Effective STL
For Woofieland.
This page intentionally left blank
Preface xi

Acknowledgments xv

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
This page intentionally left blank
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
This page intentionally left blank
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_ptr 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 dclm 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 clc 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.
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 Pattini, 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, Shlomoi Frank, Andrea Griffini, Hans Eckardt, David Smallberg, Matt Page, Andy Fyne, Vincent Stojanov, Randy Parker, Thomas Schell, Cameron Mac Minn, Mark Davis, Giora Unger, Julie Nahil, Martin Rottinger, Neil Henderson, Andrew Savige, and Molly Sharp. 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.
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 STL-eseque 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
Effective STL

Introduction

3

not familiar with a publication, of course, the Bibliography (which begins on page 225) gives you a full citation.

I cite three works often enough that I generally leave off the citation number. The first of these is the International Standard for C++ [5], which I usually refer to as simply "the Standard." The other two are my earlier books on C++, Effective C++ [1] and More Effective C++ [2].

The STL and Standards

I refer to the C++ Standard frequently, because Effective STL focuses on portable, standard-conformant C++. In theory, everything I show in this book will work with every C++ implementation. In practice, that isn’t true. Shortcomings in compiler and STL implementations conspire to prevent some valid code from compiling or from behaving the way it’s supposed to. Where that is commonly the case, I describe the problems, and I explain how you can work around them.

Sometimes, the easiest workaround is to use a different STL implementation. Appendix B gives an example of when this is the case. In fact, the more you work with the STL, the more important it becomes to distinguish between your compilers and your library implementations. When programmers run into difficulties trying to get legitimate code to compile, it’s customary for them to blame their compilers, but with the STL, compilers can be fine, while STL implementations are faulty. To emphasize the fact that you are dependent on both your compilers and your library implementations, I refer to your STL platforms. An STL platform is the combination of a particular compiler and a particular STL implementation. In this book, if I mention a compiler problem, you can be sure that I mean it’s the compiler that’s the culprit. However, if I refer to a problem with your STL platform, you should interpret that as “maybe a compiler bug, maybe a library bug, possibly both.”

I generally refer to your “compilers” — plural. That’s an outgrowth of my longstanding belief that you improve the quality (especially the portability) of your code if you ensure that it works with more than one compiler. Furthermore, using multiple compilers generally makes it easier to unravel the Gordian nature of error messages arising from improper use of the STL. (Item 49 is devoted to approaches to decoding such messages.)

Another aspect of my emphasis on standard-conforming code is my concern that you avoid constructs with undefined behavior. Such constructs may do anything at runtime. Unfortunately, this means they may do precisely what you want them to, and that can lead to a false
sense of security. Too many programmers assume that undefined behavior always leads to an obvious problem, e.g., a segmentation fault or other catastrophic failure. The results of undefined behavior can actually be much more subtle, e.g., corruption of rarely-referenced data. They can also vary across program runs. I find that a good working definition of undefined behavior is "works for me, works for you, works during development and QA, but blows up in your most important customer's face." It's important to avoid undefined behavior, so I point out common situations where it can arise. You should train yourself to be alert for such situations.

**Reference Counting**

It's close to impossible to discuss the STL without mentioning reference counting. As you'll see in Items 7 and 33, designs based on containers of pointers almost invariably lead to reference counting. In addition, many string implementations are internally reference counted, and, as Item 15 explains, this may be an implementation detail you can't afford to ignore. In this book, I assume that you are familiar with the basics of reference counting. If you're not, most intermediate and advanced C++ texts cover the topic. In *More Effective C++*, for example, the relevant material is in Items 28 and 29. If you don't know what reference counting is and you have no inclination to learn, don't worry. You'll get through this book just fine, though there may be a few sentences here and there that won't make as much sense as they otherwise would.

**string and wstring**

Whatever I say about string applies equally well to its wide-character counterpart, wstring. Similarly, any time I refer to the relationship between string and char or char*, the same is true of the relationship between wstring and wchar_t or wchar_t*. In other words, just because I don't explicitly mention wide-character strings in this book, don't assume that the STL fails to support them. It supports them as well as char-based strings. It has to. Both string and wstring are instantiations of the same template, basic_string.

**Terms, Terms, Terms**

This is not an introductory book on the STL, so I assume you know the fundamentals. Still, the following terms are sufficiently important that I feel compelled to review them:
• Vector, string, deque, and list are known as the standard sequence containers. The standard associative containers are set, multiset, map, and multimap.

• Iterators are divided into five categories, based on the operations they support. Very briefly, input iterators are read-only iterators where each iterated location may be read only once. Output iterators are write-only iterators where each iterated location may be written only once. Input and output iterators are modeled on reading and writing input and output streams (e.g., files). It’s thus unsurprising that the most common manifestations of input and output iterators are istream_iterators and ostream_iterators, respectively.

Forward iterators have the capabilities of both input and output iterators, but they can read or write a single location repeatedly. They don’t support operator--, so they can move only forward with any degree of efficiency. All standard STL containers support iterators that are more powerful than forward iterators, but, as you’ll see in Item 25, one design for hashed containers yields forward iterators. Containers for singly linked lists (considered in Item 50) also offer forward iterators.

Bidirectional iterators are just like forward iterators, except they can go backward as easily as they go forward. The standard associative containers all offer bidirectional iterators. So does list.

Random access iterators do everything bidirectional iterators do, but they also offer “iterator arithmetic,” i.e., the ability to jump forward or backward in a single step. Vector, string, and deque each provide random access iterators. Pointers into arrays act as random access iterators for the arrays.

• Any class that overloads the function call operator (i.e., operator()) is a functor class. Objects created from such classes are known as function objects or functors. Most places in the STL that work with function objects work equally well with real functions, so I often use the term “function objects” to mean both C++ functions as well as true function objects.

• The functions bind1st and bind2nd are known as binders.

A revolutionary aspect of the STL is its complexity guarantees. These guarantees bound the amount of work any STL operation is allowed to perform. This is wonderful, because it can help you determine the relative efficiency of different approaches to the same problem, regardless of the STL platform you’re using. Unfortunately, the terminology
behind the complexity guarantees can be confusing if you haven’t been formally introduced to the jargon of computer science. Here’s a quick primer on the complexity terms I use in this book. Each refers to how long it takes to do something as a function of \( n \), the number of elements in a container or range.

- An operation that runs in constant time has performance that is unaffected by changes in \( n \). For example, inserting an element into a list is a constant-time operation. Regardless of whether the list has one element or one million, the insertion takes about the same amount of time.

  Don’t take the term “constant time” too literally. It doesn’t mean that the amount of time it takes to do something is literally constant, it just means that it’s unaffected by \( n \). For example, two STL platforms might take dramatically different amounts of time to perform the same “constant-time” operation. This could happen if one library has a much more sophisticated implementation than another or if one compiler performs substantially more aggressive optimization.

  A variant of constant time complexity is amortized constant time. Operations that run in amortized constant time are usually constant-time operations, but occasionally they take time that depends on \( n \). Amortized constant time operations typically run in constant time.

- An operation that runs in logarithmic time needs more time to run as \( n \) gets larger, but the time it requires grows at a rate proportional to the logarithm of \( n \). For example, an operation on a million items would be expected to take only about three times as long as on a hundred items, because \( \log n^3 = 3 \log n \). Most search operations on associative containers (e.g., `set::find`) are logarithmic-time operations.

- The time needed to perform an operation that runs in linear time increases at a rate proportional to increases in \( n \). The standard algorithm `count` runs in linear time, because it has to look at every element of the range it’s given. If the range triples in size, it has to do three times as much work, and we’d expect it to take about three times as long to do it.

As a general rule, a constant-time operation runs faster than one requiring logarithmic time, and a logarithmic-time operation runs faster than one whose performance is linear. This is always true when \( n \) gets big enough, but for relatively small values of \( n \), it’s sometimes possible for an operation with a worse theoretical complexity to outperform an operation with a better theoretical complexity. If you’d like to know more about STL complexity guarantees, turn to Josuttis’ *The C++ Standard Library* [3].
As a final note on terminology, recall that each element in a map or multimap has two components. I generally call the first component the *key* and the second component the *value*. Given

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

for example, the string is the key and the double is the value.

**Code Examples**

This book is filled with example code, and I explain each example when I introduce it. Still, it’s worth knowing a few things in advance.

You can see from the map example above that I routinely omit `#include` statements and ignore the fact that STL components are in namespace `std`. When defining the map `m`, I could have written this,

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

but I prefer to save us both the noise.

When I declare a formal type parameter for a template, I use `typename` instead of `class`. That is, instead of writing this,

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

I write this:

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

In this context, `class` and `typename` mean exactly the same thing, but I find that `typename` more clearly expresses what I usually want to say: that *any* type will do; I need not be a class. If you prefer to use `class` to declare type parameters, go right ahead. Whether to use `typename` or `class` in this context is purely a matter of style.

It is not a matter of style in a different context. To avoid potential parsing ambiguities (the details of which I’ll spare you), you are required to use `typename` to precede type names that are dependent on formal type parameters. Such types are known as *dependent types*, and an example will help clarify what I’m talking about. Suppose you’d like to write a template for a function that, given an STL container, returns whether the last element in the container is greater than the first element. Here’s one way to do it:
template<typename C>
bool lastGreaterThanFirst(const C& container)
{
    if (container.empty()) return false;
    typename C::const_iterator begin(container.begin());
    typename C::const_iterator end(container.end());
    return *--end > *begin;
}

In this example, the local variables begin and end are of type
C::const_iterator. const_iterator is a type that is dependent on the formal
type parameter C. Because C::const_iterator is a dependent type, you
are required to precede it with the word typename. (Some compilers in-
correctly accept the code without the typenames, but such code isn’t
portable.)

