的类似hash_compare的类（也许通过从hash_compare本身派生而来），而且只要你的类定义了bucket_size、min_buckets、两个operator()函数（一个带有一个实参，一个带有两个），加上我已经省去的一些东西，你就能使用它来控制Dinkumware的hash_set或hash_multiset的配置和行为。hash_map和hash_multimap的配置控制也相似。

注意不管是SGI还是Dinkumware的设计，你都能把全部决策留给实现，并仅仅写像这样的东西：

```
hash_set<int> intTable;      // 建立一个int的散列表
```

要让这个可以编译，散列表必须容纳一个整数类型（例如int），因为默认散列函数一般局限于整数类型。（SGI的默认散列函数稍微灵活一些。条款50将告诉你在哪里可以找到全部细节。）

在后端，SGI和Dinkumware的实现方法非常不同。SGI利用常用的一个元素的单链表的指针数组（桶）组成的开放散列法。Dinkumware也利用了开放散列法，但是它的设计是基于一种新颖的数据结构——由迭代器（本质是桶）的数组组成的元素双向链表，迭代器的相邻对表示在每个桶里元素的范围。（细节可以参考Plauger相关主题的专栏，《Hash Tables》[16]。）

作为这些实现的用户，你可能会对SGI实现在单链表中储存表的元素，而Dinkumware实现使用一个双向链表的事实感兴趣。差别是值得注意的，因为它影响了用于两个实现的迭代器种类。SGI的散列容器提供了前向迭代器，因此你得放弃进行反向迭代的能力：在SGI的散列容器中没有rbegin或者rend成员函数。用于Dinkumware散列容器的迭代器是双向的，所以它们可以提供前向和反向遍历。 在内存使用量方面，SGI的设

计比Dinkumware的节俭一点点。

哪个设计最有利于你和你的程序？我不可能知道。只有你能确定，而且本条款并不试图给你足以得出一个合理结论的信息。取而代之的是，本条款的目标是让你知道虽然STL本身缺乏散列容器，兼容STL的散列容器（有不同的接口、能力和行为权衡）不难得到。就SGI和STLport的实现而言，你甚至可以免费得到它们，因为它们可以自由下载。

# 迭代器

乍看来，迭代器似乎很直观。但凑近了看，你会发现标准STL容器提供了四种不同的迭代器：iterator、const_iterator、reverse_iterator和const_reverse_iterator。很快你会注意到在这四种类型中，容器的insert和erase的某些形式只接受其中一种。那是问题的开始。为什么有四种迭代器？它们之间的关系是什么？它们可以互相转化吗？在调用算法和STL实用函数时不同类型可以混合使用吗？这些类型是怎么关联到容器和它们的成员函数的？

本章回答了这些问题，也介绍了一个比通常更值得注意的迭代器类型：istreambuf_iterator。如果你喜欢STL，但你不喜欢读取字符流时istream_iterator的性能，istreambuf_iterator可能就是你正在寻找的工具。

# 条款26：尽量用iterator代替const_iterator，reverse_iterator和const_reverse_iterator

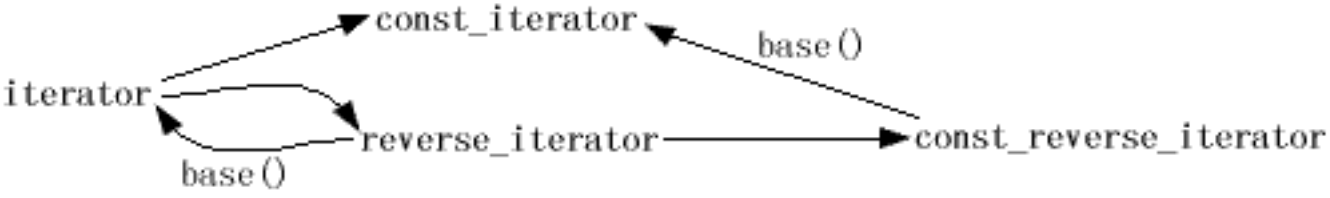
正如你所知的，每个标准容器类都提供四种迭代器类型。对于container<T>而言，iterator的作用相当于T*，而const_iterator则相当于const T*（你可能也见过T const*这样的写法：它们意思一样[1]）。增加一个iterator或者const_iterator可以在一个从容器开头趋向尾部的遍历中让你移动到容器的下一个元素。reverse_iterator与const_reverse_iterator同样相当于对应的T*和const T*，所不同的是，增加reverse_iterator或者const_reverse_iterator会在从尾到头的遍历中让你移动到容器的下一个元素。

让我向你演示两个东西。第一，看看vector<T>的insert和erase的样式：

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

每个标准容器都包含了和这差不多的函数，虽然返回类型因容器类型的不同而不同。需要注意的是：这些方法只接受iterator类型的参数，而不是const_iterator、reverse_iterator或const_reverse_iterator。总是iterator。虽然容器类支持四种迭代器类型，但其中的一种类型有着其他所没有的特权。那就是iterator，iterator比较特殊[2]。

我要让你看的第二个东西是这张图，它显示了几种迭代器之间存在的转换关系：




图中显示了从iterator到const_iterator、从iterator到reverse_iterator和从reverse_iterator到const_reverse_iterator可以进行隐式转换。并且，reverse_iterator可以通过调用其base成员函数转换为iterator。const_reverse_iterator也可以类似地通过base转换成为const_iterator。一个图中无法显示的事实是：通过base得到的也许并非你所期待的iterator。我们将会在条款28中详细讨论这一点。

你应该发现了没有办法从一个const_iterator转换得到一个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有一个const_iterator或者const_reverse_iterator，你会发现让南让它们和容器的一些成员函数合作。那些成员函数要求iterator，而你无法从const迭代器类型反过来得到iterator。当你需要指出插入位置或删除的元素时，const迭代器几乎没有用。

千万不要傻乎乎的宣称const迭代器一无是处。不，它们可以与算法默契配合，因为算法通常并不关心迭代器是什么类型，只要是适当的种类就可以了，很多容器的成员方法也接受const迭代器。只有insert和erase的一些形式有些吹毛求疵。

我写的是如果你要指出插入的位置或删除的元素时，const迭代器“几乎”没有用。这暗示了并不是完全没用。那是真的。如果你找到了一个方法可以从const_iterator或const_reverse_iterator得到一个iterator，那么它们就有用。那经常是有可能的。但这种方法并不总是行得通，而且就算可行，完成的方法也很不直观，也很缺乏效率。这个主题将足够充实一个它自己的条款，所以如果你对这个细节感兴趣的话，请转到条款27。现在，我们已经有足够的理由相信应该尽量使用iterator取代const或者reverse类型的迭代器：

- insert和erase的一些版本要求iterator。如果你需要调用这些函数，你就必须产生iterator，而不能用const或reverse iterators。
- 不可能把const_iterator隐式转换成iterator，我们将会在条款27中讨论从一个const_iterator产生一个iterator的技术并不普遍适用，而且不保证高效。
- 从reverse_iterator转换而来的iterator在转换之后可能需要相应的调整，在条款28中我们会讨论何时需要调整以及调整的原因。

所有这些东西联合起来就能看出，如果你尽量使用iterator代替const或reverse类型的迭代器，可以使得容器的使用更简单，更高效而且可以避免潜在的bug。

事实上，你可能会更多地面临在iterator与const_iterator之间的选择.iterator与reverse_iterator之间的选择显而易见——依赖于从前到后或从后到前的遍历。你可以选择你需要的一种，而且即使你选择了reverse_iterator，当你要调用需要iterator的容器成员函数时，你仍然可以通过base得到相应的iterator（可能需要一些调整，参见条款28）。

当在iterator和const_iterator之间作选择的时候，你有更充分的理由选择iterator，即使const_iterator同样可行而且即使你并不需要调用容器类的任何成员函数。其中的令人讨厌的原因包括iterator与const_iterator之间的比较。我希望我们都可以赞成这是合理的代码：

```
typedef deque<int>  IntDeque;              // typedef可以极大地简化
typedef IntDeque::iterator Iter;              // STL容器类和iterator
typedef IntDeque::const_iterator ConstIter;         // 的操作。
```

```
Iter  i;
ConstIter ci;
...                              // 同一个容器
if (i == ci) ...                          // 比较iterator和const_iterator
```

我们所做的只是同一个容器中两个迭代器之间的比较，这是STL中最基本的动作。唯一的变化是等号的一边的类型是iterator，而另一边的类型是const_iterator。这应该不是问题，因为iterator应该在比较之前隐式的转换成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。但对于其它一些实现，这段代码甚至无法通过编译。原因在于，这些实现将const_iterator的operator==作为const_iterator的一个成员函数而不是非成员函数。而问题的解决之道显得非常有趣：只要像这样交换两个iterator的位置：

```
if (ci == i)...                    // 当上面比较无法
                                 // 通过编译时的解决方法
```

不仅是比较是否相等，只要你在同一个表达式中混用iterator和const_iterator（或者reverse_iterator和const_reverse_iterator），这样的问题就可能会出现。比如，当你试图在两个随机存取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，你的（正确的）代码可能会被（错误地）拒绝。本例中最简单的解决方法是通过一个（安全的）映射把iterator转换为const_iterator：

```
if (static_cast<ConstIter>(i) - ci >= 3) ...       // 当上面的代码无法
                                 // 通过编译时的解决方法
```

避免这类问题的最简单的方法是减少混用不同类型的迭代器的机会，换句话说，又回到了尽量用iterator代替const_iterator。从常量正确性的角度来看（一个固然有价值的角度），仅仅为了避免一些潜在的STL实现的弊端（而且，这些弊端都有变通办法）而抛弃const_iterator显得有欠公允。但综合考虑到iterator与一些容器类成员函数的粘连关系，从实践得出const_iterator没有iterator好用的结论是很难避免的。更何况，有时并不值得卷入const_iterator的麻烦中去。

---

[1] 关于这个主题的完整的文章，请参考1999年2月《Embedded Systems Programming》上刊登的《const T vs. T const》，作者是Dan Saks。

[2] “iterator比较特殊”的原因并不清楚。HP的最早的STL实现包含了带有iterator参数的insert和erase，而这个设计问题在标准化的过程中并没有重新考虑。但是在以后，这可能会改变，因为程序库工作组发布了#180记录，说明了“这个问题在下一个标准版本中会作为综合回顾的const问题而被考虑”（C++程序库问题可以从http://anubis.dkuug.dk/jtc1/sc22/wg21/docs/lwg-defects.html看到）

# 条款27：用distance和advance把const_iterator转化成iterator

条款26中指出有些容器成员函数只接受iterator作为参数，而不是const_iterator。那么，如果你只有一个const_iterator，而你要在它所指向的容器位置上插入新元素呢？也就是如何把const_iterator转化为iterator呢？因为正如条款26所解释的，并不存在从const_iterator到iterator之间的隐式转换，所以你必须成为这次行动的主角。

我知道你在想什么。你正在想，“每当无路可走的时候，就举起大锤！”。在C++的世界里，你的意思只能是：映射。这种想法很可耻。真不知道你是哪儿学来的。

让我们面对困扰在你面前的问题。看看当你把一个const_iterator映射为iterator时会发生什么：

```
typedef deque<int> IntDeque;          // 方便的typedef
typedef IntDeque::iterator Iter;
typedef IntDeque::const_iterator ConstIter;
ConstIter ci;                         // ci是const_iterator
...
Iter i(ci);                           // 错误！没有从const_iterator
                                      // 到iterator隐式转换的途径
Iter i(const_cast<Iter>(ci));         // 仍是个错误！不能从const_iterator
                                      // 映射为iterator！
```

这里只是以deque为例，但是用其它容器类——list、set、multiset、map、multimap甚至条款25描述的散列表容器[1]——的结果一样。使用映射的行*也许*在vector或string的代码时能够编译，但这是我们马上要讨论的非常特殊的情形。

包含映射的代码不能通过编译的原因在于，对于这些容器而言，iterator和const_iterator是完全不同的类。它们之间并不比string和complex<float>具有更多的血缘关系。在两个毫无关联的类之间进行const_cast映射是荒谬的，所以reinterpret_cast、static_cast甚至C风格的映射也会导致同样的结果。

唉，不能编译的代码对于vector和string容器来说也许能够通过编译。那是因为通常情况下大多数实现都会采用真实的指针作为那些容器的迭代器。就这种实现而言，vector<T>::iterator是T*的typedef，而vector<T>::const_iterator是const T*的typedef，string::iterator是char*的typedef，而string::const_iterator是const char*的

typedef。在这种实现的情况下，用const_cast把const_iterator映射成iterator当然可以编译而且没有问题，因为const_iterator与iterator之间的const_cast映射被最终解释成const T*到T*的映射。但是，即使是在这种实现中，reverse_iterator和const_reverse_iterator也是真正的类，所以你仍然不能直接用const_cast把const_reverse_iterator映射成reverse_iterator。而且，正如条款50解释的，这些实现通常只会在Release模式时才使用指针表示vector和string的迭代器[2]。所有这些事实表明，把const迭代器映射为迭代器是病态的，即使是对vector和string来说也时，因为移植性很值得怀疑。

如果你得到一个const_iterator并且可以访问它所指向的容器，那么有一种安全的、可移植的方法获取它所对应的iterator[3]，而且，用不着陷入类型系统的转换。下面是解决思路的本质，虽然在它编译前还要稍作修改：

```
typedef deque<int> IntDeque;        // 和以前一样
typedef IntDeque::iterator Iter;
typedef IntDeque::const_iterator ConstIter;

IntDeque  d;
ConstIter ci;
...                         // 让ci指向d
Iter i(d.begin());                  // 初始化i为d.begin()
advance(i, distance(i, ci));        // 把i移到指向ci位置
                            // （但请留意下面关于为什么
                            // 在它编译前要调整的原因）
```

这种方法看上去非常简单，直截了当，也很让人吃惊吧。要得到与const_iterator指向同一位置的iterator，首先将iterator指向容器的起始位置，然后把它向前移到和const_iterator距离容器起始位置的偏移量一样的位置即可！这个任务得到了两个函数模板advance和distance的帮助，它们都在<iterator>中声明。distance返回两个指向同一个容器的iterator之间的距离；advance则用于将一个iterator移动指定的距离。如果i和ci指向同一个容器，那么表达式advance(i, distance(i, ci))会将i移动到与ci相同的位置上。

如果这段代码能够通过编译，它就能完成这种转换任务。但似乎事情并不那么顺利。想知道为什么，先来看看distance的定义：

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

不要被这个函数的长达56个字符的返回类型卡住，也不用理会difference_type是什么东西。取而代之的是，把

注意力集中在参数的类型InputIterator：

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

当遇到distance调用时，你的编译器需要根据使用的实参类型推断出InputIterator的类型。再来看看我所说的不太正确的distance调用：

```
advance(i, distance(i, ci));              // 调整i，指向ci位置
```

有两个参数传递给distance，i和ci。i的类型是Iter，即deque<int>::iterator的typedef。对编译器来说，这表明调用distance的InputIterator是deque<int>::iterator。但ci是ConstIter，即deque<int>::const_iterator的typedef。这表明那个InputIterator是deque<int>::const_iterator。InputIterator不可能同时有两种不同的类型，所以调用distance失败。一般会造成一些冗长的出错信息，可能会也可能不会说明是编译器无法得出InputIterator是什么类型。

要顺利地调用distance，你需要排除歧义。最简单的办法就是显式的指明distance调用的模板参数类型，从而避免编译器自己得出它们的类型：

```
advance(i, distance<ConstIter>(i, ci));
```

我们现在知道了怎么通过advance和distance获取const_iterator相应的iterator了。但另一个我们现在一直避开却很值的考虑的实际问题是：这个技巧的效率如何？答案很简单。取决于你所转换的究竟是什么样的迭代器。对于随机访问的迭代器（比如vector、string和deque的）而言，这是常数时间的操作。对于双向迭代器（也就是，所有其它容器和包括散列容器的一些实现[4]（参见条款25））而言，这是线性时间的操作。

因为它可能花费线性时间的代价来产生一个和const_iterator等价的iterator，并且因为如果不能访问const_iterator所属的容器这个操作就无法完成。从这个角度出发，也许你需要重新审视你从const_iterator产生iterator的设计。事实上那样的考虑帮助激发了条款26，它建议你当处理容器时尽量用iterator代替const和reverse迭代器。

---

[1] 两个最常见的基于散列表的STL容器实现来自于Dinkumware和SGI。你可以从P.J.Plauger 1998年11月份的CUJ专栏《Hash Tables》中找到一个Dinkumware方法的概览。我所知道的唯一的SGI的实现方法的概览来自

Effective STL的条款25，但它的接口的描述在SGI的STL网站。

[2] 当使用STLport的调试模式时会出现这种情况。你可以从STLport的网站上了解到STLport和它的调试模式。

[3] 我后来发现在这里描述的这种方法可能在使用引用计数的string实现上失败。细节请参考在http://www.aristeia.com/BookErrata/estl1e-errata.html的jep的 8/22/01关于本书第121页的注释。

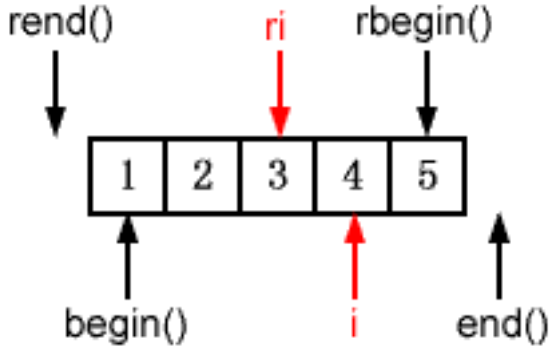
[4] Dinkumware的散列容器提供了双向的迭代器。SGI的、STLport的和Metrowerks的散列容器只提供了前向迭代器。

# 条款28：了解如何通过reverse_iterator的base得到iterator

调用reverse_iterator的base成员函数可以产生“对应的”iterator，但这句话有些辞不达意。举个例子，看一下这段代码，我们首先把从数字1-5放进一个vector中，然后产生一个指向3的reverse_iterator，并且通过reverse_iterator的base初始化一个iterator：

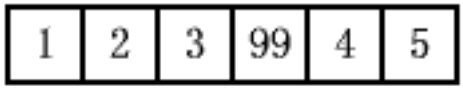
```
vector<int> v;
v.reserve(5);                    // 参见条款14
for(int i = 0；i < 5; ++ i) {          // 向vector插入1到5
    v.push_back(i);
}
vector<int>::reverse_iterator ri =          // 使ri指向3
    find(v.rbegin(), v.rend(), 3);
vector<int>::iterator i(ri.base());          // 使i和ri的base一样
```

执行上述代码后，可以想到产生的结果就像这样：




这张图很好，显示了reverse_iterator和它对应的base iterator之间特有的偏移量，就像rbegin()和rend()与相关的begin()和end()一样，但并没有说出了所有你需要知道的东西。特别是，它并没有解释怎样在ri上实现你在i上想要完成的操作。正如条款26解释的，有些容器的成员函数只接受iterator类型的参数，所以如果你想要在ri所指的位置插入一个新元素，你不能直接这么做，因为vector的insert函数不接受reverse_iterator。如果你想要删除ri所指位置上的元素也会有同样的问题。erase成员函数会拒绝reverse_iterator，坚持要求iterator。为了完成删除和一些形式的插入操作，你必须先通过base函数将reverse_iterator转换成iterator，然后用iterator来完成工作。

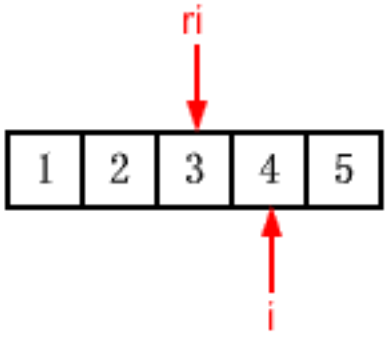
先让我们假设你要在ri指出的位置上把一个新元素插入v。特别的，我们假设你要插入的值是99。记住ri在上图中遍历的顺序是自右向左，而且插入操作会将新元素插入到ri位置，并且将原先ri位置的元素移到遍历过程的“ 下一个 ”位置，我们认为3应该出现在99的左侧。插入操作之后，v看起来像这样：




当然，我们不能用ri来指定插入的地方，因为它不是一个iterator。我们必须用i来代替。如上所述，当ri指向3时，i（就是ri.base()）指向4。如果我们用ri来指定插入位置，那么用i指向插入位置，那个假设就是正确的。结论呢？

- 要实现在一个reverse_iterator ri指出的位置上插入新元素，在ri.base()指向的位置插入就行了。对于insert操作而言，ri和ri.base()是等价的，而且ri.base()真的是ri对应的iterator。

现在再来考虑删除元素的情况。回顾一下最初的vector（也就是在插入99之前）ri与i的关系：




如果你要删除ri指向的元素，你不能直接使用i了，因为i与ri不是指向同一个元素。因此，你要删除的是i的*前*一个元素。

- 要实现在一个reverse_iterator ri指出的位置上删除元素，就应该删除ri.base()的前一个元素。对于删除操作而言，ri和ri.base()并不等价，而且ri.base()不是ri对应的iterator。

我们还是有必要看看删除操作的代码，因为它还挺令人惊讶的。

```
vector<int> v;
...                                 // 向v插入1到5，同上
vecot<int>::reverse_iterator ri =
      find(v.rbegin(), v.rend(), 3);       // 同上，ri指向3
v.erase(--ri.base());                  // 尝试删除ri.base()前面的元素；
```

```
                                    // 对于vector，一般来说编译不通过
```

这个设计并不存在什么问题。表达式--ri.base()确实能够指出我们需要删除的元素。而且，它们能够处理除了vector和string之外的其他所有容器。它可能也能处理vector和string，但对于大多数vector和string的实现，它无法通过编译。在这样的实现下，iterator（和const_iterator）会采用内建的指针来实现，所以ri.base()的结果是一个指针。

C和C++都规定了不能直接修改函数返回的指针，所以在string和vector的迭代器是指针的STL平台上，像--ri.base()这样的表达式无法通过编译。要移植从一个由reverse_iterator指出的位置删除元素时，你应该尽量避免修改base的返回值。没问题。如果你不能减少调用base的返回值，只需要先增加reverse_iterator的值，然后再调用base！

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

因为这个方法适用于所有的标准容器，这是删除一个由reverse_iterator指出的元素时首选的技巧。

现在已经很清楚了，reverse_iterator的base成员函数返回一个“对应的”iterator的说法并不准确。对于插入操作而言，的确如此；但是对于删除操作，并非如此。当需要把reverse_iterator转换成iterator的时候，有一点非常重要的是你必须知道你准备怎么处理返回的iterator，因为只有这样你才能决定你得到的iterator是否是你需要的。

# 条款29：需要一个一个字符输入时考虑使用istreambuf_iterator

假设我们要把一个文本文件拷贝到一个字符串对象中。似乎可以用一种很有道理的方法完成：

```
ifstream inputFile("interestingData.txt");
string fileData((istream_iterator<char>(inputFile)),        // 把inputFile读入
                istream_iterator<char>());          // fileData；关于为什么
                                                    // 它不是很正确请看下文
                                                    // 关于这个语法的警告
                                                    // 参见条款6
```

很快你就会发现这种方法无法把文件中的空格拷贝到字符串中。那是因为istream_iterators使用operator>>函数来进行真的读取，而且operator>>函数在默认情况下忽略空格。

假如你想保留空格，你要的做就是覆盖默认情况。只要清除输入流的skipws标志就行了：

```
ifstream inputFile("interestingData.txt");
inputFile.unset(ios::skipws);                   // 关闭inputFile的
                                                // 忽略空格标志
string fileData((istream_iterator<char>(inputFile)), istream_iterator<char>());
```

现在inputFile中的所有字符都拷贝到fileData中了。

唉，你会发现它们的拷贝速度不像你想象的那么快。istream_iterators所依靠的operator>>函数进行的是格式化输入，这意味着每次你调用的时候它们都必须做大量工作。它们必须建立和销毁岗哨（sentry）对象（为每个operator>>调用进行建立和清除活动的特殊的iostream对象），它们必须检查可能影响它们行为的流标志（比如skipws），它们必须进行全面的读取错误检查，而且如果它们遇到问题，它们必须检查流的异常掩码来决定是否该抛出一个异常。如果进行格式化输入，那些都是重要的活动，但如果你需要的只是从输入流中抓取下一个字符，那就过度了。

一个更高效的方法是使用STL最好的秘密武器之一：istreambuf_iterators。你可以像istream_iterator一样使用istreambuf_iterator，但istream_iterator<char>对象使用operator>>来从输入流中读取单个字符。istreambuf_iterator<char>对象进入流的缓冲区并直接读取下一个字符。（更明确地说，一个

istreambuf_iterator<char> 对象从一个istream s中读取会调用s.rdbuf()->sgetc()来读s的下一个字符。）把我们的文件读取代码改为使用istreambuf_iterator相当简单，大多数Visual Basic程序员都可以在两次尝试内做对：

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

注意这里不需要“ unset ”skipws标志，istreambuf_iterator不忽略任何字符。它们只抓取流缓冲区的下一个字符。

相对于istream_iterator，它们抓取得更快——在我进行的简单测试中能快40%，如果你的结果不同也不用惊奇。如果随时间流逝，速度优势不断增加也不必奇怪，因为istreambuf_iterator存在于STL的一个不常访问的角落，所以实现还没有花很多时间来优化。比如，在我用过的一个实现中，istreambuf_iterator在我的主要测试中只比istream_iterator快了大约5%。那样的实现显然还有很多余地来优化它们的istreambuf_iterator实现。如果你需要一个一个地读取流中的字符，你不需要格式化输入的威力，你关心的是它们花多少时间来读取流，和明显的性能提高相比，为每个迭代器多键入三个字符的代价是微弱的。对于无格式的一个一个字符输入，你总是应该考虑使用istreambuf_iterator。

当你了解它之后，你也应该考虑把ostreambuf_iterator用于相应的无格式一个一个字符输出的作。它们没有了ostream_iterator的开销（和灵活性），所以它们通常也做得更好。

# 算法

我注意到从第一章开始，容器就占据了STL喝彩声中最大的一份。在某种意义上，这是可以理解的。容器有着非凡的造诣，它们使大批C++程序员每天的基本生活变得简单。尽管如此，STL算法的权利也很重要，一样有能力减轻程序员的负担。事实上，有超过100个算法，很容易证明比起容器，它们提供给程序员更精巧的工具集（起码一样强）。也许它们的数量是一部分问题。搞清八种截然不同的容器类型明显比记住70个算法的名字并且弄明白哪个做了什么要容易。

在本章，我有两个主要的目标。第一，我要通过向你演示一些比较少见的算法怎么使你的生活变得简单来向你介绍它们。放心，我不会用一堆需要记忆的名字来惩罚你。我在这一章演示算法是因为它们能解决常见问题，比如进行忽略大小写的字符串比较，在容器中高效地查找n个最想要的对象，统计区间中所有对象的特性，还实现了copy_if（一个来自最初的HP STL的算法，但在标准化过程中去掉了）的行为。

我的第二个目标是向你演示怎么避免算法的常见用法问题。例如，你不能调用remove或它的兄弟remove_if和unique，除非你*真的*知道这些算法做（和不做）什么。特别是当你删除的区间容纳的是指针时。同样，很多算法只和有序区间配合，所以你需要知道它们是什么和为什么它们会有这个限制。最后，最常见的算法相关错误之一是让一个算法把它的结果写向一个不存在的位置，所以我解释了这个谬论是怎么发生的和怎么确认你没有受此折磨。

本章的结束后，你可能不会把算法和容器放在同样的高度，但我希望你会比以前更关注它们。

# 条款30：确保目标区间足够大

STL容器在被添加时（通过insert、push_front、push_back等）自动扩展它们自己来容纳新对象。这工作的很好，有些程序员因为这个信仰而被麻痹，认为他们不必担心要为容器中的对象腾出空间，因为容器自己可以照顾好这些。如果是那样就好了！当程序员想向容器中插入对象但并没有告诉STL他们所想的时，问题出现了。这是一个常见的可以自我表现的方法：

```
int transmogrify(int x);                        // 这个函数从x
                                                // 产生一些新值
vector<int> values;
...                                             // 把数据放入values
vector<int> results;                            // 把transmogrify应用于
transform(values.begin(), values.end(),         // values中的每个对象
          results.end(),                        // 把这个返回的values
          transmogrify);                        // 附加到results
                                                // 这段代码有bug！
```

