

The GNU C++ Library Manual

Paolo Carlini, Phil Edwards, Doug Gregor, Benjamin Kosnik, Dhruv Matani, Jason Merrill,
Mark Mitchell, Nathan Myers, Felix Natter, Stefan Olsson, Johannes Singler, Ami Tavory,
and Jonathan Wakely

Copyright © 2000, 2001, 2002, 2003, 2004, 2005, 2006, 2007, 2008, 2009, 2010, 2011, 2012, 2013, 2014, 2015, 2016, 2017, 2018, 2019, 2020 [FSF](#)

COLLABORATORS

	<i>TITLE :</i> The GNU C++ Library Manual		
<i>ACTION</i>	<i>NAME</i>	<i>DATE</i>	<i>SIGNATURE</i>
WRITTEN BY	Paolo Carlini, Phil Edwards, Doug Gregor, Benjamin Kosnik, Dhruv Matani, Jason Merrill, Mark Mitchell, Nathan Myers, Felix Natter, Stefan Olsson, Johannes Singler, Ami Tavory, and Jonathan Wakely	October 19, 2025	

REVISION HISTORY

NUMBER	DATE	DESCRIPTION	NAME

Contents

I	Introduction	1
1	Status	2
1.1	Implementation Status	2
1.1.1	C++ 1998/2003	2
1.1.1.1	Implementation Status	2
1.1.1.2	Implementation Specific Behavior	2
1.1.2	C++ 2011	4
1.1.2.1	Implementation Specific Behavior	4
1.1.3	C++ 2014	7
1.1.3.1	Implementation Specific Behavior	7
1.1.3.1.1	Filesystem TS	7
1.1.4	C++ 2017	9
1.1.4.1	Implementation Specific Behavior	12
1.1.4.1.1	Parallelism 2 TS	13
1.1.5	C++ 2020	14
1.1.5.1	Implementation Specific Behavior	17
1.1.6	C++ 2023	17
1.1.7	C++ TR1	17
1.1.7.1	Implementation Specific Behavior	17
1.1.8	C++ TR 24733	20
1.1.9	C++ IS 29124	20
1.1.9.1	Implementation Specific Behavior	20
1.2	License	20
1.2.1	The Code: GPL	23
1.2.2	The Documentation: GPL, FDL	24
1.3	Bugs	24
1.3.1	Implementation Bugs	24
1.3.2	Standard Bugs	24

2	Setup	31
2.1	Prerequisites	31
2.2	Configure	33
2.3	Make	36
3	Using	37
3.1	Command Options	37
3.2	Headers	37
3.2.1	Header Files	37
3.2.2	Mixing Headers	40
3.2.3	The C Headers and <code>namespace std</code>	43
3.2.4	Precompiled Headers	43
3.3	Macros	44
3.4	Dual ABI	45
3.4.1	Troubleshooting	46
3.5	Namespaces	46
3.5.1	Available Namespaces	46
3.5.2	<code>namespace std</code>	47
3.5.3	Using Namespace Composition	47
3.6	Linking	47
3.6.1	Almost Nothing	47
3.6.2	Finding Dynamic or Shared Libraries	48
3.6.3	Experimental Library Extensions	49
3.7	Concurrency	49
3.7.1	Prerequisites	49
3.7.2	Thread Safety	50
3.7.3	Atomics	51
3.7.4	IO	51
3.7.4.1	Structure	51
3.7.4.2	Defaults	52
3.7.4.3	Future	52
3.7.4.4	Alternatives	52
3.7.5	Containers	52
3.8	Exceptions	53
3.8.1	Exception Safety	53
3.8.2	Exception Neutrality	53
3.8.3	Memory allocation	54
3.8.4	Doing without	55
3.8.5	Compatibility	56

3.8.5.1	With C	56
3.8.5.2	With POSIX thread cancellation	56
3.8.6	Bibliography	56
3.9	Debugging Support	57
3.9.1	Using g++	57
3.9.2	Debug Mode	57
3.9.3	Tracking uncaught exceptions	57
3.9.4	Memory Leak Hunting	57
3.9.4.1	Non-memory leaks in Pool and MT allocators	58
3.9.5	Data Race Hunting	58
3.9.6	Using gdb	59
3.9.7	Debug Versions of Library Binary Files	59
3.9.8	Compile Time Checking	60

II Standard Contents 61

4 Support 62

4.1	Types	62
4.1.1	Fundamental Types	62
4.1.2	Numeric Properties	63
4.1.3	NULL	63
4.2	Dynamic Memory	64
4.2.1	Additional Notes	65
4.3	Termination	65
4.3.1	Termination Handlers	65
4.3.2	Verbose Terminate Handler	66

5 Diagnostics 68

5.1	Exceptions	68
5.1.1	API Reference	68
5.1.2	Adding Data to <code>exception</code>	68
5.2	Use of <code>errno</code> by the library	68
5.3	Concept Checking	69

6 Utilities 70

6.1	Functors	70
6.2	Pairs	70
6.3	Memory	71
6.3.1	Allocators	71
6.3.1.1	Requirements	71

6.3.1.2	Design Issues	71
6.3.1.3	Implementation	72
6.3.1.3.1	Interface Design	72
6.3.1.3.2	Selecting Default Allocation Policy	72
6.3.1.3.3	Disabling Memory Caching	72
6.3.1.4	Using a Specific Allocator	73
6.3.1.5	Custom Allocators	73
6.3.1.6	Extension Allocators	73
6.3.1.7	Bibliography	74
6.3.2	auto_ptr	74
6.3.2.1	Limitations	74
6.3.2.2	Use in Containers	75
6.3.3	shared_ptr	75
6.3.3.1	Requirements	75
6.3.3.2	Design Issues	75
6.3.3.3	Implementation	76
6.3.3.3.1	Class Hierarchy	76
6.3.3.3.2	Thread Safety	76
6.3.3.3.3	Selecting Lock Policy	77
6.3.3.3.4	Related functions and classes	77
6.3.3.4	Use	78
6.3.3.4.1	Examples	78
6.3.3.4.2	Unresolved Issues	78
6.3.3.5	Acknowledgments	78
6.3.3.6	Bibliography	78
6.4	Traits	78
7	Strings	79
7.1	String Classes	79
7.1.1	Simple Transformations	79
7.1.2	Case Sensitivity	80
7.1.3	Arbitrary Character Types	81
7.1.4	Tokenizing	81
7.1.5	Shrink to Fit	83
7.1.6	CString (MFC)	83

8	Localization	85
8.1	Locales	85
8.1.1	locale	85
8.1.1.1	Requirements	85
8.1.1.2	Design	85
8.1.1.3	Implementation	86
8.1.1.3.1	Interacting with "C" locales	86
8.1.1.4	Future	91
8.1.1.5	Bibliography	91
8.2	Facets	92
8.2.1	ctype	92
8.2.1.1	Implementation	92
8.2.1.1.1	Specializations	92
8.2.1.2	Future	92
8.2.1.3	Bibliography	92
8.2.2	codecvt	93
8.2.2.1	Requirements	93
8.2.2.2	Design	93
8.2.2.2.1	wchar_t Size	93
8.2.2.2.2	Support for Unicode	94
8.2.2.2.3	Other Issues	94
8.2.2.3	Implementation	95
8.2.2.4	Use	96
8.2.2.5	Future	96
8.2.2.6	Bibliography	97
8.2.3	messages	97
8.2.3.1	Requirements	97
8.2.3.2	Design	98
8.2.3.3	Implementation	98
8.2.3.3.1	Models	98
8.2.3.3.2	The GNU Model	99
8.2.3.4	Use	99
8.2.3.5	Future	100
8.2.3.6	Bibliography	100

9	Containers	102
9.1	Sequences	102
9.1.1	list	102
9.1.1.1	list::size() is O(n)	102
9.2	Associative	102
9.2.1	Insertion Hints	102
9.2.2	bitset	103
9.2.2.1	Size Variable	103
9.2.2.2	Type String	104
9.3	Unordered Associative	105
9.3.1	Insertion Hints	105
9.3.2	Hash Code	105
9.3.2.1	Hash Code Caching Policy	105
9.4	Interacting with C	106
9.4.1	Containers vs. Arrays	106
10	Iterators	108
10.1	Predefined	108
10.1.1	Iterators vs. Pointers	108
10.1.2	One Past the End	108
11	Algorithms	110
11.1	Mutating	110
11.1.1	swap	110
11.1.1.1	Specializations	110
12	Numerics	111
12.1	Complex	111
12.1.1	complex Processing	111
12.2	Generalized Operations	111
12.3	Interacting with C	112
12.3.1	Numerics vs. Arrays	112
12.3.2	C99	112
13	Input and Output	113
13.1	Iostream Objects	113
13.2	Stream Buffers	114
13.2.1	Derived streambuf Classes	114
13.2.2	Buffering	115
13.3	Memory Based Streams	116

13.3.1	Compatibility With <code>strstream</code>	116
13.4	File Based Streams	117
13.4.1	Copying a File	117
13.4.2	Binary Input and Output	117
13.5	Interacting with C	118
13.5.1	Using <code>FILE*</code> and file descriptors	118
13.5.2	Performance	119
14	Atomics	120
14.1	API Reference	120
15	Concurrency	121
15.1	API Reference	121
III	Extensions	122
16	Compile Time Checks	124
17	Debug Mode	125
17.1	Intro	125
17.2	Semantics	125
17.3	Using	126
17.3.1	Using the Debug Mode	126
17.3.2	Using a Specific Debug Container	127
17.4	Design	127
17.4.1	Goals	127
17.4.2	Methods	128
17.4.2.1	The Wrapper Model	128
17.4.2.1.1	Safe Iterators	129
17.4.2.1.2	Safe Sequences (Containers)	129
17.4.2.2	Precondition Checking	130
17.4.2.3	Release- and debug-mode coexistence	130
17.4.2.3.1	Compile-time coexistence of release- and debug-mode components	130
17.4.2.3.2	Link- and run-time coexistence of release- and debug-mode components	131
17.4.2.3.3	Alternatives for Coexistence	132
17.4.3	Other Implementations	133

18 Parallel Mode	134
18.1 Intro	134
18.2 Semantics	135
18.3 Using	136
18.3.1 Prerequisite Compiler Flags	136
18.3.2 Using Parallel Mode	136
18.3.3 Using Specific Parallel Components	136
18.4 Design	138
18.4.1 Interface Basics	138
18.4.2 Configuration and Tuning	138
18.4.2.1 Setting up the OpenMP Environment	138
18.4.2.2 Compile Time Switches	139
18.4.2.3 Run Time Settings and Defaults	139
18.4.3 Implementation Namespaces	140
18.5 Testing	140
18.6 Bibliography	141
19 The <code>mt_allocator</code>	142
19.1 Intro	142
19.2 Design Issues	142
19.2.1 Overview	142
19.3 Implementation	143
19.3.1 Tunable Parameters	143
19.3.2 Initialization	144
19.3.3 Deallocation Notes	144
19.4 Single Thread Example	145
19.5 Multiple Thread Example	146
20 The <code>bitmap_allocator</code>	148
20.1 Design	148
20.2 Implementation	148
20.2.1 Free List Store	148
20.2.2 Super Block	149
20.2.3 Super Block Data Layout	149
20.2.4 Maximum Wasted Percentage	150
20.2.5 <code>allocate</code>	150
20.2.6 <code>deallocate</code>	151
20.2.7 Questions	151
20.2.7.1 1	151
20.2.7.2 2	151
20.2.7.3 3	151
20.2.8 Locality	152
20.2.9 Overhead and Grow Policy	152

21 Policy-Based Data Structures	153
21.1 Intro	153
21.1.1 Performance Issues	153
21.1.1.1 Associative	153
21.1.1.2 Priority Que	154
21.1.2 Goals	154
21.1.2.1 Associative	154
21.1.2.1.1 Policy Choices	154
21.1.2.1.2 Underlying Data Structures	155
21.1.2.1.3 Iterators	157
21.1.2.1.3.1 Using Point Iterators for Range Operations	158
21.1.2.1.3.2 Cost to Point Iterators to Enable Range Operations	158
21.1.2.1.3.3 Invalidation Guarantees	159
21.1.2.1.4 Functional	161
21.1.2.1.4.1 erase	161
21.1.2.1.4.2 split and join	162
21.1.2.1.4.3 insert	162
21.1.2.1.4.4 operator== and operator<=	162
21.1.2.2 Priority Queues	162
21.1.2.2.1 Policy Choices	162
21.1.2.2.2 Underlying Data Structures	163
21.1.2.2.3 Binary Heaps	164
21.2 Using	165
21.2.1 Prerequisites	165
21.2.2 Organization	165
21.2.3 Tutorial	166
21.2.3.1 Basic Use	166
21.2.3.2 Configuring via Template Parameters	168
21.2.3.3 Querying Container Attributes	168
21.2.3.4 Point and Range Iteration	169
21.2.4 Examples	170
21.2.4.1 Intermediate Use	170
21.2.4.2 Querying with <code>container_traits</code>	170
21.2.4.3 By Container Method	171
21.2.4.3.1 Hash-Based	171
21.2.4.3.1.1 size Related	171
21.2.4.3.1.2 Hashing Function Related	171
21.2.4.3.2 Branch-Based	171
21.2.4.3.2.1 split or join Related	171

21.2.4.3.2.2	Node Invariants	171
21.2.4.3.2.3	trie	171
21.2.4.3.3	Priority Queues	171
21.3	Design	172
21.3.1	Concepts	172
21.3.1.1	Null Policy Classes	172
21.3.1.2	Map and Set Semantics	172
21.3.1.2.1	Distinguishing Between Maps and Sets	172
21.3.1.2.2	Alternatives to <code>std::multiset</code> and <code>std::multimap</code>	173
21.3.1.3	Iterator Semantics	176
21.3.1.3.1	Point and Range Iterators	176
21.3.1.3.2	Distinguishing Point and Range Iterators	176
21.3.1.3.3	Invalidation Guarantees	177
21.3.1.4	Genericity	178
21.3.1.4.1	Tag	179
21.3.1.4.2	Traits	180
21.3.2	By Container	180
21.3.2.1	hash	180
21.3.2.1.1	Interface	180
21.3.2.1.2	Details	181
21.3.2.1.2.1	Hash Policies	181
21.3.2.1.2.2	General	181
21.3.2.1.2.3	Range Hashing	183
21.3.2.1.2.4	Ranged Hash	184
21.3.2.1.2.5	Implementation	184
21.3.2.1.2.6	Range-Hashing and Ranged-Hashes in Collision-Chaining Tables	185
21.3.2.1.2.7	Probing tables	186
21.3.2.1.2.8	Pre-Defined Policies	186
21.3.2.1.2.9	Resize Policies	188
21.3.2.1.2.10	General	188
21.3.2.1.2.11	Size Policies	188
21.3.2.1.2.12	Trigger Policies	188
21.3.2.1.2.13	Implementation	189
21.3.2.1.2.14	Decomposition	189
21.3.2.1.2.15	Predefined Policies	194
21.3.2.1.2.16	Controlling Access to Internals	194
21.3.2.1.2.17	Policy Interactions	194
21.3.2.1.2.18	probe/size/trigger	195
21.3.2.1.2.19	hash/trigger	195

21.3.2.1.2.20	equivalence functors/storing hash values/hash	195
21.3.2.1.2.21	size/load-check trigger	195
21.3.2.2	tree	195
21.3.2.2.1	Interface	195
21.3.2.2.2	Details	196
21.3.2.2.2.1	Node Invariants	196
21.3.2.2.2.2	Node Iterators	199
21.3.2.2.2.3	Node Updator	199
21.3.2.2.2.4	Split and Join	204
21.3.2.3	Trie	204
21.3.2.3.1	Interface	204
21.3.2.3.2	Details	205
21.3.2.3.2.1	Element Access Traits	205
21.3.2.3.2.2	Node Invariants	206
21.3.2.3.2.3	Split and Join	207
21.3.2.4	List	207
21.3.2.4.1	Interface	207
21.3.2.4.2	Details	208
21.3.2.4.2.1	Underlying Data Structure	208
21.3.2.4.2.2	Policies	209
21.3.2.4.2.3	Use in Multimaps	210
21.3.2.5	Priority Queue	210
21.3.2.5.1	Interface	210
21.3.2.5.2	Details	211
21.3.2.5.2.1	Iterators	211
21.3.2.5.2.2	Underlying Data Structure	212
21.3.2.5.2.3	Traits	213
21.4	Testing	214
21.4.1	Regression	214
21.4.2	Performance	214
21.4.2.1	Hash-Based	214
21.4.2.1.1	Text <code>find</code>	214
21.4.2.1.1.1	Description	214
21.4.2.1.1.2	Results	215
21.4.2.1.1.3	Observations	216
21.4.2.1.2	Integer <code>find</code>	216
21.4.2.1.2.1	Description	216
21.4.2.1.2.2	Results	216
21.4.2.1.2.3	Observations	219

21.4.2.1.3	Integer Subscript <code>find</code>	219
21.4.2.1.3.1	Description	219
21.4.2.1.3.2	Results	219
21.4.2.1.3.3	Observations	221
21.4.2.1.4	Integer Subscript <code>insert</code>	221
21.4.2.1.4.1	Description	221
21.4.2.1.4.2	Results	222
21.4.2.1.4.3	Observations	224
21.4.2.1.5	Integer <code>find</code> with Skewed-Distribution	224
21.4.2.1.5.1	Description	224
21.4.2.1.5.2	Results	224
21.4.2.1.5.3	Observations	225
21.4.2.1.6	Erase Memory Use	226
21.4.2.1.6.1	Description	226
21.4.2.1.6.2	Results	226
21.4.2.1.6.3	Observations	227
21.4.2.2	Branch-Based	227
21.4.2.2.1	Text <code>insert</code>	227
21.4.2.2.1.1	Description	227
21.4.2.2.1.2	Results	227
21.4.2.2.1.3	Observations	230
21.4.2.2.2	Text <code>find</code>	230
21.4.2.2.2.1	Description	230
21.4.2.2.2.2	Results	230
21.4.2.2.2.3	Observations	231
21.4.2.2.3	Text <code>find</code> with Locality-of-Reference	232
21.4.2.2.3.1	Description	232
21.4.2.2.3.2	Results	232
21.4.2.2.3.3	Observations	233
21.4.2.2.4	<code>split</code> and <code>join</code>	233
21.4.2.2.4.1	Description	233
21.4.2.2.4.2	Results	233
21.4.2.2.4.3	Observations	234
21.4.2.2.5	Order-Statistics	235
21.4.2.2.5.1	Description	235
21.4.2.2.5.2	Results	235
21.4.2.2.5.3	Observations	236
21.4.2.3	Multimap	236
21.4.2.3.1	Text <code>find</code> with Small Secondary-to-Primary Key Ratios	236

21.4.2.3.1.1	Description	236
21.4.2.3.1.2	Results	236
21.4.2.3.1.3	Observations	239
21.4.2.3.2	Text <code>find</code> with Large Secondary-to-Primary Key Ratios	239
21.4.2.3.2.1	Description	239
21.4.2.3.2.2	Results	239
21.4.2.3.2.3	Observations	242
21.4.2.3.3	Text <code>insert</code> with Small Secondary-to-Primary Key Ratios	242
21.4.2.3.3.1	Description	242
21.4.2.3.3.2	Results	242
21.4.2.3.3.3	Observations	245
21.4.2.3.4	Text <code>insert</code> with Small Secondary-to-Primary Key Ratios	245
21.4.2.3.4.1	Description	245
21.4.2.3.4.2	Results	245
21.4.2.3.4.3	Observations	248
21.4.2.3.5	Text <code>insert</code> with Small Secondary-to-Primary Key Ratios Memory Use	248
21.4.2.3.5.1	Description	248
21.4.2.3.5.2	Results	248
21.4.2.3.5.3	Observations	251
21.4.2.3.6	Text <code>insert</code> with Small Secondary-to-Primary Key Ratios Memory Use	251
21.4.2.3.6.1	Description	251
21.4.2.3.6.2	Results	251
21.4.2.3.6.3	Observations	254
21.4.2.4	Priority Queue	254
21.4.2.4.1	Text <code>push</code>	254
21.4.2.4.1.1	Description	254
21.4.2.4.1.2	Results	254
21.4.2.4.1.3	Observations	256
21.4.2.4.2	Text <code>push</code> and <code>pop</code>	256
21.4.2.4.2.1	Description	256
21.4.2.4.2.2	Results	256
21.4.2.4.2.3	Observations	258
21.4.2.4.3	Integer <code>push</code>	258
21.4.2.4.3.1	Description	258
21.4.2.4.3.2	Results	259
21.4.2.4.3.3	Observations	260
21.4.2.4.4	Integer <code>push</code>	260
21.4.2.4.4.1	Description	260
21.4.2.4.4.2	Results	260

21.4.2.4.4.3	Observations	261
21.4.2.4.5	Text pop Memory Use	262
21.4.2.4.5.1	Description	262
21.4.2.4.5.2	Results	262
21.4.2.4.5.3	Observations	263
21.4.2.4.6	Text join	263
21.4.2.4.6.1	Description	263
21.4.2.4.6.2	Results	263
21.4.2.4.6.3	Observations	264
21.4.2.4.7	Text modify Up	265
21.4.2.4.7.1	Description	265
21.4.2.4.7.2	Results	265
21.4.2.4.7.3	Observations	266
21.4.2.4.8	Text modify Down	267
21.4.2.4.8.1	Description	267
21.4.2.4.8.2	Results	267
21.4.2.4.8.3	Observations	268
21.4.2.5	Observations	269
21.4.2.5.1	Associative	269
21.4.2.5.1.1	Underlying Data-Structure Families	269
21.4.2.5.1.2	Hash-Based Containers	269
21.4.2.5.1.3	Hash Policies	269
21.4.2.5.1.4	Branch-Based Containers	269
21.4.2.5.1.5	Mapping-Semantics	270
21.4.2.5.2	Priority_Queue	271
21.4.2.5.2.1	Complexity	271
21.4.2.5.2.2	Amortized push and pop operations	272
21.4.2.5.2.3	Graph Algorithms	272
21.5	Acknowledgments	272
21.6	Bibliography	273
22	HP/SGI Extensions	275
22.1	Backwards Compatibility	275
22.2	Deprecated	275
23	Utilities	277
24	Algorithms	278
25	Numerics	279

26 Iterators	280
27 Input and Output	281
27.1 Derived filebufs	281
28 Demangling	282
29 Concurrency	283
29.1 Design	283
29.1.1 Interface to Locks and Mutexes	283
29.1.2 Interface to Atomic Functions	283
29.2 Implementation	284
29.2.1 Using Built-in Atomic Functions	284
29.2.2 Thread Abstraction	285
29.3 Use	286
 IV Appendices	 287
A Contributing	288
A.1 Contributor Checklist	288
A.1.1 Reading	288
A.1.2 Assignment	288
A.1.3 Getting Sources	288
A.1.4 Submitting Patches	289
A.2 Directory Layout and Source Conventions	289
A.3 Coding Style	290
A.3.1 Bad Identifiers	290
A.3.2 By Example	293
A.4 Design Notes	301
 B Porting and Maintenance	 317
B.1 Configure and Build Hacking	317
B.1.1 Prerequisites	317
B.1.2 Overview	317
B.1.2.1 General Process	317
B.1.2.2 What Comes from Where	318
B.1.3 Configure	318
B.1.3.1 Storing Information in non-AC files (like configure.host)	318
B.1.3.2 Coding and Commenting Conventions	318
B.1.3.3 The acinclude.m4 layout	318
B.1.3.4 GLIBCXX_ENABLE, the <code>--enable maker</code>	320

B.1.3.5	Shared Library Versioning	321
B.1.4	Make	322
B.1.5	Generated files	323
B.2	Writing and Generating Documentation	323
B.2.1	Introduction	323
B.2.2	Generating Documentation	323
B.2.3	Doxygen	324
B.2.3.1	Prerequisites	324
B.2.3.2	Generating the Doxygen Files	324
B.2.3.3	Debugging Generation	325
B.2.3.4	Markup	326
B.2.4	Docbook	327
B.2.4.1	Prerequisites	327
B.2.4.2	Generating the DocBook Files	328
B.2.4.3	Debugging Generation	328
B.2.4.4	Editing and Validation	329
B.2.4.5	File Organization and Basics	329
B.2.4.6	Markup By Example	330
B.3	Porting to New Hardware or Operating Systems	330
B.3.1	Operating System	331
B.3.2	CPU	332
B.3.3	Character Types	332
B.3.4	Thread Safety	335
B.3.5	Numeric Limits	336
B.3.6	Libtool	336
B.4	Testing	336
B.4.1	Test Organization	336
B.4.1.1	Directory Layout	336
B.4.1.2	Naming Conventions	338
B.4.2	Running the Testsuite	338
B.4.2.1	Basic	338
B.4.2.2	Variations	338
B.4.2.3	Permutations	340
B.4.3	Writing a new test case	341
B.4.3.1	Examples of Test Directives	342
B.4.3.2	Directives Specific to Libstdc++ Tests	343
B.4.4	Test Harness and Utilities	343
B.4.4.1	DejaGnu Harness Details	343
B.4.4.2	Utilities	343

B.4.5	Special Topics	344
B.4.5.1	Qualifying Exception Safety Guarantees	344
B.4.5.1.1	Overview	344
B.4.5.1.2	Existing tests	345
B.4.5.1.3	C++11 Requirements Test Sequence Descriptions	345
B.5	ABI Policy and Guidelines	346
B.5.1	The C++ Interface	346
B.5.2	Versioning	346
B.5.2.1	Goals	346
B.5.2.2	History	347
B.5.2.3	Prerequisites	353
B.5.2.4	Configuring	353
B.5.2.5	Checking Active	354
B.5.3	Allowed Changes	354
B.5.4	Prohibited Changes	355
B.5.5	Implementation	355
B.5.6	Testing	356
B.5.6.1	Single ABI Testing	356
B.5.6.2	Multiple ABI Testing	356
B.5.7	Outstanding Issues	358
B.5.8	Bibliography	358
B.6	API Evolution and Deprecation History	358
B.6.1	3.0	358
B.6.2	3.1	358
B.6.3	3.2	359
B.6.4	3.3	359
B.6.5	3.4	359
B.6.6	4.0	360
B.6.7	4.1	360
B.6.8	4.2	360
B.6.9	4.3	361
B.6.10	4.4	362
B.6.11	4.5	362
B.6.12	4.6	363
B.6.13	4.7	363
B.6.14	4.8	363
B.6.15	4.9	363
B.6.16	5	363
B.6.16.1	5.3	364

B.6.17	6	364
B.6.18	7	364
B.6.18.1	7.2	364
B.6.18.2	7.3	364
B.6.19	8	364
B.6.20	9	365
B.6.21	10	365
B.6.22	11	365
B.6.23	12	366
B.6.24	12.3	366
B.6.25	13	366
B.6.26	13.3	366
B.6.27	14	366
B.6.28	15	366
B.6.29	16	366
B.7	Backwards Compatibility	367
B.7.1	First	367
B.7.2	Second	367
B.7.3	Third	367
B.7.3.1	Pre-ISO headers removed	367
B.7.3.2	Extension headers <code>hash_map</code> , <code>hash_set</code> moved to <code>ext</code> or <code>backwards</code>	367
B.7.3.3	No <code>ios::nocreate/ios::noreplace</code> .	369
B.7.3.4	No <code>stream::attach(int fd)</code>	369
B.7.3.5	Support for C++98 dialect.	369
B.7.3.6	Support for C++TR1 dialect.	370
B.7.3.7	Support for C++11 dialect.	371
B.7.3.8	<code>Container::iterator_type</code> is not necessarily <code>Container::value_type*</code>	375
C	Free Software Needs Free Documentation	376
D	GNU General Public License version 3	378
E	GNU Free Documentation License	387
30	Index	393

List of Figures

21.1 Node Invariants	155
21.2 Underlying Associative Data Structures	156
21.3 Range Iteration in Different Data Structures	158
21.4 Point Iteration in Hash Data Structures	159
21.5 Effect of erase in different underlying data structures	160
21.6 Underlying Priority Queue Data Structures	164
21.7 Exception Hierarchy	166
21.8 Non-unique Mapping Standard Containers	174
21.9 Effect of embedded lists in <code>std::multimap</code>	174
21.10 Non-unique Mapping Containers	175
21.11 Point Iterator Hierarchy	177
21.12 Invalidation Guarantee Tags Hierarchy	178
21.13 Container Tag Hierarchy	179
21.14 Hash functions, ranged-hash functions, and range-hashing functions	182
21.15 Insert hash sequence diagram	185
21.16 Insert hash sequence diagram with a null policy	186
21.17 Hash policy class diagram	187
21.18 Balls and bins	188
21.19 Insert resize sequence diagram	190
21.20 Standard resize policy trigger sequence diagram	192
21.21 Standard resize policy size sequence diagram	193
21.22 Tree node invariants	197
21.23 Tree node invalidation	198
21.24 A tree and its update policy	199
21.25 Restoring node invariants	200
21.26 Insert update sequence	201
21.27 Useless update path	203
21.28 A PATRICIA trie	206
21.29 A trie and its update policy	207
21.30 A simple list	208

21.31The counter algorithm 209

21.32Underlying Priority-Queue Data-Structures. 212

21.33Priority-Queue Data-Structure Tags. 213

B.1 Configure and Build File Dependencies 318

List of Tables

1.1	C++ 1998/2003 Implementation Status	3
1.2	C++ 2011 Implementation Status	5
1.3	C++ 2014 Implementation Status	8
1.4	C++ Technical Specifications Implementation Status	9
1.5	C++ 2017 Library Features	10
1.6	C++ 2017 Implementation Status	11
1.7	C++ Technical Specifications Implementation Status	12
1.8	Support for Extended ABI Tags	14
1.9	C++ 2020 Library Features	15
1.10	C++ 2020 Implementation Status	16
1.11	C++ 2023 Library Features	18
1.12	C++ TR1 Implementation Status	19
1.13	C++ TR 24733 Implementation Status	21
1.14	C++ Special Functions Implementation Status	22
3.1	C++ Command Options	38
3.2	C++ 1998 Library Headers	38
3.3	C++ 1998 Library Headers for C Library Facilities	38
3.4	C++ 1998 Deprecated Library Header	38
3.5	C++ 2011 Library Headers	39
3.6	C++ 2011 Library Headers for C Library Facilities	39
3.7	C++ 2014 Library Header	39
3.8	C++ 2017 Library Headers	39
3.9	C++ 2020 Library Headers	39
3.10	C++ 2020 Obsolete Headers	40
3.11	C++ 2023 Library Headers	40
3.12	C++ 2026 Library Headers	40
3.13	File System TS Header	41
3.14	Library Fundamentals TS Headers	41
3.15	Networking TS Headers	41

3.16 C++ TR 1 Library Headers	41
3.17 C++ TR 1 Library Headers for C Library Facilities	41
3.18 C++ TR 24733 Decimal Floating-Point Header	41
3.19 C++ ABI Headers	41
3.20 Extension Headers	42
3.21 Extension Debug Headers	42
3.22 Extension Parallel Headers	42
17.1 Debugging Containers	127
17.2 Debugging Containers C++11	127
18.1 Parallel Algorithms	137
20.1 Bitmap Allocator Memory Map	149
B.1 Doxygen Prerequisites	324
B.2 HTML to Doxygen Markup Comparison	327
B.3 Docbook Prerequisites	327
B.4 HTML to Docbook XML Markup Comparison	330
B.5 Docbook XML Element Use	331
B.6 Extension Allocators	360
B.7 Extension Allocators Continued	360

Part I

Introduction

Chapter 1

Status

1.1 Implementation Status

1.1.1 C++ 1998/2003

1.1.1.1 Implementation Status

This status table is based on the table of contents of ISO/IEC 14882:2003.

This section describes the C++ support in mainline GCC, not in any particular release.

1.1.1.2 Implementation Specific Behavior

The ISO standard defines the following phrase:

[1.3.5] implementation-defined behavior Behavior, for a well-formed program construct and correct data, that depends on the implementation *and that each implementation shall document*.

We do so here, for the C++ library only. Behavior of the compiler, linker, runtime loader, and other elements of "the implementation" are documented elsewhere. Everything listed in Annex B, Implementation Qualities, are also part of the compiler, not the library.

For each entry, we give the section number of the standard, when applicable. This list is probably incomplet and inkorrekt.

1.9 [intro.execution]/11 #3 If `isatty(3)` is true, then interactive stream support is implied.

17.4.4.5 [lib.reentrancy] Non-reentrant functions are probably best discussed in the various sections on multithreading (see above).

18.1 [lib.support.types]/4 The type of `NULL` is described under **Support**.

18.3 [lib.support.start.term]/8 Even though it's listed in the library sections, `libstdc++` has zero control over what the cleanup code hands back to the runtime loader. Talk to the compiler people. :-)

18.4.2.1 [lib.bad.alloc]/5 (`bad_alloc`), *18.5.2 [lib.bad.cast]/5* (`bad_cast`), *18.5.3 [lib.bad.typeid]/5* (`bad_typeid`), *18.6.1 [lib.exception]/8* (`exception`), *18.6.2.1 [lib.bad.exception]/5* (`bad_exception`): The `what()` member function of class `std::exception`, and these other classes publicly derived from it, returns the name of the class, e.g. `"std::bad_alloc"`.

18.5.1 [lib.type.info]/7 The return value of `std::type_info::name()` is the mangled type name. You will need to call `c++filt` and pass the names as command-line parameters to demangle them, or call a **runtime demangler function**.

Section	Description	Status	Comments
<i>18</i>	<i>Language support</i>		
18.1	Types	Y	
18.2	Implementation properties	Y	
18.2.1	Numeric Limits		
18.2.1.1	Class template <code>numeric_limits</code>	Y	
18.2.1.2	<code>numeric_limits</code> members	Y	
18.2.1.3	<code>float_round_style</code>	Y	
18.2.1.4	<code>float_denorm_style</code>	Y	
18.2.1.5	<code>numeric_limits</code> specializations	Y	
18.2.2	C Library	Y	
18.3	Start and termination	Y	
18.4	Dynamic memory management	Y	
18.5	Type identification		
18.5.1	Class <code>type_info</code>	Y	
18.5.2	Class <code>bad_cast</code>	Y	
18.5.3	Class <code>bad_typeid</code>	Y	
18.6	Exception handling		
18.6.1	Class <code>exception</code>	Y	
18.6.2	Violation exception-specifications	Y	
18.6.3	Abnormal termination	Y	
18.6.4	<code>uncaught_exception</code>	Y	
18.7	Other runtime support	Y	
<i>19</i>	<i>Diagnostics</i>		
19.1	Exception classes	Y	
19.2	Assertions	Y	
19.3	Error numbers	Y	
<i>20</i>	<i>General utilities</i>		
20.1	Requirements	Y	
20.2	Utility components		
20.2.1	Operators	Y	
20.2.2	<code>pair</code>	Y	
20.3	Function objects		
20.3.1	Base	Y	
20.3.2	Arithmetic operation	Y	
20.3.3	Comparisons	Y	
20.3.4	Logical operations	Y	
20.3.5	Negators	Y	
20.3.6	Binders	Y	
20.3.7	Adaptors for pointers to functions	Y	
20.3.8	Adaptors for pointers to members	Y	
20.4	Memory		
20.4.1	The default allocator	Y	
20.4.2	Raw storage iterator	Y	
20.4.3	Temporary buffers	Y	
20.4.4	Specialized algorithms	Y	
20.4.4.1	<code>uninitialized_copy</code>	Y	
20.4.4.2	<code>uninitialized_fill</code>	Y	
20.4.4.3	<code>uninitialized_fill_n</code>	Y	
20.4.5	Class template <code>auto_ptr</code>	Y	
20.4.6	C library	Y	
<i>21</i>	<i>Strings</i>		
21.1	Character traits		
21.1.1	Character traits requirements	Y	
21.1.2	traits typedef	Y	
21.1.3	<code>char_traits</code>		

20.1.5 [lib allocator.requirements]/5 "Implementors are encouraged to supply libraries that can accept allocators that encapsulate more general memory models and that support non-equal instances. In such implementations, any requirements imposed on allocators by containers beyond those requirements that appear in Table 32, and the semantics of containers and algorithms when allocator instances compare non-equal, are implementation-defined." There is experimental support for non-equal allocators in the standard containers in C++98 mode. There are no additional requirements on allocators. It is undefined behaviour to swap two containers if their allocators are not equal.

21.1.3.1 [lib.char.traits.specializations.char]/3,4, 21.1.3.2 [lib.char.traits.specializations.wchart]/2, 21.3 [lib.basic.string]/6 *basic_string::iterator*, *basic_string::const_iterator*, 23.* [lib.containers]'s *foo::iterator*, 27.* [lib.input.output]'s *foo::*_type*, others... Nope, these types are called implementation-defined because you shouldn't be taking advantage of their underlying types. Listing them here would defeat the purpose. :-)

21.1.3.1 [lib.char.traits.specializations.char]/5 I don't really know about the `mbstate_t` stuff... see the [codecvrt notes](#) for what does exist.

22.* [lib.localization] Anything and everything we have on locale implementation will be described under [Localization](#).

23.* [lib.containers] All of the containers in this clause define `size_type` as `std::size_t` and `difference_type` as `std::ptrdiff_t`.

26.2.8 [lib.complex.transcendentals]/9 I have no idea what `complex<T>`'s `pow(0,0)` returns.

27.4.2.4 [lib.ios.members.static]/2 Calling `std::ios_base::sync_with_stdio` after I/O has already been performed on the standard stream objects will flush the buffers, and destroy and recreate the underlying buffer instances. Whether or not the previously-written I/O is destroyed in this process depends mostly on the `--enable-libio` choice: for `stdio`, if the written data is already in the `stdio` buffer, the data may be completely safe!

27.6.1.1.2 [lib.istream::sentry], 27.6.2.3 [lib ostream::sentry] The I/O sentry ctor and dtor can perform additional work than the minimum required. We are not currently taking advantage of this yet.

27.7.1.3 [lib.stringbuf.virtuals]/16, 27.8.1.4 [lib.filebuf.virtuals]/10 The effects of `pubsetbuf/setbuf` are described in the [Input and Output](#) chapter.

27.8.1.4 [lib.filebuf.virtuals]/16 Calling `fstream::sync` when a get area exists will... whatever `fflush()` does, I think.

1.1.2 C++ 2011

This table is based on the table of contents of ISO/IEC JTC1 SC22 WG21 Doc No: N3290 Date: 2011-04-11 Final Draft International Standard, Standard for Programming Language C++

In this implementation the `-std=gnu++11` or `-std=c++11` flag must be used to enable language and library features. See [dialect](#) options. The pre-defined symbol `__cplusplus` is used to check for the presence of the required flag. GCC 5.1 was the first release with non-experimental C++11 support, so the API and ABI of features added in C++11 is only stable since that release.

This status table is based on the table of contents of ISO/IEC 14882:2011.

This section describes the C++11 support in mainline GCC, not in any particular release.

1.1.2.1 Implementation Specific Behavior

For behaviour which is also specified by the 1998 and 2003 standards, see [C++ 1998/2003 Implementation Specific Behavior](#). This section only documents behaviour which is new in the 2011 standard.

17.6.5.12 [res.on.exception.handling] There are no implementation-defined exception classes, only standard exception classes (or classes derived from them) will be thrown.

17.6.5.14 [value.error.codes] The `error_category` for errors originating outside the OS, and the possible error code values for each error category, should be documented here.

18.6.2.2 [new.badlength] `what()` returns `"std::bad_array_new_length"`.

20.6.9.1 [allocator.member]/5 Over-aligned types are not supported by `std::allocator`.

20.7.2.2.1 [util.smartptr.shared.const] When a `shared_ptr` constructor fails `bad_alloc` (or types derived from it) will be thrown, or when an allocator is passed to the constructor then any exceptions thrown by the allocator.

Section	Description	Status	Comments
18	<i>Language support</i>		
18.1	General		
18.2	Types	Y	
18.3	Implementation properties		
18.3.2	Numeric Limits		
18.3.2.3	Class template numeric_limits	Y	
18.3.2.4	numeric_limits members	Y	
18.3.2.5	float_round_style	N	
18.3.2.6	float_denorm_style	N	
18.3.2.7	numeric_limits specializations	Y	
18.3.3	C Library	Y	
18.4	Integer types		
18.4.1	Header <stdint> synopsis	Y	
18.5	Start and termination	Partial	C library dependency for quick_exit, at_quick_exit
18.6	Dynamic memory management	Y	
18.7	Type identification		
18.7.1	Class type_info	Y	
18.7.2	Class bad_cast	Y	
18.7.3	Class bad_typeid	Y	
18.8	Exception handling		
18.8.1	Class exception	Y	
18.8.2	Class bad_exception	Y	
18.8.3	Abnormal termination	Y	
18.8.4	uncaught_exception	Y	
18.8.5	Exception Propagation	Y	
18.8.6	nested_exception	Y	
18.9	Initializer lists		
18.9.1	Initializer list constructors	Y	
18.9.2	Initializer list access	Y	
18.9.3	Initializer list range access	Y	
18.10	Other runtime support	Y	
19	<i>Diagnostics</i>		
19.1	General		
19.2	Exception classes	Y	
19.3	Assertions	Y	
19.4	Error numbers	Y	
19.5	System error support		
19.5.1	Class error_category	Y	
19.5.2	Class error_code	Y	
19.5.3	Class error_condition	Y	
19.5.4	Comparison operators	Y	
19.5.5	Class system_error	Y	
20	<i>General utilities</i>		
20.1	General		
20.2	Utility components		
20.2.1	Operators	Y	
20.2.2	swap	Y	
20.2.3	forward/move helpers	Y	
20.2.4	Function template declval	Y	
20.3	Pairs		
20.3.1	In general		
20.3.2	Class template pair	Y	
20.3.3	Specialized algorithms	Y	
20.3.4	Tuple-like access to pair	Y	
20.3.5	Piecewise construction	Y	

20.7.2.0 [util.smartptr.weakptr] `what()` returns "bad_weak_ptr".

20.8.11.1 [func.wrap.badcall] `what()` returns "bad_function_call".

20.8.9.1.3 [func.bind.place]/1 There are 29 placeholders defined and the placeholder types are CopyAssignable.

20.11.7.1 [time.clock.system]/3, /4 Time point values are truncated to `time_t` values. There is no loss of precision for conversions in the other direction.

20.15.7 [meta.trans]/2 `aligned_storage` does not support extended alignment.

21.2.3.2 [char.traits.specializations.char16_t], 21.2.3.3 [char.traits.specializations.char32_t] The types `u16streampos` and `u32streampos` are both synonyms for `fpos<mbstate_t>`. The function `eof` returns `int_type(-1)`. `char_traits<char16_t>` will transform the "noncharacter" `U+FFFF` to `U+FFFD` (REPLACEMENT CHARACTER). This is done to ensure that `to_int_type` never returns the same value as `eof`, which is `U+FFFF`.

22.3.1 [locale] There is one global locale for the whole program, not per-thread.

22.4.5.1.2 [locale.time.get.virtuals], 22.4.5.3.2 [locale.time.put.virtuals] Additional supported formats should be documented here.

22.4.7.1.2 [locale.messages.virtuals] The mapping should be documented here.

23.3.2.1 [array.overview] `array<T, N>::iterator` is `T*` and `array<T, N>::const_iterator` is `const T*`.

23.5.4.2 [unord.map.cnstr], 23.5.5.2 [unord.multimap.cnstr], 23.5.6.2 [unord.set.cnstr], 23.5.7.2 [unord.multiset.cnstr] The default minimal bucket count is 0 for the default constructors, range constructors and initializer-list constructors.

25.3.12 [alg.random.shuffle] The two-argument overload of `random_shuffle` uses `rand` as the source of randomness.

26.5.5 [rand.predef] The type `default_random_engine` is a synonym for `minstd_rand0`.

26.5.6 [rand.device] The `token` parameter of the `random_device` constructor can be used to select a specific source of random bytes. The valid token values are shown in the list below. The default constructor uses the token "default".

"default" Select the first available source from the other entries below. This is the only token that is always valid.

"rand_s" Use the MSVCRT `rand_s` function. This token is only valid for mingw-w64 targets.

"rdseed", "rdrand" or "rdrnd" Use the IA-32 RDSEED or RDRAND instruction to read from an on-chip hardware random number generator. These tokens are only valid for x86 and x86_64 targets when both the assembler and CPU support the corresponding instruction.

"darn" Use the Power ISA-3.0 DARN ("Deliver A Random Number") instruction to read from an on-chip hardware random number generator. This token is only valid for 64-bit powerpc targets when both the assembler and CPU support the corresponding instruction.

"hw", "hardware" Use any available CPU instruction to read from an on-chip hardware random number generator. This is equivalent to trying each of the following and using the first that is supported: "rdseed" "rdrand" "darn"

"arc4random", "getentropy" Use the named C library function, if available on the target.

"/dev/urandom", "/dev/random" Use the named character special file to read random bytes from. These tokens are only valid when the device files are present and readable by the current user.

"mt19937", seed value When no source of nondeterministic random numbers is available a `mersenne_twister_engine` will be used. An integer seed value can be used as the token and will be converted to an unsigned long using `strtoul`. These tokens are only valid when no other source of random bytes is available.

An exception of type `runtime_error` will be thrown if a `random_device` object is constructed with an invalid token, or if it cannot open or read from the source of random bytes.

26.5.8.1 [rand.dist.general] The algorithms used by the distributions should be documented here.

26.8 [c.math] Whether the `rand` function introduces data races depends on the C library as the function is not provided by libstdc++.

27.8.2.1 *[stringbuf.cons]* Whether the sequence pointers are copied by the `basic_stringbuf` move constructor should be documented here.

27.9.1.2 *[filebuf.cons]* Whether the sequence pointers are copied by the `basic_filebuf` move constructor should be documented here.

28.5.1 *[re.synopt]*, 28.5.2 *[re.matchflag]*, 28.5.3 *[re.err]* `syntax_option_type`, `match_flag_type` and `error_type` are unscoped enumeration types.

28.7 *[re.traits]* The `blank` character class corresponds to the `ctype_base::blank` mask.

29.4 *[atomics.lockfree]* The values of the `ATOMIC_XXX_LOCK_FREE` macros depend on the target and cannot be listed here.

30.2.3 *[thread.req.native]/1* `native_handle_type` and `native_handle` are provided. The handle types are defined in terms of the Gthreads abstraction layer, although this is subject to change at any time. Any use of `native_handle` is inherently non-portable and not guaranteed to work between major releases of GCC.

thread The native handle type is a typedef for `__gthread_t` i.e. `pthread_t` when GCC is configured with the `posix` thread model. The value of the native handle is undefined for a thread which is not joinable.

mutex, timed_mutex The native handle type is `__gthread_mutex_t*` i.e. `pthread_mutex_t*` for the `posix` thread model.

recursive_mutex, recursive_timed_mutex The native handle type is `__gthread_recursive_mutex_t*` i.e. `pthread_mutex_t*` for the `posix` thread model.

condition_variable The native handle type is `__gthread_cond_t*` i.e. `pthread_cond_t*` for the `posix` thread model.

30.6.1 *[futures.overview]/2* `launch` is a scoped enumeration type with overloaded operators to support bitmask operations. There are no additional bitmask elements defined.

1.1.3 C++ 2014

In this implementation the `-std=gnu++14` or `-std=c++14` flag must be used to enable language and library features. See [dialect](#) options. The pre-defined symbol `__cplusplus` is used to check for the presence of the required flag. GCC 6.1 was the first release with non-experimental C++14 support, so the API and ABI of features added in C++14 is only stable since that release.

This status table is based on the table of contents of ISO/IEC 14882:2014. Some subclauses are not shown in the table where the content is unchanged since C++11 and the implementation is complete.

This section describes the C++14 and library TS support in mainline GCC, not in any particular release.

1.1.3.1 Implementation Specific Behavior

1.1.3.1.1 Filesystem TS

2.1 *POSIX conformance [fs.conform.9945]* The behavior of the filesystem library implementation will depend on the target operating system. Some features will not be supported on some targets. Symbolic links and file permissions are not supported on Windows.

15.30 *Rename [fs.op.rename]* On Windows, `experimental::filesystem::rename` is implemented by calling `MoveFileExW` and so does not meet the requirements of POSIX `rename` when one or both of the paths resolves to an existing directory. Specifically, it is possible to rename a directory so it replaces a non-directory (POSIX requires an error in that case), and it is not possible to rename a directory to replace another directory (POSIX requires that to work if the directory being replaced is empty).

Section	Description	Status	Comments
18	<i>Language support</i>		
18.1	General		
18.2	Types	Y	
18.3	Implementation properties		
18.3.2	Numeric Limits		
18.3.2.3	Class template numeric_limits	Y	
18.3.2.4	numeric_limits members	Y	
18.3.2.5	float_round_style	N	
18.3.2.6	float_denorm_style	N	
18.3.2.7	numeric_limits specializations	Y	
18.3.3	C Library	Y	
18.4	Integer types		
18.4.1	Header <cstdint> synopsis	Y	
18.5	Start and termination	Partial	C library dependency for quick_exit, at_quick_exit
18.6	Dynamic memory management	Y	
18.7	Type identification		
18.7.1	Class type_info	Y	
18.7.2	Class bad_cast	Y	
18.7.3	Class bad_typeid	Y	
18.8	Exception handling		
18.8.1	Class exception	Y	
18.8.2	Class bad_exception	Y	
18.8.3	Abnormal termination	Y	
18.8.4	uncaught_exception	Y	
18.8.5	Exception Propagation	Y	
18.8.6	nested_exception	Y	
18.9	Initializer lists		
18.9.1	Initializer list constructors	Y	
18.9.2	Initializer list access	Y	
18.9.3	Initializer list range access	Y	
18.10	Other runtime support	Y	
19	<i>Diagnostics</i>		
19.1	General		
19.2	Exception classes	Y	
19.3	Assertions	Y	
19.4	Error numbers	Y	
19.5	System error support		
19.5.1	Class error_category	Y	
19.5.2	Class error_code	Y	
19.5.3	Class error_condition	Y	
19.5.4	Comparison operators	Y	
19.5.5	Class system_error	Y	
20	<i>General utilities</i>		
20.1	General		
20.2	Utility components		
20.2.1	Operators	Y	
20.2.2	swap	Y	
20.2.3	exchange	Y	
20.2.4	forward/move helpers	Y	
20.2.5	Function template declval	Y	
20.3	Pairs	Y	
20.4	Tuples	Y	
20.5	Compile-time integer sequences		
20.5.1	Class template	Y	

Paper	Title	Status	Comments
N3662	C++ Dynamic Arrays	N	Array Extensions TS
N3793	A proposal to add a utility class to represent optional objects	Y	Library Fundamentals TS
N3804	Any library proposal	Y	Library Fundamentals TS
N3866	Invocation type traits, but dropping <code>function_call_operator</code> .	N	Library Fundamentals TS
N3905	Faster string searching (Boyer-Moore et al.)	Y	Library Fundamentals TS
N3915	<code>apply()</code> call a function with arguments from a tuple	Y	Library Fundamentals TS
N3916	Polymorphic memory resources	Partial (missing pool resource and buffer resource classes)	Library Fundamentals TS
N3920	Extending <code>shared_ptr</code> to support arrays	Y	Library Fundamentals TS
N3921	<code>string_view</code> : a non-owning reference to a string	Y	Library Fundamentals TS
N3925	A sample proposal	Y	Library Fundamentals TS
N3932	Variable Templates For Type Traits	Y	Library Fundamentals TS
N4100	File System	Y	Link with <code>-lstdc++fs</code>

Table 1.4: C++ Technical Specifications Implementation Status

1.1.4 C++ 2017

In this implementation the `-std=gnu++17` or `-std=c++17` flag must be used to enable language and library features. See [dialect](#) options. The pre-defined symbol `__cplusplus` is used to check for the presence of the required flag. GCC 9.1 was the first release with non-experimental C++17 support, so the API and ABI of features added in C++17 is only stable since that release.

This section describes the C++17 and library TS support in mainline GCC, not in any particular release.

The following table lists new library features that are included in the C++17 standard. The "Proposal" column provides a link to the ISO C++ committee proposal that describes the feature, while the "Status" column indicates the first version of GCC that contains an implementation of this feature (if it has been implemented). The "SD-6 Feature Test" column shows the corresponding macro or header from [SD-6: Feature-testing recommendations for C++](#).

Note 1: This feature is supported in GCC 7.1 and 7.2 but before GCC 7.3 the `__cpp_lib` macro is not defined, and compilation will fail if the header is included without using `-std` to enable C++17 support.

Note 2: This feature is supported in older releases but the `__cpp_lib` macro is not defined to the right value (or not defined at all) until the version shown in parentheses.

Note 3: The Parallel Algorithms have an external dependency on Intel TBB 2018 or later. If the `<execution>` header is included then `-ltbb` must be used to link to TBB.

Note 4: The mathematical special functions are enabled in C++17 mode from GCC 7.1 onwards. For GCC 6.x or for C++11/C++14 define `__STDCPP_WANT_MATH_SPEC_FUNCS__` to a non-zero value and test for `__STDCPP_MATH_SPEC_FUNCS__ >= 201003L`.

The following status table is based on the table of contents of ISO/IEC 14882:2017. Some subclauses are not shown in the table where the content is unchanged since C++14 and the implementation is complete.

Library Feature	Proposal	Status	SD-6 Feature Test
constexpr std::hardware_{constructive,destructive}_interference_size	P0154R1	12.1	__cpp_lib_hardware_interference_size >= 201603
Core Issue 1776: Replacement of class objects containing reference members	P0137R1	7.1	__cpp_lib_launders >= 201606
Wording for std::uncaught_exceptions	N4259	6.1	__cpp_lib_uncaught_exceptions >= 201411
C++17 should refer to C11 instead of C99	P0063R3	9.1	
Variant: a type-safe union for C++17	P0088R3	7.1	__has_include(<variant>), __cpp_lib_variant >= 201603 (since 7.3, see Note 1)
Library Fundamentals V1 TS Components: optional	P0220R1	7.1	__has_include(<optional>), __cpp_lib_optional >= 201603 (since 7.3, see Note 1)
Library Fundamentals V1 TS Components: any	P0220R1	7.1	__has_include(<any>), __cpp_lib_any >= 201603 (since 7.3, see Note 1)
Library Fundamentals V1 TS Components: string_view	P0220R1	7.1	__has_include(<string_view>), __cpp_lib_string_view >= 201603 (since 7.3, see Note 1)
Library Fundamentals V1 TS Components: memory_resource	P0220R1	9.1	__has_include(<memory_resource>), __cpp_lib_memory_resource >= 201603
Library Fundamentals V1 TS Components: apply	P0220R1	7.1	__cpp_lib_apply >= 201603
Library Fundamentals V1 TS Components: shared_ptr<T[]>	P0220R1	7.1	__cpp_lib_shared_ptr_arrays >= 201603
Library Fundamentals V1 TS Components: Searchers	P0220R1	7.1	__cpp_lib_boyer_moore_searcher >= 201603
Library Fundamentals V1 TS Components: Sampling	P0220R1	7.1	__cpp_lib_sample >= 201603
Constant View: A proposal for a std::as_const helper function template	P0007R1	7.1	__cpp_lib_as_const >= 201510
Improving pair and tuple	N4387	6.1	N/A
make_from_tuple: apply for construction	P0209R2	7.1	__cpp_lib_make_from_tuple >= 201606
Removing auto_ptr, random_shuffle(), And Old <functional> Stuff	N4190	No (kept for backwards compatibility)	
Deprecating Vestigial Library Parts in C++17	P0174R2	12.1	
Making std::owner_less more flexible	P0074R0	7.1	__cpp_lib_transparent_operators >= 201510
std::addressof should be constexpr	LWG2296	7.1	__cpp_lib_addressof_constexpr >= 201603
Safe conversions in unique_ptr<T[]>	N4089	6	
LWG 2228: Missing SFINAE rule in unique_ptr templated assignment	N4366	6	
Re-enabling	P0000R1	7.1	__cpp_lib_enable_shared_ptr >= 201603

Section	Description	Status	Comments
21	<i>Language support</i>		
21.1	General		
21.2	Common definitions		
21.3	Implementation properties		
21.3.1	General		
21.3.2	Header <code><limits></code> synopsis		
21.3.3	Floating-point type properties		
21.3.3.1	<code>float_round_style</code>	N	
21.3.3.2	<code>float_denorm_style</code>	N	
21.3.4	Class template <code>numeric_limits</code>	Y	
21.3.5	Header <code><climits></code> synopsis	Y	
21.3.6	Header <code><cfloat></code> synopsis	Y	
21.4	Integer types		
21.4.1	Header <code><cstdint></code> synopsis	Y	
21.5	Start and termination	Partial	C library dependency for <code>quick_exit</code> , <code>at_quick_exit</code>
21.6	Dynamic memory management		
21.6.1	Header <code><new></code> synopsis		
21.6.2	Storage allocation and deallocation	Y	
21.6.3	Storage allocation errors	Y	
21.6.4	Pointer optimization barrier	Y	
21.6.5	Hardware interference size	Y	
21.7	Type identification	Y	
21.8	Exception handling		
21.8.1	Header <code><exception></code> synopsis		
21.8.2	Class <code>exception</code>	Y	
21.8.3	Class <code>bad_exception</code>	Y	
21.8.4	Abnormal termination	Y	
21.8.5	<code>uncaught_exceptions</code>	Y	
21.8.6	Exception Propagation	Y	
21.8.7	<code>nested_exception</code>	Y	
21.9	Initializer lists	Y	
21.10	Other runtime support	Y	
22	<i>Diagnostics</i>		
22.1	General		
22.2	Exception classes	Y	
22.3	Assertions	Y	
22.4	Error numbers	Y	
22.5	System error support		
23	<i>General utilities</i>		
23.1	General		
23.2	Utility components		
23.2.1	Header <code><utility></code> synopsis		
23.2.2	Operators	Y	
23.2.3	<code>swap</code>	Y	
23.2.4	<code>exchange</code>	Y	
23.2.5	Forward/move helpers	Y	
23.2.6	Function template <code>as_const</code>	Y	
23.2.7	Function template <code>declval</code>	Y	
			Floating-point types up to 64-bit are formatted using

Paper	Title	Status	Comments
N4076	A generalized callable negator	Y	Library Fundamentals 2 TS
N4273	Uniform Container Erasure	Y	Library Fundamentals 2 TS
N4061	Greatest Common Divisor and Least Common Multiple	Y	Library Fundamentals 2 TS
N4066	Delimited iterators	Y	Library Fundamentals 2 TS
N4282	The World's Dumbest Smart Pointer	Y	Library Fundamentals 2 TS
N4388	Const-Propagating Wrapper	Y	Library Fundamentals 2 TS
N4391	make_array, revision 4	Y	Library Fundamentals 2 TS
N4502	Support for the C++ Detection Idiom, V2	Y	Library Fundamentals 2 TS
N4519	Source-Code Information Capture	Y	Library Fundamentals 2 TS
N4521	Merge Fundamentals V1 into V2	N (components from V1 are still in namespace fundamentals_v1)	Library Fundamentals 2 TS
P0013R1	Logical Operator Type Traits (revision 1)	Y	Library Fundamentals 2 TS
N4531	std::rand replacement, revision 3	Y	Library Fundamentals 2 TS
P0214R9	Data-Parallel Types	Y	Parallelism 2 TS

Table 1.7: C++ Technical Specifications Implementation Status

1.1.4.1 Implementation Specific Behavior

For behaviour which is also specified by previous standards, see [C++ 1998/2003 Implementation Specific Behavior](#) and [C++ 2011 Implementation Specific Behavior](#). This section only documents behaviour which is new in the 2017 standard.

20.5.1.2 [headers] Whether names from Annex K are declared by C++ headers depends on whether the underlying C library supports Annex K and declares the names. For the GNU C library, there is no Annex K support and so none of its names are declared by C++ headers.

23.6.5 [optional.bad_optional_access] `what()` returns "bad optional access".

23.7.3 [variant.variant] `variant` supports over-aligned types.

23.7.10 [variant.bad_access] `what()` returns one of the strings "std::get: variant is valueless", "std::get: wrong index for variant", "std::visit: variant is valueless", or "std::visit<R>: variant is valueless".

23.8.2 [any.bad_any_cast] `what()` returns "bad any_cast".

23.12.5 [mem.res.pool.options], 23.12.5 [mem.res.pool.mem] Let S equal `numeric_limits<size_t>::digits`. The limit for maximum number of blocks in a chunk is given by $2^N - 1$, where N is $\min(19, 3 + S/2)$. The largest allocation size that will be allocated from a pool is 2^{22} when $S > 20$, otherwise 3072 when $S > 16$, otherwise 768.

23.12.6.1 [mem.res.monotonic.buffer.ctor] The default `next_buffer_size` is `128 * sizeof(void*)`. The default growth factor is 1.5.

23.15.4.3 [meta.unary.prop] The predicate condition for `has_unique_object_representations` is true for all scalar types except floating point types.

23.19.3 [execpol.type], 28.4.3 [algorithms.parallel.exec] There are no implementation-defined execution policies.

24.4.2 [string.view.template] `basic_string_view<C, T>::iterator` is `C*` and `basic_string_view<C, T>::const_iterator` is `const C*`.

9.4 [parallel.simd.traits] `memory_alignment<T, U>::value` is `sizeof(U) * T::size()` rounded up to the next power-of-two value.

9.6.1 [parallel.simd.overview] On ARM, `simd<T, _VecBuiltin<Bytes>>` is supported if `__ARM_NEON` is defined and `sizeof(T) <= 4`. Additionally, `sizeof(T) == 8` with integral `T` is supported if `__ARM_ARCH >= 8`, and `double` is supported if `__aarch64__` is defined. On POWER, `simd<T, _VecBuiltin<Bytes>>` is supported if `__ALTIVEC__` is defined and `sizeof(T) < 8`. Additionally, `double` is supported if `__VSX__` is defined, and any `T` with `sizeof(T) <= 8` is supported if `__POWER8_VECTOR__` is defined. On x86, given an extended ABI tag `Abi`, `simd<T, Abi>` is supported according to the following table:

ABI tag <code>Abi</code>	value type <code>T</code>	values for Bytes	required machine option
<code>_VecBuiltin<Bytes></code>	float	8, 12, 16	"-msse"
		20, 24, 28, 32	"-mavx"
	double	16	"-msse2"
		24, 32	"-mavx"
	integral types other than bool	Bytes ≤ 16 and Bytes divisible by <code>sizeof(T)</code>	"-msse2"
		16 < Bytes ≤ 32 and Bytes divisible by <code>sizeof(T)</code>	"-mavx2"
<code>_VecBuiltin<Bytes></code> and <code>_VecBltnBtmsk<Bytes></code>	vectorizable types with <code>sizeof(T) ≥ 4</code>	32 < Bytes ≤ 64 and Bytes divisible by <code>sizeof(T)</code>	"-mavx512f"
	vectorizable types with <code>sizeof(T) < 4</code>		"-mavx512bw"
<code>_VecBltnBtmsk<Bytes></code>	vectorizable types with <code>sizeof(T) ≥ 4</code>	Bytes ≤ 32 and Bytes divisible by <code>sizeof(T)</code>	"-mavx512vl"
	vectorizable types with <code>sizeof(T) < 4</code>		"-mavx512bw" and "-mavx512vl"

Table 1.8: Support for Extended ABI Tags

1.1.5 C++ 2020

In this implementation the `-std=gnu++20` or `-std=c++20` flag must be used to enable language and library features. See [dialect](#) options. The pre-defined symbol `__cplusplus` is used to check for the presence of the required flag.

This section describes the C++20 and library TS support in mainline GCC, not in any particular release.

The following table lists new library features that are included in the C++20 standard. The "Proposal" column provides a link to the ISO C++ committee proposal that describes the feature, while the "Status" column indicates the first version of GCC that contains an implementation of this feature (if it has been implemented). A dash (—) in the status column indicates that the changes in the proposal either do not affect the code in `libstdc++`, or the changes are not required for conformance. The "SD-6 Feature Test / Notes" column shows the corresponding macro or header from [SD-6: Feature-testing recommendations for C++](#) (where applicable) or any notes about the implementation.

Note 1: This feature is supported in older releases but the `__cpp_lib` macro is not defined to the right value (or not defined at all) until the version shown in parentheses.

Note 2: The C++20 calendar types are supported since 11.1, time zones and UTC are supported since 13.1, and `chrono::parse` is supported since 14.1.

The following status table is based on the table of contents of ISO/IEC 14882:2020. Some subclauses are not shown in the table where the content is unchanged since C++17 and the implementation is complete.

