

CodeWarrior®

MSL C++ Reference



Because of last-minute changes to CodeWarrior, some of the information in this manual may be inaccurate. Please read the Release Notes on the CodeWarrior CD for the latest up-to-date information.

Revised: 990501 rdl

Metrowerks CodeWarrior copyright ©1993–1998 by Metrowerks Inc. and its licensors.
All rights reserved.

Documentation stored on the compact disk(s) may be printed by licensee for personal use. Except for the foregoing, no part of this documentation may be reproduced or transmitted in any form by any means, electronic or mechanical, including photocopying, recording, or any information storage and retrieval system, without permission in writing from Metrowerks Inc.

Metrowerks, the Metrowerks logo, CodeWarrior, and Software at Work are registered trademarks of Metrowerks Inc. PowerPlant and PowerPlant Constructor are trademarks of Metrowerks Inc.

All other trademarks and registered trademarks are the property of their respective owners.

ALL SOFTWARE AND DOCUMENTATION ON THE COMPACT DISK(S) ARE SUBJECT TO THE LICENSE AGREEMENT IN THE CD BOOKLET.

How to Contact Metrowerks:

U.S.A. and international	Metrowerks Corporation P.O. Box 334 Austin, TX 78758 U.S.A.
---------------------------------	----------------------------------------------------------------------

Canada	Metrowerks Inc. 1500 du College, Suite 300 Ville St-Laurent, QC Canada H4L 5G6
---------------	-----------------------------------------------------------------------------------------

Ordering	Voice: (800) 377–5416 Fax: (512) 873–4901
-----------------	----------------------------------------------

World Wide Web	http://www.metrowerks.com
-----------------------	-------------------------------------------------------------------

Registration information	register@metrowerks.com
---------------------------------	----------------------------------------------------------------------

Technical support	cw_support@metrowerks.com
--------------------------	--------------------------------------------------------------------------

Sales, marketing, & licensing	sales@metrowerks.com
------------------------------------------	----------------------------------------------------------------

CompuServe	goto Metrowerks
-------------------	-----------------

Table of Contents

1 Introduction	39
CodeWarrior Year 2000 Compliance	39
For additional information, visit: http://www.metrowerks.com/about/y2k.html	39
About the MSL C++ Library Reference Manual	39
2 17 C++ Library	43
Overview of the MSL C++ Reference	43
17.1 Definitions.	43
17.2 Additional Definitions.	47
17.3 Methods of Descriptions.	47
17.3.1 Structure of each subclause.	47
17.3.2 Other Conventions	48
17.4 Library-wide Requirements	49
17.4.1 Library contents and organization.	50
17.4.2 Using the library	52
17.4.3 Constraints on programs.	52
17.4.4 Conforming Implementations	54
17.4.4.8 Restrictions On Exception Handling	54
Features not implemented in MSL C++	55
Template Functionality	55
ANSI/ISO Library Functionality	55
3 Language Support Library	57
Overview of Language Support Libraries	57
Types and Macros	57
Macros	58
NULL	58
offsetof.	58
sizeof.	58
Type Implementations	58
Character types	59
Enumeration.	59
Integral types	59

Wide character types	60
Bool type	60
Floating point types	60
Void type	60
Numeric Limits.	60
Template Class numeric_limits.	61
numeric_limits::is_specialized	61
numeric_limits::min	62
numeric_limits::max	62
numeric_limits::digits.	62
numeric_limits::digits10.	62
numeric_limits::is_signed	63
numeric_limits::is_integer	63
numeric_limits::is_exact.	63
numeric_limits::radix	63
numeric_limits::epsilon	64
numeric_limits::round_error	64
numeric_limits::min_exponent	64
numeric_limits::min_exponent10.	64
numeric_limits::max_exponent.	64
numeric_limits::max_exponent10.	65
numeric_limits::has_infinity	65
numeric_limits::has_quiet_NaN	65
numeric_limits::has_signaling_NaN	65
numeric_limits::has_denorm.	66
numeric_limits::infinity	66
numeric_limits::quiet_NaN	66
numeric_limits::signaling_NaN	66
numeric_limits::denorm_min	66
numeric_limits::is_bounded	67
numeric_limits::is_modulo	67
numeric_limits::traps	67
numeric_limits::tinyness_before	67
numeric_limits::round_style	68
Dynamic Memory Storage Allocation Errors	68

Class bad_alloc	68
Constructor-bad_alloc	68
Copy Constructor-bad_alloc.	68
Assignment Operator-bad_alloc	69
bad_alloc::what	69
Class bad_cast	69
Constructor-bad_cast.	69
Copy Constructor-bad_cast	69
Assignment Operator-bad_cast	70
bad_cast::what.	70
Class bad_typeid.	70
Constructor-bad_typeid	70
Copy Constructor-bad_typeid	70
Assignment Operator-bad_typeid	71
bad_typeid::what.	71

4 Diagnostics Library 73

Overview of Diagnostics Library	73
Exception Classes.	73
Class exception	74
Constructor-exception	74
Copy Constructor-exception.	74
Assignment Operator-exception	75
Destructor-exception	75
exception::what	75
Class logic_error	75
Constructor-logic_error	75
Class domain_error.	76
Constructor-domain_error	76
Class invalid_argument	76
Constructor-invalid_argument.	76
Class length_error	76
Constructor-length_error	77
Class out_of_range	77
Constructor-out_of_range.	77

Class <code>runtime_error</code>	77
Constructor- <code>runtime_error</code>	78
Class <code>range_error</code>	78
Constructor- <code>range_error</code>	78
Class <code>overflow_error</code>	78
Constructor- <code>overflow_error</code>	78

5 General Utilities Libraries 81

Overview of General Utilities Libraries	81
Allocator Classes	81
Passing Allocators to STL Containers	82
Extracting Information from an Allocator Object	82
The Default Allocator Interface	83
Typedef Declarations	85
<code>allocator::pointer</code>	85
<code>allocator::const_pointer</code>	85
<code>allocator::reference</code>	85
<code>allocator::const_reference</code>	86
<code>allocator::value_type</code>	86
Allocator Member Functions.	86
Constructor- <code>allocator</code>	86
Destructor- <code>allocator</code>	86
<code>allocator::size_type</code>	87
<code>allocator::difference_type</code>	87
<code>allocator::address</code>	87
<code>allocator::const_address</code>	87
<code>allocator::allocate</code>	88
<code>allocator::deallocate</code>	88
<code>allocator::max_size</code>	88
Custom Allocators	89
<code>allocator::vec_default</code>	89
<code>allocator::vec_large</code>	89
<code>allocator::vec_huge</code>	89
Allocator Requirements	89
Function Objects	90

Function Adaptors	90
Arithmetic Operations.	91
plus	92
minus.	92
times	92
divides	92
modulus	92
negate	93
Comparison Operations	93
equal_to	93
not_equal_to.	93
greater	94
less.	94
greater_equal	94
less_equal.	94
Logical Operations	95
logical_and	95
logical_or	95
logical_not	95
Negator Adaptors	96
not1	96
not2	96
Binder Adaptors	96
bind1st	97
bind2nd.	97
Adaptors for Pointers to Functions	97
ptr_fun-unary	98
ptr_fun-binary.	98
Specialized Algorithms	99
uninitialized_copy	99
uninitialized_fill	99
uninitialized_fill_n	100
Template class auto_ptr	100
Constructor-auto_ptr.	101
auto_ptr Destructor.	101

auto_ptr::get	102
auto_ptr::release	102
Assignment Operator=auto_ptr	102
Dereferencing Operator*auto_ptr.	102
Association Operator->auto_ptr	102

6 21 Strings Library 105

Overview of Strings Classes	105
21.1 Character traits	105
21.1.1 Character Trait Definitions	106
21.1.2 Character Trait Requirements.	106
21.1.3 Character Trait Type Definitions	107
21.1.4 struct char_traits<T>	108
21.2 String Classes	109
21.3 Class basic_string.	115
21.3.1 Constructors and Assignments	121
21.3.2 Iterator Support	124
21.3.3 Capacity.	125
21.3.4 Element Access.	126
21.3.5 Modifiers	127
21.3.6 String Operations.	131
23.3.7 Non-Member Functions and Operators	136
21.3.7.9 Inserters and extractors	142
23.4 Null Terminated Sequence Utilities	143

7 Localization Library 147

Overview of the Localization Library	147
Class locale	148
Typedef Declarations	148
Public Data Members	150
Constructors.	150
Default Constructor	150
Overloaded and Copy Constructors	150
Public member operators	152
Assignment Operator= locale	152
Equality Operator== locale	152

Not Equal Operator!= locale	153
Grouping Operator locale	153
Public Member Functions	153
locale::name	153
locale::has	154
locale::use	154
Static Member Functions	155
locale::global.	155
locale::classic	155
locale::transparent	155
Global Operators.	156
Extractor Operator>> locale	156
Insertter Operator << locale	156
Class facet	156
Class id	158
The Numerics Category	158
Class numpunct	159
Typedef Declarations numpunct	159
numpunct::char_type	159
numpunct::string.	159
Public Member Functions numpunct	160
numpunct::decimal_point	160
numpunct::thousands_sep.	160
numpunct::grouping	160
numpunct::truename	161
numpunct::falsename.	161
Class num_get	161
Typedef Declarations num_get.	161
char_type	161
iter_type	161
ios	162
Public Member Functions num_get.	162
num_get::get.	162
Class num_put	163
Typedef Declarations num_put.	163

char_type	163
iter_type	163
ios	163
Public Member Functions num_put.	164
num_put::put	164
The Collate Category	165
Class Collate	165
Typedef Declarations collate	165
char_type	165
string	165
Public Member Functions collate	166
collate::compare	166
collate::transform.	166
collate::hash	166
Ctype Category.	167
Class ctype<charT>	167
Typedef Declarations ctype<charT>.	168
char_type	168
Public Member Functions ctype<charT>.	168
ctype::is	168
ctype::do_is	168
ctype::scan_is	169
ctype::scan_not	169
ctype::toupper	170
ctype::tolower	170
ctype::widen.	170
ctype::narrow	171
Class ctype<char> Specialization	171
Data Members ctype<char>	172
ctype::ctype_mask	172
ctype::table_	172
ctype::classic_table_	172
ctype::delete_it_	172
Public Member Functions ctype<char>	173
constructor ctype<char>	173

ctype::is for ctype<char>	173
ctype::scan_is for ctype<char>	174
ctype::scan_not for ctype<char>	174
ctype::tolower for ctype<char>.	174
ctype::toupper for ctype<char>.	175
ctype::widen for ctype<char>	175
ctype::narrow for ctype<char>	175
Class codecvt	176
Typedef Declarations codecvt	176
from_type	176
to_type	176
state_type	176
Member Functions codecvt	177
The Monetary Category	178
Class moneypunct	179
Typedef Declarations moneypunct	179
char_type	179
string_type	179
Public Member Functions moneypunct	179
moneypunct::decimal_point	179
moneypunct::thousands_sep.	180
moneypunct::groupint	180
moneypunct::curr_symbol.	180
moneypunct::positive_sign	180
moneypunct::negative_sign	181
moneypunct::frac_digits.	181
moneypunct::pos_format	181
moneypunct::neg_format	181
Class money_put	182
Typedef Declarations money_put.	182
char_type	182
iter_type	182
string	182
ios	182
Public Member Functions money_put.	182

money_put::put	183
Class money_get	183
Typedef Declarations money_get	183
char_type	183
iter_type	184
string	184
ios	184
Public Member Functions money_get	184
money_get::get.	184

8 Containers Library 187

Overview of the Containers Library	187
What Are Containers	187
Basic Design and Organization of Containers	188
Common Members of All Containers	189
Common Type Definitions in All Containers	190
Common Member Functions in all Containers	192
Sequence Container Requirements	195
Associative Container Requirements	197
Basic Design and Organization.	197
Equality of Keys	197
Additional Definitions	197
Associative Container Types and Member Functions	198
Organization of the Container Class Descriptions	202
Template class vector<T>	203
Type Definitions vector	204
Constructors, Destructors and	
Related Functions vector	206
Comparison Operations vector.	208
Element Access Member Functions vector	208
Insert Member Functions vector	210
Erase Member Functions vector	211
Notes on Insert and Erase Member Functions vector	212
Specialization Class vector<bool>	212
Public Member Functions	213
Template class deque<T>	213

Typedef Declarations deque	214
Constructors, Destructors and Related Functions deque . . .	215
Comparison Operations deque	217
Element Access Member Functions deque	217
Insert Member Functions deque	219
Erase Member Functions deque	220
Deque Class Notes	221
Storage Management	221
Complexity of Insertion	221
Notes on Erase Member Functions	222
Template class list<T>	222
Typedef Declarations list	223
Constructors, Destructors and Related Functions list . . .	225
Comparison Operations list	226
Element Access Member Functions list	227
Insert Member Functions list.	228
Erase Member Functions list.	229
Special Operations list	230
List Class Notes	232
Notes on Insert Member Functions	232
Notes on Erase Member Functions	232
Template class set<T>	233
Typedef Declarations set	233
Constructors, Destructors and Related Functions set . . .	236
Comparison Operations set	237
Element Access Member Functions set	238
Insert Member Functions set.	239
Erase Member Functions set	240
Special Operations set.	241
Template class multiset<Key>	242
Typedef Declarations multiset	243
Constructors, Destructors and Related Functions multiset . .	246
Comparison Operations multiset	247
Element Access Member Functions multiset	248
Insert Member Functions multiset	249

Erase Member Functions multiset	250
Special Operations multiset	251
Template class map<Key, T>	252
Typedef Declarations map	253
Constructors, Destructors and Related Functions map.	256
Comparison Operations map	257
Element Access Member Functions map.	258
Insert Member Functions map	260
Erase Member Functions map	261
Special Operations map	261
Template class multimap<Key, T>	263
Typedef Declarations multimap	264
Constructors, Destructors and Related Functions multimap	267
Comparison Operations multimap	268
Element Access Member Functions multimap	269
Insert Member Functions multimap.	270
Erase Member Functions multimap.	271
Special Operations multimap	272
Template class stack.	274
Public Member Functions stack	274
Comparison Operations stack	275
Template class queue	276
Public Member Functions queue	277
Comparison Operations queue.	278
Template class priority_queue	278
Constructors priority_queue.	279
Public Member Functions priority_queue	280
Comparison Operations priority_queue	281

9 Iterators Library 283

Overview of Iterators	283
Value type.	284
Distance type	284
Past-the-end values.	284
Dereferenceable values.	284

Singular values. 284
Reachability 285
Ranges 285
Mutable versus constant 285
Iterator Requirements 285
Input Iterator Requirements 286
Output Iterator Requirements 287
Forward Iterator Requirements. 288
Bidirectional Iterator Requirements. 289
Random Access Iterator Requirements 289
Stream Iterators 290
Template class istream_iterator 291
Constructor istream_iterator. 292
Default Constructor 292
Overloaded and Copy Constructors 292
Destructor. 292
Public Member Functions istream_iterator. 293
Dereferencing Operator * 293
Incrementation Operator ++ 293
Comparison Operations istream_iterator 293
Equality Operator == 293
Template class ostream_iterator. 294
Constructor ostream_iterator 294
Default Constructor 294
Overloaded and Copy Constructors 295
Destructor. 295
Public Member Functions ostream_iterator 295
Dereferencing Operator * 295
Assignment Operator = 295
Incrementation Operator ++ 295
Template class istreambuf_iterator 296
Typedef Declarations istreambuf_iterator 297
char_type 297
traits_type. 297
streambuf 297

istream 297
Placeholder proxy istreambuf_iterator 297
Constructor istreambuf_iterator 298
Default Constructor 298
Overloaded and Copy Constructor 298
Operators istreambuf_iterator 298
Dereferencing Operator * 298
Incrementation Operator ++ 298
istream_iterator::equal 299
istream_iterator::iterator_category 299
Equality Operator == 299
Not Equal Operator != 300
Template class ostreambuf_iterator 300
Typedef Declarations ostreambuf_iterator 300
char_type 301
traits_type. 301
streambuf 301
ostream 301
Constructor ostreambuf_iterator 301
Default Constructor 301
Overloaded and Copy Constructors 301
Operators ostreambuf_iterator 302
Dereferencing Operator * 302
ostreambuf_iterator::equal. 302
ostreambuf_iterator::iterator_category 302
Equality Operator == 302
Not Equal Operator != 302
Template class reverse_bidirectional_iterator 303
Constructor reverse_bidirectional_iterator 304
Default Constructor 304
Overloaded Constructors 304
Public Member Functions reverse_bidirectional_iterator. 304
reverse_bidirectional_iterator::base 304
Dereferencing Operator * 304
Incrementation Operator ++ 304

Decrementation Operator --	305
Equality Operator ==	305
Template class reverse_iterator	306
Constructor reverse_iterator	306
Default Constructor	306
Overloaded and Copy Constructors	306
Public Member Functions reverse_iterator	307
reverse_iterator::base	307
Dereferencing Operator *	307
Incrementation Operator ++	307
Decrementation Operator --	307
Add Operator +	308
Add & Assign Operator +=	308
Minus Operator -.	308
Minus & Assign Operator -=	308
Subset Operator []	309
Equality Operator ==	309
Less Than Operator <.	309
Minus Operator -.	310
Add Operator +	310
Template class back_insert_iterator	310
Constructor back_insert_iterator	311
Copy Constructor	311
Public Member Functions back_insert_iterator	311
Assignment Operator =	311
Dereferencing Operator *	311
Incrementation Operator ++	311
back_insert_iterator::back_inserter	312
Template class front_insert_iterator	312
Constructor front_insert_iterator	312
Copy Constructor	312
Public Member Functions front_insert_iterator	312
Assignment Operator =	312
Dereferencing Operator *	312
Incrementation Operator ++	313

front_insert_iterator::front_inserter	313
Template class insert_iterator.	313
Constructor insert_iterator.	313
Copy Constructor	313
Public Member Function insert_iterator	314
Assignment Operator =	314
Dereferencing Operator *	314
Incrementation Operator ++	314
insert_iterator::inserter	314

10 Algorithms Library 315

Overview of the Algorithms Library.	315
In-place and Copying Versions	315
Algorithms with Predicate Parameters	316
Binary Predicates.	316
Non Mutating Sequence Algorithms	317
for_each.	317
find.	318
find_if	318
adjacent_find	319
count	320
count_if.	320
mismatch	321
equal	322
search	323
Mutating Sequence Algorithms.	324
copy	325
copy_backward	325
swap	326
iter_swap	326
swap_ranges.	327
transform	327
replace	328
replace_if	329
replace_copy	329

replace_copy_if 330
fill 330
fill_n 331
generate. 331
generate_n 332
remove 332
remove_if 333
remove_copy 334
remove_copy_if 334
unique 335
unique_copy. 336
reverse 337
reverse_copy 337
rotate 337
rotate_copy 338
random_shuffle 339
partition 339
stable_partition 340
Sorting and Related Algorithms 341
Sorting 342
sort. 342
stable_sort. 343
partial_sort 343
partial_sort_copy. 344
nth_element 345
Binary Searching 346
binary_search 346
lower_bound 347
upper_bound 348
equal_range 349
Merging. 350
merge. 350
inplace_merge 351
Set Operations on Sorted Structures. 352
includes. 353

set_union 353
set_intersection 354
set_difference 355
set_symmetric_difference 356
Heap Operations 357
push_heap 358
pop_heap 359
make_heap 359
sort_heap 360
Finding Min and Max. 361
min. 361
max 362
min_element. 362
max_element 363
Lexicographical Comparison. 364
lexicographical_compare 364
Permutation Generators. 365
next_permutation 365
prev_permutation 366
Generalized Numeric Algorithms. 366
accumulate 367
inner_product 367
partial_sum 369
adjacent_difference 370

11 Numerics Library 373

Overview of the Numerics Library 373
Numeric Type Requirements 373
Numeric Arrays 374
Template Class valarray 375
Constructors valarray. 375
Default Constructor 375
Overloaded Constructors 376
Copy Constructor 376
Conversion Constructors 376

Destructor. 377
Assignment Operators valarray 377
Assignment Operator = 377
Overloaded Assignment Operators 377
Element Access valarray 378
Subscript Operator [] 378
Subset Operations valarray 378
Subscript Operator [] 378
Unary Operators valarray 378
Add Operator + 379
Minus Operator -. 379
Complement Operator ~ 379
Not Operator !. 379
Computed Assignment Type valarray<T> 379
Multiply & Assign Operator *=. 379
Divide and Assign Operator /= 380
Remainder & Assign Operator %= 380
Add & Assign Operator += 380
Minus and Assign Operator -= 380
XOR and Assign Operator ^= 380
And & Assign Operator &= 380
Or & Assign Operator = 380
Not Equal Operator != 380
Right Shift & Assign Operator >>= 380
Computed Assignment Type<T> 381
Multiply & Assign Operator *=. 381
Divide & Assign Operator /= 381
Remainder & Assign Operator %= 381
Add & Assign Operator += 381
Minus & Assign Operator -=. 381
XOR & Assign Operator ^= 381
And & Assign Operator &= 382
Not Equal Operator != 382
Right Shift & Assign Operator >>= 382
Non-Member Binary Operators valarray 382

Multiply Operator *	. 382
Divide Operator /	. 382
Remainder Operator %	. 383
Add Operator +	. 383
Minus Operator -	. 383
Bitwise XOR Operator ^	. 383
Bitwise And Operator &	. 384
Bitwise Or Operator	. 384
Left Shift Operator <<	. 384
Right Shift Operator >>	. 384
Logical And Operator &&	. 384
Logical Or Operator	. 385
Overloaded Binary Operators valarray	. 385
Multiply Operator *	. 385
Divide Operator /	. 385
Remainder Operator %	. 386
Add Operator +	. 386
Minus Operator -	. 386
Bitwise XOR Operator ^	. 387
Bitwise And Operator &	. 387
Bitwise Or Operator	. 387
Left Shift Operator <<	. 388
Right Shift Operator >>	. 388
Logical And Operator &&	. 388
Logical Or Operator	. 389
Comparison Operators valarray	. 389
Equality Operator ==	. 389
Less Than Operator <	. 390
Greater Than Operator >	. 390
Not Equal Operator !=	. 390
Greater Than or Equal Operator >=	. 390
Overloaded Comparison Operators valarray	. 391
Equality Operator ==	. 391
Not Equal Operator !=	. 391
Less Than Operator <	. 392

Greater Than Operator > 392
Less Than or Equal Operator <= 392
Greater Than or Equal Operator >= 393
Member Functions valarray 393
valarray::length 393
Pointer Conversion 393
valarray::sum 394
valarray::fill 394
valarray::shift 394
valarray::cshift 394
valarray::apply 395
valarray::free 395
Min And Max Functions valarray 395
valarray::min 395
valarray::max 396
Transcendentals valarray 396
valarray::abs 396
valarray::acos 396
valarray::asin 397
valarray::atan 397
valarray::atan2 397
valarray::cos 397
valarray::cosh 398
valarray::exp 398
valarray::log 398
valarray::log10 398
valarray::pow 398
valarray::sin 399
valarray::sinh 399
valarray::sqrt 399
valarray::tan 399
valarray::tanh 399
Class slice 400
Constructors slice 400
Overloaded and Copy Constructors 400

Access Functions slice.	401
slice::start	401
slice::length	401
slice::stride	401
Template class slice_array	401
Constructors slice_array.	402
Default and Copy Constructor	402
Assignment Operators slice_array	402
Assignment Operator =	402
Computed Assignment slice_array	402
Multiply & Assign Operator *=.	402
Divide & Assign Operator /=	403
Remainder & Assign Operator %=	403
Add & Assign Operator +=	403
Minus & Assign Operator -=.	403
XOR & Assign Operator ^=	403
And & Assign Operator &=	403
Or & Assign Operator =	403
Left Shift & Assign Operator <<=.	403
Right shift & Assign Operator >>=	403
Public Member Function slice_array	404
slice_array::fill	404
Class gslice	404
Constructors gslice	404
Default Constructor	404
Overloaded Constructors	405
Access Functions gslice	405
gslice::start	405
gslice::length.	405
gslice::stride	405
Template Class gslice_array	405
Constructors gslice_array	406
Default and Copy Constructors	406
Assignment gslice_array	406
Assignment Operator =	406

Computed Assignment gslice_array	406
Multiply & Assign Operator *=.	407
Divide & Assign Operator /=	407
Remainder & Assign Operator %=	407
Add & Assign Operator +=	407
Minus & Assign Operator -=.	407
XOR & Assign Operator ^=	407
And & Assign Operator &=	407
Or & Assign Operator =	407
Left Shift & Assign Operator <<=.	407
Right Shift & Assign Operator >>=	408
Public Member Function gslice_array	408
gslice_array::fill	408
Template Class mask_array	408
Constructors mask_array	409
Constructors.	409
Assignment mask_array.	409
Assignment Operator =	409
Computed Assignment mask_array.	409
Multiply & Assign Operator *=.	409
Divide & Assign Operator /=	409
Remainder & Assign Operator %=	410
Add & Assign Operator +=	410
Minus & Assign Operator -=.	410
XOR & Assign Operator ^=	410
And & Assign Operator &=	410
Or & Assign Operator =	410
Left Shift & Assign Operator <<=.	410
Right Shift & Assign Operator >>=	410
Public Member Function mask_array	411
mask_array::fill	411
Template Class indirect_array	411
Constructors indirect_array	411
Default and Copy Constructors	411
Assignment indirect_array	412

Assignment Operator =	412
Computed Assignment indirect_array.	412
Multiply & Assign Operator *=.	412
Divide & Assign Operator /=	412
Remainder & Assign Operator %=	412
Add & Assign Operator +=	413
Minus & Assign Operator -=.	413
XOR & Assign Operator ^=	413
And & Assign Operator &=.	413
Or & Assign Operator =	413
Left Shift & Assign Operator <<=.	413
Right Shift & Assign Operator >>=.	413
Public Member Function indirect_array	413
indirect_array::fill	413
Generalized Numeric Operations	414
Template Class accumulate.	414
Template Class inner_product	415
Template Class partial_sum	415
Template Class adjacent_difference	416

12 26.2 Complex Class 419

Overview of the Complex Class Library	419
_MSL_CX_LIMITED_RANGE	419
Header <complex>	420
26.2.3 Complex Specializations	424
Complex Template Class.	426
Constructors and Assignments.	426
Complex Member Functions.	427
Operators	427
Overloaded Operators and Functions	429
Complex Operators.	430
Complex Value Operations	433
Complex Transcendentals	435

13 27.1 Input and Output Library 439

Overview of Input and Output Library	439
------------------------------------------------	-----

Input and Output Library Summary	439
27.1 Iostreams requirements	440
27.1.1 Definitions	440
27.1.2 Type requirements	441
27.1.2.5 Type SZ_T	441
14 27.2 Forward Declarations	443
Overview of Input and Output Streams Forward Declarations.	443
Header <iosfwd>	443
15 27.3 Standard Iostream Objects	445
Overview of Standard Input and Output Stream Objects.	445
Header <iostream>	445
27.3.1 Narrow stream objects.	446
istream cin	446
ostream cout.	446
ostream cerr	446
ostream clog.	447
27.3.2 Wide stream objects	447
istream win	447
ostream wout	447
wostream werr.	448
wostream wlog	448
16 27.4 Iostreams Base Classes	449
Overview of Input and Output Stream Base Classes	449
Header <ios>	449
27.4.1 Typedef Declarations	450
27.4.3 Class ios_base	451
27.4.3.1 Typedef Declarations	453
27.4.3.1.1 failure	453
27.4.3.1.1.1 failure	454
failure::what.	454
27.4.3.1.2 Type fmtflags	454
27.4.3.1.3 Type iostate	456
27.4.3.1.4 Type openmode	456

27.4.3.1.5 Type seekdir	457
27.4.3.1.6 Class Init	457
Class Init Constructor.	457
27.4.3.2 ios_base fmtflags state functions.	458
flags	458
setf	461
unsetf.	462
precision	463
width.	464
27.4.3.3 ios_base locale functions	465
imbue	465
getloc.	466
27.4.3.4 ios_base storage function.	466
xalloc.	466
iword.	466
pword	467
register_callback	467
sync_with_stdio	468
27.4.3.5 ios_base Constructor.	468
27.4.4 Template class basic_ios	468
27.4.4.1 basic_ios Constructor	470
27.4.4.2 Member Functions	471
tie	471
rdbuf	473
imbue	474
fill	474
copyfmt.	475
27.4.4.3 basic_ios iostate flags functions	476
operator bool	476
operator !	476
rdstate	476
clear	478
setstate	480
good	481
eof	481

fail 483
bad. 484
exceptions. 486
27.4.5 ios_base manipulators 486
27.4.5.1 fmtflags manipulators 486
27.4.5.2 adjustfield manipulators 488
27.4.5.3 basefield manipulators. 488
27.4.5.4 floatfield manipulators. 489
Overloading Manipulators 490

17 27.5 Stream Buffers 493

Overview of Stream Buffers 493
Header <streambuf> 493
27.5.1 Stream buffer requirements. 493
27.5.2 Template class basic_streambuf<charT, traits> 494
27.5.2.1 basic_streambuf Constructor 497
27.5.2.2 basic_streambuf Public Member Functions 498
27.5.2.2.1 Locales 498
basic_streambuf::pubimbue 498
basic_streambuf::getloc 498
27.5.2.2.2 Buffer Management and Positioning 498
basic_streambuf::pubsetbuf 499
basic_streambuf::pubseekoff. 500
basic_streambuf::pubseekpos 501
basic_streambuf::pubsync 503
27.5.2.2.3 Get Area 504
basic_streambuf::in_avail 504
basic_streambuf::snextc 504
basic_streambuf::sbumpc 505
basic_streambuf::sgetc 506
basic_streambuf::sgetn 507
27.5.2.2.4 Putback 507
basic_streambuf::sputback. 507
basic_streambuf::sungetc 509
27.5.2.2.5 Put Area 509

basic_streambuf::sputc	509
basic_streambuf::sputn	510
27.5.2.3 basic_streambuf Protected Member Functions.	511
27.5.2.3.1 Get Area Access	511
basic_streambuf::eback	511
basic_streambuf::gptr	511
basic_streambuf::egptr	511
basic_streambuf::gbump	512
basic_streambuf::setg	512
27.5.2.3.2 Put Area Access	512
basic_streambuf::pbase	512
basic_streambuf::pptr	513
basic_streambuf::epptr	513
basic_streambuf::pbump	513
basic_streambuf::setp	513
27.5.2.4 basic_streambuf Virtual Functions.	514
27.5.2.4.1 Locales	514
basic_streambuf::imbue	514
27.5.2.4.2 Buffer Management and Positioning	514
basic_streambuf::setbuf	514
basic_streambuf::seekoff	515
basic_streambuf::seekpos	515
basic_streambuf::sync.	515
27.5.2.4.3 Get Area	516
basic_streambuf::showmanc	516
basic_streambuf::xsgetn	516
basic_streambuf::underflow	516
basic_streambuf::uflow	517
27.5.2.4.4 Putback	518
basic_streambuf::pbackfail.	518
27.5.2.4.5 Put Area	518
basic_streambuf::xspn	518
basic_streambuf::overflow	519

18 27.6 Formatting And Manipulators	521
Overview of Formatting and Manipulators.	521
Headers	521
Header <istream>	521
Header <ostream>	522
Header <iomanip>	523
27.6.1 Input Streams.	523
27.6.1.1 Template class basic_istream	523
27.6.1.1.1 basic_istream Constructors	526
27.6.1.1.2 Class basic_istream::sentry	527
Class basic_istream::sentry Constructor	528
sentry::Operator bool	528
27.6.1.2 Formatted input functions	529
27.6.1.2.1 Common requirements	529
27.6.1.2.2 Arithmetic Extractors Operator >>	529
27.6.1.2.3 basic_istream extractor operator >>	530
Overloading Extractors:	533
27.6.1.3 Unformatted input functions	535
basic_istream::gcount	535
basic_istream::get	537
basic_istream::getline	540
basic_istream::ignore	542
basic_istream::peek	544
basic_istream::read	544
basic_istream::readsome.	546
basic_istream::putback	547
basic_istream::unget	549
basic_istream::sync	551
basic_istream::tellg	552
basic_istream::seekg	552
27.6.1.4 Standard basic_istream manipulators	554
basic_ifstream::ws	554
27.6.1.4.1 basic_iostream Constructor	556
27.6.2 Output streams	557
27.6.2.1 Template class basic_ostream	557

27.6.2.2 basic_ostream Constructor	559
27.6.2.3 Class basic_ostream::sentry	560
Class basic_ostream::sentry Constructor	561
sentry::Operator bool	561
27.6.2.4 Formatted output functions.	562
27.6.2.4.1 Common requirements	562
27.6.2.4.2 Arithmetic Inserter Operator <<	562
27.6.2.4.3 basic_ostream::operator<<	564
Overloading Inserters.	566
27.6.2.5 Unformatted output functions	568
basic_ostream::tellp.	568
basic_ostream::seekp	569
basic_ostream::put	570
basic_ostream::write	571
basic_ostream::flush	573
27.6.2.6 Standard basic_ostream manipulators	575
basic_ostream:: endl	576
basic_ostream::ends	576
basic_ostream::flush	577
27.6.3 Standard manipulators.	580
Standard Manipulator Instantiations	580
resetiosflags	580
setiosflags	581
:setbase	582
setfill	583
setprecision	584
setw	585
Overloaded Manipulator	586

19 27.7 String Based Streams 589

Overview of String Based Stream Classes	589
Header <sstream>	589
27.7.1 Template class basic_stringbuf.	590
27.7.1.1 basic_stringbuf constructors	591
27.7.1.2 Member functions.	593

basic_stringbuf::str	593
27.7.1.3 Overridden virtual functions	594
basic_stringbuf::underflow	594
basic_stringbuf::pbackfail	595
basic_stringbuf::overflow	595
basic_stringbuf::seekoff	596
basic_stringbuf::seekpos.	596
27.7.2 Template class basic_istream.	597
27.7.2.1 basic_istream constructors.	598
27.7.2.2 Member functions.	599
basic_istream::rdbuf	599
basic_istream::str	600
27.7.2.3 Class basic_ostringstream.	601
27.7.2.4 basic_ostringstream constructors.	602
27.7.2.5 Member functions.	604
basic_ostringstream::rdbuf.	604
basic_ostringstream::str	606
27.7.3 Class basic_stringstream	607
27.7.3.4 basic_stringstream constructors	608
27.7.3.5 Member functions.	610
basic_stringstream::rdbuf	610
basic_stringstream::str	611

20 27.8 File Based Streams 613

Overview of File Based Streams	613
Header <fstream>	613
27.8.1 File streams	613
27.8.1.1 Template class basic_filebuf	614
27.8.1.2 basic_filebuf Constructors	616
27.8.1.3 Member functions.	616
basic_filebuf::is_open	616
basic_filebuf::open	617
basic_filebuf::close	619
27.8.1.4 Overridden virtual functions	619
basic_filebuf::showmanyc	619

basic_filebuf::underflow	620
basic_filebuf::pbackfail	620
basic_filebuf::overflow	620
basic_filebuf::seekoff	621
basic_filebuf::seekpos	621
basic_filebuf::setbuf	621
basic_filebuf::sync	622
basic_filebuf::imbue	622
27.8.1.5 Template class basic_ifstream	622
27.8.1.6 basic_ifstream Constructor	623
27.8.1.7 Member functions	625
basic_ifstream::rdbuf	625
basic_ifstream::is_open	626
basic_ifstream::open	626
basic_ifstream::close	628
27.8.1.8 Template class basic_ofstream	629
27.8.1.9 basic_ofstream constructor	630
27.8.1.10 Member functions	631
basic_ofstream::rdbuf	631
basic_ofstream::is_open	633
basic_ofstream::open	633
basic_ofstream::close	635
27.8.1.11 Template class basic_fstream	636
27.8.1.12 basic_fstream Constructor	637
27.8.1.13 Member Functions	638
basic_fstream::rdbuf	638
basic_fstream::is_open	640
basic_fstream::open	640
basic_fstream::close	641

21 C Library files 643

27.8.2 C Library files	643
<stdio> Macros	643
<stdio> Types	643
<stdio> Functions	644

22 Annex D Strstream	645
Overview of Strstream Classes 645
Header <strstream>. 645
Strstreambuf Class 648
Strstreambuf constructors and Destructors. 650
Strstreambuf Public Member Functions 651
freeze. 651
pcount 652
str 653
Protected Virtual Member Functions 654
setbuf. 654
seekoff 655
seekpos 655
underflow. 655
pbackfail 656
overflow 656
Istrstream Class 657
Constructors and Destructor. 658
Constructors. 658
Destructor. 659
Public Member Functions 659
rdbuf 659
str 660
Ostrstream Class 660
Constructors and Destructor. 661
Constructors. 661
Destructor. 662
Public Member Functions 663
freeze. 663
pcount 664
rdbuf 665
str 665
Strstream Class. 666
Strstream Types 667
Constructors and Destructor. 667

Constructors. 667
Destructor. 667
Public Member Functions 667
freeze. 667
pcount 668
rdbuf 668
str 668

23 Mutex.h 669

Overview of Mutex Support Library 669
Header <mutex.h> 669
null_mutex 672
Constructor and Destructor 672
Constructor 672
Destructor. 672
Public Member Functions 673
remove 673
acquire 673
release 673
mutex. 673
Constructor and Destructor 674
Constructor 674
Destructor. 674
Public Member Functions 674
remove 674
acquire 674
release 674
try_lock. 675
mutex_block 675
Constructor and Destructors. 675
Constructor 675
Destructor. 675
Public Member Functions 676
remove 676
release 676

acquire 676
mutex_arith 676
Constructors. 677
Public Member Operators 677
Prefix operator++ 677
Postfix operator++ 678
operator+= 678
Prefix operator-- 678
Postfix operator-- 678
operator-= 678
operator== 678
operator!= 679
operator>= 679
operator> 679
operator<= 679
operator< 680
operator= 680
operator TYPE 680
rw_mutex 680
Constructor and Destructor 681
Constructor 681
Destructor. 681
Public Member Functions 681
remove 681
acquire 681
release 681
try_rdlock 682
try_wrlock 682
mutex_rec 682
Constructor 683
Public Members Functions. 683
acquire 683
release 683
remove 683



Introduction

This reference manual describes the contents of the C++ standard library and what Metrowerks' library provides for its users. The C++ Standard library provides an extensible framework, and contains components for: language support, diagnostics, general utilities, strings, locales, containers, iterators, algorithms, numerics, and input/output.

CodeWarrior Year 2000 Compliance

The Products provided by Metrowerks under the License agreement process dates only to the extent that the Products use date data provided by the host or target operating system for date representations used in internal processes, such as file modifications. Any Year 2000 Compliance issues resulting from the operation of the Products are therefore necessarily subject to the Year 2000 Compliance of the relevant host or target operating system. Metrowerks directs you to the relevant statements of Microsoft Corporation, Sun Microsystems, Inc., Apple Computer, Inc., and other host or target operating systems relating to the Year 2000 Compliance of their operating systems. Except as expressly described above, the Products, in themselves, do not process date data and therefore do not implicate Year 2000 Compliance issues.

For additional information, visit: <http://www.metrowerks.com/about/y2k.html>.

About the MSL C++ Library Reference Manual

This section describes each chapter in this manual.

[“Overview of the MSL C++ Reference” on page 43](#), of this manual describes the language support library that provides components

Introduction

About the MSL C++ Library Reference Manual

that are required by certain parts of the C++ language, such as memory allocation and exception processing.

[“Overview of Language Support Libraries” on page 57](#), discusses the ANSI/ISO language support library.

[“Overview of Diagnostics Library” on page 73](#), elaborates on the diagnostics library that provides a consistent framework for reporting errors in a C++ program, including predefined exception classes.

[“Overview of General Utilities Libraries” on page 81](#), discusses the general utilities library, which includes components used by other library elements, such as predefined storage allocator for dynamic storage management.

[“Overview of Strings Classes” on page 105](#), discusses the strings components provided for manipulating text represented as sequences of type `char`, sequences of type `wchar_t`, or sequences of any other “character-like” type.

[“Overview of the Localization Library” on page 147](#), covers the localization components extend internationalization support for character classification, numeric, monetary, and date/time formatting and parsing among other things.

[“Overview of the Containers Library” on page 187](#), discusses container classes: lists, vectors, stacks, and so forth. These classes provide a C++ program with access to a subset of the most widely used algorithms and data structures.

[“Overview of Iterators” on page 283](#), discusses iterator classes.

[“Overview of the Algorithms Library” on page 315](#), discusses the algorithms library. This library provides sequence, sorting, and general numerics algorithms.

[“Overview of the Numerics Library” on page 373](#), discusses the numerics library. It describes the components for complex number types, numeric arrays, generalized numeric algorithms and facilities included from the ISO C library.

[“Overview of the Complex Class Library” on page 419](#) discusses the template class complex, for complex number storage and manipulation.

[“Overview of Input and Output Library” on page 439](#), overviews the input and output class libraries.

[“Overview of Input and Output Streams Forward Declarations.” on page 443](#), discusses the input and output streams forward declarations.

[“Overview of Standard Input and Output Stream Objects.” on page 445](#), discusses the initialized input and output objects.

[“Overview of Input and Output Stream Base Classes” on page 449](#), discusses the `iostream_base` class.

[“Overview of Stream Buffers” on page 493](#), discusses the stream buffer classes.

[“Overview of Formatting and Manipulators” on page 521](#), discusses the formatting and manipulator classes.

[“Overview of String Based Stream Classes” on page 589](#), discusses the string based stream classes.

[“Overview of File Based Streams” on page 613](#), discusses the file based stream classes.

[“27.8.2 C Library files” on page 643](#), discusses the namespace C library functions.

[“Overview of Mutex Support Library” on page 669](#), discusses the mutex classes.

[“Overview of Strstream Classes” on page 645](#), a non-templated array based stream class.

Introduction

About the MSL C++ Library Reference Manual



17 C++ Library

This chapter is an introduction to the Metrowerks Standard C++ library.

Overview of the MSL C++ Reference

This section introduces you to the definitions, conventions, terminology, and other aspects of the MSL C++ library. The topics discussed are:

- [“17.1 Definitions” on page 43](#) standard C++ terminology
- [“17.2 Additional Definitions” on page 47](#) additional terminology
- [“17.3 Methods of Descriptions” on page 47](#) standard conventions
- [“17.4 Library-wide Requirements” on page 49](#) library requirements
- [“Features not implemented in MSL C++” on page 55](#) Standard C++ features not yet implemented in Metrowerks Standard Libraries

17.1 Definitions

This section discusses the meaning of certain terms in the MSL C++ library.

- [“17.1.1 Arbitrary-Positional Stream” on page 44](#)
- [“17.1.2 Character” on page 44](#)
- [“17.1.3 Character Sequences” on page 44](#)
- [“17.1.4 Comparison Function” on page 44](#)
- [“17.1.5 Component” on page 45](#)

- [“17.1.6 Default Behavior” on page 45](#)
- [“17.1.7 Handler Function” on page 45](#)
- [“17.1.8 Iostream Class Templates” on page 45](#)
- [“17.1.9 Modifier Function” on page 45](#)
- [“17.1.10 ObjectState” on page 45](#)
- [“17.1.11 Narrow-oriented Iostream Classes” on page 45](#)
- [“17.1.12 NTCTS” on page 46](#)
- [“17.1.13 Observer Function” on page 46](#)
- [“17.1.14 Replacement Function” on page 46](#)
- [“17.1.15 Required Behavior” on page 46](#)
- [“17.1.16 Repositional Stream” on page 46](#)
- [“17.1.17 Reserved Function” on page 46](#)
- [“17.1.18 Traits” on page 46](#)
- [“17.1.19 Wide-oriented Iostream Classes” on page 47](#)

17.1.1 Arbitrary-Positional Stream

A stream that can seek to any position within the length of the stream. An arbitrary-positional stream is also a repositional stream

17.1.2 Character

Any object which, when treated sequentially, can represent text. A character can be represented by any type that provides the definitions specified.

17.1.3 Character Sequences

A class or a type used to represent a character. A character container class shall be a POD type.

17.1.4 Comparison Function

An operator function for equality or relational operators.

17.1.5 Component

A group of library entities directly related as members, parameters, or return types. For example, a class and a related non-member template function entity would be referred to as a component.

17.1.6 Default Behavior

The specific behavior provided by the implementation, for replacement and handler functions.

17.1.7 Handler Function

A non-reserved function that may be called at various points with a program through supplying a pointer to the function. The definition may be provided by a C++ program.

17.1.8 `iostream` Class Templates

Templates that take two template arguments: `charT` and `traits`. `CharT` is a character container class, and `traits` is a structure which defines additional characteristics and functions of the character type.

17.1.9 Modifier Function

A class member function other than constructors, assignment, or destructor, that alters the state of an object of the class.

17.1.10 `ObjectState`

The current value of all non-static class members of an object.

17.1.11 Narrow-oriented `iostream` Classes

The instantiations of the `iostream` class templates on the character container class. Traditional `iostream` classes are regarded as the narrow-oriented `iostream` classes.

17.1.12 NTCTS

Null Terminated Character Type Sequences. Traditional char strings are NTCTS.

17.1.13 Observer Function

A const member function that accesses the state of an object of the class, but does not alter that state.

17.1.14 Replacement Function

A non-reserved C++ function whose definition is provided by a program. Only one definition for such a function is in effect for the duration of the program's execution.

17.1.15 Required Behavior

The behavior for any replacement or handler function definition in the program replacement or handler function. If a function defined in a C++ program fails to meet the required behavior when it executes, the behavior is undefined.

17.1.16 Repositional Stream

A stream that can seek only to a position that was previously encountered.

17.1.17 Reserved Function

A function, specified as part of the C++ Standard Library, that must be defined by the implementation. If a C++ program provides a definition for any reserved function, the results are undefined.

17.1.18 Traits

A class that encapsulates a set of types and functions necessary for template classes and template functions to manipulate objects of types for which they are instantiated.

17.1.19 Wide-oriented Iostream Classes

The instantiations of the `iostream` class templates on the character container class `wchar_t` and the default value of the traits parameter.

17.2 Additional Definitions

Metrowerks Standard Libraries have additional definitions.

- [“Multi-Thread Safety” on page 47](#) precautions used with multi-threaded systems.

Multi-Thread Safety

MSL C++ Library is multi-thread safe provided that the operating system supports thread-safe system calls. Library has locks at appropriate places in the code for thread safety. The locks are implemented as a mutex class -- the implementation of which may differ from platform to platform.

This ensures that the library is MT-Safe internally. For example, if a buffer is shared between two ios class objects, then only one ios object will be able to modify the shared buffer at a given time.

Thus the library will work in the presence of multiple threads in the same way as in single thread provided the user does not share objects between threads or locks between accesses to objects that are shared.

17.3 Methods of Descriptions

Conventions used to describe the C++ Standard Library.

17.3.1 Structure of each subclause

This document follows the Standard C++ convention and numbers the library chapters by the subclause numbers.

Table 2.1 Chapter Descriptions

Chapter	Description	Chapter	Description
18	Language Support	23	Containers
19	Diagnostics	24	Iterators
20	General utilites	25	Algorithms
21	Strings	26	Numerics
22	Localizations	27	Input/Output

17.3.1.1 Summary

The Metrowerks Standard Library descriptions include a short description, notes, remarks, cross references and examples of usage.

17.3.2 Other Conventions

Some other terminology and conventions used in this reference are:

17.3.2.1.1 Character sequences

A letter is any of the 26 lowercase or 26 uppercase letters

The decimal-point character is represented by a period, '.

A character sequence is an array object of the types `char`, `unsigned char`, or `signed char`.

A character sequence can be designated by a pointer value `S` that points to its first element.

17.3.2.1.3.1 Byte strings

A null-terminated byte string, or NTBS, is a character sequence whose highest-addressed element with defined content has the value zero (the terminating null character).

The length of an NTBS is the number of elements that precede the terminating null character. An empty NTBS has a length of zero.

³The value of an NTBS is the sequence of values of the elements up to and including the terminating null character.

A static NTBS is an NTBS with static storage duration.

17.3.2.1.3.2 Multibyte strings

A null-terminated multibyte string, or NTMBS, is an NTBS that consists of multibyte characters,

A static NTMBS is an NTMBS with static storage duration.

17.3.2.1.3.3 Wide-character sequences

A wide-character sequence is an array object of type `wchar_t`

A wide character sequence can be designated by a pointer value that designates its first element.

A null-terminated wide-character string, or NTWCS, is a wide-character sequence whose highest addressed element has the value zero.

The length of an NTWCS is the number of elements that precede the terminating null wide character.

An empty NTWCS has a length of zero.

The value of an NTWCS is the sequence of values of the elements up to and including the terminating null character.

A static NTWCS is an NTWCS with static storage duration.

17.3.2.2 Functions within classes

Copy constructors, assignment operators, (non-virtual) destructors or virtual destructors that can be generated by default may not be described

17.3.2.3 Private members

To simplify understanding, where objects of certain types are required by the external specifications of their classes to store data. The declarations for such member objects are enclosed in a comment that ends with exposition only, as in:

```
// streambuf* sb; exposition only
```

17.4 Library-wide Requirements

The requirements that apply to the entire C++ Standard library.

- [“17.4.1 Library contents and organization” on page 50](#)
- [“17.4.2 Using the library” on page 52](#)
- [“17.4.3 Constraints on programs” on page 52](#)
- [“17.4.4 Conforming Implementations” on page 54](#)

17.4.1 Library contents and organization

The Metroweks Standard Libraries are organized in the same fasion as the ANSI/ISO C++ Standard.

17.4.1.1 Library Contents

Definitions are provided for Macros, Values, Types, Templates, Classes, Function and, Objects.

All library entities except macros, operator new and operator delete are defined within the namespace std or namespaces nested within namespace std.

17.4.1.2 Headers

The components of the MSL C++ Library y are declared or defined in various headers.

Table 2.2 MSL C++ Library headers:

C++	Headers	C++	Headers
<algorithm>	<bitset>	<complex>	<deque>
<exception>	<fstream>	<functional>	<iomanip>
<ios>	<iosfwd>	<iostream>	<istream>
<iterator>	<limits>	<list>	<locale>
<map>	<memory>	<new>	<numeric>
<ostream>	<queue>	<set>	<sstream>
<stack>	<stdexcept>	<streambuf>	<string>

<typeinfo>	<utility>	<valarray>	<vector>
C Style	Headers	C Style	Header
<cassert>	<cctype>	<cerrno>	<cfloat>
<ciso646>	<climits>	<locale>	<cmath>
<csetjmp>	<csignal>	<cstdarg>	<cstddef>
<cstdio>	<cstdlib>	<cstring>	<ctime>
<cwchar>	<cwctype>		

Except as may be noted, the contents of each C style header `<cname>` shall be the same as that of the corresponding header `<name.h>`. In the MSL C++ Library the declarations and definitions (except for names which are defined as macros in C) are within namespace scope of the namespace `std`.

NOTE: The names defined as macros in C include: `assert`, `errno`, `offsetof`, `setjmp`, `va_arg`, `va_end`, and `va_start`.

17.4.1.3 Freestanding Implementations

A freestanding implementation has an implementation-defined set of headers. This set shall include at least the following headers.

Table 2.3 MSL C++ Freestanding Implementation Headers

Header	Description
<cstddef>	Types
<limits>	Implementation properties
<cstdlib>	Start and termination
<new>	Dynamic memory management
<typeinfo>	Type identification

17 C++ Library

17.4 Library-wide Requirements

Header	Description
<exception>	Exception handling
<cstdarg>	Other runtime support

The Metrowerks Standard Library header <cstdlib> includes the functions abort(), atexit(), and exit().

17.4.2 Using the library

A description of how a C++ program gains access to the facilities of the C++ Standard Library.

17.4.2.1 Headers

A header's contents are made available to a translation unit when it contains the appropriate #include preprocessing directive.

A translation unit shall include a header only outside of any external declaration or definition, and shall include the header lexically before the first reference to any of the entities it declares or first defines in that translation unit.

17.4.2.2 Linkage

The Metrowerks Standard C++ Library has external "C++" linkage unless otherwise specified

Objects and functions defined in the library and required by a C++ program are included in the program prior to program startup.

17.4.3 Constraints on programs

Restrictions on C++ programs that use the facilities of the Metrowerks Standard C++ Library.

17.4.3.1 Reserved Names

Metrowerks Standard Library reserves certain sets of names and function signatures for its implementation.

Names that contains a double underscore (_ _) or begins with an underscore followed by an upper-case letter is reserved to the MSL library for it's use.

Names that begin with an underscore is reserved to the library for use as a name in the global namespace.

17.4.3.1.3 External Linkage

Each name from the Metrowerks Standard C library declared with external linkage is reserved to the implementation for use as a name with extern "C" linkage, both in namespace std and in the global namespace.

17.4.3.2 Headers

The behavior of any header file with the same name as a Metrowerks Standard Library public or private header, is undefined.

17.4.3.3 Derived classes

Virtual member function signatures defined for a base class in the C++ Standard library may be overridden in a derived class defined in the program.

17.4.3.4 Replacement Functions

If replacement definition occurs prior to program startup replacement functions are allowed.

A C++ program may provide the definition for any of eight dynamic memory allocation function signatures declared in header `<new>`.

```
operator new(size_t)
operator new(size_t, const std::nothrow_t&)
operator new[](size_t)
operator new[](size_t, const std::nothrow_t&)
```

17 C++ Library

17.4 Library-wide Requirements

```
operator delete(void*)
operator delete(void*, const std::nothrow_t&)
operator delete[](void*)
operator delete[](void*, const std::nothrow_t&)
```

17.4.3.5 Handler functions

MSL Standard C++ Library provides default versions of the following handler functions:

```
unexpected_handler *
terminate_handler
```

A C++ program may install different handler functions during execution, by supplying a pointer to a function defined in the program or the library as an argument to:

```
set_new_handler
set_unexpected
set_terminate
```

17.4.3.6 Other functions

In certain cases the Metrowerks Standard C++ Library depends on components supplied by a C++ program. If these components do not meet their requirements, the behavior is undefined.

17.4.3.7 Function arguments

If a C++ library function is passed incorrect but legal arguments the behavior is undefined.

17.4.4 Conforming Implementations

Metrowerks Standard Library is an ANSI/ISO Conforming implementation as described by the ANSI/ISO Standards in section 17.4.4

17.4.4.8 Restrictions On Exception Handling

Any of the functions defined in the Metrowerks Standard C++ Library may report a failure by throwing an exception. No destructor operation defined in the Metrowerks Standard C++ Library will throw an exception.

The Cstyle library functions all have a `throw()` exception-specification. This allows implementations to make performance optimizations based on the absence of exceptions at runtime.

The functions `qsort()` and `bsearch()` meet this condition. In particular, they can report a failure to allocate storage by throwing an exception of type `bad_alloc`, or a class derived from `bad_alloc`.

Features not implemented in MSL C++

The following sections of Standard C++ will not be implemented in the MSL C++ product.

- [“Template Functionality” on page 55](#)
- [“ANSI/ISO Library Functionality” on page 55](#)

Template Functionality

There are minimal areas of the Standard library that depend on C++ features that have so far not been implemented.

ANSI/ISO Library Functionality

The following standard library features are not part of the current release, but are being implemented:

- Messages in Locale

17 C++ Library

Features not implemented in MSL C++



Language Support Library

This chapter is a reference guide to the ANSI/ISO language support library.

Overview of Language Support Libraries

This clause in the April 1995 WPD describes the function signatures that are called implicitly, and the types of objects generated implicitly, during the execution of some C++ programs. This chapter describes the headers that declare these function signatures and define any related types. It also describes common type definitions used throughout the library, characteristics of the predefined types, functions supporting start and termination of a C++ program, support for dynamic memory management, support for dynamic type identification, support for exception processing, and other runtime support.

The topics discussed are:

- [“Types and Macros” on page 57](#)
- [“Numeric Limits” on page 60](#)
- [“Dynamic Memory Storage Allocation Errors” on page 68](#)

Types and Macros

This section discusses types and macros. The topics are:

- [“Macros” on page 58](#)
- [“Type Implementations” on page 58](#)

Macros

The macros defined are:

- [“NULL” on page 58](#)
- [“offsetof” on page 58](#)
- [“sizeof” on page 58](#)

NULL

Description The macro **NULL** is an implementation-defined C++ null-pointer constant in this International Standard. In MSL this evaluates to zero.

Source file `<stddef.h>`

offsetof

Description The macro **offsetof** accepts a restricted set of `type` arguments in this International Standard. `type` shall be a POD¹ structure or a POD union.

Source file `<stddef.h>`

sizeof

Description The **sizeof** operator yields the number of bytes in the object representation of its operand. The operand can be either an expression, which is not evaluated, or a parenthesized `type-id`.

Source file `<stddef.h>`

Type Implementations

There are several fundamental types. They are:

- [“Character types” on page 59](#)

¹A POD structure or union is a union that has no members that are of type pointer to members. The acronym for POD stands for “Plain Old Data”.

- [“Enumeration” on page 59](#)
- [“Integral types” on page 59](#)
- [“Wide character types” on page 60](#)
- [“Bool type” on page 60](#)
- [“Floating point types” on page 60](#)
- [“Void type” on page 60](#)

Character types

Objects declared as characters(**char**) shall be large enough to store any member of the implementation’s basic character set. If a character from this set is stored in a character object, its value shall be equivalent to the integer code of that character. It is implementation-defined whether a char object can take on negative values. Characters can be explicitly declared unsigned or signed. Plain char, signed char, and unsigned char are three distinct types. A char, a signed char, and an unsigned char occupy the same amount of storage and have the same alignment requirements; that is, they have the same object representation. For character types, all bits of the object representation participate in the value representation. For unsigned character types, all possible bit patterns of the value representation represent numbers.

Enumeration

An enumeration comprises a set of named integer constant values, which form the basis for an integral subrange that includes those values. Each distinct enumeration constitutes a different enumerated type. Each constant has the type of its enumeration.

Integral types

There are four signed integer types: signed char, short int, int and long int. In this list, each type provides at least as much storage as those preceding it in the list, but the implementation can otherwise make any of them equal in storage size. Plain ints have the natural size suggested by the machine architecture; the other signed integer types are provided to meet special needs.

For each of the signed integer types, there exists a corresponding unsigned integer type: `unsigned char`, `unsigned short int`, `unsigned int` and `unsigned long int`, each of which occupies the same amount of storage and has the same alignment requirements as the corresponding signed integer type.

Wide character types

Type `wchar_t` is a distinct type whose values can represent distinct codes for all members of the largest extended character set specified among the supported locales.

Bool type

Values of type `bool` are either `true(1)` or `false(0)`. There are no signed, unsigned, short, or long `bool` types or values.

Floating point types

There are three floating point types: `float`, `double`, and `long double`. The type `double` provides at least as much precision as `float`, and the type `long double` provides at least as much as precision as `double`.

Void type

The `void` type has an empty set of values. It is used as the return type for functions that do not return a value.

Numeric Limits

The `numeric_limits` component provides a C++ program with information about various properties of the implementation's representation of the fundamental types. Specializations will be provided for each fundamental type, both floating point and integer, including `bool`. The member `is_specialized` shall be `true` for all such specializations of `numeric_limits`.

The only class discussed is:

- [“Template Class numeric_limits” on page 61](#)

Template Class numeric_limits

This section discusses the numeric_limits template class. The functions discussed are:

Table 3.1 **numeric_limits functions**

numeric_limits::is_specialized	numeric_limits::max_exponent10
numeric_limits::min	numeric_limits::has_infinity
numeric_limits::max	numeric_limits::has_quiet_NaN
numeric_limits::digits	numeric_limits::has_signaling_NaN
numeric_limits::digits10	numeric_limits::has_denorm
numeric_limits::is_signed	numeric_limits::infinity
numeric_limits::is_integer	numeric_limits::quiet_NaN
numeric_limits::is_exact	numeric_limits::signaling_NaN
numeric_limits::radix	numeric_limits::denorm_min
numeric_limits::epsilon	numeric_limits::is_bounded
numeric_limits::round_error	numeric_limits::is_modulo
numeric_limits::min_exponent	numeric_limits::traps
numeric_limits::min_exponent10	numeric_limits::tinyness_before
numeric_limits::max_exponent	numeric_limits::round_style

numeric_limits::is_specialized

Description The member `is_specialized` makes it possible to distinguish between scalar types, which have specializations, and non-scalar types, which do not.

Definition `static const bool is_specialized;`

numeric_limits::min

Description This function returns the minimum finite value that can be `CHAR_MIN`, `SHRT_MIN`, `FLT_MIN`, `DBL_MIN`, etc. For this function, floating types with denormalization, return the minimum positive normalized value, `denorm_min`. The value returned by this function is meaningful for all specializations in which `is_bounded==true`, or `is_bounded==false` and `is_signed==false`.

Prototype `static T min();`

numeric_limits::max

Description This function returns the maximum finite value that can be `CHAR_MAX`, `SHRT_MAX`, `FLT_MAX`, `DBL_MAX`, etc. The value returned by this function is meaningful for all specializations in which `is_bounded==true`.

Prototype `static T max();`

numeric_limits::digits

Description It stores the number of radix digits that can be represented without change. For built-in integer types, it denotes the number of non-sign bits in the representation. For floating point types, it denotes the number of radix digits in the mantissa which can be `FLT_MANT_DIG`, `DBL_MANT_DIG` or `LDBL_MANT_DIG`.

Definition `static const int digits;`

numeric_limits::digits10

Description It stores the number of base 10 digits that can be represented without change. The value can be one among `FLT_DIG`, `DBL_DIG` or

LDBL_DIG. It is meaningful for all specializations in which `is_bounded==true`.

Definition `static const int digits10;`

numeric_limits::is_signed

Description It stores a value `true` if the type is signed. It is meaningful for all specializations.

Definition `static const bool is_signed;`

numeric_limits::is_integer

Description It stores a value that is `true`, if the type is integer. It is meaningful for all specializations.

Definition `static const bool is_integer;`

numeric_limits::is_exact

Description It stores a value `true` if the type uses an exact representation. All integer types are exact, but not vice versa. For example, rational and fixed-exponent representations are exact but not integer. It is meaningful for all specializations.

Definition `static const bool is_exact;`

numeric_limits::radix

Description It stores a value that specifies the base of radix of the exponent representation for floating types. Its value is usually 2 or can be `FLT_RADIX`. For integer types it specifies the base of the representation. It is meaningful for all specializations.

Definition `static const int radix;`

numeric_limits::epsilon

Description It returns the difference between 1 and the least value greater than 1 that is representable. The value can be either `FLT_EPSILON`, `DBL_EPSILON` or `LDBL_EPSILON`. The value returned is meaningful only for floating point types.

Prototype `static T epsilon();`

numeric_limits::round_error

Description It returns a value that denotes the maximum rounding error that is permitted. It is meaningful only for floating point types.

Prototype `static T round_error();`

numeric_limits::min_exponent

Description It stores the minimum negative integer such that `radix` raised to that power is in range. It is meaningful only for floating point types and the values can be `FLT_MIN_EXP`, `DBL_MIN_EXP`, `LDBL_MIN_EXP`.

Definition `static const int min_exponent;`

numeric_limits::min_exponent10

Description It stores the minimum negative integer such that 10 raised to that power is in range. It is meaningful only for floating point types and the values can be `FLT_MIN_10_EXP`, `DBL_MIN_10_EXP`, `LDBL_MIN_10_EXP`.

Definition `static const int min_exponent10;`

numeric_limits::max_exponent

Description It stores the maximum positive integer such that `radix` raised to that power is in range. It is meaningful only for floating point types and

the values can be `FLT_MAX_EXP`, `DBL_MAX_EXP`,
`LDBL_MAX_EXP`.

Definition `static const int max_exponent;`

numeric_limits::max_exponent10

Description It stores the maximum positive integer such that 10 raised to that power is in range. It is meaningful only for floating point types and the values can be `FLT_MAX_10_EXP`, `DBL_MAX_10_EXP`, `LDBL_MAX_10_EXP`.

Definition `static const int max_exponent10;`

numeric_limits::has_infinity

Description It stores a value `true` if the type has a representation for positive infinity. It is meaningful only for floating point types. It will be `true` for all specialization in which `is_iec559==true`.