在本例中，transform被告知它的目的区间是从results.end()开始的，所以那就是开始写在values的每个元素上调用transmogrify的结果的地方。就像所有使用目标区间的算法，transform通过对目标区间的元素赋值的方法写入结果，transform会把transmogrify应用于values[0]并把结果赋给*results.end()。然后它会把transmogrify应用于value[1]并把结果赋给*(results.end()+1)。那只能带来灾难，因为在*results.end()*没有对象*，*(results.end()+1)也没有！调用transform是错误的，因为它会给不存在的对象赋值。（条款50解释了STL的一个调试实现怎么在运行期检测这个问题。）犯了这种错误的程序员几乎总是以为他们调用算法的结果能插入目标容器。如果那是你想要发生的，你就必须说出来。STL是一个库，不是一个精神。在本例中，说“请把transform的结果放入叫做results容器的结尾”的方式是调用back_inserter来产生指定目标区间起点的迭代器：

```
vector<int> results;                            // 把transmogrify应用于
transform(values.begin(), values.end(),         // values中的每个对象，
          back_inserter(results),               // 在results的结尾
          transmogrify);                        // 插入返回的values
```

在内部，back_inserter返回的迭代器会调用push_back，所以你可以在任何提供push_back的容器上使用back_inserter（也就是任何标准序列容器：vector、string、deque和list）。如果你想让一个算法在容器的前端插

入东西，你可以使用front_inserter。在内部，front_inserter利用了push_front，所以front_insert只和提供那个成员函数的容器配合（也就是deque和list）：

```
...                                    // 同上
list<int> results;                             // results现在是list
transform(values.begin(), values.end(),        // 在results前端
          front_inserter(results),             // 以反序
          transmogrify);               // 插入transform的结果
```

因为front_inserter用push_front把每个对象添加到results，results中的对象顺序会和values中对应的对象顺序*相反*。这也是为什么front_inserter没有back_inserter那么常用的原因之一。另一个原因是vector不提供push_front，所以front_inserter不能用于vector。

如果你要transform把输出结果放在results前端，但你也要输出和values中对应的对象顺序相同，只要以相反的顺序迭代values：

```
list<int> results;                             // 同上
transform(values.rbegin(), values.rend(),              // 在results前端
          front_inserter(results),             // 插入transform的结果
          transmogrify);               // 保持相对的对象顺序
```

front_inserter让你强制算法在容器前端插入它们的结果，back_inserter让你告诉它们把结果放在容器后端，有点惊人的是inserter允许你强制算法把它们的结果插入容器中的任意位置：

```
vector<int> values;                    // 同上
...
vector<int> results;                   // 同上，除了现在
...                                    // 在调用transform前
                                       // results已经有一些数据
transform(values.begin(), values.end(),        // 把transmogrify的
    inserter(results, results.begin() + results.size()/2),  // 结果插入
    transmogrify);                     // results的中间
```

不管你是否使用了back_inserter、front_inserter或inserter，每次对目的区间的插入只完成一个对象。条款5解释了对于连续内存容器（vector、string和deque）来说这可能很昂贵，但条款5的建议解决方法（使用区间成员函数）不能应用于使用算法来完成插入的情况。在本例中，transform会对目的区间每次写入一个值，你无法改

变。

当你要插入的容器是vector或string时，你可以通过按照条款14的建议最小化这个代价，预先调用reserve。你仍然要承受每次发生插入时移动元素的开销，但至少你避免了重新分配容器的内在内存：

```
vector<int> values;                          // 同上
vector<int> results;
...
results.reserve(results.size() + values.size());          // 确定results至少
                                    // 还能装得下
                                    // values.size()个元素
transform(values.begin(), values.end(),            // 同上，
    inserter(results, results.begin() + results.size() / 2),// 但results
    transmogrify);                          // 没有任何重新分配操作
```

当使用reserve来提高一连串插入的效率时，总是应该记住reserve只增加容器的容量：容器的大小仍然没有改变。即使调用完reserve，当你想要让容器把新元素加入到vector或string时，你也必须对算法使用插入迭代器（比如，从back_inserter、front_inserter或inserter返回的迭代器之一）。

要把这些完全弄清楚，这里有一个提高本条款开始时的那个例子的效率的错误方法（就是我们把transmogrify作用于values里的数据的结果附加到results的那个例子）：

```
vector<int> values;                    // 同上
vector<int> results;
...
results.reserve(results.size() + values.size());    // 同上
transform(values.begin(), values.end(),     // 写入transmogrify的结果
          results.end(),            // 到未初始化的内存
          transmogrify);             // 行为未定义！
```

在这段代码中，transform愉快地试图对results尾部的原始的、未初始化的内存赋值。通常，这会造成运行期错误，因为赋值只在两个对象之间操作时有意义，而不是在一个对象和一块原始的比特之间。即使这段代码碰巧作了你想要它做的事情，results也不会知道transform在它的未使用容量上“创造”的新“对象”。直到results知道之前，它的大小在调用transform后仍然和原来一样。同样的，它的end迭代器仍和调用transform前指向同样的位置。结论呢？使用reserve而没有用插入迭代器会在算法内部导致未定义行为，也会弄乱容器。

正确地写这个例子的代码的方法是使用reserve和插入迭代器：

```
vector<int> values;                    // 同上
vector<int> results;
results.reserve(results.size() + values.size());      // 同上
transform(values.begin(), values.end(),       // 把transmogrify的结果
          back_inserter(results),       // 写入results的结尾，
          transmogrify);              // 处理时避免了重新分配
```

到目前为止，我都在假设你让像transform那样的算法把它们的结果作为新元素插入容器。这是通常的期望，但有时候你要覆盖现有容器的元素而不是插入新的。当这种情况时，你不需要插入迭代器，但你仍然需要按照本条款的建议来确保你的目的区间足够大。

比如，假设你让transform覆盖results的元素。如果results至少有和values一样多的元素，那很简单。如果没有，你也必须使用resize来确保它有。

```
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(),       // 覆盖values.size()个
          results.begin(),                 // results的元素
          transmogrify);
```

或者你可以清空results然后用通常的方式使用插入迭代器：

```
...
results.clear();                          // 销毁results中
                                  // 的所有元素
results.reserve(values.size());             // 保留足够空间
transform(values.begin(), values.end(),       // 把transform地返回值
          pack_inserter(results),       // 放入results
          transmogrify);
```

本条款论证了这个主题的很多变化，但我希望你能牢牢记住本质。无论何时你使用一个要求指定目的区间的算法，确保目的区间已经足够大或者在算法执行时可以增加大小。如果你选择增加大小，就使用插入迭代

器，比如ostream_iterators或从back_inserter、front_inserter或inserter返回的迭代器。这是所有你需要记住的东西。

（译注：这个问题在《C++标准程序库》322页也有讨论）

# 条款31：了解你的排序选择

*How can I sort thee? Let me count the ways.*

当很多程序员想到排序对象时，只有一个算法出现在脑海：sort。（有些程序员想到qsort，但一旦他们看了条款46，他们会放弃qsort的想法并用sort的想法取代之。）

现在，sort是一个令人称赞的算法，但如果你不需要你就没有必要浪费表情。有时候你不需要完全排序。比如，如果你有一个Widget的vector，你想选择20个质量最高的Widget发送给你最忠实的客户，你需要做的只是排序以鉴别出20个最好的Widget，剩下的可以保持无序。你需要的是部分排序，有一个算法叫做partial_sort，它能准确地完成它的名字所透露的事情：

```
bool qualityCompare(const Widget& lhs, const Widget& rhs)
{
    // 返回lhs的质量是不是比rhs的质量好
}
...
partial_sort(widgets.begin(),                    // 把最好的20个元素
             widgets.begin() + 20,        //  （按顺序）放在widgets的前端
             widgets.end(),
             qualityCompare);
...                                  // 使用widgets...
```

调用完partial_sort后，widgets的前20个元素是容器中最好的而且它们按顺序排列，也就是，质量最高的Widget是widgets[0]，第二高的是widgets[1]等。这就是你很容易把最好的Widget发送给你最好的客户，第二好的Widget给你第二好的客户等。

如果你关心的只是能把20个最好的Widget给你的20个最好的客户，但你不关心哪个Widget给哪个客户，partial_sort就给了你多于需要的东西。在那种情况下，你需要的只是任意顺序的20个最好的Widget。STL有一个算法精确的完成了你需要的，虽然名字不大可能从你的脑中迸出。它叫做nth_element。

nth_element排序一个区间，在ri位置（你指定的）的元素是如果区间被完全排序后会出现在那儿的元素。另外，当nth_element返回时，在n以上的元素没有在排序顺序上在位置n的元素之后的，而且在n以下的元素没

有在排序顺序上在位置n的元素之前的。如果这听起来很复杂，那只是因为我必须仔细地选择我的语言。一会儿我会解释为什么，但首先让我们看看怎么使用nth_element来保证最好的20个Widget在widgets vector的前端：

```
nth_element(widgets.begin(),              // 把最好的20个元素
            widgets.begin() + 19,         // 放在widgets前端，
            widgets.end(),                // 但不用担心
            qualityCompare);              // 它们的顺序
```

正如你所见，调用nth_element本质上等价于调用partial_sort。它们结果的唯一区别是partial_sort排序了在位置1-20的元素，而nth_element不。但是两个算法都把20个质量最高的Widget移动到vector前端。

那引出一个重要的问题。当有元素有同样质量的时候这些算法怎么办？比如假设有12个元素质量是1级（可能是最好的），15个元素质量是2级（第二好的）。在这种情况下，选择20个最好的Widget就是选择12个1级的和15个中的8个2级的。partial_sort和nth_element怎么判断15个中的哪些要放到最好的20个中？对于这个问题，当多个元素有等价的值时sort怎么判断元素的顺序？

partial_sort和nth_element以任何它们喜欢的方式排序值等价的元素，而且你不能控制它们在这方面行为。（条款19可以告诉你什么是两个值等价。）在我们的例子中，当需要把质量都是2级的Widget选出最后8个放到vector的前20个时，它们可以选择它们想要的任何一个。这不是没有理由的。如果你需要20个最好的Widget和一些也很好的Widget，你不该抱怨你取回的20个至少和你没有取回的一样好。

对于完整的排序，你有稍微多一些的控制权。有些排序算法是稳定的。在稳定排序中，如果一个区间中的两个元素有等价的值，它们的相对位置在排序后不改变。因此，如果在（未排序的）widgets vector中Widget A在Widget B之前，而且两者都有相同的质量等级，那么稳定排序算法会保证在这个vector排序后，Widget A仍然在Widget B之前。不稳定的算法没做这个保证。

partial_sort是不稳定的。nth_element、sort也没有提供稳定性，但是有一个算法——stable_sort——它完成了它的名字所透露的。如果当你排序的时候你需要稳定性，你可能要使用stable_sort。STL并不包含partial_sort和nth_element的稳定版本。

现在谈谈nth_element，这个名字奇怪的算法是个引人注目的多面手。除了能帮你找到区间顶部的n个元素，它也可以用于找到区间的中值或者找到在指定百分点的元素：

```
vector<Widget>::iterator begin(widgets.begin());     // 方便地表示widgets的
vector<Widget>::iterator end(widgets.end());         // 起点和终点
                                                     // 迭代器的变量
```

```
vector<Widget>::iterator goalPosition;        // 这个迭代器指示了
                                              // 下面代码要找的
                                              // 中等质量等级的Widget
                                              // 的位置
goalPosition = begin + widgets.size() / 2;            // 兴趣的Widget
                                              // 会是有序的vector的中间
nth_element(begin, goalPosition, end,         // 找到widgets中中等
            qualityCompare);                  // 质量等级的值
...                                           // goalPosition现在指向
                                              // 中等质量等级的Widget

                                              // 下面的代码能找到
                                              // 质量等级为75%的Widget
vector<Widget>::size_type goalOffset =        // 指出兴趣的Widget
    0.25 * widgets.size();                    // 离开始有多远
nth_element(begin, begin + goalOffset, end,           // 找到质量值为
            qualityCompare);                  // 75%的Widget
...                                           // begin + goalOffset现在指向
                                              // 质量等级为75%的Widget
```

如果你真的需要把东西按顺序放置，sort、stable_sort和partial_sort都很优秀，当你需要鉴别出顶部的n个元素或在一个指定位置的元素时nth_element就会买单。但有时候你需要类似nth_element的东西，但不是完全相同。比如假设，你不需要鉴别出20个质量最高的Widget。取而代之的是，你需要鉴别出所有质量等级为1或2的。当然你可以按照质量排序这个vector，然后搜索第一个质量等级比2差的。那就可以鉴别出质量差的Widget的区间起点。

但是完全排序需要很多工作，而且对于这个任务做了很多不必要的工作。一个更好的策略是使用partition算法，它重排区间中的元素以使所有满足某个标准的元素都在区间的开头。

比如，移动所有质量等级为2或更好的Widget到widgets前端，我们定义了一个函数来鉴别哪个Widget是这个级别。

```
bool hasAcceptableQuality(const Widget& w)
{
    // 返回w质量等级是否是2或更高;
}
```

传这个函数给partition：

```
vector<Widget>::iterator goodEnd =                  // 把所有满足hasAcceptableQuality
    partition(widgets.begin(),                  // 的widgets移动到widgets前端，
              widgets.end(),        // 并且返回一个指向第一个
              hasAcceptableQuality); // 不满足的widget的迭代器
```

此调用完成后，从widgets.begin()到goodEnd的区间容纳了所有质量是1或2的Widget，从goodEnd到widgets.end()的区间包含了所有质量等级更低的Widget。如果在分割时保持同样质量等级的Widget的相对位置很重要，我们自然会用stable_partition来代替partition。

算法sort、stable_sort、partial_sort和nth_element需要随机访问迭代器，所以它们可能只能用于vector、string、deque和数组。对标准关联容器排序元素没有意义，因为这样的容器使用它们的比较函数来在任何时候保持有序。唯一我们可能会但不能使用sort、stable_sort、partial_sort或nth_element的容器是list，list通过提供sort成员函数做了一些补偿。（有趣的是，list::sort提供了稳定排序。）所以，如果你想要排序一个list，你可以，但如果你想要对list中的对象进行partial_sort或nth_element，你必须间接完成。一个间接的方法是把元素拷贝到一个支持随机访问迭代器的容器中，然后对它应用需要的算法。另一个方法是建立一个list::iterator的容器，对那个容器使用算法，然后通过迭代器访问list元素。第三种方法是使用有序的迭代器容器的信息来迭代地把list的元素接合到你想让它们所处的位置。正如你所见，有很多选择。

partition和stable_partition与sort、stable_sort、partial_sort和nth_element不同，它们只需要双向迭代器。因此你可以在任何标准序列迭代器上使用partition和stable_partition。

我们总结一下你的排序选择：

- 如果你需要在vector、string、deque或数组上进行完全排序，你可以使用sort或stable_sort。
- 如果你有一个vector、string、deque或数组，你只需要排序前n个元素，应该用partial_sort。
- 如果你有一个vector、string、deque或数组，你需要鉴别出第n个元素或你需要鉴别出最前的n个元素，而不用知道它们的顺序，nth_element是你应该注意和调用的。
- 如果你需要把标准序列容器的元素或数组分隔为满足和不满足某个标准，你大概就要找partition或stable_partition。
- 如果你的数据是在list中，你可以直接使用partition和stable_partition，你可以使用list的sort来代替sort和stable_sort。如果你需要partial_sort或nth_element提供的效果，你就必须间接完成这个任务，但正如我在上面勾画的，会有很多选择。

另外，你可以通过把数据放在标准关联容器中的方法以保持在任何时候东西都有序。你也可能会考虑标准非STL容器priority_queue，它也可以总是保持它的元素有序。（priority_queue在传统上被考虑为STL的一部分，但是，正如我在导言中提到的，我对“STL”的定义是要求STL容器支持迭代器，而priority_queue并没有迭代器。）

“但性能怎么样？”，你想知道。这是极好的问题。一般来说，做更多工作的算法比做得少的要花更长时间，而必须稳定排序的算法比忽略稳定性的算法要花更长时间。我们可以把我们在本条款讨论的算法排序如下，需要更少资源（时间和空间）的算法列在需要更多的前面：

| | |
|---|---|
| 1. partition | 4. partial_sort |
| 2. stable_partition | 5. sort |
| 3. nth_element | 6. stable_sort |

我对于在这些排序算法之间作选择的建议是让你的选择基于你需要完成的任务上，而不是考虑性能。如果你选择的算法只完成了你需要的（比如用partition代替完全排序），你能得到的不仅是可以最清楚地表达了你要做的代码，而且是使用STL最高效的方法来完成它。

# 条款32：如果你真的想删除东西的话就在类似remove的算法后接上erase

我将从remove的复习开始这个条款，因为remove是STL中最糊涂的算法。误解remove很容易，驱散所有关于remove行为的疑虑——为什么它这么做，它是怎么做的——是很重要的。

这是remove的声明：

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

就像所有算法，remove接收指定它操作的元素区间的一对迭代器。它不接收一个容器，所以remove不知道它作用于哪个容器。此外，remove也不可能发现容器，因为没有办法从一个迭代器获取对应于它的容器。

想想怎么从容器中除去一个元素。唯一的方法是调用那个容器的一个成员函数，几乎都是erase的某个形式，（list有几个除去元素的成员函数不叫erase，但它们仍然是成员函数。）因为唯一从容器中除去一个元素的方法是在那个容器上调用一个成员函数，而且因为remove无法知道它正在操作的容器，所以remove*不可能从一个容器中除去元素*。这解释了另一个令人沮丧的观点——从一个容器中remove元素不会改变容器中元素的个数：

```
vector<int> v;              // 建立一个vector<int> 用1-10填充它
v.reserve(10);              //  （调用reserve的解释在条款14）
for (int i = 1; i <= 10; ++i) {
    v.push_back(i);
}

cout << v.size();                  // 打印10
v[3] = v[5] = v[9] = 99;           // 设置3个元素为99
remove(v.begin(), v.end(), 99); // 删除所有等于99的元素
cout << v.size();                  // 仍然是10！
```

要搞清这个例子的意思，记住下面这句话：

> remove并不“真的”删除东西，因为它做不到。

重复对你有好处：

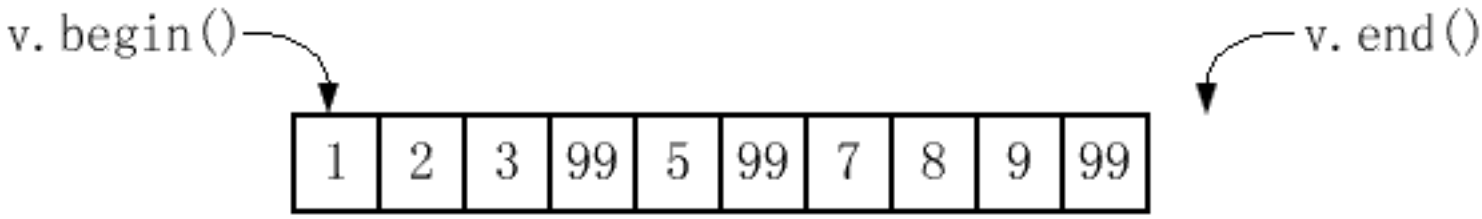
> remove并不“真的”删除东西，*因为它做不到*。

remove不知道它要从哪个容器删除东西，而没有容器，它就没有办法调用成员函数，而如果“真的”要删除东西，那就是必要的。

上面解释了remove不做什么，而且解释了为什么它不做。我们现在需要复习的是remove做了什么。

非常简要地说一下，remove移动指定区间中的元素直到所有“不删除的”元素在区间的开头（相对位置和原来它们的一样）。它返回一个指向最后一个的下一个“不删除的”元素的迭代器。返回值是区间的“新逻辑终点”。

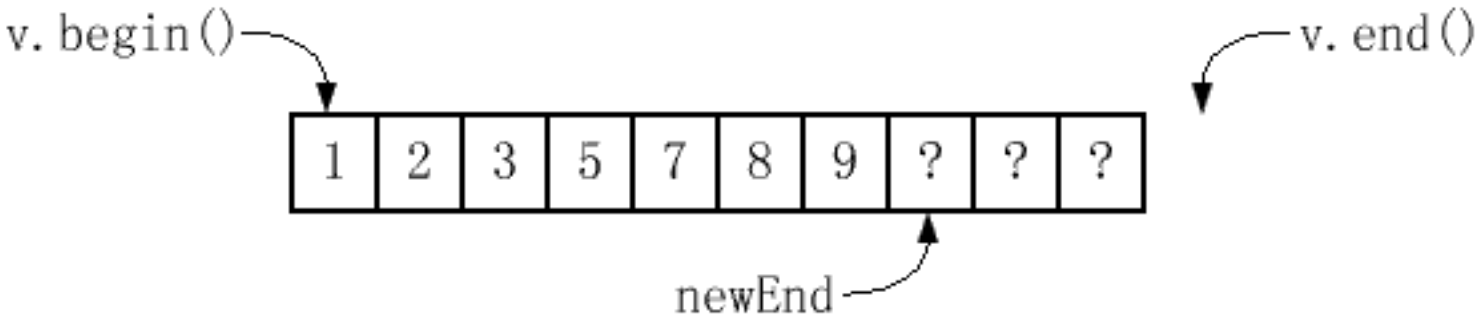
举个例子，这是v在调用remove前看起来的样子：




如果我们把remove的返回值存放在一个叫做newEnd的新迭代器中：

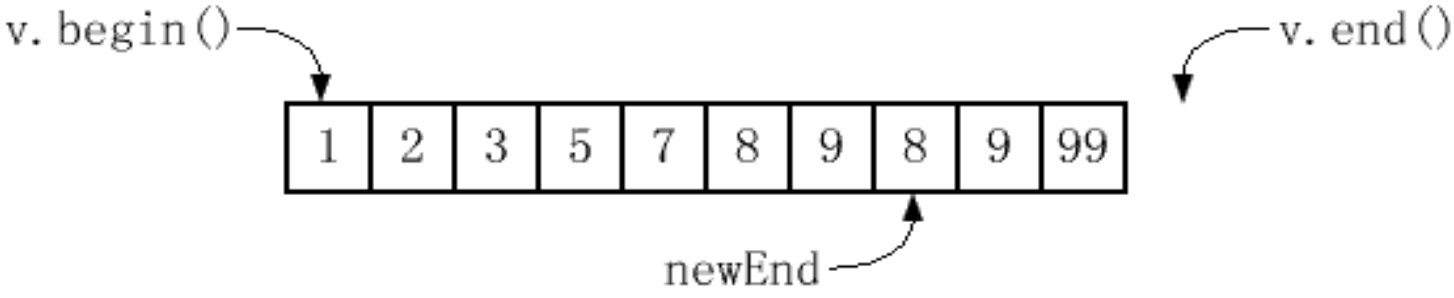
```
vector<int>::iterator newEnd(remove(v.begin(), v.end(), 99));
```

这是调用后v看起来的样子：




这里我用问号来标明那些在概念上已经从v中被删除，但继续存在的元素的值。

如果“不删除的”元素在v中的v.begin()和newEnd之间，“删除的”元素就必须在newEnd和v.end()之间——这好像很合理。*事实上不是这样！*“删除的”值完全不必再存在于v中了。remove并没有改变区间中元素的顺序，所以不会把所有“删除的”元素放在结尾，并安排所有“不删除的”值在开头。虽然标准没有要求，但一般来说区间中在新逻辑终点以后的元素*仍保持它们的原值*。调用完remove后，在我知道的所有实现中，v看起来像这样：




正如你所见，两个曾经存在于v的“99”不再在那儿了，而一个“99”仍然存在。一般来说，调用完remove后，从区间中删除的值可能是也可能不在区间中继续存在。大多数人觉得这很奇怪，但为什么？你要求remove除去一些值，所以它做了。你并没有要求它把删除的值放在一个你以后可以获取的特定位置，所以它没有做。有问题吗？（如果你不想失去任何值，你可能应该调用partition或stable_partition而不是remove，partition在条款31中描述。）

remove的行为听起来很可恶，但它只不过是算法操作的附带结果。在内部，remove遍历这个区间，把要“删除的”值覆盖为后面要保留的值。这个覆盖通过对持有被覆盖的值的元素赋值来完成。

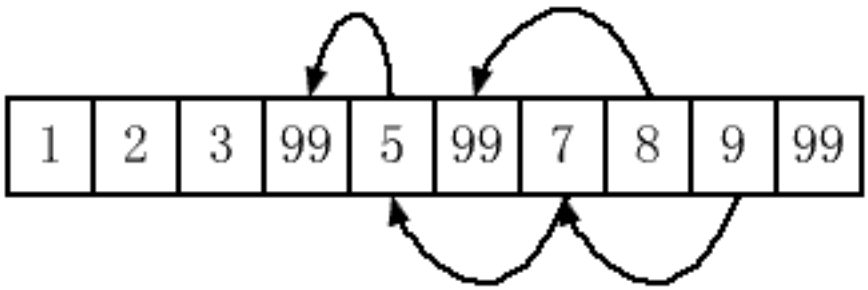
你可以想象remove完成了一种压缩，被删除的值表演了在压缩中被填充的洞的角色。对于我们的vector v，它按照下面的表演：

1. remove检测v[0]，发现它的值不是要被删除的，然后移动到v[1]。同样的情况发生在v[1]和v[2]。
2. 发现v[3]应该被删除，所以它记录下v[3]的值应该被覆盖，然后它移动到v[4]。这类似记录v[3]是一个

需要填充的“ 洞 ”。

3. 发现v[4]的值应该被保持，所以它把v[4]赋给v[3]，记录下v[4]应该被覆盖，然后移动到v[5]。继续类似的压缩，它用v[4]“ 填充 ”v[3]而且记录v[4]现在是一个洞。
4. 发现v[5]应该被删除，所以忽略并它移动到v[6]。仍然记得v[4]是一个等待填充的洞。
5. 发现v[6]是一个应该保留的值，所以把v[6]赋给v[4]。记得v[5]现在是下一个要被填充的洞，然后移到v[7]。
6. 在某种意义上类似上面的，检查v[7]、v[8]和v[9]。把v[7]赋给v[5]，v[8]赋给v[6]，忽略v[9]，因为v[9]的值是要被删除的。
7. 返回指定下一个要被覆盖的元素的迭代器，在这个例子中这个元素是v[7]。

你可以预想在v中值的移动情况像这样：




正如条款33所解释的，事实上当remove在删除时覆盖的值是指针时，会有重要的影响。但是对于本条款，知道remove不从容器中除去任何元素因为它做不到就够了。只有容器成员函数可以除去容器元素，而那是本条款的整个要点：如果你真的要删除东西的话，你应该在remove后面接上erase。

你要erase的元素很容易识别。它们是从区间的“ 新逻辑终点 ”开始持续到区间真的终点的原来区间的元素。要除去那些元素，你要做的所有事情就是用那两个迭代器调用erase的区间形式（参见条款5）。因为remove本身很方便地返回了区间新逻辑终点的迭代器，这个调用很直截了当：

```
vector<int> v;                          // 正如从前
v.erase(remove(v.begin(), v.end(), 99), v.end());          // 真的删除所有
                                        // 等于99的元素
cout << v.size();                               // 现在返回7
```

把remove的返回值作为erase区间形式第一个实参传递很常见，这是个惯用法。事实上，remove和erase是亲密联盟，这两个整合到list成员函数remove中。这是STL中唯一名叫remove又能从容器中除去元素的函数：

```
list<int> li;               // 建立一个list
                    // 放一些值进去
```

```
li.remove(99);          // 除去所有等于99的元素：
                        // 真的删除元素，
                        // 所以它的大小可能改变了
```

坦白地说，调用这个remove函数是一个STL中的矛盾。在关联容器中类似的函数叫erase，list的remove也可以叫做erase。但它没有，所以我们都必须习惯它。我们所处于的世界不是所有可能中最好的世界，但却是我们所处的。（附加一点，条款44指出，对于list，调用remove成员函数比应用erase-remove惯用法更高效。）

一旦你知道了remove不能“真的”从一个容器中删除东西，和erase联合使用就变成理所当然了。你要记住的唯一其他的东西是remove不是唯一这种情况的算法。另外有两种“类似remove”的算法：remove_if和unique。

remove和remove_if之间的相似性很直截了当。所以我不会细讲，但unique行为也像remove。它用来从一个区间删除东西（邻近的重复值）而不用访问持有区间元素的容器。结果，如果你真的要从容器中删除元素，你也必须成对调用unique和erase，unique在list中也类似于remove。正像list::remove真的删除东西（而且比erase-remove惯用法高效得多）。list::unique也真的删除邻近的重复值（也比erase-unique高效）。