I hope you’ve noticed my use of color in the examples above. It’s there
to focus your attention on parts of the code that are particularly im-
portant. Often, I highlight the differences between related examples,
such as when I showed the two possible ways to declare the parameter T in the Widget example. This use of color to call out especially
noteworthy parts of examples carries over to diagrams, too. For in-
stance, this diagram from Item 5 uses color to identify the two point-
ers that are affected when a new element is inserted into a list:

I also use color for chapter numbers, but such use is purely gratu-
itous. This being my first two-color book, I hope you’ll forgive me a lit-
tle chromatic exuberance.

Two of my favorite parameter names are lhs and rhs. They stand for
“left-hand side” and “right-hand side,” respectively, and I find them
especially useful when declaring operators. Here’s an example from
Item 19:

class Widget { ...);
bool operator==(const Widget& lhs, const Widget& rhs);
When this function is called in a context like this,

```cpp
if (x == y) ... // assume x and y are Widgets
```
x, which is on the left-hand side of the "==", is known as lhs inside op-
erator==, and y is known as rhs.

As for the class name Widget, that has nothing to do with GUIs or tool-
kits. It’s just the name I use for “some class that does something.”
Sometimes, as on page 7, Widget is a class template instead of a class.
In such cases, you may find that I still refer to Widget as a class, even
though it’s really a template. Such sloppiness about the difference be-
tween classes and class templates, structs and struct templates, and
functions and function templates hurts no one as long as there is no
ambiguity about what is being discussed. In cases where it could be
confusing, I do distinguish between templates and the classes,
structs, and functions they generate.

**Efficiency Items**

I considered including a chapter on efficiency in *Effective STL*, but I
ultimately decided that the current organization was preferable. Still,
a number of Items focus on minimizing space and runtime demands.
For your performance-enhancing convenience, here is the table of
contents for the virtual chapter on efficiency:

- **Item 4**: Call empty instead of checking size() against zero. 23
- **Item 5**: Prefer range member functions to their single-element
counterparts. 24
- **Item 14**: Use reserve to avoid unnecessary reallocations. 66
- **Item 15**: Be aware of variations in string implementations. 68
- **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
- **Item 29**: Consider istreambuf_iterators for character-by-character
input. 126
- **Item 31**: Know your sorting options. 133
- **Item 44**: Prefer member functions to algorithms with the
same names. 190
- **Item 46**: Consider function objects instead of functions as
algorithm parameters. 201
The Guidelines in *Effective STL*

The guidelines that make up the 50 Items in this book are based on the insights and advice of the world’s most experienced STL programmers. These guidelines summarize things you should almost always do — or almost always avoid doing — to get the most out of the Standard Template Library. At the same time, they’re just guidelines. Under some conditions, it makes sense to violate them. For example, the title of Item 7 tells you to invoke delete on newed pointers in a container before the container is destroyed, but the text of that Item makes clear that this applies only when the objects pointed to by those pointers should go away when the container does. This is often the case, but it’s not universally true. Similarly, the title of Item 35 beseeches you to use STL algorithms to perform simple case-insensitive string comparisons, but the text of the Item points out that in some cases, you’ll be better off using a function that’s not only outside the STL, it’s not even part of standard C++!

Only you know enough about the software you’re writing, the environment in which it will run, and the context in which it’s being created to determine whether it’s reasonable to violate the guidelines I present. Most of the time, it won’t be, and the discussions that accompany each Item explain why. In a few cases, it will. Slavish devotion to the guidelines isn’t appropriate, but neither is cavalier disregard. Before venturing off on your own, you should make sure you have a good reason.
Sure, the STL has iterators, algorithms, and function objects, but for most C++ programmers, it’s the containers that stand out. More powerful and flexible than arrays, they grow (and often shrink) dynamically, manage their own memory, keep track of how many objects they hold, bound the algorithmic complexity of the operations they support, and much, much more. Their popularity is easy to understand. They’re simply better than their competition, regardless of whether that competition comes from containers in other libraries or is a container type you’d write yourself. STL containers aren’t just good. They’re really good.

This chapter is devoted to guidelines applicable to all the STL containers. Later chapters focus on specific container types. The topics addressed here include selecting the appropriate container given the constraints you face; avoiding the delusion that code written for one container type is likely to work with other container types; the significance of copying operations for objects in containers; difficulties that arise when pointers or auto_ptrs are stored in containers; the ins and outs of erasing; what you can and cannot accomplish with custom allocators; tips on how to maximize efficiency; and considerations for using containers in a threaded environment.

That’s a lot of ground to cover, but don’t worry. The Items break it down into bite-sized chunks, and along the way, you’re almost sure to pick up several ideas you can apply to your code now.

Item 1: Choose your containers with care.

You know that C++ puts a variety of containers at your disposal, but do you realize just how varied that variety is? To make sure you haven’t overlooked any of your options, here’s a quick review.

- The **standard STL sequence containers**. vector, string, deque, and list.
item 1 containers

- The **standard STL associative containers**, set, multiset, map, and multimap.

- The **nonstandard sequence containers** slist and rope. slist is a singly linked list, and rope is essentially a heavy-duty string. (A “rope” is a heavy-duty “string.” Get it?) You’ll find a brief overview of these nonstandard (but commonly available) containers in Item 50.

- The **nonstandard associative containers** hash_set, hash_multiset, hash_map, and hash_multimap. I examine these widely available hash-table-based variants on the standard associative containers in Item 25.

- **vector<char> as a replacement for string.** Item 13 describes the conditions under which such a replacement might make sense.

- **vector as a replacement for the standard associative containers.** As Item 23 makes clear, there are times when vector can outperform the standard associative containers in both time and space.

- Several **standard non-STL containers**, including arrays, bitset, valarray, stack, queue, and priority_queue. Because these are non-STL containers, I have little to say about them in this book, though Item 16 mentions a case where arrays are preferable to STL containers and Item 18 explains why bitset may be better than vector<bool>. It’s also worth bearing in mind that arrays can be used with STL algorithms, because pointers can be used as array iterators.

That’s a panoply of options, and it’s matched in richness by the range of considerations that should go into choosing among them. Unfortunately, most discussions of the STL take a fairly narrow view of the world of containers, ignoring many issues relevant to selecting the one that is most appropriate. Even the Standard gets into this act, offering the following guidance for choosing among vector, deque, and list:

vector, list, and deque offer the programmer different complexity trade-offs and should be used accordingly. vector is the type of sequence that should be used by default. list should be used when there are frequent insertions and deletions from the middle of the sequence. deque is the data structure of choice when most insertions and deletions take place at the beginning or at the end of the sequence.

If your primary concern is algorithmic complexity, I suppose this constitutes reasonable advice, but there is so much more to be concerned with.
In a moment, we'll examine some of the important container-related issues that complement algorithmic complexity, but first I need to introduce a way of categorizing the STL containers that isn't discussed as often as it should be. That is the distinction between contiguous-memory containers and node-based containers.

**Contiguous-memory containers** (also known as *array-based containers*) store their elements in one or more (dynamically allocated) chunks of memory, each chunk holding more than one container element. If a new element is inserted or an existing element is erased, other elements in the same memory chunk have to be shifted up or down to make room for the new element or to fill the space formerly occupied by the erased element. This kind of movement affects both performance (see Items 5 and 14) and exception safety (as we'll soon see). The standard contiguous-memory containers are vector, string, and deque. The nonstandard rope is also a contiguous-memory container.

**Node-based containers** store only a single element per chunk of (dynamically allocated) memory. Insertion or erasure of a container element affects only pointers to nodes, not the contents of the nodes themselves, so element values need not be moved when something is inserted or erased. Containers representing linked lists, such as list and slist, are node-based, as are all the standard associative containers. (They're typically implemented as balanced trees.) The nonstandard hashed containers use varying node-based implementations, as you'll see in Item 25.

With this terminology out of the way, we're ready to sketch some of the questions most relevant when choosing among containers. In this discussion, I omit consideration of non-STL-like containers (e.g., arrays, bitsets, etc.), because this is, after all, a book on the STL.

- **Do you need to be able to insert a new element at an arbitrary position in the container?** If so, you need a sequence container; associative containers won't do.

- **Do you care how elements are ordered in the container?** If not, a hashed container becomes a viable choice. Otherwise, you'll want to avoid hashed containers.

- **Must the container be part of standard C++?** If so, that eliminates hashed containers, slist, and rope.

- **What category of iterators do you require?** If they must be random access iterators, you're technically limited to vector, deque, and string, but you'd probably want to consider rope, too. (See Item 50
for information on rope.) If bidirectional iterators are required, you must avoid slist (see Item 50) as well as one common implementation of the hashed containers (see Item 25).

- Is it important to avoid movement of existing container elements when insertions or erasures take place? If so, you’ll need to stay away from contiguous-memory containers (see Item 5).

- Does the data in the container need to be layout-compatible with C? If so, you’re limited to vectors (see Item 16).

- Is lookup speed a critical consideration? If so, you’ll want to look at hashed containers (see Item 25), sorted vectors (see Item 23), and the standard associative containers — probably in that order.

