原 版 风 暴 系 列

PEARSON
Addison Wesley

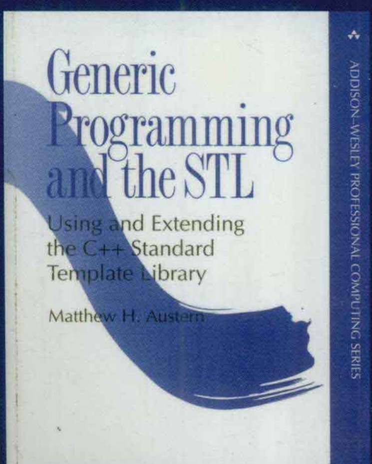
Generic Programming and the STL

Using and Extending the C++ Standard Template Library

泛型编程与STL

（影印版）

[美] Matthew H. Austern 著



Pearson Education
中国电力出版社
www.infopower.com.cn

原 版 风 暴 系 列

Generic Programming and the STL

Using and Extending the C++ Standard Template Library

泛型编程与STL

（影印版）

[美] Matthew H. Austern 著

中国电力出版社

Generic Programming and the STL: Using and Extending the C++ Standard Template Library (ISBN 0-201-30956-4)
Matthew H. Austern

北京市版权局著作合同登记号：图字：01-2003-1010

图书在版编目（CIP）数据

泛型编程与 STL /（美）奥斯滕著．—影印本．—北京：中国电力出版社，2003
（原版风暴系列）
ISBN 7-5083-1805-6
Ⅰ.泛… Ⅱ.奥… Ⅲ. C 语言－程序设计－英文 Ⅳ.TP312
中国版本图书馆 CIP 数据核字（2003）第 086084 号

丛 书 名：原版风暴系列
书　　名：泛型编程与STL（影印版）
编　　著：（美）Matthew H. Austern
责任编辑：姚贵胜
出版发行：**中国电力出版社**
　　　　　地址：北京市三里河路6号　　邮政编码：100044
　　　　　电话：（010）88515918　　传　　真：（010）88518169
印　　刷：北京丰源印刷厂
开　　本：787×1092 1/16　　印　　张：35.5
书　　号：ISBN 7-5083-1805-6
版　　次：2003 年 11 月北京第一版　　2003 年 11 月第一次印刷
定　　价：58.00 元

Preface

This is not a book about object-oriented programming.

You may think that's odd. You probably found this book in the C++ section of the bookstore, after all, and you've probably heard people use *object oriented* and *C++* synonymously, but that isn't the only way to use the C++ language. C++ supports several fundamentally different paradigms, the newest and least familiar of which is generic programming.

Like most new ideas, generic programming actually has a long history. Some of the early research papers on generic programming are nearly 25 years old, and the first experimental generic libraries were written not in C++ but in Ada [MS89a, MS89b] and Scheme [KMS88]. Yet generic programming is new enough that no textbooks on the subject exist.

The first example of generic programming to become important outside of research groups was the STL, the C++ Standard Template Library. The Standard Template Library, designed by Alexander Stepanov (then of Hewlett-Packard Laboratories) and Meng Lee, was accepted in 1994 as part of the C++ standard library. The freely available "HP implementation" [SL95], which served as a demonstration of the STL's capabilities, was released the same year.

When the Standard Template Library first became part of the C++ standard, the C++ community immediately recognized it as a library of high-quality and efficient container classes. It is always easiest to see what is familiar, and every C++ programmer is familiar with container classes. Every nontrivial program requires some way of managing a collection of objects, and every C++ programmer has written a class that implements strings or vectors or lists.

Container class libraries have been available since the earliest days of C++, and when "template" classes (parameterized types) were added to the language, one of their first uses—indeed, one of the main reasons that templates were introduced—was parameterized container classes. Many different vendors, including Borland, Microsoft, Rogue Wave, and IBM, wrote their own libraries that included `Array<T>` or its equivalent.

The fact that container classes are so familiar made the STL seem at first to be nothing more than yet another container class library. This familiarity diverted attention from the ways in which the STL was unique.

The STL is a large and extensible body of efficient, generic, and interoperable software components. It includes many of the basic algorithms and data structures of computer science, and it is written so that algorithms and data structures are decoupled from each other. Rather than a container class library, it is more accurate to think of the STL as a library of generic algorithms; containers exist so that the algorithms have something to operate on.

You can use the existing STL algorithms in your programs, just as you can use the existing STL containers. For example, you can use the generic STL `sort` as you would use the function `qsort` from the standard C library (although `sort` is simpler, more flexible, safer, and more efficient). Several books, including David Musser and Atul Saini's *STL Tutorial and Reference Guide* [MS96] and Mark Nelson's *C++ Programmer's Guide to the Standard Template Library* [Nel95], explain how to use the STL in such a way.

Even this much is useful. It is always better to reuse code than to rewrite it, and you can reuse the existing STL algorithms in your own programs. This is still, however, only one aspect of the STL. The STL was designed to be extensible; that is, it was designed so that, just as the different STL components are interoperable with each other, they are also interoperable with components you write yourself. Using the STL effectively means extending it.