（译注：《C++标准程序库》111页5.6节有remove的详细解释）

# 条款33：提防在指针的容器上使用类似remove的算法

你在管理一堆动态分配的Widgets，每一个都可能通过检验，你把结果指针保存在一个vector中：

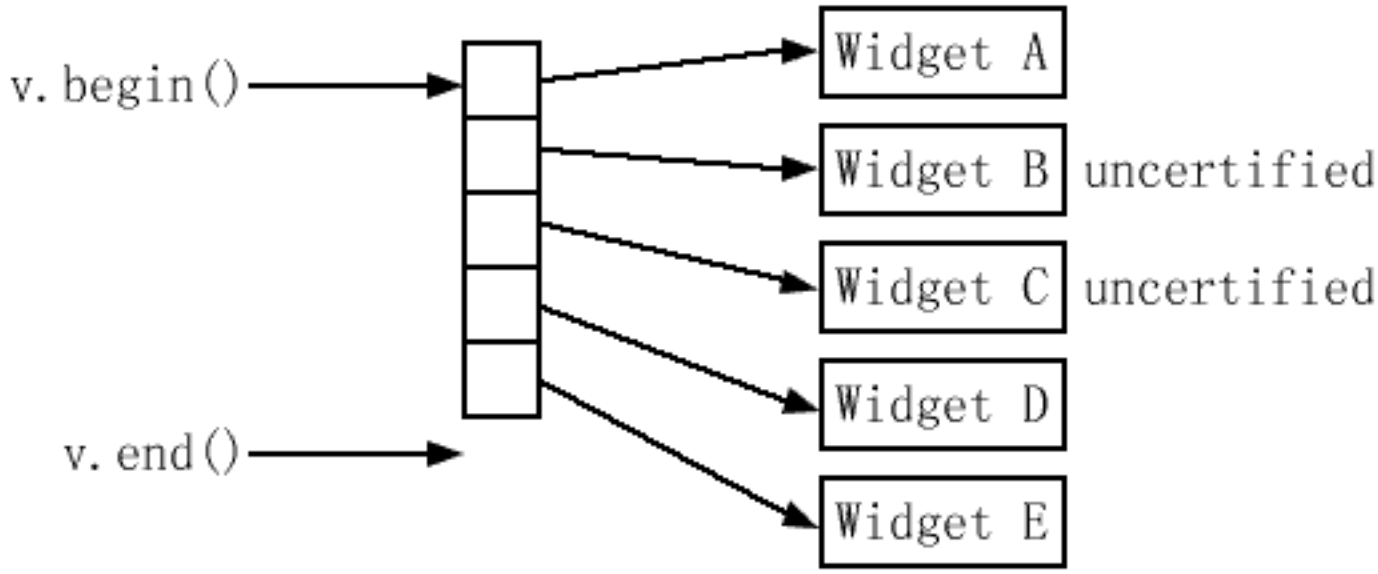
```
class Widget{
public:
    ...
    bool isCertified() const;     // 这个Widget是否通过检验
    ...
};

vector<Widget*> v;                // 建立一个vector然后用
...                       // 动态分配的Widget
v.push_back(new Widget);          // 的指针填充
```

当和v工作一段时间后，你决定除去未通过检验的Widget，因为你不再需要它们了。记住条款43的警告尽量用算法调用代替显式循环和读过的条款32关于remove和erase之间关系的描述，你自然会想到转向erase-remove惯用法，虽然这次你使用了remove_if：

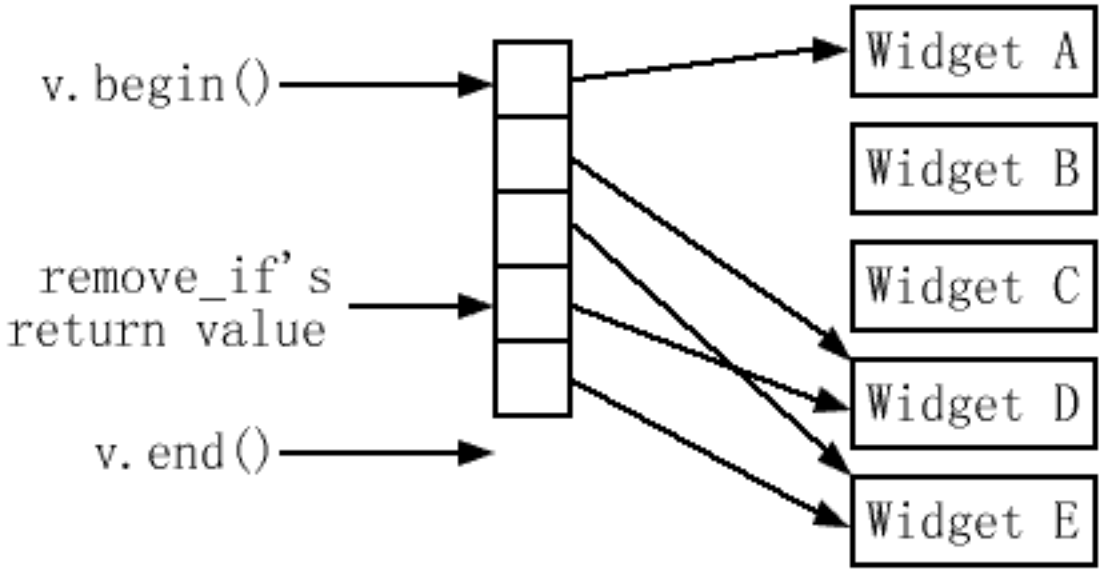
```
v.erase(remove_if(v.begin(), v.end(),                    // 删除未通过检验的
                  not1(mem_fun(&Widget::isCertified))),  // Widget指针
                  v.end());                      // 关于mem_fun的信息
                                                 // 参见条款41
```

突然你开始担心erase的调用，因为你朦胧的记起条款7关于摧毁容器中的一个指针也不会删除指针指向的东西的讨论。这是个合理的担心，但在这里，太晚了。当调用erase时，极可能你已经泄漏了资源。担心erase，是的，但首先，担心一下remove_if。

我们假设在调用remove_if前，v看起来像这样，我已经指出了未通过检验的Widget：



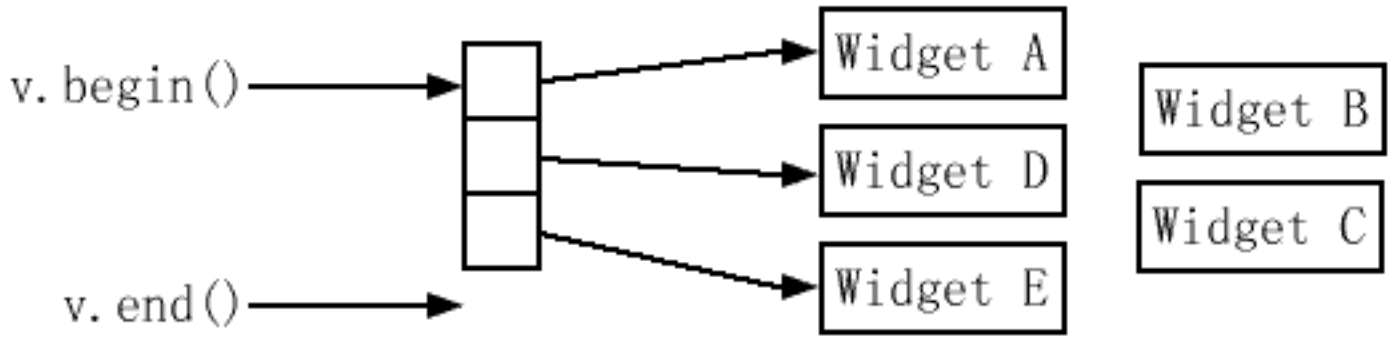

在调用remove_if后，一般来说v看起来像这样（包含从remove_if返回的迭代器）：




如果你看不懂这个转换，请转向条款32，因为它准确地解释了调用remove——或者，在这个例子里，remove_if——做了什么。

资源泄漏的理由现在很明朗了。指向Widget B和C的“删除的”指针被vector中后面的“不删除的”指针覆盖。没有什么指向两个未通过检验的Widget，它们也没有被删除，它们的内存和其他资源泄漏了。

一旦remove_if和erase返回后，情况看起来像这样：




这造成资源泄漏尤其明显了，现在你也很清楚为什么应该努力避免在动态分配的指针的容器上使用remove和类似算法（也就是，remove_if和unique）。在很多情况下，你会发现partition算法（参见条款31）是合理的替代品。

如果你无法避免在那样的容器上使用remove，排除这个问题一种方法是在应用erase-remove惯用法之前先删除指针并设置它们为空，然后除去容器中的所有空指针：

```
void delAndNullifyUncertified(Widget*& pWidget)       // 如果*pWidget是一个
{                                    // 未通过检验Widget,
    if (!pWidget->isCertified()) {              // 删除指针
        delete pWidget;                  // 并且设置它为空
        pWidget = 0;
    }
}

for_each(v.begin(), v.end(),                // 把所有指向未通过检验Widget的
         delAndNullifyUncertified);     // 指针删除并且设置为空

v.erase(remove(v.begin(), v.end(),                 // 从v中除去空指针
         static_cast<Widget*>(0)),     // 0必须映射到一个指针,
         v.end());                 // 让C++可以
                                   // 正确地推出remove的
                                   // 第三个参数的类型
```

当然，这假设vector并不容纳任何你想保留的空指针。如果有的话，你可能必须自己写循环来按你的方式删除指针。在你遍历容器时从容器中删除元素有一些细微的要注意的地方，在考虑那种方法之前确定已经读过条款9。

如果你把指针的容器替换成执行引用计数的智能指针的容器，删除相关的困难就不存在了，你可以直接使用

erase-remove惯用法：

```
template<typename T>                          // RCSP =“ 引用计数
class RCSP { ...};                           // 智能指针 ”
typedef RCSP< Widget> RCSPW;                      // RCSPW =“ RCSP to Widget ”
vector<RCSPW > v;                            // 建立一个vector，用动态
...                                  // 分配Widget的
v.push_back(RCSPW(new Widget));                   // 智能指针填充它
...
v.erase(remove_if(v.begin(), v.end(),             // erase未通过检验的
          not1 (mem_fun(&Widget::isCertified))),  // Widget的指针
               v.end());                    // 没有资源泄漏
```

要让这些工作，你的智能指针类型就必须可以（比如RCSP<Widget>）隐式转换为相应的内建指针类型（比如Widget*）。那是因为容器持有智能指针，但被调用的成员函数（比如Widget::isCertified）要的是内建指针。如果不存在隐式转换，你的编译器会抗议的。

如果在你的程序工具箱中碰巧没有一个引用计数智能指针模板，你应该从Boost库中得到shared_ptr模板。关于Boost的介绍，请看条款50。

不管你怎么选择处理动态分配指针的容器，通过引用计数智能指针、在调用类似remove的算法前手动删除和废弃指针或者一些你自己发明的技术，本条款的指导意义依然一样：提防在指针的容器上使用类似remove的算法。没有注意这个建议的人只能造成资源泄漏。

# 条款34：注意哪个算法需要有序区间

不是所有算法可以用于任意区间。比如，remove（参见条款32和33）需要前向迭代器和可以通过这些迭代器赋值的能力。所以，它不能应用于由输入迭代器划分的区间，也不能是map或multimap，也不能是set和multiset的一些实现（参见条款22）。同样，很多排序算法（参见条款31）需要随机访问迭代器，所以不可能在一个list的元素上调用这些算法。

如果你冒犯了这些规则，你的代码将不能编译，可能会出现一个非常冗长和不可理解的错误信息（参见条款49）。但其他算法的需求则更为狡猾。在这些之中，可能最常见的就是一些算法需要有序值的区间。无论何时都应该坚持这个需求，因为冒犯它不仅会导致编译器诊断，而且会造成未定义的运行期行为。

既可以和有序又可以和无序区间合作的算法很少，但当操作有序区间的时候它们最有用。你可以理解这些算法是怎么工作的，因为那会解释为什么有序区间最适合它们。

我知道你们中的一部分会用蛮力记忆，所以这里有一个只能操作有序数据的算法的表：

| | |
|---|---|
| binary_search | lower_bound |
| upper_bound | equal_range |
| set_union | set_intersection |
| set_difference | set_symmetric_difference |
| merge | inplace_merge |
| includes | |

另外，下面的算法一般用于有序区间，虽然它们不要求：

| | |
|---|---|
| unique | unique_copy |

我们马上会看到“有序”的定义是一个重要的约束，但首先，让我们熟悉一下这些算法。如果你知道了为什么需要有序区间，就很容易记起哪个算法需要那样的区间。

搜索算法binary_search、lower_bound、upper_bound和equal_range（参见条款45）需要有序区间，因为它们使

用二分法查找来搜索值。像C库中的bsearch，这些算法保证了对数时间的查找，但作为交换的是，你必须给它们已经排过序的值。

实际上，这些算法保证对数时间查找不是很正确。仅当传给它们的是随机访问迭代器时它们才能保证有那样的性能。如果给它们威力比较小的迭代器（比如双向迭代器），它们仍然进行对数次比较，但运行是线性时间的。那是因为，缺乏进行“迭代器算术（arithmetic）”的能力。它们在搜索的区间中需要花费线性时间来从一个地方移动到另一个地方。

算法set_union、set_intersection、set_difference和set_symmetric_difference的四人组提供了线性时间设置它们名字所提出的操作的性能。为什么它们需要有序区间？因为如果不是的话，它们不能以线性时间完成它们的工作。如果你开始发觉一个趋势——需要有序区间的算法为了比它们用于可能无序区间提供更好的性能而这么做，你是对的。保持协调，这个趋势会继续。

merge和inplace_merge执行了有效的单遍合并排序算法：它们读取两个有序区间，然后产生一个包含了两个源区间所有元素的新有序区间。它们以线性时间执行，如果它们不知道源区间已经有序就不能完成。

最后一个需要有序区间的算法是includes。它用来检测是否一个区间的所有对象也在另一个区间中。因为includes可能假设它的两个区间都已经有序，所以它保证了线性时间性能。没有那个保证，一般来说它会变慢。

不像我们刚才讨论过的算法，unique和unique_copy甚至在无序区间上也提供了定义良好的行为。但看看标准是怎么描述unique的行为的（斜体字是雷区）：

> 从每个相等元素的*连续*组中去除第一个以外所有的元素。

换句话说，如果你要unique从一个区间去除所有重复值（也就是，让区间中所有值“唯一”），你必须先确保所有重复值一个接着一个。猜到什么了？那是排序完成的东西之一。实际上，unique一般用于从区间中去除所有重复值，所以你几乎总是要确保你传递给unique（或unique_copy）的区间是有序的。（Unix开发者会发现STL的unique和Unix的uniq之间有惊人的相似，我想这个相似决不是巧合。）

顺便说说，unique从一个区间除去元素的方式和remove一样，也就是说它只是区分出不除去的元素。如果你不知道那是什么意思，请立刻转向条款32和33。强调理解remove和类似remove的算法（包括unique）行为的重要性永远没有过分的时候。光有基本的理解还不够。如果你不知道它们做了什么，你会陷入困境。

这让我必须说说关于有序区间的意思的难懂的条文。因为STL允许你指定用于排序的比较函数，不同的区间可能以不同的方式排序。比如，给定两个int的区间，一个可能以默认方式排序（也就是升序），而另一个使用greater<int>排序，因此是降序。给定Widget的两个区间，一个可能以价格排序而另一个可能以年龄排序。因为有很多不同的方式来排序，所以保证给STL所使用的排序相关信息一致是很重要的。如果你传一个区间给一个也带有比较函数的算法，确保你传递的比较函数行为和你用于排序这个区间的一样。

这里有一个你并不想那么做的例子：

```
vector<int> v;                          // 建立一个vector，
...                                     // 把一些数据放进去
sort(v.begin(), v.end(), greater<int>());          // 降序排列
...                                     // 使用这个vector
                                        //（没有改变它）
bool a5Exists =                         // 在这个vector中搜索5
    binary_search(v.begin(), v.end(), 5);  // 假设它是升序排列！
```

默认情况下，binary_search假设它搜索的区间是以“ < ”排序（也就是，值是升序），但在本例中，这个vector是*降序*。当你在值的排列顺序和算法所期望的不同的区间上调用binary_search (或lower_bound等）会导致未定义的结果时，你不该惊奇。

要让代码行为正确，你必须告诉binary_search要使用和sort同样的比较函数：

```
bool a5Exists =                                 // 搜索5
    binary_search(v.begin(), v.end(), 5. greater<int>());       // 把greater作为
                                                // 比较函数
```

所有需要有序区间的算法（也就是除了unique和unique_copy外本条款的所有算法）通过等价来判断两个值是否“ 相同 ”，就像标准关联容器（它们本身是有序的）。相反，unique和unique_copy判断两个对象“ 相同 ”的默认方式是通过相等，但是你可以通过传给这些算法一个定义了“ 相同 ”的意义的判断式来覆盖这个默认情况。等价和相等之间区别的详细讨论，参考条款19。

11个需要有序区间的算法为了比其他可能性提供更好的性能而这么做。只要你记住只传给它们有序区间，只要你保证用于算法的比较函数和用于排序的一致，你就会酷爱没有麻烦的查找、设置和合并操作，加上你会发现unique和unique_copy除去了所有的重复值，正如你要它们完成的一样。

# 条款35：通过mismatch或lexicographical比较实现简单的忽略大小写字符串比较

一个STL菜鸟最常问的问题是“我怎么使用STL来进行忽略大小写的字符串比较？”这是一个令人迷惑的简单问题。忽略大小写字符串比较要么真的简单要么真的困难，依赖于你要多一般地解决这个问题。如果你忽略国际化问题而且只关注于设计成字符串strcmp那样的类型，这个任务很简单。如果你要有strcmp不具有的按语言处理字符串中的字符的能力（也就是，容纳文本的字符串是除了英语以外的语言）或程序使用了区域设置而不是默认的，这个任务很困难。

在本条款中，我会作出这个问题的一个简单版本，因为那足以演示STL怎么完成任务。（这个问题一个比较难的版本也能用STL解决。准确地说，它解决了你可以在附录A中看到的有关区域设置的问题。）为了让简单的问题变得更有挑战性，我会处理两次。想要使用忽略大小写比较的程序员通常需要两种不同的调用接口，一种类似strcmp（返回一个负数、零或正数），另一种类似operator（返回true或false）。因此我会演示怎么使用STL算法实现两种调用接口。

但是首先，我们需要一种方法来确定两个字符除了大小写之外是否相等。当需要考虑国际化问题时，这是一复杂的问题。下面的字符比较显然是一个过分简单的解决方案，但它类似strcmp进行的字符串比较，因为本条款我只考虑类似strcmp的字符串比较，不考虑国际化问题，这个函数就够了：

```
int ciCharCompare(char c1, char c2)                // 忽略大小写比较字符
{                                  // c1和c2，如果c1 < c2返回-1，
                                   // 如果c1==c2返回0，如果c1 > c2返回1
    int Ic1 = tolower(static_cast<unsigned char>(c1));// 这些语句的解释
    int Ic2 = tolower(static_cast<unsigned char>(c2));// 看下文

    if (Ic1 < Ic2) return -1;
    if (lc1 > Ic2) return 1;
    return 0;
}
```

这个函数遵循了strcmp，可以返回一个负数、零或正数，依赖于c1和c2之间的关系。与strcmp不同的是，ciCharCompare在进行比较前把两个参数转化为小写。这就产生了忽略大小写的字符比较。

正如<cctype>（也是<ctype.h>）里的很多函数，tolower的参数和返回值类型是int，但除非这个int是EOF，它的值必须能表现为一个unsigned char。在C和C++中，char可能或可能不是有符号的（依赖于实现），当char有符号时，唯一确认它的值可以表现为unsigned char的方式是在调用tolower之前转换一下。这就解释了上面代码中的转换。（在char已经是无符号的实现中，这个转换没有作用。）这也解释了保存tolower返回值的是int而不是char。

给定了ciCharCompare，就很容易写出我们的第一个忽略大小写的两个字符串比较函数，提供了一个类似strcmp的接口。ciStringCompare这个函数，返回一个负数、零或正数，依赖于要比较的字符串的关系。它基于mismatch算法，因为mismatch确定了两个区间中第一个对应的不相同的值的位置。

在我们可以调用mismatch之前，我们必须先满足它的前提。特别是，我们必须确定一个字符串是否比另一个短，短的字符串作为第一个区间传递。因此我们可以把真正的工作放在一个叫做ciStringCompareImpl的函数，然后让ciStringCompare简单地确保传进去的实参顺序正确，如果实参交换了就调整返回值：

```
int ciStringCompareImpl(const string& s1,          // 实现请看下文
                        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，大部分工作由mismatch来完成。它返回一对迭代器，表示了区间中第一个对应的字符不相同的位置：

```
int ciStringCompareImpl(const string& si, const strings s2)
{
    typedef pair<string::const_iterator,              // PSCI = “ pair of
            string::const_iterator> PSCI;  // string::const_iterator ”
    PSCI p = mismatch(                          // 下文解释了
            s1.begin(), s1.end(),       // 为什么我们
            s2.begin(),                 // 需要not2；参见
            not2(ptr_fun(ciCharCompare)));  // 条款41解释了为什么
                                        // 我们需要ptr_fun
    if (p.first== s1.end()) {                   // 如果为真，s1等于
        if (p.second == s2.end()) return 0;         // s2或s1比s2短
        else return -1;
```

```
    }
    return ciCharCompare(*p.first, *p.second);          // 两个字符串的关系
}                                                       // 和不匹配的字符一样
```

幸运的是，注释把所有东西都弄清楚了。基本上，一旦你知道了字符串中第一个不同字符的位置，就可以很容易决定哪个字符串，如果有的话，在另一个前面。唯一可能感到奇怪的是传给mismatch的判断式，也就是not2(ptr_fun(ciCharCompare))。当字符匹配时这个判断式返回true，因为当判断式返回false时mismatch会停止。我们不能为此使用ciCharCompare，因为它返回-1、1或0，而*当字符匹配时它返回0*，就像strcmp。如果我们把ciCharCompare作为判断式传给mismatch，C++会把ciCharCompare的返回类型转换为bool，而当然bool中零的等价物是false，正好和我们想要的相反！同样的，当ciCharCompare返回1或-1，那会被解释成true，因为，就像C，所有非零整数值都看作true。这再次和我们想要的相反。要修正这个语义倒置，我们在ciCharCompare前面放上not2和ptr_fun，而且我们都会一直很快乐地生活。

我们的第二个方法ciStringCompare是产生一个合适的STL判断式：可以在关联容器中用作比较函数的函数。这个实现很短也很好，因为我们需要做的是把ciCharCompare修改为一个有判断式接口的字符比较函数，然后把进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是否
                                                        // 在c2前面；
    tolower(static_cast<unsigned char>(c1)) <           // 条款46解释了为什么
        tolower(static_cast<unsigned char>(c2));        // 一个函数对象可能
}                                                       // 比函数好

bool ciStringCompare(const string& s1, const string& s2)
{
    return lexicographical_compare(s1.begin(), s1.end(),        // 关于这个
                                   s2.begin(), s2.end(),  // 算法调用的
                                   ciCharLess);         // 讨论在下文
}
```

不，我不会让你再有悬念了。名字最长的算法是set_symmetric_difference。

如果你熟悉lexicographical_compare的行为，上面的代码在清楚不过了。如果你不，可能像隔着一块混凝土一样。幸运的是，要把混凝土换成玻璃并不难。

lexicographical_compare是strcmp的泛型版本。strcmp只对字符数组起作用，但lexicographical_compare对所有任

何类型的值的区间都起作用。同时，strcmp总是比较两个字符来看看它们的关系是相等、小于或大于另一个。lexicographical_compare可以传入一个决定两个值是否满足一个用户定义标准的二元判断式。

在上面的调用中，lexicographical_compare用来寻找s1和s2第一个不同的位置，基于调用ciCharLess的结果。如果，使用那个位置的字符，ciCharLess返回true，lexicographical_compare也是；如果，在第一个字符不同的位置，从第一个字符串来的字符先于对应的来自第二个字符串的字符，第一个字符串就先于第二个。就像strcmp，lexicographical_compare认为两个相等值的区间是相等的，因此它对于这样的两个区间返回false：第一个区间不在第二个之前。也像strcmp，如果第一个区间在发现不同的对应值之前就结束了，lexicographical_compare返回true：一个先于任何区间的前缀是一个前缀。

谈了很多关于mismatch和lexicographical_compare。虽然我专注于本书的可移植性，但如果我没有提到忽略大小写字符串比较函数作为对标准C库的非标准扩展而广泛存在，我可能是玩忽职守的。它们一般有stricmp或strcmpi这样的名字，而且它们一般和我们在本条款开发的函数一样没有提供更多对国际化的支持。如果你想牺牲一些移植性，你知道你的字符串没有包含嵌入的null，而且你不关心国际化，你可以找到完全不用STL实现一个忽略大小写字符串比较最简单的方式。取而代之的是，它把两个字符串都转换为const char*指针（参见条款16），然后对指针使用stricmp或strcmpi：

```
int ciStringCompare(const string& s1, const string& s2)
{
    return stricmp(s1.c_str(), s2.c_str());          // 你的系统上的
}                                                    // 函数名可能
                                                     // 不是stricmp
```

有的人可能称此为技巧（hack），但stricmp/strcmpi被优化为只做一件事情，对长字符串运行起来一般比通用的算法mismatch和lexicographical_compare快得多。如果那对你很重要，你可能不在乎你用非标准C函数完成标准STL算法。有时候最有效地使用STL的方法是认识到其他方法更好。

# 条款36：了解copy_if的正确实现

STL有很多有趣的地方，其中一个是虽然有11个名字带“ copy ”的算法：

| | |
|---|---|
| copy | copy_backward |
| replace_copy | reverse_copy |
| replace_copy_if | unique_copy |
| remove_copy | rotate_copy |
| remove_copy_if | partial_sort_copy |
| unintialized_copy | |

但没有一个是copy_if。这意味着你可以replace_copy_if，你可以remove_copy_if，你可以copy_backward或者reverse_copy，但如果你只是简单地想要拷贝一个区间中满足某个判断式的元素，你只能自己做。

比如，假设你有一个函数来决定一个Widget是否有缺陷的：

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

而且你希望把一个vector中所有有缺陷的Widget写到cerr。如果存在copy_if，你可以简单地这么做：

```
vector<Widget> widgets;
...
copy_if(widgets.begin(), widgets.end(),          // 这无法编译：
        ostream_iterator<Widget>(cerr, "\n"),    // STL中并没有copy_if
        isDefective);
```

具有讽刺意味的事，copy_if是最初的惠普STL的一部分，惠普STL形成了STL的基础，而STL现在是标准C++库的一部分。有些怪癖会使历史变得有趣，在从HP STL中为标准筛选出一些大小易于管理的东西时，copy_if成了其中一个在剪接室中被遗弃的东西。

在《The C++ Programming Language》[7]，Stroustrup强调写copy_if是微不足道的，他是对的，但那不意味着

实现正确的琐事很简单。比如，这里有一个合理的看待copy_if，很多人（包括我）曾经知道的实现：

```
template<typename InputIterator,                    // 一个不很正确的
         typename OutputIterator,           // copy_if实现
         typename Predicate>
OutputIterator copy_if(InputIterator begin,
                       InputIterator end,
                       OutputIterator destBegin, Predicate p)
{
    return remove_copy_if(begin, end, destBegin, not1(p));
}
```

这个方法是基于这个观点——虽然STL并没有让你说“拷贝每个判断式为true的东西”，但它的确让你说了“拷贝除了判断式不为true以外的每个东西”。要实现copy_if，似乎我们需要做的就只是加一个not1在我们希望传给copy_if的判断式前面，然后把这个结果判断式传给remove_copy_if，结果就是上面的代码。

如果上面的理由有效，我们就可以用这种方法写出有缺陷的Widget：

```
copy_if(widgets.begin(), widgets.end(),            // well-intentioned code
        ostream_iterator<Widget>(cerr, "\n"),  // that will not compile
        isDefective);
```

你的STL平台将会敌视这段代码，因为它试图对isDefective应用not1（这个应用出现在copy_if内部）。正如条款41试图将清楚的，not1不能直接应用于一个函数指针，函数指针必须先传给ptr_fun。要调用这个copy_if实现，你必须传递的不仅是一个函数对象，而且是一个*可适配*的函数对象。这够简单了，但是想要成为STL算法用户的人应该不必这样。标准STL算法从来不要求它们的仿函数是可适配的，copy_if也不应该要求。上面的实现是好的，但不够好。

这是copy_if正确的微不足道的实现：

```
template<typename InputIterator,                    // 一个copy_if的
         typename OutputIterator,           // 正确实现
         typename Predicate>
OutputIterator copy_if(InputIterator begin,
                       InputIterator end,
                       OutputIterator destBegin,
                       Predicate p) {
```

```
    while (begin != end) {
        if (p(*begin))*destBegin++ = *begin;
        ++begin;
    }

    return destBegin;
}
```

告诉你copy_if多有用，加上新STL程序员趋向于希望无论如何它应该存在的事实，所以好的做法是把copy_if——正确的那个！——放在你局部的STL相关工具库中，而且只要合适就使用。

# 条款37：用accumulate或for_each来统计区间

有时候你需要把整个区间提炼成一个单独的数，或，更一般地，一个单独的对象。对于一般需要的信息，有特殊目的的算法来完成这个任务，比如，count告诉你区间中有多少等于某个值的元素，而count_if告诉你有多少元素满足一个判断式。区间中的最小和最大值可以通过min_element和max_element获得。