- Do you mind if the underlying container uses reference counting? If so, you’ll want to steer clear of string, because many string implementations are reference-counted (see Item 13). You’ll need to avoid rope, too, because the definitive rope implementation is based on reference counting (see Item 50). You have to represent your strings somehow, of course, so you’ll want to consider vector<char>.

- Do you need transactional semantics for insertions and erasures? That is, do you require the ability to reliably roll back insertions and erasures? If so, you’ll want to use a node-based container. If you need transactional semantics for multiple-element insertions (e.g., the range form — see Item 5), you’ll want to choose list, because list is the only standard container that offers transactional semantics for multiple-element insertions. Transactional semantics are particularly important for programmers interested in writing exception-safe code. (Transactional semantics can be achieved with contiguous-memory containers, too, but there is a performance cost, and the code is not as straightforward. To learn more about this, consult Item 17 of Sutter’s Exceptional C++ [8].)

- Do you need to minimize iterator, pointer, and reference invalidation? If so, you’ll want to use node-based containers, because insertions and erasures on such containers never invalidate iterators, pointers, or references (unless they point to an element you are erasing). In general, insertions or erasures on contiguous-memory containers may invalidate all iterators, pointers, and references into the container.

- Do you care if using swap on containers invalidates iterators, pointers, or references? If so, you’ll need to avoid string, because string is alone in the STL in invalidating iterators, pointers, and references during swaps.
Would it be helpful to have a sequence container with random access iterators where pointers and references to the data are not invalidated as long as nothing is erased and insertions take place only at the ends of the container? This is a very special case, but if it’s your case, deque is the container of your dreams. (Interestingly, deque’s iterators may be invalidated when insertions are made only at the ends of the container. deque is the only standard STL container whose iterators may be invalidated without also invalidating its pointers and references.)

These questions are hardly the end of the matter. For example, they don’t take into account the varying memory allocation strategies employed by the different container types. (Items 10 and 14 discuss some aspects of such strategies.) Still, they should be enough to convince you that, unless you have no interest in element ordering, standards conformance, iterator capabilities, layout compatibility with C, lookup speed, behavioral anomalies due to reference counting, the ease of implementing transactional semantics, or the conditions under which iterators are invalidated, you have more to think about than simply the algorithmic complexity of container operations. Such complexity is important, of course, but it’s far from the entire story.

The STL gives you lots of options when it comes to containers. If you look beyond the bounds of the STL, there are even more options. Before choosing a container, be sure to consider all your options. A “default container”? I don’t think so.

Item 2: Beware the illusion of container-independent code.

The STL is based on generalization. Arrays are generalized into containers and parameterized on the types of objects they contain. Functions are generalized into algorithms and parameterized on the types of iterators they use. Pointers are generalized into iterators and parameterized on the type of objects they point to.

That’s just the beginning. Individual container types are generalized into sequence and associative containers, and similar containers are given similar functionality. Standard contiguous-memory containers (see Item 1) offer random-access iterators, while standard node-based containers (again, see Item 1) provide bidirectional iterators. Sequence containers support push_front and/or push_back, while associative containers don’t. Associative containers offer logarithmic-time lower_bound, upper_bound, and equal_range member functions, but sequence containers don’t.
With all this generalization going on, it’s natural to want to join the movement. This sentiment is laudable, and when you write your own containers, iterators, and algorithms, you’ll certainly want to pursue it. Alas, many programmers try to pursue it in a different manner. Instead of committing to particular types of containers in their software, they try to generalize the notion of a container so that they can use, say, a vector, but still preserve the option of replacing it with something like a deque or a list later — all without changing the code that uses it. That is, they strive to write container-independent code. This kind of generalization, well-intentioned though it is, is almost always misguided.

Even the most ardent advocate of container-independent code soon realizes that it makes little sense to try to write software that will work with both sequence and associative containers. Many member functions exist for only one category of container, e.g., only sequence containers support \texttt{push_front} or \texttt{push_back}, and only associative containers support \texttt{count} and \texttt{lower_bound}, etc. Even such basics as \texttt{insert} and \texttt{erase} have signatures and semantics that vary from category to category. For example, when you insert an object into a sequence container, it stays where you put it, but if you insert an object into an associative container, the container moves the object to where it belongs in the container’s sort order. For another example, the form of \texttt{erase} taking an iterator returns a new iterator when invoked on a sequence container, but it returns nothing when invoked on an associative container. (Item 9 gives an example of how this can affect the code you write.)

Suppose, then, you aspire to write code that can be used with the most common sequence containers: vector, deque, and list. Clearly, you must program to the intersection of their capabilities, and that means no uses of \texttt{reserve} or \texttt{capacity} (see Item 14), because deque and list don’t offer them. The presence of list also means you give up \texttt{operator[]}, and you limit yourself to the capabilities of bidirectional iterators. That, in turn, means you must stay away from algorithms that demand random access iterators, including \texttt{sort}, \texttt{stable_sort}, \texttt{partial_sort}, and \texttt{nth_element} (see Item 31).

On the other hand, your desire to support vector rules out use of \texttt{push_front} and \texttt{pop_front}, and both vector and deque put the kibosh on \texttt{splice} and the member form of \texttt{sort}. In conjunction with the constraints above, this latter prohibition means that there is no form of \texttt{sort} you can call on your “generalized sequence container.”
That’s the obvious stuff. If you violate any of those restrictions, your code will fail to compile with at least one of the containers you want to be able to use. The code that will compile is more insidious.

The main culprit is the different rules for invalidation of iterators, pointers, and references that apply to different sequence containers. To write code that will work correctly with vector, deque, and list, you must assume that any operation invalidating iterators, pointers, or references in any of those containers invalidates them in the container you’re using. Thus, you must assume that every call to insert invalidates everything, because deque::insert invalidates all iterators and, lacking the ability to call capacity, vector::insert must be assumed to invalidate all pointers and references. (Item 1 explains that deque is unique in sometimes invalidating its iterators without invalidating its pointers and references.) Similar reasoning leads to the conclusion that, unless you’re erasing the last element of a container, calls to erase must also be assumed to invalidate everything.

Want more? You can’t pass the data in the container to a C interface, because only vector supports that (see Item 16). You can’t instantiate your container with bool as the type of objects to be stored, because, as Item 18 explains, vector<bool> doesn’t always behave like a vector, and it never actually stores bools. You can’t assume list’s constant-time insertions and erasures, because vector and deque take linear time to perform those operations.

When all is said and done, you’re left with a “generalized sequence container” where you can’t call reserve, capacity, operator[], push_front, pop_front, splice, or any algorithm requiring random access iterators; a container where every call to insert and erase takes linear time and invalidates all iterators, pointers, and references; and a container incompatible with C where bools can’t be stored. Is that really the kind of container you want to use in your applications? I suspect not.

If you rein in your ambition and decide you’re willing to drop support for list, you still give up reserve, capacity, push_front, and pop_front; you still must assume that all calls to insert and erase take linear time and invalidate everything; you still lose layout compatibility with C; and you still can’t store bools.

If you abandon the sequence containers and shoot instead for code that can work with different associative containers, the situation isn’t much better. Writing for both set and map is close to impossible, because sets store single objects while maps store pairs of objects. Even writing for both set and multiset (or map and multimap) is tough. The insert member function taking only a value has different return types for sets/maps than for their multi cousins, and you must reli-
giously avoid making any assumptions about how many copies of a value are stored in a container. With map and multimap, you must avoid using operator[], because that member function exists only for map.

Face the truth: it’s not worth it. The different containers are different, and they have strengths and weaknesses that vary in significant ways. They’re not designed to be interchangeable, and there’s little you can do to paper that over. If you try, you’re merely tempting fate, and fate doesn’t like to be tempted.

Still, the day will dawn when you’ll realize that a container choice you made was, er, suboptimal, and you’ll need to use a different container type. You now know that when you change container types, you’ll not only need to fix whatever problems your compilers diagnose, you’ll also need to examine all the code using the container to see what needs to be changed in light of the new container’s performance characteristics and rules for invalidation of iterators, pointers, and references. If you switch from a vector to something else, you’ll also have to make sure you’re no longer relying on vector’s C-compatible memory layout, and if you switch to a vector, you’ll have to ensure that you’re not using it to store bools.

Given the inevitability of having to change container types from time to time, you can facilitate such changes in the usual manner: by encapsulating, encapsulating, encapsulating. One of the easiest ways to do this is through the liberal use of typedefs for container types. Hence, instead of writing this,

```cpp
class Widget { ... }
vector<Widget> vw;
Widget bestWidget;
...
vector<Widget>::iterator i = find(vw.begin(), vw.end(), bestWidget);
```

write this:

```cpp
class Widget { ... }
typedef vector<Widget> WidgetContainer;
WidgetContainer cw;
Widget bestWidget;
...
WidgetContainer::iterator i = find(cw.begin(), cw.end(), bestWidget);
```
This makes it a lot easier to change container types, something that’s especially convenient if the change in question is simply to add a custom allocator. (Such a change doesn’t affect the rules for iterator/pointer/reference invalidation.)

```cpp
class Widget { ...; };

// see Item 10 for why this is needed to be a template
template<typename T> class SpecialAllocator { ...; }

typedef vector<Widget, SpecialAllocator<Widget>> WidgetContainer;

WidgetContainer cw; // still works

Widget bestWidget;
```

If the encapsulating aspects of typedefs mean nothing to you, you’re still likely to appreciate the work they can save, especially for iterator types. For example, if you have an object of type

```cpp
map<string, vector<Widget>::iterator, CIStringCompare> // CIStringCompare is “case-insensitive string compare;”
```

and you want to walk through the map using const_iterators, do you really want to spell out

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

more than once? Once you’ve used the STL a little while, you’ll realize that typedefs are your friends.

