HZ BOOKS

PEARSON Addison Wesley

Modern C++ Design C++ 设计新思维

STL高效编程

Effective STL

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

(英文版)

(美) Scott Meyers 著

机械工业出版社
China Machine Press

Modern C++ Design
C++
设计新思维

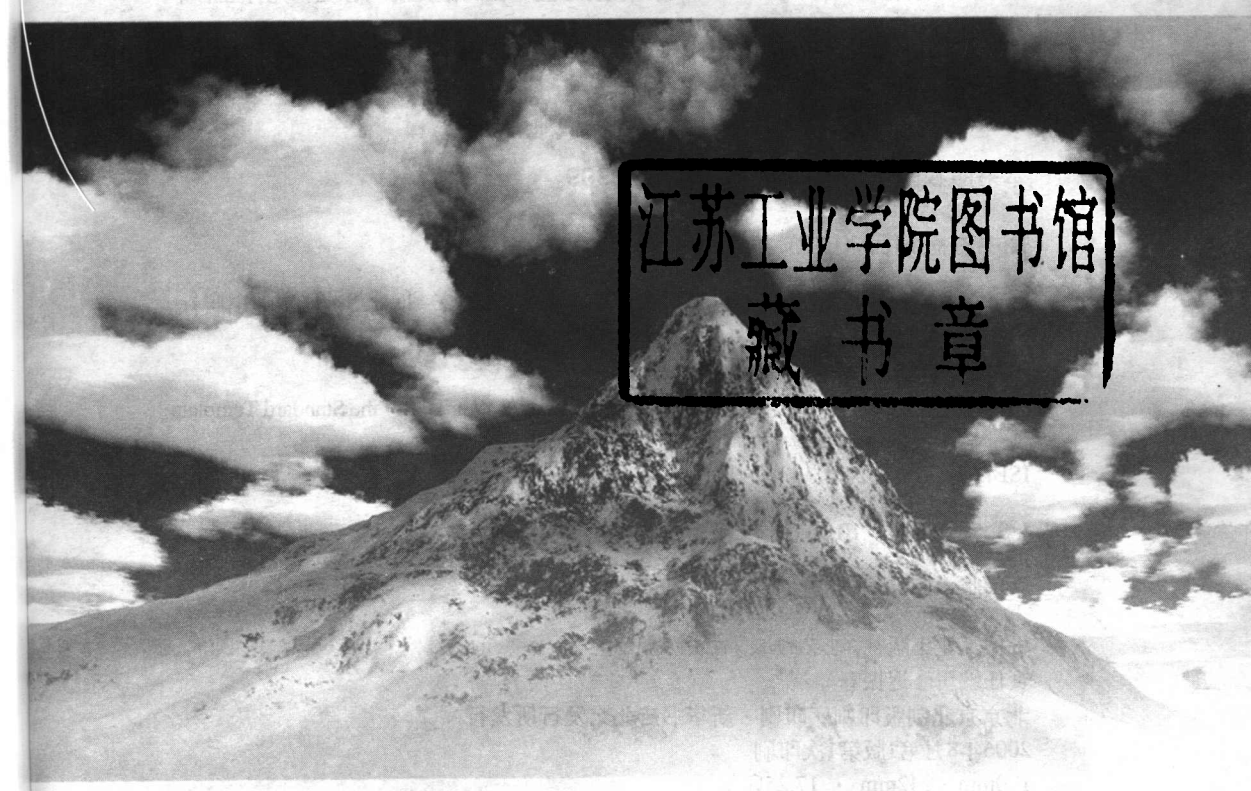
STL高效编程

（英文版）

Effective STL

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

（美） Scott Meyers 著



江苏工业学院图书馆
藏书章

机械工业出版社
China Machine Press

English reprint edition copyright © 2006 by Pearson Education Asia Limited and China Machine Press.

