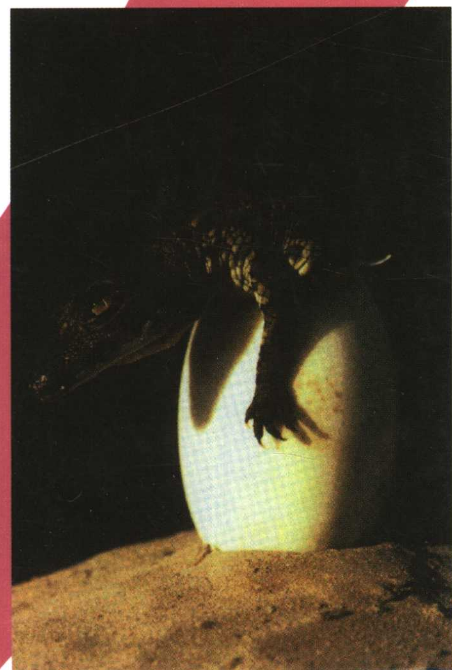
PEARSON
Addison Wesley

Effective STL 中文版

50条有效使用STL的经验

[美] Scott Meyers 著
潘爱民 陈铭 邹开红 译



ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

清华大学出版社

Effective STL 中文版

50 条有效使用 STL 的经验

[美]Scott Meyers 著

潘爱民 陈 铭 邹开红 译

清华大学出版社

北 京

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

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

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

EISBN: 0-201-74962-9

All Rights Reserved.

Published by arrangement with the original publisher, Pearson Education, Inc., publishing as **Addison-Wesley.**
This edition is authorized for sale only in the People's Republic of China (excluding the Special Administrative Region of Hong Kong and Macao).
本书中文简体翻译版由 Pearson Education 授权给清华大学出版社在中国境内(不包括中国香港、澳门特别行政区)出版发行。

北京市版权局著作权合同登记号　图字：01-2004-5452

版权所有，翻印必究。举报电话：010-62782989　13501256678　13801310933
本书封面贴有 Pearson Education(培生教育出版集团)激光防伪标签，无标签者不得销售。

图书在版编目(CIP)数据

Effective STL 中文版：50 条有效使用 STL 的经验 / （美）迈耶斯（Meyers,S.）著；潘爱民，陈铭，邹开红译. —北京：清华大学出版社，2006.4
书名原文：Effective STL
ISBN 7-302-12695-X

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

中国版本图书馆 CIP 数据核字（2006）第 020616 号

出 版 者：清华大学出版社　　**地　址**：北京清华大学学研大厦
http://www.tup.com.cn　　**邮　编**：100084
社 总 机：010-62770175　　**客户服务**：010-62776969
责任编辑：汤斌浩
印 刷 者：北京中科印刷有限公司
装 订 者：北京市密云县京文制本装订厂
发 行 者：新华书店总店北京发行所
开　本：185×230　**印张**：13.75　**字数**：309 千字
版　次：2006 年 4 月第 1 版　2006 年 4 月第1 次印刷
书　号：ISBN 7-302-12695-X/TP・8109
印　数：1～4000
定　价：30.00 元

译　序

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

C++是一门易学难用的编程语言，从学会使用C++到用好C++需要经过不断的实践。Scott Meyers的这三本姊妹作分别从各个不同的角度来帮助你缩短这个过程。C++语言经过了近20年的发展，正在渐趋完善。尽管如此，在使用C++语言的时候，仍然有许多陷阱，有的陷阱非常显然，一经点拨就可以明白；而有的陷阱则不那么直截了当，需要仔细地分析才能揭开那层神秘的面纱。