A typedef is just a synonym for some other type, so the encapsulation it affords is purely lexical. A typedef doesn’t prevent a client from doing (or depending on) anything they couldn’t already do (or depend on). You need bigger ammunition if you want to limit client exposure to the container choices you’ve made. You need classes.

To limit the code that may require modification if you replace one container type with another, hide the container in a class, and limit the amount of container-specific information visible through the class interface. For example, if you need to create a customer list, don’t use a list directly. Instead, create a CustomerList class, and hide a list in its private section:
class CustomerList {
private:
    typedef list<Customer> CustomerContainer;
    typedef CustomerContainer::iterator CCIterator;
    CustomerContainer customers;

public: // limit the amount of list-specific
    ... // information visible through
    }; // this interface

At first, this may seem silly. After all a customer list is a *list*, right? Well, maybe. Later you may discover that you don’t need to insert or erase customers from the middle of the list as often as you’d anticipated, but you do need to quickly identify the top 20% of your customers — a task tailor-made for the *nth_element* algorithm (see Item 31). But *nth_element* requires random access iterators. It won’t work with a list. In that case, your customer “list” might be better implemented as a *vector* or a *deque*.

When you consider this kind of change, you still have to check every *CustomerList* member function and every friend to see how they’ll be affected (in terms of performance and iterator/pointer/reference invalidation, etc.), but if you’ve done a good job of encapsulating *CustomerList*’s implementation details, the impact on *CustomerList* clients should be small. You can’t write container-independent code, but they might be able to.

**Item 3:** Make copying cheap and correct for objects in containers.

Containers hold objects, but not the ones you give them. Instead, when you add an object to a container (via, e.g., insert or push_back, etc.), what goes into the container is a *copy* of the object you specify.

Once an object is in a container, it’s not uncommon for it to be copied further. If you insert something into or erase something from a *vector*, *string*, or *deque*, existing container elements are typically moved (copied) around (see Items 5 and 14). If you use any of the sorting algorithms (see Item 31): *next_permutation* or *previous_permutation*; remove, unique, or their ilk (see Item 32): rotate or reverse, etc., objects will be moved (copied) around. Yes, copying objects is the STL way.
It may interest you to know how all this copying is accomplished. That’s easy. An object is copied by using its copying member functions, in particular, its copy constructor and its copy assignment operator. (Clever names, no?) For a user-defined class like Widget, these functions are traditionally declared like this:

```cpp
class Widget {
public:
    ... Widget(const Widget&); // copy constructor
    Widget& operator=(const Widget&); // copy assignment operator
    ...}
```

As always, if you don’t declare these functions yourself, your compilers will declare them for you. Also as always, the copying of built-in types (e.g., ints, pointers, etc.) is accomplished by simply copying the underlying bits. (For details on copy constructors and assignment operators, consult any introductory book on C++. In *Effective C++*, Items 11 and 27 focus on the behavior of these functions.)

With all this copying taking place, the motivation for this Item should now be clear. If you fill a container with objects where copying is expensive, the simple act of putting the objects into the container could prove to be a performance bottleneck. The more things get moved around in the container, the more memory and cycles you’ll blow on making copies. Furthermore, if you have objects where “copying” has an unconventional meaning, putting such objects into a container will invariably lead to grief. (For an example of the kind of grief it can lead to, see Item 8.)

In the presence of inheritance, of course, copying leads to slicing. That is, if you create a container of base class objects and you try to insert derived class objects into it, the derivedness of the objects will be removed as the objects are copied (via the base class copy constructor) into the container:

```cpp
vector<Widget> vw;
class SpecialWidget: // SpecialWidget inherits from public Widget { ... }; // Widget above SpecialWidget sw;
vw.push_back(sw); // sw is copied as a base class // object into vw. Its specialness // is lost during the copying
```

The slicing problem suggests that inserting a derived class object into a container of base class objects is almost always an error. If you want
the resulting object to act like a derived class object, e.g., invoke derived class virtual functions, etc., it is always an error. (For more background on the slicing problem, consult *Effective C++*, Item 22. For another example of where it arises in the STL, see Item 38.)

An easy way to make copying efficient, correct, and immune to the slicing problem is to create containers of *pointers* instead of containers of objects. That is, instead of creating a container of *Widget*, create a container of *Widget*\*\*. Copying pointers is fast, it always does exactly what you expect (it copies the bits making up the pointer), and nothing gets sliced when a pointer is copied. Unfortunately, containers of pointers have their own STL-related headaches. You can read about them in Items 7 and 33. As you seek to avoid those headaches while still dodging efficiency, correctness, and slicing concerns, you’ll probably discover that containers of *smart pointers* are an attractive option. To learn more about this option, turn to Item 7.

If all this makes it sound like the STL is copy-crazy, think again. Yes, the STL makes lots of copies, but it’s generally designed to avoid copying objects *unnecessarily*. In fact, it’s generally designed to avoid *creating* objects unnecessarily. Contrast this with the behavior of C’s and C++’s only built-in container, the lowly array:

```
Widget w[maxNumWidgets]; // create an array of maxNumWidgets
                         // Widgets, default-constructing each one
```

This constructs *maxNumWidgets* *Widget* objects, even if we normally expect to use only a few of them or we expect to immediately overwrite each default-constructed value with values we get from someplace else (e.g., a file). Using the STL instead of an array, we can use a vector that grows when it needs to:

```
vector<Widget> vw;   // create a vector with zero Widget
                     // objects that will expand as needed
```

We can also create an empty vector that contains enough space for *maxNumWidgets* *Widgets*, but where zero *Widgets* have been constructed:

```
vector<Widget> vw;
vw.reserve(maxNumWidgets);   // see Item 14 for details on reserve
```

Compared to arrays, STL containers are much more civilized. They create (by copying) only as many objects as you ask for, they do it only when you direct them to, and they use a default constructor only when you say they should. Yes, STL containers make copies, and yes, you need to understand that, but don’t lose sight of the fact that they’re still a big step up from arrays.
Item 4: Call empty instead of checking size() against zero.

For any container c, writing

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

is essentially equivalent to writing

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

That being the case, you might wonder why one construct should be preferred to the other, especially in view of the fact that empty is typically implemented as an inline function that simply returns whether size returns 0.

You should prefer the construct using empty, and the reason is simple: empty is a constant-time operation for all standard containers, but for some list implementations, size may take linear time.

But what makes list so troublesome? Why can’t it, too, offer a constant-time size? The answer has much to do with the range form of list’s unique splicing functions. Consider this code:

```cpp
list<int> list1;
list<int> list2;
...
list1.splice( // move all nodes in list2
    list1.end(), list2, // from the first occurrence
    find(list2.begin(), list2.end(), 5), // of 5 through the last
    find(list2.rbegin(), list2.rend(), 10).base() // occurrence of 10 to the
); // end of list1. See Item 28
```

This code won’t work unless list2 contains a 10 somewhere beyond a 5, but let’s assume that’s not a problem. Instead, let’s focus on this question: how many elements are in list1 after the splice? Clearly, list1 after the splice has as many elements as it did before the splice plus however many elements were spliced into it. But how many elements were spliced into it? As many as were in the range defined by find(list2.begin(), list2.end(), 5) and find(list2.rbegin(), list2.rend(), 10).base().

Okay, how many is that? Without traversing the range and counting them, there’s no way to know. And therein lies the problem.

Suppose you’re responsible for implementing list. list isn’t just any container, it’s a standard container, so you know your class will be widely used. You naturally want your implementation to be as efficient as possible. You figure that clients will commonly want to find out how many elements are in a list, so you’d like to make size a constant-
time operation. You’d thus like to design list so it always knows how many elements it contains.

At the same time, you know that of all the standard containers, only list offers the ability to splice elements from one place to another without copying any data. You reason that many list clients will choose list specifically because it offers high-efficiency splicing. They know that splicing a range from one list to another can be accomplished in constant time, and you know that they know it, so you certainly want to meet their expectation that splice is a constant-time member function.

This puts you in a quandary. If size is to be a constant-time operation, each list member function must update the sizes of the lists on which it operates. That includes splice. But the only way for the range version of splice to update the sizes of the lists it modifies is for it to count the number of elements being spliced, and doing that would prevent it from achieving the constant-time performance you want for it. If you eliminate the requirement that the range form of splice update the sizes of the lists it’s modifying, splice can be made constant-time, but then size becomes a linear-time operation. In general, it will have to traverse its entire data structure to see how many elements it contains. No matter how you look at it, something — size or the range form of splice — has to give. One or the other can be a constant-time operation, but not both.

Different list implementations resolve this conflict in different ways, depending on whether their authors choose to maximize the efficiency of size or the range form of splice. If you happen to be using a list implementation where a constant-time range form of splice was given higher priority than a constant-time size, you’ll be better off calling empty than size, because empty is always a constant-time operation. Even if you’re not using such an implementation, you might find yourself using such an implementation in the future. For example, you might port your code to a different platform where a different implementation of the STL is available, or you might just decide to switch to a different STL implementation for your current platform.

No matter what happens, you can’t go wrong if you call empty instead of checking to see if size() == 0. So call empty whenever you need to know whether a container has zero elements.