Library Feature	Proposal	Status	SD-6 Feature Test / Notes
Compile-time programming			
Add constexpr modifiers to functions in <code><algorithm></code> and <code><utility></code> Headers	P0202R3	10.1	<code>__cpp_lib_constexpr_algorithm</code> >= 201703L
Constexpr for swap and swap related functions	P0879R0	10.1	<code>__cpp_lib_constexpr_algorithm</code> >= 201806L
Constexpr for <code>std::complex</code>	P0415R1	9.1	<code>__cpp_lib_constexpr_complex</code> >= 201711L (since 9.4, see Note 1)
<code>std::is_constant_evaluated</code>	P0595R2	9.1	<code>__cpp_lib_is_constant_evaluated</code> >= 201811L
More constexpr containers	P0784R7	10.1	<code>__cpp_lib_constexpr_dynamically_resizable_container</code> >= 201907L
Making <code>std::string</code> constexpr	P0980R1	12.1	<code>__cpp_lib_constexpr_string</code> >= 201907L
Making <code>std::vector</code> constexpr	P1004R2	12.1	<code>__cpp_lib_constexpr_vector</code> >= 201907L
Constexpr in <code>std::pointer_traits</code>	P1006R1	9.1	<code>__cpp_lib_constexpr_memory</code> >= 201811L (since 9.4, see Note 1)
constexpr for <code><numeric></code> algorithms	P1645R1	10.1	<code>__cpp_lib_constexpr_numeric</code> >= 201911L
Constexpr iterator requirements	P0858R0	9.1	<code>__cpp_lib_array_constexpr_iterator</code> >= 201803L <code>__cpp_lib_string_view</code> >= 201803L (both since 9.4, see Note 1)
constexpr comparison operators for <code>std::array</code>	P1023R0	10.1	<code>__cpp_lib_array_constexpr_comparison_operators</code> >= 201806
Misc constexpr bits	P1032R1	10.1	<code>__cpp_lib_array_constexpr_iterator</code> >= 201811L <code>__cpp_lib_constexpr_function</code> >= 201811L <code>__cpp_lib_constexpr_iterator</code> >= 201811L <code>__cpp_lib_constexpr_string</code> >= 201811L <code>__cpp_lib_constexpr_tuple</code> >= 201811L <code>__cpp_lib_constexpr_utility</code> >= 201811L
constexpr <code>INVOKE</code>	P1065R2	10.1	<code>__cpp_lib_constexpr_function</code> >= 201907L
Transformation Trait <code>remove_cvref</code>	P0550R2	9.1	<code>__cpp_lib_remove_cvref</code> >= 201711L (since 9.4, see Note 1)
Implicit conversion traits and utility functions	P0758R1	9.1	<code>__cpp_lib_is_nothrow_conversion</code> >= 201806L (since 9.4, see Note 1)
The identity metafunction	P0887R1	9.1	<code>__cpp_lib_type_identity</code> >= 201806L (since 9.4, see Note 1)
<code>unwrap_ref_decay</code> and <code>unwrap_reference</code>	P0318R1	9.1	<code>__cpp_lib_unwrap_ref</code> >= 201811L (since 9.4, see Note 1)
Improving Completeness Requirements for Type Traits	P1285R0	—	Most misuses are diagnosed, but not all.
Missing feature test macros	P1353R0	9.1	
Making			

Section	Description	Status	Comments
17	<i>Language support library</i>		
17.1	General		
17.2	Common definitions		
17.2.1	Header <code><cstdint></code> synopsis	Y	
17.2.2	Header <code><cstdlib></code> synopsis	Y	
17.2.3	Null pointers	Y	
17.2.4	Sizes, alignments, and offsets	Y	
17.2.5	byte type operations	Y	
17.3	Implementation properties		
17.3.1	General		
17.3.2	Header <code><version></code> synopsis	Y	
17.3.3	Header <code><limits></code> synopsis	Y	
17.3.4	Floating-point type properties		
17.3.4.1	Type <code>float_round_style</code>	N	
17.3.4.2	Type <code>float_denorm_style</code>	N	
17.3.5	Class template <code>numeric_limits</code>	Y	
17.3.6	Header <code><climits></code> synopsis	Y	
17.3.7	Header <code><cfloat></code> synopsis	Y	
17.4	Integer types		
17.4.1	General		
17.4.2	Header <code><cstdint></code> synopsis	Y	
17.5	Startup and termination	Partial	C library dependency for <code>quick_exit</code> , <code>at_quick_exit</code>
17.6	Dynamic memory management	Y	
17.7	Type identification	Y	
17.8	Source location		
17.8.1	Header <code><source_location></code> synopsis	Y	
17.8.2	Class <code>source_location</code>	Y	
17.9	Exception handling	Y	
17.10	Initializer lists	Y	
17.11	Comparisons		
17.11.1	Header <code><compare></code> synopsis	Y	
17.11.2	Comparison category types	Y	
17.11.3	Class template <code>common_comparison_category</code>	Y	
17.11.4	Concept <code>three_way_comparable</code>	Y	
17.11.5	Result of three-way comparison	Y	
17.11.6	Comparison algorithms	Y	
17.12	Coroutines		
17.12.1	General		
17.12.2	Header <code><coroutine></code> synopsis	Y	
17.12.3	Coroutine traits	Y	
17.12.4	Class template	Y	

1.1.5.1 Implementation Specific Behavior

For behaviour which is also specified by previous standards, see [C++ 1998/2003 Implementation Specific Behavior](#), [C++ 2011 Implementation Specific Behavior](#), and [C++ 2017 Implementation Specific Behavior](#). This section only documents behaviour which is new in the 2020 standard.

16.4.2.4 [compliance] The implementation is freestanding if the `-ffreestanding` compiler flag is used, and hosted otherwise.

16.4.2.4 [compliance] The support for always lock-free integral atomic types and presence of `atomic_signed_lock_free` and `atomic_unsigned_lock_free` type aliases depends on the target.

27.5.11 [time.duration.io] The `"μs"` (`"\u00b5\u0073"`) is used for `std::microPeriod::type` if the macro `_GLIBCXX_USE` is defined to a value other than zero before inclusion of the `chrono` header, `"us"` is used otherwise.

29.8.2.2 [stringbuf.cons] Sequence pointers are initialized to null pointers by the `basic_stringbuf(ios_base::openmode)` constructor.

31.7.1 [atomics.ref.generic.general], 31.7.3 [atomics.ref.int], 31.7.4 [atomics.ref.float], 31.7.5 [atomics.ref.pointer] The values of `is_always_lock_free` members depend on the target and cannot be listed here.

31.7.1 [atomics.ref.generic.general] If `sizeof(T)` is equal to either 1, 2, 4, 8, or 16, then the value of `required_alignment` member is equal to the maximum of `alignof(T)` and `sizeof(T)`. Otherwise `required_alignment` value is `alignof(T)`.

31.7.3 [atomics.ref.int] The value of `required_alignment` member is equal to the maximum of `alignof(value_type)` and `sizeof(value_type)`.

31.7.4 [atomics.ref.float], 31.7.5 [atomics.ref.pointer] The value of `required_alignment` member is equal to `alignof(value_type)`.

32.7.3 [thread.sema.cnt] The value of default argument for the `least_max_value` parameter depends on the target operating system and platform, however the value of `counting_semaphore<>::max()` is greater than or equal to `numeric_limits<int>`.

1.1.6 C++ 2023

In this implementation the `-std=gnu++23` or `-std=c++23` flag must be used to enable language and library features. See [dialect](#) options. The pre-defined symbol `__cplusplus` is used to check for the presence of the required flag.

This section describes the C++23 and library TS support in mainline GCC, not in any particular release.

The following table lists new library features that have been accepted into the C++23 working draft. The "Proposal" column provides a link to the ISO C++ committee proposal that describes the feature, while the "Status" column indicates the first version of GCC that contains an implementation of this feature (if it has been implemented). A dash (—) in the status column indicates that the changes in the proposal either do not affect the code in `libstdc++`, or the changes are not required for conformance. The "SD-6 Feature Test / Notes" column shows the corresponding macro or header from [SD-6: Feature-testing recommendations for C++](#) (where applicable) or any notes about the implementation.

1.1.7 C++ TR1

This table is based on the table of contents of ISO/IEC DTR 19768, Doc No: N1836=05-0096, Date: 2005-06-24, "Draft Technical Report on C++ Library Extensions".

In this implementation the header names are prefixed by `tr1/`, for instance `<tr1/functional>`, `<tr1/memory>`, and so on.

This page describes the TR1 support in mainline GCC, not in any particular release.

1.1.7.1 Implementation Specific Behavior

For behaviour which is specified by the 1998 and 2003 standards, see [C++ 1998/2003 Implementation Specific Behavior](#). This section documents behaviour which is required by TR1.

3.6.4 [tr.func.bind.place]/1 There are 29 placeholders defined and the placeholder types are Assignable.

Library Feature	Proposal	Status	SD-6 Feature Test / Notes
Ranges and Views			
Range constructor for <code>std::string_view</code>	P1989R2	11.1	
<code>join_view</code> should join all views of ranges	P2328R1	11.2	
Clarifying range adaptor objects	P2281R1	11.1	
Views should not be required to be default constructible	P2325R3	11.3	<code>__cpp_lib_ranges</code> >= 202106L
Conditionally borrowed ranges	P2017R1	11.1	
Require <code>span</code> & <code>basic_string_view</code> to be Trivially Copyable	P2251R1	Yes	
Repairing input range adaptors and <code>counted_iterator</code>	P2259R1	12.1	
Superior String Splitting	P2210R2	12.1	
What is a view?	P2415R2	12.1	<code>__cpp_lib_ranges</code> >= 202110L
Fix <code>istream_view</code>	P2432R1	12.1	
<code>starts_with</code> and <code>ends_with</code>	P1659R3		<code>__cpp_lib_ranges_starts_en</code> >= 202106L
<code>zip</code>	P2321R2	13.1	<code>__cpp_lib_ranges_zip</code> >= 202110L
<code>views::repeat</code>	P2474R2	13.1	<code>__cpp_lib_ranges_repeat</code> >= 202207L
<code>views::enumerate</code>	P2164R9	13.1	<code>__cpp_lib_ranges_enumerate</code> >= 202302L
<code>views::join_with</code>	P2441R2	13.1	<code>__cpp_lib_ranges_join_with</code> >= 202202L
Windowing range adaptors: <code>views::chunk</code> and <code>views::slide</code>	P2442R1	13.1	<code>__cpp_lib_ranges_slide</code> >= 202202L
<code>views::chunk_by</code>	P2443R1	13.1	<code>__cpp_lib_ranges_chunk_by</code> >= 202202L
<code>views::stride</code>	P1899R3	13.1	<code>__cpp_lib_ranges_stride</code> >= 202207L
<code>views::cartesian_product</code>	P2374R4	13.1	<code>__cpp_lib_ranges_cartesian</code> >= 202207L
Empty Product for certain Views	P2540R1	13.1	<code>__cpp_lib_ranges_cartesian</code> >= 202207L
<code>views::as_rvalue</code>	P2446R2	13.1	<code>__cpp_lib_ranges_as_rvalue</code> >= 202207L
<code>cbegin</code> should always return a constant iterator	P2278R4	13.1	<code>__cpp_lib_ranges_as_const</code> >= 202207L
<code>ranges::to</code>	P1206R7	14.1 (<code>ranges::to</code> function) 15.1 (new members in containers)	<code>__cpp_lib_ranges_to_contai</code> >= 202202L, <code>__cpp_lib_containers_range</code> >= 202202L
Ranges iterators as inputs to non-Ranges algorithms	P2408R5		<code>__cpp_lib_algorithm_iterat</code> >= 202207L
Pipe support for user-defined range adaptors	P2387R3	13.1	<code>__cpp_lib_bind_pack</code> >= 202202L, <code>__cpp_lib_ranges</code> >= 202202L
<code>ranges::iota</code> , <code>ranges::shift_left</code> , and <code>ranges::shift_right</code>	P2440R1	13.1 (<code>ranges::iota</code>)	<code>__cpp_lib_ranges_iota</code> >= 202202L, <code>__cpp_lib_shift</code> >= 202202L
<code>ranges::find_last</code>	P1223R5	13.1	<code>__cpp_lib_ranges_find_last</code>

Section	Description	Status	Comments
2	<i>General Utilities</i>		
2.1	Reference wrappers		
2.1.1	Additions to header <functional> synopsis	Y	
2.1.2	Class template reference_wrapper		
2.1.2.1	reference_wrapper construct/copy/destroy	Y	
2.1.2.2	reference_wrapper assignment	Y	
2.1.2.3	reference_wrapper access	Y	
2.1.2.4	reference_wrapper invocation	Y	
2.1.2.5	reference_wrapper helper functions	Y	
2.2	Smart pointers		
2.2.1	Additions to header <memory> synopsis	Y	
2.2.2	Class bad_weak_ptr	Y	
2.2.3	Class template shared_ptr		Uses code from boost::shared_ptr.
2.2.3.1	shared_ptr constructors	Y	
2.2.3.2	shared_ptr destructor	Y	
2.2.3.3	shared_ptr assignment	Y	
2.2.3.4	shared_ptr modifiers	Y	
2.2.3.5	shared_ptr observers	Y	
2.2.3.6	shared_ptr comparison	Y	
2.2.3.7	shared_ptr I/O	Y	
2.2.3.8	shared_ptr specialized algorithms	Y	
2.2.3.9	shared_ptr casts	Y	
2.2.3.10	get_deleter	Y	
2.2.4	Class template weak_ptr		
2.2.4.1	weak_ptr constructors	Y	
2.2.4.2	weak_ptr destructor	Y	
2.2.4.3	weak_ptr assignment	Y	
2.2.4.4	weak_ptr modifiers	Y	
2.2.4.5	weak_ptr observers	Y	
2.2.4.6	weak_ptr comparison	Y	
2.2.4.7	weak_ptr specialized algorithms	Y	
2.2.5	Class template enable_shared_from_this	Y	
3	<i>Function Objects</i>		
3.1	Definitions	Y	
3.2	Additions to <functional> synopsis	Y	
3.3	Requirements	Y	
3.4	Function return types	Y	
3.5	Function template mem_fn	Y	
3.6	Function object binders		
3.6.1	Class template is_bind_expression	Y	
3.6.2	Class template is_placeholder	Y	
3.6.3	Function template bind	Y	
3.6.4	Placeholders	Y	
3.7	Polymorphic function wrappers		
3.7.1	Class	Y	

1.1.8 C++ TR 24733

This table is based on the table of contents of ISO/IEC TR 24733:2011, "Extensions for the programming language C++ to support decimal floating-point arithmetic".

This page describes the TR 24733 support in mainline GCC, not in any particular release.

1.1.9 C++ IS 29124

This table is based on the table of contents of ISO/IEC FDIS 29124, Doc No: N3060, Date: 2010-03-06, "Extensions to the C++ Library to support mathematical special functions".

Complete support for IS 29124 is in GCC 6.1 and later releases, when using at least C++11 (for older releases or C++98/C++03 use TR1 instead). For C++11 and C++14 the additions to the library are not declared by their respective headers unless `__STDCPP_WANT_MATH_SPEC_FUNCS__` is defined as a macro that expands to a non-zero integer constant. For C++17 the special functions are always declared (since GCC 7.1).

When the special functions are declared the macro `__STDCPP_MATH_SPEC_FUNCS__` is defined to 201003L.

In addition to the special functions defined in IS 29124, for non-strict modes (i.e. `-std=gnu++NN` modes) the hypergeometric functions and confluent hypergeometric functions from TR1 are also provided, defined in namespace `__gnu_cxx`.

1.1.9.1 Implementation Specific Behavior

For behaviour which is specified by the 2011 standard, see [C++ 2011 Implementation Specific Behavior](#). This section documents behaviour which is required by IS 29124.

7.2 [macro.user]/3/4 The functions declared in Clause 8 are only declared when `__STDCPP_WANT_MATH_SPEC_FUNCS__ == 1` (or in C++17 mode, for GCC 7.1 and later).

8.1.1 [sf.cmath.Lnm]/1 The effect of calling these functions with $n \geq 128$ or $m \geq 128$ should be described here.

8.1.2 [sf.cmath.Plm]/3 The effect of calling these functions with $l \geq 128$ should be described here.

8.1.3 [sf.cmath.I]/3 The effect of calling these functions with $nu \geq 128$ should be described here.

8.1.8 [sf.cmath.J]/3 The effect of calling these functions with $nu \geq 128$ should be described here.

8.1.9 [sf.cmath.K]/3 The effect of calling these functions with $nu \geq 128$ should be described here.

8.1.10 [sf.cmath.N]/3 The effect of calling these functions with $nu \geq 128$ should be described here.

8.1.15 [sf.cmath.Hn]/3 The effect of calling these functions with $n \geq 128$ should be described here.

8.1.16 [sf.cmath.Ln]/3 The effect of calling these functions with $n \geq 128$ should be described here.

8.1.17 [sf.cmath.Pl]/3 The effect of calling these functions with $l \geq 128$ should be described here.

8.1.19 [sf.cmath.j]/3 The effect of calling these functions with $n \geq 128$ should be described here.

8.1.20 [sf.cmath.Ylm]/3 The effect of calling these functions with $l \geq 128$ should be described here.

8.1.21 [sf.cmath.n]/3 The effect of calling these functions with $n \geq 128$ should be described here.

1.2 License

There are two licenses affecting GNU libstdc++: one for the code, and one for the documentation.

There is a license section in the FAQ regarding common [questions](#). If you have more questions, ask the FSF or the [gcc mailing list](#).

Section	Description	Status	Comments
0	<i>Introduction</i>		
1	<i>Normative references</i>		
2	<i>Conventions</i>		
3	<i>Decimal floating-point types</i>		
3.1	Characteristics of decimal floating-point types		
3.2	Decimal Types		
3.2.1	Class <code>decimal</code> synopsis	Partial	Missing declarations for formatted input/output; non-conforming extension for functions converting to integral type
3.2.2	Class <code>decimal32</code>	Partial	Missing 3.2.2.5 conversion to integral type; conforming extension for conversion from scalar decimal floating-point
3.2.3	Class <code>decimal64</code>	Partial	Missing 3.2.3.5 conversion to integral type; conforming extension for conversion from scalar decimal floating-point
3.2.4	Class <code>decimal128</code>	Partial	Missing 3.2.4.5 conversion to integral type; conforming extension for conversion from scalar decimal floating-point
3.2.5	Initialization from coefficient and exponent	Y	
3.2.6	Conversion to generic floating-point type	Y	
3.2.7	Unary arithmetic operators	Y	
3.2.8	Binary arithmetic operators	Y	
3.2.9	Comparison operators	Y	
3.2.10	Formatted input	N	
3.2.11	Formatted output	N	
3.3	Additions to header <code>limits</code>	N	
3.4	Headers <code>cfloat</code> and <code>float.h</code>		
3.4.2	Additions to header <code>cfloat</code> synopsis	Y	
3.4.3	Additions to header <code>float.h</code> synopsis	N	
3.4.4	Maximum finite value	Y	
3.4.5	Epsilon	Y	
3.4.6	Minimum positive normal value	Y	
3.4.7	Minimum positive subnormal value	Y	
3.4.8	Evaluation format	Y	
3.5	Additions to <code>cfenv</code> and <code>fenv.h</code>	Outside the scope of GCC	
3.6	Additions to <code>cmath</code> and <code>math.h</code>	Outside the scope of GCC	
3.7	Additions to <code>cstdio</code> and <code>stdio.h</code>	Outside the scope of GCC	
3.8	Additions to <code>cstdlib</code> and <code>stdlib.h</code>	Outside the scope of GCC	
3.9	Additions to <code>cwchar</code> and <code>wchar.h</code>	Outside the scope of GCC	
3.10	Facets	N	
3.11	Type traits	N	

Section	Description	Status	Comments
7	Macro names	Partial	No diagnostic for inconsistent definitions of <code>__STDCPP_WANT_MATH_SPEC_FUNCS</code>
8	Mathematical special functions	Y	
8.1	Additions to header <code><cmath></code> synopsis	Y	
8.1.1	associated Laguerre polynomials	Y	
8.1.2	associated Legendre functions	Y	
8.1.3	beta function	Y	
8.1.4	(complete) elliptic integral of the first kind	Y	
8.1.5	(complete) elliptic integral of the second kind	Y	
8.1.6	(complete) elliptic integral of the third kind	Y	
8.1.7	regular modified cylindrical Bessel functions	Y	
8.1.8	cylindrical Bessel functions (of the first kind)	Y	
8.1.9	irregular modified cylindrical Bessel functions	Y	
8.1.10	cylindrical Neumann functions	Y	
8.1.11	(incomplete) elliptic integral of the first kind	Y	
8.1.12	(incomplete) elliptic integral of the second kind	Y	
8.1.13	(incomplete) elliptic integral of the third kind	Y	
8.1.14	exponential integral	Y	
8.1.15	Hermite polynomials	Y	
8.1.16	Laguerre polynomials	Y	
8.1.17	Legendre polynomials	Y	
8.1.18	Riemann zeta function	Y	
8.1.19	spherical Bessel functions (of the first kind)	Y	
8.1.20	spherical associated Legendre functions	Y	
8.1.21	spherical Neumann functions	Y	
8.2	Additions to header <code><math.h></code>	Y	
8.3	The header <code><ctgmath></code>	Partial	Conflicts with C++ 2011 requirements.
8.4	The header <code><tgmath.h></code>	N	Conflicts with C++ 2011 requirements.

Table 1.14: C++ Special Functions Implementation Status

1.2.1 The Code: GPL

The source code is distributed under the **GNU General Public License version 3**, with the addition under section 7 of an exception described in the “GCC Runtime Library Exception, version 3.1” as follows (or see the file COPYING.RUNTIME):

GCC RUNTIME LIBRARY EXCEPTION

Version 3.1, 31 March 2009

Copyright (C) 2009 Free Software Foundation, Inc.

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

This GCC Runtime Library Exception ("Exception") is an additional permission under section 7 of the GNU General Public License, version 3 ("GPLv3"). It applies to a given file (the "Runtime Library") that bears a notice placed by the copyright holder of the file stating that the file is governed by GPLv3 along with this Exception.

When you use GCC to compile a program, GCC may combine portions of certain GCC header files and runtime libraries with the compiled program. The purpose of this Exception is to allow compilation of non-GPL (including proprietary) programs to use, in this way, the header files and runtime libraries covered by this Exception.

0. Definitions.

A file is an "Independent Module" if it either requires the Runtime Library for execution after a Compilation Process, or makes use of an interface provided by the Runtime Library, but is not otherwise based on the Runtime Library.

"GCC" means a version of the GNU Compiler Collection, with or without modifications, governed by version 3 (or a specified later version) of the GNU General Public License (GPL) with the option of using any subsequent versions published by the FSF.

"GPL-compatible Software" is software whose conditions of propagation, modification and use would permit combination with GCC in accord with the license of GCC.

"Target Code" refers to output from any compiler for a real or virtual target processor architecture, in executable form or suitable for input to an assembler, loader, linker and/or execution phase. Notwithstanding that, Target Code does not include data in any format that is used as a compiler intermediate representation, or used for producing a compiler intermediate representation.

The "Compilation Process" transforms code entirely represented in non-intermediate languages designed for human-written code, and/or in Java Virtual Machine byte code, into Target Code. Thus, for example, use of source code generators and preprocessors need not be considered part of the Compilation Process, since the Compilation Process can be understood as starting with the output of the generators or preprocessors.

A Compilation Process is "Eligible" if it is done using GCC, alone or with other GPL-compatible software, or if it is done without using any work based on GCC. For example, using non-GPL-compatible Software to optimize any GCC intermediate representations would not qualify as an Eligible Compilation Process.

1. Grant of Additional Permission.

You have permission to propagate a work of Target Code formed by combining the Runtime Library with Independent Modules, even if such propagation would otherwise violate the terms of GPLv3, provided that all Target Code was generated by Eligible Compilation Processes. You may then convey such a combination under terms of your choice, consistent with the licensing of the Independent Modules.

2. No Weakening of GCC Copyleft.

The availability of this Exception does not imply any general presumption that third-party software is unaffected by the copyleft requirements of the license of GCC.

Hopefully that text is self-explanatory. If it isn't, you need to speak to your lawyer, or the Free Software Foundation.

1.2.2 The Documentation: GPL, FDL

The documentation shipped with the library and made available over the web, excluding the pages generated from source comments, are copyrighted by the Free Software Foundation, and placed under the [GNU Free Documentation License version 1.3](#). There are no Front-Cover Texts, no Back-Cover Texts, and no Invariant Sections.

For documentation generated by doxygen or other automated tools via processing source code comments and markup, the original source code license applies to the generated files. Thus, the doxygen documents are licensed [GPL](#).

If you plan on making copies of the documentation, please let us know. We can probably offer suggestions.

1.3 Bugs

1.3.1 Implementation Bugs

Information on known bugs, details on efforts to fix them, and fixed bugs are all available as part of the [GCC bug tracking system](#), under the component "libstdc++".

1.3.2 Standard Bugs

Everybody's got issues. Even the C++ Standard Library.

The Library Working Group, or LWG, is the ISO subcommittee responsible for making changes to the library. They periodically publish an Issues List containing problems and possible solutions. As they reach a consensus on proposed solutions, we often incorporate the solution.

Here are the issues which have resulted in code changes to the library. The links are to the full version of the Issues List. You can read the full version online at the [ISO C++ Committee homepage](#).

If a DR is not listed here, we may simply not have gotten to it yet; feel free to submit a patch. Search the `include` and `src` directories for appearances of `_GLIBCXX_RESOLVE_LIB_DEFECTS` for examples of style. Note that we usually do not make changes to the code until an issue has reached [DR](#) status.

- 5: `string::compare` specification questionable** This should be two overloaded functions rather than a single function.
- 17: Bad bool parsing** Apparently extracting Boolean values was messed up...
- 19: "Noconv" definition too vague** If `codecvt::do_in` returns `noconv` there are no changes to the values in `[to, to_limit)`.
- 22: Member `open` vs flags** Re-opening a file stream does *not* clear the state flags.
- 23: `Num_get` overflow result** Implement the proposed resolution.
- 25: String operator<< uses `width()` value wrong** Padding issues.
- 48: Use of non-existent exception constructor** An instance of `ios_base::failure` is constructed instead.
- 49: Underspecification of `ios_base::sync_with_stdio`** The return type is the *previous* state of synchronization.
- 50: Copy constructor and assignment operator of `ios_base`** These members functions are declared `private` and are thus inaccessible. Specifying the correct semantics of "copying stream state" was deemed too complicated.
- 60: What is a formatted input function?** This DR made many widespread changes to `basic_istream` and `basic_ostream` all of which have been implemented.
- 63: Exception-handling policy for unformatted output** Make the policy consistent with that of formatted input, unformatted input, and formatted output.
- 68: Extractors for `char*` should store null at end** And they do now. An editing glitch in the last item in the list of [27.6.1.2.3]/7.
- 74: Garbled text for `codecvt::do_max_length`** The text of the standard was gibberish. Typos gone rampant.
- 75: Contradiction in `codecvt::length`'s argument types** Change the first parameter to `stateT&` and implement the new effects paragraph.
- 83: `string::npos` vs. `string::max_size()`** Safety checks on the size of the string should test against `max_size()` rather than `npos`.
- 90: Incorrect description of operator>> for strings** The effect contain `isspace(c, getloc())` which must be replaced by `isspace(c, is.getloc())`.
- 91: Description of operator>> and `getline()` for `string<>` might cause endless loop** They behave as a formatted input function and as an unformatted input function, respectively (except that `getline` is not required to set `gcount`).
- 103: `set::iterator` is required to be modifiable, but this allows modification of keys.** For associative containers where the value type is the same as the key type, both `iterator` and `const_iterator` are constant iterators.
- 109: Missing binders for non-const sequence elements** The `binder1st` and `binder2nd` didn't have an `operator()` taking a non-const parameter.
- 110: `istreambuf_iterator::equal` not const** This was not a const member function. Note that the DR says to replace the function with a const one; we have instead provided an overloaded version with identical contents.
- 117: `basic_ostream` uses nonexistent `num_put` member functions** `num_put::put()` was overloaded on the wrong types.
- 118: `basic_istream` uses nonexistent `num_get` member functions** Same as 117, but for `num_get::get()`.
- 129: Need error indication from `seekp()` and `seekg()`** These functions set `failbit` on error now.
- 130: Return type of `container::erase(iterator)` differs for associative containers** Make member `erase` return iterator for `set`, `multiset`, `map`, `multimap`.
- 136: `seekp`, `seekg` setting wrong streams?** `seekp` should only set the output stream, and `seekg` should only set the input stream.
- 167: Improper use of `traits_type::length()`** `op<<` with a `const char*` was calculating an incorrect number of characters to write.
- 169: Bad efficiency of `overflow()` mandated** Grow efficiently the internal array object.

- 171: Strange seekpos() semantics due to joint position** Quite complex to summarize...
- 181: make_pair() unintended behavior** This function used to take its arguments as reference-to-const, now it copies them (pass by value).
- 195: Should basic_istream::sentry's constructor ever set eofbit?** Yes, it can, specifically if EOF is reached while skipping whitespace.
- 206: operator new(size_t, nothrow) may become unlinked to ordinary operator new if ordinary version replaced** The `nothrow` forms of `new` and `delete` were changed to call the throwing forms, handling any exception by catching it and returning a null pointer.
- 211: operator>>(istream&, string&) doesn't set failbit** If nothing is extracted into the string, `op>>` now sets `failbit` (which can cause an exception, etc., etc.).
- 214: set::find() missing const overload** Both `set` and `multiset` were missing overloaded `find`, `lower_bound`, `upper_bound`, and `equal_range` functions for `const` instances.
- 231: Precision in iostream?** For conversion from a floating-point type, `str.precision()` is specified in the conversion specification.
- 233: Insertion hints in associative containers** Implement N1780, first check before then check after, insert as close to hint as possible.
- 235: No specification of default ctor for reverse_iterator** The declaration of `reverse_iterator` lists a default constructor. However, no specification is given what this constructor should do.
- 241: Does unique_copy() require CopyConstructible and Assignable?** Add a helper for `forward_iterator/output_iterator`, fix the existing one for `input_iterator/output_iterator` to not rely on `Assignability`.
- 243: get and getline when sentry reports failure** Store a null character only if the character array has a non-zero size.
- 251: basic_stringbuf missing allocator_type** This nested typedef was originally not specified.
- 253: valarray helper functions are almost entirely useless** Make the copy constructor and copy-assignment operator declarations public in `gslice_array`, `indirect_array`, `mask_array`, `slice_array`; provide definitions.
- 265: std::pair::pair() effects overly restrictive** The default ctor would build its members from copies of temporaries; now it simply uses their respective default ctors.
- 266: bad_exception::~bad_exception() missing Effects clause** The `bad_*` classes no longer have destructors (they are trivial), since no description of them was ever given.
- 271: basic_istream missing typedefs** The typedefs it inherits from its base classes can't be used, since (for example) `basic_istream` is ambiguous.
- 275: Wrong type in num_get::get() overloads** Similar to 118.
- 280: Comparison of reverse_iterator to const reverse_iterator** Add global functions with two template parameters. (NB: not added for now a templated assignment operator)
- 292: Effects of a.copypmt (a)** If `(this == &rhs)` do nothing.
- 300: List::merge() specification incomplete** If `(this == &x)` do nothing.
- 303: Bitset input operator underspecified** Basically, compare the input character to `is.widen(0)` and `is.widen(1)`.
- 305: Default behavior of codecvt<wchar_t, char, mbstate_t>::length()** Do not specify what `codecvt<wchar_t, char, mbstate_t>::do_length` must return.
- 328: Bad sprintf format modifier in money_put<>::do_put()** Change the format string to `"%.0Lf"`.
- 365: Lack of const-qualification in clause 27** Add `const` overloads of `is_open`.
- 387: std::complex over-encapsulated** Add the `real(T)` and `imag(T)` members; in C++11 mode, also adjust the existing `real()` and `imag()` members and free functions.

- 389: Const overload of `valarray::operator[]` returns by value** Change it to return a `const T&`.
- 396: what are characters zero and one** Implement the proposed resolution.
- 402: Wrong new expression in `[some_]allocator::construct`** Replace "new" with "::new".
- 408: Is `vector<reverse_iterator<char*>>` forbidden?** Tweak the debug-mode checks in `_Safe_iterator`.
- 409: Closing an `fstream` should clear the error state** Have `open` clear the error flags.
- 415: Behavior of `std::ws`** Change it to be an unformatted input function (i.e. construct a sentry and catch exceptions).
- 431: Swapping containers with unequal allocators** Implement Option 3, as per N1599.
- 432: `stringbuf::overflow()` makes only one write position available** Implement the resolution, beyond DR 169.
- 434: `bitset::to_string()` hard to use** Add three overloads, taking fewer template arguments.
- 438: Ambiguity in the "do the right thing" clause** Implement the resolution, basically cast less.
- 445: `iterator_traits::reference` unspecified for some iterator categories** Change `istreambuf_iterator::reference` in C++11 mode.
- 453: `basic_stringbuf::seekoff` need not always fail for an empty stream** Don't fail if the next pointer is null and `newoff` is zero.
- 455: `cerr::tie()` and `wcerr::tie()` are overspecified** Initialize `cerr` tied to `cout` and `wcerr` tied to `wcout`.
- 464: Suggestion for new member functions in standard containers** Add `data()` to `std::vector` and `at(const key_type)` to `std::map`.
- 467: `char_traits::lt()`, `compare()`, and `memcmp()`** Change `lt`.
- 508: Bad parameters for `ranlux64_base_01`** Fix the parameters.
- 512: Seeding `subtract_with_carry_01` from a single unsigned long** Construct a `linear_congruential` engine and seed with it.
- 526: Is it undefined if a function in the standard changes in parameters?** Use `&value`.
- 538: 241 again: Does `unique_copy()` require `CopyConstructible` and `Assignable`?** In case of `input_iterator/output_iterator` rely on Assignability of `input_iterator`' `value_type`.
- 539: `partial_sum` and `adjacent_difference` should mention requirements** We were almost doing the right thing, just use `std::move` in `adjacent_difference`.
- 541: `shared_ptr` template assignment and `void`** Add an `auto_ptr<void>` specialization.
- 543: `valarray` slice default constructor** Follow the straightforward proposed resolution.
- 550: What should the return type of `pow(float,int)` be?** In C++11 mode, remove the `pow(float,int)`, etc., signatures.
- 581: `flush()` not unformatted function** Change it to be a unformatted output function (i.e. construct a sentry and catch exceptions).
- 586: `string inserter` not a formatted function** Change it to be a formatted output function (i.e. catch exceptions).
- 596: 27.8.1.3 Table 112 omits "a+" and "a+b" modes** Add the missing modes to `fopen_mode`.
- 630: arrays of `valarray`** Implement the simple resolution.
- 660: Missing bitwise operations** Add the missing operations.
- 691: `const_local_iterator` `cbegin`, `cend` missing from TR1** In C++11 mode add `cbegin(size_type)` and `cend(size_type)` to the unordered containers.
- 693: `std::bitset::all()` missing** Add it, consistently with the discussion.
-

- 695: `ctype<char>::classic_table()` not accessible** Make the member functions `table` and `classic_table` public.
- 696: `istream::operator>>(int&)` broken** Implement the straightforward resolution.
- 761: `unordered_map` needs an `at()` member function** In C++11 mode, add `at()` and `at()` const.
- 775: Tuple indexing should be unsigned?** Implement the `int -> size_t` replacements.
- 776: Undescribed assign function of `std::array`** In C++11 mode, remove `assign`, add `fill`.
- 781: `std::complex` should add missing C99 functions** In C++11 mode, add `std::proj`.
- 809: `std::swap` should be overloaded for array types** Add the overload.
- 853: `to_string` needs updating with zero and one** Update / add the signatures.
- 865: More algorithms that throw away information** The traditional HP / SGI return type and value is blessed by the resolution of the DR.
- 1203: More useful rvalue stream insertion** Return the stream as its original type, not the base class.
- 1339: `uninitialized_fill_n` should return the end of its range** Return the end of the filled range.
- 2021: Further incorrect uses of `result_of`** Correctly decay types in signature of `std::async`.
- 2049: `is_destructible` underspecified** Handle non-object types.
- 2056: `future_errc` enums start with value 0 (invalid value for `broken_promise`)** Reorder enumerators.
- 2059: C++0x ambiguity problem with `map::erase`** Add additional overloads.
- 2062: 2062. Effect contradictions w/o no-throw guarantee of `std::function` swaps** Add `noexcept` to swap functions.
- 2063: Contradictory requirements for string move assignment** Respect propagation trait for move assignment.
- 2064: More `noexcept` issues in `basic_string`** Add `noexcept` to the comparison operators.
- 2067: `packaged_task` should have deleted copy c'tor with `const` parameter** Fix signatures.
- 2101: Some transformation types can produce impossible types** Use the referenceable type concept.
- 2106: `move_iterator` wrapping iterators returning prvalues** Change the `reference` type.
- 2108: No way to identify allocator types that always compare equal** Define and use `is_always_equal` even for C++11.
- 2118: `unique_ptr` for array does not support cv qualification conversion of actual argument** Adjust constraints to allow safe conversions.
- 2127: Move-construction with `raw_storage_iterator`** Add assignment operator taking an rvalue.
- 2132: `std::function` ambiguity** Constrain the constructor to only accept callable types.
- 2141: `common_type` trait produces reference types** Use `decay` for the result type.
- 2144: Missing `noexcept` specification in `type_index`** Add `noexcept`
- 2145: `error_category` default constructor** Declare a public `constexpr` constructor.
- 2162: `allocator_traits::max_size` missing `noexcept`** Add `noexcept`.
- 2187: `vector<bool>` is missing `emplace` and `emplace_back` member functions** Add `emplace` and `emplace_back` member functions.
- 2192: Validity and return type of `std::abs(0u)` is unclear** Move all declarations to a common header and remove the generic `abs` which accepted unsigned arguments.
- 2196: Specification of `is_*[copy/move]_[constructible/assignable]` unclear for non-referencable types** Use the referenceable type concept.
-

- 2212: tuple_size for const pair request <tuple> header** The tuple_size and tuple_element partial specializations are defined in <utility> which is included by <array>.
- 2296: std::addressof should be constexpr** Use __builtin_addressof and add constexpr to addressof for C++17 and later.
- 2306: match_results::reference should be value_type&, not const value_type&** Change typedef.
- 2313: tuple_size should always derive from integral_constant<size_t, N>** Update definitions of the partial specializations for const and volatile types.
- 2328: Rvalue stream extraction should use perfect forwarding** Use perfect forwarding for right operand.
- 2329: regex_match()/regex_search() with match_results should forbid temporary strings** Add deleted overloads for rvalue strings.
- 2332: regex_iterator/regex_token_iterator should forbid temporary regexes** Add deleted constructors.
- 2332: Unnecessary copying when inserting into maps with braced-init syntax** Add overloads of insert taking value_type&& rvalues.
- 2399: shared_ptr's constructor from unique_ptr should be constrained** Constrain the constructor to require convertibility.
- 2400: shared_ptr's get_deleter() should use addressof()** Use addressof.
- 2401: std::function needs more noexcept** Add noexcept to the assignment and comparisons.
- 2407: packaged_task(allocator_arg_t, const Allocator&, F&&) should neither be constrained nor explicit** Remove explicit from the constructor.
- 2408: SFINAE-friendly common_type/iterator_traits is missing in C++14** Make iterator_traits empty if any of the types is not present in the iterator. Make common_type<> empty.
- 2415: Inconsistency between unique_ptr and shared_ptr** Create empty an shared_ptr from an empty unique_ptr.
- 2418: apply does not work with member pointers** Use mem_fn for member pointers.
- 2440: seed_seq::size() should be noexcept** Add noexcept.
- 2441: Exact-width atomic typedefs should be provided** Define the typedefs.
- 2442: call_once() shouldn't DECAY_COPY()** Remove indirection through call wrapper that made copies of arguments and forward arguments straight to std::invoke.
- 2454: Add raw_storage_iterator::base() member** Add the base() member function.
- 2455: Allocator default construction should be allowed to throw** Make noexcept specifications conditional.
- 2458: N3778 and new library deallocation signatures** Remove unused overloads.
- 2459: std::polar should require a non-negative rho** Add debug mode assertion.
- 2465: SFINAE-friendly common_type is nearly impossible to specialize correctly and regresses key functionality** Detect whether decay_t changes either type and use the decayed types if so.
- 2466: allocator_traits::max_size() default behavior is incorrect** Divide by the object type.
- 2484: rethrow_if_nested() is doubly unimplementable** Avoid using dynamic_cast when it would be ill-formed.
- 2487: bind() should be const-overloaded not cv-overloaded** Deprecate volatile-qualified operator() for C++17, make it ill-formed for C++20.
- 2499: operator>>(basic_istream&, CharT*) makes it hard to avoid buffer overflows** Replace operator>>(basic_istream&, CharT*) and other overloads writing through pointers.

- 2537: Constructors for `priority_queue` taking allocators should call `make_heap`** Call `make_heap`.
- 2566: Requirements on the first template parameter of container adaptors** Add static assertions to enforce the requirement.
- 2583: There is no way to supply an allocator for `basic_string(str, pos)`** Add new constructor.
- 2586: Wrong value category used in `scoped_allocator_adaptor::construct()`** Change internal helper for uses-allocator construction to always check using const lvalue allocators.
- 2684: `priority_queue` lacking comparator typedef** Define the `value_compare` typedef.
- 2735: `std::abs(short)`, `std::abs(signed char)` and others should return `int` instead of `double` in order to be com**
Resolved by the changes for [2192](#).
- 2770: `tuple_size<const T>` specialization is not SFINAE compatible and breaks decomposition declarations** Safely detect `tuple_size<T>::value` and only use it if valid.
- 2781: Contradictory requirements for `std::function` and `std::reference_wrapper`** Remove special handling for `reference_wrapper` arguments and store them directly as the target object.
- 2802: Add `noexcept` to several `shared_ptr` related functions** Add `noexcept`.
- 2873: `shared_ptr` constructor requirements for a deleter** Use rvalues for deleters.
- 2921: `packaged_task` and type-erased allocators** For C++17 mode, remove the constructors taking an allocator argument.
- 2942: LWG 2873's resolution missed `weak_ptr::owner_before`** Add `noexcept`.
- 2996: Missing rvalue overloads for `shared_ptr` operations** Add additional constructor and cast overloads.
- 2993: `reference_wrapper<T>` conversion from `T&&`** Replaced the constructors with a constrained template, to prevent participation in overload resolution when not valid.
- 3074: Non-member functions for `valarray` should only deduce from the `valarray`** Change scalar operands to be non-deduced context, so that they will allow conversions from other types to the `value_type`.
- 3076: `basic_string` CTAD ambiguity** Change constructors to constrained templates.
- 3096: `path::lexically_relative` is confused by trailing slashes** Implement the fix for trailing slashes.
- 3656: Inconsistent bit operations returning a count** Changed `bit_width` to return `int`.
-

Chapter 2

Setup

Transforming libstdc++ sources into installed include files and properly built binaries useful for linking to other software is done as part of building GCC. Building libstdc++ separately from the rest of GCC is not supported.

The general outline of commands to build GCC is something like:

```
get gcc sources
extract into gccsrcdir
mkdir gccbuilddir
cd gccbuilddir
gccsrcdir/configure --prefix=destdir --other-opts...
make
make check
make install
```

Each step is described in more detail in the following sections.

2.1 Prerequisites

Because libstdc++ is part of GCC, the primary source for installation instructions is [the GCC install page](#). In particular, list of prerequisite software needed to build the library [starts with those requirements](#). The same pages also list the tools you will need if you wish to modify the source.

Additional data is given here only where it applies to libstdc++.

To take full advantage of useful space-saving features and bug-fixes you should use a recent binutils whenever possible. The configure process will automatically detect and use these features if the underlying support is present.

To generate the API documentation from the sources you will need Doxygen, see [Documentation Hacking](#) in the appendix for full details.

Finally, a few system-specific requirements:

linux The 'gnu' locale model makes use of `iconv` for character set conversions. The relevant functions are provided by Glibc and so are always available, however they can also be provided by the separate GNU libiconv library. If GNU libiconv is found when GCC is built (e.g., because its headers are installed in `/usr/local/include`) then the `libstdc++.so.6` library will have a run-time dependency on `libiconv.so.2`. If you do not want that run-time dependency then you should do one of the following:

- Uninstall the libiconv headers before building GCC. Glibc already provides `iconv` so you should not need libiconv anyway.
- [Download](#) the libiconv sources and extract them into the top level of the GCC source tree, e.g.,

```
wget https://ftp.gnu.org/pub/gnu/libiconv/libiconv-1.16.tar.gz
tar xf libiconv-1.16.tar.gz
ln -s libiconv-1.16 libiconv
```

This will build libiconv as part of building GCC and link to it statically, so there is no `libiconv.so.2` dependency.

- Configure GCC with `--with-libiconv-type=static`. This requires the static `libiconv.a` library, which is not installed by default. You might need to reinstall libiconv using the `--enable-static` configure option to get the static library.

If GCC 3.1.0 or later on is being used on GNU/Linux, an attempt will be made to use "C" library functionality necessary for C++ named locale support, e.g. the `newlocale` and `uselocale` functions. For GCC 4.6.0 and later, this means that glibc 2.3 or later is required.

If the 'gnu' locale model is being used, the following locales are used and tested in the libstdc++ testsuites. The first column is the name of the locale, the second is the character set it is expected to use.

de_DE	ISO-8859-1
de_DE@euro	ISO-8859-15
en_GB	ISO-8859-1
en_HK	ISO-8859-1
en_PH	ISO-8859-1
en_US	ISO-8859-1
en_US.ISO-8859-1	ISO-8859-1
en_US.ISO-8859-15	ISO-8859-15
en_US.UTF-8	UTF-8
es_ES	ISO-8859-1
es_MX	ISO-8859-1
fr_FR	ISO-8859-1
fr_FR@euro	ISO-8859-15
is_IS	UTF-8
it_IT	ISO-8859-1
ja_JP.eucjp	EUC-JP
ru_RU.ISO-8859-5	ISO-8859-5
ru_RU.UTF-8	UTF-8
se_NO.UTF-8	UTF-8
ta_IN	UTF-8
zh_TW	BIG5

Failure to have installed the underlying "C" library locale information for any of the above regions means that the corresponding C++ named locale will not work: because of this, the libstdc++ testsuite will skip named locale tests which need missing information. If this isn't an issue, don't worry about it. If a named locale is needed, the underlying locale information must be installed. Note that rebuilding libstdc++ after "C" locales are installed is not necessary.

To install support for locales, do only one of the following:

- install all locales, e.g., run `dnf install glibc-all-langpacks` for Fedora and related distributions.
- install just the necessary locales
 - with Debian GNU/Linux:
Add the above list, as shown, to the file `/etc/locale.gen`
`run /usr/sbin/locale-gen`
 - on most Unix-like operating systems:
`localedef -i de_DE -f ISO-8859-1 de_DE`
(repeat for each entry in the above list)
 - Instructions for other operating systems solicited.

Some tests for the `std::messages` facet require a message catalog created by the **msgfmt** utility. That is usually installed as part of the GNU gettext library. If **msgfmt** is not available, some tests under the `22_locale/messages` directory will fail.

- enable-long-long** The long long type was introduced in C99. It is provided as a GNU extension to C++98 in g++. This flag builds support for "long long" into the library (specialized templates and the like for iostreams). This option is on by default: if enabled, users will have to either use the new-style "C" headers by default (i.e., `<cmath>` not `<math.h>`) or add appropriate compile-time flags to all compile lines to allow "C" visibility of this feature (on GNU/Linux, the flag is `-D_ISOC99_SOURCE`, which is added automatically via `CPLUSPLUS_CPP_SPEC`'s addition of `_GNU_SOURCE`). This option can change the library ABI.
- enable-fully-dynamic-string** This option enables a special version of `basic_string` avoiding the optimization that allocates empty objects in static memory. Mostly useful together with shared memory allocators, see PR libstdc++/16612 for details.
- enable-concept-checks** This turns on additional compile-time checks for instantiated library templates, in the form of specialized templates described in the [Concept Checking](#) section. They can help users discover when they break the rules of the STL, before their programs run. These checks are based on C++03 rules and some of them are not compatible with correct C++11 code.
- enable-symvers[=style]** In 3.1 and later, tries to turn on symbol versioning in the shared library (if a shared library has been requested). Values for 'style' that are currently supported are 'gnu', 'gnu-versioned-namespace', 'darwin', 'darwin-export', and 'sun'. Both gnu- options require that a recent version of the GNU linker be in use. Both darwin options are equivalent. With no style given, the configure script will try to guess correct defaults for the host system, probe to see if additional requirements are necessary and present for activation, and if so, will turn symbol versioning on. This option can change the library ABI.
- enable-libstdcxx-visibility** In 4.2 and later, enables or disables visibility attributes. If enabled (as by default), and the compiler seems capable of passing the simple sanity checks thrown at it, adjusts items in namespace `std`, namespace `std::tr1`, namespace `std::tr2`, and namespace `__gnu_cxx` to have `visibility ("default")` so that `-fvisibility options` can be used without affecting the normal external-visibility of namespace `std` entities. Prior to 4.7 this option was spelled `--enable-visibility`.
- enable-libstdcxx-pch** In 3.4 and later, tries to turn on the generation of `stdc++.h.gch`, a pre-compiled file including all the standard C++ includes. If enabled (as by default), and the compiler seems capable of passing the simple sanity checks thrown at it, try to build `stdc++.h.gch` as part of the make process. In addition, this generated file is used later on (by appending `-include bits/stdc++.h` to `CXXFLAGS`) when running the testsuite.
- enable-extern-template[default]** Use extern template to pre-instantiate all required specializations for certain types defined in the standard libraries. These types include `string` and dependents like `char_traits`, the templated IO classes, `allocator`, and others. Disabling means that implicit template generation will be used when compiling these types. By default, this option is on. This option can change the library ABI.
- disable-hosted-libstdcxx** By default, a complete *hosted* C++ library is built. The C++ Standard also describes a *freestanding* environment, in which only a minimal set of headers are provided. This option builds such an environment. Note that a hosted library installs headers that still can be used in non hosted environments, as the library checks for `__STDC_HOSTED__`, however, a library configured with `--disable-hosted-libstdcxx` will not install unusable headers.
- disable-libstdcxx-hosted** This is an alias for `--disable-hosted-libstdcxx`.
- disable-libstdcxx-verbose** By default, the library is configured to write descriptive messages to standard error for certain events such as calling a pure virtual function or the invocation of the standard terminate handler. Those messages cause the library to depend on the demangler and standard I/O facilities, which might be undesirable in a low-memory environment or when standard error is not available. This option disables those messages. This option does not change the library ABI.
- disable-libstdcxx-dual-abi** Disable support for the new, C++11-conforming implementations of `std::string`, `std::list` etc. so that the library only provides definitions of types using the old ABI (see [Dual ABI](#)). This option changes the library ABI.
- with-default-libstdcxx-abi=OPTION** Set the default value for the `_GLIBCXX_USE_CXX11_ABI` macro (see [Macros](#)). The default is `OPTION=new` which sets the macro to 1, use `OPTION=gcc4-compatible` to set it to 0. This option does not change the library ABI.

- with-libstdcxx-lock-policy=OPTION** Sets the lock policy that controls how `shared_ptr` reference counting is synchronized. The choice `OPTION=atomic` enables use of atomics for updates to `shared_ptr` reference counts. The choice `OPTION=mutex` enables use of a mutex to synchronize updates to `shared_ptr` reference counts. If the compiler's thread model is "single" then this option has no effect, as no synchronization is used for the reference counts. The default is `OPTION=auto`, which checks for the availability of compiler built-ins for 2-byte and 4-byte atomic compare-and-swap, and uses `OPTION=atomic` if they're available, `OPTION=mutex` otherwise. This option can change the library ABI. If the library is configured to use atomics and user programs are compiled using a target that doesn't natively support the atomic operations (e.g. the library is configured for `armv7` and then code is compiled with `-march=armv5t`) then the program might rely on support in `libgcc` to provide the atomics.
- enable-vtable-verify** Use `-fvtable-verify=std` to compile the C++ runtime with instrumentation for vtable verification. All virtual functions in the standard library will be verified at runtime. Types impacted include `locale` and `iostream`, and others. Disabling means that the C++ runtime is compiled without support for vtable verification. By default, this option is off.
- enable-libstdcxx-file-system-ts[default]** Build `libstdc++fs.a` as well as the usual `libstdc++` and `libsupc++` libraries. This is enabled by default on select POSIX targets where it is known to work and disabled otherwise.
- enable-libstdcxx-static-eh-pool** Use a fixed-size static buffer for the emergency exception handling pool (see [Memory allocation for exceptions](#)). The default is to allocate the pool on program startup using `malloc`. With this option, a static buffer will be provided by `libstdc++` instead. This does not change the library ABI.
- with-libstdcxx-eh-pool-obj-count=NUM** Set the size of the emergency exception handling pool. `NUM` is the number of simultaneous allocated exceptions to support. This does not change the library ABI.
- with-libstdcxx-zoneinfo=OPTION** Choose how `std::chrono::tzdb` will obtain the time zone info. The library requires a copy of the `tzdata.zi` and `leapseconds` files from the [IANA Time Zone Database](#). The choice `OPTION=static` will embed a copy of the files into the library, and use that static data when time zone information is required. The choice `OPTION=dir` will use the files `dir/tzdata.zi` and `dir/leapseconds` (which must exist when a program tries to access time zone information). The choice `OPTION=dir,static` will try to use files in `dir` but if they are not available the embedded static data will be used instead. The default choice is `OPTION=yes`. This is equivalent to `OPTION=dir,static` with a system-specific default directory (if a suitable default for the target is known). The choice `OPTION=no` will disable all code for loading time zone info from file or from the embedded static data, which means that only the "UTC" and "GMT" time zones are defined. Using `OPTION=no` results in a smaller library, so is suitable for systems that will never need to query the time zone database. This does not change the library ABI.

2.3 Make

If you have never done this before, you should read the basic [GCC Installation Instructions](#) first. Read *all of them*. *Twice*.

Then type: **make**, and congratulations, you've started to build.

Chapter 3

Using

3.1 Command Options

The set of features available in the GNU C++ library is shaped by several [GCC Command Options](#). Options that impact `libstdc++` are enumerated and detailed in the table below.

The standard library conforms to the dialect of C++ specified by the `-std` option passed to the compiler. By default, `g++` is equivalent to `g++ -std=gnu++17` since GCC 11, and `g++ -std=gnu++14` in GCC 6, 7, 8, 9, and 10, and `g++ -std=gnu++98` for older releases.

3.2 Headers

3.2.1 Header Files

The C++ standard specifies the entire set of header files that must be available to all hosted implementations. Actually, the word "files" is a misnomer, since the contents of the headers don't necessarily have to be in any kind of external file. The only rule is that when one `#includes` a header, the contents of that header become available, no matter how.

That said, in practice files are used.

There are two main types of include files: header files related to a specific version of the ISO C++ standard (called Standard Headers), and all others (TS, TR1, C++ ABI, and Extensions).

Multiple dialects of standard headers are supported, corresponding to the 1998 standard as updated for 2003, the 2011 standard, the 2014 standard, and so on.

Table [3.2](#) and Table [3.3](#) and Table [3.4](#) show the C++98/03 include files. These are available in the C++98 compilation mode, i.e. `-std=c++98` or `-std=gnu++98`. Unless specified otherwise below, they are also available in later modes (C++11, C++14 etc).

The following header is deprecated and might be removed from a future C++ standard.

Table [3.5](#) and Table [3.6](#) show the C++11 include files. These are available in C++11 compilation mode, i.e. `-std=c++11` or `-std=gnu++11`. Including these headers in C++98/03 mode may result in compilation errors. Unless specified otherwise below, they are also available in later modes (C++14 etc).

Table [3.7](#) shows the C++14 include file. This is available in C++14 compilation mode, i.e. `-std=c++14` or `-std=gnu++14`. Including this header in C++98/03 mode or C++11 will not result in compilation errors, but will not define anything. Unless specified otherwise below, it is also available in later modes (C++17 etc).

Table [3.8](#) shows the C++17 include files. These are available in C++17 compilation mode, i.e. `-std=c++17` or `-std=gnu++17`. Including these headers in earlier modes will not result in compilation errors, but will not define anything. Unless specified otherwise below, they are also available in later modes (C++20 etc).

Option Flags	Description
<code>-std</code>	Select the C++ standard, and whether to use the base standard or GNU dialect.
<code>-fno-exceptions</code>	See exception-free dialect
<code>-fno-rtti</code>	As above, but RTTI-free dialect.
<code>-pthread</code>	For ISO C++11 <code><thread></code> , <code><future></code> , <code><mutex></code> , or <code><condition_variable></code> .
<code>-latomic</code>	Linking to <code>libatomic</code> is required for some uses of ISO C++11 <code><atomic></code> .
<code>-lstdc++exp</code>	Linking to <code>libstdc++exp.a</code> is required for use of experimental C++ library features. This currently provides support for the C++23 types defined in the <code><stacktrace></code> header, the Filesystem library extensions defined in the <code><experimental/filesystem></code> header, and the Contracts extensions enabled by <code>-fcontracts</code> .
<code>-lstdc++fs</code>	Linking to <code>libstdc++fs.a</code> is another way to use the Filesystem library extensions defined in the <code><experimental/filesystem></code> header. The <code>libstdc++exp.a</code> library also provides all the symbols contained in this library.
<code>-fopenmp</code>	For parallel mode.
<code>-ltbb</code>	Linking to <code>tbb</code> (Thread Building Blocks) is required for use of the Parallel Standard Algorithms and execution policies in <code><execution></code> .
<code>-ffreestanding</code>	Limits the library to its freestanding subset. Headers that are not supported in freestanding will emit a "This header is not available in freestanding mode" error. Headers that are in the freestanding subset partially will not expose functionality that is not part of the freestanding subset.

Table 3.1: C++ Command Options

<code>algorithm</code>	<code>bitset</code>	<code>complex</code>	<code>deque</code>	<code>exception</code>
<code>fstream</code>	<code>functional</code>	<code>iomanip</code>	<code>ios</code>	<code>iosfwd</code>
<code>iostream</code>	<code>istream</code>	<code>iterator</code>	<code>limits</code>	<code>list</code>
<code>locale</code>	<code>map</code>	<code>memory</code>	<code>new</code>	<code>numeric</code>
<code>ostream</code>	<code>queue</code>	<code>set</code>	<code>sstream</code>	<code>stack</code>
<code>stdexcept</code>	<code>streambuf</code>	<code>string</code>	<code>utility</code>	<code>typeinfo</code>
<code>valarray</code>	<code>vector</code>			

Table 3.2: C++ 1998 Library Headers

<code>cassert</code>	<code>cerrno</code>	<code>cctype</code>	<code>cfloat</code>	<code>ciso646</code>
<code>climits</code>	<code>ctype</code>	<code>cmath</code>	<code>csetjmp</code>	<code>csignal</code>
<code>cstdarg</code>	<code>cstddef</code>	<code>cstdio</code>	<code>cstdlib</code>	<code>cstring</code>
<code>ctime</code>	<code>wchar</code>	<code>wctype</code>		

Table 3.3: C++ 1998 Library Headers for C Library Facilities

<code>strstream</code>

Table 3.4: C++ 1998 Deprecated Library Header

array	atomic	chrono	codecvt	condition_variable
forward_list	future	initializer_list	mutex	random
ratio	regex	scoped_allocator	system_error	thread
tuple	typeindex	type_traits	unordered_map	unordered_set

Table 3.5: C++ 2011 Library Headers

ccomplex	cfenv	cinttypes	cstdalign	cstdbool
cstdint	ctgmath	cuchar		

Table 3.6: C++ 2011 Library Headers for C Library Facilities

shared_mutex

Table 3.7: C++ 2014 Library Header

any	charconv	execution	filesystem	memory_resource
optional	string_view	variant		

Table 3.8: C++ 2017 Library Headers

barrier	bit	charconv	compare	concepts
coroutine	format	latch	numbers	ranges
semaphore	source_location	span	stop_token	syncstream
version				

Table 3.9: C++ 2020 Library Headers

Table 3.9 shows the C++20 include files. These are available in C++20 compilation mode, i.e. `-std=c++20` or `-std=gnu++20`. Including these headers in earlier modes will not result in compilation errors, but will not define anything.

The following headers have been removed in the C++20 standard. They are still available when using this implementation, but in future they might start to produce warnings or errors when included in C++20 mode. Programs that intend to be portable should not include them.

<code>ccomplex</code>	<code>ciso646</code>	<code>cstdalign</code>	<code>cstdbool</code>	<code>ctgmath</code>
-----------------------	----------------------	------------------------	-----------------------	----------------------

Table 3.10: C++ 2020 Obsolete Headers

Table 3.11 shows the C++23 include files. These are available in C++23 compilation mode, i.e. `-std=c++23` or `-std=gnu++23`. Including these headers in earlier modes will not result in compilation errors, but will not define anything.

<code>expected</code>	<code>generator</code>	<code>print</code>	<code>spanstream</code>	<code>stacktrace</code>
<code>stdatomic.h</code>	<code>stdfloat</code>			

Table 3.11: C++ 2023 Library Headers

Table 3.12 shows the C++26 include files. These are available in C++26 compilation mode, i.e. `-std=c++26` or `-std=gnu++26`. Including these headers in earlier modes will not result in compilation errors, but will not define anything.

<code>text_encoding</code>

Table 3.12: C++ 2026 Library Headers

Table 3.13, shows the additional include file define by the File System Technical Specification, ISO/IEC TS 18822:2015. This is available in C++11 and later compilation modes. Including this header in earlier modes will not result in compilation errors, but will not define anything.

Table 3.14, shows the additional include files define by the C++ Extensions for Library Fundamentals Technical Specification, ISO/IEC TS 19568:2015, ISO/IEC TS 19568:2017, and ISO/IEC TS 19568:2024. These are available in C++14 and later compilation modes, except for `<experimental/scope>` which is available in C++20 and later compilation modes. Including these headers in earlier modes will not result in compilation errors, but will not define anything.

Table 3.15, shows the additional include files define by the Networking Technical Specification, ISO/IEC TS 19216:2018. These are available in C++14 and later compilation modes. Including these headers in earlier modes will not result in compilation errors, but will not define anything.

In addition, TR1 includes as:

Decimal floating-point arithmetic is available if the C++ compiler supports scalar decimal floating-point types defined via `__attribute__((mode(SD|DD|LD)))`.

Also included are files for the C++ ABI interface:

And a large variety of extensions.

3.2.2 Mixing Headers

A few simple rules.

First, mixing different dialects of the standard headers is not possible. It's an all-or-nothing affair. Thus, code like

```
#include <array>
#include <functional>
```

Implies C++11 mode. To use the entities in `<array>`, the C++11 compilation mode must be used, which implies the C++11 functionality (and deprecations) in `<functional>` will be present.

experimental/filesystem

Table 3.13: File System TS Header

experimental/ algorithm	experimental/ any	experimental/ array	experimental/ chrono	experimental/ deque
experimental/ forward_list	experimental/ functional	experimental/ iterator	experimental/ list	experimental/ map
experimental/ memory	experimental/ memory_ resource	experimental/ numeric	experimental/ optional	experimental/ propagate_ const
experimental/ random	experimental/ ratio	experimental/ regex	experimental/ scope	experimental/ set
experimental/ source_ location	experimental/ string	experimental/ string_view	experimental/ system_error	experimental/ tuple
experimental/ type_traits	experimental/ unordered_map	experimental/ unordered_set	experimental/ utility	experimental/ vector

Table 3.14: Library Fundamentals TS Headers

experimental/ buffer	experimental/ executor	experimental/ internet	experimental/io_ context
experimental/net	experimental/ netfwd	experimental/ socket	experimental/timer

Table 3.15: Networking TS Headers

tr1/array	tr1/complex	tr1/memory	tr1/functional	tr1/random
tr1/regex	tr1/tuple	tr1/type_ traits	tr1/unordered_ map	tr1/unordered_ set
tr1/utility				

Table 3.16: C++ TR 1 Library Headers

tr1/complex	tr1/cfenv	tr1/cfloat	tr1/cmath	tr1/cinttypes
tr1/climits	tr1/cstdarg	tr1/cstdbool	tr1/cstdint	tr1/cstdio
tr1/cstdlib	tr1/ctgmth	tr1/ctime	tr1/cwchar	tr1/cwctype

Table 3.17: C++ TR 1 Library Headers for C Library Facilities

decimal/decimal

Table 3.18: C++ TR 24733 Decimal Floating-Point Header

cxxabi.h	cxxabi_forced.h
----------	-----------------

Table 3.19: C++ ABI Headers

ext/algorithm	ext/atomicity.h	ext/bitmap_allocator.h	ext/cast.h	
ext/codecv_t_specializations.h	ext/concurrence.h	ext/debug_allocator.h	ext/enc_filebuf.h	ext/extptr_allocator.h
ext/functional	ext/iterator	ext/malloc_allocator.h	ext/memory	ext/mt_allocator.h
ext/new_allocator.h	ext/numeric	ext/numeric_traits.h	ext/pb_ds/assoc_container.h	ext/pb_ds/priority_queue.h
ext/pod_char_traits.h	ext/pool_allocator.h	ext/rb_tree	ext/rope	ext/slist
ext/stdio_filebuf.h	ext/stdio_sync_filebuf.h	ext/throw_allocator.h	ext/typelist.h	ext/type_traits.h
ext/vstring.h				

Table 3.20: Extension Headers

debug/array	debug/bitset	debug/deque	debug/forward_list	debug/list
debug/map	debug/set	debug/string	debug/unordered_map	debug/unordered_set
debug/vector				

Table 3.21: Extension Debug Headers

parallel/algorithm	parallel/numeric
--------------------	------------------

Table 3.22: Extension Parallel Headers

Second, the other headers can be included with either dialect of the standard headers, although features and types specific to C++11 are still only enabled when in C++11 compilation mode. So, to use rvalue references with `__gnu_cxx::vstring`, or to use the debug-mode versions of `std::unordered_map`, one must use the `std=gnu++11` compiler flag. (Or `std=c++11`, of course.)

