
Scott Meyers, *Effective STL*¹

Book Review and Commentary
by David R. Miller
July, 2001

The “Effective” Series

This is Scott Meyers’ third “Effective” book. His first book, *Effective C++*², was a popular success with “50 Specific Ways to Improve Your Programs and Designs.” Its popularity inspired the author to publish 35 more C++ guidelines in *More Effective C++*³.

In his first two books, Meyers gives us concise, pithy maxims that apply to common issues faced by C++ programmers. The table of contents for *Effective C++* reads like a series of sound bites; rules so simple and concrete that some organizations incorporate them into their coding style guidelines:⁴

- “Prefer `const` and `inline` to `#define`”
- “Prefer `new` and `delete` to `malloc` and `free`”
- “Avoid casts down the inheritance hierarchy”

However, only a handful of the fifty topics in *Effective STL* are simple “guidelines.” Some are broad topics that are expressed in terms of “understand this,” such as:

- “Understand how to use a `reverse_iterator`’s base iterator” (Item 28)
- “Understand the proper implementation of `copy_if`” (Item 36)
- “Distinguish among `count`, `find`, `binary_search`, `lower_bound`, `upper_bound`, and `equal_range`. (Item 45)

¹ Meyers, Scott, *Effective STL: 50 Specific Ways to Improve Your Use of the Standard Template Library*. Addison-Wesley, 2001. ISBN 0-201-74962-9.

² *Effective C++: 50 Specific Ways to Improve Your Programs and Designs, Second Edition*, Addison-Wesley, 1998. ISBN 0-201-92488-9.

³ *More Effective C++: 35 New Ways to Improve Your Programs and Designs*, Addison-Wesley, 1996. ISBN 0-201-63371-X.

⁴ ParaSoft markets an automated tool called CodeWizard that checks source code for violations of the guidelines in *Effective C++*. (<http://www.parasoft.com/products/wizard/>). However, while I believe that an understanding of the guidelines in *Effective C++* is valuable, blind application without understanding could do as much harm as good.

This is a book for programmers already familiar with STL concepts who want to explore the STL in more depth. Here you will find fifty interesting discussions of various corners of the Standard Template Library, not a simple checklist.

A Little History

The Standard Template Library was originally developed by researchers at Hewlett Packard. With some tweaking and tuning, it was incorporated in to the C++ Standards after the C++ standardization was under way.⁵ Though the Committee did a commendable job of fitting it with the rest of the C++ library and language, it is still often regarded as a somewhat distinct subset of the C++ library, with its own personality, features, and flaws. It is also easily ignored by C++ programmers. There isn't anything that the STL does that can't be done some other less generic way.

Audience

This book is not a tutorial, because it assumes a prior working knowledge of STL concepts. It isn't a reference book, because it is too narrative and doesn't offer a complete coverage of the subject. This book may overwhelm the novice STL programmer, and it doesn't *have* to be part of the STL expert's library. But in the range of expertise between novice and expert, there are programmers who will appreciate and enjoy these fifty little journeys into the cracks and crevices of the STL.

Before spending money on this book, spend first on tutorials of the C++ language. Become familiar with class construction and inheritance, user-defined operator overloading, and template specialization. Then spend your money on tutorials concerning generic programming techniques. Become familiar with the basic concepts of iterators, containers, functors, and traits—especially as implemented with C++ templates.

Then buy this book. Some parts may seem too introductory, others too obscure. But the parts that strike at just the right spot will be worth the price of the book.

Examples

Most of the Items in *Effective STL* are interesting, but not all of them apply to everyday STL programming problems. In Item 24, for example, Meyers devotes five pages to the subtle performance differences between `operator[]` and member function `insert()` on `map` containers. This is an interesting piece of trivia worthy of a footnote⁶ or a couple of sentences,⁷ but I question whether it is among the fifty most important guidelines for STL programmers.⁸

⁵ Alex Stepanov and David Musser were principal designers of the original STL. Stepanov and Meng Lee made the proposal to the ANSI/ISO C++ Standards Committee in 1994 to include the STL in the C++ Standard Library.

⁶ As in footnote 9 in "More on Iterators," by Thomas Becker in *C/C++ Users Journal*, June 2001.

⁷ The subject gets two sentences in *The Standard C++ Library: A Tutorial and Reference*, by Nicolai M. Josuttis, p. 207.

⁸ The difference, by the way, is that if you insert a new element with `map`'s subscript operator, it calls the default constructor on the new element followed by a call to the element's assignment operator. The member function `insert()` skips the call to the default constructor and calls the copy constructor instead. If performance is a problem, you will want to discover all the circumstances in which STL functions create, destroy, and copy

The general nature of the book can be inferred from an examination of Items 26 and 27. Item 26 tells us:

“Prefer `iterator` to `const_iterator`, `reverse_iterator` and `const_reverse_iterator`.”

Meyers would have us ignore const-correctness with iterators. He points out that there is no implicit or explicit conversion from `const_iterator` to `iterator`⁹. If you dutifully define a `const_iterator` that you can pass with confidence to non-modifying algorithms, you shouldn't be surprised that you cannot pass that iterator to a modifying algorithm. But you also cannot cast off its constness. Alas, if you have not designed your program with this in mind, you might find that the conversion from `const_iterator` to `iterator` is expensive and non-obvious.

