

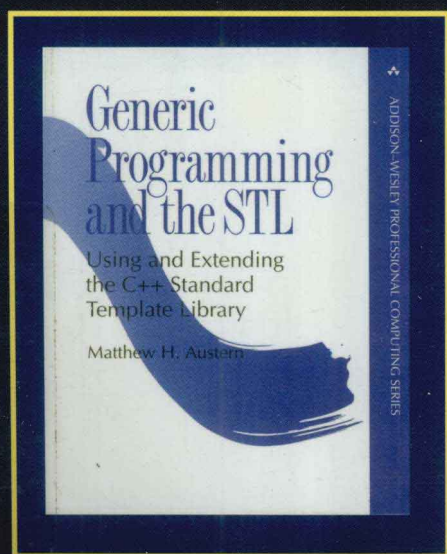
Generic Programming and the STL

Using and Extending the C++ Standard Template Library

泛型编程与 STL

(影印版)

[美] Matthew H. Austern 著



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

Copyright © 1999 Addison Wesley Longman, Inc.

Original English Language Edition Published by Addison Wesley Longman, Inc.

All rights reserved.

Reprinting edition published by PEARSON EDUCATION ASIA LTD and CHINA
ELECTRIC POWER PRESS, Copyright © 2003.

本书影印版由 Pearson Education 授权中国电力出版社在中国境内（香港、澳门特别行政区和台湾
地区除外）独家出版、发行。

未经出版者书面许可，不得以任何方式复制或抄袭本书的任何部分。

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

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

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

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

版权所有 翻印必究

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 $O(N)$.

*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

Preface	xv
Part I Introduction to Generic Programming	1
Chapter 1 A Tour of the STL	3
1.1 A Simple Example	3
1.2 Summary	7
Chapter 2 Algorithms and Ranges	9
2.1 Linear Search	9
2.1.1 Linear Search in C	10
2.1.2 Ranges	12
2.1.3 Linear Search in C++	13
2.2 Concepts and Modeling	16
2.3 Iterators	19
2.3.1 Input Iterators	20
2.3.2 Output Iterators	22
2.3.3 Forward Iterators	24
2.3.4 Bidirectional Iterators	27
2.3.5 Random Access Iterators	28
2.4 Refinement	29
2.5 Summary	31
Chapter 3 More about Iterators	33
3.1 Iterator Traits and Associated Types	33
3.1.1 Value Types	33
3.1.2 Difference Type	36
3.1.3 Reference and Pointer Types	37
3.1.4 Dispatching Algorithms and Iterator Tags	38
3.1.5 Putting It All Together	41
3.1.6 Iterator Traits without <code>iterator_traits</code>	43
3.2 Defining New Components	44
3.2.1 Iterator Adaptors	46
3.2.2 Advice for Defining an Iterator	47

3.2.3	Advice for Defining an Algorithm	47
3.3	Summary	48
Chapter 4	Function Objects	49
4.1	Generalizing Linear Search	49
4.2	Function Object Concepts	52
4.2.1	Unary and Binary Function Objects	52
4.2.2	Predicates and Binary Predicates	53
4.2.3	Associated Types	54
4.3	Function Object Adaptors	56
4.4	Predefined Function Objects	58
4.5	Summary	58
Chapter 5	Containers	59
5.1	A Simple Container	59
5.1.1	An Array Class	60
5.1.2	How It Works	63
5.1.3	Finishing Touches	63
5.2	Container Concepts	67
5.2.1	Containment of Elements	68
5.2.2	Iterators	68
5.2.3	The Hierarchy of Containers	70
5.2.4	The Trivial Container	71
5.3	Variable Size Container Concepts	72
5.3.1	Sequences	73
5.3.2	Associative Containers	75
5.3.3	Allocators	78
5.4	Summary	78
5.4.1	Which Container Should You Use?	78
5.4.2	Defining Your Own Container	79
Part II	Reference Manual: STL Concepts	81
Chapter 6	Basic Concepts	83
6.1	Assignable	83
6.2	Default Constructible	84
6.3	Equality Comparable	85
6.4	Ordering	86
6.4.1	LessThan Comparable	86
6.4.2	Strict Weakly Comparable	88
Chapter 7	Iterators	91
7.1	Trivial Iterator	91
7.2	Input Iterator	94
7.3	Output Iterator	96
7.4	Forward Iterator	100