Generic Programming

The STL is not just a collection of useful components. Its other aspect, which is less widely recognized and understood, is that it is a formal hierarchy of abstract requirements that describe software components. The reason that the STL's components are interoperable and extensible, and the reason that you can add new algorithms and new containers and can be confident that the new pieces and the old can be used together, is that all STL components are written to conform to precisely specified requirements.

Most of the important advances in computer science have been the discoveries of new kinds of abstractions. One crucial abstraction supported by all contemporary computer languages is the subroutine (a.k.a. the procedure or function—different languages use different terminology). Another abstraction supported by C++ is that of abstract data typing. In C++, it is possible to define a new data type together with that type's basic operations.

The combination of code and data forms an abstract data type, one that is always manipulated through a well-defined interface. Subroutines are an important abstraction because using a subroutine doesn't require that you depend on (or even necessarily know) its exact implementation; similarly, you can use an abstract data type—you can manipulate and even create values—without depending on the actual representation of the data. Only the interface is important.

C++ also supports object-oriented programming [Boo94, Mey97], which involves hierarchies of polymorphic data types related by inheritance. Object-oriented programming has one more layer of indirection than abstract data typing, thus it achieves one more step in abstraction. In some circumstances you can refer to a value and manipulate it without needing to specify its exact type. You can write a single function that will operate on a number of types within an inheritance hierarchy.

Generic programming, too, means identifying a new kind of abstraction. The central abstraction of generic programming is less tangible than earlier abstractions like the subroutine or the class or the module. It is a *set of requirements* on data types. This is a difficult abstraction to grasp because it isn't tied to a specific C++ language feature. There is no keyword in C++ (or, for that matter, in any contemporary computer language) for declaring a set of abstract requirements.

What generic programming provides in return for understanding an abstraction that at first seems frustratingly nebulous is an unprecedented level of flexibility. Just as important, it achieves abstraction without loss of efficiency. Generic programming, unlike object-oriented programming, does not require you to call functions through extra levels of indirection; it allows you to write a fully general and reusable algorithm that is just as efficient as an algorithm handcrafted for a specific data type.

A generic algorithm is written by abstracting algorithms on specific types and specific data structures so that they apply to arguments whose types are as general as possible. This means that a generic algorithm actually has two parts: the actual instructions that describe the steps of the algorithm and the set of requirements that specify precisely which properties its argument types must satisfy.

The central innovation of the STL is the recognition that these type requirements can be specified and systematized. That is, it is possible to define a set of abstract *concepts* and to say that a type conforms to one of those concepts if it satisfies a certain set of requirements. These concepts are important because most of the assumptions that algorithms make about their types can be expressed both in terms of conformance to concepts and in terms of the relationships between different concepts. Additionally, these concepts form a well-defined hierarchy, one reminiscent of inheritance in traditional object-oriented programming but purely abstract.

This hierarchy of concepts is the conceptual structure of the STL. It is the most important part of the STL, and it is what makes reuse and interoperability possible. The conceptual structure would be important purely as a formal taxonomy of software components, even without its embodiment in code. The STL does include concrete data structures, such as `pair` and `list`, but to use those data structure effectively you must understand the conceptual structure they are built upon.

Defining abstract concepts and writing algorithms and data structures in terms of abstract concepts is the essence of generic programming.

How to Read This Book

This book describes the Standard Template Library as a library of abstract concepts. It defines the fundamental concepts and abstractions of the STL and shows what it means for a type to model one of those concepts or for an algorithm to be written in terms of a concept's interface. It discusses the classes and algorithms that are part of the basic STL, and it explains how you can write your own STL-compliant classes and algorithms and when you might want to do so. Finally, it includes a complete reference manual of all of the STL's concepts, classes, and algorithms.

Everyone should read Part I, which introduces the main ideas of the STL and of generic programming. It shows how to use and write a generic algorithm, and it explains what it means for an algorithm to be generic. Genericity has implications that go far beyond the ability to operate on multiple data types.

Exploring the idea of a generic algorithm leads naturally to the central ideas of *concepts*, *modeling*, and *refinement*, ideas that are as basic to generic programming as polymorphism and inheritance are to object-oriented programming. Generic algorithms on one-dimensional ranges, meanwhile, lead to the fundamental concepts of the STL: iterators, containers, and function objects.

Part I introduces the notation and the typographical conventions that are used throughout the remainder of the book: the terminology of modeling and refinement, the asymmetrical notation for ranges, and the special typeface for concept names.

The STL defines many concepts, some of which differ from each other only in technical details. Part I is an overview, and it discusses the broad outlines of STL concepts. Part II is a detailed reference manual that contains a precise definition of each STL concept. You may not wish to read Part II all the way through and, instead, may find it more useful to look up a particular concept only when you need to refer to its definition. (You should refer to Part II whenever you write a new type that conforms to an STL concept.)

Part III is also a reference manual. It documents the STL's predefined algorithms and classes. It relies heavily on the concept definitions of Part II. All STL algorithms and almost all concrete types are templates, and every template parameter can be characterized as the model of some concept. The definitions in Part III are cross-referenced to the appropriate sections of Part II.