Original English language title: *Effective STL: 50 Specific Ways to Improve Your Use of the Standard Template Library* (ISBN 0-201-74962-9) by Scott Meyers, Copyright © 2001 by Addison-Wesley.

All rights reserved.

Published by arrangement with the original publisher, Pearson Education, Inc., publishing as Addison-Wesley.

For sale and distribution in the People's Republic of China exclusively (except Taiwan, Hong Kong SAR and Macau SAR).

本书英文影印版由Pearson Education Asia Ltd.授权机械工业出版社独家出版。未经出版者书面许可，不得以任何方式复制或抄袭本书内容。

仅限于中华人民共和国境内（不包括中国香港、澳门特别行政区和中国台湾地区）销售发行。

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

版权所有，侵权必究。

本书法律顾问　北京市展达律师事务所

本书版权登记号：图字：01-2006-3999

图书在版编目（CIP）数据

STL高效编程（英文版）/（美）迈耶斯（Meyers, S.）著. -北京：机械工业出版社，2006.8

（C++设计新思维）

书名原文：Effective STL: 50 Specific Ways to Improve Your Use of the Standard Template Library

ISBN 7-111-19624-4

I. S… II. 迈… III. C语言－程序设计－英文 IV. TP312

中国版本图书馆CIP数据核字（2006）第080925号

机械工业出版社（北京市西城区百万庄大街22号　邮政编码　100037）

责任编辑：迟振春

北京京北制版印刷厂印刷 · 新华书店北京发行所发行

2006年8月第1版第1次印刷

170mm × 242mm · 17.25印张

定价：32元

凡购本书，如有倒页、脱页、缺页，由本社发行部调换

本社购书热线：（010）68326294

“C++设计新思维”丛书前言

自C++诞生尤其是ISO/ANSI C++标准问世以来，以Bjarne Stroustrup为首的C++社群领袖一直不遗余力地倡导采用“新风格”教学和使用C++。事实证明，除了兼容于C的低阶特性外，C++提供的高级特性以及在此基础上发展的各种惯用法可以让我们编写出更加简洁、优雅、高效、健壮的程序。

这些高级特性和惯用法包括精致且高效的标准库和各种“准标准库”，与效率、健壮性、异常安全等主题有关的各种惯用法，以及在C++的未来占据更重要地位的模板和泛型程序设计技术等。它们发展于力量强大的C++社群，并被这个社群中最负声望的专家提炼、升华成一本本精彩的著作。毫无疑问，这些学术成果必将促进C++社群创造出更多的实践成果。

我个人认为，包括操作系统、设备驱动、编译器、系统工具、图像处理、数据库系统以及通用办公软件等在内的基础软件更能够代表一个国家的软件产业发展质量，迄今为止，此类基础性的软件恰好是C++所擅长开发的，因此，可以感性地说，C++的应用水平在一定程度上可以折射出一个国家的软件产业发展水平和健康程度。

前些年国内曾引进出版了一大批优秀的C++书籍，它们拓宽了中国C++程序员的视野，并在很大程度上纠正了长期以来存在于C++的教育、学习和使用方面的种种误解，对C++相关的产业发展起到了一定的促进作用。然而在过去的两年中，随着.NET、Java技术吸引越来越多的注意力，中国软件产业业务化、项目化的状况愈发加剧，擅长于“系统编程”的C++语言的应用领域似有进一步缩减的趋势，这也导致人们对C++的出版教育工作失去了应有的重视。

机械工业出版社华章分社决定继续为中国C++“现代化”教育推波助澜，从2006年起将陆续推出一套“C++设计新思维”丛书。这套丛书秉持精品、高端的理念，其作译者为包括Herb Sutter在内的国内外知名C++技术专家和研究者、教育者，议题紧密围绕现代C++特性，以实用性为主，兼顾实验性和探索性，形式上则是原版影印、中文译著和原创兼收并蓄。每一本书相对独立且交叉引用，篇幅短小却内容深入。作为这套丛书的特邀技术编辑，我衷心希望它们所展示的技术、技巧和理念能够为中国C++社群注入新的活力。

荣 耀
2005年12月
南京师范大学
www.royaloo.com

For Woofieland.