**Item 5:** Prefer range member functions to their single-element counterparts.

Quick! Given two vectors, v1 and v2, what’s the easiest way to make v1’s contents be the same as the second half of v2’s? Don’t agonize
The example classes and class templates declared or defined in this book are indexed under example classes/templates. The example functions and function templates are indexed under example functions/templates.

Before A
__default_alloc_template 211

A
Abrahams, David xvii
abstraction bonus 203
abstraction penalty 201
accumulate
  function objects for 158
  initial value and 157, 159
  side effects and 160
adaptability
  algorithm function objects and 156
  definition of 170
  functor classes and 169–173
  overloading operator() and 173
add or update functionality, in map 107
adjacent_difference 157
Adobe, demonic software spewed by xvii
advance
  efficiency of 122
  to create iterators from
    const_iterators 120–123
Alexandrescu, Andrei xvii, 227, 230
algorithms
  accumulate 156–161
  function objects for 158
  initial value and 157, 159
  side effects and 160
  adaptable function objects and 156
  adjacent_difference 157
  as a vocabulary 186
binary_search 192–201
  container mem funcs vs. 190–192
  copy, eliminating calls to 26
  copy_if, implementing 154–156
  copying func objects within 166–168
  count 156, 192–201
    equal_range vs., for sorted ranges 197
  count_if 157
  efficiency, vs. explicit loops 182–184
  equal_range 192–201
    count vs., for sorted ranges 197
  find 192–201
    count in multiset, multimap vs. 199
    lower_bound in multiset, multimap vs. 201
  using equivalence vs. equality 86
for_each 156–161
  side effects and 160
  function call syntax in 175
  function parameters to 201–205
  hand-written loops vs. 181–189
  includes 148
  inner_product 157
  inplace_merge 148
  lexicographical_compare 153
  longest name 153
  loops and 192
  lower_bound 192–201
  equality vs. equivalence and 195
max_element 157
merge 148, 192
min_element 157
mismatch 151
nth_element 133–138
optimizations in 183
partial_sort 133–138
partial_sum 157
partition 133–138
  containers of pointers and 145
remove 139–143
  on containers of pointers 143–146
remove_copy_if 44
remove_if 44, 144, 145
  see also remove
possible implementation of 167
set_difference 148
set_intersection 148
set_symmetric_difference 148, 153
set_union 148
sort 133–138
sorted ranges and 146–150
sorting 133–138
  alternatives to 138
stable_partition 133–138
stable_sort 133–138
string member functions vs. 230
transform 129
unique 145, 148, 149
  see also remove
unique_copy 148, 149
upper_bound 197–198

All your base are belong to us
allocations
  minimizing via reserve 66–68
  minimum, in string 72
allocator
  allocate interface, vs. operator new 51
allocators
  boilerplate code for 54
  conventions and restrictions 48–54
  fraud in interface 51
  in string 69
  legitimate uses 54–58
  never being called 52
  permitted assumptions about 49
  rebind template within 53
  stateful, portability and 50, 51, 212
  summary of guidelines for 54
  typedefs within 48
  URLs for examples 227, 228
allusions
to Candide 143
to Do Not Go Gentle into that Good Night 187
to Martin Luther King, Jr. 51
to Matthew 25:32 222
to The Little Engine that Could 63
amortized constant time complexity 6
ANSI Standard for C++
  see C++ Standard, The
anteater 88
argument_type 170
array-based containers, definition of 13
arrays
  as part of the STL 2
  as STL containers 64
  containers vs. 22
  vector and string vs. 63–66
assignments
  assign vs. operator= 25
  superfluous, avoiding 31
  via range member functions 33
associative containers
  see standard associative containers, hashed containers
Austern, Matt xvi, xvii, 54, 226, 227, 228
  see also Generic Programming and the STL
author, contacting the xii
auto_ptr
  as container element 40–43
  semantics of copying 41
  sorting 41
  URL for update page on 228
average, finding, for a range 159–161
Avery, Katrina xvii

B
back_insert_iterator 216
back_inserter 130, 216
push_back and 130
backwards order, inserting elements in 184
Barron, Carl xvi
base, see reverse_iterator
basic_ostream, relation to ostream 230
basic_string
  relation to string 210, 229
  relation to string and wstring 4
Becker, Pete xvii
Becker, Thomas xvii
begin/end, relation to rbegin/rend 123
bidirectional iterators
  binary search algorithms and 148
  definition of 5
  standard associative containers and 5
binary_function 170–172
  pointer parameters and 172
  reference parameters and 171
binary_search 199
  bsearch vs. 194
  related functions vs. 192–201
bind1st 216
  adaptability and 170
bind2nd 210, 216
  adaptability and 170
  reference-to-reference problem and 222
binder1st 216
Index

binder2nd 216
binders, definition of 5
bitfields 80
bitset as alternative to vector<
  boost::
bitset> 81
as part of the STL 2
Boost 170, 172, 221–223
shared_array 222
shared_ptr 39, 146, 165, 178, 222
web site URL 217, 227
Bridge Pattern 165
Bruce, Robin xviii
bsearch, vs. binary_search 194
bugs
  Dinkumware list for MSVC 244
  in this book, reporting xii
Bulka, Dov 226
  see also Efficient C++
Buzard, Susannah xviii

C
C++ Programming Language, The 61, 155
bibliography entry for 226
C++ Standard Library, The 2, 6, 94, 113,
  207
bibliographic entry for 225
C++ Standard, The
  bibliography entry for 226
  citation number omitted for 3
  guidance on choosing containers 12
  reference counting and 64
  URL for purchasing 226
Campbell, Aaron xvii
Candide, allusion to 143
capacity
  cost of increasing for vector/string 66
  minimizing in vector and string 77-79
  capacity, vs. size 66-67
Carey, Alicia xviii
case conversions, when not one-for-
one 234
case-insensitive
  string class 229-230
  string comparisons, see string
casts
  const_iterator to iterator 120
  creating temporary objects via 98
  to references 98
  to remove constness 98
  to remove map/multimap constness 99
categories, for iterators 5
char_traits 113, 211, 230
choosing
  among binary_search, lower_bound,
    upper_bound, and
    equal_range 192-201
  among containers 12
  among iterator types 116–119
  vector vs. string 64
citations, in this book 2
class vs. typename 7
classes, vs. structs for functor classes 171
clustering, in node-based containers 103
COAPs, see containers, auto_ptr
collate facet 234
color, use in this book 8
comp.lang.c++.moderated xv
comp.std.c++ xv
comparison functions
  consistency and 149
  equal values and 92-94
  for pairs 104
comparisons
  see also string, comparisons, compar-
sion functions
  iterators with const_iterators 118
  lexicographic 231
compilers
  diagnostics, deciphering 210–217
  implementations, vs. STL impls 3
  independence, value of 3
  optimizations, inlining and 202
  problems, see workarounds
complexity
  amortized constant time 6
  constant time 6
  guarantees, in the STL 5-6
  linear time 6
  logarithmic time 6
compose1 219
compose2 187, 219
const_iterator
  casting to iterator 120
  comparisons with iterators 118
  converting to iterator 120–123
  other iterator types vs. 116-119
const_reverse_iterator
  other iterator types vs. 116-119
constant time complexity 6
  amortized, see amortized constant time
constness
  casting away 98
  of map/multimap elements 95
  of set/multiset elements 95
construction, via range mem funcs 31
contacting the author xii
container adapters, as part of the STL 2
container-independent code 16, 47
illusory nature of 15-20
containers
  arrays as 64
  arrays vs. 22
  assign vs. operator= in 25
  associative, see standard associative
  containers
Index

local class 189
logarithmic time complexity 6
monomorphic function object 164
node-based container 13
output iterator 5
predicate 166
predicate class 166
pure function 166
random access iterator 5
range member function 25
resource acquisition is initialization 61
stability, in sorting 135
standard associative container 5
standard sequence container 5
STL platform 3
transactional semantics 14
delete
delete[] vs. 63
using wrong form 64
deleting
objects more than once 64
pointers in containers 36–40
dependent types, typename and 7–8
deque
unique invalidation rules for 15
see also iterators, invalidation
deque<bool>, as alternative to vector<bool> 81
derereferencing functor class 90
Derge, Gillmer xvii
Design Patterns 80, 165
Design Patterns CD
bibliography entry for 226
destination range, ensuring adequate size 129–133
destructors, calling explicitly 56
Dewhurst, Stephen xvii
dictionary order comparison, see lexicographic comparison
Dinkumware
bug list for MSVC STL 244
hashed containers implementation 114
interface for hashed containers 113
slist implementation 218
STL replacement for MSVC6 243
web site for 243
disambiguating function and object declarations 35
distance
declaration for 122
efficiency of 122
explicit template argument specification and 122
to convert const_iterators to iterators 120–123
Do Not Go Gentle into that Good Night, allusion to 187
documentation, on-line, for the STL 217
Dr. Seuss xi