但有时，你需要用一些自定义的方式统计（summarize）区间，而且在那些情况中，你需要比count、count_if、min_element或max_element更灵活的东西。比如，你可能想要对一个容器中的字符串长度求和。你可能想要数的区间的乘积。你可能想要point区间的平均坐标。在那些情况中，你需要*统计*一个区间，但你需要有定义你需要统计的东西的能力。没问题。STL为你准备了那样的算法，它叫作accumulate。你可能不熟悉accumulate，因为，不像大部分算法，它不存在于<algorithm>。取而代之的是，它和其他三个“数值算法”都在<numeric>中。（那三个其它的算法是inner_product、adjacent_difference和partial_sum。）

就像很多算法，accumulate存在两种形式。带有一对迭代器和初始值的形式可以返回初始值加由迭代器划分出的区间中值的和：

```
list<double> ld;                                    // 建立一个list，放
...                                  // 一些double进去
double sum = accumulate(ld.begin(), Id.end(), 0.0);          // 计算它们的和，
                                     // 从0.0开始
```

在本例中，注意初始值指定为0.0，不是简单的0。这很重要。0.0的类型是double，所以accumulate内部使用了一个double类型的变量来存储计算的和。如果这么写这个调用：

```
double sum = accumulate(ld.begin(), Id.end(), 0);           // 计算它们的和，
                                     // 从0开始；
                                     // 这不正确！
```

如果初始值是int 0，所以accumulate内部就会使用一个int来保存它计算的值。那个int最后变成accumulate的返回值，而且它用来初始化和变量。这代码可以编译和运行，但和的值可能不对。不是保存真的double的list的和，它可能保存了所有的double加起来的结果，但每次加法后把结果转换为一个int。

accumulate只需要输入迭代器，所以你甚至可以使用istream_iterator和istreambuf_iterator（参见条款29）：

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

进行数值算法是accumulate的默认行为。但当使用accumulate的另一种形式，带有一个初始和值与一个任意的统计函数，这变得一般很多。

比如，考虑怎么使用accumulate来计算容器中的字符串的长度和。要计算这个和，accumulate需要知道两个东西。第一，同上，它必须知道和的开始。在我们的例子中，它是0。第二，它必须知道每次看到一个新的字符串时怎么更新这个和。要完成这个任务，我们写一个函数，它带有目前的和与新的字符串，而且返回更新的和：

```
string::size_type                              // string::size_type的内容
stringLengthSum(string::size_type sumSoFar,          // 请看下文
                const string& s)
{
    return sumSoFar + s.size();
}
```

这个函数的函数体非常简单，但你可能发现自己陷于string::size_type的出现。不要那样。每个标准STL容器都有一个typedef叫做size_type，那是容器计量东西的类型。比如，这是容器的size函数的返回类型。对于所有的标准容器，size_type必须是size_t，但理论上非标准STL兼容的容器可能让size_type使用一个不同的类型（虽然我花了很多时间来想为什么它们要那么做）。对于标准容器，你可以把Container::size_type看作size_t写法的一个奇异方式。

stringLengthSum是accmulate使用的统计函数的代表。它带有到目前为止区间的统计值和区间的下一个元素，它返回新的统计值。一般来说，那意味着函数会带不同类型的参数。那就是这里所做的。到目前为止的统计值（已经看到的字符串的长度和）是string::size_type类型，而要检查的元素的类型是string。典型地在这个例子中，这里的返回类型和函数的第一个参数相同，因为它是更新了的统计值（加上了最后计算的元素）。

我们可以让accumulate这么使用stringLengthSum：

```
set<string> ss;                        // 建立字符串的容器，
...                                 // 进行一些操作
string::size_type lengthSum =              // 把lengthSum设为对
```

```
    accumulate(ss.begin(), ss.end(),          // ss中的每个元素调用
               0, stringLengthSum);  // stringLengthSum的结果，使用0
                                     // 作为初始统计值
```

很好，不是吗？计算数值区间的积甚至更简单，因为我们不用写自己的求和函数。我们可以使用标准multiplies仿函数类：

```
vector<float> vf;                          // 建立float的容器
...                                  // 进行一些操作
float product =                            // 把product设为对vf
    accumulate(vf.begin(), vf.end(),       // 中的每个元素调用
               1.0f, multiplies<float>());  // multiplies<float>的结果，用1.0f
                                     // 作为初始统计值
```

这里唯一需要小心的东西是记得把1（作为float，不是int！）作为初始统计值，而不是0。如果我们使用0作为开始值，结果会总是0，因为0乘以任何东西也是0，不是吗？

我们的最后一个例子有一些晦涩。它完成寻找point的区间的平均值，point看起来像这样：

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

求和函数应该是一个叫做PointAverage的仿函数类的对象，但在我们察看PointAverage之前，让我们看看它在调用accumulate中的使用方法：

```
list<Point> lp;
...
Point avg =                                // 对Ip中的point求平均值
    accumulate(lp.begin(), lp.end(),
               Point(0, 0), PointAverage());
```

简单而直接，我们最喜欢的方式。在这个例子中，初始和值是在原点的point对象，我们需要记得的是当计算区间的平均值时不要考虑那个点。

PointAverage通过记录它看到的point的个数和它们x和y部分的和的来工作。每次调用时，它更新那些值并返回目前检查过的point的平均坐标，因为它对于区间中的每个点只调用一次，它把x与y的和除以区间中的point的个数，忽略传给accumulate的初始point值，它就应该是这样：

```
class PointAverage:
        public binary_function<Point, Point, Point> {        // 参见条款40
public:
    PointAverage(): numPoints(0), xSum(0), ySum(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;
};
```

这工作得很好，而且仅因为我有时候和一些非常狂热的人打交道（他们中的很多都在标准委员会），所以我才会预见到它可能失败的STL实现。不过，PointAverage和标准的第26.4.1节第2段冲突，我知道你想起来了，那就是禁止传给accumulate的函数中有副作用。成员变量numPoints、xSum和ySum的修改造成了一个副作用，所以，技术上讲，我刚展示的代码会导致结果未定义。实际上，很难想象它无法工作，但当我这么写时我被险恶的语言律师包围着，所以我别无选择，只能在这个问题上写出难懂的条文。

那很好，因为它给我了一个机会来提起for_each，另一个可以用于统计区间而且没有accumulate那么多限制的算法。正如accumulate，for_each带有一个区间和一个函数（一般是一个函数对象）来调用区间中的每个元素，但传给for_each的函数只接收一个实参（当前的区间元素），而且当完成时for_each返回它的函数。（实际上，它返回它的函数的一个拷贝——参见条款38。）值得注意的是，传给（而且后来要返回）for_each的函数可能有副作用。

除了副作用问题，for_each和accumulate的不同主要在两个方面。首先，accumulate的名字表示它是一个产生区间统计的算法，for_each听起来好像你只是要对区间的每个元素进行一些操作，而且，当然，那是那个算法的主要应用。用for_each来统计一个区间是合法的，但是它没有accumulate清楚。

其次，accumulate直接返回那些我们想要的统计值，而for_each返回一个函数对象，我们必须从这个对象中提

取想要的统计信息。在C++里，那意味着我们必须给仿函数类添加一个成员函数，让我们找回我们追求的统计信息。

这又是最后一个例子，这次使用for_each而不是accumulate：

```
struct Point {...};                          // 同上
class PointAverage:
        public unary_function<Point, void> {        // 参见条款40
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;
```

就个人来说，我更喜欢用accumulate来统计，因为我认为它最清楚地表达了正在做什么，但是for_each也可以，而且不像accumulate，副作用的问题并不跟随for_each。两个算法都能用来统计区间。使用最适合你的那个。

你可能想知道为什么for_each的函数参数允许有副作用，而accumulate不允许。这是一个刺向STL心脏的探针问题。唉，尊敬的读者，有一些秘密总是在我们的知识范围之外。为什么accumulate和for_each之间有差别？我尚待听到一个令人信服的解释。

# 仿函数、仿函数类、函数等

无论喜欢或不喜欢，函数和类似函数的对象——*仿函数*——遍布STL。关联容器使用它们来使元素保持有序；find_if这样的算法使用它们来控制它们的行为；如果缺少它们，那么比如for_each和transform这样的组件就没有意义了；比如not1和bind2nd这样的适配器会积极地产生它们。

是的，在你看到的STL中的每个地方，你都可以看见仿函数和仿函数类。包括你的源代码中。如果不知道怎么写行为良好的仿函数就不可能有效地使用STL。由于这样的情况，本章的大部分专注于解释怎么使你的仿函数行为和STL期望的方式一样。但有一个条款，专注于不同的主题，那个条款肯定会受到因需要用ptr_fun、mem_fun和mem_fun_ref弄乱他们的代码而感到惊讶的人的关注。如果你喜欢，你可以从那个条款（条款41）开始，但请别以它为终止。一旦你了解了那些函数，你会需要剩下条款的信息来确认你的仿函数完全地配合它们和STL的其他部分。

# 条款38：把仿函数类设计为用于值传递

C和C++都不允许你真的把函数作为参数传递给其他函数。取而代之的是，你必须传*指针*给函数。比如，这里有一个标准库函数qsort的声明：

```
void qsort(void *base, size_t nmemb, size_t size,
           int (*cmpfcn)(const void*, const void*));
```

条款46解释了为什么sort算法一般来说是比qsort函数更好的选择，但在这里那不是问题。问题是qsort声明的参数cmpfcn。一旦你忽略了所有的星号，就可以清楚地看出作为cmpfcn传递的实参，一个指向函数的指针，是从调用端拷贝（也就是，值传递）给qsort。这是C和C++标准库都遵循的一般准则，也就是，函数指针是值传递。

STL函数对象在函数指针之后成型，所以STL中的习惯是当传给函数和从函数返回时函数对象也是值传递的（也就是拷贝）。最好的证据是标准的for_each声明，这个算法通过值传递获取和返回函数对象：

```
template<class InputIterator,
         class Function>
Function                        // 注意值返回
for_each(InputIterator first,
         InputIterator last,
         Function f);           // 注意值传递
```

实际上，值传递的情况并不是完全打不破的，因为for_each的调用者在调用点可以显式指定参数类型。比如，下面的代码可以使for_each通过引用传递和返回它的仿函数：

```
class DoSomething:
    public unary_function<int, void> {        // 条款40解释了这个基类
public:
    void operator()(int x) {...}
    ...
};
```

```
typedef deque<int>::iterator DequeIntIter;          // 方便的typedef
deque<int> di;
...
DoSomething d;                                      // 建立一个函数对象
...
for_each<DequeIntIter,                              // 调用for_each，参数
         DoSomething&>(di.begin(),                  // 类型是DequeIntIter
         di.end(),                                  // 和DoSomething&；
         d);                                        // 这迫使d按引用
                                                    // 传递和返回
```

但是STL的用户不能做这样的事，如果函数对象是引用传递，有些STL算法的实现甚至不能编译。在本条款的剩余部分，我会继续假设函数对象总是值传递。实际上，这事实上总是真的。

因为函数对象以值传递和返回，你的任务就是确保当那么传递（也就是拷贝）时你的函数对象行为良好。这暗示了两个东西。第一，你的函数对象应该很小。否则它们的拷贝会很昂贵。第二，你的函数对象必须单态（也就是，非多态）——它们不能用虚函数。那是因为派生类对象以值传递代入基类类型的参数会造成切割问题：在拷贝时，它们的派生部分被删除。（切割问题怎么影响你使用STL的另一个例子参见条款3。）

当然效率很重要，避免切割问题也是，但不是所有的仿函数都是小的、单态的。函数对象比真的函数优越的的原因之一是仿函数可以包含你需要的所有状态。有些函数对象自然会很重，保持传这样的仿函数给STL算法和传它们的函数版本一样容易是很重要的。

禁止多态仿函数是不切实际的。C++支持继承层次和动态绑定，这些特性在设计仿函数类和其他东西的时候一样有用。仿函数类如果缺少继承就像C++缺少“ ++ ”。的确有办法让大的和/或多态的函数对象仍然允许它们把以值传递仿函数的方式遍布STL。

这就是了。带着你要放进你的仿函数类的数据和/或多态，把它们移到另一个类中。然后给你的仿函数一个指向这个新类的指针。比如，如果你想要建立一个包含很多数据的多态仿函数类。

```
template<typename T>
class BPFC:                                         // BPFC = “ Big Polymorphic
    public                                          // Functor Class ”
        unary_function<T, void> {                   // 条款40解释了这个基类
private:
    Widget w;                                       // 这个类有很多数据，
    Int x;                                          // 所以用值传递
```

```
    ...                                 // 会影响效率

public:
    virtual void operator()(const T& val) const;   // 这是一个虚函数，
    ...                                 // 所以切割时会出问题
};
```

建立一个包含一个指向实现类的指针的小而单态的类，然后把所有数据和虚函数放到实现类：

```
template<typename T>                        // 用于修改的BPFC
class BPFCImpl
    public unary_function<T, void> {        // 的新实现类
private:
    Widget w;                               // 以前在BPFC里的所有数据
    int x;                                  // 现在在这里
    ...
    virtual ~BPFCImpl();                    // 多态类需要
                                            // 虚析构函数
    virtual void operator()(const T& val) const;
    friend class BPFC<T>;                   // 让BPFC可以访问这些数据
};

template<typename T>
class BPFC:                                 // 小的，单态版的BPFC
    public unary_function<T, void> {
private:
    BPFCImpl<T> *pImpl;                     // 这是BPFC唯一的数据

public:
    void operator()(const T& val) const     // 现在非虚；
    {                                       // 调用BPFCImpl的
        pImpl->operator() (val);
    }
    ...
};
```

BPFC::operator()的实现例证了BPFC所有的虚函数是怎么实现的：它们调用了在BPFCImpl中它们真的虚函数。结果是仿函数类（BPFC）是小而单态的，但可以访问大量状态而且行为多态。

我在这里忽略了很多细节，因为我勾勒出的基本技术在C++圈子中已经广为人知了。《Effective C++》的条款34中有。在Gamma等的《设计模式》[6]中，这叫做“ Bridge模式 ”。Sutter在他的《Exceptional C++》[8]中叫它“ Pimpl惯用法 ”.

从STL的视角看来，要记住的最重要的东西是使用这种技术的仿函数类必须支持合理方式的拷贝。如果你是上面BPFC的作者，你就必须保证它的拷贝构造函数对指向的BPFCImpl对象做了合理的事情。也许最简单的合理的东西是引用计数，使用类似Boost的shared_ptr，你可以在条款50中了解它.

实际上，对于本条款的目的，唯一你必须担心的是BPFC的拷贝构造函数的行为，因为当在STL中被传递或从一个函数返回时，函数对象总是被拷贝——值传递，记得吗？那意味着两件事。让它们小，而且让它们单态。

# 条款39：用纯函数做判断式

我讨厌为你做这些，但我们必须从一个简短的词汇课开始：

- *判断式*是返回bool（或者其他可以隐式转化为bool的东西）。判断式在STL中广泛使用。标准关联容器的比较函数是判断式，判断式函数常常作为参数传递给算法，比如find_if和多种排序算法。（排序算法的概览可以在条款31找到。）
- *纯函数*是返回值只依赖于参数的函数。如果f是一个纯函数，x和y是对象，f(x, y)的返回值仅当x或y的值改变的时候才会改变。
  在C++中，由纯函数引用的所有数据不是作为参数传进的就是在函数生存期内是常量。（一般，这样的常量应该声明为const。）如果一个纯函数引用的数据在不同次调用中可能改变，在不同的时候用同样的参数调用这个函数可能导致不同的结果，那就与纯函数的定义相反。

现在已经很清楚用纯函数作判断式是什么意思了。我要做的所有事情就是使你相信我的建议是有根据的。要帮我完成这件事，我希望你能原谅我再增加一个术语所给你带来的负担。

- 一个*判断式类*是一个仿函数类，它的operator()函数是一个判断式，也就是，它的operator()返回true或false（或其他可以隐式转换到true或false的东西）。正如你可以预料到的，任何STL想要一个判断式的地方，它都会接受一个真的判断式或一个判断式类对象。

就这些了，我保证！现在我们已经准备好学习为什么这个条款提供了有遵循价值的指引。

条款38解释了函数对象是传值，所以你应该设计可以拷贝的函数对象。用于判断式的函数对象，有另一个理由设计当它们拷贝时行为良好。算法可能拷贝仿函数，在使用前暂时保存它们，而且有些算法实现利用了这个自由。这个论点的一个重要结果是*判断式函数必须是纯函数*。

想知道这是为什么，先让我们假设你想要违反这个约束。考虑下面（坏的实现）的判断式类。不管传递的是什么实参，它严格地只返回一次true：第三次被调用的时候。其他时候它返回假。

```
class BadPredicate:                    // 关于这个基类的更多信息
    public unary_function<Widget, bool> {  // 请参见条款40
public:
    BadPredicate(): timesCalled(0) {}          // 把timesCalled初始化为0
    bool operator()(const Widget&)
```

```
    {
        return ++timesCalled == 3;
    }

private:
    size_t timesCalled;
};
```

假设我们用这个类来从一个vector<Widget>中除去第三个Widget：

```
vector<Widget> vw;                    // 建立vector，然后
                                      // 放一些Widgets进去
vw.erase(remove_if(vw.begin(),        // 去掉第三个Widget;
                   vw.end(),          // 关于erase和remove_if的关系
                   BadPredicate()),   // 请参见条款32
                   vw.end());
```

这段代码看起来很合理，但对于很多STL实现，它不仅会从vw中除去第三个元素，它也会除去第六个！

要知道这是怎么发生的，就该看看remove_if一般是怎么实现的。记住remove_if*不是一定要*这么实现：

```
template <typename FwdIterator, typename Predicate>
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);
    }
}
```

这段代码的细节不重要，但注意判断式p先传给find_if，后传给remove_copy_if。当然，在两种情况中，p是传值——是*拷贝*——到那些算法中的。（技术上说，这不需要是真的，但实际上，*是真的*。详细资料请参考条款38。）

最初调用remove_if（用户代码中要从vw中除去第三个元素的那次调用）建立一个匿名BadPredicate对象，它把内部的timesCalled成员清零。这个对象（在remove_if内部叫做p）然后被拷贝到find_if，所以find_if也接收了一个timesCalled等于0的BadPredicate对象。find_if“调用”那个对象直到它返回true，所以调用了三次，find_if然后返回控制权到remove_if。remove_if继续运行后面的调用remove_copy_if，传p的另一个拷贝作为一个判断式。但p的timesCalled成员仍然是0！find_if没有调用p，它调用的只是p的拷贝。结果，第三次remove_copy_if调用它的判断式，它也将会返回true。这就是为什么remove_if最终会从vw中删除两个Widgets而不是一个。

最简单的使你自己不摔跟头而进入语言陷阱的方法是在判断式类中把你的operator()函数声明为const。如果你这么做了，你的编译器不会让你改变任何类数据成员。

```
class BadPredicate:
    public unary_function<Widget, bool> {
public:
    bool operator()(const Widget&) const
    {
        return ++timesCalled == 3;          // 错误！在const成员函数中
    }                                       // 不能改变局部数据
};
```

因为这是避免我们刚测试过的问题的一个直截了当的方法，我几乎可以把本条款的题目改为“在判断式类中使operator()成为const”。但那走得不够远。甚至const成员函数可以访问multable数据成员、非const局部静态对象、非const类静态对象、名字空间域的非const对象和非const全局对象。一个设计良好的判断式类也保证它的operator()函数独立于任何那类对象。在判断式类中把operator()声明为const对于正确的行为来说*是必要的*，但*不够充分*。一个行为良好的operator()当然是const，但不只如此。它也得是一个纯函数。

本条款的前面，我强调了任何STL想要一个判断式的地方，它都会接受一个真的判断式或一个判断式类对象。它在两个方向上都是对的。在STL任何可以接受一个判断式类对象的地方，一个判断式函数（可能由ptr_fun改变——参见条款41）也是受欢迎的。你现在明白判断式类中的operator()函数应该是纯函数，所以这个限制也扩展到判断式函数。作为一个判断式，这个函数和从BadPredicate类产生的对象一样糟：

```
bool anotherBadPredicate(const Widget&, const Widget&)
{
    static int timesCalled = 0;             // 不！不！不！不！不！不！不！
    return ++timesCalled == 3;              // 判断式应该是纯函数，
}                                           // 纯函数没有状态
```

不管你怎么写你的判断式，它们都应该是纯函数。

# 条款40：使仿函数类可适配

假设我有一个Widget*指针的list和一个函数来决定这样的指针是否确定一个有趣的Widget：

```
list<Widget*> widgetPtrs;
bool isInteresting(const Widget *pw);
```

如果我要在list中找第一个指向有趣的Widget的指针，这很简单：

```
list<Widget*>::iterator i = find_if(widgetPtrs.begin(), widgetPtrs.end(),
                                    isInteresting);
if (i != widgetPtrs.end()) {
    ...                                 // 处理第一个
}                                       // 有趣的指向
                                        // Widget的指针
```

但如果我想要找第一个指向不有趣的Widget的指针，显而易见的方法却编译失败：

```
list<Widget*>::iterator i =
    find_if(widgetPtrs.begin(), widgetPtrs.end(),
            not1(isInteresting));           // 错误！不能编译
```

取而代之的是，我必须对isInteresting应用ptr_fun在应用not1之前：

```
list<Widget*>::iterator i =
    find_if(widgetPtrs.begin(), widgetPtrs.end(),
            not1(ptr_func(isInteresting)));     // 没问题
if (i != widgetPtrs.end()) {
    ...                                 // 处理第一个
}                                       // 指向Widget的指针
```

那会引出一些问题。为什么我必须在应用not1前对isInteresting应用ptr_fun？ptr_fun为我做了什么，怎么完成上面的工作的？

答案多少有些令人惊讶。ptr_fun做的唯一的事是使一些typedef有效。就是这样。not1需要这些typedef，这就是为什么可以把not1应用于ptr_fun，但不能直接对isInteresting应用not1。因为是低级的函数指针，isInteresting缺乏not1需要的typedef。

not1不是STL中唯一有那些要求的组件。四个标准函数适配器（not1、not2、bind1st和bind2nd）都需要存在某些typedef，一些其他人写的非标准STL兼容的适配器（比如来自SGI和Boost的——参见条款50）也需要。提供这些必要的typedef的函数对象称为*可适配*的，而缺乏那些typedef的函数对象不可适配。可适配的比不可适配的函数对象可以用于更多的场景，所以只要能做到你就应该使你的函数对象可适配。这不花费你任何东西，而它可以为你仿函数类的客户购买一个便利的世界。

我知道，我知道。我在卖弄，经常提及“某些typedef”而没有告诉你是什么。问题中的typedef是argument_type、first_argument_type、second_argument_type和result_type，但不是那么直截了当，因为不同类型仿函数类需要提供那些名字的不同子集。总的来说，除非你在写你自己的适配器（本书没有覆盖的主题），你才不需要知道任何关于那些typedef的事情。那是因为提供它们的正规方法是从一个基类，或，更精确地说，一个基结构，继承它们。operator()带一个实参的仿函数类，要继承的结构是std::unary_function。operator()带有两个实参的仿函数类，要继承的结构是std::binary_function。

好，简单来说，unary_function和binary_function是模板，所以你不能直接继承它们。取而代之的是，你必须从它们产生的类继承，而那就需要你指定一些类型实参。对于unary_function，你必须指定的是由你的仿函数类的operator()所带的参数的类型和它的返回类型。对于binary_function，你要指定三个类型：你的operator的第一个和第二个参数的类型，和你的operator地返回类型。

这里有两个例子：

```
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;
};
```

在两种情况下，注意传给unary_function或binary_function的类型与传给仿函数类的operator()和从那里返回的一样，虽然operator的返回类型作为最后一个实参被传递给unary_function或binary_function有一点古怪。

你可能注意到了MeetsThreshold是一个类，而WidgetNameCompare是一个结构。MeetsThreshold有内部状态（它的阈值数据成员），而类是封装那些信息的合理方法。WidgetNameCompare没有状态，因此不需要任何private的东西。所有东西都是public的仿函数类的作者经常把它们声明为struct而不是class，也许只因为可以避免在基类和operator()函数前面输入“ public ”。把这样的仿函数声明为class还是struct纯粹是一个个人风格问题。如果你仍然在精炼你的个人风格，想找一些仿效的对象，看看无状态STL自己的仿函数类（比如，less<T>、plus<T>等）一般写为struct。再看看WidgetNameCompare：

```
struct WidgetNameCompare:
    public std::binary_function<Widget, Widget, bool> {
    bool operator()(cost Widget& lhs, const Widget& rhs) const;
}
```

虽然operator的实参类型是const Widget&，但传给binary_function的是Widget。一般来说，传给unary_function或binary_function的非指针类型都去掉了const和引用。（不要问为什么。理由不很好也不很有趣。如果你真的想知道，写一些没有去掉它们的程序，然后去解剖编译器诊断结果。如果完成了这步，你*仍然*对这个问题感兴趣，访问boost.org（参见条款50）然后看看他们关于特性和函数对象适配器的工作。）

当operator()的参数是指针时这个规则变了。这里有一个和WidgetNameCompare相似的结构，但这个使用Widget*指针：

```
struct PtrWidgetNameCompare:
    public std::binary_function<const Widget*, const Widget*, bool> {
    bool operator()(const Widget* lhs, const Widget* rhs) const;
};
```

在这里，传给binary_function的类型和operator()所带的类型*一样*。用于带有或返回指针的仿函数的一般规则是传给unary_function或binary_function的类型是operator()带有或返回的类型。

不要忘记所有使用这些unary_function和binary_function基类基本理由的冗繁的文字。这些类提供函数对象适配

器需要的typedef，所以从那些类继承产生可适配的函数对象。那使我们这么做：

```
list<Widget> widgets;
...
list<Widget>::reverse_iterator i1 =                    // 找到最后一个不
    find_if(widgets.rbegin(), widgets.rend(),          // 适合阈值10的widget
            not1(MeetsThreshold<int>(10)));// （不管意味着什么）

Widget w(构造函数实参);
list<Widget>::iterator i2 =                            // 找到第一个在由
    find_if(widgets.begin(), widgets.end(),      // WidgetNameCompare定义
            bind2nd(WidgetNameCompare(), w);     // 的排序顺序上先于w的widget
```

如果我们没有把仿函数类继承自unary_function或binary_function，这些例子都不能编译，因为not1和bind2nd都只和可适配的函数对象合作。

STL函数对象模仿了C++函数，而一个C++函数只有一套参数类型和一个返回类型。结果，STL暗中假设每个仿函数类只有一个operator()函数，而且这个函数的参数和返回类型要被传给unary_function或binary_function（与我们刚讨论过的引用和指针类型的规则一致）。这意味着，虽然可能很诱人，但你不能通过建立一个单独的含有两个operator()函数的struct试图组合WidgetNameCompare和PtrWidgetNameCompare的功能。如果你那么做了，这个仿函数可能可以和最多一种它的调用形式（你传参数给binary_function的那个）适配，而一个只能一半适配的仿函数可能只比完全不能适配要好。

有时候有必要给一个仿函数类多个调用形式（因此得放弃可适配性），条款7、20、23和25给了这种情况的例子。但是那种仿函数类是例外，不是规则。可适配性是重要的，每次你写仿函数类时都应该努力促进它。

# 条款41：了解使用ptr_fun、mem_fun和mem_fun_ref的原因

ptr_fun/mem_fun/mem_fun_ref系列是什么意思的？有时候你必须使用这些函数，有时候不用，总之，它们是做什么的？它们似乎只是坐在那里，没用地挂在函数名周围就像不合身的外衣。它们不好输入，影响阅读，而且难以理解。这些东西是STL古董的附加例子（正如在条款10和18中描述的那样），或者只是一些标准委员会的成员因为太闲和扭曲的幽默感而强加给我们的语法玩笑？

冷静一下。虽然ptr_fun、mem_fun和mem_fun_ref的名字不太好听，但它们做了很重要的工作，而不是语法玩笑，这些函数的主要任务之一是覆盖C++固有的语法矛盾之一。

如果我有一个函数f和一个对象x，我希望在x上调用f，而且我在x的成员函数之外。C++给我三种不同的语法来实现这个调用：

```
f(x);            // 语法#1：当f是一个非成员函数
x.f();           // 语法#2：当f是一个成员函数
                 // 而且x是一个对象或一个对象的引用
p->f();          // 语法#3：当f是一个成员函数
                 // 而且p是一个对象的指针
```

现在，假设我有一个可以测试Widget的函数，

```
void test(Widget& w);       // 测试w，如果没通过
                            // 就标记为“failed”
```

而且我有一个Widget的容器：

```
vector<Widget> vw;              // vw容纳Widget
```

要测试vw中的每个Widget，我很显然可以这么使用for_each：

```
for_each(vw.begin(), vw.end(), test);  // 调用#1（可以编译）
```