Definition `static const bool has_infinity;`

numeric_limits::has_quiet_NaN

Description It stores a value `true` if the type has a representation for a quiet(non-signaling) Not a Number. It is meaningful only for floating point types. The value will be `true` for all specializations in which `is_iec559==true`.

Definition `static const bool has_quiet_NaN;`

numeric_limits::has_signaling_NaN

Description It stores a value `true` if the type has a representation for a signaling Not a Number. It is meaningful only for floating point types. The value will be `true` for all specializations in which `is_iec559==true`.

Definition `static const bool has_signaling_NaN;`

numeric_limits::has_denorm

Description It stores `true` if the type allows denormalized values (i.e. variable number of exponent bits). The value is meaningful only for floating point types.

Definition `static const bool has_denorm;`

numeric_limits::infinity

Description It returns the representation of positive infinity, if available. It is meaningful only for specializations for which `has_infinity==true`. Required in specializations for which `is_iec559==true`.

Prototype `static T infinity();`

numeric_limits::quiet_NaN

Description This function returns the representation of a quiet Not a Number if `has_quiet_NaN==true`.

Prototype `static T quiet_NaN();`

numeric_limits::signaling_NaN

Description This function returns the representation of a signaling Not a Number if `has_signaling_NaN==true`.

Prototype `static T signaling_NaN();`

numeric_limits::denorm_min

Description It returns the minimum positive denormalized value. It is meaningful for all floating point types. In this function specialization for

which `has_denorm==false`, returns the minimum positive normalized value.

Prototype `static T denorm_min();`

numeric_limits::is_bounded

Description It stores a value `true` if the set of values representable by the type is finite. All built-in types are bounded, this member would be `false` for arbitrary precision types. It is meaningful for all specializations.

Definition `static const bool is_bounded;`

numeric_limits::is_modulo

Description It stores a value `true` if the type is modulo. A type is modulo if it is possible to add two positive numbers and have a result that wraps around to a third number that is less than either of the two numbers. This has a value `false` for all floating types, `true` for unsigned integers, and `true` for signed integers on most machines and is meaningful for all specializations.

Definition `static const bool is_modulo;`

numeric_limits::traps

Description It stores a value `true` if trapping is implemented for the type. It is meaningful for all specializations.

Definition `static const bool traps;`

numeric_limits::tinyness_before

Description It stores a value `true` if tinyness is detected before rounding. It is meaningful only for floating point types.

Definition `static const bool tinyness_before;`

numeric_limits::round_style

Description It stores the rounding style for the type. The value stored is meaningful for all floating point types. Specializations for integer types shall return `round_toward_zero`.

Definition `static const float_round_style round_style;`

Dynamic Memory Storage Allocation Errors

This section discusses classes related to memory allocation errors. The classes discussed are:

- [“Class `bad_alloc`” on page 68](#)
- [“Class `bad_cast`” on page 69](#)
- [“Class `bad_typeid`” on page 70](#)

Class `bad_alloc`

The class `bad_alloc`, derived from the class `exception`, defines the type of objects thrown as exceptions by the implementation to report a failure to allocate storage.

Table 3.2 The member functions discussed are:

Constructor–<code>bad_cast</code>	Copy Constructor–<code>bad_alloc</code>
Assignment Operator–<code>bad_alloc</code>	bad_alloc::what

Constructor–`bad_alloc`

Description This function constructs an object of class `bad_alloc`.

Prototype `bad_alloc() throw();`

Copy Constructor–`bad_alloc`

Description This function copies an object of class `bad_alloc`.

Prototype `bad_alloc(const bad_alloc&) throw();`

Assignment Operator–bad_alloc

Description This function copies an object of class `bad_alloc`.

Prototype `bad_alloc& operator=(const bad_alloc&) throw();`

bad_alloc::what

Description Our implementation returns a string object instead of `const char*`. This function returns the string associated with the exception if some allocation is done, else it returns `string()`.

Prototype `virtual const char* what() const throw();`

Class bad_cast

The class `bad_cast`, derived from the class `exception`, defines the type of objects thrown as exceptions by the implementation to report the execution of an invalid dynamic-cast expression.

Table 3.3 **The member functions discussed are:**

<u>Constructor–bad_cast</u>	<u>Copy Constructor–bad_cast</u>
<u>Assignment Operator–bad_cast</u>	<u>bad_cast::what</u>

Constructor–bad_cast

Description This function constructs an object of class `bad_cast`.

Prototype `bad_cast() throw();`

Copy Constructor–bad_cast

Description This function copies or constructs an object of class `bad_cast`.

Prototype `bad_cast(const bad_cast&) throw();`

Assignment Operator–bad_cast

Description This function copies an object of class `bad_cast`.

Prototype `bad_cast& operator=(const bad_cast&) throw();`

bad_cast::what

Description Our implementation returns a string object instead of `const char*`. This function returns the string associated with the exception if some allocation is done, else it returns `string()`.

Prototype `virtual const char* what() const throw();`

Class bad_typeid

The class `bad_typeid`, derived from the class `exception`, defines the type of objects thrown as exceptions by the implementation to report a null pointer in a `typeid` expression.

Table 3.4 The member functions discussed are:

[Constructor–bad_typeid](#)

[Copy Constructor–bad_typeid](#)

[Assignment Operator–
bad_typeid](#)

[bad_typeid::what](#)

Constructor–bad_typeid

Description This function constructs an object of class `bad_typeid`.

Prototype `bad_typeid() throw();`

Copy Constructor–bad_typeid

Description These functions copy an object of class `bad_typeid`.

Prototype `bad_typeid(const bad_typeid&) throw();`

Assignment Operator—bad_typeid

Description This function copies an object of class `bad_typeid`.

Prototype `bad_typeid& operator=(const bad_typeid&) throw();`

bad_typeid::what

Description Our implementation returns a `string` object instead of `const char*`. This function returns the string associated with the exception if some allocation is done, else it returns `string()`.

Prototype `virtual const char* what() const throw();`



Diagnostics Library

This chapter is a reference guide to the ANSI/ISO exception classes, which are used for reporting several kinds of exceptional conditions, documenting program assertions, and a global variable for error number codes.

Overview of Diagnostics Library

The standard C++ library provides classes to be used to report errors in a C++ program. In the error model reflected in these classes, errors are divided into two broad categories: logic errors and runtime errors. The distinguishing characteristic of logic errors is that they are due to errors in the internal logic of the program, and are preventable. In contrast, runtime errors are due to events beyond the scope of the program. They cannot be easily predicted in advance. The header `<stdexcept.h>` defines several types of predefined exceptions for reporting errors in C++ program. These exceptions are related via inheritance.

The only group of classes in the diagnostics library are:

- [“Exception Classes” on page 73](#)

Exception Classes

There are several exception-related classes in the diagnostics library. The base class is `Class exception`. Other exception classes derive from `exception`.

The classes are:

- [“Class exception” on page 74](#)
- [“Class domain_error” on page 76](#)
- [“Class invalid_argument” on page 76](#)

- [“Class logic_error” on page 75](#)
- [“Class out_of_range” on page 77](#)
- [“Class runtime_error” on page 77](#)
- [“Class length_error” on page 76](#)
- [“Class overflow_error” on page 78](#)
- [“Class range_error” on page 78](#)

Class exception

Description The class exception defines the base class for the types of objects thrown as exceptions by C++ standard library components, and certain expressions, to report errors detected during program execution.

Table 4.1 The member functions discussed are:

[Constructor-exception](#)

[Copy Constructor-exception](#)

[Assignment Operator-exception](#)

[Destructor-exception](#)

[exception::what](#)

Constructor-exception

Description This function constructs an object of class exception. It does not throw any exceptions.

Prototype `exception() throw();`

Copy Constructor-exception

Description This function copies an exception object.

Prototype `exception& exception(const exception&) throw();`

Assignment Operator–exception

Description This function copies an exception object.

Prototype `exception& operator=(const exception&) throw();`

Destructor–exception

Description This function destroys an object of class `exception`.

Prototype `virtual ~exception() throw();`

exception::what

Description Our implementation returns a string object instead of `const char*`. This function returns the string associated with the exception if some allocation is done, else it returns `string()`.

Prototype `virtual const char* what() const throw();`

Class `logic_error`

Description The class `logic_error`, derived from [“Class `exception`” on page 74](#), defines the type of objects thrown as exceptions to report errors presumably detectable before the program executes, such as violations of logical preconditions or class invariants.

Table 4.2 The member functions discussed are:

[Constructor–`logic_error`](#)

Constructor–`logic_error`

Description This function constructs an object of class `logic_error`.

Prototype `logic_error(const string& what_arg);`

Class `domain_error`

Description The class `domain_error`, derived from [“Class `logic_error`” on page 75](#), defines the type of objects thrown as exceptions to report domain errors.

Table 4.3 The member functions discussed are:

[Constructor–`domain_error`](#)

Constructor–`domain_error`

Description This function constructs an object of class `domain_error`.

Prototype `domain_error(const string& what_arg);`

Class `invalid_argument`

Description The class `invalid_argument` defines the type of objects thrown as exceptions to report an invalid argument.

Table 4.4 The member functions discussed are:

[Constructor–`invalid_argument`](#)

Constructor–`invalid_argument`

Description This function constructs an object of class `invalid_argument`.

Prototype `invalid_argument(const string& what_arg);`

Class `length_error`

Description The class `length_error`, derived from [“Class `logic_error`” on page 75](#), defines the type of objects thrown as exceptions to report an attempt to produce an object whose length equals or exceeds its maximum allowable size.

Table 4.5 **The member functions discussed are:**

[Constructor-length_error](#)

Constructor-length_error

Description This function constructs an object of class `length_error`.

Prototype `length_error(const string& what_arg);`

Class out_of_range

Description The class `out_of_range`, derived from [“Class logic_error” on page 75](#), defines the type of objects thrown as exceptions to report an argument value not in its expected range.

Table 4.6 **The member functions discussed are:**

[Constructor-out_of_range](#)

Constructor-out_of_range

Description This function constructs an object of class `out_of_range`.

Prototype `out_of_range(const string& what_arg);`

Class runtime_error

Description The class `runtime_error`, derived from [“Class exception” on page 74](#), defines the type of objects thrown as exceptions to report errors presumably detectable only when the program executes.

Table 4.7 **The member functions discussed are:**

[Constructor-runtime_error](#)

Constructor–runtime_error

Description This function constructs an object of class `runtime_error`.

Prototype `runtime_error(const string& what_arg);`

Class range_error

Description The class `range_error`, derived from [“Class runtime_error” on page 77](#), defines the type of objects thrown as exceptions to report range errors.

Table 4.8 The member functions discussed are:

[Constructor–range_error](#)

Constructor–range_error

Description This function constructs an object of class `range_error`.

Prototype `range_error(const string& what_arg);`

Class overflow_error

Description The class `overflow_error`, derived from [“Class runtime_error” on page 77](#), defines the type of objects thrown as exceptions to report an arithmetic overflow error.

Table 4.9 The member functions discussed are:

[Constructor–overflow_error](#)

Constructor–overflow_error

Description This function constructs an object of class `overflow_error`.

Prototype `overflow_error(const string& what_arg);`

Diagnostics Library

Exception Classes



General Utilities Libraries

This chapter discusses the `Allocator` class.

Overview of General Utilities Libraries

Every STL container class uses an `Allocator` class to encapsulate information about the memory model being used by the program. The topics in this chapter are:

- [“Allocator Classes” on page 81](#)
- [“The Default Allocator Interface” on page 83](#)
- [“Arithmetic Operations” on page 91](#)
- [“Comparison Operations” on page 93](#)
- [“Logical Operations” on page 95](#)
- [“Negator Adaptors” on page 96](#)
- [“Binder Adaptors” on page 96](#)
- [“Adaptors for Pointers to Functions” on page 97](#)
- [“Specialized Algorithms” on page 99](#)
- [“Template class `auto_ptr`” on page 100](#)

Allocator Classes

Different memory models have different requirements for pointers, references, integer sizes, etc. The `Allocator` class encapsulates information about pointers, constant pointers, references, constant references, sizes of objects, difference types between pointers, allocation and deallocation functions, and some other functions. The

exact set of types and functions defined within the allocator are explained in the Default Allocator Interface, later in this chapter.

Since memory model information can be encapsulated in an allocator, STL containers can work with different memory models by simply providing different allocators. All operations on allocators are expected to be amortized constant time.

Additional topics are:

- [“Passing Allocators to STL Containers” on page 82](#)
- [“Extracting Information from an Allocator Object” on page 82](#)

Passing Allocators to STL Containers

Once an allocator class for a particular memory model has been written, it must be passed on to the STL container for that container to work properly in the concerned memory model. This is done by passing the allocator to the STL container as a template parameter.

For example, the vector container has the following interface:

```
template<class T, class Allocator = allocator>
class vector;
```

Here the Allocator parameter defaults to allocator. All the STL containers are not yet parameterized on Allocator. In all places where Allocator parameter is specified in the draft, default allocator is used.

Extracting Information from an Allocator Object

Allocator is obtained by a container by a macro of the same name. Once the allocator is known, the container must somehow extract the memory model information from the Allocator class. This information is extracted by simply accessing the typedefs and member functions of the Allocator class.

For example, the public interface of the vector class mentioned above contains the following typedefs to extract information about references and pointers from the Allocator class:

```
typedef Allocator<T>::reference reference;  
typedef Allocator<T>::const_reference  
    const_reference;  
typedef Allocator<T>::pointer iterator;  
typedef Allocator<T>::const_pointer  
    const_iterator;
```

NOTE: If for different memory models, the types `Allocator<T>::pointer`, `Allocator<T>::size_type`, etc., will in general be different. However, the point is that these differences do not affect the vector container (or any other STL container), since the changes are completely encapsulated in the `Allocator` class.

The information passed on from the `Allocator` class to the STL container includes the types of pointers, constant pointers, references, constant references, etc., together with some member functions. Complete details of the types and functions encapsulated by the default allocator are provided in the next section.

The Default Allocator Interface

[“Class allocator declaration” on page 83](#) contains the Class allocator declaration.

Listing 5.1 Class allocator declaration

```
class allocator {  
    public:  
        typedef size_t size_type;  
        typedef ptrdiff_t difference_type;  
  
        template<class T>  
        class types {
```

General Utilities Libraries

The Default Allocator Interface

```
typedef T* pointer;
typedef const T* const_pointer;
typedef T& reference;
typedef const T& const_reference;
typedef T value_type;
};

allocator();
~allocator();

template<class T> typename
    types<T>::pointer
    address(types<T>::reference x) const;

template<class T> typename
    types<T>::const_pointer
    address(types<T>::const_reference x) const;

template<class T, class U> typename
    types<T>::pointer allocate(size_type,
    types<U>::const_pointer hint);

template<class T>
    void deallocate(types<T>::pointer p);

size_type max_size() const;
};

class allocator::types<void> { // specialization
public:
    typedef void* pointer;
    typedef const void* const_pointer;
    typedef void value_type;
};

void* operator new(size_t N, allocator& a);
```

Description In the allocator interface it can be seen that the type information is encapsulated in the nested template class types. Since nested class templates are not yet supported by any compilers, our implementa-

tion provides the following workaround by templatizing the class `allocator`.

The topics in this section discuss:

- [“Typedef Declarations” on page 85](#)
- [“Allocator Member Functions” on page 86](#)
- [“Custom Allocators” on page 89](#)
- [“Allocator Requirements” on page 89](#)

Typedef Declarations

Table 5.1 The following typedef’s are defined in the class `allocator<T>`

<code>allocator::pointer</code>	<code>allocator::const_pointer</code>
<code>allocator::reference</code>	<code>allocator::const_reference</code>
<code>allocator::value_type</code>	<code>allocator::const_address</code>

`allocator::pointer`

Description The type of a pointer in the memory model.

Definition `typedef T* pointer;`

`allocator::const_pointer`

Description The type of a constant pointer in the memory model.

Definition `typedef const T* const_pointer;`

`allocator::reference`

Description The type of a reference in the memory model.

Definition `typedef T& reference;`

allocator::const_reference

Description The type of a constant reference in the memory model.

Definition `typedef const T& const_reference`

allocator::value_type

Description `value_type` refers to the type of the objects in the container. By default, containers contain objects of the type with which they are instantiated. For example, `vector<int*>` is a declaration of a vector of pointers to integers.

Definition `typedef T value_type;`

Allocator Member Functions

Table 5.2 The Class `allocator` has the following member functions

[Constructor-allocator](#)

[Destructor-allocator](#)

[allocator::size_type](#)

[allocator::difference_type](#)

[allocator::address](#)

[allocator::const_address](#)

[allocator::allocate](#)

[allocator::deallocate](#)

[allocator::max_size](#)

Constructor-allocator

Description Constructs an allocator object.

Prototype `allocator();`

Destructor-allocator

Description Destroys an allocator object.

Prototype `~allocator();`

The class `allocator` has the following other members:

`allocator::size_type`

Description `size_type` is the type that can represent the size of the largest object in the memory model.

Definition `typedef size_t size_type;`

`allocator::difference_type`

Description This is the type that can represent the difference between any two pointers in the memory model.

Definition `typedef ptrdiff_t difference_type;`

`allocator::address`

Description Returns a pointer to the referenced object `x`.

Prototype

```
template
    < class T >
allocator<T>::pointer address
    (allocator<T>::reference x);
```

`allocator::const_address`

Description Returns a constant pointer to the referenced object `x`.

Prototype

```
template
    < class T >
allocator<T>::const_pointer const_address
    (allocator<T>::const_reference x);
```

allocator::allocate

Description This member function allocates memory for `n` objects of type `size_type` but the objects are not constructed. It uses the global operator `new`.

NOTE: If that different memory models require different allocate functions (which is why the function has been encapsulated in the memory allocator class). `allocate` may raise an appropriate exception.

Prototype

```
template
    < class T >
allocator<T>::pointer allocate
    (size_type n);
```

allocator::deallocate

Description Deallocates all of the storage pointed to by the pointer `p` using the global operator `delete`. All objects in the area pointed to by `p` should be destroyed prior to the call of `deallocate`. The function is templated to allow it to be specialized for particular types in custom allocators.

Prototype

```
template
    < class T >
void deallocate
    (allocator<T>::pointer p);
```

allocator::max_size

Description Returns the largest positive value of `difference_type`. This is the same as the largest number of elements that the container can hold in the given memory model.

Prototype

```
size_type max_size();
```


Custom Allocators

The data types representing pointers (`size_type`) as well as the difference between two pointers (`difference_type`) differ across memory models.

Table 5.3 The data types representing pointers

[`allocator::vec_default`](#) [`allocator::vec_large`](#)
[`allocator::vec_huge`](#)

`allocator::vec_default`

Description A vector of 100 integers using the default allocator.

Prototype `vector<int> vec_default(100);`

`allocator::vec_large`

Description A vector of 1000 integers using the far allocator.

Prototype `vector<int, far_allocator> vec_large(1000);`

Remarks In case of the far allocator, the addressable range is 64 K. The `size_type` is a 32 bit value and `difference_type` is a 16-bit value.

`allocator::vec_huge`

Description A vector of 100,000 integers using the huge allocator.

Prototype `vector<int, huge_allocator> vec_huge(100000);`

Allocator Requirements

This section discusses

- [“Function Objects” on page 90](#)

- [“Function Adaptors” on page 90](#)

Function Objects

Description A function object encapsulates a function in an object for use by other components. This is done by overloading the function call operator, `operator()`, of the corresponding class.

Passing a function object to an algorithm is similar to passing a pointer to a function, with one important difference. Function objects are classes that have `operator()` overloaded, which makes it possible to

- pass function objects to algorithms at compile time, and
- increase efficiency, by inlining the corresponding call.

These costs make a difference when the functions involved are very simple ones, such as integer additions or comparisons.

For example, if we want to have a by-element addition of two integer vectors `a` and `b`, with the result being placed in `a`, we can do:

```
transform(a.begin(), a.end(), b.begin(),  
          a.begin(), plus<int>());
```

If we want to negate every element of `a`, we do:

```
transform(a.begin(), a.end(),  
          a.begin(), negate<int>());
```

In both of the above examples, the addition and negation will be inlined.

Function Adaptors

Description Function adaptors are STL classes that allow users to construct a wider variety of function objects. Using function adaptors is often easier than direct construction of a new function object type with a struct or class definition.

For example, a binder is a kind of function adapter that converts binary function objects into unary function objects by binding an argument to some particular value. The code fragment:

```
int x[1000];
int* where = find_if(x, x+1000,
    bind2nd(greater<int>(), 200));
```

finds the first integer in array `x` that is greater than 200. The base function object `greater<int>()` takes two arguments `x` and `y` and returns the greater value. By applying the binder `bind2nd` to this function object and the number 200, we produce a function object that defines a unary function that takes a single argument `x` and returns true if `x > 200`. This function object is then used as a parameter to `find_if`, to find the first element in the array that is greater than 200.

Besides binders, the library defines two other kinds of function adaptors:

- Negators are function objects that reverse the sense of predicate function objects.
- Adaptors for pointers to functions allow pointers to (unary and binary) functions to work with function adaptors that the library provides.

Arithmetic Operations

Files `#include <function.h>`

Description STL provides basic function object classes for all of the arithmetic operators in the language. The functionality of the operators is described below. The function object classes are:

Table 5.4 The functionality of the operators

<u>plus</u>	<u>times</u>
<u>divides</u>	<u>minus</u>
<u>modulus</u>	<u>negate</u>

plus

Description The plus function object accepts two operands x and y of type T , and returns the result of the computation of $x + y$.

Prototype `template< class T >
struct plus<T>`

minus

Description The minus function object accepts two operands x and y of type T , and returns the result of the computation of $x - y$.

Prototype `template< class T >
struct minus<T>`

times

Description The times function object accepts two operands x and y of type T , and returns the result of the computation of $x * y$.

Prototype `template< class T >
struct times<T>`

divides

Description The divides function object accepts two operands x and y of type T , and returns the result of the computation of x / y .

Prototype `template< class T >
struct divides<T>`

modulus

Description The modulus function object accepts two operands, x and y , of type T , and returns their result of the computation $x \% y$.

Prototype `template< class T >`

```
struct modulus<T>
```

negate

Description negate is a unary function that accepts a single operand x of type T, and returns the result of the computation of -x.

Prototype

```
template< class T >  
struct negate<T>
```

Comparison Operations

Files `#include <function.h>`

Description STL provides basic function object classes for all of the comparison operators in the language. The basic functionality of the comparison objects is described below.

Table 5.5 The operation classes are:

<u>equal_to</u>	<u>not_equal_to</u>
<u>greater</u>	<u>less</u>
<u>greater_equal</u>	<u>less_equal</u>

equal_to

Description An object of this type accepts two parameters, x and y, of type T, and returns true if x == y.

Prototype

```
template< class T >  
struct equal_to<T>
```

not_equal_to

Description An object of this type accepts two parameters, x and y, of type T, and returns true if x != y.

General Utilities Libraries

Comparison Operations

Prototype `template< class T >`
 `struct not_equal_to<T>`

greater

Description An object of this type accepts two parameters, x and y, of type T, and returns true if x > y.

Prototype `template< class T >`
 `struct greater<T>`

less

Description An object of this type accepts two parameters, x and y, of type T, and returns true if x < y.

Prototype `template< class T >`
 `struct less<T>`

greater_equal

Description An object of this type accepts two parameters, x and y, of type T, and returns true if x >= y.

Prototype `template< class T >`
 `struct greater_equal<T>`

less_equal

Description An object of this type accepts two parameters, x and y, of type T, and returns true if x <= y.

Prototype `template< class T >`
 `struct less_equal<T>`

Logical Operations

Files `#include <function.h>`

Description STL provides basic function object classes for the following logical operators in the language: and, or, not. The basic functionality of the logical operators is described below.

Table 5.6 **The logical operation classes are:**

[logical_and](#) [logical_or](#)
[logical_not](#)

logical_and

Description An object of this type accepts two parameters, x and y, of type T, and returns the boolean result of the logical and operation: `x && y`.

Prototype `template< class T >`
 `struct logical_and<T>`

logical_or

Description An object of this type accepts two parameters, x and y, of type T, and returns the boolean result of the logical or operation: `x || y`.

Prototype `template< class T >`
 `struct logical_or<T>`

logical_not

Description An object of this type accepts a single parameter, x, of type T, and returns the boolean result of the logical not operation: `!x`.

Prototype `template< class T >`
 `struct logical_not<T>`

Negator Adaptors

Files `#include <function.h>`

Description Negators are function adaptors that take a predicate and return its complement. STL provides the negators `not1` and `not2` that take a unary and binary predicate respectively and return their complements.

Table 5.7 **The negator function adaptors are:**

[not1](#)

[not2](#)

not1

Description This function accepts a unary predicate `x` as input and returns its complement `!x`.

Prototype

```
template
    < class Predicate >
    unary_negate<Predicate> not1
        (const predicate& x);
```

not2

Description This function accepts a binary predicate `x` as input and returns its complement, `!x`.

Prototype

```
template
    < class Predicate >
    binary_negate<Predicate> not2
        (const predicate& x);
```

Binder Adaptors

Files `#include <function.h>`

Description Binders are function adaptors that convert binary function objects into unary function objects by binding an argument to some particular value. STL provides two binders `bind1st` and `bind2nd`, which are described below.

Table 5.8 The binder function adaptors are:

[`bind1st`](#)

[`bind2nd`](#)

`bind1st`

Description This adapter accepts a function object `op` of two arguments and a value `x`, of type `T`. It returns a function object of one argument, constructed out of `op` with the *first* argument bound to `x`.

Prototype

```
template
< class Operation,
  class T >
binder1st<Operation> bind1st
    (const Operation& op,
     const T& x);
```

`bind2nd`

Description This adapter accepts a function object `op` of two arguments and a value `x`, of type `T`. It returns a function object of one argument, constructed out of `op` with the second argument bound to `x`.

Prototype

```
template
< class Operation,
  class T >
binder2nd<Operation> bind2nd
    (const Operation& op,
     const T& x);
```

Adaptors for Pointers to Functions

Files `#include <function>`

General Utilities Libraries

Adaptors for Pointers to Functions

Description Adaptors for pointers to functions are provided to allow pointers to unary and binary functions to work with the function adaptors provided in the library. They also can help avoid the “code bloat” problem arising from multiple template instances in the same program.

Table 5.9 The function pointer adaptors are:

[ptr_fun-unary](#)

[ptr_fun-binary](#)

Description STL provides two adaptors for pointers to functions: one for unary and the other for binary functions. Both the functions have the same name (which is overloaded).

ptr_fun-unary

Description This function adapter accepts a pointer to a unary function that takes an argument of type `Arg` and returns a result of type `Result`. A function object of type `pointer_to_unary_function<Arg, Result>` is constructed out of this argument, and returned.

Prototype

```
template
    < class Arg,
      class Result >
ptr_fun(Result (*x) (Arg));
```

ptr_fun-binary

Description This function adapter accepts a pointer to a binary function that accepts arguments of type `Arg1` and `Arg2` and returns a result of type `Result`. A function object of type `pointer_to_binary_function<Arg, Result>` is constructed out of this argument, and returned.

Prototype

```
template
    < class Arg,
      class Result >
ptr_fun(Result (*x) (Arg1, Arg2));
```

Specialized Algorithms

All the iterators that are used as formal template parameters in these algorithms are required to have their `operator*` return an object for which `operator&` is defined and returns a pointer to T. See [“Overview of Iterators” on page 283](#) where algorithms are discussed in detail.

Table 5.10 **The uninitialized copy and fill functions are:**

[uninitialized_copy](#) [uninitialized_fill](#)
[uninitialized_fill_n](#)

uninitialized_copy

Description This function behaves as follows: while (`first != last`) construct (`&*result++`, `*first++`); This function returns result.

Prototype

```
template
    < class InputIterator,
      class ForwardIterator >
ForwardIterator uninitialized_copy
    (InputIterator first,
     InputIterator last,
     ForwardIterator result);
```

uninitialized_fill

Description This function behaves as follows:
 while (`first != last`)
 construct (`&*first++`, `x`);

Prototype

```
template
< class ForwardIterator,
  class T >
void uninitialized_fill
    (ForwardIterator first,
```

General Utilities Libraries

Template class *auto_ptr*

```
ForwardIterator last,  
const T& x);
```

uninitialized_fill_n

Description This function behaves as follows:
while (n--)
 construct (&*first++, x);

Prototype template
 < class ForwardIterator,
 class Size,
 class T>
void uninitialized_fill_n
 (ForwardIterator first, Size n, const T& x);

Template class *auto_ptr*

Description Template class *auto_ptr* holds onto a pointer obtained via *new* and deletes that object when it itself is destroyed when it goes out of scope. The *auto_ptr* provides semantics of strict ownership. An object may be safely pointed to by only one *auto_ptr*, so copying an *auto_ptr* copies the pointer and transfers ownership to the destination. The declaration of *auto_ptr* is given below.

```
namespace std {  
    template<class X> class auto_ptr {  
    public:  
        typedef X element_type;  
  
        explicit auto_ptr(X* p =0) throw();  
        auto_ptr(const auto_ptr&) throw();  
        template<class Y> auto_ptr(const auto_ptr<Y>&) throw();  
  
        auto_ptr& operator=(const auto_ptr&) throw();  
        template<class Y> auto_ptr& operator=  
            (const auto_ptr<Y>&) throw();  
  
        ~auto_ptr();  
    }  
};
```

```

X& operator*() const throw();
X* operator->() const throw();
X* get() const throw();
X* release() const throw();
};
}

```

Constructor—*auto_ptr*

Description This function requires *p* to point to an object of class *X* or a class derived from *X* for which *delete p* is defined and accessible. or else *p* is a null pointer.

Prototype `explicit auto_ptr(X* p =0) throw();`

Postconditions: *this holds the pointer to *p*. *this owns *get() if and only if *p* is not a null pointer.

Prototype `auto_ptr(const auto_ptr&) throw();`

Postconditions: *this holds the pointer returned from *a.release()*. *this owns *get() if and only if *a* owns **a*.

Remarks Calls *a.release()*.

Prototype `template<class Y> auto_ptr
(const auto_ptr<Y>& a) throw();`

Postconditions: *this holds the pointer returned from *a.release()*. *this owns *get() if and only if *a* owns **a*.

Remarks

- *Y** can be implicitly converted to *X**.
- Calls *a.release()*.

auto_ptr Destructor

Description Destroys the auto pointer.

General Utilities Libraries

Template class *auto_ptr*

Prototype `~auto_ptr();`

Remarks If **this* owns **get()* then delete *get()*.

auto_ptr::get

Description This function returns the pointer *p* specified as the argument to the constructor `auto_ptr (X* p)` or as the argument to the most recent call to `reset (X* p)`.

Prototype `X* get () const;`

auto_ptr::release

Description This function releases the pointer and after a call to this function `get ()` will return 0.

Prototype `X* release ();`

Assignment Operator=auto_ptr

Description Copies the argument *a* to **this*.

Prototype `auto_ptr& operator=
 (const auto_ptr& a) throw();
template<class Y> auto_ptr& operator=
 (const auto_ptr<Y>& a) throw();`

Dereferencing Operator*auto_ptr

Description This returns **get ()* provided `get ()` does not return 0.

Prototype `X& operator* () const;`

Association Operator->auto_ptr

Description Returns the pointer associated.

Prototype `x* operator-> () const;`

General Utilities Libraries

Template class auto_ptr



21 Strings Library

This chapter is a reference guide to the ANSI/ISO String class that describes components for manipulating sequences of characters, where characters may be of type `char`, `wchar_t`, or of a type defined in a C++ program.

Overview of Strings Classes

The strings library is based on the ANSI/ISO string class description and includes

- [“21.1 Character traits” on page 105](#) defines types and facilities for character manipulations
- [“21.2 String Classes” on page 109](#) lists string and character structures and classes
- [“21.3 Class basic_string” on page 115](#) defines facilities for character sequence manipulations.
- [“23.4 Null Terminated Sequence Utilities” on page 143](#) lists facilities for Null terminated character sequence strings.

21.1 Character traits

This section defines a class template `char_traits<charT>` and two specializations for `char` and `wchar_t` types. These types are required by string and stream classes and are passed to these classes as formal parameters `charT` and `traits`.

The topics in this section are:

- [“21.1.1 Character Trait Definitions” on page 106](#)
- [“21.1.2 Character Trait Requirements” on page 106](#)
- [“21.1.3 Character Trait Type Definitions” on page 107](#)
- [“21.1.4 struct char_traits<T>” on page 108](#)

21 Strings Library

21.1 Character traits

21.1.1 Character Trait Definitions

character Any object when treated sequentially can represent text. This term is not restricted to just `char` and `wchar_t` types

character container type A class or type used to represent a character. This object must be POD (Plain Old Data).

traits A class that defines types and functions necessary for handling characteristics.

NTCTS A null character termination string is a character sequence that proceeds the null character value `charT(0)`.

21.1.2 Character Trait Requirements

These types are required by string and stream classes and are passed to these classes as formal parameters `charT` and `traits`.

assign Used for character type assignment.

```
static void assign  
    (char_type, const char_type);
```

eq Used for `bool` equality checking.

```
static bool eq  
    (const char_type&, const char_tye&);
```

lt Used for `bool` less than checking.

```
static bool lt(const char_type&, const char_type&);
```

compare Used for NTCTS comparison.

```
static int compare  
    (const char_type*, const char_type*, size_t n);
```

length Used when determining the length of a NTCTS.

```
static size_t length  
    (const char_type*);
```

find Used to find a character type in an array

```
static const char_type* find  
    (const char_type*, int n, const char_type&);
```

move Used to move one NTCTS to another even if the receiver contains the sting already.

```
static char_type* move  
    (char_type*, const char_type*, size_t);
```

copy Used for copying a NTCTS that does not contain the NTCTS already.

```
static char_type* copy  
    (char_type*, const char_type*, size_t);
```

not_eof Used for bool inequality checking.

```
static int_type not_eof  
    (const int_type&);
```

to_char_type Used to convert to a char type from an int_type

```
static char_type to_char_type  
    (const int_type&);
```

to_int_type Used to convert from a char type to an int_type.

```
static int_type to_int_type  
    (const char_type&);
```

eq_int_type Used to test for deletion.

```
static bool eq_int_type  
    (const int_type&, const int_type& );
```

get_state Used to store the state of the file buffer.

```
static state_type get_state  
    (pos_type pos);
```

eof Used to test for the end of a file.

```
static int_type eof();
```

21.1.3 Character Trait Type Definitions

There are several types defined in the char_traits structure for both wide and conventional char types.

Table 6.1 The functions are:

Type	Defined	Use
char	char_type	char values
int	int_type	integral values of char types including eof
streamoff	off_type	stream offset values

21 Strings Library

21.1 Character traits

streampos	pos_type	stream position values
mbstate_t	state_type	file state values

21.1.4 struct char_traits<T>

Description The template structure is overloaded for both the `wchar_t` type `struct char_traits<wchar_t>`. This specialization is used for string and stream usage.

NOTE: The `assign`, `eq` and `lt` are the same as the `=`, `==` and `<` operators.

Prototype

```
namespace std {
template<>
// struct char_traits<wchar_t> or
struct char_traits<char> {
    typedef char char_type;
    typedef int int_type;
    typedef streamoff off_type;
    typedef streampos pos_type;
    typedef mbstate_t state_type;

    static void assign(char_type, const char_type);
    static bool eg(const char_type&, const
char_tye&);
    static bool lt(const char_type&, const
char_type&);

    static int compar(const char_type*, const
char_type*, size_t n);
    static size_t length(const char_type*);
    static const char_type* find(const char_type*,
int n, const char_type&);

    static char_type* move(char_type*, const
char_type*, size_t);
```

```

static char_type* copy(char_type*, const
char_type*, size_t);
static char_type* assign(char_type*, size_t,
char_type);

static int_type not_eof(const int_type&);
static char_type to_char_type(const int_type&);
static int_type to_int_type(const char_type&);
static bool eq_int_type(const int_type&, const
int_type& );
static state_type get_state(pos_type pos);
static int_type eof();
};
}

```

21.2 String Classes

The header `<string>` define string and trait classes used to manipulate character and wide character like template arguments.

```

namespace std {

template<class charT> struct char_traits;
template <> struct char_traits<char>;
template <> struct char_traits<wchar_t>;

template
<class charT,
class traits =char_traits<charT>,
class Allocator = allocator<charT> >
class basic_string;

template
<class charT, class traits, class Allocator>
basic_string <charT,traits, Allocator>
operator+
    (const basic_string
      <charT,traits,Allocator>& lhs,
      const basic_string

```

21 Strings Library

21.2 String Classes

```
<charT, traits, Allocator>& rhs);
```

```
template
<class charT, class traits, class Allocator>
basic_string<charT,traits,Allocator>
operator+
    const charT* lhs,
    const basic_string
        <charT, traits, Allocator>& rhs);
```

```
template
<class charT, class traits, class Allocator>
basic_string<charT,traits,Allocator>
operator+
    (charT lhs, const basic_string
        <charT,traits,Allocator>& rhs);
```

```
template
<class charT, class traits, class Allocator>
basic_string<charT,traits,Allocator>
operator+
    (const basic_string
        <charT,traits,Allocator>& lhs,
    const charT* rhs);
```

```
template
<class charT, class traits, class Allocator>
basic_string<charT,traits,Allocator>
operator+
    (const basic_string
        <charT,traits,Allocator>& lhs, charT rhs);
```

```
template
<class charT, class traits, class Allocator>
bool operator==
    (const basic_string
        <charT,traits,Allocator>& lhs,
    const basic_string
        <charT,traits,Allocator>& rhs);
```

```
template
<class charT, class traits, class Allocator>
bool operator==
    (const charT* lhs,
     const basic_string
         <charT,traits,Allocator>& rhs);
```

```
template
<class charT, class traits, class Allocator>
bool operator==
    (const basic_string<charT,
     traits,Allocator>& lhs,
     const charT* rhs);
```

```
template
<class charT, class traits, class Allocator>
bool operator!=
    (const basic_string
         <charT,traits,Allocator>& lhs,
     const basic_string
         <charT,traits, Allocator>& rhs);
```

```
template
<class charT, class traits, class Allocator>
bool operator!=
    (const charT* lhs,
     const basic_string
         <charT,traits,Allocator>& rhs);
```

```
template
<class charT, class traits, class Allocator>
bool operator!=
    (const basic_string
         <charT,traits,Allocator>& lhs,
     const charT* rhs);
```

```
template
<class charT, class traits, class Allocator>
bool operator<
    (const basic_string
```

21 Strings Library

21.2 String Classes

```
        <charT,traits,Allocator>& lhs,  
        const basic_string  
        <charT,traits,Allocator>& rhs);
```

```
template  
<class charT, class traits, class Allocator>  
bool operator<  
    (const basic_string  
        <charT,traits,Allocator>& lhs,  
    const charT* rhs);
```

```
template  
<class charT, class traits, class Allocator>  
bool operator<  
    (const charT* lhs,  
    const basic_string  
        <charT,traits,Allocator>& rhs);
```

```
template  
<class charT, class traits, class Allocator>  
bool operator>  
    (const basic_string  
        <charT,traits,Allocator>& lhs,  
    const basic_string  
        <charT,traits,Allocator>& rhs);
```

```
template  
<class charT, class traits, class Allocator>  
bool operator>  
    (const basic_string  
        <charT,traits,Allocator>& lhs,  
    const charT* rhs);
```

```
template  
<class charT, class traits, class Allocator>  
bool operator>  
    (const charT* lhs,  
    const basic_string  
        <charT,traits,Allocator>& rhs);
```



```
template
<class charT, class traits, class Allocator>
bool operator<=
    (const basic_string
     <charT,traits,Allocator>& lhs,
     const basic_string
     <charT,traits,Allocator>& rhs);
```

```
template
<class charT, class traits, class Allocator>
bool operator<=
    (const basic_string
     <charT,traits,Allocator>& lhs,
     const charT* rhs);
```

```
template
<class charT, class traits, class Allocator>
bool operator<=
    (const charT* lhs,
     const basic_string
     <charT,traits,Allocator>& rhs);
```

```
template
<class charT, class traits, class Allocator>
bool operator>=
    (const basic_string
     <charT,traits,Allocator>& lhs,
     const basic_string
     <charT,traits,Allocator>& rhs);
```

```
template
<class charT, class traits, class Allocator>
bool operator>=
    (const basic_string
     <charT,traits,Allocator>& lhs,
     const charT* rhs);
```

```
template
<class charT, class traits, class Allocator>
bool operator>=
```

21 Strings Library

21.2 String Classes

```
(const charT* lhs,  
const basic_string  
    <charT,traits,Allocator>& rhs);
```

```
template  
<class charT, class traits, class Allocator>  
void swap  
    (basic_string<charT,traits,Allocator>& lhs,  
     basic_string<charT,traits,Allocator>& rhs);
```

```
template  
<class charT, class traits, class Allocator>  
basic_istream<charT,traits>&  
operator>>  
    (basic_istream<charT,traits>& is,  
     basic_string<charT,traits,Allocator>& str);
```

```
template  
<class charT, class traits, class Allocator>  
basic_ostream<charT, traits>&  
operator<<  
    (basic_ostream<charT, traits>& os,  
     const basic_string  
         <charT,traits,Allocator>& str);
```

```
template  
<class charT, class traits, class Allocator>  
basic_istream<charT,traits>&  
getline  
    (basic_istream<charT,traits>& is,  
     basic_string<charT,traits,Allocator>& str,  
     charT delim);
```

```
template  
<class charT, class traits, class Allocator>  
basic_istream<charT,traits>&  
getline  
    (basic_istream<charT,traits>& is,  
     basic_string<charT,traits,Allocator>& str);
```

```
typedef basic_string<char> string;  
typedef basic_string<wchar_t> wstring;  
};  
}
```

21.3 Class `basic_string`

The class `basic_string` is used to store and manipulate a sequence of character like types of varying length known as strings.

Remarks Memory for a string is allocated and deallocated as necessary by member functions.

The first element of the sequence is at position zero.

The iterators used by `basic_string` are random iterators and as such qualifies as a reversible container

The topics in this section include:

- [“21.3.1 Constructors and Assignments” on page 121](#)
- [“21.3.2 Iterator Support” on page 124](#)
- [“21.3.3 Capacity” on page 125](#)
- [“21.3.4 Element Access” on page 126](#)
- [“21.3.5 Modifiers” on page 127](#)
- [“21.3.6 String Operations” on page 131](#)
- [“23.3.7 Non-Member Functions and Operators” on page 136](#)

NOTE: In general, the string size can be constrained by memory restrictions.

Prototype

```
namespace std {  
    template  
        <class charT, class traits = char_traits<charT>,  
          class Allocator = allocator<charT> >  
        class basic_string {
```

21 Strings Library

21.3 Class *basic_string*

```
    public:
typedef traits traits_type;
typedef typename traits::char_type value_type;
typedef Allocator allocator_type;
typedef typename Allocator::size_type size_type;
typedef typename Allocator::difference_type
    difference_type;
typedef typename Allocator::reference reference;
typedef typename Allocator::const_reference
    const_reference;
typedef typename Allocator::pointer pointer;
typedef typename Allocator::const_pointer
    const_pointer;
// typedef implementation defined iterator;
// typedef implementation defined const_iterator;
typedef std::reverse_iterator<iterator>
    reverse_iterator;
typedef std::reverse_iterator<const_iterator>
    const_reverse_iterator;

static const size_type npos = -1;

explicit basic_string
    (const Allocator& a = Allocator());
basic_string
    (const basic_string& str,
     size_type pos = 0,
     size_type n = npos,
     const Allocator& a = Allocator());
basic_string
    (const charT* s,
     size_type n,
     const Allocator& a = Allocator());
basic_string
    (const charT* s,
     const Allocator& a = Allocator());
basic_string
    (size_type n,
     charT c,
```

```
    const Allocator& a = Allocator();
template<class InputIterator>
basic_string
    (InputIterator begin,
     InputIterator end,
     const Allocator& a = Allocator());

~basic_string();

basic_string& operator= (const basic_string& str);
basic_string& operator=(const charT* s);
basic_string& operator=(charT c);

iterator begin();
const_iterator begin() const;
iterator end();
const_iterator end() const;
reverse_iterator rbegin();
const_reverse_iterator rbegin() const;
reverse_iterator rend();
const_reverse_iterator rend() const;

size_type size() const;
size_type length() const;
size_type max_size() const;
void resize(size_type n, charT c);
void resize(size_type n);
size_type capacity() const;
void reserve(size_type res_arg = 0);
void clear();
bool empty() const;

const_reference operator[](size_type pos) const;
reference operator[](size_type pos);
const_reference at(size_type n) const;
reference at(size_type n);

basic_string& operator+=(const basic_string& str);
basic_string& operator+=(const charT* s);
basic_string& operator+=(charT c);
```

21 Strings Library

21.3 Class *basic_string*

```
basic_string& append(const basic_string& str);
basic_string& append(
    const basic_string& str,
    size_type pos, size_type n);
basic_string& append(const charT* s, size_type n);
basic_string& append(const charT* s);
basic_string& append(size_type n, charT c);
template<class InputIterator>
basic_string& append
    (InputIterator first, InputIterator last);
void push_back(const charT);
basic_string& assign(const basic_string&);
basic_string& assign
    (const basic_string& str,
    size_type pos, size_type n);
basic_string& assign(const charT* s, size_type n);
basic_string& assign(const charT* s);
basic_string& assign(size_type n, charT c);
template<class InputIterator>
basic_string& assign
    (InputIterator first, InputIterator last);
basic_string& insert
    (size_type pos1, const basic_string& str);
basic_string& insert
    (size_type pos1, const basic_string& str,
    size_type pos2, size_type n);
basic_string& insert
    (size_type pos, const charT* s, size_type n);
basic_string& insert
    (size_type pos, const charT* s);
basic_string& insert
    (size_type pos, size_type n, charT c);
iterator insert(iterator p, charT c = charT());
void insert(iterator p, size_type n, charT c);
template<class InputIterator>
void insert
    (iterator p, InputIterator first,
    InputIterator last);
basic_string& erase
    (size_type pos = 0, size_type n = npos);
```

```
iterator erase(iterator position);
iterator erase(iterator first, iterator last);
basic_string& replace
    (size_type pos1, size_type n1,
     const basic_string& str);
basic_string& replace
    (size_type pos1, size_type n1,
     const basic_string& str,
     size_type pos2, size_type n2);
basic_string& replace
    (size_type pos, size_type n1,
     const charT* s, size_type n2);
basic_string& replace
    (size_type pos, size_type n1, const charT* s);
basic_string& replace
    (size_type pos, size_type n1,
     size_type n2, charT c);
basic_string& replace
    (iterator i1, iterator i2,
     const basic_string& str);
basic_string& replace
    (iterator i1, iterator i2,
     const charT* s, size_type n);
basic_string& replace(
    iterator i1, iterator i2, const charT* s);
basic_string& replace
    (iterator i1, iterator i2,
     size_type n, charT c);
template<class InputIterator>
basic_string& replace
    (iterator i1, iterator i2,
     InputIterator j1, InputIterator j2);

size_type copy
    (charT* s, size_type n,
     size_type pos = 0) const;
void swap(basic_string<charT, traits, Allocator>&);

const charT* c_str() const;
const charT* data() const;
```

21 Strings Library

21.3 Class *basic_string*

```
allocator_type get_allocator() const;
size_type find
    (const basic_string& str,
     size_type pos = 0) const;
size_type find
    (const charT* s, size_type pos,
     size_type n) const;
size_type find
    (const charT* s, size_type pos = 0) const;
size_type find (charT c, size_type pos = 0) const;
size_type rfind
    (const basic_string& str,
     size_type pos = npos) const;
size_type rfind
    (const charT* s, size_type pos,
     size_type n) const;
size_type rfind
    (const charT* s, size_type pos = npos) const;
size_type rfind
    (charT c, size_type pos = npos) const;
size_type find_first_of
    (const basic_string& str,
     size_type pos = 0) const;
size_type find_first_of
    (const charT* s, size_type pos,
     size_type n) const;
size_type find_first_of
    (const charT* s, size_type pos = 0) const;
size_type find_first_of
    (charT c, size_type pos = 0) const;
size_type find_last_of
    (const basic_string& str,
     size_type pos = npos) const;
size_type find_last_of
    (const charT* s,
     size_type pos, size_type n) const;
size_type find_last_of
    (const charT* s, size_type pos = npos) const;
size_type find_last_of
    (charT c, size_type pos = npos) const;
```



```
size_type find_first_not_of
    (const basic_string& str,
     size_type pos = 0) const;
size_type find_first_not_of
    (const charT* s, size_type pos,
     size_type n) const;
size_type find_first_not_of
    (const charT* s, size_type pos = 0) const;
size_type find_first_not_of
    (charT c, size_type pos = 0) const;
size_type find_last_not_of
    (const basic_string& str,
     size_type pos = npos) const;
size_type find_last_not_of
    (const charT* s,
     size_type pos, size_type n) const;
size_type find_last_not_of
    (const charT* s, size_type pos = npos) const;
size_type find_last_not_of
    (charT c, size_type pos = npos) const;
basic_string substr
    (size_type pos = 0, size_type n = npos) const;
int compare(const basic_string& str) const;
int compare(
    size_type pos1, size_type n1,
    const basic_string& str) const;
int compare
    (size_type pos1, size_type n1,
     const basic_string& str,
     size_type pos2, size_type n2) const;
int compare(const charT* s) const;
int compare
    (size_type pos1, size_type n1, const charT* s,
     size_type n2 = npos) const;
};
}
```

21.3.1 Constructors and Assignments

Constructor, destructor and assignment operators and functions.

21 Strings Library

21.3 Class *basic_string*

Constructors

Description The various `basic_string` constructors construct a string object for character sequence manipulations. All constructors include an `Allocator` argument that is used for memory allocation.

Prototype `explicit basic_string
(const Allocator& a = Allocator());`

Remarks This default constructor, constructs an empty string. A zero sized string that may be copied to is created.

```
basic_string  
(const basic_string& str,  
 size_type pos = 0,  
 size_type n = npos,  
 const Allocator& a = Allocator());
```

Remarks This constructor takes a string class argument and creates a copy of that string, with size of the length of that string and a capacity at least as large as that string.

- An exception is thrown upon failure

```
basic_string  
(const charT* s,  
 size_type n,  
 const Allocator& a = Allocator());
```

Remarks This constructor takes a `const char` array argument and creates a copy of that array with the size limited to the `size_type` argument.

- The `charT*` argument shall not be a null pointer
- An exception is thrown upon failure

```
basic_string  
(const charT* s,  
 const Allocator& a = Allocator());
```

Remarks This constructor takes an `const char` array argument. The size is determined by the size of the `char` array.

- The `charT*` argument shall not be a null pointer

```
basic_string  
    (size_type n,  
     charT c,  
     const Allocator& a = Allocator());
```

Remarks This constructor creates a string of size_type n size repeating charT c as the filler.

- A `length_error` is thrown if n is less than `npos`.

```
template<class InputIterator>  
basic_string  
    (InputIterator begin,  
     InputIterator end,  
     const Allocator& a = Allocator());
```

Remarks This iterator string takes InputIterator arguments and creates a string with its first position starts with begin and its ending position is end. Size is the distance between beginning and end.

Destructor

Description Deallocates the memory referenced by the `basic_string` object.

Prototype `~basic_string ();`

Assignment Operator

Description Assigns the input string, char array or char type to the current string.

Prototype `basic_string& operator= (const basic_string& str);`

Remarks If *this and str are the same object has it has no effect.

```
basic_string& operator=(const charT* s);
```

Remarks Used to assign a NCTCS to a string.

```
basic_string& operator=(charT c);
```

Remarks Used to assign a single char type to a string.

21 Strings Library

21.3 Class *basic_string*

Assignment & Addition Operator *basic_string*

Description	Appends the string <i>rhs</i> to the current string.
Prototype	<pre>string& operator+= (const string& rhs); string& operator+= (const charT* s); string& operator+= (charT s);</pre>
Remarks	Both of the overloaded functions construct a string object from the input <i>s</i> , and append it to the current string.
Return	The assignment operator returns the <code>this</code> pointer.

21.3.2 Iterator Support

Member functions for string iterator support.

begin

Description	Returns an iterator to the first character in the string
Prototype	<pre>iterator begin(); const_iterator begin() const;</pre>

end

Description	Returns an iterator that is past the end value.
Prototype	<pre>iterator end(); const_iterator end() const;</pre>

rbegin

Description	Returns an iterator that is equivalent to <code>reverse_iterator(end())</code> .
Prototype	<pre>reverse_iterator rbegin();</pre>

```
const_reverse_iterator rbegin() const;
```

rend

Description Returns an iterator that is equivalent to `reverse_iterator(begin())`.

Prototype `reverse_iterator rend();`
`const_reverse_iterator rend() const;`

21.3.3 Capacity

Member functions for determining a strings capacity.

size

Description Returns the size of the string.

Prototype `size_type size() const;`

length

Description Returns the length of the string

Prototype `size_type length() const;`

max_size

Description Returns the maximum size of the string.

Prototype `size_type max_size() const;`

resize

Description Resizes the string to size `n`.

Prototype `void resize(size_type n);`
`void resize(size_type n, charT c);`

21 Strings Library

21.3 Class *basic_string*

Remarks If the size of the string is longer than `size_type n`, it shortens the string to `n`, if the size of the string is shorter than `n` it appends the string to size `n` with `charT c` or `charT()` if no filler is specified.

capacity

Description Returns the memory storage capacity.

Prototype `size_type capacity() const;`

reserve

Description A directive that indicates a planned change in memory size to allow for better memory management.

Prototype `void reserve(size_type res_arg = 0);`

clear

Description Erases from `begin()` to `end()`.

Prototype `void clear();`

empty

Description Empties the string stored.

Prototype `bool empty() const;`

Returns Returns `true` if the size is equal to zero, otherwise `false`.

21.3.4 Element Access

Member functions and operators for accessing individual string elements.

operator[]

Description An operator used to access an indexed element of the string.

Prototype `const_reference operator[](size_type pos) const;
reference operator[](size_type pos);`

at

Description A function used to access an indexed element of the string.

Prototype `const_reference at(size_type n) const;
reference at(size_type n);`

21.3.5 Modifiers

Operators for appending a string.

21.3.5.1 operator+=

Description An Operator used to append to the end of a string.

Prototype `basic_string& operator+=(const basic_string& str);

basic_string& operator+=(const charT* s);

basic_string& operator+=(charT c);`

21.3.5.2 append

Member functions for appending to the end of a string.

Description A function used to append to the end of a string.

Prototype `basic_string& append(const basic_string& str);

basic_string& append(
 const basic_string& str,
 size_type pos, size_type n);

basic_string& append(const charT* s, size_type n);

basic_string& append(const charT* s);`

21 Strings Library

21.3 Class *basic_string*

```
basic_string& append(size_type n, charT c);
```

```
template<class InputIterator>  
basic_string& append  
    (InputIterator first, InputIterator last);
```

21.5.3.3 assign

Description Assigns a string, Null Terminated Character Type Sequence or char type to the string.

Prototype `basic_string& assign(const basic_string&);`

```
basic_string& assign  
    (const basic_string& str,  
     size_type pos, size_type n);
```

```
basic_string& assign(const charT* s, size_type n);
```

```
basic_string& assign(const charT* s);
```

```
basic_string& assign(size_type n, charT c);
```

```
template<class InputIterator>  
basic_string& assign  
    (InputIterator first, InputIterator last);
```

Remarks If there is a size argument whichever is smaller the string size or argument value will be assigned.

21.3.5.4 insert

Description Inserts a string, Null Terminated Character Type Sequence or char type into the string.

Prototype `basic_string& insert
 (size_type pos1, const basic_string& str);`

```
basic_string& insert  
    (size_type pos1, const basic_string& str,
```



```
size_type pos2, size_type n);

basic_string& insert
(size_type pos, const charT* s, size_type n);

basic_string& insert
(size_type pos, const charT* s);

basic_string& insert
(size_type pos, size_type n, charT c);
iterator insert(iterator p, charT c = charT());

void insert(iterator p, size_type n, charT c);

template<class InputIterator>
void insert
(iterator p, InputIterator first,
InputIterator last);
```

Remarks May throw an exception,

21.3.5.5 erase

Description Erases the string

Prototype `basic_string& erase`
`(size_type pos = 0, size_type n = npos);`
`iterator erase(iterator position);`
`iterator erase(iterator first, iterator last);`

Remarks May throw an exception,

21.3.5.6 replace

Description Replaces the string with a `string`, Null Terminated Character Type Sequence or `char` type.

Prototype `basic_string& replace`
`(size_type pos1, size_type n1,`
`const basic_string& str);`

21 Strings Library

21.3 Class *basic_string*

```
basic_string& replace
    (size_type pos1, size_type n1,
     const basic_string& str,
     size_type pos2, size_type n2);

basic_string& replace
    (size_type pos, size_type n1,
     const charT* s, size_type n2);

basic_string& replace
    (size_type pos, size_type n1, const charT* s);

basic_string& replace
    (size_type pos, size_type n1,
     size_type n2, charT c);

basic_string& replace
    (iterator i1, iterator i2,
     const basic_string& str);

basic_string& replace
    (iterator i1, iterator i2,
     const charT* s, size_type n);

basic_string& replace(
    iterator i1, iterator i2, const charT* s);

basic_string& replace
    (iterator i1, iterator i2,
     size_type n, charT c);

template<class InputIterator>
basic_string& replace
    (iterator i1, iterator i2,
     InputIterator j1, InputIterator j2);
```

Remarks May throw an exception,

21.3.5.7 copy

Description Copies a Null Terminated Character Type Sequence to a string up to the size designated.

Prototype `size_type copy
(charT* s, size_type n,
size_type pos = 0) const;`

Remarks The function `copy` does not pad the string with Null characters.

21.3.5.8 swap

Description Swaps one string for another.

Prototype `void swap(basic_string<charT,traits,Allocator>&);`

21.3.6 String Operations

Member functions for sequences of character operations.

`c_str`

Description Returns the string as a Null terminated character type sequence.

Prototype `const charT* c_str() const;`

`data`

Description Returns the string as an array without a Null terminator.

Prototype `const charT* data() const;`

`get_allocator`

Description Returns a copy of the `allocator` object used to create the string.

21 Strings Library

21.3 Class *basic_string*

Prototype `allocator_type get_allocator() const;`

21.3.6.1 find

Description Finds a string, Null Terminated Character Type Sequence or char type in a string starting from the beginning.

Prototype `size_type find
 (const basic_string& str,
 size_type pos = 0) const;`

`size_type find
 (const charT* s, size_type pos,
 size_type n) const;`

`size_type find
 (const charT* s, size_type pos = 0) const;`

`size_type find (charT c, size_type pos = 0) const;`

Returns The found position or `npos` if not found.

21.3.6.2 rfind

Description Finds a string, Null Terminated Character Type Sequence or char type in a string testing backwards from the end.

Prototype `size_type rfind
 (const basic_string& str,
 size_type pos = npos) const;`

`size_type rfind
 (const charT* s, size_type pos,
 size_type n) const;`

`size_type rfind
 (const charT* s, size_type pos = npos) const;`

`size_type rfind`

```
(charT c, size_type pos = npos) const;
```

Returns The found position or `npos` if not found.

21.6.3.3 `find_first_of`

Description Finds the first position of one of the elements in the function's argument starting from the beginning.

Prototype

```
size_type find_first_of
(const basic_string& str,
 size_type pos = 0) const;

size_type find_first_of
(const charT* s, size_type pos,
 size_type n) const;

size_type find_first_of
(const charT* s, size_type pos = 0) const;

size_type find_first_of
(charT c, size_type pos = 0) const;
```

Returns The found position or `npos` if not found.

21.3.6.4 `find_last_of`

Description Finds the last position of one of the elements in the function's argument starting from the beginning.

Prototype

```
size_type find_last_of
(const basic_string& str,
 size_type pos = npos) const;

size_type find_last_of
(const charT* s,
 size_type pos, size_type n) const;

size_type find_last_of
(const charT* s, size_type pos = npos) const;
```

21 Strings Library

21.3 Class *basic_string*

```
size_type find_last_of  
(charT c, size_type pos = npos) const;
```

Returns The found position or `npos` if not found.

21.3.6.5 `find_first_not_of`

Description Finds the first position that is not one of the elements in the function's argument starting from the beginning.

Prototype

```
size_type find_first_not_of  
(const basic_string& str,  
size_type pos = 0) const;
```

```
size_type find_first_not_of  
(const charT* s, size_type pos,  
size_type n) const;
```

```
size_type find_first_not_of  
(const charT* s, size_type pos = 0) const;
```

```
size_type find_first_not_of  
(charT c, size_type pos = 0) const;
```

Returns The found position or `npos` if not found.

21.3.6.6 `find_last_not_of`

Description Finds the last position that is not one of the elements in the function's argument starting from the beginning.

Prototype

```
size_type find_last_not_of  
(const basic_string& str,  
size_type pos = npos) const;
```

```
size_type find_last_not_of  
(const charT* s,  
size_type pos, size_type n) const;
```

```
size_type find_last_not_of
```

```
(const charT* s, size_type pos = npos) const;
```

```
size_type find_last_not_of  
(charT c, size_type pos = npos) const;
```

Returns The found position or `npos` if not found.

21.3.6.7 substr

Description Returns a string if possible from beginning at the first arguments position to the last position.

Prototype `basic_string substr`
`(size_type pos = 0, size_type n = npos) const;`

Remarks May throw an exception,

21.3.6.8 compare

Description Compares a string, substring or Null Terminated Character Type Sequence with a lexicographical comparison.

Prototype `int compare(const basic_string& str) const;`

```
int compare(  
    size_type pos1, size_type n1,  
    const basic_string& str) const;
```

```
int compare  
    (size_type pos1, size_type n1,  
    const basic_string& str,  
    size_type pos2, size_type n2) const;
```

```
int compare(const charT* s) const;
```

```
int compare  
    (size_type pos1, size_type n1, const charT* s,  
    size_type n2 = npos) const;
```

21 Strings Library

21.3 Class *basic_string*

Returns Less than zero if the string is smaller than the argument lexicographically, zero if the string is the same size as the argument lexicographically and greater than zero if the string is larger than the argument lexicographically.

23.3.7 Non-Member Functions and Operators

21.3.7.1 `operator+`

Description Appends one string to another.

Prototype

```
template
    <class charT, class traits, class Allocator>
    basic_string<charT,traits,Allocator>
operator+
    (const basic_string
     <charT,traits, Allocator>& lhs,
     const basic_string
     <charT,traits,Allocator>& rhs);

template
    <class charT, class traits, class Allocator>
    basic_string<charT,traits,Allocator>

operator+
    (const charT* lhs,
     const basic_string
     <charT,traits,Allocator>& rhs);

template
    <class charT, class traits, class Allocator>
    basic_string<charT,traits,Allocator>
operator+
    (charT lhs,
     const basic_string
     <charT,traits,Allocator>& rhs);

template
    <class charT, class traits, class Allocator>
```



```

        basic_string<charT,traits,Allocator>
operator+
    (const basic_string
     <charT,traits,Allocator>& lhs,
     const charT* rhs);

template
    <class charT, class traits, class Allocator>
    basic_string<charT,traits,Allocator>
operator+
    (const basic_string
     <charT,traits,Allocator>& lhs, charT rhs);

```

Returns The combined strings.

21.3.7.2 operator==

Description Test for lexicographical equality.

Prototype

```

template
    <class charT, class traits, class Allocator>
bool operator==
    (const basic_string
     <charT,traits,Allocator>& lhs,
     const basic_string
     <charT,traits,Allocator>& rhs);

template
    <class charT, class traits, class Allocator>
bool operator==
    (const charT* lhs,
     const basic_string
     <charT,traits,Allocator>& rhs);

template
    <class charT, class traits, class Allocator>
bool operator==
    (const basic_string
     <charT,traits,Allocator>& lhs,
     const charT* rhs);

```

21 Strings Library

21.3 Class *basic_string*

Returns True if the strings match otherwise false.

21.3.7.3 operator!=

Description Test for lexicographical inequality.

Prototype

```
template
    <class charT, class traits, class Allocator>
bool operator!=
    (const basic_string
     <charT,traits,Allocator>& lhs,
     const basic_string
     <charT,traits,Allocator>& rhs);

template
    <class charT, class traits, class Allocator>
bool operator!=
    (const charT* lhs,
     const basic_string
     <charT,traits,Allocator>& rhs);

template
    <class charT, class traits, class Allocator>
bool operator!=
    (const basic_string
     <charT,traits,Allocator>& lhs,
     const charT* rhs);
```

Returns True if the strings do not match otherwise false.