7.5	Bidirectional Iterator	102
7.6	Random Access Iterator	103
Chapter 8	Function Objects	109
8.1	Basic Function Objects	110
8.1.1	Generator	110
8.1.2	Unary Function	111
8.1.3	Binary Function	112
8.2	Adaptable Function Objects	113
8.2.1	Adaptable Generator	113
8.2.2	Adaptable Unary Function	114
8.2.3	Adaptable Binary Function	115
8.3	Predicates	116
8.3.1	Predicate	116
8.3.2	Binary Predicate	117
8.3.3	Adaptable Predicate	118
8.3.4	Adaptable Binary Predicate	119
8.3.5	Strict Weak Ordering	119
8.4	Specialized Concepts	122
8.4.1	Random Number Generator	122
8.4.2	Hash Function	123
Chapter 9	Containers	125
9.1	General Container Concepts	125
9.1.1	Container	125
9.1.2	Forward Container	131
9.1.3	Reversible Container	133
9.1.4	Random Access Container	135
9.2	Sequences	136
9.2.1	Sequence	136
9.2.2	Front Insertion Sequence	141
9.2.3	Back Insertion Sequence	143
9.3	Associative Containers	145
9.3.1	Associative Container	145
9.3.2	Unique Associative Container	149
9.3.3	Multiple Associative Container	152
9.3.4	Simple Associative Container	153
9.3.5	Pair Associative Container	155
9.3.6	Sorted Associative Container	156
9.3.7	Hashed Associative Container	161
9.4	Allocator	166

Part III Reference Manual: Algorithms and Classes	173
Chapter 10 Basic Components	175
10.1 pair	175
10.2 Iterator Primitives	177
10.2.1 iterator_traits	177
10.2.2 Iterator Tag Classes	179
10.2.3 distance	181
10.2.4 advance	183
10.2.5 Iterator Base Class	185
10.3 allocator	187
10.4 Memory Management Primitives	189
10.4.1 construct	189
10.4.2 destroy	190
10.4.3 uninitialized_copy	192
10.4.4 uninitialized_fill	194
10.4.5 uninitialized_fill_n	195
10.5 Temporary Buffers	196
10.5.1 get_temporary_buffer	197
10.5.2 return_temporary_buffer	198
Chapter 11 Nonmutating Algorithms	199
11.1 Linear Search	199
11.1.1 find	199
11.1.2 find_if	200
11.1.3 adjacent_find	202
11.1.4 find_first_of	204
11.2 Subsequence Matching	206
11.2.1 search	206
11.2.2 find_end	209
11.2.3 search_n	211
11.3 Counting Elements	214
11.3.1 count	214
11.3.2 count_if	216
11.4 for_each	218
11.5 Comparing Two Ranges	220
11.5.1 equal	220
11.5.2 mismatch	222
11.5.3 lexicographical_compare	225
11.6 Minimum and Maximum	227
11.6.1 min	227
11.6.2 max	228
11.6.3 min_element	229
11.6.4 max_element	231

Chapter 12 Basic Mutating Algorithms	233
12.1 Copying Ranges	233
12.1.1 copy	233
12.1.2 copy_backward	236
12.2 Swapping Elements	237
12.2.1 swap	237
12.2.2 iter_swap	238
12.2.3 swap_ranges	239
12.3 transform	240
12.4 Replacing Elements	243
12.4.1 replace	243
12.4.2 replace_if	244
12.4.3 replace_copy	246
12.4.4 replace_copy_if	248
12.5 Filling Ranges	249
12.5.1 fill	249
12.5.2 fill_n	250
12.5.3 generate	251
12.5.4 generate_n	252
12.6 Removing Elements	253
12.6.1 remove	253
12.6.2 remove_if	255
12.6.3 remove_copy	256
12.6.4 remove_copy_if	258
12.6.5 unique	259
12.6.6 unique_copy	262
12.7 Permuting Algorithms	264
12.7.1 reverse	264
12.7.2 reverse_copy	265
12.7.3 rotate	266
12.7.4 rotate_copy	268
12.7.5 next_permutation	269
12.7.6 prev_permutation	271
12.8 Partitions	273
12.8.1 partition	273
12.8.2 stable_partition	274
12.9 Random Shuffling and Sampling	275
12.9.1 random_shuffle	276
12.9.2 random_sample	277
12.9.3 random_sample_n	279
12.10 Generalized Numeric Algorithms	281
12.10.1 accumulate	281
12.10.2 inner_product	283
12.10.3 partial_sum	285
12.10.4 adjacent_difference	287