但想象test是一个Widget的成员函数而不是一个非成员函数，也就是说，Widget支持自我测试：

```
class Widget {
public:
    void test();        // 进行自我测试；如果没通过
                        // 就把*this标记为“ failed ”
};
```

在一个完美的世界，我也将能使用for_each对vw中的每个对象调用Widget::test：

```
for_each(vw.begin(), vw.end(),
         &Widget::test);      // 调用#2（不能编译）
```

实际上，如果世界真的完美，我将也可以使用for_each来在Widget*指针的容器上调用Widget::test：

```
list<Widget*> lpw;                    // lpw容纳Widget的指针
for_each(lpw.begin(), lpw.end(),
         &Widget::test);      // 调用#3（也不能编译）
```

但是想想在这个完美的世界里必须发生的。在调用#1的for_each函数内部，我们以一个对象为参数调用一个非成员函数，因此我们必须使用语法#1。在调用#2的for_each函数内部，我们必须使用语法#2，因为我们有一个对象和一个成员函数。而调用#3的for_each函数内部，我们需要使用语法#3，因为我们将面对一个成员函数和一个对象的指针。因此我们需要三个不同版本的for_each，而那个世界将有多完美？

在我们拥有的世界上，我们只有一个版本的for_each。想一种实现不难：

```
template<typename InputIterator, typename Function>
Function for_each(InputIterator begin, InputIterator end, Function f)
{
    while (begin != end) f(*begin++);
}
```

这里，我强调当调用时for_each是用语法#1这个事实。这是STL里的一个普遍习惯：函数和函数对象总使用用于非成员函数的语法形式调用。这解释了为什么调用#1可以编译而调用#2和#3不。这是因为STL算法（包括for_each）牢牢地绑定在句法#1上，而且只有调用#1与那语法兼容。

也许现在清楚为什么mem_fun和mem_fun_ref存在了。它们让成员函数（通常必须使用句法#2或者#3来调用的）使用句法1调用。

mem_fun和mem_fun_ref完成这个的方式很简单，虽然如果你看一下这些函数之一的声明会稍微清楚些。它们是真的函数模板，而且存在mem_fun和mem_fun_ref模板的几个变体，对应于它们适配的不同的参数个数和常量性（或缺乏）的成员函数。看一个声明就足以理解事情怎样形成整体的：

```
template<typename R, typename C>          // 用于不带参数的non-const成员函数
mem_fun_t<R,C>                  // 的mem_fun声明。
mem_fun(R(C::*pmf)());            // C是类，R是被指向
                        // 的成员函数的返回类型
```

mem_fun带有一个到成员函数的指针，pmf，并返回一个mem_fun_t类型的对象。这是一个仿函数类，容纳成员函数指针并提供一个operator()，它调用指向在传给operator()的对象上的成员函数。例如，在这段代码中，

```
list<Widget*> lpw;                  // 同上
...
for_each(lpw.begin(), lpw.end(),
         mem_fun(&Widget::test));     // 这个现在可以编译了
```

for_each接受一个mem_fun_t类型的对象，持有一个Widget::test的指针。对于在lpw里的每个Widget*指针，for_each使用语法#1“调用”mem_func_t，而那个对象立刻在Widget*指针上使用句法#3调用Widget::test。

总的来说，mem_fun适配语法#3——也就是当和Widget*指针配合时Widget::test要求的——到语法1，也就是for_each用的。因此也不奇怪像mem_fun_t这样的类被称为*函数对象适配器*。知道这个不应该使你惊讶，完全类似上述的，mem_fun_ref函数适配语法#2到语法#1，并产生mem_fun_ref_t类型的适配器对象。

mem_fun和mem_fun_ref产生的对象不仅允许STL组件假设所有函数都使用单一的语法调用。它们也提供重要的typedef，正如ptr_fun产生的对象一样。条款40告诉了你这些typedef后面的故事，所以我将不在这里重复它。但是，这使我们能够理解为什么这个调用可以编译。

```
for_each(vw.begin(), vw.end(), test);       // 同上，调用#1；
                        // 这个可以编译
```

而这些不能：

```
for_each(vw.begin(), vw.end(), &Widget::test); // 同上，调用#2；
                                               // 不能编译
for_each(lpw.begin(), lpw.end(), &Widget::test);     // 同上，调用#3；
                                               // 不能编译
```

第一个调用（调用#1）传递一个真的函数，因此用于for_each时不需要适配它的调用语法；这个算法将固有地使用适当的语法调用它。而且，for_each没有使用ptr_fun增加的typedef，所以当把test传给for_each时不必使用ptr_fun。另一方面，增加的typedef不会造成任何损伤，所以这和上面的调用做相同的事情：

```
for_each(vw.begin(), vw.end(), ptr_fun(test)); // 可以编译，行为
                                               // 就像上面的调用#1
```

如果你关于什么时候使用ptr_fun什么时候不使用而感到困惑，那就考虑每当你传递一个函数给STL组件时都使用它。STL将不在乎，并且没有运行期的惩罚。可能出现的最坏的情况就是一些读你代码的人当看见不必要的ptr_fun使用时，可能会扬起眉毛。我认为，那有多让你操心依赖于你对扬起眉毛的敏感性。

一个与ptr_fun有关的可选策略是只有当你被迫时才使用它。如果当typedef是必要时你忽略了它，你的编译器将退回你的代码。然后你得返回去添加它。

mem_fun和mem_fun_ref的情况则完全不同。只要你传一个成员函数给STL组件，你就必须使用它们，因为，除了增加typedef（可能是或可能不是必须的）之外，它们把调用语法从一个通常用于成员函数的适配到在STL中到处使用的。当传递成员函数指针时如果你不使用它们，你的代码将永远不能编译。

现在只留下成员函数适配器的名字，而在这里，最后，我们有一个真实的STL历史产物。当对这些种适配器的需求开始变得明显时，建立STL的人们正专注于指针的容器。（这种容器的缺点在条款7、20和33描述，这看起来可能很惊人，但是记住指针的容器支持多态，而对象的容器不支持。）他们需要用于成员函数的适配器，所以他们选择了mem_fun。但很快他们意识到需要一个用于对象的容器的另一个适配器，所以他们使用了mem_fun_ref。是的，它非常不优雅，但是这些事情发生了。告诉我你从未给你的任何组件一个你过后意识到，呃，很难概括的名字……

# 条款42：确定less<T>表示operator<

正如所有了解零件（Widget）的人所知道的，Widget有重量和最高速度：

```
class Widget {
public:
    ...
    size_t weight() const;
    size_t maxSpeed() const;
    ...
};
```

此外众所周知的是给Widget排序的自然方法是按重量。用于Widget的operator<表现成这样：

```
bool operator<(const Widget& lhs, const Widget& rhs)
{
    return lhs.weight() < rhs.weight();
}
```

但是假设我们想建立一个按照最高速度排序Widget的multiset<Widget>。我们知道multiset<Widget>的默认比较函数是less<Widget>，而且我们知道默认的less<Widget>通过调用Widget的operator<来工作。鉴于这种情况，好像得到以最高速度排序的multiset<Widget>很明显一种方法是通过特化less<Widget>来切断less<Widget>和operator<之间的纽带，让它只关注Widget的最高速度：

```
template<>                              // 这是一个std::less
struct std::less<Widget>:                       // 的Widget的特化；
    public                               // 也是非常坏的主意
    std::binary_function<Widget,
                         Widget,        // 关于这个基类更多
                         bool> {        // 的信息参见条款40
    bool operator()(const Widget& lhs, const Widget& rhs) const
    {
```

```
        return lhs.maxSpeed() < rhs.maxSpeed();
    }
};
```

这看起来既有欠考虑又确实有欠考虑，但你想到的理由可能不欠考虑。你是不是很奇怪它完全可以编译？很多程序员指出上述不只是一个模板的特化，而且是在特化std namespace中的模板。“std不应该是神圣的，为库的实现保留，而且超出了一般程序员可以达到的范围？”他们问，“编译器不该拒绝这个干预C++不朽的运转的尝试吗？”他们很奇怪。

通常，试图修改std里的组件确实是禁止的（而且这么做通常被认为是行为未定义的范畴），但是在一些情况下，修补是允许的。具体来说，程序员被允许用自定义类型特化std内的模板。特化std模板的选择几乎总是更为优先，但很少发生，这确实合理的。例如，智能指针类的作者经常想让他们的类在排序的时候行为表现得像内建指针，因此用于智能指针类型的std::less特化并不罕见。例如下面内容，是Boost库的shared_ptr的一部分，你可以在条款7和50中获悉智能指针：

```
namespace std {
    template<typename T>                    // 这是一个用于boost::shared_ptr<T>
    struct less<boost::shared_ptr<T> >:             // 的std::less的特化
        public                          //  （boost是一个namespace）
        binary function<boost::shared_ptr<T>,
                        boost::shared_ptr<T>,// 这是惯例的
                        bool> {       // 基类（参见条款40）
        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对象内的
                                        // 内建指针
    };
}
```

这不过分，当然也没有什么奇怪的，因为这个less的特化仅仅在排序上保证智能指针的行为与它们的内建兄弟相同。哎，我们试验性的用于Widget的less特化却比较奇怪。

C++程序员可以被原谅存在某些假设。例如，他们假设拷贝构造函数拷贝。（就像条款8证明的，没有依照这个约定可能导致惊人的行为。）他们假设对一个对象取地址就会产生一个指向那个物体的指针。（转向条款18去了解当这不为真时将发生什么。）他们假设像bind1st和not2这样的适配器可能应用于函数对象。（条款40

解释了当不是这样时，事情是如何被破坏的。）他们假设operator+是加法（除了string，但是使用“+”表示字符串的连接已经有悠久的历史了），operator-是减法，operator==作比较。而且他们假设使用less等价于使用operator<。

operator<不仅是实现less的默认方式，它*还是程序员希望less做的*。让less做除operator<以外的事情是对程序员预期的无故破坏。它与所被称为“最小惊讶的原则”相反。它是冷淡的。它是低劣的。它是坏的。你不该那么做。

特别是当没有理由时。在STL中没有一个用到less的地方你不能指定一个不同的比较类型。回到我们原先以最高速度排序的multiset<Widget>的例子，我们要得到希望的结果所需的所有事情就是建立一个叫做*除了*less以外的几乎任何名字的仿函数类来对我们感兴趣的东西进行比较。嗨，这里有一个例子：

```
struct MaxSpeedCompare:
    public binary_function<Widget, Widget, bool> {
    bool operator()(const Widget& lhs, const Widget& rhs) const
    {
        return lhs.maxSpeed() < rhs.maxSpeed();
    }
};
```

要创造我们的multiset，我们使用MaxSpeedCompare作为比较类型，因此避免了默认比较类型的使用（当然也就是less<Widget>）：

```
multiset<Widget, MaxSpeedCompare> widgets;
```

这条代码确切地说出了它的意思。它建立了一个Widget的multiset，按照仿函数类MaxSpeedCompare所定义方法排序。

对比这个：

```
multiset<Widget> widgets;
```

这个表示widgets是一个以默认方式排序的Widget的multiset。在技术上，那表示它使用了less<Widget>，但是实际上每人都要假设那真的意味着它是按operator<来排序。

不要通过把less的定义当儿戏来误导那些程序员。如果你使用less（明确或者隐含），保证它表示operator<。如果你想要使用一些其他标准排序对象，建立一个特殊的*不叫做*less的仿函数类。它真的很简单。

条款42：确定less<T>表示operator<

# 使用STL编程

总结由容器、迭代器、算法和函数对象组成的STL是个惯例，但使用STL编程远不止那些。运用STL编程要知道什么时候使用循环，什么时候使用算法，什么时候使用容器成员函数。要知道equal_range什么时候是比lower_bound更好的搜索方式，要知道lower_bound什么时候比find更优越，要知道find什么时候击败equal_range。要知道怎么通过用仿函数替代做同一件事的函数以改进算法性能。要知道怎么避免不可移植或不能理解的代码。甚至要知道怎么读懂用上千个字符表示的编译器错误信息。要知道STL文档、STL扩展甚至完整的STL实现的因特网资源。

是的，使用STL编程需要知道很多东西。本章将给你很多你需要的这些知识。

# 条款43：尽量用算法调用代替手写循环

每个算法接受至少一对用来指示将被操作的对象区间的迭代器。比如，min_element可以找出此区间中的最小的值，而accumulate则对区间内的元素作某种形式的整体求和运算（参见条款37），partition将区间内的元素分割为满足和不满足某判决条件的两个部分（参见条款31）。当算法被执行时，它们必须检查指示给它的区间中的每个元素，并且是按你所期望的方式进行的：从区间的起始点循还到结束点。有一些算法，比如find和find_if，可能在遍历完成前就返回了，但即使是这些算法，内部都包含一个循环。毕竟，即使是find和find_if也必须在查看过了每个元素后，才能断定它们所寻找的元素在不在此区间内。

所以，算法内部是一个循环。此外，STL算法的广泛涉及面意味着很多你本来要用循环来实现的任务，现在可以改用算法来实现了。比如，如果你有一个支持重画的Widget类：

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

并且，你想重画一个list中的所有Widget对象，你可能会这样使用这样一个循环：

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

但是你也可以用for_each算法来完成：

```
for_each(lw.begin(), lw.end(),
```

```
    mem_fun_ref(&Widget::redraw));
```

对许多C++程序员而言，使用循环比调用算法的想法自然多了，并且读循环比弄懂mem_fun_ref的意义和取Widget::redraw的地址要舒服多了。但是，本条款将说明调用算法更可取。事实上，本条款将证明调用算法通常比手写的循环更优越。为什么？

有三个理由：

- 效率：算法通常比程序员产生的循环更高效。
- 正确性：写循环时比调用算法更容易产生错误。
- 可维护性：算法通常使代码比相应的显式循环更干净、更直观。

本条款的余下部分将对此予以例证。

从效率方面看，算法在三个方面可以打败显式循环，两个主要的，一个次要的。次要的包括消除了多余的计算。回头看一下我们刚才写的循环：

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

我已经加亮了循环终止测试语句，以强调每次循环，i都要与lw.end()作检查。也就是说，每次的循环，都要调用函数list::end。但我们不需要调用end()一次以上的，因为我们不准备修改这个list，end调用一次就够了。而我们转过来看一下算法调用，就可以看到只对end作了正确的求值次数：

```
for_each(lw.begin(), lw.end(),       // 调用lw.end()求值
     mem_fun_ref(&Widget::redraw)); // 只有一次
```

公平的说，STL的实现者知道begin和end（以及类似的函数，比如size）用得很频繁，所以他们尽可能把它们实现得最高效。几乎肯定会inline它们，并努力编码使得绝大部分编译器都可以通过将计算结果提到循环外（译注：频度优化的一种）来避免重复计算。然而，经验表明，这样的实现不是总可以成功的，而且当不成功时，对重复计算的避免足以让算法比手写的循环具有性能优势。

但这只是影响效率的次要因素。第一个主要影响因素是库的实现者可以利用他们知道容器的具体实现的优

势，用库的使用者无法采用的方式来优化遍历。比如，在deque中的对象通常存储在（内部的）一个或多个固定大小的数组上。基于指针的遍历比基于迭代器的遍历更快，但只有库的实现者可以使用基于指针的遍历，因为只有他们知道内部数组的大小以及如何从一个数组移向下一个。一些STL容器和算法的实现版本将它们的deque的内部数据结构引入了account，而且已经知道，这样的实现比“通常”的实现快20%。

这点不是说STL实现这为deque（或者其他特定的容器类型）做了优化，而是实现者对于他们的实现比你知道得更多，而且他们可以把这些知识用在算法实现上。如果你避开了算法调用，而只喜欢自己写循环，你就相当于放弃了得到任何他们所提供的特定实现优点的机会。

第二个主要的效率因素是，除了最微不足道的STL算法，所有的STL算法使用的计算机科学都比一般的C++程序员能拿得出来的算法复杂——有时候会复杂得多。几乎不可能被打败的sort及其同族算法（比如，stable_sort()，nth_element()等，参见条款31）；适用于有序区间的搜索算法（比如，binary_search，lower_bound等，参见条款34和35）也一样好；就算是很平凡的任务，比如从连续内存容器中除去一些对象，使用erase-remove惯用法都比绝大多数程序员写的循环更高效（参见条款9）。

如果算法的效率因素说服不了你，也许你更愿意接受基于正确性的考虑。写循环时，比较麻烦的事在于确保所使用的迭代器（a）有效，并且（b）指向你所期望的地方。举例来说，假设有一个数组（大概是因为遗留的C API——参见条款16），你想获得其中的每一个元素，把它加上41，然后将结果插入一个deque的前端。用循环，你可能这样写：（这是来自条款16的一个例子的变体）：

```
// C API：这个函数的参数是一个能放arraySize
// 个double的数组的指针，
// 函数向这个数组写入数据。它返回
// 写入double的个数
size_t fillArray(double *pArray, size_t arraySize);
double data[maxNumDoubles];          // 建立本地数组，
                                     // 大小是最大可能的大小

deque<double> d;                     // 建立deque，
...                                  // 把数据放进去

size_t numDoubles =
    fillArray(data, maxNumDoubles);  // 从API获取数组数据

for (size_t i = 0; i < numDoubles; ++i) {   // 对于每一个数据，
    d.insert(d.begin(), data[i] + 41);      // 在d的前端插入data[i]+41
}                                    // 这段代码有一个bug！
```

这可以执行，只要你能满意于插入的元素于在data中对应的元素是反序的。因为每次的插入点都是d.begin()，最后一个被插入的元素将位于deque的前端！

如果这不是你想要的（还是承认吧，它肯定不是），你可能想这样修改：

```
deque<double>::iterator insertLocation = d.begin();    // 记下d的
                                                       // 起始迭代器

for (size_t i = 0; i < numDoubles; ++i) {                  // 在insertLocation
    d.insert(insertLocation++, data[i] + 41);              // 插入data[i]+41，然后
}                                                          // insertLocation递增；
                                                           // 这段代码也有bug！
```

看起来象双赢，它不只是递增了指示插入位置的，还避免了循环每次对begin的调用；这就像我们前面讨论过的一样，消除了影响效率的次要因素。唉，这种方法陷入了另外一个的问题中：它导致了未定义的结果。每次调用deque::insert，都将导致所有指向deque内部的迭代器无效，包括上面的insertLocation。在第一次调用insert后，insertLocation就无效了，后面的循环迭代器可以直接让人发疯。

注意到这个问题后（可能得到了STLport调试模式的帮助，在条款50描述），你可能会这样做：

```
deque<double>::iterator insertLocation =
    d.begin();                                 // 同上

for (size_t i = 0; i < numDoubles; ++i) {                  // 每次调用insert后
    insertLocation =                                       // 都更新insertLocation
        d.insert(insertLocation, data[i] + 41);            // 以保证迭代器有效
    ++insertLocation;                                      // 然后递增它
}
```

这样的代码确实完成了你相要的功能，但回想一下费了多大劲才达到这一步！和调用算法transform对比一下：

```
// 把所有元素从data拷贝到d的前端
// 每个增加上41
transform(data, data + numDoubles,
          inserter(d, d.begin()),
          bind2nd(plus<double>(), 41));
```

这个“ bind2nd(plus<double>(), 41) ”可能会花上一些时间才能看明白（尤其是如果不常用STL的bind族的话），但是与迭代器相关的唯一烦扰就是指出源区间的起始点和结束点（而这从不会成为问题），并确保在目的区间的起始点上使用inserter（参见条款30）。实际上，为源区间和目的区间指出正确的初始迭代器通常都很容易，至少比确保循环体没有于无意中将需要持续使用的迭代器变得无效要容易得多。

难以正确实现循环的情况太多了，这个例子只是比较有代表性。因为在使用迭代器前，必须时刻关注它们是否被不正确地操纵或变得无效。要看忽略迭代器失效会导致的麻烦的另一个例子，转到条款9，那里描述了手写循环和调用erase只见的微妙之处。假设使用无效的迭代器会导致未定义的行为，又假设未定义的行为在开发和测试期间会有表现出令人讨厌的行为，为什么要冒不必要的危险？将迭代器扔给算法，让它们去考虑操纵迭代器时的各种诡异行为吧。

我已经解释了算法为什么可以比手写的循环更高效，也描述了为什么循环将艰难地穿行于与迭代器相关的荆棘丛中，而算法正避免了这一点。运气好的话，你现在已是一个算法的信徒了。然而运气是变化无常的，在我休息前，我想更确定些。因此，让我们继续行进到代码清晰性的议题。毕竟，最好软件应该是那些最清晰的、最容易懂的、能容易增强、维护和适用于新的环境的软件。虽然循环比较熟悉，但算法在这个长期的竞争中具有优势。

关键在于已知词汇的力量。在STL中约有70个算法的名字——总共超过100个不同的函数模板，每个重载都算一个。每个算法都完成一些精心定义的任务，而且有理由认为专业的C++程序员知道（或应该去看一下）每个算法都完成了什么。因此，当程序员调用transform时，他们认为对区间内的每个元素都施加了某个函数，而结果将被写到另外一个地方。当程序员调用replace_if时，他（她）知道区间内满足判定条件的对象都将被修改。当调用partition时，她（他）明白区间中所有满足判定条件的对象将被聚集在一起（参见条款31）。STL算法的名字传达了大量的语义信息，这使得它们比随机的循环清晰多了。

当你看见for、while或do的时候，你能知道的只是这是一种循环。如果要获得哪怕一点关于这个循环作了什么的信息，你就得审视它。算法则不用。一旦你看见调用一个算法，独一无二的名字勾勒出了它所作所为的轮廓。当然要了解它真正做了什么，你必须检查传给算法的实参，但这一般比去研究一个普通的循环结构要轻松得多。

明摆着，算法的名字暗示了其功能。“ for ”、“ while ”和“ do ”却做不到这一点。事实上，这一点对标准C语言或C++语言运行库的所有组件都成立。毫无疑问地，你能自己实现strlen， memset或bsearch，但你不会这么做。为什么不会？因为（1）已经有人帮你实现了它们，因此没必要你自己再做一遍；（2）名字是标准的，因此，每个人都知道它们做什么用的；和（3）你猜测程序库的实现者知道一些你不知道的关于效率方面的技巧，因此你不愿意错过熟练的程序库实现者可能能提供的优化。正如你不会去写strlen等函数的自己的版本，同样没道理用循环来实现出已存在的STL算法的等价版本。

我很希望故事就此结束，因为我认为这个收尾很有说服力的。唉，有句老话叫好事多磨。算法的名字比光溜

溜的循环有意义多了，这是事实，但使用循环更能让人明白加诸于迭代器上的操作。举例来说，假设想要找出vector中第一个比x大又比y小的元素。这是使用循环的实现：

```
vector<int> v;
int x, y;
...
vector<int>::iterator i = v.begin();    // 从v.begin()开始迭代，直到
for( ; i != v.end(); ++i) {              // 找到了适当的值或
 if (*i > x && *i < y) break;  // 到v.end()了
}
...                                      // i现在指向那个值或等于v.end()
```

将同样的逻辑传给find_if是可能的，但是需要使用一个非标准函数对象适配器，比如SGI的compose2[1]（参见条款50）：

```
vector<int>::iterator i =
 find_if(v.begin(), v.end(),             // 查找第一个find the first value val
    compose2(logical_and<bool>(),   // 使val > x
       bind2nd(greater<int>(), x),  // 和val < y都
       bind2nd(less<int>(), y)));   // 为真的值val
```

即使没使用非标准组件，许多程序员也会反对说它远不及循环清晰，我也不得不同意这个观点（参见条款47）。

find_if的调用可以不显得那么复杂，只要将测试的逻辑封装入一个独立的仿函数类中：

```
template<typename T>
class BetweenValues:
public unary_function<T, bool> {          // 参见条款40
public:
    BetweenValues(const T& lowValue,
        const T& highValue)               // 使构造函数保存上下界
        :lowVal(lowValue), highVal(highValue)
    {}
    bool operator()(const T& val) const    //返回val是否在保存的值之间
    {
```

```
        return val > lowVal && val < highVal;
    }

private:
    T lowVal;
    T highVal;
};
...
vector<int>::iterator i = find_if(v.begin(), v.end(),
                    BetweenValues<int>(x, y));
```

但这种方法有它自己的缺陷。首先，创建BetweenValues模板比写循环体要多出很多工作。就光数一下行数。循环体：1行；BetweenValues模板：24行。太不成比例了。第二，find_if正在找寻是什么的细节被从调用上完全割裂出去了，要想真的明白对find_if的这个调用，还必须得查看BetweenValues的定义，但BetweenValues一定被定义在调用find_if的函数之外。如果试图将BetweenValues声明在这个函数的调用内部，就像这样，

```
{ // 函数开始

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

你会发现这不能编译，因为模板不能声明在函数内部。如果试图把BetweenValues做成一个类而不是模板来避开这个限制：

```
{ // 函数开始
 ...
 class BetweenValues:
  public std::unary_function<int, bool> { ... };
 vector<int> iterator i =
  find_if(v.begin(), v.end(),
```

```
        BetweenValues(x, y));
  ...
}// 函数结束
```

你会发现仍然运气不佳，因为定义在函数内部的类是个局部类，而局部类不能绑定在模板的类型实参上（比如find_if所需要的仿函数类型）。很失望吧，仿函数类和仿函数类模板不能被定义在函数内部，不管它实现起来有多方便。

在算法调用与手写循环正在进行的较量中，关于代码清晰度的底线是：这完全取决于你想在循环里做的是什么。如果你要做的是算法已经提供了的，或者非常接近于它提供的，调用泛型算法更清晰。如果循环里要做的事非常简单，但调用算法时却需要使用绑定和适配器或者需要独立的仿函数类，你恐怕还是写循环比较好。最后，如果你在循环里做的事相当长或相当复杂，天平再次倾向于算法。因为长的、复杂的通常总应该封装入独立的函数。只要将循环体一封装入独立函数，你几乎总能找到方法将这个函数传给一个算法（通常是for_each），以使得最终代码直截了当。

如果你同意算法调用通常优于手写循环这个条款，并且如果你也同意条款5的区间成员函数优于循环调用单元素的成员函数，一个有趣的结论出现了：使用STL容器的C++精致程序中的循环比不使用STL的等价程序少多了。这是好事。只要能用高层次的术语——如insert、find和for_each，取代了低层次的词汇——如for、while和do，我们就提升了软件的抽象层次，并因此使得它更容易实现、文档化、增强和维护。

---

[1] 要了解更多关于compose2的信息，请参考SGI的STL网站[21]或Matt Austern的书《Generic Programming and the STL》（Addison-Wesley，1999）[4]

# 条款44：尽量用成员函数代替同名的算法

有些容器拥有和STL算法同名的成员函数。关联容器提供了count、find、lower_bound、upper_bound和equal_range，而list提供了remove、remove_if、unique、sort、merge和reverse。大多数情况下，你应该用成员函数代替算法。这样做有两个理由。首先，成员函数更快。其次，比起算法来，它们与容器结合得更好（尤其是关联容器）。那是因为同名的算法和成员函数通常并不是是一样的。

我们以对关联容器的实验开始。假如有一个set<int>，它容纳了一百万个元素，而你想找到元素727的第一个出现位置（如果存在的话）。这儿有两个最自然的方法来执行搜索：

```
set<int> s;                          // 建立set，放入1,000,000个数据
...