21.3.7.4 operator<

Description Tests for a lexicographically less than condition.

Prototype

```
template
    <class charT, class traits, class Allocator>
bool operator<
    (const basic_string
     <charT,traits,Allocator>& lhs,
     const basic_string
```

```
<charT,traits,Allocator>& rhs);
```

```
template
    <class charT, class traits, class Allocator>
bool operator<
    (const charT* lhs,
     const basic_string
        <charT,traits,Allocator>& rhs);
```

```
template
    <class charT, class traits, class Allocator>
bool operator<
    (const basic_string
        <charT,traits,Allocator>& lhs,
     const charT* rhs);
```

Returns True if the first argument is lexicographically less than the second argument otherwise false.

21.3.7.5 operator>

Description Tests for a lexicographically greater than condition.

Prototype

```
template
    <class charT, class traits, class Allocator>
bool operator>
    (const basic_string
        <charT,traits,Allocator>& lhs,
     const basic_string
        <charT,traits,Allocator>& rhs);
```

```
template
    <class charT, class traits, class Allocator>
bool operator>
    (const charT* lhs,
     const basic_string
        <charT,traits,Allocator>& rhs);
```

```
template
    <class charT, class traits, class Allocator>
```

21 Strings Library

21.3 Class *basic_string*

```
bool operator>
    (const basic_string
     <charT,traits,Allocator>& lhs,
     const charT* rhs);
```

Returns True if the first argument is lexicographically greater than the second argument otherwise false.

21.3.7.6 operator<=

Description Tests for a lexicographically less than or equal to condition.

Prototype

```
template
    <class charT, class traits, class Allocator>
bool operator<=
    (const basic_string
     <charT,traits,Allocator>& lhs,
     const basic_string
     <charT,traits,Allocator>& rhs);
```

```
template
    <class charT, class traits, class Allocator>
bool operator<=
    (const charT* lhs,
     const basic_string
     <charT,traits,Allocator>& rhs);
```

```
template
    <class charT, class traits, class Allocator>
bool operator<=
    (const basic_string
     <charT,traits,Allocator>& lhs,
     const charT* rhs);
```

Returns True if the first argument is lexicographically less than or equal to the second argument otherwise false.

21.3.7.7 operator>=

Description Tests for a lexicographically greater than or equal to condition.

Prototype

```
template
    <class charT, class traits, class Allocator>
bool operator>=
    (const basic_string
     <charT,traits,Allocator>& lhs,
     const basic_string
     <charT,traits,Allocator>& rhs);

template
    <class charT, class traits, class Allocator>
bool operator>=
    (const charT* lhs,
     const basic_string
     <charT,traits,Allocator>& rhs);

template
    <class charT, class traits, class Allocator>
bool operator>=
    (const basic_string
     <charT,traits,Allocator>& lhs,
     const charT* rhs);
```

Returns True if the first argument is lexicographically greater than or equal to the second argument otherwise false.

21.3.7.8 swap

Description This non member `swap` exchanges the first and second arguments.

Prototype

```
template
    <class charT, class traits, class Allocator>
void swap
    (basic_string<charT,traits,Allocator>& lhs,
     basic_string
     <charT,traits,Allocator>& rhs);
```

21.3.7.9 Inserters and extractors

Overloaded inserters and extractors for *basic_string* types.

operator>>

Description Overloaded `extractor` for stream input operations.

Prototype

```
template
<class charT, class traits, class Allocator>
basic_istream<charT,traits>&
operator>>
(basic_istream<charT,traits>& is,
 basic_string<charT,traits,Allocator>& str);
```

Remarks Characters are extracted and appended until `n` characters are stored or `end-of-file` occurs on the input sequence;

operator<<

Description Inserts characters from a string object from into a output stream.

Prototype

```
template
<class charT, class traits, class Allocator>
basic_ostream<charT, traits>&
operator<<
(basic_ostream<charT, traits>& os,
 const basic_string
 <charT,traits,Allocator>& str);
```

getline

Description Extracts characters from a `stream` and appends them to a `string`.

Prototype

```
template
<class charT, class traits, class Allocator>
basic_istream<charT,traits>&
getline
(basic_istream<charT,traits>& is,
```

```
basic_string
    <charT,traits,Allocator>& str,
    charT delim);
template
    <class charT, class traits, class Allocator>
    basic_istream<charT,traits>&
getline
    (basic_istream<charT,traits>& is,
     basic_string<charT,traits,Allocator>& str)
```

Remarks Extracts characters from a stream and appends them to the string until the end-of-file occurs on the input sequence (in which case, the `getline` function calls `setstate(eofbit)` or the delimiter is encountered in which case, the delimiter is extracted but not appended.

If the function extracts no characters, it calls `setstate(failbit)` in which case it may throw an exception.

23.4 Null Terminated Sequence Utilities

The standard requires C++ versions of the standard libraries for use with characters and Null Terminated Character Type Sequences.

Character Support

The standard provides for namespace and character type support.

Table 6.2 Headers `<cctype.h>` and `<cwctype.h>`

<code><cctype.h></code>	<code><cwctype.h></code>	<code><cctype.h></code>	<code><cwctype.h></code>
<code>isalnum</code>	<code>iswalnum</code>	<code>isprint</code>	<code>iswprint</code>
<code>isalpha</code>	<code>iswalpha</code>	<code>ispunct</code>	<code>iswpunct</code>
<code>iscntrl</code>	<code>iswcntrl</code>	<code>isspace</code>	<code>iswspace</code>
<code>isdigit</code>	<code>iswdigit</code>	<code>isupper</code>	<code>iswupper</code>
<code>isgraph</code>	<code>iswgraph</code>	<code>isxdigit</code>	<code>iswxdigit</code>

<cctype.h>	<cwctype.h>	<cctype.h>	<cwctype.h>
isalnum	iswalnum	isprint	iswprint
islower	iswlower		
tolower	towlower	toupper	towupper
	wctype		iswctype
	wctrans		towctrans
Macros		EOF	WEOF

String Support

The standard provides for namespace and wide character type for Null Terminated Character Type Sequence functionality.

Table 6.3 Headers <cstring.h> and <wchar.h>

<cstring.h>	<wchar.h>	<cstring.h>	<wchar.h>
memchr	wmemchr	strerror	
memcmp	wmemcmp	strlen	wcslen
memcpy	wmemcpy	strncat	wcsncat
memmove	wmemmove	strncmp	wcsncmp
memset	wmemset	strncpy	wcsncpy
strcat	wscat	strpbrk	wcspbrk
strchr	wcschr	strrchr	wcsrchr
strcmp	wcscmp	strspn	wcsspn
strcoll	wscoll	strstr	wcsstr
strcpy	wscpy	strtok	wcstok
strcspn	wcscspn	strxfrm	wcsxfrm
Type	mbstate_t	size_t	wint_t

<cstring.h>	<wchar.h>	<cstring.h>	<wchar.h>
memchr	wmemchr	strerror	
memcmp	wmemcmp	strlen	wcslen
memcpy	wmemcpy	strncat	wcsncat
memmove	wmemmove	strncmp	wcsncmp
memset	wmemset	strncpy	wcsncpy
Macro	NULL	NULL	WCHAR_MAX
Macro			WCHAR_MIN

Input and Output Manipulations

The standard provides for namespace and wide character support for manipulation and conversions of input and output and character and character sequences.

Table 6.4 Additional <wchar.h> and <stdlib.h> support

wchar.h	wchar.h	wchar.h	<stdlib.h>
btowc	mbrtowc	wcrtomb	atol
fgetwc	mbsinit	wcscoll	atof
fgetws	mbsrtowcs	wcsftime	atoi
fputwc	putwc	wcstod	mblen
fputws	putwchar	wcstol	mbstowcs
fwide	swscanf	wcsrtombs	mbtowc
fwprintf	swprintf	wcstoul	strtod
fwscanf	ungetwc	wctob	strtol
getwc	vfwprintf	wprintf	strtoul
getwchar	vwprintf	wscanf	wctomb
mbrlen	vswprintf		wcstombs

21 Strings Library

23.4 Null Terminated Sequence Utilities



Localization Library

This chapter is a reference guide to the ANSI/ISO standard C++ Localization library and is based on the April 1995 Draft Working Paper of the ANSI/ISO committee.

Overview of the Localization Library

C++ localization library extends the internationalization facilities provided by the C library in such a way that will help programmers to encapsulate the cultural differences.

Files `#include <mlocale.h>`
 `#include <locale.h>`

Description The localization library is a set of classes that help C++ programmers to encapsulate the cultural differences that come up especially while porting an application across different user communities. Hence, this library provides a set of classes that include internationalization support for character classification, string collation, formatting and parsing of date, time, numeric and monetary quantities, message retrieval etc.

In different countries people use different formats for formatting date, time, currency etc., For example in India, the date is written using the DD/MM/YY (D-Day, M-Month and Y-Year) format while in the USA, it is the MM/DD/YY format. This difference (though, “minor”) may cause problems when any software system is being used by people belonging to different communities. So, as soon as the ISO C standard was published, the work on improvising the internationalization support began and this work was published as the ISO C Amendment I.

The C++ support for internationalization is summarized in two header files. The header `<mlocale>` declares the set of classes like

Localization Library

Class locale

locale, facet etc., which are part of the C++ standard library and the header `<locale>` declares the elements of localization library from the standard C library.

This section summarizes the classes and convenience interfaces provided by the C++ library for internationalization. The topics in this chapter are:

- [“Class locale” on page 148](#)
- [“Class facet” on page 156](#)
- [“Class id” on page 158](#)
- [“The Numerics Category” on page 158](#)
- [“The Collate Category” on page 165](#)
- [“Ctype Category” on page 167](#)
- [“The Monetary Category” on page 178](#)

Class locale

Class `locale` is used for implementing a type-safe polymorphic collection of facets (feature-sets) indexed by facet types. The following sections illustrate the components of the class `locale`.

The topics are:

- [“Typedef Declarations” on page 148](#)
- [“Public Data Members” on page 150](#)
- [“Constructors” on page 150](#)
- [“Public member operators” on page 152](#)
- [“Public Member Functions” on page 153](#)
- [“Static Member Functions” on page 155](#)
- [“Global Operators” on page 156](#)

Typedef Declarations

Description	Category, when given a legal value, represents a collection of facets. Valid category values include 0 and the <code>locale</code> member bit-
--------------------	------------------------------------------------------------------------------------------------------------------------------------------------

mask elements `collate`, `ctype`, `monetary`, `numeric`, `time` and `messages`. In addition, `locale` member `all` is defined such that the expression

```
(collate | ctype | monetary | numeric | time | messages) == all
```

The categories and their corresponding facets are given below.

Definition `typedef unsigned category;`

Table 7.1 Categories and corresponding facets

Category	Include Facets
<code>collate</code>	<code>collate<char></code> , <code>collate<wchar_t></code>
<code>ctype</code>	<code>ctype<char></code> , <code>ctype<wchar_t></code> , <code>codecvt<char,</code> <code> wchar_t, mbstate_t></code> , <code>codecvt<wchar_t,</code> <code> char, mbstate_t></code>
<code>monetary</code>	<code>moneypunct <char, bool></code> , <code>moneypunct <wchar_t, bool></code> , <code>money_get<char, bool, II></code> , <code>money_get<wchar_t, bool, II></code> , <code>money_put<char, bool, II></code> , <code>money_put<wchar_t, bool, II></code>
<code>numeric</code>	<code>numput<char></code> , <code>numput<wchar_t></code> , <code>num_get<char, II></code> , <code>num_get<wchar_t, II></code> , <code>num_put<char, OI></code> , <code>num_put<wchar_t, OI></code>

Localization Library

Class locale

Category	Include Facets
time	time_get<char, bool, II>, time_get<wchar_t, bool, II> time_put<char, bool, II>, time_put<wchar_t, bool, II>
messages	messages<char>, messages<wchar_t>

Public Data Members

The class `locale` has two nested classes in it, ["Class facet" on page 156](#) and ["Class id" on page 158](#). The ["Class facet" on page 156](#) acts as a base class for all locale facets.

Constructors

Default Constructor

Description Default constructor constructs a locale which is a snapshot of the current global locale. The current global locale can be set by calling either the standard "C" function `setlocale()` or `locale::global(const locale&)`.

Prototype `locale () throw();`

Overloaded and Copy Constructors

Prototype `locale (const locale& other) throw();`

Description Copy constructor constructs an instance of the class `locale` which is a copy of the locale `other`.

Prototype `explicit locale (const char* std_name);`

Remarks Constructs a locale using the standard C locale names (for example, "POSIX"). If the argument `std_name` passed to it is not a valid

name, then a `runtime_error` exception is thrown. The valid set of `std_name` include "C", null string and any other implementation-defined value.

Prototype `locale`
 `(const locale& other,`
 `const char* std_name,`
 `category cat);`

Remarks Creates a locale as a copy of `other` except for the facets identified by the category argument `cat`, for which the semantics will be the same as that of the second argument `std_name`. The resulting locale will have a name if and only if `other` has a name.

Prototype `template`
 `< class Facet >`
 `locale`
 `(const locale& other,`
 `Facet* f);`

Remarks A locale is constructed which does not have a name. The constructed locale has all the features set from the first locale argument `other`, except that of the type `Facet`, for which the second argument is used.

Prototype `template`
 `< class Facet >`
 `locale`
 `(const locale& other,`
 `const locale& one);`

Remarks A locale which does not have a name is constructed such that all the facets are incorporated from the first argument `other` and that facet which is identified by the template parameter `Facet`, is incorporated from the second argument `one`. If the second argument does not have a facet of that particular type, `runtime_error` exception is thrown.

Prototype `locale`
 `(const locale& other,`

Localization Library

Class locale

```
const locale& one,  
category cat);
```

Remarks A locale is constructed which has all the facets from the first argument *other*, except those facet(s) that are specified by the category argument *cat*. These facets that are specified by the categories are installed from the second argument. The constructed locale will have a name if and only if the first two locales are named.

Public member operators

The locale class has four public member operators.

- [Assignment Operator= locale](#)
- [Equality Operator== locale](#)
- [Not Equal Operator!= locale](#)
- [Grouping Operator locale](#)

Assignment Operator= locale

Description Creates a copy of *other*, replacing the current value.

Prototype

```
const locale& operator =  
    (const locale& other) const;
```

Equality Operator== locale

Description The function returns true if and only if any of the following conditions are satisfied.

- Both arguments are the same locale
- If one is a copy of the other
- Both locales have a name and they are identical

Prototype

```
bool operator == (const locale& other) const;
```


Not Equal Operator!= locale

Description Evaluates the following expression and returns the result: `!(*this == other)`.

Prototype `bool operator != (const locale& other) const;`

Grouping Operator locale

Description Compares two strings according to the `collate<charT>` facet of `locale`. This member operator satisfied requirements for a comparator predicate template argument as applied to strings.

Prototype `template <class charT, class IS_Traits>
bool operator()
(const basic_string<charT,
IS_Traits>& s1, const basic_string<charT,
IS_Traits>& s2) const;`

Remarks Since member templates are not supported, our library does not provide this function yet.

Public Member Functions

Table 7.2 The public member functions are

locale::name

[locale::name](#)

[locale::has](#)

[locale::use](#)

Description The name of `*this`, if it has one; otherwise, the string `""`.

Prototype `const basic_string<char>& name() const;`

locale::has

Description An indication whether the facet requested is present in **this*. Also, see *use*.

Prototype

```
template <class Facet> bool has() const;  
template <class Facet> bool has (  
    const locale& loc, Facet* f);
```

Remarks The semantics of this function is the same as explained above, except that locale argument is also passed as an argument, instead of being implicit.

See Also ["locale::use" on page 154](#).

locale::use

Description The function returns a reference to the Facet of the locale. If the facet requested is not present in the locale on which the function was applied but present in the current global locale, returns the global locale's instance of Facet.

Prototype

```
template <class Facet> const Facet& use() const;  
template <class Facet> const Facet& use (  
    const locale& loc, Facet* f);
```

Remarks The semantics of this function is the same as explained above, except that locale argument is also passed as an argument, instead of being implicit.

See Also ["locale::has" on page 154](#).

Static Member Functions

Table 7.3 The static member functions are:

locale::global

[locale::global](#)

[locale::classic](#)

[locale::transparent](#)

Description The function sets the global locale to its argument. Subsequent calls to the default constructor and of other library functions affected by the function `setlocale()`, use the locale `loc` until the next call to this function or `setlocale()`.

Prototype `static locale global (const locale& loc);`

locale::classic

Description A call to this member returns the standard “C” locale. This is equivalent to constructing a locale using a call to the constructor `locale(“C”)`.

Prototype `static const locale& classic();`

locale::transparent

Description This function returns the continuously updated global locale. A locale which implements semantics that vary dynamically as the global locale is changed.

Prototype `static locale transparent();`

Global Operators

Table 7.4 The global operators are:

Extractor Operator >> locale

[Extractor Operator >> locale](#)

[Inserter Operator << locale](#)

Description This function tries to read a line into a string and construct a locale from it. If either operation fails, sets failbit of streams. This operator is not yet implemented in our library.

Prototype

```
template
    < class charT,
      class Traits>
basic_istream<charT, Traits>& operator >>
    (basic_istream<charT,
     Traits>& s,
     locale& loc);
```

Inserter Operator << locale

Description The usual stream output operator for locales.

Prototype

```
template
    < class charT,
      class Traits >
basic_ostream<charT, Traits>& operator <<
    (basic_ostream<charT,
     Traits>& s,
     locale& loc);
```

Class facet

Description Class facet is the base class for locale feature sets. Declaration of the class facet is given below.

Prototype class locale::facet {

```
protected :
    facet (size_t refs = 0);
    virtual ~facet ();
private :
    facet (const facet&); // not defined
    void operator= (const facet&); // not
defined
};
```

Remarks A class is a facet if it is publicly derived from another facet, or if it is a class derived from `locale::facet` and containing a declaration as follows:
`static locale::id id;`

See also [“Class id” on page 158](#)

A program that passes a type that is not a facet (explicit or deduced) as a parameter to a locale function expecting a facet, is ill-formed.

The `refs` argument to the constructor is used for lifetime management. If (`refs == 0`) the facet’s lifetime is managed by the locale or locales it is incorporated into. If (`refs == 1`) its lifetime is until explicitly deleted.

For some standard facets (say `FACET`), a standard “`FACET_byname`” class, derived from it, implements the semantics equivalent to that facet of the locale constructed by `locale (const char*)`. Each `FACET_byname` facet provides a constructor,
`FACET_byname(const char* name, size_t refs = 0);`

where `name` names the locale, and the `refs` argument is passed to the base class constructor. If there is no “`..._byname`” version of a particular facet, the base class implements such semantics itself.

Prototype

```
class Facet : public locale::facet {
    protected :
        virtual <return_type> do_F (...) const;
        //....
    public :
        <return_type> F(...) const
            { return do_F (...); }
```

Localization Library

Class id

```
//...

Facet(size_t refs = 1) : locale::facet(refs){}

protected :
    ~Facet () {}
};
```

Remarks There are 6 different categories in which the facets provided by C++ localization library are divided. They are `collate`, `ctype`, `monetary`, `numeric`, `time`, `messages`. Each of the standard categories includes a family of facets. See also:

- [“The Numerics Category” on page 158](#)
- [“The Collate Category” on page 165](#)
- [“Ctype Category” on page 167](#)
- [“The Monetary Category” on page 178](#)

Class id

Description This class is used for identification of a locale facet. A mandatory object of this class used as an index in every derived facet facilitates lookup and initialization of facets. This type-id mechanism ensures that every facet type installed in a locale is assigned a unique id. Id is for every facet type and not for every facet object.

See also [“Class facet” on page 156](#)

The Numerics Category

The numerics category consists of three templated classes. These classes handle all the numeric formatting and parsing. These classes are publicly derived from `locale::facet`.

The classes are:

- [“Class numpunct” on page 159](#)
- [“Class num_get” on page 161](#)

- [“Class num_put” on page 163](#)

Classes `num_get` and `num_put` use `numpunct` for numeric formatting and parsing.

Class `numpunct`

Description This class specifies numeric punctuation. The base class provides classic “C” numeric formats, while `numpunct_byname` version supports named locale (e.g. POSIX, X/OPEN) numeric formatting semantics. Other two numeric facets, `num_get` and `num_put` use `numpunct` facet installed in a particular locale to parse/format numeric quantities.

The topics discussed for this class are:

- [“Typedef Declarations `numpunct`” on page 159](#)
- [“Public Member Functions `numpunct`” on page 160](#)

Typedef Declarations `numpunct`

Description The following typedef’s are defined in the class `numpunct`.

`numpunct::char_type`

Definition `typedef charT char_type;`

`numpunct::string`

Definition `typedef basic_string<charT> string;`

Public Member Functions *numpunct*

Table 7.5 The public member functions are:

numpunct::decimal_point

[*numpunct::decimal_point*](#)

[*numpunct::thousands_sep*](#)

[*numpunct::grouping*](#)

[*numpunct::truename*](#)

[*numpunct::falsename*](#)

Description The function returns a string for use as the decimal radix separator. The base class implementation of this member returns a ".". The `num_get<charT, InputIterator>` class is not required to recognize numbers formatted using a decimal radix separator if it is not a one-character string.

Prototype `string decimal_point() const;`

numpunct::thousands_sep

Description The function returns a string which is the thousand separator. The base class implementation of this member returns the empty string. The `num_get<charT, InputIterator>` class is not required to recognize the numbers formatted using a thousand separator if is not a one-character string.

Prototype `string thousands_sep() const;`

numpunct::grouping

Description This function returns a vector `vec` in which `vec[i]` represents the number of digits in the group at position `i` starting with 0 as the rightmost group. If `vec.size() <= i`, the number is the same as `group(i-1)`; if `(i < 0 || vec[i] <= 0)`, the size of the digit group is unlimited. The base-class implementation this returns the empty vector.

Prototype `vector<char> grouping() const;`

num_punct::truename

Description Returns a string representing the name of the boolean value `true`. The base class implementation return the strings `"true"`.

Prototype `string truename() const;`

num_punct::falsename

Description Returns a string representing the name of the boolean value `false`. The base class implementation return the strings `"false"`.

Prototype `string falsename() const;`

Class num_get

Description Template class `num_get` has the following set of typedef's and public member functions.

- ["Typedef Declarations num_get" on page 161](#)
- ["Public Member Functions num_get" on page 162](#)

Typedef Declarations num_get

Description The following typedef's are defined in the class `num_get`.

char_type

Definition `typedef charT char_type;`

iter_type

Definition `typedef InputIterator iter_type;`

ios

Definition `typedef basic_ios<charT> ios;`

Public Member Functions num_get

There is one function.

num_get::get

Description All the above functions read characters from `in`, interpreting them according to flags set in `str` and the `ctype<charT>` facet and `num_punct<charT>` facet installed in the locale `loc`. These functions ignore the state of `str` initially. But they indicate failure by calling `str.setstate (ios_base::failbit)`. If an error occurs, `v` is not changed; otherwise it is set to the resulting value. Digit grouping separators are optional; if present, digit grouping is checked after the entire number is read. When reading a non-numeric boolean value, the names are compared exactly.

Prototype `iter_type get
 (iter_type in,
 iter_type end,
 ios& str,
 const locale&
 bool& v) const;`
`iter_type get
 (iter_type in,
 iter_type end,
 ios& str,
 const locale&
 long& v) const;`
`iter_type get
 (iter_type in,
 iter_type end,
 ios& str,
 const locale&
 unsigned long& v) const;`
`iter_type get`

```
        (iter_type in,  
         iter_type end,  
         ios& str,  
         const locale&,  
         double& v) const;  
iter_type get  
        (iter_type in,  
         iter_type end,  
         ios& str,  
         const locale&,  
         long double& v) const;
```

Class `num_put`

Description Template class `num_put` has the following set of typedef's and public member functions.

- [“Typedef Declarations `num_put`” on page 163](#)
- [“Public Member Functions `num_put`” on page 164](#)

Typedef Declarations `num_put`

Description The following typedef's are defined in the class `num_put`.

`char_type`

Definition `typedef charT char_type;`

`iter_type`

Definition `typedef OutputIterator iter_type;`

`ios`

Definition `typedef basic_ios<charT> ios;`

Public Member Functions *num_put*

There is one function.

num_put::put

Description All the above functions write characters to the sequence *out*, formatting *val* according to the flags set in *str*, *ctype*<charT> and *numput*<charT> facets installed inside locale *loc*. These functions insert digit group separators as specified by *numput*<charT>::grouping().

Prototype

```
iter_type put
    (iter_type out,
     ios& str,
     const locale& loc,
     bool val) const;
iter_type put
    (iter_type out,
     ios& str,
     const locale& loc,
     long val) const;
iter_type put
    (iter_type out,
     ios& str,
     const locale& loc,
     unsigned long val) const;
iter_type put
    (iter_type out,
     ios& str,
     const locale& loc,
     double val) const;
iter_type put
    (iter_type out,
     ios& str,
     const locale& loc,
     long double val) const;
```

Remarks All the above functions ignore and do not change the stream state. They return an iterator pointing immediately after the last character produced.

The Collate Category

Collate category consists of one template class `collate` which provides features for use in the collation (comparison) and hashing of strings.

See Also [“Class Collate” on page 165](#)

Class Collate

Description This class provides features for use in the collation (comparison) and hashing of strings. A locale member function template, `operator()`, uses the `collate` facet to allow a locale to act directly as the predicate argument for standard algorithms (see Chapter 9) and containers operating on strings. The base class implementation applied lexicographic ordering (see 9.30).

- [“Typedef Declarations `collate`” on page 165](#)
- [“Public Member Functions `collate`” on page 166](#)

Typedef Declarations `collate`

Description The following typedef’s are defined in the class `collate`.

`char_type`

Definition `typedef charT char_type;`

`string`

Definition `typedef basic_string<charT> string;`

Public Member Functions collate

Table 7.6 The public member functions are:

collate::compare

[collate::compare](#)

[collate::transform](#)

[collate::hash](#)

Description Compares the two strings and returns 1 if the first string is greater than the second, -1 if less; zero otherwise.

Prototype

```
int compare
    (const char_type* low1,
     const char_type* high1,
     const char_type* low2,
     const char_type* high2) const;
```

collate::transform

Description This function returns a string that, when compared lexicographically with the result of calling `transform()` on another string, yields the same result as calling `compare()` on the same two strings.

Prototype

```
string transform
    (const char* low,
     const char* high) const;
```

collate::hash

Description Returns an integer value equal to the result of calling `hash()` on any other string for which `compare()` returns 0 (i.e. equal) when passed the two strings.

Prototype

```
long hash
    (const char* low,
     const char* high) const;
```

Ctype Category

In this category, there are two templated classes, `ctype<charT>` and `codecvt<fromtT, toT, stateT>`. There is also a specialization of `ctype<charT>` for the `char` data type.

Class `locale::ctype` encapsulates the C library `ctype` features. This class is typically used by rest of the library classes for character classing. A specialization `locale::ctype<char>` is provided, so that the member functions on type `char` may be implemented inline.

The two templated classes in this category are:

- [“Class `ctype<charT>`” on page 167](#)
- [“Class `ctype<char>` Specialization” on page 171](#)
- [“Class `codecvt`” on page 176](#)

Class `ctype<charT>`

Description The definition of class `ctype_base` which is the base class for `ctype` is given below.

Prototype

```
class ctype_base {
    public :
    enum ctype_mask
    {
        space, print, cntrl, upper, lower,
        alpha, digit, punct, xdigit,
        alnum = alpha | digit, graph = alnum | punct
    };
};
```

NOTE: The type `ctype_mask` is a bitmask type. As noted before, `locale's ctype` facet encapsulates the C library `ctype` features. This facet is derived from `locale::facet` and `ctype_base`.

- [“Typedef Declarations `ctype<charT>`” on page 168](#)

- [“Public Member Functions `ctype<charT>`” on page 168](#)

Typedef Declarations `ctype<charT>`

Description The following typedef’s are defined in the class `Ctype`.

`char_type`

Definition `typedef charT char_type;`

Public Member Functions `ctype<charT>`

Table 7.7 The public member functions are

`ctype::is`

[`ctype::is`](#)

[`ctype::do_is`](#)

[`ctype::scan_is`](#)

[`ctype::scan_not`](#)

[`ctype::toupper`](#)

[`ctype::tolower`](#)

[`ctype::widen`](#)

[`ctype::narrow`](#)

Description This function classify a character or sequence of characters. For each argument character, this function identifies a value `M` of type `ctype_mask`. The function `is()` returns the result `(M & mask) != 0`.

Prototype `bool is
 (ctype_mask mask,
 char_type c) const;`

See Also `do_is()`

`ctype::do_is`

Description This function classify a character or sequence of characters. For each argument character, this function identifies a value `M` of type

ctype_mask. The do_is() function simply places M for all *p where (low <= p && p < high), into vec[p-low] and returns high.

Prototype `const char_type* do_is
 (const char_type* low,
 const char_type* high,
 ctype_mask* vec) const;`

See Also is()

ctype::scan_is

Description This function locates a character in the buffer [low, high) that conforms to the classification mask, mask. It returns the smallest pointer p in the range [low, high) such that (*p) would return true; otherwise, returns high.

Prototype `const char* scan_is
 (ctype_mask mask,
 const char_type* low,
 const char_type* high) const;`

ctype::scan_not

Description This function locates a character in the buffer [low, high) that fails to the classification mask, mask. It returns the smallest pointer p in the range [low, high) such that (*p) would return false; otherwise, returns high.

Prototype `const char* scan_not
 (ctype_mask mask,
 const char_type* low,
 const char_type* high) const;`

ctype::toupper

Description These functions convert a character or sequence of characters to upper-case. The first function returns the corresponding upper-case character if it is known to exist, or its argument if not. The second form replaces each character **p* in the range [*low*, *high*) for which a corresponding upper-case character exists, with that character and returns *high*.

Prototype

```
charT toupper (char_type c) const;  
const char_type* toupper  
    (char_type* low,  
    const char_type* high) const;
```

ctype::tolower

Description These functions convert a character or sequence of characters to lower-case. The first function returns the corresponding lower-case character if it is known to exist, or its argument if not. The second form replaces each character **p* in the range [*low*, *high*) for which a corresponding lower-case character exists, with that character and returns *high*.

Prototype

```
charT tolower (char_type c) const;  
const char_type* tolower  
    (char_type* low,  
    const char_type* high) const;
```

ctype::widen

Description These function apply the simplest reasonable transformation from a *char* value or sequence of *char* values to the corresponding *char_type* value or values. The only characters for which unique transformations are required are the digits, alphabetic characters, '-', '+', newline and space. For any character *c*, the transformed character of *c* is not a member of any character classification that *c* is not also a member of. The first function returns the transformed value and the second form transforms each character **p* in the range

[low, high) placing the result in dest[p-low] and returns high.

Prototype `char_type widen (char c) const;
const char* widen
 (const char* low,
 const char* high,
 char_type* dest) const;`

ctype::narrow

Description These function apply the simplest reasonable transformation from a char_type value or sequence of char_type values to the corresponding char value or values. The only characters for which unique transformations are required are the digits, alphabetic characters, '-', '+', newline and space. For any character c, the transformed character of c is not a member of any character classification that c is not also a member of. In addition, the expression (narrow(c) - '0') evaluates to the digit value of the character. The first function returns the transformed value or default if no mapping is readily available and the second form transforms each character *p in the range [low, high) placing the result(or default if no simplest transformation is readily available) in dest[p-low] and returns high.

Prototype `char narrow
 (char_type c,
 char default) const;
const char_type* narrow
 (const char_type* low,
 const char_type* high,
 char default,
 char* dest) const;`

Class ctype<char> Specialization

ctype<char> specialization is provided so that the member functions on type char can be implemented inline. This specializa-

tion is provided because it affects the derivation interface for ctype<char>.

- [“Data Members ctype<char>” on page 172](#)
- [“Public Member Functions ctype<char>” on page 173](#)

Data Members ctype<char>

The following are the protected or private data members of ctype<char>.

ctype::ctype_mask

Definition `static const ctype_mask;`

ctype::table_

Definition `const ctype_mask* const table_;`

ctype::classic_table_

Definition `static const mask classic_table_[UCHAR_MAX+1];`

ctype::delete_it_

Definition `bool delete_it_;`

Remarks The type `delete_it_` is a private flag,

Public Member Functions ctype<char>

Table 7.8 The public member functions are:

constructor ctype<char>

<u>constructor ctype<char></u>	<u>ctype::is for ctype<char></u>
<u>ctype::scan is for ctype<char></u>	<u>ctype::scan not for ctype<char></u>
<u>ctype::tolower for ctype<char></u>	<u>ctype::toupper for ctype<char></u>
<u>ctype::widen for ctype<char></u>	<u>ctype::narrow for ctype<char></u>

Description This constructor initializes the protected member `table_` with the `tab` argument if nonzero, or the static value `classic_table_` otherwise, and initializes the private member `delete_it_` to `(tab && del)`. The `refs` argument is passed to the base class constructor.

Prototype `ctype`
 `(const ctype_mask* tab = 0,`
 `bool del = false,`
 `size_t refs = 0);`

ctype::is for ctype<char>

Description The first function returns `table_[(unsigned char)c] & mask`. The second function, for all `*p` in the range `[low, high)` assigns `vec[p-low]` to `table_[(unsigned char)*p]` and it returns `high`.

Prototype `bool is`
 `(ctype_mask mask,`
 `char c) const;`
 `const char* is`
 `(const char* low,`

```
const char* high,  
ctype_mask* vec) const;
```

ctype::scan_is for ctype<char>

Description This function returns the smallest *p* in the range [*low*, *high*) such that `table_[(unsigned char)*p] & mask) == true`.

Prototype

```
const char* scan_is  
    (ctype_mask mask,  
    const char* low,  
    const char* high) const;
```

ctype::scan_not for ctype<char>

Description This function returns the smallest *p* in the range [*low*, *high*) such that `table_[(unsigned char)*p] & mask) == false`.

Prototype

```
const char* scan_not  
    (ctype_mask mask,  
    const char* low,  
    const char* high) const;
```

ctype::tolower for ctype<char>

Description These functions convert a character or sequence of characters to lower-case. The first function returns the corresponding lower-case character if it is known to exist, or its argument if not. The second form replaces each character **p* in the range [*low*, *high*) for which a corresponding lower-case character exists, with that character and returns *high*.

Prototype

```
char tolower (char c) const;  
const char* tolower  
    (char* low,  
    const char* high) const;
```

ctype::toupper for ctype<char>

Description These functions convert a character or sequence of characters to upper-case. The first function returns the corresponding upper-case character if it is known to exist, or its argument if not. The second form replaces each character *p in the range [low, high) for which a corresponding upper-case character exists, with that character and returns high.

Prototype

```
char toupper (char c) const;
const char* toupper
    (char* low,
     const char* high) const;
```

ctype::widen for ctype<char>

Description The second function copies contents of the range [low, high) to dest and returns high. The first function returns c.

Prototype

```
char widen (char c) const;
const char* widen
    (const char* low,
     const char* high,
     char* dest) const;
```

ctype::narrow for ctype<char>

Description The second function copies contents of the range [low, high) to dest and returns high. The first function returns c.

Prototype

```
char narrow
    (char c,
     char /* dfault */) const;
const char* narrow (
    const char* low,
    const char* high,
    char /* dfault */,
    char* dest) const;
```

Class codecvt

Description Definition of class `codecvt_base`, used as a base class for `codecvt` facet is given below.

Prototype

```
class codecvt_base {
    public :
        enum result {ok, partial, error, noconv};
};
```

The class `codecvt<fromT, toT, stateT>` is for use when converting from one codeset to another, such as from wide characters to multibyte characters, or vice versa. Instances of this facet are typically used in pairs instantiated oppositely. The `stateT` argument selects the pair of codesets being mapped between. This implementation provides the following two specializations for `codecvt` facet `codecvt<char, wchar_t, mbstate_t>` and `codecvt<wchar_t, char, mbstate_t>`.

- [“Typedef Declarations codecvt” on page 176](#)
- [“Member Functions codecvt” on page 177](#)

Typedef Declarations codecvt

Description The following typedef’s are defined in the class `codecvt`.

from_type

Definition `typedef fromT from_type;`

to_type

Definition `typedef toT to_type;`

state_type

Definition `typedef stateT state_type;`

Member Functions codecvt

There is one function.

Prototype `codecvt::result convert`
 (
 state_type& state,
 const from_type* from,
 const from_type* from_end,
 const from_type*& from_next,
 to_type* to,
 to_type* to_limit,
 to_type*& to_next
) const;

NOTE: This function requires the following conditions to hold true. (`from <= from_end` && `to <= to_end`) and state initialized if at the beginning of a sequence, or else equal to the result of converting the preceding characters in the sequence.

This function translates characters in the range (`from`, `from_end`), placing the results starting at `to`. It stops when it runs out of character to translate or space to put the results, or if it encounters a character it cannot convert. It always leaves the `from_next` and `to_next` pointers pointing one beyond the last character successfully converted. If no translation is needed (returns `codecvt_base::noconv`), sets `to_next` equal to `to`. In any case, this function does not write into `*to_limit`. This function returns an enumeration value of type `codecvt_base::result` as summarized below:

Table 7.9 convert result values

The Monetary Category

Value	Meaning
ok	completed the conversion
partial	ran out of space in the destination
error	encountered a from_type character it could not convert
noconv	no conversion was needed

The monetary category consists of three template classes `money_punct`, `money_put` and `money_get` to handle monetary formatting and parsing. In all the three classes, a template parameter indicates whether local or international monetary formats are to be used. The `money_get<>` and `money_put<>` facets use `money_punct<>` members to determine all formatting details. `money_punct<>` provides basic format information for money processing.

The declaration of the class `money_base`, which is a base class for `money_punct` is given below.

Prototype

```
class money_base {
    public :
        enum part
            {none, space, symbol, sign, value};
        struct pattern
            { char field[4];};
};
```

The classes in the monetary category are:

- [“Class money_punct” on page 179](#)
- [“Class money_put” on page 182](#)
- [“Class money_get” on page 183](#)

Class *money_punct*

Description This class provides money punctuation, similar to `num_punct<>` of the numerics category.

- [“Typedef Declarations *money_punct*” on page 179](#)
- [“Public Member Functions *money_punct*” on page 179](#)

Typedef Declarations *money_punct*

Description The following typedef’s are defined in the class *money_punct*.

char_type

Definition `typedef charT char_type;`

string_type

Definition `typedef basic_string<charT> string_type;`

Public Member Functions *money_punct*

Table 7.10 The public member functions are:

money_punct::decimal_point

money_punct::decimal_point	money_punct::thousands_sep
money_punct::grouping	money_punct::curr_symbol
money_punct::positive_sign	money_punct::negative_sign
money_punct::frac_digits	money_punct::neg_format
money_punct::frac_digits	

Description This function returns the radix separator to use in case `frac_digits()` is greater than zero.

Localization Library

Class *moneypunct*

Prototype `basic_string<charT,traits>
 decimal_point() const;`

`moneypunct::thousands_sep`

Description This function returns the digit group separator to use in case `grouping()` specifies a digit grouping pattern.

Prototype `basic_string<charT,traits>
 thousands_sep() const;`

Remarks The two functions above have been changed to return `charT`, in the April 1995 draft. But this implementation still returns a `string`.

`moneypunct::groupint`

Description This function returns a vector `vec` in which `vec[i]` represents the number of digits in the group at position `i` starting with 0 as the rightmost group. If `vec.size() <= i`, the number is the same as `group(i-1)`; if `(i < 0 || vec[i] <= 0)`, the size of the digit group is unlimited. The base-class implementation this returns the empty vector.

Prototype `vector<char> grouping () const;`

`moneypunct::curr_symbol`

Description This function returns the string to use as the currency identifier symbol.

Prototype `string_type curr_symbol () const;`

`moneypunct::positive_sign`

Description This function returns the string to use to indicate a positive monetary value.

Prototype `string_type positive_sign () const;`

moneypunct::negative_sign

Description This function returns the string to use to indicate a negative monetary value. If this is a one-char-string containing '(', it is paired with a matching ')'.
Prototype `string_type negative_sign () const;`

moneypunct::frac_digits

Description This function returns the number of digits after the decimal radix separator, if any.
Prototype `int frac_digits () const;`

moneypunct::pos_format

Description This function returns a four-element array specifying the order in which the syntactic elements appear in the monetary format. In this array, each element is an enumeration value of type `money_base::part`. Each enumeration value appears exactly once. `none`, if present, is not first; `space`, if present, is neither first nor last. Otherwise, the elements may appear in any order. In international instantiations, the result is always { `symbol`, `sign`, `none`, `value` }.
Prototype `money_base::pattern pos_format () const;`

moneypunct::neg_format

Description This function returns a four-element array specifying the order in which the syntactic elements appear in the monetary format. In this array, each element is an enumeration value of type `money_base::part`. Each enumeration value appears exactly once. `none`, if present, is not first; `space`, if present, is neither first nor last. Otherwise, the elements may appear in any order. In international instantiations, the result is always { `symbol`, `sign`, `none`, `value` }.

Prototype `money_base::pattern neg_format () const;`

Class money_put

Description Class `money_put` contains the following set of typedefs and public member functions.

- [“Typedef Declarations money_put” on page 182](#)
- [“Public Member Functions money_put” on page 182](#)

Typedef Declarations money_put

Description The following typedef’s are defined in the class `money_put`.

char_type

Definition `typedef charT char_type;`

iter_type

Definition `typedef OutputIterator iter_type;`

string

Definition `typedef basic_string<charT> string;`

ios

Definition `typedef basic_ios<charT> ios;`

Public Member Functions money_put

There is one function.

money_put::put

Description Writes characters to *s*, according to the format specified by the `money_punct<charT>` facet of `loc`, and `f.flags()`. Ignores any fractional part of units, or any characters in digits beyond the (optional) leading `'-` and immediately subsequent digits. If format flags specify filling with internal space, the fill characters are placed where `none` or `space` appears in the formatting pattern. Returns an iterator pointing immediately after the last character produced.

Prototype

```
iter_type put
    (iter_type s,
     ios& f,
     const locale& loc,
     double units&) const;
iter_type put
    (iter_type s, ios& f,
     const locale& loc,
     const string& SixDgts) const
```

Class money_get

Description Class `money_get` contains the following set of typedefs and public member functions.

- [“Typedef Declarations money_get” on page 183](#)
- [“Public Member Functions money_get” on page 184](#)

Typedef Declarations money_get

Description The following typedef's are defined in the class `money_get`.

char_type

Definition `typedef charT char_type;`

iter_type

Definition `typedef InputIterator iter_type;`

string

Definition `typedef basic_string<charT> string;`

ios

Definition `typedef basic_ios<charT> ios;`

Public Member Functions money_get

There is one function.

money_get::get

Description These functions read characters from `s` until they have constructed a monetary value, as specified in `str.flags()` and the `money_punct<charT>` facet of `loc`, or until it encounters an error or runs out of characters. The result is a pure sequence of digits, representing a count of the smallest unit of currency representable. Digit group separators are optional; if present, digit grouping is checked after all syntactic elements have been read. Where space or none appear in the format pattern, except at the end optional whitespace is consumed. These functions set `units` or `digits` from the sequence of digits found. `units` is negated, or `digits` is preceded by '-' for a negative value. These functions indicate a failure by calling `str.setstate (failbit)`. On error, `units` or `digits` argument is unchanged. These function return an iterator pointing immediately beyond the last character recognized as part of a valid monetary quantity.

Prototype `iter_type get
 (iter_type s,
 iter_type end,
 ios& str,`


```
        const locale& loc,  
        double& units) const  
iter_type get  
    (iter_type s,  
     iter_type end,  
     ios& str,  
     const locale& loc,  
     string& digits) const
```

Localization Library

Class money_get



Containers Library

This chapter discusses the containers library. These classes support lists, sets, maps, stacks, queues, and more.

Overview of the Containers Library

Standard C++ containers are divided into two broad categories: sequence containers and associative containers.

The topics in this chapter are:

- [“What Are Containers” on page 187](#)
- [“Organization of the Container Class Descriptions” on page 202](#)
- [“Template class vector<T>” on page 203](#)
- [“Template class deque<T>” on page 213](#)
- [“Template class list<T>” on page 222](#)
- [“Template class set<T>” on page 233](#)
- [“Template class multiset<Key>” on page 242](#)
- [“Template class map<Key, T>” on page 252](#)
- [“Template class multimap<Key, T>” on page 263](#)
- [“Template class stack” on page 274](#)
- [“Template class queue” on page 276](#)
- [“Template class priority_queue” on page 278](#)

What Are Containers

This section discusses the concept of a container, and how they work in the containers library. There are several common features of

all the container classes. These features are discussed in this section as well.

The topics in this section are:

- [“Basic Design and Organization of Containers” on page 188](#)
- [“Common Type Definitions in All Containers” on page 190](#)
- [“Common Members of All Containers” on page 189](#)
- [“Common Member Functions in all Containers” on page 192](#)
- [“Sequence Container Requirements” on page 195](#)
- [“Associative Container Requirements” on page 197](#)
- [“Associative Container Types and Member Functions” on page 198](#)

Basic Design and Organization of Containers

Sequence containers include vectors, lists and deques. These contain elements of a single type, organized in a strictly linear arrangement. Although only the three most basic sequence containers are provided in this version of STL, it is possible to construct other sequence containers efficiently using these basic containers through the use of container adaptors, which are STL classes that provide interface mappings. STL provides adaptors for stacks, queues and priority queues.

The second STL container category consists of associative containers, which include sets, multisets, maps and multimaps. Associative containers allow for the fast retrieval of data based on keys. For example, a map (also known as an associative array) allows a user to retrieve an object of type T based on a key of some other type, while sets allow for the fast retrieval of the keys themselves.

All of the STL containers have three important characteristics:

- Every container allocates and manages its own storage.
- Every container provides a minimal set of operations (as member functions) to access and maintain its storage. The provided set of member functions includes:

- Constructors and destructors: these functions allow users to construct and destroy instances of the container. Most containers have several kinds of constructors.
- Element access member functions: these allow users to access the container elements. In most instances, the element access member functions do not change the container.
- Insertion member functions: these are used to insert elements into the container.
- Erase member functions: used to delete elements from the container.
- Each container has an allocator object associated with it. This allocator object encapsulates information about the memory model currently being used, and allows the classes to be portable across various platforms.

The same naming convention is used for the member functions of all containers, resulting in a very uniform interface to all the classes.

There is a great deal of similarity in the interfaces of all STL containers. Some differences exist between sequence and associative container interfaces, which we examine after taking a look at the common components.

Common Members of All Containers

The public members of STL containers fall into a two level hierarchy. The first level defines members that are common to *all* containers, while the second level contains two categories:

- members common to sequence containers (vectors, lists, deques).
- members common to associative containers (sets, maps, multisets, multimaps).

The common members of all STL containers fall into two distinct categories: type definitions and member functions. We take a look at each in turn.

Common Type Definitions in All Containers

The common type definitions found in each STL container are presented below. It is assumed that `X` is a container class containing objects of type `T`, `a` and `b` are values of `X`, `u` is an identifier and `r` is a value of `X&`.

value_type

Description Type of values the container holds.

Definition `X::value_type`

reference

Description Type that can be used for storing into `X::value_type` objects. This type is usually `X::value_type&`.

Definition `X::reference`

const_reference

Description A constant reference type identical to `X::reference`.

Definition `X::const_reference`

pointer

Description A pointer type pointing to `X::reference`.

Definition `X::pointer`

iterator

Description An iterator type that points to `X` instances. It is either a random access iterator type (for vector or deque) or a bidirectional iterator type (for other containers).

Definition `X::iterator`

const_iterator

Description A iterator type that can be used with constant instances of type X. It is either a constant random access iterator type (for vector or deque) or a constant bidirectional iterator type (for other containers).

Definition `X::const_iterator`

reverse_iterator

Description An iterator type identical to `X::iterator` except that traversal direction is reversed (`X::reverse_iterator::operator++` is `X::iterator::operator--`, etc.).

Definition `X::reverse_iterator`

const_reverse_iterator

Description A constant iterator type identical to `X::const_iterator` except that traversal direction is reversed.

Definition `X::const_reverse_iterator`

difference_type

Description The type that can represent the difference between any two X iterator objects (varies with the memory model).

Definition `X::difference_type`

size_type

Description The type that can represent the size of any X instance (varies with the memory model).

Definition `X::size_type`

Common Member Functions in all Containers

The common member functions required to be in each STL container are outlined below. In the descriptions, it is assumed that *X* is a container class containing objects of type *T*, *a* and *b* are values of *X*, *u* is an identifier and *r* is a value of *X*&.

Default Constructor

Description The default constructor. Takes constant time.

Prototype `X()`

Overloaded and Copy Constructors

Prototype `X(a) ;`

Remarks Constructor. Takes linear time.

Prototype `X u(a) ;`

Remarks Copy Constructor. Takes linear time.

Destructor

Description Destructor. The destructor is applied to every element of *a*, and all the memory is returned. Takes linear time.

Prototype `(&a) -> ~X() ;`

begin

Description Returns an iterator (*const_iterator* for constant *a*), that can be used to begin traversing all locations in the container.

Prototype `a.begin() ;`

end

Description Returns an iterator (const_iterator for constant a), that can be used in a comparison for ending traversal through the container.

Prototype `a.end();`

rbegin

Description Returns a reverse_iterator (const_reverse_iterator for constant a) that can be used to begin traversing through all locations of the container in the reverse of the normal order.

Prototype `a.rbegin();`

rend

Description Returns a reverse_iterator (const_reverse_iterator for constant a) that can be used in a comparison for ending a reverse-direction traversal through all locations in the container.

Prototype `a.rend();`

Equality Operator ==

Description Equality operation on containers of the same type. Returns true when the sequences of elements in a and b are element wise equal (using `X::value_type::operator==`). Takes linear time.

Prototype `a == b`

Not Equal Operator !=

Description The opposite of the equality operation. Takes linear time.

Prototype `a != b`

Assignment Operator =

Description The assignment operator for containers. Takes linear time.

Containers Library

What Are Containers

Prototype `r = a`

size

Description Returns the number of elements in the container.

Prototype `a.size();`

max_size

Description `size()` of the largest possible container.

Prototype `a.max_size();`

empty

Description Returns true if the container is empty (i.e., if `a.size() == 0`).

Prototype `a.empty();`

Less Than Operator <

Description Compares two containers lexicographically. Takes linear time.

NOTE: lexicographical comparisons are described in chapter 10
Set Operations on Sorted Structures

Prototype `a < b`

Greater Than Operator >

Description Returns true if `b < a`, as defined above. Takes linear time.

Prototype `a > b`

Less Than or Equal Operator <=

Description Returns true if `!(a > b)`. Takes linear time.

Prototype `a <= b`

Greater Than or Equal`>=`

Description Returns true if `!(a < b)`. Takes linear time.

Prototype `a >= b`

swap

Description Swaps two containers of the same type in constant time.

Prototype `a.swap(b);`

Remarks From the above definitions, we note that several comparison functions, constructors, type definitions and other member functions are shared between all STL containers, allowing their interfaces to be very uniform.

NOTE: Shallow Copies: It must be noted that the assignment operators of all STL containers make shallow copies. This means that the assignment operator will simply copy the assigned container exactly as is, and will not traverse pointers downwards to make copies recursively of elements that the container elements might possibly point to.

Sequence Container Requirements

All STL Sequence containers define two constructors, three insert member functions and two erase member functions in addition to the common types and member functions mentioned in the previous section.

The additional members are defined below. In the definitions, `x` is assumed to be a sequence class (e.g., a vector, list or deque), `i` and `j` satisfy input iterator requirements, `[i, j)` is a valid range, `n` is a value of `x::size_type`, `p` is a valid iterator to `a`, `q`, `q1` and `q2`

Containers Library

What Are Containers

are valid dereferenceable iterators to a , $[q1, q2)$ is a valid range, t is a value of $X::value_type$.

Prototype `X(n, t);`
`X a(n, t);`

Remarks Constructs a sequence with n copies of t .

`X(i, j);`
`X a(i, j);`

Remarks Constructs a sequence equal to the range $[i, j)$.

insert

Description Inserts a copy or copies of an element.

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

Remarks Inserts a copy of t before p . Returns an iterator pointing to the inserted copy.

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

Remarks Inserts n copies of t before p .

Prototype `a.insert(p, i, j);`

Remarks Inserts copies of elements in $[i, j)$ before p .

erase

Description Erases the element or elements.

Prototype `a.erase(q);`

Erases the element pointed to by q .
`a.erase(q1, q2);`

Erases the elements in the range $[q1, q2)$.

NOTE: These additional member functions only define some basic insert, erase and construction operations. All other operations on the containers (such as sorting, searching, transformations, etc.) are carried out by `generic algorithms` provided by the library.

Associative Container Requirements

STL provides four basic kinds of associative containers: set, multiset, map and multimap.

Before taking a detailed look at the type definitions and member functions provided by the associative containers, we need to define a few terms and explain some of the ideas behind the design.

Basic Design and Organization

All associative containers are parameterized on a type `Key` and an ordering relation `Compare` that induces a total ordering on elements of `Key`. In addition, map and multimap associate an arbitrary type `T` with the `Key`. An object of type `Compare` is called the *comparison object* of the container.

Equality of Keys

For associative containers, it is important to note that equality of keys depends on the equivalence relation imposed by the comparison, and not on the operator `==` on keys. Thus, two keys `k1` and `k2` are considered equal if, for a comparison object `comp`,

```
comp(k1,k2)==false && comp(k2,k1)==false.
```

Additional Definitions

The set and map containers support unique keys; they can store at most one element of each key. multiset and multimap containers support equal keys; i.e., they can store multiple elements that have the same key.

Containers Library

What Are Containers

For set and multiset, the value type is the same as the key type; i.e., the values stored in sets and multisets are basically the keys themselves. For map and multimap, the value type is `pair<const Key, T>`; i.e., the elements stored in maps and multimaps are pairs whose first elements are `const Key` values and whose second elements are `T` values.

Finally, an iterator of an associative container is of the bidirectional iterator category. insert operations do not affect the validity of iterators and references to the container, and erase operations only invalidate iterators and references to the erased elements.

Listed below are the type definitions and member functions defined by associative containers in addition to the common container members outlined previously.

In all the definitions, we assume that `X` is an associative container class, `a` is a value of `X`, `a_uniq` is a value of `X` when `X` supports unique keys and `a_eq` is a value of `X` when `X` supports equal keys. `i` and `j` satisfy input iterator requirements and refer to elements of `value_type`. `[i, j)` is a valid range, `p` is a valid iterator to `a`, `q`, `q1`, and `q2` are valid dereferenceable iterators to `a`, `[q1, q2)` is a valid range, `t` is a value of `X::value_type` and `k` is a value of `X::key_type`.

Associative Container Types and Member Functions

key_type

Description The type of keys, `Key`, with which the container is instantiated.

Definition `X::key_type`

key_compare

Description The comparison object type, `Compare`, with which the container is instantiated.

Definition `X::key_compare`

value_compare

Description A type for comparing objects of `X::value_type`. This is the same as `key_compare` for set and multiset, while for map and multimap it is a type that compares pairs of Key and T values by comparing their keys using `X::key_compare`.

Definition `X::value_compare`

Constructors

Description Constructs a container object.

Prototype `X();`
`X a;`

Remarks Constructs an empty container, using `Compare()` as a comparison object. Takes constant time.

Prototype `X(c);`
`X a(c);`

Remarks Constructs an empty container, using `c` as a comparison object. Takes constant time.

`X(i,j,c);`
`X a(i,j,c);`

Remarks Constructs an empty container, using `c` as a comparison object, and inserts elements from the range `[i,j)` in it. Takes $N \log N$ time in general, where N is the distance from `i` to `j`; linear if `[i,j)` is sorted with `value_comp()`.

`X(i,j);`
`X a(i,j);`

Remarks Same as above, but uses `Compare()` as a comparison object.

key_comp

Description Returns the comparison object, of type `X::key_compare`, out of which `a` was constructed.

Containers Library

What Are Containers

Prototype `a.key_comp();`

value_comp

Description Returns an object of type `X::value_compare` constructed out of the comparison object.

Prototype `a.value_comp();`

insert

Description Inserts elements under various and specific conditions.

Prototype `a_uniq.insert(t);`

Remarks Inserts `t` if and only if there is no element in the container with key equal to the key of `t`. Returns a `pair<iterator, bool>` whose `bool` component indicates whether the insertion was made and whose `iterator` component points to the element with key equal to the key of `t`. Takes time logarithmic in the size of the container.
`a_eq.insert(p, t);`

Remarks Inserts `t` into the container and returns the iterator pointing to the newly inserted element.
`a.insert(p, t);`

Remarks Inserts `t` if and only if there is no element with key equal to the key of `t` in containers that support unique keys (i.e., sets and maps). Always inserts `t` in containers that support equal keys (i.e., multisets and multimaps). Iterator `p` is a hint pointing to where the insert should start to search. Takes time logarithmic in the size of the container in general, but amortized constant if `t` is inserted right after `p`.
`a.insert(i, j);`

Remarks Inserts the elements from the range `[i,j)` into the container. Takes $N\log N$ time in general, where N is the distance from `i` to `j`. Linear if `[i,j)` is sorted according to `value_comp()`.

erase

- Description** Erases and element under various specific conditions.
- Prototype** `a.erase(k);`
- Remarks** Erases all elements in the container with key equal to k. Returns the number of erased elements.
`a.erase(q);`
- Remarks** Erases the element pointed to by q.
`a.erase(q1, q2);`
- Remarks** Erases all the elements in the range [q1,q2). Takes $\log(\text{size}() + N)$ time, where N is the distance from q1 to q2.

find

- Description** Returns an iterator pointing to an element with key equal to k, or `a.end()` if such an element is not found.
- Prototype** `a.find(k);`

count

- Description** Returns the number of elements with key equal to k.
- Prototype** `a.count(k);`

lower_bound

- Description** Returns an iterator pointing to the first element with key not less than k.
- Prototype** `a.lower_bound(k);`

upper_bound

- Description** Returns an iterator pointing to the first element with key greater than k.

Containers Library

Organization of the Container Class Descriptions

Prototype `a.upper_bound(k);`

equal_range

Description Returns a pair of iterators (const iterators if `a` is constant), the first equal to `lower_bound(k)` and the second equal to `upper_bound(k)`.

Prototype `a.equal_range(k);`

NOTE: It must be noted that associative containers provide two constructors to copy ranges: `X(i,j,c)` and `X(i,j)`. The first version, `X(i,j,c)`, uses `c` as a comparison object, while the second constructor, `X(i,j)`, uses the comparison object `Compare()` constructed from the `Compare` type with which `X` is instantiated.

Organization of the Container Class Descriptions

Description The remaining sections of this chapter describe the specific requirements for the three sequence containers (vector, list, deque) and associative containers (set, multiset, map, multimap). Each of these container class descriptions contains the following subsections:

- **Files**—shows the header file to be included in programs that use the class.
- **Class Declaration**—the class name and template parameters are shown.
- **Description**—describes the basic functionality of the class. It serves as a short introduction to the particular container being described.
- **Type Definitions**—explains the type definitions in the public interface of the class.
- **Constructors, Destructors and Related Functions**—contains descriptions of constructors and destructors in the class. Some classes also have other related functions that deal with allocation and deallocation issues, and these are explained wherever required.

- **Element Access Member Functions**—explains the functionality of all member functions that are used to access elements in the container.
- **Insert Member Functions**—explains all member functions that are used to insert elements into the container.
- **Erase Member Functions**—details all member functions that are used to erase elements from the container.
- **Additional Notes Section(s)**—this section or sections contain details such as implementation dependencies, time complexity discussions for insert and erase member functions, memory model dependencies, etc. Any important information that is not included in the other sections is included in the notes sections.

Template class vector<T>

Files `#include <vector.h>`

Declaration `template
 < class T >
class vector;`

Description Vectors are containers that arrange elements of a given type in a strictly linear arrangement, and allow fast random access to any element (i.e., any element can be accessed in constant time).

Vectors allow constant time insertions and deletions at the end of the sequence. Inserting and/or deleting elements in the middle of a vector requires linear time. Further details of the time complexity of vector insertion can be found in the notes section.

The topics in this section are:

- [“Type Definitions vector” on page 204](#)
- [“Constructors, Destructors and Related Functions vector” on page 206](#)
- [“Comparison Operations vector” on page 208](#)
- [“Element Access Member Functions vector” on page 208](#)

- [“Erase Member Functions vector” on page 211](#)
- [“Insert Member Functions vector” on page 210](#)
- [“Notes on Insert and Erase Member Functions vector” on page 212](#)
- [“Specialization Class vector<bool>” on page 212](#)

Type Definitions vector

Iterator

Description iterator is a random access iterator type referring to T.

Definition `typedef iterator;`

const_iterator

Description const_iterator is a constant random access iterator type referring to const T.

Definition `typedef const_iterator;`

Remarks It is guaranteed that there is a constructor for const_iterator out of iterator.

T*

Description The type T* (pointer to T).

Definition `typedef Allocator<T>::pointer pointer;`

reference

Description The type T& that can be used for storing into T objects.

Definition `typedef Allocator<T>::reference reference;`

const_reference

Description `typedef Allocator<T>::const_reference`

`const_reference;`

Definition The type `const T&` that can be used for storing into T objects.

`size_type`

Description `size_type` is an unsigned integral type that can represent the size of any vector instance.

Definition `typedef size_type;`

`difference_type`

Description A signed integral type that can represent the difference between any two pointers to `vector::iterator` objects.

Definition `typedef difference_type;`

`value_type`

Description The type of values the vector holds. This is simply T.

Definition `typedef T value_type;`

`reverse_iterator`

Description Non-constant reverse random access iterator.

Definition `typedef reverse_iterator;`

`const_reverse_iterator`

Description Constant reverse random access iterator.

Definition `typedef const_reverse_iterator;`

Constructors, Destructors and Related Functions `vector`

Default Constructor

Description The default constructor. Constructs a vector of size zero.

Prototype `vector();`

Overloaded and Copy Constructors

Prototype `explicit vector
 (size_type n,
 const T& value = T());`

Remarks Constructs a vector of size `n` and initializes all its elements with `value`. If the second argument is not supplied, `value` is obtained with the default constructor, `T()`, for the element value type `T`.

Prototype `vector(const vector<T>& x);`

Remarks The vector copy constructor. Constructs a vector and initializes it with copies of the elements of vector `x`.

`vector
 (const_iterator first,
 const_iterator last);`

Remarks Constructs a vector of size `last-first` and initializes it with copies of elements in the range `[first,last)`.

Assignment Operator `=`

Description The vector assignment operator. Replaces the contents of the current vector with a copy of the parameter vector `x`.

Prototype `vector<T>& operator=
 (const vector<T>& x);`

Reserve

Description This member function is a directive that informs the vector of a planned change in size, so storage can be managed accordingly. It does not change the size of the vector, and it takes time at most linear in the size of the vector. Reallocation happens at this point if and only if the current capacity is less than the argument of reserve (capacity is a vector member function that returns the size of the allocated storage in the vector). After a call to reserve, the capacity is greater than or equal to the argument of reserve if reallocation happens, and equal to the previous capacity otherwise.

Prototype `void reserve(size_type n);`

Remarks Reallocation invalidates all the references, pointers, and iterators referring to the elements in the vector. It is guaranteed that no reallocation takes place during the insertions that happen after reserve takes place till the time when the size of the vector reaches the size specified by reserve.

Destructor

Description The vector destructor. Returns all allocated storage back to the free store.

Prototype `~vector();`

vector::swap

Description Swaps the contents of the current vector with those of the input vector `x`. The current vector replaces `x` and vice versa.

Prototype `void swap(vector<T>& x);`

Comparison Operations `vector`

Equality Operator `==`

Description Equality operation on vectors. Returns true if the sequences of elements in `x` and `y` are element-wise equal (using `T::operator==`). Takes linear time.

Prototype

```
bool operator==(
    (const vector<T>& x,
    const vector<T>& y);
```

Less Than Operator `<`

Description Returns true if `x` is lexicographically less than `y`, false otherwise. Takes linear time.