A special case of the second rule is the mixing of TR1 and C++11 facilities. It is possible (although not especially prudent) to include both the TR1 version and the C++11 version of header in the same translation unit:

```
#include <tr1/type_traits>
#include <type_traits>
```

Several parts of C++11 diverge quite substantially from TR1 predecessors.

3.2.3 The C Headers and namespace `std`

The standard specifies that if one includes the C-style header (`<math.h>` in this case), the symbols will be available in the global namespace and perhaps in namespace `std::` (but this is no longer a firm requirement.) On the other hand, including the C++-style header (`<cmath>`) guarantees that the entities will be found in namespace `std` and perhaps in the global namespace.

Usage of C++-style headers is recommended, as then C-linkage names can be disambiguated by explicit qualification, such as by `std::abort`. In addition, the C++-style headers can use function overloading to provide a simpler interface to certain families of C-functions. For instance in `<cmath>`, the function `std::sin` has overloads for all the builtin floating-point types. This means that `std::sin` can be used uniformly, instead of a combination of `std::sinf`, `std::sin`, and `std::sinl`.

3.2.4 Precompiled Headers

There are three base header files that are provided. They can be used to precompile the standard headers and extensions into binary files that may then be used to speed up compilations that use these headers.

- `stdc++.h`
Includes all standard headers. Actual content varies depending on **language dialect**.
- `stdtr1c++.h`
Includes all of `<stdc++.h>`, and adds all the TR1 headers.
- `extc++.h`
Includes all of `<stdc++.h>`, and adds all the Extension headers (and in C++98 mode also adds all the TR1 headers by including all of `<stdtr1c++.h>`).

To construct a `.gch` file from one of these base header files, first find the include directory for the compiler. One way to do this is:

```
g++ -v hello.cc

#include <...> search starts here:
 /mnt/share/bld/H-x86-gcc.20071201/include/c++/4.3.0
...
End of search list.
```

Then, create a precompiled header file with the same flags that will be used to compile other projects.

```
g++ -Winvalid-pch -x c++-header -g -O2 -o ./stdc++.h.gch /mnt/share/bld/H-x86-gcc.20071201/ ↵
include/c++/4.3.0/x86_64-unknown-linux-gnu/bits/stdc++.h
```

The resulting file will be quite large: the current size is around thirty megabytes.

How to use the resulting file.

```
g++ -I. -include stdc++.h -H -g -O2 hello.cc
```


- `_GLIBCXX_CONCEPT_CHECKS`** Undefined by default. Configurable via `--enable-concept-checks`. When defined, performs compile-time checking on certain template instantiations to detect violations of the requirements of the standard. This macro has no effect for freestanding implementations. This is described in more detail in [Compile Time Checks](#).
- `_GLIBCXX_ASSERTIONS`** Defined by default when compiling with no optimization, undefined by default when compiling with optimization. When defined, enables extra error checking in the form of precondition assertions, such as bounds checking in strings and null pointer checks when dereferencing smart pointers.
- `_GLIBCXX_NO_ASSERTIONS`** Undefined by default. When defined, prevents the implicit definition of `_GLIBCXX_ASSERTIONS` when compiling with no optimization.
- `_GLIBCXX_DEBUG`** Undefined by default. When defined, compiles user code using the [debug mode](#). When defined, `_GLIBCXX_ASSERTIONS` is defined automatically, so all the assertions enabled by that macro are also enabled in debug mode.
- `_GLIBCXX_DEBUG_PEDANTIC`** Undefined by default. When defined while compiling with the [debug mode](#), makes the debug mode extremely picky by making the use of libstdc++ extensions and libstdc++-specific behavior into errors.
- `_GLIBCXX_DEBUG_BACKTRACE`** Undefined by default. Considered only if libstdc++ has been configured with `--enable-libstdc++backtrace` and if `_GLIBCXX_DEBUG` is defined. When defined display backtraces on [debug mode](#) assertions.
- `_GLIBCXX_PARALLEL`** Undefined by default. When defined, compiles user code using the [parallel mode](#).
- `_GLIBCXX_PARALLEL_ASSERTIONS`** Undefined by default, but when any parallel mode header is included this macro will be defined to a non-zero value if `_GLIBCXX_ASSERTIONS` has a non-zero value, otherwise to zero. When defined to a non-zero value, it enables extra error checking and assertions in the parallel mode.
- `_STDCPP_WANT_MATH_SPEC_FUNCS`** Undefined by default. When defined to a non-zero integer constant, enables support for ISO/IEC 29124 Special Math Functions.
- `_GLIBCXX_SANITIZE_VECTOR`** Undefined by default. When defined, `std::vector` operations will be annotated so that AddressSanitizer can detect invalid accesses to the unused capacity of a `std::vector`. These annotations are only enabled for `std::vector<T, std::allocator<T>>` and only when `std::allocator` is derived from [new_allocator](#) or [malloc_allocator](#). The annotations must be present on all vector operations or none, so this macro must be defined to the same value for all translation units that create, destroy, or modify vectors.
- `_GLIBCXX_NO_FREESTANDING_CHRONO`** Undefined by default. When defined, the `<chrono>` header cannot be used with `-ffreestanding`. When not defined, durations, time points, and calendar types are available for freestanding, but the standard clocks and the time zone database are not (because they require OS support).

3.4 Dual ABI

In the GCC 5.1 release libstdc++ introduced a new library ABI that includes new implementations of `std::string` and `std::list`. These changes were necessary to conform to the 2011 C++ standard which forbids Copy-On-Write strings and requires lists to keep track of their size.

In order to maintain backwards compatibility for existing code linked to libstdc++ the library's soname has not changed and the old implementations are still supported in parallel with the new ones. This is achieved by defining the new implementations in an inline namespace so they have different names for linkage purposes, e.g. the new version of `std::list<int>` is actually defined as `std::__cxx11::list<int>`. Because the symbols for the new implementations have different names the definitions for both versions can be present in the same library.

The `_GLIBCXX_USE_CXX11_ABI` macro (see [Macros](#)) controls whether the declarations in the library headers use the old or new ABI. So the decision of which ABI to use can be made separately for each source file being compiled. Using the default configuration options for GCC the default value of the macro is 1 which causes the new ABI to be active, so to use the old ABI you must explicitly define the macro to 0 before including any library headers. (Be aware that some GNU/Linux distributions configured GCC 5 differently so that the default value of the macro is 0 and users must define it to 1 to enable the new ABI.)

Although the changes were made for C++11 conformance, the choice of ABI to use is independent of the `-std` option used to compile your code, i.e. for a given GCC build the default value of the `_GLIBCXX_USE_CXX11_ABI` macro is the same for

all dialects. This ensures that the `-std` does not change the ABI, so that it is straightforward to link C++03 and C++11 code together.

Because `std::string` is used extensively throughout the library a number of other types are also defined twice, including the stringstream classes and several facets used by `std::locale`. The standard facets which are always installed in a locale may be present twice, with both ABIs, to ensure that code like `std::use_facet<std::time_get<char>>(locale);` will work correctly for both `std::time_get` and `std::__cxx11::time_get` (even if a user-defined facet that derives from one or other version of `time_get` is installed in the locale).

Although the standard exception types defined in `<stdexcept>` use strings, most are not defined twice, so that a `std::out_of_range` exception thrown in one file can always be caught by a suitable handler in another file, even if the two files are compiled with different ABIs.

One exception type does change when using the new ABI, namely `std::ios_base::failure`. This is necessary because the 2011 standard changed its base class from `std::exception` to `std::system_error`, which causes its layout to change. Exceptions due to iostream errors are thrown by a function inside `libstdc++.so`, so whether the thrown exception uses the old `std::ios_base::failure` type or the new one depends on the ABI that was active when `libstdc++.so` was built, *not* the ABI active in the user code that is using iostreams. This means that for a given build of GCC the type thrown is fixed. In current releases the library throws a special type that can be caught by handlers for either the old or new type, but for GCC 7.1, 7.2 and 7.3 the library throws the new `std::ios_base::failure` type, and for GCC 5.x and 6.x the library throws the old type. Catch handlers of type `std::ios_base::failure` will only catch the exceptions if using a newer release, or if the handler is compiled with the same ABI as the type thrown by the library. Handlers for `std::exception` will always catch iostreams exceptions, because the old and new type both inherit from `std::exception`.

Some features are not supported when using the old ABI, including:

- Using `std::string::const_iterator` for positional arguments to member functions such as `std::string::erase`.
- Allocator propagation in `std::string`.
- Using `std::string` at compile-time in `constexpr` functions.
- Class `std::chrono::time_zone` and all related APIs.
- The `<syncstream>` header.

3.4.1 Troubleshooting

If you get linker errors about undefined references to symbols that involve types in the `std::__cxx11` namespace or the tag `[abi:cxx11]` then it probably indicates that you are trying to link together object files that were compiled with different values for the `_GLIBCXX_USE_CXX11_ABI` macro. This commonly happens when linking to a third-party library that was compiled with an older version of GCC. If the third-party library cannot be rebuilt with the new ABI then you will need to recompile your code with the old ABI.

Not all uses of the new ABI will cause changes in symbol names, for example a class with a `std::string` member variable will have the same mangled name whether compiled with the old or new ABI. In order to detect such problems the new types and functions are annotated with the `abi_tag` attribute, allowing the compiler to warn about potential ABI incompatibilities in code using them. Those warnings can be enabled with the `-Wabi-tag` option.

3.5 Namespaces

3.5.1 Available Namespaces

There are three main namespaces.

- `std`

The ISO C++ standards specify that "all library entities are defined within namespace `std`." This includes namespaces nested within namespace `std`, such as namespace `std::chrono`.

- `abi`

Specified by the C++ ABI. This ABI specifies a number of type and function APIs supplemental to those required by the ISO C++ Standard, but necessary for interoperability.

- `__gnu__`

Indicating one of several GNU extensions. Choices include `__gnu_cxx`, `__gnu_debug`, `__gnu_parallel`, and `__gnu_pbds`.

The library uses a number of inline namespaces as implementation details that are not intended for users to refer to directly, these include `std::__detail`, `std::__cxx11` and `std::_V2`.

A complete list of implementation namespaces (including namespace contents) is available in the generated source [documentation](#).

3.5.2 namespace std

One standard requirement is that the library components are defined in namespace `std`. Thus, in order to use these types or functions, one must do one of two things:

- put a kind of *using-declaration* in your source (either `using namespace std;` or i.e. `using std::string;`) This approach works well for individual source files, but should not be used in a global context, like header files.
- use a *fully qualified name* for each library symbol (i.e. `std::string`, `std::cout`) Always can be used, and usually enhanced, by strategic use of typedefs. (In the cases where the qualified verbiage becomes unwieldy.)

3.5.3 Using Namespace Composition

Best practice in programming suggests sequestering new data or functionality in a sanely-named, unique namespace whenever possible. This is considered an advantage over dumping everything in the global namespace, as then name look-up can be explicitly enabled or disabled as above, symbols are consistently mangled without repetitive naming prefixes or macros, etc.

For instance, consider a project that defines most of its classes in namespace `gtk`. It is possible to adapt namespace `gtk` to namespace `std` by using a C++-feature called *namespace composition*. This is what happens if a *using-declaration* is put into a namespace-definition: the imported symbol(s) gets imported into the currently active namespace(s). For example:

```
namespace gtk
{
    using std::string;
    using std::tr1::array;

    class Window { ... };
}
```

In this example, `std::string` gets imported into namespace `gtk`. The result is that use of `std::string` inside namespace `gtk` can just use `string`, without the explicit qualification. As an added bonus, `std::string` does not get imported into the global namespace. Additionally, a more elaborate arrangement can be made for backwards compatibility and portability, whereby the *using-declarations* can be wrapped in macros that are set based on autoconf-tests to either `""` or i.e. `using std::string;` (depending on whether the system has `libstdc++` in `std::` or not). (ideas from Llewellyn and Karl Nelson)

3.6 Linking

3.6.1 Almost Nothing

Or as close as it gets: freestanding. This is a minimal configuration, with only partial support for the standard library. Assume only the following header files can be used:

- `cstdarg`
- `cstddef`
- `cstdlib`
- `exception`
- `limits`
- `new`
- `exception`
- `typeinfo`

In addition, throw in

- `cxxabi.h`.

In the C++11 **dialect** add

- `initializer_list`
- `type_traits`

As of GCC 13, `libstdc++` implements P1642, which brings in many more headers, as well a quite a few ones not covered by the paper. In general, if a feature does not require traditionally `libc`-provided facilities, or dynamic memory allocation, it's enabled in the freestanding subset. In addition, if only a subset of a header requires such features, it is partially included. Some examples include:

- `string_view`
- `tuple`
- `bitset`

Currently, this subset includes all of the iterator APIs (including the ranges APIs) that do not involve streams, the entire C++ algorithms library, excluding parallel algorithms, and a large part of the utilities library. This is on top of the headers included in the lists above.

If you're using a `libstdc++` configured for hosted environments, and would like to not involve the libraries `libstdc++` would depend on in your programs, you will need to use **gcc** to link your application with only `libsupc++.a`, like so:

gcc -ffreestanding foo.cc -lsupc++

If you configured `libstdc++` with `--disable-hosted-libstdcxx`, however, you can use the normal **g++** command to link, as this configuration provides a (nearly) empty `libstdc++.a`.

3.6.2 Finding Dynamic or Shared Libraries

If the only library built is the static library (`libstdc++.a`), or if specifying static linking, this section can be skipped. But if building or using a shared library (`libstdc++.so`), then additional location information will need to be provided.

But how?

A quick read of the relevant part of the GCC manual, [Compiling C++ Programs](#), specifies linking against a C++ library. More details from the GCC [FAQ](#), which states *GCC does not, by default, specify a location so that the dynamic linker can find dynamic libraries at runtime*.

Users will have to provide this information.

Methods vary for different platforms and different styles, and are printed to the screen during installation. To summarize:

3.7.2 Thread Safety

In the terms of the 2011 C++ standard a thread-safe program is one which does not perform any conflicting non-atomic operations on memory locations and so does not contain any data races. The standard places requirements on the library to ensure that no data races are caused by the library itself or by programs which use the library correctly (as described below). The C++11 memory model and library requirements are a more formal version of the [SGI STL](#) definition of thread safety, which the library used prior to the 2011 standard.

The library strives to be thread-safe when all of the following conditions are met:

- The system's libc is itself thread-safe,
- The compiler in use reports a thread model other than 'single'. This can be tested via output from `gcc -v`. Multi-thread capable versions of gcc output something like this:

```
%gcc -v
Using built-in specs.
...
Thread model: posix
gcc version 4.1.2 20070925 (Red Hat 4.1.2-33)
```

Look for "Thread model" lines that aren't equal to "single."

- Requisite command-line flags are used for atomic operations and threading. Examples of this include `-pthread` and `-march=native`, although specifics vary depending on the host environment. See [Command Options](#) and [Machine Dependent Options](#).
- An implementation of the `atomicity.h` functions exists for the architecture in question. See the [internals documentation](#) for more details.

The user code must guard against concurrent function calls which access any particular library object's state when one or more of those accesses modifies the state. An object will be modified by invoking a non-const member function on it or passing it as a non-const argument to a library function. An object will not be modified by invoking a const member function on it or passing it to a function as a pointer- or reference-to-const. Typically, the application programmer may infer what object locks must be held based on the objects referenced in a function call and whether the objects are accessed as const or non-const. Without getting into great detail, here is an example which requires user-level locks:

```
library_class_a shared_object_a;

void thread_main () {
    library_class_b *object_b = new library_class_b;
    shared_object_a.add_b (object_b);    // must hold lock for shared_object_a
    shared_object_a.mutate ();           // must hold lock for shared_object_a
}

// Multiple copies of thread_main() are started in independent threads.
```

Under the assumption that `object_a` and `object_b` are never exposed to another thread, here is an example that does not require any user-level locks:

```
void thread_main () {
    library_class_a object_a;
    library_class_b *object_b = new library_class_b;
    object_a.add_b (object_b);
    object_a.mutate ();
}
```

All library types are safe to use in a multithreaded program if objects are not shared between threads or as long each thread carefully locks out access by any other thread while it modifies any object visible to another thread. Unless otherwise documented, the only exceptions to these rules are atomic operations on the types in `<atomic>` and lock/unlock operations on the standard mutex types in `<mutex>`. These atomic operations allow concurrent accesses to the same object without introducing data races.

The following member functions of standard containers can be considered to be `const` for the purposes of avoiding data races: `begin`, `end`, `rbegin`, `rend`, `front`, `back`, `data`, `find`, `lower_bound`, `upper_bound`, `equal_range`, `at` and, except in associative or unordered associative containers, `operator[]`. In other words, although they are non-`const` so that they can return mutable iterators, those member functions will not modify the container. Accessing an iterator might cause a non-modifying access to the container the iterator refers to (for example incrementing a list iterator must access the pointers between nodes, which are part of the container and so conflict with other accesses to the container).

The Copy-On-Write `std::string` implementation used before GCC 5 (and with `_GLIBCXX_USE_CXX11_ABI=0`) is not a standard container and does not conform to the data race avoidance rules described above. For the Copy-On-Write `std::string`, non-`const` member functions such as `begin()` are considered to be modifying accesses and so must not be used concurrently with any other accesses to the same object.

Programs which follow the rules above will not encounter data races in library code, even when using library types which share state between distinct objects. In the example below the `shared_ptr` objects share a reference count, but because the code does not perform any non-`const` operations on the globally-visible object, the library ensures that the reference count updates are atomic and do not introduce data races:

```
std::shared_ptr<int> global_sp;

void thread_main() {
    auto local_sp = global_sp; // OK, copy constructor's parameter is reference-to-const

    int i = *global_sp;        // OK, operator* is const
    int j = *local_sp;         // OK, does not operate on global_sp

    // *global_sp = 2;          // NOT OK, modifies int visible to other threads
    // *local_sp = 2;           // NOT OK, modifies int visible to other threads

    // global_sp.reset();       // NOT OK, reset is non-const
    local_sp.reset();           // OK, does not operate on global_sp
}

int main() {
    global_sp.reset(new int(1));
    std::thread t1(thread_main);
    std::thread t2(thread_main);
    t1.join();
    t2.join();
}
```

For further details of the C++11 memory model see Hans-J. Boehm's [Threads and memory model for C++](#) pages, particularly the [introduction](#) and [FAQ](#).

3.7.3 Atomics

3.7.4 IO

This gets a bit tricky. Please read carefully, and bear with me.

3.7.4.1 Structure

A wrapper type called `__basic_file` provides our abstraction layer for the `std::filebuf` classes. Nearly all decisions dealing with actual input and output must be made in `__basic_file`.

A generic locking mechanism is somewhat in place at the filebuf layer, but is not used in the current code. Providing locking at any higher level is akin to providing locking within containers, and is not done for the same reasons (see the links above).

Since the container implementation of `libstdc++` uses the SGI code, we use the same definition of thread safety as SGI when discussing design. A key point that beginners may miss is the fourth major paragraph of the first page mentioned above (*For most clients...*), which points out that locking must nearly always be done outside the container, by client code (that'd be you, not us). There is a notable exceptions to this rule. Allocators called while a container or element is constructed uses an internal lock obtained and released solely within `libstdc++` code (in fact, this is the reason STL requires any knowledge of the thread configuration).

For implementing a container which does its own locking, it is trivial to provide a wrapper class which obtains the lock (as SGI suggests), performs the container operation, and then releases the lock. This could be templated *to a certain extent*, on the underlying container and/or a locking mechanism. Trying to provide a catch-all general template solution would probably be more trouble than it's worth.

The library implementation may be configured to use the high-speed caching memory allocator, which complicates thread safety issues. For all details about how to globally override this at application run-time see [here](#). Also useful are details on [allocator](#) options and capabilities.

3.8 Exceptions

The C++ language provides language support for stack unwinding with `try` and `catch` blocks and the `throw` keyword.

These are very powerful constructs, and require some thought when applied to the standard library in order to yield components that work efficiently while cleaning up resources when unexpectedly killed via exceptional circumstances.

Two general topics of discussion follow: exception neutrality and exception safety.

3.8.1 Exception Safety

What is exception-safe code?

Will define this as reasonable and well-defined behavior by classes and functions from the standard library when used by user-defined classes and functions that are themselves exception safe.

Please note that using exceptions in combination with templates imposes an additional requirement for exception safety. Instantiating types are required to have destructors that do no throw.

Using the layered approach from Abrahams, can classify library components as providing set levels of safety. These will be called exception guarantees, and can be divided into three categories.

- One. Don't throw.
As specified in 23.2.1 general container requirements. Applicable to container and string classes.
Member functions `erase`, `pop_back`, `pop_front`, `swap`, `clear`. And iterator copy constructor and assignment operator.
- Two. Don't leak resources when exceptions are thrown. This is also referred to as the "basic" exception safety guarantee.
This applicable throughout the standard library.
- Three. Commit-or-rollback semantics. This is referred to as "strong" exception safety guarantee.
As specified in 23.2.1 general container requirements. Applicable to container and string classes.
Member functions `insert` of a single element, `push_back`, `push_front`, and `rehash`.

3.8.2 Exception Neutrality

Simply put, once thrown an exception object should continue in flight unless handled explicitly. In practice, this means propagating exceptions should not be swallowed in gratuitous `catch(...)` blocks. Instead, matching `try` and `catch` blocks should have specific catch handlers and allow un-handled exception objects to propagate. If a terminating `catch(...)` blocks exist then it should end with a `throw` to re-throw the current exception.

3.8.4 Doing without

C++ is a language that strives to be as efficient as is possible in delivering features. As such, considerable care is used by both language implementer and designers to make sure unused features do not impose hidden or unexpected costs. The GNU system tries to be as flexible and as configurable as possible. So, it should come as no surprise that GNU C++ provides an optional language extension, spelled `-fno-exceptions`, as a way to excise the implicitly generated magic necessary to support `try` and `catch` blocks and thrown objects. (Language support for `-fno-exceptions` is documented in the [GCC manual](#).)

Before detailing the library support for `-fno-exceptions`, first a passing note on the things lost when this flag is used: it will break exceptions trying to pass through code compiled with `-fno-exceptions` whether or not that code has any `try` or `catch` constructs. If you might have some code that throws, you shouldn't use `-fno-exceptions`. If you have some code that uses `try` or `catch`, you shouldn't use `-fno-exceptions`.

And what is to be gained, tinkering in the back alleys with a language like this? Exception handling overhead can be measured in the size of the executable binary, and varies with the capabilities of the underlying operating system and specific configuration of the C++ compiler. On recent hardware with GNU system software of the same age, the combined code and data size overhead for enabling exception handling is around 7%. Of course, if code size is of singular concern than using the appropriate optimizer setting with exception handling enabled (ie, `-Os -fexceptions`) may save up to twice that, and preserve error checking.

So. Hell bent, we race down the slippery track, knowing the brakes are a little soft and that the right front wheel has a tendency to wobble at speed. Go on: detail the standard library support for `-fno-exceptions`.

In sum, valid C++ code with exception handling is transformed into a dialect without exception handling. In detailed steps: all use of the C++ keywords `try`, `catch`, and `throw` in the standard library have been permanently replaced with the pre-processor controlled equivalents spelled `__try`, `__catch`, and `__throw_exception_again`. They are defined as follows.

```
#if __cpp_exceptions
# define __try      try
# define __catch(X) catch(X)
# define __throw_exception_again throw
#else
# define __try      if (true)
# define __catch(X) if (false)
# define __throw_exception_again
#endif
```

In addition, for most of the classes derived from class `exception`, there exists a corresponding function with C language linkage. An example:

```
#if __cpp_exceptions
void __throw_bad_exception()
{ throw bad_exception(); }
#else
void __throw_bad_exception()
{ abort(); }
#endif
```

The last language feature needing to be transformed by `-fno-exceptions` is treatment of exception specifications on member functions. Fortunately, the compiler deals with this by ignoring exception specifications and so no alternate source markup is needed.

By using this combination of language re-specification by the compiler, and the pre-processor tricks and the functional indirection layer for thrown exception objects by the library, `libstdc++` files can be compiled with `-fno-exceptions`.

User code that uses C++ keywords like `throw`, `try`, and `catch` will produce errors even if the user code has included `libstdc++` headers and is using constructs like `basic_istream`. Even though the standard library has been transformed, user code may need modification. User code that attempts or expects to do error checking on standard library components compiled with exception handling disabled should be evaluated and potentially made conditional.

Some issues remain with this approach (see [bugzilla entry 25191](#)). Code paths are not equivalent, in particular `catch` blocks are not evaluated. Also problematic are `throw` expressions expecting a user-defined throw handler. Known problem areas in the standard library include using an instance of `basic_istream` with `exceptions` set to specific `ios_base::iostate` conditions, or cascading `catch` blocks that dispatch error handling or recovery efforts based on the type of exception object thrown.

Oh, and by the way: none of this hackery is at all special. (Although perhaps well-deserving of a raised eyebrow.) Support continues to evolve and may change in the future. Similar and even additional techniques are used in other C++ libraries and compilers.

C++ hackers with a bent for language and control-flow purity have been successfully consoled by grizzled C veterans lamenting the substitution of the C language keyword `const` with the uglified doppelganger `__const`.

3.8.5 Compatibility

3.8.5.1 With C

C language code that is expecting to interoperate with C++ should be compiled with `-fexceptions`. This will make debugging a C language function called as part of C++-induced stack unwinding possible.

In particular, unwinding into a frame with no exception handling data will cause a runtime abort. If the unwinder runs out of unwind info before it finds a handler, `std::terminate()` is called.

Please note that most development environments should take care of getting these details right. For GNU systems, all appropriate parts of the GNU C library are already compiled with `-fexceptions`.

3.8.5.2 With POSIX thread cancellation

GNU systems re-use some of the exception handling mechanisms to track control flow for POSIX thread cancellation.

Cancellation points are functions defined by POSIX as worthy of special treatment. The standard library may use some of these functions to implement parts of the ISO C++ standard or depend on them for extensions.

Of note:

`nanosleep`, `read`, `write`, `open`, `close`, and `wait`.

The parts of `libstdc++` that use C library functions marked as cancellation points should take pains to be exception neutral. Failing this, `catch` blocks have been augmented to show that the POSIX cancellation object is in flight.

This augmentation adds a `catch` block for `__cxxabiv1::__forced_unwind`, which is the object representing the POSIX cancellation object. Like so:

```
catch(const __cxxabiv1::__forced_unwind&)
{
    this->_M_setstate(ios_base::badbit);
    throw;
}
catch(...)
{ this->_M_setstate(ios_base::badbit); }
```

3.8.6 Bibliography

- [1] *System Interface Definitions, Issue 7 (IEEE Std. 1003.1-2008)* , 2.9.5 Thread Cancellation , Copyright © 2008 The Open Group/The Institute of Electrical and Electronics Engineers, Inc. .
- [2] David Abrahams , *Error and Exception Handling* , Boost .
- [3] David Abrahams , *Exception-Safety in Generic Components* , Boost .
- [4] Matt Austern , *Standard Library Exception Policy* , WG21 N1077 .
- [5] Richard Henderson , *ia64 c++ abi exception handling* , GNU .
- [6] Bjarne Stroustrup , *Appendix E: Standard-Library Exception Safety*
- [7] Herb Sutter , Exception-Safety Issues and Techniques .
- [8] *GCC Bug 25191: exception_defines.h #defines try/catch*
- [9] *Tunables, The GNU C Library*

3.9 Debugging Support

There are numerous things that can be done to improve the ease with which C++ binaries are debugged when using the GNU tool chain. Here are some of them.

3.9.1 Using g++

Compiler flags determine how debug information is transmitted between compilation and debug or analysis tools.

The default optimizations and debug flags for a libstdc++ build are `-g -O2`. However, both debug and optimization flags can be varied to change debugging characteristics. For instance, turning off all optimization via the `-g -O0 -fno-inline` flags will disable inlining and optimizations, and include debugging information, so that stepping through all functions, (including inlined constructors and destructors) is possible. In addition, `-fno-eliminate-unused-debug-types` can be used when additional debug information, such as nested class info, is desired.

Or, the debug format that the compiler and debugger use to communicate information about source constructs can be changed via `-gdwarf-2` or `-gstabs` flags: some debugging formats permit more expressive type and scope information to be shown in GDB. Expressiveness can be enhanced by flags like `-g3`. The default debug information for a particular platform can be identified via the value set by the `PREFERRED_DEBUGGING_TYPE` macro in the GCC sources.

Many other options are available: please see ["Options for Debugging Your Program"](#) in Using the GNU Compiler Collection (GCC) for a complete list.

3.9.2 Debug Mode

The **Debug Mode** has compile and run-time checks for many containers.

There are also lightweight assertions for checking function preconditions, such as checking for out-of-bounds indices when accessing a `std::vector`. These can be enabled without using the full Debug Mode, by using `-D_GLIBCXX_ASSERTIONS` (see [Macros](#)).

3.9.3 Tracking uncaught exceptions

The **verbose termination handler** gives information about uncaught exceptions which kill the program.

3.9.4 Memory Leak Hunting

On many targets GCC supports AddressSanitizer, a fast memory error detector, which is enabled by the `-fsanitize=address` option.

The `std::vector` implementation has additional instrumentation to work with AddressSanitizer, but this has to be enabled explicitly by using `-D_GLIBCXX_SANITIZE_VECTOR` (see [Macros](#)).

There are also various third party memory tracing and debug utilities that can be used to provide detailed memory allocation information about C++ code. An exhaustive list of tools is not going to be attempted, but includes `mtrace`, `valgrind`, `mudflap` (no longer supported since GCC 4.9.0), `ElectricFence`, and the non-free commercial product `purify`. In addition, `libcwld`, `jemalloc` and `TCMalloc` have replacements for the global `new` and `delete` operators that can track memory allocation and deallocation and provide useful memory statistics.

For `valgrind`, there are some specific items to keep in mind. First of all, use a version of `valgrind` that will work with current GNU C++ tools: the first that can do this is `valgrind 1.0.4`, but later versions should work better. Second, using an unoptimized build might avoid confusing `valgrind`.

Third, it may be necessary to force deallocation in other libraries as well, namely the "C" library. On GNU/Linux, this can be accomplished with the appropriate use of the `__cxa_atexit` or `atexit` functions.

```
#include <cstdlib>

extern "C" void __libc_freeres(void);

void do_something() { }

int main()
{
    atexit(__libc_freeres);
    do_something();
    return 0;
}
```

or, using `__cxa_atexit`:

```
extern "C" void __libc_freeres(void);
extern "C" int __cxa_atexit(void (*func) (void *), void *arg, void *d);

void do_something() { }

int main()
{
    extern void* __dso_handle __attribute__((__weak__));
    __cxa_atexit((void (*) (void *)) __libc_freeres, NULL,
        &__dso_handle ? __dso_handle : NULL);
    do_test();
    return 0;
}
```

Suggested valgrind flags, given the suggestions above about setting up the runtime environment, library, and test file, might be:

```
valgrind -v --num-callers=20 --leak-check=yes --leak-resolution=high --show-reachable= ↵
yes a.out
```

3.9.4.1 Non-memory leaks in Pool and MT allocators

There are different kinds of allocation schemes that can be used by `std::allocator`. Prior to GCC 3.4.0 the default was to use a pooling allocator, `pool_allocator`, which is still available as the optional `__pool_alloc` extension. Another optional extension, `__mt_alloc`, is a high-performance pool allocator.

In a suspect executable these pooling allocators can give the mistaken impression that memory is being leaked, when in reality the memory "leak" is a pool being used by the library's allocator and is reclaimed after program termination.

If you're using memory debugging tools on a program that uses one of these pooling allocators, you can set the environment variable `GLIBCXX_FORCE_NEW` to keep extraneous pool allocation noise from cluttering debug information. For more details, see the [mt allocator](#) documentation and look specifically for `GLIBCXX_FORCE_NEW`.

3.9.5 Data Race Hunting

All synchronization primitives used in the library internals need to be understood by race detectors so that they do not produce false reports.

Two annotation macros are used to explain low-level synchronization to race detectors: `_GLIBCXX_SYNCHRONIZATION_HAPPENS` and `_GLIBCXX_SYNCHRONIZATION_HAPPENS_AFTER()`. By default, these macros are defined empty -- anyone who wants to use a race detector needs to redefine them to call an appropriate API. Since these macros are empty by default when the library is built, redefining them will only affect inline functions and template instantiations which are compiled in user code. This allows annotation of templates such as `shared_ptr`, but not code which is only instantiated in the library. Code which is only instantiated in the library needs to be recompiled with the annotation macros defined. That can be done by rebuilding the entire


```
--enable-libstdcxx-debug-flags='...'
```

Both the normal build and the debug build will persist, without having to specify `CXXFLAGS`, and the debug library will be installed in a separate directory tree, in `(prefix)/lib/debug`. For more information, look at the [configuration](#) section.

A second approach is to use the configuration flags

```
make CXXFLAGS='-g3 -fno-inline -O0' all
```

3.9.8 Compile Time Checking

The [Compile-Time Checks](#) extension has compile-time checks for many algorithms. These checks were designed for C++98 and have not been updated to work with C++11 and later standards. They might be removed at a future date.

Part II

Standard Contents

Chapter 4

Support

This part deals with the functions called and objects created automatically during the course of a program's existence.

While we can't reproduce the contents of the Standard here (you need to get your own copy from your nation's member body; see our homepage for help), we can mention a couple of changes in what kind of support a C++ program gets from the Standard Library.

4.1 Types

4.1.1 Fundamental Types

C++ has the following builtin types:

- `char`
- `signed char`
- `unsigned char`
- `signed short`
- `signed int`
- `signed long`
- `unsigned short`
- `unsigned int`
- `unsigned long`
- `bool`
- `wchar_t`
- `float`
- `double`
- `long double`

These fundamental types are always available, without having to include a header file. These types are exactly the same in either C++ or in C.

Specializing parts of the library on these types is prohibited: instead, use a POD.

4.1.2 Numeric Properties

The header `<limits>` defines traits classes to give access to various implementation defined-aspects of the fundamental types. The traits classes -- fourteen in total -- are all specializations of the class template `numeric_limits` and defined as follows:

```
template<typename T>
struct class
{
    static const bool is_specialized;
    static T max() throw();
    static T min() throw();

    static const int digits;
    static const int digits10;
    static const bool is_signed;
    static const bool is_integer;
    static const bool is_exact;
    static const int radix;
    static T epsilon() throw();
    static T round_error() throw();

    static const int min_exponent;
    static const int min_exponent10;
    static const int max_exponent;
    static const int max_exponent10;

    static const bool has_infinity;
    static const bool has_quiet_NaN;
    static const bool has_signaling_NaN;
    static const float_denorm_style has_denorm;
    static const bool has_denorm_loss;
    static T infinity() throw();
    static T quiet_NaN() throw();
    static T denorm_min() throw();

    static const bool is_iec559;
    static const bool is_bounded;
    static const bool is_modulo;

    static const bool traps;
    static const bool tinyness_before;
    static const float_round_style round_style;
};
```

4.1.3 NULL

The only change that might affect people is the type of `NULL`: while it is required to be a macro, the definition of that macro is *not* allowed to be an expression with pointer type such as `(void*)0`, which is often used in C.

For **g++**, `NULL` is `#define'd` to be `__null`, a magic keyword extension of **g++** that is slightly safer than a plain integer.

The biggest problem of `#defining` `NULL` to be something like “0L” is that the compiler will view that as a long integer before it views it as a pointer, so overloading won’t do what you expect. It might not even have the same size as a pointer, so passing `NULL` to a varargs function where a pointer is expected might not even work correctly if `sizeof(NULL) < sizeof(void*)`. The **G++** `__null` extension is defined so that `sizeof(__null) == sizeof(void*)` to avoid this problem.

Scott Meyers explains this in more detail in his book *Effective Modern C++* and as a guideline to solve this problem recommends to not overload on pointer-vs-integer types to begin with.

The C++ 2011 standard added the `nullptr` keyword, which is a null pointer constant of a special type, `std::nullptr_t`. Values of this type can be implicitly converted to *any* pointer type, and cannot convert to integer types or be deduced as an integer type. Unless you need to be compatible with C++98/C++03 or C you should prefer to use `nullptr` instead of `NULL`.


```

while (true)
{
    if (void* p = /* try to allocate memory */)
        return p;
    else if (std::new_handler h = std::get_new_handler ())
        h ();
    else
        throw bad_alloc{};
}

```

This means you can influence what happens on allocation failure by writing your own new-handler and then registering it with `std::set_new_handler`:

```

typedef void (*PFV)();

static char*  safety;
static PFV    old_handler;

void my_new_handler ()
{
    delete[] safety;
    safety = nullptr;
    popup_window ("Dude, you are running low on heap memory.  You"
        " should, like, close some windows, or something."
        " The next time you run out, we're gonna burn!");
    set_new_handler (old_handler);
    return;
}

int main ()
{
    safety = new char[500000];
    old_handler = set_new_handler (&my_new_handler);
    ...
}

```

4.2.1 Additional Notes

Remember that it is perfectly okay to delete a null pointer! Nothing happens, by definition. That is not the same thing as deleting a pointer twice.

`std::bad_alloc` is derived from the base `std::exception` class, see [Exceptions](#).

4.3 Termination

4.3.1 Termination Handlers

Not many changes here to `<cstdlib>`. You should note that the `abort()` function does not call the destructors of automatic nor static objects, so if you're depending on those to do cleanup, it isn't going to happen. (The functions registered with `atexit()` don't get called either, so you can forget about that possibility, too.)

The good old `exit()` function can be a bit funky, too, until you look closer. Basically, three points to remember are:

1. Static objects are destroyed in reverse order of their creation.
2. Functions registered with `atexit()` are called in reverse order of registration, once per registration call. (This isn't actually new.)

3. The previous two actions are “interleaved,” that is, given this pseudocode:

```
extern "C or C++" void f1 ();
extern "C or C++" void f2 ();

static Thing obj1;
atexit(f1);
static Thing obj2;
atexit(f2);
```

then at a call of `exit()`, `f2` will be called, then `obj2` will be destroyed, then `f1` will be called, and finally `obj1` will be destroyed. If `f1` or `f2` allow an exception to propagate out of them, Bad Things happen.

Note also that `atexit()` is only required to store 32 functions, and the compiler/library might already be using some of those slots. If you think you may run out, we recommend using the `xatexit/xexit` combination from `libiberty`, which has no such limit.

4.3.2 Verbose Terminate Handler

If you are having difficulty with uncaught exceptions and want a little bit of help debugging the causes of the core dumps, you can make use of a GNU extension, the verbose terminate handler.

The verbose terminate handler is only available for hosted environments (see [Configuring](#)) and will be used by default unless the library is built with `--disable-libstdcxx-verbose` or with exceptions disabled. If you need to enable it explicitly you can do so by calling the `std::set_terminate` function.

```
#include <exception>

int main()
{
    std::set_terminate(__gnu_cxx::__verbose_terminate_handler);
    ...

    throw anything;
}
```

The `__verbose_terminate_handler` function obtains the name of the current exception, attempts to demangle it, and prints it to `stderr`. If the exception is derived from `std::exception` then the output from `what()` will be included.

Any replacement termination function is required to kill the program without returning; this one calls `std::abort`.

For example:

```
#include <exception>
#include <stdexcept>

struct argument_error : public std::runtime_error
{
    argument_error(const std::string& s): std::runtime_error(s) { }
};

int main(int argc)
{
    std::set_terminate(__gnu_cxx::__verbose_terminate_handler);
    if (argc > 5)
        throw argument_error("argc is greater than 5!");
    else
        throw argc;
}
```

With the verbose terminate handler active, this gives:

```
% ./a.out
terminate called after throwing a `int'
Aborted
% ./a.out f f f f f f f f f f
terminate called after throwing an instance of `argument_error'
what(): argc is greater than 5!
Aborted
```

The 'Aborted' line is printed by the shell after the process exits by calling `abort()`.

As this is the default termination handler, nothing need be done to use it. To go back to the previous “silent death” method, simply include `<exception>` and `<cstdlib>`, and call

```
std::set_terminate(std::abort);
```

After this, all calls to `terminate` will use `abort` as the terminate handler.

Note: the verbose terminate handler will attempt to write to `stderr`. If your application closes `stderr` or redirects it to an inappropriate location, `__verbose_terminate_handler` will behave in an unspecified manner.

Chapter 5

Diagnostics

5.1 Exceptions

5.1.1 API Reference

Most exception classes are defined in one of the standard headers `<exception>`, `<stdexcept>`, `<new>`, and `<typeinfo>`. The C++ 2011 revision of the standard added more exception types in the headers `<functional>`, `<future>`, `<regex>`, and `<system_error>`. The C++ 2017 revision of the standard added more exception types in the headers `<any>`, `<filesystem>`, `<optional>`, and `<variant>`.

All exceptions thrown by the library have a base class of type `std::exception`, defined in `<exception>`. This type has no `std::string` member.

Derived from this are several classes that may have a `std::string` member. A full hierarchy can be found in the source documentation.

5.1.2 Adding Data to `exception`

The standard exception classes carry with them a single string as data (usually describing what went wrong or where the 'throw' took place). It's good to remember that you can add your own data to these exceptions when extending the hierarchy:

```
struct My_Exception : public std::runtime_error
{
    public:
        My_Exception (const string& whatarg)
        : std::runtime_error(whatarg), e(errno), id(GetDataBaseID()) { }
        int  errno_at_time_of_throw() const { return e; }
        DBID id_of_thing_that_threw() const { return id; }
    protected:
        int      e;
        DBID     id;      // some user-defined type
};
```

5.2 Use of `errno` by the library

The C and POSIX standards guarantee that `errno` is never set to zero by any library function. The C++ standard has less to say about when `errno` is or isn't set, but `libstdc++` follows the same rule and never sets it to zero.

On the other hand, there are few guarantees about when the C++ library sets `errno` on error, beyond what is specified for functions that come from the C library. For example, when `std::stoi` throws an exception of type `std::out_of_range`, `errno` may or may not have been set to `ERANGE`.

Parts of the C++ library may be implemented in terms of C library functions, which may result in `errno` being set with no explicit call to a C function. For example, on a target where `operator new` uses `malloc` a failed memory allocation with `operator new` might set `errno` to `ENOMEM`. Which C++ library functions can set `errno` in this way is unspecified because it may vary between platforms and between releases.

5.3 Concept Checking

In 1999, SGI added “concept checkers” to their implementation of the STL: code which checked the template parameters of instantiated pieces of the STL, in order to insure that the parameters being used met the requirements of the standard. For example, the Standard requires that types passed as template parameters to `vector` be “Assignable” (which means what you think it means). The checking was done during compilation, and none of the code was executed at runtime.

Unfortunately, the size of the compiler files grew significantly as a result. The checking code itself was cumbersome. And bugs were found in it on more than one occasion.

The primary author of the checking code, Jeremy Siek, had already started work on a replacement implementation. The new code was formally reviewed and accepted into [the Boost libraries](#), and we are pleased to incorporate it into the GNU C++ library.

The new version imposes a much smaller space overhead on the generated object file. The checks are also cleaner and easier to read and understand.

They are off by default for all versions of GCC. They can be enabled at configure time with `--enable-concept-checks`. You can enable them on a per-translation-unit basis with `-D_GLIBCXX_CONCEPT_CHECKS`.

Please note that the checks are based on the requirements in the original C++ standard, many of which were relaxed in the C++11 standard and so valid C++11 code may be incorrectly rejected by the concept checks. Additionally, some correct C++03 code might be rejected by the concept checks, for example template argument types may need to be complete when used in a template definition, rather than at the point of instantiation. There are no plans to address these shortcomings.

Chapter 6

Utilities

6.1 Functors

If you don't know what functors are, you're not alone. Many people get slightly the wrong idea. In the interest of not reinventing the wheel, we will refer you to the introduction to the functor concept written by SGI as part of their STL, in [their <https://web.archive.org/web/20171209002754/http://www.sgi.com/tech/stl/functors.html>](https://web.archive.org/web/20171209002754/http://www.sgi.com/tech/stl/functors.html).

6.2 Pairs

The `pair<T1, T2>` is a simple and handy way to carry around a pair of objects. One is of type `T1`, and another of type `T2`; they may be the same type, but you don't get anything extra if they are. The two members can be accessed directly, as `.first` and `.second`.

Construction is simple. The default ctor initializes each member with its respective default ctor. The other simple ctor,

```
pair (const T1& x, const T2& y);
```

does what you think it does, `first` getting `x` and `second` getting `y`.

There is a constructor template for copying pairs of other types:

```
template <class U, class V> pair (const pair<U,V>& p);
```

The compiler will convert as necessary from `U` to `T1` and from `V` to `T2` in order to perform the respective initializations.

The comparison operators are done for you. Equality of two `pair<T1, T2>`s is defined as both `first` members comparing equal and both `second` members comparing equal; this simply delegates responsibility to the respective `operator==` functions (for types like `MyClass`) or builtin comparisons (for types like `int`, `char`, etc).

The less-than operator is a bit odd the first time you see it. It is defined as evaluating to:

```
x.first < y.first ||  
( !(y.first < x.first) && x.second < y.second )
```

The other operators are not defined using the `rel_ops` functions above, but their semantics are the same.

Finally, there is a template function called `make_pair` that takes two references-to-const objects and returns an instance of a pair instantiated on their respective types:

```
pair<int, MyClass> p = make_pair(4, myobject);
```

6.3 Memory

Memory contains three general areas. First, function and operator calls via `new` and `delete` operator or member function calls. Second, allocation via `allocator`. And finally, smart pointer and intelligent pointer abstractions.

6.3.1 Allocators

Memory management for Standard Library entities is encapsulated in a class template called `allocator`. The `allocator` abstraction is used throughout the library in `string`, container classes, algorithms, and parts of `iostreams`. This class, and base classes of it, are the superset of available free store (“heap”) management classes.

6.3.1.1 Requirements

The C++ standard only gives a few directives in this area:

- When you add elements to a container, and the container must allocate more memory to hold them, the container makes the request via its `Allocator` template parameter, which is usually aliased to `allocator_type`. This includes adding chars to the `string` class, which acts as a regular STL container in this respect.
- The default `Allocator` argument of every container-of-`T` is `allocator<T>`.
- The interface of the `allocator<T>` class is extremely simple. It has about 20 public declarations (nested typedefs, member functions, etc), but the two which concern us most are:

```
T*    allocate    (size_type n, const void* hint = 0);
void deallocate  (T* p, size_type n);
```

The `n` arguments in both those functions is a *count* of the number of `T`'s to allocate space for, *not their total size*. (This is a simplification; the real signatures use nested typedefs.)

- The storage is obtained by calling `::operator new`, but it is unspecified when or how often this function is called. The use of the `hint` is unspecified, but intended as an aid to locality if an implementation so desires. [20.4.1.1]/6

Complete details can be found in the C++ standard, look in [20.4 Memory].

6.3.1.2 Design Issues

The easiest way of fulfilling the requirements is to call `operator new` each time a container needs memory, and to call `operator delete` each time the container releases memory. This method may be **slower** than caching the allocations and re-using previously-allocated memory, but has the advantage of working correctly across a wide variety of hardware and operating systems, including large clusters. The `__gnu_cxx::new_allocator` implements the simple `operator new` and `operator delete` semantics, while `__gnu_cxx::malloc_allocator` implements much the same thing, only with the C language functions `std::malloc` and `std::free`.

Another approach is to use intelligence within the `allocator` class to cache allocations. This extra machinery can take a variety of forms: a bitmap index, an index into an exponentially increasing power-of-two-sized buckets, or simpler fixed-size pooling cache. The cache is shared among all the containers in the program: when your program's `std::vector<int>` gets cut in half and frees a bunch of its storage, that memory can be reused by the private `std::list<WonkyWidget>` brought in from a KDE library that you linked against. And operators `new` and `delete` are not always called to pass the memory on, either, which is a speed bonus. Examples of allocators that use these techniques are `__gnu_cxx::bitmap_allocator`, `__gnu_cxx::pool_allocator`, and `__gnu_cxx::__mt_alloc`.

Depending on the implementation techniques used, the underlying operating system, and compilation environment, scaling caching allocators can be tricky. In particular, order-of-destruction and order-of-creation for memory pools may be difficult to pin down with certainty, which may create problems when used with plugins or loading and unloading shared objects in memory. As such, using caching allocators on systems that do not support `abi::__cxa_atexit` is not recommended.

6.3.1.3 Implementation

6.3.1.3.1 Interface Design

The only allocator interface that is supported is the standard C++ interface. As such, all STL containers have been adjusted, and all external allocators have been modified to support this change.

The class `allocator` just has typedef, constructor, and rebind members. It inherits from one of the high-speed extension allocators, covered below. Thus, all allocation and deallocation depends on the base class.

The choice of base class that `allocator` is derived from is fixed at the time when GCC is built, and the different choices are not ABI compatible.

6.3.1.3.2 Selecting Default Allocation Policy

It's difficult to pick an allocation strategy that will provide maximum utility, without excessively penalizing some behavior. In fact, it's difficult just deciding which typical actions to measure for speed.

Three synthetic benchmarks have been created that provide data that is used to compare different C++ allocators. These tests are:

1. Insertion.

Over multiple iterations, various STL container objects have elements inserted to some maximum amount. A variety of allocators are tested. Test source for [sequence](#) and [associative](#) containers.

2. Insertion and erasure in a multi-threaded environment.

This test shows the ability of the allocator to reclaim memory on a per-thread basis, as well as measuring thread contention for memory resources. Test source [here](#).

3. A threaded producer/consumer model.

Test source for [sequence](#) and [associative](#) containers.

Since GCC 12 the default choice for `allocator` is `std::__new_allocator`. Before GCC 12 it was the `__gnu_cxx::new_allocator` extension (which has identical behaviour).

6.3.1.3.3 Disabling Memory Caching

In use, `allocator` may allocate and deallocate using implementation-specific strategies and heuristics. Because of this, a given call to an allocator object's `allocate` member function may not actually call the global operator `new` and a given call to the `deallocate` member function may not call operator `delete`.

This can be confusing.

In particular, this can make debugging memory errors more difficult, especially when using third-party tools like valgrind or debug versions of `new`.

There are various ways to solve this problem. One would be to use a custom allocator that just called operators `new` and `delete` directly, for every allocation. (See the default allocator, `include/ext/new_allocator.h`, for instance.) However, that option may involve changing source code to use a non-default allocator. Another option is to force the default allocator to remove caching and pools, and to directly allocate with every call of `allocate` and directly deallocate with every call of `deallocate`, regardless of efficiency. As it turns out, this last option is also available.

To globally disable memory caching within the library for some of the optional non-default allocators, merely set `GLIBCXX_FORCE_NEW` (with any value) in the system's environment before running the program. If your program crashes with `GLIBCXX_FORCE_NEW` in the environment, it likely means that you linked against objects built against the older library (objects which might still using the cached allocations...).

6.3.1.4 Using a Specific Allocator

You can specify different memory management schemes on a per-container basis, by overriding the default Allocator template parameter. For example, an easy (but non-portable) method of specifying that only `malloc` or `free` should be used instead of the default node allocator is:

```
std::list<int, __gnu_cxx::malloc_allocator<int>> malloc_list;
```

Likewise, a debugging form of whichever allocator is currently in use:

```
std::deque<int, __gnu_cxx::debug_allocator<std::allocator<int>>> debug_deque;
```

6.3.1.5 Custom Allocators

Writing a portable C++ allocator would dictate that the interface would look much like the one specified for `allocator`. Additional member functions, but not subtractions, would be permissible.

Probably the best place to start would be to copy one of the extension allocators: say a simple one like `new_allocator`.

Since C++11 the minimal interface required for an allocator is much smaller, as `std::allocator_traits` can provide default for much of the interface.

6.3.1.6 Extension Allocators

Several other allocators are provided as part of this implementation. The location of the extension allocators and their names have changed, but in all cases, functionality is equivalent. Starting with gcc-3.4, all extension allocators are standard style. Before this point, SGI style was the norm. Because of this, the number of template arguments also changed. Table B.6 tracks the changes.

More details on each of these extension allocators follows.

1. `new_allocator`

Simply wraps `::operator new` and `::operator delete`.

2. `malloc_allocator`

Simply wraps `malloc` and `free`. There is also a hook for an out-of-memory handler (for `new/delete` this is taken care of elsewhere).

3. `debug_allocator`

A wrapper around an arbitrary allocator `A`. It passes on slightly increased size requests to `A`, and uses the extra memory to store size information. When a pointer is passed to `deallocate()`, the stored size is checked, and `assert()` is used to guarantee they match.

4. `throw_allocator`

Includes memory tracking and marking abilities as well as hooks for throwing exceptions at configurable intervals (including random, all, none).

5. `__pool_alloc`

A high-performance, single pool allocator. The reusable memory is shared among identical instantiations of this type. It calls through `::operator new` to obtain new memory when its lists run out. If a client container requests a block larger than a certain threshold size, then the pool is bypassed, and the `allocate/deallocate` request is passed to `::operator new` directly.

For thread-enabled configurations, the pool is locked with a single big lock. In some situations, this implementation detail may result in severe performance degradation.

(Note that the GCC thread abstraction layer allows us to provide safe zero-overhead stubs for the threading routines, if threads were disabled at configuration time.)

6. `__mt_alloc`

A high-performance fixed-size allocator with exponentially-increasing allocations. It has its own [chapter](#) in the documentation.

7. `bitmap_allocator`

A high-performance allocator that uses a bit-map to keep track of the used and unused memory locations. It has its own [chapter](#) in the documentation.

6.3.1.7 Bibliography

- [10] Matt Austern, *The Standard Librarian: What Are Allocators Good For?* , C/C++ Users Journal , 2000-12.
- [11] Emery Berger, *The Hoard Memory Allocator*
- [12] Emery BergerBen ZornKathryn McKinley, *Reconsidering Custom Memory Allocation* , Copyright © 2002 OOPSLA.
- [13] Klaus KreftAngelika Langer, *Allocator Types* , C/C++ Users Journal .
- [14] Bjarne Stroustrup, Copyright © 2000 , 19.4 Allocators, Addison Wesley .
- [15] Felix Yen

[isoc++_1998] , 20.4 Memory.

6.3.2 `auto_ptr`

6.3.2.1 Limitations

Explaining all of the fun and delicious things that can happen with misuse of the `auto_ptr` class template (called AP here) would take some time. Suffice it to say that the use of AP safely in the presence of copying has some subtleties.

The AP class is a really nifty idea for a smart pointer, but it is one of the dumbest of all the smart pointers -- and that's fine.

AP is not meant to be a supersmart solution to all resource leaks everywhere. Neither is it meant to be an effective form of garbage collection (although it can help, a little bit). And it can *not* be used for arrays!

AP is meant to prevent nasty leaks in the presence of exceptions. That's *all*. This code is AP-friendly:

```
// Not a recommend naming scheme, but good for web-based FAQs.
typedef std::auto_ptr<MyClass>  APMC;

extern function_taking_MyClass_pointer (MyClass*);
extern some_throwable_function ();

void func (int data)
{
    APMC  ap (new MyClass(data));

    some_throwable_function();    // this will throw an exception

    function_taking_MyClass_pointer (ap.get());
}
```

When an exception gets thrown, the instance of `MyClass` that's been created on the heap will be `delete`'d as the stack is unwound past `func()`.

Changing that code as follows is not AP-friendly:

```
APMC  ap (new MyClass[22]);
```

You will get the same problems as you would without the use of AP:

```
char* array = new char[10];           // array new...
...
delete array;                         // ...but single-object delete
```

AP cannot tell whether the pointer you’ve passed at creation points to one or many things. If it points to many things, you are about to die. AP is trivial to write, however, so you could write your own `auto_array_ptr` for that situation (in fact, this has been done many times; check the mailing lists, Usenet, Boost, etc).

6.3.2.2 Use in Containers

All of the **containers** described in the standard library require their contained types to have, among other things, a copy constructor like this:

```
struct My_Type
{
    My_Type (My_Type const&);
};
```

Note the `const` keyword; the object being copied shouldn’t change. The template class `auto_ptr` (called AP here) does not meet this requirement. Creating a new AP by copying an existing one transfers ownership of the pointed-to object, which means that the AP being copied must change, which in turn means that the copy ctors of AP do not take `const` objects.

The resulting rule is simple: *Never ever use a container of `auto_ptr` objects.* The standard says that “undefined” behavior is the result, but it is guaranteed to be messy.

To prevent you from doing this to yourself, the **concept checks** built in to this implementation will issue an error if you try to compile code like this:

```
#include <vector>
#include <memory>

void f()
{
    std::vector< std::auto_ptr<int> >   vec_ap_int;
}
```

Should you try this with the checks enabled, you will see an error.

6.3.3 `shared_ptr`

The `shared_ptr` class template stores a pointer, usually obtained via `new`, and implements shared ownership semantics.

6.3.3.1 Requirements

The standard deliberately doesn’t require a reference-counted implementation, allowing other techniques such as a circular-linked-list.

6.3.3.2 Design Issues

The `shared_ptr` code is kindly donated to GCC by the Boost project and the original authors of the code. The basic design and algorithms are from Boost, the notes below describe details specific to the GCC implementation. Names have been uglified in this implementation, but the design should be recognisable to anyone familiar with the Boost 1.32 `shared_ptr`.

The basic design is an abstract base class, `_Sp_counted_base` that does the reference-counting and calls virtual functions when the count drops to zero. Derived classes override those functions to destroy resources in a context where the correct dynamic type is known. This is an application of the technique known as type erasure.

6.3.3.3 Implementation

6.3.3.3.1 Class Hierarchy

A `shared_ptr<T>` contains a pointer of type `T*` and an object of type `__shared_count`. The `shared_count` contains a pointer of type `_Sp_counted_base*` which points to the object that maintains the reference-counts and destroys the managed resource.

`_Sp_counted_base<Lp>` The base of the hierarchy is parameterized on the lock policy (see below.) `_Sp_counted_base` doesn't depend on the type of pointer being managed, it only maintains the reference counts and calls virtual functions when the counts drop to zero. The managed object is destroyed when the last strong reference is dropped, but the `_Sp_counted_base` itself must exist until the last weak reference is dropped.

`_Sp_counted_base_impl<Ptr, Deleter, Lp>` Inherits from `_Sp_counted_base` and stores a pointer of type `Ptr` and a deleter of type `Deleter`. `_Sp_deleter` is used when the user doesn't supply a custom deleter. Unlike Boost's, this default deleter is not "checked" because GCC already issues a warning if `delete` is used with an incomplete type. This is the only derived type used by `tr1::shared_ptr<Ptr>` and it is never used by `std::shared_ptr`, which uses one of the following types, depending on how the `shared_ptr` is constructed.

`_Sp_counted_ptr<Ptr, Lp>` Inherits from `_Sp_counted_base` and stores a pointer of type `Ptr`, which is passed to `delete` when the last reference is dropped. This is the simplest form and is used when there is no custom deleter or allocator.

`_Sp_counted_deleter<Ptr, Deleter, Alloc>` Inherits from `_Sp_counted_ptr` and adds support for custom deleter and allocator. Empty Base Optimization is used for the allocator. This class is used even when the user only provides a custom deleter, in which case `allocator` is used as the allocator.

`_Sp_counted_ptr_inplace<Tp, Alloc, Lp>` Used by `allocate_shared` and `make_shared`. Contains aligned storage to hold an object of type `Tp`, which is constructed in-place with placement `new`. Has a variadic template constructor allowing any number of arguments to be forwarded to `Tp`'s constructor. Unlike the other `_Sp_counted_*` classes, this one is parameterized on the type of object, not the type of pointer; this is purely a convenience that simplifies the implementation slightly.

C++11-only features are: rvalue-ref/move support, allocator support, aliasing constructor, `make_shared` & `allocate_shared`. Additionally, the constructors taking `auto_ptr` parameters are deprecated in C++11 mode.

6.3.3.3.2 Thread Safety

The [Thread Safety](#) section of the Boost `shared_ptr` documentation says "shared_ptr objects offer the same level of thread safety as built-in types." The implementation must ensure that concurrent updates to separate `shared_ptr` instances are correct even when those instances share a reference count e.g.

```
shared_ptr<A> a(new A);
shared_ptr<A> b(a);

// Thread 1      // Thread 2
a.reset();      b.reset();
```

The dynamically-allocated object must be destroyed by exactly one of the threads. Weak references make things even more interesting. The shared state used to implement `shared_ptr` must be transparent to the user and invariants must be preserved at all times. The key pieces of shared state are the strong and weak reference counts. Updates to these need to be atomic and visible to all threads to ensure correct cleanup of the managed resource (which is, after all, `shared_ptr`'s job!) On multi-processor systems memory synchronisation may be needed so that reference-count updates and the destruction of the managed resource are race-free.

The function `_Sp_counted_base::_M_add_ref_lock()`, called when obtaining a `shared_ptr` from a `weak_ptr`, has to test if the managed resource still exists and either increment the reference count or throw `bad_weak_ptr`. In a multi-threaded program there is a potential race condition if the last reference is dropped (and the managed resource destroyed) between testing the reference count and incrementing it, which could result in a `shared_ptr` pointing to invalid memory.

The Boost `shared_ptr` (as used in GCC) features a clever lock-free algorithm to avoid the race condition, but this relies on the processor supporting an atomic *Compare-And-Swap* instruction. For other platforms there are fall-backs using mutex locks. Boost (as of version 1.35) includes several different implementations and the preprocessor selects one based on the compiler, standard library, platform etc. For the version of `shared_ptr` in libstdc++ the compiler and library are fixed, which makes things much simpler: we have an atomic CAS or we don't, see Lock Policy below for details.

6.3.3.3 Selecting Lock Policy

There is a single `_Sp_counted_base` class, which is a template parameterized on the enum `__gnu_cxx::_Lock_policy`. The entire family of classes is parameterized on the lock policy, right up to `__shared_ptr`, `__weak_ptr` and `__enable_shared_from_this`. The actual `std::shared_ptr` class inherits from `__shared_ptr` with the lock policy parameter selected automatically based on the thread model and platform that libstdc++ is configured for, so that the best available template specialization will be used. This design is necessary because it would not be conforming for `shared_ptr` to have an extra template parameter, even if it had a default value. The available policies are:

1. `_S_atomic`

Selected when GCC supports a builtin atomic compare-and-swap operation on the target processor (see [Atomic Builtins](#).) The reference counts are maintained using a lock-free algorithm and GCC's atomic builtins, which provide the required memory synchronisation.

2. `_S_mutex`

The `_Sp_counted_base` specialization for this policy contains a mutex, which is locked in `add_ref_lock()`. This policy is used when GCC's atomic builtins aren't available so explicit memory barriers are needed in places.

3. `_S_single`

This policy uses a non-reentrant `add_ref_lock()` with no locking. It is used when libstdc++ is built without `--enable-threads`.

For all three policies, reference count increments and decrements are done via the functions in `ext/atomicity.h`, which detect if the program is multi-threaded. If only one thread of execution exists in the program then less expensive non-atomic operations are used.

6.3.3.4 Related functions and classes

dynamic_pointer_cast, static_pointer_cast, const_pointer_cast As noted in N2351, these functions can be implemented non-intrusively using the alias constructor. However the aliasing constructor is only available in C++11 mode, so in TR1 mode these casts rely on three non-standard constructors in `shared_ptr` and `__shared_ptr`. In C++11 mode these constructors and the related tag types are not needed.

enable_shared_from_this The clever overload to detect a base class of type `enable_shared_from_this` comes straight from Boost. There is an extra overload for `__enable_shared_from_this` to work smoothly with `__shared_ptr<Tp>` using any lock policy.

make_shared, allocate_shared `make_shared` simply forwards to `allocate_shared` with `std::allocator` as the allocator. Although these functions can be implemented non-intrusively using the alias constructor, if they have access to the implementation then it is possible to save storage and reduce the number of heap allocations. The newly constructed object and the `_Sp_counted_*` can be allocated in a single block and the standard says implementations are "encouraged, but not required," to do so. This implementation provides additional non-standard constructors (selected with the type `_Sp_make_shared_tag`) which create an object of type `_Sp_counted_ptr_inplace` to hold the new object. The returned `shared_ptr<A>` needs to know the address of the new A object embedded in the `_Sp_counted_ptr_inplace`, but it has no way to access it. This implementation uses a "covert channel" to return the address of the embedded object when `get_deleter<_Sp_make_shared_tag>()` is called. Users should not try to use this. As well as the extra constructors, this implementation also needs some members of `_Sp_counted_deleter` to be protected where they could otherwise be private.

6.3.3.4 Use

6.3.3.4.1 Examples

Examples of use can be found in the testsuite, under `testsuite/tr1/2_general_utilities/shared_ptr`, `testsuite/20_util/shared_ptr` and `testsuite/20_util/weak_ptr`.

6.3.3.4.2 Unresolved Issues

The *shared_ptr atomic access* clause in the C++11 standard is not implemented in GCC.

Unlike Boost, this implementation does not use separate classes for the pointer+deleter and pointer+deleter+allocator cases in C++11 mode, combining both into `_Sp_counted_deleter` and using `allocator` when the user doesn't specify an allocator. If it was found to be beneficial an additional class could easily be added. With the current implementation, the `_Sp_counted_deleter` and `__shared_count` constructors taking a custom deleter but no allocator are technically redundant and could be removed, changing callers to always specify an allocator. If a separate pointer+deleter class was added the `__shared_count` constructor would be needed, so it has been kept for now.