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()) ...
```

find成员函数运行花费对数时间，所以不管727是否存在于此set中，set::find只需执行不超过40次比较来查找它，而一般只需要大约20次。相反，find算法运行花费线性时间，所以如果727不在此set中，它需要执行1,000,000次比较。即使727在此set中，也平均需要执行500,000次比较来找到它。效率得分如下：

```
find成员：大约40（最坏的情况）至大约20（平均情况）
find算法：1,000,000（最坏的情况）至500,000（平均情况）
```

和高尔夫比赛一样，分值低的赢。正如你所见，这场比赛结果没什么可说的。

我必须对find成员函数所需的比较次数表示小小的谨慎，因为它有些依赖于关联容器的实现。绝大部分的实现是使用的红黑树——平衡树的一种——失衡度可能达到2。在这样的实现中，对一百万个元素的set进行搜索所需最多的比较次数是38次。但对绝大部分的搜索情况而言，只需要不超过22次。一个基于完全平衡树的实现绝不需要超过21次比较，但在实践中，完全平衡树的效率总的来说不如红黑树。这就是为什么大多数的STL实现都使用红黑树。

效率不是find成员函数和find算法间的唯一差别。正如条款19所解释的，STL算法判断两个对象是否相同的方法是检查的是它们是否相等，而关联容器是用等价来测试它们的“同一性”。因此，find算法搜索727用的是相等，而find成员函数用的是等价。相等和等价间的区别可能造成成功搜索和不成功搜索的区别。比如说，条款19演示了用find算法在关联容器搜索失败而用find成员函数却搜索成功的情况！因此，如果使用关联容器的话，你应该尽量使用成员函数形式的find、count、lower_bound等等，而不是同名的算法，因为这些成员函数版本提供了和其它成员函数一致的行为。由于相等和等价间的差别，算法不能提供这样的一致行为。

这一差别对map和multimap尤其明显，因为它们容纳的是对象对（pair object），而它们的成员函数只在意对象对的key部分。因此，count成员函数只统计key值匹配的对象对的数目（所谓“匹配”，自然是检测等价情况）；对象对的value部份被忽略。成员函数find、lower_bound、upper_bound和equal_range也是如此。但是如果你使用count算法，它的寻找将基于（a）相等和（b）对象对的全部组成部分；算法find、lower_bound等同样如此。要想让算法只关注于对象对的key部分，必须要跳出条款23描述的限制（那儿介绍了用相等测试代替等价测试的方法）。

另一方面，如果你真的关心效率，你可以采用条款23中的技巧，联合条款34中讲的对数时间搜索算法。相对于性能的提升，这只是一个很小的代价。再者，如果你*真的*在乎效率，你应该考虑非标准的散列容器（在条款25进行了描述），只是，你将再次面对相等和等价的区别。

对于标准的关联容器，选择成员函数而不是同名的算法有几个好处。首先，你得到的是对数时间而不是线性时间的性能。其次，你判断两个元素“相同”使用的是等价，这是关联容器的默认定义。第三，当操纵map和multimap时，你可以自动地只处理key值而不是(key, value)对。这三点给了优先使用成员函数完美的铁甲。

让我们转到list的与算法同名的成员函数身上。这里的故事几乎全部是关于效率的。每个被list作了特化的算法（remove、remove_if、unique、sort、merge和reverse）都要拷贝对象，而list的特别版本什么都没有拷贝；它们只是简单地操纵连接list的节点的指针。算法和成员函数的算法复杂度是相同的，但如果操纵指针比拷贝对象的代价小的话，list的版本应该提供更好的性能。

牢牢记住这一点很重要：list成员函数的行为和它们的算法兄弟的行为经常不相同。正如条款32所解释的，如果你真的想从容器中清除对象的话，调用remove、remove_if和unique算法后，必须紧接着调用erase函数；但list的remove、remove_if和unique成员函数真的去掉了元素，后面不需要接着调用erase。

在sort算法和list的sort成员函数间的一个重要区别是前者不能用于list。作为单纯的双向迭代器，list的迭代器不能传给sort算法。merge算法和list的merge成员函数之间也同样存在巨大差异。这个算法被限制为不能修改源范围，但list::merge总是修改它的宿主list。

所以，你明白了吧。当面临着STL算法和同名的容器成员函数间进行选择时，你应该尽量使用成员函数。几乎可以肯定它更高效，而且它看起来也和容器的惯常行为集成得更好。

# 条款45：注意count、find、binary_search、lower_bound、upper_bound和equal_range的区别

你要寻找什么，而且你有一个容器或者你有一个由迭代器划分出来的区间——你要找的东西就在里面。你要怎么完成搜索呢？你箭袋中的箭有这些：count、count_if、find、find_if、binary_search、lower_bound、upper_bound和equal_range。面对着它们，你要怎么做出选择？

简单。你寻找的是能又快又简单的东西。越快越简单的越好。

暂时，我假设你有一对指定了搜索区间的迭代器。然后，我会考虑到你有的是一个容器而不是一个区间的情况。

要选择搜索策略，必须依赖于你的迭代器是否定义了一个有序区间。如果是，你就可以通过binary_search、lower_bound、upper_bound和equal_range来加速（通常是对数时间——参见条款34）搜索。如果迭代器并没有划分一个有序区间，你就只能用线性时间的算法count、count_if、find和find_if。在下文中，我会忽略掉count和find是否有_if的不同，就像我会忽略掉binary_search、lower_bound、upper_bound和equal_range是否带有判断式的不同。你是依赖默认的搜索谓词还是指定一个自己的，对选择搜索算法的考虑是一样的。

如果你有一个无序区间，你的选择是count或着find。它们分别可以回答略微不同的问题，所以值得仔细去区分它们。count回答的问题是：“ 是否存在这个值，如果有，那么存在几份拷贝？ ”而find回答的问题是：“ 是否存在，如果有，那么它在哪儿？ ”

假设你想知道的东西是，是否有一个特定的Widget值w在list中。如果用count，代码看起来像这样：

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

这里示范了一种惯用法：把count用来作为是否存在的检查。count返回零或者一个正数，所以我们把非零转化为true而把零转化为false。如果这样能使我们要做的更加显而易见：

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

而且有些程序员这样写，但是使用隐式转换则更常见，就像最初的例子。

和最初的代码比较，使用find略微更难懂些，因为你必须检查find的返回值和list的end迭代器是否相等：

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

如果是为了检查是否存在，count这个惯用法编码起来比较简单。但是，当搜索成功时，它的效率比较低，因为当找到匹配的值后find就停止了，而count必须继续搜索，直到区间的结尾以寻找其他匹配的值。对大多数程序员来说，find在效率上的优势足以证明略微增加复杂度是合适的。

通常，只知道区间内是否有某个值是不够的。取而代之的是，你想获得区间中的第一个等于该值的对象。比如，你可能想打印出这个对象，你可能想在它前面插入什么，或者你可能想要删除它（但当迭代时删除的引导参见条款9）。当你需要知道的不止是某个值是否存在，而且要知道哪个对象（或哪些对象）拥有该值，你就得用find：

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

对于有序区间，你有其他的选择，而且你应该明确的使用它们。count和find是线性时间的，但有序区间的搜索算法（binary_search、lower_bound、upper_bound和equal_range）是对数时间的。

从无序区间迁移到有序区间导致了另一个迁移：从使用相等来判断两个值是否相同到使用等价来判断。条款19由一个详细地讲述了相等和等价的区别，所以我在这里不会重复。取而代之的是，我会简单地说明count和

find算法都用相等来搜索，而binary_search、lower_bound、upper_bound和equal_range则用等价。

要测试在有序区间中是否存在一个值，使用binary_search。不像标准C库中的（因此也是标准C++库中的）bsearch，binary_search只返回一个bool：这个值是否找到了。binary_search回答这个问题：“它在吗？”它的回答只能是是或者否。如果你需要比这样更多的信息，你需要一个不同的算法。

这里有一个binary_search应用于有序vector的例子（你可以从条款23中知道有序vector的优点）：

```
vector<Widget> vw;                  // 建立vector，放入
...                                 // 数据，
sort(vw.begin(), vw.end());         // 把数据排序
Widget w;                           // 要找的值
...
if (binary_search(vw.begin(), vw.end(), w)) {
    ...                             // w在vw中
} else {
    ...                             // 不在
}
```

如果你有一个有序区间而且你的问题是：“它在吗，如果是，那么在哪儿？”你就需要equal_range，但你可能想要用lower_bound。我会很快讨论equal_range，但首先，让我们看看怎么用lower_bound来在区间中定位某个值。

当你用lower_bound来寻找一个值的时候，它返回一个迭代器，这个迭代器指向这个值的第一个拷贝（如果找到的话）或者到可以插入这个值的位置（如果没找到）。因此lower_bound回答这个问题：“它在吗？如果是，第一个拷贝在哪里？如果不是，它将在哪里？”和find一样，你必须测试lower_bound的结果，来看看它是否指向你要寻找的值。但又不像find，你不能只是检测lower_bound的返回值是否等于end迭代器。取而代之的是，你必须检测lower_bound所标示出的对象是不是你需要的值。

很多程序员这么用lower_bound：

```
vector<Widget>::iterator i = lower_bound(vw.begin(), vw.end(), w);
if (i != vw.end() && *i == w) { // 保证i指向一个对象；
                                // 也就保证了这个对象有正确的值。
                                // 这是个bug！
    ...                         // 找到这个值，i指向
                                // 第一个等于该值的对象
} else {
```

```
    ...                 // 没找到
}
```

大部分情况下这是行得通的，但不是真的完全正确。再看一遍检测需要的值是否找到的代码：

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

这是一个*相*等的测试，但lower_bound搜索用的是等*价*。大部分情况下，等价测试和相等测试产生的结果相同，但就像条款19论证的，相等和等价的结果不同的情况并不难见到。在这种情况下，上面的代码就是错的。

要完全完成，你就必须检测lower_bound返回的迭代器指向的对象的值是否和你要寻找的值等价。你可以手动完成（条款19演示了你该怎么做，当它值得一做时条款24提供了一个例子），但可以更狡猾地完成，因为你必须确认使用了和lower_bound使用的相同的比较函数。一般而言，那可以是一个任意的函数（或函数对象）。如果你传递一个比较函数给lower_bound，你必须确认和你的手写的等价检测代码使用了相同的比较函数。这意味着如果你改变了你传递给lower_bound的比较函数，你也得对你的等价检测部分作出修改。保持比较函数同步不是火箭发射，但却是另一个要记住的东西，而且我想你已经有很多需要你记的东西了。

这儿有一个简单的方法：使用equal_range。equal_range返回一对迭代器，第一个等于lower_bound返回的迭代器，第二个等于upper_bound返回的（也就是，等价于要搜索值区间的末迭代器的下一个）。因此，equal_range，返回了一对划分出了和你要搜索的值等价的区间的迭代器。一个名字很好的算法，不是吗？（当然，也许叫equivalent_range会更好，但叫equal_range也非常好。）

对于equal_range的返回值，有两个重要的地方。第一，如果这两个迭代器相同，就意味着对象的区间是空的；这个只没有找到。这个结果是用equal_range来回答“它在吗？”这个问题的答案。你可以这么用：

```
vector<Widget> vw;
...
sort(vw.begin(), vw.end());
typedef vector<Widget>::iterator VWIter;      // 方便的typedef
typedef pair<VWIter, VWIter> VWIterPair;
VWIterPair p = equal_range(vw.begin(), vw.end(), w);
if (p.first != p.second) {                    // 如果equal_range不返回
                          // 空的区间...
    ...                   // 说明找到了，p.first指向
                          // 第一个而p.second
                          // 指向最后一个的下一个
```

```
} else {
    ...                    // 没找到，p.first和
                           // p.second都指向搜索值
}                          // 的插入位置
```

这段代码只用等价，所以总是正确的。

第二个要注意的是equal_range返回的东西是两个迭代器，对它们作distance就等于区间中对象的数目，也就是，等价于要寻找的值的对象。结果，equal_range不光完成了搜索有序区间的任务，而且完成了计数。比如说，要在vw中找到等价于w的Widget，然后打印出来有多少这样的Widget存在，你可以这么做：

```
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，我们所讨论的都是假设我们要在一个区间内搜索一个值，但是有时候我们更感兴趣于在区间中寻找一个位置。比如，假设我们有一个Timestamp类和一个Timestamp的vector，它按照老的timestamp放在前面的方法排序：

```
class Timestamp { ... };
bool operator<(const Timestamp& lhs,       // 返回在时间上lhs
    const Timestamp& rhs);       // 是否在rhs前面
vector<Timestamp> vt;            // 建立vector，填充数据，
...                              // 排序，使老的时间
sort(vt.begin(), vt.end());              // 在新的前面
```

现在假设我们有一个特殊的timestamp——ageLimit，而且我们从vt中删除所有比ageLimit老的timestamp。在这种情况下，我们不需要在vt中搜索和ageLimit等价的Timestamp，因为可能不存在任何等价于这个精确值的元素。 取而代之的是，我们需要在vt中找到一个位置：第一个不比ageLimit更老的元素。这是再简单不过的了，因为lower_bound会给我们答案的：

```
Timestamp ageLimit;
...
vt.erase(vt.begin(), lower_bound(vt.begin(),   // 从vt中排除所有
    vt.end(),                    // 排在ageLimit的值
    ageLimit));              // 前面的对象
```

如果我们的需求稍微改变了一点，我们要排除所有至少和ageLimit一样老的timestamp，也就是我们需要找到第一个比ageLimit年轻的timestamp的位置。这是一个为upper_bound特制的任务：

```
vt.erase(vt.begin(), upper_bound(vt.begin(),   // 从vt中除去所有
    vt.end(),                    // 排在ageLimit的值前面
    ageLimit));              // 或者等价的对象
```

如果你要把东西插入一个有序区间，而且对象的插入位置是在有序的等价关系下它应该在的地方时，upper_bound也很有用。比如，你可能有一个有序的Person对象的list，对象按照name排序：

```
class Person {
public:
    ...
    const string& name() const;
    ...
};

struct PersonNameLess:
public binary_function<Person, Person, bool> {  // 参见条款40
    bool operator()(const Person& lhs, const Person& rhs) const
    {
        return lhs.name() < rhs.name();
    }
};

list<Person> lp;
...
lp.sort(PersonNameLess());              // 使用PersonNameLess排序lp
```

要保持list仍然是我们希望的顺序（按照name，插入后等价的名字仍然按顺序排列），我们可以用upper_bound来指定插入位置：

```
Person newPerson;
...
lp.insert(upper_bound(lp.begin(),          // 在lp中排在newPerson
    lp.end(),                    // 之前或者等价
```

```
    newPerson,              // 的最后一个
    PersonNameLess()),              // 对象后面
    newPerson);             // 插入newPerson
```

这工作的很好而且很方便，但很重要的是不要被误导——错误地认为upper_bound的这种用法让我们魔术般地在一个list里在对数时间内找到了插入位置。我们并没有——条款34解释了因为我们用了list，查找花费线性时间，但是它只用了对数次的比较。

一直到这里，我都只考虑我们有一对定义了搜索区间的迭代器的情况。通常我们有一个容器，而不是一个区间。在这种情况下，我们必须区别序列和关联容器。对于标准的序列容器（vector、string、deque和list），你应该遵循我在本条款提出的建议，使用容器的begin和end迭代器来划分出区间。

这种情况对标准关联容器（set、multiset、map和multimap）来说是不同的，因为它们提供了搜索的成员函数，它们往往是比用STL算法更好的选择。条款44详细说明了为什么它们是更好的选择，简要地说，是因为它们更快行为更自然。幸运的是，成员函数通常和相应的算法有同样的名字，所以前面的讨论推荐你使用的算法count、find、equal_range、lower_bound或upper_bound，在搜索关联容器时你都可以简单的用同名的成员函数来代替。

调用binary_search的策略不同，因为这个算法没有提供对应的成员函数。要测试在set或map中是否存在某个值，使用count的惯用方法来对成员进行检测：

```
set<Widget> s;      // 建立set，放入数据
...
Widget w;               // w仍然是保存要搜索的值
...
if (s.count(w)) {
    ...         // 存在和w等价的值
} else {
    ...         // 不存在这样的值
}
```

要测试某个值在multiset或multimap中是否存在，find往往比count好，因为一旦找到等于期望值的单个对象，find就可以停下了，而count，在最遭的情况下，必须检测容器里的每一个对象。（对于set和map，这不是问题，因为set不允许重复的值，而map不允许重复的键。）

但是，count给关联容器计数是可靠的。特别，它比调用equal_range然后应用distance到结果迭代器更好。首先，它更清晰：count 意味着“ 计数 ”。第二，它更简单；不用建立一对迭代器然后把它的组成（译注：就

是first和second）传给distance。第三，它可能更快一点。

要给出所有我们在本条款中所考虑到的，我们的从哪儿着手？下面的表格道出了一切。

| 你想知道的 | 使用的算法 | | 使用的成员函数 | |
|---|---|---|---|---|
| | 在无序区间 | 在有序区间 | 在set或map上 | 在multiset或multimap上 |
| 期望值是否存在？ | find | binary_search | count | find |
| 期望值是否存在？如果有，第一个等于这个值的对象在哪里？ | find | equal_range | find | find或lower_bound（参见下面） |
| 第一个不在期望值之前的对象在哪里？ | find_if | lower_bound | lower_bound | lower_bound |
| 第一个在期望值之后的对象在哪里？ | find_if | upper_bound | upper_bound | upper_bound |
| 有多少对象等于期望值？ | count | equal_range，然后distance | count | count |
| 等于期望值的所有对象在哪里？ | find（迭代） | equal_range | equal_range | equal_range |

上表总结了要怎么操作有序区间，equal_range的出现频率可能令人吃惊。当搜索时，这个频率因为等价检测的重要性而上升了。对于lower_bound和upper_bound，它很容易在相等检测中退却，但对于equal_range，只检测等价是很自然的。在第二行有序区间，equal_range打败了find还因为一个理由：equal_range花费对数时间，而find花费线性时间。

对于multiset和multimap，当你在搜索第一个等于特定值的对象的那一行，这个表列出了find和lower_bound两个算法作为候选。 已对于这个任务find是通常的选择，而且你可能已经注意到在set和map那一列里，这项只有find。但是对于multi容器，如果不只有一个值存在，find并不保证能识别出容器里的等于给定值的第一个元素；它只识别这些元素中的一个。如果你真的需要找到等于给定值的第一个元素，你应该使用lower_bound，而且你必须手动的对第二部分做等价检测，条款19的内容可以帮你确认你已经找到了你要找的值。（你可以用equal_range来避免作手动等价检测，但是调用equal_range的花费比调用lower_bound多得多。）

在count、find、binary_search、lower_bound、upper_bound和equal_range中做出选择很简单。当你调用时，选择算法还是成员函数可以给你需要的行为和性能，而且是最少的工作。按照这个建议做（或参考那个表格），你就不会再有困惑。

# 条款46：考虑使用函数对象代替函数作算法的参数

一个关于用高级语言编程的抱怨是抽象层次越高，产生的代码效率就越低。事实上，Alexander Stepanov（STL的发明者）有一次作了一组小测试，试图测量出相对C，C++的“抽象惩罚”。其中，测试结果显示基本上操作包含一个double的类产生的代码效率比对应的直接操作一个double的代码低。因此你可能会奇怪地发现把STL函数对象——化装成函数的对象——传递给算法所产生的代码一般比传递真的函数高效。

比如，假设你需要以降序排序一个double的vector。最直接的STL方式是通过sort算法和greater<double>类型的函数对象：

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

如果你注意了抽象惩罚，你可能决定避开函数对象而使用真的函数，一个不光是真的，而且是内联的函数：

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

有趣的是，如果你比较了两个sort调用（一个使用greater<double>，一个使用doubleGreater）的性能，你基本上可以明确地知道使用greater<double>的那个更快。实际来说，我在四个不同的STL平台上测量了对一百万个double的vector的两个sort调用，每个都设为针对速度优化，使用greater<double>的版本每次都比较快。最差的情况下，快50%，最好的快了160%。抽象惩罚真多啊。

这个行为的解释很简单：内联。如果一个函数对象的operator()函数被声明为内联（不管显式地通过inline或者隐式地通过定义在它的类定义中），编译器就可以获得那个函数的函数体，而且大部分编译器喜欢在调用算法的模板实例化时内联那个函数。在上面的例子中，greater<double>::operator()是一个内联函数，所以编译器在实例化sort时内联展开它。结果，sort没有包含一次函数调用，而且编译器可以对这个没有调用操作的代码

进行其他情况下不经常进行的优化。（内联和编译器优化之间交互的讨论，参见《Effective C++》的条款33和Bulka与Mayhew的《Efficient C++》[10]的第8-10章。）

使用doubleGreater调用sort的情况则不同。要知道有什么不同，我们必须知道不可能把一个函数作为参数传给另一个函数。当我们试图把一个函数作为参数时，编译器默默地把函数转化为一个指向那个函数的*指针*，而那个指针是我们真正传递的。因此，这个调用

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

不是真的把doubleGreater传给sort，它传了一个doubleGreater的指针。当sort模板实例化时，这是产生函数的声明：

```
void sort(vector<double>::iterator first,                    // 区间起点
          vector<double>::iterator last,         // 区间终点
          bool (*comp)(double, double));         // 比较函数
```

因为comp是一个指向函数的指针，每次在sort中用到时，编译器产生一个间接函数调用——通过指针调用。大部分编译器不会试图去内联通过函数指针调用的函数，甚至，正如本例中，那个函数已经声明为inline而且这个优化看起来很直接。为什么不？可能因为编译器厂商从来没有觉得值得实现这个优化。你得稍微同情一下编译器厂商。他们有很多需求，而他们不能做每一件事。你的需要并不能让他们实现那个优化。

把函数指针作为参数会抑制内联的事实解释了一个长期使用C的程序员经常发现却难以相信的现象：在速度上，C++的sort实际上总是使C的qsort感到窘迫。当然，C++有函数、实例化的类模板和看起来很有趣的operator()函数需要调用，而C只是进行简单的函数调用，但所有的C++“开销”都在编译期被吸收。在运行期，sort内联调用它的比较函数（假设比较函数已经被声明为inline而且它的函数体在编译期可以得到）而qsort通过一个指针调用它的比较函数。结果是sort运行得更快。在我的测试的一百万个double的vector上，它快670%，但不要光相信我的话，亲自试试。很容易证实当比较函数对象和真的函数作为算法的参数时，那是一个纯红利。

还有另一个使用函数对象代替函数作为算法参数的理由，而它与效率无关。它涉及到让你的程序可以编译。对于任何理由，STL平台经常完全拒绝有效代码，即使编译器或库或两者都没问题。比如，一个广泛使用的STL平台拒绝了下面（有效的）代码来从cout打印出set中每个字符串的长度：

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

```
            mem_fun_ref(&string::size));
```

这个问题的原因是这个特定的STL平台在处理const成员函数时有bug（比如string::size）。一个变通方法是改用函数对象：

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

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

解决这问题的还有其他变通办法，但这个不仅在我知道的每个STL平台都可以编译。而且它简化了string::size的内联调用，那是几乎不会在上面把mem_fun_ref(&string::size)传给transform的代码中发生的事情。换句话说，仿函数类StringSize的创造不仅避开了编译器一致性问题，而且可能会带来性能提升。

另一个用函数对象代替函数的原因是它们可以帮助你避免细微的语言陷阱。有时候，看起来合理代码被编译器由于合法性的原因——但很模糊——而拒绝。有很多这样的情况，比如，当一个函数模板实例化的名字不等价于函数名。这是一种这样的情况：

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

template<typename InputIter1,
         typename InputIter2>
void writeAverages(InputIter1 begin1,              // 把两个序列的
                   InputIter1 end1,               // 成对的平均值
                   InputIter2 begin2,             // 写入一个流
                   ostream& s)
```

```
{
    transform(
        begin1, end1, begin2,
        ostream_iterator<typename iterator_traits
        <lnputIter1>::value_type> (s, "\n"),
        average<typename iteraror_traits
                <InputIter1>::value_type>     // 有错？
    );
}
```

很多编译器接受这段代码，但是C++标准倾向于禁止它。理由是理论上有可能存在另一带有一个类型参数的函数模板叫做average。如果有，表达式average<typename iterator_traits<lnputIter1>::value_type>将是模棱两可的，因为它不清楚将实例化哪个模板。在这个特殊的例子里，不存在模棱两可，但是无论如何，有些编译器会拒绝这段代码，而且那么做是允许的。没有关系。解决这个问题的方法是依靠一个函数对象：

```
template<typename FPType>
struct Average:
    public binary_function<FPType, FPType, FPType>{      // 参见条款40
FPType operator()(FPType val1. FPType val2) const
{
    return average(val 1 , val2);
}

template<typename InputIter1, typename InputIter2>
void writeAverages(lnputIter1 begin1, InputIter1 end1,
                   InputIter2 begin2, ostream& s)
{
    transform(
        begin1, end1, begin2,
        ostream_iterator<typename iterator_traits<lnputIter1>
                         ::value_type>(s, "\n"),
        Average<typename iterator_traits<InputIter1>
                ::value_type>()
    );
}
```

每个编译器都会接受这条修正的代码。而且在transform里调用Average::operator()符合内联的条件，有些事情对于实例化上面的average不成立，因为averate是一个函数模板，不是函数*对象*。

把函数对象作为算法的参数所带来的不仅是巨大的效率提升。在让你的代码可以编译方面，它们也更稳健。当然，真函数很有用，但是当涉及有效的STL编程时，函数对象经常更有用。

# 条款47：避免产生只写代码

假设你有一个vector<int>，你想去掉vector中值小于x而出现在至少和y一样大的最后一个元素之后的所有元素。下面代码立刻出现在你脑中吗？

```
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());
```

一条语句就完成了工作。清楚并直接了当。没问题。对吗？

好，让我们先退回一步。对你来说它是合理的、好维护的代码？“不！”大部分C++程序员惊叫，他们的声音中充满了害怕和讨厌。“是！”他们中的一部分显然高兴地说。但那儿有一个问题。一个程序员很有表现力的方法是另一个程序员的噩梦。

正如我所见，有上面代码涉及到两个问题。首先，它是函数调用的鼠巢。要知道我是什么意思，这里有一个相同的语句，但所有的函数名都替换为fn，每个n对应一个函数：

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

这看起来不自然地复杂，因为我去掉了出现在原例中的缩进，但我想它可以很明白的表示了如果一条语句使用了12次函数调用，分别调用了10个不同的函数，那会被大部分C++程序开发人员认为很过分。但曾经用过比如Scheme的函数式语言（functional language）的程序员可能感觉不同，而我的经验是大多数看到原来的代码而没有感到惊讶的程序员有一个很强的函数式语言背景。大部分C++程序员缺乏这样的背景，所以除非你的同事精通深度嵌套（nest）的函数调用的方法，类似上面erase调用的代码几乎可以肯定会打败下一个想要弄明白你写的代码的人。

这段代码的第二个缺点是需要很多STL背景才能明白它。它使用了不常见的_if形式的find和remove，它使用了逆向迭代器（参见条款26），它把reverse_iterator转换为iterator（参见条款28），它使用了bind2nd，它建立了匿名函数对象而且它使用了erase-remove惯用法（参见条款32）。有经验的STL程序员可以轻易地吞下那些组合，但更多C++开发者在一口吃掉之前会一遍一遍看。如果你的同事对在一条语句中使用erase、remove_if、find_if、base和bind2ndSTL的这种方法都很熟悉，那可能很好，但如果你想让你的代码易于被主流背景更多的C++程序员了解，我鼓励你把它分解为易于消化的块。

这是一种你可以使用的方法。（注释不光为了本书。我也会把它们放入代码。）

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

// 把rangeBegin初始化为指向最后一个
// 出现一个大于等于y的值后面的元素。
// 如果没有那样的值，
// 把rangeBegin初始化为v.begin()。如果那个值的
// 最后一次出现是v中的最后一个元素，
// 就把rangeBegin初始化为v.end()。
VecIntIter rangeBegin = find_if(v.rbegin(), v.rend(),
                                bind2nd(greater_equal<int>(), y)).base();

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

这仍然可能把一些人弄糊涂，因为它依赖于对erase-remove惯用法的了解，但在代码的注释和一本好的STL参考（比如，Josuttis的《The C++ Standard Library》[3]或SGI的STL网站[21])之间，每个C++程序员应该可以不太困难地指出它作了什么。

当转换代码时，要注意我并没有放弃算法并试图自己写的循环。条款43解释了为什么那一般是一个劣等的选择，它的论点在这里。当写源代码时，目标是写出对编译器和人都有意义的代码，并提供可接受的性能。算法基本上总是最好地达到那个目标的方式。但是，条款43也解释了增加算法的时候会自然导致增加嵌套函数调用并大量使用绑定器和其他仿函数适配器。再看看本条款开头的问题描述：

> 假设你有一个vector<int>，你想去掉vector中值小于x而出现在至少和y一样大的最后一个元素之后的所有元素。

一个方案的轮廓跳入脑中：

- 找到vector中一个值的最后一次出现需要用逆向迭代器调用find或find_if的某种形式。
- 去掉元素需要调用erase或erase-remove惯用法。

把这两个想法加在一起，你可以得到这个伪代码，其中“ something ”表示一个用于还没有填充的代码的占位符：

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

一旦你完成了这个，确定something就不是很难了，下一个事情你是知道的，你有原例中的代码。那是为什么这种语句一般被称为“ 只写（write-only） ”代码。当你写代码时，它似乎很直截了当，因为它是一些基本想法（也就是，erase-remove惯用法加上使用逆向迭代器调用find的想法）的自然产物。但是，*读者们*很难把最后的产物分解回它基于的想法。这就被称为只写代码：很容易写，但很难读和理解。

代码是否是只写依赖于读它的人。正如我提到的，有些C++程序员不反感本条款中的代码。如果这是你工作环境的典型而且你希望它在未来也典型，那就自由释放出你最多的高级STL编程爱好。但是，如果你的同时不适应函数式编程风格而且STL经验很少，收回你的野心，写一些接近我前面演示的两条语句的方法的东西。

代码的读比写更经常，这是软件工程的真理。也就是说软件的维护比开发花费多得多的时间。不能读和理解的软件不能被维护，不能维护的软件几乎没有不值得拥有。你用STL越多，你会感到它越来越舒适，而且你会越来越多的使用嵌套函数调用和即时（on the fly）建立函数对象。这没有什么错的，但永远记住你今天写的代码会被某个人——也可能是你——在未来的某一天读到。为那天做准备吧。

当然，使用STL，好好使用，有效使用。但避免产生只写代码。最后，这样的代码完全不高效。

# 条款48：总是#include适当的头文件

STL编程的次要麻烦之一是虽然可以很容易地建立可以在一个平台上编译的软件，但在其它平台上则需要附加的#include指示。这个烦恼来自一个事实：C++标准（不像C标准）未能指定哪一个标准头文件必须或者可能被其他标准头文件#include。由于有了这样的灵活性，不同的实现就会选择去做不同的东西。

这在实践中意味着什么？我可以给你一些的概念。我使用了五个STL平台（咱们叫它们A、B、C、D和E），花了一些时间在它们上测试了一些小程序来看看我可以在忽略哪个标准头文件的情况下仍然成功编译。这间接地告诉我哪个头文件#include了其他的。这是我所发现的：

- 对于A和C，<vector> #includes <string>.
- 对于C，<algorithm> #includes <string>.
- 对于C和D, <iostream> #includes <iterator>.
- 对于D, <iostream> #includes <string> and <vector>.
- 对于D和E, <string> #includes <algorithm>.
- 对于所有的五个实现，<set> #includes <functional>.

除了<set> #include <functional>外，我无法使缺少头文件的程序通过实现B。按照Murphy定律，你总是会在像A、C、D或E那样的平台上开发，然后移植到像B那样的平台，尤其是当移植的压力很大而且完成的时间很紧的情况下。

但是请别指责你将要移植的编译器或库实现。如果你缺少了需要的头文件，这就是你的过错。无论何时你引用了std名字空间里的元素，你就应该对#include合适的头文件负责。如果你遗漏了它们，你的代码也可能编译，但你仍然缺少了必要的头文件，而其他STL平台可能正好会抵制你的代码。

要帮你记起需要的东西，关于在每个标准STL相关的头文件中都有什么，这里有一个快速概要：

- 几乎所有的容器都在同名的头文件里，比如，vector在<vector>中声明，list在<list>中声明等。例外的是<set>和<map>。<set>声明了set和multiset，<map>声明了map和multimap。
- 除了四个算法外，所有的算法都在<algorithm>中声明。例外的是accumulate（参见条款37）、inner_product、adjacent_difference和partial_sum。这些算法在<numeric>中声明。
- 特殊的迭代器，包括istream_iterators和istreambuf_iterators（参见条款29），在<iterator>中声明。
- 标准仿函数（比如less<T>）和仿函数适配器（比如not1、bind2nd）在<functional>中声明。

无论何时你使用了一个头文件中的任意组件，就要确定提供了相应的#include指示，就算你的开发平台允许你不用它也能通过编译。当你发现移植到一个不同的平台时这么做可以减少压力，你的勤奋将因而得到回报。

# 条款49：学习破解有关STL的编译器诊断信息

用一个特定的大小定义一个vector是完全合法的，