Prototype

```
bool operator<
    (const vector<T>& x,
    const vector<T>& y);
```

Element Access Member Functions `vector`

`vector::begin`

Description Returns an iterator (`const_iterator` for constant vector) that can be used to begin traversing through the vector.

Prototype

```
iterator begin();
const_iterator begin() const;
```

`vector::end`

Description Returns an iterator (`const_iterator` for constant vector) that can be used in a comparison for ending traversal through the vector.

Prototype

```
iterator end();
const_iterator end() const;
```


vector::rbegin

Description Returns a reverse_iterator (const_reverse_iterator for constant vectors) that can be used to begin traversing the vector in the reverse of the normal order.

Prototype `reverse_iterator rbegin();
const_reverse_iterator rbegin();`

vector::rend

Description Returns a reverse_iterator (const_reverse_iterator for constant vectors) that can be used in a comparison for ending reverse-direction traversal through the vector.

Prototype `reverse_iterator rend();
const_reverse_iterator rend();`

vector::size

Description Returns the number of elements currently stored in the vector.

Prototype `size_type size() const;`

vector::max_size

Description Returns the maximum possible size of the vector.

Prototype `size_type max_size() const;`

vector::capacity

Description Returns the largest number of elements that the vector can store without reallocation. See also the reserve member function.

Prototype `size_type capacity() const;`

Containers Library

Template class `vector<T>`

`vector::empty`

Description Returns true if the vector contains no elements (i.e., if `begin() == end()`), false otherwise.

Prototype `bool empty() const;`

Subset Operator []

Description Returns the *n*th element from the beginning of the vector in constant time.

Prototype `reference operator[](size_type n);`
`const reference operator[](size_type n) const;`

`vector::front`

Description Returns the first element of the vector; i.e., the element referred to by the iterator `begin()`. Undefined if the vector is empty.

Prototype `reference front();`
`const_reference front() const;`

`vector::back`

Description Returns the last element of the vector; i.e., the element pointed to by the iterator `end()-1`. Undefined if the vector is empty.

Prototype `reference back();`
`const_reference back() const;`

Insert Member Functions `vector`

The time complexities of all insert member functions are described in the notes subsections at the end of this section.

`vector::push_back`

Description Adds the element *x* at the end of the vector.

Prototype `void push_back(const T& x);`

vector::insert

Description Inserts an element or elements into position or positions referred to in the vector object.

Prototype `iterator insert(iterator position, const T& x);`

Remarks Inserts the element `x` at the position in the vector referred to by the iterator `position`. Elements already in the vector are moved as required. The iterator returned refers to the position at which the element was inserted.

Prototype `void insert
 (iterator position,
 size_type n,
 const T& x = T());`

Remarks Inserts `n` copies of the element `x` starting at the position referred to by the iterator `position`.

Prototype `void insert
 (iterator position,
 const T* first,
 const T* last);`

Remarks Copies of elements in the range `[first,last)` are inserted into the vector at the position referred to by the iterator `position`.

Erase Member Functions vector

vector::pop_back

Description Erases the last element of the vector.

Prototype
`void pop_back();`

`vector::erase`

Description	Erases one or more elements from a vector.
Prototype	<code>void erase(iterator position);</code>
Remarks	Erases the element of the vector pointed to by the iterator position.
Prototype	<code>void erase(iterator first, iterator last);</code>
Remarks	The iterators first and last are assumed to point into the vector, and all elements in the range [first,last) are erased from the vector.

Notes on Insert and Erase Member Functions

vector

Inserting a single element into a vector is linear in the distance from the insertion point to the end of the vector. The amortized complexity of inserting a single element at the end of a vector is constant (see Section 1.4.2 for discussion of amortized complexity).

Insertion of multiple elements into a vector with a single call of the insert member function is linear in the sum of the number of elements plus the distance to the end of the vector. This means that it is much faster to insert many elements into the middle of a vector at once than to do the insertions one at a time.

All insert member functions cause reallocation if the new size is greater than the old capacity. If no reallocation happens, all the iterators and references before the insertion point remain valid.

`erase` invalidates all iterators and references after the point of the erase. The destructor of `T` is called for each erased element and the assignment operator of `T` is called the number of times equal to the number of elements in the vector after the erased elements.

Specialization Class `vector<bool>`

Files `#include <vector.h>`

Description This class is a specialization of the template class `vector<T>` so as to optimize space allocation. All the member functions of template class `vector<T>` are defined for this class also and it has an extra member function `swap()`.

In this implementation `vector<bool>` has been replaced by a class `bit_vector` since `bool` has not been implemented.

Public Member Functions

`vector::swap`

Description In this implementation, this function is a global function which swaps the contents of `x` and `y`.

Prototype `void swap(reference x, reference y);`

Template class deque<T>

Files `#include <deque.h>`

Declaration `template <class T> class deque;`

Description This class implements a deque of objects of type `T`. Deques are very much like vectors, except that they can be expanded in both directions: they allow constant time insertion and deletion of objects at either end.

Like vectors, deques allow fast random access to any element in constant time.

The topics in this section are:

- [“Typedef Declarations deque” on page 214](#)
- [“Constructors, Destructors and Related Functions deque” on page 215](#)
- [“Comparison Operations deque” on page 217](#)
- [“Element Access Member Functions deque” on page 217](#)

- [“Insert Member Functions deque” on page 219](#)
- [“Erase Member Functions deque” on page 220](#)
- [“Deque Class Notes” on page 221](#)

Typedef Declarations deque

Description The following typedef’s are defined in the class `deque<T>`.

iterator

Definition `typedef iterator;`

const_iterator

Definition `typedef const_iterator;`

`iterator` is a random access iterator referring to `T`. `const_iterator` is a constant random access iterator referring to `const T`.

pointer

Description The type `T*` (pointer to `T`).

Definition `typedef Allocator<T>::pointer pointer;`

reference

Description The type `T&` that can be used for storing into `T` objects.

Definition `typedef Allocator<T>::reference reference;`

const_reference

Description The type `const T&` for const references that can be used for storing into `T` objects.

Definition `typedef Allocator<T>::const_reference
const_reference;`

size_type

Description size_type is an unsigned integral type that can represent the size of any deque instance.

Definition `typedef size_type;`

difference_type

Description A signed integral type that can represent the difference between any two deque::iterator objects.

Definition `typedef difference_type;`

value_type

Description The type of values the deque holds. This is simply T.

Definition `typedef T value_type;`

reverse_iterator

Description Non-constant reverse random access iterators

Definition `typedef reverse_iterator;`

const_reverse_iterator

Description Constant reverse random access iterators

Definition `typedef const_reverse_iterator;`

Constructors, Destructors and Related Functions deque

Default Constructor

Description The default constructor. Constructs a deque with size zero.

Containers Library

Template class *deque*<T>

Prototype `deque() ;`

Overloaded and Copy Constructors

Prototype `explicit deque
 (size_type n,
 const T& value = T());`

Remarks Constructs a deque of size *n*, and initializes all its elements with *value*. The default for *value* is set to *T()*, where *T()* is the default constructor of the type passed to the deque class template.

Prototype `deque(const deque<T>& x);`

Remarks The deque copy constructor. Constructs a deque and initializes it with copies of the elements of deque *x*.

Prototype `deque
 (const_iterator first,
 const_iterator last);`

Remarks Constructs a deque of size *last-first* and initializes it with copies of elements in the range [*first*,*last*).

Assignment Operator =

Prototype `deque<T>& operator=
 (const deque<T>& x);`

Remarks The deque assignment operator. Replaces the contents of the current deque with a copy of the parameter deque *x*.

Destructor

Description The set destructor. Returns all allocated storage back to the free store.

Prototype `~deque() ;`

deque::swap

Description Swaps the contents of the current deque with those of the input deque x. The current deque replaces x and vice versa.

Prototype `void swap(deque<T>& x);`

Comparison Operations deque

Equality Operator ==

Description Equality operation on deques. Returns true if the sequences of elements in x and y are element-wise equal (using `T::operator==`). Takes linear time.

Prototype `bool operator==
 (const deque<T>& x,
 const deque<T>& y);`

Less Than Operator <

Description Returns true if x is *lexicographically* less than y, false otherwise. Takes linear time.

Prototype `bool operator<
 (const deque<T>& x,
 const deque<T>& y);`

Element Access Member Functions deque

deque::begin

Description Returns an iterator (`const_iterator` for constant deque) that can be used to begin traversing through all locations in the deque.

Prototype `iterator begin()
const_iterator begin() const;`

Containers Library

Template class *deque*<T>

deque::end

Description Returns an iterator (`const_iterator` for constant deque) that can be used in a comparison for ending traversal through the deque.

Prototype `iterator end();`
`const_iterator end() const;`

deque::rbegin

Description Returns a `reverse_iterator` (`const_reverse_iterator` for constant deques), that can be used to begin traversing all locations in the deque in the reverse of the normal order.

Prototype `reverse_iterator rbegin();`
`const_reverse_iterator rbegin() const;`

deque::rend

Description Returns a `reverse_iterator` (`const_reverse_iterator` for constant deques), that can be used in a comparison for ending reverse-direction traversal through all locations in the deque.

Prototype `reverse_iterator rend();`
`const_reverse_iterator rend();`

deque::size

Description Returns the number of elements in the deque.

Prototype `size_type size() const;`

deque::max_size

Description Returns the maximum possible size of the deque.

Prototype `size_type max_size() const;`

deque::empty

Description Returns true if the deque contains no elements (i.e., if `begin() == end()`), false otherwise

Prototype `bool empty() const;`

Subset Operator []

Description Allows constant time access to the *n*th element of the deque.

Prototype `reference operator[](size_type n);`
`const_reference operator[](size_type n) const;`

deque::front

Description Returns the first element of the deque; i.e., the element pointed to by the iterator `begin()`.

Prototype `reference front();`
`const_reference front() const;`

deque::back

Description Returns the last element of the deque; i.e., the element pointed to by the iterator `end()-1`.

Prototype `reference back();`
`const_reference back() const;`

Insert Member Functions deque

deque::push_front

Description Adds the element *x* at the beginning of the deque.

Prototype `void push_front(const T& x);`

Containers Library

Template class *deque*<T>

deque::push_back

Description Adds the element *x* at the end of the deque.

Prototype `void push_back(const T& x);`

deque::insert

Description Inserts one or more elements into a position in the deque.

Prototype `iterator insert(iterator position, const T& x);`

Remarks Inserts the element *x* at the position in the deque pointed to by the iterator *position*. The iterator returned points to the position that contains the inserted element.

Prototype `void insert
(iterator position,
size_type n,
const T& x = T());`

Description Inserts *n* copies of the element *x* starting at the position pointed to by the iterator *position*.

Prototype `void insert
(iterator position,
const T* first,
const T* last;`

Description Inserts elements into the deque before the position pointed to by the iterator *position*. Copies of elements in the range *[first,last)* are inserted into the deque.

Erase Member Functions deque

deque::pop_front

Description

Prototype

```
void pop_front();
```

deque::pop_back

Description Erases the last element of the deque.

Prototype `void pop_back();`

deque::erase

Description Erases one or more elements of the deque.

Prototype `void erase(iterator position);`

Remarks Erases the element of the deque pointed to by the iterator position.
`void erase(iterator first, iterator last);`

Remarks The iterators first and last are assumed to point into the deque, and all elements in the range [first,last) are erased from the deque.

Deque Class Notes

Storage Management

As with all STL containers, all storage management in deques is handled automatically. Deques are implemented using segmented storage (unlike vectors). This means that all deque elements are not necessarily kept in contiguous locations in memory.

Complexity of Insertion

Deques are specially optimized for insertion of single elements at either the beginning or the end of the data structure. Such insertions always take constant time and cause a single call to the copy constructor of T, where T is the type of the inserted object.

If an element is inserted into the middle of the deque, then in the worst case the time taken is linear in the minimum of the distance

from the insertion point to the beginning of the deque and the distance from the insertion point to the end of the deque.

The insert and push member functions invalidate all the iterators and references to the deque.

Notes on Erase Member Functions

The erase and pop invalidate all the iterators and references to the deque. The number of calls to the destructor (of the erased type `T`) is the same as the number of elements erased, but the number of calls to the assignment operator of `T` is equal to the minimum of the number of elements before the erased elements and the number of elements after the erased elements.

Template class `list<T>`

Files `#include <list.h>`

Declaration `template
 < class T >
 class list;`

Description This class implements the sequence abstraction as a linked list. All lists are “doubly-linked” and may be traversed in either direction.

Lists should be used in preference to other sequence abstractions when there are frequent insertions and deletions in the middle of sequences. As with all STL containers, storage management is handled automatically.

Unlike vectors or deques, lists are not random-access data structures. For this reason, some STL generic algorithms such as `sort`, `random_shuffle`, etc., cannot operate on lists. The list class provides its own `sort` member function.

Besides `sort`, lists also include some other special member functions for splicing two lists, reversing lists, making all list elements unique

and for merging two lists. All of these special member functions are discussed on [“Special Operations list” on page 230](#).

The topics in this section are:

- [“Typedef Declarations list” on page 223](#)
- [“Constructors, Destructors and Related Functions list” on page 225](#)
- [“Comparison Operations list” on page 226](#)
- [“Element Access Member Functions list” on page 227](#)
- [“Insert Member Functions list” on page 228](#)
- [“Erase Member Functions list” on page 229](#)
- [“Special Operations list” on page 230](#)
- [“List Class Notes” on page 232](#)

Typedef Declarations list

Description The following typedef’s are defined in the class `list<T>`.

iterator

Description The type `iterator` is a bidirectional iterator referring to `T`.

Definition `typedef iterator;`

const_iterator

Description The type `const_iterator` is a constant bidirectional iterator referring to `const T`. It is guaranteed that there is a constructor for `const_iterator` out of `iterator`.

Definition `typedef const_iterator;`

pointer

Description The type `T*` (pointer to `T`).

Definition `typedef Allocator<T>::pointer pointer;`

Containers Library

Template class `list<T>`

reference

Description The type `T&` that can be used for storing into `T` objects.

Definition `typedef Allocator<T>::reference reference;`

const_reference

Description The type `const T&` for const references that can be used for storing into `T` objects.

Definition `typedef Allocator<T>::const_reference
const_reference;`

size_type

Description `size_type` is an unsigned integral type that can represent the size of any `list` instance.

Definition `typedef size_type;`

difference_type

Description A signed integral type that can represent the difference between any two `list::iterator` objects.

Definition `typedef difference_type;`

value_type

Description The type `T` of values the list holds.

Definition `typedef T value_type;`

reverse_iterator

Description Non-constant reverse bidirectional iterator type.

Definition `typedef reverse_iterator;`

`const_reverse_iterator`

Description Constant reverse bidirectional iterator type.

Definition `typedef const_reverse_iterator;`

Constructors, Destructors and Related Functions

Default Constructor

Description The default constructor. Constructs an empty list.

Prototype `list();`

Overloaded and Copy Constructors

Description Constructs a list of size `n`, and initializes all its elements with value.

Prototype `explicit list
 (size_type n,
 const T& value = T());`

Prototype `list(const list<T>& x);`

Remarks The list copy constructor. Constructs a list and initializes it with copies of the elements of list `x`.

Prototype `list(const T* first, const T* last);`

Remarks Constructs a list of size `last-first` and initializes it with copies of elements in the range `[first,last)`.

Assignment Operator =

Description The list assignment operator. Replaces the contents of the current list with a copy of the parameter list `x`.

Prototype `list<T>& operator=(const list<T>& x);`

Containers Library

Template class `list<T>`

Destructor

Description The list destructor. Returns all allocated storage back to the free store.

Prototype `~list();`

`list::swap`

Description Swaps the contents of the current list with those of the input list `x`. The current list replaces `x` and vice versa.

Prototype `void swap(list<T>& x);`

Comparison Operations list

Equality Operator `==`

Description Equality operation on lists. Returns true if the sequences of elements in `x` and `y` are element-wise equal (using `T::operator==`). Takes linear time.

Prototype `bool operator==
 (const list<T>& x,
 const list<T>& y);`

Less Than Operator `<`

Description Returns true if `x` is lexicographically less than `y`, false otherwise. Takes linear time

Prototype `bool operator<
 (const list<T>& x,
 const list<T>& y);`

Element Access Member Functions list

list::begin

Description Returns an iterator (`const_iterator` for constant list) that can be used to begin traversing through the list.

Prototype `iterator begin();`
`const_iterator begin() const;`

list::end

Description Returns an iterator (`const_iterator` for constant list) that can be used in a comparison for ending traversal through the list.

Prototype `iterator end();`
`const_iterator end() const;`

list::rbegin

Description Returns a `reverse_iterator` (`const_reverse_iterator` for constant lists), that can be used to begin traversing the list in the reverse of the normal order.

Prototype `reverse_iterator rbegin();`
`const_reverse_iterator rbegin() const;`

list::rend

Description Returns a `reverse_iterator` (`const_reverse_iterator` for constant lists), that can be used in a comparison for ending reverse-direction traversal through the list.

Prototype `reverse_iterator rend();`
`const_reverse_iterator rend() const;`

list::size

Description Returns the number of elements currently stored in the list.

Containers Library

Template class *list*<T>

Prototype `size_type size() const;`

list::max_size

Description Returns the maximum possible size of the list.

Prototype `size_type max_size() const;`

list::empty

Description Returns true if the list contains no elements (i.e., if `begin() == end()`), false otherwise

Prototype `bool empty() const;`

list::front

Description Returns the first element of the list; i.e., the element pointed to by the iterator `begin()` .

Prototype `reference front();`
`const_reference front() const;`

list::back

Description Returns the last element of the list; i.e., the element pointed to by the iterator `end()-1` .

Prototype `reference back();`
`const_reference back() const;`

Insert Member Functions list

list::push

Description Inserts the element `x` at the beginning of the list .

Prototype `void push_front(const T& x);`

list::push_back

Description Inserts the element x at the end of the list.

Prototype `void push_back(const T& x);`

list::insert

Description Inserts one or more elements into the list.

Prototype `iterator insert(iterator position, const T& x);`

Remarks Inserts the element x at the position in the list pointed to by the iterator position. The iterator returned points to the position that contains the inserted element.

Prototype `void insert
 (iterator position,
 size_type n,
 const T& x = T());`

Remarks Inserts n copies of the element x starting at the position pointed to by the iterator position

`void insert
 (iterator position,
 const T* first,
 const T* last);`

Remarks Inserts elements into the list before the position pointed to by the iterator position. Copies of elements in the range [first,last) are inserted into the list.

Erase Member Functions list

list::pop_front

Description Erases the first element of the list.

Prototype `void pop_front();`

Containers Library

Template class `list<T>`

`list::pop_back`

Description Erases the last element of the list.

Prototype `void pop_back();`

`list::erase`

Description Erases one or more elements of the list pointed to by the iterator position.

Prototype `void erase(iterator position);`

Remarks Erases the element of the list pointed to by the iterator position.

Prototype `void erase(iterator first, iterator last);`

Remarks The iterators `first` and `last` are assumed to point into the list, and all elements in the range `[first, last)` are erased from the list.

Special Operations list

`list::splice`

Description These member functions inserts the contents of list `x` before the iterator position, and `x` becomes empty.

Prototype `void splice(iterator position, list<T>& x);`

Remarks This member function inserts the contents of list `x` before the iterator position, and `x` becomes empty. The operation takes constant time. Essentially, the contents of `x` are transferred into the current list.

Prototype `void splice
 (iterator position,
 list<T>& x,
 iterator x_elem);`

Remarks Inserts the element pointed to by iterator `x_elem` from list `x` before position, and removes the element from `x`. It takes constant time. `x_elem` is assumed to be a valid iterator of the list `x`. The function basically transfers a single element from the list `x` into the current list.

Prototype

```
void splice
    (iterator position,
     list<T>& x,
     iterator first,
     iterator last);
```

Remarks Inserts the elements in the range `[first, last)` before the iterator position, and removes the elements from list `x`. The operation takes linear time. The range `[first,last)` is assumed to be a valid range in `x`.

`list::remove`

Description This function removes all elements in the list that are equal to `value`, using `T::operator==`. The relative order of other elements is not affected. The entire list is traversed exactly once.

Prototype

```
void remove(const T& value);
```

`list::unique`

Description This function erases all but the first element from every consecutive group of equal elements in the list. Exactly `size()-1` applications of `T::operator==` are done. This function is most useful when the list is sorted, so that all elements that are equal appear in consecutive positions. In that case, each element in the resulting list is unique.

Prototype

```
void unique();
```

`list::merge`

Description This function merges the argument list `x` into the current list. It is assumed that both lists are sorted according to the `operator<` of type `T`. The merge is stable; i.e., for equal elements in the two lists, the elements from the current list always precede the elements from the

Containers Library

Template class `list<T>`

argument list `x`. `x` becomes empty after the merge. At most `size() + x.size() - 1` comparisons are done.

Prototype `void merge(list<T>& x);`

`list::reverse`

Description Reverses the order of the elements in the list. It takes linear time.

Prototype `void reverse();`

`list::sort`

Description Sorts the list according to the operator< of type `T`. The sort is stable; i.e., the relative order of equal elements is preserved. Approximately $N\log N$ comparisons are done, where N is equal to `size()`.

Prototype `void sort();`

List Class Notes

Notes on Insert Member Functions

List insert operations do not affect the validity of iterators and references to other elements of the list. Insertion of a single element of type `T` into a list takes constant time and makes only one call to the copy constructor of `T`. Insertion of multiple elements into a list is linear in the number of elements inserted, and the number of calls to the copy constructor of `T` is exactly equal to the number of elements inserted.

Notes on Erase Member Functions

`erase` invalidates only the iterators and references to the erased elements. Erasing a single element of type `T` is a constant time operation, with a single call to the destructor of `T`. Erasing a range in a list takes time linear in the size of the range, and the number of calls to the destructor of type `T` is exactly equal to the size of the range.

Template class `set<T>`

Files `#include <set.h>`

Declaration

```
template
    < class Key,
      class Compare = less<Key> >
class set;
```

Description A `set<Key, Compare>` stores unique elements of type `Key`, and allows for the retrieval of the elements themselves. All elements in the set are ordered by the ordering relation `Compare`, which induces a total ordering on the elements.

As with all STL containers, the set container only allocates storage and provides a minimal set of operations (such as insert, erase, find, count, etc.). The set does not itself provide operations for union, intersection, difference etc. These operations are handled by generic algorithms in STL.

The topics in this section are:

- [“Typedef Declarations set” on page 233](#)
- [“Constructors, Destructors and Related Functions set” on page 236](#)
- [“Comparison Operations set” on page 237](#)
- [“Element Access Member Functions set” on page 238](#)
- [“Insert Member Functions set” on page 239](#)
- [“Erase Member Functions set” on page 240](#)
- [“Special Operations set” on page 241](#)

Typedef Declarations `set`

Description The following typedef’s are defined in the class `set`.

key_type

Description The type of the keys with which the set is instantiated.

Containers Library

Template class `set<T>`

Definition `typedef Key key_type;`

value_type

Description `value_type` represents the type of the values stored in the set. This is the same as `key_type`.

Definition `typedef Key value_type;`

`value_type` represents the type of the values stored in the set. This is the same as `key_type`.

pointer

Description The type `Key*` (pointer to `Key`).

Definition `typedef Allocator<Key>::pointer pointer;`

The type `Key*` (pointer to `Key`).

reference

Description The type `Key&` that can be used for storing into `Key` objects.

Definition `typedef Allocator<Key>::reference reference;`

const_reference

Description The type `Const Key&` for const references that can be used for storing into `Key` objects.

Definition `typedef Allocator<Key>::const_reference
const_reference;`

The type `Key&` (`const Key&` for const references) that can be used for storing into `Key` objects.

compare_key

Description The comparison object type, `Compare`, with which the set is instantiated. This type is used to order the keys in the set.

Definition `typedef Compare key_compare;`

value_compare

Description This is the ordering relation that is used to order the values stored in the set. Its type is the same as `key_compare`, since the type of a value stored in a set is the same as the type of the key.

Definition `typedef Compare value_compare;`

iterator

Description The type `iterator` is a constant bidirectional iterator referring to `const value_type`.

Definition `typedef iterator;`

const_iterator

Description The type `const_iterator` is the same type as `iterator`.

Definition `typedef const_iterator;`

size_type

Description `size_type` is an unsigned integral type that can represent the size of any set instance.

Definition `typedef size_type;`

difference_type

Description A signed integral type that can represent the difference between any two `set::iterator` objects.

Definition `typedef difference_type;`

reverse_iterator

Description Non-constant reverse bidirectional iterator type.

Containers Library

Template class `set<T>`

Definition `typedef reverse_iterator;`

`const_reverse_iterator`

Description Constant reverse bidirectional iterator type.

Definition `typedef const_reverse_iterator;`

Constructors, Destructors and Related Functions `set`

Default Constructor

Description The default constructor. Constructs an empty set using the relation `comp` to order the elements.

Prototype `set(const Compare& comp = Compare());`

Overloaded and Copy Constructors

Prototype `set(const set<Key, Compare>& x);`

Remarks The set copy constructor. Constructs a set and initializes it with copies of the elements of set `x`.

Prototype `set
 (const value_type* first,
 const value_type* last,
 const Compare& comp = Compare());`

Remarks Constructs an empty set and initializes it with copies of elements in the range `[first,last)`. The ordering relation `comp` is used to order the elements of the set.

Assignment Operator `=`

Description The set assignment operator. Replaces the contents of the current set with a copy of the parameter set `x`.

Prototype `set<Key, Compare>& operator=
 (const set<Key,
 Compare>& x);`

set::swap

Description Swaps the contents of the current set with those of the input set x. The current set replaces x and vice versa.

Prototype `void swap(set<Key, Compare>& x);`

Destructor

Description The set destructor. Returns all allocated storage back to the free store.

Prototype `~set();`

Comparison Operations set

Equality Operator ==

Description Equality operation on sets. Returns true if the sequences of elements in x and y are element-wise equal (using `T::operator==`). Takes linear time.

Prototype `bool operator==
 (const set<Key,
 Compare>& x,
 const set<Key,
 Compare>& y);`

Less Than Operator <

Description Returns true if x is lexicographically less than y, false otherwise. Takes linear time.

Prototype `bool operator<
 (const set<Key,
 Compare>& x,`

```
const set<Key,  
Compare>& y);
```

Element Access Member Functions `set`

`set::key_compare`

Description This function returns the comparison object of the set. The comparison object is an object of class `Compare`, which represents the ordering relation used to construct the set.

Prototype `key_compare key_comp() const;`

`set::value_comp`

Description Returns an object of type `value_compare` constructed out of the comparison object. For sets, this is simply an object of type `Compare`.

Prototype `value_compare value_comp() const;`

`set::begin`

Description Returns the iterator that can be used to begin traversing through all locations in the set.

Prototype `iterator begin() const;`

`set::end`

Description Returns an iterator that can be used in a comparison for ending traversal through the set.

Prototype `iterator end() const;`

`set::rbegin`

Description Returns a `reverse_iterator`, that can be used to begin traversing all locations in the set in the reverse of the normal order.

Prototype `reverse_iterator rbegin();`

set::rend

Description Returns a `reverse_iterator`, that can be used in a comparison for ending reverse-direction traversal through all locations in the set.

Prototype `reverse_iterator rend();`

set::empty

Description Returns true if the set is empty, false otherwise.

Prototype `bool empty() const;`

set::size

Description Returns the number of elements in the set.

Prototype `size_type size() const;`

set::max_size

Description Returns the maximum possible size of the set. The maximum size is simply the total number of elements of type `Key` that can be represented in the memory model used.

Prototype `size_type max_size() const;`

Insert Member Functions set

set::insert

Description Inserts one or more elements into the set.

Prototype `iterator insert
 (iterator position,
 const value_type& x);`

Containers Library

Template class `set<T>`

Remarks Inserts the element `x` into the set if `x` is not already present in the set. The iterator position is a hint, indicating where the insert function should start to search to do the insert. The search is necessary since sets are ordered containers.

The insertion takes $O(\log N)$ time, where N is the number of elements in the set, but is amortized constant if `x` is inserted right after the iterator position.

Prototype `pair<iterator, bool> insert
(const value_type& x);`

Remarks Inserts the element `x` into the set if `x` is not already present in the set. The returned value is a pair, whose bool component indicates whether the insertion has taken place, and whose iterator component points to the just inserted element in the set, if the insertion takes place, otherwise to the element `x` already present.

The insertion takes $O(\log N)$ time, where N is the number of elements in the set.

Prototype `void insert
(const value_type* first,
const value_type* last);`

Remarks Copies of elements in the range `[first,last)` are inserted into the set. This insert member function allows elements from other containers to be inserted into the set.

In general, the time taken for this insertion is $N \log(\text{size}() + N)$, where N is the distance from `first` to `last`, and linear if the range `[first, last)` is sorted according to the set ordering relation `value_comp()`.

Erase Member Functions `set`

`set::erase`

Description Erases one or more set elements.

Prototype `void erase(iterator position);`

Remarks Erases the set element pointed to by the iterator position. The time taken is amortized constant.

Prototype `size_type erase(const key_type& x);`

Remarks Erases all the set elements with key equal to *x* (i.e., removes all *x*'s from the set). Returns the number of erased elements, which is 1 if *x* is present in the set, and 0 otherwise (since sets do not store duplicate elements). In general, this function takes time proportional to $\log(\text{size}()) + \text{count}(x)$, where $\text{count}(k)$ is a set member function that returns the number of elements with key equal to *k*.

Prototype `void erase(iterator first, iterator last);`

Remarks The iterators *first* and *last* are assumed to point into the set, and all elements in the range $[\text{first}, \text{last})$ are erased from the set. The time taken is $\log(\text{size}()) + N$, where *N* is the distance from *first* to *last*.

Special Operations set

set::find

Description Searches for the element *x* in the set. If *x* is found, the function returns the iterator pointing to it. Otherwise, `end()` is returned. The function takes $O(\log N)$ time, where *N* is the number of elements in the set.

Prototype `iterator find(const key_type& x) const;`

set::count

Description Returns the number of elements in the set that are equal to *x*. If *x* is present in the set, this number is always 1; otherwise, it is 0. The function takes $O(\log(\text{size}()))$ time.

Prototype `size_type count(const key_type& x) const;`

Containers Library

Template class *multiset*<Key>

set::lower_bound

Description Returns an iterator pointing to the first set element whose key is not less than x. Since set elements are not repeated, the returned iterator points to x itself if x is present in the set. If x is not present in the set, end() is returned. The function takes O(log N) time, where N is the number of elements in the set.

Prototype `iterator lower_bound(const key_type& x) const;`

set::upper_bound

Description The upper_bound function returns an iterator to the first set element whose key is greater than x. If no such element is found, end() is returned. The function takes O(log N) time, where N is the number of elements in the set.

Prototype `iterator upper_bound(const key_type& x) const;`

set::equal_range

Description This function returns the pair(lower_bound(x), upper_bound(x)). The function takes O(log N) time, where N is the number of elements in the set.

Prototype `pair<iterator,iterator> equal_range
(const key_type& x) const;`

Template class **multiset**<Key>

Files `#include <multiset.h>`

Declaration `template
 < class Key,
 class Compare = less<Key> >
class multiset;`

Description A multiset is an associative container that can store multiple copies of the same key. As with regular sets, all elements in the multiset are

ordered by the ordering relation `Compare`, which induces a total ordering on the elements.

Multisets are necessary because we sometimes need to store elements, all of which are alike in most ways but which differ only in certain known characteristics. For example, a set of cars sorted by the make of the car would be a multiset, since there could be several cars in the set with the same manufacturer, but different in other aspects, such as engine capacity, price, etc.

The interface of the multiset class is exactly the same as that of the regular set class. The only difference is that multisets possibly contain multiple values of the same key value. As a result, some of the member functions also have slightly different semantics.

The topics in this section are:

- [“`Typedef Declarations multiset`” on page 243](#)
- [“`Constructors, Destructors and Related Functions multiset`” on page 246](#)
- [“`Comparison Operations multiset`” on page 247](#)
- [“`Element Access Member Functions multiset`” on page 248](#)
- [“`Insert Member Functions multiset`” on page 249](#)
- [“`Erase Member Functions multiset`” on page 250](#)
- [“`Special Operations multiset`” on page 251](#)

Typedef Declarations multiset

Description The following typedef’s are defined in the class `multiset`.

key_type

Description The type of the keys with which the multiset is instantiated.

Definition

```
typedef Key key_type;
```

Containers Library

Template class *multiset*<Key>

value_type

Description value_type represents the type of the values stored in the multiset. This is the same as key_type.

Definition `typedef Key value_type;`

pointer

Description The type Key* (pointer to Key).

Definition `typedef Allocator<Key>::pointer pointer`

reference

Description The type Key& that can be used for storing into Key objects.

Definition `typedef Allocator<Key>::reference reference;`

const_reference

Description The type const Key& for const references that can be used for storing into Key objects.

Definition `typedef Allocator<Key>::const_reference
const_reference;`

key_compare

Description The comparison object type, Compare, with which the multiset is instantiated. This type is used to order the keys in the multiset.

Definition `typedef Compare key_compare;`

value_compare

Description This is the ordering relation that is used to order the values stored in the multiset. Its type is the same as key_compare, since the type of a value stored in a multiset is the same as the type of the key.

Definition `typedef Compare value_compare;`

iterator

Description The type `iterator` is a constant bidirectional iterator referring to `const value_type`.

Definition `typedef iterator;`

const_iterator

Description The type `const_iterator` is the same type as `iterator`.

Definition `typedef const_iterator;`

size_type

Description `size_type` is an unsigned integral type that can represent the size of any `multiset` instance.

Definition `typedef size_type;`

difference_type

Description A signed integral type that can represent the difference between any two `multiset::iterator` objects.

Definition `typedef difference_type;`

reverse_iterator

Description Non-constant reverse bidirectional iterator types.

Definition `typedef reverse_iterator;`

const_reverse_iterator

Description Constant reverse bidirectional iterator types.

Definition `typedef const_reverse_iterator;`

Constructors, Destructors and Related Functions multiset

Default Constructor

Description The default constructor. Constructs an empty multiset using the relation comp to order the elements.

Prototype `multiset(const Compare& comp = Compare());`

Overloaded and Copy Constructors

Prototype `multiset(const multiset<Key, Compare>& x);`

Remarks The multiset copy constructor. Constructs a multiset and initializes it with copies of the elements of multiset x.

Prototype `multiset
 (const value_type* first,
 const value_type* last,
 const Compare& comp = Compare());`

Remarks Constructs an empty multiset and initializes it with copies of elements in the range [first,last). The ordering relation comp is used to order the elements of the multiset.

Assignment Operator =

Description The multiset assignment operator. Replaces the contents of the current multiset with a copy of the parameter multiset x.

Prototype `multiset<Key, Compare>& operator=
 (const multiset<Key,
 Compare>& x);`

multiset::swap

Description Swaps the contents of the current multiset with those of the input multiset x. The current multiset replaces x and vice versa.

Prototype `void swap(multiset<Key, Compare>& x);`

Destructor

Description The multiset destructor. Returns all allocated storage back to the free store.

Prototype `~multiset();`

Comparison Operations multiset

Equality Operator ==

Description Equality operation on multisets. Returns true if the sequences of elements in x and y are element-wise equal (using `T::operator==`). Takes linear time.

Prototype `bool operator==
 (const multiset<Key,
 Compare>& x,
 const multiset<Key,
 Compare>& y);`

Less Than Operator <

Description Returns true if x is lexicographically less than y, false otherwise. Takes linear time.

Prototype `bool operator<
 (const multiset<Key,
 Compare>& x,
 const multiset<Key,
 Compare>& y);`

Element Access Member Functions multiset

multiset::key_comp

Description This function returns the comparison object of the multiset. The comparison object is an object of class Compare, which represents the ordering relation used to construct the multiset.

Prototype `key_compare key_comp() const;`

multiset::value_comp

Description Returns an object of type value_compare constructed out of the comparison object. For multisets, this is simply an object of type Compare.

Prototype `value_compare value_comp() const;`

multiset::begin

Description Returns the iterator that can be used to begin traversing through all locations in the multiset.

Prototype `iterator begin() const;`

multiset::end

Description Returns an iterator that can be used in a comparison for ending traversal through the multiset.

Prototype `iterator end() const;`

multiset::rbegin

Description Returns a reverse_iterator, that can be used to begin traversing all locations in the multiset in the reverse of the normal order.

Prototype `reverse_iterator rbegin();`

multiset::rend

Description Returns a reverse_iterator, that can be used in a comparison for ending reverse-direction traversal through all locations in the multiset.

Prototype `reverse_iterator rend();`

multiset::empty

Description Returns true if the multiset is empty, false otherwise.

Prototype `bool empty() const;`

multiset::size

Description Returns the number of elements in the multiset.

Prototype `size_type size() const;`

multiset::max_size

Description Returns the maximum possible size of the multiset. The maximum size is simply the total number of elements of type Key that can be represented in the memory model used.

Prototype `size_type max_size() const;`

Insert Member Functions multiset

multiset::insert

Description Inserts one more elements into the multiset.

Prototype `iterator insert
 (iterator position,
 const value_type& x);`

Remarks Inserts the element x into the multiset if x is not already present in the multiset. The iterator position is a hint, indicating where the insert function should start to search to do the insert. This insertion

Containers Library

Template class *multiset*<Key>

takes $O(\log N)$ time in general, where N is the number of elements in the multiset, but is amortized constant if x is inserted right after the iterator position.

Prototype `iterator insert(const value_type& x);`

Remarks Inserts the element x into the multiset and returns the iterator pointing to the newly inserted element. The function takes $O(\log N)$ time, where N is the number of elements in the multiset.

Prototype `void insert
(const value_type* first,
const value_type* last);`

Remarks Copies of elements in the range $[first, last)$ are inserted into the multiset. This insert member function allows elements from other containers to be inserted into the multiset.

In general, the function takes $O(N \log(\text{size()} + N))$ time, where N is the distance from $first$ to $last$, and $O(N)$ time if the range $[first, last)$ is sorted according to the multiset ordering relation `value_comp()`.

Erase Member Functions *multiset*

multiset::erase

Description Erases one or more multiset elements.

Prototype `void erase(iterator position);`

Remarks Erases the multiset element pointed to by the iterator position. The time taken is amortized constant.

Prototype `size_type erase(const key_type& x);`

Remarks Erases the multiset element with key equal to x (i.e., removes all x 's from the multiset). Returns the number of erased elements. In general, this function takes time proportional to $\log(\text{size}()) + \text{count}(x)$,

where `count(k)` is a multiset member function that returns the number of elements with key equal to `k`.

Prototype `void erase(iterator first, iterator last);`

Remarks The iterators `first` and `last` are assumed to point into the multiset, and all elements in the range `[first,last)` are erased from the multiset. The time taken is $\log(\text{size}()) + N$, where N is the distance from `first` to `last`.

Special Operations multiset

`multiset::find`

Description Searches for the element `x` in the multiset. If `x` is found, the function returns the iterator pointing to it. Otherwise, `end()` is returned. The function takes $O(\log N)$ time, where N is the number of elements in the multiset.

Prototype `iterator find(const key_type& x) const;`

`multiset::count`

Description Returns the number of elements in the multiset that are equal to `x`. The function takes $O(\log N)$ time, where N is the number of elements in the multiset.

Prototype `size_type count(const key_type& x) const;`

`multiset::lower_bound`

Description Returns an iterator pointing to the first multiset element whose key is not less than `x`. If no such element is found, `end()` is returned. The function takes $O(\log N)$ time, where N is the number of elements in the multiset.

Prototype `iterator lower_bound(const key_type& x) const;`

Containers Library

Template class *map*<Key, T>

multiset::upper_bound

Description The `upper_bound` function returns an iterator to the first multiset element whose key is greater than `x`. If no such element is found, `end()` is returned. The function takes $O(\log N)$ time, where N is the number of elements in the multiset.

Prototype `iterator upper_bound(const key_type& x) const;`

multiset::equal_range

Description This function returns the `pair(lower_bound(x), upper_bound(x))`. The function takes $O(\log N)$ time, where N is the number of elements in the multiset.

Prototype `pair<iterator, iterator> equal_range
(const key_type& x) const;`

Template class *map*<Key, T>

Files `#include <map.h>`

Declaration

```
template
    < class Key,
      class T,
      class Compare = less<Key> >
class map;
```

Description A map is an associative container that supports unique keys of a given type `Key`, and provides for fast retrieval of values of another type `T` based on the stored keys. As in all other STL associative containers, the ordering relation `Compare` is used to order the elements of the map.

Maps are necessary because we often need to associate elements of one type with values of another. For example, consider a telephone directory, which contains associations of names (string types) and phone numbers (integers). A `map<string, long>` can be used to pro-

vide for fast retrieval of a phone-number that corresponds to a given name.

Elements are stored in maps as pairs in which each Key has an associated value of type T. Since maps store only unique keys, each map contains at most one <Key, T> pair for each Key value. It is not possible to associate a single Key with more than one value.

The topics in this section are:

- [“Typedef Declarations map” on page 253](#)
- [“Constructors, Destructors and Related Functions map” on page 256](#)
- [“Comparison Operations map” on page 257](#)
- [“Element Access Member Functions map” on page 258](#)
- [“Insert Member Functions map” on page 260](#)
- [“Erase Member Functions map” on page 261](#)
- [“Special Operations map” on page 261](#)

Typedef Declarations map

Description The following typedef’s are defined in the class `map`.

key_type

Description `key_type` represents the type of the keys in the map.

Definition `typedef Key key_type;`

value_type

Description `value_type` represents the type of the values stored in the map. Since maps store pairs of values, `value_type` is a pair which associates every key (of type Key) with a value of type T.

Definition `typedef pair<const Key, T> value_type;`

Containers Library

Template class `map<Key, T>`

key_compare

Description This is the comparison object type, `Compare`, with which the map is instantiated. It is used to order keys in the map.

Definition `typedef Compare key_compare;`

value_compare

Description A class for comparing objects of `map::value_type` (i.e., objects of type `pair<const Key, T>`), by comparing their keys using `map::key_compare`.

Definition `class value_compare;`

iterator

Description `iterator` is a bidirectional iterator referring to `value_type`. It is guaranteed that there is a constructor for `const_iterator` out of `iterator`.

Definition `typedef iterator;`

const_iterator

Description `Const_iterator` is a constant bidirectional iterator referring to `const value_type`. It is guaranteed that there is a constructor for `const_iterator` out of `iterator`.

Definition `typedef const_iterator;`

pointer

Description The type `value_type*`. (i.e., `pair<const Key, T>*`).

Definition `typedef Allocator<value_type>::pointer pointer;`

reference

Description The type `pair<const Key, T>&` that can be used for storing into `map::value_type` objects.

Definition `typedef Allocator<value_type>::reference
 reference;`

const_reference

Description The type `const Key,T>&` (`const pair<const Key,T>&` for `const` references that can be used for storing into `map::value_type` objects.

Definition `typedef Allocator<value_type>::const_reference
 const_reference;`

size_type

Description `size_type` is an unsigned integral type that can represent the size of any `map` instance.

Definition `typedef size_type;`

difference_type

Description A signed integral type that can represent the difference between any two `map::iterator` objects.

Definition `typedef difference_type;`

reverse_iterator

Description Non-constant reverse bidirectional iterator types.

Definition `typedef reverse_iterator;`

const_reverse_iterator

Description Constant reverse bidirectional iterator types.

Definition `typedef const_reverse_iterator;`

Constructors, Destructors and Related Functions map

Default Constructor

Description The default constructor. Constructs an empty map using the relation `comp` to order the elements.

Prototype `map(const Compare& comp = Compare());`

Overloaded and Copy Constructors

Prototype `map(const map<Key, T, Compare>& x);`

Remarks The map copy constructor. Constructs a map and initializes it with copies of the elements of map `x`.

Prototype `map
 (const value_type* first,
 const value_type* last,
 const Compare& comp = Compare());`

Remarks Constructs an empty map and initializes it with copies of elements in the range `[first,last)`. The ordering relation `comp` is used to order the elements of the map.

Assignment Operator =

Description The map assignment operator. Replaces the contents of the current map with a copy of the parameter map `x`.

Prototype `map<Key, T, Compare>& operator=
 (const map<Key,
 Compare>& x);`

map::swap

Description Swaps the contents of the current map with those of the input map `x`. The current map replaces `x` and vice versa.

Prototype `void swap(map<Key, T, Compare>& x);`

Destructor

Description The map destructor. Returns all allocated storage back to the free store.

Prototype `~map();`

Comparison Operations map

Equality Operator ==

Description Equality operation on maps. Returns true if the sequences of elements in x and y are element-wise equal (using T::operator==). Takes linear time.

Prototype `bool operator==
 (const map<Key,
 Compare>& x,
 const map<Key,
 Compare>& y);`

Less Than Operator <

Description Returns true if x is lexicographically less than y, false otherwise. Takes linear time.

Prototype `bool operator<
 (const map<Key,
 Compare>& x,
 const map<Key,
 Compare>& y);`

Element Access Member Functions map

map::key_comp

Description This function returns the comparison object of the map. The comparison object is an object of class Compare, which represents the ordering relation used to construct the map.

Prototype `key_compare key_comp() const;`

map::value_comp

Description Returns an object of type value_compare constructed out of the comparison object. For maps, value_compare is a class that can be used to compare values stored as pairs in the map.

Prototype `value_compare value_comp() const;`

map::begin

Description Returns an iterator (const_iterator for constant map) that can be used to begin traversing through all locations in the map.

Prototype `iterator begin()
const_iterator begin() const;`

map::end

Description Returns an iterator (const_iterator for constant map) that can be used in a comparison for ending traversal through the map.

Prototype `iterator end()
const_iterator end() const;`

map::rbegin

Description Returns a reverse_iterator (const_reverse_iterator for constant maps), that can be used to begin traversing all locations in the map in the reverse of the normal order.

Prototype `reverse_iterator rbegin();`
 `const_reverse_iterator rbegin() const;`

map::rend

Description Returns a `reverse_iterator` (`const_reverse_iterator` for constant maps), that can be used in a comparison for ending reverse-direction traversal through all locations in the map.

Prototype `reverse_iterator rend();`
 `const_reverse_iterator rend() const;`

map::empty

Description Returns true if the map is empty, false otherwise.

Prototype `bool empty() const;`

map::size

Description Returns the number of elements in the map.

Prototype `size_type size() const;`

map::max_size

Description Returns the maximum possible size of the map. The maximum size is simply the total number of elements of type `Key` that can be represented in the memory model used.

Prototype `size_type max_size() const;`

Sub Operator []

Description For a `map<Key, T, Compare>`, this operator returns the element of type `T` that is associated with the key `Key`.

Prototype `reference operator[](const key_type& x);`

Remarks The map subscripting operator is different from the subscripting operator of vectors and deques in that if the map contains no element of type `T` associated with Key `x`, then the pair `(x, T())` is inserted into the map.

Insert Member Functions `map`

`map::insert`

Description Inserts one or more elements or values into the map.

Prototype

```
iterator insert
    (iterator position,
     const value_type& x);
```

Remarks Inserts the value `x` into the map if `x` is not already present in the map. The iterator position is a hint, indicating where the `insert` function should start to search to do the insert. This insertion takes $O(\log N)$ time in general, where N is the number of elements in the set, but is amortized constant if `x` is inserted right after the iterator position.

Prototype

```
pair<iterator, bool> insert
    (const value_type& x);
```

Remarks Inserts the value `x` into the map if `x` is not already present in the map. The returned value is a `pair`, whose `bool` component indicates whether the insertion has taken place, and whose iterator component points to the just inserted value in the map, if the insertion takes place, otherwise to the value `x` already present.

NOTE: If Notice, that `x` is a pair of type `pair< Key, T>`. The function takes $O(\log N)$ time, where N is the number of elements in the set.

Prototype

```
void insert
    (const value_type* first,
     const value_type* last);
```

Remarks Copies of elements in the range [first,last) are inserted into the map. This insert member function allows elements from other containers to be inserted into the map. In general, this insertion takes $O(N\log(\text{size}()+N))$ time, where N is the distance from first to last, and $O(N)$ time if the range [first,last) is sorted according to the map ordering relation `value_comp()`.

Erase Member Functions map

`map::erase`

Description Erases one or more map elements.

Prototype `void erase(iterator position);`

Remarks Erases the map element pointed to by the iterator position. The time taken is amortized constant.

Prototype `size_type erase(const key_type& x);`

Remarks Erases the map element with key equal to x (i.e., removes all pairs whose first element is x from the map). Returns the number of erased elements, which is 1 if x is present in the map, and 0 otherwise. In general, this function takes time proportional to $\log(\text{size}()) + \text{count}(x)$, where `count(k)` is a map member function that returns the number of elements with key equal to k .

Prototype `void erase(iterator first, iterator last);`

Remarks The iterators `first` and `last` are assumed to point into the map, and all elements in the range [first,last) are erased from the map. The time taken is $\log(\text{size}())+N$, where N is the distance from first to last.

Special Operations map

`map::find`

Description Searches the map for an element with Key equal to x . If such an element is found, the function returns the iterator (`const_iterator` for

Containers Library

Template class `map<Key, T>`

constant maps) pointing to it. Otherwise, `end()` is returned. The function takes $O(\log N)$ time, where N is the number of elements in the map.

Prototype `iterator find(const key_type& x);`
`const_iterator find(const key_type& x) const;`

map::count

Description Returns the number of elements in the map with Key equal to x . If an element with Key equal to x has been inserted into the map then this number is always 1; otherwise, it is 0.

Prototype `size_type count(const key_type& x) const;`

map::lower_bound

Description Returns an iterator (`const_iterator` for constant maps) pointing to the first map element whose key is not less than x . If the map contains an element with key not less than x , then the returned iterator points to this element. If such an element is not present in the map, `end()` is returned. The function takes $O(\log N)$ time, where N is the number of elements in the map.

Prototype `iterator lower_bound`
`(const key_type& x);`
`const_iterator lower_bound`
`(const key_type& x) const;`

map::upper_bound

Description The `upper_bound` function returns an iterator (`const_iterator` for constant maps) to the first map element whose key is greater than x . If no such element is found, `end()` is returned. The function takes $O(\log N)$ time, where N is the number of elements in the map.

Prototype `iterator upper_bound`
`(const key_type& x)`
`const_iterator upper_bound`
`(const key_type& x) const;`

map::equal_range

Description This function returns the pair(lower_bound(x), upper_bound(x)) . The function takes O(log N) time, where N is the number of elements in the map.

Prototype

```
pair<iterator, iterator> equal_range
    (const key_type& x);
pair<const_iterator, const_iterator> equal_range
    (const key_type& x) const;
```

Template class multimap<Key, T>

Files `#include <multimap.h>`

Declaration

```
template
    < class Key,
      class T,
      class Compare = less<Key> >
class multimap;
```

It is assumed that the operators operator== and an operator< are defined on the type Key.

Description A multimap is an associative container that stores allows users to store multiple keys of a given type Key, and to efficiently retrieve values of another type T based on the stored Key. As in all other STL associative containers, the ordering relation Compare is used to order the elements of the map.

Multimaps are necessary because we often need to associate more than one object of type T with each Key.

For example, consider a telephone directory organized by last names. Here we might need to associate different telephone numbers (of type integer) with all names ending in Smith. A multimap<name, integer> can be used to hold this information (where name is an appropriately defined type). Corresponding to each name, there might be several telephone numbers, allowing us to

Containers Library

Template class `multimap<Key, T>`

easily determine the telephone numbers of different people all of whose last names are Smith.

Elements are stored in multimaps as pairs in which each Key has an associated value of type T. Since multimaps allow multiple keys, it is possible to associate a single Key with more than one value (as was done in the example above).

The topics in this section are:

- [“Typedef Declarations multimap” on page 264](#)
- [“Constructors, Destructors and Related Functions multimap” on page 267](#)
- [“Comparison Operations multimap” on page 268](#)
- [“Element Access Member Functions multimap” on page 269](#)
- [“Insert Member Functions multimap” on page 270](#)
- [“Erase Member Functions multimap” on page 271](#)
- [“Special Operations multimap” on page 272](#)

Typedef Declarations multimap

Description The following typedef’s are defined in the class `multimap`.

key_type

Description `key_type` represents the type of the keys in the multimap.

Definition `typedef Key key_type;`

value_type

Description `value_type` represents the type of the values stored in the multimap. Since multimaps store pairs of values, `value_type` is a pair which associates every key (of type `Key`) with a value of type T.

Definition `typedef pair<const Key, T> value_type;`

key_compare

Description This is the comparison object type, `Compare`, with which the map is instantiated. It is used to order keys in the map.

Definition `typedef Compare key_compare;`

value_compare

Description A class for comparing objects of `multimap::value_type` (i.e., objects of type `pair<const Key,T>`), by comparing their keys using `multimap::key_compare`.

Definition `class value_compare;`

iterator

Description Iterator is a bidirectional iterator referring to `value_type`. It is guaranteed that there is a constructor for `const_iterator` out of `iterator`.

Definition `typedef iterator,;`

const_iterator

Description `Const_iterator` is a constant bidirectional iterator referring to `const value_type`. It is guaranteed that there is a constructor for `const_iterator` out of `iterator`.

Definition `typedef const_iterator;`

value_type*

Description The type `value_type*` (i.e., `pair<const Key,T>*`).

Definition `typedef Allocator<value_type>::pointer pointer;`

reference

Description The type `pair<const Key,T>&` that can be used for storing into `map::value_type` objects.

Containers Library

Template class *multimap*<Key, T>

Definition `typedef Allocator<value_type>::reference
 reference;`

const_reference

Description The type `const pair<const Key,T>&` for const references that can be used for storing into `map::value_type` objects.

Definition `typedef Allocator<value_type>::const_reference
 const_reference;`

size_type

Description `size_type` is an unsigned integral type that can represent the size of any `map` instance.

Definition `typedef size_type;`

difference_type

Description A signed integral type that can represent the difference between any two `map::iterator` objects.

Definition `typedef difference_type;`

reverse_iterator

Description Non-constant reverse bidirectional iterator type.

Definition `typedef reverse_iterator;`

const_reverse_iterator

Description Constant reverse bidirectional iterator type.

Definition `typedef const_reverse_iterator;`

Constructors, Destructors and Related Functions `multimap`

Default Constructor

Description The default constructor. Constructs an empty `multimap` using the `relationcomp` to order the elements.

Prototype `multimap(const Compare& comp = Compare());`

Overloaded and Copy Constructors

Prototype `multimap
 (const multimap<Key,
 T,
 Compare>& x);`

Remarks The `multimap` copy constructor. Constructs a `multimap` and initializes it with copies of the elements of `multimap x`.

```
multimap  
    (const value_type* first,  
    const value_type* last,  
    const Compare& comp = Compare());
```

Remarks Constructs an empty `multimap` and initializes it with copies of elements in the range `[first,last)`. The ordering relation `comp` is used to order the elements of the `multimap`.

Assignment Operator `=`

Description The `multimap` assignment operator. Replaces the contents of the current `multimap` with a copy of the parameter `multimap x`.

Prototype `multimap<Key, T, Compare>& operator=
 (const multimap<Key,
 Compare>& x);`

Containers Library

Template class *multimap*<Key, T>

multimap::swap

Description Swaps the contents of the current multimap with those of the input multimap x. The current multimap replaces x and vice versa.

Prototype `void swap(multimap<Key, T, Compare>& x);`

Destructor

Description The multimap destructor. Returns all allocated storage back to the free store.

Prototype `~multimap();`

Comparison Operations multimap

Equality Operator ==

Description Equality operation on multimaps. Returns true if the sequences of elements in x and y are element-wise equal (using T::operator==). Takes linear time.

Prototype `bool operator==(const multimap<Key, Compare>& x, const multimap<Key, Compare>& y);`

Less Than Operator <

Description Returns true if x is lexicographically less than y, false otherwise. Takes linear time.

Prototype `bool operator<(const multimap<Key, Compare>& x, const multimap<Key, Compare>& y);`

Element Access Member Functions `multimap`

`multimap::key_comp`

Description This function returns the comparison object of the `multimap`. The comparison object is an object of class `Compare`, which represents the ordering relation used to construct the `multimap`.

Prototype `key_compare key_comp() const;`

`multimap::value_comp`

Description Returns an object of type `value_compare` constructed out of the comparison object. For `multimaps`, `value_compare` is a class that can be used to compare values stored as pairs in the `multimap`.

Prototype `value_compare value_comp() const;`

`multimap::begin`

Description Returns an iterator (`const_iterator` for constant `multimap`) that can be used to begin traversing through all locations in the `multimap`.

Prototype `iterator begin();`
`const_iterator begin() const;`

`multimap::end`

Description Returns an iterator (`const_iterator` for constant `multimap`) that can be used in a comparison for ending traversal through the `multimap`.

Prototype `iterator end();`
`const_iterator end() const;`

`multimap::rbegin`

Description Returns a `reverse_iterator` (`const_reverse_iterator` for constant `multimaps`), that can be used to begin traversing all locations in the vector in the reverse of the normal order.

Containers Library

Template class *multimap*<Key, T>

Prototype `reverse_iterator rbegin();`
 `const_reverse_iterator rbegin() const;`

multimap::rend

Description Returns a `reverse_iterator` (`const_reverse_iterator` for constant `multimaps`), that can be used in a comparison for ending reverse-direction traversal through all locations in the `multimap`.

Prototype `reverse_iterator rend();`
 `const_reverse_iterator rend() const;`

multimap::empty

Description Returns true if the `multimap` is empty, false otherwise.

Prototype `bool empty() const;`

multimap::size

Description Returns the number of elements in the `multimap`.

Prototype `size_type size() const;`

multimap::max_size

Description Returns the maximum possible size of the `multimap`. The maximum size is simply the total number of elements of type `Key` that can be represented in the memory model used.

Prototype `size_type max_size() const;`

Insert Member Functions multimap

multimap::Insert

Description Inserts a value or elements into a `multimap` object.

Prototype `iterator insert`
 `(iterator position,`

```
const value_type& x);
```

Remarks Inserts the value `x` into the `multimap` if `x` is not already present in the `multimap`. The iterator position is a hint, indicating where the `insert` function should start to search to do the insert. The insertion takes $O(\log N)$ time in general, where N is the number of elements in the map, but is amortized constant if `x` is inserted right after the iterator position.

Prototype `iterator insert(const value_type& x);`

Remarks Inserts the value `x` into the `multimap` and returns the iterator pointing to the newly inserted value.

NOTE: The value `x` is a pair of the form `pair<const Key, T>`. The insertion takes $O(\log N)$ time, where N is the number of elements in the map.

```
void insert
    (const value_type* first,
     const value_type* last);
```

Remarks Copies of elements in the range `[first,last)` are inserted into the `multimap`. This `insert` member function allows elements from other containers to be inserted into the `multimap`. In general, the time taken for this insertion is $O(N \log(\text{size}() + N))$, where N is the distance from `first` to `last`, and $O(N)$ if the range `[first,last)` is sorted according to the `multimap` ordering relation `value_comp()`.

Erase Member Functions `multimap`

`multimap::erase`

Description Erases one or more elements or values from the `multimap` object.

Prototype `void erase(iterator position);`

Remarks Erases the `multimap` element pointed to by the iterator position. The time taken is amortized constant.

Containers Library

Template class *multimap*<Key, T>

Prototype `size_type erase(const key_type& x);`

Remarks Erases the multimap element with key equal to *x* (i.e., removes all pairs whose first element is *x* from the multimap). Returns the number of erased elements.

NOTE: There could be more than one multimap element with key equal to *x*, since multimaps allow multiple keys.

In general, this function takes time proportional to $\log(\text{size}()) + \text{count}(x)$, where $\text{count}(k)$ is a multimap member function that returns the number of elements with key equal to *k*.

Prototype `void erase(iterator first, iterator last);`

Remarks The iterators *first* and *last* are assumed to point into the multimap, and all elements in the range $[\text{first}, \text{last})$ are erased from the multimap. The time taken is $O(\log(\text{size}()) + N)$, where *N* is the distance from *first* to *last*.

Special Operations multimap

multimap::find

Description Searches the multimap for an element with Key equal to *x*. If such an element is found, the function returns the iterator (`const_iterator` for constant multimaps) pointing to it. Otherwise, `end()` is returned. The function takes $O(\log N)$ time, where *N* is the number of elements in the multimap.

Prototype `iterator find(const key_type& x);`
`const_iterator find(const key_type& x) const;`

multimap::count

Description Returns the number of elements in the multimap with key equal to *x*.

Prototype `size_type count(const key_type& x) const;`

`multimap::lower_bound`

Description Returns an iterator (`const_iterator` for constant `multimaps`) pointing to the first `multimap` element whose key is not less than `x`. If the `multimap` contains an element with Key not less than `x`, then the returned iterator points to this single element. If such an element is not present in the `multimap`, `end()` is returned. The function takes $O(\log N)$ time, where N is the number of elements in the `multimap`.

Prototype `iterator lower_bound`
 `(const key_type& x) const;`
`const_iterator lower_bound`
 `(const key_type& x) const;`

`multimap::upper_bound`

Description The `upper_bound` function returns an iterator (`const_iterator` for constant `multimaps`) to the first `multimap` element whose key is greater than `x`. If no such element is found, `end()` is returned. The function takes $O(\log N)$ time, where N is the number of elements in the `multimap`.

Prototype `iterator upper_bound`
 `(const key_type& x) const;`
`const_iterator upper_bound`
 `(const key_type& x) const;`

`multimap::equal_range`

Description This function returns the `pair(lower_bound(x), upper_bound(x))`. The function takes $O(\log N)$ time, where N is the number of elements in the `multimap`.

Prototype `pair<iterator, iterator> equal_range`
 `(const key_type& x) const;`
`pair<const_iterator, const_iterator> equal_range`
 `(const key_type& x) const;`

Template class stack

Files `#include <stack.h>`

Declaration `template <class Container> class stack;`

NOTE: It is assumed that the operators `operator==` and `operator<` are defined for objects of type `Container`.

Description A stack is a data structure that allows the following operations: insertion at one end, deletion from the same end, retrieving the value at the end, and testing for emptiness. Thus, stacks provide a “last-in/first-out” service. The element deleted or retrieved is always the last one inserted.

STL provides a stack container adapter, which can be used to instantiate a stack with any container that supports the following operations: `back`, `push_back`, and `pop_back`. In particular, vectors, lists and deques can be used to instantiate stacks.

The topics in this section are:

- [“Public Member Functions stack” on page 274](#)
- [“Comparison Operations stack” on page 275](#)

Public Member Functions stack

stack::empty

Description Returns true if the stack is empty, false otherwise.

Prototype `bool empty() const;`

stack::size

Description Returns the current size of the stack (i.e the number of elements the stack currently holds).

Prototype `size_type size() const;`

stack::top

Description Returns the element at the top of the stack. The stack remains unchanged.

Prototype `value_type& top() const;,
const value_type& top() const;`

stack::push

Description Inserts the value `x` at the top of the stack.

Prototype `void push(const value_type& x);`

stack::pop

Description Removes the element at the top of the stack.

Prototype `void pop();`

Comparison Operations stack

Equality Operator ==

Description Equality operation on stacks. Returns true if the sequences of elements in `x` and `y` are element-wise equal (using `T::operator==`). Takes linear time.

Prototype `bool operator==
 (const stack<container>& x,
 const stack<container>& y);`

Less Than Operator <

Description Returns true if `x` is lexicographically less than `y`, false otherwise. Takes linear time.

Prototype `bool operator<`

```
(const stack<container>& x,  
const stack<container>& y);
```

Template class queue

Files `#include <queue.h>`

Declaration `template <class Container> class queue;`

It is assumed that the operators `operator==` and an `operator<` are defined for objects of type `Container`.

Description A queue is a data structure in which elements are inserted at one end and removed from the opposite end. The order of removal is the same as the order of insertion.

STL provides a queue container adapter, which can be used to instantiate a queue with any container that supports the following operations: `empty`, `size`, `front`, `back`, `push_back`, and `pop_front`. In particular, lists and deques can be used to instantiate queues:

`queue< list<int> >`, declares a queue of integers with an underlying list implementation

`queue< deque<float> >`, declares a queue of floats with an underlying deque implementation.

NOTE: The vectors cannot be used to instantiate queues, since they do not provide a `pop_front` function. This function is not provided for vectors, since it would be highly inefficient for long vectors.