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

STL的源码规模并不大，但是它蕴含的思想非常深刻。在C++标准化的过程中，STL也被定格和统一。对于每一个STL实现，我们所看到的被分为两部分：一是STL的接口，这是应用程序赖以打交道的基础，也是我们所熟知的STL；二是STL的实现，特别是一些内部的机理，有的机理是C++标准所规定的，但是有的却是实现者自主选择的。在软件设计领域中有一条普遍适用的规则是“接口与实现分离”，但是对于STL，你不能简单地使用这条规则。虽然你写出来的程序代码只跟STL的接口打交道，但是用好STL则需要建立在充分了解STL实现的基础之上。你不仅需要了解对所有STL实现都通用的知识，也要了解针对你所使用的STL实现的特殊知识。那么，应该如何来把握接口与实现之间的关系呢？本书讲述了许多既涉及接口也关系到具体实现的STL用法，通过对这些用法的讲解，读者可以更加清楚地了解应该如何来看待这些与实现相关的知识。

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

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

- 如何选择容器的类型。STL中容器的类型并不多，但是不同的容器有不同的特点，

所以选择恰当的容器类型往往是解决问题的起点。本书中还特别介绍了与 vector 和 string 两种容器有关的一些注意事项。

- 涉及到关联容器有更多的陷阱，一不留神就可能陷入其中。作者专门指出了关联容器中一些并不直观的要点，还介绍了一种非标准的关联容器——哈希容器。
- 迭代器是 STL 中指针的泛化形式，也是程序员访问容器的重要途径。本书讨论了与 const_iterator 和 reverse_iterator 有关的一些问题。以我个人之见，本书这部分内容略显单薄，毕竟迭代器在 STL 中是一个非常关键的组件。
- STL 算法是体现 STL 功能的地方，一个简单的算法调用或许完成了一件极为复杂的事情，但是要用好 STL 中众多的算法并不容易，本书给出了一些重要的启示。
- 函数对象是 STL 中用到的关键武器之一，它使得 STL 中每一个算法都具有极强的扩展性，本书也特别讨论了涉及到函数和函数对象的一些要点。
- 其他的方方面面，包括在算法和同名成员函数之间如何进行区别，如何考虑程序的效率，如何保持程序的可读性，如何解读调试信息，关于移植性问题的考虑，等等。

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

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

对于每一个期望将 STL 用得更好的人，这本书值得一读。

潘爱民

2003 年 6 月 16 日

北京大学燕北园

试读结束：需要全本请在线购买： www.ertongbook.com

前　　言

> It came without ribbons! It came without tags!
> It came without packages, boxes or bags!
>
> —Dr. Seuss, *How the Grinch Stole Christmas!*, Random House, 1957

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

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

然后，我开始注意到了一些让我非常惊讶的事情。尽管 STL 存在可移植性问题，尽管它的文档并不完整，尽管编译器的诊断信息有如传输线上的噪音一样，但是，我的许多咨询客户正在使用 STL。而且，他们并不只是把 STL 拿来玩一玩，而是在用它开发实际的产品。这是一个很重要的启示。过去我知道 STL 是一个设计非常考究的模板库，这时我逐渐感觉到，既然程序员们愿意忍受移植性的麻烦、不够完整的文档以及难以理解的错误消息，那么这个库除了良好的设计以外，一定还有其他更多的优势。随着专业程序员的数量越来越多，我意识到，即使是一个很差的 STL 实现，也胜过没有实现。

更进一步，我知道 STL 的境况正在好转。C++库和编译器越来越多地遵从 C++标准，

好的文档也开始出现了(请参考本书后面所附的参考资料目录)，而且编译器的诊断信息也在改进(不过我们还需要等待它们改进得更好，在此期间，你可以参考第 49 条给出的一些针对如何处理诊断信息的建议)。因此我决定投身到这场 STL 运动中，尽我的一份绵薄之力。本书就是我努力的结果：50 条有效使用 STL 的经验。

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

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

我为本书维护了一份自第一次印刷以来的修订纪录，其中包括错误的改正情况、一些说明以及技术更新。你可以在本书的勘误页面上找到这份纪录：http://www.aristeia.com/BookErrata/estl1e-errata.html。