```
vector<int> v(10);    // 建立一个大小为10的vector
```

而string在很多方面像vector，所以你可能希望可以这么做：

```
string s(10);        // 常识建立一个大小为10的string
```

这不能编译。string没有带有一个int实参的构造函数。我的一个STL平台像这样告诉我那一点：

```
example.cpp(20): error C2664:'__thiscall std::basic_string<char, struct
std::char_traits<char>,class std::allocator<char> >::std::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
```

是不是很奇妙？消息的第一部分看起来像一只猫走过键盘，第二部分神秘地提到了一个从未在源代码中涉及的分配器，第三部分说构造函数调用是错的。当然，第三部分是准确的，但首先让我们关注于号称猫咪散步的结果上，因为当使用string时，这是你经常遇到的诊断信息的典型。

string不是一个类，它是typedef。实际上，它是这个的typedef：

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

这是因为字符串的C++观念已经被泛化为表示带有任意字符特性（“traits”）的任意字符类型的序列并储存在以任意分配器分类的内存中。在C++里所有类似字符串的对象实际上都是basic_string模板的实例，这就是为什么当大多数编译器发出关于“程序错误使用string”的诊断信息时会涉及类型basic_string。（一些编译器

很善良，在诊断信息中会使用string的名字，但大多数不会。）通常，那样的诊断信息会明确指出basic_string（以及服务助手模板char_traits和allocator）在std名字空间里，所以常常看到错误调用string会产生提及这种类型的诊断信息：

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

这十分接近于上面编译器里使用的诊断信息，但不同的编译器使用这个主题的不同变体。我使用的另一个STL平台以这种方式表示string，

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

string_char_traits和__default_alloc_template的名字是非标准的，但是那是生活。一些STL实现背离了标准。如果你不喜欢你当前STL实现里的背离，考虑用另一个来替换它。条款50给了你可以找到可选择实现的位置的例子。

不管编译器诊断信息怎样表示string类型，把诊断信息减少到有意义的东西的技术是一样的：用文字“ string ”全局替换冗繁难解的basic_string。如果你使用的是命令行编译器，通常可以很容易地用一个类似sed的程序或一种脚本语言比如perl、python或ruby来完成。（你可以在Zolman的文章——《Visual C++的STL错误信息解码器》[26]——里找到一个这样的脚本的例子。）就上面的诊断信息而言，我们用string全局替换

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

可以得到这个：

```
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> &'
```

这会清楚（或至少比较清楚）地说明问题是在传给string构造函数的参数类型里，即使仍然神秘地提及allocator<char>，但比较容易使人发现不存在只带有大小的string构造函数形式。

顺便说一下，神秘地提到分配器的原因是每个标准容器都有一个只带有分配器的构造函数。就string而论，是三个可以用一个实参调用的构造函数之一，但由于某种原因，编译器指出带有分配器的那个是你试图调用的。编译器指错了，而诊断信息也令人误解。哦哟。

至于只带有分配器的构造函数，请不要使用它。那个构造函数是为了容易构造类型相同但分配器不等价的容器。通常，那是不好的，非常不好。要知道为什么，转向条款11。

现在让我们对付更富于挑战性的诊断信息。假定你正在实现一个允许用户通过昵称而不是电子邮件地址查找人的电子邮件程序。例如，这样的程序将可能使用“ The Big Cheese ”作为美国总统（碰巧是president@whitehouse.gov）电子邮件地址的同义词。这样的程序可以使用一个从昵称到电子邮件地址的映射，并可能提供一个成员函数showEmailAddress，显示和给定的昵称关联的电子邮件地址：

```
class NiftyEmailProgram {
private:
    typedef map<string, string> NicknameMap;
    NicknameMap nicknames;                    // 从昵称到电子邮件
                                              // 地址的映射
public:
    ...
    void showEmailAddress(const string& nickname) const;
};
```

在showEmailAddress内部，你需要找到与一个特定的昵称关联的映射入口，所以你可能这么写：

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

编译器不喜欢这个，而且有好的原因，但原因不明显。为了帮助你指出它，这是一个STL平台有帮助地发出的：

```
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,structstd::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,structstd::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
```

有2095个字符长，这条消息看起来相当可怕，但我看过更糟的。对于这个例子我最喜欢的STL平台之一产生了一个4812个字节的诊断信息。正如你所猜测的，错误信息以外的特性是造成我喜爱它的原因。

让我们把这团乱麻减少成容易处理的东西。我们从把basic_string乱语替换成string开始。可以产生这个：

```
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
```

好多了。现在瘦身到745个字符了，我们可以真正地开始看消息了。很可能引起我们注意的东西之一是模板std::_Tree。标准没有说过一个叫_Tree的模板，但名字中的前导下划线随后有一个大写字母唤起了我们的记忆——这样的名字是为实现而保留。这是用来实现STL一些部分的一个内部模板。

实际上，几乎所有STL实现都使用某种内在的模板来实现标准关联容器（set、multiset、map和multimap）。就像使用string的源代码通常导致诊断信息提及basic_string一样，使用标准关联容器的源代码经常会导致诊断信息提及一些内在的树模板。在这里，它叫做_Tree，但我知道的其他实现使用__tree或__rb_tree，后者反映出使用红-黑树——在STL实现中最常使用的平衡树类型。（译注：红黑树的知识可以在数据结构或算法的相关书籍里找到。）

把_Tree先放到一边，上面的消息提到了一个我们得认出的类型：std::map<class string, class string, struct std::less<class string>, class std::allocator<class string > >。这正好是我们正使用的map类型，除了显示了比较和分配器类型（我们定义map时没有指定它们）。如果我们用那个类型的typedef——NicknameMap——替换了它，错误信息将更容易明白。于是产生了这个：

```
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
```

这条信息更短，但清楚得多。我们需要对_Tree做一些事情。因为_Tree是一个实现特定（implementation-specific）的模板，知道它的模板参数的意思的唯一的方法是去读源代码，而如果不必，没有理由要去翻寻实现特定的源代码。让我们试着只是用SOMETHING替换传给的_Tree的全部东西来看看我们得到什么。这是结果：

```
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
```

这是我们能够处理的东西。编译器抱怨我们试图把某种const_iterator转换成iterator，一次对常数正确性的明显破坏。让我们再次看看那段讨厌的代码，我已经高亮了引起编译器怒火的那行：

```
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 != nicknames.end())...
    ...
}
```

有意义的唯一解释是我们试图用一个从map::find返回的const_iterator初始化i（是iterator）。那好像很古怪，因为我们是在nicknames上调用find，而nicknames在非常量对象，find因此应该返回非常量iterator。

再看看。是的，nicknames被声明为一个非常量map，但showEmailAddress是一个*const*成员函数，而在一个const成员函数内部，类的所有非静态数据成员都变成常量！在showEmailAddress内部，nicknames是一个常量map。突然错误信息有意义了。我们试图产生一个进入我们许诺不要修改的map中的iterator。要解决这个问题，我们必须把i改为const_iterator或我们必须使showEmailAddress成为一个非const成员函数。这两个解决方案的挑战性或许比发现错误信息的意思更少。

在本条款中，我演示了用原文替换降低错误信息的复杂度，但一旦你稍微实践，多数时间里你将可以在头脑中进行替换。我不是音乐家（我连开收音机都有困难），但别人告诉我好的音乐家可以在一瞥之间视读几个小节；他们不需要看独立的音符。有经验的STL程序员发展出一项类似的技能。他们可以不假思索地在内部把比如std::basic_string<char, struct std::char_traits<char>, class std::allocator<char> >翻译为string。你也要发展这项技能，但在你能做到之前，记得你总是可以通过用更短的记忆术替换冗长的基于模板的类型名字来把编译器诊断信息降低到可以理解的东西。在许多场合，你要做的就是用你已经使用的typedef名字替换typedef展开。那是我们用NicknameMap替换std::map<class string, class string, struct std::less<class string >, class::allocator<class string > >时所做的。

这里有一些应该能帮助你理解有关STL的编译器消息的其它提示：

- 对于vector和string，迭代器有时是指针，所以如果你用迭代器犯了错误，编译器诊断信息可能会提及涉及指针类型。例如，如果你的源代码涉及vector<double>::iterator，编译器消息有时会提及double*指针。（一个值得注意的例外是当你使用来自STLport的STL实现，而且你运行在调试模式。那样的话，vector和string的迭代器干脆不是指针。对STLport和它调试模式的更多信息，转向条款50。）
- 提到back_insert_iterator、front_insert_iterator或insert_iterator的消息经常意味着你错误调用了back_inserter、front_inserter或inserter，一一对应，（back_inserter返回back_insert_iterator类型的对象，front_inserter返回front_insert_iterator类型的对象，而inserter返回insert_iterator类型的对象。关于使用这些inserter的信息，参考条款30。）如果你没有调用这些函数，你（直接或间接）调用的一些函数做了。
- 类似地，如果你得到的一条消息提及binder1st或binder2nd，你或许错误地使用了bind1st或bind2nd。（bind1st返回binder1st类型的对象，而bind2nd返回binder2nd类型的对象。）
- 输出迭代器（例如ostream_iterator、ostreambuf_iterators（参见条款29），和从back_inserter、front_inserter和inserter返回的迭代器）在赋值操作符内部做输出或插入工作，所以如果你错误使用了这些迭代器类型之一，你很可能得到一条消息，抱怨在你从未听说过的一个赋值操作符里的某个东西。为了明白我的意思，试着编译这段代码：

```
vector<string*> v;                          // 试图打印一个
copy(v.begin(), v.end(),                    // string*指针的容器，
     ostream_iterator<string>(cout, "\n")); // 被当作string对象
```

- 你得到一条源于STL算法实现内部的错误信息（即，源代码引发的错误在<algorithm>中），也许是你试图给那算法用的类型出错了。例如，你可能传了错误种类的迭代器。要看看这样的用法错误是怎样报告的，通过把这段代码喂给你的编译器来启发（并愉快！）自己：

```
list<int>::iterator i1, i2;         // 把双向迭代器
sort(i1, i2);                       // 传给一个需要
                                    // 随机访问迭代器的算法
```

- 你使用常见的STL组件比如vector、string或for_each算法，而编译器说不知道你在说什么，你也许没有#include一个需要的头文件。正如条款48的解释，这问题会降临在长期以来都可以顺利编译而刚移植到新平台的代码。

# 条款50：让你自己熟悉有关STL的网站

因特网充满了STL的信息。用你最喜欢的搜索引擎寻找“STL”，它一定会返回几百个链接，其中有一些可能实际上是相关的。不过，对于大多数STL程序员，没有必要搜寻。下列网站应该要提升到几乎每个人的最常使用列表的顶端：

- SGI STL网站，http://www.sgi.com/tech/stl/。
- STLport网站，http://www.stlport.org/。
- Boost网站，http://www.boost.org/。（译注：如果访问不了，可以试试http://boost.sourceforge.net/）

下面是为什么这些网站值得收藏的简要描述。

## SGI STL网站

SGI的STL网站名列前茅，而且有很好的理由。它对STL的每个组件提供了全面的文档。对于很多程序员，不管他们使用的是哪个STL 平台，这个网站都是他们的在线参考手册。（参考文档由Matt Austern编制，后来把它扩充并精炼为《Generic Programming and the STL》[4]。）这里的材料不仅包括了STL的组件本身。例如，《Effective STL》里关于容器线程安全的讨论（参见条款12）就是基于SGI STL网站上的一个主题。

SGI网站为STL程序员提供了其它东西：一个可以自由下载的STL实现。这个实现只被移植到少数编译器，但广泛移植的STLport分发也是基于SGI分发的，我马上要写更多关于STLport的东西。此外，STL的SGI实现提供了可以让STL编程更强大、更灵活而且更有趣的许多非标准组件。其中最著名的是这些：

- 散列关联容器hash_set、hash_multiset、hash_map和hash_multimap。关于这些容器的更多信息，转向条款25。
- 单链表容器，slist。正如你所想的那样实现，而且迭代器指向你期望它们指向的列表节点。不幸的是，这使实现insert和erase成员函数变得昂贵，因为两者都要求调整迭代器指向的节点*前*一个节点的next指针。在双向链表里（比如标准list容器），这不是问题，但在单链表里，“后退”一个节点是线性时间的操作。对于SGI的slist，insert和erase花费线性而不是常数时间，这是一个相当大的缺点。SGI通过非标准（但常数时间）成员函数insert_after和erase_after解决这个问题。SGI注解到，

  > 如果你发现insert_after和erase_after不能满足你的需要，而且你经常需要在列表中间使用insert和erase，你或许应该使用list来代替slist。

Dinkumware也提供了一个叫做slist的单链表容器，但是它使用了一个保持线性时间性能的insert和erase的不同迭代器实现。关于Dinkumware的更多信息，参考附录B。

- 用于超长字符串的类似字符串的容器。这个容器叫做rope，因为绳子（rope）是重型的线（string），看见了吗？SGI以这种方式描述rope：

  > rope是可伸缩的string实现：它们被设计为用于把string看作一个整体的高效操作。比如赋值、串联和子串的操作所花的时间差不多不依赖字符串的长度。与C的字符串不同，rope是超长字符串的一个合理的表现，比如编辑缓冲区或邮件信息。

  在后端，rope被实现为引用计数子串的树，而且每个子串都存储为字符数组。rope接口的一个有趣方面是begin和end成员函数总是返回const_iterator。这是为了阻止客户进行改变单个字符的操作。这样的操作是昂贵的，rope针对涉及整个字符串的动作（如上所述，例如，赋值、串联和获取子串）优化；单个字符操作表现很差。

- 多个非标准函数对象和适配器。原先的HP STL实现比它成为标准C++后包含更多的函数对象。有两个是被很多老STL黑客所想念的，select1st和select2nd，因为它们对于map和multimap非常有用。给定一个pair，select1st返回它的第一个组件，而select2nd返回它的第二个。这些非标准仿函数类模板可以这么使用：

```
map<int, string> m;
...
// 把所有的map键写到cout
transform(m.begin(), m.end(),
          ostream_iterator<int>(cout, "\n"),
          select1st<map<int, string>::value_type>());

// 建立一个vector，把map中所有的值拷贝进去
vector<string> v;
transform(m.begin(), m.end(), back_inserter(v),
          select2nd<map<int, string>::value_type>());
```

如你所见，select1st和select2nd简化了用算法调用代替你可能必须写自己的循环（参见条款43），但是，如果你使用这些仿函数，它们是非标准的事实使你可能写出不可移植和无法维护的代码（参见条款47）。狂热的STL迷不在乎。他们认为select1st和select2nd没有首先进入标准是不公平的。

作为SGI实现一部分的其它非标准函数对象包括identity、project1st、project2nd、compose1和compose2。要知道这些做了什么，你可以访问网站，虽然你可以在本书的第187页找到使用compose2例子。现在，我希望你清楚访问SGI网站必然是有益的。

SGI的库实现超越了STL。他们的目标是开发一个标准C++库的完整实现，除了从C继承的部分（SGI认为你已经有一个可供支配的标准C库）。因此，可以从SGI获得的另一个值得注意的下载是C++ iostream库的实现。正如你期望的，这个实现与STL的SGI实现结合得很好，但在特征性能方面也优于很多伴随C++编译器的iostream实现。

## STLport网站

STLport的主要卖点是，它提供一个可以移植到超过20个编译器的SGI STL实现（包括iostream等）的修改版本。和SGI的库一样，STLport可以免费下载。如果你写的代码必须在多个平台工作，你可以通过以STLport实现为标准并让你所有的编译器都使用它的方法来给自己节省一堆麻烦。

大多数STLport对SGI代码基础的修改都专注于改进移植性上，但STLport的STL也是我知道的唯一一个提供“调试模式”来帮助侦测STL的不正确用法的实现——可以编译但导致未定义的运行期行为的用法。例如，条款30关于在超过容器结尾的地方写入这个常见错误的讨论中用到了这个例子：

```
int transmogrify(int x);                    // 这个函数从x
                                  // 产生一些新值
vector<int> values;
...                               // 把数据放入values
vector<int> results;
transform(values.begin(), values.end(), // 这回尝试在
                results.end(),       // 超过results结尾
                transmogrify);       // 的地方写入！
```

这可以编辑，但是当运行时，它产生未定义结果。如果你运气好，可怕的东西将在transform调用内部发生，而调试这个问题相对简单。如果你不运气好，transform调用会把数据倾泻在你的地址空间某处，但你得到后来才会发现。在那时，确定内存错误的原因——我们将说？——*挑战*。

STLport的调试模式会消除这个挑战。当上面的transform执行时，会产生下列消息（假设STLport安装在目录C:\STLport）：

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

然后程序停止了，因为如果STLport的调试模式遇到用法错误就会调用abort。如果你喜欢改为抛出一个异常，你可以把STLport配置成你的方式。

无可否认，上述错误信息没有它可能的清楚，而且不幸的是，报告的文件和行对应于内部STL断言的位置而不是调用transform的行， 但这也仍然比越过transform调用的运行，然后试图指出你的数据结构为什么是错误的要好。通过STLport的调试模式，你需要做的所有事情就是发动你的调试器，把调用堆栈返回到你写的代码，然后确定你做错了什么。发现厌恶的源代码行一般不是问题。

STLport的调试模式检测多种常见错误，包括把无效的区间传给算法，试图从一个空的容器里读取，使用来自一只容器的迭代器作为第二个迭代器成员函数的实参，等等。它通过迭代器和它们的容器间彼此跟踪来完成这个魔术。给定两个迭代器，这样就使检查它们是否来自同一个容器成为可能，而且当一个容器被修改时，这使适当的迭代器集失效成为可能。

因为STLport在调试模式使用特殊的迭代器实现，vector和string的迭代器就是类对象而不是原始指针。因此，使用STLport并在调试模式编译是确认没有人草率地处理指针和这些容器类型的迭代器之间区别的一种好方法。单单这点就足够成为给STLport调试方式一个机会的理由。

## Boost网站

在1997年，当关闭通向C++国际标准的道路的钟声响起时，一些人对他们提倡的库特性没有入选而感到失望。这些人中的一部分本身就是委员会的成员，所以他们开始在第二轮标准化期间为标准库的扩充打下基础。结果就是Boost，一个任务是“ 提供免费、同行评议的C++库。重点在于可以和C++标准库配合良好的可移植库 ”的网站。在任务后面是一个动机：

> 一个库变成“ 现有实践 ”的程度，某人把它提交给未来标准的可能性就增加了。提交一个库给Boost.org是建立现有实践的一种方法……

换句话说，当一个库可能增加到标准C++库时，Boost把自己作为帮助区分好坏的鉴别机制。这是一个可敬的服务，而且我们全都应该感激。

让人感激的另一个原因是你可以在Boost中找到的库集合。我并不想在这里全部解释它们，尤其是因为当你读这些话时，无疑已经增加了许多新的库。不过，对于STL用户，两个库特别有用。第一个是智能指针库，包含用于引用计数智能指针的模板shared_ptr，与标准库的auto_ptr不同，它可以安全的储存在STL容器里（参见条款8）。Boost的智能指针库也提供shared_array，一个用于动态分配数组的引用计数智能指针，但条款13论述了动态分配数组不如vector和string，而且我希望你发现它的论点有说服力。

Boost第二个吸引STL迷的是它有关STL的函数对象和相关工具的群。这些库包含的基本原则是重新设计和重新实现STL函数对象和适配器后面的思想，这个结果消除了很多对标准仿函数功效的人为约束。作为这样的

一个约束的例子，你将发现如果你试图把bind2nd和mem_fun或mem_fun_ref一起用（参见条款41）来把一个对象绑定到一个成员函数的参数，而且那个成员函数通过引用获取它的参数，你的代码不可能编译。如果你试图把not1或not2和ptr_fun一起用而且一个函数声明了通过引用的参数，你将发现一样的结果。在两中情况里的原因是在模板实例化的过程中，大多数STL平台产生到引用的引用，而到引用的引用在C++里不合法。（标准委员会正在酝酿标准里的一个改变来解决这个问题。）这里有一个被认为是“到引用的引用的问题”的例子：

```
class Widget {
public:
    ...
    int readStream(istream& stream);                  // readStream用
    ...                                               // 引用获取参数
};

vector<Widget*> vw;
...