试读结束：需要全本请在线购买： www.ertongbook.com

Preface

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

I first wrote about the Standard Template Library in 1995, when I concluded the final Item of *More Effective C++* with a brief STL overview. I should have known better. Shortly thereafter, I began receiving mail asking when I'd write *Effective STL*.

I resisted the idea for several years. At first, I wasn't familiar enough with the STL to offer advice on it, but as time went on and my experience with it grew, this concern gave way to other reservations. There was never any question that the library represented a breakthrough in efficient and extensible design, but when it came to *using* the STL, there were practical problems I couldn't overlook. Porting all but the simplest STL programs was a challenge, not only because library implementations varied, but also because template support in the underlying compilers ranged from good to awful. STL tutorials were hard to come by, so learning "the STL way of programming" was difficult, and once that hurdle was overcome, finding comprehensible and accurate reference documentation was a challenge. Perhaps most daunting, even the smallest STL usage error often led to a blizzard of compiler diagnostics, each thousands of characters long, most referring to classes, functions, or templates not mentioned in the offending source code, almost all incomprehensible. Though I had great admiration for the STL and for the people behind it, I felt uncomfortable recommending it to practicing programmers. I wasn't sure it was *possible* to use the STL effectively.

Then I began to notice something that took me by surprise. Despite the portability problems, despite the dismal documentation, despite the compiler diagnostics resembling transmission line noise, many of

my consulting clients were using the STL anyway. Furthermore, they weren't just playing with it, they were using it in production code! That was a revelation. I knew that the STL featured an elegant design, but any library for which programmers are willing to endure portability headaches, poor documentation, and incomprehensible error messages has a lot more going for it than just good design. For an increasingly large number of professional programmers, I realized, even a bad implementation of the STL was preferable to no implementation at all.

Furthermore, I knew that the situation regarding the STL would only get better. Libraries and compilers would grow more conformant with the Standard (they have), better documentation would become available (it has — consult the bibliography beginning on page 225), and compiler diagnostics would improve (for the most part, we're still waiting, but Item 49 offers suggestions for how to cope while we wait). I therefore decided to chip in and do my part for the STL movement. This book is the result: 50 specific ways to improve your use of C++'s Standard Template Library.

My original plan was to write the book in the second half of 1999, and with that thought in mind, I put together an outline. But then I changed course. I suspended work on the book and developed an introductory training course on the STL, which I then taught several times to groups of programmers. About a year later, I returned to the book, significantly revising the outline based on my experiences with the training course. In the same way that my *Effective C++* has been successful by being grounded in the problems faced by real programmers, it's my hope that *Effective STL* similarly addresses the practical aspects of STL programming — the aspects most important to professional developers.

I am always on the lookout for ways to improve my understanding of C++. If you have suggestions for new guidelines for STL programming or if you have comments on the guidelines in this book, please let me know. In addition, it is my continuing goal to make this book as accurate as possible, so for each error in this book that is reported to me — be it technical, grammatical, typographical, or otherwise — I will, in future printings, gladly add to the acknowledgments the name of the first person to bring that error to my attention. Send your suggested guidelines, your comments, and your criticisms to estl@aristeia.com.

I maintain a list of changes to this book since its first printing, including bug-fixes, clarifications, and technical updates. The list is available at the *Effective STL Errata* web site, http://www.aristeia.com/BookErrata/estl1e-errata.html.

If you'd like to be notified when I make changes to this book, I encourage you to join my mailing list. I use the list to make announcements likely to be of interest to people who follow my work on C++. For details, consult http://www.aristeia.com/MailingList/.