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

Scott Douglas Meyers
http://www.aristeia.com/
STAFFORD, OREGON
2001.4

致　谢

我差不多用了两年时间才真正对 STL 有所认识，同时设计了一门关于 STL 的培训课程以及著写了本书，在此过程中，我得到了来自许多途径的帮助。在所有的帮助之中，有两个尤其重要。第一是 Mark Rodgers，当我设计培训材料的时候，Mark 总是自愿审查这些材料，而且，我从他身上学到的关于 STL 的知识，比从其他任何人身上学到的都要多得多。他还担当了本书的技术审稿人，给出了许多富有洞察力的意见和建议，几乎每一个条款都得益于他的这些意见和建议。

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

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

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

在第 1 条中，我的见解“基于节点的容器为事务语义提供了更好的支持”建立在 Josuttis 的 *The C++ Standard Library*[3]的 5.11.2 小节的基础之上。第 2 条包含的一个例子来自于 Mark Rodgers 的关于 typedef 如何在分配子改变的情况下能有所帮助的论述。第 5 条得到了 Reeves 在 *C++ Report* 上的专栏“STL Gotchas”[17]的启发。第 8 条来源于 Sutter 的 *Exceptional C++*[8]中的第 37 条，以及 Kevlin Henney 提供的关于“auto_ptr 的容器在实践中如何未能工作”的重要细节。Matt Austern 在 Usenet 的帖子中提供了一些关于分配子何时有用的例子，我把他的例子包含在第 11 条中。第 12 条建立在 SGI STL Web 站点[21]上关于线程安全性的讨论的基础之上。第 13 条中有关在多线程环境下引用计数技术性能问题的材料来自于 Sutter 在这个话题上的文章[20]。第 15 条的想法来自于 Reeves 在 *C++ Report* 上的专栏“Using Standard string in the Real World, Part 2”[18]。在第 16 条中，Mark Rodgers 提出了我所展示的技术，即，让一个 C API 将数据直接写到一个 vector 中。第 17 条包含了

Siemel Naran 和 Carl Barron 在 Usenet 上张贴的信息。我窃取了 Sutter 在 *C++ Report* 上的专栏“When Is a Container Not a Container?”[12]作为第 18 条。在第 20 条中，Mark Rodgers 贡献了“通过一个解除指针引用函数子把一个指针转换成一个对象”的想法，Scott Lewandowski 提出了我所展示的 DereferenceLess 的版本。第 21 条起源于 Doug Harrison 张贴在 microsoft.public.vc.stl 上的内容，但是将该条款的焦点限定在等值上的决定则是我自己做出的。第 22 条则是建立在 Sutter 在 *C++ Report* 上的专栏“Standard Library News: sets and maps”[13]的基础之上的；Matt Austern 帮助我理解了标准化委员会的 Library Issue #103 的状况。第 23 条得到了 Austern 在 *C++ Report* 上的文章“Why you Shouldn't Use set – and What to Use Instead”[15]的启发；David Smallberg 为我的 DataCompare 实现做了更为精细的加工。我介绍的 Dinkumware 哈希容器建立在 Plauger 在 *C/C++ Users Journal* 上的专栏“Hash Tables”[16]的基础之上。Mark Rodgers 并不赞成第 26 条的全部建议，但是该条款原先的一个动机是，他观察到有些容器的成员函数只接受 iterator 类型的实参。我选择第 29 条是因为 Usenet 上 Matt Austern 和 James Kanze 所参与的一些讨论，同时我也受到了 Kreft 和 Langer 发表在 *C++ Report* 上的文章“A Sophisticated Implementation of User-Defined Inserters and Extractors”[25]的影响。第 30 条是由于 Josuttis 在 *The C++ Standard Library*[3]的 5.4.2 小节中的讨论。在第 31 条中，Marco Dalla Gasperina 贡献了利用 nth_element 来计算中间值的示例用法，通过该算法来找到百分比的用法则直接来自于 Stroustrup 的 *The C++ Programming Language*[7]的 18.7.1 小节。第 31 条受到了 Josuttis 在 *The C++ Standard Library*[3]的 5.6.1 小节中的材料的影响。第 35 条起源于 Austern 在 *C++ Report* 上的专栏“How to Do Case-Insensitive String Comparison”[11]，而且，James Kanze 以及 John Potter 在 clcm 上的帖子帮助我加深了对于所涉及到的各个问题的理解。我在第 36 条中所展示的 copy_if 实现来自于 Stroustrup 的 *The C++ Programming Language*[7]。第 37 条在很大程度上得到了 Josuttis 的多份出版物的启发，他在 *The C++ Standard Library*[3]、在 Standard Library Issue #92，以及在其 *C++ Report* 的文章“Predicates vs. Function Objects”[14]中讲述了关于“stateful predicates”的内容。在我的介绍中，我使用了他的例子，并且推荐了他提出的一种方案，不过，我使用了我自己的术语“纯函数”。Matt Austern 证实了我在第 41 条中关于术语 mem_fun 和 mem_fun_ref 的历史的猜测。第 42 条可以追溯到当我考虑是否可以违反该指导原则时，我从 Mark Rodgers 处得到的一份讲稿。Mark Rodgers 也贡献了第 44 条中的见解：在 map 和 multimap 上的非成员搜索操作会检查每个元素的两个组件，而成员搜索操作则只检查每个元素的第一个组件(键)。第 45 条包含了众多 clcm 发贴者贡献的信息，其中包括 John Potter、Marcin Kasperski、Pete Becker、Dennis Yelle 和 David Abrahams。David Smallberg 提醒我，在执行基于等价性的搜索，以及在排序的序列容器上进行计数时，要注意 equal_range 的用法。Andrei Alexandrescu 帮助我更好地理解了第 50 条中讲述的“指向引用的引用”问题所发生的条件；针对此问题，我在 Mark Rodgers 所提供的例子(在 Boost Web 站点[22])的基础上，也模仿了一个类似的例子。