Meyers fills four pages with this discussion, much of which is devoted to the point that `const_iterators` and `iterators` are not guaranteed to mix well in expressions. By itself, this is useful information, but Meyers does not explain in much depth *why* iterators of different types don't mix well in expressions. The novice STL programmer can memorize this guideline by rote. The experienced STL programmer will have already deduced this rule from an understanding of two principals: (1) how a proxy for a pointer type is implemented, and (2) the C++ conversion rules for matching a template argument with the declared template parameter type. I believe it is more valuable to learn the underlying principals than to memorize rules. I fear that there is not enough explanation of the “why” to enable a novice to remember this guideline, and not enough “meat” to hold the interest of the experienced programmer. I fear that some programmers will discard all hope of const-correctness because of potential inconveniences. Better than a rigid rule is the understanding of how to avoid these inconveniences while maintaining const-correctness.

In the final analysis about `const iterators`, Meyers concludes,

“...sometimes they're just not worth the trouble.”

Just in case you do need to convert a `const_iterator` to a non-const `iterator`, Item 27 gives you a cookbook formula for doing so. Meyers first explains that different types of iterators may be implemented as different, unrelated classes, with no facility prescribed for constructing a non-const `iterator` from a `const_iterator`. His cookbook solution is neat, it works, and it makes for an interesting discussion. Given container `c` of type `Container` and a `const_iterator` `ci`, you can cause `iterator` `i` to point to the same element in the following way:

```
Container::iterator i(c.begin());
advance(i, distance<Container::const_iterator>(i, ci));
```

First, `distance()` returns the number of elements from the beginning of the container to the present position of `ci`. Add that integer value to the location of `i` and *voilà!* You have a non-const `iterator` pointing to the same location as `ci`.

Meyers spends three pages on this one-line solution. He warns that while this is a constant-time operation on containers `vector`, `string`, and `deque`, it is a time-consuming linear operation on

elements. The case of insertion into a `map` is just one arbitrary illustration of how different techniques that achieve the same goal have different side effects.

⁹ If it is known that a particular `const_iterator` `cIter` on a particular compiler for a particular type `T` is a simple `const T*`, the conversion is `const_cast<T*>(cIter)`. But Meyers generally does not advocate non-portable code constructs.

containers `list`, `set`, `multiset`, `map`, `multimap`, and on most of the ubiquitous but non-standard hash containers. He doesn't tell you that for the non-linear cases this expression takes *twice* as long as necessary. Linear complexity applies when the `distance()` function must walk a sequence of nodes, one at a time, from the beginning of the container to the current position of `ci`. Then the `advance()` function walks the same sequence of nodes a second time so that `i` is advanced to the same position as `ci`. This can be made twice as fast by walking the nodes just once:

```
Container::iterator i(c.begin());
while (&*i !=&*ci)
    ++i;
```

This solution is twice as fast in the linear cases, but unnecessarily slow for random-access containers. Neither formula is optimum for all cases. Meyers fails to mention that the optimum solution – The STL Way – is based on iterator category tags. Iterators expose an identification of their “category” so that a template class or template function can be specialized for different iterator categories. Or functions may be overloaded on the iterator category type. For random access iterators, the `advance(distance())` idiom is best. For all other categories, the `while` loop is best. This gets to the essence of generic programming: one interface to work with all types. This also gets to the essence of my only objection to *Effective STL*.

Meyers begins his book with the disclaimer – or warning – that this book does not explain how to extend the STL:

“...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.”

The Standard Template Library is the best that the software industry has to offer for generic programming. The STL has been carefully crafted to allow extensions. Even the manner in which extensions are granted is generic. Limitations deny genericity, and this is how the book creates a conflict. Typically, the programmer just beginning to learn the STL does not yet appreciate the incredible power of extending the generic library. But the novice also does not have the background to comprehend intuitively many of the discussions in this book. A slightly more experienced STL programmer can see that the STL is a starting point, an embodiment of a new programming paradigm with potential bursting at its seams. The programmer with the generic programming mind-set is not satisfied to remain in the confines of the functions, iterators, and containers defined in the C++ STL. For such a programmer, the discussions in this book may feel like excursions through a few narrow corridors of a narrowly defined library.

Summary

For the programmer who wants an introduction to STL concepts while also brushing up on features of the C++ core language, there are two books I recommend that represent the extremes of verbosity and parsimony. *A C++ Primer, 3rd Edition* by Stanley Lippman and Josée Lajoie¹⁰ contains over 1200 pages of clearly written explanations and examples of the C++ core language and all of the C++ Standard Library. On the other extreme, the slim 320-page *Accelerated C++* by Andrew Koenig and Barbara E. Moo¹¹ will have you programming with containers and iterators in the first fifty pages, even if your C++ is rusty. All these authors have been major players in the creation of the C++ language or

¹⁰ Addison Wesley, 1998. ISBN 0-201-82470-1.

¹¹ Addison-Wesley, 2000. ISBN 0-201-70353-X.

in the C++ standardization process. For a reference book of the entire C++ Standard Library including the STL, but not of the core language, there is no book equal to *The C++ Standard Library* by Nicolai M. Josuttis¹²

Scott Meyers has an excellent reputation as author and instructor. While *Effective STL* lives up to his reputation, it is confined by a limited approach to the STL. If you are interested in the nuances of STL, then buy this book and study it, but remember that the STL is just a starting point for a whole new programming paradigm.

¹² Addison-Wesley, 1999. ISBN 0-201-137926-0.