SCOTT DOUGLAS MEYERS
http://www.aristeia.com/

STAFFORD, OREGON
APRIL 2001

Acknowledgments

I had an enormous amount of help during the roughly two years it took me to make some sense of the STL, create a training course on it, and write this book. Of all my sources of assistance, two were particularly important. The first is Mark Rodgers. Mark generously volunteered to review my training materials as I created them, and I learned more about the STL from him than from anybody else. He also acted as a technical reviewer for this book, again providing observations and insights that improved virtually every Item.

The other outstanding source of information was several C++-related Usenet newsgroups, especially comp.lang.c++.moderated ("clcm"), comp.std.c++, and microsoft.public.vc.stl. For well over a decade, I've depended on the participants in newsgroups like these to answer my questions and challenge my thinking, and it's difficult to imagine what I'd do without them. I am deeply grateful to the Usenet community for their help with both this book and my prior publications on C++.

My understanding of the STL was shaped by a variety of publications, the most important of which are listed in the Bibliography. I leaned especially heavily on Josuttis' *The C++ Standard Library* [3].

This book is fundamentally a summary of insights and observations made by others, though a few of the ideas are my own. I've tried to keep track of where I learned what, but the task is hopeless, because a typical Item contains information garnered from many sources over a long period of time. What follows is incomplete, but it's the best I can do. Please note that my goal here is to summarize where *I* first learned of an idea or technique, not where the idea or technique was originally developed or who came up with it.

In Item 1, my observation that node-based containers offer better support for transactional semantics is based on section 5.11.2 of Josuttis' *The C++ Standard Library* [3]. Item 2 includes an example from Mark Rodgers on how typedefs help when allocator types are changed.

Item 5 was motivated by Reeves' *C++ Report* column, "STL Gotchas" [17]. Item 8 sprang from Item 37 in Sutter's *Exceptional C++* [8], and Kevlin Henney provided important details on how containers of auto_ptrs fail in practice. In Usenet postings, Matt Austern provided examples of when allocators are useful, and I include his examples in Item 11. Item 12 is based on the discussion of thread safety at the SGI STL web site [21]. The material in Item 13 on the performance implications of reference counting in a multithreaded environment is drawn from Sutter's writings on this topic [20]. The idea for Item 15 came from Reeves' *C++ Report* column, "Using Standard string in the Real World, Part 2," [18]. In Item 16, Mark Rodgers came up with the technique I show for having a C API write data directly into a vector. Item 17 includes information from Usenet postings by Siemel Naran and Carl Barron. I stole Item 18 from Sutter's *C++ Report* column, "When Is a Container Not a Container?" [12]. In Item 20, Mark Rodgers contributed the idea of transforming a pointer into an object via a dereferencing functor, and Scott Lewandowski came up with the version of DereferenceLess I present. Item 21 originated in a Doug Harrison posting to microsoft.public.vc.stl, but the decision to restrict the focus of that Item to equality was mine. I based Item 22 on Sutter's *C++ Report* column, "Standard Library News: sets and maps" [13]; Matt Austern helped me understand the status of the Standardization Committee's Library Issue #103. Item 23 was inspired by Austern's *C++ Report* article, "Why You Shouldn't Use set — and What to Use Instead" [15]; David Smallberg provided a neat refinement for my implementation of DataCompare. My description of Dinkumware's hashed containers is based on Plauger's *C/C++ Users Journal* column, "Hash Tables" [16]. Mark Rodgers doesn't agree with the overall advice of Item 26, but an early motivation for that Item was his observation that some container member functions accept only arguments of type iterator. My treatment of Item 29 was motivated and informed by Usenet discussions involving Matt Austern and James Kanze; I was also influenced by Kreft and Langer's *C++ Report* article, "A Sophisticated Implementation of User-Defined Inserters and Extractors" [25]. Item 30 is due to a discussion in section 5.4.2 of Josuttis' *The C++ Standard Library* [3]. In Item 31, Marco Dalla Gasperina contributed the example use of nth_element to calculate medians, and use of that algorithm for finding percentiles comes straight out of section 18.7.1 of Stroustrup's *The C++ Programming Language* [7]. Item 32 was influenced by the material in section 5.6.1 of Josuttis' *The C++ Standard Library* [3]. Item 35 originated in Austern's *C++ Report* column "How to Do Case-Insensitive String Comparison" [11], and James Kanze's and John Potter's clcm postings helped me refine my understanding of the issues involved. Stroustrup's implementation for copy_if, which I