for_each(                                             // 大多数STL平台
        vw.begin(), vw.end(),                         // 试图在这个调用中
        bind2nd(mem_fun(&Widget::readStream), cin)    // 产生一个
                                                      // 到引用的引用；
                                                      // 那样的代码
                                                      // 不能编译
```

Boost的函数对象避免了这个和其它问题，此外它们相当地扩大了函数对象的表现力。

如果你对STL函数对象的潜力感兴趣而且你想要更进一步探索它，赶快马上转到Boost。如果你痛恨函数对象而且认为它们的存在只是为了安慰少数转为C++程序员的Lisp拥护者，那也赶快转向Boost。Boost的函数对象库很重要，但它们只是你能在那个网站发现的一小部分而已。

# 参考书目

下面的大部分出版物都在本书中被引用，虽然很多引用只在致谢中出现。没有引用的出版物前面加的是点而不是编号。

给一个不稳定的因特网URL，我在把它们包含到本参考书目前很踌躇。最后，我觉得即使是一个你每次尝试都无效的URL，知道一个文档曾经在哪里也对在另一个URL找到它有帮助。

## 我写的东西


[1] Scott Meyers，《Effective C++: 50 Specific Ways to Improve Your Programs and Designs》（第二版），Addison-Wesley，1998，ISBN 0-201-92488-9。也可以在《Effective C++ CD》里找到（参见下面）。

[2] Scott Meyers，《More Effective C++: 35 New Ways to Improve Your Programs and Designs》，Addison-Wesley 1996，ISBN 0-201-63371-X。也可以在《Effective C++ CD》里找到（参见下面）。

· Scott Meyers，《Effective C++ CD: 85 Specific Ways to Improve Your Programs and Designs》, Addison-Wesley，1999，ISBN 0-201-31015-5。包含上面两本书，一些相关杂志文章，和一些电子出版的创新。要试用这张CD，访问http://meyerscd.awl.com/。要知道它的创新，看看http://zing.ncsl.nist.gov/hfweb/proceedings/meyers-jones/和http://www.microsoft.com/Mind/1099/browsing/browsing.htm。


## 我没写的东西（但我希望我有）


[3] Nicolai M. Josuitis，《The C++ Standard Library: A Tutorial and Reference》，Addison-Wesley，1999，ISBN 0-201-37926-0。一本不可缺少的书。每个C++程序员都应该有一份拷贝。

[4] Matthew H. Austern，《Generic Programming and the STL》，Addison-Wesley，1999，ISBN 0-201-30956-4。这本书基本上是SGI STL网站http://www.sgi.com/tech/stl/上材料的打印版本。

[5] ISO/IEC《International Standard, Programming Languages -- C++》，参考号码ISO/IEC 14882:1998(E)，1998。描述C++的官方文档。在ANSI花18美元就可以得到PDF，http://webstore.ansi.org/ansidocstore/default.asp。

[6] Erich Gamma、Richard Helm、Ralph Johnson和John Vlissides，《Design Patterns: Elements of Reusable Object-Oriented Software》，Addison-Wesley，1995，ISBN 0-201-63361-2。也可以看《Design Patterns CD》，Addison-Wesley，1998，ISBN 0-201-63498-8。设计模式的权威著作。每个实际的C++程序员应该熟悉这里描述的模式，而且应该可以很容易地翻阅这本书或CD。

[7] Bjarne Stroustrup，《The C++ Programming Language》（第三版），Addison-Wesley，1997，ISBN 0-201-88954-4。我在条款12提到的“资源获取即初始化”惯用法在这本书的第14.4.1节讨论，我在条款36引用的代码在第530页。

[8] Herb Sutter，《Exceptional C++ : 47 Engineering Puzzles, progromming Problems, and Solutions》，Addison-Wesley，2000，ISBN 0-201-61562-2。对我Effective系列的一个值得推崇的补充，即使当时Herb没有让我为它写序，我也会赞美它。

[9] Herb Sutter，《More Exceptional C++: 40 More Engineering Puzzles, Programming Problems, and Solutions》，Addison-Wesley，预计在2001年出版，暂定ISBN 0-201-70434-X。基于我看过的草稿，他看起来完全和它的前辈[8]一样好。

[10] Dov Bulka和David Mayhew，《Efficient C++: Performance Programming Techniques》，Addison-Wesley，2000，ISBN 0-201-37950-3。唯一一本专注于C++效率的书，因此也是最好的。

[11] Matt Austern，“How to Do Case-Insensitive String Comparison”，《C++ Report》，2000年5月。这篇文章很重要，他被复制为本书的附录A。

[12] Herb Sutter，“When Is a Container Not a Container?”，《C++ Report》，1999年5月。可以在http://www.gotw.ca/publications/mill09.htm找到。修订和更新为《More Exceptional C++》[9]的条款6。

[13] Herb Sutter，“Standard Library News: sets and maps”，《C++ Report》，1999年10月。可以在http://www.gotw.ca/publications/mill11.htm找到。修订和更新为《More Exceptional C++》[9]的条款8。

[14] Nicolai M. Josuttis，“Predicates vs. Function Objects”，《C++ Report》，2000年6月。

[15] Matt Austern，“Why You Shouldn't Use set -- and What to Use Instead”，《C++ Report》，2000年4月。

[16] P. J. Plauger，“Hash Tables”，《C/C++ Users Journal》，1998年11月。描述了Dinkumware散列容器的方法（在条款25讨论）以及它与其他竞争者的设计区别。

[17] Jack Reeves，“STL Gotcha's”，《C++ Report》, 1997年1月。可以在http://www.bleading-edge.com/Publications/C++Report/v9701/abstract.htm找到。

[18] Jack Reeves，“Using Standard string in the Real World, Part 2”，《C++ Report》，1999年1月。可以在http://www.bleading-edge.com/Publications/C++Report/v9901/abstract.htm找到。

[19] Andrei Alexandreseu，“Traits: The else-if-then of Types”，《C++ Report》，2000年4月。可以在http://www.creport.com/html/from-pages/view-recent_articles_c.cfm?ArticleID=402找到。

[20] Herb Sutter，“Optimizations That Aren't (In a Multithreaded World)”，《C/C++ Users Journal》，1999年6月。可以在http://www.gotw.ca/publications/optimizations.htm找到。修订和更新为《More Exceptional C++》[9]的条款16。

[21] SGI STL网站，http://www.sgi.com/tech/stl/。条款50总结了这个重要网站的材料。关于STL容器线程安全（它是条款12的动机）的页在http://www.sgi.com/tech/stl/thread_safety.html。

[22] Boost网站，http://www.boost.org/。条款50总结了这个重要网站的材料。

[23] Nicolai M. Josuttis，“User-Defined Allocator”，http://www.josuttis.com/cppcode/allocator.html。这页是Josuttis关于C++标准库[3]优秀的书的网站的一部分。

[24] Matt Austern，“The Standard Librarian: What Are Allocators Good For?”，《C/C++ Users Journal》的C++专家论坛（这个杂志的在线扩展），2000年12月，http://www.cuj.com/documents/s=8000/cujcexp1812austern/。分配器的好资料很难得到。这个专栏补充了条款10和11的材料。它也包含了一个分配器的实现样例。

[25] Klaus Kreft和Angelika Langer，“A Sophisticated Implementation of User-Defined Inserters and Extractors”，《C++ Report》，2000年2月。

[26] Leor Zolman，“An STL Error Message Decryptor for Visual C++”，《C/C++ Users Journal》，2001年7月。这篇文章和它描述的软件可以在http://www.bdsoft.com/tools/stlfilt.html找到。

[27] Bjarne Stroustrup，“Sixteen Ways to Stack a Cat”，《C++ Report》，1990年10月。可以在http://www.research.att.com/~bs/stack_cat.pdf找到。

- Herb Sutter，“Guru of the Week #74: Uses and Abuses of vector”，2000年9月，http://www.gotw.ca/gotw/074.htm。这个测试（和伴随的解决方案）很好地让你想想那些vector相关的问题比如size和capacity（参见条款14），但它也讨论了为什么算法调用比手写循环优越（参见条款43）。
- Matt Austern，“The Standard Librarian: Bitsets and Bit Vectors?”，《C/C++ Users Journal》的C++专家论坛（这个杂志的在线扩展），2001年5月，http://www.cuj.com/experts/1905/austern.htm。这篇文章提供了bitset的信息与它们和vector<bool>的比较，这是我在条款18简要考查的主题。

# 我写过的（但我希望我没有）

- 《Effective C++》勘误表，http://www.aristeia.com/BookErrata/ec++2e-errata.html。

[28] 《More Effective C++》勘误表，http://www.aristeia.com/BookErrata/mec++-errata.html。

- 《Effective C++ CD》勘误表，http://www.aristeia.com/BookErrata/cd1e-errata.html。

[29] 《More Effective C++》auto_ptr更新页，http://www.awl.com/cseng/titles/0-201-63371-X/auto_ptr.html。

# 区域设置和忽略大小写的字符串比较

条款35解释了怎样使用mismatch和lexicographical_comapre实现忽略大小写的字符串比较，但是它也指出一个真正通用的解决方案必须考虑区域设置。本书是关于STL的，而非国际化的，因此我几乎没有提到任何关于区域设置的东西。不过，Matt Austern，《Generic Programming and the STL》[4]的作者，在2000年5月《C++ Report》[11]的专栏里提到了涉及区域设置的忽略大小写字符串比较。为了完整地讲述这个重要的主题，我很高兴能在这里再版他的专栏，而且我感谢Matt和101communications准许我这么做。

## 如何进行忽略大小写的字符串比较

Matt Austern

如果你写的程序曾经用到过string（谁没有吗？），有时候可能你需要处理两个除了大小写不同其他都相同的字符串。即，你需要让比较——相等、小于、子串匹配、排序——都忽略大小写。而且，的确，关于标准C++库的最常见问题之一是怎样使string忽略大小写。这个问题已经被回答很多次了。很多答案是错误的。

首先，让我们放弃试图写忽略大小写string类的想法。是的，它在技术上多多少少是可能的。标准库类型std::string其实只是一个模板std::basic_string<char, std::char_traits<char>, std::allocator<char> >的别名。所有的比较都使用了特性参数，所以，通过提供一个正确地重定义了相等和小于的特性参数的方法，你可以实例化basic_string，使<和==操作符是忽略大小写的。你可以这么做，但没必要那么麻烦。

- *你将不能做I/O*，至少不能再没有很多痛苦的情况下进行。标准库里的I/O类，比如std::basic_istream和std::basic_ostream，与std::basic_string一样在字符类型和特性上模板化。（再次强调，std::ostream只是一个std::basic_ostream<char, char_traits<char> >的别名。）特性参数必须匹配。如果你对字符串使用std::basic_string<char, my_traits_class>，你就必须对流使用std::basic_ostream<char, my_traits_class>。你不能使用比如cin和cout那样普通的流对象。
- *忽略大小写不涉及一个对象，而涉及你怎样使用一个对象。*你可能在一些情况下非常需要把一个string当作关注大小写而在其他情况下忽略大小写。（或许取决于用户控制的选项。）为这两种应用定义不同的类型是在它们之间放置人造障碍。
- *它并不合适。*正如所有的特性类[1]，char_traits是小的、简单的和无状态的。我们可以在本专栏的后面看到，正确的忽略大小写比较不是这样的东西。
- *它不够。*即使所有basic_string本身的成员函数都忽略大小写，当你需要使用非成员泛型算法比如std::search和std::find_end时，将仍然没有帮助。如果你决定，出于效率的考虑，从basic_string对象的容器改为字符串表，它也将没有帮助。

更自然地融合入标准库设计的更好的解决方案是在当你需要时才进行忽略大小写的比较。不要为string::

find_first_of和string::rfind那样的成员函数而烦恼；它们的功能都存在于非成员泛型算法。同时，泛型算法灵活得足以适应忽略大小写的字符串。例如，如果你需要以忽略大小写的顺序排序一个字符串的集合，你需要做的就是提供适当的比较函数对象：

```
std::sort(C.begin(), C.end(), compare_without_case);
```

本专栏的剩余部分将致力于怎样写那个函数对象。

## 第一次尝试

有不止一种方法来按字母顺序排列单词。下次你在书店时，注意作者的名字是怎么安排的：Mary McCarthy在Bernard Malamud之前，还是之后？（这是习惯的问题，而且这两种方式我都看到过。）但是，字符串比较的最简单方式是我们都在小学学过的那个：词典或者“ 字典顺序 ”比较，我们从一个一个字符的比较建立了字符串比较。

词典比较可能不适合专业应用（不是唯一的方法；库可能以不同的方式排序人名和地名），但它适合大部分情况，而且这是字符串比较在C++里的默认意思。字符串是字符的序列，如果x和y的类型是std::string，表达式x < y等价于这个表达式

```
std::lexicographical_compare(x.begin(), x.end(), y.begin(), y.end()).
```

在这个表达式中，lexicographical_compare使用operator<比较单个字符，但还有一个版本的lexicographical_compare让你选择自己的比较字符的方法。那个版本带有五个实参，而不是四个；最后一个实参是一个函数对象，确定两个字符哪个应该在另一个之前的二元判定式。然后，为了用lexicographical_compare进行忽略大小写比较，我们需要把它和忽略大小写的字符比较函数对象结合起来。

在字符的忽略大小写的比较后面的一般的想法是把两个字符都转化成大写字母然后比较结果。这里把那个想法显而易见的转换成一个C++函数对象，使用了来自标准C库的一个广为人知的函数：

```
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));
    }
};
```

“任何复杂问题都有一个简单、整洁而且错误的解决方案。”写关于C++的书的人都喜欢这个类，因为它是一个好的、简单的例子。我像其他人一样心虚；我在我的书中多次使用了它。它几乎是正确的，但是不够好。问题是微妙的。

这里有一个你可以开始发现问题的例子：

```
int main()
{
    const char* s1 = "GEW\334RZTRAMINER";
    const char* s2 = "gew\374rztraminer";
    printf("s1 = %s, s2 = %s\n", s1, s2);
    printf("sl < s2: %s\n",
        std::lexicographical_compare(s1, s1 + 14, s2, s2 + 14, lt_nocase())
        ? "true" :"false");
}
```

你应该在你的系统上试试看。在我的系统上（一台运行IRIX 6.5的Silicon Graphics O2），这是输出：

```
s1 = GEWÜRZTRAMINER, s2 = gew ü rztraminer
s1< s2: true
```

噢，多古怪。如果你做忽略大小写比较，难道“gew ü rztraminer”和“GEWÜRZTRAMINER”不同吗？现在做一个轻微的变化：如果你在printf语句之前插入这行

```
setlocale(LC_ALL, "de");
```

，突然输出改变了：

```
s1 = GEWÜRZTRAMINER, s2 = gew ü rztraminer
s1 < s2: false
```

忽略大小写的字符串比较比看起来更复杂。这表面上正确的程序非常依赖于大多数人经常忽略的东西：区域设置。

## 区域设置

一个char真的无异于一个小的整数。我们可以选择把一个小的整数解释成一个字符，但这种解释并不通用。一些特定的数字应该被解释为一个字母、一个标点符号还是一个不能打印的控制字符？

没有一个正确的答案，而且直到关系到C和C++语言核心之前它们没有任何不同。需要靠一些库函数产生那些区别：例如，isalpha确定了一个字符是否是字母，toupper把小写字母转换成大写而对大写字母或不是字母的字符则什么都不做。所有那些都取决于本地文化和语言习惯：字母和非字母之间的区别在英语中是一个意思，在瑞典语则是另一个意思。从小写到大写的转换在罗马和斯拉夫字母表中表示不同的东西，而在希伯来语中则没有任何意义。

默认情况下，字符操作函数适用于简单的英语文字字符集。字符'\374'不受toupper影响，因为它不是一个字母；在一些系统上打印时它可能看起来像ü，但那和操作英语文字的C库程序不相干。在ASCII字符集里没有ü字符。这行

```
setlocale(LC_ALL, "de");
```

告诉C库开始根据德语习惯操作。（至少它在IRIX上是那样。区域的名字没有标准化。）德语中有字符ü，因此toupper把ü改为Ü。

如果这还不使你紧张，那么马上就会。虽然toupper可能看起来像带有一个实参的简单函数，但它也依赖于一个全局变量——不好，一个隐藏的全局变量。这引发了所有常见的困难：使用toupper的函数潜在地依赖于整个程序中的任何一个其他函数。

如果你把toupper用于忽略大小写的字符串比较，这可能是灾难性的。如果你有一个依赖于有序list的算法（比如binary_search），然后一个新的区域设置引发了它后面的排序顺序的改变，那将发生什么？像这样的代码不可复用：只是勉强可用。你不能在库里使用它——库可以用于任何种类的程序，不只是从未调用setlocale的程序。你可能在一个大的程序里使用它却侥幸逃过一劫，但你将有一个维护问题：或许你能证明没有其他模块调用了setlocale，但你能证明在程序明年的版本里没有其他模块调用setlocale吗？

这个问题在C里没有好的解决方案。C库只有一个全局的区域设置，没别的了。在C++里有一个解决方案。

## C++中的区域设置

C++标准库里的区域设置不是深深地埋藏在库实现里的全局数据。它是一个std::locale类型的对象，而且你可以建立它和把它传给函数，就像其他任何对象一样。例如，你要建立一个表示通常区域的区域设置对象可以这么写

```
std::locale L = std::locale::classic();
```

```
```

或者你通过这么写建立一个德语区域设置

```
std::locale L("de");
```

（和C库里的一样，区域的名字没有标准化。检查你的实现文档来查明提供了哪些有名字的区域设置。）

C++里的区域设置分成多个*方面*（facet），每个方面处理一个国际化的不同方向，而函数std::use_facet从区域设置对象[2]中提取一个特定的方面。ctype方面处理字符分类，包括大小写转换。最后，如果c1和c2是char类型，这段代码将以适合区域设置L并以忽略大小写的方式比较它们。

```
const std::ctype<char>& ct = std::use_facet<std::ctype<char> >(L);
bool result = ct.toupper(c1) < ct.toupper(c2);
```

也有一个特别的缩写词：你可以写

```
std::toupper(c, L)
```

意思和

```
std::use_facet<std::ctype<char> >(L).toupper(c)
```

相同（如果c是char类型）。不过，最小化调用use_facet的次数是值得的，因为它可能相当昂贵。

正如词典比较不能适合于所有应用一样，一个一个字符的大小写转换也不总是适合的。（例如，在德语里，小写字母“ß”对应着大写序列“SS”。）但是，不幸的是，一个一个字符的大小写转换是我们拥有的全部。C和C++标准库都没有提供任何一次用于不止一个字符的字符串转换形式。如果你的目的不能接受这个限制，那么就已经在标准库的范围之外了。

## 离题一下：另一个方面

如果你已经熟悉C++里的区域设置，你可能想果用另一种方式进行字符串比较：collate方面的存在正好封装了排序的细节，而它有一个接口很像C库函数strcmp的成员函数。甚至有一个方便的特征：如果L是一个区域设置对象，你可以通过写L(x，y)而不是通过讨厌地调用use_facet然后调用collate成员函数来比较两个字符串。

“ classic ”区域设置有一个进行词典排序的collate方面，和字符串的operator<做的一样，但其他区域设置进行任何种比较都是合适的。如果你的系统正好有一个对任何你感兴趣的语言进行忽略大小写比较的区域设置，你可以使用它。那个区域设置甚至可能做出比一个一个字符比较更智能化的事情！

不幸的是，这个建议，可能是真的，并不能帮助像我们这些没有那样的系统的人。或许有一天一套这样的区域设置可以标准化，但现在它们还没有。如果没有人为你写了一种忽略大小写的比较，你就必须亲自写它。

## 忽略大小写字符串比较

使用ctype，用忽略大小写字符比较构造忽略大小写字符串比较是很简单的。这个版本不是最优的，但至少它是正确的。它基本上使用和以前相同的技术：使用lexicographical_compare比较两个字符串，而且通过把两个字符都转换成大写来比较它们。不过，这次我们小心地使用区域设置对象而不是全局变量。（另外说一下，把两个字符都转化成大写不一定总是等于把两个字符都变成小写的结果：没有保证操作是可逆的。例如，在法语里，通常忽略大写字符上的重音标记。在法语区域设置中，toupper有理由是有损转换；它可以把“ é ”和“ e ”都转换成同样的大写字符，“ E ”。那么，在这样的区域设置里，使用toupper的忽略大小写比较将说“ é ”和“ E ”是等价字符，而tolower将说它们不是。哪个是正确的答案？或许是前者，但它取决于语言，取决于当地习惯，取决于你的应用程序。）

```
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));
    }
};
```

这还不很好；它比应该的慢。问题是讨厌的和技术性的：我们在循环内调用toupper，而C++标准要求toupper是虚函数调用。一些优化器可能聪明得足以把虚函数开销移到循环之外，但是大多数不是。循环内的虚函数调用应该避免。

在这里，避免它不是很简单。你可能想到正确答案是ctype的另一个成员函数，

```
const char* ctype<char>::toupper(char* f, char* l) const
```

这改变了区间[f, l)内的字符大小写。不幸的是，这不完全是我们的目标的正确接口。使用它来比较两个字符串要求把两个字符串都拷贝到缓冲区，然后把缓冲区转化成大写。那些缓冲区从哪里来？它们不能是固定大小的数组（多大才足够大？），但动态数组需要昂贵的内存分配。

另一个解决方案是每次对一个字符进行大小写转换并缓存结果。这不是一个完全通用的解决方案——例如，如果你用的是32位UCS 4字符，它将完全不能工作。不过，如果你用char（大部分系统上是8位），在比较函数对象里维护一个256字节的大小写转换信息不是没有道理的。

```
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));
    }
};
```

正如你看见的，lt_str_1和lt_str_2不是非常不同。前者有一个直接使用ctype方面的字符比较函数对象，而后者使用一张预先算好的大写转换表的字符比较函数对象。如果你建立lt_str_2函数对象，使用它比较一些短字符串，然后放弃它，可能会比较慢。不过，对任何实际的使用来说，lt_str_2将明显比lt_str_1快。在我的系统上差别不止两倍：用lt_str_1排序一个23,791个单词的list花费0.86秒，而用lt_str_2花费0.4秒。

我们从所有这些里学到了什么？

- 忽略大小写的字符串类是错误的抽象层面。C++标准库中的泛型算法是由策略参数化的，而你应该利用这个事实。
- 词典字符串比较建立在字符比较之上。一旦你有了一个忽略大小写的字符比较函数对象，问题就解决了。（而且你可以把那个函数对象重用于比较其他类型的字符序列，比如vector<char>，或字符串表，或原始的C字符串。）
- 忽略大小写的字符比较比看起来难。除了在一个特定区域设置的场景之外，它没有意义，所以字符比较函数对象需要储存区域设置信息。如果关系到速度，你应该写避免重复调用昂贵的方面操作的函数对象。

正确的忽略大小写比较花费了大量手段，但是你只须把它写一次。你或许不想考虑locale；大多数人不。（谁想在1990年考虑千年虫？）如果你依赖区域设置的代码正确了，那么你忽视区域设置的可能性将大于你写出消除了这个依赖性的代码。

---

[1] 参见Andrei Alexandrescu在《C++ Report》2000年4月的专栏[19]。 [2] 警告：use_facet是一个函数模板，它的模板参数值出现在返回类型，而不是任何实参。使用一个叫做*显式模板参数特化*的语言特性来调用它，而一些C++编译器尚未支持那个特性。如果你使用了一个不支持的编译器，你的库实现可能提供了变通办法，所以你可以用某种方式调用use_facet。

# 在微软STL平台上的注意事项

在这本书的开头几页里，我提到了术语*STL平台*是指一个特定编译器和一个标准模板库特定实现的组合。如果你在使用版本6或更早的Microsoft Visual C++编译器（即，伴随版本6或更早的Microsoft Visual Studio的编译器），在编译器和库之间的区别特别重要，因为编译器有时比伴随的STL实现更有能力。在本附录中，我描述了旧的微软STL平台的一个重要缺点，而且我提供可以明显改进你STL经验的变通办法。

下面是给使用Microsoft Visual C++（MSVC）4-6版的开发者的信息。如果你正使用Visual C++ .NET，你的STL平台没有下面描述的那些问题，你可以略过这个附录。

## STL里的成员函数模板

假设你有两个Widget的vector，你想把一个vector里的Widget拷贝到另一个的末端。那很容易。只要使用vector的区间insert函数（参见条款5）：

```
vector<Widget> vw1, vw2;
...
vw1.insert(vw1.end(), vw2.begin(), vw2.end());       // 把vw2里Widget的副本
                                                     // 追加到vw1
```

如果你有一个vector和一个deque，你一样可以这么做：

```
vector<Widget> vw;
deque<Widget> dw;
vw.insert(vw.end(), dw.begin(), dw.end());           // 把dw里Widget的副本
                                                     // 追加到vw
```

实际上，你不必理会被拷贝的容器容纳的是什么类型的对象。即使自定义容器也可以工作：

```
vector<Widget> vw;
...
list <Widget> lw;
...
vw.insert(vw.begin(), lw.begin(), lw.end());         // 把lw里Widget的副本
```

```
                                        // 追加到vw
set<Widget> sw;
...
vw.insert(vw.begin(), sw.begin(), sw.end());            // 把sw里Widget的副本
                                        // 追加到vw

template<typename T,                            // 用于自定义
      typename Allocator = allocator<T> >               // 兼容STL的
class SpecialContainer { ... };                 // 容器模板
SpecialContainer<Widget> scw;
...
vw.insert(vw.end(), scw.begin(), scw.end());            // 把scw里Widget的副本
                                        // 追加到vw
```

这种灵活性是可能的，因为vector的区间insert函数完全不是一个函数。相反，它是一个*成员函数模板*，可以用任何*迭代器类型*实例化，以产生一个具体的区间insert函数。对于vector，标准像这样声明insert模板：

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

每个标准容器都要求提供这个模板化的区间insert。容器也要求提供类似成员函数模板的区间构造函数和assign的区间形式（两者都在条款5讨论）。

# MSVC版本4-6

不幸的是，伴随MSVC版本4-6的STL实现没有声明成员函数模板。这个库最初为MSVC版本4开发，而那个编译器，像它所在时代的大多数编译器一样，缺乏成员函数模板的能力。在MSVC4到MSVC6之间，编译器增加了对这些模板的支持，但是，由于法律诉讼的影响，微软没有直接包含它们，库基本保持冻结。

因为伴随MSVC4-6的STL实现是为一个缺乏成员函数模板的编译器设计的，库的作者通过用具体函数替换每个模板的方法来近似这样的功能性，也就是只接受和容器的迭代器类型相同的迭代器。例如，对于insert，这

个成员函数模板被替换这样：

```
void insert(iterator position,                    // “ iterator ”是
            iterator first, iterator last);       // 容器的迭代器类型
```

受限的区间成员函数形式可以进行从一个vector<Widget>到一个vector<Widget>或从一个list<int>到一个list<int>的区间插入，但不能从一个vector<Widget>到一个list<Widget>或从一个set<int>到一个deque<int>。甚至不可能进行从一个vector<long>到一个vector<int>的区间insert（或assign或构造），因为vector<long>::iterator与vector<int>::iterator类型不同。结果，下面十分有效的代码不能用MSVC4-6编译：

```
istream iterator<Widget> begin(cin), end;                 // 建立用于从cin
                                  // 读取Widget的
                                  // begin和end迭代器
                                  // （参见条款6）
vector<Widget> vw(begin, end);                    // 把cin的Widget读入vw
                                  // （再次参见条款6）；在MSVC4-6
                                  // 不能编译
list<Widget> lw;
...
lw.assign(vw.rbegin(), vw.rend());                      // 把vw的内容赋值给lw
                                  // （以反序）；在MSVC4-6
                                  // 不能编译
SpecialContainer<Widget> scw;
...
scw.insert(scw.end(), lw.begin(), lw.end());              // 把lw中的Widget的副本
                                  // 插入scw的末端；
                                  // 在MSVC4-6不能编译
```

那如果你必须使用MSVC4-6，你该怎么办？那取决于你使用的MSVC版本和是否你被迫使用伴随编译器的STL实现。

## MSVC4-5的变通办法

再次看看不能用伴随MSVC4-6的STL编译的有效代码例子：

```
vector<Widget> vw(begin, end);                    // 被MSVC4-6的
```

```
                                    // STL实现拒绝
list<Widget> lw;
...
lw.assign(vw.rbegin(), vw.rend());                  // 也拒绝

SpecialContainer<Widget> scw;
...
scw.insert(scw.end(), lw.begin(), lw.end());   // 同上
```

这些调用看起来相当不同，但它们全都由于相同的原因而失败：在STL实现里缺乏成员函数模板。对它们有一种单独的治疗方法：使用copy和插入迭代器（参见条款30）。例如，这里是上面例子的变通办法：

```
istream_iterator<Widget> begin(cin), end;
vector<Widget> vw;                                  // 默认构造vw；
copy(begin, end, back_inserter(vw));                    // 然后把cin中的
                                    // Widget拷贝进去
list<Widget> lw;
...
lw.clear();                                 // 去除lw的老
copy(vw.rbegin(), vw.rend(), back_inserter(lw));            // Widget；把
                                    // vw的Widget拷贝进去（以
                                    // 反序）

SpecialContainer<Widget> scw;
...
copy(lw.begin(), lw.end(),                          // 把lw的Widget拷贝到
     inserter(scw, scw.end()));                     // scw的结尾
```

我鼓励你在伴随MSVC4-5的库上使用这样的基于copy的变通办法，但是注意！不要满足于这个变通办法，你忘记了它们只是*变通办法*。正如条款5解释的，使用copy算法几乎总是不如使用一个区间成员函数，所以一旦你有机会把你的STL平台升级到支持成员函数模板的版本，就在区间成员函数是正确方法的地方停止使用copy。

## 用于MSVC6的另一个变通办法

你也可以对MSVC6使用MSVC4-5的变通办法，但对于MSVC6有另一个选择。作为MSVC4-5一部分的编译器没有提供有意义的成员函数模板，所以STL实现缺乏它们的事实是无关紧要的。MSVC6的形势则不同，因为

MSVC6的编译器支持成员函数模板。因此有理由考虑用提供标准指定的成员函数模板的STL实现替换伴随MSVC6的。

条款50解释了SGI和STLport都提供了可以自由下载的STL实现，而且那两个实现都把MSVC6编译器作为将配合的编译器之一。你也可以从Dinkumware购买最新的兼容MSVC的STL实现。每种选择各有利弊。

SGI的和STLport的实现是自由的，我想你知道那在对软件的官方支持上代表什么：完全没有。而且，因为SGI和STLport把他们的库设计为使用多种编译器，你或许必须手工配置它们的实现来最有效地使用MSVC6。特别是，你可能必须明确地启用成员函数模板的支持，因为，它们要使用很多编译器，SGI和/或STLport默认可能不启用它。你可能也得为与其他MSVC6库（特别是DLL）链接而担心，包括保证你使用合适的线程和调试构建，等等。

如果你被那些事吓着了，或如果你听过你负担不起自由软件的牢骚，你可能要看看Dinkumware用于MSVC6的替代库。它被设计为提高原生MSVC6 STL的兼容性，并使作为STL平台的MSVC6对标准的支持最大化。因为Dinkumware写了伴随MSVC6的STL，所以他们最新的STL实现有很大的可能性真的是一个合适的替代品。要了解更多关于Dinkumware STL实现的信息，访问公司的网站：http://www.dinkumware.com/。

不管你选择的是SGI的、STLport的还是Dinkumware的实现作为STL的替代品，你将得到的不只是带有成员函数模板的STL。你也将在库的其他地方旁路一致性问题，比如没有声明push_back的string。此外，你可以访问有用的STL扩展，包括散列容器（参见条款25）和单链表（slists）。SGI的和STLport的实现也提供了多种非标准的仿函数类，比如select1st和select2nd（参见条款50）。

即使你被伴随MSVC6的STL实现困住，访问Dinkumware网站或许也是值得的。那个网站列举了在MSVC6的库实现里的已知漏洞并解释怎样修改你的库副本来减少它的缺陷。不用说，编辑你的库头文件是让你自己冒险的事。如果你遇到麻烦，不要责备我。

# 词汇表

本表列出的是一些单词在本书中使用的翻译。这些单词大部分是专业术语，一部分是字典上没有的。

| 英文 | 中文 |
| --- | --- |
| adapter | 适配器 |
| algorithm | 算法 |
| allocate | 分配 |
| allocator | 分配器 |
| amortize | 分摊 |
| argument | 实参 |
| associative container | 关联容器 |
| cast | 映射 |
| category | 种类 |
| component | 组件 |
| context | 场景 |
| constness | 常数性 |
| container | 容器 |
| deallocate | 回收 |
| dereference | 解引用 |
| diagnostics | 诊断信息 |
| encapsulation | 封装 |
| equality | 相等 |
| equivalence | 等价 |
| exception safey | 异常安全 |
| function object | 函数对象 |
| functionality | 功能性 |
| functor | 仿函数 |
| generalize | 泛化 |

| | |
|---|---|
| hash | 散列 |
| implementation | 实现 |
| instantiation | 实例化 |
| iterator | 迭代器 |
| key | 键 |
| lexical | 词法 |
| locale | 区域设置 |
| namespace | 名字空间 |
| pivot element | 主元 |
| parameter | 参数 |
| predicate | 判断式 |
| range | 区间 |
| reference counting | 引用计数 |
| roll back | 回退 |
| scope | 作用域 |
| semantics | 语义 |
| sequence container | 序列容器 |
| slice | 分割 |
| smart pointer | 智能指针 |
| specification | 特化 |
| splice | 接合 |
| trait | 特性 |
| transactional | 事务性 |
| workaround | 变通办法 |

# 关于本电子书

制作本书的目的是为了方便大家的阅读。转载时请保持本电子书的完整性。

前言、条款2、16、21、44根据从Addison-Wesley出版社下载的开放条款翻译。条款26、27、28、45根据从Scott Meyers的网站下载的《Three Guidelines for Effective Iterator Usage》一文翻译。条款43根据从C/C++ Users Journal网站下载的《STL Algorithms vs. Hand-Written Loops》一文翻译，条款45根据从C/C++ Users Journal网站下载的《Distinguishing STL Search Algorithms》一文翻译。其余部分根据epubcn放出的电子书制作。

翻译制作：龚敏敏 2004/1/2
游戏引擎开发网
PDF制作： 抚琴舞剑 2005/4/26

由于目录用到了javascript，所以在WinXP SP2安装后会出现所谓的“安全保护”，这是只需选择“允许阻止的内容...”即可。强烈推荐使用FireFox 0.9以上观看，从开放源代码、技术含量、安全性、速度、对标准的支持度等角度看，FireFox都远远优于IE。而且将会遇到麻烦还少一些。