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

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

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

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

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

本书的印刷比以前的印刷要好得多，因为有一些目光敏锐的读者将问题指出来，所以我有机会解决这些问题，他们是：Jon Webb、Michael Hawkins、Derek Price、Jim Scheller、Carl Manaster、Herb Sutter、Albert Franklin、George King、Dave Miller、Harold Howe、John Fuller、Tim McCarthy、John Hershberger、Igor Mikolic-Torreira、Stephen Bergmann、Robert Allan Schwartz、John Potter、David Grigsby、Sanjay Pattni、Jesper Andersen、Jing Tao Wang、André Blavier、Dan Schmidt、Bradley White、Adam Petersen、Wayne Goertel、Gabriel Netterdag、Jason Kenny、Scott Blachowicz、Seyed H. Haeri、Gareth McCaughan、Giulio Agostini、Fraser Ross、Wolfram Burkhardt、Keith Stanley、Leor Zolman、Chan Ki Lok、Motti Abramsky、Kevlin Henney、Stefan Kuhlins、Phillip Ngan、Jim Phillips、Ruediger Dreier、Guru Chander、Charles Brockman，以及 Day Barr。我要感谢他们，正是他们的帮助改进了 *Effective STL* 的印刷。

我在 Addison-Wesley 的合作者包括 John Wait（我的编辑，现在已经是一位高级副总裁了）、Alicia Carey 和 Susannah Buzard（他的第 n 位和第 n+1 位助手）、John Fuller（产品协调人）、Karin Hansen（封面设计者）、Jason Jones（全才的技术高手，尤其是在 Adobe 开发的恐怖的排版软件方面）、Marty Rabinowitz（他们的老板，但是他自己也工作），以及 Curt Johnson、Chanda Leary-Coutu 和 Robin Bruce（都是市场人才，但都非常友善）。