show in Item 36, is from section 18.6.1 of his *The C++ Programming Language* [7]. Item 39 was largely motivated by the publications of Josuttis, who has written about "stateful predicates" in his *The C++ Standard Library* [3], in Standard Library Issue #92, and in his *C++ Report* article, "Predicates vs. Function Objects" [14]. In my treatment, I use his example and recommend a solution he has proposed, though the use of the term "pure function" is my own. Matt Austern confirmed my suspicion in Item 41 about the history of the terms mem_fun and mem_fun_ref. Item 42 can be traced to a lecture I got from Mark Rodgers when I considered violating that guideline. Mark Rodgers is also responsible for the insight in Item 44 that non-member searches over maps and multimaps examine both components of each pair, while member searches examine only the first (key) component. Item 45 contains information from various clcm contributors, including John Potter, Marcin Kasperski, Pete Becker, Dennis Yelle, and David Abrahams. David Smallberg alerted me to the utility of equal_range in performing equivalence-based searches and counts over sorted sequence containers. Andrei Alexandrescu helped me understand the conditions under which "the reference-to-reference problem" I describe in Item 50 arises, and I modeled my example of the problem on a similar example provided by Mark Rodgers at the Boost Web Site [22].

Credit for the material in Appendix A goes to Matt Austern, of course. I'm grateful that he not only gave me permission to include it in this book, he also tweaked it to make it even better than the original.

Good technical books require a thorough pre-publication vetting, and I was fortunate to benefit from the insights of an unusually talented group of technical reviewers. Brian Kernighan and Cliff Green offered early comments on a partial draft, and complete versions of the manuscript were scrutinized by 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, and, of course, Mark Rodgers. Katrina Avery did the copyediting.

One of the most challenging parts of preparing a book is finding good technical reviewers. I thank John Potter for introducing me to Jared Manning and Aaron Campbell.

Herb Sutter kindly agreed to act as my surrogate in compiling, running, and reporting on the behavior of some STL test programs under a beta version of Microsoft's Visual Studio .NET, while Leor Zolman undertook the herculean task of testing all the code in this book. Any errors that remain are my fault, of course, not Herb's or Leor's.

试读结束：需要全本请在线购买： www.ertongbook.com

Angelika Langer opened my eyes to the indeterminate status of some aspects of STL function objects. This book has less to say about function objects than it otherwise might, but what it does say is more likely to remain true. At least I hope it is.

This printing of the book is better than earlier printings, because I was able to address problems identified by the following sharp-eyed readers: 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, Stephan 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 Chandar, Charles Brockman, Day Barr, Eric Niebler, Sharad Kala, Declan Moran, Nick de Smith, David Callaway, Shlomi Frank, Andrea Griffini, Hans Eckardt, David Smallberg, Matt Page, and Andy Fyfe. I'm grateful for their help in improving *Effective STL*.

My collaborators at Addison-Wesley included John Wait (my editor and now a senior VP), Alicia Carey and Susannah Buzard (his assistants *n* and *n*+1), John Fuller (the production coordinator), Karin Hansen (the cover designer), Jason Jones (all-around technical guru, especially with respect to the demonic software spewed forth by Adobe), Marty Rabinowitz (their boss, but he works, too), and Curt Johnson, Chanda Leary-Coutu, and Robin Bruce (all marketing people, but still very nice).