The hack used to get the address of the managed object from `_Sp_counted_ptr_inplace::_M_get_deleter()` is accessible to users. This could be prevented if `get_deleter<_Sp_make_shared_tag>()` always returned `NULL`, since the hack only needs to work at a lower level, not in the public API. This wouldn't be difficult, but hasn't been done since there is no danger of accidental misuse: users already know they are relying on unsupported features if they refer to implementation details such as `_Sp_make_shared_tag`.

`tr1::_Sp_deleter` could be a private member of `tr1::__shared_count` but it would alter the ABI.

6.3.3.5 Acknowledgments

The original authors of the Boost `shared_ptr`, which is really nice code to work with, Peter Dimov in particular for his help and invaluable advice on thread safety. Phillip Jordan and Paolo Carlini for the lock policy implementation.

6.3.3.6 Bibliography

- [16] *Improving shared_ptr for C++0x, Revision 2* , N2351 .
- [17] *C++ Standard Library Active Issues List* , N2456 .
- [18] *Working Draft, Standard for Programming Language C++* , N2461 .
- [19] *Boost C++ Libraries documentation, shared_ptr* , N2461 .

6.4 Traits

Chapter 7

Strings

7.1 String Classes

7.1.1 Simple Transformations

Here are Standard, simple, and portable ways to perform common transformations on a `string` instance, such as "convert to all upper case." The word transformations is especially apt, because the standard template function `transform<>` is used.

This code will go through some iterations. Here's a simple version:

```
#include <string>
#include <algorithm>
#include <cctype>      // old <ctype.h>

struct ToLower
{
    char operator() (char c) const { return std::tolower(c); }
};

struct ToUpper
{
    char operator() (char c) const { return std::toupper(c); }
};

int main()
{
    std::string s ("Some Kind Of Initial Input Goes Here");

    // Change everything into upper case
    std::transform (s.begin(), s.end(), s.begin(), ToUpper());

    // Change everything into lower case
    std::transform (s.begin(), s.end(), s.begin(), ToLower());

    // Change everything back into upper case, but store the
    // result in a different string
    std::string capital_s;
    capital_s.resize(s.size());
    std::transform (s.begin(), s.end(), capital_s.begin(), ToUpper());
}
```

Note that these calls all involve the global C locale through the use of the C functions `toupper/tolower`. This is absolutely guaranteed to work -- but *only* if the string contains *only* characters from the basic source character set, and there are *only* 96 of

those. Which means that not even all English text can be represented (certain British spellings, proper names, and so forth). So, if all your input forevermore consists of only those 96 characters (hahahahahaha), then you're done.

Note that the `ToUpper` and `ToLower` function objects are needed because `toupper` and `tolower` are overloaded names (declared in `<cctype>` and `<locale>`) so the template-arguments for `transform<>` cannot be deduced, as explained in [this message](#). At minimum, you can write short wrappers like

```
char toLower (char c)
{
    // std::tolower(c) is undefined if c < 0 so cast to unsigned char.
    return std::tolower((unsigned char)c);
}
```

(Thanks to James Kanze for assistance and suggestions on all of this.)

Since C++11 the wrapper can be replaced with a lambda expression, which can perform the conversion to `unsigned char` and also ensure the single-argument form of `std::lower` is used:

```
std::transform (s.begin(), s.end(), capital_s.begin(),
               [](unsigned char c) { return std::tolower(c); });
```

Another common operation is trimming off excess whitespace. Much like transformations, this task is trivial with the use of `string`'s `find` family. These examples are broken into multiple statements for readability:

```
std::string str (" \t blah blah blah \n ");

// trim leading whitespace
string::size_type notwhite = str.find_first_not_of(" \t\n");
str.erase(0,notwhite);

// trim trailing whitespace
notwhite = str.find_last_not_of(" \t\n");
str.erase(notwhite+1);
```

Obviously, the calls to `find` could be inserted directly into the calls to `erase`, in case your compiler does not optimize named temporaries out of existence.

7.1.2 Case Sensitivity

The well-known-and-if-it-isn't-well-known-it-ought-to-be [Guru of the Week](#) discussions held on Usenet covered this topic in January of 1998. Briefly, the challenge was, "write a `'ci_string'` class which is identical to the standard `'string'` class, but is case-insensitive in the same way as the (common but nonstandard) C function `stricmp()`".

```
ci_string s( "AbCdE" );

// case insensitive
assert( s == "abcde" );
assert( s == "ABCDE" );

// still case-preserving, of course
assert( strcmp( s.c_str(), "AbCdE" ) == 0 );
assert( strcmp( s.c_str(), "abcde" ) != 0 );
```

The solution is surprisingly easy. The original answer was posted on Usenet, and a revised version appears in Herb Sutter's book *Exceptional C++* and on his website as [GotW 29](#).

See? Told you it was easy!

Added June 2000: The May 2000 issue of C++ Report contains a fascinating [article](#) by Matt Austern (yes, *the* Matt Austern) on why case-insensitive comparisons are not as easy as they seem, and why creating a class is the *wrong* way to go about it in production code. (The GotW answer mentions one of the principle difficulties; his article mentions more.)

Basically, this is "easy" only if you ignore some things, things which may be too important to your program to ignore. (I chose to ignore them when originally writing this entry, and am surprised that nobody ever called me on it...) The GotW question and answer remain useful instructional tools, however.

Added September 2000: James Kanze provided a link to a [Unicode Technical Report discussing case handling](#), which provides some very good information.

7.1.3 Arbitrary Character Types

The `std::basic_string` is tantalizingly general, in that it is parameterized on the type of the characters which it holds. In theory, you could whip up a Unicode character class and instantiate `std::basic_string<my_unicode_char>`, or assuming that integers are wider than characters on your platform, maybe just declare variables of type `std::basic_string<int>`.

That's the theory. Remember however that `basic_string` has additional type parameters, which take default arguments based on the character type (called `CharT` here):

```
template <typename CharT,
typename Traits = char_traits<CharT>,
typename Alloc = allocator<CharT> >
class basic_string { .... };
```

Now, `allocator<CharT>` will probably Do The Right Thing by default, unless you need to implement your own allocator for your characters.

But `char_traits` takes more work. The `char_traits` template is *declared* but not *defined*. That means there is only

```
template <typename CharT>
struct char_traits
{
    static void foo (type1 x, type2 y);
    ...
};
```

and functions such as `char_traits<CharT>::foo()` are not actually defined anywhere for the general case. The C++ standard permits this, because writing such a definition to fit all possible `CharT`'s cannot be done.

The C++ standard also requires that `char_traits` be specialized for instantiations of `char` and `wchar_t`, and it is these template specializations that permit entities like `basic_string<char, char_traits<char>>` to work.

If you want to use character types other than `char` and `wchar_t`, such as `unsigned char` and `int`, you will need suitable specializations for them. For a time, in earlier versions of GCC, there was a mostly-correct implementation that let programmers be lazy but it broke under many situations, so it was removed. GCC 3.4 introduced a new implementation that mostly works and can be specialized even for `int` and other built-in types.

If you want to use your own special character class, then you have [a lot of work to do](#), especially if you wish to use i18n features (facets require traits information but don't have a traits argument).

Another example of how to specialize `char_traits` was given [on the mailing list](#) and at a later date was put into the file `include/ext/p`. We agree that the way it's used with `basic_string` (scroll down to `main()`) doesn't look nice, but that's because [the nice-looking first attempt](#) turned out to [not be conforming C++](#), due to the rule that `CharT` must be a POD. (See how tricky this is?)

7.1.4 Tokenizing

The Standard C (and C++) function `strtok()` leaves a lot to be desired in terms of user-friendliness. It's unintuitive, it destroys the character string on which it operates, and it requires you to handle all the memory problems. But it does let the client code decide what to use to break the string into pieces; it allows you to choose the "whitespace," so to speak.

A C++ implementation lets us keep the good things and fix those annoyances. The implementation here is more intuitive (you only call it once, not in a loop with varying argument), it does not affect the original string at all, and all the memory allocation is handled for you.

It's called `stringtok`, and it's a template function. Sources are as below, in a less-portable form than it could be, to keep this example simple (for example, see the comments on what kind of string it will accept).

```

#include <string>
template <typename Container>
void
stringtok(Container &container, string const &in,
          const char * const delimiters = " \t\n")
{
    const string::size_type len = in.length();
    string::size_type i = 0;

    while (i < len)
    {
        // Eat leading whitespace
        i = in.find_first_not_of(delimiters, i);
        if (i == string::npos)
            return;    // Nothing left but white space

        // Find the end of the token
        string::size_type j = in.find_first_of(delimiters, i);

        // Push token
        if (j == string::npos)
        {
            container.push_back(in.substr(i));
            return;
        }
        else
            container.push_back(in.substr(i, j-i));

        // Set up for next loop
        i = j + 1;
    }
}

```

The author uses a more general (but less readable) form of it for parsing command strings and the like. If you compiled and ran this code using it:

```

std::list<string> ls;
stringtok (ls, " this \t is\t\n a test ");
for (std::list<string>const_iterator i = ls.begin();
i != ls.end(); ++i)
{
    std::cerr << ':' << (*i) << ":\n";
}

```

You would see this as output:

```

:this:
:is:
:a:
:test:

```

with all the whitespace removed. The original `s` is still available for use, `ls` will clean up after itself, and `ls.size()` will return how many tokens there were.

As always, there is a price paid here, in that `stringtok` is not as fast as `strtok`. The other benefits usually outweigh that, however.

Added February 2001: Mark Wilden pointed out that the standard `std::getline()` function can be used with standard `istreamstreams` to perform tokenizing as well. Build an `istringstream` from the input text, and then use `std::getline` with varying delimiters (the three-argument signature) to extract tokens into a string.

7.1.5 Shrink to Fit

From GCC 3.4 calling `s.reserve(res)` on a string `s` with `res < s.capacity()` will reduce the string's capacity to `std::max(s.size(), res)`.

This behaviour is suggested, but not required by the standard. Prior to GCC 3.4 the following alternative can be used instead

```
std::string(str.data(), str.size()).swap(str);
```

This is similar to the idiom for reducing a `vector`'s memory usage (see [this FAQ entry](#)) but the regular copy constructor cannot be used because `libstdc++`'s `string` is Copy-On-Write in GCC 3.

From GCC 4.5 in **C++11** mode you can call `s.shrink_to_fit()` to achieve the same effect as `s.reserve(s.size())`.

7.1.6 CString (MFC)

A common lament seen in various newsgroups deals with the Standard `string` class as opposed to the Microsoft Foundation Class called `CString`. Often programmers realize that a standard portable answer is better than a proprietary nonportable one, but in porting their application from a Win32 platform, they discover that they are relying on special functions offered by the `CString` class.

Things are not as bad as they seem. In [this message](#), Joe Buck points out a few very important things:

- The Standard `string` supports all the operations that `CString` does, with three exceptions.
- Two of those exceptions (whitespace trimming and case conversion) are trivial to implement. In fact, we do so on this page.
- The third is `CString::Format`, which allows formatting in the style of `sprintf`. This deserves some mention:

The old `libg++` library had a function called `form()`, which did much the same thing. But for a Standard solution, you should use the `stringstream` classes. These are the bridge between the `iostream` hierarchy and the `string` class, and they operate with regular streams seamlessly because they inherit from the `iostream` hierarchy. An quick example:

```
#include <iostream>
#include <string>
#include <sstream>

string f (string& incoming)      // incoming is "foo N"
{
    istreamstream    incoming_stream(incoming);
    string           the_word;
    int             the_number;

    incoming_stream >> the_word      // extract "foo"
                   >> the_number;   // extract N

    ostreamstream    output_stream;
    output_stream << "The word was " << the_word
                  << " and 3*N was " << (3*the_number);

    return output_stream.str();
}
```

A serious problem with `CString` is a design bug in its memory allocation. Specifically, quoting from that same message:

`CString` suffers from a common programming error that results in poor performance. Consider the following code:

```
CString n_copies_of (const CString& foo, unsigned n)
{
    CString tmp;
```

```
for (unsigned i = 0; i < n; i++)
    tmp += foo;
return tmp;
}
```

This function is $O(n^2)$, not $O(n)$. The reason is that each `+=` causes a reallocation and copy of the existing string. Microsoft applications are full of this kind of thing (quadratic performance on tasks that can be done in linear time) -- on the other hand, we should be thankful, as it's created such a big market for high-end ix86 hardware. :-)

If you replace `CString` with `string` in the above function, the performance is $O(n)$.

Joe Buck also pointed out some other things to keep in mind when comparing `CString` and the Standard `string` class:

- `CString` permits access to its internal representation; coders who exploited that may have problems moving to `string`.
- Microsoft ships the source to `CString` (in the files `MFC\SRC\Str{core,ex}.cpp`), so you could fix the allocation bug and rebuild your MFC libraries. Note: *It looks like the `CString` shipped with VC++6.0 has fixed this, although it may in fact have been one of the VC++ SPs that did it.*
- `string` operations like this have $O(n)$ complexity *if the implementors do it correctly*. The `libstdc++` implementors did it correctly. Other vendors might not.
- While parts of the SGI STL are used in `libstdc++`, their `string` class is not. The SGI `string` is essentially `vector<char>` and does not do any reference counting like `libstdc++`'s does. (It is $O(n)$, though.) So if you're thinking about SGI's `string` or `rope` classes, you're now looking at four possibilities: `CString`, the `libstdc++` `string`, the SGI `string`, and the SGI `rope`, and this is all before any allocator or traits customizations! (More choices than you can shake a stick at -- want fries with that?)

Chapter 8

Localization

8.1 Locales

8.1.1 locale

Describes the basic locale object, including nested classes `id`, `facet`, and the reference-counted implementation object, class `_Impl`.

8.1.1.1 Requirements

Class `locale` is non-templatized and has two distinct types nested inside of it:

class facet 22.1.1.1.2 Class locale::facet

Facets actually implement locale functionality. For instance, a facet called `numput` is the data object that can be used to query for the thousands separator in the locale.

Literally, a facet is strictly defined:

- Containing the following public data member:

```
static locale::id id;
```

- Derived from another facet:

```
class gnu_codecvt: public std::ctype<user-defined-type>
```

Of interest in this class are the memory management options explicitly specified as an argument to facet's constructor. Each constructor of a facet class takes a `std::size_t __refs` argument: if `__refs == 0`, the facet is deleted when the locale containing it is destroyed. If `__refs == 1`, the facet is not destroyed, even when it is no longer referenced.

class id 22.1.1.1.3 - Class locale::id

Provides an index for looking up specific facets.

8.1.1.2 Design

The major design challenge is fitting an object-orientated and non-global locale design on top of POSIX and other relevant standards, which include the Single Unix (nee X/Open.)

Because C and earlier versions of POSIX fall down so completely, portability is an issue.

8.1.1.3 Implementation

8.1.1.3.1 Interacting with "C" locales

- `locale -a` displays available locales.

```
af_ZA
ar_AE
ar_AE.utf8
ar_BH
ar_BH.utf8
ar_DZ
ar_DZ.utf8
ar_EG
ar_EG.utf8
ar_IN
ar_IQ
ar_IQ.utf8
ar_JO
ar_JO.utf8
ar_KW
ar_KW.utf8
ar_LB
ar_LB.utf8
ar_LY
ar_LY.utf8
ar_MA
ar_MA.utf8
ar_OM
ar_OM.utf8
ar_QA
ar_QA.utf8
ar_SA
ar_SA.utf8
ar_SD
ar_SD.utf8
ar_SY
ar_SY.utf8
ar_TN
ar_TN.utf8
ar_YE
ar_YE.utf8
be_BY
be_BY.utf8
bg_BG
bg_BG.utf8
br_FR
bs_BA
C
ca_ES
ca_ES@euro
ca_ES.utf8
ca_ES.utf8@euro
cs_CZ
cs_CZ.utf8
cy_GB
da_DK
da_DK.iso885915
da_DK.utf8
de_AT
de_AT@euro
```

```
de_AT.utf8
de_AT.utf8@euro
de_BE
de_BE@euro
de_BE.utf8
de_BE.utf8@euro
de_CH
de_CH.utf8
de_DE
de_DE@euro
de_DE.utf8
de_DE.utf8@euro
de_LU
de_LU@euro
de_LU.utf8
de_LU.utf8@euro
el_GR
el_GR.utf8
en_AU
en_AU.utf8
en_BW
en_BW.utf8
en_CA
en_CA.utf8
en_DK
en_DK.utf8
en_GB
en_GB.iso885915
en_GB.utf8
en_HK
en_HK.utf8
en_IE
en_IE@euro
en_IE.utf8
en_IE.utf8@euro
en_IN
en_NZ
en_NZ.utf8
en_PH
en_PH.utf8
en_SG
en_SG.utf8
en_US
en_US.iso885915
en_US.utf8
en_ZA
en_ZA.utf8
en_ZW
en_ZW.utf8
es_AR
es_AR.utf8
es_BO
es_BO.utf8
es_CL
es_CL.utf8
es_CO
es_CO.utf8
es_CR
es_CR.utf8
es_DO
es_DO.utf8
es_EC
```

```
es_EC.utf8
es_ES
es_ES@euro
es_ES.utf8
es_ES.utf8@euro
es_GT
es_GT.utf8
es_HN
es_HN.utf8
es_MX
es_MX.utf8
es_NI
es_NI.utf8
es_PA
es_PA.utf8
es_PE
es_PE.utf8
es_PR
es_PR.utf8
es_PY
es_PY.utf8
es_SV
es_SV.utf8
es_US
es_US.utf8
es_UY
es_UY.utf8
es_VE
es_VE.utf8
et_EE
et_EE.utf8
eu_ES
eu_ES@euro
eu_ES.utf8
eu_ES.utf8@euro
fa_IR
fi_FI
fi_FI@euro
fi_FI.utf8
fi_FI.utf8@euro
fo_FO
fo_FO.utf8
fr_BE
fr_BE@euro
fr_BE.utf8
fr_BE.utf8@euro
fr_CA
fr_CA.utf8
fr_CH
fr_CH.utf8
fr_FR
fr_FR@euro
fr_FR.utf8
fr_FR.utf8@euro
fr_LU
fr_LU@euro
fr_LU.utf8
fr_LU.utf8@euro
ga_IE
ga_IE@euro
ga_IE.utf8
ga_IE.utf8@euro
```

```
gl_ES
gl_ES@euro
gl_ES.utf8
gl_ES.utf8@euro
gv_GB
gv_GB.utf8
he_IL
he_IL.utf8
hi_IN
hr_HR
hr_HR.utf8
hu_HU
hu_HU.utf8
id_ID
id_ID.utf8
is_IS
is_IS.utf8
it_CH
it_CH.utf8
it_IT
it_IT@euro
it_IT.utf8
it_IT.utf8@euro
iw_IL
iw_IL.utf8
ja_JP.eucjp
ja_JP.utf8
ka_GE
kl_GL
kl_GL.utf8
ko_KR.euckr
ko_KR.utf8
kw_GB
kw_GB.utf8
lt_LT
lt_LT.utf8
lv_LV
lv_LV.utf8
mi_NZ
mk_MK
mk_MK.utf8
mr_IN
ms_MY
ms_MY.utf8
mt_MT
mt_MT.utf8
nl_BE
nl_BE@euro
nl_BE.utf8
nl_BE.utf8@euro
nl_NL
nl_NL@euro
nl_NL.utf8
nl_NL.utf8@euro
nn_NO
nn_NO.utf8
no_NO
no_NO.utf8
oc_FR
pl_PL
pl_PL.utf8
POSIX
```

```
pt_BR
pt_BR.utf8
pt_PT
pt_PT@euro
pt_PT.utf8
pt_PT.utf8@euro
ro_RO
ro_RO.utf8
ru_RU
ru_RU.koi8r
ru_RU.utf8
ru_UA
ru_UA.utf8
se_NO
sk_SK
sk_SK.utf8
sl_SI
sl_SI.utf8
sq_AL
sq_AL.utf8
sr_YU
sr_YU@cyrillic
sr_YU.utf8
sr_YU.utf8@cyrillic
sv_FI
sv_FI@euro
sv_FI.utf8
sv_FI.utf8@euro
sv_SE
sv_SE.iso885915
sv_SE.utf8
ta_IN
te_IN
tg_TJ
th_TH
th_TH.utf8
tl_PH
tr_TR
tr_TR.utf8
uk_UA
uk_UA.utf8
ur_PK
uz_UZ
vi_VN
vi_VN.tcvn
wa_BE
wa_BE@euro
yi_US
zh_CN
zh_CN.gb18030
zh_CN.gbk
zh_CN.utf8
zh_HK
zh_HK.utf8
zh_TW
zh_TW.euctw
zh_TW.utf8
```

- ``locale`` displays environmental variables that impact how `locale("")` will be deduced.

```
LANG=en_US
```

```
LC_CTYPE="en_US"
LC_NUMERIC="en_US"
LC_TIME="en_US"
LC_COLLATE="en_US"
LC_MONETARY="en_US"
LC_MESSAGES="en_US"
LC_PAPER="en_US"
LC_NAME="en_US"
LC_ADDRESS="en_US"
LC_TELEPHONE="en_US"
LC_MEASUREMENT="en_US"
LC_IDENTIFICATION="en_US"
LC_ALL=
```

From Josuttis, p. 697-698, which says, that "there is only *one* relation (of the C++ locale mechanism) to the C locale mechanism: the global C locale is modified if a named C++ locale object is set as the global locale" (emphasis Paolo), that is:

```
std::locale::global(std::locale(""));
```

affects the C functions as if the following call was made:

```
std::setlocale(LC_ALL, "");
```

On the other hand, there is *no* vice versa, that is, calling `setlocale` has *no* whatsoever on the C++ locale mechanism, in particular on the working of `locale("")`, which constructs the locale object from the environment of the running program, that is, in practice, the set of `LC_ALL`, `LANG`, etc. variable of the shell.

8.1.1.4 Future

- Locale initialization: at what point does `_S_classic`, `_S_global` get initialized? Can named locales assume this initialization has already taken place?
- Document how named locales error check when filling data members. I.e., a `fr_FR` locale that doesn't have `numpunct::truename()`: does it use "true"? Or is it a blank string? What's the convention?
- Explain how locale aliasing happens. When does "de_DE" use "de" information? What is the rule for locales composed of just an ISO language code (say, "de") and locales with both an ISO language code and ISO country code (say, "de_DE").
- What should non-required facet instantiations do? If the generic implementation is provided, then how to end-users provide specializations?

8.1.1.5 Bibliography

- [20] Roland McGrathUlrich Drepper, Copyright © 2007 FSF, Chapters 6 Character Set Handling and 7 Locales and Internationalization .
- [21] Ulrich Drepper, Copyright © 2002 .
- [22] , Copyright © 1998 ISO.
- [23] , Copyright © 1999 ISO.
- [24] *System Interface Definitions, Issue 7 (IEEE Std. 1003.1-2008)* , Copyright © 2008 The Open Group/The Institute of Electrical and Electronics Engineers, Inc. .
- [25] Bjarne Stroustrup, Copyright © 2000 Addison Wesley, Inc., Appendix D, Addison Wesley .
- [26] Angelika LangerKlaus Kreft, Advanced Programmer's Guide and Reference , Copyright © 2000 Addison Wesley Longman, Inc., Addison Wesley Longman .

8.2 Facets

8.2.1 ctype

8.2.1.1 Implementation

8.2.1.1.1 Specializations

For the required specialization `codecvt<wchar_t, char, mbstate_t>`, conversions are made between the internal character set (always UCS4 on GNU/Linux) and whatever the currently selected locale for the `LC_CTYPE` category implements.

The two required specializations are implemented as follows:

```
ctype<char>
```

This is simple specialization. Implementing this was a piece of cake.

```
ctype<wchar_t>
```

This specialization, by specifying all the template parameters, pretty much ties the hands of implementors. As such, the implementation is straightforward, involving `wcsrtombs` for the conversions between `char` to `wchar_t` and `wcsrtombs` for conversions between `wchar_t` and `char`.

Neither of these two required specializations deals with Unicode characters.

8.2.1.2 Future

- How to deal with the global locale issue?
- How to deal with types other than `char`, `wchar_t`?
- Overlap between `codecvt/ctype`: narrow/widen
- mask typedef in `codecvt_base`, argument types in `codecvt`. what is know about this type?
- Why mask* argument in `codecvt`?
- Can this be made (more) generic? is there a simple way to straighten out the configure-time mess that is a by-product of this class?
- Get the `ctype<wchar_t>::mask` stuff under control. Need to make some kind of static table, and not do lookup every time somebody hits the `do_is...` functions. Too bad we can't just redefine mask for `ctype<wchar_t>`
- Rename abstract base class. See if just smash-overriding is a better approach. Clarify, add sanity to naming.

8.2.1.3 Bibliography

- [27] Roland McGrath Ulrich Drepper, Copyright © 2007 FSF, Chapters 6 Character Set Handling and 7 Locales and Internationalization.
- [28] Ulrich Drepper, Copyright © 2002 .
- [29] , Copyright © 1998 ISO.
- [30] , Copyright © 1999 ISO.
- [31] *The Open Group Base Specifications, Issue 6 (IEEE Std. 1003.1-2004)* , Copyright © 1999 The Open Group/The Institute of Electrical and Electronics Engineers, Inc..
- [32] Bjarne Stroustrup, Copyright © 2000 Addison Wesley, Inc., Appendix D, Addison Wesley .
- [33] Angelika LangerKlaus Kreft, Advanced Programmer's Guide and Reference , Copyright © 2000 Addison Wesley Longman, Inc., Addison Wesley Longman .

Provides a way to see if the given `encoding_state` object has been properly initialized. If the string literals describing the desired internal and external encoding are not valid, initialization will fail, and this will return false. If the internal and external encodings are valid, but `iconv_open` could not allocate conversion descriptors, this will also return false. Otherwise, the object is ready to convert and will return true.

```
encoding_state(const encoding_state&)
```

As `iconv` allocates memory and sets up conversion descriptors, the copy constructor can only copy the member data pertaining to the internal and external code conversions, and not the conversion descriptors themselves.

Definitions for all the required `codecvt` member functions are provided for this specialization, and usage of `codecvt<internal character type, external character type, encoding_state>` is consistent with other `codecvt` usage.

8.2.2.4 Use

A conversion involving a string literal.

```
typedef codecvt_base::result          result;
typedef unsigned short                unicode_t;
typedef unicode_t                     int_type;
typedef char                           ext_type;
typedef encoding_state                state_type;
typedef codecvt<int_type, ext_type, state_type> unicode_codecvt;

const ext_type*      e_lit = "black pearl jasmine tea";
int                  size = strlen(e_lit);
int_type             i_lit_base[24] =
{ 25088, 27648, 24832, 25344, 27392, 8192, 28672, 25856, 24832, 29184,
  27648, 8192, 27136, 24832, 29440, 27904, 26880, 28160, 25856, 8192, 29696,
  25856, 24832, 2560
};
const int_type*      i_lit = i_lit_base;
const ext_type*      efrom_next;
const int_type*      ifrom_next;
ext_type*            e_arr = new ext_type[size + 1];
ext_type*            eto_next;
int_type*            i_arr = new int_type[size + 1];
int_type*            ito_next;

// construct a locale object with the specialized facet.
locale                loc(locale::classic(), new unicode_codecvt);
// sanity check the constructed locale has the specialized facet.
VERIFY( has_facet<unicode_codecvt>(loc) );
const unicode_codecvt& cvt = use_facet<unicode_codecvt>(loc);
// convert between const char* and unicode strings
unicode_codecvt::state_type state01("UNICODE", "ISO_8859-1");
initialize_state(state01);
result r1 = cvt.in(state01, e_lit, e_lit + size, efrom_next,
    i_arr, i_arr + size, ito_next);
VERIFY( r1 == codecvt_base::ok );
VERIFY( !int_traits::compare(i_arr, i_lit, size) );
VERIFY( efrom_next == e_lit + size );
VERIFY( ito_next == i_arr + size );
```

8.2.2.5 Future

- a. things that are sketchy, or remain unimplemented: `do_encoding`, `max_length` and `length` member functions are only weakly implemented. I have no idea how to do this correctly, and in a generic manner. Nathan?
- b. conversions involving `std::string`

- how should operators `!=` and `==` work for string of different/same encoding?
- what is equal? A byte by byte comparison or an encoding then byte comparison?
- conversions between narrow, wide, and unicode strings
- c. conversions involving `std::filebuf` and `std::ostream`
 - how to initialize the state object in a standards-conformant manner?
 - how to synchronize the "C" and "C++" conversion information?
 - `wchar_t/char` internal buffers and conversions between internal/external buffers?

8.2.2.6 Bibliography

- [34] Roland McGrath Ulrich Drepper, Copyright © 2007 FSF, Chapters 6 Character Set Handling and 7 Locales and Internationalization .
- [35] Ulrich Drepper, Copyright © 2002 .
- [36] , Copyright © 1998 ISO.
- [37] , Copyright © 1999 ISO.
- [38] *System Interface Definitions, Issue 7 (IEEE Std. 1003.1-2008)* , Copyright © 2008 The Open Group/The Institute of Electrical and Electronics Engineers, Inc. .
- [39] Bjarne Stroustrup, Copyright © 2000 Addison Wesley, Inc., Appendix D, Addison Wesley .
- [40] Angelika LangerKlaus Kreft, Advanced Programmer's Guide and Reference , Copyright © 2000 Addison Wesley Longman, Inc., Addison Wesley Longman .
- [41] Clive Feather, *A brief description of Normative Addendum 1* , Extended Character Sets.
- [42] Bruno Haible, *The Unicode HOWTO*
- [43] Markus Khun, *UTF-8 and Unicode FAQ for Unix/Linux*

8.2.3 messages

The `std::messages` facet implements message retrieval functionality equivalent to Java's `java.text.MessageFormat` using either GNU `gettext` or IEEE 1003.1-200 functions.

8.2.3.1 Requirements

The `std::messages` facet is probably the most vaguely defined facet in the standard library. It's assumed that this facility was built into the standard library in order to convert string literals from one locale to the other. For instance, converting the "C" locale's `const char* c = "please"` to a German-localized "bitte" during program execution.

22.2.7.1 - Template class `messages` [`lib.locale.messages`]

This class has three public member functions, which directly correspond to three protected virtual member functions.

The public member functions are:

```
catalog open(const string&, const locale&) const
string_type get(catalog, int, int, const string_type&) const
void close(catalog) const
```

While the virtual functions are:

```
catalog do_open(const string& name, const locale& loc) const
```


-
- [51] *API Specifications, Java Platform* , `java.util.Properties`, `java.text.MessageFormat`, `java.util.Locale`, `java.util.ResourceBundle` .
- [52] *GNU gettext tools, version 0.10.38, Native Language Support Library and Tools.*
-

Chapter 9

Containers

9.1 Sequences

9.1.1 list

9.1.1.1 list::size() is O(n)

Yes it is, at least using the **old ABI**, and that's okay. This is a decision that we preserved when we imported SGI's STL implementation. The following is quoted from **their FAQ**:

The size() member function, for list and slist, takes time proportional to the number of elements in the list. This was a deliberate tradeoff. The only way to get a constant-time size() for linked lists would be to maintain an extra member variable containing the list's size. This would require taking extra time to update that variable (it would make splice() a linear time operation, for example), and it would also make the list larger. Many list algorithms don't require that extra word (algorithms that do require it might do better with vectors than with lists), and, when it is necessary to maintain an explicit size count, it's something that users can do themselves.

This choice is permitted by the C++ standard. The standard says that size() “should” be constant time, and “should” does not mean the same thing as “shall”. This is the officially recommended ISO wording for saying that an implementation is supposed to do something unless there is a good reason not to.

One implication of linear time size(): you should never write

```
if (L.size() == 0)
    ...
```

Instead, you should write

```
if (L.empty())
    ...
```

9.2 Associative

9.2.1 Insertion Hints

Section [23.1.2], Table 69, of the C++ standard lists this function for all of the associative containers (map, set, etc):

```
a.insert(p,t);
```



```

namespace std
{
    template<>
        struct __is_fast_hash<hasher> : std::false_type
        { };
}

```

9.4 Interacting with C

9.4.1 Containers vs. Arrays

You're writing some code and can't decide whether to use builtin arrays or some kind of container. There are compelling reasons to use one of the container classes, but you're afraid that you'll eventually run into difficulties, change everything back to arrays, and then have to change all the code that uses those data types to keep up with the change.

If your code makes use of the standard algorithms, this isn't as scary as it sounds. The algorithms don't know, nor care, about the kind of "container" on which they work, since the algorithms are only given endpoints to work with. For the container classes, these are iterators (usually `begin()` and `end()`, but not always). For builtin arrays, these are the address of the first element and the **past-the-end** element.

Some very simple wrapper functions can hide all of that from the rest of the code. For example, a pair of functions called `beginof` can be written, one that takes an array, another that takes a vector. The first returns a pointer to the first element, and the second returns the vector's `begin()` iterator.

The functions should be made template functions, and should also be declared inline. As pointed out in the comments in the code below, this can lead to `beginof` being optimized out of existence, so you pay absolutely nothing in terms of increased code size or execution time.

The result is that if all your algorithm calls look like

```
std::transform(beginof(foo), endof(foo), beginof(foo), SomeFunction);
```

then the type of `foo` can change from an array of ints to a vector of ints to a deque of ints and back again, without ever changing any client code.

```

// beginof
template<typename T>
    inline typename vector<T>::iterator
    beginof(vector<T> &v)
    { return v.begin(); }

template<typename T, unsigned int sz>
    inline T*
    beginof(T (&array)[sz]) { return array; }

// endof
template<typename T>
    inline typename vector<T>::iterator
    endof(vector<T> &v)
    { return v.end(); }

template<typename T, unsigned int sz>
    inline T*
    endof(T (&array)[sz]) { return array + sz; }

// lengthof
template<typename T>
    inline typename vector<T>::size_type
    lengthof(vector<T> &v)

```

```
{ return v.size(); }
```

```
template<typename T, unsigned int sz>  
    inline unsigned int  
    lengthof(T (&)[sz]) { return sz; }
```

Astute readers will notice two things at once: first, that the container class is still a `vector<T>` instead of a more general `Container<T>`. This would mean that three functions for `deque` would have to be added, another three for `list`, and so on. This is due to problems with getting template resolution correct; I find it easier just to give the extra three lines and avoid confusion.

Second, the line

```
    inline unsigned int lengthof (T (&)[sz]) { return sz; }
```

looks just weird! Hint: unused parameters can be left nameless.

Chapter 10

Iterators

10.1 Predefined

10.1.1 Iterators vs. Pointers

The following FAQ [entry](#) points out that iterators are not implemented as pointers. They are a generalization of pointers, but they are implemented in libstdc++ as separate classes.

Keeping that simple fact in mind as you design your code will prevent a whole lot of difficult-to-understand bugs.

You can think of it the other way 'round, even. Since iterators are a generalization, that means that *pointers* are *iterators*, and that pointers can be used whenever an iterator would be. All those functions in the Algorithms section of the Standard will work just as well on plain arrays and their pointers.

That doesn't mean that when you pass in a pointer, it gets wrapped into some special delegating iterator-to-pointer class with a layer of overhead. (If you think that's the case anywhere, you don't understand templates to begin with...) Oh, no; if you pass in a pointer, then the compiler will instantiate that template using `T*` as a type, and good old high-speed pointer arithmetic as its operations, so the resulting code will be doing exactly the same things as it would be doing if you had hand-coded it yourself (for the 273rd time).

How much overhead *is* there when using an iterator class? Very little. Most of the layering classes contain nothing but typedefs, and typedefs are "meta-information" that simply tell the compiler some nicknames; they don't create code. That information gets passed down through inheritance, so while the compiler has to do work looking up all the names, your runtime code does not. (This has been a prime concern from the beginning.)

10.1.2 One Past the End

This starts off sounding complicated, but is actually very easy, especially towards the end. Trust me.

Beginners usually have a little trouble understand the whole 'past-the-end' thing, until they remember their early algebra classes (see, they *told* you that stuff would come in handy!) and the concept of half-open ranges.

First, some history, and a reminder of some of the funkier rules in C and C++ for builtin arrays. The following rules have always been true for both languages:

1. You can point anywhere in the array, *or to the first element past the end of the array*. A pointer that points to one past the end of the array is guaranteed to be as unique as a pointer to somewhere inside the array, so that you can compare such pointers safely.
2. You can only dereference a pointer that points into an array. If your array pointer points outside the array -- even to just one past the end -- and you dereference it, Bad Things happen.

Chapter 11

Algorithms

The neatest accomplishment of the algorithms section is that all the work is done via iterators, not containers directly. This means two important things:

1. Anything that behaves like an iterator can be used in one of these algorithms. Raw pointers make great candidates, thus built-in arrays are fine containers, as well as your own iterators.
2. The algorithms do not (and cannot) affect the container as a whole; only the things between the two iterator endpoints. If you pass a range of iterators only enclosing the middle third of a container, then anything outside that range is inviolate.

Even strings can be fed through the algorithms here, although the string class has specialized versions of many of these functions (for example, `string::find()`). Most of the examples on this page will use simple arrays of integers as a playground for algorithms, just to keep things simple. The use of N as a size in the examples is to keep things easy to read but probably won't be valid code. You can use wrappers such as those described in the [containers section](#) to keep real code readable.

The single thing that trips people up the most is the definition of *range* used with iterators; the famous "past-the-end" rule that everybody loves to hate. The [iterators section](#) of this document has a complete explanation of this simple rule that seems to cause so much confusion. Once you get *range* into your head (it's not that hard, honest!), then the algorithms are a cakewalk.

11.1 Mutating

11.1.1 swap

11.1.1.1 Specializations

If you call `std::swap(x, y);` where x and y are standard containers, then the call will automatically be replaced by a call to `x.swap(y);` instead.

This allows member functions of each container class to take over, and containers' swap functions should have $O(1)$ complexity according to the standard. (And while "should" allows implementations to behave otherwise and remain compliant, this implementation does in fact use constant-time swaps.) This should not be surprising, since for two containers of the same type to swap contents, only some internal pointers to storage need to be exchanged.

Chapter 12

Numerics

12.1 Complex

12.1.1 complex Processing

Using `complex<>` becomes even more comple- er, sorry, *complicated*, with the not-quite-gratuitously-incompatible addition of complex types to the C language. David Tribble has compiled a list of C++98 and C99 conflict points; his description of C's new type versus those of C++ and how to get them playing together nicely is [here](#).

`complex<>` is intended to be instantiated with a floating-point type. As long as you meet that and some other basic requirements, then the resulting instantiation has all of the usual math operators defined, as well as definitions of `op<<` and `op>>` that work with iostreams: `op<<` prints `(u, v)` and `op>>` can read `u`, `(u)`, and `(u, v)`.

As an extension to C++11 and for increased compatibility with C, `<complex.h>` includes both `<complex>` and the C99 `<complex.h>` (if the C library provides it).

12.2 Generalized Operations

There are four generalized functions in the `<numeric>` header that follow the same conventions as those in `<algorithm>`. Each of them is overloaded: one signature for common default operations, and a second for fully general operations. Their names are self-explanatory to anyone who works with numerics on a regular basis:

- `accumulate`
- `inner_product`
- `partial_sum`
- `adjacent_difference`

Here is a simple example of the two forms of `accumulate`.

```
int    ar[50];
int    someval = somefunction();

// ...initialize members of ar to something...

int    sum      = std::accumulate(ar, ar+50, 0);
int    sum_stuff = std::accumulate(ar, ar+50, someval);
int    product  = std::accumulate(ar, ar+50, 1, std::multiplies<int>());
```

The first call adds all the members of the array, using zero as an initial value for `sum`. The second does the same, but uses `someval` as the starting value (thus, `sum_stuff == sum + someval`). The final call uses the second of the two signatures, and multiplies all the members of the array; here we must obviously use 1 as a starting value instead of 0.

The other three functions have similar dual-signature forms.

12.3 Interacting with C

12.3.1 Numerics vs. Arrays

One of the major reasons why FORTRAN can chew through numbers so well is that it is defined to be free of pointer aliasing, an assumption that C89 is not allowed to make, and neither is C++98. C99 adds a new keyword, `restrict`, to apply to individual pointers. The C++ solution is contained in the library rather than the language (although many vendors can be expected to add this to their compilers as an extension).

That library solution is a set of two classes, five template classes, and "a whole bunch" of functions. The classes are required to be free of pointer aliasing, so compilers can optimize the daylight out of them the same way that they have been for FORTRAN. They are collectively called `valarray`, although strictly speaking this is only one of the five template classes, and they are designed to be familiar to people who have worked with the BLAS libraries before.

12.3.2 C99

In addition to the other topics on this page, we'll note here some of the C99 features that appear in `libstdc++`.

The C99 features depend on the `--enable-c99` configure flag. This flag is already on by default, but it can be disabled by the user. Also, the configuration machinery will disable it if the necessary support for C99 (e.g., header files) cannot be found.

As of GCC 3.0, C99 support includes classification functions such as `isnormal`, `isgreater`, `isnan`, etc. The functions used for 'long long' support such as `strtoll` are supported, as is the `lldiv_t` typedef. Also supported are the wide character functions using 'long long', like `wcstoll`.

Chapter 13

Input and Output

13.1 `iostream` Objects

To minimize the time you have to wait on the compiler, it's good to only include the headers you really need. Many people simply include `<iostream>` when they don't need to -- and that can *penalize your runtime as well*. Here are some tips on which header to use for which situations, starting with the simplest.

`<iosfwd>` should be included whenever you simply need the *name* of an I/O-related class, such as "ofstream" or "basic_streambuf". Like the name implies, these are forward declarations. (A word to all you fellow old school programmers: trying to forward declare classes like "class istream;" won't work. Look in the `<iosfwd>` header if you'd like to know why.) For example,

```
#include <iosfwd>

class MyClass
{
....
std::ifstream&    input_file;
};

extern std::ostream& operator<< (std::ostream&, MyClass&);
```

`<ios>` declares the base classes for the entire I/O stream hierarchy, `std::ios_base` and `std::basic_ios<charT>`, the counting types `std::streamoff` and `std::streamsize`, the file positioning type `std::fpos`, and the various manipulators like `std::hex`, `std::fixed`, `std::noshowbase`, and so forth.

The `ios_base` class is what holds the format flags, the state flags, and the functions which change them (`setf()`, `width()`, `precision()`, etc). You can also store extra data and register callback functions through `ios_base`, but that has been historically underused. Anything which doesn't depend on the type of characters stored is consolidated here.

The class template `basic_ios` is the highest class template in the hierarchy; it is the first one depending on the character type, and holds all general state associated with that type: the pointer to the polymorphic stream buffer, the facet information, etc.

`<streambuf>` declares the class template `basic_streambuf`, and two standard instantiations, `streambuf` and `wstreambuf`. If you need to work with the vastly useful and capable stream buffer classes, e.g., to create a new form of storage transport, this header is the one to include.

`<istream>` and `<ostream>` are the headers to include when you are using the overloaded `>>` and `<<` operators, or any of the other abstract stream formatting functions. For example,

```
#include <istream>

std::ostream& operator<< (std::ostream& os, MyClass& c)
{
    return os << c.data1() << c.data2();
}
```



```

#include <locale>
#include <cstdio>

class outbuf : public std::streambuf
{
    protected:
/* central output function
 * - print characters in uppercase mode
 */
virtual int_type overflow (int_type c) {
    if (c != EOF) {
        // convert lowercase to uppercase
        c = std::toupper(static_cast<char>(c), getloc());

        // and write the character to the standard output
        if (putchar(c) == EOF) {
            return EOF;
        }
        return c;
    }
};

int main()
{
    // create special output buffer
    outbuf ob;
    // initialize output stream with that output buffer
    std::ostream out(&ob);

    out << "31 hexadecimal: "
        << std::hex << 31 << std::endl;
    return 0;
}

```

Try it yourself! More examples can be found in 3.1.x code, in `include/ext/*_filebuf.h`, and in the article [Filtering Streambufs](#) by James Kanze.

13.2.2 Buffering

First, are you sure that you understand buffering? Particularly the fact that C++ may not, in fact, have anything to do with it?

The rules for buffering can be a little odd, but they aren't any different from those of C. (Maybe that's why they can be a bit odd.) Many people think that writing a newline to an output stream automatically flushes the output buffer. This is true only when the output stream is, in fact, a terminal and not a file or some other device -- and *that* may not even be true since C++ says nothing about files nor terminals. All of that is system-dependent. (The "newline-buffer-flushing only occurring on terminals" thing is mostly true on Unix systems, though.)

Some people also believe that sending `endl` down an output stream only writes a newline. This is incorrect; after a newline is written, the buffer is also flushed. Perhaps this is the effect you want when writing to a screen -- get the text out as soon as possible, etc -- but the buffering is largely wasted when doing this to a file:

```

output << "a line of text" << endl;
output << some_data_variable << endl;
output << "another line of text" << endl;

```

The proper thing to do in this case to just write the data out and let the libraries and the system worry about the buffering. If you need a newline, just write a newline:

```

output << "a line of text\n"
    << some_data_variable << '\n'

```


13.5.2 Performance

Pathetic Performance? Ditch C.

It sounds like a flame on C, but it isn't. Really. Calm down. I'm just saying it to get your attention.

Because the C++ library includes the C library, both C-style and C++-style I/O have to work at the same time. For example:

```
#include <iostream>
#include <cstdio>

std::cout << "Hel";
std::printf ("lo, worl");
std::cout << "d!\n";
```

This must do what you think it does.

Alert members of the audience will immediately notice that buffering is going to make a hash of the output unless special steps are taken.

The special steps taken by libstdc++, at least for version 3.0, involve doing very little buffering for the standard streams, leaving most of the buffering to the underlying C library. (This kind of thing is tricky to get right.) The upside is that correctness is ensured. The downside is that writing through `cout` can quite easily lead to awful performance when the C++ I/O library is layered on top of the C I/O library (as it is for 3.0 by default). Some patches have been applied which improve the situation for 3.1.

However, the C and C++ standard streams only need to be kept in sync when both libraries' facilities are in use. If your program only uses C++ I/O, then there's no need to sync with the C streams. The right thing to do in this case is to call

```
#include any of the I/O headers such as ios, iostream, etc

std::ios::sync_with_stdio(false);
```

You must do this before performing any I/O via the C++ stream objects. Once you call this, the C++ streams will operate independently of the (unused) C streams. For GCC 3.x, this means that `cout` and company will become fully buffered on their own.

Note, by the way, that the synchronization requirement only applies to the standard streams (`cin`, `cout`, `cerr`, `clog`, and their wide-character counterparts). File stream objects that you declare yourself have no such requirement and are fully buffered.

Chapter 14

Atomics

Facilities for atomic operations.

14.1 API Reference

All items are declared in the standard header file `atomic`.

Set of typedefs that map `int` to `atomic_int`, and so on for all builtin integral types. Global enumeration `memory_order` to control memory ordering. Also includes `atomic`, a class template with member functions such as `load` and `store` that is instantiable such that `atomic_int` is the base class of `atomic<int>`.

Full API details.

Chapter 15

Concurrency

Facilities for concurrent operation, and control thereof.

15.1 API Reference

All items are declared in one of four standard header files.

In header `mutex`, class template `mutex` and variants, class `once_flag`, and class template `unique_lock`.

In header `condition_variable`, classes `condition_variable` and `condition_variable_any`.

In header `thread`, class `thread` and namespace `this_thread`.

In header `future`, class template `future` and class template `shared_future`, class template `promise`, and `packaged_task`.

Full API details.

Part III

Extensions

Here we will make an attempt at describing the non-Standard extensions to the library. Some of these are from older versions of standard library components, namely SGI's STL, and some of these are GNU's.

Before you leap in and use any of these extensions, be aware of two things:

1. Non-Standard means exactly that.

The behavior, and the very existence, of these extensions may change with little or no warning. (Ideally, the really good ones will appear in the next revision of C++.) Also, other platforms, other compilers, other versions of g++ or libstdc++ may not recognize these names, or treat them differently, or...

2. You should know how to access these headers properly.

Chapter 16

Compile Time Checks

Also known as concept checking.

In 1999, SGI added *concept checkers* to their implementation of the STL: code which checked the template parameters of instantiated pieces of the STL, in order to insure that the parameters being used met the requirements of the standard. For example, the Standard requires that types passed as template parameters to `vector` be “Assignable” (which means what you think it means). The checking was done during compilation, and none of the code was executed at runtime.

Unfortunately, the size of the compiler files grew significantly as a result. The checking code itself was cumbersome. And bugs were found in it on more than one occasion.

The primary author of the checking code, Jeremy Siek, had already started work on a replacement implementation. The new code has been formally reviewed and accepted into [the Boost libraries](#), and we are pleased to incorporate it into the GNU C++ library.

The new version imposes a much smaller space overhead on the generated object file. The checks are also cleaner and easier to read and understand.

They are off by default for all GCC 3.0 and all later versions. They can be enabled at configure time with `--enable-concept-checks`. You can enable them on a per-translation-unit basis with `#define _GLIBCXX_CONCEPT_CHECKS` for GCC 3.4 and higher (or with `#define _GLIBCPP_CONCEPT_CHECKS` for versions 3.1, 3.2 and 3.3).

Please note that the concept checks only validate the requirements of the old C++03 standard and reject some valid code that meets the relaxed requirements of C++11 and later standards. C++11 was expected to have first-class support for template parameter constraints based on concepts in the core language. This would have obviated the need for the library-simulated concept checking described above, but was not part of C++11. C++20 adds a different model of concepts, which is now used to constrain some new parts of the C++20 library, e.g. the `<ranges>` header and the new overloads in the `<algorithm>` header for working with ranges. The old library-simulated concept checks might be removed at a future date.

Chapter 17

Debug Mode

17.1 Intro

By default, `libstdc++` is built with efficiency in mind, and therefore performs little or no error checking that is not required by the C++ standard. This means that programs that incorrectly use the C++ standard library will exhibit behavior that is not portable and may not even be predictable, because they tread into implementation-specific or undefined behavior. To detect some of these errors before they can become problematic, `libstdc++` offers a debug mode that provides additional checking of library facilities, and will report errors in the use of `libstdc++` as soon as they can be detected by emitting a description of the problem to standard error and aborting the program. This debug mode is available with GCC 3.4.0 and later versions.

The `libstdc++` debug mode performs checking for many areas of the C++ standard, but the focus is on checking interactions among standard iterators, containers, and algorithms, including:

- *Safe iterators*: Iterators keep track of the container whose elements they reference, so errors such as incrementing a past-the-end iterator or dereferencing an iterator that points to a container that has been destructed are diagnosed immediately.
- *Algorithm preconditions*: Algorithms attempt to validate their input parameters to detect errors as early as possible. For instance, the `set_intersection` algorithm requires that its iterator parameters `first1` and `last1` form a valid iterator range, and that the sequence `[first1, last1)` is sorted according to the same predicate that was passed to `set_intersection`; the `libstdc++` debug mode will detect an error if the sequence is not sorted or was sorted by a different predicate.

17.2 Semantics

A program that uses the C++ standard library correctly will maintain the same semantics under debug mode as it had with the normal (release) library. All functional and exception-handling guarantees made by the normal library also hold for the debug mode library, with one exception: performance guarantees made by the normal library may not hold in the debug mode library. For instance, erasing an element in a `std::list` is a constant-time operation in normal library, but in debug mode it is linear in the number of iterators that reference that particular list. So while your (correct) program won't change its results, it is likely to execute more slowly.

`libstdc++` includes many extensions to the C++ standard library. In some cases the extensions are obvious, such as the hashed associative containers, whereas other extensions give predictable results to behavior that would otherwise be undefined, such as throwing an exception when a `std::basic_string` is constructed from a NULL character pointer. This latter category also includes implementation-defined and unspecified semantics, such as the growth rate of a vector. Use of these extensions is not considered incorrect, so code that relies on them will not be rejected by debug mode. However, use of these extensions may affect the portability of code to other implementations of the C++ standard library, and is therefore somewhat hazardous. For this reason, the `libstdc++` debug mode offers a "pedantic" mode (similar to GCC's `-pedantic` compiler flag) that attempts to emulate the semantics guaranteed by the C++ standard. For instance, constructing a `std::basic_string` with a NULL character pointer would result in an exception under normal mode or non-pedantic debug mode (this is a `libstdc++` extension),

whereas under pedantic debug mode `libstdc++` would signal an error. To enable the pedantic debug mode, compile your program with both `-D_GLIBCXX_DEBUG` and `-D_GLIBCXX_DEBUG_PEDANTIC`. (N.B. In GCC 3.4.x and 4.0.0, due to a bug, `-D_GLIBCXX_DEBUG_PEDANTIC` was also needed. The problem has been fixed in GCC 4.0.1 and later versions.)

The following library components provide extra debugging capabilities in debug mode:

- `std::array` (no safe iterators)
- `std::basic_string` (no safe iterators and see note below)
- `std::bitset`
- `std::deque`
- `std::list`
- `std::map`
- `std::multimap`
- `std::multiset`
- `std::set`
- `std::vector`
- `std::unordered_map`
- `std::unordered_multimap`
- `std::unordered_set`
- `std::unordered_multiset`

N.B. although there are precondition checks for some string operations, e.g. `operator[]`, they will not always be run when using the `char` and `wchar_t` specializations (`std::string` and `std::wstring`). This is because `libstdc++` uses GCC's `extern template` extension to provide explicit instantiations of `std::string` and `std::wstring`, and those explicit instantiations don't include the debug-mode checks. If the containing functions are inlined then the checks will run, so compiling with `-O1` might be enough to enable them. Alternatively `-D_GLIBCXX_EXTERN_TEMPLATE=0` will suppress the declarations of the explicit instantiations and cause the functions to be instantiated with the debug-mode checks included, but this is unsupported and not guaranteed to work. For full debug-mode support you can use the `__gnu_debug::basic_string` debugging container directly, which always works correctly.

17.3 Using

17.3.1 Using the Debug Mode

To use the `libstdc++` debug mode, compile your application with the compiler flag `-D_GLIBCXX_DEBUG`. Note that this flag changes the sizes and behavior of standard class templates such as `std::vector`, and therefore you can only link code compiled with debug mode and code compiled without debug mode if no instantiation of a container is passed between the two translation units.

By default, error messages are formatted to fit on lines of about 78 characters. The environment variable `GLIBCXX_DEBUG_MESSAGE_LENGTH` can be used to request a different length.

Note that `libstdc++` is able to produce backtraces on error. To enable these, compile with `-D_GLIBCXX_DEBUG_BACKTRACE` and then link with `-lstdc++exp`. These backtraces are not supported on all platforms.

operation is correct (aborting with a diagnostic if an error is found) and will then forward to the underlying release-mode container that will perform the actual work. This design decision ensures that we cannot regress release-mode performance (because the release-mode containers are left untouched) and partially enables **mixing debug and release code** at link time, although that will not be discussed at this time.

Two types of wrappers are used in the implementation of the debug mode: container wrappers and iterator wrappers. The two types of wrappers interact to maintain relationships between iterators and their associated containers, which are necessary to detect certain types of standard library usage errors such as dereferencing past-the-end iterators or inserting into a container using an iterator from a different container.

17.4.2.1.1 Safe Iterators

Iterator wrappers provide a debugging layer over any iterator that is attached to a particular container, and will manage the information detailing the iterator's state (singular, dereferenceable, etc.) and tracking the container to which the iterator is attached. Because iterators have a well-defined, common interface the iterator wrapper is implemented with the iterator adaptor class template `__gnu_debug::_Safe_iterator`, which takes two template parameters:

- **Iterator:** The underlying iterator type, which must be either the `iterator` or `const_iterator` typedef from the sequence type this iterator can reference.
- **Sequence:** The type of sequence that this iterator references. This sequence must be a safe sequence (discussed below) whose `iterator` or `const_iterator` typedef is the type of the safe iterator.

17.4.2.1.2 Safe Sequences (Containers)

Container wrappers provide a debugging layer over a particular container type. Because containers vary greatly in the member functions they support and the semantics of those member functions (especially in the area of iterator invalidation), container wrappers are tailored to the container they reference, e.g., the debugging version of `std::list` duplicates the entire interface of `std::list`, adding additional semantic checks and then forwarding operations to the real `std::list` (a public base class of the debugging version) as appropriate. However, all safe containers inherit from the class template `__gnu_debug::_Safe_sequence`, instantiated with the type of the safe container itself (an instance of the curiously recurring template pattern).

The iterators of a container wrapper will be **safe iterators** that reference sequences of this type and wrap the iterators provided by the release-mode base class. The debugging container will use only the safe iterators within its own interface (therefore requiring the user to use safe iterators, although this does not change correct user code) and will communicate with the release-mode base class with only the underlying, unsafe, release-mode iterators that the base class exports.

The debugging version of `std::list` will have the following basic structure:

```
template<typename _Tp, typename _Allocator = allocator<_Tp>
class debug-list :
    public release-list<_Tp, _Allocator>,
    public __gnu_debug::_Safe_sequence<debug-list<_Tp, _Allocator> >
{
    typedef release-list<_Tp, _Allocator> _Base;
    typedef debug-list<_Tp, _Allocator> _Self;

public:
    typedef __gnu_debug::_Safe_iterator<typename _Base::iterator, _Self> iterator;
    typedef __gnu_debug::_Safe_iterator<typename _Base::const_iterator, _Self> ←
        const_iterator;

    // duplicate std::list interface with debugging semantics
};
```



```

{
    template<typename _Tp, typename _Alloc = allocator<_Tp> >
        class list
        {
        // ...
        };
    } // namespace __gnu_norm

    namespace __debug
    {
        template<typename _Tp, typename _Alloc = allocator<_Tp> >
            class list
            : public __cxx1998::list<_Tp, _Alloc>,
            public __gnu_debug::_Safe_sequence<list<_Tp, _Alloc> >
            {
            // ...
            };
        } // namespace __cxx1998

        inline namespace __debug { }
    }

```

17.4.2.3.2 Link- and run-time coexistence of release- and debug-mode components

Because each component has a distinct and separate release and debug implementation, there is no issue with link-time coexistence: the separate namespaces result in different mangled names, and thus unique linkage.

However, components that are defined and used within the C++ standard library itself face additional constraints. For instance, some of the member functions of `std::moneypunct` return `std::basic_string`. Normally, this is not a problem, but with a mixed mode standard library that could be using either debug-mode or release-mode `basic_string` objects, things get more complicated. As the return value of a function is not encoded into the mangled name, there is no way to specify a release-mode or a debug-mode string. In practice, this results in runtime errors. A simplified example of this problem is as follows.

Take this translation unit, compiled in debug-mode:

```

// -D_GLIBCXX_DEBUG
#include <string>

std::string test02();

std::string test01()
{
    return test02();
}

int main()
{
    test01();
    return 0;
}

```

... and linked to this translation unit, compiled in release mode:

```

#include <string>

std::string
test02()
{
    return std::string("toast");
}

```


Other options may exist for implementing the debug mode, many of which have probably been considered and others that may still be lurking. This list may be expanded over time to include other options that we could have implemented, but in all cases the full ramifications of the approach (as measured against the design goals for a libstdc++ debug mode) should be considered first. The DejaGNU testsuite includes some testcases that check for known problems with some solutions (e.g., the `using` declaration solution that breaks user specialization), and additional testcases will be added as we are able to identify other typical problem cases. These test cases will serve as a benchmark by which we can compare debug mode implementations.

17.4.3 Other Implementations

There are several existing implementations of debug modes for C++ standard library implementations, although none of them directly supports debugging for programs using libstdc++. The existing implementations include:

- **SafeSTL**: SafeSTL was the original debugging version of the Standard Template Library (STL), implemented by Cay S. Horstmann on top of the Hewlett-Packard STL. Though it inspired much work in this area, it has not been kept up-to-date for use with modern compilers or C++ standard library implementations.
- **STLport**: STLport is a free implementation of the C++ standard library derived from the **SGI implementation**, and ported to many other platforms. It includes a debug mode that uses a wrapper model (that in some ways inspired the libstdc++ debug mode design), although at the time of this writing the debug mode is somewhat incomplete and meets only the "Full user recompilation" (2) recompilation guarantee by requiring the user to link against a different library in debug mode vs. release mode.
- **Metrowerks CodeWarrior**: The C++ standard library that ships with Metrowerks CodeWarrior includes a debug mode. It is a full debug-mode implementation (including debugging for CodeWarrior extensions) and is easy to use, although it meets only the "Full recompilation" (1) recompilation guarantee.

Chapter 18

Parallel Mode

The `libstdc++` parallel mode is an experimental parallel implementation of many algorithms of the C++ Standard Library.

Several of the standard algorithms, for instance `std::sort`, are made parallel using OpenMP annotations. These parallel mode constructs can be invoked by explicit source declaration or by compiling existing sources with a specific compiler flag.

Note

The parallel mode has not been kept up to date with recent C++ standards and so it only conforms to the C++03 requirements. That means that move-only predicates may not work with parallel mode algorithms, and for C++20 most of the algorithms cannot be used in `constexpr` functions.

For C++17 and above there are new overloads of the standard algorithms which take an execution policy argument. You should consider using those instead of the non-standard parallel mode extensions.

18.1 Intro

The following library components in the include `numeric` are included in the parallel mode:

- `std::accumulate`
- `std::adjacent_difference`
- `std::inner_product`
- `std::partial_sum`

The following library components in the include `algorithm` are included in the parallel mode:

- `std::adjacent_find`
 - `std::count`
 - `std::count_if`
 - `std::equal`
 - `std::find`
 - `std::find_if`
 - `std::find_first_of`
 - `std::for_each`
-

- `std::generate`
- `std::generate_n`
- `std::lexicographical_compare`
- `std::mismatch`
- `std::search`
- `std::search_n`
- `std::transform`
- `std::replace`
- `std::replace_if`
- `std::max_element`
- `std::merge`
- `std::min_element`
- `std::nth_element`
- `std::partial_sort`
- `std::partition`
- `std::random_shuffle`
- `std::set_union`
- `std::set_intersection`
- `std::set_symmetric_difference`
- `std::set_difference`
- `std::sort`
- `std::stable_sort`
- `std::unique_copy`

18.2 Semantics

The parallel mode STL algorithms are currently not exception-safe, i.e. user-defined functors must not throw exceptions. Also, the order of execution is not guaranteed for some functions, of course. Therefore, user-defined functors should not have any concurrent side effects.

Since the current GCC OpenMP implementation does not support OpenMP parallel regions in concurrent threads, it is not possible to call parallel STL algorithm in concurrent threads, either. It might work with other compilers, though.

18.3 Using

18.3.1 Prerequisite Compiler Flags

Any use of parallel functionality requires additional compiler and runtime support, in particular support for OpenMP. Adding this support is not difficult: just compile your application with the compiler flag `-fopenmp`. This will link in `libgomp`, the [GNU Offloading and Multi Processing Runtime Library](#), whose presence is mandatory.

In addition, hardware that supports atomic operations and a compiler capable of producing atomic operations is mandatory: GCC defaults to no support for atomic operations on some common hardware architectures. Activating atomic operations may require explicit compiler flags on some targets (like `sparc` and `x86`), such as `-march=i686`, `-march=native` or `-mcpu=v9`. See the GCC manual for more information.

18.3.2 Using Parallel Mode

To use the `libstdc++` parallel mode, compile your application with the prerequisite flags as detailed above, and in addition add `-D_GLIBCXX_PARALLEL`. This will convert all use of the standard (sequential) algorithms to the appropriate parallel equivalents. Please note that this doesn't necessarily mean that everything will end up being executed in a parallel manner, but rather that the heuristics and settings coded into the parallel versions will be used to determine if all, some, or no algorithms will be executed using parallel variants.

Note that the `_GLIBCXX_PARALLEL` define may change the sizes and behavior of standard class templates such as `std::search`, and therefore one can only link code compiled with parallel mode and code compiled without parallel mode if no instantiation of a container is passed between the two translation units. Parallel mode functionality has distinct linkage, and cannot be confused with normal mode symbols.

18.3.3 Using Specific Parallel Components

When it is not feasible to recompile your entire application, or only specific algorithms need to be parallel-aware, individual parallel algorithms can be made available explicitly. These parallel algorithms are functionally equivalent to the standard drop-in algorithms used in parallel mode, but they are available in a separate namespace as GNU extensions and may be used in programs compiled with either release mode or with parallel mode.

An example of using a parallel version of `std::sort`, but no other parallel algorithms, is:

```
#include <vector>
#include <parallel/algorithm>

int main()
{
    std::vector<int> v(100);

    // ...

    // Explicitly force a call to parallel sort.
    __gnu_parallel::sort(v.begin(), v.end());
    return 0;
}
```

Then compile this code with the prerequisite compiler flags (`-fopenmp` and any necessary architecture-specific flags for atomic operations.)

The following table provides the names and headers of all the parallel algorithms that can be used in a similar manner:

18.4 Design

18.4.1 Interface Basics

All parallel algorithms are intended to have signatures that are equivalent to the ISO C++ algorithms replaced. For instance, the `std::adjacent_find` function is declared as:

```
namespace std
{
    template<typename _FIter>
        _FIter
        adjacent_find(_FIter, _FIter);
}
```

Which means that there should be something equivalent for the parallel version. Indeed, this is the case:

```
namespace std
{
    namespace __parallel
    {
        template<typename _FIter>
            _FIter
            adjacent_find(_FIter, _FIter);

        ...
    }
}
```

But.... why the ellipses?

The ellipses in the example above represent additional overloads required for the parallel version of the function. These additional overloads are used to dispatch calls from the ISO C++ function signature to the appropriate parallel function (or sequential function, if no parallel functions are deemed worthy), based on either compile-time or run-time conditions.

