Effective STL

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

Scott Meyers

ADDISON–WESLEY PROFESSIONAL COMPUTING SERIES

Effective STL

Effective STL

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

Scott Meyers

ADDISON-WESLEY

Boston • San Francisco • New York • Toronto • Montreal
London • Munich • Paris • Madrid
Capetown • Sydney • Tokyo • Singapore • Mexico City

E-book ISBN 13: 978-0-321-51580-3
E-book ISBN 10: 0-321-51580-3
Second e-book release, April 2011 (essentially identical to the 13th Paper Printing).

For Woofieland.

This page intentionally left blank

Contents

This page intentionally left blank

Preface

It came without ribbons! It came without tags!
It came without packages, boxes or bags!

— Dr. Seuss, *How the Grinch Stole Christmas!*, Random House, 1957

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

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

Then I began to notice something that took me by surprise. Despite the portability problems, despite the dismal documentation, despite the compiler diagnostics resembling transmission line noise, many of my consulting clients were using the STL anyway. Furthermore, they weren't just playing with it, they were using it in production code! That was a revelation. I knew that the STL featured an elegant design, but any library for which programmers are willing to endure portability headaches, poor documentation, and incomprehensible error messages has a lot more going for it than just good design. For an increasingly large number of professional programmers, I realized, even a bad implementation of the STL was preferable to no implementation at all.

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

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

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

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

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

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

STAFFORD, OREGON
APRIL 2001

Acknowledgments

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

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

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

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

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

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

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

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

Good technical books require a thorough pre-publication vetting, and I was fortunate to benefit from the insights of an unusually talented group of technical reviewers. Brian Kernighan and Cliff Green offered early comments on a partial draft, and complete versions of the manuscript were scrutinized by Doug Harrison, Brian Kernighan, Tim Johnson, Francis Glassborow, Andrei Alexandrescu, David Smallberg, Aaron Campbell, Jared Manning, Herb Sutter, Stephen Dewhurst, Matt Austern, Gillmer Derge, Aaron Moore, Thomas Becker, Victor Von, and, of course, Mark Rodgers. Katrina Avery did the copyediting.

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

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

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

This printing of the book is better than earlier printings, because I was able to address problems identified by the following sharp-eyed readers: Jon Webb, Michael Hawkins, Derek Price, Jim Scheller, Carl Manaster, Herb Sutter, Albert Franklin, George King, Dave Miller, Harold Howe, John Fuller, Tim McCarthy, John Hershberger, Igor Mikolic-Torreira, Stephan Bergmann, Robert Allan Schwartz, John Potter, David Grigsby, Sanjay Pattni, Jesper Andersen, Jing Tao Wang, André Blavier, Dan Schmidt, Bradley White, Adam Petersen, Wayne Goertel, Gabriel Netterdag, Jason Kenny, Scott Blachowicz, Seyed H. Haeri, Gareth McCaughan, Giulio Agostini, Fraser Ross, Wolfram Burkhardt, Keith Stanley, Leor Zolman, Chan Ki Lok, Motti Abramsky, Kevlin Henney, Stefan Kuhlins, Phillip Ngan, Jim Phillips, Ruediger Dreier, Guru Chandar, Charles Brockman, Day Barr, Eric Niebler, Sharad Kala, Declan Moran, Nick de Smith, David Callaway, Shlomi Frank, Andrea Griffini, Hans Eckardt, David Smallberg, Matt Page, Andy Fyfe, 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.

Introduction

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

I wrote this book for you.

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

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

Defining, Using, and Extending the STL

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

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

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

Citations

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

```
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 #includes and ignore the fact that STL components are in namespace std. When defining the map m, I could have written this,

```
#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,

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

I write this:

```
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; T 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:

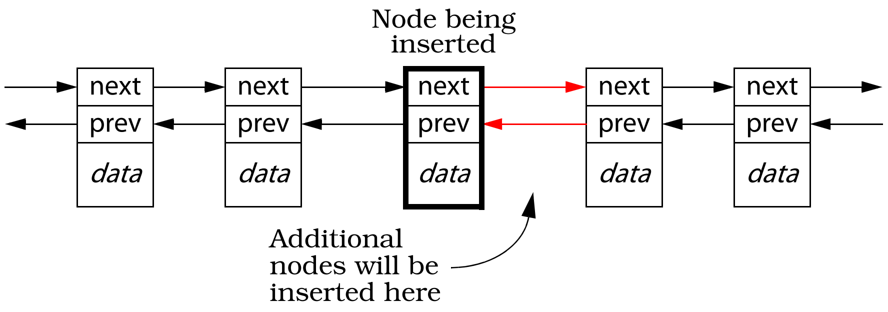
```
template<typename C>
bool lastGreaterThanFirst(const C& container)
{
   if (container.empty()) return false;

   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 incorrectly 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 important. 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 instance, this diagram from Item 5 uses color to identify the two pointers that are affected when a new element is inserted into a list:



I also use color for chapter numbers, but such use is purely gratuitous. This being my first two-color book, I hope you'll forgive me a little 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,

```
if (x == y) ...                          // assume x and y are Widgets
```

x, which is on the left-hand side of the "==", is known as lhs inside operator==, and y is known as rhs.

As for the class name Widget, that has nothing to do with GUIs or toolkits. 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 between 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:

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.