Abbi Staley made Sunday lunches a routinely pleasurable experience.

As she has for the six books and one CD that came before it, my wife, Nancy, tolerated the demands of my research and writing with her usual forbearance and offered me encouragement and support when I needed it most. She never fails to remind me that there's more to life than C++ and software.

And then there's our dog, Persephone. As I write this, it is her sixth birthday. Tonight, she and Nancy and I will visit Baskin-Robbins for ice cream. Persephone will have vanilla. One scoop. In a cup. To go.

Contents

Preface v

Acknowledgments ix

Introduction 1

Chapter 1: Containers 11

Item 1: Choose your containers with care. 11
Item 2: Beware the illusion of container-independent code. 15
Item 3: Make copying cheap and correct for objects in containers. 20
Item 4: Call empty instead of checking size() against zero. 23
Item 5: Prefer range member functions to their single-element counterparts. 24
Item 6: Be alert for C++'s most vexing parse. 33
Item 7: When using containers of newed pointers, remember to delete the pointers before the container is destroyed. 36
Item 8: Never create containers of auto_ptrs. 40
Item 9: Choose carefully among erasing options. 43
Item 10: Be aware of allocator conventions and restrictions. 48
Item 11: Understand the legitimate uses of custom allocators. 54
Item 12: Have realistic expectations about the thread safety of STL containers. 58

Chapter 2: vector and string 63

Item 13: Prefer vector and string to dynamically allocated arrays. 63
Item 14: Use reserve to avoid unnecessary reallocations. 66
Item 15: Be aware of variations in string implementations. 68

Item 16: Know how to pass vector and string data to legacy APIs. 74
Item 17: Use "the swap trick" to trim excess capacity. 77
Item 18: Avoid using vector<bool>. 79

Chapter 3: Associative Containers **83**

Item 19: Understand the difference between equality and equivalence. 83
Item 20: Specify comparison types for associative containers of pointers. 88
Item 21: Always have comparison functions return false for equal values. 92
Item 22: Avoid in-place key modification in set and multiset. 95
Item 23: Consider replacing associative containers with sorted vectors. 100
Item 24: Choose carefully between map::operator[] and map::insert when efficiency is important. 106
Item 25: Familiarize yourself with the nonstandard hashed containers. 111

Chapter 4: Iterators **116**

Item 26: Prefer iterator to const_iterator, reverse_iterator, and const_reverse_iterator. 116
Item 27: Use distance and advance to convert a container's const_iterators to iterators. 120
Item 28: Understand how to use a reverse_iterator's base iterator. 123
Item 29: Consider istreambuf_iterators for character-by-character input. 126

Chapter 5: Algorithms **128**

Item 30: Make sure destination ranges are big enough. 129
Item 31: Know your sorting options. 133
Item 32: Follow remove-like algorithms by erase if you really want to remove something. 139
Item 33: Be wary of remove-like algorithms on containers of pointers. 143
Item 34: Note which algorithms expect sorted ranges. 146
Item 35: Implement simple case-insensitive string comparisons via mismatch or lexicographical_compare. 150
Item 36: Understand the proper implementation of copy_if. 154

Item 37: Use accumulate or for_each to summarize ranges. 156

Chapter 6: Functors, Functor Classes, Functions, etc. **162**

Item 38: Design functor classes for pass-by-value. 162
Item 39: Make predicates pure functions. 166
Item 40: Make functor classes adaptable. 169
Item 41: Understand the reasons for ptr_fun, mem_fun, and mem_fun_ref. 173
Item 42: Make sure less<T> means operator<. 177

Chapter 7: Programming with the STL **181**

Item 43: Prefer algorithm calls to hand-written loops. 181
Item 44: Prefer member functions to algorithms with the same names. 190
Item 45: Distinguish among count, find, binary_search, lower_bound, upper_bound, and equal_range. 192
Item 46: Consider function objects instead of functions as algorithm parameters. 201
Item 47: Avoid producing write-only code. 206
Item 48: Always #include the proper headers. 209
Item 49: Learn to decipher STL-related compiler diagnostics. 210
Item 50: Familiarize yourself with STL-related web sites. 217