The available signature options are specific for the different algorithms/algorithm classes.

The general view of overloads for the parallel algorithms look like this:

- ISO C++ signature
- ISO C++ signature + `sequential_tag` argument
- ISO C++ signature + algorithm-specific tag type (several signatures)

Please note that the implementation may use additional functions (designated with the `__switch` suffix) to dispatch from the ISO C++ signature to the correct parallel version. Also, some of the algorithms do not have support for run-time conditions, so the last overload is therefore missing.

18.4.2 Configuration and Tuning

18.4.2.1 Setting up the OpenMP Environment

Several aspects of the overall runtime environment can be manipulated by standard OpenMP function calls.

To specify the number of threads to be used for the algorithms globally, use the function `omp_set_num_threads`. An example:


```
make check-parallel
```

The log and summary files for conformance testing are in the `testsuite/parallel` directory.

To run the performance tests with the parallel mode active,

```
make check-performance-parallel
```

The result file for performance testing are in the `testsuite` directory, in the file `libstdc++_performance.sum`. In addition, the policy-based containers have their own visualizations, which have additional software dependencies than the usual bare-boned text file, and can be generated by using the `make doc-performance` rule in the `testsuite`'s Makefile.

18.6 Bibliography

- [53] Johannes SinglerLeonor Frias, Copyright © 2007 , Workshop on Highly Parallel Processing on a Chip (HPPC) 2007. (LNCS) .
- [54] Johannes SinglerPeter SandersFelix Putze, Copyright © 2007 , Euro-Par 2007: Parallel Processing. (LNCS 4641) .

Chapter 19

The `mt_allocator`

19.1 Intro

The `mt_allocator` [hereinafter referred to simply as "the allocator"] is a fixed size (power of two) allocator that was initially developed specifically to suit the needs of multi-threaded applications [hereinafter referred to as an MT application]. Over time the allocator has evolved and been improved in many ways, in particular it now also does a good job in single-threaded applications [hereinafter referred to as an ST application]. (Note: In this document, when referring to single-threaded applications this also includes applications that are compiled with `gcc` without thread support enabled. This is accomplished using `ifdef`'s on `__GTHREADS`). This allocator is tunable, very flexible, and capable of high-performance.

The aim of this document is to describe - from an application point of view - the "inner workings" of the allocator.

19.2 Design Issues

19.2.1 Overview

There are three general components to the allocator: a datum describing the characteristics of the memory pool, a policy class containing this pool that links instantiation types to common or individual pools, and a class inheriting from the policy class that is the actual allocator.

The datum describing pools characteristics is

```
template<bool __Thread>
class __pool
```

This class is parametrized on thread support, and is explicitly specialized for both multiple threads (with `bool==true`) and single threads (via `bool==false`.) It is possible to use a custom pool datum instead of the default class that is provided.

There are two distinct policy classes, each of which can be used with either type of underlying pool datum.

```
template<bool __Thread>
struct __common_pool_policy

template<typename _Tp, bool __Thread>
struct __per_type_pool_policy
```

The first policy, `__common_pool_policy`, implements a common pool. This means that allocators that are instantiated with different types, say `char` and `long` will both use the same pool. This is the default policy.

The second policy, `__per_type_pool_policy`, implements a separate pool for each instantiating type. Thus, `char` and `long` will use separate pools. This allows per-type tuning, for instance.

Putting this all together, the actual allocator class is

```
template<typename _Tp, typename _Poolp = __default_policy>
class __mt_alloc : public __mt_alloc_base<_Tp>, _Poolp
```

This class has the interface required for standard library allocator classes, namely member functions `allocate` and `deallocate`, plus others.

19.3 Implementation

19.3.1 Tunable Parameters

Certain allocation parameters can be modified, or tuned. There exists a nested struct `__pool_base::_Tune` that contains all these parameters, which include settings for

- Alignment
- Maximum bytes before calling `::operator new` directly
- Minimum bytes
- Size of underlying global allocations
- Maximum number of supported threads
- Migration of deallocations to the global free list
- Shunt for global `new` and `delete`

Adjusting parameters for a given instance of an allocator can only happen before any allocations take place, when the allocator itself is initialized. For instance:

```
#include <ext/mt_allocator.h>

struct pod
{
    int i;
    int j;
};

int main()
{
    typedef pod value_type;
    typedef __gnu_cxx::__mt_alloc<value_type> allocator_type;
    typedef __gnu_cxx::__pool_base::_Tune tune_type;

    tune_type t_default;
    tune_type t_opt(16, 5120, 32, 5120, 20, 10, false);
    tune_type t_single(16, 5120, 32, 5120, 1, 10, false);

    tune_type t;
    t = allocator_type::_M_get_options();
    allocator_type::_M_set_options(t_opt);
    t = allocator_type::_M_get_options();

    allocator_type a;
    allocator_type::pointer p1 = a.allocate(128);
    allocator_type::pointer p2 = a.allocate(5128);

    a.deallocate(p1, 128);
    a.deallocate(p2, 5128);
```


systems will reclaim allocated memory at program termination anyway. If sidestepping this kind of noise is desired, there are three options: use an allocator, like `new_allocator` that releases memory while debugging, use `GLIBCXX_FORCE_NEW` to bypass the allocator's internal pools, or use a custom pool datum that releases resources on destruction.

On systems with the function `__cxa_atexit`, the allocator can be forced to free all memory allocated before program termination with the member function `__pool_type::_M_destroy`. However, because this member function relies on the precise and exactly-conforming ordering of static destructors, including those of a static local `__pool` object, it should not be used, ever, on systems that don't have the necessary underlying support. In addition, in practice, forcing deallocation can be tricky, as it requires the `__pool` object to be fully-constructed before the object that uses it is fully constructed. For most (but not all) STL containers, this works, as an instance of the allocator is constructed as part of a container's constructor. However, this assumption is implementation-specific, and subject to change. For an example of a pool that frees memory, see the [ext/mt_allocator/deallocate_local-6.cc](#) example.

19.4 Single Thread Example

Let's start by describing how the data on a freelist is laid out in memory. This is the first two blocks in freelist for thread id 3 in bin 3 (8 bytes):

```
+-----+
| next*  -----|--+  (_S_bin[ 3 ].first[ 3 ] points here)
|           |   |
|           |   |
+-----+-----+
| thread_id = 3 |   |
|           |   |
|           |   |
+-----+-----+
| DATA        |   | (A pointer to here is what is returned to the
|           |   | the application when needed)
|           |   |
|           |   |
|           |   |
+-----+-----+
| next*        |<--+ (If next == NULL it's the last one on the list)
|           |   |
|           |   |
+-----+-----+
| thread_id = 3 |   |
|           |   |
|           |   |
+-----+-----+
| DATA        |   |
|           |   |
|           |   |
|           |   |
+-----+-----+
```


The reason that the number of blocks moved to the current threads freelist is limited to `block_count` is to minimize the chance that a subsequent `deallocate()` call will return the excess blocks to the global freelist (based on the `_S_freelist_headroom` calculation, see below).

However if there isn't any memory on the global pool we need to get memory from the system - this is done in exactly the same way as in a single threaded application with one major difference; the list built in the newly allocated memory (of `_S_chunk_size` size) is added to the current threads freelist instead of to the global.

The basic process of a deallocation call is simple: always add the block to the front of the current threads freelist and update the counters and pointers (as described earlier with the specific check of ownership that causes the used counter of the thread that originally allocated the block to be decreased instead of the current threads counter).

And here comes the free and used counters to service. Each time a `deallocate()` call is made, the length of the current threads freelist is compared to the amount memory in use by this thread.

Let's go back to the example of an application that has one thread that does all the allocations and one that deallocates. Both these threads use say 516 32-byte blocks that was allocated during thread creation for example. Their used counters will both say 516 at this point. The allocation thread now grabs 1000 32-byte blocks and puts them in a shared container. The used counter for this thread is now 1516.

The deallocation thread now deallocates 500 of these blocks. For each deallocation made the used counter of the allocating thread is decreased and the freelist of the deallocation thread gets longer and longer. But the calculation made in `deallocate()` will limit the length of the freelist in the deallocation thread to `_S_freelist_headroom %` of it's used counter. In this case, when the freelist (given that the `_S_freelist_headroom` is at it's default value of 10%) exceeds 52 (516/10) blocks will be returned to the global pool where the allocating thread may pick them up and reuse them.

In order to reduce lock contention (since this requires this bins mutex to be locked) this operation is also made in chunks of blocks (just like when chunks of blocks are moved from the global freelist to a threads freelist mentioned above). The "formula" used can probably be improved to further reduce the risk of blocks being "bounced back and forth" between freelists.

Chapter 20

The bitmap_allocator

20.1 Design

As this name suggests, this allocator uses a bit-map to keep track of the used and unused memory locations for its book-keeping purposes.

This allocator will make use of 1 single bit to keep track of whether it has been allocated or not. A bit 1 indicates free, while 0 indicates allocated. This has been done so that you can easily check a collection of bits for a free block. This kind of Bitmapped strategy works best for single object allocations, and with the STL type parameterized allocators, we do not need to choose any size for the block which will be represented by a single bit. This will be the size of the parameter around which the allocator has been parameterized. Thus, close to optimal performance will result. Hence, this should be used for node based containers which call the allocate function with an argument of 1.

The bitmapped allocator's internal pool is exponentially growing. Meaning that internally, the blocks acquired from the Free List Store will double every time the bitmapped allocator runs out of memory.

The macro `__GTHREADS` decides whether to use Mutex Protection around every allocation/deallocation. The state of the macro is picked up automatically from the `gthr` abstraction layer.

20.2 Implementation

20.2.1 Free List Store

The Free List Store (referred to as FLS for the remaining part of this document) is the Global memory pool that is shared by all instances of the bitmapped allocator instantiated for any type. This maintains a sorted order of all free memory blocks given back to it by the bitmapped allocator, and is also responsible for giving memory to the bitmapped allocator when it asks for more.

Internally, there is a Free List threshold which indicates the Maximum number of free lists that the FLS can hold internally (cache). Currently, this value is set at 64. So, if there are more than 64 free lists coming in, then some of them will be given back to the OS using operator delete so that at any given time the Free List's size does not exceed 64 entries. This is done because a Binary Search is used to locate an entry in a free list when a request for memory comes along. Thus, the run-time complexity of the search would go up given an increasing size, for 64 entries however, $\lg(64) == 6$ comparisons are enough to locate the correct free list if it exists.

Suppose the free list size has reached its threshold, then the largest block from among those in the list and the new block will be selected and given back to the OS. This is done because it reduces external fragmentation, and allows the OS to use the larger blocks later in an orderly fashion, possibly merging them later. Also, on some systems, large blocks are obtained via calls to `mmap`, so giving them back to free system resources becomes most important.

The function `_S_should_i_give` decides the policy that determines whether the current block of memory should be given to the allocator for the request that it has made. That's because we may not always have exact fits for the memory size that the allocator requests. We do this mainly to prevent external fragmentation at the cost of a little internal fragmentation. Now, the value of this

20.2.8 Locality

Another issue would be whether to keep the all bitmaps in a separate area in memory, or to keep them near the actual blocks that will be given out or allocated for the client. After some testing, I've decided to keep these bitmaps close to the actual blocks. This will help in 2 ways.

1. Constant time access for the bitmap themselves, since no kind of look up will be needed to find the correct bitmap list or its equivalent.
2. And also this would preserve the cache as far as possible.

So in effect, this kind of an allocator might prove beneficial from a purely cache point of view. But this allocator has been made to try and roll out the defects of the `node_allocator`, wherein the nodes get skewed about in memory, if they are not returned in the exact reverse order or in the same order in which they were allocated. Also, the `new_allocator`'s book keeping overhead is too much for small objects and single object allocations, though it preserves the locality of blocks very well when they are returned back to the allocator.

20.2.9 Overhead and Grow Policy

Expected overhead per block would be 1 bit in memory. Also, once the address of the free list has been found, the cost for allocation/deallocation would be negligible, and is supposed to be constant time. For these very reasons, it is very important to minimize the linear time costs, which include finding a free list with a free block while allocating, and finding the corresponding free list for a block while deallocating. Therefore, I have decided that the growth of the internal pool for this allocator will be exponential as compared to linear for `node_allocator`. There, linear time works well, because we are mainly concerned with speed of allocation/deallocation and memory consumption, whereas here, the allocation/deallocation part does have some linear/logarithmic complexity components in it. Thus, to try and minimize them would be a good thing to do at the cost of a little bit of memory.

Another thing to be noted is the pool size will double every time the internal pool gets exhausted, and all the free blocks have been given away. The initial size of the pool would be `sizeof(size_t) x 8` which is the number of bits in an integer, which can fit exactly in a CPU register. Hence, the term given is exponential growth of the internal pool.

Chapter 21

Policy-Based Data Structures

21.1 Intro

This is a library of policy-based elementary data structures: associative containers and priority queues. It is designed for high-performance, flexibility, semantic safety, and conformance to the corresponding containers in `std` and `std::tr1` (except for some points where it differs by design).

21.1.1 Performance Issues

An attempt is made to categorize the wide variety of possible container designs in terms of performance-impacting factors. These performance factors are translated into design policies and incorporated into container design.

There is tension between unravelling factors into a coherent set of policies. Every attempt is made to make a minimal set of factors. However, in many cases multiple factors make for long template names. Every attempt is made to alias and use typedefs in the source files, but the generated names for external symbols can be large for binary files or debuggers.

In many cases, the longer names allow capabilities and behaviours controlled by macros to also be unambiguously emitted as distinct generated names.

Specific issues found while unraveling performance factors in the design of associative containers and priority queues follow.

21.1.1.1 Associative

Associative containers depend on their composite policies to a very large extent. Implicitly hard-wiring policies can hamper their performance and limit their functionality. An efficient hash-based container, for example, requires policies for testing key equivalence, hashing keys, translating hash values into positions within the hash table, and determining when and how to resize the table internally. A tree-based container can efficiently support order statistics, i.e. the ability to query what is the order of each key within the sequence of keys in the container, but only if the container is supplied with a policy to internally update meta-data. There are many other such examples.

Ideally, all associative containers would share the same interface. Unfortunately, underlying data structures and mapping semantics differentiate between different containers. For example, suppose one writes a generic function manipulating an associative container.

```
template<typename Cntnr>
void
some_op_sequence(Cntnr& r_cnt)
{
    ...
}
```

Given this, then what can one assume about the instantiating container? The answer varies according to its underlying data structure. If the underlying data structure of `Cntnr` is based on a tree or trie, then the order of elements is well defined; otherwise, it is not, in general. If the underlying data structure of `Cntnr` is based on a collision-chaining hash table, then modifying `r_Cntnr` will not invalidate its iterators' order; if the underlying data structure is a probing hash table, then this is not the case. If the underlying data structure is based on a tree or trie, then a reference to the container can efficiently be split; otherwise, it cannot, in general. If the underlying data structure is a red-black tree, then splitting a reference to the container is exception-free; if it is an ordered-vector tree, exceptions can be thrown.

21.1.1.2 Priority Que

Priority queues are useful when one needs to efficiently access a minimum (or maximum) value as the set of values changes.

Most useful data structures for priority queues have a relatively simple structure, as they are geared toward relatively simple requirements. Unfortunately, these structures do not support access to an arbitrary value, which turns out to be necessary in many algorithms. Say, decreasing an arbitrary value in a graph algorithm. Therefore, some extra mechanism is necessary and must be invented for accessing arbitrary values. There are at least two alternatives: embedding an associative container in a priority queue, or allowing cross-referencing through iterators. The first solution adds significant overhead; the second solution requires a precise definition of iterator invalidation. Which is the next point...

Priority queues, like hash-based containers, store values in an order that is meaningless and undefined externally. For example, a `push` operation can internally reorganize the values. Because of this characteristic, describing a priority queues' iterator is difficult: on one hand, the values to which iterators point can remain valid, but on the other, the logical order of iterators can change unpredictably.

Roughly speaking, any element that is both inserted to a priority queue (e.g. through `push`) and removed from it (e.g., through `pop`), incurs a logarithmic overhead (in the amortized sense). Different underlying data structures place the actual cost differently: some are optimized for amortized complexity, whereas others guarantee that specific operations only have a constant cost. One underlying data structure might be chosen if modifying a value is frequent (Dijkstra's shortest-path algorithm), whereas a different one might be chosen otherwise. Unfortunately, an array-based binary heap - an underlying data structure that optimizes (in the amortized sense) `push` and `pop` operations, differs from the others in terms of its invalidation guarantees. Other design decisions also impact the cost and placement of the overhead, at the expense of more difference in the kinds of operations that the underlying data structure can support. These differences pose a challenge when creating a uniform interface for priority queues.

21.1.2 Goals

Many fine associative-container libraries were already written, most notably, the C++ standard's associative containers. Why then write another library? This section shows some possible advantages of this library, when considering the challenges in the introduction. Many of these points stem from the fact that the ISO C++ process introduced associative-containers in a two-step process (first standardizing tree-based containers, only then adding hash-based containers, which are fundamentally different), did not standardize priority queues as containers, and (in our opinion) overloads the iterator concept.

21.1.2.1 Associative

21.1.2.1.1 Policy Choices

Associative containers require a relatively large number of policies to function efficiently in various settings. In some cases this is needed for making their common operations more efficient, and in other cases this allows them to support a larger set of operations

1. Hash-based containers, for example, support look-up and insertion methods (`find` and `insert`). In order to locate elements quickly, they are supplied a hash functor, which instruct how to transform a key object into some size type; a hash functor might transform "hello" into 1123002298. A hash table, though, requires transforming each key object into some size-type type in some specific domain; a hash table with a 128-long table might transform "hello" into position 63. The policy by which the hash value is transformed into a position within the table can dramatically affect performance. Hash-based containers also do not resize naturally (as opposed to tree-based containers, for example). The appropriate resize policy is unfortunately intertwined with the policy that transforms hash value into a position within the table.

2. Tree-based containers, for example, also support look-up and insertion methods, and are primarily useful when maintaining order between elements is important. In some cases, though, one can utilize their balancing algorithms for completely different purposes.

Figure A shows a tree whose each node contains two entries: a floating-point key, and some size-type *metadata* (in bold beneath it) that is the number of nodes in the sub-tree. (The root has key 0.99, and has 5 nodes (including itself) in its sub-tree.) A container based on this data structure can obviously answer efficiently whether 0.3 is in the container object, but it can also answer what is the order of 0.3 among all those in the container object: see [66].

As another example, Figure B shows a tree whose each node contains two entries: a half-open geometric line interval, and a number *metadata* (in bold beneath it) that is the largest endpoint of all intervals in its sub-tree. (The root describes the interval $[20, 36)$, and the largest endpoint in its sub-tree is 99.) A container based on this data structure can obviously answer efficiently whether $[3, 41)$ is in the container object, but it can also answer efficiently whether the container object has intervals that intersect $[3, 41)$. These types of queries are very useful in geometric algorithms and lease-management algorithms.

It is important to note, however, that as the trees are modified, their internal structure changes. To maintain these invariants, one must supply some policy that is aware of these changes. Without this, it would be better to use a linked list (in itself very efficient for these purposes).



Figure 21.1: Node Invariants

21.1.2.1.2 Underlying Data Structures

The standard C++ library contains associative containers based on red-black trees and collision-chaining hash tables. These are very useful, but they are not ideal for all types of settings.

The figure below shows the different underlying data structures currently supported in this library.



Figure 21.2: Underlying Associative Data Structures

A shows a collision-chaining hash-table, B shows a probing hash-table, C shows a red-black tree, D shows a splay tree, E shows a tree based on an ordered vector (implicit in the order of the elements), F shows a PATRICIA trie, and G shows a list-based container with update policies.

Each of these data structures has some performance benefits, in terms of speed, size or both. For now, note that vector-based trees and probing hash tables manipulate memory more efficiently than red-black trees and collision-chaining hash tables, and that list-based associative containers are very useful for constructing "multimaps".

Now consider a function manipulating a generic associative container,

```
template<class Cntnr>
int
some_op_sequence(Cntnr &r_cnt)
{
    ...
}
```

Ideally, the underlying data structure of `Cntnr` would not affect what can be done with `r_cnt`. Unfortunately, this is not the case.

For example, if `Cntnr` is `std::map`, then the function can use

```
std::for_each(r_cnt.find(foo), r_cnt.find(bar), foobar)
```

in order to apply `foobar` to all elements between `foo` and `bar`. If `Cntnr` is a hash-based container, then this call's results are undefined.

Also, if `Cntnr` is tree-based, the type and object of the comparison functor can be accessed. If `Cntnr` is hash based, these queries are nonsensical.

There are various other differences based on the container's underlying data structure. For one, they can be constructed by, and queried for, different policies. Furthermore:

1. Containers based on C, D, E and F store elements in a meaningful order; the others store elements in a meaningless (and probably time-varying) order. By implication, only containers based on C, D, E and F can support `erase` operations taking an iterator and returning an iterator to the following element without performance loss.
2. Containers based on C, D, E, and F can be split and joined efficiently, while the others cannot. Containers based on C and D, furthermore, can guarantee that this is exception-free; containers based on E cannot guarantee this.
3. Containers based on all but E can guarantee that erasing an element is exception free; containers based on E cannot guarantee this. Containers based on all but B and E can guarantee that modifying an object of their type does not invalidate iterators or references to their elements, while containers based on B and E cannot. Containers based on C, D, and E can furthermore make a stronger guarantee, namely that modifying an object of their type does not affect the order of iterators.

A unified tag and traits system (as used for the C++ standard library iterators, for example) can ease generic manipulation of associative containers based on different underlying data structures.

21.1.2.1.3 Iterators

Iterators are centric to the design of the standard library containers, because of the container/algorithm/iterator decomposition that allows an algorithm to operate on a range through iterators of some sequence. Iterators, then, are useful because they allow going over a specific *sequence*. The standard library also uses iterators for accessing a specific *element*: when an associative container returns one through `find`. The standard library consistently uses the same types of iterators for both purposes: going over a range, and accessing a specific found element. Before the introduction of hash-based containers to the standard library, this made sense (with the exception of priority queues, which are discussed later).

Using the standard associative containers together with non-order-preserving associative containers (and also because of priority-queues container), there is a possible need for different types of iterators for self-organizing containers: the iterator concept seems overloaded to mean two different things (in some cases).

21.1.2.1.3.1 Using Point Iterators for Range Operations

Suppose `cntnr` is some associative container, and say `c` is an object of type `cntnr`. Then what will be the outcome of

```
std::for_each(c.find(1), c.find(5), foo);
```

If `cntnr` is a tree-based container object, then an in-order walk will apply `foo` to the relevant elements, as in the graphic below, label A. If `c` is a hash-based container, then the order of elements between any two elements is undefined (and probably time-varying); there is no guarantee that the elements traversed will coincide with the *logical* elements between 1 and 5, as in label B.



Figure 21.3: Range Iteration in Different Data Structures

In our opinion, this problem is not caused just because red-black trees are order preserving while collision-chaining hash tables are (generally) not - it is more fundamental. Most of the standard's containers order sequences in a well-defined manner that is determined by their *interface*: calling `insert` on a tree-based container modifies its sequence in a predictable way, as does calling `push_back` on a list or a vector. Conversely, collision-chaining hash tables, probing hash tables, priority queues, and list-based containers (which are very useful for "multimaps") are self-organizing data structures; the effect of each operation modifies their sequences in a manner that is (practically) determined by their *implementation*.

Consequently, applying an algorithm to a sequence obtained from most containers may or may not make sense, but applying it to a sub-sequence of a self-organizing container does not.

21.1.2.1.3.2 Cost to Point Iterators to Enable Range Operations

Suppose `c` is some collision-chaining hash-based container object, and one calls

```
c.find(3)
```

Then what composes the returned iterator?

In the graphic below, label A shows the simplest (and most efficient) implementation of a collision-chaining hash table. The little box marked `point_iterator` shows an object that contains a pointer to the element's node. Note that this "iterator" has no way to move to the next element (it cannot support `operator++`). Conversely, the little box marked `iterator` stores both a pointer to the element, as well as some other information (the bucket number of the element). the second iterator, then, is "heavier" than the first one- it requires more time and space. If we were to use a different container to cross-reference into this hash-table using these iterators - it would take much more space. As noted above, nothing much can be done by incrementing these iterators, so why is this extra information needed?

Alternatively, one might create a collision-chaining hash-table where the lists might be linked, forming a monolithic total-element list, as in the graphic below, label B. Here the iterators are as light as can be, but the hash-table's operations are more complicated.



Figure 21.4: Point Iteration in Hash Data Structures

It should be noted that containers based on collision-chaining hash-tables are not the only ones with this type of behavior; many other self-organizing data structures display it as well.

21.1.2.1.3.3 Invalidation Guarantees

Consider the following snippet:

```
it = c.find(3);
c.erase(5);
```

Following the call to `erase`, what is the validity of `it`: can it be de-referenced? can it be incremented?

The answer depends on the underlying data structure of the container. The graphic below shows three cases: A1 and A2 show a red-black tree; B1 and B2 show a probing hash-table; C1 and C2 show a collision-chaining hash table.



Figure 21.5: Effect of erase in different underlying data structures

1. Erasing 5 from A1 yields A2. Clearly, an iterator to 3 can be de-referenced and incremented. The sequence of iterators changed, but in a way that is well-defined by the interface.

- Erasing 5 from B1 yields B2. Clearly, an iterator to 3 is not valid at all - it cannot be de-referenced or incremented; the order of iterators changed in a way that is (practically) determined by the implementation and not by the interface.
- Erasing 5 from C1 yields C2. Here the situation is more complicated. On the one hand, there is no problem in de-referencing `it`. On the other hand, the order of iterators changed in a way that is (practically) determined by the implementation and not by the interface.

So in the standard library containers, it is not always possible to express whether `it` is valid or not. This is true also for `insert`. Again, the iterator concept seems overloaded.

21.1.2.1.4 Functional

The design of the functional overlay to the underlying data structures differs slightly from some of the conventions used in the C++ standard. A strict public interface of methods that comprise only operations which depend on the class's internal structure; other operations are best designed as external functions. (See [84]). With this rubric, the standard associative containers lack some useful methods, and provide other methods which would be better removed.

21.1.2.1.4.1 `erase`

- Order-preserving standard associative containers provide the method

```
iterator
erase(iterator it)
```

which takes an iterator, erases the corresponding element, and returns an iterator to the following element. Also standard hash-based associative containers provide this method. This seemingly increases genericity between associative containers, since it is possible to use

```
typename C::iterator it = c.begin();
typename C::iterator e_it = c.end();

while(it != e_it)
    it = pred(*it)? c.erase(it) : ++it;
```

in order to erase from a container object `c` all element which match a predicate `pred`. However, in a different sense this actually decreases genericity: an integral implication of this method is that tree-based associative containers' memory use is linear in the total number of elements they store, while hash-based containers' memory use is unbounded in the total number of elements they store. Assume a hash-based container is allowed to decrease its size when an element is erased. Then the elements might be rehashed, which means that there is no "next" element - it is simply undefined. Consequently, it is possible to infer from the fact that the standard library's hash-based containers provide this method that they cannot downsize when elements are erased. As a consequence, different code is needed to manipulate different containers, assuming that memory should be conserved. Therefore, this library's non-order preserving associative containers omit this method.

- All associative containers include a conditional-erase method

```
template<
class Pred>
size_type
erase_if
(Pred pred)
```

which erases all elements matching a predicate. This is probably the only way to ensure linear-time multiple-item erase which can actually downsize a container.

- The standard associative containers provide methods for multiple-item erase of the form

```
size_type
erase(It b, It e)
```

erasing a range of elements given by a pair of iterators. For tree-based or trie-based containers, this can be implemented more efficiently as a (small) sequence of split and join operations. For other, unordered, containers, this method isn't much better than an external loop. Moreover, if `c` is a hash-based container, then

```
c.erase(c.find(2), c.find(5))
```

is almost certain to do something different than erasing all elements whose keys are between 2 and 5, and is likely to produce other undefined behavior.

21.1.2.1.4.2 `split` and `join`

It is well-known that tree-based and trie-based container objects can be efficiently split or joined (See [66]). Externally splitting or joining trees is super-linear, and, furthermore, can throw exceptions. Split and join methods, consequently, seem good choices for tree-based container methods, especially, since as noted just before, they are efficient replacements for erasing sub-sequences.

21.1.2.1.4.3 `insert`

The standard associative containers provide methods of the form

```
template<class It>
size_type
insert(It b, It e);
```

for inserting a range of elements given by a pair of iterators. At best, this can be implemented as an external loop, or, even more efficiently, as a join operation (for the case of tree-based or trie-based containers). Moreover, these methods seem similar to constructors taking a range given by a pair of iterators; the constructors, however, are transactional, whereas the insert methods are not; this is possibly confusing.

21.1.2.1.4.4 `operator==` and `operator<=`

Associative containers are parametrized by policies allowing to test key equivalence: a hash-based container can do this through its equivalence functor, and a tree-based container can do this through its comparison functor. In addition, some standard associative containers have global function operators, like `operator==` and `operator<=`, that allow comparing entire associative containers.

In our opinion, these functions are better left out. To begin with, they do not significantly improve over an external loop. More importantly, however, they are possibly misleading - `operator==`, for example, usually checks for equivalence, or interchangeability, but the associative container cannot check for values' equivalence, only keys' equivalence; also, are two containers considered equivalent if they store the same values in different order? this is an arbitrary decision.

21.1.2.2 Priority Queues

21.1.2.2.1 Policy Choices

Priority queues are containers that allow efficiently inserting values and accessing the maximal value (in the sense of the container's comparison functor). Their interface supports `push` and `pop`. The standard container `std::priority_queue` indeed support these methods, but little else. For algorithmic and software-engineering purposes, other methods are needed:

1. Many graph algorithms (see [66]) require increasing a value in a priority queue (again, in the sense of the container's comparison functor), or joining two priority-queue objects.
2. The return type of `priority_queue`'s `push` method is a point-type iterator, which can be used for modifying or erasing arbitrary values. For example:

```
priority_queue<int> p;
priority_queue<int>::point_iterator it = p.push(3);
p.modify(it, 4);
```

These types of cross-referencing operations are necessary for making priority queues useful for different applications, especially graph applications.

3. It is sometimes necessary to erase an arbitrary value in a priority queue. For example, consider the `select` function for monitoring file descriptors:

```
int
select(int nfd, fd_set *readfds, fd_set *writefds, fd_set *errorfds,
       struct timeval *timeout);
```

then, as the `select` documentation states:

“The `nfd` argument specifies the range of file descriptors to be tested. The `select()` function tests file descriptors in the range of 0 to `nfd-1`.”

It stands to reason, therefore, that we might wish to maintain a minimal value for `nfd`, and priority queues immediately come to mind. Note, though, that when a socket is closed, the minimal file description might change; in the absence of an efficient means to erase an arbitrary value from a priority queue, we might as well avoid its use altogether.

The standard containers typically support iterators. It is somewhat unusual for `std::priority_queue` to omit them (See [83]). One might ask why do priority queues need to support iterators, since they are self-organizing containers with a different purpose than abstracting sequences. There are several reasons:

- (a) Iterators (even in self-organizing containers) are useful for many purposes: cross-referencing containers, serialization, and debugging code that uses these containers.
- (b) The standard library’s hash-based containers support iterators, even though they too are self-organizing containers with a different purpose than abstracting sequences.
- (c) In standard-library-like containers, it is natural to specify the interface of operations for modifying a value or erasing a value (discussed previously) in terms of iterators. It should be noted that the standard containers also use iterators for accessing and manipulating a specific value. In hash-based containers, one checks the existence of a key by comparing the iterator returned by `find` to the iterator returned by `end`, and not by comparing a pointer returned by `find` to `NULL`.

21.1.2.2.2 Underlying Data Structures

There are three main implementations of priority queues: the first employs a binary heap, typically one which uses a sequence; the second uses a tree (or forest of trees), which is typically less structured than an associative container’s tree; the third simply uses an associative container. These are shown in the figure below with labels A1 and A2, B, and C.



Figure 21.6: Underlying Priority Queue Data Structures

No single implementation can completely replace any of the others. Some have better `push` and `pop` amortized performance, some have better bounded (worst case) response time than others, some optimize a single method at the expense of others, etc. In general the "best" implementation is dictated by the specific problem.

As with associative containers, the more implementations co-exist, the more necessary a traits mechanism is for handling generic containers safely and efficiently. This is especially important for priority queues, since the invalidation guarantees of one of the most useful data structures - binary heaps - is markedly different than those of most of the others.

21.1.2.2.3 Binary Heaps

Binary heaps are one of the most useful underlying data structures for priority queues. They are very efficient in terms of memory (since they don't require per-value structure metadata), and have the best amortized `push` and `pop` performance for primitive types like `int`.

The standard library's `priority_queue` implements this data structure as an adapter over a sequence, typically `std::vector` or `std::deque`, which correspond to labels A1 and A2 respectively in the graphic above.

This is indeed an elegant example of the adapter concept and the algorithm/container/iterator decomposition. (See [89]). There are several reasons why a binary-heap priority queue may be better implemented as a container instead of a sequence adapter:

1. `std::priority_queue` cannot erase values from its adapted sequence (irrespective of the sequence type). This means that the memory use of an `std::priority_queue` object is always proportional to the maximal number of values it ever contained, and not to the number of values that it currently contains. (See `performance/priority_queue_text_pop_mem_usage.cc`.) This implementation of binary heaps acts very differently than other underlying data structures (See also pairing heaps).

2. Some combinations of adapted sequences and value types are very inefficient or just don't make sense. If one uses `std::priority_queue<std::vector<std::string>>`, for example, then not only will each operation perform a logarithmic number of `std::string` assignments, but, furthermore, any operation (including `pop`) can render the container useless due to exceptions. Conversely, if one uses `std::priority_queue<std::deque<int>>`, then each operation uses incurs a logarithmic number of indirect accesses (through pointers) unnecessarily. It might be better to let the container make a conservative deduction whether to use the structure in the graphic above, labels A1 or A2.
3. There does not seem to be a systematic way to determine what exactly can be done with the priority queue.
 - (a) If `p` is a priority queue adapting an `std::vector`, then it is possible to iterate over all values by using `&p.top()` and `&p.top() + p.size()`, but this will not work if `p` is adapting an `std::deque`; in any case, one cannot use `p.begin()` and `p.end()`. If a different sequence is adapted, it is even more difficult to determine what can be done.
 - (b) If `p` is a priority queue adapting an `std::deque`, then the reference return by


```
p.top()
```

 will remain valid until it is popped, but if `p` adapts an `std::vector`, the next push will invalidate it. If a different sequence is adapted, it is even more difficult to determine what can be done.
4. Sequence-based binary heaps can still implement linear-time `erase` and `modify` operations. This means that if one needs to erase a small (say logarithmic) number of values, then one might still choose this underlying data structure. Using `std::priority_queue`, however, this will generally change the order of growth of the entire sequence of operations.

21.2 Using

21.2.1 Prerequisites

The library contains only header files, and does not require any other libraries except the standard C++ library. All classes are defined in namespace `__gnu_pbds`. The library internally uses macros beginning with `PB_DS`, but `#undefs` anything it `#defines` (except for header guards). Compiling the library in an environment where macros beginning in `PB_DS` are defined, may yield unpredictable results in compilation, execution, or both.

Further dependencies are necessary to create the visual output for the performance tests. To create these graphs, an additional package is needed: **pychart**.

21.2.2 Organization

The various data structures are organized as follows.

- Branch-Based
 - `basic_branch` is an abstract base class for branched-based associative-containers
 - `tree` is a concrete base class for tree-based associative-containers
 - `trie` is a concrete base class trie-based associative-containers
- Hash-Based
 - `basic_hash_table` is an abstract base class for hash-based associative-containers
 - `cc_hash_table` is a concrete collision-chaining hash-based associative-containers
 - `gp_hash_table` is a concrete (general) probing hash-based associative-containers
- List-Based

- `list_update` list-based update-policy associative container
- Heap-Based
 - `priority_queue` A priority queue.

The hierarchy is composed naturally so that commonality is captured by base classes. Thus `operator[]` is defined at the base of any hierarchy, since all derived containers support it. Conversely `split` is defined in `basic_branch`, since only tree-like containers support it.

In addition, there are the following diagnostics classes, used to report errors specific to this library's data structures.



Figure 21.7: Exception Hierarchy

21.2.3 Tutorial

21.2.3.1 Basic Use

For the most part, the policy-based containers containers in namespace `__gnu_pbds` have the same interface as the equivalent containers in the standard C++ library, except for the names used for the container classes themselves. For example, this shows basic operations on a collision-chaining hash-based container:

```
#include <ext/pb_ds/assoc_container.h>

int main()
{
    __gnu_pbds::cc_hash_table<int, char> c;
    c[2] = 'b';
    assert(c.find(1) == c.end());
};
```

The container is called `__gnu_pbds::cc_hash_table` instead of `std::unordered_map`, since “unordered map” does not necessarily mean a hash-based map as implied by the C++ library (C++11 or TR1). For example, list-based associative containers, which are very useful for the construction of “multimaps,” are also unordered.

This snippet shows a red-black tree based container:

```
#include <ext/pb_ds/assoc_container.h>

int main()
{
    __gnu_pbds::tree<int, char> c;
    c[2] = 'b';
    assert(c.find(2) != c.end());
};
```

The container is called `tree` instead of `map` since the underlying data structures are being named with specificity.

The member function naming convention is to strive to be the same as the equivalent member functions in other C++ standard library containers. The familiar methods are unchanged: `begin`, `end`, `size`, `empty`, and `clear`.

This isn't to say that things are exactly as one would expect, given the container requirements and interfaces in the C++ standard. The names of containers' policies and policy accessors are different than the usual. For example, if `hash_type` is some type of hash-based container, then

```
hash_type::hash_fn
```

gives the type of its hash functor, and if `obj` is some hash-based container object, then

```
obj.get_hash_fn()
```

will return a reference to its hash-functor object.

Similarly, if `tree_type` is some type of tree-based container, then

```
tree_type::cmp_fn
```

gives the type of its comparison functor, and if `obj` is some tree-based container object, then

```
obj.get_cmp_fn()
```

will return a reference to its comparison-functor object.

It would be nice to give names consistent with those in the existing C++ standard (inclusive of TR1). Unfortunately, these standard containers don't consistently name types and methods. For example, `std::tr1::unordered_map` uses `hasher` for the hash functor, but `std::map` uses `key_compare` for the comparison functor. Also, we could not find an accessor for `std::tr1::unordered_map`'s hash functor, but `std::map` uses `compare` for accessing the comparison functor.

Instead, `__gnu_pbds` attempts to be internally consistent, and uses standard-derived terminology if possible.

Another source of difference is in scope: `__gnu_pbds` contains more types of associative containers than the standard C++ library, and more opportunities to configure these new containers, since different types of associative containers are useful in different settings.

Namespace `__gnu_pbds` contains different classes for hash-based containers, tree-based containers, trie-based containers, and list-based containers.

Since associative containers share parts of their interface, they are organized as a class hierarchy.

Each type or method is defined in the most-common ancestor in which it makes sense.

For example, all associative containers support iteration expressed in the following form:

```
const_iterator
begin() const;

iterator
begin();

const_iterator
end() const;

iterator
end();
```

But not all containers contain or use hash functors. Yet, both collision-chaining and (general) probing hash-based associative containers have a hash functor, so `basic_hash_table` contains the interface:

```
const hash_fn&
get_hash_fn() const;

hash_fn&
get_hash_fn();
```

so all hash-based associative containers inherit the same hash-functor accessor methods.

21.2.3.2 Configuring via Template Parameters

In general, each of this library's containers is parametrized by more policies than those of the standard library. For example, the standard hash-based container is parametrized as follows:

```
template<typename Key, typename Mapped, typename Hash,
        typename Pred, typename Allocator, bool Cache_Hashe_Code>
class unordered_map;
```

and so can be configured by key type, mapped type, a functor that translates keys to unsigned integral types, an equivalence predicate, an allocator, and an indicator whether to store hash values with each entry. this library's collision-chaining hash-based container is parametrized as

```
template<typename Key, typename Mapped, typename Hash_Fn,
        typename Eq_Fn, typename Comb_Hash_Fn,
        typename Resize_Policy, bool Store_Hash,
        typename Allocator>
class cc_hash_table;
```

and so can be configured by the first four types of `std::tr1::unordered_map`, then a policy for translating the key-hash result into a position within the table, then a policy by which the table resizes, an indicator whether to store hash values with each entry, and an allocator (which is typically the last template parameter in standard containers).

Nearly all policy parameters have default values, so this need not be considered for casual use. It is important to note, however, that hash-based containers' policies can dramatically alter their performance in different settings, and that tree-based containers' policies can make them useful for other purposes than just look-up.

As opposed to associative containers, priority queues have relatively few configuration options. The priority queue is parametrized as follows:

```
template<typename Value_Type, typename Cmp_Fn, typename Tag,
        typename Allocator>
class priority_queue;
```

The `Value_Type`, `Cmp_Fn`, and `Allocator` parameters are the container's value type, comparison-functor type, and allocator type, respectively; these are very similar to the standard's priority queue. The `Tag` parameter is different: there are a number of pre-defined tag types corresponding to binary heaps, binomial heaps, etc., and `Tag` should be instantiated by one of them.

Note that as opposed to the `std::priority_queue`, `__gnu_pbds::priority_queue` is not a sequence-adapter; it is a regular container.

21.2.3.3 Querying Container Attributes

A containers underlying data structure affect their performance; Unfortunately, they can also affect their interface. When manipulating generically associative containers, it is often useful to be able to statically determine what they can support and what the cannot.

Happily, the standard provides a good solution to a similar problem - that of the different behavior of iterators. If `It` is an iterator, then

```
typename std::iterator_traits<It>::iterator_category
```

is one of a small number of pre-defined tag classes, and

```
typename std::iterator_traits<It>::value_type
```

is the value type to which the iterator "points".

Similarly, in this library, if `C` is a container, then `container_traits` is a trait class that stores information about the kind of container that is implemented.

```
typename container_traits<C>::container_category
```

is one of a small number of predefined tag structures that uniquely identifies the type of underlying data structure.

In most cases, however, the exact underlying data structure is not really important, but what is important is one of its other attributes: whether it guarantees storing elements by key order, for example. For this one can use

```
typename container_traits<C>::order_preserving
```

Also,

```
typename container_traits<C>::invalidation_guarantee
```

is the container's invalidation guarantee. Invalidation guarantees are especially important regarding priority queues, since in this library's design, iterators are practically the only way to manipulate them.

21.2.3.4 Point and Range Iteration

This library differentiates between two types of methods and iterators: point-type, and range-type. For example, `find` and `insert` are point-type methods, since they each deal with a specific element; their returned iterators are point-type iterators. `begin` and `end` are range-type methods, since they are not used to find a specific element, but rather to go over all elements in a container object; their returned iterators are range-type iterators.

Most containers store elements in an order that is determined by their interface. Correspondingly, it is fine that their point-type iterators are synonymous with their range-type iterators. For example, in the following snippet

```
std::for_each(c.find(1), c.find(5), foo);
```

two point-type iterators (returned by `find`) are used for a range-type purpose - going over all elements whose key is between 1 and 5.

Conversely, the above snippet makes no sense for self-organizing containers - ones that order (and reorder) their elements by implementation. It would be nice to have a uniform iterator system that would allow the above snippet to compile only if it made sense.

This could trivially be done by specializing `std::for_each` for the case of iterators returned by `std::tr1::unordered_map`, but this would only solve the problem for one algorithm and one container. Fundamentally, the problem is that one can loop using a self-organizing container's point-type iterators.

This library's containers define two families of iterators: `point_const_iterator` and `point_iterator` are the iterator types returned by point-type methods; `const_iterator` and `iterator` are the iterator types returned by range-type methods.

```
class <- some container ->
{
public:
...

typedef <- something -> const_iterator;

typedef <- something -> iterator;

typedef <- something -> point_const_iterator;

typedef <- something -> point_iterator;

...

public:
...

const_iterator begin () const;

iterator begin();
```

```
point_const_iterator find(...) const;

point_iterator find(...);
};
```

For containers whose interface defines sequence order, it is very simple: point-type and range-type iterators are exactly the same, which means that the above snippet will compile if it is used for an order-preserving associative container.

For self-organizing containers, however, (hash-based containers as a special example), the preceding snippet will not compile, because their point-type iterators do not support `operator++`.

In any case, both for order-preserving and self-organizing containers, the following snippet will compile:

```
typename Cntnr::point_iterator it = c.find(2);
```

because a range-type iterator can always be converted to a point-type iterator.

Distinguishing between iterator types also raises the point that a container's iterators might have different invalidation rules concerning their de-referencing abilities and movement abilities. This now corresponds exactly to the question of whether point-type and range-type iterators are valid. As explained above, `container_traits` allows querying a container for its data structure attributes. The iterator-invalidation guarantees are certainly a property of the underlying data structure, and so

```
container_traits<C>::invalidation_guarantee
```

gives one of three pre-determined types that answer this query.

21.2.4 Examples

Additional code examples are provided in the source distribution, as part of the regression and performance testsuite.

21.2.4.1 Intermediate Use

- Basic use of maps: `basic_map.cc`
- Basic use of sets: `basic_set.cc`
- Conditionally erasing values from an associative container object: `erase_if.cc`
- Basic use of multimaps: `basic_multimap.cc`
- Basic use of multisets: `basic_multiset.cc`
- Basic use of priority queues: `basic_priority_queue.cc`
- Splitting and joining priority queues: `priority_queue_split_join.cc`
- Conditionally erasing values from a priority queue: `priority_queue_erase_if.cc`

21.2.4.2 Querying with `container_traits`

- Using `container_traits` to query about underlying data structure behavior: `assoc_container_traits.cc`
- A non-compiling example showing wrong use of finding keys in hash-based containers: `hash_find_neg.cc`
- Using `container_traits` to query about underlying data structure behavior: `priority_queue_container_traits.cc`

21.2.4.3 By Container Method

21.2.4.3.1 Hash-Based

21.2.4.3.1.1 size Related

- Setting the initial size of a hash-based container object: `hash_initial_size.cc`
- A non-compiling example showing how not to resize a hash-based container object: `hash_resize_neg.cc`
- Resizing the size of a hash-based container object: `hash_resize.cc`
- Showing an illegal resize of a hash-based container object: `hash_illegal_resize.cc`
- Changing the load factors of a hash-based container object: `hash_load_set_change.cc`

21.2.4.3.1.2 Hashing Function Related

- Using a modulo range-hashing function for the case of an unknown skewed key distribution: `hash_mod.cc`
- Writing a range-hashing functor for the case of a known skewed key distribution: `shift_mask.cc`
- Storing the hash value along with each key: `store_hash.cc`
- Writing a ranged-hash functor: `ranged_hash.cc`

21.2.4.3.2 Branch-Based

21.2.4.3.2.1 split or join Related

- Joining two tree-based container objects: `tree_join.cc`
- Splitting a PATRICIA trie container object: `trie_split.cc`
- Order statistics while joining two tree-based container objects: `tree_order_statistics_join.cc`

21.2.4.3.2.2 Node Invariants

- Using trees for order statistics: `tree_order_statistics.cc`
- Augmenting trees to support operations on line intervals: `tree_intervals.cc`

21.2.4.3.2.3 trie

- Using a PATRICIA trie for DNA strings: `trie_dna.cc`
- Using a PATRICIA trie for finding all entries whose key matches a given prefix: `trie_prefix_search.cc`

21.2.4.3.3 Priority Queues

- Cross referencing an associative container and a priority queue: `priority_queue_xref.cc`
 - Cross referencing a vector and a priority queue using a very simple version of Dijkstra's shortest path algorithm: `priority_queue_dijkstra.cc`
-

21.3 Design

21.3.1 Concepts

21.3.1.1 Null Policy Classes

Associative containers are typically parametrized by various policies. For example, a hash-based associative container is parametrized by a hash-functor, transforming each key into a non-negative numerical type. Each such value is then further mapped into a position within the table. The mapping of a key into a position within the table is therefore a two-step process.

In some cases, instantiations are redundant. For example, when the keys are integers, it is possible to use a redundant hash policy, which transforms each key into its value.

In some other cases, these policies are irrelevant. For example, a hash-based associative container might transform keys into positions within a table by a different method than the two-step method described above. In such a case, the hash functor is simply irrelevant.

When a policy is either redundant or irrelevant, it can be replaced by `null_type`.

For example, a `set` is an associative container with one of its template parameters (the one for the mapped type) replaced with `null_type`. Other places simplifications are made possible with this technique include node updates in tree and trie data structures, and hash and probe functions for hash data structures.

21.3.1.2 Map and Set Semantics

21.3.1.2.1 Distinguishing Between Maps and Sets

Anyone familiar with the standard knows that there are four kinds of associative containers: maps, sets, multimaps, and multisets. The map datatype associates each key to some data.

Sets are associative containers that simply store keys - they do not map them to anything. In the standard, each map class has a corresponding set class. E.g., `std::map<int, char>` maps each `int` to a `char`, but `std::set<int, char>` simply stores `ints`. In this library, however, there are no distinct classes for maps and sets. Instead, an associative container's Mapped template parameter is a policy: if it is instantiated by `null_type`, then it is a "set"; otherwise, it is a "map". E.g.,

```
cc_hash_table<int, char>
```

is a "map" mapping each `int` value to a `char`, but

```
cc_hash_table<int, null_type>
```

is a type that uniquely stores `int` values.

Once the Mapped template parameter is instantiated by `null_type`, then the "set" acts very similarly to the standard's sets - it does not map each key to a distinct `null_type` object. Also, , the container's `value_type` is essentially its `key_type` - just as with the standard's sets .

The standard's multimaps and multisets allow, respectively, non-uniquely mapping keys and non-uniquely storing keys. As discussed, the reasons why this might be necessary are 1) that a key might be decomposed into a primary key and a secondary key, 2) that a key might appear more than once, or 3) any arbitrary combination of 1)s and 2)s. Correspondingly, one should use 1) "maps" mapping primary keys to secondary keys, 2) "maps" mapping keys to size types, or 3) any arbitrary combination of 1)s and 2)s. Thus, for example, an `std::multiset<int>` might be used to store multiple instances of integers, but using this library's containers, one might use

```
tree<int, size_t>
```

i.e., a map of `ints` to `size_ts`.

These "multimaps" and "multisets" might be confusing to anyone familiar with the standard's `std::multimap` and `std::multiset` because there is no clear correspondence between the two. For example, in some cases where one uses `std::multiset` in the standard, one might use in this library a "multimap" of "multisets" - i.e., a container that maps primary keys each to an associative container that maps each secondary key to the number of times it occurs.

When one uses a "multimap," one should choose with care the type of container used for secondary keys.

21.3.1.2.2 Alternatives to `std::multiset` and `std::multimap`

Brace oneself: this library does not contain containers like `std::multimap` or `std::multiset`. Instead, these data structures can be synthesized via manipulation of the Mapped template parameter.

One maps the unique part of a key - the primary key, into an associative-container of the (originally) non-unique parts of the key - the secondary key. A primary associative-container is an associative container of primary keys; a secondary associative-container is an associative container of secondary keys.

Stepping back a bit, and starting in from the beginning.

Maps (or sets) allow mapping (or storing) unique-key values. The standard library also supplies associative containers which map (or store) multiple values with equivalent keys: `std::multimap`, `std::multiset`, `std::tr1::unordered_multimap`, and `unordered_multiset`. We first discuss how these might be used, then why we think it is best to avoid them.

Suppose one builds a simple bank-account application that records for each client (identified by an `std::string`) and account-id (marked by an unsigned long) - the balance in the account (described by a float). Suppose further that ordering this information is not useful, so a hash-based container is preferable to a tree based container. Then one can use

```
std::tr1::unordered_map<std::pair<std::string, unsigned long>, float, ...>
```

which hashes every combination of client and account-id. This might work well, except for the fact that it is now impossible to efficiently list all of the accounts of a specific client (this would practically require iterating over all entries). Instead, one can use

```
std::tr1::unordered_multimap<std::pair<std::string, unsigned long>, float, ...>
```

which hashes every client, and decides equivalence based on client only. This will ensure that all accounts belonging to a specific user are stored consecutively.

Also, suppose one wants an integers' priority queue (a container that supports `push`, `pop`, and `top` operations, the last of which returns the largest int) that also supports operations such as `find` and `lower_bound`. A reasonable solution is to build an adapter over `std::set<int>`. In this adapter, `push` will just call the tree-based associative container's `insert` method; `pop` will call its `end` method, and use it to return the preceding element (which must be the largest). Then this might work well, except that the container object cannot hold multiple instances of the same integer (`push(4)`, will be a no-op if 4 is already in the container object). If multiple keys are necessary, then one might build the adapter over an `std::multiset<int>`.

The standard library's non-unique-mapping containers are useful when (1) a key can be decomposed in to a primary key and a secondary key, (2) a key is needed multiple times, or (3) any combination of (1) and (2).

The graphic below shows how the standard library's container design works internally; in this figure nodes shaded equally represent equivalent-key values. Equivalent keys are stored consecutively using the properties of the underlying data structure: binary search trees (label A) store equivalent-key values consecutively (in the sense of an in-order walk) naturally; collision-chaining hash tables (label B) store equivalent-key values in the same bucket, the bucket can be arranged so that equivalent-key values are consecutive.



Figure 21.8: Non-unique Mapping Standard Containers

Put differently, the standards' non-unique mapping associative-containers are associative containers that map primary keys to linked lists that are embedded into the container. The graphic below shows again the two containers from the first graphic above, this time with the embedded linked lists of the grayed nodes marked explicitly.

Figure 21.9: Effect of embedded lists in `std::multimap`

These embedded linked lists have several disadvantages.

1. The underlying data structure embeds the linked lists according to its own consideration, which means that the search path for a value might include several different equivalent-key values. For example, the search path for the the black node in either of the first graphic, labels A or B, includes more than a single gray node.
2. The links of the linked lists are the underlying data structures' nodes, which typically are quite structured. In the case of tree-based containers (the graphic above, label B), each "link" is actually a node with three pointers (one to a parent and two to children), and a relatively-complicated iteration algorithm. The linked lists, therefore, can take up quite a lot of memory, and iterating over all values equal to a given key (through the return value of the standard library's `equal_range`) can be expensive.
3. The primary key is stored multiply; this uses more memory.
4. Finally, the interface of this design excludes several useful underlying data structures. Of all the unordered self-organizing data structures, practically only collision-chaining hash tables can (efficiently) guarantee that equivalent-key values are stored consecutively.

The above reasons hold even when the ratio of secondary keys to primary keys (or average number of identical keys) is small, but when it is large, there are more severe problems:

1. The underlying data structures order the links inside each embedded linked-lists according to their internal considerations, which effectively means that each of the links is unordered. Irrespective of the underlying data structure, searching for a specific value can degrade to linear complexity.
2. Similarly to the above point, it is impossible to apply to the secondary keys considerations that apply to primary keys. For example, it is not possible to maintain secondary keys by sorted order.
3. While the interface "understands" that all equivalent-key values constitute a distinct list (through `equal_range`), the underlying data structure typically does not. This means that operations such as erasing from a tree-based container all values whose keys are equivalent to a given key can be super-linear in the size of the tree; this is also true also for several other operations that target a specific list.

In this library, all associative containers map (or store) unique-key values. One can (1) map primary keys to secondary associative-containers (containers of secondary keys) or non-associative containers (2) map identical keys to a size-type representing the number of times they occur, or (3) any combination of (1) and (2). Instead of allowing multiple equivalent-key values, this library supplies associative containers based on underlying data structures that are suitable as secondary associative-containers.

In the figure below, labels A and B show the equivalent underlying data structures in this library, as mapped to the first graphic above. Labels A and B, respectively. Each shaded box represents some size-type or secondary associative-container.



Figure 21.10: Non-unique Mapping Containers

In the first example above, then, one would use an associative container mapping each user to an associative container which maps each application id to a start time (see `example/basic_multimap.cc`); in the second example, one would use an associative container mapping each `int` to some size-type indicating the number of times it logically occurs (see `example/basic_multiset.cc`).

See the discussion in list-based container types for containers especially suited as secondary associative-containers.

21.3.1.3 Iterator Semantics

21.3.1.3.1 Point and Range Iterators

Iterator concepts are bifurcated in this design, and are comprised of point-type and range-type iteration.

A point-type iterator is an iterator that refers to a specific element as returned through an associative-container's `find` method.

A range-type iterator is an iterator that is used to go over a sequence of elements, as returned by a container's `find` method.

A point-type method is a method that returns a point-type iterator; a range-type method is a method that returns a range-type iterator.

For most containers, these types are synonymous; for self-organizing containers, such as hash-based containers or priority queues, these are inherently different (in any implementation, including that of C++ standard library components), but in this design, it is made explicit. They are distinct types.

21.3.1.3.2 Distinguishing Point and Range Iterators

When using this library, is necessary to differentiate between two types of methods and iterators: point-type methods and iterators, and range-type methods and iterators. Each associative container's interface includes the methods:

```
point_const_iterator
find(const_key_reference r_key) const;

point_iterator
find(const_key_reference r_key);

std::pair<point_iterator, bool>
insert(const_reference r_val);
```

The relationship between these iterator types varies between container types. The figure below shows the most general invariant between point-type and range-type iterators: In *A* iterator, can always be converted to `point_iterator`. In *B* shows invariants for order-preserving containers: point-type iterators are synonymous with range-type iterators. Orthogonally, *C* shows invariants for "set" containers: iterators are synonymous with const iterators.



Figure 21.11: Point Iterator Hierarchy

Note that point-type iterators in self-organizing containers (hash-based associative containers) lack movement operators, such as `operator++` - in fact, this is the reason why this library differentiates from the standard C++ library's design on this point.

Typically, one can determine an iterator's movement capabilities using `std::iterator_traits<It>::iterator_category`, which is a `struct` indicating the iterator's movement capabilities. Unfortunately, none of the standard predefined categories reflect a pointer's *not* having any movement capabilities whatsoever. Consequently, `pb_ds` adds a type `trivial_iterator_tag` (whose name is taken from a concept in C++ standardese, which is the category of iterators with no movement capabilities.) All other standard C++ library tags, such as `forward_iterator_tag` retain their common use.

21.3.1.3.3 Invalidation Guarantees

If one manipulates a container object, then iterators previously obtained from it can be invalidated. In some cases a previously-obtained iterator cannot be de-referenced; in other cases, the iterator's next or previous element might have changed unpredictably. This corresponds exactly to the question whether a point-type or range-type iterator (see previous concept) is valid or not. In this design, one can query a container (in compile time) about its invalidation guarantees.

Given three different types of associative containers, a modifying operation (in that example, `erase`) invalidated iterators in three different ways: the iterator of one container remained completely valid - it could be de-referenced and incremented; the iterator of a different container could not even be de-referenced; the iterator of the third container could be de-referenced, but its "next" iterator changed unpredictably.

Distinguishing between `find` and `range` types allows fine-grained invalidation guarantees, because these questions correspond exactly to the question of whether point-type iterators and range-type iterators are valid. The graphic below shows tags corresponding to different types of invalidation guarantees.



Figure 21.12: Invalidation Guarantee Tags Hierarchy

- `basic_invalidation_guarantee` corresponds to a basic guarantee that a point-type iterator, a found pointer, or a found reference, remains valid as long as the container object is not modified.
- `point_invalidation_guarantee` corresponds to a guarantee that a point-type iterator, a found pointer, or a found reference, remains valid even if the container object is modified.
- `range_invalidation_guarantee` corresponds to a guarantee that a range-type iterator remains valid even if the container object is modified.

To find the invalidation guarantee of a container, one can use

```
typename container_traits<Cntnr>::invalidation_guarantee
```

Note that this hierarchy corresponds to the logic it represents: if a container has range-invalidation guarantees, then it must also have find invalidation guarantees; correspondingly, its invalidation guarantee (in this case `range_invalidation_guarantee`) can be cast to its base class (in this case `point_invalidation_guarantee`). This means that this hierarchy can be used easily using standard metaprogramming techniques, by specializing on the type of `invalidation_guarantee`.

These types of problems were addressed, in a more general setting, in [81] - Item 2. In our opinion, an invalidation-guarantee hierarchy would solve these problems in all container types - not just associative containers.

21.3.1.4 Genericity

The design attempts to address the following problem of data-structure genericity. When writing a function manipulating a generic container object, what is the behavior of the object? Suppose one writes

```
template<typename Cntnr>
void
some_op_sequence(Cntnr &r_container)
{
    ...
}
```

then one needs to address the following questions in the body of `some_op_sequence`:

- Which types and methods does `Cntnr` support? Containers based on hash tables can be queried for the hash-functor type and object; this is meaningless for tree-based containers. Containers based on trees can be split, joined, or can erase iterators and return the following iterator; this cannot be done by hash-based containers.
- What are the exception and invalidation guarantees of `Cntnr`? A container based on a probing hash-table invalidates all iterators when it is modified; this is not the case for containers based on node-based trees. Containers based on a node-based tree can be split or joined without exceptions; this is not the case for containers based on vector-based trees.
- How does the container maintain its elements? Tree-based and Trie-based containers store elements by key order; others, typically, do not. A container based on a splay trees or lists with update policies "cache" "frequently accessed" elements; containers based on most other underlying data structures do not.
- How does one query a container about characteristics and capabilities? What is the relationship between two different data structures, if anything?

The remainder of this section explains these issues in detail.

21.3.1.4.1 Tag

Tags are very useful for manipulating generic types. For example, if `It` is an iterator class, then `typename It::iterator_category` or `typename std::iterator_traits<It>::iterator_category` will yield its category, and `typename std::iterator_traits<It>::value_type` will yield its value type.

This library contains a container tag hierarchy corresponding to the diagram below.



Figure 21.13: Container Tag Hierarchy

Given any container `Cntnr`, the tag of the underlying data structure can be found via `typename Cntnr::container_category`.

21.3.1.4.2 Traits

Additionally, a traits mechanism can be used to query a container type for its attributes. Given any container `Cntnr`, then `<Cntnr>` is a traits class identifying the properties of the container.

To find if a container can throw when a key is erased (which is true for vector-based trees, for example), one can use

```
container_traits<Cntnr>::erase_can_throw
```

Some of the definitions in `container_traits` are dependent on other definitions. If `container_traits<Cntnr>::order_p` is true (which is the case for containers based on trees and tries), then the container can be split or joined; in this case, `container_traits<Cntnr>::split_join_can_throw` indicates whether splits or joins can throw exceptions (which is true for vector-based trees); otherwise `container_traits<Cntnr>::split_join_can_throw` will yield a compilation error. (This is somewhat similar to a compile-time version of the COM model).

21.3.2 By Container

21.3.2.1 hash

21.3.2.1.1 Interface

The collision-chaining hash-based container has the following declaration.

```
template<
    typename Key,
    typename Mapped,
    typename Hash_Fn = std::hash<Key>,
    typename Eq_Fn = std::equal_to<Key>,
    typename Comb_Hash_Fn = direct_mask_range_hashing<>
    typename Resize_Policy = default explained below.
    bool Store_Hash = false,
    typename Allocator = std::allocator<char> >
    class cc_hash_table;
```

The parameters have the following meaning:

1. `Key` is the key type.
2. `Mapped` is the mapped-policy.
3. `Hash_Fn` is a key hashing functor.
4. `Eq_Fn` is a key equivalence functor.
5. `Comb_Hash_Fn` is a range-hashing functor; it describes how to translate hash values into positions within the table.
6. `Resize_Policy` describes how a container object should change its internal size.
7. `Store_Hash` indicates whether the hash value should be stored with each entry.
8. `Allocator` is an allocator type.

The probing hash-based container has the following declaration.

```
template<
    typename Key,
    typename Mapped,
    typename Hash_Fn = std::hash<Key>,
    typename Eq_Fn = std::equal_to<Key>,
    typename Comb_Probe_Fn = direct_mask_range_hashing<>
    typename Probe_Fn = default explained below.
```

```
typename Resize_Policy = default explained below.  
bool Store_Hash = false,  
typename Allocator = std::allocator<char> >  
class gp_hash_table;
```

The parameters are identical to those of the collision-chaining container, except for the following.

1. `Comb_Probe_Fn` describes how to transform a probe sequence into a sequence of positions within the table.
2. `Probe_Fn` describes a probe sequence policy.

Some of the default template values depend on the values of other parameters, and are explained below.

21.3.2.1.2 Details

21.3.2.1.2.1 Hash Policies

21.3.2.1.2.2 General

Following is an explanation of some functions which hashing involves. The graphic below illustrates the discussion.



Figure 21.14: Hash functions, ranged-hash functions, and range-hashing functions

Let U be a domain (e.g., the integers, or the strings of 3 characters). A hash-table algorithm needs to map elements of U "uniformly" into the range $[0, \dots, m - 1]$ (where m is a non-negative integral value, and is, in general, time varying). I.e., the algorithm needs a ranged-hash function

$$f : U \times \mathbb{Z}_+ \rightarrow \mathbb{Z}_+$$

such that for any u in U ,

$$0 \leq f(u, m) \leq m - 1$$

and which has "good uniformity" properties (say [77].) One common solution is to use the composition of the hash function

$$h : U \rightarrow \mathbb{Z}_+,$$

which maps elements of U into the non-negative integers, and

$$g : \mathbb{Z}_+ \times \mathbb{Z}_+ \rightarrow \mathbb{Z}_+,$$

which maps a non-negative hash value, and a non-negative range upper-bound into a non-negative integral in the range between 0 (inclusive) and the range upper bound (exclusive), i.e., for any r in \mathbb{Z}_+ ,

$$0 \leq g(r, m) \leq m - 1$$

The resulting ranged-hash function, is

$$f(u, m) = g(h(u), m)$$

EQUATION 21.1: Ranged Hash Function

From the above, it is obvious that given g and h , f can always be composed (however the converse is not true). The standard's hash-based containers allow specifying a hash function, and use a hard-wired range-hashing function; the ranged-hash function is implicitly composed.