Chapter 13	Sorting and Searching	291
13.1	Sorting Ranges	291
13.1.1	sort	292
13.1.2	stable_sort	294
13.1.3	partial_sort	297
13.1.4	partial_sort_copy	300
13.1.5	nth_element	301
13.1.6	is_sorted	303
13.2	Operations on Sorted Ranges	305
13.2.1	Binary Search	305
13.2.1.1	binary_search	306
13.2.1.2	lower_bound	308
13.2.1.3	upper_bound	310
13.2.1.4	equal_range	313
13.2.2	Merging Two Sorted Ranges	316
13.2.2.1	merge	316
13.2.2.2	inplace_merge	318
13.2.3	Set Operations on Sorted Ranges	320
13.2.3.1	includes	321
13.2.3.2	set_union	324
13.2.3.3	set_intersection	327
13.2.3.4	set_difference	330
13.2.3.5	set_symmetric_difference	333
13.3	Heap Operations	336
13.3.1	make_heap	336
13.3.2	push_heap	338
13.3.3	pop_heap	339
13.3.4	sort_heap	342
13.3.5	is_heap	343
Chapter 14	Iterator Classes	345
14.1	Insert Iterators	345
14.1.1	front_insert_iterator	345
14.1.2	back_insert_iterator	348
14.1.3	insert_iterator	351
14.2	Stream Iterators	354
14.2.1	istream_iterator	354
14.2.2	ostream_iterator	357
14.2.3	istreambuf_iterator	359
14.2.4	ostreambuf_iterator	362
14.3	reverse_iterator	363
14.4	raw_storage_iterator	368

Chapter 15	Function Object Classes	371
15.1	Function Object Base Classes	371
15.1.1	unary_function	371
15.1.2	binary_function	372
15.2	Arithmetic Operations	373
15.2.1	plus	373
15.2.2	minus	375
15.2.3	multiplies	376
15.2.4	divides	378
15.2.5	modulus	379
15.2.6	negate	380
15.3	Comparisons	382
15.3.1	equal_to	382
15.3.2	not_equal_to	383
15.3.3	less	384
15.3.4	greater	386
15.3.5	less_equal	387
15.3.6	greater_equal	388
15.4	Logical Operations	390
15.4.1	logical_and	390
15.4.2	logical_or	391
15.4.3	logical_not	393
15.5	Identity and Projection	394
15.5.1	identity	394
15.5.2	project1st	395
15.5.3	project2nd	397
15.5.4	select1st	398
15.5.5	select2nd	399
15.6	Specialized Function Objects	400
15.6.1	hash	400
15.6.2	subtractive_rng	402
15.7	Member Function Adaptors	403
15.7.1	mem_fun_t	404
15.7.2	mem_fun_ref_t	406
15.7.3	mem_fun1_t	408
15.7.4	mem_fun1_ref_t	410
15.7.5	const_mem_fun_t	412
15.7.6	const_mem_fun_ref_t	414
15.7.7	const_mem_fun1_t	416
15.7.8	const_mem_fun1_ref_t	418
15.8	Other Adaptors	421
15.8.1	binder1st	421
15.8.2	binder2nd	422
15.8.3	pointer_to_unary_function	424
15.8.4	pointer_to_binary_function	426
15.8.5	unary_negate	428

15.8.6	<code>binary_negate</code>	429
15.8.7	<code>unary_compose</code>	431
15.8.8	<code>binary_compose</code>	433
Chapter 16	Container Classes	435
16.1	Sequences	435
16.1.1	<code>vector</code>	435
16.1.2	<code>list</code>	441
16.1.3	<code>slist</code>	448
16.1.4	<code>deque</code>	455
16.2	Associative Containers	460
16.2.1	<code>set</code>	461
16.2.2	<code>map</code>	466
16.2.3	<code>multiset</code>	473
16.2.4	<code>multimap</code>	478
16.2.5	<code>hash_set</code>	484
16.2.6	<code>hash_map</code>	488
16.2.7	<code>hash_multiset</code>	494
16.2.8	<code>hash_multimap</code>	499
16.3	Container Adaptors	504
16.3.1	<code>stack</code>	505
16.3.2	<code>queue</code>	507
16.3.3	<code>priority_queue</code>	510
Appendix A	Portability and Standardization	515
A.1	Language Changes	516
A.1.1	The Template Compilation Model	516
A.1.2	Default Template Parameters	517
A.1.3	Member Templates	518
A.1.4	Partial Specialization	519
A.1.5	New Keywords	521
A.2	Library Changes	524
A.2.1	Allocators	524
A.2.2	Container Adaptors	525
A.2.3	Minor Library Changes	526
A.3	Naming and Packaging	527
Bibliography	531
Index	535