Bibliography **225**

Appendix A: Locales and Case-Insensitive String Comparisons **229**

Appendix B: Remarks on Microsoft's STL Platforms **239**

Index **245**

Introduction

You're already familiar with the STL. You know how to create containers, iterate over their contents, add and remove elements, and apply common algorithms, such as find and sort. But you're not satisfied. You can't shake the sensation that the STL offers more than you're taking advantage of. Tasks that should be simple aren't. Operations that should be straightforward leak resources or behave erratically. Procedures that should be efficient demand more time or memory than you're willing to give them. Yes, you know how to use the STL, but you're not sure you're using it *effectively*.

I wrote this book for you.

In *Effective STL*, I explain how to combine STL components to take full advantage of the library's design. Such information allows you to develop simple, straightforward solutions to simple, straightforward problems, and it also helps you design elegant approaches to more complicated problems. I describe common STL usage errors, and I show you how to avoid them. That helps you dodge resource leaks, code that won't port, and behavior that is undefined. I discuss ways to optimize your code, so you can make the STL perform like the fast, sleek machine it is intended to be.

The information in this book will make you a better STL programmer. It will make you a more productive programmer. And it will make you a happier programmer. Using the STL is fun, but using it effectively is outrageous fun, the kind of fun where they have to drag you away from the keyboard, because you just can't believe the good time you're having. Even a cursory glance at the STL reveals that it is a wondrously cool library, but the coolness runs broader and deeper than you probably imagine. One of my primary goals in this book is to convey to you just how amazing the library is, because in the nearly 30 years I've been programming, I've never seen anything like the STL. You probably haven't either.

Defining, Using, and Extending the STL

There is no official definition of "the STL," and different people mean different things when they use the term. In this book, "the STL" means the parts of C++'s Standard Library that work with iterators. That includes the standard containers (including string), parts of the iostream library, function objects, and algorithms. It excludes the standard container adapters (stack, queue, and priority_queue) as well as the containers bitset and valarray, because they lack iterator support. It doesn't include arrays, either. True, arrays support iterators in the form of pointers, but arrays are part of the C++ *language*, not the library.

Technically, my definition of the STL excludes extensions of the standard C++ library, notably hashed containers, singly linked lists, ropes, and a variety of nonstandard function objects. Even so, an effective STL programmer needs to be aware of such extensions, so I mention them where it's appropriate. Indeed, Item 25 is devoted to an overview of nonstandard hashed containers. They're not in the STL now, but something similar to them is almost certain to make it into the next version of the standard C++ library, and there's value in glimpsing the future.

One of the reasons for the existence of STL extensions is that the STL is a library designed to be extended. In this book, however, I focus on *using* the STL, not on adding new components to it. You'll find, for example, that I have little to say about writing your own algorithms, and I offer no guidance at all on writing new containers and iterators. I believe that it's important to master what the STL already provides before you embark on increasing its capabilities, so that's what I focus on in *Effective STL*. When you decide to create your own STLesque components, you'll find advice on how to do it in books like Josuttis' *The C++ Standard Library* [3] and Austern's *Generic Programming and the STL* [4]. One aspect of STL extension I *do* discuss in this book is writing your own function objects. You can't use the STL effectively without knowing how to do that, so I've devoted an entire chapter to the topic (Chapter 6).

Citations

The references to the books by Josuttis and Austern in the preceding paragraph demonstrate how I handle bibliographic citations. In general, I try to mention enough of a cited work to identify it for people who are already familiar with it. If you already know about these authors' books, for example, you don't have to turn to the Bibliography to find out that [3] and [4] refer to books you already know. If you're

试读结束：需要全本请在线购买： www.ertongbook.com