The above describes the case where a key is to be mapped into a single position within a hash table, e.g., in a collision-chaining table. In other cases, a key is to be mapped into a sequence of positions within a table, e.g., in a probing table. Similar terms apply in this case: the table requires a ranged probe function, mapping a key into a sequence of positions within the table. This is typically achieved by composing a hash function mapping the key into a non-negative integral type, a probe function transforming the hash value into a sequence of hash values, and a range-hashing function transforming the sequence of hash values into a sequence of positions.

21.3.2.1.2.3 Range Hashing

Some common choices for range-hashing functions are the division, multiplication, and middle-square methods ([77]), defined as

$$g(r, m) = r \bmod m$$

EQUATION 21.2: Range-Hashing, Division Method

$$g(r, m) = \lceil u/v (a r \bmod v) \rceil$$

and

$$g(r, m) = \lceil u/v (r^2 \bmod v) \rceil$$

respectively, for some positive integrals u and v (typically powers of 2), and some a . Each of these range-hashing functions works best for some different setting.

The division method (see above) is a very common choice. However, even this single method can be implemented in two very different ways. It is possible to implement using the low level `%` (modulo) operation (for any m), or the low level `&` (bit-mask) operation (for the case where m is a power of 2), i.e.,

$$g(r, m) = r \% m$$

EQUATION 21.3: Division via Prime Modulo

and

$$g(r, m) = r \& m - 1, \text{ (with } m = 2^k \text{ for some } k)$$

EQUATION 21.4: Division via Bit Mask

respectively.

The `%` (modulo) implementation has the advantage that for m a prime far from a power of 2, $g(r, m)$ is affected by all the bits of r (minimizing the chance of collision). It has the disadvantage of using the costly modulo operation. This method is hard-wired into SGI's implementation.

The `&` (bit-mask) implementation has the advantage of relying on the fast bit-wise and operation. It has the disadvantage that for $g(r, m)$ is affected only by the low order bits of r . This method is hard-wired into Dinkumware's implementation.

21.3.2.1.2.4 Ranged Hash

In cases it is beneficial to allow the client to directly specify a ranged-hash hash function. It is true, that the writer of the ranged-hash function cannot rely on the values of m having specific numerical properties suitable for hashing (in the sense used in [77]), since the values of m are determined by a resize policy with possibly orthogonal considerations.

There are two cases where a ranged-hash function can be superior. The first is when using perfect hashing: the second is when the values of m can be used to estimate the "general" number of distinct values required. This is described in the following.

Let

$$s = [s_0, \dots, s_{t-1}]$$

be a string of t characters, each of which is from domain S . Consider the following ranged-hash function:

$$f_1(s, m) = \sum_{i=0}^{t-1} s_i a^i \bmod m$$

EQUATION 21.5: A Standard String Hash Function

where a is some non-negative integral value. This is the standard string-hashing function used in SGI's implementation (with $a = 5$). Its advantage is that it takes into account all of the characters of the string.

Now assume that s is the string representation of a long DNA sequence (and so $S = \{'A', 'C', 'G', 'T'\}$). In this case, scanning the entire string might be prohibitively expensive. A possible alternative might be to use only the first k characters of the string, where

$$|S|^k \geq m,$$

i.e., using the hash function

$$f_2(s, m) = \sum_{i=0}^{k-1} s_i a^i \bmod m$$

EQUATION 21.6: Only k String DNA Hash

requiring scanning over only

$$k = \log_4(m)$$

characters.

Other more elaborate hash-functions might scan k characters starting at a random position (determined at each resize), or scanning k random positions (determined at each resize), i.e., using

$$f_3(s, m) = \sum_{i=r}^{r_0+k-1} s_i a^i \bmod m,$$

or

$$f_4(s, m) = \sum_{i=0}^{k-1} s_{r_i} a^{r_i} \bmod m,$$

respectively, for r_0, \dots, r_{k-1} each in the (inclusive) range $[0, \dots, t-1]$.

It should be noted that the above functions cannot be decomposed as per a ranged hash composed of hash and range hashing.

21.3.2.1.2.5 Implementation

This sub-subsection describes the implementation of the above in this library. It first explains range-hashing functions in collision-chaining tables, then ranged-hash functions in collision-chaining tables, then probing-based tables, and finally lists the relevant classes in this library.

21.3.2.1.2.6 Range-Hashing and Ranged-Hashes in Collision-Chaining Tables

`cc_hash_table` is parametrized by `Hash_Fn` and `Comb_Hash_Fn`, a hash functor and a combining hash functor, respectively.

In general, `Comb_Hash_Fn` is considered a range-hashing functor. `cc_hash_table` synthesizes a ranged-hash function from `Hash_Fn` and `Comb_Hash_Fn`. The figure below shows an `insert` sequence diagram for this case. The user inserts an element (point A), the container transforms the key into a non-negative integral using the hash functor (points B and C), and transforms the result into a position using the combining functor (points D and E).



Figure 21.15: Insert hash sequence diagram

If `cc_hash_table`'s hash-functor, `Hash_Fn` is instantiated by `null_type`, then `Comb_Hash_Fn` is taken to be a ranged-hash function. The graphic below shows an `insert` sequence diagram. The user inserts an element (point A), the container transforms the key into a position using the combining functor (points B and C).



Figure 21.16: Insert hash sequence diagram with a null policy

21.3.2.1.2.7 Probing tables

`gp_hash_table` is parametrized by `Hash_Fn`, `Probe_Fn`, and `Comb_Probe_Fn`. As before, if `Hash_Fn` and `Probe_Fn` are both `null_type`, then `Comb_Probe_Fn` is a ranged-probe functor. Otherwise, `Hash_Fn` is a hash functor, `Probe_Fn` is a functor for offsets from a hash value, and `Comb_Probe_Fn` transforms a probe sequence into a sequence of positions within the table.

21.3.2.1.2.8 Pre-Defined Policies

This library contains some pre-defined classes implementing range-hashing and probing functions:

1. `direct_mask_range_hashing` and `direct_mod_range_hashing` are range-hashing functions based on a bit-mask and a modulo operation, respectively.
2. `linear_probe_fn`, and `quadratic_probe_fn` are a linear probe and a quadratic probe function, respectively.

The graphic below shows the relationships.



Figure 21.17: Hash policy class diagram

21.3.2.1.2.9 Resize Policies

21.3.2.1.2.10 General

Hash-tables, as opposed to trees, do not naturally grow or shrink. It is necessary to specify policies to determine how and when a hash table should change its size. Usually, resize policies can be decomposed into orthogonal policies:

1. A size policy indicating how a hash table should grow (e.g., it should multiply by powers of 2).
2. A trigger policy indicating when a hash table should grow (e.g., a load factor is exceeded).

21.3.2.1.2.11 Size Policies

Size policies determine how a hash table changes size. These policies are simple, and there are relatively few sensible options. An exponential-size policy (with the initial size and growth factors both powers of 2) works well with a mask-based range-hashing function, and is the hard-wired policy used by Dinkumware. A prime-list based policy works well with a modulo-prime range hashing function and is the hard-wired policy used by SGI's implementation.

21.3.2.1.2.12 Trigger Policies

Trigger policies determine when a hash table changes size. Following is a description of two policies: load-check policies, and collision-check policies.

Load-check policies are straightforward. The user specifies two factors, A_{\min} and A_{\max} , and the hash table maintains the invariant that

$$A_{\min} \leq (\text{number of stored elements}) / (\text{hash-table size}) \leq A_{\max}$$

Collision-check policies work in the opposite direction of load-check policies. They focus on keeping the number of collisions moderate and hoping that the size of the table will not grow very large, instead of keeping a moderate load-factor and hoping that the number of collisions will be small. A maximal collision-check policy resizes when the longest probe-sequence grows too large.

Consider the graphic below. Let the size of the hash table be denoted by m , the length of a probe sequence be denoted by k , and some load factor be denoted by A . We would like to calculate the minimal length of k , such that if there were $A \cdot m$ elements in the hash table, a probe sequence of length k would be found with probability at most $1/m$.



Figure 21.18: Balls and bins

Denote the probability that a probe sequence of length k appears in bin i by p_i , the length of the probe sequence of bin i by l_i , and assume uniform distribution. Then

$$p_1 =$$

$$\text{EQUATION 21.7: Probability of Probe Sequence of Length } k$$

$$P(l_i \geq k) =$$

$$P(l_i \geq \alpha (1 + k / \alpha - 1)) \leq (a)$$

$$e^{-(\alpha (k / \alpha - 1)^2) / 2}$$

where (a) follows from the Chernoff bound ([85]). To calculate the probability that some bin contains a probe sequence greater than k , we note that the l_i are negatively-dependent ([67]). Let $I(\cdot)$ denote the indicator function. Then

$$P(\text{exists } i: l_i \geq k) =$$

EQUATION 21.8: Probability Probe Sequence in Some Bin

$$P(\sum_{i=1}^m I(l_i \geq k) \geq 1) =$$

$$P(\sum_{i=1}^m I(l_i \geq k) \geq m p_1 (1 + 1 / (m p_1) - 1)) \leq (a)$$

$$e^{-(m p_1 (1 / (m p_1) - 1)^2) / 2},$$

where (a) follows from the fact that the Chernoff bound can be applied to negatively-dependent variables ([67]). Inserting the first probability equation into the second one, and equating with $1/m$, we obtain

$$k \sim \sqrt{(2 \alpha \ln 2 m \ln(m))}.$$

21.3.2.1.2.13 Implementation

This sub-subsection describes the implementation of the above in this library. It first describes resize policies and their decomposition into trigger and size policies, then describes pre-defined classes, and finally discusses controlled access the policies' internals.

21.3.2.1.2.14 Decomposition

Each hash-based container is parametrized by a `Resize_Policy` parameter; the container derives publicly from `Resize_Policy`. For example:

```
cc_hash_table<typename Key,
typename Mapped,
...
typename Resize_Policy
...> : public Resize_Policy
```

As a container object is modified, it continuously notifies its `Resize_Policy` base of internal changes (e.g., collisions encountered and elements being inserted). It queries its `Resize_Policy` base whether it needs to be resized, and if so, to what size.

The graphic below shows a (possible) sequence diagram of an insert operation. The user inserts an element; the hash table notifies its resize policy that a search has started (point A); in this case, a single collision is encountered - the table notifies its resize policy of this (point B); the container finally notifies its resize policy that the search has ended (point C); it then queries its resize policy whether a resize is needed, and if so, what is the new size (points D to G); following the resize, it notifies the policy that a resize has completed (point H); finally, the element is inserted, and the policy notified (point I).



Figure 21.19: Insert resize sequence diagram

In practice, a resize policy can be usually orthogonally decomposed to a size policy and a trigger policy. Consequently, the library contains a single class for instantiating a resize policy: `hash_standard_resize_policy` is parametrized by `Size_Policy` and `Trigger_Policy`, derives publicly from both, and acts as a standard delegate ([71]) to these policies.

The two graphics immediately below show sequence diagrams illustrating the interaction between the standard resize policy and its trigger and size policies, respectively.



Figure 21.20: Standard resize policy trigger sequence diagram



Figure 21.21: Standard resize policy size sequence diagram

21.3.2.1.2.15 Predefined Policies

The library includes the following instantiations of size and trigger policies:

1. `hash_load_check_resize_trigger` implements a load check trigger policy.
2. `cc_hash_max_collision_check_resize_trigger` implements a collision check trigger policy.
3. `hash_exponential_size_policy` implements an exponential-size policy (which should be used with mask range hashing).
4. `hash_prime_size_policy` implementing a size policy based on a sequence of primes (which should be used with mod range hashing).

The graphic below gives an overall picture of the resize-related classes. `basic_hash_table` is parametrized by `Resize_Policy`, which it subclasses publicly. This class is currently instantiated only by `hash_standard_resize_policy`. `hash_standard_resize_policy` itself is parametrized by `Trigger_Policy` and `Size_Policy`. Currently, `Trigger_Policy` is instantiated by `hash_load_check_resize_trigger` or `cc_hash_max_collision_check_resize_trigger`; `Size_Policy` is instantiated by `hash_exponential_size_policy` or `hash_prime_size_policy`.

21.3.2.1.2.16 Controlling Access to Internals

There are cases where (controlled) access to resize policies' internals is beneficial. E.g., it is sometimes useful to query a hash-table for the table's actual size (as opposed to its `size()` - the number of values it currently holds); it is sometimes useful to set a table's initial size, externally resize it, or change load factors.

Clearly, supporting such methods both decreases the encapsulation of hash-based containers, and increases the diversity between different associative-containers' interfaces. Conversely, omitting such methods can decrease containers' flexibility.

In order to avoid, to the extent possible, the above conflict, the hash-based containers themselves do not address any of these questions; this is deferred to the resize policies, which are easier to change or replace. Thus, for example, neither `cc_hash_table` nor `gp_hash_table` contain methods for querying the actual size of the table; this is deferred to `hash_standard_resize_policy`.

Furthermore, the policies themselves are parametrized by template arguments that determine the methods they support ([56] shows techniques for doing so). `hash_standard_resize_policy` is parametrized by `External_Size_Access` that determines whether it supports methods for querying the actual size of the table or resizing it. `hash_load_check_resize_trigger` is parametrized by `External_Load_Access` that determines whether it supports methods for querying or modifying the loads. `cc_hash_max_collision_check_resize_trigger` is parametrized by `External_Load_Access` that determines whether it supports methods for querying the load.

Some operations, for example, resizing a container at run time, or changing the load factors of a load-check trigger policy, require the container itself to resize. As mentioned above, the hash-based containers themselves do not contain these types of methods, only their resize policies. Consequently, there must be some mechanism for a resize policy to manipulate the hash-based container. As the hash-based container is a subclass of the resize policy, this is done through virtual methods. Each hash-based container has a private virtual method:

```
virtual void
do_resize
(size_type new_size);
```

which resizes the container. Implementations of `Resize_Policy` can export public methods for resizing the container externally; these methods internally call `do_resize` to resize the table.

21.3.2.1.2.17 Policy Interactions

Hash-tables are unfortunately especially susceptible to choice of policies. One of the more complicated aspects of this is that poor combinations of good policies can form a poor container. Following are some considerations.

21.3.2.1.2.18 probe/size/trigger

Some combinations do not work well for probing containers. For example, combining a quadratic probe policy with an exponential size policy can yield a poor container: when an element is inserted, a trigger policy might decide that there is no need to resize, as the table still contains unused entries; the probe sequence, however, might never reach any of the unused entries.

Unfortunately, this library cannot detect such problems at compilation (they are halting reducible). It therefore defines an exception class `insert_error` to throw an exception in this case.

21.3.2.1.2.19 hash/trigger

Some trigger policies are especially susceptible to poor hash functions. Suppose, as an extreme case, that the hash function transforms each key to the same hash value. After some inserts, a collision detecting policy will always indicate that the container needs to grow.

The library, therefore, by design, limits each operation to one resize. For each `insert`, for example, it queries only once whether a resize is needed.

21.3.2.1.2.20 equivalence functors/storing hash values/hash

`cc_hash_table` and `gp_hash_table` are parametrized by an equivalence functor and by a `Store_Hash` parameter. If the latter parameter is `true`, then the container stores with each entry a hash value, and uses this value in case of collisions to determine whether to apply a hash value. This can lower the cost of collision for some types, but increase the cost of collisions for other types.

If a ranged-hash function or ranged probe function is directly supplied, however, then it makes no sense to store the hash value with each entry. This library's container will fail at compilation, by design, if this is attempted.

21.3.2.1.2.21 size/load-check trigger

Assume a size policy issues an increasing sequence of sizes a , $a q$, $a q^1$, $a q^2$, ... For example, an exponential size policy might issue the sequence of sizes 8, 16, 32, 64, ...

If a load-check trigger policy is used, with loads α_{\min} and α_{\max} , respectively, then it is a good idea to have:

1. $\alpha_{\max} \sim 1 / q$
2. $\alpha_{\min} < 1 / (2 q)$

This will ensure that the amortized hash cost of each modifying operation is at most approximately 3.

$\alpha_{\min} \sim \alpha_{\max}$ is, in any case, a bad choice, and $\alpha_{\min} > \alpha_{\max}$ is horrendous.

21.3.2.2 tree

21.3.2.2.1 Interface

The tree-based container has the following declaration:

```
template<
    typename Key,
    typename Mapped,
    typename Cmp_Fn = std::less<Key>,
    typename Tag = rb_tree_tag,
    template<
        typename Const_Node_Iterator,
        typename Node_Iterator,
        typename Cmp_Fn_,
        typename Allocator_>
        class Node_Update = null_node_update,
    typename Allocator = std::allocator<char> >
    class tree;
```

The parameters have the following meaning:

1. `Key` is the key type.
2. `Mapped` is the mapped-policy.
3. `Cmp_Fn` is a key comparison functor
4. `Tag` specifies which underlying data structure to use.
5. `Node_Update` is a policy for updating node invariants.
6. `Allocator` is an allocator type.

The `Tag` parameter specifies which underlying data structure to use. Instantiating it by `rb_tree_tag`, `splay_tree_tag`, or `ov_tree_tag`, specifies an underlying red-black tree, splay tree, or ordered-vector tree, respectively; any other tag is illegal. Note that containers based on the former two contain more types and methods than the latter (e.g., `reverse_iterator` and `rbegin`), and different exception and invalidation guarantees.

21.3.2.2.2 Details

21.3.2.2.2.1 Node Invariants

Consider the two trees in the graphic below, labels A and B. The first is a tree of floats; the second is a tree of pairs, each signifying a geometric line interval. Each element in a tree is referred to as a node of the tree. Of course, each of these trees can support the usual queries: the first can easily search for `0.4`; the second can easily search for `std::make_pair(10, 41)`.

Each of these trees can efficiently support other queries. The first can efficiently determine that the 2rd key in the tree is `0.3`; the second can efficiently determine whether any of its intervals overlaps

```
std::make_pair(29, 42)
```

(useful in geometric applications or distributed file systems with leases, for example). It should be noted that an `std::set` can only solve these types of problems with linear complexity.

In order to do so, each tree stores some metadata in each node, and maintains node invariants (see [66].) The first stores in each node the size of the sub-tree rooted at the node; the second stores at each node the maximal endpoint of the intervals at the sub-tree rooted at the node.



Figure 21.22: Tree node invariants

Supporting such trees is difficult for a number of reasons:

1. There must be a way to specify what a node's metadata should be (if any).
2. Various operations can invalidate node invariants. The graphic below shows how a right rotation, performed on A, results in B, with nodes x and y having corrupted invariants (the grayed nodes in C). The graphic shows how an insert, performed on D, results in E, with nodes x and y having corrupted invariants (the grayed nodes in F). It is not feasible to know outside the tree the effect of an operation on the nodes of the tree.
3. The search paths of standard associative containers are defined by comparisons between keys, and not through metadata.
4. It is not feasible to know in advance which methods trees can support. Besides the usual `find` method, the first tree can support a `find_by_order` method, while the second can support an `overlaps` method.



Figure 21.23: Tree node invalidation

These problems are solved by a combination of two means: node iterators, and template-template node updater parameters.

21.3.2.2.2 Node Iterators

Each tree-based container defines two additional iterator types, `const_node_iterator` and `node_iterator`. These iterators allow descending from a node to one of its children. Node iterator allow search paths different than those determined by the comparison functor. The `tree` supports the methods:

```
const_node_iterator
node_begin() const;

node_iterator
node_begin();

const_node_iterator
node_end() const;

node_iterator
node_end();
```

The first pairs return node iterators corresponding to the root node of the tree; the latter pair returns node iterators corresponding to a just-after-leaf node.

21.3.2.2.3 Node Updator

The tree-based containers are parametrized by a `Node_Update` template-template parameter. A tree-based container instantiates `Node_Update` to some `node_update` class, and publicly subclasses `node_update`. The graphic below shows this scheme, as well as some predefined policies (which are explained below).



Figure 21.24: A tree and its update policy

`node_update` (an instantiation of `Node_Update`) must define `metadata_type` as the type of metadata it requires. For order statistics, e.g., `metadata_type` might be `size_t`. The tree defines within each node a `metadata_type` object.

`node_update` must also define the following method for restoring node invariants:

```
void
operator()(node_iterator nd_it, const_node_iterator end_nd_it)
```

In this method, `nd_it` is a `node_iterator` corresponding to a node whose A) all descendants have valid invariants, and B) its own invariants might be violated; `end_nd_it` is a `const_node_iterator` corresponding to a just-after-leaf node. This method should correct the node invariants of the node pointed to by `nd_it`. For example, say node `x` in the graphic below label A has an invalid invariant, but its' children, `y` and `z` have valid invariants. After the invocation, all three nodes should have valid invariants, as in label B.



Figure 21.25: Restoring node invariants

When a tree operation might invalidate some node invariant, it invokes this method in its `node_update` base to restore the invariant. For example, the graphic below shows an `insert` operation (point A); the tree performs some operations, and calls the update functor three times (points B, C, and D). (It is well known that any `insert`, `erase`, `split` or `join`, can restore all node invariants by a small number of node invariant updates ([66]).



Figure 21.26: Insert update sequence

To complete the description of the scheme, three questions need to be answered:

1. How can a tree which supports order statistics define a method such as `find_by_order`?
2. How can the node updater base access methods of the tree?
3. How can the following cyclic dependency be resolved? `node_update` is a base class of the tree, yet it uses node iterators defined in the tree (its child).

The first two questions are answered by the fact that `node_update` (an instantiation of `Node_Update`) is a *public* base class of the tree. Consequently:

1. Any public methods of `node_update` are automatically methods of the tree ([56]). Thus an order-statistics node updater, `tree_order_statistics_node_update` defines the `find_by_order` method; any tree instantiated by this policy consequently supports this method as well.
2. In C++, if a base class declares a method as `virtual`, it is `virtual` in its subclasses. If `node_update` needs to access one of the tree's methods, say the member function `end`, it simply declares that method as `virtual abstract`.

The cyclic dependency is solved through template-template parameters. `Node_Update` is parametrized by the tree's node iterators, its comparison functor, and its allocator type. Thus, instantiations of `Node_Update` have all information required.

This library assumes that constructing a metadata object and modifying it are exception free. Suppose that during some method, say `insert`, a metadata-related operation (e.g., changing the value of a metadata) throws an exception. Ack! Rolling back the method is unusually complex.

Previously, a distinction was made between redundant policies and null policies. Node invariants show a case where null policies are required.

Assume a regular tree is required, one which need not support order statistics or interval overlap queries. Seemingly, in this case a redundant policy - a policy which doesn't affect nodes' contents would suffice. This, would lead to the following drawbacks:

1. Each node would carry a useless metadata object, wasting space.
2. The tree cannot know if its `Node_Update` policy actually modifies a node's metadata (this is halting reducible). In the graphic below, assume the shaded node is inserted. The tree would have to traverse the useless path shown to the root, applying redundant updates all the way.



Figure 21.27: Useless update path

A null policy class, `null_node_update` solves both these problems. The tree detects that node invariants are irrelevant, and defines all accordingly.

21.3.2.2.4 Split and Join

Tree-based containers support split and join methods. It is possible to split a tree so that it passes all nodes with keys larger than a given key to a different tree. These methods have the following advantages over the alternative of externally inserting to the destination tree and erasing from the source tree:

1. These methods are efficient - red-black trees are split and joined in poly-logarithmic complexity; ordered-vector trees are split and joined at linear complexity. The alternatives have super-linear complexity.
2. Aside from orders of growth, these operations perform few allocations and de-allocations. For red-black trees, allocations are not performed, and the methods are exception-free.

21.3.2.3 Trie

21.3.2.3.1 Interface

The trie-based container has the following declaration:

```
template<typename Key,
typename Mapped,
typename Cmp_Fn = std::less<Key>,
typename Tag = pat_trie_tag,
template<typename Const_Node_Iterator,
typename Node_Iterator,
typename E_Access_Traits_,
typename Allocator_>
class Node_Update = null_node_update,
typename Allocator = std::allocator<char> >
class trie;
```

The parameters have the following meaning:

1. `Key` is the key type.
2. `Mapped` is the mapped-policy.
3. `E_Access_Traits` is described in below.
4. `Tag` specifies which underlying data structure to use, and is described shortly.
5. `Node_Update` is a policy for updating node invariants. This is described below.
6. `Allocator` is an allocator type.

The `Tag` parameter specifies which underlying data structure to use. Instantiating it by `pat_trie_tag`, specifies an underlying PATRICIA trie (explained shortly); any other tag is currently illegal.

Following is a description of a (PATRICIA) trie (this implementation follows [90] and [69]).

A (PATRICIA) trie is similar to a tree, but with the following differences:

1. It explicitly views keys as a sequence of elements. E.g., a trie can view a string as a sequence of characters; a trie can view a number as a sequence of bits.
2. It is not (necessarily) binary. Each node has fan-out $n + 1$, where n is the number of distinct elements.
3. It stores values only at leaf nodes.
4. Internal nodes have the properties that A) each has at least two children, and B) each shares the same prefix with any of its descendant.

A (PATRICIA) trie has some useful properties:

1. It can be configured to use large node fan-out, giving it very efficient find performance (albeit at insertion complexity and size).
2. It works well for common-prefix keys.
3. It can support efficiently queries such as which keys match a certain prefix. This is sometimes useful in file systems and routers, and for "type-ahead" aka predictive text matching on mobile devices.

21.3.2.3.2 Details

21.3.2.3.2.1 Element Access Traits

A trie inherently views its keys as sequences of elements. For example, a trie can view a string as a sequence of characters. A trie needs to map each of n elements to a number in $\{0, n - 1\}$. For example, a trie can map a character c to

```
static_cast<size_t>(c)
```

.

Seemingly, then, a trie can assume that its keys support (const) iterators, and that the `value_type` of this iterator can be cast to a `size_t`. There are several reasons, though, to decouple the mechanism by which the trie accesses its keys' elements from the trie:

1. In some cases, the numerical value of an element is inappropriate. Consider a trie storing DNA strings. It is logical to use a trie with a fan-out of $5 = 1 + |\{'A', 'C', 'G', 'T'\}|$. This requires mapping 'T' to 3, though.
2. In some cases the keys' iterators are different than what is needed. For example, a trie can be used to search for common suffixes, by using strings' `reverse_iterator`. As another example, a trie mapping UNICODE strings would have a huge fan-out if each node would branch on a UNICODE character; instead, one can define an iterator iterating over 8-bit (or less) groups.

trie is, consequently, parametrized by `E_Access_Traits`-traits which instruct how to access sequences' elements. `string_trie` is a traits class for strings. Each such traits define some types, like:

```
typename E_Access_Traits::const_iterator
```

is a const iterator iterating over a key's elements. The traits class must also define methods for obtaining an iterator to the first and last element of a key.

The graphic below shows a (PATRICIA) trie resulting from inserting the words: "I wish that I could ever see a poem lovely as a trie" (which, unfortunately, does not rhyme).

The leaf nodes contain values; each internal node contains two `typename E_Access_Traits::const_iterator` objects, indicating the maximal common prefix of all keys in the sub-tree. For example, the shaded internal node roots a sub-tree with leafs "a" and "as". The maximal common prefix is "a". The internal node contains, consequently, two const iterators, one pointing to 'a', and the other to 's'.



Figure 21.28: A PATRICIA trie

21.3.2.3.2.2 Node Invariants

Trie-based containers support node invariants, as do tree-based containers. There are two minor differences, though, which, unfortunately, thwart sharing them sharing the same node-updating policies:

1. A trie's `Node_Update` template-template parameter is parametrized by `E_Access_Traits`, while a tree's `Node_Update` template-template parameter is parametrized by `Cmp_Fn`.
2. Tree-based containers store values in all nodes, while trie-based containers (at least in this implementation) store values in leafs.

The graphic below shows the scheme, as well as some predefined policies (which are explained below).



Figure 21.29: A trie and its update policy

This library offers the following pre-defined trie node updating policies:

1. `trie_order_statistics_node_update` supports order statistics.
2. `trie_prefix_search_node_update` supports searching for ranges that match a given prefix.
3. `null_node_update` is the null node updater.

21.3.2.3.2.3 Split and Join

Trie-based containers support split and join methods; the rationale is equal to that of tree-based containers supporting these methods.

21.3.2.4 List

21.3.2.4.1 Interface

The list-based container has the following declaration:

```
template<typename Key,
        typename Mapped,
        typename Eq_Fn = std::equal_to<Key>,
        typename Update_Policy = move_to_front_lu_policy<>,
        typename Allocator = std::allocator<char> >
class list_update;
```

The parameters have the following meaning:

1. `Key` is the key type.
2. `Mapped` is the mapped-policy.
3. `Eq_Fn` is a key equivalence functor.
4. `Update_Policy` is a policy updating positions in the list based on access patterns. It is described in the following subsection.
5. `Allocator` is an allocator type.

A list-based associative container is a container that stores elements in a linked-list. It does not order the elements by any particular order related to the keys. List-based containers are primarily useful for creating "multimaps". In fact, list-based containers are designed in this library expressly for this purpose.

List-based containers might also be useful for some rare cases, where a key is encapsulated to the extent that only key-equivalence can be tested. Hash-based containers need to know how to transform a key into a size type, and tree-based containers need to know if some key is larger than another. List-based associative containers, conversely, only need to know if two keys are equivalent.

Since a list-based associative container does not order elements by keys, is it possible to order the list in some useful manner? Remarkably, many on-line competitive algorithms exist for reordering lists to reflect access prediction. (See [85] and [57]).

21.3.2.4.2 Details

21.3.2.4.2.1 Underlying Data Structure

The graphic below shows a simple list of integer keys. If we search for the integer 6, we are paying an overhead: the link with key 6 is only the fifth link; if it were the first link, it could be accessed faster.



Figure 21.30: A simple list

List-update algorithms reorder lists as elements are accessed. They try to determine, by the access history, which keys to move to the front of the list. Some of these algorithms require adding some metadata alongside each entry.

For example, in the graphic below label A shows the counter algorithm. Each node contains both a key and a count metadata (shown in bold). When an element is accessed (e.g. 6) its count is incremented, as shown in label B. If the count reaches some predetermined value, say 10, as shown in label C, the count is set to 0 and the node is moved to the front of the list, as in label D.



Figure 21.31: The counter algorithm

21.3.2.4.2.2 Policies

this library allows instantiating lists with policies implementing any algorithm moving nodes to the front of the list (policies implementing algorithms interchanging nodes are unsupported).

Associative containers based on lists are parametrized by a `Update_Policy` parameter. This parameter defines the type of metadata each node contains, how to create the metadata, and how to decide, using this metadata, whether to move a node to the front of the list. A list-based associative container object derives (publicly) from its update policy.

An instantiation of `Update_Policy` must define internally `update_metadata` as the metadata it requires. Internally, each node of the list contains, besides the usual key and data, an instance of `typename Update_Policy::update_metadata`.

An instantiation of `Update_Policy` must define internally two operators:

```
update_metadata
operator() ();

bool
operator() (update_metadata &);
```

The first is called by the container object, when creating a new node, to create the node's metadata. The second is called by the container object, when a node is accessed (when a find operation's key is equivalent to the key of the node), to determine whether to move the node to the front of the list.

The library contains two predefined implementations of list-update policies. The first is `lu_counter_policy`, which implements the counter algorithm described above. The second is `lu_move_to_front_policy`, which unconditionally move an accessed element to the front of the list. The latter type is very useful in this library, since there is no need to associate metadata with each element. (See [57])

21.3.2.4.2.3 Use in Multimaps

In this library, there are no equivalents for the standard's multimaps and multisets; instead one uses an associative container mapping primary keys to secondary keys.

List-based containers are especially useful as associative containers for secondary keys. In fact, they are implemented here expressly for this purpose.

To begin with, these containers use very little per-entry structure memory overhead, since they can be implemented as singly-linked lists. (Arrays use even lower per-entry memory overhead, but they are less flexible in moving around entries, and have weaker invalidation guarantees).

More importantly, though, list-based containers use very little per-container memory overhead. The memory overhead of an empty list-based container is practically that of a pointer. This is important for when they are used as secondary associative-containers in situations where the average ratio of secondary keys to primary keys is low (or even 1).

In order to reduce the per-container memory overhead as much as possible, they are implemented as closely as possible to singly-linked lists.

1. List-based containers do not store internally the number of values that they hold. This means that their `size` method has linear complexity (just like `std::list`). Note that finding the number of equivalent-key values in a standard multimap also has linear complexity (because it must be done, via `std::distance` of the multimap's `equal_range` method), but usually with higher constants.
2. Most associative-container objects each hold a policy object (a hash-based container object holds a hash functor). List-based containers, conversely, only have class-wide policy objects.

21.3.2.5 Priority Queue

21.3.2.5.1 Interface

The priority queue container has the following declaration:

```
template<typename Value_Type,
        typename Cmp_Fn = std::less<Value_Type>,
        typename Tag = pairing_heap_tag,
        typename Allocator = std::allocator<char> > >
class priority_queue;
```

The parameters have the following meaning:

1. `Value_Type` is the value type.
2. `Cmp_Fn` is a value comparison functor
3. `Tag` specifies which underlying data structure to use.
4. `Allocator` is an allocator type.

The `Tag` parameter specifies which underlying data structure to use. Instantiating it by `pairing_heap_tag`, `binary_heap_tag`, `binomial_heap_tag`, `rc_binomial_heap_tag`, or `thin_heap_tag`, specifies, respectively, an underlying pairing heap ([70]), binary heap ([66]), binomial heap ([66]), a binomial heap with a redundant binary counter ([80]), or a thin heap ([75]).

As mentioned in the tutorial, `__gnu_pbds::priority_queue` shares most of the same interface with `std::priority_queue`. E.g. if `q` is a priority queue of type `Q`, then `q.top()` will return the "largest" value in the container (according to `typename`

`Q::cmp_fn`). `__gnu_pbds::priority_queue` has a larger (and very slightly different) interface than `std::priority_queue` however, since typically `push` and `pop` are deemed insufficient for manipulating priority-queues.

Different settings require different priority-queue implementations which are described in later; see `traits` discusses ways to differentiate between the different traits of different implementations.

21.3.2.5.2 Details

21.3.2.5.2.1 Iterators

There are many different underlying-data structures for implementing priority queues. Unfortunately, most such structures are oriented towards making `push` and `top` efficient, and consequently don't allow efficient access of other elements: for instance, they cannot support an efficient `find` method. In the use case where it is important to both access and "do something with" an arbitrary value, one would be out of luck. For example, many graph algorithms require modifying a value (typically increasing it in the sense of the priority queue's comparison functor).

In order to access and manipulate an arbitrary value in a priority queue, one needs to reference the internals of the priority queue from some form of an associative container - this is unavoidable. Of course, in order to maintain the encapsulation of the priority queue, this needs to be done in a way that minimizes exposure to implementation internals.

In this library the priority queue's `insert` method returns an iterator, which if valid can be used for subsequent `modify` and `erase` operations. This both preserves the priority queue's encapsulation, and allows accessing arbitrary values (since the returned iterators from the `push` operation can be stored in some form of associative container).

Priority queues' iterators present a problem regarding their invalidation guarantees. One assumes that calling `operator++` on an iterator will associate it with the "next" value. Priority-queues are self-organizing: each operation changes what the "next" value means. Consequently, it does not make sense that `push` will return an iterator that can be incremented - this can have no possible use. Also, as in the case of hash-based containers, it is awkward to define if a subsequent `push` operation invalidates a prior returned iterator: it invalidates it in the sense that its "next" value is not related to what it previously considered to be its "next" value. However, it might not invalidate it, in the sense that it can be de-referenced and used for `modify` and `erase` operations.

Similarly to the case of the other unordered associative containers, this library uses a distinction between point-type and range type iterators. A priority queue's `iterator` can always be converted to a `point_iterator`, and a `const_iterator` can always be converted to a `point_const_iterator`.

The following snippet demonstrates manipulating an arbitrary value:

```
// A priority queue of integers.
priority_queue<int > p;

// Insert some values into the priority queue.
priority_queue<int >::point_iterator it = p.push(0);

p.push(1);
p.push(2);

// Now modify a value.
p.modify(it, 3);

assert(p.top() == 3);
```

It should be noted that an alternative design could embed an associative container in a priority queue. Could, but most probably should not. To begin with, it should be noted that one could always encapsulate a priority queue and an associative container mapping values to priority queue iterators with no performance loss. One cannot, however, "un-encapsulate" a priority queue embedding an associative container, which might lead to performance loss. Assume, that one needs to associate each value with some data unrelated to priority queues. Then using this library's design, one could use an associative container mapping each value to a pair consisting of this data and a priority queue's iterator. Using the embedded method would need to use two associative containers. Similar problems might arise in cases where a value can reside simultaneously in many priority queues.

21.3.2.5.2.2 Underlying Data Structure

There are three main implementations of priority queues: the first employs a binary heap, typically one which uses a sequence; the second uses a tree (or forest of trees), which is typically less structured than an associative container's tree; the third simply uses an associative container. These are shown in the graphic below, in labels A1 and A2, label B, and label C.



Figure 21.32: Underlying Priority-Queue Data-Structures.

Roughly speaking, any value that is both pushed and popped from a priority queue must incur a logarithmic expense (in the amortized sense). Any priority queue implementation that would avoid this, would violate known bounds on comparison-based sorting (see [66] and [64]).

Most implementations do not differ in the asymptotic amortized complexity of `push` and `pop` operations, but they differ in the constants involved, in the complexity of other operations (e.g., `modify`), and in the worst-case complexity of single operations. In general, the more "structured" an implementation (i.e., the more internal invariants it possesses) - the higher its amortized complexity of `push` and `pop` operations.

This library implements different algorithms using a single class: `priority_queue`. Instantiating the `Tag` template parameter, "selects" the implementation:

1. Instantiating `Tag = binary_heap_tag` creates a binary heap of the form in represented in the graphic with labels A1 or A2. The former is internally selected by `priority_queue` if `Value_Type` is instantiated by a primitive type (e.g., an `int`); the latter is internally selected for all other types (e.g., `std::string`). This implementations is relatively unstructured, and so has good `push` and `pop` performance; it is the "best-in-kind" for primitive types, e.g., ints. Conversely, it has high worst-case performance, and can support only linear-time `modify` and `erase` operations.
2. Instantiating `Tag = pairing_heap_tag` creates a pairing heap of the form in represented by label B in the graphic above. This implementations too is relatively unstructured, and so has good `push` and `pop` performance; it is the "best-in-kind" for non-primitive types, e.g., `std::strings`. It also has very good worst-case `push` and `join` performance ($O(1)$), but has high worst-case `pop` complexity.

3. Instantiating `Tag = binomial_heap_tag` creates a binomial heap of the form represented by label B in the graphic above. This implementation is more structured than a pairing heap, and so has worse `push` and `pop` performance. Conversely, it has sub-linear worst-case bounds for `pop`, e.g., and so it might be preferred in cases where responsiveness is important.
4. Instantiating `Tag = rc_binomial_heap_tag` creates a binomial heap of the form represented by label B above, accompanied by a redundant counter which governs the trees. This implementation is therefore more structured than a binomial heap, and so has worse `push` and `pop` performance. Conversely, it guarantees $O(1)$ `push` complexity, and so it might be preferred in cases where the responsiveness of a binomial heap is insufficient.
5. Instantiating `Tag = thin_heap_tag` creates a thin heap of the form represented by the label B in the graphic above. This implementation too is more structured than a pairing heap, and so has worse `push` and `pop` performance. Conversely, it has better worst-case and identical amortized complexities than a Fibonacci heap, and so might be more appropriate for some graph algorithms.

Of course, one can use any order-preserving associative container as a priority queue, as in the graphic above label C, possibly by creating an adapter class over the associative container (much as `std::priority_queue` can adapt `std::vector`). This has the advantage that no cross-referencing is necessary at all; the priority queue itself is an associative container. Most associative containers are too structured to compete with priority queues in terms of `push` and `pop` performance.

21.3.2.5.2.3 Traits

It would be nice if all priority queues could share exactly the same behavior regardless of implementation. Sadly, this is not possible. Just one for instance is in join operations: joining two binary heaps might throw an exception (not corrupt any of the heaps on which it operates), but joining two pairing heaps is exception free.

Tags and traits are very useful for manipulating generic types. `__gnu_pbds::priority_queue` publicly defines `container_tag` as one of the tags. Given any container `Cntnr`, the tag of the underlying data structure can be found via `typename Cntnr::container_tag`. This is one of the possible tags shown in the graphic below.



Figure 21.33: Priority-Queue Data-Structure Tags.

Additionally, a traits mechanism can be used to query a container type for its attributes. Given any container `Cntnr`, then

```
__gnu_pbds::container_traits<Cntnr>
```

is a traits class identifying the properties of the container.

To find if a container might throw if two of its objects are joined, one can use

```
container_traits<Cntnr>::split_join_can_throw
```

Different priority-queue implementations have different invalidation guarantees. This is especially important, since there is no way to access an arbitrary value of priority queues except for iterators. Similarly to associative containers, one can use

```
container_traits<Cntnr>::invalidation_guarantee
```

to get the invalidation guarantee type of a priority queue.

It is easy to understand from the graphic above, what `container_traits<Cntnr>::invalidation_guarantee` will be for different implementations. All implementations of type represented by label B have `point_invalidation_guarantee`: the container can freely internally reorganize the nodes - range-type iterators are invalidated, but point-type iterators are always valid. Implementations of type represented by labels A1 and A2 have `basic_invalidation_guarantee`: the container can freely internally reallocate the array - both point-type and range-type iterators might be invalidated.

This has major implications, and constitutes a good reason to avoid using binary heaps. A binary heap can perform `modify` or `erase` efficiently given a valid point-type iterator. However, in order to supply it with a valid point-type iterator, one needs to iterate (linearly) over all values, then supply the relevant iterator (recall that a range-type iterator can always be converted to a point-type iterator). This means that if the number of `modify` or `erase` operations is non-negligible (say super-logarithmic in the total sequence of operations) - binary heaps will perform badly.

21.4 Testing

21.4.1 Regression

The library contains a single comprehensive regression test. For a given container type in this library, the test creates an object of the container type and an object of the corresponding standard type (e.g., `std::set`). It then performs a random sequence of methods with random arguments (e.g., inserts, erases, and so forth) on both objects. At each operation, the test checks the return value of the method, and optionally both compares this library's object with the standard's object as well as performing other consistency checks on this library's object (e.g., order preservation, when applicable, or node invariants, when applicable).

Additionally, the test integrally checks exception safety and resource leaks. This is done as follows. A special allocator type, written for the purpose of the test, both randomly throws an exceptions when allocations are performed, and tracks allocations and de-allocations. The exceptions thrown at allocations simulate memory-allocation failures; the tracking mechanism checks for memory-related bugs (e.g., resource leaks and multiple de-allocations). Both this library's containers and the containers' value-types are configured to use this allocator.

For granularity, the test is split into the several sources, each checking only some containers.

For more details, consult the files in `testsuite/ext/pb_ds/regression`.

21.4.2 Performance

21.4.2.1 Hash-Based

21.4.2.1.1 Text find

21.4.2.1.1.1 Description

This test inserts a number of values with keys from an arbitrary text ([98]) into a container, then performs a series of finds using `find`. It measures the average time for `find` as a function of the number of values inserted.

It uses the test file: `performance/ext/pb_ds/text_find_timing_test.cc`

And uses the data file: `filethirty_years_among_the_dead_preproc.txt`

The test checks the effect of different range-hashing functions, trigger policies, and cache-hashing policies.

21.4.2.1.1.2 Results

The graphic below show the results for the native and collision-chaining hash types the function applied being a text find timing test using `find`.



The abbreviated names in the legend of the graphic above are instantiated with the types in the following table.

Name/Instantiating Type	Parameter	Details	Parameter	Details
n_hash_map_ncah				
std::tr1::unordered_map	hash_code	false		
cc_hash_mod_prime_1div1_nsth_map				
cc_hash_table	Comb_Hash_Fn	direct_mod_range_hashing		
	Resize_Policy	hash_standard_resize_policy	Size_Policy	hash_prime_size_policy
			Trigger_Policy	hash_load_check_resize_policy with $\alpha_{\min} = 1/8$ and $\alpha_{\max} = 1/1$
cc_hash_mask_exp_1div2_sth_map				
cc_hash_table	Comb_Hash_Fn	direct_mask_range_hashing		
	Resize_Policy	hash_standard_resize_policy	Size_Policy	hash_exponential_size_policy
			Trigger_Policy	hash_load_check_resize_policy with $\alpha_{\min} = 1/8$ and $\alpha_{\max} = 1/2$
cc_hash_mask_exp_1div1_nsth_map				
cc_hash_table	Comb_Hash_Fn	direct_mask_range_hashing		
	Resize_Policy	hash_standard_resize_policy	Size_Policy	hash_exponential_size_policy
			Trigger_Policy	hash_load_check_resize_policy with $\alpha_{\min} = 1/8$ and $\alpha_{\max} = 1/1$
cc_hash_mask_exp_1div2_nsth_map				

<i>Name/Instantiating Type</i>	<i>Parameter</i>	<i>Details</i>	<i>Parameter</i>	<i>Details</i>
cc_hash_table	Comb_Hash_Fn	direct_mask_range_hashing		
	Resize_Policy	hash_standard_resize_policy	Size_Policy	hash_exponential_size
			Trigger_Policy	hash_load_check_resize with $\alpha_{\min} = 1/8$ and $\alpha_{\max} = 1/2$

21.4.2.1.1.3 Observations

In this setting, the range-hashing scheme affects performance more than other policies. As the results show, containers using mod-based range-hashing (including the native hash-based container, which is currently hard-wired to this scheme) have lower performance than those using mask-based range-hashing. A modulo-based range-hashing scheme’s main benefit is that it takes into account all hash-value bits. Standard string hash-functions are designed to create hash values that are nearly-uniform as ([77]).

Trigger policies, i.e. the load-checks constants, affect performance to a lesser extent.

Perhaps surprisingly, storing the hash value alongside each entry affects performance only marginally, at least in this library’s implementation. (Unfortunately, it was not possible to run the tests with `std::tr1::unordered_map`’s `cache_hash_code = true`, as it appeared to malfunction.)

21.4.2.1.2 Integer find

21.4.2.1.2.1 Description

This test inserts a number of values with uniform integer keys into a container, then performs a series of finds using `find`. It measures the average time for `find` as a function of the number of values inserted.

It uses the test file: `performance/ext/pb_ds/random_int_find_timing.cc`

The test checks the effect of different underlying hash-tables, range-hashing functions, and trigger policies.

21.4.2.1.2.2 Results

There are two sets of results for this type, one for collision-chaining hashes, and one for general-probe hashes.

The first graphic below shows the results for the native and collision-chaining hash types. The function applied being a random integer timing test using `find`.



▼ n_hash_map_ncah
 ▽ cc_hash_mod_prime_nea_lc_1div8_1div1_nsth_map
 ▴ cc_hash_mod_prime_nea_lc_1div8_1div2_nsth_map
 ◇ cc_hash_mask_exp_nea_lc_1div8_1div1_nsth_map
 × cc_hash_mask_exp_nea_lc_1div8_1div2_nsth_map

The abbreviated names in the legend of the graphic above are instantiated with the types in the following table.

Name/Instantiating Type	Parameter	Details	Parameter	Details
n_hash_map_ncah				
std::tr1::unordered_map	hash_code	false		
cc_hash_mod_prime_1div1_nsth_map				
cc_hash_table	Comb_Hash_Fn	direct_mod_range_hashing		
	Resize_Policy	hash_standard_resize_policy	Size_Policy	hash_prime_size_policy
			Trigger_Policy	hash_load_check_resize_policy with $\alpha_{\min} = 1/8$ and $\alpha_{\max} = 1/1$
cc_hash_mod_prime_1div2_nsth_map				
cc_hash_table	Comb_Hash_Fn	direct_mod_range_hashing		
	Resize_Policy	hash_standard_resize_policy	Size_Policy	hash_prime_size_policy
			Trigger_Policy	hash_load_check_resize_policy with $\alpha_{\min} = 1/8$ and $\alpha_{\max} = 1/2$
cc_hash_mask_exp_1div1_nsth_map				
cc_hash_table	Comb_Hash_Fn	direct_mask_range_hashing		
	Resize_Policy	hash_standard_resize_policy	Size_Policy	hash_exponential_size_policy
			Trigger_Policy	hash_load_check_resize_policy with $\alpha_{\min} = 1/8$ and $\alpha_{\max} = 1/1$
cc_hash_mask_exp_1div2_nsth_map				
cc_hash_table	Comb_Hash_Fn	direct_mask_range_hashing		
	Resize_Policy	hash_standard_resize_policy	Size_Policy	hash_exponential_size_policy

<i>Name/Instantiating Type</i>	<i>Parameter</i>	<i>Details</i>	<i>Parameter</i>	<i>Details</i>
			Trigger_Policy	hash_load_check_resize_policy with $\alpha_{\min} = 1/8$ and $\alpha_{\max} = 1/2$

And the second graphic shows the results for the native and general-probe hash types. The function applied being a random integer timing test using `find`.



△gp_hash_mod_quadp_prime_nea_lc_1div8_1div2_nsth_map
▼n_hash_map_ncah
×gp_hash_mask_linp_exp_nea_lc_1div8_1div2_nsth_map

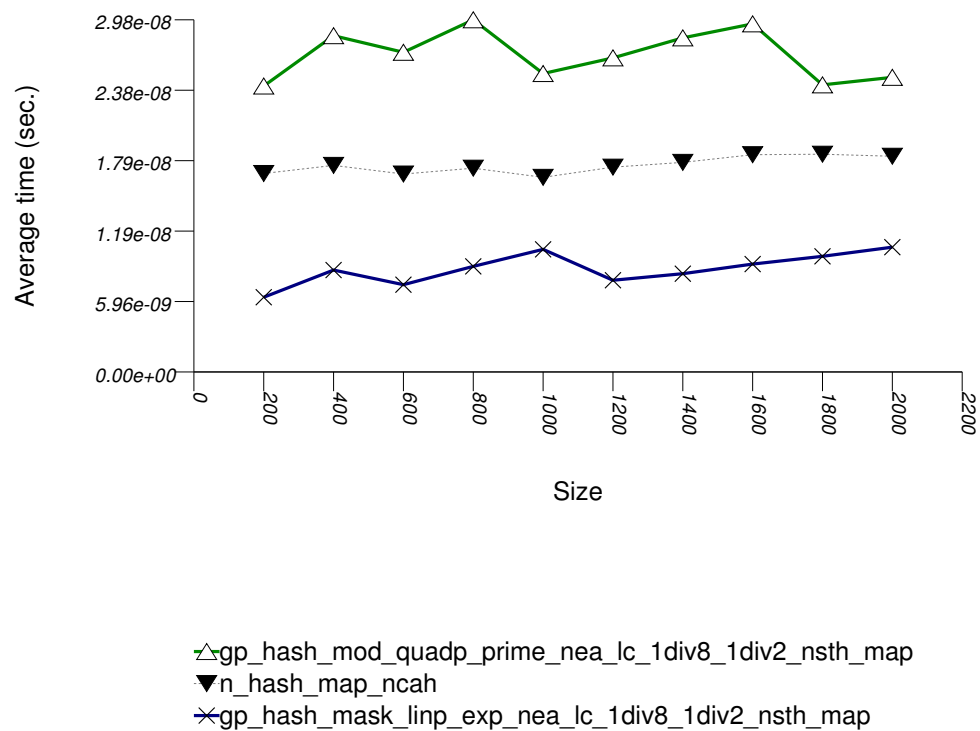
The abbreviated names in the legend of the graphic above are instantiated with the types in the following table.

<i>Name/Instantiating Type</i>	<i>Parameter</i>	<i>Details</i>	<i>Parameter</i>	<i>Details</i>
n_hash_map_ncah				
std::tr1::unordered_map	hash_code	false		
gp_hash_mod_quadp_prime_1div2_nsth_map				
gp_hash_table	Comb_Hash_Fn	direct_mod_range_hashing		
	Probe_Fn	quadratic_probe_fn		
	Resize_Policy	hash_standard_resize_policy	Size_Policy	hash_prime_size_policy
			Trigger_Policy	hash_load_check_resize_policy with $\alpha_{\min} = 1/8$ and $\alpha_{\max} = 1/2$
gp_hash_mask_linp_exp_1div2_nsth_map				
gp_hash_table	Comb_Hash_Fn	direct_mask_range_hashing		
	Probe_Fn	linear_probe_fn		
	Resize_Policy	hash_standard_resize_policy	Size_Policy	hash_exponential_size_policy
			Trigger_Policy	hash_load_check_resize_policy with $\alpha_{\min} = 1/8$ and $\alpha_{\max} = 1/2$

The abbreviated names in the legend of the graphic above are instantiated with the types in the following table.

<i>Name/Instantiating Type</i>	<i>Parameter</i>	<i>Details</i>	<i>Parameter</i>	<i>Details</i>
n_hash_map_ncah				
std::tr1::unordered_map	reacnash_code	false		
cc_hash_mod_prime_1div1_nsth_map				
cc_hash_table	Comb_Hash_Fn	direct_mod_range_hashing		
	Resize_Policy	hash_standard_resize_policy	Size_Policy	hash_prime_size_policy
			Trigger_Policy	hash_load_check_resize_policy with $\alpha_{\min} = 1/8$ and $\alpha_{\max} = 1/1$
cc_hash_mod_prime_1div2_nsth_map				
cc_hash_table	Comb_Hash_Fn	direct_mod_range_hashing		
	Resize_Policy	hash_standard_resize_policy	Size_Policy	hash_prime_size_policy
			Trigger_Policy	hash_load_check_resize_policy with $\alpha_{\min} = 1/8$ and $\alpha_{\max} = 1/2$
cc_hash_mask_exp_1div1_nsth_map				
cc_hash_table	Comb_Hash_Fn	direct_mask_range_hashing		
	Resize_Policy	hash_standard_resize_policy	Size_Policy	hash_exponential_size_policy
			Trigger_Policy	hash_load_check_resize_policy with $\alpha_{\min} = 1/8$ and $\alpha_{\max} = 1/1$
cc_hash_mask_exp_1div2_nsth_map				
cc_hash_table	Comb_Hash_Fn	direct_mask_range_hashing		
	Resize_Policy	hash_standard_resize_policy	Size_Policy	hash_exponential_size_policy
			Trigger_Policy	hash_load_check_resize_policy with $\alpha_{\min} = 1/8$ and $\alpha_{\max} = 1/2$

And the second graphic shows the results for the native and general-probe hash types. The function applied being a random integer timing test using `find`.



The abbreviated names in the legend of the graphic above are instantiated with the types in the following table.

Name/Instantiating Type	Parameter	Details	Parameter	Details
n_hash_map_ncah				
std::tr1::unordered_map	rehash_code	false		
gp_hash_mod_quadp_prime_1div2_nsth_map				
gp_hash_table	Comb_Hash_Fn	direct_mod_range_hashing		
	Probe_Fn	quadratic_probe_fn		
	Resize_Policy	hash_standard_resize_policy	Size_Policy	hash_prime_size_policy
			Trigger_Policy	hash_load_check_resize_policy with $\alpha_{\min} = 1/8$ and $\alpha_{\max} = 1/2$
gp_hash_mask_linp_exp_1div2_nsth_map				
gp_hash_table	Comb_Hash_Fn	direct_mask_range_hashing		
	Probe_Fn	linear_probe_fn		
	Resize_Policy	hash_standard_resize_policy	Size_Policy	hash_exponential_size_policy
			Trigger_Policy	hash_load_check_resize_policy with $\alpha_{\min} = 1/8$ and $\alpha_{\max} = 1/2$

21.4.2.1.3.3 Observations

This test shows similar results to Hash-Based Integer find Find Timing test.

21.4.2.1.4 Integer Subscript insert

21.4.2.1.4.1 Description

This test inserts a number of values with uniform i.i.d. integer keys into a container, using `operator[]`. It measures the average time for `operator[]` as a function of the number of values inserted.

It uses the test file: `performance/ext/pb_ds/random_int_subscript_insert_timing.cc`

The test checks the effect of different underlying hash-tables.

21.4.2.1.4.2 Results

There are two sets of results for this type, one for collision-chaining hashes, and one for general-probe hashes.

The first graphic below shows the results for the native and collision-chaining hash types, using as the function applied an integer subscript timing test with `insert`.



The abbreviated names in the legend of the graphic above are instantiated with the types in the following table.

Name/Instantiating Type	Parameter	Details	Parameter	Details
n_hash_map_ncah				
std::tr1::unordered_map	hash_code	false		
cc_hash_mod_prime_1div1_nsth_map				
cc_hash_table	Comb_Hash_Fn	direct_mod_range_hashing		
	Resize_Policy	hash_standard_resize_policy	Size_Policy	hash_prime_size_policy
			Trigger_Policy	hash_load_check_resize_policy with $\alpha_{\min} = 1/8$ and $\alpha_{\max} = 1/1$
cc_hash_mod_prime_1div2_nsth_map				
cc_hash_table	Comb_Hash_Fn	direct_mod_range_hashing		
	Resize_Policy	hash_standard_resize_policy	Size_Policy	hash_prime_size_policy
			Trigger_Policy	hash_load_check_resize_policy with $\alpha_{\min} = 1/8$ and $\alpha_{\max} = 1/2$
cc_hash_mask_exp_1div1_nsth_map				

<i>Name/Instantiating Type</i>	<i>Parameter</i>	<i>Details</i>	<i>Parameter</i>	<i>Details</i>
gp_hash_mask_linp_exp_1div2_nsth_map				
gp_hash_table	Comb_Hash_Fn	direct_mask_range	hashing	
	Probe_Fn	linear_probe_fn		
	Resize_Policy	hash_standard_resize_policy	Size_Policy	hash_exponential_size
			Trigger_Policy	hash_load_check_resize with $\alpha_{\min} = 1/8$ and $\alpha_{\max} = 1/2$

21.4.2.1.4.3 Observations

In this setting, as in Hash-Based Text `find` Find Timing test and Hash-Based Integer `find` Find Timing test , the choice of underlying hash-table underlying hash-table affects performance most, then the range-hashing scheme, and finally any other policies.

There are some differences, however:

1. In this setting, probing tables function sometimes more efficiently than collision-chaining tables. This is explained shortly.
2. The performance graphs have a "saw-tooth" shape. The average insert time rises and falls. As values are inserted into the container, the load factor grows larger. Eventually, a resize occurs. The reallocations and rehashing are relatively expensive. After this, the load factor is smaller than before.

Collision-chaining containers use indirection for greater flexibility; probing containers store values contiguously, in an array (see Figure Motivation::Different underlying data structures A and B, respectively). It follows that for simple data types, probing containers access their allocator less frequently than collision-chaining containers, (although they still have less efficient probing sequences). This explains why some probing containers fare better than collision-chaining containers in this case.

Within each type of hash-table, the range-hashing scheme affects performance more than other policies. This is similar to the situation in Hash-Based Text `find` Find Timing Test and Hash-Based Integer `find` Find Timing Test. Unsurprisingly, however, containers with lower α_{\max} perform worse in this case, since more re-hashes are performed.

21.4.2.1.5 Integer `find` with Skewed-Distribution

21.4.2.1.5.1 Description

This test inserts a number of values with a markedly non-uniform integer keys into a container, then performs a series of finds using `find`. It measures the average time for `find` as a function of the number of values in the containers. The keys are generated as follows. First, a uniform integer is created. Then it is then shifted left 8 bits.

It uses the test file: `performance/ext/pb_ds/hash_zlob_random_int_find_timing.cc`

The test checks the effect of different range-hashing functions and trigger policies.

21.4.2.1.5.2 Results

The graphic below show the results for the native, collision-chaining, and general-probing hash types.

The range-hashing scheme affects performance dramatically. A mask-based range-hashing scheme effectively maps all values into the same bucket. Access degenerates into a search within an unordered linked-list. In the graphic above, it should be noted that `std::tr1::unordered_map` is hard-wired currently to mod-based and mask-based schemes, respectively.

When observing the settings of this test, it is apparent that the keys' distribution is far from natural. One might ask if the test is not contrived to show that, in some cases, mod-based range hashing does better than mask-based range hashing. This is, in fact just the case. A more natural case in which mod-based range hashing is better was not encountered. Thus the inescapable conclusion: real-life key distributions are handled better with an appropriate hash function and a mask-based range-hashing function. (`pb_ds/example/hash_shift_mask.cc` shows an example of handling this a-priori known skewed distribution with a mask-based range-hashing function). If hash performance is bad, a χ^2 test can be used to check how to transform it into a more uniform distribution.

For this reason, this library's default range-hashing function is mask-based.

21.4.2.1.6 Erase Memory Use

21.4.2.1.6.1 Description

This test inserts a number of uniform integer keys into a container, then erases all keys except one. It measures the final size of the container.

It uses the test file: `performance/ext/pb_ds/hash_random_int_erase_mem_usage.cc`

The test checks how containers adjust internally as their logical size decreases.

21.4.2.1.6.2 Results

The graphic below show the results for the native, collision-chaining, and general-probing hash types.



The abbreviated names in the legend of the graphic above are instantiated with the types in the following table.

<i>Name/Instantiating Type</i>	<i>Parameter</i>	<i>Details</i>	<i>Parameter</i>	<i>Details</i>
n_hash_map_ncah				
<code>std::tr1::unordered_map</code>	<code>rehash_code</code>	<code>false</code>		
cc_hash_mod_prime_1div1_nsth_map				
<code>cc_hash_table</code>	<code>Comb_Hash_Fn</code>	<code>direct_mod_range_hashing</code>		
	<code>Resize_Policy</code>	<code>hash_standard_resize_policy</code>	<code>Size_Policy</code>	<code>hash_prime_size_policy</code>
			<code>Trigger_Policy</code>	<code>hash_load_check_resize_policy</code> with $\alpha_{\min} = 1/8$ and $\alpha_{\max} = 1/1$
cc_hash_mask_exp_1div2_nsth_map				
<code>cc_hash_table</code>	<code>Comb_Hash_Fn</code>	<code>direct_mask_range_hashing</code>		
	<code>Resize_Policy</code>	<code>hash_standard_resize_policy</code>	<code>Size_Policy</code>	<code>hash_exponential_size_policy</code>
			<code>Trigger_Policy</code>	<code>hash_load_check_resize_policy</code> with $\alpha_{\min} = 1/8$ and $\alpha_{\max} = 1/2$
gp_hash_mask_linp_exp_1div2_nsth_set				
<code>gp_hash_table</code>	<code>Comb_Hash_Fn</code>	<code>direct_mask_range_hashing</code>		
	<code>Probe_Fn</code>	<code>linear_probe_fn</code>		
	<code>Resize_Policy</code>	<code>hash_standard_resize_policy</code>	<code>Size_Policy</code>	<code>hash_exponential_size_policy</code>
			<code>Trigger_Policy</code>	<code>hash_load_check_resize_policy</code> with $\alpha_{\min} = 1/8$ and $\alpha_{\max} = 1/2$

21.4.2.1.6.3 Observations

The standard's hash-based containers act very differently than trees in this respect. When erasing numerous keys from an standard associative-container, the resulting memory user varies greatly depending on whether the container is tree-based or hash-based. This is a fundamental consequence of the standard's interface for associative containers, and it is not due to a specific implementation.

21.4.2.2 Branch-Based

21.4.2.2.1 Text insert

21.4.2.2.1.1 Description

This test inserts a number of values with keys from an arbitrary text ([wickland96thirty]) into a container using `insert`. It measures the average time for `insert` as a function of the number of values inserted.

The test checks the effect of different underlying data structures.

It uses the test file: `performance/ext/pb_ds/tree_text_insert_timing.cc`

21.4.2.2.1.2 Results

The three graphics below show the results for the native tree and this library's node-based trees, the native tree and this library's vector-based trees, and the native tree and this library's PATRICIA-trie, respectively.

The graphic immediately below shows the results for the native tree type and several node-based tree types.



The abbreviated names in the legend of the graphic above are instantiated with the types in the following table.

<i>Name/Instantiating Type</i>	<i>Parameter</i>	<i>Details</i>
n_map		
std::map		
splay_tree_map		
tree	Tag	splay_tree_tag
	Node_update	null_node_update
rb_tree_map		
tree	Tag	rb_tree_tag
	Node_update	null_node_update

The graphic below shows the results for the native tree type and a vector-based tree type.



The abbreviated names in the legend of the graphic above are instantiated with the types in the following table.

<i>Name/Instantiating Type</i>	<i>Parameter</i>	<i>Details</i>
n_map		
std::map		
ov_tree_map		
tree	Tag	ov_tree_tag
	Node_update	null_node_update

The graphic below shows the results for the native tree type and a PATRICIA trie type.



The abbreviated names in the legend of the graphic above are instantiated with the types in the following table.