Effective C++
bibliography entry for 225
citation number omitted for 3
inheritance from class without a virtual destructor discussion in 37
inlining discussion in 202
object copying discussions in 21
pointer to implementation class discussion in 165
slicing problem discussion in 22
URL for errata list for 228
Effective C++ CD
bibliography entry for 225
URL for errata list for 228
Effective STL, web site for xii
efficiency
see also optimizations
advance and 122
algorithms vs. explicit loops 182–184
associative container vs. sorted vector 100-106
case-insensitive string comparisons and 154
comparative, of sorting algorithms 138
copy vs. range insert 26
copying
function objects and 164
objects and 21
distance and 122
empty vs. size 23–24
erasing from contiguous-memory containers 32
function objects vs. functions 201–205
hashed containers, overview of 111–115
hashing and 101
“hint” form of insert and 110
improving via custom allocators 54
increasing vector/string capacity 66
inlining and 202
inserting into contiguous-memory containers 28
istreambuf_iterators and 126–127
list::remove vs. the erase-remove idiom 43
logarithmic vs. linear 190
map::operator[] vs. map::insert 106-111
mem funcs vs. algorithms 190–192
minimizing reallocs via reserve 66-68
ostreambuf_iterators and 127
range vs. single-element member functions 24-33
small string optimization 71
sort vs. qsort 203
sorting algorithms, overview of 133–138
string implementation trade-offs 68–73
toupper and 236–237
use_facet and 234
Effective C++ 202
bibliography entry for 226
Einstein, Albert 69
Emacs 27
e-mail address for comments on this book xii
for the President of the USA 212
embedded nulls 75, 154
empty, vs. size 23–24
encapsulating containers 19
equal_range 196–197
count vs., for sorted ranges 197
related functions vs. 192–201
equal_to 86, 112
equality definition of 84
equivalence vs. 83–88
in hashed containers 113
lower_bound and 195
equivalence definition of 84–85
equality vs. 83–88
in hashed containers 113
lower_bound and 195
equivalent values, inserting in order 198
erase
see also erase-remove idiom
relation to remove algorithm 139–143
return types for 32
return value for standard sequence containers 46
erase_after 218
erase-remove idiom 43, 47, 142, 145, 146, 184, 207
limitations of 46
list::remove vs. 43
remove_if variant 144
erasing
see also containers, erasing
base iterators and 124
elements in containers 43–48
rolling back 14
via range member functions 32
errata list
for Effective C++ 228
for Effective C++ CD 228
for More Effective C++ 228
for this book xii
error messages, deciphering 210–217
example classes/templates
  Average 205
  BadPredicate 167, 168
  BetweenValues 188, 189
  BPFC 164, 165
  BPFCImpl 165
  CIIStringCompare 85
  Contestant 77
  CustomerList 20
  DataCompare 105
  DeleteObject 37, 38
  Dereference 90
  DereferenceLess 91
  DoSomething 163
  Employee 95
  Heap1 57
  Heap2 57
  IDNumberLess 96
  list 52
  list::ListNode 52
  Lock 60
  lt_nocase 231
  lt_str_1 235
  lt_str_1::lt_char 235
  lt_str_2 236
  lt_str_2::lt_char 236
  MaxSpeedCompare 179
  MeetsThreshold 171
  NiftyEmailProgram 212, 215
  Person 198
  PersonNameLess 198
  Point 159, 161
  PointAverage 160, 161
  PtrWidgetNameCompare 172
  RCSP 146
  SharedMemoryAllocator 55
  SpecialAllocator 19, 49
  SpecialContainer 240
  SpecialString 37
  SpecialWidget 21
  SpecificHeapAllocator 57
  std::less<Widget> 178
  StringPtrGreater 93
  StringPtrLess 89
  StringSize 204
  Timestamp 197
  vector 240
  Widget 7, 18, 19, 21, 35, 84, 106, 111, 143, 174, 177, 182, 222
  WidgetNameCompare 171
eexample functions/templates
  anotherBadPredicate 169
  average 204
  Average::operator() 205
  BadPredicate::BadPredicate() 167
  BadPredicate::operator() 167, 168
  BetweenValues::BetweenValues 188
  BetweenValues::operator() 188
  BPFC::operator() 164, 165
  BPFCImpl::BPFCImpl 165
  BPFCImpl::operator() 165
Effective STL

SharedMemoryAllocator::allocate 55
SpecificHeapAllocator::allocate 57
SpecificHeapAllocator::deallocate 57
std::less<Widget>::operator() 178
stringLengthSum 158
StringPtrGreater::operator() 93
stringPtrLess 91
StringPtrLess::operator() 89
StringSize::operator() 204
test 174
transmogrify 129, 220
vector< bool >::operator[] 80
vector< bool >::reference 80
Widget::isCertified 143
Widget::maxSpeed 177
Widget::operator= 21, 106, 111
Widget::readStream 222
Widget::redraw 182
Widget::test 174
Widget::weight 177
Widget::Widget 21, 106
widgetAPCompare 41
TypeNameCompare::operator() 171
writeAverages 204, 205

exception safety 14, 37, 39, 50, 61
Exceptional C++ 14, 165
bibliography entry for 226
exceptions, to guidelines in this book 10
extension template argument specification
distance and 122
for_each and 163
use_facet and 234

extending the STL 2

F

facets, locales and 234–235
find
count in multiset, multimap vs. 199
lower_bound in multiset, multimap
vs. 201
related functions vs. 192–201
using equivalence vs. equality 86
first_argument_type 170
for_each
declaration for 163
explicit template argument specification
and 163
possible implementation of 174
side effects and 160
forward iterators
definition of 5
operator-- and 5
fragmentation, memory, reducing 54
fraud, in allocator interface 51
free STL implementations 217, 220
front_insert_iterator 216
front_inserter 130, 216
push_front and 130
 Fuller, John xviii
function objects
  as workaround for compiler
    problems 204
  definition of 5
  dereferencing, generic 90
  for accumulate 158
  functions vs. 201–205
  monomorphic, definition of 164
  pass-by-value and 162–166
  slicing 164
functional programming 206
functions
  calling forms 173
  calling syntax in the STL 175
  comparison
    equal values and 92–94
    for pointers 88–91
  declaration forms 33–35
  declaring templates in 188
  function objects vs. 201–205
  in <ctype>, conventions of 151
  in <type.h>, conventions of 151
  pointers to, as formal parameters 34
  predicates, need to be pure 166–169
  pure, definition of 166
  range vs. single-element 24–33
functor classes
  adaptability and 169–173
  classes vs. structs 171
  definition of 5
  overloading operator() in 114
  pass-by-value and 162–166
funcor, see function objects
Gamma, Erich 226
  see also Design Patterns
Generic Programming and the STL 2, 94, 217, 229
  bibliography entry for 226
gewürztraminer 232
glass, broken, crawling on 95, 97
Glassborow, Francis xvii
Green, Cliff xvii
growth factor, for vector/string 66
H
hand-written loops
  algorithms vs. 181–189
  iterator invalidation and 185
Hansen, Karin xviii
Harrison, Doug xvi, xvii
hashed containers 111–115
  Dinkumware interface for 113
  equality vs. equivalence in 113
  SGI interface for 112
  two implementation approaches to 114
headers
  #includeing the proper ones 209–210
  <algorithm> 210
  <ctypes> 151
  <type.h> 151
  <functional> 210
  <iterator> 210
  <list> 209
  <map> 209
  <numeric> 210
  <numeric> 157
  <set> 209
  <vector> 209
  failing to #include 217
  summary of 209–210
heaps, separate, allocators and 56–58
Helm, Richard 226
  see also Design Patterns
Henney, Kevlin xvi
“hint” form of insert 100, 110
How the Grinch Stole Christmas! xi
I
identifiers, reserved 213
identity 219
implementations
  compilers vs. the STL 3
  variations for string 68–73
includes 148
#includes, portability and 209–210
inlining 202
  function pointers and 203
inner_product 157
inplace_merge 148
input iterators
  definition of 5
  range insert and 29
insert
  as member template 240
  “hint” form 100, 110
  operator() in map vs. 106–111
  return types for 32
insert iterators
  see also inserter, back_inserter,
    front_inserter
  container::reserve and 131
  copy algorithm and 26
insert_after 218
insert_iterator 216
inserter 130, 216
inserting
see also containers, insertions
base iterators and 124
equivalent values in order 198
in reverse order 184
via range member functions 32
internationalization
see also locales
strcmp and 150
stricmp/strcmpi and 154
invalidation, see iterators, invalidation
ios::skipws 126
iostreams library, SGI implementation
of 220
ISO Standard for C++
see C++ Standard, The
istream_iterators 157, 210
operator>> and 126
parsing ambiguities and 35
istreambuf_iterators 157, 210
use for efficient I/O 126–127
Items on efficiency, list of 9
iterator
comparisons with const iterators 118
other iterator types vs. 116–119
reverse_iterator’s base and 123–125
iterator_traits 113
value_type typedef 42
iterators
see also istreambuf_iterators,
ostreambuf_iterators
base, erasing and 124
base, insertion and 124
casting 120
categories 5
see also input iterators, output iterator,
forward iterators, bidirectional iterators, random access iterators
in hashed containers 114
choosing among types 116–119
conversions among types 117
dererereferencing function object for 91
implemented as pointers 120
invalidation
during deque::insert 185
during erasing 14, 45, 46
during insertion 14
during vector reallocation 59
during vector/string insert 68
during vector/string reallocation 66
during vector::insert 27
in hand-written loops 185
in standard sequence containers 17
in STLPport STL implementation 221
predicting in vector/string 68
undefined behavior from 27, 45, 46, 185
unique rules for deque 15
pointers in vector vs. 75
relationship between iterator and
reverse_iterator’s base 123–125
typedefs for 18–19
types in containers
see also iterator, const_iterator,
reverse_iterator, const_reverse_iterator
casting among 121
types, mixing 119