The topics in this section are:

- [“Public Member Functions queue” on page 277](#)
- [“Comparison Operations queue” on page 278](#)

Public Member Functions queue

queue::empty

Description Returns true if the queue is empty, false otherwise.

Prototype `bool empty() const;`

queue::size

Description Returns the current size of the queue (i.e the number of elements the queue currently holds).

Prototype `size_type size() const;`

queue::front

Description Returns the element at the front of the queue. The queue remains unchanged.

Prototype `value_type& front() const;`
`const value_type& front() const;`

queue::back

Description Returns the element at the end of the queue. This is the element that was last inserted into the queue. The queue remains unchanged.

Prototype `value_type& back() const;`
`const value_type& back() const;`

queue::push

Description Adds the element x at the end of the queue.

Prototype `void push(const value_type& x);`

queue::pop

Description Removes the element at the front of the queue.

Containers Library

Template class *priority_queue*

Prototype `void pop()`

Comparison Operations `queue`

Equality Operator `==`

Description Equality operation on queues. Returns true if the sequences of elements in `x` and `y` are element-wise equal (using `T::operator==`). Takes linear time.

Prototype `bool operator==
 (const queue<container>& x,
 const queue<container>& y);`

Less Than Operator `<`

Description Returns true if `x` is lexicographically less than `y`, false otherwise. Takes linear time.

Prototype `bool operator<
 (const queue<container>& x,
 const queue<container>& y);`

Template class `priority_queue`

Files `#include <queue.h>`

Declaration `template <class Container> class priority_queue;`

It is assumed that the operators `operator==` and `operator<` are defined for objects of type `Container`.

Description A priority queue is a container in which the element immediately available for retrieval is the largest of those in the container, for some particular way of ordering the elements. The order of removal is the same as the order of insertion.

STL provides a `priority_queue` container adapter, which can be used to instantiate a `priority_queue` with any container that sup-

ports the following operations: `empty`, `size`, `front`, `push_back`, and `pop_back`. In particular, vectors and deques can be used to instantiate `priority_queues`.

NOTE: Since `priority_queues` involve an ordering on their elements, a comparison function object `comp` needs to be supplied to instantiate a `priority_queue`. For example:

`priority_queue< vector<int>, less<int> >`, declares a `priority_queue` of integers with a vector implementation and using the built-in `<` operation for integers to compare the objects.

`priority_queue< deque<float>, greater<float> >`, declares a `priority_queue` of floats with a deque implementation, using the `>` operation on floats for comparisons.

NOTE: Since `>` is used instead of `<`, the element available for retrieval at any time is actually the smallest element rather than the largest.

The topics in this section are:

- [“Constructors `priority_queue`” on page 279](#)
- [“Public Member Functions `priority_queue`” on page 280](#)
- [“Comparison Operations `priority_queue`” on page 281](#)

Constructors `priority_queue`

Default Constructor

Description	The default constructor. Constructs a <code>priority_queue</code> using a comparison function object of type <code>Compare</code> .
Prototype	<code>priority_queue (const Compare& x = Compare());</code>

Overloaded and Copy Constructors

Prototype `priority_queue`
 `(const value_type* first,`
 `const value_type* last,`
 `const Compare& x = Compare());`

Remarks Constructs a priority queue whose elements are copies of elements in the range [first,last). The default comparison function used is Compare.

Public Member Functions *priority_queue*

priority_queue::empty

Description Returns true if the *priority_queue* is empty, false otherwise.

Prototype `bool empty() const;`

priority_queue::size

Description Returns the current size of the *priority_queue* (i.e the number of elements the *priority_queue* currently holds).

Prototype `size_type size() const;`

priority_queue::top

Description Returns the element at the top of the *priority_queue*. The *priority_queue* remains unchanged.

Prototype `value_type& top() const;`
 `const value_type& top() const;`

priority_queue::push

Description Adds the element *x* to the *priority_queue*.

Prototype `void push(const value_type& x);`

`priority_queue::pop`

Description Removes the element at the top of the `priority_queue`.

Prototype `void pop();`

Comparison Operations `priority_queue`

Equality and comparison operations are not provided for `priority_queue`s.

Containers Library

Template class priority_queue



Iterators Library

This chapter presents the concept of iterators in detail, defining and illustrating the five iterator categories of input iterators, output iterators, forward iterators, bidirectional iterators and random access iterators.

Overview of Iterators

This chapter is a reference guide to the requirements that must be satisfied by a class or a built-in type to be used as an iterator. The – iterators of a particular category include: stream iterator classes, iterator adaptors (reverse iterators and insert iterators).

The principle sections in this chapter are:

- [“Iterator Requirements” on page 285](#)
- [“Stream Iterators” on page 290](#)
- [“Template class istream_iterator” on page 291](#)
- [“Template class ostream_iterator” on page 294](#)
- [“Template class istreambuf_iterator” on page 296](#)
- [“Template class ostreambuf_iterator” on page 300](#)
- [“Template class reverse_bidirectional_iterator” on page 303](#)
- [“Template class reverse_iterator” on page 306](#)
- [“Template class back_insert_iterator” on page 310](#)
- [“Template class front_insert_iterator” on page 312](#)
- [“Template class insert_iterator” on page 313](#)

The following terminology is used in the statement of iterator requirements.

Value type

Iterators are objects that have `operator*` returning a value of some class or built-in type `T` called the `value type` of the iterator.

Distance type

For every iterator type for which equality is defined, there is a corresponding signed integral type called the `distance type` of the iterator.

Past-the-end values

Just as a regular pointer to an array guarantees that there is a valid pointer value pointing past the last element of the array, so for any iterator type there is an iterator value that points past the last element of a corresponding container. These values are called `past-the-end` values.

Dereferenceable values.

Values of the iterator for which `operator*` is defined are called `dereferenceable`. STL components never assume that `past-the-end` values are `dereferenceable`.

Singular values.

Iterators might also have `singular` values that are not associated with any container. For example, after the declaration of an uninitialized pointer `x` (as with `int* x;`), `x` should always be assumed to have a `singular` value of a pointer. Results of most expressions are undefined for `singular` values. The only exception is an assignment of a `non-singular` value to an iterator that holds a `singular` value. In this case the `singular` value is overwritten the same way as any other value. `Dereferenceable` and `past-the-end` values are always `non-singular`.

Reachability

An iterator `j` is called reachable from an iterator `i` if and only if there is a finite sequence of applications of `operator++` to `i` that makes `i == j`. If `i` is reachable from `j`, they refer to the same container.

Ranges

Most of the library's algorithmic templates that operate on containers have interfaces that use ranges. A range is a pair of iterators that serve as beginning and end markers for a computation. A range `[i, i)` is an empty range; in general, a range `[i, j)` refers to the positions in a container starting with the one referred to by `i` up to but not including the one pointed to by `j`. Range `[i, j)` is valid if and only if `j` is reachable from `i`. The result of the application of the algorithms in the library to invalid ranges is undefined.

Mutable versus constant

Iterators can be mutable or constant depending on whether the result of `operator*` behaves as a reference or as a reference to a constant. Constant iterators do not satisfy the requirements for output iterators.

NOTE: For all iterator operations that are required in each category, the computing time requirement is `constant time` (amortized). For this reason, we will not mention computing times separately in any of the following sections on requirements.

Iterator Requirements

This section discusses the requirements for all five types of iterators. The topics in this section are:

- [“Input Iterator Requirements” on page 286](#)
- [“Output Iterator Requirements” on page 287](#)

- [“Forward Iterator Requirements” on page 288](#)
- [“Random Access Iterator Requirements” on page 289](#)
- [“Bidirectional Iterator Requirements” on page 289](#)

In this and the following four requirements sections, for each iterator type X we will assume

- a and b denote values of type X ,
- n denotes a value of the distance type for X ,
- r denotes a value of $X\&$,
- t denotes a value of value type T , and
- u , tmp , and m denote identifiers.

Input Iterator Requirements

A class or a built-in type X satisfies the requirements of an input iterator for the value type T if and only if the expressions described below are valid.

$X(a)$ The copy constructor, which makes $X(a) == a$. A destructor is assumed.

$X\ u(a); \ X\ u = a;$ Either of these results in $u == a$.

$a == b$ The return type must be convertible to `bool`, and `==` must be an equivalence relation.

$a != b$ The return type must be convertible to `bool`, and the result must be the same as `!(a == b)`.

$*a$ The return type must be convertible to T . It is assumed that a is dereferenceable. If $a == b$, then it must be the case that $*a == *b$.

$++r$ The return type must be convertible to `const X&`. It is assumed that r is dereferenceable. The result is that r is either dereferenceable or r is the past-the-end value of the container, and `&r == &++r`.

$r++$ The return type must be convertible to `const X&`. The result must be the same as that of `{ X tmp = r; ++r; return tmp; }`

$*++r \ *r++$ The return type must be convertible to T .

NOTE: For input iterators, `a == b` does not imply `++a == ++b`. The main consequence is that algorithms on input iterators should be single pass algorithms; i.e., they should never attempt to copy the value of an iterator and use it to pass through the same position twice. Furthermore, value type `T` is not required to be an lvalue type, so algorithms on input iterators should not attempt to assign through them. (Forward iterators remove these restrictions.)

Output Iterator Requirements

A class or a built-in type `X` satisfies the requirements of an output iterator for the value type `T` if and only if the expressions described below are valid.

`X(a); *a = t` is equivalent to `*X(a) = t`. Further, a destructor is assumed in this case.

`X u(a); X u = a;` The result is that `u` is a copy of `a`.

NOTE: However that equality and inequality are not necessarily defined, and algorithms should not attempt to use output iterators to pass through a position twice (i.e., should be single-pass).

`*a = t` `t` is assigned through the iterator to the position to which `a` refers. The result of this operation is not used.

`++r` The return type must be convertible to `const X&`. It is assumed that `r` is dereferenceable on the left hand side of an assignment. The result is that `r` is either dereferenceable on the left hand side of an assignment or `r` is the past-the-end value of the container, and `&r == &++r`.

`r++` The return type must be convertible to `const X&`. The result must be the same as that of `{ X tmp = r; ++r; return tmp; }`

`*++r` `*r++` The return type must be convertible to `T`.

NOTE: The only valid use of `operator*` on output iterators is on the left hand side of an assignment statement. As with input it-

erators, algorithms that use output iterators should be single-pass. Equality and inequality operators might not be defined. Algorithms that use output iterators can be used with ostream as the destination for placing data via the `ostream_iterator` class as well as with insert iterators and insert pointers.

Forward Iterator Requirements

A class or a built-in type `X` satisfies the requirements of a forward iterator for the value type `T` if and only if the expressions described below are valid.

`X u;` The resulting value of `u` might be singular. A destructor is assumed.

`X();` `X()` might be singular.

`X(a);` The result is required to satisfy `a == X(a)`.

`X u(a);` **`X u = a;`** The result is required to satisfy `u == a`.

`a == b` The return type must be convertible to `bool`, and `==` must be an equivalence relation.

`a != b` The return type must be convertible to `bool`, and the result must be the same as `!(a == b)`.

`r = a` The return type is `X&` and the result must satisfy `r == a`.

`*a` The return type must be convertible to `T`. It is assumed that `a` is dereferenceable. If `a == b`, then it must be the case also that `*a == *b`. If `X` is mutable, `*a = t` is valid.

`++r` The return type must be convertible to `X&`. It is assumed that `r` is dereferenceable, and the result is that `r` is either dereferenceable or is the past-the-end value, and `&r == &++r`. Moreover, `r == s` and `r` is dereferenceable implies `++r == ++s`.

`r++` The return type must be convertible to `const X&`. The result must be the same as that of `{ X tmp = r; ++r; return tmp; }`

`*++r` **`*r++`** The return type must be convertible to `T`.

NOTE: The condition that `a == b` implies `++a == ++b` (which is not true for input or output iterators) and the removal of the restrictions on the number of the assignments through the iterator (which applies to output iterators) allows the use of multi-pass one-directional algorithms with forward iterators.

Bidirectional Iterator Requirements

A class or a built-in type `X` satisfies the requirements of a bidirectional iterator for the value type `T` if and only if the expressions described below are valid, in addition to the requirements that are described in the previous section, forward iterators.

--r The return type is `X&`. It is assumed that there exists `s` such that `r == ++s`, and the result is `r` refers to the same position as `s`, is dereferenceable and `&r == &--r`. Both of the following properties must hold: `--(++r) == r`, and if `--r == --s` implies that `r == s`.

r-- The return type must be convertible to `const X&`.

***r--** The return type must be convertible to `T`.

Random Access Iterator Requirements

A class or a built-in type `X` satisfies the requirements of a random access iterator for the value type `T` if and only if the expressions described below are valid, in addition to the requirements of a bidirectional iterator type.

r += n The return type must be `X&`. The result must be the same as would be computed by

```
{ Distance m = n;
  if (m >= 0)
    while (m-- > 0) ++r;
  else
    while (m++ < 0) --r;
  return r; }
```

but is computed in constant time.

a + n n + a The return type must be X. The result must be the same as would be computed by

`{ X tmp = a; return tmp += n; }.`

r -= n The return type must be X&. The result must be the same as would be computed by `r += -n`.

a - n The return type must be X. The result must be the same as would be computed by `{ X tmp = a; return tmp -= n; }.`

b - a The return type must be Distance. It is assumed that there exists a value `n` of the type Distance such that `a + n == b`; the result returned is `n`.

a[n] The return type must be convertible to T.

a < b The return type must be convertible to bool, and `<` must be a total ordering relation.

a > b The return type must be convertible to bool, and `>` must be a total ordering relation opposite to `<`.

a >= b The return type is convertible to bool, and the result must be the same as that of `!(a < b)`.

a <= b The return type is convertible to bool, and the result must be the same as that of `!(b < a)`.

Stream Iterators

The library provides `stream iterators`, defined by template classes, to allow algorithms to work directly with input/output streams. The `istream_iterator` class defines input iterator types and the `ostream_iterator` class defines output iterator types. For example, the following code fragment:

```
istream_iterator<int> end_of_stream;  
partial_sum_copy(istream_iterator<int>(cin),  
end_of_stream, ostream_iterator<int>(cout, "\n"));
```

reads a file containing integers from the input stream `cin`, and prints the partial sums to `cout`, separated by newline characters.

The two stream iterators are:

- [“Template class istream_iterator” on page 291](#)

- [“Template class ostream_iterator” on page 294](#)

The `istream_iterators` are used to read values from the input stream for which they are constructed. The `ostream_iterators`, are used to write values into the output stream for which they are constructed.

Template class `istream_iterator`

Files `#include <iterator.h>`

Description An `istream_iterator<T>` reads (using `operator>>`) successive elements of type `T` from the input stream for which it was constructed. Each time `++` is used on a constructed `istream_iterator<T>` object, the iterator reads and stores a value of `T`. The end of stream value is reached when `operator void*()` on the stream returns `false`. In this case, the iterator becomes equal to the `end-of-stream` iterator value. This end-of-stream value can only be constructed using the constructor with no arguments: `istream_iterator<T>()`. Two end-of-stream iterators are always equal. An end-of-stream iterator is not equal to a non-end-of-stream iterator. Two non-end-of-stream iterators are equal when they are constructed from the same stream.

One can only use `istream_iterators` to read values; it is impossible to store anything into a position referred to by an `istream_iterator` value.

The main peculiarity of `istream` iterators is that fact that `++` operators are not equality-preserving; that is, `i == j` does not guarantee that `++i == ++j`. Every time `++` is used a new value is read from the associated `istream`. The practical consequence of this fact is that `istream` iterators can only be used with single-pass algorithms.

Prototype `template`
 `template`
 `< class T,`
 `class charT,`
 `class traits = char_traits<charT>`

Iterators Library

Template class *istream_iterator*

```
class Distance = ptrdiff_t >
class istream_iterator: input_iterator<T,Distance>;
```

The topics in this section are:

- [“Constructor istream_iterator” on page 292](#)
- [“Public Member Functions istream_iterator” on page 293](#)
- [“Comparison Operations istream_iterator” on page 293](#)

Constructor *istream_iterator*

Default Constructor

Description Constructs the end-of-stream iterator value.

NOTE: The two end-of-stream iterators are always equal.

Prototype `istream_iterator();`

Overloaded and Copy Constructors

Prototype `istream_iterator
(istream& s);`

Remarks Constructs an `istream_iterator<T>` object that reads values from the input stream `s`.

```
istream_iterator  
(const istream_iterator<T,  
    Distance>& x);
```

Remarks Copy constructor.

Destructor

Prototype `~istream_iterator();`

Public Member Functions istream_iterator

Dereferencing Operator *

Description Dereferencing operator. By returning a reference to `const T`, it ensures that it cannot be used to write values to the input stream for which the iterator is constructed.

Prototype `const T& operator*() const;`

Incrementation Operator ++

Description Incrementation operators.

Prototype `istream_iterator<T, Distance>& operator++();`

Remarks This operator reads and stores a value of `T` each time it is called.

Prototype `istream_iterator <T, Distance> operator++(int);`

Remarks This operator reads and stores `x` values of `T` each time it is called.

Comparison Operations istream_iterator

Equality Operator ==

Description Equality operator. Two end-of-stream iterators are always equal. An end-of-stream iterator is not equal to a non-end-of-stream iterator. Two non-end-of-stream iterators are equal when they are constructed from the same stream.

Prototype

```
template
    < class T,
      class Distance>
bool operator==
    (const istream_iterator<T,Distance>& x,
     const istream_iterator<T,Distance>& y);
```

Template class ostream_iterator

Files `#include <iterator.h>`

Description An `ostream_iterator<T>` object writes (using `operator<<`) successive elements onto the output stream for which it was constructed. If it is constructed with `char*` as a constructor argument, then this `delimiter` string is written to the stream after each `T` value is written.

Prototype

```
template
    < class T,
      class charT,
      class traits = char_traits<charT> >
class ostream_iterator : public output_iterator;
```

Remarks It is not possible to read a value of the output iterator. It can only be used to write values out to an output stream for which it is constructed.

The topics in this section are:

- [“Constructor ostream_iterator” on page 294](#)
- [“Public Member Functions ostream_iterator” on page 295](#)

Constructor ostream_iterator

Default Constructor

Description Constructs an iterator that can be used to write to the output stream `s`.

Prototype `ostream_iterator(ostream& s);`

Overloaded and Copy Constructors

Description Constructs an iterator that can be used to write to the output stream *s*. The character string *delimiter* is written out after every value (of type *T*) written to *s*.

Prototype `ostream_iterator(ostream& s, const char* delim);`
`ostream_iterator(const ostream_iterator<T>& x);`

Remarks Copy constructor.

Destructor

Prototype `~ostream_iterator ();`

Public Member Functions ostream_iterator

Dereferencing Operator *

Description Dereferencing operator. An assignment `*o = t` through an output iterator *o* causes *t* to be written to the output stream and the stream pointer advanced in preparation for the next write.

Prototype `ostream_iterator<T>& operator*();`

Assignment Operator =

Description Assignment operator. Replaces the current iterator with a copy of the iterator *x*.

Prototype `ostream_iterator<T>& operator=`
`(const ostream_iterator<T>& x);`

Incrementation Operator ++

Description These operators are present to allow ostream iterators to be used with algorithms that both assign through an output iterator and ad-

Iterators Library

Template class *istreambuf_iterator*

vance the iterator; they actually do nothing, since assignments through the iterator advance the stream pointer also.

Prototype `ostream_iterator <T>& operator++();`
 `ostream_iterator <T> operator++(int x);`

Template class *istreambuf_iterator*

Declaration `template`
 `<class charT,`
 `class traits = char_traits<charT> >`
 `class istreambuf_iterator;`

Description The template class *istreambuf_iterator* reads successive characters from the *streambuf* for which it was constructed. *operator** provides access to the current input character, if any. Each time *operator++* is evaluated, the iterator advanced to the next input character. If the end of stream is reached (*streambuf::sgetc ()* returns *traits::eof()*), the iterator becomes equal to the end of stream iterator value. The default constructor *istreambuf_iterator ()* and the constructor *istreambuf_iterator(0)* both construct an end of stream iterator object suitable for use as an end-of-range.

The result of *operator* ()* on an end of stream is undefined. For any other iterator value a *char_type* is returned. It is impossible to assign a character via an input iterator. In input iterators, *++* operators are not equality preserving, that is, *i == j* does not guarantee at all that *++i == ++j*. Every time *++* is evaluated a new value is used. A practical consequence of this fact is that an *istreambuf_iterator* object can be used only for one-pass algorithms. Two end of stream iterators are always equal. An end of stream iterator is not equal to a non-end of stream iterator.

The topics in this section are:

- [“Typedef Declarations *istreambuf_iterator*” on page 297](#)
- [“Placeholder proxy *istreambuf_iterator*” on page 297](#)
- [“Constructor *istreambuf_iterator*” on page 298](#)

- [“Operators istreambuf_iterator” on page 298](#)

Typedef Declarations `istreambuf_iterator`

Description The following typedef’s are defined in the class `istreambuf_iterator`.

`char_type`

Prototype `typedef charT char_type;`

`traits_type`

Prototype `typedef traits traits_type;`

`streambuf`

Prototype `typedef basic_streambuf<charT, traits>
streambuf;`

`istream`

Prototype `typedef basic_istream<charT,
traits istream;`

Placeholder proxy `istreambuf_iterator`

Class `istreambuf_iterator<charT, traits>::proxy` provides a temporary placeholder as the return value of the post-increment operator (`operator++`). It keeps the character pointed to by the previous value of the iterator for some possible future access to get the character.

Constructor *istreambuf_iterator*

Default Constructor

Description Constructs an end-of-stream iterator.

Prototype `istreambuf_iterator ();`

Overloaded and Copy Constructor

Description

Prototype `istreambuf_iterator
(basic_istream<charT, traits>& s);`

Remarks This constructs the `istream_iterator` pointing to the `basic_streambuf` object `*(s.rdbuf())`.
`istreambuf_iterator (const proxy& p);`

Remarks This constructs the `istreambuf_iterator` pointing to the `basic_streambuf` object related to the proxy object `p`.

Operators *istreambuf_iterator*

Dereferencing Operator *

Description This operator extracts one character pointed to by the `streambuf` `*sbuf_`.

Prototype `charT operator* ();`

Incrementation Operator ++

Description These operators advances the iterator.

Prototype `istreambuf_iterator<charT, traits>&
istreambuf_iterator<charT, traits>::operator++`

() ;

Remarks This operator advances the iterator and returns the result.

Prototype `proxy istreambuf_iterator
<charT,
traits>::operator++
(int);`

Remarks This operator advances the iterator and returns the proxy object keeping the character pointed to by the previous iterator.

istream_iterator::equal

Description This function returns true if and only if both iterators are either at end-of-stream, or are the end-of-stream value, regardless of what streambuf they iterator over.

Prototype `bool equal
(istreambuf_iterator<charT, traits>& b);`

istream_iterator::iterator_category

Description Returns the category of the iterator s.

Prototype `input_iterator_tag iterator_category
(const istreambuf_iterator& s);`

Equality Operator ==

Description `operator==` returns `a.equal (b)`.

Prototype `template
< class charT,
class traits >
bool operator==
(istreambuf_iterator<charT, traits>& a,
istreambuf_iterator<charT, traits>& b);`

Not Equal Operator !=

Description This returns `!a.equal(b)`.

Prototype

```
template
    < class charT,
      class traits >
bool operator!=
    (ostreambuf_iterator<charT,
      traits>& a,
     ostreambuf_iterator<charT, traits>& b);
```

Template class ostreambuf_iterator

Declaration

```
template
    <class charT,
      class traits = char_traits<charT> >
class ostreambuf_iterator;
```

Description The template class `ostreambuf_iterator` writes successive characters onto the output stream from which it was constructed. It is not possible to get a value out of the output iterator. Two output iterators are equal if they are constructed with the same output streambuf.

The topics in this section are:

- [“Typedef Declarations ostreambuf_iterator” on page 300](#)
- [“Constructor ostreambuf_iterator” on page 301](#)
- [“Operators ostreambuf_iterator” on page 302](#)

Typedef Declarations ostreambuf_iterator

Description The following typedef's are defined in the class `ostreambuf_iterator`.

char_type

Prototype `typedef charT char_type;`

traits_type

Prototype `typedef traits traits_type;`

streambuf

Prototype `typedef basic_streambuf<charT, traits>
 streambuf;`

ostream

Prototype `typedef basic_ostream<charT, traits> ostream;`

Constructor ostreambuf_iterator

Default Constructor

Description Constructs an iterator with sbuf_ set to 0.

Prototype `ostreambuf_iterator ();`

Overloaded and Copy Constructors

Prototype `ostreambuf_iterator (ostream& s);`

Remarks This constructs the ostream_iterator pointing to the basic_streambuf object `*(s.rdbuf())`.

Prototype `ostreambuf_iterator (streambuf* s);`

Remarks This constructs the ostreambuf_iterator pointing to the basic_streambuf object `s`.

Operators *ostreambuf_iterator*

Dereferencing Operator *

Description This operator returns `*this`.

Prototype `ostreambuf_iterator<charT, traits>& operator*
();`

ostreambuf_iterator::equal

Description This function returns true if `sbuf_ == b.sbuf_`.

Prototype `bool equal
(ostreambuf_iterator<charT, traits>& b);`

ostreambuf_iterator::iterator_category

Description Returns `output_iterator_tag()`.

Prototype `output_iterator_tag iterator_category
(const ostreambuf_iterator& s);`

Equality Operator ==

Description `operator==` returns `a.equal (b)`.

Prototype `template <class charT, class traits>
bool operator==
(ostreambuf_iterator<charT,
traits>& a,
ostreambuf_iterator<charT,
traits> & b);`

Not Equal Operator !=

Description This returns `!a.equal (b)`.

Prototype `template
 < class charT,
 class traits >
 bool operator!=
 (ostreambuf_iterator<charT,
 traits>& a,
 ostreambuf_iterator<charT,
 traits>& b);`

Template class reverse_bidirectional_iterator

Bidirectional iterators and random access iterators have a corresponding reverse iterator adaptor. These adaptors produce iterators that can be used for traversing through a data structure in the opposite of the normal direction. This section describes reverse_bidirectional_iterator and the following section describes reverse_iterator (for reversing a random access iterator).

The reverse_bidirectional_iterator adaptor takes a bidirectional iterator and produces a new bidirectional iterator for traversal in the opposite direction.

Declaration A template class for reverse bidirectional iterators.

Prototype `template
 < class BidirectionalIterator,
 class T,
 class Reference = T&,
 class Distance = ptrdiff_t >
class reverse_bidirectional_iterator : public
 bidirectional_iterator<T, Distance>`

The topics in this section are:

- [“Constructor reverse_bidirectional_iterator” on page 304](#)
- [“Public Member Functions reverse_bidirectional_iterator” on page 304](#)

Constructor *reverse_bidirectional_iterator*

Default Constructor

Prototype `reverse_bidirectional_iterator();`

Overloaded Constructors

Prototype `explicit reverse_bidirectional_iterator
 (BidirectionalIterator x);`

Remarks This constructor initializes the value of `current` with `x`.

Public Member Functions *reverse_bidirectional_iterator*

reverse_bidirectional_iterator::base

Prototype `BidirectionalIterator base();`

Dereferencing Operator `*`

Prototype `Dereference operator *();`

Incrementation Operator `++`

Prototype `reverse_bidirectional_iterator
 < BidirectionalIterator,
 T,
 Reference,
 Distance >&
operator ++();`

Prototype `reverse_bidirectional_iterator
 < BidirectionalIterator,
 T,
 Reference,`


```
Distance >
operator ++(int);
```

Decrementation Operator --

Prototype `reverse_bidirectional_iterator`
 `< BidirectionalIterator,`
 `T,`
 `Reference,`
 `Distance>&`
`operator --();`

Prototype `reverse_bidirectional_iterator`
 `<BidirectionalIterator,`
 `T,`
 `Reference,`
 `Distance>`
`operator --(int);`

Equality Operator ==

Description Return a true if the `reverse_bidirectional_iterator` `x` is
 equal to `reverse_bidirectional_iterator` `y`.

Prototype

```
template
< class BidirectionalIterator, class T, class Distance >
bool operator ==
(const reverse_bidirectional_iterator
  <BidirectionalIterator, T, Reference, Distance >& x,
const reverse_bidirectional_iterator
  <BidirectionalIterator, T, Reference, Distance>& y);
```

Template class reverse_iterator

The `reverse_iterator` adaptor takes a random access iterator and produces a new random access iterator for traversal in the opposite direction.

Description A template class for reverse iterators.

Prototype

```
template
    < class RandomAccessIterator,
      class T,
      class Reference = T&,
      class Distance = ptrdiff_t >
class reverse_iterator : public
    random_access_iterator <T, Distance>;
```

The topics in this section are:

- [“Constructor reverse_iterator” on page 306](#)
- [“Public Member Functions reverse_iterator” on page 307](#)

Constructor reverse_iterator

Default Constructor

Prototype `reverse_iterator();`

Overloaded and Copy Constructors

Description This constructor initializes the value of current with `x`.

Prototype

```
explicit reverse_iterator
    (RandomAccessIterator x);
```

Public Member Functions reverse_iterator

reverse_iterator::base

Prototype `RandomAccessIterator base();`

Dereferencing Operator *

Prototype `Reference operator *();`

Incrementation Operator ++

Prototype `reverse_iterator
 < RandomAccessIterator,
 T,
 Reference,
 Distance >&
operator ++();`

Prototype `reverse_iterator
 < RandomAccessIterator,
 T,
 Reference,
 Distance >
operator ++(int);`

Decrementation Operator --

Prototype `reverse_iterator
 < RandomAccessIterator,
 T,
 Reference,
 Distance>&
operator --();`

Prototype `reverse_iterator<RandomAccessIterator, T,
 Reference, Distance> operator --(int);`

Add Operator +

Prototype `reverse_iterator`
 `< RandomAccessIterator,`
 `T,`
 `Reference,`
 `Distance >`
`operator+`
 `(Distance n) const;`

Add & Assign Operator +=

Prototype `reverse_iterator`
 `< RandomAccessIterator,`
 `T,`
 `Reference,`
 `Distance >&`
`operator +=`
 `(Distance n);`

Minus Operator -

Prototype `reverse_iterator`
 `< RandomAccessIterator,`
 `T,`
 `Reference,`
 `Distance >`
`operator -`
 `(Distance n) const;`

Minus & Assign Operator -=

Prototype `reverse_iterator`
 `< RandomAccessIterator,`
 `T,`
 `Reference,`
 `Distance >&`
`operator -=`

(Distance n);

Subset Operator []

Prototype Reference operator[] (Distance n);

Equality Operator ==

Prototype

```
template
< class RandomAccessIterator,
  class T,
  class Reference,
  class Distance >
bool operator ==
(const reverse_iterator
  <RandomAccessIterator, T, Reference, Distance>& x,
const reverse_iterator
  < RandomAccessIterator, T, Reference, Distance>& y);
```

Less Than Operator <

Prototype

```
template
< class RandomAccessIterator,
  class T, class Reference,
  class Distance >
bool operator <
(const reverse_iterator
  < RandomAccessIterator,T,Reference, Distance >& x,
const reverse_iterator
  < RandomAccessIterator,T,Reference, Distance >& y);
```

Minus Operator -

Prototype

```
template
< class RandomAccessIterator,
  class T,
  class Reference,
  class Distance >
Distance operator -
(const reverse_iterator
  < RandomAccessIterator, T, Reference, Distance >& x,
  const reverse_iterator
  < RandomAccessIterator, T, Reference, Distance >& y);
```

Add Operator +

Prototype

```
template
< class RandomAccessIterator,
  class T,
  class Reference,
  class Distance >
Reverse_iterator< RandomAccessIterator,T, Reference, Distance >
operator +
(Distance n,
  const reverse_iterator
  <RandomAccessIterator,T,Reference,Distance>& x);
```

Template class back_insert_iterator

Insert iterators are provided in the STL to deal with the problem of insertion similar to writing in an array. These iterator are iterator adaptors. There are three types of insert iterator adaptors. They are `back_insert_iterator`, `front_insert_iterator` and `insert_iterator`. In this section, we shall study the interface of these three iterator adaptors.

Description A template class for back insert iterators.

Prototype

```
template
    < class Container >
class back_insert_iterator : public
output_iterator;
```

The topics in this section are:

- [“Constructor back insert iterator” on page 311](#)
- [“Public Member Functions back insert iterator” on page 311](#)

Constructor back_insert_iterator

Copy Constructor

Prototype

```
explicit back_insert_iterator(Container& x);
```

Public Member Functions back_insert_iterator

Assignment Operator =

Prototype

```
back_insert_iterator
    < Containe r>&
operator=
    (const typename
    Container::value_type& value);
```

Dereferencing Operator *

Prototype

```
back_insert_iterator<Container>& operator*();
```

Incrementation Operator ++

Prototype

```
back_insert_iterator<Container>& operator++();
```

Prototype

```
back_insert_iterator<Container> operator++(int);
```

`back_insert_iterator::back_inserter`

Prototype `template
 < class Container >
back_insert_iterator
 < Container > back_inserter(Container& x);`

Template class `front_insert_iterator`

Description A template class for front insert iterators.

Prototype `template
 < class Container >
class front_insert_iterator : public
 output_iterator;`

The topics in this section are:

- [“Constructor `front_insert_iterator`” on page 312](#)
- [“Public Member Functions `front_insert_iterator`” on page 312](#)

Constructor `front_insert_iterator`

Copy Constructor

Prototype `explicit front_insert_iterator(Container& x);`

Public Member Functions `front_insert_iterator`

Assignment Operator =

Prototype `front_insert_iterator<Container>& operator =
 (const typename Container::value_type& value);`

Dereferencing Operator *

Prototype `front_insert_iterator<Container>& operator*();`

Incrementation Operator ++

Prototype `front_insert_iterator<Container>& operator++();`

Prototype `front_insert_iterator
 < Container >
operator++(int);`

front_insert_iterator::front_inserter

Prototype `template
 < class Container >
front_insert_iterator< Container >
 front_inserter(Container& x);`

Template class insert_iterator

Description A template class for insert iterators

Prototype `template
 < class Container >
class insert_iterator : public output_iterator;`

The topics in this section are:

- [“Constructor insert_iterator” on page 313](#)
- [“Public Member Function insert_iterator” on page 314](#)

Constructor insert_iterator

Copy Constructor

Prototype `insert_iterator
 (Container& x,
 typename Container::iterator i);`

Public Member Function insert_iterator

Assignment Operator =

Prototype `insert_iterator< Container >& operator =
 (const typename
 Container::value_type& value);`

Dereferencing Operator *

Prototype `insert_iterator< Container >& operator*();`

Incrementation Operator ++

Prototype `insert_iterator< Container >& operator++();`

Prototype `insert_iterator<Container> operator++(int);`

insert_iterator::inserter

Prototype `template
 < class Container,
 class Iterator >
insert_iterator< Container > inserter
 (Container& x, Iterator i);`



Algorithms Library

This chapter discusses the algorithms library. These algorithms cover sequences, sorting, and numerics.

Overview of the Algorithms Library

The algorithms library contains 32 distinct algorithms, divided into four main categories:

- [“Non Mutating Sequence Algorithms” on page 317](#)
- [“Mutating Sequence Algorithms” on page 324](#)
- [“Sorting and Related Algorithms” on page 341](#)
- [“Generalized Numeric Algorithms” on page 366](#)

All of the library algorithms are generic, in the sense that they can operate on a variety of data structures. The algorithms are not directly parameterized in terms of data structures. Instead, they are parameterized by iterator types. This allows the algorithms to work with user-defined data structures, as long as these data structures have iterator types satisfying the assumptions of the algorithms.

The remaining topics in this overview are:

- [“In-place and Copying Versions” on page 315](#)
- [“Algorithms with Predicate Parameters” on page 316](#)
- [“Binary Predicates” on page 316](#)

In-place and Copying Versions

Both in-place and copying versions are provided for certain algorithms. The decision whether to include a copying version is based on complexity considerations.

Algorithms Library

Overview of the Algorithms Library

For example, `sort_copy` is not provided, since the cost of sorting is much more significant, and users might as well do `copy` followed by `sort`. On the other hand,

`replace_copy` is provided, since the cost of copying is greater than the cost of replacing a value in a container.

Whenever, a copying version is provided for algorithm, it is called `algorithm_copy`.

Finally, algorithms that take predicates end with the suffix `_if` (which follows the suffix `_copy`, for copying algorithms).

Algorithms with Predicate Parameters

Several algorithms accept function objects as parameters. These function objects are applied to the result of dereferencing the iterators accepted by the algorithm, with the requirement that the resulting value be testable as `true`. A unary `Predicate` class is used in the definition of such algorithms.

In other words, if an algorithm takes a `Predicate pred` as its argument, and `first` as its iterator argument, it should work correctly in the construct

```
if (pred(*first)) {.....}
```

The function object `pred` is assumed not to apply any non-constant function through the dereferenced iterator.

Binary Predicates

A `BinaryPredicate` class is used whenever an algorithm expects a function object that when applied to the result of dereferencing two corresponding iterators or to dereferencing an iterator and a type `T` (where `T` is part of the function signature), returns a value testable as `true`.

In other words, if an algorithm takes
`BinaryPredicate binary_pred`

as its argument and `first1` and `first2` as its iterator arguments, it should work correctly in the construct

```
if (binary_pred(*first1, *first2)) {...}
```

`BinaryPredicate` always takes the first iterator type as its first argument. That is, in those cases when `T` value is part of the function signature, `binary_pred` can be used as follows:

```
if (binary_pred(first, value)) {...}
```

It is expected that `binary_pred` will not apply any non-constant function through the dereferenced iterators.

Non Mutating Sequence Algorithms

Non-mutating sequence algorithms are those that do not directly modify the containers they operate on. The algorithms in this category are:

- [“for_each” on page 317](#)
- [“find_if” on page 318](#)
- [“adjacent_find” on page 319](#)
- [“count” on page 320](#)
- [“count_if” on page 320](#)
- [“mismatch” on page 321](#)
- [“equal” on page 322](#)
- [“search” on page 323](#)

Each of these algorithms, except `for_each`, has two versions: a “normal” version, which uses `operator<` or `operator==` for comparisons, and a “predicate” version, which uses an appropriate function object for comparisons.

for_each

Description	The <code>for_each</code> algorithm applies a specified function to each element of the input container.
--------------------	----------------------------------------------------------------------------------------------------------

Algorithms Library

Non Mutating Sequence Algorithms

Prototype `template
 < class InputIterator,
 class Function >
Function for_each
 (InputIterator first,
 InputIterator last,
 Function f);`

Remarks The function `f` is applied to the result of dereferencing every iterator in the range `[first, last)`. It is assumed that the function `f` does not apply any non-constant function through the dereferenced iterator. `f` is applied exactly `last-first` times. If `f` returns a result, the result is ignored.

Complexity Time complexity is linear. If `n` is the size of `[first, last)`, then exactly `n` applications of `f` are made. Space complexity is constant.

find

Description The first version of the algorithm traverses the iterators `(first, last]` and returns the first iterator `i` such that `*i == value`. In either case, if such an iterator is not found then the iterator `last` is returned.

Prototype `template
 < class InputIterator,
 class T >
InputIterator find
 (InputIterator first,
 InputIterator last,
 const T& value);`

find_if

Description The second version returns the first iterator `i` such that `pred(*i) == true`. In either case, if such an iterator is not found then the iterator `last` is returned.

Prototype `template`

```
    < class InputIterator,
      class Predicate >
InputIterator find_if
    (InputIterator first,
     InputIterator last,
     Predicate pred);
```

Complexity Time complexity is linear. The number of applications of operator `!=` (or `pred`, for `find_if`) is the size of the range `[first, last)`. Space complexity is constant.

adjacent_find

Description The `adjacent_find` algorithm returns an iterator `i` referring to the first consecutive duplicate element in the range `[first, last)`, or `last` if there is no such element. By consecutive duplicate, it is meant that an element is equal to the element immediately following it in the range.

Prototype

```
template
    < class InputIterator >
InputIterator adjacent_find
    (InputIterator first,
     InputIterator last);
template
    < class InputIterator,
      class BinaryPredicate >
InputIterator adjacent_find
    (InputIterator first,
     InputIterator last,
     BinaryPredicate binary_pred);
```

Remarks Comparisons are done using `operator==` in the first version of the algorithm and a function object `binary_pred` in the second version.

Complexity Time complexity is linear. The number of comparisons done is the size of the range `[first, i)`. Space complexity is constant.

count

Description The `count` algorithm adds the number of elements in the range `[first, last)` that are equal to `value`, and places the result into the reference argument `n`.

Prototype

```
template
    < class InputIterator,
      class T,
      class Size >
void count(InputIterator first,
           InputIterator last,
           const T& value,
           Size& n);
```

Remarks `Count` must store the result into a reference argument since it cannot deduce the size type from the built-in iterator types, such as `int*`.

Complexity Time complexity is linear. The number of equality operations performed, or of applications of the predicate `pred`, is the size of the range `[first, last)`. Space complexity is constant.

count_if

Description The `count_if` algorithm adds to `n` the number of iterators in the range `[first, last)` for which the condition `pred(*i) == true` is satisfied.

Prototype

```
template
    < class InputIterator,
      class Predicate,
      class Size >
void count_if
    (InputIterator first,
     InputIterator last,
     Predicate pred,
     Size& n);
```


Complexity Time complexity is linear. The number of equality operations performed, or of applications of the predicate `pred`, is the size of the range `[first, last)`. Space complexity is constant.

mismatch

Description `mismatch` compares corresponding pairs of elements from two ranges, and returns the first mismatched pair.

Prototype

```
template
    < class InputIterator1,
      class InputIterator2 >
pair<InputIterator1, InputIterator2> mismatch
    (InputIterator1 first1,
     InputIterator1 last1,
     InputIterator2 first2);

template
    < class InputIterator1,
      class InputIterator2,
      class BinaryPredicate >
pair<InputIterator1, InputIterator2> mismatch
    (InputIterator1 first1,
     InputIterator1 last1,
     InputIterator2 first2,
     BinaryPredicate binary_pred);
```

Remarks The algorithm finds the first position at which the values in the range `[first1, last1)` disagree with the values in the range starting at `first2`. It returns a pair of iterators `i` and `j` which satisfy the following conditions:

- `i` points into the range `[first, last)`
- `j` points into the range beginning at `first2`
- `i` and `j` are both equidistant from the beginning of their corresponding ranges.
- `*i != *j`, or `binary_pred(i, j) == false`, depending on the version of `mismatch` invoked. In the first version, checks for equality are made with `operator==` and in the

Algorithms Library

Non Mutating Sequence Algorithms

second version they are made with the function object `binary_pred`.

Complexity Time complexity is linear. The number of equality operations or applications of the binary predicate is the size of the range `[first1, i)`.

equal

Description `equal` returns true if the ranges `[first1, last1)` and the range of size `first1-last1` beginning at `first2` contain the same elements in the same order, false otherwise.

Prototype

```
template
    < class InputIterator1,
      class InputIterator2 >
bool equal
    (InputIterator1 first1,
     InputIterator1 last1,
     InputIterator2 first2);
template
    < class InputIterator1,
      class InputIterator2,
      class BinaryPredicate >
bool equal
    (InputIterator1 first1,
     InputIterator1 last1,
     InputIterator2 first2,
     BinaryPredicate binary_pred);
```

Remarks In the first version of `equal`, checks for element equality are made with `operator==`, and in the second version they are made with the function object `pred`.

Complexity Time complexity is linear. The number of equality operations is the size of the range `[first1, i)`, where `i` is the iterator referring to the first element in the range `[first1, last1)` that does not match the corresponding element in the range beginning at `first2`. Space complexity is constant.

search

Description `search` checks whether the second range `[first2, last2)` is contained in the first range `[first1, last1)`. If so, the iterator `i` in `[first1, last1)` that represents the start of the second range in the first is returned. Otherwise, the past-the-end location of the first range, (i.e. `last1`), is returned.

Prototype

```
template
    < class ForwardIterator1,
      class ForwardIterator2 >
void search
    (ForwardIterator1 first1,
     ForwardIterator1 last1,
     ForwardIterator2 first2,
     ForwardIterator2 last2);

template
    < class ForwardIterator1,
      class ForwardIterator2,
      class BinaryPredicate >
void search
    (ForwardIterator1 first1,
     ForwardIterator1 last1,
     ForwardIterator2 first2,
     ForwardIterator2 last2,
     BinaryPredicate binary_pred);
```

Remarks In the first version of the algorithm, checks for element equality are made with `operator==`, while in the second they are made with the function object `binary_pred`.

Complexity Time complexity is quadratic. If `n` is the size of the range `[first1, last1)`, and `m` is the size of the range `[first2, last2)`, then the number of applications of `operator!=` or `binary_pred` is $(n-m)*m$, which is less than or equal to $n*n/4$. If $m > n$, the time taken is 0 (i.e. no match can be found in this case).

The implementation does not use the Knuth-Morris-Pratt algorithm. The KMP algorithm guarantees linear time, but tends to be slower in most practical cases than the naive algorithm with worst case

quadratic behavior. The worst case is extremely unlikely. Space complexity is constant.

Mutating Sequence Algorithms

Mutating sequence algorithms typically modify the containers they operate on. The algorithms in this category are:

- ["copy" on page 325](#)
- ["copy backward" on page 325](#)
- ["fill" on page 330](#)
- ["fill n" on page 331](#)
- ["generate" on page 331](#)
- ["generate n" on page 332](#)
- ["iter swap" on page 326](#)
- ["partition" on page 339](#)
- ["random shuffle" on page 339](#)
- ["remove" on page 332](#)
- ["remove copy" on page 334](#)
- ["remove copy if" on page 334](#)
- ["remove if" on page 333](#)
- ["replace" on page 328](#)
- ["replace copy" on page 329](#)
- ["replace copy if" on page 330](#)
- ["replace if" on page 329](#)
- ["reverse" on page 337](#)
- ["reverse copy" on page 337](#)
- ["rotate" on page 337](#)
- ["rotate copy" on page 338](#)
- ["stable partition" on page 340](#)
- ["swap" on page 326](#)
- ["swap ranges" on page 327](#)

- [“transform” on page 327](#)
- [“unique” on page 335](#)
- [“unique_copy” on page 336](#)

copy

Description `copy` copies elements from the sequence `[first, last)` to the sequence of size `last - first` beginning at the iterator `result`, and returns the past-the-end iterator, `result + last - first`. For each non-negative integer $n < (last - first)$, the operation `*(result + n) = *(first + n)` is performed. The result of `copy` is undefined if `result` is in the range `[first, last)`.

Prototype

```
template
    < class InputIterator,
      class OutputIterator >
void copy
    (InputIterator first,
     InputIterator last,
     OutputIterator result);
```

Complexity Time complexity is linear for both copy algorithms. At most n assignments are performed, where n is the size of the range `[first, last)`. Space complexity is constant.

copy_backward

Description `copy_backward` copies all of the values in the range `[first, last)` to the range of size `last - first` starting at the iterator `result`, and returns the iterator that contains the last element copied (i.e. the beginning of the sequence). For `copy_backward`, the source and destination ranges may overlap if `result >= last`.

Prototype

```
template
    < class BidirectionalIterator1,
      class BidirectionalIterator2 >
void copy_backward
    (InputIterator first,
```

```
InputIterator last,  
OutputIterator result);
```

Complexity Time complexity is linear for both copy algorithms. At most n assignments are performed, where n is the size of the range `[first, last)`. Space complexity is constant.

swap

Description The `swap` function exchanges two elements.

Prototype

```
template  
    < class T >  
void swap  
    (T &a,  
     T &b);
```

Complexity The time required is constant for `swap` and `iter_swap`. For `swap_ranges`, the time complexity is linear. The number of swaps performed is the size of the range `[first, last)`. Space complexity is constant for all the swap algorithms.

iter_swap

Description The function `iter_swap` exchanges values pointed to by two iterators.

Prototype

```
template  
    < class ForwardIterator1,  
      ForwardIterator2 >  
void iter_swap  
    (ForwardIterator1 a,  
     ForwardIterator2 b);
```

Complexity The time required is constant for `swap` and `iter_swap`. For `swap_ranges`, the time complexity is linear. The number of swaps performed is the size of the range `[first, last)`. Space complexity is constant for all the swap algorithms.

swap_ranges

Description The function `swap_ranges` exchanges the elements in the range `[first, last)` with those in the range of size `last - first` beginning at `first2`. `swap_ranges` returns the past-the-end iterator, `first2+last-first`.

Prototype

```
template
    < class ForwardIterator1,
      class ForwardIterator2 >
ForwardIterator swap_ranges
    (ForwardIterator1 first1,
     ForwardIterator1 last1,
     ForwardIterator2 first2);
```

Complexity The time required is constant for `swap` and `iter_swap`. For `swap_ranges`, the time complexity is linear. The number of swaps performed is the size of the range `[first, last)`. Space complexity is constant for all the swap algorithms.

transform

Description The first version of `transform` generates an output sequence of elements by applying a unary function `op` to each element of the input sequence `[first, last)`.

The second version of `transform` accepts the input sequence `[first1, last1)` and the sequence of length `last1 - first1` starting at `first2`, and generates an output by applying a binary operation `binary_op` to each corresponding pair of elements from the input sequences.

Prototype

```
template
    < class InputIterator,
      class OutputIterator,
      class UnaryOperation >
OutputIterator transform
    (InputIterator first,
     InputIterator last,
```

Algorithms Library

Mutating Sequence Algorithms

```
        OutputIterator result,  
        UnaryOperation op);  
template  
    < class InputIterator1,  
      class InputIterator2,  
      class OutputIterator,  
      class BinaryOperation >  
OutputIterator transform  
    (InputIterator1 first1,  
     InputIterator1 last1,  
     InputIterator2 first2,  
     OutputIterator result,  
     BinaryOperation binary_op);
```

Remarks For both versions of transform, the resulting sequence is placed starting at the position result, and the past-the-end iterator is returned.

Complexity Time complexity is linear. The number of applications of op is the size of the range [first, last) and the number of applications of binary_op is the size of the range [first1, last1). Space complexity is constant.

replace

Description The replace algorithm modifies the range [first, last) so that all elements equal to old_value are replaced by new_value, while other values remain unchanged.

Prototype

```
template  
    < class ForwardIterator,  
      class T >  
void replace  
    (ForwardIterator first,  
     ForwardIterator last,  
     const T& old_value,  
     const T& new_value);
```


Complexity Time complexity is linear for all replace algorithms. The number of equality operations performed, or of applications of the predicate `pred`, is the size of the range `[first, last)`. Space complexity is constant.

replace_if

Description `replace_if` modifies the range `[first, last)` so that all elements that satisfy the predicate `pred` are replaced by `new_value`, while other values remain unchanged.

Prototype

```
template
    < class ForwardIterator,
      class Predicate, class T >
void replace_if
    (ForwardIterator first,
     ForwardIterator last,
     Predicate pred,
     const T& new_value);
```

Complexity Time complexity is linear for all replace algorithms. The number of equality operations performed, or of applications of the predicate `pred`, is the size of the range `[first, last)`. Space complexity is constant.

replace_copy

Description `replace_copy` is similar to `replace`, except that the original sequence is not modified. Rather, the altered sequence is placed in the range of size `last - first` beginning at `result`.

Prototype

```
template
    < class InputIterator,
      class OutputIterator,
      class T >
OutputIterator replace_copy
    (InputIterator first,
     InputIterator last,
     OutputIterator result,
```

```
const T& old_value,  
const T& new_value);
```

Complexity Time complexity is linear for all replace algorithms. The number of equality operations performed, or of applications of the predicate `pred`, is the size of the range `[first, last)`. Space complexity is constant.

replace_copy_if

Description `replace_copy_if` is similar to `replace_if`, except that the original sequence is not modified. Rather, the altered sequence is placed in the range of size `last - first` beginning at `result`.

Prototype

```
template  
    < class InputIterator,  
      class OutputIterator,  
      class Predicate,  
      class T >  
OutputIterator replace_copy_if  
    (InputIterator first,  
     InputIterator last,  
     OutputIterator result,  
     Predicate pred,  
     const T& new_value);
```

Complexity Time complexity is linear for all replace algorithms. The number of equality operations performed, or of applications of the predicate `pred`, is the size of the range `[first, last)`. Space complexity is constant.

fill

Description `fill` assigns `value` through all the iterators in the range `[first, last)`.

Prototype

```
template  
    < class ForwardIterator,  
      class T >
```

```
void fill
    (ForwardIterator first,
     ForwardIterator
     last,
     const T& value);
```

Complexity Time complexity is linear. The number of assignments for both versions of the algorithm is the size of the range `[first, last)`. Space complexity is constant.

fill_n

Description `fill_n` assigns value through all the iterators in the range `[first, first + n)`.

Prototype

```
template
    < class ForwardIterator,
      class Size,
      class T >
void fill_n
    (ForwardIterator first,
     Size n,
     const T& value);
```

Complexity Time complexity is linear. The number of assignments for both versions of the algorithm is the size of the range `[first, last)`. Space complexity is constant.

generate

Description `generate` fills the range `[first, last)` with the sequence generated by `last - first` successive calls to the function object `gen`.

Prototype

```
template
    < class ForwardIterator,
      class Generator >
void generate
    (ForwardIterator first,
     ForwardIterator last,
```

```
Generator gen);
```

Complexity Time complexity is linear. The number of assignments for `generate` is the size of the range `[first, last)`, while for `generate_n` the number of assignments is `n`. Space complexity is constant.

generate_n

Description `generate_n` fills the range of size `n` beginning at `first` with the sequence generated by `n` successive calls to `gen`.

Prototype

```
template
    < class ForwardIterator,
      class Size,
      class Generator >
void generate_n
    (ForwardIterator first,
     Size n,
     Generator gen);
```

Complexity Time complexity is linear. The number of assignments for `generate` is the size of the range `[first, last)`, while for `generate_n` the number of assignments is `n`. Space complexity is constant.

remove

Description The function `remove` removes those elements from the range `[first, last)` that are equal to `value`, and returns the location `i` that is the past-the-end iterator for the resulting range of values that are not equal to `value`.

Prototype

```
template
    < class ForwardIterator,
      class T >
ForwardIterator remove
    (ForwardIterator first,
     ForwardIterator last,
     const T& value);
```

It is important to note that neither `remove` nor `remove_if` alters the size of the original container: the algorithms operate by copying (with assignments) the final generated elements into the range `[first, i)`. No calls are made to the `insert` or `erase` member functions of the containers operated on by the algorithms.

All versions of `remove` are stable, that is, the relative order of the elements that are not removed is the same as their relative order in the original range.

Complexity Time complexity is linear for all versions of `remove`. The number of assignments is the number of elements not removed, at most the size of the range `[first, last)`. Space complexity is constant for all the `remove` algorithms.

`remove_if`

Description The function `remove_if` removes those elements from the range `[first, last)` which satisfy the predicate `pred`, and returns the location `i` that is the past-the-end iterator for the resulting range of values that are not equal to `value`.

Prototype

```
template
    < class ForwardIterator,
      class Predicate >
ForwardIterator remove_if
    (ForwardIterator first,
     ForwardIterator last,
     Predicate pred);
```

Remarks It is important to note that neither `remove` nor `remove_if` alters the size of the original container: the algorithms operate by copying (with assignments) the final generated elements into the range `[first, i)`. No calls are made to the `insert` or `erase` member functions of the containers operated on by the algorithms.

All versions of `remove` are stable, that is, the relative order of the elements that are not removed is the same as their relative order in the original range.

Complexity Time complexity is linear for all versions of `remove`. The number of assignments is the number of elements not removed, at most the size of the range `[first, last)`. Space complexity is constant for all the `remove` algorithms.

remove_copy

Description `remove_copy` is similar to `remove`, except that the final resulting sequences are copied into the range beginning at `result`.

Prototype

```
template
    < class InputIterator,
      class OutputIterator,
      class T >
OutputIterator remove_copy
    (InputIterator first,
     InputIterator last,
     OutputIterator result,
     const T& value);
```

Remarks All versions of `remove` are stable, that is, the relative order of the elements that are not removed is the same as their relative order in the original range.

Complexity Time complexity is linear for all versions of `remove`. The number of assignments is the number of elements not removed, at most the size of the range `[first, last)`. Space complexity is constant for all the `remove` algorithms.

remove_copy_if

Description `remove_copy_if` is similar to `remove_if`, except that the final resulting sequences are copied into the range beginning at `result`.

Prototype

```
template
    < class InputIterator,
      class OutputIterator,
      class Predicate >
OutputIterator remove_copy_if
```

```
(InputIterator first,  
InputIterator last,  
OutputIterator result,  
Predicate pred);
```

Remarks All versions of `remove` are stable, that is, the relative order of the elements that are not removed is the same as their relative order in the original range.

Complexity Time complexity is linear for all versions of `remove`. The number of assignments is the number of elements not removed, at most the size of the range `[first, last)`. Space complexity is constant for all the `remove` algorithms.

unique

Description `unique` eliminates consecutive duplicates from the range `[first, last)`. An element is considered to be a consecutive duplicate if it is equal to an element in the location to its immediate right in the range.

In the first version of `unique`, checks for equality are made using `operator==`, while in the second they are made with the function object `binary_pred`.

Prototype

```
template  
    < class ForwardIterator >  
ForwardIterator unique  
    (ForwardIterator first,  
     ForwardIterator last);
```

Prototype

```
template  
    < class ForwardIterator,  
      class BinaryPredicate >  
ForwardIterator unique  
    (ForwardIterator first,  
     ForwardIterator last,  
     BinaryPredicate binary_pred);
```

Remarks All versions of `unique` return the end of the resulting range.

The `unique` algorithms are typically applied to a sorted range, since in this case all duplicates are consecutive duplicates.

Complexity Time complexity is linear for all versions of `unique`. Exactly `last - first` applications of the corresponding predicates are done. Space complexity is constant for `unique`, and linear for `unique_copy`.

`unique_copy`

Description `unique_copy` is similar to `unique`, except that the resulting sequence is copied into the range starting at `result`.

Prototype

```
template
    < class InputIterator,
      class OutputIterator >
OutputIterator unique_copy
    (InputIterator first,
     InputIterator last,
     OutputIterator result);

template
    < class ForwardIterator,
      class BinaryPredicate >
OutputIterator unique_copy
    (InputIterator first,
     InputIterator last,
     OutputIterator result,
     BinaryPredicate binary_pred);
```

Remarks All versions of `unique` return the end of the resulting range.

The `unique` algorithms are typically applied to a sorted range, since in this case all duplicates are consecutive duplicates.

Complexity Time complexity is linear for all versions of `unique`. Exactly `last - first` applications of the corresponding predicates are done. Space complexity is constant for `unique`, and linear for `unique_copy`.

reverse

Description The reverse algorithm reverses the order of elements in the range `[first, last)`.

Prototype

```
template
    < class BidirectionalIterator >
void reverse
    (BidirectionalIterator first,
     BidirectionalIterator last);
```

Complexity Time complexity is linear. For reverse, exactly $[n/2]$ element exchanges are performed, where n is the size `[first, last)`. For `reverse_copy`, exactly `last-first` assignments are done. Space complexity is constant.

reverse_copy

Description The `reverse_copy` algorithm reverses the sequence `[first, last)` and copies the resulting sequence into the range beginning `result`.

Prototype

```
template
    < class BidirectionalIterator,
      class OutputIterator >
OutputIterator reverse_copy
    (BidirectionalIterator first,
     BidirectionalIterator last,
     OutputIterator result);
```

Complexity Time complexity is linear. For reverse, exactly $[n/2]$ element exchanges are performed, where n is the size `[first, last)`. For `reverse_copy`, exactly `last-first` assignments are done. Space complexity is constant.

rotate

Description The rotate algorithm shifts elements in a sequence leftward as follows: for each non-negative integer $i < (\text{last} - \text{first})$, rotate

places the element from the position `first + i` into position `first + (i + (middle - first)) % (last - first)`.

After the `rotate` operation, an element originally at location `i` in the sequence `[first, last)` is finally placed at location `(i+n-m) mod n`, where `m` is the size of the range `[first, middle)`, and `n` is the size of the range `[first, last)`.

Prototype

```
template
    < class ForwardIterator >
void rotate
    (ForwardIterator first,
     ForwardIterator middle,
     ForwardIterator last);
```

Complexity Time complexity is linear.

For `rotate`, exactly $2 * [n/2] + [m/2] + [(n-m)/2]$ element exchanges are performed, where `m` is the size of the range `[first, middle)` and `n` is the size `[first, last)`. Space complexity is constant.

rotate_copy

Description `rotate_copy` is similar to `rotate`, except that it copies the elements of the resulting sequence into a range of size `last-first` starting at the location `result`.

Prototype

```
template
    < class ForwardIterator,
      class OutputIterator >
void rotate_copy
    (ForwardIterator first,
     ForwardIterator middle,
     ForwardIterator last,
     OutputIterator result);
```

Complexity Time complexity is linear.

For `rotate_copy`, exactly `n` assignment operations are performed, where `n` is the size `[first, last)`. Space complexity is constant.

random_shuffle

Description `random_shuffle` shuffles the elements in the range `[first, last)` with uniform distribution. `random_shuffle` can take a particular random number generating function object `rand` such that `rand` returns a randomly chosen double in the interval `[0, 1)`.

Prototype

```
template
    < class RandomAccessIterator >
void random_shuffle
    (RandomAccessIterator first,
     RandomAccessIterator last);
```

Prototype

```
template
    < class RandomAccessIterator,
      class RandomNumberGenerator >
void random_shuffle
    (RandomAccessIterator first,
     RandomAccessIterator last,
     RandomNumberGenerator& rand);
```

Complexity Time complexity is linear. The algorithm performs exactly `(last - first) - 1` swaps. Space complexity is constant.

partition

Description The `partition` algorithm places all elements in the range `[first, last)` that satisfy `pred` before all elements that do not satisfy it.

Prototype

```
template
    < class BidirectionalIterator,
      class Predicate >
void partition
    (BidirectionalIterator first,
     BidirectionalIterator last,
     Predicate pred);
```

Remarks Both algorithms return an iterator `i` such that for any iterator `j` in the range `[first, i)`, `pred(*j) == true`, and for any iterator `k` in the range `[i, last)`, `pred(*k) == false`.

For `partition`, exactly $\lfloor n/2 \rfloor$ element exchanges are performed, where n is the size `[first, last)`. Exactly `last - first` applications of the predicate are performed.

Complexity Time complexity is linear for both versions of the algorithm. Space complexity is constant.

stable_partition

Prototype

```
template
    < class BidirectionalIterator,
      class Predicate >
void stable_partition
    (BidirectionalIterator first,
     BidirectionalIterator last,
     Predicate pred);
```

Description In `stable_partition`, the relative positions of the elements in both groups are preserved. `partition` does not guarantee this.

Remarks Both algorithms return an iterator `i` such that for any iterator `j` in the range `[first, i)`, `pred(*j) == true`, and for any iterator `k` in the range `[i, last)`, `pred(*k) == false`.

If the available memory for a buffer is smaller than the range `[first, last)`, the `stable_partition` function requires $O(n \log n)$ time and performs $n \log n$ swaps, where n is the size of `[first, last)`. If there is enough available memory for the buffer to contain all the elements in the range `[first, last)`, then the `stable_partition` function requires linear time, performing $n + m$ assignment operations and applying the predicate exactly n times, where m is the size of the range `[i, last)` and n is the size of `[first, last)`.

Complexity Time complexity is linear for both versions of the algorithm. For `stable_partition`, the time and space complexity varies with the available memory.

Sorting and Related Algorithms

There are several distinct sets of algorithms related to sorting. Each individual set contains a collection of related algorithms. The sets are:

- [“Sorting” on page 342](#)
- [“Binary Searching” on page 346](#)
- [“Merging” on page 350](#)
- [“Set Operations on Sorted Structures” on page 352](#)
- [“Heap Operations” on page 357](#)
- [“Finding Min and Max” on page 361](#)
- [“Lexicographical Comparison” on page 364](#)
- [“Permutation Generators” on page 365](#)

All of the algorithms have two versions: one that uses `operator<` for comparisons and another that uses a function object of type `Compare`.

`Compare` is used as a function object which accepts two arguments, returns `true` if the first argument is less than the second, and returns `false` otherwise. `Compare comp` is used throughout for algorithms assuming an ordering relation. It is assumed that `comp` will not apply any non-constant function through the dereferenced iterator.