而 Abbi Staley 则让我觉得周日的午餐是一种美好的享受。

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

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

试读结束：需要全本请在线购买： www.ertongbook.com

目　　录

引言……1

第 1 章　容器……9

第 1 条：慎重选择容器类型。……9

第 2 条：不要试图编写独立于容器类型的代码。……12

第 3 条：确保容器中的对象拷贝正确而高效。……16

第 4 条：调用 empty 而不是检查 size()是否为 0。……18

第 5 条：区间成员函数优先于与之对应的单元素成员函数。……19

第 6 条：当心 C++编译器最烦人的分析机制。……26

第 7 条：如果容器中包含了通过 new 操作创建的指针，切记在容器对象析构前将指针 delete 掉。……28

第 8 条：切勿创建包含 auto_ptr 的容器对象。……32

第 9 条：慎重选择删除元素的方法。……34

第 10 条：了解分配子(allocator)的约定和限制。……38

第 11 条：理解自定义分配子的合理用法。……43

第 12 条：切勿对 STL 容器的线程安全性有不切实际的依赖。……46

第 2 章　vector 和 string……51

第 13 条：vector 和 string 优先于动态分配的数组。……51

第 14 条：使用 reserve 来避免不必要的重新分配。……53

第 15 条：注意 string 实现的多样性。……55

第 16 条：了解如何把 vector 和 string 数据传给旧的 API。……59

第 17 条：使用“swap 技巧”除去多余的容量。……62

第 18 条：避免使用 vector<bool>。……64

第 3 章　关联容器……67

第 19 条：理解相等(equality)和等价(equivalence)的区别。……67

第 20 条：为包含指针的关联容器指定比较类型。……71

第 21 条：总是让比较函数在等值情况下返回 false。……74

第 22 条：切勿直接修改 set 或 multiset 中的键。……77

第 23 条：考虑用排序的 vector 替代关联容器。……82

第 24 条：当效率至关重要时，请在 map::operator[]与 map::insert 之间谨慎做出选择。……87

第 25 条：熟悉非标准的哈希容器。 …… 91
第 4 章 迭代器 …… 95
第 26 条：iterator 优先于 const_iterator、reverse_iterator 以及 const_reverse_iterator。 …… 95
第 27 条：使用 distance 和 advance 将容器的 const_iterator 转换成 iterator。 …… 98
第 28 条：正确理解由 reverse_iterator 的 base()成员函数所产生的 iterator 的用法。 …… 101
第 29 条：对于逐个字符的输入请考虑使用 istreambuf_iterator。 …… 103
第 5 章 算法 …… 106
第 30 条：确保目标区间足够大。 …… 106
第 31 条：了解各种与排序有关的选择。 …… 110
第 32 条：如果确实需要删除元素，则需要在 remove 这一类算法之后调用 erase。 …… 115
第 33 条：对包含指针的容器使用 remove 这一类算法时要特别小心。 …… 118
第 34 条：了解哪些算法要求使用排序的区间作为参数。 …… 121
第 35 条：通过 mismatch 或 lexicographical_compare 实现简单的忽略大小写的字符串比较。 …… 124
第 36 条：理解 copy_if 算法的正确实现。 …… 128
第 37 条：使用 accumulate 或者 for_each 进行区间统计。 …… 130
第 6 章 函数子、函数子类、函数及其他 …… 135
第 38 条：遵循按值传递的原则来设计函数子类。 …… 135
第 39 条：确保判别式是“纯函数”。 …… 139
第 40 条：若一个类是函数子，则应使它可配接。 …… 142
第 41 条：理解 ptr_fun、mem_fun 和 mem_fun_ref 的来由。 …… 145
第 42 条：确保 less<T>与 operator<具有相同的语义。 …… 149
第 7 章 在程序中使用 STL …… 153
第 43 条：算法调用优先于手写的循环。 …… 153
第 44 条：容器的成员函数优先于同名的算法。 …… 160
第 45 条：正确区分 count、find、binary_search、lower_ bound、upper_bound 和 equal_range。 …… 162
第 46 条：考虑使用函数对象而不是函数作为 STL 算法的参数。 …… 170
第 47 条：避免产生“直写型”(write-only)的代码。 …… 174
第 48 条：总是包含(#include)正确的头文件。 …… 177
第 49 条：学会分析与 STL 相关的编译器诊断信息。 …… 178
第 50 条：熟悉与 STL 相关的 Web 站点。 …… 185
参考书目 …… 191
附录 A：地域性与忽略大小写的字符串比较 …… 195
附录 B：对 Microsoft 的 STL 平台的说明 …… 204