J
Johnson, Curt xviii
Johnson, Ralph 226
see also Design Patterns
Johnson, Tim xvii
Jones, Jason xviii
Josuttis, Nicolai xv, xvii, 225, 227
see also C++ Standard Library, The

K
Kanze, James xvi
Kasperski, Marcin xvii
Kernighan, Brian xvii
key_comp 85, 110
keys, for set/multiset, modifying 95–100
King, Martin Luther, Jr., allusion to 51
Kreft, Klaus xvi, 228

L
Langer, Angelika xvi, xviii, 228
leaks, see resource leaks
Leary-Coutu, Chanda xviii
legacy APIs
filling containers from 77
sorted vectors and 76
vector and string and 74–77
vector<bool> and 81
lemur 88
less 210
operator< and 177–180
less_equal 92
Lewandowski, Scott xvi
lexicographic comparison 231
lexicographical_compare 153
strcmp and 153
use for case-insensitive string
comparisons 150–154
lhs, as parameter name 8–9
linear time complexity 6
  for binary search algorithms with bi-de-
  rectional iterators 148
logarithmic complexity vs. 190
list
  algorithm specializations 192
  iterator invalidation in, see iterators,
    invalidation
  merge 192
remove 142–143
  vs. the erase-remove idiom 43
  sort 137
splice
  exception safety of 50
  vs. size 23–24
unique 143
Little Engine that Could, The, allusion
to 63
local classes
  definition of 189
  type parameters and 189
locales 232–233
  case-insensitive string comparisons
    and 229–237
facets and 234–235
locality of reference 103
  improving via allocators 55
locking objects 60
logarithmic time complexity
  linear complexity vs. 190
  meaning for binary search
  algorithms 147
longest algorithm name 153
lookup speed, maximizing 100
lower_bound
  equality vs. equivalence and 195
  related functions vs. 192–201

M
mailing list for Scott Meyers xiii
Manning, Jared xvii
map
  add or update functionality in 107
  constness of elements 99
  iterator invalidation in, see iterators,
    invalidation
  key, casting away constness 99
  key_comp member function 110
  value_type typedef 108
Matthew 25:32, allusion to 222
max_element 157
max_size 66
Mayhew, David 226
  see also Efficient C++
mem_fun
  declaration for 175
  reasons for 173–177
  reference-to-reference problem and 222
mem_fun_ref
  reasons for 173–177
  reference-to-reference problem and 222
mem_fun_ref_t 175
mem_fun_t 175
member funs, vs. algorithms 190–192
member function templates
  see member templates
member templates
  avoiding client redundancy with 38
  in the STL 239–240
  Microsoft’s STL platforms and 239–244
    vector::insert as 240
  workaround for when missing 242
memory fragmentation, allocators and 54
memory layout
  for string 69–71, 75
  for vector 74
memory leaks, see resource leaks
memory, shared, allocators and 55–56
merge 148, 192
Meyers, Scott
  mailing list for xiii
  web site for xiii
Microsoft’s STL platforms 239–244
  Dinkumware replacement library
    for 243
microsoft.public.vc.stl xv
min_element 157
mismatch 151
  use for case-insensitive string
    comparisons 150–154
mixing iterator types 119
modifying
  components in std 178
  const objects 99
  set or multiset keys 95–100
monomorphic function objects 164
Moore, Aaron xvii
More Effective C++
  auto_ptr and 40
  bibliography entry for 225
  citation number omitted for 3
  errata list 228
  smart pointers and 39
placement new discussion in 56
proxy objects discussion in 49, 80
reference counting discussion in 4, 71
resource acquisition is initialization
  discussion in 61
smart pointer discussion in 39
Index

STL overview in xi
URL for auto_ptr update page for 228
URL for errata list for 228
More Exceptional C++
  bibliography entry for 226
multimap
  constness of elements 95
  find vs. count in 199
  find vs. lower_bound in 201
  indeterminate traversal order in 87
  iterator invalidation in, see iterators, invalidation
  key, casting away constness 99
  value_type typedef 108
multiple deletes 64
multiset
  constness of elements 95
  corrupting via element modification 97
  find vs. count in 199
  find vs. lower_bound in 201
  indeterminate traversal order in 87
  iterator invalidation in, see iterators, invalidation
  keys, modifying 95-100
multithreading
  allocators and 54
  containers and 58-62
  reference counting and 64-65
  string and 64-65

N

Naran, Siemel xvi
newsgroups xv
  comp.lang.c++.moderated xv
  comp.std.c++ xv
  microsoft.public.vc.stl xv
node-based containers
  see also standard associative containers, list, slist, hashed containers
  allocators and 52
  clustering in 103
  definition of 13
nonstandard containers
  see hashed containers, slist, rope
not1 155, 156, 169, 170, 172, 210, 222
  adaptability and 170
not2 152, 222
  adaptability and 170
nth_element 133-138
nulls, embedded 75, 154
<numeric> 157

Effective STL

O

objects
  copying, in containers 20-22
  for locking 60
  slicing 21-22, 164
  temporary, created via casts 98
One True Editor, the, see Emacs
operator new, interface,
  vs. allocator::allocate 51
operator() 5
  declaring const 168
  functor class and 5
  inlining and 202
  overloading
    adaptability and 173
    in functor classes 114
operator++, side effects in 45
operator<, forward iterators and 5
operator. ("operator dot") 49
operator<, less and 177-180
operator>>
  istream_iterators and 126
  sentry objects and 126
  whitespace and 126
operator[], vs. insert in map 106-111
optimizations
  algorithms and 183
  function pointers and 203
  inlining and 202
  istreambuf_iterators and 127
  range insertions and 31
  reference counting and 64
  small strings and 71
  stricmp/strcmpi and 154
  to reduce default allocator size 70
ostream, relation to basic_ostream 230
ostream_iterators 216
ostreambuf_iterators 216
  efficiency and 127
output iterator, definition of 5
overloading, operator() in functor classes 114

P

page faults 102, 103
pair, comparison functions for 104
parameters
  function objects vs. functions 201-205
  pointers to functions 34
  type, local classes and 189
parentheses
  ignored, around parameter names 33
to distinguish function and object declarations 35
parse, most vexing in C++ 33–35
parsing, objects vs. functions 33–35
partial_sort 133–138
partial_sum 157
partition 133–138
containers of pointers and 145
remove vs. 141
pass-by-value
function objects and 163
functor classes and 162–166
penguin 88
Perfect Woman, see Woman, Perfect
performance, see efficiency
Persephone xviii
Pimpl Idiom 165
placement new 56
Plauger, P. J. xvi, 114, 227
pointers
allocator typedef for 48
as iterators 120
as return type from vector::begin 75
assignments, avoiding superfluous 31
comparison functions for 88–91
deleting in containers 36–40
dereferencing function object for 91
destructors for 36
invalidation, see iterators, invalidation
iterators in vector vs. 75
parameters, binary_function and 172
parameters, unary_function and 172
returned from reverse_iterator::base 125
smart, see smart pointers
to bitfields 80
to functions, as formal parameters 34
portability
#include and 209–210
casting const iterator to iterators 121
container:<auto_ptr> and 41
explicit template argument specification and 163
hashed containers, code using 112
identity, project1st, project2nd,
compose1, compose2, select1st,
select2nd and 219
multiple compilers and 3
range construction with
::istream_iterator and 35
reverse_iterator::base and 125
set/multiset key modification and 98
stateful allocators and 50, 51
STLP, STL implementation and 220
stricmp/strcmpi 154
Potter, John xvi, xvii
predicate class, definition of 166
predicates
definition of 166
need to be pure functions 166–169
predicting iterator invalidation, in vector/string 68
principle of least astonishment, the 179
priority_queue 138
as part of the STL 2
project1st 219
project2nd 219
proxy objects 49
containers of 82
vector<bool> and 80
ptr_fun, reasons for 173–177
pure function, definition of 166
push_back, back_inserter and 130
push_front, front_inserter and 130
qsort 162
declaration for 162
sort vs. 203
queue, as part of the STL 2
Rabinowitz, Marty xviii
random access iterators
definition of 5
sorting algorithms requiring 137
range
destination, ensuring adequate
size 129–133
member functions 25
input iterators and 29
single-element versions vs. 24–33
summary of 31–33
pointer assignments in list and 31
sorted, algorithms requiring 146–150
summarizing 156–161
raw_storage_iterator 52
RB trees, see red-black trees
rbegin/rend, relation to begin/end 123
reallocations
invalidation of iterators during 59
minimizing via reserve 66–68
rebinding allocators 53
red-black trees 190, 191, 214
redundant computations, avoiding via
algorithm calls 182
Reeves, Jack xvi, 227
reference counting
disabling, for string 65
multithreading and 64–65
smart pointers 39, 146
see also Boost, shared_ptr
string and 64–65
The C++ Standard and 64
this book and 4
references
allocator typedef for 48
casting to 98
invalidation, see iterators, invalidation
parameters, binary_function and 171
parameters, unary_function and 171
to bitfields 80
reference-to-reference problem 222
remove 139–142
see also erase-remove idiom
on containers of pointers 143–146
partition vs. 141
remove_copy_if 44
remove_if 44, 144, 145
see also remove
possible implementation of 167
replace_if 186
replacing STL implementations 243
reporting bugs in this book xii
reserve
insert iterators and 131
resize vs. 67
resize
reallocation and 67
reserve vs. 67
resource acquisition is initialization 61
resource leaks 36, 39, 63, 144, 145
avoiding via smart pointers 39, 146
preventing via classes 61
result_type 170
return type
allocator::allocate vs. operator new 51
for container::begin 75
for distance 122
for erase 32, 117
for function objects for accumulate 158
for insert 17, 32, 117
for vector::operator[] 80
reverse order
inserting elements in 184
reverse_iterator
base member function 123–125
other iterator types vs. 116–119
rhs, as parameter name 8–9
Rodgers, Mark xv, xvi, xvii
rolling back, insertions and erasures 14
rope 218