<i>Name/Instantiating Type</i>	<i>Parameter</i>	<i>Details</i>
n_map		
std::map		
pat_trie_map		
tree	Tag	pat_trie_tag
	Node_update	null_node_update

21.4.2.2.1.3 Observations

Observing the first graphic implies that for this setting, a splay tree (tree with Tag = splay_tree_tag) does not do well. See also the Branch-Based Text find Find Timing Test. The two red-black trees perform better.

Observing the second graphic, an ordered-vector tree (tree with Tag = ov_tree_tag) performs abysmally. Inserting into this type of tree has linear complexity [austern00noset].

Observing the third and last graphic, A PATRICIA trie (trie with Tag = pat_trie_tag) has abysmal performance, as well. This is not that surprising, since a large-fan-out PATRICIA trie works like a hash table with collisions resolved by a sub-trie. Each time a collision is encountered, a new "hash-table" is built A large fan-out PATRICIA trie, however, does well in look-ups (see Branch-Based Text find Find Timing Test). It may be beneficial in semi-static settings.

21.4.2.2.2 Text find

21.4.2.2.2.1 Description

This test inserts a number of values with keys from an arbitrary text ([wickland96thirty]) into a container, then performs a series of finds using find. It measures the average time for find as a function of the number of values inserted.

It uses the test file: performance/ext/pb_ds/text_find_timing.cc

The test checks the effect of different underlying data structures.

21.4.2.2.2.2 Results

The graphic immediately below shows the results for the native tree type and several other tree types.



The abbreviated names in the legend of the graphic above are instantiated with the types in the following table.

<i>Name/Instantiating Type</i>	<i>Parameter</i>	<i>Details</i>
n_map		
std::map		
splay_tree_map		
tree	Tag	splay_tree_tag
	Node_Update	null_node_update
rb_tree_map		
tree	Tag	rb_tree_tag
	Node_Update	null_node_update
ov_tree_map		
tree	Tag	ov_tree_tag
	Node_Update	null_node_update
pat_trie_map		
tree	Tag	pat_trie_tag
	Node_Update	null_node_update

21.4.2.2.3 Observations

For this setting, a splay tree (tree with Tag = splay_tree_tag) does not do well. This is possibly due to two reasons:

1. A splay tree is not guaranteed to be balanced [motwani95random]. If a splay tree contains n nodes, its average root-leaf path can be $m \gg \log(n)$.
2. Assume a specific root-leaf search path has length m , and the search-target node has distance m' from the root. A red-black tree will require $m + 1$ comparisons to find the required node; a splay tree will require $2 m'$ comparisons. A splay tree, consequently, can perform many more comparisons than a red-black tree.

An ordered-vector tree (`tree` with `Tag = ov_tree_tag`), a red-black tree (`tree` with `Tag = rb_tree_tag`), and the native red-black tree all share approximately the same performance.

An ordered-vector tree is slightly slower than red-black trees, since it requires, in order to find a key, more math operations than they do. Conversely, an ordered-vector tree requires far lower space than the others. ([austern00noset], however, seems to have an implementation that is also faster than a red-black tree).

A PATRICIA trie (`trie` with `Tag = pat_trie_tag`) has good look-up performance, due to its large fan-out in this case. In this setting, a PATRICIA trie has look-up performance comparable to a hash table (see Hash-Based Text `find` Timing Test), but it is order preserving. This is not that surprising, since a large-fan-out PATRICIA trie works like a hash table with collisions resolved by a sub-trie. A large-fan-out PATRICIA trie does not do well on modifications (see Tree-Based and Trie-Based Text Insert Timing Test). Therefore, it is possibly beneficial in semi-static settings.

21.4.2.2.3 Text `find` with Locality-of-Reference

21.4.2.2.3.1 Description

This test inserts a number of values with keys from an arbitrary text ([wickland96thirty]) into a container, then performs a series of finds using `find`. It is different than Tree-Based and Trie-Based Text `find` Find Timing Test in the sequence of finds it performs: this test performs multiple `find`s on the same key before moving on to the next key. It measures the average time for `find` as a function of the number of values inserted.

It uses the test file: `performance/ext/pb_ds/tree_text_lor_find_timing.cc`

The test checks the effect of different underlying data structures in a locality-of-reference setting.

21.4.2.2.3.2 Results

The graphic immediately below shows the results for the native tree type and several other tree types.



The abbreviated names in the legend of the graphic above are instantiated with the types in the following table.

<i>Name/Instantiating Type</i>	<i>Parameter</i>	<i>Details</i>
n_map		
std::map		
splay_tree_map		
tree	Tag	splay_tree_tag
	Node_Update	null_node_update
rb_tree_map		
tree	Tag	rb_tree_tag
	Node_Update	null_node_update
ov_tree_map		
tree	Tag	ov_tree_tag
	Node_Update	null_node_update
pat_trie_map		
tree	Tag	pat_trie_tag
	Node_Update	null_node_update

21.4.2.2.3.3 Observations

For this setting, an ordered-vector tree (tree with Tag = ov_tree_tag), a red-black tree (tree with Tag = rb_tree_tag), and the native red-black tree all share approximately the same performance.

A splay tree (tree with Tag = splay_tree_tag) does much better, since each (successful) find "bubbles" the corresponding node to the root of the tree.

21.4.2.2.4 split and join

21.4.2.2.4.1 Description

This test a container, inserts into a number of values, splits the container at the median, and joins the two containers. (If the containers are one of this library's trees, it splits and joins with the `split` and `join` method; otherwise, it uses the `erase` and `insert` methods.) It measures the time for splitting and joining the containers as a function of the number of values inserted.

It uses the test file: `performance/ext/pb_ds/tree_split_join_timing.cc`

The test checks the performance difference of `join` as opposed to a sequence of `insert` operations; by implication, this test checks the most efficient way to erase a sub-sequence from a tree-like-based container, since this can always be performed by a small sequence of splits and joins.

21.4.2.2.4.2 Results

The graphic immediately below shows the results for the native tree type and several other tree types.



The abbreviated names in the legend of the graphic above are instantiated with the types in the following table.

<i>Name/Instantiating Type</i>	<i>Parameter</i>	<i>Details</i>
n_set		
std::set		
splay_tree_set		
tree	Tag	splay_tree_tag
	Node_Update	null_node_update
rb_tree_set		
tree	Tag	rb_tree_tag
	Node_Update	null_node_update
ov_tree_set		
tree	Tag	ov_tree_tag
	Node_Update	null_node_update
pat_trie_map		
tree	Tag	pat_trie_tag
	Node_Update	null_node_update

21.4.2.2.4.3 Observations

In this test, the native red-black trees must be split and joined externally, through a sequence of `erase` and `insert` operations. This is clearly super-linear, and it is not that surprising that the cost is high.

This library's tree-based containers use in this test the `split` and `join` methods, which have lower complexity: the `join` method of a splay tree (tree with `Tag = splay_tree_tag`) is quadratic in the length of the longest root-leaf path, and linear in the total number of elements; the `join` method of a red-black tree (tree with `Tag = rb_tree_tag`) or an ordered-vector tree (tree with `Tag = ov_tree_tag`) is linear in the number of elements.

Asides from orders of growth, this library's trees access their allocator very little in these operations, and some of them do not access it at all. This leads to lower constants in their complexity, and, for some containers, to exception-free splits and joins (which can be determined via `container_traits`).

It is important to note that `split` and `join` are not esoteric methods - they are the most efficient means of erasing a contiguous range of values from a tree based container.

21.4.2.2.5 Order-Statistics

21.4.2.2.5.1 Description

This test creates a container, inserts random integers into the the container, and then checks the order-statistics of the container's values. (If the container is one of this library's trees, it does this with the `order_of_key` method of `tree_order_statistics_node_update`; otherwise, it uses the `find` method and `std::distance`.) It measures the average time for such queries as a function of the number of values inserted.

It uses the test file: `performance/ext/pb_ds/tree_order_statistics_timing.cc`

The test checks the performance difference of policies based on node-invariant as opposed to a external functions.

21.4.2.2.5.2 Results

The graphic immediately below shows the results for the native tree type and several other tree types.



The abbreviated names in the legend of the graphic above are instantiated with the types in the following table.

Name/Instantiating Type	Parameter	Details
n_set		
std::set		
splay_tree_ost_set		
tree	Tag	splay_tree_tag
	Node_Update	tree_order_statistics_node_update
rb_tree_ost_set		
tree	Tag	rb_tree_tag
	Node_Update	tree_order_statistics_node_update

21.4.2.2.5.3 Observations

In this test, the native red-black tree can support order-statistics queries only externally, by performing a `find` (alternatively, `lower_bound` or `upper_bound`) and then using `std::distance`. This is clearly linear, and it is not that surprising that the cost is high.

This library's tree-based containers use in this test the `order_of_key` method of `tree_order_statistics_node_update`. This method has only linear complexity in the length of the root-node path. Unfortunately, the average path of a splay tree (tree with `Tag = splay_tree_tag`) can be higher than logarithmic; the longest path of a red-black tree (tree with `Tag = rb_tree_tag`) is logarithmic in the number of elements. Consequently, the splay tree has worse performance than the red-black tree.

21.4.2.3 Multimap

21.4.2.3.1 Text `find` with Small Secondary-to-Primary Key Ratios

21.4.2.3.1.1 Description

This test inserts a number of pairs into a container. The first item of each pair is a string from an arbitrary text ([`wickland96thirty`]), and the second is a uniform i.i.d. integer. The container is a "multimap" - it considers the first member of each pair as a primary key, and the second member of each pair as a secondary key (see `Motivation::Associative Containers::Alternative to Multiple Equivalent Keys`). There are 400 distinct primary keys, and the ratio of secondary keys to primary keys ranges from 1 to 5.

The test measures the average find-time as a function of the number of values inserted. For this library's containers, it finds the secondary key from a container obtained from finding a primary key. For the native multimaps, it searches a range obtained using `std::equal_range` on a primary key.

It uses the test file: `performance/ext/pb_ds/multimap_text_find_timing_small.cc`

The test checks the find-time scalability of different "multimap" designs.

21.4.2.3.1.2 Results

The graphic below show the results for "multimaps" which use a tree-based container for primary keys.

<i>Name/Instantiating Type</i>	<i>Parameter</i>	<i>Details</i>	<i>Parameter</i>	<i>Details</i>	<i>Parameter</i>	<i>Details</i>
					Trigger_Policy	hash_load_check_1 with $\alpha_{\min} = 1/8$ and $\alpha_{\max} = 1/2$

21.4.2.3.1.3 Observations

See Observations::Mapping-Semantics Considerations.

21.4.2.3.2 Text find with Large Secondary-to-Primary Key Ratios

21.4.2.3.2.1 Description

This test inserts a number of pairs into a container. The first item of each pair is a string from an arbitrary text ([wickland96thirty]), and the second is a uniform integer. The container is a "multimap" - it considers the first member of each pair as a primary key, and the second member of each pair as a secondary key. There are 400 distinct primary keys, and the ratio of secondary keys to primary keys ranges from 1 to 5.

The test measures the average find-time as a function of the number of values inserted. For this library's containers, it finds the secondary key from a container obtained from finding a primary key. For the native multimaps, it searches a range obtained using `std::equal_range` on a primary key.

It uses the test file: `performance/ext/pb_ds/multimap_text_find_timing_large.cc`

The test checks the find-time scalability of different "multimap" designs.

21.4.2.3.2.2 Results

The graphic below show the results for "multimaps" which use a tree-based container for primary keys.



The abbreviated names in the legend of the graphic above are instantiated with the types in the following table.

Name/Instantiating Type	Parameter	Details	Parameter	Details	Parameter	Details
n_hash_mmap						
std::tr1::unordered_multimap						
rb_tree_mmap_lu_mtf_set						
cc_hash_table	Comb_Hash_Fn	direct_mask	range_hashing			
	Resize_Policy	hash_standard	Size_Policy	hash_exponential_size_policy		
			Trigger_Policy	hash_load_check_resize_trigger with $\alpha_{\min} = 1/8$ and $\alpha_{\max} = 1/2$		
	Mapped	list_update	Update_Policy	yu_move_to_front_policy		
rb_tree_mmap_cc_hash_mask_exp_1div2_nsth_set						
cc_hash_table	Comb_Hash_Fn	direct_mask	range_hashing			
	Resize_Policy	hash_standard	Size_Policy	hash_exponential_size_policy		
			Trigger_Policy	hash_load_check_resize_trigger with $\alpha_{\min} = 1/8$ and $\alpha_{\max} = 1/2$		
	Mapped	cc_hash_table	Comb_Hash_Fn	direct_mask	range_hashing	
			Resize_Policy	hash_standard	Size_Policy	hash_exponential

<i>Name/Instantiating Type</i>	<i>Parameter</i>	<i>Details</i>	<i>Parameter</i>	<i>Details</i>	<i>Parameter</i>	<i>Details</i>
					Trigger_Policy	hash_load_check_1 with $\alpha_{\min} = 1/8$ and $\alpha_{\max} = 1/2$

21.4.2.3.2.3 Observations

See Observations::Mapping-Semantics Considerations.

21.4.2.3.3 Text insert with Small Secondary-to-Primary Key Ratios

21.4.2.3.3.1 Description

This test inserts a number of pairs into a container. The first item of each pair is a string from an arbitrary text ([wickland96thirty]), and the second is a uniform integer. The container is a "multimap" - it considers the first member of each pair as a primary key, and the second member of each pair as a secondary key. There are 400 distinct primary keys, and the ratio of secondary keys to primary keys ranges from 1 to 5.

The test measures the average insert-time as a function of the number of values inserted. For this library's containers, it inserts a primary key into the primary associative container, then a secondary key into the secondary associative container. For the native multimaps, it obtains a range using `std::equal_range`, and inserts a value only if it was not contained already.

It uses the test file: `performance/ext/pb_ds/multimap_text_insert_timing_small.cc`

The test checks the insert-time scalability of different "multimap" designs.

21.4.2.3.3.2 Results

The graphic below show the results for "multimaps" which use a tree-based container for primary keys.



The abbreviated names in the legend of the graphic above are instantiated with the types in the following table.

Name/Instantiating Type	Parameter	Details	Parameter	Details	Parameter	Details
n_hash_mmap						
std::tr1::unordered_multimap						
rb_tree_mmap_lu_mtf_set						
cc_hash_table	Comb_Hash_Fn	direct_mask	range_hashing			
	Resize_Policy	hash_standard	Size_Policy	hash_exponential_size_policy		
			Trigger_Policy	hash_load_check_resize_trigger with $\alpha_{\min} = 1/8$ and $\alpha_{\max} = 1/2$		
	Mapped	list_update	Update_Policy	lu_move_to_front_policy		
rb_tree_mmap_cc_hash_mask_exp_1div2_nsth_set						
cc_hash_table	Comb_Hash_Fn	direct_mask	range_hashing			
	Resize_Policy	hash_standard	Size_Policy	hash_exponential_size_policy		
			Trigger_Policy	hash_load_check_resize_trigger with $\alpha_{\min} = 1/8$ and $\alpha_{\max} = 1/2$		
	Mapped	cc_hash_table	Comb_Hash_Fn	direct_mask	range_hashing	
			Resize_Policy	hash_standard	Size_Policy	hash_exponential

<i>Name/Instantiating Type</i>	<i>Parameter</i>	<i>Details</i>	<i>Parameter</i>	<i>Details</i>	<i>Parameter</i>	<i>Details</i>
					Trigger_Policy	hash_load_check_1 with $\alpha_{\min} = 1/8$ and $\alpha_{\max} = 1/2$

21.4.2.3.3.3 Observations

See Observations::Mapping-Semantics Considerations.

21.4.2.3.4 Text insert with Small Secondary-to-Primary Key Ratios

21.4.2.3.4.1 Description

This test inserts a number of pairs into a container. The first item of each pair is a string from an arbitrary text ([wickland96thirty]), and the second is a uniform integer. The container is a "multimap" - it considers the first member of each pair as a primary key, and the second member of each pair as a secondary key. There are 400 distinct primary keys, and the ratio of secondary keys to primary keys ranges from 1 to 5.

The test measures the average insert-time as a function of the number of values inserted. For this library's containers, it inserts a primary key into the primary associative container, then a secondary key into the secondary associative container. For the native multimaps, it obtains a range using `std::equal_range`, and inserts a value only if it was not contained already.

It uses the test file: `performance/ext/pb_ds/multimap_text_insert_timing_large.cc`

The test checks the insert-time scalability of different "multimap" designs.

21.4.2.3.4.2 Results

The graphic below show the results for "multimaps" which use a tree-based container for primary keys.



The abbreviated names in the legend of the graphic above are instantiated with the types in the following table.

Name/Instantiating Type	Parameter	Details	Parameter	Details	Parameter	Details
n_hash_mmap						
std::tr1::unordered_multimap						
rb_tree_mmap_lu_mtf_set						
cc_hash_table	Comb_Hash_Fn	direct_mask	range_hashing			
	Resize_Policy	hash_standard_resize_policy	hash_exponential_size_policy			
			Trigger_Policy	hash_load_check_resize_trigger with $\alpha_{\min} = 1/8$ and $\alpha_{\max} = 1/2$		
	Mapped	list_update	Update_Policy	yu_move_to_front_policy		
rb_tree_mmap_cc_hash_mask_exp_1div2_nsth_set						
cc_hash_table	Comb_Hash_Fn	direct_mask	range_hashing			
	Resize_Policy	hash_standard_resize_policy	hash_exponential_size_policy			
			Trigger_Policy	hash_load_check_resize_trigger with $\alpha_{\min} = 1/8$ and $\alpha_{\max} = 1/2$		
	Mapped	cc_hash_table	Comb_Hash_Fn	direct_mask	range_hashing	
			Resize_Policy	hash_standard_resize_policy	hash_exponential_size_policy	

<i>Name/Instantiating Type</i>	<i>Parameter</i>	<i>Details</i>	<i>Parameter</i>	<i>Details</i>	<i>Parameter</i>	<i>Details</i>
					Trigger_Policy	hash_load_check_1 with $\alpha_{\min} = 1/8$ and $\alpha_{\max} = 1/2$

21.4.2.3.4.3 Observations

See Observations::Mapping-Semantics Considerations.

21.4.2.3.5 Text insert with Small Secondary-to-Primary Key Ratios Memory Use

21.4.2.3.5.1 Description

This test inserts a number of pairs into a container. The first item of each pair is a string from an arbitrary text ([wickland96thirty]), and the second is a uniform integer. The container is a "multimap" - it considers the first member of each pair as a primary key, and the second member of each pair as a secondary key. There are 100 distinct primary keys, and the ratio of secondary keys to primary keys ranges to about 20.

The test measures the memory use as a function of the number of values inserted.

It uses the test file: performance/ext/pb_ds/multimap_text_insert_mem_usage_small.cc

The test checks the memory scalability of different "multimap" designs.

21.4.2.3.5.2 Results

The graphic below show the results for "multimaps" which use a tree-based container for primary keys.

<i>Name/Instantiating Type</i>	<i>Parameter</i>	<i>Details</i>	<i>Parameter</i>	<i>Details</i>	<i>Parameter</i>	<i>Details</i>
					Trigger_Policy	hash_load_check_1 with $\alpha_{\min} = 1/8$ and $\alpha_{\max} = 1/2$

21.4.2.3.5.3 Observations

See Observations::Mapping-Semantics Considerations.

21.4.2.3.6 Text insert with Small Secondary-to-Primary Key Ratios Memory Use

21.4.2.3.6.1 Description

This test inserts a number of pairs into a container. The first item of each pair is a string from an arbitrary text ([wickland96thirty]), and the second is a uniform integer. The container is a "multimap" - it considers the first member of each pair as a primary key, and the second member of each pair as a secondary key. There are 100 distinct primary keys, and the ratio of secondary keys to primary keys ranges to about 20.

The test measures the memory use as a function of the number of values inserted.

It uses the test file: performance/ext/pb_ds/multimap_text_insert_mem_usage_large.cc

The test checks the memory scalability of different "multimap" designs.

21.4.2.3.6.2 Results

The graphic below show the results for "multimaps" which use a tree-based container for primary keys.

<i>Name/Instantiating Type</i>	<i>Parameter</i>	<i>Details</i>
pairing_heap		
priority_queue	Tag	pairing_heap_tag

21.4.2.4.1.3 Observations

Pairing heaps (`priority_queue` with `Tag = pairing_heap_tag`) are the most suited for sequences of `push` and `pop` operations of non-primitive types (e.g. `std::strings`). (See [Priority Queue Text push and pop Timing Test](#).) They are less constrained than binomial heaps, e.g., and since they are node-based, they outperform binary heaps. (See [Priority Queue Random Integer push Timing Test](#) for the case of primitive types.)

The standard's priority queues do not seem to perform well in this case: the `std::vector` implementation needs to perform a logarithmic sequence of string operations for each operation, and the deque implementation is possibly hampered by its need to manipulate a relatively-complex type (deques support a $O(1)$ `push_front`, even though it is not used by `std::priority_queue`.)

21.4.2.4.2 Text push and pop

21.4.2.4.2.1 Description

This test inserts a number of values with keys from an arbitrary text ([`wickland96thirty`]) into a container using `push`, then removes them using `pop`. It measures the average time for `push` as a function of the number of values.

It uses the test file: `performance/ext/pb_ds/priority_queue_text_push_pop_timing.cc`

The test checks the effect of different underlying data structures.

21.4.2.4.2.2 Results

The two graphics below show the results for the native `priority_queues` and this library's `priority_queues`.

The graphic immediately below shows the results for the native `priority_queue` type instantiated with different underlying container types versus several different versions of library's `priority_queues`.



The abbreviated names in the legend of the graphic above are instantiated with the types in the following table.

<i>Name/Instantiating Type</i>	<i>Parameter</i>	<i>Details</i>
n_pq_vector		
std::priority_queue	Sequence	std::vector
n_pq_deque		
std::priority_queue	Sequence	std::deque
binary_heap		
priority_queue	Tag	binary_heap_tag
binomial_heap		
priority_queue	Tag	binomial_heap_tag
rc_binomial_heap		
priority_queue	Tag	rc_binomial_heap_tag
thin_heap		
priority_queue	Tag	thin_heap_tag
pairing_heap		
priority_queue	Tag	pairing_heap_tag

The graphic below shows the results for the native priority queues and this library's pairing-heap priority_queue data structures.

<i>Name/Instantiating Type</i>	<i>Parameter</i>	<i>Details</i>
<code>priority_queue</code>	Tag	<code>binary_heap_tag</code>
<code>binomial_heap</code>		
<code>priority_queue</code>	Tag	<code>binomial_heap_tag</code>
<code>rc_binomial_heap</code>		
<code>priority_queue</code>	Tag	<code>rc_binomial_heap_tag</code>
<code>thin_heap</code>		
<code>priority_queue</code>	Tag	<code>thin_heap_tag</code>
<code>pairing_heap</code>		
<code>priority_queue</code>	Tag	<code>pairing_heap_tag</code>

21.4.2.4.5.3 Observations

The priority queue implementations (excluding the standard's) use memory proportionally to the number of values they hold: node-based implementations (e.g., a pairing heap) do so naturally; this library's binary heap de-allocates memory when a certain lower threshold is exceeded.

Note from Priority Queue Text `push` and `pop` Timing Test and Priority Queue Random Integer `push` and `pop` Timing Test that this does not impede performance compared to the standard's priority queues.

See Hash-Based Erase Memory Use Test for a similar phenomenon regarding priority queues.

21.4.2.4.6 Text `join`

21.4.2.4.6.1 Description

This test inserts a number of values with keys from an arbitrary text ([wickland96thirty]) into two containers, then merges the containers. It uses `join` for this library's priority queues; for the standard's priority queues, it successively pops values from one container and pushes them into the other. The test measures the average time as a function of the number of values.

It uses the test file: `performance/ext/pb_ds/priority_queue_text_join_timing.cc`

The test checks the effect of different underlying data structures.

21.4.2.4.6.2 Results

The graphic immediately below shows the results for the native `priority_queue` type instantiated with different underlying container types versus several different versions of library's `priority_queues`.

21.4.2.4.7 Text modify Up

21.4.2.4.7.1 Description

This test inserts a number of values with keys from an arbitrary text ([wickland96thirty]) into into a container then modifies each one "up" (i.e., it makes it larger). It uses `modify` for this library's priority queues; for the standard's priority queues, it pops values from a container until it reaches the value that should be modified, then pushes values back in. It measures the average time for `modify` as a function of the number of values.

It uses the test file: `performance/ext/pb_ds/priority_queue_text_modify_up_timing.cc`

The test checks the effect of different underlying data structures for graph algorithms settings. Note that making an arbitrary value larger (in the sense of the priority queue's comparison functor) corresponds to decrease-key in standard graph algorithms [clrs2001].

21.4.2.4.7.2 Results

The two graphics below show the results for the native `priority_queues` and this library's `priority_queues`.

The graphic immediately below shows the results for the native `priority_queue` type instantiated with different underlying container types versus several different versions of library's `priority_queues`.



The abbreviated names in the legend of the graphic above are instantiated with the types in the following table.

<i>Name/Instantiating Type</i>	<i>Parameter</i>	<i>Details</i>
n_pq_vector		
std::priority_queue	Sequence	std::vector
n_pq_deque		

perform an `erase` operation followed by a `push` operation. As the other tests show, a pairing heap is usually far more efficient than a thin heap, so this is not surprising.

Most algorithms that involve priority queues increase values (in the sense of the priority queue's comparison functor), and so Priority Queue Text `modify` Up Timing Test - is more interesting than this test.

21.4.2.5 Observations

21.4.2.5.1 Associative

21.4.2.5.1.1 Underlying Data-Structure Families

In general, hash-based containers have better timing performance than containers based on different underlying-data structures. The main reason to choose a tree-based or trie-based container is if a byproduct of the tree-like structure is required: either order-preservation, or the ability to utilize node invariants. If memory-use is the major factor, an ordered-vector tree gives optimal results (albeit with high modification costs), and a list-based container gives reasonable results.

21.4.2.5.1.2 Hash-Based Containers

Hash-based containers are typically either collision chaining or probing. Collision-chaining containers are more flexible internally, and so offer better timing performance. Probing containers, if used for simple value-types, manage memory more efficiently (they perform far fewer allocation-related calls). In general, therefore, a collision-chaining table should be used. A probing container, conversely, might be used efficiently for operations such as eliminating duplicates in a sequence, or counting the number of occurrences within a sequence. Probing containers might be more useful also in multithreaded applications where each thread manipulates a hash-based container: in the standard, allocators have class-wise semantics (see [meyers96more] - Item 10); a probing container might incur less contention in this case.

21.4.2.5.1.3 Hash Policies

In hash-based containers, the range-hashing scheme seems to affect performance more than other considerations. In most settings, a mask-based scheme works well (or can be made to work well). If the key-distribution can be estimated a-priori, a simple hash function can produce nearly uniform hash-value distribution. In many other cases (e.g., text hashing, floating-point hashing), the hash function is powerful enough to generate hash values with good uniformity properties [knuth98sorting]; a modulo-based scheme, taking into account all bits of the hash value, appears to overlap the hash function in its effort.

The range-hashing scheme determines many of the other policies. A mask-based scheme works well with an exponential-size policy; for probing-based containers, it goes well with a linear-probe function.

An orthogonal consideration is the trigger policy. This presents difficult tradeoffs. E.g., different load factors in a load-check trigger policy yield a space/amortized-cost tradeoff.

21.4.2.5.1.4 Branch-Based Containers

In general, there are several families of tree-based underlying data structures: balanced node-based trees (e.g., red-black or AVL trees), high-probability balanced node-based trees (e.g., random treaps or skip-lists), competitive node-based trees (e.g., splay trees), vector-based "trees", and tries. (Additionally, there are disk-residing or network-residing trees, such as B-Trees and their numerous variants. An interface for this would have to deal with the execution model and ACID guarantees; this is out of the scope of this library.) Following are some observations on their application to different settings.

Of the balanced node-based trees, this library includes a red-black tree, as does standard (in practice). This type of tree is the "workhorse" of tree-based containers: it offers both reasonable modification and reasonable lookup time. Unfortunately, this data structure stores a huge amount of metadata. Each node must contain, besides a value, three pointers and a boolean. This type might be avoided if space is at a premium [austern00noset].

High-probability balanced node-based trees suffer the drawbacks of deterministic balanced trees. Although they are fascinating data structures, preliminary tests with them showed their performance was worse than red-black trees. The library does not contain any such trees, therefore.

Competitive node-based trees have two drawbacks. They are usually somewhat unbalanced, and they perform a large number of comparisons. Balanced trees perform one comparison per each node they encounter on a search path; a splay tree performs two comparisons. If the keys are complex objects, e.g., `std::string`, this can increase the running time. Conversely, such trees do well when there is much locality of reference. It is difficult to determine in which case to prefer such trees over balanced trees. This library includes a splay tree.

Ordered-vector trees use very little space [austern00noset]. They do not have any other advantages (at least in this implementation).

Large-fan-out PATRICIA tries have excellent lookup performance, but they do so through maintaining, for each node, a miniature "hash-table". Their space efficiency is low, and their modification performance is bad. These tries might be used for semi-static settings, where order preservation is important. Alternatively, red-black trees cross-referenced with hash tables can be used. [okasaki98mereable] discusses small-fan-out PATRICIA tries for integers, but the cited results seem to indicate that the amortized cost of maintaining such trees is higher than that of balanced trees. Moderate-fan-out trees might be useful for sequences where each element has a limited number of choices, e.g., DNA strings.

21.4.2.5.1.5 Mapping-Semantics

Different mapping semantics were discussed in the introduction and design sections. Here the focus will be on the case where a keys can be composed into primary keys and secondary keys. (In the case where some keys are completely identical, it is trivial that one should use an associative container mapping values to size types.) In this case there are (at least) five possibilities:

1. Use an associative container that allows equivalent-key values (such as `std::multimap`)
2. Use a unique-key value associative container that maps each primary key to some complex associative container of secondary keys, say a tree-based or hash-based container.
3. Use a unique-key value associative container that maps each primary key to some simple associative container of secondary keys, say a list-based container.
4. Use a unique-key value associative container that maps each primary key to some non-associative container (e.g., `std::vector`)
5. Use a unique-key value associative container that takes into account both primary and secondary keys.

Stated simply: there is a simple answer for this. (Excluding option 1, which should be avoided in all cases).

If the expected ratio of secondary keys to primary keys is small, then 3 and 4 seem reasonable. Both types of secondary containers are relatively lightweight (in terms of memory use and construction time), and so creating an entire container object for each primary key is not too expensive. Option 4 might be preferable to option 3 if changing the secondary key of some primary key is frequent - one cannot modify an associative container's key, and the only possibility, therefore, is erasing the secondary key and inserting another one instead; a non-associative container, conversely, can support in-place modification. The actual cost of erasing a secondary key and inserting another one depends also on the allocator used for secondary associative-containers (The tests above used the standard allocator, but in practice one might choose to use, e.g., [boost_pool]). Option 2 is definitely an overkill in this case. Option 1 loses out either immediately (when there is one secondary key per primary key) or almost immediately after that. Option 5 has the same drawbacks as option 2, but it has the additional drawback that finding all values whose primary key is equivalent to some key, might be linear in the total number of values stored (for example, if using a hash-based container).

If the expected ratio of secondary keys to primary keys is large, then the answer is more complicated. It depends on the distribution of secondary keys to primary keys, the distribution of accesses according to primary keys, and the types of operations most frequent.

To be more precise, assume there are m primary keys, primary key i is mapped to n_i secondary keys, and each primary key is mapped, on average, to n secondary keys (i.e., $E(n_i) = n$).

Suppose one wants to find a specific pair of primary and secondary keys. Using 1 with a tree based container (`std::multimap`), the expected cost is $E(\Theta(\log(m) + n_i)) = \Theta(\log(m) + n)$; using 1 with a hash-based container (`std::tr1::unordered_multimap`), the expected cost is $\Theta(n)$. Using 2 with a primary hash-based container and secondary hash-based containers, the expected cost is $O(1)$; using 2 with a primary tree-based container and secondary tree-based containers, the expected cost is (using the Jensen

21.4.2.5.2.2 Amortized push and pop operations

In many cases, a priority queue is needed primarily for sequences of `push` and `pop` operations. All of the underlying data structures have the same amortized logarithmic complexity, but they differ in terms of constants.

The table above shows that the different data structures are "constrained" in some respects. In general, if a data structure has lower worst-case complexity than another, then it will perform slower in the amortized sense. Thus, for example a redundant-counter binomial heap (`priority_queue` with `Tag = rc_binomial_heap_tag`) has lower worst-case `push` performance than a binomial heap (`priority_queue` with `Tag = binomial_heap_tag`), and so its amortized `push` performance is slower in terms of constants.

As the table shows, the "least constrained" underlying data structures are binary heaps and pairing heaps. Consequently, it is not surprising that they perform best in terms of amortized constants.

1. Pairing heaps seem to perform best for non-primitive types (e.g., `std::strings`), as shown by Priority Queue Text `push` Timing Test and Priority Queue Text `push` and `pop` Timing Test
2. binary heaps seem to perform best for primitive types (e.g., `ints`), as shown by Priority Queue Random Integer `push` Timing Test and Priority Queue Random Integer `push` and `pop` Timing Test.

21.4.2.5.2.3 Graph Algorithms

In some graph algorithms, a decrease-key operation is required [clrs2001]; this operation is identical to `modify` if a value is increased (in the sense of the priority queue's comparison functor). The table above and Priority Queue Text `modify` Up Timing Test show that a thin heap (`priority_queue` with `Tag = thin_heap_tag`) outperforms a pairing heap (`priority_queue` with `Tag = Tag = pairing_heap_tag`), but the rest of the tests show otherwise.

This makes it difficult to decide which implementation to use in this case. Dijkstra's shortest-path algorithm, for example, requires $\Theta(n)$ `push` and `pop` operations (in the number of vertices), but $O(n^2)$ `modify` operations, which can be in practice $\Theta(n)$ as well. It is difficult to find an a-priori characterization of graphs in which the actual number of `modify` operations will dwarf the number of `push` and `pop` operations.

21.5 Acknowledgments

Written by Ami Tavory and Vladimir Dreizin (IBM Haifa Research Laboratories), and Benjamin Kosnik (Red Hat).

This library was partially written at IBM's Haifa Research Labs. It is based heavily on policy-based design and uses many useful techniques from Modern C++ Design: Generic Programming and Design Patterns Applied by Andrei Alexandrescu.

Two ideas are borrowed from the SGI-STL implementation:

1. The prime-based resize policies use a list of primes taken from the SGI-STL implementation.
2. The red-black trees contain both a root node and a header node (containing metadata), connected in a way that forward and reverse iteration can be performed efficiently.

Some test utilities borrow ideas from `boost::timer`.

We would like to thank Scott Meyers for useful comments (without attributing to him any flaws in the design or implementation of the library).

We would like to thank Matt Austern for the suggestion to include tries.

21.6 Bibliography

- [55] Dave Abrahams , *STL Exception Handling Contract* , 1997, ISO SC22/WG21 .
 - [56] Andrei Alexandrescu , *Modern C++ Design: Generic Programming and Design Patterns Applied* , 2001 , Addison-Wesley Publishing Company .
 - [57] K. Andrew and D. Gleich , *MTF, Bit, and COMB: A Guide to Deterministic and Randomized Algorithms for the List Update Problem*
 - [58] Matthew Austern , *Why You Shouldn't Use set - and What You Should Use Instead* , April, 2000 , C++ Report .
 - [59] Matthew Austern , *A Proposal to Add Hashtables to the Standard Library* , 2001 , ISO SC22/WG21 .
 - [60] Matthew Austern , *Segmented iterators and hierarchical algorithms* , April, 1998 , Generic Programming .
 - [61] Beeman Dawes , *Boost Timer Library* , Boost .
 - [62] Stephen Cleary , *Boost Pool Library* , Boost .
 - [63] Maddock John and Stephen Cleary , *Boost Type Traits Library* , Boost .
 - [64] Gerth Stolting Brodal , *Worst-case efficient priority queues*
 - [65] D. Bulka and D. Mayhew , *Efficient C++ Programming Techniques* , 1997 , Addison-Wesley Publishing Company .
 - [66] T. H. Cormen , C. E. Leiserson , R. L. Rivest , and C. Stein , *Introduction to Algorithms, 2nd edition* , 2001 , MIT Press .
 - [67] D. Dubashi and D. Ranjan , *Balls and bins: A study in negative dependence* , 1998 , Random Structures and Algorithms 13 .
 - [68] R. Fagin , J. Nievergelt , N. Pippenger , and H. R. Strong , *Extendible hashing - a fast access method for dynamic files* , 1979 , ACM Trans. Database Syst. 4 .
 - [69] Jean-Christophe Filliatre , *Ptset: Sets of integers implemented as Patricia trees* , 2000 .
 - [70] M. L. Fredman , R. Sedgewick , D. D. Sleator , and R. E. Tarjan , *The pairing heap: a new form of self-adjusting heap* , 1986 .
 - [71] E. Gamma , R. Helm , R. Johnson , and J. Vlissides , *Design Patterns - Elements of Reusable Object-Oriented Software* , 1995 , Addison-Wesley Publishing Company .
 - [72] A. K. Garg and C. C. Gotlieb , *Order-preserving key transformations* , 1986 , Trans. Database Syst. 11 .
 - [73] J. Hyslop and Herb Sutter , *Making a real hash of things* , May 2002 , C++ Report .
 - [74] N. M. Jossutis , *The C++ Standard Library - A Tutorial and Reference* , 2001 , Addison-Wesley Publishing Company .
 - [75] Haim Kaplan and Robert E. Tarjan , *New Heap Data Structures* , 1999 .
 - [76] Angelika Langer and Klaus Kleft , *Are Set Iterators Mutable or Immutable?* , October 2000 , C/C++ Users Journal .
 - [77] D. E. Knuth , *The Art of Computer Programming - Sorting and Searching* , 1998 , Addison-Wesley Publishing Company .
 - [78] B. Liskov , *Data abstraction and hierarchy* , May 1998 , SIGPLAN Notices 23 .
 - [79] W. Litwin , *Linear hashing: A new tool for file and table addressing* , June 1980 , Proceedings of International Conference on Very Large Data Bases .
 - [80] Maverick Woo , *Deamortization - Part 2: Binomial Heaps* , 2005 .
-

- [81] Scott Meyers , *More Effective C++: 35 New Ways to Improve Your Programs and Designs* , 1996 , Addison-Wesley Publishing Company .
 - [82] Scott Meyers , *How Non-Member Functions Improve Encapsulation* , 2000 , C/C++ Users Journal .
 - [83] Scott Meyers , *Effective STL: 50 Specific Ways to Improve Your Use of the Standard Template Library* , 2001 , Addison-Wesley Publishing Company .
 - [84] Scott Meyers , *Class Template, Member Template - or Both?* , 2003 , C/C++ Users Journal .
 - [85] R. Motwani and P. Raghavan , *Randomized Algorithms* , 2003 , Cambridge University Press .
 - [86] *The Component Object Model* , Microsoft .
 - [87] David R. Musser , *Rationale for Adding Hash Tables to the C++ Standard Template Library* , 1995 .
 - [88] David R. Musser and A. Saini , *STL Tutorial and Reference Guide* , 1996 , Addison-Wesley Publishing Company .
 - [89] Mark Nelson , *Priority Queues and the STL* , January 1996 , Dr. Dobbs Journal .
 - [90] C. Okasaki and A. Gill , *Fast mergeable integer maps* , September 1998 , In Workshop on ML .
 - [91] Matt Austern , *Standard Template Library Programmer's Guide* , SGI .
 - [92] *select*
 - [93] D. D. Sleator and R. E. Tarjan , *Amortized Efficiency of List Update Problems* , 1984 , ACM Symposium on Theory of Computing .
 - [94] D. D. Sleator and R. E. Tarjan , *Self-Adjusting Binary Search Trees* , 1985 , ACM Symposium on Theory of Computing .
 - [95] A. A. Stepanov and M. Lee , *The Standard Template Library* , 1984 .
 - [96] Bjarne Stroustrup , *The C++ Programming Language* , 1997 , Addison-Wesley Publishing Company .
 - [97] D. Vandevor and N. M. Josuttis , *C++ Templates: The Complete Guide* , 2002 , Addison-Wesley Publishing Company .
 - [98] C. A. Wickland , *Thirty Years Among the Dead* , 1996 , National Psychological Institute .
-

Chapter 22

HP/SGI Extensions

22.1 Backwards Compatibility

A few extensions and nods to backwards-compatibility have been made with containers. Those dealing with older SGI-style allocators are dealt with elsewhere. The remaining ones all deal with bits:

The old pre-standard `bit_vector` class is present for backwards compatibility. It is simply a typedef for the `vector<bool>` specialization.

The `bitset` class has a number of extensions, described in the rest of this item. First, we'll mention that this implementation of `bitset<N>` is specialized for cases where N number of bits will fit into a single word of storage. If your choice of N is within that range (≤ 32 on i686-pc-linux-gnu, for example), then all of the operations will be faster.

There are versions of single-bit test, set, reset, and flip member functions which do no range-checking. If we call them member functions of an instantiation of `bitset<N>`, then their names and signatures are:

```
bitset<N>&    _Unchecked_set    (size_t pos);
bitset<N>&    _Unchecked_set    (size_t pos, int val);
bitset<N>&    _Unchecked_reset  (size_t pos);
bitset<N>&    _Unchecked_flip   (size_t pos);
bool         _Unchecked_test   (size_t pos);
```

Note that these may in fact be removed in the future, although we have no present plans to do so (and there doesn't seem to be any immediate reason to).

The member function `operator[]` on a const `bitset` returns a `bool`, and for a non-const `bitset` returns a reference (a nested type). No range-checking is done on the index argument, in keeping with other containers' `operator[]` requirements.

Finally, two additional searching functions have been added. They return the index of the first "on" bit, and the index of the first "on" bit that is after `prev`, respectively:

```
size_t _Find_first() const;
size_t _Find_next (size_t prev) const;
```

The same caveat given for the `_Unchecked_*` functions applies here also.

22.2 Deprecated

The SGI hashing classes `hash_set` and `hash_multiset` have been deprecated by the `unordered_set`, `unordered_multiset`, `unordered_map`, `unordered_multimap` containers in TR1 and C++11, and may be removed in future releases.

The SGI headers

```
<hash_map>
<hash_set>
<rope>
<slist>
<rb_tree>
```

are all here; `<backwards/hash_map>` and `<backwards/hash_set>` are deprecated but available as backwards-compatible extensions, as discussed further below. `<ext/rope>` is the SGI specialization for large strings ("rope," "large strings," get it? Love that geeky humor.) `<ext/slist>` (superseded in C++11 by `<forward_list>`) is a singly-linked list, for when the doubly-linked `list<>` is too much space overhead, and `<ext/rb_tree>` exposes the red-black tree classes used in the implementation of the standard maps and sets.

Each of the associative containers map, multimap, set, and multiset have a counterpart which uses a **hashing function** to do the arranging, instead of a strict weak ordering function. The classes take as one of their template parameters a function object that will return the hash value; by default, an instantiation of **hash**. You should specialize this functor for your class, or define your own, before trying to use one of the hashing classes.

The hashing classes support all the usual associative container functions, as well as some extra constructors specifying the number of buckets, etc.

Why would you want to use a hashing class instead of the “normal” implementations? Matt Austern writes:

[W]ith a well chosen hash function, hash tables generally provide much better average-case performance than binary search trees, and much worse worst-case performance. So if your implementation has hash_map, if you don't mind using nonstandard components, and if you aren't scared about the possibility of pathological cases, you'll probably get better performance from hash_map.

The deprecated hash tables are superseded by the standard unordered associative containers defined in the ISO C++ 2011 standard in the headers `<unordered_map>` and `<unordered_set>`.

Chapter 23

Utilities

The `<functional>` header contains many additional functors and helper functions, extending section 20.3. They are implemented in the file `stl_function.h`:

- `identity_element` for addition and multiplication.
- The functor `identity`, whose `operator()` returns the argument unchanged.
- Composition functors `unary_function` and `binary_function`, and their helpers `compose1` and `compose2`.
- `select1st` and `select2nd`, to strip pairs.
- `project1st` and `project2nd`.
- A set of functors/functions which always return the same result. They are `constant_void_fun`, `constant_binary_fun`, `constant_unary_fun`, `constant0`, `constant1`, and `constant2`.
- The class `subtractive_rng`.
- `mem_fun` adaptor helpers `mem_fun1` and `mem_fun1_ref` are provided for backwards compatibility.

20.4.1 can use several different allocators; they are described on the main extensions page.

20.4.3 is extended with a special version of `get_temporary_buffer` taking a second argument. The argument is a pointer, which is ignored, but can be used to specify the template type (instead of using explicit function template arguments like the standard version does). That is, in addition to

```
get_temporary_buffer<int>(5);
```

you can also use

```
get_temporary_buffer(5, (int*)0);
```

A class `temporary_buffer` is given in `stl_tempbuf.h`.

The specialized algorithms of section 20.4.4 are extended with `uninitialized_copy_n`.

Chapter 24

Algorithms

25.1.6 (`count`, `count_if`) is extended with two more versions of `count` and `count_if`. The standard versions return their results. The additional signatures return `void`, but take a final parameter by reference to which they assign their results, e.g.,

```
void count (first, last, value, n);
```

25.2 (mutating algorithms) is extended with two families of signatures, `random_sample` and `random_sample_n`.

25.2.1 (`copy`) is extended with

```
copy_n (_InputIter first, _Size count, _OutputIter result);
```

which copies the first 'count' elements at 'first' into 'result'.

25.3 (sorting 'n' heaps 'n' stuff) is extended with some helper predicates. Look in the doxygen-generated pages for notes on these.

- `is_heap` tests whether or not a range is a heap.
- `is_sorted` tests whether or not a range is sorted in nondescending order.

25.3.8 (`lexicographical_compare`) is extended with

```
lexicographical_compare_3way(_InputIter1 first1, _InputIter1 last1,  
                             _InputIter2 first2, _InputIter2 last2)
```

which does... what?

Chapter 25

Numerics

26.4, the generalized numeric operations such as `accumulate`, are extended with the following functions:

```
power (x, n);  
power (x, n, monoid_operation);
```

Returns, in FORTRAN syntax, "`x ** n`" where `n >= 0`. In the case of `n == 0`, returns the identity element for the monoid operation. The two-argument signature uses multiplication (for a true "power" implementation), but addition is supported as well. The operation functor must be associative.

The `iota` function wins the award for Extension With the Coolest Name (the name comes from Ken Iverson's APL language.) As described in the [SGI documentation](#), it "assigns sequentially increasing values to a range. That is, it assigns `value` to `*first`, `value + 1` to `*(first + 1)` and so on."

```
void iota(_ForwardIter first, _ForwardIter last, _Tp value);
```

The `iota` function is included in the ISO C++ 2011 standard.

Chapter 26

Iterators

24.3.2 describes `struct iterator`, which didn't exist in the original HP STL implementation (the language wasn't rich enough at the time). For backwards compatibility, base classes are provided which declare the same nested typedefs:

- `input_iterator`
- `output_iterator`
- `forward_iterator`
- `bidirectional_iterator`
- `random_access_iterator`

24.3.4 describes iterator operation `distance`, which takes two iterators and returns a result. It is extended by another signature which takes two iterators and a reference to a result. The result is modified, and the function returns nothing.

Chapter 27

Input and Output

Extensions allowing `filebufs` to be constructed from "C" types like `FILE*`s and file descriptors.

27.1 Derived filebufs

The v2 library included non-standard extensions to construct `std::filebufs` from C stdio types such as `FILE*`s and POSIX file descriptors. Today the recommended way to use stdio types with libstdc++ `IOStreams` is via the `stdio_filebuf` class (see below), but earlier releases provided slightly different mechanisms.

- 3.0.x `filebufs` have another ctor with this signature: `basic_filebuf(__c_file_type*, ios_base::openmode, int_type);` This comes in very handy in a number of places, such as attaching Unix sockets, pipes, and anything else which uses file descriptors, into the `IOStream` buffering classes. The three arguments are as follows:

- `__c_file_type* F` // the `__c_file_type` typedef usually boils down to stdio's `FILE`
- `ios_base::openmode M` // same as all the other uses of `openmode`
- `int_type B` // buffer size, defaults to `BUFSIZ` if not specified

For those wanting to use file descriptors instead of `FILE*`'s, I invite you to contemplate the mysteries of C's `fdopen()`.

- In library snapshot 3.0.95 and later, `filebufs` bring back an old extension: the `fd()` member function. The integer returned from this function can be used for whatever file descriptors can be used for on your platform. Naturally, the library cannot track what you do on your own with a file descriptor, so if you perform any I/O directly, don't expect the library to be aware of it.
- Beginning with 3.1, the extra `basic_filebuf` constructor and the `fd()` function were removed from the standard `filebuf`. Instead, `<ext/stdio_filebuf.h>` contains a derived class template called `__gnu_cxx::stdio_filebuf`. This class can be constructed from a C `FILE*` or a file descriptor, and provides the `fd()` function.

Chapter 28

Demangling

Transforming C++ ABI identifiers (like RTTI symbols) into the original C++ source identifiers is called “demangling.”

If you have read the [source documentation for namespace abi](#) then you are aware of the cross-vendor C++ ABI in use by GCC. One of the exposed functions is used for demangling, `abi::__cxx_demangle`.

In programs like **c++filt**, the linker, and other tools have the ability to decode C++ ABI names, and now so can you.

(The function itself might use different demanglers, but that’s the whole point of abstract interfaces. If we change the implementation, you won’t notice.)

Probably the only time you’ll be interested in demangling at runtime is when you’re seeing `typeid` strings in RTTI. For example:

```
#include <iostream>
#include <cstdlib>
#include <cxxabi.h>

struct empty { };

template <typename T, int N>
    struct bar { };

int main()
{
    int      status;
    char     *realname;

    // typeid
    bar<empty,17>      u;
    const std::type_info &ti = typeid(u);

    realname = abi::__cxx_demangle(ti.name(), NULL, NULL, &status);
    std::cout << ti.name() << "\t=> " << realname << "\t: " << status << '\n';
    std::free(realname);
}
```

This prints

```
3barI5emptyLi17EE      => bar<empty, 17>      : 0
```

The demangler interface is described in the source documentation linked to above. It is actually written in C, so you don’t need to be writing C++ in order to demangle C++. (That also means we have to use crummy memory management facilities, so don’t forget to `free()` the returned char array.)

Chapter 29

Concurrency

29.1 Design

29.1.1 Interface to Locks and Mutexes

The file `<ext/concurrency.h>` contains all the higher-level constructs for playing with threads. In contrast to the atomics layer, the concurrence layer consists largely of types. All types are defined within namespace `__gnu_cxx`.

These types can be used in a portable manner, regardless of the specific environment. They are carefully designed to provide optimum efficiency and speed, abstracting out underlying thread calls and accesses when compiling for single-threaded situations (even on hosts that support multiple threads.)

The enumerated type `_Lock_policy` details the set of available locking policies: `_S_single`, `_S_mutex`, and `_S_atomic`.

- `_S_single`
Indicates single-threaded code that does not need locking.
- `_S_mutex`
Indicates multi-threaded code using thread-layer abstractions.
- `_S_atomic`
Indicates multi-threaded code using atomic operations.

The compile-time constant `__default_lock_policy` is set to one of the three values above, depending on characteristics of the host environment and the current compilation flags.

Two more datatypes make up the rest of the interface: `__mutex`, and `__scoped_lock`.

The scoped lock idiom is well-discussed within the C++ community. This version takes a `__mutex` reference, and locks it during construction of `__scoped_lock` and unlocks it during destruction. This is an efficient way of locking critical sections, while retaining exception-safety. These types have been superseded in the ISO C++ 2011 standard by the `mutex` and `lock` types defined in the header `<mutex>`.

29.1.2 Interface to Atomic Functions

Two functions and one type form the base of atomic support.

The type `_Atomic_word` is a signed integral type supporting atomic operations.

The two functions functions are:

```

_Atomic_word
__exchange_and_add_dispatch(volatile _Atomic_word*, int);

void
__atomic_add_dispatch(volatile _Atomic_word*, int);

```

Both of these functions are declared in the header file `<ext/atomicity.h>`, and are in namespace `__gnu_cxx`.

- `__exchange_and_add_dispatch`
Adds the second argument's value to the first argument. Returns the old value.
- `__atomic_add_dispatch`
Adds the second argument's value to the first argument. Has no return value.

These functions forward to one of several specialized helper functions, depending on the circumstances. For instance,

`__exchange_and_add_dispatch`

Calls through to either of:

- `__exchange_and_add`
Multi-thread version. Inlined if compiler-generated builtin atomics can be used, otherwise resolved at link time to a non-builtin code sequence.
- `__exchange_and_add_single`
Single threaded version. Inlined.

However, only `__exchange_and_add_dispatch` and `__atomic_add_dispatch` should be used. These functions can be used in a portable manner, regardless of the specific environment. They are carefully designed to provide optimum efficiency and speed, abstracting out atomic accesses when they are not required (even on hosts that support compiler intrinsics for atomic operations.)

In addition, there are two macros

```

_GLIBCXX_READ_MEM_BARRIER
_GLIBCXX_WRITE_MEM_BARRIER

```

Which expand to the appropriate write and read barrier required by the host hardware and operating system.

29.2 Implementation

29.2.1 Using Built-in Atomic Functions

The functions for atomic operations described above are either implemented via compiler intrinsics (if the underlying host is capable) or by library fallbacks.

Compiler intrinsics (builtins) are always preferred. However, as the compiler builtins for atomics are not universally implemented, using them directly is problematic, and can result in undefined function calls.

Prior to GCC 4.7 the older `__sync` intrinsics were used. An example of an undefined symbol from the use of `__sync_fetch_and_add_4` on an unsupported host is a missing reference to `__sync_fetch_and_add_4`.

Current releases use the newer `__atomic` intrinsics, which are implemented by library calls if the hardware doesn't support them. Undefined references to functions like `__atomic_is_lock_free` should be resolved by linking to `libatomic`, which is usually installed alongside `libstdc++`.

 In addition, on some hosts the compiler intrinsics are enabled conditionally, via the `-march` command line flag. This makes the available intrinsics vary depending on the target hardware and the flags used during compile.

If builtins are possible for bool-sized integral types, `ATOMIC_BOOL_LOCK_FREE` will be defined. If builtins are possible for int-sized integral types, `ATOMIC_INT_LOCK_FREE` will be defined.

For the following hosts, intrinsics are enabled by default.

- alpha
- ia64
- powerpc
- s390

For others, some form of `-march` may work. On non-ancient x86 hardware, `-march=native` usually does the trick.

For hosts without compiler intrinsics, but with capable hardware, hand-crafted assembly is selected. This is the case for the following hosts:

- cris
- hppa
- i386
- i486
- m48k
- mips
- sparc

And for the rest, a simulated atomic lock via pthreads.

Detailed information about compiler intrinsics for atomic operations can be found in the GCC [documentation](#).

More details on the library fallbacks from the porting [section](#).

29.2.2 Thread Abstraction

A thin layer above IEEE 1003.1 (i.e. pthreads) is used to abstract the thread interface for GCC. This layer is called "gthread," and is comprised of one header file that wraps the host's default thread layer with a POSIX-like interface.

The file `<gthr-default.h>` points to the deduced wrapper for the current host. In libstdc++ implementation files, `<bits/gthr.h>` is used to select the proper gthreads file.

Within libstdc++ sources, all calls to underlying thread functionality use this layer. More detail as to the specific interface can be found in the source [documentation](#).

By design, the gthread layer is interoperable with the types, functions, and usage found in the usual `<pthread.h>` file, including `pthread_t`, `pthread_once_t`, `pthread_create`, etc.

29.3 Use

Typical usage of the last two constructs is demonstrated as follows:

```
#include <ext/concurrency.h>

namespace
{
    __gnu_cxx::__mutex safe_base_mutex;
} // anonymous namespace

namespace other
{
    void
    foo()
    {
        __gnu_cxx::__scoped_lock sentry(safe_base_mutex);
        for (int i = 0; i < max; ++i)
        {
            _Safe_iterator_base* __old = __iter;
            __iter = __iter-<_M_next;
            __old-<_M_detach_single();
        }
    }
}
```

In this sample code, an anonymous namespace is used to keep the `__mutex` private to the compilation unit, and `__scoped_lock` is used to guard access to the critical section within the for loop, locking the mutex on creation and freeing the mutex as control moves out of this block.

Several exception classes are used to keep track of concurrency-related errors. These classes are: `__concurrency_lock_error`, `__concurrency_unlock_error`, `__concurrency_wait_error`, and `__concurrency_broadcast_error`.

Part IV

Appendices

Appendix A

Contributing

The GNU C++ Library is part of GCC and follows the same development model, so the general rules for [contributing to GCC](#) apply. Active contributors are assigned maintainership responsibility, and given write access to the source repository. First-time contributors should follow this procedure:

A.1 Contributor Checklist

A.1.1 Reading

- Get and read the relevant sections of the C++ language specification. Copies of the full ISO 14882 standard are available on line via the ISO mirror site for committee members. Non-members, or those who have not paid for the privilege of sitting on the committee and sustained their two meeting commitment for voting rights, may get a copy of the standard from their respective national standards organization. In the USA, this national standards organization is [ANSI](#). (And if you've already registered with them you can [buy the standard on-line](#).)
- The library working group bugs, and known defects, can be obtained here: <https://www.open-std.org/jtc1/sc22/wg21>
- Peruse the [GNU Coding Standards](#), and chuckle when you hit the part about “Using Languages Other Than C”.
- Be familiar with the extensions that preceded these general GNU rules. These style issues for libstdc++ can be found in [Coding Style](#).
- And last but certainly not least, read the library-specific information found in [Porting and Maintenance](#).

A.1.2 Assignment

See the [legal prerequisites](#) for all GCC contributions.

Historically, the libstdc++ assignment form added the following question:

“ Which Belgian comic book character is better, Tintin or Asterix, and why? ”

While not strictly necessary, humoring the maintainers and answering this question would be appreciated.

Please contact Jonathan Wakely at jwakely+assign@redhat.com if you are confused about the assignment or have general licensing questions. When requesting an assignment form from assign@gnu.org, please CC the libstdc++ maintainer above so that progress can be monitored.

A.1.3 Getting Sources

[Getting write access \(look for "Write after approval"\)](#)

A.1.4 Submitting Patches

Every patch must have several pieces of information before it can be properly evaluated. Ideally (and to ensure the fastest possible response from the maintainers) it would have all of these pieces:

- A description of the bug and how your patch fixes this bug. For new features a description of the feature and your implementation.
- A ChangeLog entry as part of the Git commit message. Check some recent commits for format and content. The `contrib/mklog.py` script can be used to generate a ChangeLog template for commit messages. See [Read-write Git access](#) for scripts and aliases that are useful here.
- A testsuite submission or sample program that will easily and simply show the existing error or test new functionality.
- The patch itself. If you are using the Git repository use **git show** or **git format-patch** to produce a patch; otherwise, use **diff -cp OLD NEW**. If your version of diff does not support these options, then get the latest version of GNU diff.
- When you have all these pieces, bundle them up in a mail message and send it to `libstdc++@gcc.gnu.org`. All patches and related discussion should be sent to the libstdc++ mailing list. In common with the rest of GCC, patches should also be sent to the gcc-patches mailing list. So you could send your email To:libstdc++@gcc.gnu.org and Cc:gcc-patches@gcc.gnu.org for example.

A.2 Directory Layout and Source Conventions

The `libstdc++-v3` directory in the GCC sources contains the files needed to create the GNU C++ Library.

It has subdirectories:

doc Files in HTML and text format that document usage, quirks of the implementation, and contributor checklists.

include All header files for the C++ library are within this directory, modulo specific runtime-related files that are in the `libsupc++` directory.

include/std Files meant to be found by `#include <name>` directives in standard-conforming user programs.

include/c Headers intended to directly include standard C headers. [NB: this can be enabled via `--enable-headers=c`]

include/c_global Headers intended to include standard C headers in the global namespace, and put select names into the `std::` namespace. [NB: this is the default, and is the same as `--enable-headers=c_global`]

include/c_std Headers intended to include standard C headers already in namespace `std`, and put select names into the `std::` namespace. [NB: this is the same as `--enable-headers=c_std`]

include/bits Files included by standard headers and by other files in the `bits` directory.

include/backward Headers provided for backward compatibility, such as `<backward/hash_map>`. They are not used in this library.

include/ext Headers that define extensions to the standard library. No standard header refers to any of them, in theory (there are some exceptions).

include/debug, include/parallel, and Headers that implement the Debug Mode and Parallel Mode extensions.

scripts Scripts that are used during the configure, build, make, or test process.

src Files that are used in constructing the library, but are not installed.

src/c++98 Source files compiled using `-std=gnu++98`.

src/c++11 Source files compiled using `-std=gnu++11`.

src/filesystem Source files for the Filesystem TS.

src/shared Source code included by other files under both `src/c++98` and `src/c++11`

testsuites/[backward, demangle, ext, performance, thread, 17_* to 30_*] Test programs are here, and may be used to begin to exercise the library. Support for "make check" and "make check-install" is complete, and runs through all the subdirectories here when this command is issued from the build directory. Please note that "make check" requires DejaGnu 1.4 or later to be installed, or for extra **permutations** DejaGnu 1.5.3 or later.

Other subdirectories contain variant versions of certain files that are meant to be copied or linked by the configure script. Currently these are:

```
config/abi
config/allocator
config/cpu
config/io
config/locale
config/os
```

In addition, a subdirectory holds the convenience library `libsupc++`.

libsupc++ Contains the runtime library for C++, including exception handling and memory allocation and deallocation, RTTI, terminate handlers, etc.

Note that `glibc` also has a `bits/` subdirectory. We need to be careful not to collide with names in its `bits/` directory. For example `<bits/std_mutex.h>` has to be renamed from `<bits/mutex.h>`. Another solution would be to rename `bits` to (e.g.) `cppbits`.

In files throughout the system, lines marked with an "XXX" indicate a bug or incompletely-implemented feature. Lines marked "XXX MT" indicate a place that may require attention for multi-thread safety.

A.3 Coding Style

A.3.1 Bad Identifiers

Identifiers that conflict and should be avoided.

This is the list of names reserved to the implementation that have been claimed by certain compilers and system headers of interest, and should not be used in the library. It will grow, of course. We generally are interested in names that are not all-caps, except for those like `"_T"`

For Solaris:

```
_B
_C
_L
_N
_P
_S
_U
_X
_E1
..
_E24
```

Irix adds:

_A
_G

MS adds:

_T
__deref

BSD adds:

__used
__unused
__inline
_Complex
__istype
__maskrune
__tolower
__toupper
__wchar_t
__wint_t
_res
_res_ext
__tg_*

VxWorks adds:

_C2

For GCC:

[Note that this list is out of date. It applies to the old name-mangling; in G++ 3.0 and higher a different name-mangling is used. In addition, many of the bugs relating to G++ interpreting these names as operators have been fixed.]

The full set of __* identifiers (combined from gcc/cp/lex.c and gcc/cplus-dem.c) that are either old or new, but are definitely recognized by the demangler, is:

__aa
__aad
__ad
__addr
__adv
__aer
__als
__alshift
__amd
__ami
__aml
__amu
__aor
__apl
__array
__ars
__arshift
__as
__bit_and
__bit_ior
__bit_not

__bit_xor
__call
__cl
__cm
__cn
__co
__component
__compound
__cond
__convert
__delete
__dl
__dv
__eq
__er
__ge
__gt
__indirect
__le
__ls
__lt
__max
__md
__method_call
__mi
__min
__minus
__ml
__mm
__mn
__mult
__mx
__ne
__negate
__new
__nop
__nt
__nw
__oo
__op
__or
__pl
__plus
__postdecrement
__postincrement
__pp
__pt
__rf
__rm
__rs
__sz
__trunc_div
__trunc_mod
__truth_andif
__truth_not
__truth_orif
__vc

```
__vd
__vn

SGI badnames:
__builtin_alloca
__builtin_fsqrt
__builtin_sqrt
__builtin_fabs
__builtin_dabs
__builtin_cast_f2i
__builtin_cast_i2f
__builtin_cast_d2ll
__builtin_cast_ll2d
__builtin_copy_dhi2i
__builtin_copy_i2dhi
__builtin_copy_dlo2i
__builtin_copy_i2dlo
__add_and_fetch
__sub_and_fetch
__or_and_fetch
__xor_and_fetch
__and_and_fetch
__nand_and_fetch
__mpy_and_fetch
__min_and_fetch
__max_and_fetch
__fetch_and_add
__fetch_and_sub
__fetch_and_or
__fetch_and_xor
__fetch_and_and
__fetch_and_nand
__fetch_and_mpy
__fetch_and_min
__fetch_and_max
__lock_test_and_set
__lock_release
__lock_acquire
__compare_and_swap
__synchronize
__high_multiply
__unix
__sgi
__linux__
__i386__
__i486__
__cplusplus
__embedded_cplusplus
// long double conversion members mangled as __opr
// http://gcc.gnu.org/ml/libstdc++/1999-q4/msg00060.html
__opr
```

A.3.2 By Example

This library is written to appropriate C++ coding standards. As such,

it is intended to precede the recommendations of the GNU Coding Standard, which can be referenced in full here:

<https://www.gnu.org/prep/standards/standards.html#Formatting>

The rest of this is also interesting reading, but skip the "Design Advice" part.

The GCC coding conventions are here, and are also useful:

<https://gcc.gnu.org/codingconventions.html>

In addition, because it doesn't seem to be stated explicitly anywhere else, there is an 80 column source limit.