引　言

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

所以我为你写了这本书。

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

本书中的信息将会使你成为一位更优秀的 STL 程序员；它会使你成为一位高效率、高产出的程序员；它还会使你成为一位快乐的程序员。使用 STL 很令人开心，但是有效地使用它则令人更开心，这种开心来源于它会使你有更多的时间离开键盘，因为你可能不相信自己会节省这么多时间。即便是对 STL 粗粗浏览一遍，也能发现这是一个非常酷的库，但你可能想象不到实际上它还要酷得多(无论是深度还是广度)。本书的一个主要目标是向你展示这个库是多么令人惊奇，因为在我从事程序设计近三十年来，我从来没看到过可以与 STL 相媲美的代码库。可能你也没见过。

定义、使用和扩展 STL

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

从技术上讲，我对 STL 的定义不包括标准 C++库的扩展部分，尤其是哈希容器、单向链表、rope 以及许多非标准的函数对象。即便如此，一个高效的 STL 程序员需要意识到这

种扩展，所以在适当的时候我也会提及。实际上，第 25 条是专门针对非标准的哈希容器的一般性介绍。现在它们还不在 STL 中，但是一些与之类似的东西肯定会进入到下一个版本的标准 C++库中，我们展望一下未来总是有价值的。

STL 之所以存在扩展，其中一个原因是，STL 的设计目的就是为了便于扩展。但在本书中，我将把焦点放在*如何使用* STL 上，而不是如何向其中添加新的部件。比如，你会发现，我将很少讲述如何编写自己的算法，对于如何编写新的容器和迭代器也没有给出任何建议。我相信，在考虑增强 STL 的能力之前，首先重要的是掌握 STL 已经提供了什么，而这正是本书的焦点所在。当你决定创建自己的类似 STL 的部件时，你可以在 Josuttis 的 *The C++ Standard Library*[3]和 Austern 的 *Generic Programming and the STL*[4]中找到相关的建议，它们会告诉你如何做到这一点。然而，在本书中，我还是会讨论到 STL 扩展的一个方面，即怎样编写自己的函数对象。如果不知道怎样编写自己的函数对象，你就无法有效地使用 STL，所以我将花一整章的篇幅(第 6 章)来重点讲述这一话题。

引文

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

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

STL 和标准

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

有时候，最直接的方式是使用不同的 STL 实现。附录 B 给出了一个这样的例子。你对 STL 的使用越多，就越有必要区分你的编译器和你的库实现。当程序员试图使合法的代码

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

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

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

引用计数

如果不提到引用计数技术而来讨论 STL，这几乎是不可能的。在第 7 条和第 31 条中你将会看到，凡是涉及指针容器的设计几乎无一例外地会用到引用计数。另外，很多 string 实现的内部也使用了引用计数技术，正如在第 15 条中指出的那样，这是你无法忽略的一个实现细节。在本书中，我将假设你熟悉有关引用计数的一些基本知识。如果你不熟悉，大多数中级和高级的 C++书籍都涉及到了这一话题。比如，在 *More Effective C++*中，相关的材料在第 28 条和第 29 条中。如果你不知道引用计数是什么，而且你也不想知道，那么，请不要着急，你仍然可以读懂本书，尽管会在这里或那里有一些句子你可能不太懂。

string 和 wstring

我所说的关于 string 的内容同样也适用于与它对应的宽字节字符串 wstring。同样，当提

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

术语，术语，术语

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

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

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

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

  随机访问迭代器(random access iterator)有双向迭代器的所有功能，而且，它还提供了“迭代器算术”，即在一步内向前或向后跳跃的能力。vector、string 和 deque 都提供了随机访问迭代器。指向数组内部的指针对于数组来说也是随机访问迭代器。
- 所有重载了函数调用操作符(即 operator())的类都是一个函数子类(functor class)。从这些类创建的对象被称为函数对象(function object)或函数子(functor)。在 STL 中，大多数使用函数对象的地方同样也可以使用实际的函数，所以我经常使用“函数对象”(function object)这个术语既表示 C++函数，也表示真正的函数对象。
- 函数 bind1st 和 bind2nd 被称为绑定器(binder)。

试读结束：需要全本请在线购买： www.ertongbook.com