For all algorithms that take `Compare`, there is a version that uses `operator<` instead. That is, `comp(*i, *j) == true` defaults to `*i < *j == true`. For the algorithms to work correctly, `comp` has to induce a total ordering on the values.

A sequence is sorted with respect to a comparator `comp` if for any iterator `i` pointing to the sequence and any non-negative integer `n`

such that `i+n` is a valid iterator pointing to an element of the sequence, `comp(*(i+n), *i) == false`.

In the descriptions of the functions that deal with ordering relationships, we frequently use a notion of equality to describe concepts such as stability. The equality to which we refer is not necessarily an `operator==`, but an equality relation induced by the total ordering. That is, two elements `a` and `b` are considered equal if and only if `!(a < b) && !(b < a)`.

Sorting

Four different sorting algorithms are provided by the library: `sort`, `stable_sort`, `partial_sort` and `partial_sort_copy`.

`sort` uses the quicksort algorithm, which is generally the fastest for a randomly shuffled sequence. It should be used as the default sorting algorithm. However, in the worst case, `sort` might take quadratic time. This worst case occurs if the original input to the algorithm is already sorted.

If worst case behavior is absolutely critical, then `stable_sort` should be used instead of `sort`.

sort

Description `sort` sorts the elements in the range `[first, last)`. It uses a fast quicksort algorithm. If worst case behavior is important `stable_sort` or `partial_sort` should be used.

Prototype

```
template
    < class RandomAccessIterator >
void sort
    (RandomAccessIterator first,
     RandomAccessIterator last);
template
    < class RandomAccessIterator,
      class Compare >
void sort
    (RandomAccessIterator first,
```

```
RandomAccessIterator last,  
Compare comp);
```

Complexity Sort does approximately $N \log N$ (where N is $(\text{last} - \text{first})$) comparisons on the average. Space complexity is constant.

stable_sort

Description `stable_sort` sorts the elements in the range $[\text{first}, \text{last})$, and ensures that the relative order of the equal elements is preserved.

Prototype

```
template  
< class RandomAccessIterator >  
void stable_sort  
(RandomAccessIterator first,  
RandomAccessIterator last);
```

Prototype

```
template  
< class RandomAccessIterator,  
class Compare >  
void stable_sort  
(RandomAccessIterator first,  
RandomAccessIterator last,  
Compare comp);
```

Remarks `stable_sort` performs at most $N \log N * \log N$ (where N is $\text{last} - \text{first}$) comparisons. If adequate memory is available, then the number of comparisons is $N \log N$. Space complexity for `stable_sort` is variable, at most $O(N)$.

Complexity Sort does approximately $N \log N$ (where N is $(\text{last} - \text{first})$) comparisons on the average. Space complexity is constant.

partial_sort

Description `partial_sort` sorts only a subsequence of the input range. It places the first $\text{middle} - \text{first}$ sorted elements from the range $[\text{first}, \text{last})$ into the range $[\text{first}, \text{middle})$. The rest of the

elements (i.e. those in the range `[middle, last)`) are placed in an undefined order.

Prototype

```
template
    < class RandomAccessIterator >
void partial_sort
    (RandomAccessIterator first,
     RandomAccessIterator middle,
     RandomAccessIterator last);
template
    < class RandomAccessIterator,
      class Compare >
void partial_sort
    (RandomAccessIterator first,
     RandomAccessIterator middle,
     RandomAccessIterator last,
     Compare comp);
```

Remarks `partial_sort` does approximately $(last - first) * \log(middle - first)$ comparisons. Space complexity is constant.

Complexity Sort does approximately $N \log N$ (where N is $(last - first)$) comparisons on the average. Space complexity is constant.

partial_sort_copy

Description `partial_sort_copy` is similar to `partial_sort`. The algorithm places the first $\min(last - first, result_last - result_first)$ sorted elements from the range `[first, last)` into the sequence beginning at `result_first`.

Prototype

```
template
    < class InputIterator,
      class RandomAccessIterator
> RandomAccessIterator partial_sort_copy
    (InputIterator first, InputIterator last,
     RandomAccessIterator result_first,
     RandomAccessIterator result_last);
template
```



```
    < class InputIterator,
      class RandomAccessIterator,
      class Compare >
RandomAccessIterator partial_sort_copy
    (InputIterator first,
     InputIterator last,
     RandomAccessIterator result_first,
     RandomAccessIterator result_last,
     Compare comp);
```

Remarks `partial_sort_copy` does approximately $(last - first) * \log(\min(last - first, result_last - result_first))$ comparisons. Space complexity is proportional to the length of the sequence copied.

Complexity Sort does approximately $N \log N$ (where N is $(last - first)$) comparisons on the average. Space complexity is constant.

`nth_element`

Description The `nth_element` algorithm is a restricted form of the sort algorithm. It places an element of a sequence in the location where it would be if the sequence were sorted.

Prototype

```
template
    < class RandomAccessIterator >
void nth_element
    (RandomAccessIterator first,
     RandomAccessIterator nth,
     RandomAccessIterator last);
template
    < class RandomAccessIterator,
      class Compare >
void nth_element
    (RandomAccessIterator first,
     RandomAccessIterator nth,
     RandomAccessIterator last,
     Compare comp);
```

Remarks In the first version of the algorithm, element comparisons are done using `operator<`, while in the second version they are done using the function object `comp`.

After a call to `nth_element`, for any iterator `i` in the range `[first, nth)` and any iterator `j` in the range `[nth, last)` it holds that `!(i > j)` or `comp(*i, *j) == false`. That is, the algorithm partitions the elements of the sequence according to size: elements to the left of the `nth` element are all less than those to its right.

Complexity The algorithm takes $O(N)$ time on the average, where N is the size of the range `[first, last)`. Space complexity is constant.

Binary Searching

All of the algorithms in this section are versions of binary search.

Although binary search is typically efficient (i.e., performs in logarithmic time) only for random access data structures (such as vectors, deques, etc.), the algorithms here have been written so as to also work on non-random access data structures such as lists. For all non-random access data structures, the total time taken is linear in the size of the container, but the number of comparisons is only logarithmic in the size of the container.

binary_search

Description The `binary_search` function returns true if value is in the range `[first, last)` and false otherwise.

Prototype

```
template
    < class ForwardIterator,
      class T >
bool binary_search
    (ForwardIterator first,
     ForwardIterator last,
     const T& value);
template
    < class ForwardIterator,
      class T,
```

```
class Compare >
bool binary_search
    (ForwardIterator first,
     ForwardIterator last,
     const T& value,
     Compare comp);
```

Remarks In the first function of each pair of functions, element comparisons are done using `operator<`, while in the second they are done using the function object `comp`.

Complexity For random access iterators only, the number of comparisons will be

- at most $\log(N)+1$, for `lower_bound` and `upper_bound`
- at most $\log(N)+2$, for `binary_search`
- at most $2*\log(N)+1$, for `equal_range`

where N is the size of the range `[first, last)`.

For all iterators other than random access, time complexity is linear, since the algorithm has to sequentially traverse through the data structure. Space complexity is constant for all the binary search algorithms.

lower_bound

Description The `lower_bound` functions return an iterator `i` referring to the first position in the sorted sequence in the range `[first, last)` into which `value` may be inserted while maintaining the sorted ordering.

Prototype

```
template
    < class ForwardIterator,
      class T >
ForwardIterator lower_bound
    (ForwardIterator first,
     ForwardIterator last,
     const T& value);
template
```

Algorithms Library

Sorting and Related Algorithms

```
    < class ForwardIterator,
      class T,
      class Compare >
ForwardIterator lower_bound
    (ForwardIterator first,
     ForwardIterator last,
     const T& value,
     Compare comp);
```

Remarks In the first function of each pair of functions, element comparisons are done using `operator<`, while in the second they are done using the function object `comp`.

Complexity For random access iterators only, the number of comparisons will be

- at most $\log(N)+1$, for `lower_bound` and `upper_bound`
- at most $\log(N)+2$, for `binary_search`
- at most $2*\log(N)+1$, for `equal_range`

where N is the size of the range `[first, last)`.

For all iterators other than random access, time complexity is linear, since the algorithm has to sequentially traverse through the data structure. Space complexity is constant for all the binary search algorithms.

upper_bound

Description The `upper_bound` functions return an iterator `i` referring to the last position in the sorted sequence in the range `[first, last)` into which `value` may be inserted while maintaining the sorted ordering.

Prototype

```
template
    < class ForwardIterator,
      class T >
ForwardIterator upper_bound
    (ForwardIterator first,
     ForwardIterator last,
```

```
        const T& value);  
template  
    < class ForwardIterator,  
      class T,  
      class Compare >  
ForwardIterator upper_bound  
    (ForwardIterator first,  
     ForwardIterator last,  
     const T& value,  
     Compare comp);
```

Remarks In the first function of each pair of functions, element comparisons are done using operator<, while in the second they are done using the function object comp.

Complexity For random access iterators only, the number of comparisons will be

- at most $\log(N)+1$, for lower_bound and upper_bound
- at most $\log(N)+2$, for binary_search
- at most $2*\log(N)+1$, for equal_range

where N is the size of the range [first, last).

For all iterators other than random access, time complexity is linear, since the algorithm has to sequentially traverse through the data structure. Space complexity is constant for all the binary search algorithms.

equal_range

Description The equal_range functions return a pair of iterators i and j referring to the first and last positions in the sorted sequence in the range [first, last) into which value may be inserted while maintaining the sorted ordering.

Prototype

```
template  
    < class ForwardIterator, class T >  
pair< ForwardIterator, ForwardIterator >  
equal_range
```

```
        (ForwardIterator first,
         ForwardIterator last,
         const T& value);
template
    < class ForwardIterator, class T >
pair < ForwardIterator, ForwardIterator >
equal_range
    (ForwardIterator first,
     ForwardIterator last,
     const T& value,
     Compare comp);
```

Remarks In the first function of each pair of functions, element comparisons are done using `operator<`, while in the second they are done using the function object `comp`.

Complexity For random access iterators only, the number of comparisons will be

- at most $\log(N)+1$, for `lower_bound` and `upper_bound`
- at most $\log(N)+2$, for `binary_search`
- at most $2*\log(N)+1$, for `equal_range`

where N is the size of the range `[first, last)`.

For all iterators other than random access, time complexity is linear, since the algorithm has to sequentially traverse through the data structure. Space complexity is constant for all the binary search algorithms.

Merging

The merge algorithms join two sorted ranges.

merge

Description `merge` merges two sorted ranges `[first1, last1)` and `[first2, last2)` into the range `[result, result + (last1 - first1) + (last2 - first2))`. The merge is stable, that is, for equal ele-

ments in the two ranges, the elements from the first range always precede the elements from the second. `merge` returns `result + (last1 - first1) + (last2 - first2)`. The result of `merge` is undefined if the resulting range overlaps with either of the original ranges.

Prototype

```
template
    < class InputIterator1,
      class InputIterator2,
      class OutputIterator >
OutputIterator merge
    (InputIterator1 first1,
     InputIterator1 last1,
     InputIterator2 first2,
     InputIterator last2,
     OutputIterator result);
template
    < class InputIterator1,
      class InputIterator2,
      class OutputIterator,
      class Compare >
OutputIterator merge
    (InputIterator1 first1,
     InputIterator1 last1,
     InputIterator2 first2,
     InputIterator last2,
     OutputIterator result,
     Compare comp);
```

Complexity For `merge`, at most $(last1 - first1) + (last2 - first2) - 1$ comparisons are performed.

inplace_merge

Description `inplace_merge` merges two sorted consecutive ranges `[first, middle)` and `[middle, last)` putting the result of the merge into the range `[first, last)`. The merge is stable, that is, for equal elements in the two ranges, the elements from the first range always precede the elements from the second.

Prototype

```
template
    < class BidirectionalIterator >
void inplace_merge
    (BidirectionalIterator first,
     BidirectionalIterator middle,
     BidirectionalIterator last);
template
    < class BidirectionalIterator,
      class Compare >
void inplace_merge
    (BidirectionalIterator first,
     BidirectionalIterator middle,
     BidirectionalIterator last,
     Compare comp);
```

Complexity For `inplace_merge`, at most `last - first` comparisons are performed. If no additional memory is available, the number of assignments can be equal to $N \log N$ where N is equal to `last - first`.

Set Operations on Sorted Structures

The library provides five different types of set operations:

- ["includes" on page 353](#)
- ["set union" on page 353](#)
- ["set intersection" on page 354](#)
- ["set difference" on page 355](#)
- ["set symmetric difference" on page 356](#)

The operations work on sorted structures, such as all STL associative containers.

The algorithms even work with multisets containing multiple copies of equal elements. The semantics of the operations have been generalized to multisets in a standard way, by defining union to contain the maximum number of occurrences of every element, intersection to contain the minimum, and so on.

includes

Description includes checks if the second sequence is a subset of the first sequence (both ranges are assumed to be sorted). The algorithm returns true if every element in the range [first2, last2) is contained in the range [first1, last1), and false otherwise.

Prototype

```
template
    < class InputIterator1,
      class InputIterator2 >
bool includes
    (InputIterator1 first1,
     InputIterator1 last1,
     InputIterator2 first2,
     InputIterator2 last2);
template
    < class InputIterator1,
      class InputIterator2,
      class Compare >
bool includes
    (InputIterator1 first1,
     InputIterator1 last1,
     InputIterator2 first2,
     InputIterator2 last2,
     Compare comp);
```

Complexity TAll of the set operations functions require linear time. In all cases, at most $((last1 - first1) + (last2 - first2)) * 2 - 1$ comparisons are performed. Space complexity is constant.

set_union

Description set_union constructs a sorted union of the elements from the two ranges. set_union is stable, that is, if an element is present in both ranges, the one from the first range is copied.

Prototype

```
template
    < class InputIterator1,
      class InputIterator2,
```

```
        class OutputIterator >
OutputIterator set_union
    (InputIterator1 first1,
     InputIterator1 last1,
     InputIterator2 first2,
     InputIterator2 last2,
     OutputIterator result);
template
    < class InputIterator1,
      class InputIterator2,
      class OutputIterator,
      class Compare >
OutputIterator set_union
    (InputIterator1 first1,
     InputIterator1 last1,
     InputIterator2 first2,
     InputIterator2 last2,
     OutputIterator result,
     Compare comp);
```

Complexity The union, intersection, difference and symmetric difference algorithms all return the end of the constructed range. The result of each of these algorithms is undefined if the resulting range overlaps with either of the original ranges.

All of the set operations functions require linear time. In all cases, at most $((last1 - first1) + (last2 - first2)) * 2 - 1$ comparisons are performed. Space complexity is constant.

set_intersection

Description `set_intersection` constructs a sorted intersection of the elements from the two ranges. `set_intersection` is guaranteed to be stable, that is, if an element is present in both ranges, the one from the first range is copied into the intersection.

Prototype

```
template
    < class InputIterator1,
      class InputIterator2, class OutputIterator >
OutputIterator set_intersection
```

```
        (InputIterator1 first1,
         InputIterator1 last1,
         InputIterator2 first2,
         InputIterator2 last2,
         OutputIterator result);
template
    < class InputIterator1,
      class InputIterator2,
      class OutputIterator,
      class Compare >
OutputIterator set_intersection
    (InputIterator1 first1,
     InputIterator1 last1,
     InputIterator2 first2,
     InputIterator2 last2,
     OutputIterator result,
     Compare comp);
```

Remarks The union, intersection, difference and symmetric difference algorithms all return the end of the constructed range. The result of each of these algorithms is undefined if the resulting range overlaps with either of the original ranges.

Complexity All of the set operations functions require linear time. In all cases, at most $((last1 - first1) + (last2 - first2)) * 2 - 1$ comparisons are performed. Space complexity is constant.

set_difference

Description `set_difference` constructs a sorted difference of the elements from the two ranges. This difference contains elements that are present in the first set but not in the second.

Prototype

```
template
    < class InputIterator1,
      class InputIterator2,
      class OutputIterator >
OutputIterator set_difference
    (InputIterator1 first1,
```

Algorithms Library

Sorting and Related Algorithms

```
        InputIterator1 last1,
        InputIterator2 first2,
        InputIterator2 last2,
        OutputIterator result);
template
    < class InputIterator1,
      class InputIterator2,
      class OutputIterator,
      class Compare >
OutputIterator set_difference
    (InputIterator1 first1,
     InputIterator1 last1,
     InputIterator2 first2,
     InputIterator2 last2,
     OutputIterator result,
     Compare comp);
```

Remarks The union, intersection, difference and symmetric difference algorithms all return the end of the constructed range. The result of each of these algorithms is undefined if the resulting range overlaps with either of the original ranges.

Complexity All of the set operations functions require linear time. In all cases, at most $((\text{last1} - \text{first1}) + (\text{last2} - \text{first2})) * 2 - 1$ comparisons are performed. Space complexity is constant.

set_symmetric_difference

Prototype

```
template
    < class InputIterator1,
      class InputIterator2,
      class OutputIterator >
OutputIterator set_symmetric_difference
    (InputIterator1 first1,
     InputIterator1 last1,
     InputIterator2 first2,
     InputIterator2 last2,
     OutputIterator result);
template
    < class InputIterator1,
```

```
class InputIterator2,
class OutputIterator,
class Compare >
OutputIterator set_symmetric_difference
(InputIterator1 first1,
InputIterator1 last1,
InputIterator2 first2,
InputIterator2 last2,
OutputIterator result,
Compare comp);
```

- Description** `set_symmetric_difference` constructs a sorted symmetric difference of the elements from the two ranges (i.e. a combination of the set of elements that are in the first range but not in the second and vice-versa).
- Remarks** The union, intersection, difference and symmetric difference algorithms all return the end of the constructed range. The result of each of these algorithms is undefined if the resulting range overlaps with either of the original ranges.
- Complexity** All of the set operations functions require linear time. In all cases, at most $((last1 - first1) + (last2 - first2)) * 2 - 1$ comparisons are performed. Space complexity is constant.

Heap Operations

A heap represents a particular organization of a random access data structure (such as a vector or a deque). Given a range `[first, last)`, where `first` and `last` are random access iterators, we say that the elements in the range represent a heap if two key properties are satisfied:

- The value pointed to by the iterator `first` is the largest element in the range
- The value pointed to by the iterator `first` may be removed by `pop_heap`, or a new element added by `push_heap`, in logarithmic time. Both `pop_heap` and `push_heap` return valid heaps.

These properties allow heaps to be used as priority queues.

In addition to `pop_heap` and `push_heap` there are two more heap algorithms: `make_heap` and `sort_heap`.

push_heap

Description	<code>push_heap</code> assumes the range <code>[first1, last - 1)</code> is a valid heap and properly places the value in the location <code>last - 1</code> into the resulting heap <code>[first, last)</code> .
Prototype	<pre>template < class RandomAccessIterator > void push_heap (RandomAccessIterator first, RandomAccessIterator last); template < class RandomAccessIterator, class Compare > void push_heap (RandomAccessIterator first, RandomAccessIterator last, Compare comp);</pre>
Remarks	In the first function of each pair of functions, element comparisons are done using <code>operator<</code> , while in the second they are done using the function object <code>comp</code> .
Complexity	<p>In all of the complexity descriptions below, the number <code>N</code> represents the size of the range <code>[first, last)</code>.</p> <p>The <code>push_heap</code> and <code>pop_heap</code> functions require logarithmic time. The <code>push_heap</code> functions require at most $\log N$ time, and the <code>pop_heap</code> functions require at most $2 * \log N$ comparisons.</p> <p>Space complexity is constant for all heap algorithms.</p>

pop_heap

Description `pop_heap` assumes the range `[first, last)` is a valid heap, then swaps the value in the location `first` with the value in the location `last - 1` and makes `[first, last - 1)` into a heap.

Prototype

```
template
    < class RandomAccessIterator >
void pop_heap
    (RandomAccessIterator first,
     RandomAccessIterator last);
template
    < class RandomAccessIterator,
      class Compare >
void pop_heap
    (RandomAccessIterator first,
     RandomAccessIterator last,
     Compare comp);
```

Remarks In the first function of each pair of functions, element comparisons are done using `operator<`, while in the second they are done using the function object `comp`.

Complexity In all of the complexity descriptions below, the number `N` represents the size of the range `[first, last)`.

The `push_heap` and `pop_heap` functions require logarithmic time. The `push_heap` functions require at most $\log N$ time, and the `pop_heap` functions require at most $2 * \log N$ comparisons.

Space complexity is constant for all heap algorithms.

make_heap

Description The `make_heap` functions construct a heap in the range `[first, last)` using the elements in the range `[first, last)`.

Prototype

```
template
    < class RandomAccessIterator >
```

Algorithms Library

Sorting and Related Algorithms

```
void make_heap(
    RandomAccessIterator first,
    RandomAccessIterator last);
template
    < class RandomAccessIterator,
      class Compare >
void make_heap(RandomAccessIterator first,
    RandomAccessIterator last,
    Compare comp);
```

Remarks In the first function of each pair of functions, element comparisons are done using `operator<`, while in the second they are done using the function object `comp`.

Complexity In all of the complexity descriptions below, the number N represents the size of the range `[first, last)`.

The `make_heap` functions require linear time and require at most $3 \cdot N$ comparisons.

Space complexity is constant for all heap algorithms.

sort_heap

Description The `sort_heap` functions sort the elements that are stored in the heap represented in the range `[first, last)`.

Prototype

```
template
    < class RandomAccessIterator >
void sort_heap
    (RandomAccessIterator first,
     RandomAccessIterator last);
template
    < class RandomAccessIterator,
      class Compare >
void sort_heap
    (RandomAccessIterator first,
     RandomAccessIterator last,
     Compare comp);
```


Remarks	In the first function of each pair of functions, element comparisons are done using <code>operator<</code> , while in the second they are done using the function object <code>comp</code> .
Complexity	<p>Tn all of the complexity descriptions below, the number <code>N</code> represents the size of the range <code>[first, last)</code>.</p> <p>The <code>sort_heap</code> functions require $N\log N$ time and $N\log N$ comparisons.</p> <p>Space complexity is constant for all heap algorithms.</p>

Finding Min and Max

The min and max algorithms identify the larger or smaller of two elements, or of elements in a range.

min

Description	The <code>min</code> function is passed two elements, and returns the one that is smaller.
Prototype	<pre>template < class T > T min (const T &a, const T &b); template < class T, class Compare > T min (const T &a, const T &b, Compare comp);</pre>
Remarks	In the first function of each pair of functions, element comparisons are done using <code>operator<</code> , while in the second they are done using the function object <code>comp</code> .

Algorithms Library

Sorting and Related Algorithms

Complexity min and max take constant time. Space complexity is constant for all min and max algorithms.

max

Description The max function is passed two elements, and returns the one that is larger.

Prototype

```
template
    < class T >
T max
    (const T &a,
     const T &b);
template
    < class T,
      class Compare >
T max
    (const T &a,
     const T &b,
     Compare comp);
```

Remarks In the first function of each pair of functions, element comparisons are done using operator<, while in the second they are done using the function object comp.

Complexity min and max take constant time. Space complexity is constant for all min and max algorithms.

min_element

Description The min_element function returns an iterator i referring to the minimum of the elements in the range [first, last).

Prototype

```
template
    < class InputIterator >
InputIterator min_element
    (InputIterator first,
     InputIterator last);
template
```

```
        < class InputIterator,
        class Compare >
InputIterator min_element
    (InputIterator first,
     InputIterator last,
     Compare comp);
```

Remarks In the first function of each pair of functions, element comparisons are done using operator<, while in the second they are done using the function object comp.

Complexity Tmin_element takes linear time, with the number of element comparisons being the size of the range [first, last). Space complexity is constant for all min and max algorithms.

max_element

Description The max_element functions returns an iterator i referring to the maximum of the elements in the range [first, last).

Prototype

```
template
    < class InputIterator >
InputIterator max_element
    (InputIterator first,
     InputIterator last);
template
    < class InputIterator,
    class Compare >
InputIterator max_element
    (InputIterator first,
     InputIterator last,
     Compare comp);
```

Complexity Tmax_element takes linear time, with the number of element comparisons being the size of the range [first, last). Space complexity is constant for all min and max algorithms.

Lexicographical Comparison

The lexicographical comparison algorithm compares two sequences of elements.

lexicographical_compare

Description The lexicographical comparison of two sequences `[first1, last1)` and `[first2, last2)` is defined as follows: traverse the sequences, comparing corresponding pairs of elements `e1` and `e2`. If `e1 < e2`, stop and return `true`. If `e2 < e1`, stop and return `false`. Otherwise, continue to the next corresponding pair of elements.

If the first sequence is exhausted but the second is not, then return `true`, otherwise return `false`.

Prototype

```
template
    < class InputIterator1,
      class InputIterator2 >
bool lexicographical_compare
    (InputIterator1 first1,
     InputIterator1 last1,
     InputIterator2 first2,
     InputIterator2 last2);

template
    < class InputIterator1,
      class InputIterator2,
      class Compare >
bool lexicographical_compare
    (InputIterator1 first1,
     InputIterator1 last1,
     InputIterator2 first2,
     InputIterator2 last2,
     Compare comp);
```

Remarks Comparisons of `e1` and `e2` are done with operator`<` in the first version of the algorithm, and with the function object `comp` in the second version.

Complexity Time complexity is linear. The number of comparisons done is at most i , where i is the smallest index at which a disagreement occurs. Space complexity is constant.

Permutation Generators

The library provides two permutation generation algorithms: `next_permutation` and `prev_permutation`. As with all sorting related operations, there are two versions of each algorithm: one that uses `operator<` for comparisons, and one that uses a function object `comp`.

`next_permutation`

Description `next_permutation` takes a sequence defined by the range `[first, last)` and transforms it into the next permutation. The next permutation is found by assuming that the set of all permutations is lexicographically sorted with respect to `operator<` or `comp`. If such a permutation exists, the algorithm returns `true`. Otherwise, it transforms the sequence into the smallest permutation (i.e., the one sorted in an ascending order), and returns `false`.

Prototype

```
template
    < class BidirectionalIterator >
bool next_permutation
    (BidirectionalIterator first,
     BidirectionalIterator last);
template
    < class BidirectionalIterator,
      class Compare >
bool next_permutation
    (BidirectionalIterator first,
     BidirectionalIterator last,
     Compare comp);
```

Complexity Time complexity is linear for both algorithms. The number of comparisons done is at most n where n is half the range `[first, last)`. Space complexity is constant.

prev_permutation

Description `prev_permutation` takes a sequence defined by the range `[first, last)` and transforms it into the previous permutation. The previous permutation is found by assuming that the set of all permutations is lexicographically sorted with respect to operator `<` or `comp`. If such a permutation exists, it returns `true`. Otherwise, it transforms the sequence into the largest permutation, (i.e., the descending sorted one), and returns `false`.

Prototype

```
template
    < class BidirectionalIterator >
bool prev_permutation
    (BidirectionalIterator first,
     BidirectionalIterator last);
template
    < class BidirectionalIterator,
      class Compare >
bool prev_permutation
    (BidirectionalIterator first,
     BidirectionalIterator last,
     Compare comp);
```

Complexity Time complexity is linear for both algorithms. The number of comparisons done is at most n where n is half the range `[first, last)`. Space complexity is constant.

Generalized Numeric Algorithms

The library provides four algorithms for numeric processing:

- [“accumulate” on page 367](#)
- [“inner product” on page 367](#)
- [“partial sum” on page 369](#)
- [“adjacent difference” on page 370](#)

See also [“Generalized Numeric Algorithms” on page 366](#).

accumulate

Description `accumulate` is similar to the APL reduction operator and the Common Lisp `reduce` function, but avoids the difficulty of defining the result of reduction on an empty sequence by always requiring an initial value.

Prototype

```
template
    < class InputIterator, class T >
T accumulate
    (InputIterator first,
     InputIterator last,
     T init);
template
    < class InputIterator,
      class T,
      class BinaryOperation >
T accumulate
    (InputIterator first,
     InputIterator last,
     T init,
     BinaryOperation binary_op);
```

Remarks Accumulation is done by initializing the accumulator `acc` with the initial value `init` and then modifying it with `acc = acc + *i` or `acc = binary_op(acc, *i)` for every iterator `i` in the range `[first, last)` in order. `binary_op` is assumed not to cause any side effects.

Complexity Time complexity is linear. The number of applications of operator+ or `binary_op` done is `n` where `n` is the range `[first, last)`. Space complexity is constant.

inner_product

Description The `inner_product` algorithm computes its result by initializing the accumulator `acc` with initial value `init` and then modifying it as follows:
`acc = acc + (*i1) * (*i2)`

Algorithms Library

Generalized Numeric Algorithms

or
`acc = binary_op1(acc, binary_op2(*i1, *i2))`

for every iterator `i1` in the range
`[first1, last1)`

and `i2` in the range
`[first2, first2 + (last1 - first1))`

in order.

`binary_op1` and `binary_op2` are assumed to cause no side effects.

Prototype

```
template
    < class InputIterator1,
      class InputIterator2,
      class T >
T inner_product
    (InputIterator1 first1,
     InputIterator1 last1,
     InputIterator2 first2,
     T init);
template
    < class InputIterator1,
      class InputIterator2,
      class T,
      class BinaryOperation1,
      class BinaryOperation2 >
T inner_product
    (InputIterator1 first1,
     InputIterator1 last1,
     InputIterator2 first2,
     T init,
     BinaryOperation1 binary_op1,
     BinaryOperation2 binary_op2);
```

Complexity Time complexity is linear. The number of applications of operator+ or `binary_op1`, and the number of applications of operator* or `binary_op2` is `n` where `n` is the size of the range `[first, last)`. Space complexity is constant.

partial_sum

Description The `partial_sum` algorithm computes partial sums of the input sequence and stores the result in the output sequence.

Formally, `partial_sum` assigns to every iterator `i` in the range `[result, result + (last - first))`

a value correspondingly equal to

$$((...(*first + *(first + 1)) + ...) + (*first + (i - result)))$$

or

```
binary_op(binary_op  
(., binary_op(*first + *(first + 1)) +...) + (*first + (i - result))).
```

`binary_op` is expected not to have any side effects.

Prototype

```
template  
    < class InputIterator,  
      class OutputIterator >  
InputIterator partial_sum  
    (InputIterator first,  
     InputIterator last,  
     OutputIterator result);  
template  
    < class InputIterator,  
      class OutputIterator,  
      class BinaryOperation >  
InputIterator partial_sum  
    (InputIterator first,  
     InputIterator last,  
     OutputIterator result,  
     BinaryOperation binary_op);
```

Remarks The `partial_sum` algorithm returns an iterator referring to the past-the-end location of the result sequence.

NOTE: The result may be equal to `first`; i.e., it is possible for the algorithm to work “in-place”, meaning that the algorithm can generate the partial sums and replace the original sequence with them.

Complexity Time complexity is linear. The number of applications of `operator+ or binary_op` is one less than `n`, where `n` is the size of the range `[first, last)`. Space complexity is constant.

adjacent_difference

Description `adjacent_difference` assigns to every element referred to by iterator `i` in the range `[result + 1, result + (last - first))`

a value correspondingly equal to

$$*(first + (i - result)) - *(first + (i - result) - 1)$$

or

$$binary_op(*(first + (i - result)), *(first + (i - result) - 1)).$$

`binary_op` is expected not to have any side effects.

Prototype

```
template
    < class InputIterator,
      class OutputIterator >
InputIterator adjacent_difference
    (InputIterator first,
     InputIterator last,
     OutputIterator result);
template
    < class InputIterator,
      class OutputIterator,
      class BinaryOperation>
InputIterator adjacent_difference
    (InputIterator first,
     InputIterator last,
```

```
OutputIterator result,  
BinaryOperation binary_op);
```

Remarks `result` may be equal to `first`; i.e., the algorithm can work “in-place”.

Complexity Time complexity is linear. The number of applications of operator- or `binary_op` is one less than `n` where `n` is the range `[first, last)`. Space complexity is constant.



Numerics Library

This chapter is a reference guide to the ANSI/ISO standard Numeric classes which are used to perform the semi-numerical operations.

Overview of the Numerics Library

This library contains the classes for complex number types, numeric arrays, generalized numeric algorithms, and facilities included from the ISO C library.

The sections in this chapter are:

- [“Numeric Type Requirements” on page 373](#)
- [“Numeric Arrays” on page 374](#)
- [“Generalized Numeric Operations” on page 414](#)

Numeric Type Requirements

The `complex` and `valarray` components are parameterized by the type of information they contain and manipulate. A C++ program shall instantiate these components with types that satisfy the following requirements:

- T is not an abstract class, i.e it has no pure virtual member functions;
- T is not a reference type;
- T is not cv-qualified;
- If T is a class, it has a public default constructor;
- If T is a class, it has a public copy constructor with the signature `T::T(const T&);`
- If T is a class, it has a public destructor;

- If `T` is a class, it has a public assignment operator whose signature is either

`T& T::operator=(const T&) or T& T::operator=(T);`

If `T` is a class, its assignment operator, copy and default constructors, and destructor must correspond to each other in the following sense: Initialization of raw storage using the default constructor, followed by assignment, is semantically equivalent to initialization of raw storage using the copy constructor. Destruction of an object, followed by initialization of its raw storage using the copy constructor, is semantically equivalent to assignment to the original object.

In addition, many member and related functions of `valarray<T>` can be successfully instantiated and will exhibit well-defined behavior if and only if `T` satisfies additional requirements specified for each such member or related function.

Numeric Arrays

The header `<valarray>` defines five template classes: `valarray`, `slice_array`, `gslice_array`, `mask_array` and `indirect_array`, two classes: `slice` and `gslice`, and a series of related function signatures for representing and manipulating arrays of values.

Files `#include <valarray.h>`

The array classes are:

- [“Template Class `valarray`” on page 375](#)
- [“Class `slice`” on page 400](#)
- [“Template class `slice_array`” on page 401](#)
- [“Class `gslice`” on page 404](#)
- [“Template Class `gslice_array`” on page 405](#)
- [“Template Class `mask_array`” on page 408](#)
- [“Template Class `indirect_array`” on page 411](#)

Template Class valarray

Description The template class `valarray<T>` is a one-dimensional smart array, with elements numbered sequentially from zero. It is a representation of the mathematical concept of an ordered set of values. The illusion of higher dimensionality may be produced by the familiar idiom of computed indices, together with the powerful subsetting capabilities provided by the generalized subscript operators.

The topics in this section are:

- [“Constructors valarray” on page 375](#)
- [“Element Access valarray” on page 378](#)
- [“Subset Operations valarray” on page 378](#)
- [“Assignment Operators valarray” on page 377](#)
- [“Unary Operators valarray” on page 378](#)
- [“Computed Assignment Type valarray<T>” on page 379](#)
- [“Computed Assignment Type<T>” on page 381](#)
- [“Overloaded Binary Operators valarray” on page 385](#)
- [“Comparison Operators valarray” on page 389](#)
- [“Overloaded Comparison Operators valarray” on page 391](#)
- [“Member Functions valarray” on page 393](#)
- [“Min And Max Functions valarray” on page 395](#)
- [“Transcendentals valarray” on page 396](#)

Constructors valarray

Default Constructor

Description This constructs an object of class `valarray<T>`, which has zero length until it is passed into a library function as a modifiable lvalue or through a non-constant this pointer.

Prototype `valarray() ;`

Overloaded Constructors

Prototype `valarray (size_t);`

Remarks This constructor sets the array length equal to the value of the argument. The elements of the array are constructed using the default constructor for the instantiating type T.

`valarray (const T&, size_t);`

Remarks This constructor sets the array equal to the value of the second argument and initializes all the elements with the value of the first argument.

`valarray (const T*, size_t);`

Remarks The array created by this constructor has a length equal to the second argument. The values of the elements of the array are initialized with the first n values pointed to by the first argument. It is necessary that the value of the second argument is greater than the number of vales pointed to by the first argument.

Copy Constructor

Description Copy constructor creates a distinct array rather than a alias.

Prototype `valarray (const valarray<T>&);`

Conversion Constructors

Description These are conversion constructors which convert one of the four reference templates to valarray.

Prototype `valarray (const slice_array<T>&);`
`valarray (const gslice_array<T>&);`
`valarray (const mask_array<T>&);`
`valarray (const indirect_array<T>&);`

Destructor

Description Destructor for valarray.

Prototype `~valarray ();`

Assignment Operators valarray

Assignment Operator =

Description The assignment operator modifies the length of the `*this` array to be equal to that of the argument array. Each element of `*this` array is then assigned the value of the corresponding element of the argument array. Assignment is the usual way to change the length of an array after initialization. Assignment results in a distinct array rather than an alias.

Prototype `valarray<T>& operator= (const valarray<T>&);`

Overloaded Assignment Operators

Prototype `valarray<T>& operator=
 (const slice_array<T>&);
valarray<T>& operator=
 (const gslice_array<T>&);
valarray<T>& operator=
 (const mask_array<T>&);
valarray<T>& operator=
 (const indirect_array<T>&);`

Remarks These operators allow the results of a generalized subscripting operation to be assigned directly to a valarray.

Element Access valarray

Subscript Operator []

Description For const valarray objects, the subscript operator returns the value of the corresponding element of the array. For non-const valarray objects, a reference to the corresponding element of the array is returned.

Prototype `T operator[] (size_t) const;`
`T& operator[] (size_t);`

Subset Operations valarray

Subscript Operator []

Description Each of these operations returns a subset of the array. The const-qualified versions return this subset as a new valarray. The non-const versions return a class template object which has reference semantics to the original array.

Prototype `valarray<T> operator[](slice) const;`
`slice_array<T> operator[](slice);`
`valarray<T> operator[](const gslice&) const;`
`gslice_array<T> operator[] (const gslice&);`
`valarray<T> operator[]`
`(const valarray<bool>&) const;`
`mask_array<T> operator[] (const valarray<bool>&);`
`valarray<T> operator[]`
`(const valarray<size_t>&) const;`
`indirect_array<T> operator[]`
`(const valarray<size_t>&);`

Unary Operators valarray

Description Each of these operators may only be instantiated for a type T to which the indicated operator can be applied and for which the indicated operator returns a value which is of type &T or which may be

unambiguously converted to type T. Each of these operators returns an array whose length is equal to the length of the array. Each element of the returned array is initialized with the result of applying the indicated operator to the corresponding element of the array.

Add Operator +

Prototype `valarray<T> operator+() const;`

Minus Operator -

Prototype `valarray<T> operator-() const;`

Complement Operator ~

Prototype `valarray<T> operator~() const;`

Not Operator !

Prototype `valarray<T> operator!() const;`

Computed Assignment Type valarray<T>

Description Each of these operators may only be instantiated for a type T to which the indicated operator can be applied. Each of these operators performs the indicated operation on each of its elements and the corresponding element of the argument array. The array is then returned by reference. If the array and the argument array do not have the same length, the behavior is undefined. The appearance of an array on the left hand side of a computed assignment does not invalidate references or pointers.

Multiply & Assign Operator *=

Prototype `valarray<T>& operator*= (const valarray<T>&);`

Divide and Assign Operator /=

Prototype `valarray<T>& operator/= (const valarray<T>&);`

Remainder & Assign Operator %=

Prototype `valarray<T>& operator%= (const valarray<T>&);`

Add & Assign Operator +=

Prototype `valarray<T>& operator+= (const valarray<T>&);`

Minus and Assign Operator -=

Prototype `valarray<T>& operator-= (const valarray<T>&);`

XOR and Assign Operator ^=

Prototype `valarray<T>& operator^= (const valarray<T>&);`

And & Assign Operator &=

Prototype `valarray<T>& operator&= (const valarray<T>&);`

Or & Assign Operator |=

Prototype `valarray<T>& operator|= (const valarray<T>&);`

Not Equal Operator !=

Prototype `valarray<T>& operator!= (const valarray<T>&);`

Right Shift & Assign Operator >>=

Prototype `valarray<T>& operator>>=(const valarray<T>&);`

Computed Assignment Type<T>

Description Each of these operators may only be instantiated for a type T to which the indicated operator can be applied. Each of these operators applies the indicated operation to each element of the array and the scalar argument. The array is then returned by reference. The appearance of an array on the left hand side of a computed assignment does not invalidate references or pointers to the elements of the array.

Multiply & Assign Operator *=

Prototype `valarray<T>& operator*= (const T&);`

Divide & Assign Operator /=

Prototype `valarray<T>& operator/= (const T&);`

Remainder & Assign Operator %=

Prototype `valarray<T>& operator%= (const T&);`

Add & Assign Operator +=

Prototype `valarray<T>& operator+= (const T&);`

Minus & Assign Operator -=

Prototype `valarray<T>& operator-= (const T&);`

XOR & Assign Operator ^=

Prototype `valarray<T>& operator^= (const T&);`

And & Assign Operator &=

Prototype `valarray<T>& operator&= (const T&);`

Not Equal Operator !=

Prototype `valarray<T>& operator|= (const T&);`

Right Shift & Assign Operator >>=

Prototype `valarray<T>& operator>>=(const T&);`

Non-Member Binary Operators valarray

Description Each of these operators may only be instantiated for a type T to which the indicated operator can be applied and for which the indicated operator returns a value which is of type T or which can be unambiguously converted to type T. Each of these operators returns an array whose length is equal to the lengths of the argument arrays. Each element of the returned array is initialized with the result of applying the indicated operator to the corresponding elements of the argument arrays. If the argument arrays do not have the same length, the behavior is undefined.

Multiply Operator *

Prototype

```
template
    < class T >
valarray<T> operator*
    (const valarray<T>&,
     const valarray<T>&);
```

Divide Operator /

Prototype

```
template
    < class T >
valarray<T> operator/
```

```
(const valarray<T>&,
const valarray<T>&);
```

Remainder Operator %

Prototype `template`
 `< class T >`
`valarray<T> operator%`
 `(const valarray<T>&,`
 `const valarray<T>&);`

Add Operator +

Prototype `template`
 `< class T >`
`valarray<T> operator+`
 `(const valarray<T>&,`
 `const valarray<T>&);`

Minus Operator -

Prototype `template`
 `< class T >`
 `valarray<T>`
`operator-`
 `(const valarray<T>&,`
 `const valarray<T>&);`

Bitwise XOR Operator ^

Prototype `template`
 `< class T >`
`valarray<T> operator^`
 `(const valarray<T>&,`
 `const valarray<T>&);`

Bitwise And Operator &

Prototype `template
 < class T >
valarray<T> operator&
 (const valarray<T>&,
 const valarray<T>&);`

Bitwise Or Operator

Prototype `template
 < class T >
valarray<T> operator|
 (const valarray<T>&,
 const valarray<T>&);`

Left Shift Operator <<

Prototype `template
 < class T >
valarray<T> operator<<
 (const valarray<T>&,
 const valarray<T>&);`

Right Shift Operator >>

Prototype `template
 < class T >
valarray<T> operator>>
 (const valarray<T>&,
 const valarray<T>&);`

Logical And Operator &&

Prototype `template
 < class T >
valarray<T> operator&&
 (const valarray<T>&,`


```
const valarray<T>&);
```

Logical Or Operator ||

Prototype

```
template
< class T >
valarray<T> operator||
    (const valarray<T>&,
     const valarray<T>&);
```

Overloaded Binary Operators valarray

Description Each of these operators may only be instantiated for a type T to which the indicated operator can be applied and for which the indicated operator returns a value which is of type T or which can be unambiguously converted to type T. Each of these operators returns an array whose length is equal to the length of the array argument. Each element of the returned array is initialized with the result of applying the indicated operator to the corresponding element of the array argument and the scalar argument.

Multiply Operator *

Prototype

```
template
< class T >
valarray<T> operator*
    (const valarray<T>&,
     const T&);
template
< class T >
valarray<T> operator*
    (const T&, const
     valarray<T>&);
```

Divide Operator /

Prototype

```
template
< class T >
```

```
valarray<T> operator/  
    (const valarray<T>&,  
     const T&);  
template  
    < class T>  
valarray<T> operator/  
    (const T&,  
     const valarray<T>&);
```

Remainder Operator %

Prototype

```
template  
    < class T >  
valarray<T >operator%  
    (const valarray<T>&,  
     const T&);  
template  
    < class T >  
valarray<T> operator%  
    (const T&,  
     const valarray<T>&);
```

Add Operator +

Prototype

```
template  
    < class T >  
valarray<T> operator+  
    (const valarray<T>&,  
     const T&);  
template  
    < class T>  
valarray<T> operator+  
    (const T&,  
     const valarray<T>&);
```

Minus Operator -

Prototype

```
template  
    < class T >
```

```
valarray<T> operator-
    (const valarray<T>&,
     const T&);
template
    < class T > v
alarray<T> operator-
    (const T&,
     const valarray<T>&);
```

Bitwise XOR Operator ^

Prototype

```
template
    < class T >
valarray<T> operator^
    (const valarray<T>&,
     const T&);
template
    < class T >
valarray<T> operator^
    (const T&,
     const valarray<T>&);
```

Bitwise And Operator &

Prototype

```
template
    < class T >
valarray<T> operator&
    (const valarray<T>&,
     const T&);
template
    < class T >
valarray<T> operator&
    (const T&,
     const valarray<T>&);
```

Bitwise Or Operator |

Prototype

```
template
    < class T >
```

```
valarray<T> operator|
    (const valarray<T>&,
     const T&);
template
    < class T >
valarray<T> operator|
    (const T&,
     const valarray<T>&);
```

Left Shift Operator <<

Prototype

```
template
    < class T >
valarray<T> operator<<
    (const valarray<T>&, const T&);
template
    < class T>
valarray<T> operator<<
    (const T&,
     const valarray<T>&);
```

Right Shift Operator >>

Prototype

```
template
    < class T >
valarray<T> operator>>
    (const valarray<T>&,
     const T&);
template
    < class T >
valarray<T> operator>>
    (const T&, const
     valarray<T>&);
```

Logical And Operator &&

Prototype

```
template
    < class T >
valarray<T> operator&&
```

```
        (const valarray<T>&,
         const T&);
template
    < class T >
valarray<T> operator&&
    (const T&,
     const valarray<T>&);
```

Logical Or Operator ||

Prototype

```
template
    < class T >
valarray<T> operator||
    (const valarray<T>&,
     const T&);
template
    < class T >
valarray<T> operator||
    (const T&,
     const valarray<T>&);
```

Comparison Operators valarray

Description Each of these operators may only be instantiated for a type T to which the indicated operator can be applied and for which the indicated operator returns a value which is of type bool or which can be unambiguously converted to type bool. Each of these operators returns a bool array whose length is equal to the length of the array arguments. Each element of the returned array is initialized with the result of applying the indicated operator to the corresponding elements of the argument arrays. If the two array arguments do not have the same length, the behavior is undefined.

Equality Operator ==

Prototype

```
template
    < class T >
valarray<bool> operator==
    (const valarray<T>&,
     const valarray<T>&);
```

```
        const valarray<T>&);  
template  
    < class T>  
valarray<bool> operator!=  
    (const valarray<T>&,  
     const valarray<T>&);
```

Less Than Operator <

Prototype

```
template  
    < class T >  
valarray<bool> operator<  
    (const valarray<T>&,  
     const valarray<T>&);
```

Greater Than Operator >

Prototype

```
template  
    < class T >  
valarray<bool> operator>  
    (const valarray<T>&,  
     const valarray<T>&);
```

Not Equal Operator !=

Prototype

```
template  
    < class T >  
valarray<bool> operator!=  
    (const valarray<T>&,  
     const valarray<T>&);
```

Greater Than or Equal Operator >=

Prototype

```
template  
    < class T >  
valarray<bool> operator>=  
    (const valarray<T>&,  
     const valarray<T>&);
```

Overloaded Comparison Operators valarray

Description Each of these operators may only be instantiated for a type T to which the indicated operator can be applied and for which the indicated operator returns a value which is of type bool or which can be unambiguously converted to type bool. Each of these operators returns a bool array whose length is equal to the length of the array argument. Each element of the returned array is initialized with the result of applying the indicated operator to the corresponding element of the array and the scalar argument.

Equality Operator ==

Prototype

```
template
    < class T >
valarray<bool> operator==
    (const valarray&,
     const T&);
template
    < class T>
valarray<bool> operator==
    (const T&,
     const valarray&);
```

Not Equal Operator !=

Prototype

```
template
    < class T >
valarray<bool> operator!=
    (const valarray&,
     const T&);
template
    < class T > valarray<bool> operator!=
    (const T&,
     const valarray&);
```

Less Than Operator <

Prototype

```
template
    < class T >
valarray<bool> operator<
    (const valarray&,
     const T&);
template
    < class T >
valarray<bool> operator<
    (const T&,
     const valarray&);
```

Greater Than Operator >

Prototype

```
template
    < class T >
valarray<bool> operator>
    (const valarray&,
     const T&);
template
    < class T >
valarray<bool> operator>
    (const T&,
     const valarray&);
```

Less Than or Equal Operator <=

Prototype

```
template
    < class T >
valarray<bool> operator<=
    (const valarray&,
     const T&);
template
    < class T >
valarray<bool> operator<=
    (const T&,
     const valarray&);
```


Greater Than or Equal Operator >=

Prototype

```
template
    < class T >
valarray<bool> operator>=
    (const valarray&,
     const T&);
template
    < class T >
valarray<bool> operator>=
    (const T&,
     const valarray&);
```

Member Functions valarray

valarray::length

Description This function returns the number of elements in the array.

Prototype `size_t length() const;`

Pointer Conversion

Description A non-constant array may be converted to a pointer to the instantiating type. A constant array may be converted to a pointer to the instantiating type, qualified by `const`. It is guaranteed that `&a[0] == (T*)a` for any non-constant `valarray<T> a`. The pointer returned for a non-constant array (whether or not it points to a type qualified by `const`) is valid for the same duration as a reference returned by the `size_t` subscript operator. The pointer returned for a constant array is valid for the lifetime of the array.

Prototype

```
operator T*();
operator const T*() const;
```

valarray::sum

Description This function may only be instantiated for a type `T` to which operator`+=` can be applied. This function returns the sum of all the elements of the array. If the array has length 0, the behavior is undefined. If the array has length 1, `sum` returns the value of element 0. Otherwise, the returned value is calculated by applying operator`+=` to a copy of an element of the array and all other elements of the array in an unspecified order.

Prototype `T sum() const;`

valarray::fill

Description This function assigns the value of the argument to all the elements of the array. The length of the array is not changed, nor are any pointers or references to the elements of the array invalidated.

Prototype `void fill (const T&);`

valarray::shift

Prototype `valarray<T> shift(int) const;`

Description This function returns an array whose length is identical to the array, but whose element values are shifted the number of places indicated by the argument. A positive argument value results in a left shift, a negative value in a right shift, and a zero value in no shift.

valarray::cshift

Description This function returns an array whose length is identical to the array, but whose element values are shifted in a circular fashion the number of places indicated by the argument. A positive argument value results in a left shift, a negative value in a right shift, and a zero value in no shift.

Prototype `valarray<T> cshift(int) const;`

valarray::apply

Description These functions return an array whose length is equal to the array. Each element of the returned array is assigned the value returned by applying the argument function to the corresponding element of the array.

Prototype `valarray<T> apply(T func(T)) const;
valarray<T> apply(T func(const T&)) const;`

valarray::free

Description This function sets the length of an array to zero.

Prototype `void free();`

Min And Max Functions valarray

valarray::min

Description This function may only be instantiated for a type T to which operator> and operator< may be applied and for which operator> and operator< return a value which is of type bool or which can be unambiguously converted to type bool. This function returns the minimum (a.min()) value found in the argument array a. The value returned for an array of length 0 is undefined. For an array of length 1, the value of element 0 is returned. For all other array lengths, the determination is made using operator> and operator<, in a manner analogous to the application of operator+= for the sum function.

Prototype `template
 < class T >
T min(const valarray<T>& a);`

valarray::max

Description This function may only be instantiated for a type `T` to which `operator>` and `operator<` may be applied and for which `operator>` and `operator<` return a value which is of type `bool` or which can be unambiguously converted to type `bool`. These functions return the maximum (`a.max()`) value found in the argument array `a`. The value returned for an array of length 0 is undefined. For an array of length 1, the value of element 0 is returned. For all other array lengths, the determination is made using `operator>` and `operator<`, in a manner analogous to the application of `operator+=` for the `sum` function.

Prototype

```
template
    < class T>
T max(const valarray<T>& a);
```

Transcendentals valarray

Description Each of these functions may only be instantiated for a type `T` to which a unique function with the indicated name can be applied. This function must return a value which is of type `T` or which can be unambiguously converted to type `T`.

valarray::abs

Prototype

```
template
    < class T >
valarray<T> abs(const valarray<T>&);
```

valarray::acos

Prototype

```
template
    < class T>
valarray<T> acos(const valarray<T>&);
```

valarray::asin

Prototype `template
 < class T >
valarray<T> asin(const valarray<T>&);`

valarray::atan

Prototype `template
 < class T >
valarray<T> atan(const valarray<T>&);`

valarray::atan2

Prototype `template
 < class T >
valarray<T> atan2
 (const valarray<T>&,
 const valarray<T>&);`

Prototype `template
 < class T >
valarray<T> atan2
 (const valarray<T>&,
 const T&);`

Prototype `template
 < class T >
valarray<T> atan2
 (const T&,
 const valarray<T>&);`

valarray::cos

Prototype `template
 < class T >
valarray<T> cos (const valarray<T>&);`

valarray::cosh

Prototype `template
 < class T >
valarray<T> cosh(const valarray<T>&);`

valarray::exp

Prototype `template
 < class T >
valarray<T> exp(const valarray<T>&);`

valarray::log

Prototype `template
 < class T >
valarray<T> log(const valarray<T>&);`

valarray::log10

Prototype `template
 < class T >
valarray<T> log10(const valarray<T>&);`

valarray::pow

Prototype `template
 < class T >
valarray<T> pow
 (const valarray<T>&,
 const valarray<T>&);`

Prototype `template
 < class T >
valarray<T> pow
 (const valarray<T>&,
 const T&);`

Prototype `template
 < class T >
valarray<T> pow
 (const T&,
 const valarray<T>&);`

valarray::sin

Prototype `template
 < class T >
valarray<T> sin(const valarray<T>&);`

valarray::sinh

Prototype `template
 < class T >
valarray<T> sinh(const valarray<T>&);`

valarray::sqrt

Prototype `template
 < class T >
valarray<T> sqrt (const valarray<T>&);`

valarray::tan

Prototype `template
 < class T >
valarray<T> tan(const valarray<T>&);`

valarray::tanh

Prototype `template
 < class T >
valarray<T> tanh(const valarray<T>&);`

Class slice

Description The slice class represents a BLAS-like slice from an array. Such a slice is specified by a starting index, a length, and a stride.

Prototype

```
class slice {  
    public:  
        slice();  
        slice(size_t, size_t, size_t);  
  
        size_t start() const;  
        size_t length() const;  
        size_t stride() const;  
};
```

The topics in this section are:

- [“Constructors slice” on page 400](#)
- [“Access Functions slice” on page 401](#)

Constructors slice

Default Constructor

Description The default constructor for slice creates a slice which specifies no elements. A default constructor is provided only to permit the declaration of arrays of slices.

Prototype `slice();`

Overloaded and Copy Constructors

Description The constructor with arguments for a slice takes a start, length, and stride parameter.

Prototype

```
slice(  
    size_t start,  
    size_t length,  
    size_t stride);
```



```
slice(const slice&);
```

Access Functions slice

Description These functions return the start, length, or stride specified by a slice object.

slice::start

Prototype `size_t start() const;`

slice::length

Prototype `size_t length() const;`

slice::stride

Prototype `size_t stride() const;`

Template class slice_array

Description The `slice_array` template is a helper template used by the slice subscript operator `slice_array<T> valarray<T>::operator[](slice)`; It has reference semantics to a subset of an array specified by a slice object.

The topics in this section are:

- [“Constructors slice_array” on page 402](#)
- [“Assignment Operators slice_array” on page 402](#)
- [“Computed Assignment slice_array” on page 402](#)
- [“Public Member Function slice_array” on page 404](#)—the `fill` function

Constructors slice_array

Default and Copy Constructor

Description The `slice_array` template has no public constructors. These constructors are declared to be private. These constructors need not be defined.

Prototype `slice_array();`

Prototype `slice_array(const slice_array&);`

Assignment Operators slice_array

Assignment Operator =

Description The second of these two assignment operators is declared private and need not be defined. The first has reference semantics, assigning the values of the argument array elements to selected elements of the `valarray<T>` object to which the `slice_array` object refers.

Prototype `void operator= (const valarray<T>&) const;`
`slice_array& operator=(const slice_array&);`

Computed Assignment slice_array

Description These computed assignments have reference semantics, applying the indicated operation to the elements of the argument array and selected elements of the `valarray<T>` object to which the `slice_array` object refers.

Multiply & Assign Operator *=

Prototype `void operator*= (const valarray<T>&) const;`

Divide & Assign Operator /=

Prototype `void operator/= (const valarray<T>&) const;`

Remainder & Assign Operator %=

Prototype `void operator%= (const valarray<T>&) const;`

Add & Assign Operator +=

Prototype `void operator+= (const valarray<T>&) const;`

Minus & Assign Operator -=

Prototype `void operator-= (const valarray<T>&) const;`

XOR & Assign Operator ^=

Prototype `void operator^= (const valarray<T>&) const;`

And & Assign Operator &=

Prototype `void operator&= (const valarray<T>&) const;`

Or & Assign Operator |=

Prototype `void operator|= (const valarray<T>&) const;`

Left Shift & Assign Operator <<=

Prototype `void operator<<=(const valarray<T>&) const;`

Right shift & Assign Operator >>=

Prototype `void operator>>=(const valarray<T>&) const;`

Public Member Function slice_array

slice_array::fill

Description This function has reference semantics, assigning the value of its argument to the elements of the `valarray<T>` object to which the `slice_array` object refers.

Prototype `void fill(const T&);`

Class gslice

Description This class represents a generalized slice out of an array. A `gslice` is defined by a starting offset (s), a set of lengths (l_j), and a set of strides (d_j). The number of lengths must equal the number of strides. A `gslice` represents a mapping from a set of indices (i_j), equal in number to the number of strides, to a single index k . It is useful for building multidimensional array classes using the `valarray` template, which is one-dimensional. The set of one-dimensional index values specified by a `gslice` are $k = s + \sum i_j d_j$ where the multidimensional indices i_j range in value from 0 to $l_j - 1$. It is possible to have degenerate generalized slices in which an address is repeated. If a degenerate slice is used as the argument to the non-const version of `operator[](const gslice&)`, the resulting behavior is undefined.

The topics in this section are:

- [“Constructors gslice” on page 404](#)
- [“Access Functions gslice” on page 405](#)

Constructors gslice

Default Constructor

Description The default constructor creates a `gslice` which specifies no elements.

Prototype `gslice ();`

Overloaded Constructors

Description The constructor with arguments builds a `gslice` based on a specification of start, lengths, and strides, as explained in the previous section.

Prototype

```
gslice (size_t start,  
        const valarray<size_t>& lengths,  
        const valarray<size_t>& strides);  
gslice (const gslice&);
```

Access Functions `gslice`

Description These access functions return the representation of the start, lengths, or strides specified for the `gslice`.

`gslice::start`

Prototype `size_t start() const;`

`gslice::length`

Prototype `valarray<size_t> length() const;`

`gslice::stride`

Prototype `valarray<size_t> stride() const;`

Template Class `gslice_array`

Description This template is a helper template used by the slice subscript operator `gslice_array<T> valarray<T>::operator[] (const gslice&);` It has reference semantics to a subset of an array specified by a `gslice` object. Thus, the expression `a[gslice(1, length, stride)] = b` has the effect of assigning the elements of `b` to a generalized slice of the elements in `a`.

The topics in this section are:

- [“Constructors *gslice_array*” on page 406](#)
- [“Assignment *gslice_array*” on page 406](#)
- [“Computed Assignment *gslice_array*” on page 406](#)
- [“Public Member Function *gslice_array*” on page 408](#)—the `fill` function

Constructors *gslice_array*

Default and Copy Constructors

Description The *gslice_array* template has no public constructors. It declares the constructor to be private.

Prototype `gslice_array();`
`gslice_array(const gslice_array&);`

Assignment *gslice_array*

Assignment Operator =

Description The second of these two assignment operators is declared private and need not be defined. The first has reference semantics, assigning the values of the argument array elements to selected elements of the `valarray<T>` object to which the *gslice_array* refers.

Prototype `void operator=(const valarray<T>&) const;`
`gslice_array& operator=(const gslice_array&);`

Computed Assignment *gslice_array*

Description These computed assignments have reference semantics, applying the indicated operation to the elements of the argument array and selected elements of the `valarray<T>` object to which the *gslice_array* object refers.

Multiply & Assign Operator *=

Prototype `void operator*= (const valarray<T>&) const;`

Divide & Assign Operator /=

Prototype `void operator/= (const valarray<T>&) const;`

Remainder & Assign Operator %=

Prototype `void operator%= (const valarray<T>&) const;`

Add & Assign Operator +=

Prototype `void operator+= (const valarray<T>&) const;`

Minus & Assign Operator -=

Prototype `void operator-= (const valarray<T>&) const;`

XOR & Assign Operator ^=

Prototype `void operator^= (const valarray<T>&) const;`

And & Assign Operator &=

Prototype `void operator&= (const valarray<T>&) const;`

Or & Assign Operator |=

Prototype `void operator|= (const valarray<T>&) const;`

Left Shift & Assign Operator <<=

Prototype `void operator<<=(const valarray<T>&) const;`

Right Shift & Assign Operator >>=

Prototype `void operator>>=(const valarray<T>&) const;`

Public Member Function gslice_array

gslice_array::fill

Description This function has reference semantics, assigning the value of its argument to the elements of the `valarray<T>` object to which the `gslice_array` object refers.

Prototype `void fill(const T&);`

Template Class mask_array

Description This template is a helper template used by the mask subscript operator: `mask_array<T> valarray<T>::operator[](const valarray<bool>&)`. It has reference semantics to a subset of an array specified by a `boolean` mask. Thus, the expression `a[mask] = b;` has the effect of assigning the elements of `b` to the masked elements in `a` (those for which the corresponding element in `mask` is true).

The topics in this section are:

- [“Constructors mask_array” on page 409](#)
- [“Assignment mask_array” on page 409](#)
- [“Computed Assignment mask_array” on page 409](#)
- [“Public Member Function mask_array” on page 411](#)—the `fill` function

Constructors mask_array

Constructors

Description The `mask_array` template has no public constructors. It declares the above constructors to be private. These constructors need not be defined.

Prototype `mask_array();`
`mask_array(const mask_array&);`

Assignment mask_array

Assignment Operator =

Description The second of these two assignment operators is declared private and need not be defined. The first has reference semantics, assigning the values of the argument array elements to selected elements of the `valarray<T>` object to which it refers.

Prototype `void operator= (const valarray<T>&) const;`
`mask_array& operator= (const mask_array&);`

Computed Assignment mask_array

Description These computed assignments have reference semantics, applying the indicated operation to the elements of the argument array and selected elements of the `valarray<T>` object to which the mask object refers.

Multiply & Assign Operator *=

Prototype `void operator*= (const valarray<T>&) const;`

Divide & Assign Operator /=

Prototype `void operator/= (const valarray<T>&) const;`

Remainder & Assign Operator %=

Prototype `void operator%= (const valarray<T>&) const;`

Add & Assign Operator +=

Prototype `void operator+= (const valarray<T>&) const;`

Minus & Assign Operator -=

Prototype `void operator-= (const valarray<T>&) const;`

XOR & Assign Operator ^=

Prototype `void operator^= (const valarray<T>&) const;`

And & Assign Operator &=

Prototype `void operator&= (const valarray<T>&) const;`

Or & Assign Operator |=

Prototype `void operator|= (const valarray<T>&) const;`

Left Shift & Assign Operator <<=

Prototype `void operator<<=(const valarray<T>&) const;`

Right Shift & Assign Operator >>=

Prototype `void operator>>=(const valarray<T>&) const;`

Public Member Function mask_array

mask_array::fill

Prototype `void fill(const T&);`

Description This function has reference semantics, assigning the value of its argument to the elements of the `valarray<T>` object to which the `mask_array` object refers.

Template Class indirect_array

Description This template is a helper template used by the indirect subscript operator `indirect_array<T> valarray<T>::operator[](const valarray<int>&)`. It has reference semantics to a subset of an array specified by an `indirect_array`. Thus the expression `a[indirect] = b`; has the effect of assigning the elements of `b` to the elements in `a` whose indices appear in `indirect`.