In an ideal world, the book would end with Part III. Unfortunately, reality demands one more section, an appendix that discusses portability concerns. When the STL was first released, portability was not an issue because only one implementation existed. That is no longer the case, and whenever more than one implementation of any language or library exists, anyone who cares about portability must be aware of the differences between them.

The old HP implementation is still available by anonymous FTP from butler.hpl.hp.com, but it is no longer being maintained. A newer free implementation, from Silicon Graphics Computer Systems (SGI) is available at http://www.sgi.com/Technology/STL, and a port of the SGI STL to a variety of compilers, maintained by Boris Fomitchev, is available at http://www.metabyte.com/~fbp/stl. Finally, there are several different commercial STL implementations.

If you are writing real programs, it isn't enough to understand the theoretical design of the library; you also have to understand how the various STL implementations and the various C++ compilers differ. These unglamorous but necessary details are the subject of Appendix A.

Who Should Read This Book

While this book is largely about algorithms written in C++, it is neither an introductory textbook on algorithms nor a C++ tutorial. It does explain some of the unfamiliar aspects of both subjects. In particular, since the STL uses templates in ways that are uncommon in other sorts of C++ programs, it discusses some advanced techniques of programming with templates. This should not be your first C++ book, nor should it be your first exposure to an analysis of algorithms. You should know how to write basic C++ programs, and you should know the meaning of notation like $\mathcal{O}(N)$.

Two of the standard references on algorithms and data structures are Donald Knuth's *The Art of Computer Programming* [Knu97, Knu98a, Knu98b], and *Introduction to Algorithms*, by Cormen, Leiserson, and Rivest [CLR90]. Two of the best introductory C++ books are *The C++ Programming Language*, by Bjarne Stroustrup [Str97], and *A C++ Primer*, by Stanley Lippman and Josée Lajoie [LL98].

How This Book Came About

I joined the compiler group at Silicon Graphics Computer Systems (SGI) in 1996. Alex Stepanov had left HP to join SGI several months before. At the time, SGI's C++ compiler did not include an implementation of the Standard Template Library. Using the original HP implementation as our source base, Alex, Hans Boehm, and I wrote the version of the STL that was shipped with release 7.1 (and subsequent releases) of SGI's MIPSpro compiler.

The SGI Standard Template Library [Aus97] included many new and extended features, such as efficient and thread-safe memory allocation, hash tables, and algorithmic improvements. If these enhancements had remained proprietary, they would have been of no value to SGI's customers, so the SGI STL was made freely available to the public. It is distributed on the World Wide Web, along with its documentation, at http://www.sgi.com/Technology/STL.

The documentation, a set of Web pages, treats the STL's conceptual structure as central. It describes the abstract concepts that comprise the structure, and it documents the STL's algorithms and data structures in terms of the abstract concepts. We received many requests for an expanded form of the documentation, and this book is a response to those requests. The reference sections of this book, Parts II and III, are an outgrowth of the SGI STL Web pages.

The Web pages were written for and are copyrighted by SGI. I am using them with the kind permission of my management.

Acknowledgments

First and foremost, this book could not possibly have existed without the work of Alex Stepanov. Alex was involved with this book at every stage: he brought me to SGI, he taught me almost everything I know about generic programming, he participated in the development of the SGI STL and the SGI STL Web pages, and he encouraged me to turn the Web pages into a book. I am grateful to Alex for all of his help and encouragement.

I also wish to thank Bjarne Stroustrup and Andy Koenig for helping me to understand C++ and Dave Musser for his numerous contributions (some of which can be found in the bibliography) to generic programming, to the STL, and to this book. Dave used an early version of the SGI STL Web pages as part of his course material, and the Web pages were greatly improved through his and his students' comments.

Similarly, this book was greatly improved through the comments of reviewers, including Tom Becker, Steve Clamage, Jay Gischer, Brian Kernighan, Jak Kirman, Andy Koenig, Angelika Langer, Dave Musser, Sibylle Schupp, and Alex Stepanov, who read early versions. This book is more focused than it would have been without them, and it contains far fewer errors. Any mistakes that remain are my own.

Several mistakes in the first, second, and third printings of this book have now been corrected, and I wish to thank Sam Bradsher, Bruce Eckel, Guy Gascoigne, Ed James-Beckham, Jon Jagger, Nate Lewis, CH Lin, Shawn D. Pautz, John Potter, George Reilly, Manos Renieris, Peter Roth, Dieter Rothmeier, Andreas Scherer, and Jürgen Zeller, for bringing these errors to my attention.

I am also indebted to the staff at Addison-Wesley, including John Fuller, Mike Hendrickson, Marina Lang, and Genevieve Rajewski, for guiding me through the writing process, and to Karen Tongish for her careful copyediting.

Finally, I am grateful to my fiancée, Janet Lafler, for her love and support and for her patience during the many evenings and weekends that I spent writing.

Our cats, Randy and Oliver, tried to help by walking over my keyboard, but in the end I deleted most of their contributions.

Contents