ChangeLog entries for member functions should use the `classname::member function name` syntax as follows:

1999-04-15 Dennis Ritchie <dr@att.com>

```
* src/basic_file.cc (__basic_file::open): Fix thinko in
_G_HAVE_IO_FILE_OPEN bits.
```

Notable areas of divergence from what may be previous local practice (particularly for GNU C) include:

01. Pointers and references

```
char* p = "flop";
char& c = *p;
- NOT -
char *p = "flop"; // wrong
char &c = *p;      // wrong
```

Reason: In C++, definitions are mixed with executable code. Here, `p` is being initialized, not `*p`. This is near-universal practice among C++ programmers; it is normal for C hackers to switch spontaneously as they gain experience.

02. Operator names and parentheses

```
operator==(type)
- NOT -
operator == (type) // wrong
```

Reason: The `==` is part of the function name. Separating it makes the declaration look like an expression.

03. Function names and parentheses

```
void mangle()
- NOT -
void mangle () // wrong
```

Reason: no space before parentheses (except after a control-flow keyword) is near-universal practice for C++. It identifies the parentheses as the function-call operator or declarator, as opposed to an expression or other overloaded use of parentheses.

04. Template function indentation

```
template<typename T>
    void
    template_function(args)
    { }
- NOT -
template<class T>
void template_function(args) {};
```

Reason: In class definitions, without indentation whitespace is needed both above and below the declaration to distinguish it visually from other members. (Also, re: "typename" rather than "class".) T often could be int, which is not a class. ("class", here, is an anachronism.)

05. Template class indentation

```
template<typename _CharT, typename _Traits>
    class basic_ios : public ios_base
    {
    public:
        // Types:
    };
- NOT -
template<class _CharT, class _Traits>
class basic_ios : public ios_base
{
    public:
        // Types:
    };
- NOT -
template<class _CharT, class _Traits>
    class basic_ios : public ios_base
    {
    public:
        // Types:
    };
};
```

06. Enumerators

```
enum
{
    space = _ISspace,
    print = _ISprint,
    cntrl = _IScntrl
};
- NOT -
enum { space = _ISspace, print = _ISprint, cntrl = _IScntrl };
```

07. Member initialization lists

All one line, separate from class name.

```
gribble::gribble()
: _M_private_data(0), _M_more_stuff(0), _M_helper(0)
{ }
- NOT -
gribble::gribble() : _M_private_data(0), _M_more_stuff(0), _M_helper(0)
{ }
```

08. Try/Catch blocks

```
try
{
    //
}
catch (...)
{
    //
}
- NOT -
try {
    //
} catch (...) {
    //
}
```

09. Member functions declarations and definitions

Keywords such as `extern`, `static`, `export`, `explicit`, `inline`, etc go on the line above the function name. Thus

```
virtual int
foo()
- NOT -
virtual int foo()
```

Reason: GNU coding conventions dictate return types for functions are on a separate line than the function name and parameter list for definitions. For C++, where we have member functions that can be either inline definitions or declarations, keeping to this standard allows all member function names for a given class to be aligned to the same margin, increasing readability.

10. Invocation of member functions with "this->"

For non-uglified names, use `this->name` to call the function.

```
this->sync()
- NOT -
```

```
sync()
```

Reason: Koenig lookup.

11. Namespaces

```
namespace std
{
    blah blah blah;
} // namespace std
```

-NOT-

```
namespace std {
    blah blah blah;
} // namespace std
```

12. Spacing under protected and private in class declarations: space above, none below i.e.

```
public:
    int foo;
```

-NOT-

```
public:

    int foo;
```

13. Spacing WRT return statements. no extra spacing before returns, no parenthesis i.e.

```
}
return __ret;
```

-NOT-

```
}

return __ret;
```

-NOT-

```
}
return (__ret);
```

14. Location of global variables. All global variables of class type, whether in the "user visible" space (e.g., cin) or the implementation namespace, must be defined as a character array with the appropriate alignment and then later

re-initialized to the correct value.

This is due to startup issues on certain platforms, such as AIX. For more explanation and examples, see `src/globals.cc`. All such variables should be contained in that file, for simplicity.

15. Exception abstractions

Use the exception abstractions found in `functexcept.h`, which allow C++ programmers to use this library with `-fno-exceptions`. (Even if that is rarely advisable, it's a necessary evil for backwards compatibility.)

16. Exception error messages

All start with the name of the function where the exception is thrown, and then (optional) descriptive text is added. Example:

```
__throw_logic_error(__N("basic_string::_S_construct NULL not valid"));
```

Reason: The verbose terminate handler prints out `exception::what()`, as well as the `typeinfo` for the thrown exception. As this is the default terminate handler, by putting location info into the exception string, a very useful error message is printed out for uncaught exceptions. So useful, in fact, that non-programmers can give useful error messages, and programmers can intelligently speculate what went wrong without even using a debugger.

17. The doxygen style guide to comments is a separate document, see `index`.

The library currently has a mixture of GNU-C and modern C++ coding styles. The GNU C usages will be combed out gradually.

Name patterns:

For nonstandard names appearing in Standard headers, we are constrained to use names that begin with underscores. This is called "uglification". The convention is:

Local and argument names: `__[a-z].*`

Examples: `__count` `__ix` `__s1`

Type names and template formal-argument names: `__[A-Z][^_].*`

Examples: `_Helper` `_CharT` `_Nm`

Member data and function names: `__M_.*`

Examples: `__M_num_elements` `__M_initialize ()`

Static data and function members, constants, and enumerations: `__S_.*`

Examples: `__S_max_elements` `__S_default_value`

Don't use names in the same scope that differ only in the prefix,

e.g. `_S_top` and `_M_top`. See `BADNAMES` for a list of forbidden names. (The most tempting of these seem to be `"_T"` and `"_N"`.)

Names must never have `"__"` internally; it would confuse name unmanglers on some targets. Also, never use `"__[0-9]"`, same reason.

[BY EXAMPLE]

```
#ifndef _HEADER_
#define _HEADER_ 1

namespace std
{
    class gribble
    {
    public:
        gribble() throw();

        gribble(const gribble&);

        explicit
        gribble(int __howmany);

        gribble&
        operator=(const gribble&);

        virtual
        ~gribble() throw ();

        // Start with a capital letter, end with a period.
        inline void
        public_member(const char* __arg) const;

        // In-class function definitions should be restricted to one-liners.
        int
        one_line() { return 0 }

        int
        two_lines(const char* arg)
        { return strchr(arg, 'a'); }

        inline int
        three_lines(); // inline, but defined below.

        // Note indentation.
        template<typename _Formal_argument>
        void
        public_template() const throw();

        template<typename _Iterator>
        void
        other_template();

    private:
```

```

class _Helper;

int _M_private_data;
int _M_more_stuff;
_Helper* _M_helper;
int _M_private_function();

enum _Enum
{
    _S_one,
    _S_two
};

static void
_S_initialize_library();
};

// More-or-less-standard language features described by lack, not presence ↵
.
# ifndef _G_NO_LONGLONG
extern long long _G_global_with_a_good_long_name; // avoid globals!
# endif

// Avoid in-class inline definitions, define separately;
// likewise for member class definitions:
inline int
gribble::public_member() const
{ int __local = 0; return __local; }

class gribble::_Helper
{
    int _M_stuff;

    friend class gribble;
};

// Names beginning with "__": only for arguments and
//   local variables; never use "__" in a type name, or
//   within any name; never use "__[0-9]".

#endif /* _HEADER_ */

namespace std
{
    template<typename T> // notice: "typename", not "class", no space
        long_return_value_type<with_many, args>
        function_name(char* pointer, // "char *pointer" is wrong.
                      char* argument,
                      const Reference& ref)
        {
            // int a_local; /* wrong; see below. */
            if (test)
            {
                nested code
            }
        }
    }

```

```

    int a_local = 0;  // declare variable at first use.

    // char a, b, *p;  /* wrong */
    char a = 'a';
    char b = a + 1;
    char* c = "abc";  // each variable goes on its own line, always.

    // except maybe here...
    for (unsigned i = 0, mask = 1; mask; ++i, mask <= 1) {
        // ...
    }
}

gribble::gribble()
: _M_private_data(0), _M_more_stuff(0), _M_helper(0)
{ }

int
gribble::three_lines()
{
    // doesn't fit in one line.
}
} // namespace std

```

A.4 Design Notes

The Library

This paper covers two major areas:

- Features and policies not mentioned in the standard that the quality of the library implementation depends on, including extensions and "implementation-defined" features;
- Plans for required but unimplemented library features and optimizations to them.

Overhead

The standard defines a large library, much larger than the standard C library. A naive implementation would suffer substantial overhead in compile time, executable size, and speed, rendering it unusable in many (particularly embedded) applications. The alternative demands care in construction, and some compiler support, but there is no need for library subsets.

What are the sources of this overhead? There are four main causes:

- The library is specified almost entirely as templates, which with current compilers must be included in-line, resulting in

very slow builds as tens or hundreds of thousands of lines of function definitions are read for each user source file. Indeed, the entire SGI STL, as well as the dos Reis valarray, are provided purely as header files, largely for simplicity in porting. Iostream/locale is (or will be) as large again.

- The library is very flexible, specifying a multitude of hooks where users can insert their own code in place of defaults. When these hooks are not used, any time and code expended to support that flexibility is wasted.
- Templates are often described as causing to "code bloat". In practice, this refers (when it refers to anything real) to several independent processes. First, when a class template is manually instantiated in its entirety, current compilers place the definitions for all members in a single object file, so that a program linking to one member gets definitions of all. Second, template functions which do not actually depend on the template argument are, under current compilers, generated anew for each instantiation, rather than being shared with other instantiations. Third, some of the flexibility mentioned above comes from virtual functions (both in regular classes and template classes) which current linkers add to the executable file even when they manifestly cannot be called.
- The library is specified to use a language feature, exceptions, which in the current gcc compiler ABI imposes a run time and code space cost to handle the possibility of exceptions even when they are not used. Under the new ABI (accessed with `-fnew-abi`), there is a space overhead and a small reduction in code efficiency resulting from lost optimization opportunities associated with non-local branches associated with exceptions.

What can be done to eliminate this overhead? A variety of coding techniques, and compiler, linker and library improvements and extensions may be used, as covered below. Most are not difficult, and some are already implemented in varying degrees.

Overhead: Compilation Time

Providing "ready-instantiated" template code in object code archives allows us to avoid generating and optimizing template instantiations in each compilation unit which uses them. However, the number of such instantiations that are useful to provide is limited, and anyway this is not enough, by itself, to minimize compilation time. In particular, it does not reduce time spent parsing conforming headers.

Quicker header parsing will depend on library extensions and compiler improvements. One approach is some variation on the techniques previously marketed as "pre-compiled headers", now standardized as support for the "export" keyword. "Exported" template definitions can be placed (once) in a "repository" -- really just a library, but of template definitions rather than object code -- to be drawn upon at link time when an instantiation is needed, rather than placed in header files to be parsed along with every compilation unit.

Until "export" is implemented we can put some of the lengthy template

definitions in `#if` guards or alternative headers so that users can skip over the full definitions when they need only the ready-instantiated specializations.

To be precise, this means that certain headers which define templates which users normally use only for certain arguments can be instrumented to avoid exposing the template definitions to the compiler unless a macro is defined. For example, in `<string>`, we might have:

```
template <class _CharT, ... > class basic_string {
... // member declarations
};
... // operator declarations

#ifdef _STRICT_ISO_
# if _G_NO_TEMPLATE_EXPORT
#   include <bits/std_locale.h> // headers needed by definitions
#   ...
#   include <bits/string.tcc> // member and global template definitions.
# endif
#endif
```

Users who compile without specifying a strict-ISO-conforming flag would not see many of the template definitions they now see, and rely instead on ready-instantiated specializations in the library. This technique would be useful for the following substantial components: `string`, `locale/iostreams`, `valarray`. It would *not* be useful or usable with the following: `containers`, `algorithms`, `iterators`, `allocator`. Since these constitute a large (though decreasing) fraction of the library, the benefit the technique offers is limited.

The language specifies the semantics of the "export" keyword, but the gcc compiler does not yet support it. When it does, problems with large template inclusions can largely disappear, given some minor library reorganization, along with the need for the apparatus described above.

Overhead: Flexibility Cost

The library offers many places where users can specify operations to be performed by the library in place of defaults. Sometimes this seems to require that the library use a more-roundabout, and possibly slower, way to accomplish the default requirements than would be used otherwise.

The primary protection against this overhead is thorough compiler optimization, to crush out layers of inline function interfaces. Kuck & Associates has demonstrated the practicality of this kind of optimization.

The second line of defense against this overhead is explicit specialization. By defining helper function templates, and writing specialized code for the default case, overhead can be eliminated for that case without sacrificing flexibility. This takes full

advantage of any ability of the optimizer to crush out degenerate code.

The library specifies many virtual functions which current linkers load even when they cannot be called. Some minor improvements to the compiler and to ld would eliminate any such overhead by simply omitting virtual functions that the complete program does not call. A prototype of this work has already been done. For targets where GNU ld is not used, a "pre-linker" could do the same job.

The main areas in the standard interface where user flexibility can result in overhead are:

- Allocators: Containers are specified to use user-definable allocator types and objects, making tuning for the container characteristics tricky.
- Locales: the standard specifies locale objects used to implement iostream operations, involving many virtual functions which use streambuf iterators.
- Algorithms and containers: these may be instantiated on any type, frequently duplicating code for identical operations.
- Iostreams and strings: users are permitted to use these on their own types, and specify the operations the stream must use on these types.

Note that these sources of overhead are avoidable. The techniques to avoid them are covered below.

Code Bloat -----

In the SGI STL, and in some other headers, many of the templates are defined "inline" -- either explicitly or by their placement in class definitions -- which should not be inline. This is a source of code bloat. Matt had remarked that he was relying on the compiler to recognize what was too big to benefit from inlining, and generate it out-of-line automatically. However, this also can result in code bloat except where the linker can eliminate the extra copies.

Fixing these cases will require an audit of all inline functions defined in the library to determine which merit inlining, and moving the rest out of line. This is an issue mainly in clauses 23, 25, and 27. Of course it can be done incrementally, and we should generally accept patches that move large functions out of line and into ".tcc" files, which can later be pulled into a repository. Compiler/linker improvements to recognize very large inline functions and move them out-of-line, but shared among compilation units, could make this work unnecessary.

Pre-instantiating template specializations currently produces large amounts of dead code which bloats statically linked programs. The current state of the static library, libstdc++.a, is intolerable on this account, and will fuel further confused speculation about a need

for a library "subset". A compiler improvement that treats each instantiated function as a separate object file, for linking purposes, would be one solution to this problem. An alternative would be to split up the manual instantiation files into dozens upon dozens of little files, each compiled separately, but an abortive attempt at this was done for `<string>` and, though it is far from complete, it is already a nuisance. A better interim solution (just until we have "export") is badly needed.

When building a shared library, the current compiler/linker cannot automatically generate the instantiations needed. This creates a miserable situation; it means any time something is changed in the library, before a shared library can be built someone must manually copy the declarations of all templates that are needed by other parts of the library to an "instantiation" file, and add it to the build system to be compiled and linked to the library. This process is readily automated, and should be automated as soon as possible. Users building their own shared libraries experience identical frustrations.

Sharing common aspects of template definitions among instantiations can radically reduce code bloat. The compiler could help a great deal here by recognizing when a function depends on nothing about a template parameter, or only on its size, and giving the resulting function a link-name "equate" that allows it to be shared with other instantiations. Implementation code could take advantage of the capability by factoring out code that does not depend on the template argument into separate functions to be merged by the compiler.

Until such a compiler optimization is implemented, much can be done manually (if tediously) in this direction. One such optimization is to derive class templates from non-template classes, and move as much implementation as possible into the base class. Another is to partial-specialize certain common instantiations, such as `vector<T*>`, to share code for instantiations on all types `T`. While these techniques work, they are far from the complete solution that a compiler improvement would afford.

Overhead: Expensive Language Features

The main "expensive" language feature used in the standard library is exception support, which requires compiling in cleanup code with static table data to locate it, and linking in library code to use the table. For small embedded programs the amount of such library code and table data is assumed by some to be excessive. Under the "new" ABI this perception is generally exaggerated, although in some cases it may actually be excessive.

To implement a library which does not use exceptions directly is not difficult given minor compiler support (to "turn off" exceptions and ignore exception constructs), and results in no great library maintenance difficulties. To be precise, given `"-fno-exceptions"`, the compiler should treat "try" blocks as ordinary blocks, and "catch" blocks as dead code to ignore or eliminate. Compiler support is not strictly necessary, except in the case of "function try blocks"; otherwise the following macros almost suffice:

```
#define throw(X)
#define try      if (true)
#define catch(X) else if (false)
```

However, there may be a need to use function try blocks in the library implementation, and use of macros in this way can make correct diagnostics impossible. Furthermore, use of this scheme would require the library to call a function to re-throw exceptions from a try block. Implementing the above semantics in the compiler is preferable.

Given the support above (however implemented) it only remains to replace code that "throws" with a call to a well-documented "handler" function in a separate compilation unit which may be replaced by the user. The main source of exceptions that would be difficult for users to avoid is memory allocation failures, but users can define their own memory allocation primitives that never throw. Otherwise, the complete list of such handlers, and which library functions may call them, would be needed for users to be able to implement the necessary substitutes. (Fortunately, they have the source code.)

Opportunities

The template capabilities of C++ offer enormous opportunities for optimizing common library operations, well beyond what would be considered "eliminating overhead". In particular, many operations done in Glibc with macros that depend on proprietary language extensions can be implemented in pristine Standard C++. For example, the chapter 25 algorithms, and even C library functions such as `strchr`, can be specialized for the case of static arrays of known (small) size.

Detailed optimization opportunities are identified below where the component where they would appear is discussed. Of course new opportunities will be identified during implementation.

Unimplemented Required Library Features

The standard specifies hundreds of components, grouped broadly by chapter. These are listed in excruciating detail in the CHECKLIST file.

```
17 general
18 support
19 diagnostics
20 utilities
21 string
22 locale
23 containers
24 iterators
25 algorithms
26 numerics
27 iostreams
Annex D backward compatibility
```

Anyone participating in implementation of the library should obtain a copy of the standard, ISO 14882. People in the U.S. can obtain an electronic copy for US\$18 from ANSI's web site. Those from other countries should visit <http://www.iso.org/> to find out the location of their country's representation in ISO, in order to know who can sell them a copy.

The emphasis in the following sections is on unimplemented features and optimization opportunities.

Chapter 17 General

Chapter 17 concerns overall library requirements.

The standard doesn't mention threads. A multi-thread (MT) extension primarily affects operators new and delete (18), allocator (20), string (21), locale (22), and iostreams (27). The common underlying support needed for this is discussed under chapter 20.

The standard requirements on names from the C headers create a lot of work, mostly done. Names in the C headers must be visible in the `std::` and sometimes the global namespace; the names in the two scopes must refer to the same object. More stringent is that Koenig lookup implies that any types specified as defined in `std::` really are defined in `std::`. Names optionally implemented as macros in C cannot be macros in C++. (An overview may be read at <http://www.cantrip.org/cheaders.html>). The scripts "inclosure" and "mkcshadow", and the directories shadow/ and cshadow/, are the beginning of an effort to conform in this area.

A correct conforming definition of C header names based on underlying C library headers, and practical linking of conforming namespaced customer code with third-party C libraries depends ultimately on an ABI change, allowing namespaced C type names to be mangled into type names as if they were global, somewhat as C function names in a namespace, or C++ global variable names, are left unmangled. Perhaps another "extern" mode, such as 'extern "C-global"' would be an appropriate place for such type definitions. Such a type would affect mangling as follows:

```
namespace A {
struct X {};
extern "C-global" { // or maybe just 'extern "C"'
struct Y {};
};
}
void f(A::X*); // mangles to f__FPQ21A1X
void f(A::Y*); // mangles to f__FP1Y
```

(It may be that this is really the appropriate semantics for regular 'extern "C"', and 'extern "C-global"', as an extension, would not be necessary.) This would allow functions declared in non-standard C headers (and thus fixable by neither us nor users) to link properly with functions declared using C types defined in properly-namespaced headers. The problem this solves is that C headers (which C++ programmers do persist

in using) frequently forward-declare C struct tags without including the header where the type is defined, as in

```
struct tm;  
void munge(tm*);
```

Without some compiler accommodation, munge cannot be called by correct C++ code using a pointer to a correctly-scoped tm* value.

The current C headers use the preprocessor extension "#include_next", which the compiler complains about when run "-pedantic". (Incidentally, it appears that "-fpedantic" is currently ignored, probably a bug.) The solution in the C compiler is to use "-isystem" rather than "-I", but unfortunately in g++ this seems also to wrap the whole header in an 'extern "C"' block, so it's unusable for C++ headers. The correct solution appears to be to allow the various special include-directory options, if not given an argument, to affect subsequent include-directory options additively, so that if one said

```
-pedantic -iprefix $(prefix) \  
-idirafter -ino-pedantic -ino-extern-c -iwithprefix -I g++-v3 \  
-iwithprefix -I g++-v3/ext
```

the compiler would search \$(prefix)/g++-v3 and not report pedantic warnings for files found there, but treat files in \$(prefix)/g++-v3/ext pedantically. (The undocumented semantics of "-isystem" in g++ stink. Can they be rescinded? If not it must be replaced with something more rationally behaved.)

All the C headers need the treatment above; in the standard these headers are mentioned in various clauses. Below, I have only mentioned those that present interesting implementation issues.

The components identified as "mostly complete", below, have not been audited for conformance. In many cases where the library passes conformance tests we have non-conforming extensions that must be wrapped in #if guards for "pedantic" use, and in some cases renamed in a conforming way for continued use in the implementation regardless of conformance flags.

The STL portion of the library still depends on a header stl/bits/stl_config.h full of #ifdef clauses. This apparatus should be replaced with autoconf/automake machinery.

The SGI STL defines a type_traits<> template, specialized for many types in their code including the built-in numeric and pointer types and some library types, to direct optimizations of standard functions. The SGI compiler has been extended to generate specializations of this template automatically for user types, so that use of STL templates on user types can take advantage of these optimizations. Specializations for other, non-STL, types would make more optimizations possible, but extending the gcc compiler in the same way would be much better. Probably the next round of standardization will ratify this, but probably with changes, so it probably should be renamed to place it in the implementation namespace.

The SGI STL also defines a large number of extensions visible in standard headers. (Other extensions that appear in separate headers have been sequestered in subdirectories `ext/` and `backward/`.) All these extensions should be moved to other headers where possible, and in any case wrapped in a namespace (not `std!`), and (where kept in a standard header) girded about with macro guards. Some cannot be moved out of standard headers because they are used to implement standard features. The canonical method for accommodating these is to use a protected name, aliased in macro guards to a user-space name. Unfortunately C++ offers no satisfactory template typedef mechanism, so very ad-hoc and unsatisfactory aliasing must be used instead.

Implementation of a template typedef mechanism should have the highest priority among possible extensions, on the same level as implementation of the template "export" feature.

Chapter 18 Language support

Headers: `<limits>` `<new>` `<typeinfo>` `<exception>`
 C headers: `<cstdint>` `<climits>` `<float>` `<cstdlib>` `<setjmp>`
`<ctime>` `<csignal>` `<stdlib>` (also 21, 25, 26)

This defines the built-in exceptions, `rtti`, `numeric_limits<>`, operator `new` and `delete`. Much of this is provided by the compiler in its static runtime library.

Work to do includes defining `numeric_limits<>` specializations in separate files for all target architectures. Values for integer types except for `bool` and `wchar_t` are readily obtained from the C header `<limits.h>`, but values for the remaining numeric types (`bool`, `wchar_t`, `float`, `double`, `long double`) must be entered manually. This is largely dog work except for those members whose values are not easily deduced from available documentation. Also, this involves some work in target configuration to identify the correct choice of file to build against and to install.

The definitions of the various operators `new` and `delete` must be made thread-safe, which depends on a portable exclusion mechanism, discussed under chapter 20. Of course there is always plenty of room for improvements to the speed of operators `new` and `delete`.

`<cstdlib>`, in Glibc, defines some macros that gcc does not allow to be wrapped into an inline function. Probably this header will demand attention whenever a new target is chosen. The functions `atexit()`, `exit()`, and `abort()` in `cstdlib` have different semantics in C++, so must be re-implemented for C++.

Chapter 19 Diagnostics

Headers: `<stdexcept>`
 C headers: `<cassert>` `<cerrno>`

This defines the standard exception objects, which are "mostly complete".

```

<book>
<part>
<chapter>
<section>
</section>

<sect1>
</sect1>

<sect1>
<sect2>
</sect2>
</sect1>
</chapter>

<chapter>
</chapter>
</part>
</book>

</set>

```

B.2.4.6 Markup By Example

Complete details on Docbook markup can be found in the [DocBook Element Reference](#). An incomplete reference for HTML to Docbook conversion is detailed in the table below.

HTML	Docbook
<p>	<para>
<pre>	<computeroutput>, <programlisting>, <literallayout>
	<itemizedlist>
	<orderedlist>
	<listitem>
<dl>	<variablelist>
<dt>	<term>
<dd>	<listitem>
	<ulink url="">
<code>	<literal>, <programlisting>
	<emphasis>
	<emphasis>
"	<quote>

Table B.4: HTML to Docbook XML Markup Comparison

And examples of detailed markup for which there are no real HTML equivalents are listed in the table below.

B.3 Porting to New Hardware or Operating Systems

This document explains how to port libstdc++ (the GNU C++ library) to a new target.

In order to make the GNU C++ library (libstdc++) work with a new target, you must edit some configuration files and provide some new header files. Unless this is done, libstdc++ will use generic settings which may not be correct for your target; even if they are correct, they will likely be inefficient.


```
char
ctype<char>::do_tolower(char __c) const
{ return _tolower(__c); }
```

Your C library provides equivalents to IRIX's `_toupper` and `_tolower`. If you initialized `_M_toupper` and `_M_tolower` above, then you could use those tables instead.

Finally, you have to provide two utility functions that convert strings of characters. The versions provided here will always work - but you could use specialized routines for greater performance if you have machinery to do that on your system:

```
const char*
ctype<char>::do_toupper(char* __low, const char* __high) const
{
    while (__low < __high)
    {
        *__low = do_toupper(*__low);
        ++__low;
    }
    return __high;
}

const char*
ctype<char>::do_tolower(char* __low, const char* __high) const
{
    while (__low < __high)
    {
        *__low = do_tolower(*__low);
        ++__low;
    }
    return __high;
}
```

You must also provide the `ctype_inline.h` file, which contains a few more functions. On most systems, you can just copy `config/os/generic/ctype_inline.h` and use it on your system.

In detail, the functions provided test characters for particular properties; they are analogous to the functions like `isalpha` and `islower` provided by the C library.

The first function is implemented like this on IRIX:

```
bool
ctype<char>::
is(mask __m, char __c) const throw()
{ return (_M_table)[(unsigned char)(__c)] & __m; }
```

The `_M_table` is the table passed in above, in the constructor. This is the table that contains the bitmasks for each character. The implementation here should work on all systems.

The next function is:

```
const char*
ctype<char>::
is(const char* __low, const char* __high, mask* __vec) const throw()
{
    while (__low < __high)
        *__vec++ = (_M_table)[(unsigned char)(*__low++)];
    return __high;
}
```

This function is similar; it copies the masks for all the characters from `__low` up until `__high` into the vector given by `__vec`.

The last two functions again are entirely generic:

- GCC 4.7.0: GCC_4.7.0
- GCC 4.8.0: GCC_4.8.0
- GCC 7.1.0: GCC_7.0.0
- GCC 9.1.0: GCC_9.0.0
- GCC 11.1.0: GCC_11.0
- GCC 12.1.0: GCC_12.0.0
- GCC 13.1.0: GCC_13.0.0

3. Release versioning on the libstdc++.so binary, implemented in the same way as the libgcc_s.so binary above. Listed is the filename: DT_SONAME can be deduced from the filename by removing the last two period-delimited numbers. For example, filename libstdc++.so.5.0.4 corresponds to a DT_SONAME of libstdc++.so.5. Binaries with equivalent DT_SONAMES are forward-compatible: in the table below, releases incompatible with the previous one are explicitly noted. If a particular release is not listed, its libstdc++.so binary has the same filename and DT_SONAME as the preceding release.

It is versioned as follows:

- GCC 3.0.0: libstdc++.so.3.0.0
- GCC 3.0.1: libstdc++.so.3.0.1
- GCC 3.0.2: libstdc++.so.3.0.2
- GCC 3.0.3: libstdc++.so.3.0.2 (See Note 1)
- GCC 3.0.4: libstdc++.so.3.0.4
- GCC 3.1.0: libstdc++.so.4.0.0 (*Incompatible with previous*)
- GCC 3.1.1: libstdc++.so.4.0.1
- GCC 3.2.0: libstdc++.so.5.0.0 (*Incompatible with previous*)
- GCC 3.2.1: libstdc++.so.5.0.1
- GCC 3.2.2: libstdc++.so.5.0.2
- GCC 3.2.3: libstdc++.so.5.0.3 (See Note 2)
- GCC 3.3.0: libstdc++.so.5.0.4
- GCC 3.3.1: libstdc++.so.5.0.5
- GCC 3.4.0: libstdc++.so.6.0.0 (*Incompatible with previous*)
- GCC 3.4.1: libstdc++.so.6.0.1
- GCC 3.4.2: libstdc++.so.6.0.2
- GCC 3.4.3: libstdc++.so.6.0.3
- GCC 4.0.0: libstdc++.so.6.0.4
- GCC 4.0.1: libstdc++.so.6.0.5
- GCC 4.0.2: libstdc++.so.6.0.6
- GCC 4.0.3: libstdc++.so.6.0.7
- GCC 4.1.0: libstdc++.so.6.0.7
- GCC 4.1.1: libstdc++.so.6.0.8
- GCC 4.2.0: libstdc++.so.6.0.9
- GCC 4.2.1: libstdc++.so.6.0.9 (See Note 3)
- GCC 4.2.2: libstdc++.so.6.0.9
- GCC 4.3.0: libstdc++.so.6.0.10
- GCC 4.4.0: libstdc++.so.6.0.11
- GCC 4.4.1: libstdc++.so.6.0.12
- GCC 4.4.2: libstdc++.so.6.0.13

B.5.2.5 Checking Active

When the GNU C++ library is being built with symbol versioning on, you should see the following at configure time for libstdc++ (showing either 'gnu' or another of the supported styles):

```
checking versioning on shared library symbols... gnu
```

If you don't see this line in the configure output, or if this line appears but the last word is 'no', then you are out of luck.

If the compiler is pre-installed, a quick way to test is to compile the following (or any) simple C++ file and link it to the shared libstdc++ library:

```
#include <iostream>

int main()
{ std::cout << "hello" << std::endl; return 0; }

%g++ hello.cc -o hello.out

%ldd hello.out
libstdc++.so.5 => /usr/lib/libstdc++.so.5 (0x00764000)
libm.so.6 => /lib/tls/libm.so.6 (0x004a8000)
libgcc_s.so.1 => /mnt/hd/bld/gcc/gcc/libgcc_s.so.1 (0x40016000)
libc.so.6 => /lib/tls/libc.so.6 (0x0036d000)
/lib/ld-linux.so.2 => /lib/ld-linux.so.2 (0x00355000)

%nm hello.out
```

If you see symbols in the resulting output with "GLIBCXX_3" as part of the name, then the executable is versioned. Here's an example:

```
U _ZNSt8ios_base4InitC1Ev@@GLIBCXX_3.4
```

On Solaris 2, you can use `pvs -r` instead:

```
%g++ hello.cc -o hello.out

%pvs -r hello.out
libstdc++.so.6 (GLIBCXX_3.4, GLIBCXX_3.4.12);
libgcc_s.so.1 (GCC_3.0);
libc.so.1 (SUNWprivate_1.1, SYSVABI_1.3);
```

`ldd -v` works too, but is very verbose.

B.5.3 Allowed Changes

The following will cause the library minor version number to increase, say from "libstdc++.so.3.0.4" to "libstdc++.so.3.0.5".

1. Adding an exported global or static data member
2. Adding an exported function, static or non-virtual member function
3. Adding an exported symbol or symbols by additional instantiations

Other allowed changes are possible.

B.5.4 Prohibited Changes

The following non-exhaustive list will cause the library major version number to increase, say from "libstdc++.so.3.0.4" to "libstdc++.so.4.0.0".

1. Changes in the gcc/g++ compiler ABI
2. Changing size of an exported symbol
3. Changing alignment of an exported symbol
4. Changing the layout of an exported symbol
5. Changing mangling on an exported symbol
6. Deleting an exported symbol
7. Changing the inheritance properties of a type by adding or removing base classes
8. Changing the size, alignment, or layout of types specified in the C++ standard. These may not necessarily be instantiated or otherwise exported in the library binary, and include all the required locale facets, as well as things like `std::basic_streambuf`, et al.
9. Adding an explicit copy constructor or destructor to a class that would otherwise have implicit versions. This will change the way the compiler deals with this class in by-value return statements or parameters: instead of passing instances of this class in registers, the compiler will be forced to use memory. See the section on [Function Calling Conventions and APIs](#) of the C++ ABI documentation for further details.

B.5.5 Implementation

1. Separation of interface and implementation

This is accomplished by two techniques that separate the API from the ABI: forcing undefined references to link against a library binary for definitions.

Include files have declarations, source files have defines For non-templated types, such as much of `class locale`, the appropriate standard C++ include, say `locale`, can contain full declarations, while various source files (say `locale.cc`, `locale_init.cc`, `localename.cc`) contain definitions.

Extern template on required types For parts of the standard that have an explicit list of required instantiations, the GNU extension syntax `extern template` can be used to control where template definitions reside. By marking required instantiations as `extern template` in include files, and providing explicit instantiations in the appropriate instantiation files, non-inlined template functions can be versioned. This technique is mostly used on parts of the standard that require `char` and `wchar_t` instantiations, and includes `basic_string`, the locale facets, and the types in `iostreams`.

In addition, these techniques have the additional benefit that they reduce binary size, which can increase runtime performance.

2. Namespaces linking symbol definitions to export mapfiles

All symbols in the shared library binary are processed by a linker script at build time that either allows or disallows external linkage. Because of this, some symbols, regardless of normal C/C++ linkage, are not visible. Symbols that are internal have several appealing characteristics: by not exporting the symbols, there are no relocations when the shared library is started and thus this makes for faster runtime loading performance by the underlying dynamic loading mechanism. In addition, they have the possibility of changing without impacting ABI compatibility.

The following namespaces are transformed by the mapfile:

namespace std Defaults to exporting all symbols in label `GLIBCXX` that do not begin with an underscore, i.e., `__test_func` would not be exported by default. Select exceptional symbols are allowed to be visible.

B.6.10 4.4

C++0X features.

- Added.
`<atomic>`, `<chrono>`, `<condition_variable>`, `<forward_list>`, `<initializer_list>`, `<mutex>`, `<ratio>`,
`<thread>`
- Updated and improved.
`<algorithm>`, `<system_error>`, `<type_traits>`
- Use of the GNU extension namespace association converted to inline namespaces.
- Preliminary support for `initializer_list` and defaulted and deleted constructors in container classes.
- `unique_ptr`.
- Support for new character types `char16_t` and `char32_t` added to `char_traits`, `basic_string`, `numeric_limits`, and assorted compile-time type traits.
- Support for string conversions `to_string` and `to_wstring`.
- Member functions taking string arguments were added to iostreams including `basic_filebuf`, `basic_ofstream`, and `basic_ifstream`.
- Exception propagation support, including `exception_ptr`, `current_exception`, `copy_exception`, and `rethrow_exception`.

Uglification of `try` to `__try` and `catch` to `__catch`.

Audit of internal mutex usage, conversion to functions returning static local mutex.

Extensions added: `<ext/pointer.h>` and `<ext/extptr_allocator.h>`. Support for non-standard pointer types has been added to `vector` and `forward_list`.

B.6.11 4.5

C++0X features.

- Added.
`<functional>`, `<future>`, `<random>`
- Updated and improved.
`<atomic>`, `<system_error>`, `<type_traits>`
- Add support for explicit operators and standard layout types.

Profile mode first appears.

Support for decimal floating-point arithmetic, including `decimal32`, `decimal64`, and `decimal128`.

Python pretty-printers are added for use with appropriately-advanced versions of **gdb**.

Audit for application of function attributes `nothrow`, `const`, `pure`, and `noreturn`.

The default behavior for comparing `typeinfo` names changed, so in `<typeinfo>`, `__GXX_MERGED_TYPEINFO_NAMES` now defaults to zero.

Extensions modified: `<ext/throw_allocator.h>`.

```

    #ifdef __GNUC__
    #if __GNUC__ < 3
#include <hash_map.h>
namespace extension { using ::hash_map; }; // inherit globals
    #else
#include <backward/hash_map>
    #if __GNUC__ == 3 && __GNUC_MINOR__ == 0
        namespace extension = std;                // GCC 3.0
    #else
        namespace extension = ::__gnu_cxx;          // GCC 3.1 and later
    #endif
    #endif
    #else // ... there are other compilers, right?
namespace extension = std;
    #endif

    extension::hash_map<int,int> my_map;

```

This is a bit cleaner than defining typedefs for all the instantiations you might need.

The following autoconf tests check for working HP/SGI hash containers.

```

# AC_HEADER_EXT_HASH_MAP
AC_DEFUN([AC_HEADER_EXT_HASH_MAP], [
    AC_CACHE_CHECK(for ext/hash_map,
    ac_cv_cxx_ext_hash_map,
    [AC_LANG_SAVE
    AC_LANG_CPLUSPLUS
    ac_save_CXXFLAGS="$CXXFLAGS"
    CXXFLAGS="$CXXFLAGS -Werror"
    AC_TRY_COMPILE([#include <ext/hash_map>], [using __gnu_cxx::hash_map;],
    ac_cv_cxx_ext_hash_map=yes, ac_cv_cxx_ext_hash_map=no)
    CXXFLAGS="$ac_save_CXXFLAGS"
    AC_LANG_RESTORE
    ])
    if test "$ac_cv_cxx_ext_hash_map" = yes; then
        AC_DEFINE(HAVE_EXT_HASH_MAP,,[Define if ext/hash_map is present. ])
    fi
])

```

```

# AC_HEADER_EXT_HASH_SET
AC_DEFUN([AC_HEADER_EXT_HASH_SET], [
    AC_CACHE_CHECK(for ext/hash_set,
    ac_cv_cxx_ext_hash_set,
    [AC_LANG_SAVE
    AC_LANG_CPLUSPLUS
    ac_save_CXXFLAGS="$CXXFLAGS"
    CXXFLAGS="$CXXFLAGS -Werror"
    AC_TRY_COMPILE([#include <ext/hash_set>], [using __gnu_cxx::hash_set;],
    ac_cv_cxx_ext_hash_set=yes, ac_cv_cxx_ext_hash_set=no)
    CXXFLAGS="$ac_save_CXXFLAGS"
    AC_LANG_RESTORE
    ])
    if test "$ac_cv_cxx_ext_hash_set" = yes; then
        AC_DEFINE(HAVE_EXT_HASH_SET,,[Define if ext/hash_set is present. ])
    fi
])

```

B.7.3.3 No `ios::nocreate/ios::noreplace`.

Historically these flags were used with iostreams to control whether new files are created or not when opening a file stream, similar to the `O_CREAT` and `O_EXCL` flags for the `open(2)` system call. Because iostream modes correspond to `fopen(3)` modes these flags are not supported. For input streams a new file will not be created anyway, so `ios::nocreate` is not needed. For output streams, a new file will be created if it does not exist, which is consistent with the behaviour of `fopen`.

When one of these flags is needed a possible alternative is to attempt to open the file using `std::ifstream` first to determine whether the file already exists or not. This may not be reliable however, because whether the file exists or not could change between opening the `std::istream` and re-opening with an output stream. If you need to check for existence and open a file as a single operation then you will need to use OS-specific facilities outside the C++ standard library, such as `open(2)`.

B.7.3.4 No `stream::attach(int fd)`

Phil Edwards writes: It was considered and rejected for the ISO standard. Not all environments use file descriptors. Of those that do, not all of them use integers to represent them.

For a portable solution (among systems which use file descriptors), you need to implement a subclass of `std::streambuf` (or `std::basic_streambuf<...>`) which opens a file given a descriptor, and then pass an instance of this to the stream-constructor.

An extension is available that implements this. `<ext/stdio_filebuf.h>` contains a derived class called `__gnu_cxx::stdio_filebuf`. This class can be constructed from a `C FILE*` or a file descriptor, and provides the `fd()` function.

For another example of this, refer to [fdstream example](#) by Nicolai Josuttis.

B.7.3.5 Support for C++98 dialect.

Check for complete library coverage of the C++1998/2003 standard.

```
# AC_HEADER_STDCXX_98
AC_DEFUN([AC_HEADER_STDCXX_98], [
  AC_CACHE_CHECK(for ISO C++ 98 include files,
    ac_cv_cxx_stdcxx_98,
    [AC_LANG_SAVE
     AC_LANG_CPLUSPLUS
     AC_TRY_COMPILE([
       #include <cassert>
       #include <cctype>
       #include <cerrno>
       #include <cfloat>
       #include <ciso646>
       #include <climits>
       #include <locale>
       #include <cmath>
       #include <setjmp>
       #include <signal>
       #include <stdarg>
       #include <stddef>
       #include <stdio>
       #include <stdlib>
       #include <string>
       #include <time>

       #include <algorithm>
       #include <bitset>
       #include <complex>
       #include <deque>
       #include <exception>
       #include <fstream>
       #include <functional>
```

```

#include <iomanip>
#include <ios>
#include <iosfwd>
#include <iostream>
#include <istream>
#include <iterator>
#include <limits>
#include <list>
#include <locale>
#include <map>
#include <memory>
#include <new>
#include <numeric>
#include <ostream>
#include <queue>
#include <set>
#include <sstream>
#include <stack>
#include <stdexcept>
#include <streambuf>
#include <string>
#include <typeinfo>
#include <utility>
#include <valarray>
#include <vector>
],,
ac_cv_cxx_stdcxx_98=yes, ac_cv_cxx_stdcxx_98=no)
AC_LANG_RESTORE
])
if test "$ac_cv_cxx_stdcxx_98" = yes; then
    AC_DEFINE(STDCCXX_98_HEADERS,, [Define if ISO C++ 1998 header files are present. ])
fi
])

```

B.7.3.6 Support for C++TR1 dialect.

Check for library coverage of the TR1 standard.

```

# AC_HEADER_STDCXX_TR1
AC_DEFUN([AC_HEADER_STDCXX_TR1], [
    AC_CACHE_CHECK(for ISO C++ TR1 include files,
        ac_cv_cxx_stdcxx_tr1,
        [AC_LANG_SAVE
         AC_LANG_CPLUSPLUS
         AC_TRY_COMPILE([
             #include <tr1/array>
             #include <tr1/complex>
             #include <tr1/cctype>
             #include <tr1/cfenv>
             #include <tr1/cfloat>
             #include <tr1/cinttypes>
             #include <tr1/climits>
             #include <tr1/cmath>
             #include <tr1/complex>
             #include <tr1/cstdarg>
             #include <tr1/cstdbool>
             #include <tr1/cstdint>
             #include <tr1/stdio>
             #include <tr1/stdlib>
             #include <tr1/ctgmath>

```

```

#include <tr1/ctime>
#include <tr1/cwchar>
#include <tr1/cwctype>
#include <tr1/functional>
#include <tr1/memory>
#include <tr1/random>
#include <tr1/regex>
#include <tr1/tuple>
#include <tr1/type_traits>
#include <tr1/unordered_set>
#include <tr1/unordered_map>
#include <tr1/utility>
],,
ac_cv_cxx_stdcxx_tr1=yes, ac_cv_cxx_stdcxx_tr1=no)
AC_LANG_RESTORE
])
if test "$ac_cv_cxx_stdcxx_tr1" = yes; then
    AC_DEFINE(STDCCXX_TR1_HEADERS,,[Define if ISO C++ TR1 header files are present. ])
fi
])

```

An alternative is to check just for specific TR1 includes, such as `<unordered_map>` and `<unordered_set>`.

```

# AC_HEADER_TR1_UNORDERED_MAP
AC_DEFUN([AC_HEADER_TR1_UNORDERED_MAP], [
    AC_CACHE_CHECK(for tr1/unordered_map,
        ac_cv_cxx_tr1_unordered_map,
        [AC_LANG_SAVE
         AC_LANG_CPLUSPLUS
         AC_TRY_COMPILE([#include <tr1/unordered_map>], [using std::tr1::unordered_map;],
            ac_cv_cxx_tr1_unordered_map=yes, ac_cv_cxx_tr1_unordered_map=no)
         AC_LANG_RESTORE
        ])
    if test "$ac_cv_cxx_tr1_unordered_map" = yes; then
        AC_DEFINE(HAVE_TR1_UNORDERED_MAP,,[Define if tr1/unordered_map is present. ])
    fi
])

```

```

# AC_HEADER_TR1_UNORDERED_SET
AC_DEFUN([AC_HEADER_TR1_UNORDERED_SET], [
    AC_CACHE_CHECK(for tr1/unordered_set,
        ac_cv_cxx_tr1_unordered_set,
        [AC_LANG_SAVE
         AC_LANG_CPLUSPLUS
         AC_TRY_COMPILE([#include <tr1/unordered_set>], [using std::tr1::unordered_set;],
            ac_cv_cxx_tr1_unordered_set=yes, ac_cv_cxx_tr1_unordered_set=no)
         AC_LANG_RESTORE
        ])
    if test "$ac_cv_cxx_tr1_unordered_set" = yes; then
        AC_DEFINE(HAVE_TR1_UNORDERED_SET,,[Define if tr1/unordered_set is present. ])
    fi
])

```

B.7.3.7 Support for C++11 dialect.

Check for baseline language coverage in the compiler for the C++11 standard.

```

# AC_COMPILE_STDCXX_11
AC_DEFUN([AC_COMPILE_STDCXX_11], [
    AC_CACHE_CHECK(if g++ supports C++11 features without additional flags,

```

```

ac_cv_cxx_compile_cxx11_native,
[AC_LANG_SAVE
AC_LANG_CPLUSPLUS
AC_TRY_COMPILE([
template <typename T>
struct check final
{
    static constexpr T value{ __cplusplus };
};

typedef check<check<bool>> right_angle_brackets;

int a;
decltype(a) b;

typedef check<int> check_type;
check_type c{};
check_type&& cr = static_cast<check_type&&>(c);

static_assert(check_type::value == 201103L, "C++11 compiler");],,
ac_cv_cxx_compile_cxx11_native=yes, ac_cv_cxx_compile_cxx11_native=no)
AC_LANG_RESTORE
])

AC_CACHE_CHECK(if g++ supports C++11 features with -std=c++11,
ac_cv_cxx_compile_cxx11_cxx,
[AC_LANG_SAVE
AC_LANG_CPLUSPLUS
ac_save_CXXFLAGS="$CXXFLAGS"
CXXFLAGS="$CXXFLAGS -std=c++11"
AC_TRY_COMPILE([
template <typename T>
struct check final
{
    static constexpr T value{ __cplusplus };
};

typedef check<check<bool>> right_angle_brackets;

int a;
decltype(a) b;

typedef check<int> check_type;
check_type c{};
check_type&& cr = static_cast<check_type&&>(c);

static_assert(check_type::value == 201103L, "C++11 compiler");],,
ac_cv_cxx_compile_cxx11_cxx=yes, ac_cv_cxx_compile_cxx11_cxx=no)
CXXFLAGS="$ac_save_CXXFLAGS"
AC_LANG_RESTORE
])

AC_CACHE_CHECK(if g++ supports C++11 features with -std=gnu++11,
ac_cv_cxx_compile_cxx11_gxx,
[AC_LANG_SAVE
AC_LANG_CPLUSPLUS
ac_save_CXXFLAGS="$CXXFLAGS"
CXXFLAGS="$CXXFLAGS -std=gnu++11"
AC_TRY_COMPILE([
template <typename T>
struct check final
{

```

```

    static constexpr T value{ __cplusplus };
};

typedef check<check<bool>> right_angle_brackets;

int a;
decltype(a) b;

typedef check<int> check_type;
check_type c{};
check_type&& cr = static_cast<check_type&&>(c);

    static_assert(check_type::value == 201103L, "C++11 compiler");],,
ac_cv_cxx_compile_cxx11_gxx=yes, ac_cv_cxx_compile_cxx11_gxx=no)
CXXFLAGS="$ac_save_CXXFLAGS"
AC_LANG_RESTORE
])

if test "$ac_cv_cxx_compile_cxx11_native" = yes ||
    test "$ac_cv_cxx_compile_cxx11_cxx" = yes ||
    test "$ac_cv_cxx_compile_cxx11_gxx" = yes; then
    AC_DEFINE(HAVE_STDCXX_11,,[Define if g++ supports C++11 features. ])
fi
])

```

Check for library coverage of the C++2011 standard. (Some library headers are commented out in this check, they are not currently provided by libstdc++).

```

# AC_HEADER_STDCXX_11
AC_DEFUN([AC_HEADER_STDCXX_11], [
    AC_CACHE_CHECK(for ISO C++11 include files,
        ac_cv_cxx_stdcxx_11,
        [AC_REQUIRE([AC_COMPILE_STDCXX_11])
        AC_LANG_SAVE
        AC_LANG_CPLUSPLUS
        ac_save_CXXFLAGS="$CXXFLAGS"
        CXXFLAGS="$CXXFLAGS -std=gnu++11"

        AC_TRY_COMPILE([
            #include <cassert>
            #include <ccomplex>
            #include <cctype>
            #include <cerrno>
            #include <cfenv>
            #include <cfloat>
            #include <cinttypes>
            #include <ciso646>
            #include <climits>
            #include <locale>
            #include <cmath>
            #include <setjmp>
            #include <signal>
            #include <stdalign>
            #include <stdarg>
            #include <stdbool>
            #include <stddef>
            #include <stdint>
            #include <stdio>
            #include <stdlib>
            #include <string>
            #include <tgmath>
            #include <time>

```

```
// #include <cuchar>
#include <cwchar>
#include <cwctype>

#include <algorithm>
#include <array>
#include <atomic>
#include <bitset>
#include <chrono>
// #include <codecvt>
#include <complex>
#include <condition_variable>
#include <deque>
#include <exception>
#include <forward_list>
#include <fstream>
#include <functional>
#include <future>
#include <initializer_list>
#include <iomanip>
#include <ios>
#include <iosfwd>
#include <iostream>
#include <istream>
#include <iterator>
#include <limits>
#include <list>
#include <locale>
#include <map>
#include <memory>
#include <mutex>
#include <new>
#include <numeric>
#include <ostream>
#include <queue>
#include <random>
#include <ratio>
#include <regex>
#include <scoped_allocator>
#include <set>
#include <sstream>
#include <stack>
#include <stdexcept>
#include <streambuf>
#include <string>
#include <system_error>
#include <thread>
#include <tuple>
#include <typeindex>
#include <typeinfo>
#include <type_traits>
#include <unordered_map>
#include <unordered_set>
#include <utility>
#include <valarray>
#include <vector>

],,
ac_cv_cxx_stdcxx_11=yes, ac_cv_cxx_stdcxx_11=no)
AC_LANG_RESTORE
CXXFLAGS="$ac_save_CXXFLAGS"
])
if test "$ac_cv_cxx_stdcxx_11" = yes; then
```

```

    AC_DEFINE(STDCXX_11_HEADERS,,[Define if ISO C++11 header files are present. ])
  fi
})

```

As is the case for TR1 support, these autoconf macros can be made for a finer-grained, per-header-file check. For `<unordered_map>`

```

# AC_HEADER_UNORDERED_MAP
AC_DEFUN([AC_HEADER_UNORDERED_MAP], [
  AC_CACHE_CHECK(for unordered_map,
    ac_cv_cxx_unordered_map,
    [AC_REQUIRE([AC_COMPILE_STDCXX_11])
     AC_LANG_SAVE
     AC_LANG_CPLUSPLUS
     ac_save_CXXFLAGS="$CXXFLAGS"
     CXXFLAGS="$CXXFLAGS -std=gnu++11"
     AC_TRY_COMPILE([#include <unordered_map>], [using std::unordered_map;],
       ac_cv_cxx_unordered_map=yes, ac_cv_cxx_unordered_map=no)
     CXXFLAGS="$ac_save_CXXFLAGS"
     AC_LANG_RESTORE
    ])
  if test "$ac_cv_cxx_unordered_map" = yes; then
    AC_DEFINE(HAVE_UNORDERED_MAP,,[Define if unordered_map is present. ])
  fi
])

```

```

# AC_HEADER_UNORDERED_SET
AC_DEFUN([AC_HEADER_UNORDERED_SET], [
  AC_CACHE_CHECK(for unordered_set,
    ac_cv_cxx_unordered_set,
    [AC_REQUIRE([AC_COMPILE_STDCXX_11])
     AC_LANG_SAVE
     AC_LANG_CPLUSPLUS
     ac_save_CXXFLAGS="$CXXFLAGS"
     CXXFLAGS="$CXXFLAGS -std=gnu++11"
     AC_TRY_COMPILE([#include <unordered_set>], [using std::unordered_set;],
       ac_cv_cxx_unordered_set=yes, ac_cv_cxx_unordered_set=no)
     CXXFLAGS="$ac_save_CXXFLAGS"
     AC_LANG_RESTORE
    ])
  if test "$ac_cv_cxx_unordered_set" = yes; then
    AC_DEFINE(HAVE_UNORDERED_SET,,[Define if unordered_set is present. ])
  fi
])

```

Some C++11 features first appeared in GCC 4.3 and could be enabled by `-std=c++0x` and `-std=gnu++0x` for GCC releases which pre-date the 2011 standard. Those C++11 features and GCC's support for them were still changing until the 2011 standard was finished, but the autoconf checks above could be extended to test for incomplete C++11 support with `-std=c++0x` and `-std=gnu++0x`.

B.7.3.8 `Container::iterator_type` is not necessarily `Container::value_type*`

This is a change in behavior from older versions. Now, most `iterator_type` typedefs in container classes are POD objects, not `value_type` pointers.

Appendix C

Free Software Needs Free Documentation

The biggest deficiency in free operating systems is not in the software--it is the lack of good free manuals that we can include in these systems. Many of our most important programs do not come with full manuals. Documentation is an essential part of any software package; when an important free software package does not come with a free manual, that is a major gap. We have many such gaps today.

Once upon a time, many years ago, I thought I would learn Perl. I got a copy of a free manual, but I found it hard to read. When I asked Perl users about alternatives, they told me that there were better introductory manuals--but those were not free.

Why was this? The authors of the good manuals had written them for O'Reilly Associates, which published them with restrictive terms--no copying, no modification, source files not available--which exclude them from the free software community.

That wasn't the first time this sort of thing has happened, and (to our community's great loss) it was far from the last. Proprietary manual publishers have enticed a great many authors to restrict their manuals since then. Many times I have heard a GNU user eagerly tell me about a manual that he is writing, with which he expects to help the GNU project--and then had my hopes dashed, as he proceeded to explain that he had signed a contract with a publisher that would restrict it so that we cannot use it.

Given that writing good English is a rare skill among programmers, we can ill afford to lose manuals this way.

Free documentation, like free software, is a matter of freedom, not price. The problem with these manuals was not that O'Reilly Associates charged a price for printed copies--that in itself is fine. (The Free Software Foundation [sells printed copies](#) of free GNU manuals, too.) But GNU manuals are available in source code form, while these manuals are available only on paper. GNU manuals come with permission to copy and modify; the Perl manuals do not. These restrictions are the problems.

The criterion for a free manual is pretty much the same as for free software: it is a matter of giving all users certain freedoms. Redistribution (including commercial redistribution) must be permitted, so that the manual can accompany every copy of the program, on-line or on paper. Permission for modification is crucial too.

As a general rule, I don't believe that it is essential for people to have permission to modify all sorts of articles and books. The issues for writings are not necessarily the same as those for software. For example, I don't think you or I are obliged to give permission to modify articles like this one, which describe our actions and our views.

But there is a particular reason why the freedom to modify is crucial for documentation for free software. When people exercise their right to modify the software, and add or change its features, if they are conscientious they will change the manual too--so they can provide accurate and usable documentation with the modified program. A manual which forbids programmers to be conscientious and finish the job, or more precisely requires them to write a new manual from scratch if they change the program, does not fill our community's needs.

While a blanket prohibition on modification is unacceptable, some kinds of limits on the method of modification pose no problem. For example, requirements to preserve the original author's copyright notice, the distribution terms, or the list of authors, are ok. It is also no problem to require modified versions to include notice that they were modified, even to have entire sections that may not be deleted or changed, as long as these sections deal with nontechnical topics. (Some GNU manuals have them.)

These kinds of restrictions are not a problem because, as a practical matter, they don't stop the conscientious programmer from adapting the manual to fit the modified program. In other words, they don't block the free software community from making full use of the manual.

However, it must be possible to modify all the *technical* content of the manual, and then distribute the result in all the usual media, through all the usual channels; otherwise, the restrictions do block the community, the manual is not free, and so we need another manual.

Unfortunately, it is often hard to find someone to write another manual when a proprietary manual exists. The obstacle is that many users think that a proprietary manual is good enough--so they don't see the need to write a free manual. They do not see that the free operating system has a gap that needs filling.

Why do users think that proprietary manuals are good enough? Some have not considered the issue. I hope this article will do something to change that.

Other users consider proprietary manuals acceptable for the same reason so many people consider proprietary software acceptable: they judge in purely practical terms, not using freedom as a criterion. These people are entitled to their opinions, but since those opinions spring from values which do not include freedom, they are no guide for those of us who do value freedom.

Please spread the word about this issue. We continue to lose manuals to proprietary publishing. If we spread the word that proprietary manuals are not sufficient, perhaps the next person who wants to help GNU by writing documentation will realize, before it is too late, that he must above all make it free.

We can also encourage commercial publishers to sell free, copylefted manuals instead of proprietary ones. One way you can help this is to check the distribution terms of a manual before you buy it, and prefer copylefted manuals to non-copylefted ones.

[Note: We now maintain a [web page that lists free books available from other publishers](#)].

Copyright © 2004, 2005, 2006, 2007 Free Software Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301, USA

Verbatim copying and distribution of this entire article are permitted worldwide, without royalty, in any medium, provided this notice is preserved.

Report any problems or suggestions to webmaster@fsf.org.

Appendix D

GNU General Public License version 3

Version 3, 29 June 2007

Copyright © 2007 Free Software Foundation, Inc. <https://www.fsf.org>

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

Preamble

The GNU General Public License is a free, copyleft license for software and other kinds of works.

The licenses for most software and other practical works are designed to take away your freedom to share and change the works. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change all versions of a program—to make sure it remains free software for all its users. We, the Free Software Foundation, use the GNU General Public License for most of our software; it applies also to any other work released this way by its authors. You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for them if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs, and that you know you can do these things.

To protect your rights, we need to prevent others from denying you these rights or asking you to surrender the rights. Therefore, you have certain responsibilities if you distribute copies of the software, or if you modify it: responsibilities to respect the freedom of others.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must pass on to the recipients the same freedoms that you received. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

Developers that use the GNU GPL protect your rights with two steps: (1) assert copyright on the software, and (2) offer you this License giving you legal permission to copy, distribute and/or modify it.

For the developers' and authors' protection, the GPL clearly explains that there is no warranty for this free software. For both users' and authors' sake, the GPL requires that modified versions be marked as changed, so that their problems will not be attributed erroneously to authors of previous versions.

Some devices are designed to deny users access to install or run modified versions of the software inside them, although the manufacturer can do so. This is fundamentally incompatible with the aim of protecting users' freedom to change the software. The systematic pattern of such abuse occurs in the area of products for individuals to use, which is precisely where it is most unacceptable. Therefore, we have designed this version of the GPL to prohibit the practice for those products. If such problems arise substantially in other domains, we stand ready to extend this provision to those domains in future versions of the GPL, as needed to protect the freedom of users.

Finally, every program is threatened constantly by software patents. States should not allow patents to restrict development and use of software on general-purpose computers, but in those that do, we wish to avoid the special danger that patents applied to a free program could make it effectively proprietary. To prevent this, the GPL assures that patents cannot be used to render the program non-free.

The precise terms and conditions for copying, distribution and modification follow.

TERMS AND CONDITIONS

0. Definitions.

“This License” refers to version 3 of the GNU General Public License.

“Copyright” also means copyright-like laws that apply to other kinds of works, such as semiconductor masks.

“The Program” refers to any copyrightable work licensed under this License. Each licensee is addressed as “you”. “Licensees” and “recipients” may be individuals or organizations.

To “modify” a work means to copy from or adapt all or part of the work in a fashion requiring copyright permission, other than the making of an exact copy. The resulting work is called a “modified version” of the earlier work or a work “based on” the earlier work.

A “covered work” means either the unmodified Program or a work based on the Program.

To “propagate” a work means to do anything with it that, without permission, would make you directly or secondarily liable for infringement under applicable copyright law, except executing it on a computer or modifying a private copy. Propagation includes copying, distribution (with or without modification), making available to the public, and in some countries other activities as well.

To “convey” a work means any kind of propagation that enables other parties to make or receive copies. Mere interaction with a user through a computer network, with no transfer of a copy, is not conveying.

An interactive user interface displays “Appropriate Legal Notices” to the extent that it includes a convenient and prominently visible feature that (1) displays an appropriate copyright notice, and (2) tells the user that there is no warranty for the work (except to the extent that warranties are provided), that licensees may convey the work under this License, and how to view a copy of this License. If the interface presents a list of user commands or options, such as a menu, a prominent item in the list meets this criterion.

1. Source Code.

The “source code” for a work means the preferred form of the work for making modifications to it. “Object code” means any non-source form of a work.

A “Standard Interface” means an interface that either is an official standard defined by a recognized standards body, or, in the case of interfaces specified for a particular programming language, one that is widely used among developers working in that language.

The “System Libraries” of an executable work include anything, other than the work as a whole, that (a) is included in the normal form of packaging a Major Component, but which is not part of that Major Component, and (b) serves only to enable use of the work with that Major Component, or to implement a Standard Interface for which an implementation is available to the public in source code form. A “Major Component”, in this context, means a major essential component (kernel, window system, and so on) of the specific operating system (if any) on which the executable work runs, or a compiler used to produce the work, or an object code interpreter used to run it.

The “Corresponding Source” for a work in object code form means all the source code needed to generate, install, and (for an executable work) run the object code and to modify the work, including scripts to control those activities. However, it does not include the work’s System Libraries, or general-purpose tools or generally available free programs which are used unmodified in performing those activities but which are not part of the work. For example, Corresponding Source includes interface definition files associated with source files for the work, and the source code for shared libraries and dynamically linked subprograms that

the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program. If not, see <http://www.gnu.org/licenses/>.

Also add information on how to contact you by electronic and paper mail.

If the program does terminal interaction, make it output a short notice like this when it starts in an interactive mode:

```
<i>program</i> Copyright (C) <i>year</i> <i>name of author</i>
This program comes with ABSOLUTELY NO WARRANTY; for details type `show w'.
This is free software, and you are welcome to redistribute it
under certain conditions; type `show c' for details.
```

The hypothetical commands ‘show w’ and ‘show c’ should show the appropriate parts of the General Public License. Of course, your program’s commands might be different; for a GUI interface, you would use an “about box”.

You should also get your employer (if you work as a programmer) or school, if any, to sign a “copyright disclaimer” for the program, if necessary. For more information on this, and how to apply and follow the GNU GPL, see <http://www.gnu.org/licenses/>.

The GNU General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Lesser General Public License instead of this License. But first, please read <https://www.gnu.org/licenses/why-not-lgpl.html>.

copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

- A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.
- B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.
- C. State on the Title page the name of the publisher of the Modified Version, as the publisher.
- D. Preserve all the copyright notices of the Document.
- E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.
- F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.
- G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.
- H. Include an unaltered copy of this License.
- I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.
- J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.
- K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.
- L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.
- M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.
- N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.
- O. Preserve any Warranty Disclaimers.

The operator of an MMC Site may republish an MMC contained in the site under CC-BY-SA on the same site at any time before August 1, 2009, provided the MMC is eligible for relicensing.

ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

```
Copyright © YEAR YOUR NAME
```

```
Permission is granted to copy, distribute and/or modify this document under the
terms of the GNU Free Documentation License, Version 1.3 or any later version
published by the Free Software Foundation; with no Invariant Sections, no
Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in
the section entitled "GNU Free Documentation License".
```

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the “with... Texts.” line with this:

```
with the Invariant Sections being LIST THEIR TITLES, with the Front-Cover Texts
being LIST, and with the Back-Cover Texts being LIST.
```

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

Chapter 30

Index

A

Algorithms, [110](#)

Appendix

Contributing, [288](#)

Free Documentation, [376](#)

Porting and Maintenance, [317](#)

Atomics, [120](#)

C

Concurrency, [121](#)

Containers, [102](#)

D

Diagnostics, [68](#)

E

Extensions, [122](#)

I

Input and Output, [113](#)

Introduction, [1](#)

Iterators, [108](#)

L

Localization, [85](#)

N

Numerics, [111](#)

S

Strings, [79](#)

Support, [62](#)

T

Test

Exception Safety, [344](#)

U

Utilities, [70](#)