The topics in this section are:

- [“Constructors indirect_array” on page 411](#)
- [“Assignment indirect_array” on page 412](#)
- [“Computed Assignment indirect_array” on page 412](#)
- [“Public Member Function indirect_array” on page 413](#)—the `fill` function

Constructors indirect_array

Default and Copy Constructors

Description The `indirect_array` template has no public constructors. The constructors listed above are private. These constructors need not be defined.

Prototype `indirect_array();`
`indirect_array(const indirect_array&);`

Assignment indirect_array

Assignment Operator =

Description The second of these two assignment operators is declared private and need not be defined. The first has reference semantics, assigning the values of the argument array elements to selected elements of the `valarray<T>` object to which it refers. If the `indirect_array` specifies an element in the `valarray<T>` object to which it refers more than once, the behavior is undefined.

Prototype

```
void operator=
    (const valarray<T>&) const;
indirect_array& operator=
    (const indirect_array&);
```

Computed Assignment indirect_array

Description These computed assignments have reference semantics, applying the indicated operation to the elements of the argument array and selected elements of the `valarray<T>` object to which the `indirect_array` object refers. If the `indirect_array` specifies an element in the `valarray<T>` object to which it refers more than once, the behavior is undefined.

Multiply & Assign Operator *=

Prototype

```
void operator*= (const valarray<T>&) const;
```

Divide & Assign Operator /=

Prototype

```
void operator/= (const valarray<T>&) const;
```

Remainder & Assign Operator %=

Prototype

```
void operator%= (const valarray<T>&) const;
```

Add & Assign Operator +=

Prototype `void operator+= (const valarray<T>&) const;`

Minus & Assign Operator -=

Prototype `void operator-= (const valarray<T>&) const;`

XOR & Assign Operator ^=

Prototype `void operator^= (const valarray<T>&) const;`

And & Assign Operator &=

Prototype `void operator&= (const valarray<T>&) const;`

Or & Assign Operator |=

Prototype `void operator|= (const valarray<T>&) const;`

Left Shift & Assign Operator <<=

Prototype `void operator<<=(const valarray<T>&) const;`

Right Shift & Assign Operator >>=

Prototype `void operator>>=(const valarray<T>&) const;`

Public Member Function indirect_array

indirect_array::fill

Description This function has reference semantics, assigning the value of its argument to the elements of the `valarray<T>` object to which the `indirect_array` object refers.

Prototype `void fill(const T&);`

Generalized Numeric Operations

The classes are:

- [“Template Class accumulate” on page 414](#)
- [“Template Class inner_product” on page 415](#)
- [“Template Class partial_sum” on page 415](#)
- [“Template Class adjacent_difference” on page 416](#)

See also [“Generalized Numeric Algorithms” on page 366](#).

Template Class accumulate

Description Initializes the accumulator `acc` with the initial value `init` and then modifies it with `acc = acc + *i` or `acc = binary_op(acc, *i)` for every iterator `i` in the range `[first, last)` in order. This function requires that `binary_op` shall not cause side effects.

Prototype

```
template
    < class InputIterator,
      class T >
T accumulate
    (InputIterator first,
     InputIterator last,
     T init);

template
    < class InputIterator,
      class T,
      class BinaryOperation >
T accumulate
    (InputIterator first,
     InputIterator last,
     T init,
     BinaryOperation binary_op);
```

Template Class inner_product

Description Computes its result by initializing the accumulator `acc` with the initial value `init` and then modifying it with `acc = acc + (*i1) * (*i2)` or `acc = binary_op1(acc, binary_op2(*i1, *i2))` for every iterator `i1` in the range `[first, last)` and iterator `i2` in the range `[first2, first2 + (last - first))` in order. This function requires that `binary_op1` and `binary_op2` shall not cause side effects.

Prototype

```
template
    < class InputIterator1,
      class InputIterator2,
      class T >
T inner_product
    (InputIterator1 first1,
     InputIterator1 last1,
     InputIterator2 first2,
     T init);

template
    < class InputIterator1,
      class InputIterator2,
      class T,
      class BinaryOperation1,
      class BinaryOperation2 >
T inner_product
    (InputIterator1 first1,
     InputIterator1 last1,
     InputIterator2 first2, T init,
     BinaryOperation1 binary_op1,
     BinaryOperation2 binary_op2);
```

Template Class partial_sum

Description Assigns to every iterator `i` in the range `[result, result + (last - first))` a value correspondingly equal to `((...(*first + *(first + 1)) + ...) + *(first + (i - result)))` or `binary_op(binary_op(...,`

Numerics Library

Template Class *adjacent_difference*

`binary_op(*first, *(first + 1), ...), *(first + (i - result)))`. This function returns `result + (last - first)`. The complexity of this function is exactly `(last - first) - 1` applications of `binary_op`. This function requires `binary_op` is expected not to have any side effects. The result may be equal to `first`.

Prototype

```
template
    < class InputIterator,
      class OutputIterator >
OutputIterator partial_sum
    (InputIterator first,
     InputIterator last,
     OutputIterator result);
template
    < class InputIterator,
      class OutputIterator,
      class BinaryOperation >
OutputIterator partial_sum
    (InputIterator first,
     InputIterator last,
     OutputIterator result,
     BinaryOperation binary_op);
```

Template Class *adjacent_difference*

Description Assigns to every element referred to by iterator `i` in the range `[result + 1, result + (last - first))` a value correspondingly equal to `*(first + (i - result)) - *(first + (i - result) - 1)` or `binary_op(*(first + (i - result)), *(first + (i - result) - 1))`. `Result` gets the value of `*first`. This function requires `binary_op` shall not have any side effects. `Result` may be equal to `first`. This function returns `result + (last - first)`. The complexity of this function is exactly `(last - first) - 1` applications of `binary_op`.

Prototype

```
template
    < class InputIterator,
      class OutputIterator >
```



```
OutputIterator adjacent_difference
    (InputIterator first,
     InputIterator last,
     OutputIterator result);
template
    < class InputIterator,
      class OutputIterator,
      class BinaryOperation >
OutputIterator adjacent_difference
    (InputIterator first,
     InputIterator last,
     OutputIterator result,
     BinaryOperation binary_op);
```

Numerics Library

Template Class adjacent_difference



26.2 Complex Class

The header `<complex>` defines a template class, and facilities for representing and manipulating complex numbers.

Overview of the Complex Class Library

The header `<complex>` defines classes, operators, and functions for representing and manipulating complex numbers

The topics in this section are:

- [“Header `<complex>`” on page 420](#), shows the complex header class declarations
- [“26.2.3 Complex Specializations” on page 424](#), lists the float, double and long double specializations
- [“Complex Template Class” on page 426](#), is a template class for complex numbers.

`_MSL_CX_LIMITED_RANGE`

This flag effects the `*` and `/` operators of complex.

When defined, the “normal” formulas for multiplication and division are used. They may execute faster on some machines. However, infinities will not be properly calculated, and there is more roundoff error potential.

If the flag is undefined (default), then more complicated algorithms (from the C standard) are used which have better overflow and underflow characteristics and properly propagate infinity. Flipping this switch requires recompilation of the C++ library.

26.2 Complex Class

Header <complex>

NOTE: It is recommend that the `ansi_prefix.xxx.h` is the place to define this flag if you want the simpler and faster multiplication and division algorithms.

Header <complex>

The header <complex> defines classes, operators, and functions for representing and manipulating complex numbers

Listing 12.1 Header <complex> forward declarations

```
namespace std {  
    // forward declarations  
    template<class T> class complex;  
    template<> class complex<float>;  
    template<> class complex<double>;  
    template<> class complex<long double>;  
}
```

26.2.6 operators:

```
template<class T> complex<T> operator+  
    (const complex<T>&, const complex<T>&);  
template<class T> complex<T> operator+  
    (const complex<T>&, const T&);  
template<class T> complex<T> operator+  
    (const T&, const complex<T>&);  
template<class T> complex<T> operator-  
    (const complex<T>&, const complex<T>&);  
template<class T> complex<T> operator-  
    (const complex<T>&, const T&);  
template<class T> complex<T> operator-  
    (const T&, const complex<T>&);  
template<class T> complex<T> operator*  
    (const complex<T>&, const complex<T>&);  
template<class T> complex<T> operator*  
    (const complex<T>&, const T&);  
template<class T> complex<T> operator*  
    (const T&, const complex<T>&);
```

```
template<class T> complex<T> operator/
    (const complex<T>&, const complex<T>&);
template<class T> complex<T> operator/
    (const complex<T>&, const T&);
template<class T> complex<T> operator/
    (const T&, const complex<T>&);
template<class T> complex<T> operator+
    (const complex<T>&);
template<class T> complex<T> operator-
    (const complex<T>&);
template<class T> bool operator==
    (const complex<T>&, const complex<T>&);
template<class T> bool operator==
    (const complex<T>&, const T&);
template<class T> bool operator==
    (const T&, const complex<T>&);
template<class T> bool operator!=
    (const complex<T>&, const complex<T>&);
template<class T> bool operator!=
    (const complex<T>&, const T&);
template<class T> bool operator!=
    (const T&, const complex<T>&);
template<class T, class charT, class traits>
basic_istream<charT, traits>& operator>>
    (basic_istream<charT, traits>&, complex<T>&);
template<class T, class charT, class traits>
basic_ostream<charT, traits>& operator<<
    (basic_ostream<charT, traits>&, const complex<T>&);
```

26.2.7 values:

```
template<class T> T real(const complex<T>&);
template<class T> T imag(const complex<T>&);
template<class T> T abs(const complex<T>&);
template<class T> T arg(const complex<T>&);
template<class T> T norm(const complex<T>&);
template<class T> complex<T> conj(const complex<T>&);
template<class T> complex<T> polar(const T&, const T&);
```

26.2 Complex Class

Header `<complex>`

26.2.8 transcendentals:

```
template<class T> complex<T> cos (const complex<T>&);
template<class T> complex<T> cosh (const complex<T>&);
template<class T> complex<T> exp (const complex<T>&);
template<class T> complex<T> log (const complex<T>&);
template<class T> complex<T> log10(const complex<T>&);
template<class T> complex<T> pow(const complex<T>&, int);
template<class T> complex<T> pow(const complex<T>&, const T&);
template<class T> complex<T> pow
    (const complex<T>&, const complex<T>&);
template<class T> complex<T> pow(const T&, const complex<T>&);
template<class T> complex<T> sin (const complex<T>&);
template<class T> complex<T> sinh (const complex<T>&);
template<class T> complex<T> sqrt (const complex<T>&);
template<class T> complex<T> tan (const complex<T>&);
template<class T> complex<T> tanh (const complex<T>&);
}
```

Template Class Complex

```
namespace std {
template<class T>
class complex {
public:
    typedef T value_type;
    complex(const T& re = T(), const T& im = T());
    complex(const complex&);
    template<class X> complex(const complex<X>&);
    T real() const;
    T imag() const;
    complex<T>& operator= (const T&);
    complex<T>& operator+=(const T&);
    complex<T>& operator-=(const T&);
    complex<T>& operator*=(const T&);
    complex<T>& operator/=(const T&);
    complex& operator=(const complex&);
    template<class X> complex<T>& operator= (const complex<X>&);
    template<class X> complex<T>& operator+=(const complex<X>&);
    template<class X> complex<T>& operator-=(const complex<X>&);
};
```

```

template<class X> complex<T>& operator*=(const complex<X>&);
template<class X> complex<T>& operator/=(const complex<X>&);
};
template<class T> complex<T> operator+
    (const complex<T>&, const complex<T>&);
template<class T> complex<T> operator+
    (const complex<T>&, const T&);
template<class T> complex<T> operator+
    (const T&, const complex<T>&);
template<class T> complex<T> operator-
    (const complex<T>&, const complex<T>&);
template<class T> complex<T> operator-
    (const complex<T>&, const T&);
template<class T> complex<T> operator-
    (const T&, const complex<T>&);
template<class T> complex<T> operator*
    (const complex<T>&, const complex<T>&);
template<class T> complex<T> operator*
    (const complex<T>&, const T&);
template<class T> complex<T> operator*
    (const T&, const complex<T>&);
template<class T> complex<T> operator/
    (const complex<T>&, const complex<T>&);
template<class T> complex<T> operator/
    (const complex<T>&, const T&);
template<class T> complex<T> operator/
    (const T&, const complex<T>&);
template<class T> complex<T> operator+
    (const complex<T>&);
template<class T> complex<T> operator-
    (const complex<T>&);
template<class T> bool operator==
    (const complex<T>&, const complex<T>&);
template<class T> bool operator==
    (const complex<T>&, const T&);
template<class T> bool operator==
    (const T&, const complex<T>&);
template<class T> bool operator!=
    (const complex<T>&, const complex<T>&);
template<class T> bool operator!=

```

26.2 Complex Class

26.2.3 Complex Specializations

```
(const complex<T>&, const T&);
template<class T> bool operator!=
    (const T&, const complex<T>&);
template<class T, class charT, class traits>
basic_istream<charT, traits>& operator>>
    (basic_istream<charT, traits>&, complex<T>&);
template<class T, class charT, class traits>
basic_ostream<charT, traits>& operator<<
    (basic_ostream<charT, traits>&, const complex<T>&);
};
```

26.2.3 Complex Specializations

The standard specialize the template complex class for float, double and long double types.

Listing 12.2 Float specializations

```
template<> class complex<float> {
public:
    typedef float value_type;
    complex(float re = 0.0f, float im = 0.0f);
    explicit complex(const complex<double>&);
    explicit complex(const complex<long double>&);
    float real() const;
    float imag() const;
    complex<float>& operator= (float);
    complex<float>& operator+=(float);
    complex<float>& operator-=(float);
    complex<float>& operator*=(float);
    complex<float>& operator/=(float);
    complex<float>& operator=(const complex<float>&);
    template<class X> complex<float>& operator= (const complex<X>&);
    template<class X> complex<float>& operator+=(const complex<X>&);
    template<class X> complex<float>& operator-=(const complex<X>&);
    template<class X> complex<float>& operator*=(const complex<X>&);
    template<class X> complex<float>& operator/=(const complex<X>&);
};
```

Listing 12.3 Double specializations

```
template<> class complex<double> {
public:
    typedef double value_type;
    complex(double re = 0.0, double im = 0.0);
    complex(const complex<float>&);
    explicit complex(const complex<long double>&);
    double real() const;
    double imag() const;
    complex<double>& operator= (double);
    complex<double>& operator+=(double);
    complex<double>& operator-=(double);
    complex<double>& operator*=(double);
    complex<double>& operator/=(double);
    complex<double>& operator=(const complex<double>&);
    template<class X> complex<double>& operator= (const complex<X>&);
    template<class X> complex<double>& operator+=(const complex<X>&);
    template<class X> complex<double>& operator-=(const complex<X>&);
    template<class X> complex<double>& operator*=(const complex<X>&);
    template<class X> complex<double>& operator/=(const complex<X>&);
};
```

Listing 12.4 Long Double specializations

```
template<> class complex<long double> {
public:
    typedef long double value_type;
    complex(long double re = 0.0L, long double im = 0.0L);
    complex(const complex<float>&);
    complex(const complex<double>&);
    long double real() const;
    long double imag() const;
    complex<long double>& operator=(const complex<long double>&);
    complex<long double>& operator= (long double);
    complex<long double>& operator+=(long double);
    complex<long double>& operator-=(long double);
    complex<long double>& operator*=(long double);
    complex<long double>& operator/=(long double);
    template<class X> complex<long double>& operator=
```

26.2 Complex Class

Complex Template Class

```
(const complex<X>&);  
template<class X> complex<long double>& operator+=  
(const complex<X>&);  
template<class X> complex<long double>& operator-=  
(const complex<X>&);  
template<class X> complex<long double>& operator*=  
(const complex<X>&);  
template<class X> complex<long double>& operator/=  
(const complex<X>&);  
};
```

Complex Template Class

The template class `complex` contains Cartesian components `real` and `imag` for a complex number.

Remarks The effect of instantiating the template `complex` for any type other than `float`, `double` or `long double` is unspecified.

If the result of a function is not mathematically defined or not in the range of representable values for its type, the behavior is undefined.

The `complex` class consists of:

- [“Constructors and Assignments” on page 426.](#)
- [“Complex Member Functions” on page 427.](#)
- [“Operators” on page 427.](#)

Constructors and Assignments

Constructor, destructor and assignment operators and functions.

Constructors

Description Construct an object of a complex class.

Prototype `complex(const T& re = T(), const T& im = T());`
`complex(const complex&);`

```
template<class X> complex(const complex<X>&);
```

Remarks After construction real equal re and imag equals im.

Assignment Operator

Description An assignment operator for complex classes.

Prototype

```
complex<T>& operator= (const T&);  
complex& operator= (const complex&);  
template<class X> complex<T>& operator=  
    (const complex<X>&);
```

Remarks Assigns a floating point type to the Cartesian complex class.

Complex Member Functions

There are two public member functions

- [“real” on page 427](#)
- [“imag” on page 427](#)

real

Description Retrieves the real component.

Prototype `T real() const;`

imag

Description Retrieves the imag component.

Prototype `T imag() const;`

Operators

Several assignment operators are overloaded for the complex class manipulations.

- [“operator +=” on page 428](#)

26.2 Complex Class

Complex Template Class

- [“operator -=” on page 428](#)
- [“operator *=” on page 429](#)
- [“operator /=” on page 429](#)

operator +=

Description Adds and assigns to a complex class.

Prototype

```
complex<T>& operator+=(const T&);  
template<class X> complex<T>& operator+=  
    (const complex<X>&);
```

Remarks The first operator with a scalar argument adds the scalar value of the right hand side to the real component and stores the result in the object. The imaginary component is left alone.

The second operator with a complex type, adds the complex value of the right hand side to the object and stores the resultant in the object.

Returns The `this` pointer is returned.

operator -=

Description Subtracts and assigns from a complex class.

Prototype

```
complex<T>& operator-=(const T&);  
template<class X> complex<T>& operator-=  
    (const complex<X>&);
```

Remarks The first operator with a scalar argument subtracts the scalar value of the right hand side from the real component and stores the result in the object. The imaginary component is left alone.

The second operator with a complex type, subtracts the complex value of the right hand side from the object and stores the resultant in the object.

Returns The `this` pointer is returned.

operator *=

Description	Multiplies by and assigns to a complex class.
Prototype	<pre>complex<T>& operator*=(const T&); template<class X> complex<T>& operator*= (const complex<X>&);</pre>
Remarks	<p>The first operator with a scalar argument multiplies the scalar value of the right hand side to class object and stores result in the object.</p> <p>The second operator with a complex type, multiplies the complex value of the right hand side to the object and stores the resultant in the object.</p>
Returns	The <code>this</code> pointer is returned.

operator /=

Description	Divides by and assigns to a complex class.
Prototype	<pre>complex<T>& operator/=(const T&); template<class X> complex<T>& operator/= (const complex<X>&);</pre>
Remarks	<p>The first operator with a scalar argument divides the scalar value of the right hand side to class object and stores result in the object.</p> <p>The second operator with a complex type, divides the complex value of the right hand side into the object and stores the resultant in the object.</p>
Returns	The <code>this</code> pointer is returned.

Overloaded Operators and Functions

There are several non member functions and overloaded operators in the complex class library.

["Complex Operators" on page 430](#)

26.2 Complex Class

Complex Template Class

[“Complex Value Operations” on page 433](#)

[“Complex Transcendentals” on page 435](#)

Complex Operators

The overloaded complex operators consists of:

- [“operator +” on page 430](#)
- [“operator -” on page 430](#)
- [“operator /” on page 431](#)
- [“operator !=” on page 432](#)
- [“operator >>” on page 432](#)
- [“operator <<” on page 433](#)

operator +

Description Adds to the complex class.

Prototype

```
template<class T> complex<T> operator+
    (const complex<T>&, const complex<T>&);
template<class T> complex<T> operator+
    (const complex<T>&, const T&);
template<class T> complex<T> operator+
    (const T&, const complex<T>&);

template<class T> complex<T> operator+
    (const complex<T>&);
```

Remarks The addition performs an += operation.

Returns The complex class after the addition.

operator -

Description Subtracts from the complex class.

Prototype

```
template<class T> complex<T> operator-
    (const complex<T>&, const complex<T>&);
```

```
template<class T> complex<T> operator-
    (const complex<T>&, const T&);
template<class T> complex<T> operator-
    (const T&, const complex<T>&);

template<class T> complex<T> operator-
    (const complex<T>&);
```

Remarks The subtraction performs a -= operation.

Returns The complex class after the Subtraction.

operator *

Description Multiplies the complex class.

Prototype

```
template<class T> complex<T> operator*
    (const complex<T>&, const complex<T>&);
template<class T> complex<T> operator*
    (const complex<T>&, const T&);
template<class T> complex<T> operator*
    (const T&, const complex<T>&);
```

Remarks The multiplication performs a *= operation.

Returns The complex class after the multiplication.

operator /

Description Divides from the complex class.

Prototype

```
template<class T> complex<T> operator/
    (const complex<T>&, const complex<T>&);
template<class T> complex<T> operator/
    (const complex<T>&, const T&);
template<class T> complex<T> operator/
    (const T&, const complex<T>&);
```

Remarks The division performs an /= operation.

26.2 Complex Class

Complex Template Class

Returns The complex class after the division.

operator ==

Description A boolean equality comparison.

Prototype

```
template<class T> bool operator==  
    (const complex<T>&, const complex<T>&);  
template<class T> bool operator==  
    (const complex<T>&, const T&);  
template<class T> bool operator==  
    (const T&, const complex<T>&);
```

Remarks Returns true if the real and the imaginary components are equal.

operator !=

Description A boolean non equality comparison.

Prototype

```
template<class T> bool operator!=  
    (const complex<T>&, const complex<T>&);  
template<class T> bool operator!=  
    (const complex<T>&, const T&);  
template<class T> bool operator!=  
    (const T&, const complex<T>&);
```

Remarks Returns true if the real or the imaginary components are not equal.

operator >>

Description Extracts a complex type from a stream.

Prototype

```
template<class T, class charT, class traits>  
basic_istream<charT, traits>& operator>>  
    (basic_istream<charT, traits>&, complex<T>&);
```

Remarks Extracts in the form of u, (u), or (u,v) where u is the real part and v is the imaginary part.

Any failure in extraction will set the failbit and result in undefined behavior.

operator <<

Description Inserts a complex number into a stream.

Prototype `template<class T, class charT, class traits>
basic_ostream<charT, traits>& operator<<
 (basic_ostream<charT, traits>&
 const complex<T>&);`

Complex Value Operations

The complex value operations consists of:

- [“real” on page 433](#)
- [“imag” on page 433](#)
- [“abs” on page 434](#)
- [“arg” on page 434](#)
- [“norm” on page 434](#)
- [“conj” on page 434](#)
- [“polar” on page 435](#)

real

Description Retrieves the real component of a complex class.

Prototype `template<class T>
T real(const complex<T>&);`

Remarks Returns the real component of the argument.

imag

Description Retrieves the imaginary component of a complex class.

Prototype `template<class T>`

26.2 Complex Class

Complex Template Class

```
T imag(const complex<T>&);
```

Remarks Returns the imaginary component of the argument.

abs

Description Determines the absolute value of a complex class.

Prototype

```
template<class T>
    T abs(const complex<T>&);
```

Remarks Returns the absolute value of the complex class argument.

arg

Description Determines the phase angle.

Prototype

```
template<class T>
    T arg(const complex<T>&);
```

Remarks Returns the phase angle of the complex class argument or `atan2(imag(x), real(x))`.

norm

Description Determines the squared magnitude.

Prototype

```
template<class T>
    T norm(const complex<T>&);
```

Remarks The squared magnitude of the complex class.

conj

Description Determines the complex conjugate.

Prototype

```
template<class T>
    complex<T> conj(const complex<T>&);
```

Remarks Returns the complex conjugate of the complex class argument.

polar

Description	Determines the polar coordinates.
Prototype	<pre>template<class T> complex<T> polar(const T&, const T&);</pre>
Remarks	Returns the complex value corresponding to a complex number whose magnitude is the first argument and whose phase angle is the second argument.

Complex Transcendentals

The complex transcendentals consists of:

- [“cos” on page 435](#)
- [“cosh” on page 436](#)
- [“exp” on page 436](#)
- [“log” on page 436](#)
- [“log10” on page 436](#)
- [“pow” on page 437](#)
- [“sin” on page 437](#)
- [“sinh” on page 437](#)
- [“sqrt” on page 437](#)
- [“tan” on page 438](#)
- [“tanh” on page 438](#)

cos

Description	Determines the cosine.
Prototype	<pre>template<class T> complex<T> cos (const complex<T>&);</pre>
Remarks	Returns the cosine of the complex class argument.

26.2 Complex Class

Complex Template Class

cosh

Description Determines the hyperbolic cosine.

Prototype `template<class T>
complex<T> cosh (const complex<T>&);`

Remarks Returns the cosine of the complex class argument.

exp

Description Determines the exponential.

Prototype `template<class T>
complex<T> exp (const complex<T>&);`

Remarks Returns the base exponential of the complex class argument.

log

Description Determines the natural base logarithm.

Prototype `template<class T>
complex<T> log (const complex<T>&);`

Remarks Returns the natural base logarithm of the complex class argument, in the range of a strip mathematically unbounded along the real axis and in the interval of $[-i\pi, i\pi]$ along the imaginary axis. Where the argument is a negative real number, $\text{imag}(\log(\text{cpx}))$, is π .

log10

Description Determines the logarithm to base ten.

Prototype `template<class T>
complex<T> log10(const complex<T>&);`

Remarks Returns the logarithm base(10) of the argument `cpx` defined as $\log(\text{cpx}) / \log(10)$.

pow

Description Raises the complex class to a set power.

Prototype

```
template<class T>
    complex<T> pow(const complex<T>&, int);
template<class T>
    complex<T> pow(const complex<T>&, const T&);
template<class T> complex<T> pow
    (const complex<T>&, const complex<T>&);
template<class T>
    complex<T> pow(const T&, const complex<T>&);
```

Remarks Returns the complex class raised to the power of second argument defined as the exponent of (the second argument times the log of the first argument).

The value for `pow(0,0)` will return (nan, nan).

sin

Description Determines the sine.

Prototype

```
template<class T>
    complex<T> sin (const complex<T>&);
```

Remarks Returns the sine of the complex class argument.

sinh

Description Determines the hyperbolic sine.

Prototype

```
template<class T>
    complex<T> sinh (const complex<T>&);
```

Remarks Returns the hyperbolic sine of the complex class argument.

sqrt

Description Determines the square root.

26.2 Complex Class

Complex Template Class

Prototype `template<class T>
 complex<T> sqrt (const complex<T>&);`

Remarks Returns the square root of the complex class argument in the range of right half plane. If the argument is a negative real number, the value returned lies on the positive imaginary axis.

tan

Description Determines the tangent.

Prototype `template<class T>
 complex<T> tan (const complex<T>&);`

Remarks Returns the tangent of the complex class argument.

tanh

Description Determines the hyperbolic tangent.

Prototype `template<class T>
 complex<T> tanh (const complex<T>&);`

Remarks Returns the hyperbolic tangent of the complex class argument.



27.1 Input and Output Library

A listing of the set of components that C++ programs may use to perform input/output operations.

Overview of Input and Output Library

The sections in this chapter are:

- [“Input and Output Library Summary” on page 439](#)
- [“27.1 Iostreams requirements” on page 440](#)

Input and Output Library Summary

This library includes the headers.

Table 13.1 **Input/Output Library Summary**

Include	Purpose
<code><iosfwd></code>	Forward declarations
<code><iostream></code>	Standard iostream objects
<code><ios></code>	Iostream base classes
<code><streambuf></code>	Stream buffers
<code><istream></code>	Formatting and manipulators
<code><ostream></code>	
<code><iomanip></code>	
<code><sstream></code>	String streams

27.1 Input and Output Library

27.1 *Iostreams requirements*

Include	Purpose
<code><cstdlib></code>	
<code><fstream></code>	File Streams
<code><cstdio></code>	
<code><wchar></code>	

27.1 *iostreams requirements*

No requirements library has been defined.

Topics in this section are:

- [“27.1.1 Definitions” on page 440](#)
- [“27.1.2 Type requirements” on page 441](#)
- [“27.1.2.5 Type SZ T” on page 441](#)

27.1.1 Definitions

Additional definitions are:

- `character` - A unit that can represent text
- `character container type` - A class or type used to represent a character.
- `iostream class templates` - A templates that take two arguments: `charT` and `traits`. The argument `charT` is a character container type. The argument `traits` is a structure which defines characteristics and functions of the `charT` type.
- `narrow-oriented iostream classes` - These classes are template instantiation classes. The traditional `iostream` classes are narrow-oriented `iostream` classes.
- `wide-oriented iostream classes` - These classes are template instantiation classes. They are used for the character container class `wchar_t`.
- `repositional streams and arbitrary-positional streams` - A repositional stream can seek to only a pre-

viously encountered position. An arbitrary-positional stream can integral position within the length of the stream.

27.1.2 Type requirements

Several types are required by the standards, they are consolidated in strings (chapter 21.)

27.1.2.5 Type `SZ_T`

A type that represents one of the signed basic integral types. It is used to represent the number of characters transferred in and input/output operation or for the size of the input/output buffers.

27.1 Input and Output Library

27.1 *Iostreams requirements*



27.2 Forward Declarations

The header `<iosfwd>` is used for forward declarations of template classes.

Overview of Input and Output Streams Forward Declarations.

The ANSI/ISO standard calls for forward declarations of input and output streams for basic input and output, basic input and basic output. This is for both normal and wide character formats.

Header `<iosfwd>`

Prototype

```
namespace std {
    template<class charT> class basic_ios;
    template<class charT> class basic_istream;
    template<class charT> class basic_ostream;

    typedef basic_ios<char> ios;
    typedef basic_ios<wchar> wios;

    typedef basic_istream<char> istream;
    typedef basic_istream<wchar_t> istream;

    typedef basic_ostream<char> ostream;
    typedef basic_ostream<char> wostream;
}
```

Remarks The template class `basic_ios<charT, traits>` serves as a base class for class `basic_istream` and `basic_ostream`.

27.2 Forward Declarations

Header `<iosfwd>`

The class `ios` is an instantiation of `basic_ios` specialized by the type `char`.

The class `wios` is an instantiation of `basic_ios` specialized by the type `wchar_t`.



27.3 Standard Iostream Objects

The include header `<iostream>` declared input and output stream objects. The declared objects are associated with the standard C streams provided for by the functions in `<stdio>`.

Overview of Standard Input and Output Stream Objects.

The ANSI/ISO standard calls for predetermined objects for standard input, output, logging and error reporting. This is initialized for normal and wide character formats.

Header `<iostream>`

Description Declaration of standard objects

Prototype

```
Header <iostream>
namespace std{
extern istream cin;
extern ostream cout;
extern ostream cerr;
extern ostream clog;

extern wistream wcin;
extern wostream wcout;
extern wostream cerr;
extern wostream wclog;
}
```

Additional topics are:

27.3 Standard Iostream Objects

Header `<iostream>`

- [“27.3.1 Narrow stream objects” on page 446](#)
- [“27.3.2 Wide stream objects” on page 447](#)

27.3.1 Narrow stream objects

Description Narrow stream objects provide unbuffered input and output associated with standard input and output declared in `<cstdio>`.

istream cin

Description An unbuffered input stream.

Prototype `istream cin;`

Remarks The object `cin` controls input from an unbuffered stream buffer associated with `stdin` declared in `<cstdio>`. After `cin` is initialized `cin.tie()` returns `cout`.

Return An `istream` object;

ostream cout

Description An unbuffered output stream.

Prototype `ostream cout;`

Remarks The object `cout` controls output to an unbuffered stream buffer associated with `stdout` declared in `<cstdio>`.

Return An `ostream` object;

ostream cerr

Description Controls output to an unbuffered stream.

Prototype `ostream cerr;`

Remarks The object `cerr` controls output to an unbuffered stream buffer associated with `stderr` declared in `<cstdio>`. After `err` is initialized, `err.flags()` and `unitbuf` is nonzero.

Return An ostream object;

ostream clog

Description Controls output to a stream buffer.

Prototype `ostream clog;`

Remarks The object `clog` controls output to a stream buffer associated with `cerr` declared in `<cstdio>`.

Return An ostream object;

27.3.2 Wide stream objects

Description Narrow stream objects provide unbuffered input and output associated with standard input and output declared in `<cstdio>`.

istream win

Description An unbuffered input stream.

Prototype `wistream win;`

Remarks The object `win` controls input from an unbuffered stream buffer associated with `stdin` declared in `<cstdio>`. After `cin` is initialized `win.tie()` returns `wout`.

Return An wistream object;

ostream wout

Description An unbuffered output stream.

27.3 Standard Iostream Objects

Header `<iostream>`

Prototype `wostream wout;`

Remarks The object `cout` controls output to an unbuffered stream buffer associated with `stdout` declared in `<cstdio>`.

Return An `wostream` object;

wostream werr

Description Controls output to an unbuffered stream.

Prototype `wostream werr;`

Remarks The object `werr` controls output to an unbuffered stream buffer associated with `stderr` declared in `<cstdio>`. After `werr` is initialized, `werr.flags()` and `unitbuf` is nonzero.

Return An `ostream` object;

wostream wlog

Description Controls output to a stream buffer.

Prototype `wostream wlog;`

Remarks The object `wlog` controls output to a stream buffer associated with `cerr` declared in `<cstdio>`.

Return An `ostream` object



27.4 `iostreams` Base Classes

The include header `<iostream>` contains the basic class definitions, types, and enumerations necessary for input and output stream reading writing and other manipulations.

Overview of Input and Output Stream Base Classes

The sections in this chapter are:

- [“Header `<iostream>`” on page 449](#)
- [“27.4.1 Typedef Declarations” on page 450](#)
- [“27.4.3 Class `ios_base`” on page 451](#)
- [“27.4.4 Template class `basic_iostream`” on page 468](#)
- [“27.4.5 `ios_base` manipulators” on page 486](#)

Header `<iostream>`

Description The header file `<iostream>` provides for implementation of stream objects for standard input and output.

Prototype Header `<iostream>`

```
typedef OFF_T streamoff;
typedef SZ_T streamsize;

class ios_base;
template <class charT, class traits =
ios_traits<charT> >
```

27.4 Iostreams Base Classes

27.4.1 Typedef Declarations

```
class basic_ios

typedef basic_ios<char> ios;
typedef basic_ios<wchar_t> wios;

ios_base& boolalpha (ios_base& str)
ios_base& noboolalpha (ios_base& str)

ios_base& showbase (ios_base& str)
ios_base& noshowbase (ios_base& str)

ios_base& showpoint (ios_base& str)
ios_base& noshowpoint (ios_base& str)

ios_base& showpos (ios_base& str)
ios_base& noshowpos (ios_base& str)

ios_base& skipws (ios_base& str)
ios_base& noskipws (ios_base& str)

ios_base& uppercase (ios_base& str)
ios_base& nouppercase (ios_base& str)

ios_base& internal (ios_base& str)
ios_base& left (ios_base& str)
ios_base& right (ios_base& str)

ios_base& dec (ios_base& str)
ios_base& hex (ios_base& str)
ios_base& oct (ios_base& str)

ios_base& fixed (ios_base& str)
ios_base& scientific (ios_base& str)
```

27.4.1 Typedef Declarations

Description The following typedef's are defined in the class `ios_base`.

Definition `typedef OFF_T wstreamoff;`

Definition `typedef POS_T wstreampos;`

Definition `typedef SZ_T streamsize;`

27.4.3 Class ios_base

Description A base class for input and output stream mechanisms

The prototype is listed below. Additional topics in this section are:

- [“27.4.3.1 Typedef Declarations” on page 453](#)
- [“27.4.3.1.1 failure” on page 453](#)
- [“27.4.3.1.1.1 failure” on page 454](#)
- [“27.4.3.1.2 Type fmtflags” on page 454](#)
- [“27.4.3.1.3 Type iostate” on page 456](#)
- [“27.4.3.1.4 Type openmode” on page 456](#)
- [“27.4.3.1.5 Type seekdir” on page 457](#)
- [“27.4.3.1.6 Class Init” on page 457](#)
- [“Class Init Constructor” on page 457](#)
- [“27.4.3.2 ios_base fmtflags state functions” on page 458](#)
- [“27.4.3.3 ios_base locale functions” on page 465](#)
- [“27.4.3.4 ios_base storage function” on page 466](#)
- [“27.4.3.5 ios_base Constructor” on page 468](#)

Prototype

```
namespace std{
    class ios_base{
    public: class failure;

    typedef T1 fmtflags;
    static const fmtflags boolalpha;
    static const fmtflags dec;
    static const fmtflags fixed;
    static const fmtflags hex;
    static const fmtflags internal;
```

27.4 Iostreams Base Classes

27.4.3 Class *ios_base*

```
static const fmtflags left;
static const fmtflags oct;
static const fmtflags right;
static const fmtflags scientific;
static const fmtflags showbase;
static const fmtflags showpoint;
static const fmtflags showpos;
static const fmtflags skipws;
static const fmtflags unitbuf;
static const fmtflags uppercase;
static const fmtflags adjustfield;
static const fmtflags basefield;
static const fmtflags floatfield;

typedef T2 iostate;
static const iostate badbit;
static const iostate eofbit;
static const iostate failbit;
static const iostate goodbit;

typedef T3 openmode;
static const openmode app;
static const openmode ate;
static const openmode binary;
static const openmode in;
static const openmode out;
static const openmode trunc;

typedef T4 seekdir;
static const seekdir beg;
static const seekdir cur;
static const seekdir end;

class Init;
fmtflags flags() const;
fmtflags flags(fmtflags fmtfl);
fmtflags setf(fmtflags fmtfl);
fmtflags setf(fmtflags fmtfl, fmtflags mask);
void unsetf(fmtflags mask);
streamsize precision() const;
```

```

streamsize precision(streamsize prec);
streamsize width() const;
streamsize width(streamsize wide);
locale imbue(const locale& loc);
locale getloc() const;
static int xalloc();
long&  iword(int index);
void*& pword(int index);

~ios_base();

enum event { erase_event, imbue_event, copyfmt_event };
typedef void (*event_callback)(event, ios_base&, int index);
void register_callback(event_call_back fn, int index);
static bool sync_with_stdio(bool sync = true);

protected:
    ios_base();

private:
    // static int index;    exposition only
    // long* iarray;        exposition only
    // void** parray;       exposition only
};
}

```

Remarks The `ios_base` class is a base class and includes many enumerations and mechanisms necessary for input and output operations.

27.4.3.1 Typedef Declarations

No types are specified in the current standards.

27.4.3.1.1 failure

Description Define a base class for types of object thrown as exceptions.

Prototype `namespace std {
 class ios_base::failure : public exception {`

27.4 Iostreams Base Classes

27.4.3 Class *ios_base*

```
    public:  
        explicit failure(const string&)  
        virtual ~failure();  
        virtual const char* what() const;  
    };  
}
```

27.4.3.1.1 failure

Description Construct a class failure.

Prototype `explicit failure(const string& msg);`

Remarks The function `failure()` construct a class failure initializing with `exception(msg)`.

failure::what

Description To return the exception message.

Prototype `const char *what() const;`

Remarks The function `what()` is use to deliver the `msg.str()`.

Returns Returns the message with which the exception was created.

27.4.3.1.2 Type `fmtflags`

An enumeration used to set various formatting flags for reading and writing of streams.

Table 16.1 Format Flags Enumerations

Flag	Effects when set
<code>boolalpha</code>	insert and extract bool type in alphabetic form
<code>dec</code>	decimal output

Flag	Effects when set
<code>fixed</code>	when set show floating point numbers in normal manner by default that is six decimal places
<code>hex</code>	hexadecimal output
<code>oct</code>	octal output
<code>left</code>	left justified
<code>right</code>	right justified
<code>internal</code>	pad a field between signs or base characters
<code>scientific</code>	show scientific notation for floating point numbers
<code>showbase</code>	show the bases numeric values
<code>showpoint</code>	show the decimal point and trailing zeros
<code>showpos</code>	show the leading plus sign for positive numbers
<code>skipws</code>	skip leading white spaces with input
<code>unitbuf</code>	buffer the output and flush after insertion operation
<code>uppercase</code>	show the scientific notation, x or o in uppercase

Table 16.2 Format flag field constants

Constants	Allowable values
<code>adjustfield</code>	<code>left</code> <code>right</code> <code>internal</code>
<code>basefield</code>	<code>dec</code> <code>oct</code> <code>hex</code>
<code>floatfield</code>	<code>scientific</code> <code>fixed</code>

27.4 Iostreams Base Classes

27.4.3 Class ios_base

Listing 16.1 Example of ios format flags usage

see `basic_ios::setf()` and `basic_ios::unsetf()`

27.4.3.1.3 Type iostate

An enumeration that is used to define the various states of a stream.

Table 16.3 Enumeration iostate

Flags	Usage
<code>badbit</code>	<code>iostate</code> improper read/write
<code>failbit</code>	<code>iostate</code> failure
<code>eofbit</code>	end of file bit set note: see variance from AT&T Standards

Listing 16.2 Example of ios iostate flags usage:

See `basic_ios::setstate()` and `basic_ios::rdstate()`

27.4.3.1.4 Type openmode

An enumeration that is used to specify various file opening modes.

Table 16.4 Enumeration openmode

Mode	Definition
<code>app</code>	Start the read or write at end of the file
<code>ate</code>	Start the read or write immediately at the end
<code>binary</code>	binary file
<code>in</code>	Start the read at end of the file
<code>out</code>	Start the write at the beginning of the file
<code>trunc</code>	Start the read or write at the beginning of the file

27.4.3.1.5 Type `seekdir`

An enumeration to position a pointer to a specific place in a file stream.

Table 16.5 Enumeration `seekdir`

Enumeration	Position
<code>beg</code>	Begging of stream
<code>cur</code>	Current position of stream
<code>end</code>	End of stream

Listing 16.3 Example of `ios seekdir` usage:

See: `streambuf::pubseekoff`

27.4.3.1.6 Class `Init`

Description An object that associates `<iostream>` object buffers with standard stream declared in `<cstdio>`.

Prototype

```
namespace std {
class ios_base::Init {
    public:
        Init();
        ~Init();
    private:
        // static int
    };
}
```

Class `Init` Constructor

Default Constructor

Description To construct an object of class `Init`;

27.4 Iostreams Base Classes

27.4.3 Class *ios_base*

Prototype `Init();`

Remarks The constructor `Init()` constructs an object of class `Init`. If `init_cnt` is zero the function stores the value one and constructs `cin`, `cout`, `cerr`, `clog`, `win`, `wout`, `werr` and `wlog`. In any case the constructor then adds one to `init_cnt`.

Destructor

Prototype `~Init();`

Remarks The destructor subtracts one from `init_cnt` and if the result is one calls `cout.flush()`, `cerr.flush()` and `clog.flush()`.

27.4.3.2 *ios_base* `fmtflags` state functions

Description To set the state of the *ios_base* format flags.

flags

Description To alter formatting flags using a mask.

Prototype `fmtflags flags() const`
 `fmtflags flags(fmtflags)`

Remarks Use `flags()` when you would like to use a mask of several flags, or would like to save the current format configuration. The return value of `flags()` returns the current `fmtflags`. The overloaded `flags(fmtflags)` alters the format flags but will return the value prior to the flags being changed.

Returns The `fmtflags` type before alterations.

NOTE: See `ios` enumerators for a list of `fmtflags`.

See Also: `setiosflags()` and `resetiosflags()`

Listing 16.4 Example of `flags()` usage:

```
#include <iostream>

// showf() displays flag settings
void showf();

int main()
{
using namespace std;
    showf(); // show format flags

    cout << "press enter to continue" << endl;
    cin.get();

    cout.setf(ios::right|ios::showpoint|ios::fixed);
    showf();
    return 0;
}

// showf() displays flag settings
void showf()
{
using namespace std;

    char fflags[][12] = {
        "boolalpha",
        "dec",
        "fixed",
        "hex",
        "internal",
        "left",
        "oct",
        "right",
        "scientific",
        "showbase",
        "showpoint",
        "showpos",
        "skipws",
        "unitbuf",
```

27.4 iostreams Base Classes

27.4.3 Class *ios_base*

```
        "uppercase"
};

long f = cout.flags();    // get flag settings
cout.width(9); // for demonstration
    // check each flag
for(long i=1, j =0; i<=0x4000; i = i<<1, j++)
{
    cout.width(10); // for demonstration
    if(i & f)
        cout << fflags[j] << " is on \n";
    else
        cout << fflags[j] << " is off \n";
}

cout << "\n";
}
```

Result:

```
boolalpha  is off
dec        is on
fixed      is off
hex        is off
internal   is off
left       is off
oct        is off
right      is off
scientific is off
showbase   is off
showpoint  is off
showpos    is off
skipws     is on
unitbuf    is off
uppercase  is off
```

press enter to continue

```
boolalpha is off
    dec is on
    fixed is on
```

```

        hex is off
    internal is off
        left is off
        oct is off
        right is on
scientific is off
    showbase is off
    showpoint is on
    showpos is off
    skipws is on
    unitbuf is off
uppercase is off

```

setf

Description Set the stream format flags.

Prototype `fmtflags setf(fmtflags)`
`fmtflags setf(fmtflags, fmtflags)`

Remarks You should use the function `setf()` to set the formatting flags for input/output. It is overloaded. The single argument form of `setf()` sets the flags in the mask. The two argument form of `setf()` clears the flags in the first argument before setting the flags with the second argument.

Returns type `basic_ios::fmtflags`

Listing 16.5 Example of `setf()` usage:

```

#include <iostream>

int main()
{
    using namespace std;

    double d = 10.01;

    cout.setf(ios::showpos | ios::showpoint);

```

27.4 Iostreams Base Classes

27.4.3 Class *ios_base*

```
cout << d << endl;
cout.setf(ios::showpoint, ios::showpos | ios::showpoint);
cout << d << endl;

return 0;
}
```

Result:
+10.01
10.01

unsetf

Description	To un-set previously set formatting flags.
Prototype	<code>void unsetf(fmtflags)</code>
Remarks	Use the <code>unsetf()</code> function to reset any format flags to a previous condition. You would normally store the return value of <code>setf()</code> in order to achieve this task.
Returns	There is no return.

Listing 16.6 Example of `unsetf()` usage:

```
#include <iostream>

int main()
{
using namespace std;

double d = 10.01;

cout.setf(ios::showpos | ios::showpoint);
cout << d << endl;

cout.unsetf(ios::showpoint);
cout << d << endl;
return 0;
}
```

```
}
```

```
Result:
```

```
+10.01
```

```
+10.01
```

precision

Description Set and return the current format precision.

Prototype `streamsize precision() const`
`streamsize precision(streamsize prec)`

Remarks Use the `precision()` function with floating point numbers to limit the number of digits in the output. You may use `precision()` with scientific or non-scientific floating point numbers. You may use the overloaded `precision()` to retrieve the current precision that is set.

With the flag `ios::floatfield` set the number in `precision` refers to the total number of significant digits generated. If the settings are for either `ios::scientific` or `ios::fixed` then the precision refers to the number of digits after the decimal place.

NOTE: This means that `ios::scientific` will have one more significant digit than `ios::floatfield`, and `ios::fixed` will have a varying number of digits.

Returns The current value set.

See Also `setprecision()`

Listing 16.7 Example of `precision()` usage:

```
#include <iostream>
```

```
extern double pi;
```

27.4 Iostreams Base Classes

27.4.3 Class *ios_base*

```
int main()
{
    using namespace std;

    double TenPi = 10*pi;

    cout.precision(5);
    cout.setf(0, ios::floatfield);
    cout << "floatfield:\t" << TenPi << endl;
    cout.setf(ios::scientific, ios::floatfield);
    cout << "scientific:\t" << TenPi << endl;
    cout.setf(ios::fixed, ios::floatfield);
    cout << "fixed:\t\t" << TenPi << endl;
    return 0;
}
```

Result:

floatfield: 31.416

scientific: 3.14159e+01

fixed: 31.41593

width

Description To set the width of the output field.

Prototype `streamsize width() const`
`streamsize width(streamsize wide)`

Remarks Use the `width()` function to set the field size for output. The function is overloaded to return just the current width setting if there is no parameter or to store and then return the previous setting before changing the fields width to the new parameter.

Returns The previous width setting is returned.

Listing 16.8 Example of `width()` usage:

```
#include <iostream>

int main()
{
    using namespace std;

    int width;

    cout.width(8);
    width = cout.width();
    cout.fill('*');
    cout << "Hi!" << '\n';

    // reset to left justified blank filler
    cout<< "Hi!" << '\n';

    cout.width(width);
    cout<< "Hi!" << endl;

    return 0;
}
```

Result:
 Hi!*****
 Hi!
 Hi!*****

27.4.3.3 ios_base locale functions

Description Sets the locale for input output operations.

imbue

Description Stores a value representing the locale.

Prototype `locale imbue(const locale loc);`

27.4 Iostreams Base Classes

27.4.3 Class *ios_base*

Remarks The precondition of the argument `loc` is equal to `getloc()`.

Return The previous value of `getloc()`.

getloc

Description Determined the imbued locale for input output operations.

Prototype `locale getloc() const;`

Return The global C++ locale if no locale has been imbued. Otherwise it returns the locale of the input and output operations.

27.4.3.4 *ios_base* storage function

Description To allocate storage pointers.

xalloc

Description Allocation function.

Prototype `static int xalloc()`

Return `index++`.

yword

Description Allocate an array of `int` and store a pointer.

Remark If `iarray` is a null pointer allocate an array and store a pointer to the first element. The function extends the array as necessary to include `iarray[idx]`. Each new allocated element is initialized to the return value may be invalid.

NOTE: After a subsequent call to `yword()` for the same object the return value may be invalid.

Return `irray[idx]`

pword

Description Allocate an array of pointers.

Prototype `void * &pword(int idx)`

Remarks If parray is a null pointer allocates an array of void pointers. Then extends parray as necessary to include the element parray[idx].

NOTE: After a subsequent call to pword() for the same object the return value may be invalid.

Return `parray[idx].`

register_callback

Description Registers functions when an event occurs.

Prototype `void register_callback
 (event_callback fn,
 int index);`

Remarks Registers the pair (fn, index) such that during calls to imbue(), copyfmt() or ~ios_base() the function fn is called with argument index. Function registered are called when an event occurs, in opposite order of registration. Functions registered while a callback function is active are not called until the next event.

NOTE: Identical pairs are not merged and a function registered twice will be called twice.

27.4 Iostreams Base Classes

27.4.4 Template class *basic_ios*

sync_with_stdio

Description	Synchronizes stream input output with 'C' input and output functions.
Prototype	<code>static bool sync_with_stdio(bool sync = true);</code>
Remarks	Is not supported in the Metrowerks Standard Library.
Returns	Always returns <code>true</code> indicating that the MSLstreams are always synchronized with the C streams.

27.4.3.5 ios_base Constructor

Default Constructor

Description	Construct and destruct an object of class <code>ios_base</code>
Prototype	<pre>protected: ios_base();</pre>
Remarks	The <code>ios_base</code> constructor is protected so it may only be derived from. If the values of the <code>ios_base</code> members are undermined.

Destructor

Prototype	<code>~ios_base();</code>
Remarks	Calls registered callbacks and destroys an object of class <code>ios_base</code> .

27.4.4 Template class *basic_ios*

Description	A template class for input and output streams. The prototype is listed below. Additional topics in this section are: <ul style="list-style-type: none">• “27.4.4.1 basic_ios Constructor” on page 470• “27.4.4.2 Member Functions” on page 471
--------------------	--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

- [“27.4.4.3 basic_ios iostate flags functions” on page 476](#)

Prototype

```

namespace std{
template<class charT,
        class traits = ios_traits<charT> >
class basic_ios : public ios_base {
public:
    typedef charT char_type;
    typedef typename traits::int_type int_type;
    typedef typename traits::pos_type pos_type;
    typedef typename traits::off_type off_type;

    operator bool() const;
    bool operator!() const;
    iostate rdbuf() const;
    void clear(iostate state = goodbit);
    void setstate(iostate state);
    bool good() const;
    bool eof() const;
    bool fail() const;
    bool bad() const;

    explicit basic_ios
        (basic_streambuf<charT, traits>,
         traits *sb);

    virtual ~basic_ios();

    basic_ostream<charT, traits>* tie() const;
    basic_ostream<charT, traits>*
        tie(basic_streambuf<charT, traits>* sb);

    basic_streambuf<charT, traits>* rdbuf() const;
    basic_streambuf<charT, traits>*
        rdbuf(basic_streambuf<charT, traits>* sb);

    basic_ios& copyfmt(const basic_ios& rhs);

    char_type fill()const;
    char_type fill(char_type ch);

```

27.4 Iostreams Base Classes

27.4.4 Template class *basic_ios*

```
        locale imbue(const locale& loc);

protected:
    basic_ios();
    void init(basic_streambuf<charT, traits>* sb);
    };
}
```

Remarks The `basic_ios` template class is a base class and includes many enumerations and mechanisms necessary for input and output operations.

27.4.4.1 `basic_ios` Constructor

Default and Overloaded Constructor

Description Construct an object of class `basic_ios` and assign values.

Prototype

```
public:
    explicit basic_ios
        (basic_streambuf<charT,traits>* sb);
protected:
    basic_ios();
```

Remarks The `basic_ios` constructor creates an object of class `basic_ios` and assigns values to its member functions by calling `init()`.

Destructor

Prototype `virtual ~basic_ios();`

Remarks Destroys an object of type `basic_ios`.

Remarks The conditions of the member functions after `init()` are shown in the following table.

Table 16.6 **Conditions after init()**

Member	Postcondition Value
rdbuf()	sb
tie()	zero
rdstate()	goodbit if stream buffer is not a null pointer otherwise badbit.
exceptions()	goodbit
flags()	skipws dec
width()	zero
precision()	six
fill()	the space character
getloc()	locale::classic()
index	undefined
iarray	a null pointer
parray	a null pointer

27.4.4.2 Member Functions

tie

Description To tie an ostream to the calling stream.

Prototype `basic_ostream<charT, traits>* tie() const;`
`basic_ostream<charT, traits>* tie`
`(basic_ostream<charT, traits>* tiestr);`

Remarks Any stream can have an ostream tied to it to ensure that the ostream is flushed before any operation. The standard input and output objects cin and cout are tied to ensure that cout is flushed before any cin operation. The function tie() is overloaded the parameterless version returns the current ostream that is tied if any.

27.4 Iostreams Base Classes

27.4.4 Template class *basic_ios*

The `tie()` function with an argument ties the new object to the ostream and returns a pointer if any from the first. The postcondition of `tie()` function that takes the argument `tiestr` is that `tiestr` is equal to `tie()`;

Returns A pointer to type ostream that is or previously was tied, or zero if there was none.

Listing 16.9 Example of `tie()` usage:

The file MW Reference contains
Metrowerks CodeWarrior "Software at Work"

```
#include <iostream>
#include <fstream>
#include <cstdlib>

char inFile[] = "MW Reference";

int main()
{
    using namespace std;

    ifstream inOut(inFile, ios::in | ios::out);
    if(!inOut.is_open())
        { cout << "file is not open"; exit(1); }
    ostream Out(inOut.rdbuf());

    if(inOut.tie())
        cout << "The streams are tied\n";
    else cout << "The streams are not tied\n";

    inOut.tie(&Out);
    inOut.rdbuf()->pubseekoff(0, ios::end);

    char str[] = "\nRegistered Trademark";
    Out << str;

    if(inOut.tie())
        cout << "The streams are tied\n";
```

```

else cout << "The streams are not tied\n";

inOut.close();
return 0;
}

```

Result:

```

The streams are not tied
The streams are tied

```

The file MW Reference now contains
Metrowerks CodeWarrior "Software at Work"
Registered Trademark

rdbuf

Description To retrieve a pointer to the stream buffer.

Prototype `basic_streambuf<charT, traits>* rdbuf() const;`
`basic_streambuf<charT, traits>* rdbuf`
`(basic_streambuf<charT, traits>* sb);`

Remarks To manipulate a stream for random access or synchronization it is necessary to retrieve a pointer to the streams buffer. The function `rdbuf()` allows you to retrieve this pointer. The `rdbuf()` function that takes an argument has the postcondition of `sb` is equal to `rdbuf()`.

Returns A pointer to `basic_streambuf` object.

Listing 16.10 Example of `rdbuf()` usage:

```

#include <iostream>

struct address {
    int number;
    char street[40];
} addbook;

```

27.4 istreams Base Classes

27.4.4 Template class *basic_ios*

```
int main()
{
    using namespace std;

    cout << "Enter your street number: ";
    cin >> addbook.number;

    cin.rdbuf()->pubsync(); // buffer flush

    cout << "Enter your street name: ";
    cin.get(addbook.street, 40);

    cout << "Your address is: "
         << addbook.number << " " << addbook.street;

    return 0;
}
```

Result:

```
Enter your street number: 2201
Enter your street name: Donley Drive
Your address is: 2201 Donley Drive
```

imbue

Description	Stores a value representing the locale.
Prototype	<code>locale imbue(const locale& rhs);</code>
Remarks	The function <code>imbue()</code> calls <code>ios_base::imbue()</code> and <code>rdbuf()->pubimbue()</code> .
Returns	The current locale.

fill

Description	To insert characters into the stream's unused spaces.
--------------------	-------------------------------------------------------

Prototype	<code>char_type fill() const</code> <code>char_type fill(char_type)</code>
Remarks	Use <code>fill(char_type)</code> in output to fill blank spaces with a character. The function <code>fill()</code> is overloaded to return the current filler without altering it.
Returns	The current character being used as a filler.
See Also	manipulator <code>setfill()</code>

Listing 16.11 Example of fill() usage:

```
#include <iostream>

int main()
{
    using namespace std;

    char fill;

    cout.width(8);
    cout.fill('*');
    fill = cout.fill();
    cout<< "Hi!" << "\n";
    cout << "The filler is a " << fill << endl;

    return 0;
}
```

```
Result:
Hi!*****
The filler is a *
```

copyfmt

Description Copies a `basic_ios` object.

Prototype `basic_ios& copyfmt(const basic_ios& rhs);`

27.4 Iostreams Base Classes

27.4.4 Template class *basic_ios*

Remarks Assigns members of `*this` object the corresponding objects of the `rhs` argument with certain exceptions. The exceptions are `rdstate()` is unchanged, `exceptions()` is altered last, and the contents of `pword` and `word` arrays are copied not the pointers themselves.

Returns The `this` pointer.

27.4.4.3 *basic_ios* `iosstate` flags functions

Description To set flags pertaining to the state of the input and output streams.

`operator bool`

Description A `bool` operator.

Prototype `operator bool() const;`

Returns `!fail()`

`operator !`

Description A `bool` not operator.

Prototype `bool operator ! ();`

Return `fail()`.

`rdstate`

Description To retrieve the state of the current formatting flags.

Prototype `iosstate rdstate() const`

Remarks This member function allows you to read and check the current status of the input and output formatting flags. The returned value may be stored for use in the function `ios::setstate()` to reset the flags at a later date.

Returns Type `iosstate` used in `ios::setstate()`

See Also `ios::setstate()`

Listing 16.12 Example of `rdstate()` usage:

The file `MW Reference` contains:
`ABCDEFGHIJKLMNOPQRSTUVWXYZ`

```
#include <iostream>
#include <fstream>
#include <cstdlib>

char * inFile = "MW Reference";

using namespace std;

void status(ifstream &in);

int main()
{
    ifstream in(inFile);
    if(!in.is_open())
    {
        cout << "could not open file for input";
        exit(1);
    }

    int count = 0;
    int c;
    while((c = in.get()) != EOF)
    {
        // simulate a bad bit
        if(count++ == 12) in.setstate(ios::badbit);
        status(in);
    }

    status(in);
    in.close();
}
```

27.4 istreams Base Classes

27.4.4 Template class *basic_ios*

```
    return 0;
}

void status(istream &in)
{
    int i = in.rdstate();
    switch (i) {
        case ios::eofbit : cout << "EOF encountered \n";
                           break;
        case ios::failbit : cout << "Non-Fatal I/O Error n";
                           break;
        case ios::goodbit : cout << "GoodBit set \n";
                           break;
        case ios::badbit : cout << "Fatal I/O Error \n";
                           break;
    }
}
```

Result:

```
GoodBit set
GoodBit set
GoodBit set
GoodBit set
GoodBit set
GoodBit set
GoodBit set
GoodBit set
GoodBit set
GoodBit set
GoodBit set
GoodBit set
GoodBit set
Fatal I/O Error
```

clear

Description Clears iostate field.

Prototype `void clear`
 `(iostate state = goodbit) throw failure;`

Remarks Use `clear()` to reset the failbit, eofbit or a badbit that may have been set inadvertently when you wish to override for continuation of your processing. Postcondition of `clear` is the argument is equal to `rdstate()`.

NOTE: If `rdstate()` and `exceptions()` $\neq 0$ an exception is thrown.

Returns No value is returned.

Listing 16.13 Example of `clear()` usage:

The file MW Reference contains:
ABCDEFGH

```
#include <iostream>
#include <fstream>
#include <cstdlib>

char * inFile = "MW Reference";

using namespace std;

void status(ifstream &in);

int main()
{
    ifstream in(inFile);
    if(!in.is_open())
    {
        cout << "could not open file for input";
        exit(1);
    }

    int count = 0;
    int c;
    while((c = in.get()) != EOF) {
        if(count++ == 4)
        {
```

27.4 Iostreams Base Classes

27.4.4 Template class *basic_ios*

```
        // simulate a failed state
        in.setstate(ios::failbit);
        in.clear();
    }
    status(in);
}

status(in);
in.close();
return 0;
}

void status(ifstream &in)
{
    // note: eof() is not needed in this example
    // if(in.eof()) cout << "EOF encountered \n"
    if(in.fail()) cout << "Non-Fatal I/O Error \n";
    if(in.good()) cout << "GoodBit set \n";
    if(in.bad()) cout << "Fatal I/O Error \n";
}
```

Result:

```
GoodBit set
GoodBit set
GoodBit set
GoodBit set
GoodBit set
GoodBit set
GoodBit set
GoodBit set
Non-Fatal I/O Error
```

setstate

Description To set the state of the format flags.

Prototype void setstate(iostate state) throw(failure);

Remarks Calls `clear(rdstate() | state)` and may throw an exception.

Returns No Return

Listing 16.14 Example of `setstate()` usage:

See `ios::rdstate()`

good

Description To test for the lack of error bits being set.

Prototype `bool good() const;`

Remarks Use the function `good()` to test for the lack of error bits being set.

Returns

True if `rdstate() == 0`.

Listing 16.15 Example of `good()` usage:

See `basic_ios::bad()`

eof

Description To test for the eofbit setting.

Prototype `bool eof() const`

Remarks Use the `eof()` function to test for an eofbit setting in a stream being processed under some conditions. This end of file bit is not set by stream opening or closing, but only for operations that detect an end of file condition.

Returns True if eofbit is set in `rdstate()`.

27.4 istreams Base Classes

27.4.4 Template class *basic_ios*

Listing 16.16 Example of eof() usage:

MW Reference is simply a one line text document
ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz

```
#include <iostream>
#include <fstream>
#include <cstdlib>

const char* TheText = "MW Reference";

int main()
{
    using namespace std;

    ifstream in(TheText);
    if(!in.is_open())
    {
        cout << "Couldn't open file for input";
        exit(1);
    }

    int i = 0;
    char c;
    cout.setf(ios::uppercase);

    //eofbit is not set under normal file opening
    while(!in.eof())
    {
        c = in.get();
        cout << c << " " << hex << int(c) << "\n";

        // simulate an end of file state
        if(++i == 5) in.setstate(ios::eofbit);
    }
    return 0;
}
```

Result:

A 41

B 42

C 43

D 44

E 45

fail

Description To test for stream reading failure from any cause.

Prototype `bool fail() const`

Remarks The member function `fail()` will test for `failbit` and `badbit`.

Returns True if `failbit` or `badbit` is set in `rdstate()`.