S
Scheme 206
second_argument_type 170
select1st 219
select2nd 219
sentry objects, operator<< and 126
separate heaps, allocators and 96–58
sequence containers
see standard sequence containers
set
constness of elements 95
corrupting via element modification 97
iterator invalidation in, see iterators, invalidation
keys, modifying 95–100
membership test, idiomatic 199
set_difference 148
set_intersection 148
set_symmetric_difference 148, 153
set_union 148
set getc 127
SGI
hashed containers implementation 114
iostreams implementation 220
slist implementation 218
STL web site 94, 207, 217–220
thread-safety definition at 58
URL for 217, 227
shared memory, allocators and 55–56
shared_ptr, see Boost, shared_ptr
shrink to fit, see swap trick, the
side effects
accumulate and 160
for_each and 160
in operator++ 45
size
vs. capacity 66–67
vs. empty 23–24
size_type 158
sizeof, variations when applied to string 69
skipws 126
slicing problem 21–22, 164
slist 218
small string optimization 71
Smallberg, David xvi, xvii
smart pointers
see also Boost, shared_ptr
avoiding resource leaks with 39, 146
dereferencing function object for 91
implicit conversions and 146
sort 133–138
qsort vs. 203
sorted range
algorithms requiring 146–150
sorted vectors
associative containers vs. 100–106
legacy APIs and 76
sorting
algorithms for 133–138
auto_ptr/s 41
consistency and 149
stability, in sorting 135
stable_partition 133–138
stable_sort 133–138
stack, as part of the STL 2
Staley, Abbi xviii
standard associative containers
see also containers
bidirectional iterators and 5
comparison funcs for pointers 88–91
definition of 5
“hint” form of insert 100, 110
iterator invalidation in, see iterators, invalidation
key_comp member function 85
search complexity of 6
sorted vector vs. 100–106
typical implementation 52, 190
Standard for C++
see C++ Standard, The
standard sequence containers
see also containers
definition of 5
erase’s return value 46
iterator invalidation in, see iterators, invalidation
push_back, back_inserter and 130
push_front, front_inserter and 130
Standard Template Library, see STL
Stepanov, Alexander 201
STL
algorithms, vs. string member functions 230
arrays and 2
bitset and 2
complexity guarantees in 5–6
container adapters and 2
containers, selecting among 11–15
definition of 2
documentation, on-line 217
early usage problems with xi
extending 2
free implementations of 217, 220
function call syntax in 175
implementations
compiler implementations vs. 3
Dinkumware bug list for MSVC 244
replacing 243
member templates in 239–240
platform, see STL platform
priority_queue and 2
queue and 2
stack and 2
thread safety and 58–62
valarray and 2
web sites about 217–223
wide-character strings and 4
STL platform
definition of 3
Microsoft’s, remarks on 239–244
STLport 220–221
debg mode 185, 216
detecting invalidated iterators in 221
hashed containers at 112
URL for 217
strcmp 152, 234
internationalization issues and 150
lexicographical_compare and 153
strcmpl 154
streams, relation to string 230
stricmp 154
strict weak ordering 94
string
allocators in 69
alternatives to 65
arrays vs. 63–66
as typedef for basic_string 65
c_str member function 75
comparisons
case-insensitive 150–154, 235–237
using locales 229–237
cost of increasing capacity 66
disabling reference counting 65
embedded nulls and 75, 154
growth factor 66
implementation variations 68–73
inheriting from 37
iterator invalidation in, see iterators, invalidation
iterators as pointers 120
legacy APIs and 74–77
mem funcs vs. algorithms 230
memory layout for 75
minimum allocation for 72
multithreading and 64–65
reference counting and 64–65
relation to basic_string 4, 210, 229
relation to streams 230
reserve, input iterators and 131
resize vs. reserve 67
shrink to fit 78–79
size vs. capacity 66–67
size_type typedef 158
sizeof, variations in 69
small, optimization for 71
summing lengths of 158
trimming extra capacity from 77–79
vector vs. 64
vector<char> vs. 65
whether reference-counted 65
wstring and 4
string_char_traits 211
strings
  case-insensitive 229–230
  wide-character, see wstring
Stroustrup, Bjarne xvi, 68, 226, 228
  see also C++ Programming Language, The
structs, vs. classes for functor classes 171
summarizing ranges 156–161
Sutter, Herb xvi, xvii, 65, 226, 227, 228
  see also Exceptional C++
swap trick, the 77–79

T

templates
declaring inside functions 188
explicit argument specification for 122, 163, 234
instantiating with local classes 189
member
  in the STL 239–240
  Microsoft’s platforms and 239–244
parameters, declared via class vs. typename 7
temporary objects, created via casts 98
thread safety, in containers 58–62
  see also multithreading
tolower 151
  as inverse of toupper 235
toupper
  as inverse of tolower 235
  cost of calling 236–237
traits classes 113, 211, 230
transactional semantics 14
transform 129, 186
traversal order, in multiset, multimap 87
trees, red-black 190, 191, 214
typedefs
  allocator::pointer 48
  allocator::reference 48
  argument_type 170
  container::size_type 158
  container::value_type 36
  first_argument_type 170
  for container and iterator types 18–19
  mem_fun and 176
  mem_fun_ref and 176
  ptr_fun and 170
  result_type 170
  second_argument_type 170
  string as 65
  wstring as 65
typename
  class vs. 7
  dependent types and 7–8

U

Unary function 170–172
  pointer parameters and 172
  reference parameters and 171
undefined behavior
  accessing v[0] when v is empty 74
  applying some algorithms to ranges of unsorted values 147
  associative container comparisonfuncs yielding true for equal values 92
  attempting to modify objects defined to be const 99
  changing a set or multiset key 97
  deleting an object more than once 64
  deleting derived object via ptr-to-base with a nonvirtual destructor 38
  detecting via STLport’s debug mode 220–221
  modifying components in std 178
  multithreading and 59
  practical meaning of 3–4
  side effects inside accumulate’s function object 160
  specifying uninitialized memory as destination range for algorithms 132
  using algorithms with inconsistent sort orders 149
  using the wrong form of delete 64
  when using invalidated iterator 27, 45, 46, 185
underscores, in identifiers 213
uninitialized_fill 52
uniq 148
unique 145, 148, 149
  see also remove
unique_copy 148, 149
unsigned char, use in cctype and <ctype.h> 151
upper_bound 197–198
  related functions vs. 192–201
Urbano, Nancy L., see Perfect Woman
URLs
  for Austern’s sample allocator 228
  for auto_ptr update page 228
  for Boost web site 217, 227
  for Dinkumware web site 243
  for Effective C++ CD errata list 228
  for Effective C++ errata list 228
  for Effective STL errata list xii
  for Josuttis’ sample allocator 227
  for More Effective C++ errata list 228
  for Persephone’s web site 71
  for purchasing The C++ Standard 226
  for Scott Meyers’ mailing list xiii
  for Scott Meyers’ web site xiii
  for SGI STL web site 217, 227
Index

for STLport web site 217
for this book’s errata list xii
for this book’s web site xii

use_facet 234
- cost of calling 234
- explicit template argument specification and 234

Usenet newsgroups, see newsgroups

valarray, as part of the STL 2

value_type typedef
- in containers 36, 108
- in iterator_traits 42

vector
- see also vector<bool>, vector<char>
- arrays vs. 63–66
- contiguous memory for 74
- cost of increasing capacity 66
- growth factor 66
- iterator invalidation in, see iterators, invalidation
- iterators as pointers 120
- legacy APIs and 74–77
- reserve, input iterators and 131
- resize vs. reserve 67
- return type from begin 75
- shrink to fit 78–79
- size vs. capacity 66–67
- sorted
- legacy APIs and 76
- vs. associative containers 100–106
- string vs. 64
- trimming extra capacity from 77–79
- vector::insert, as member template 240

vector<bool>
- alternatives to 81
- legacy APIs and 81
- problems with 79–82
- proxy objects and 80
- vector<char>, vs. string 65
- virtual functions, toupper and 236–237

Visual Basic 127

Visual C++, see Microsoft’s STL platforms

Vlissides, John 226
- see also Design Patterns

vocabulary, algorithms as 186

Von, Victor xvii

Wait, John xviii

wchar_t 4

web sites
- see also URLs
- for STL-related resources 217–223

Effective STL

whitespace, operator<< and 126

wide-character strings, see wstring

Widget, use in this book 9

Wizard of Oz, The 83

Woman, Perfect, see Urbano, Nancy L.
wombat 88

workarounds
- for improperly declared iterator-related functions 119
- for Microsoft’s STL platforms 242–244
- for missing member templates 242
- for use_facet 234
- function objects as 204
- write-only code, avoiding 206–208

writing the author xii

wstring
- as typedef for basic_string 65
- relation to basic_string 4
- string and 4
- this book and 4

Y

Yelle, Dennis xvii

Z

Zolman, Leor xvii, 211, 228

W