Listing 16.17 Example of `fail()` usage:

MW Reference file for input contains.

float 33.33 double 3.16e+10 Integer 789 character C

```
#include <iostream>
```

```
#include <fstream>
```

```
#include <cstdlib>
```

```
int main()
```

```
{
```

```
using namespace std;
```

```
    char inFile[] = "MW Reference";
```

```
    ifstream in(inFile);
```

```
    if(!in.is_open())
```

```
    {cout << "Cannot open input file"; exit(1);}
```

```
    char ch = 0;
```

```
    while(!in.fail())
```

```
    {
```

```
        if(ch)cout.put(ch);
```

```
        in.get(ch);
```

```
    }
```

27.4 istreams Base Classes

27.4.4 Template class *basic_ios*

```
    return 0;
}
```

Result:

float 33.33 double 3.16e+10 integer 789 character C

bad

Description To test for fatal I/O error.

Prototype `bool bad() const`

Remarks Use the member function `bad()` to test if a fatal input or output error occurred which sets the `badbit` flag in the stream.

Returns True if `badbit` is set in `rdstate()`.

See Also `basic_ios::fail()`

Listing 16.18 Example of `bad()` usage:

The File MW Reference contains:

abcdefghijklmnopqrstuvwxyz

```
#include <iostream>
#include <fstream>
#include <cstdlib>
```

```
char * inFile = "MW Reference";
```

```
using namespace std;
```

```
void status(istream &in);
```

```
int main()
{
    ifstream in(inFile);
    if(!in.is_open())
```

```
{
    cout << "could not open file for input";
    exit(1);
}

int count = 0;
int c;
while((c = in.get()) != EOF)
{
    // simulate a failed state
    if(count++ == 4) in.setstate(ios::failbit);
    status(in);
}

status(in);
in.close();
return 0;
}

void status(ifstream &in)
{
    // note: eof() is not needed in this example
    // if(in.eof()) cout << "EOF encountered \n";

    if(in.fail()) cout << "Non-Fatal I/O Error \n";
    if(in.good()) cout << "GoodBit set \n";
    if(in.bad()) cout << "Fatal I/O Error \n";
}
```

Result:

```
GoodBit set
GoodBit set
GoodBit set
GoodBit set
Non-Fatal I/O Error
Non-Fatal I/O Error
```

27.4 Iostreams Base Classes

27.4.5 ios_base manipulators

exceptions

Description	To handle basic_ios exceptions.
Prototype	<pre>iostate exceptions() const; void exceptions(iostate except);</pre>
Remarks	The function <code>exceptions()</code> determines what elements in <code>rdstate()</code> cause exceptions to be thrown. The overloaded <code>exceptions(iostate)</code> calls <code>clear(rdstate())</code> and leaves the argument <code>except</code> equal to <code>exceptions()</code> .
Return	A mask that determines what elements set in <code>rdstate()</code> cause undefined behavior.

27.4.5 ios_base manipulators

Description	To provide an in line input and output formatting mechanism.
--------------------	--------------------------------------------------------------

The topics in this section are:

- [“27.4.5.1 fmtflags manipulators” on page 486](#)
- [“27.4.5.2 adjustfield manipulators” on page 488](#)
- [“27.4.5.3 basefield manipulators” on page 488](#)
- [“27.4.5.4 floatfield manipulators” on page 489](#)

27.4.5.1 fmtflags manipulators

Description	To provide an in line input and output numerical formatting mechanism.
--------------------	------------------------------------------------------------------------

Table 16.7 **Prototype of ios_base manipulators**

Manipulator	Definition
<code>ios_base& boolalpha(ios_base&)</code>	insert and extract bool type in alphabetic format
<code>ios_base& noboolalpha(ios_base&)</code>	unsets insert and extract bool type in alphabetic format
<code>ios_base& showbase(ios_base& b)</code>	set the number base to parameter b
<code>ios_base& noshowbase(ios_base&)</code>	remove show base
<code>ios_base& show-point(ios_base&)</code>	show decimal point
<code>ios_base& noshow-point(ios_base&)</code>	do not show decimal point
<code>ios_base& showpos(ios_base&)</code>	show the positive sign
<code>ios_base& noshowpos(ios_base&)</code>	do not show positive sign
<code>ios_base& skipws(ios_base&)</code>	input only skip white spaces
<code>ios_base& noskipws(ios_base&)</code>	input only no skip white spaces
<code>ios_base& uppercase(ios_base&)</code>	show scientific in uppercase
<code>ios_base& nouppercase(ios_base&)</code>	do not show scientific in uppercase

27.4 Iostreams Base Classes

27.4.5 ios_base manipulators

	Manipulator	Definition
	<code>ios_base& unitbuf</code> <code>(ios_base::unitbuf)</code>	set the unitbuf flag
	<code>ios_base& nunitbuf</code> <code>(ios_base::unitbuf)</code>	unset the unitbuf flag
Remarks	Manipulators are used in the stream to alter the formatting of the stream.	
Returns	A reference to an object of type <code>ios_base</code> is returned to the stream. (The <code>this</code> pointer.)	

27.4.5.2 adjustfield manipulators

Description To provide an in line input and output orientation formatting mechanism.

Table 16.8 Adjustfield manipulators

	Manipulator	Definition
	<code>ios_base& internal(ios_base&)</code>	fill between indicator and value
	<code>ios_base& left(ios_base&)</code>	left justify in a field
	<code>ios_base& right(ios_base&)</code>	right justify in a field
Remarks	Manipulators are used in the stream to alter the formatting of the stream.	
Returns	A reference to an object of type <code>ios_base</code> is returned to the stream. (The <code>this</code> pointer.)	

27.4.5.3 basefield manipulators

Description To provide an in line input and output numerical formatting mechanism.

Table 16.9 Basefield manipulators

Manipulator	Definition
<code>ios_base& dec(ios_base&)</code>	format output data as a decimal
<code>ios_base& oct(ios_base&)</code>	format output data as octal
<code>ios_base& hex(ios_base&)</code>	format output data as hexadecimal
Remarks	Manipulators are used in the stream to alter the formatting of the stream.
Returns	A reference to an object of type <code>ios_base</code> is returned to the stream. (The <code>this</code> pointer.)

27.4.5.4 floatfield manipulators

Description To provide an in line input and output numerical formatting mechanism.

Table 16.10 Floatfield manipulators

Manipulator	Definition
<code>ios_base& fixed(ios_base&)</code>	format in fixed point notation
<code>ios_base& scientific(ios_base&)</code>	use scientific notation
Remarks	Manipulators are used in the stream to alter the formatting of the stream.
Returns	A reference to an object of type <code>ios_base</code> is returned to the stream. (The <code>this</code> pointer.)

Listing 16.19 Example of manipulator usage:

```
#include <iostream>
#include <iomanip>

int main()
```

27.4 istreams Base Classes

27.4.5 ios_base manipulators

```
{
using namespace std;

    long number = 64;

    cout << "Original Number is "
         << number << "\n\n";
    cout << showbase;
    cout << setw(30) << "Hexadecimal :"
         << hex << setw(10) << right
         << number << '\n';
    cout << setw(30) << "Octal :" << oct
         << setw(10) << left
         << number << '\n';
    cout << setw(30) << "Decimal :" << dec
         << setw(10) << right
         << number << endl;

    return 0;
}
```

Result:

Original Number is 64

Hexadecimal : 0x40

Octal :0100

Decimal : 64

Overloading Manipulators

Description To provide an in line formatting mechanism.

Prototype The basic template for parameterless manipulators,
ostream &manip-name(ostream &stream)
{
 // coding
 return stream;
}

27.4 lostreams Base Classes

27.4.5 ios_base manipulators

Remarks Use overloaded manipulators to provide specific and unique formatting methods relative to one class.

Returns A reference to ostream. (Usually the this pointer.)

See Also `<iomanip>` for manipulators with parameters

Listing 16.20 Example of overloaded manipulator usage:

```
#include <iostream>

using namespace std;

ostream &rJus(ostream &stream);

int main()
{
    cout << "align right " << rJus << "for column";
    return 0;
}

ostream &rJus(ostream &stream)
{
    stream.width(30);
    stream.setf(ios::right);
    return stream;
}
```

```
Result:
align right          for column
```

27.4 **iostreams Base Classes**

27.4.5 *ios_base manipulators*



27.5 Stream Buffers

The header `<streambuf>` defines types that control input and output to character sequences.

Overview of Stream Buffers

The sections in this chapter are:

- [“Header `<streambuf>`” on page 493](#)
- [“27.5.1 Stream buffer requirements” on page 493](#)
- [“27.5.2 Template class `basic_streambuf<charT, traits>`” on page 494](#)

Header `<streambuf>`

```
Prototype  namespace std {
            template <class charT, class traits =
            char_traits<charT> >
              class basic_streambuf;
            typedef basic_streambuf<char> streambuf;
            typedef basic_streambuf<wchar_t> wstreambuf;
            }
```

27.5.1 Stream buffer requirements

Stream buffers can impose constraints. The constraints include:

- The input sequence can be not readable
- The output sequence can be not writable
- The sequences can be association with other presentations such as external files

27.5 Stream Buffers

27.5.2 Template class `basic_streambuf<charT, traits>`

- The sequences can support operations to or from associated sequences.
- The sequences can impose limitations on how the program can read and write characters to and from a sequence or alter the stream position.

There are three pointers that control the operations performed on a sequence or associated sequences. These are used for read, writes and stream position alteration. If not `null` all pointers point to the same `charT` array object.

- The beginning pointer or lowest element in an array. - (beg)
- The next pointer of next element addressed for read or write. - (next)
- The end pointer of first element addressed beyond the end of the array. - (end)

27.5.2 Template class `basic_streambuf<charT, traits>`

The prototype is listed below. Additional topics in this section are:

- [“27.5.2.1 basic_streambuf Constructor” on page 497](#)
- [“27.5.2.2 basic_streambuf Public Member Functions” on page 498](#)
- [“27.5.2.2.1 Locales” on page 498](#)
- [“27.5.2.2.2 Buffer Management and Positioning” on page 498](#)
- [“27.5.2.2.3 Get Area” on page 504](#)
- [“27.5.2.2.4 Putback” on page 507](#)
- [“27.5.2.2.5 Put Area” on page 509](#)
- [“27.5.2.3 basic_streambuf Protected Member Functions” on page 511](#)
- [“27.5.2.3.1 Get Area Access” on page 511](#)
- [“27.5.2.3.2 Put Area Access” on page 512](#)
- [“27.5.2.4 basic_streambuf Virtual Functions” on page 514](#)
- [“27.5.2.4.1 Locales” on page 514](#)

- [“27.5.2.4.2 Buffer Management and Positioning” on page 514](#)
- [“27.5.2.4.3 Get Area” on page 516](#)
- [“27.5.2.4.4 Putback” on page 518](#)
- [“27.5.2.4.5 Put Area” on page 518](#)

Prototype

```
namespace std {
template< class charT, class traits = char_traits<charT> >
class basic_streambuf {
public:

    typedef charT char_type;
    typedef typename traits::int_type int_type;
    typedef typename traits::pos_type pos_type;
    typedef typename traits::off_type off_type;

    virtual ~basic_streambuf();

    locale pubimbue(const locale &loc);
    locale getloc() const;

    basic_streambuf<char_type, traits> * pubsetbuf
        (char_type* s, streamsize n);

    pos_type pubseekoff
        (off_type off,
         ios_base::seekdir way,
         ios_base::openmode which = ios_base::in | ios_base::out);
    pos_type pubseekpos
        (pos_type sp,
         ios_base::openmode which = ios::in | ios::out);

    int pubsync();

    streamsize in_avail();

    int_type snextc();
    int_type sbumpc();
    int_type sgetc();
```

27.5 Stream Buffers

27.5.2 Template class *basic_streambuf*<charT, traits>

```
streamsize sgetn(char_type *s, streamsize n);

int_type sputback(char_type C);
int_type sungetc();

int_type sputc(char_type c);
int_type sputn(char_type *s, streamsize n);

protected:
    basic_streambuf();

    char_type* eback() const;
    char_type* gptr() const;
    char_type* egptr() const;
    void gbump(int n);
    void setg
        (char_type *gbeg,
         char_type *gnext,
         char_type *gend);

    char_type* pbase() const;
    char_type* pptr() const;
    char_type* epptr() const;

    void pbump(int n);
    void setp(char_type *pbeg, char_type *pend);

    virtual void imbue(const locale &loc);

    virtual basic_streambuf<char_type, traits>* setbuf
        (char_type* s, streamsize n);

    virtual pos_type seekoff
        (off_type off,
         ios_base::seekdir way,
         ios_base::openmode which = ios::in | ios::out);
    virtual pos_type seekpos
        (pos_type sp,
         ios_base::openmode which = ios::in | ios::out);
    virtual int sync();
```



```

virtual int showmanyc();
virtual streamsize xsgetn(char_type *s,
    streamsize n);
virtual int_type underflow();
virtual int_type uflow();

virtual int_type
    pbackfail(int_type c = traits::eof());

virtual streamsize xspn
    (const char_type *s, streamsize n);
virtual int_type overflow
    (int_type c = traits::eof());
};
}

```

Remarks The template class `basic_streambuf` is an abstract class for deriving various stream buffers whose objects control input and output sequences. The type `streambuf` is an instantiation of `char` type. the type `wstreambuf` is an instantiation of `wchar_t` type.

27.5.2.1 `basic_streambuf` Constructor

Default Constructor

Description Construct and destruct an object of type `basic_streambuf`.

Prototype `protected:`
`basic_streambuf();`

Remarks The constructor sets all pointer member objects to null pointers and calls `getloc()` to copy the global locale at the time of construction.

Destructor

Prototype `virtual ~basic_streambuf();`

Remarks Removes the object from memory.

27.5 Stream Buffers

27.5.2 Template class *basic_streambuf*<charT, traits>

27.5.2.2 *basic_streambuf* Public Member Functions

Description The public member functions allow access to member functions from derived classes.

27.5.2.2.1 Locales

Locales are used for encapsulation and manipulation of information to a particular locale.

basic_streambuf::pubimbue

Description To set the locale.

Prototype `locale pubimbue(const locale &loc);`

Remarks The function *pubimbue* calls *imbue*(loc).

Return The previous value of `getloc()`.

basic_streambuf::getloc

Description To get the locale.

Prototype `locale getloc() const;`

Return If *pubimbue* has already been called one it returns the last value of *loc* supplied otherwise the current one. If *pubimbue* has been called but has not returned a value it from *imbue*, it then returns the previous value.

27.5.2.2.2 Buffer Management and Positioning

Functions used to manipulate the buffer and the input and output positioning pointers.

basic_streambuf::pubsetbuf

Description	To set an allocation after construction.
Prototype	<code>basic_streambuf<char_type, traits> *pubsetbuf (char_type* s, streamsize n);</code>
Remarks	The first argument is used in an another function by a <code>filebuf</code> derived class. See <code>setbuf()</code> . The second argument is used to set the size of a dynamic allocated buffer.
Return	A pointer to <code>basic_streambuf<char_type, traits></code> via <code>setbuf(s, n)</code> .

Listing 17.1 Example of basic_streambuf::pubsetbuf() usage:

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

const int size = 100;
char temp[size] = "\0";

int main()
{
    using namespace std;

    stringbuf strbuf;
    strbuf.pubsetbuf('\0', size);
    strbuf.sputn("Metrowerks CodeWarrior", 50);
    strbuf.sgetn(temp, 50);
    cout << temp;

    return 0;
}
```

Result:
Metrowerks CodeWarrior

27.5 Stream Buffers

27.5.2 Template class `basic_streambuf<charT, traits>`

`basic_streambuf::pubseekoff`

Description Determines the position of the get pointer.

Prototype `pos_type pubseekoff`
`(off_type off,`
`ios_base::seekdir way, ios_base::openmode`
`which = ios_base::in | ios_base::out);`

Remarks The member function `pubseekoff()` is used to find the difference in bytes of the get pointer from a known position (such as the beginning or end of a stream). The function `pubseekoff()` returns a type `pos_type` which holds all the necessary information.

Return A `pos_type` via `seekoff(off, way, which)`

See Also `pubseekpos()`

Listing 17.2 Example of `basic_streambuf::pubseekoff()` usage:

The MW Reference file contains originally
Metrowerks CodeWarrior "Software at Work"

```
#include <iostream>
#include <fstream>
#include <stdlib.h>

char inFile[] = "MW Reference";

int main()
{
    using namespace std;

    ifstream inOut(inFile, ios::in | ios::out);
    if(!inOut.is_open())
        {cout << "Could not open file"; exit(1);}
    ostream Out(inOut.rdbuf());

    char str[] = "\nRegistered Trademark";
```

```

inOut.rdbuf()->pubseekoff(0, ios::end);

Out << str;

inOut.close();
return 0;
}

```

Result:

The File now reads:

Metrowerks CodeWarrior "Software at Work"

Registered Trademark

basic_streambuf::pubseekpos

Description Determine and move to a desired offset.

Prototype `pos_type pubseekpos
(pos_type sp,
ios_base::openmode which = ios::in | ios::out);`

Remarks The function `pubseekpos()` is use to move to a desired offset using a type `pos_type`, which holds all necessary information.

Return A `pos_type` via `seekpos(sb, which)`

See Also `pubseekoff()`, `seekoff()`

Listing 17.3 Example of streambuf::pubseekpos() usage:

The file MW Reference contains:

ABCDEFGHIJKLMNOPQRSTUVWXYZ

```

#include <iostream>
#include <fstream>
#include <cstdlib>

```

```

int main()

```

27.5 Stream Buffers

27.5.2 Template class `basic_streambuf<charT, traits>`

```
{
using namespace std;

    ifstream in("MW Reference");
    if(!in.is_open())
        {cout << "could not open file"; exit(1);}

    streampos spEnd(0), spStart(0), aCheck(0);
    spEnd = spStart = 5;

    aCheck = in.rdbuf()->pubseekpos(spStart,ios::in);
    cout << "The offset at the start of the reading"
        << " in bytes is "
        << static_cast<streamoff>(aCheck) << endl;

    char ch;
    while(spEnd != spStart+10)
    {
        in.get(ch);
        cout << ch;
        spEnd = in.rdbuf()->pubseekoff(0, ios::cur);
    }

    aCheck = in.rdbuf()->pubseekoff(0,ios::cur);
    cout << "\nThe final position's offset"
        << " in bytes now is "
        << static_cast<streamoff>(aCheck) << endl;

    in.close();

    return 0;
}
```

Result:

The offset for the start of the reading in bytes is 5

FGHIJKLMNOP

The final position's offset in bytes now is 15

basic_streambuf::pubsync

Description	To synchronize the <code>streambuf</code> object with its input/output.
Prototype	<code>int pubsync();</code>
Remarks	The function <code>pubsync()</code> will attempt to synchronize the <code>streambuf</code> input and output.
Returns	Zero if successful or EOF if not via <code>sync()</code> .

Listing 17.4 Example of `streambuf::pubsync()` usage:

```
#include <iostream>

struct address {
    int number;
    char street[40];
}addbook;

int main()
{
    using namespace std;

    cout << "Enter your street number: ";
    cin >> addbook.number;

    cin.rdbuf()->pubsync(); // buffer flush

    cout << "Enter your street name: ";
    cin.get(addbook.street, 40);

    cout << "Your address is: "
         << addbook.number << " " << addbook.street;

    return 0;
}
```

Result:

Enter your street number: 2201

27.5 Stream Buffers

27.5.2 Template class `basic_streambuf<charT, traits>`

Enter your street name: Donley Drive

Your address is: 2201 Donley Drive

27.5.2.2.3 Get Area

Public functions for retrieving input from a buffer.

`basic_streambuf::in_avail`

Description To test for availability of input stream.

Prototype `streamsize in_avail();`

Return If a read is permitted returns size of stream as a type `streamsize`.

`basic_streambuf::snextc`

Description To retrieve the next character in a stream.

Prototype `int_type snextc();`

Remarks The function `snextc()` calls `sbumpc()` to extract the next character in a stream. After the operation, the get pointer references the character following the last character extracted.

Return If `sbumpc` returns `traits::eof` returns that, otherwise returns `sgetc()`.

Listing 17.5 Example of `streambuf::snextc()` usage:

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

const int size = 100;

int main()
{
    using namespace std;
```



```

stringbuf strbuf;
strbuf.pubsetbuf('\0', size);
strbuf.sputn("ABCDE", 50);

char ch;
    // look ahead at the next character
ch =strbuf.snextc();
cout << ch;
    // get pointer was not returned after peeking
ch = strbuf.snextc();
cout << ch;

return 0;
}

```

Result:
BC

basic_streambuf::sbumpc

Description	To move the get pointer.
Prototype	<code>int_type sbumpc();</code>
Remarks	The function <code>sbumpc()</code> moves the get pointer one element when called.
Return	The value of the character at the <code>get</code> pointer. It returns <code>ufLOW()</code> if it fails to move the pointer.
See Also	<code>sgetc()</code>

Listing 17.6 Example of `streambuf::sbumpc()` usage:

```

#include <iostream>
#include <sstream>

const int size = 100;

```

27.5 Stream Buffers

27.5.2 Template class `basic_streambuf<charT, traits>`

```
std::string buf = "Metrowerks CodeWarrior --Software at Work--";

int main()
{
    using namespace std;

    stringbuf strbuf(buf);

    int ch;
    for (int i = 0; i < 23; i++)
    {
        ch = strbuf.sgetc();
        strbuf.sbumpc();
        cout.put(ch);
    }
    cout << endl;
    cout << strbuf.str() << endl;
    return 0;
}
```

Result:

Metrowerks CodeWarrior

Metrowerks CodeWarrior --Software at Work--

`basic_streambuf::sgetc`

Description To extract a character from the stream.

Prototype `int_type sgetc();`

Remarks The function `sgetc()` extracts a single character, without moving the get pointer.

Return A `int_type` type at the get pointer if available otherwise returns `underflow()`.

Listing 17.7 Example of `streambuf::sgetc()` usage:

See `streambuf::sbumpc()`

`basic_streambuf::sgetn`

Description	To extract a series of characters from the stream.
Prototype	<code>streamsize sgetn(char_type *s, streamsize n);</code>
Remarks	The public member function <code>sgetn()</code> is used to extract a series of characters from the stream buffer. After the operation, the <code>get</code> pointer references the character following the last character extracted.
Return	A <code>streamsize</code> type as returned from the function <code>xsgetn(s,n)</code> .

Listing 17.8 Example of `streambuf::sgetn()` usage:

See `pubsetbuf()`

27.5.2.2.4 Putback

Public functions to return a value to a stream.

`basic_streambuf::sputback`

Description	To put a character back into the stream.
Prototype	<code>int_type sputback(char_type c);</code>
Remarks	The function <code>sputbackc()</code> will replace a character extracted from the stream with another character. The results are not assured if the putback is not immediately done or a different character is used.
Return	If successful returns a pointer to the <code>get</code> pointer as an <code>int_type</code> otherwise returns <code>pbackfail(c)</code> .

27.5 Stream Buffers

27.5.2 Template class *basic_streambuf*<charT, traits>

Listing 17.9 Example of *streambuf::sputbackc()* usage:

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

std::string buffer = "ABCDEF";

int main()
{
    using namespace std;

    stringbuf strbuf(buffer);
    char ch;

    ch = strbuf.sgetc(); // extract first character
    cout << ch; // show it

    //get the next character
    ch = strbuf.snextc();

    // if second char is B replace first char with x
    if(ch == 'B') strbuf.sputbackc('x');

    // read the first character now x
    cout << (char)strbuf.sgetc();

    strbuf.sbumpc(); // increment get pointer
    // read second character
    cout << (char)strbuf.sgetc();

    strbuf.sbumpc(); // increment get pointer
    // read third character
    cout << (char)strbuf.sgetc();

    // show the new stream after alteration
    strbuf.pubseekoff(0, ios::beg);
    cout << endl;

    cout << (char)strbuf.sgetc();
```

```

while( (ch = strbuf.snextc()) != EOF)
    cout << ch;

    return 0;
}

```

Result:

AxBC

xBCDEF

basic_streambuf::sungetc

Description	To restore a character extracted.
Prototype	<code>int_type sungetc();</code>
Remarks	The function <code>sungetc()</code> restores the previously extracted character. After the operation, the <code>get</code> pointer references the last character extracted.
Return	If successful returns a pointer to the <code>get</code> pointer as an <code>int_type</code> otherwise returns <code>ebackfail(c)</code> .

Listing 17.10 Example of `streambuf::sungetc()` usage:

See: `streambuf::sputbackc()`

27.5.2.2.5 Put Area

Public functions for inputting characters into a buffer.

basic_streambuf::sputc

Description	To insert a character in the stream.
--------------------	--------------------------------------

27.5 Stream Buffers

27.5.2 Template class `basic_streambuf<charT, traits>`

Prototype `int_type sputc(char_type c);`

Remarks The function `sputc()` inserts a character into the stream. After the operation, the get pointer references the character following the last character extracted.

Return If successful returns `c` as an `int_type` otherwise returns `overflow(c)`.

Listing 17.11 Example of `streambuf::sputc()` usage:

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

int main()
{
    using namespace std;

    stringbuf strbuf;
    strbuf.sputc('A');

    char ch;
    ch = strbuf.sgetc();
    cout << ch;

    return 0;
}
```

Result:

A

`basic_streambuf::sputn`

Description To insert a series of characters into a stream.

Prototype `int_type sputn(char_type *s, streamsize n);`

Remarks The function `sputn()` inserts a series of characters into a stream. After the operation, the get pointer references the character following the last character extracted.

Return A `streamsize` type returned from a call to `xputn(s,n)`.

27.5.2.3 **basic_streambuf Protected Member Functions**

Protected member functions that are used for stream buffer manipulations by the `basic_streambuf` class and derived classes from it.

27.5.2.3.1 **Get Area Access**

Member functions for extracting information from a stream.

basic_streambuf::eback

Description Retrieve the beginning pointer for stream input.

Prototype `char_type* eback() const;`

Return The beginning pointer.

basic_streambuf::gptr

Description Retrieve the next pointer for stream input.

Prototype `char_type* gptr() const;`

Return The next pointer.

basic_streambuf::egptr

Description Retrieve the end pointer for stream input.

Prototype `char_type* egptr() const;`

27.5 Stream Buffers

27.5.2 Template class *basic_streambuf*<charT, traits>

Return The end pointer.

`basic_streambuf::gbump`

Description Advances the next pointer for stream input.

Prototype `void gbump(int n);`

Remarks The function `gbump()` advances the input pointer by the value of the `int n` argument.

`basic_streambuf::setg`

Description To set the beginning, next and end pointers.

Prototype `void setg
 (char_type *gbeg,
 char_type *gnext,
 char_type *gend);`

Remarks After the call to `setg()` the `gbeg` pointer equals `eback()`, the `gnext` pointer equals `gptr()`, and the `gend` pointer equals `egptr()`.

27.5.2.3.2 Put Area Access

Protected member functions for stream output sequences.

`basic_streambuf::pbase`

Description To retrieve the beginning pointer for stream output.

Prototype `char_type* pbase() const;`

Return The beginning pointer.

basic_streambuf::pptr

Description To retrieve the next pointer for stream output.

Prototype `char_type* pptr() const;`

Return The next pointer.

basic_streambuf::epptr

Description To retrieve the end pointer for stream output.

Prototype `char_type* epptr() const;`

Return The end pointer.

basic_streambuf::pbump

Description To advance the next pointer for stream output.

Prototype `void pbump(int n);`

Remarks The function `pbump()` advances the next pointer by the value of the `int` argument `n`.

basic_streambuf::setp

Description To set the values for the beginning, next and end pointers.

Prototype `void setp
 (char_type* pbeg,
 char_type* pend);`

Remarks After the call to `setp()`, `pbeg` equals `pbase()`, `pbeg` equals `pptr()` and `pend` equals `epptr()`.

27.5 Stream Buffers

27.5.2 Template class `basic_streambuf<charT, traits>`

27.5.2.4 `basic_streambuf` Virtual Functions

Description The virtual functions in `basic_streambuf` class are to be overloaded in any derived class.

27.5.2.4.1 Locales

Description To get and set the stream locale. These functions should be overridden in derived classes.

`basic_streambuf::imbue`

Description To change any translations base on locale.

Prototype `virtual void imbue(const locale &loc);`

Remarks The `imbue()` function allows the derived class to be informed in changes of locale and to cache results of calls to locale functions.

27.5.2.4.2 Buffer Management and Positioning

Virtual functions for positioning and manipulating the stream buffer. These functions should be overridden in derived classes.

`basic_streambuf::setbuf`

Description To set a buffer for stream input and output sequences.

Prototype `virtual basic_streambuf<char_type, traits> *
setbuf
(char_type* s, streamsize n);`

Remarks The function `setbuf()` is overridden in `basic_stringbuf` and `basic_filebuf` classes.

Return The `this` pointer.

basic_streambuf::seekoff

Description To return an offset of the current pointer in an input or output streams.

Prototype `virtual pos_type seekoff
(off_type off,
ios_base::seekdir way,
ios_base::openmode which = ios::in | ios::out);`

Remarks The function `seekoff()` is overridden in `basic_stringbuf` and `basic_filebuf` classes.

Return A `pos_type` value, which is an invalid stream position.

basic_streambuf::seekpos

Description To alter an input or output stream position.

Prototype `virtual pos_type seekpos
(pos_type sp,
ios_base::openmode which = ios::in | ios::out);`

Remarks The function `seekpos()` is overridden in `basic_stringbuf` and `basic_filebuf` classes.

Return A `pos_type` value, which is an invalid stream position.

basic_streambuf::sync

Description To synchronize the controlled sequences in arrays.

Prototype `virtual int sync();`

Remarks If `pbase()` is non null the characters between `pbase()` and `pptr()` are written to the control sequence. The function `setbuf()` is overridden the `basic_filebuf` class.

27.5 Stream Buffers

27.5.2 Template class *basic_streambuf*<charT, traits>

Return Zero if successful and -1 if failure occurs.

27.5.2.4.3 Get Area

Description Virtual functions for extracting information from an input stream buffer. These functions should be overridden in derived classes.

basic_streambuf::showmanyc

Description Shows how many characters in an input stream

Prototype `virtual int showmanyc();`

Remarks If the function `showmanyc()` returns a positive value then calls to `underflow()` will succeed. If `showmanyc()` returns a negative number any calls to the functions `underflow()` and `uflow()` will fail.

Return Zero for normal behavior and negative or positive one.

basic_streambuf::xsgetn

Description To read a number of characters from an input stream buffer.

Prototype `virtual streamsize xsgetn
(char_type *s, streamsize n);`

Remarks The characters are read by repeated calls to `sbumpc()` until either `n` characters have been assigned or `EOF` is encountered.

Return The number of characters read.

basic_streambuf::underflow

Description To show an underflow condition and not increment the get pointer.

Prototype `virtual int_type underflow();`

Remarks The function `underflow()` is called when a character is not available for `sgetc()`.

There are many constraints for `underflow()`.

- The pending sequence of characters is a concatenation of end pointer minus the get pointer plus some sequence of characters to be read from input.
- The result character if the sequence is not empty the first character in the sequence or the next character in the sequence.
- The backup sequence if the beginning pointer is null, the sequence is empty, otherwise the sequence is the get pointer minus the beginning pointer.

Return The first character of the pending sequence and does not increment the get pointer. If the position is null returns `traits::eof()` to indicate failure.

`basic_streambuf::uflow`

Description To show a underflow condition for a single character and increment the get pointer.

Prototype `virtual int_type uflow();`

Remarks The function `uflow()` is called when a character is not available for `sbumpc()`.

The constraints are the same as `underflow()`, with the exceptions that the resultant character is transferred from the pending sequence to the back up sequence and the pending sequence may not be empty.

Return Calls `underflow()` and if `traits::eof` is not returned returns the integer value of the get pointer and increments the next pointer for input.

27.5 Stream Buffers

27.5.2 Template class *basic_streambuf*<charT, traits>

27.5.2.4.4 Putback

Virtual functions for replacing data to a stream. These functions should be overridden in derived classes.

basic_streambuf::pbackfail

Description	To show a failure in a put back operation.
Prototype	<pre>virtual int_type pbackfail (int_type c = traits::eof());</pre>
Remarks	The resulting conditions are the same as the function <code>underflow()</code> .
Return	The function <code>pbackfail()</code> is only called when a put back operation really has failed and returns <code>traits::eof</code> . If success occurs the return is undefined.

27.5.2.4.5 Put Area

Virtual function for inserting data into an output stream buffer. These functions should be overridden in derived classes.

basic_streambuf::xspn

Description	Write a number of characters to an output buffer.
Prototype	<pre>virtual streamsize xspn (const char_type *s, streamsize n);</pre>
Remarks	The function <code>xspn()</code> writes to the output character by using repeated calls to <code>sputc(c)</code> . Write stops when <code>n</code> characters have been written or <code>EOF</code> is encountered.
Return	The number of characters written in a type <code>streamsize</code> .

basic_streambuf::overflow

Description Consumes the pending characters of an output sequence.

Prototype `virtual int_type overflow
(int_type c = traits::eof());`

Remarks The pending sequence is defined as the concatenation of the `put pointer` minus the `beginning pointer` plus either the sequence of characters or an empty sequence, unless the `beginning pointer` is null in which case the pending sequence is an empty sequence.

This function is called by `sputc()` and `sputn()` when the buffer is not large enough to hold the output sequence.

Overriding this function requires that:

- When overridden by a derived class how characters are consumed must be specified.
- After the overflow either the `beginning pointer` must be null or the `beginning` and `put pointer` must both be set to the same non-null value.
- The function may fail if appending characters to an output stream fails or failure to set the previous requirement occurs.

Return The function returns `traits::eof()` for failure or some unspecified result to indicate success.

27.5 Stream Buffers

27.5.2 Template class *basic_streambuf*<charT, traits>



27.6 Formatting And Manipulators

This chapter discusses formatting and manipulators in the input/output library.

Overview of Formatting and Manipulators

There are three headers—`<istream>`, `<ostream>`, and `<iomanip>`—that contain stream formatting and manipulator routines and implementations.

The sections in this chapter are:

- [“Headers” on page 521](#)
- [“27.6.1 Input Streams” on page 523](#)
- [“27.6.2 Output streams” on page 557](#)
- [“27.6.3 Standard manipulators” on page 580](#)

Headers

This section lists the header for `istream`, `ostream`, and `iomanip`.

Header `<istream>`

Prototype

```
#include <ios>
namespace std{
template
    <class charT, class traits = ios_traits<charT> >
class basic_istream;
```

27.6 Formatting And Manipulators

Headers

```
typedef basic_istream<char> istream;
typedef basic_istream<wchar_t> wistream;

template
    <class charT, class traits>
basic_istream<charT, traits> &ws
    (basic_istream<charT,traits> (is);
}
```

Header <ostream>

```
#include <ios>
namespace std{
template
    <class charT, class traits = ios_traits<charT> >
class basic_ostream;

typedef basic_ostream<char> ostream;
typedef basic_ostream<wchar_t> wostream;

template
    <class charT, class traits>
basic_ostream<charT, traits> &endl
    (basic_ostream<charT,traits>& os);

template
    <class charT, class traits>
basic_ostream<charT, traits> &ends
    (basic_ostream<charT,traits>& os);

template
    <class charT, class traits>
basic_ostream<charT, traits> &flush
    (basic_ostream<charT,traits>& os);
}
```

Header <iomanip>

```
#include <ios>
namespace std {
// return types are unspecified
T1 resetiosflags(ios_base::fmtflags mask);
T2 setiosflags (ios_base::fmtflag mask);
T3 setbase(int base);
T4 setfill(int c);
T5 setprecision(int n);
T6 setw(int n);
}
```

27.6.1 Input Streams

The header <istream> controls input from a stream buffer.

The topics in this section are:

- [“27.6.1.1 Template class basic_istream” on page 523](#)
- [“27.6.1.1.1 basic_istream Constructors” on page 526](#)
- [“27.6.1.1.2 Class basic_istream::sentry” on page 527](#)
- [“27.6.1.2 Formatted input functions” on page 529](#)
- [“27.6.1.2.1 Common requirements” on page 529](#)
- [“27.6.1.2.2 Arithmetic Extractors Operator >>” on page 529](#)
- [“27.6.1.2.3 basic_istream extractor operator >>” on page 530](#)
- [“27.6.1.3 Unformatted input functions” on page 535](#)
- [“27.6.1.4 Standard basic_istream manipulators” on page 554](#)
- [“27.6.1.4.1 basic_iostream Constructor” on page 556](#)

27.6.1.1 Template class basic_istream

Description	A class that defines several functions for stream input mechanisms from a controlled stream buffer.
--------------------	-----------------------------------------------------------------------------------------------------

27.6 Formatting And Manipulators

27.6.1 Input Streams

Prototype

```
namespace std{
template
    <class charT, class traits = ios_traits<charT> >
class basic_istream : virtual public basic_ios<charT, traits> {
public:
    typedef charT
    typedef typename traits::int_type int_type;
    typedef typename traits::pos_type pos_type;
    typedef typename traits::off_type off_type;

    explicit basic_istream
    (basic_streambuf<charT, traits>* sb);

    virtual ~basic_istream();

    class sentry;

    basic_istream<charT, traits>& operator >>
        (basic_istream<charT, traits>& (*pf)
        (basic_istream<charT, traits>&))
    basic_istream<charT, traits>& operator >>
        (basic_ios<charT, traits>& (*pf)
        (basic_ios<charT, traits>&))
    basic_istream<charT, traits>& operator >>
        (char_type *s);
    basic_istream<charT, traits>& operator >>
        (char_type& c);
    basic_istream<charT, traits>& operator >>
        (bool& n);
    basic_istream<charT, traits>& operator >>
        (short& n);
    basic_istream<charT, traits>& operator >>
        (unsigned short& n);
    basic_istream<charT, traits>& operator >>
        (int& n);
    basic_istream<charT, traits>& operator >>
        (unsigned int& n);
    basic_istream<charT, traits>& operator >>
        ( long& n);
```

```
basic_istream<charT, traits>& operator >>
    (unsigned long& n);
basic_istream<charT, traits>& operator >>
    (float& f);
basic_istream<charT, traits>& operator >>
    (double& f);
basic_istream<charT, traits>& operator >>
    (long double & f);
basic_istream<charT, traits>& operator >>
    (void*& p);
basic_istream<charT, traits>& operator >>
    (basic_streambuf<char_type, traits>* sb);

streamsize gcount() const;
int_type get();
basic_istream<charT, traits>& get
    (char_type& c);
basic_istream<charT, traits>& get
    (char_type* s,
     streamsize n,
     char_type delim = traits::newline());
basic_istream<charT, traits>& get
    (basic_streambuf<char_type,
     traits>& sb,
     char_type delim = traits::newline());

basic_istream<charT, traits>& getline
    (char_type* s,
     streamsize n,
     char_type delim = traits::newline());

basic_istream<charT, traits>& ignore
    (streamsize n = 1,
     int_type delim = traits::eof());

int_type peek();

basic_istream<charT, traits>& read
    (char_type* s, streamsize n);
streamsize readsome(charT_type* s, streamsize n);
```

27.6 Formatting And Manipulators

27.6.1 Input Streams

```
basic_istream<charT, traits>& putback(char_type c);
basic_istream<charT, traits>& unget();

int sync();

pos_type tellg();
basic_istream<charT, traits>& seekg
    (pos_type);
basic_istream<charT, traits>& seekg
    (off_type, ios_base::seekdir);
};
}
```

Remarks The `basic_istream` class is derived from the `basic_ios` class and provides many functions for input operations.

27.6.1.1.1 `basic_istream` Constructors

constructor

Description Creates an `basic_istream` object.

Prototype `explicit basic_istream
(basic_streambuf<charT, traits>* sb);`

Remarks The `basic_istream` constructor is overloaded. It can be created as a base class with no arguments. It may be a simple input class initialized to a previous object's stream buffer.

Destructor

Description Destroy the `basic_istream` object.

Prototype `virtual ~basic_istream()`

Remarks The `basic_istream` destructor removes from memory the `basic_istream` object.

Listing 18.1 Example of basic_istream() usage:

MW Reference file contains

Ask the teacher anything you want to know

```
#include <iostream>
#include <fstream>
#include <cstdlib>

int main()
{
    using namespace std;

    ofstream out("MW Reference", ios::out | ios::in);
    if(!out.is_open())
        {cout << "file did not open"; exit(1);}

    istream inOut(out.rdbuf());

    char c;
    while(inOut.get(c)) cout.put(c);

    return 0;
}
```

Result:

Ask the teacher anything you want to know

27.6.1.1.2 Class basic_istream::sentry

Description A class for exception safe prefix and suffix operations.

Prototype

```
namespace std {
template
    <class charT,
    class traits = char_traits<charT> >
class basic_istream<charT, traits>::sentry {
    bool ok_;
public:
    explicit sentry
```

27.6 Formatting And Manipulators

27.6.1 Input Streams

```
        (basic_istream<charT,  
        traits>& is,  
        bool noskipws = false);  
    ~sentry();  
    operator bool() {return ok_;}  
};  
}
```

Class `basic_istream::sentry` Constructor

Constructor

Description Prepare for formatted or unformatted input

Prototype `explicit sentry
(basic_istream<charT,
traits>& is,
bool noskipws = false);`

Remarks If after the operation `is.good()` is true `ok_ equals true` otherwise `ok_ equals false`. The constructor may call `set-state(failbit)` which may throw an exception.

Destructor

Prototype `~sentry();`

Remarks The destructor has no effects.

`sentry::Operator bool`

Description To return the value of the data member `ok_`.

Prototype `operator bool();`

Return Operator `bool` returns the value of `ok_`

27.6.1.2 Formatted input functions

Formatted function provide mechanisms for input operations of specific types.

27.6.1.2.1 Common requirements

Each formatted input function begins by calling `ipfx()` and if the scan fails for any reason calls `setstate(failbit)`. The behavior of the scan functions are “as if” it was `fscanf()`.

27.6.1.2.2 Arithmetic Extractors Operator >>

Description Extractors that provide formatted arithmetic input operation.

Prototype

```
basic_istream<charT, traits>& operator >>
    (bool & n);
basic_istream<charT, traits>& operator >>
    (short &n);
```

Remarks: Extracts a short integer value and stores it in `n`.

```
basic_istream<charT, traits>& operator >>
    (unsigned short & n);
basic_istream<charT, traits>& operator >>
    (int & n);
basic_istream<charT, traits>& operator >>
    (unsigned int &n);
basic_istream<charT, traits>& operator >>
    (long & n);
basic_istream<charT, traits>& operator >>
    (unsigned long & n);
basic_istream<charT, traits>& operator >>
    (float & f);
basic_istream<charT, traits>& operator >>
    (double& f);
basic_istream<charT, traits>& operator >>(
    long double& f);
```

Remarks The Arithmetic extractors extract a specific type from the input stream and store it in the address provided

27.6 Formatting And Manipulators

27.6.1 Input Streams

Table 18.1 States and stdio equivalents

state	stdio equivalent
(flags() & basefield) == oct	%o
(flags() & basefield) == hex	%x
(flags() & basefield) != 0	%x
(flags() & basefield) == 0	%i
Otherwise	
signed integral type	%d
unsigned integral type	%u

27.6.1.2.3 basic_istream extractor operator >>

Description Extracts characters or sequences of characters and converts if necessary to numerical data.

Prototype `basic_istream<charT, traits>& operator >>
basic_istream<charT, traits>& (*pf)
(basic_istream<charT, traits>&)`

Remarks Returns pf(*this).
`basic_istream<charT, traits>& operator >>
(basic_ios<charT, traits>& (*pf)
(basic_ios<charT, traits>&))`

Remarks Calls pf(*this) then returns *this.
`basic_istream<charT, traits>& operator >>
(char_type *s);`

Remarks Extracts a char array and stores it in s if possible otherwise call setstate(failbit). If width() is set greater than zero width()-1 elements are extracted else up to size of s-1 elements are extracted. Scan stops with a whitespace “as if” in fscanf().
`basic_istream<charT, traits>& operator >>
(char_type& c);`

Remarks Extracts a single character and stores it in c if possible otherwise call setstate(failbit).

```
basic_istream<charT, traits>& operator >>
(void*& p);
```

Remarks Converts a pointer to void and stores it in p.

```
basic_istream<charT, traits>& operator >>
(basic_streambuf<char_type, traits>* sb);
```

Remarks Extracts a `basic_streambuf` type and stores it in sb if possible otherwise call `setstate(failbit)`.

Remarks The various overloaded extractors are used to obtain formatted input dependent upon the type of the argument. Since they return a reference to the calling stream they may be chained in a series of extractions. The overloaded extractors work “as if” like `fscanf()` in standard C and read until a white space character or EOF is encountered.

NOTE: The white space character is not extracted and is not discarded, but simply ignored. Be careful when mixing unformatted input operations with the formatted extractor operators. Such as when using console input.

Returns The `this` pointer is returned.

See Also `basic_ostream::operator <<`

Listing 18.2 Example of `basic_istream::` extractor usage:

The MW Reference input file contains
float 33.33 double 3.16e+10 Integer 789 character C

```
#include <iostream>
#include <fstream>
#include <cstdlib>
```

```
char ioFile[81] = "MW Reference";
```

```
int main()
{
using namespace std;
```

27.6 Formatting And Manipulators

27.6.1 Input Streams

```
ifstream in(ioFile);
if(!in.is_open())
{cout << "cannot open file for input"; exit(1);}

char type[20];
double d;
int i;
char ch;

in  >> type >> d;
cout << type << " " << d << endl;
in  >> type >> d;
cout << type << " " << d << endl;
in  >> type >> i;
cout << type << " " << i << endl;
in  >> type >> ch;
cout << type << " " << ch << endl;

cout << "\nEnter an integer: ";
cin >> i;
cout << "Enter a word: ";
cin >> type;
cout << "Enter a character \ "
    << "then a space then a double: ";
cin >> ch >> d;

cout << i << " " << type << " "
    << ch << " " << d << endl;

in.close();

return 0;
}
```

Result:

```
float 33.33
double 3.16e+10
Integer 789
character C
```

```
Enter an integer: 123 <enter>
Enter a word: Metrowerks <enter>
Enter a character then a space then a double: a 12.34 <enter>
123 Metrowerks a 12.34
```

Overloading Extractors:

Description	To provide custom formatted data retrieval.
Prototype	<pre>extractor prototype Basic_istream &operator >> (basic_istream &s,const imanip<T>&) { // procedures return s; }</pre>
Remarks	You may overload the extractor operator to tailor the specific needs of a particular class.
Returns	The this pointer is returned.

Listing 18.3 Example of basic_istream overloaded extractor usage:

```
#include <iostream>
#include <iomanip>
#include <cstdlib>
#include <cstring>

class phonebook {
    friend std::ostream &operator<<(std::ostream &stream,
        phonebook o);
    friend std::istream &operator>>(std::istream &stream,
        phonebook &o);

private:
    char name[80];
    int areacode;
```

27.6 Formatting And Manipulators

27.6.1 Input Streams

```
int exchange;
int num;

public:
void putname() {std::cout << num;}
phonebook() {} // default constructor
phonebook(char *n, int a, int p, int nm)
    {std::strcpy(name, n); areacode = a;
     exchange = p; num = nm;}
};

int main()
{
using namespace std;
    phonebook a;

    cin >> a;
    cout << a;

    return 0;
}

std::ostream &operator<<(std::ostream &stream, phonebook o)
{
using namespace std;

    stream << o.name << " ";
    stream << "(" << o.areacode << ") ";
    stream << o.exchange << "-";
    cout << setfill('0') << setw(4) << o.num << "\n";
    return stream;
}

std::istream &operator>>(std::istream &stream, phonebook &o)
{
using namespace std;

    char buf[5];
    cout << "Enter the name: ";
    stream >> o.name;
```

```
cout << "Enter the area code: ";
stream >> o.areacode;
cout << "Enter exchange: ";
stream >> o.exchange;
cout << "Enter number: ";
stream >> buf;
o.num = atoi(buf);
cout << "\n";
return stream;
}
```

Result:

```
Enter the name: Metrowerks
Enter the area code: 512
Enter exchange: 873
Enter number: 4700
```

```
Metrowerks (512) 873-4700
```

27.6.1.3 Unformatted input functions

The various unformatted input functions all begin by construction an object of type `basic_istream::sentry` and ends by destroying the `sentry` object.

NOTE: Older versions of the library may begin by calling `ipfx()` and end by calling `isfx()` and returning the value specified.

basic_istream::gcount

Description To obtain the number of bytes read.

Prototype `streamsize gcount() const;`

Remarks Use the function `gcount ()` to obtain the number of bytes read by the last unformatted input function called by that object.

27.6 Formatting And Manipulators

27.6.1 Input Streams

Returns An int type count of the bytes read.

Listing 18.4 Example of basic_istream::gcount() usage:

```
#include <iostream>
#include <fstream>

const SIZE = 4;

struct stArray {
    int index;
    double dNum;
};

int main()
{
    using namespace std;

    ofstream fOut("test");
    if(!fOut.is_open())
        {cout << "can't open out file"; return 1;}

    stArray arr;
    short i;

    for(i = 1; i < SIZE+1; i++)
    {
        arr.index = i;
        arr.dNum = i * 3.14;
        fOut.write((char *) &arr, sizeof(stArray));
    }
    fOut.close();

    stArray aIn[SIZE];

    ifstream fIn("test");
    if(!fIn.is_open())
        {cout << "can't open in file"; return 2;}

    long count =0;
```



```
for(i = 0; i < SIZE; i++)
{
    fIn.read((char *) &aIn[i], sizeof(stArray));

    count+=fIn.gcount();
}

cout << count << " bytes read " << endl;
cout << "The size of the structure is "
    << sizeof(stArray) << endl;
for(i = 0; i < SIZE; i++)
cout << aIn[i].index << " " << aIn[i].dNum
    << endl;

fIn.close();

return 0;
}
```

Result:

```
48 bytes read
The size of the structure is 12
1 3.14
2 6.28
3 9.42
4 12.56
```

basic_istream::get

Description Overloaded functions to retrieve a `char` or a `char` sequence from an input stream.

Prototype `int_type get();`

Remarks Extracts a character if available and returns that value. Else, calls `setstate(failbit)` and returns `eof()`.

`basic_istream<charT, traits>& get(char_type& c);`

Remarks Extracts a character and assigns it to `c` if possible else calls `setstate(failbit)`.

`basic_istream<charT, traits>& get`

27.6 Formatting And Manipulators

27.6.1 Input Streams

```
(char_type* s,  
streamsize n,  
char_type delim = traits::newline());
```

Remarks Extracts characters and stores them in a `char` array at an address pointed to by `s`, until

- A limit (the second argument minus one) or the number of characters to be stored is reached
- A delimiter (the default value is the newline character) is met. In which case, the delimiter is not extracted.
- If `end_of_file` is encountered in which case `setstate(eofbit)` is called.

If no characters are extracted calls `setstate(failbit)`. In any case it stores a `null` character in the next available location of array `s`.

```
basic_istream<charT, traits>& get  
(basic_streambuf<char_type,  
traits>& sb,  
char_type delim = traits::newline());
```

Remarks Extracts a characters and assigns them to the `basic_streambuf` object `sb` if possible else calls `setstate(failbit)`. Extraction stops if...

- an insertion fails
- end-of-file is encountered.
- an exception is thrown
- the next the next available character `c == delim` (in which case `c` is not extracted.)

Returns An integer when used with no argument. When used with an argument if a character is extracted the `get()` function returns The `this` pointer. If no character is extracted `setstate(failbit)` is called. In any case a `null char` is appended to the array.

See Also `getline()`

Listing 18.5 Example of `basic_istream::get()` usage:

READ ONE CHARACTER:

MW Reference file for input

float 33.33 double 3.16e+10 Integer 789 character C

```
#include <iostream>
```

```
#include <fstream>
```

```
#include <cstdlib>
```

```
int main()
```

```
{
```

```
using namespace std;
```

```
    char inFile[] = "MW Reference";
```

```
    ifstream in(inFile);
```

```
    if(!in.is_open())
```

```
    {cout << "Cannot open input file"; exit(1);}
```

```
    char ch;
```

```
    while(in.get(ch)) cout << ch;
```

```
    return 0;
```

```
}
```

Result:

float 33.33 double 3.16e+10 Integer 789 character C

READ ONE LINE:

```
#include <iostream>
```

```
#include <iostream>
```

```
const int size = 100;
```

```
char buf[size];
```

```
int main()
```

```
{
```

```
using namespace std;
```

27.6 Formatting And Manipulators

27.6.1 Input Streams

```
cout << " Enter your name: ";
cin.get(buf, size);
cout << buf;

return 0;
}
```

Result:

```
Enter your name: Metrowerks CodeWarrior <enter>
Metrowerks CodeWarrior
```

basic_istream::getline

Description To obtain a delimiter terminated character sequence from an input stream.

Prototype `basic_istream<charT, traits>& getline
(char_type* s,
streamsize n,
char_type delim = traits::newline());`

Remarks The unformatted `getline()` function retrieves character input, and stores it in a character array buffer `s` if possible until the following conditions evaluated in this order occur. If no characters are extracted `setstate(failbit)` is called.

- end-of-file occurs in which case `setstate eofbit)` is called.
- A delimiter (default value is the newline character) is encountered. In which case the delimiter is read and extracted but not stored.
- A limit (the second argument minus one) is read.
- if `n-1` chars are read that `failbit` gets set.

In any case it stores a null char into the next successive location of the array.

Returns The this pointer is returned.

See Also `basic_ostream::flush()`

Listing 18.6 Example of `basic_istream::getline()` usage:

```
#include <iostream>
```

```
const int size = 120;
```

```
int main()
```

```
{
```

```
using namespace std;
```

```
    char compiler[size];
```

```
    cout << "Enter your compiler: ";
```

```
    cin.getline(compiler, size);
```

```
    cout << "You use " << compiler;
```

```
    return 0;
```

```
}
```

Result:

Enter your compiler: *Metrowerks CodeWarrior* <enter>

You use Metrowerks CodeWarrior

```
#include <iostream>
```

```
const int size = 120;
```

```
#define TAB '\t'
```

```
int main()
```

```
{
```

```
using namespace std;
```

```
    cout << "What kind of Compiler do you use: ";
```

```
    char compiler[size];
```

```
    cin.getline(compiler, size, TAB);
```

```
    cout << compiler;
```

```
    cout << "\nsecond input not needed\n";
```

27.6 Formatting And Manipulators

27.6.1 Input Streams

```
cin >> compiler;
cout << compiler;

return 0;
}
```

Result:

```
What kind of Compiler do you use:
Metrowerks CodeWarrior<tab>Why?
Metrowerks CodeWarrior
second input not needed
Why?
```

basic_istream::ignore

Description To extract and discard a number of characters.

Prototype `basic_istream<charT, traits>& ignore
(streamsize n = 1,
int_type delim = traits::eof());`

Remarks The function `ignore()` will extract and discard characters until

- A limit is met (the first argument)
- end-of-file is encountered (in which case `setstate(eofbit)` is called.)
- The next character `c` is equal to the delimiter `delim`, in which case it is extracted except when `c` is equal to `traits::eof()`;

Returns The `this` pointer is returned.

Listing 18.7 Example of `basic_istream::ignore()` usage:

The file `MW Reference` contains:

```
char ch; // to save char
/*This C comment will remain */
while((ch = in.get())!= EOF) cout.put(ch);
// read until failure
```

27.6 Formatting And Manipulators

27.6.1 Input Streams

```
/* the C++ comments won't */

#include <iostream>
#include <fstream>
#include <cstdlib>

char inFile[] = "MW Reference";
char bslash = '/';

int main()
{
using namespace std;

    ifstream in(inFile);
    if(!in.is_open())
        {cout << "file not opened"; exit(1);}

    char ch;
    while((ch = in.get()) != EOF)
    {
        if(ch == bslash && in.peek() == bslash)
        {
            in.ignore(100, '\n');
            cout << '\n';
        }
        else    cout << ch;
    }

    return 0;
}
```

Result:

```
char ch;
    /*This C comment will remain */
while((ch = in.get())!= EOF) cout.put(ch);

/* the C++ comments won't */
```

27.6 Formatting And Manipulators

27.6.1 Input Streams

basic_istream::peek

Description	To view at the next character to be extracted.
Prototype	<code>int_type peek();</code>
Remarks	The function <code>peek()</code> allows you to look ahead at the next character in a stream to be extracted without extracting it.
Returns	If <code>good()</code> is false returns <code>traits::eof()</code> else returns the value of the next character in the stream.

Listing 18.8 Example of basic_istream::peek() usage:

See `basic_istream::ignore()`

basic_istream::read

Description	To obtain a block of binary data from and input stream.
Prototype	<code>basic_istream<charT, traits>& read (char_type* s, streamsize n);</code>
Remarks	<p>The function <code>read()</code> will attempt to extract a block of binary data until the following conditions are met.</p> <ul style="list-style-type: none">• A limit of <code>n</code> number of characters are stored.• end-of-file is encountered on the input (in which case <code>setstate(failbit)</code> is called.
Returns	The <code>this</code> pointer is returned.
See Also	<code>write()</code>

Listing 18.9 Example of basic_istream::read() usage:

```
#include <iostream>
#include <fstream>
#include <iomanip>
```



```
#include <cstdlib>
#include <cstring>

struct stock {
    char name[80];
    double price;
    long trades;
};

char *Exchange = "BBSE";
char *Company = "Big Bucks Inc.";

int main()
{
    using namespace std;

    stock Opening, Closing;

    strcpy(Opening.name, Company);
    Opening.price = 180.25;
    Opening.trades = 581300;

    // open file for output
    ofstream Market(Exchange, ios::out | ios::trunc | ios::binary);
    if(!Market.is_open())
        {cout << "can't open file for output"; exit(1);}

    Market.write((char*) &Opening, sizeof(stock));
    Market.close();

    // open file for input
    ifstream Market2(Exchange, ios::in | ios::binary);
    if(!Market2.is_open())
        {cout << "can't open file for input"; exit(2);}

    Market2.read((char*) &Closing, sizeof(stock));

    cout << Closing.name << "\n"
         << "The number of trades was: " << Closing.trades << "\n";
    cout << fixed << setprecision(2)
```

27.6 Formatting And Manipulators

27.6.1 Input Streams

```
<< "The closing price is: $" << Closing.price << endl;

Market2.close();

return 0;
}
```

Result:

Big Bucks Inc.

The number of trades was: 581300

The closing price is: \$180.25

basic_istream::readsome

Description Extracts characters and stores them in an array.

Prototype `streamsize readsome
(charT_type* s, streamsize n);`

Remarks The function `readsome` extracts and stores characters storing them in the buffer pointed to by `s` until the following conditions are met.

- end-of-file is encountered (in which case `setstate(eofbit)` is called.)
- No characters are extracted.
- A limit of characters is extracted either `n` or the size of the buffer.

Returns The number of characters extracted.

Listing 18.10 Example of `basic_istream::readsome()` usage.

The file MW Reference contains:
Metrowerks CodeWarrior
Software at Work
Registered Trademark

```
#include <iostream>
#include <fstream>
#include <sstream>
#include <cstdlib>

const short SIZE = 81;

int main()
{
    using namespace std;

    ifstream in("MW Reference");
    if(!in.is_open())
        {cout << "can't open file for input"; exit(1);}

    char Buffer[SIZE] = "\0";
    ostringstream Paragraph;

    while(in.good() && (in.peek() != EOF))
    {
        in.readsome(Buffer, 5);
        Paragraph << Buffer;
    }

    cout << Paragraph.str();

    in.close();
    return 0;
}
```

Result:

Metrowerks CodeWarrior
Software at Work
Registered Trademark

basic_istream::putback

Description To replace a previously extracted character.

27.6 Formatting And Manipulators

27.6.1 Input Streams

Prototype `basic_istream<charT, traits>& putback
 (char_type c);`

Remarks The function `putback()` allows you to replace the last character extracted by calling `rdbuf()->sungetc()`. If the buffer is empty, or if `sungetc()` returns `eof`, `setstate(failbit)` may be called.

Returns The `this` pointer is returned.

See Also `sungetc()`

Listing 18.11 Example of `basic_istream::putback` usage:

The file `MW Reference` contains.

```
char ch;                // to save char
    /* comment will remain */
while((ch = in.get()) != EOF) cout.put(ch);
// read until failure
```

```
#include <iostream>
#include <fstream>
#include <stdlib.h>
```

```
char inFile[] = "MW Reference";
char bslash = '/';
```

```
int main()
{
using namespace std;
```

```
    ifstream in(inFile);
    if(!in.is_open())
    {cout << "file not opened"; exit(1);}
}
```

```
char ch, tmp;
while((ch = in.get()) != EOF)
{
    if(ch == bslash)
    {
        in.get(tmp);
    }
}
```

```
        if(tmp != bslash)
            in.putback(tmp);
        else continue;
    }
    cout << ch;
}

return 0;
}
```

Result:

```
char ch;                to save char
    /* comment will remain */
while((ch = in.get()) != EOF) cout.put(ch);
    read until failure
```

basic_istream::unget

Description To replace a previously extracted character.

Prototype `basic_istream<charT, traits>&unget();`

Remarks Use the function `unget()` to return the previously extracted character. If `rdbuf()` is null or if end-of-file is encountered `setstate(badbit)` is called.

Returns The this pointer is returned.

See Also `putback()`, `ignore()`

Listing 18.12 Example of basic_istream::unget() usage:

The file MW Reference contains:

```
char ch;                // to save char
    /* comment will remain */
    // read until failure
while((ch = in.get()) != EOF) cout.put(ch);
```

27.6 Formatting And Manipulators

27.6.1 Input Streams

```
#include <iostream>
#include <fstream>
#include <cstdlib>

char inFile[] = "MW Reference";
char bslash = '/';

int main()
{
    using namespace std;

    ifstream in(inFile);
    if(!in.is_open())
    {cout << "file not opened"; exit(1);}

    char ch, tmp;
    while((ch = in.get()) != EOF)
    {
        if(ch == bslash)
        {
            in.get(tmp);
            if(tmp != bslash)
                in.unget();
            else continue;
        }
        cout << ch;
    }

    return 0;
}
```

Result:

```
char ch;                to save char
    /* comment will remain */
    read until failure
while((ch = in.get()) != EOF) cout.put(ch);
```

basic_istream::sync

Description	To synchronize input and output
Prototype	<code>int sync();</code>
Remarks	<p>This functions attempts to make the input source consistent with the stream being extracted.</p> <p>If <code>rdbuf()->pubsync()</code> returns <code>-1</code> <code>setstate(badbit)</code> is called and <code>traits::eof</code> is returned.</p>
Returns	If <code>rdbuf()</code> is <code>Null</code> returns <code>-1</code> otherwise returns zero.

Listing 18.13 Example of basic_istream::sync() usage:

The file MW Reference contains:
 This functions attempts to make the input source
 consistent with the stream being extracted.
 --
 Metrowerks CodeWarrior "Software at Work"

```
#include <iostream>
#include <fstream>
#include <cstdlib>

char inFile[] = "MW Reference";

int main()
{
    using namespace std;

    ifstream in(inFile);
    if(!in.is_open())
        {cout << "could not open file"; exit(1);}

    char str[10];
    if(in.sync())// return 0 if successful
        { cout << "cannot sync"; exit(1); }
    while (in.good())
```

27.6 Formatting And Manipulators

27.6.1 Input Streams

```
{
    in.get(str, 10, EOF);
    cout <<str;
}
return 0;
}
```

Result:

This functions attempts to make the input source consistent with the stream being extracted.

--

Metrowerks CodeWarrior "Software at Work"

basic_istream::tellg

Description To determine the offset of the get pointer in a stream

Prototype `pos_type tellg();`

Remarks The function `tellg` calls `rdbuf()->pubseekoff(0, cur, in)`.

Returns The current offset as a `pos_type` if successful else returns `-1`.

See Also `basic_streambuf::pubseekoff()`

Listing 18.14 Example of `basic_istream::tellg()` usage:

See `basic_istream::seekg()`

basic_istream::seekg

Description To move to a variable position in a stream.

Prototype `basic_istream<charT, traits>& seekg(pos_type);`
`basic_istream<charT, traits>& seekg`
`(off_type, ios_base::seekdir dir);`

Remarks The function `seekg` is overloaded to take a `pos_type` object, or an `off_type` object (defined in `basic_ios` class.) The function is used to set the position of the `get` pointer of a stream to a random location for character extraction.

Returns The `this` pointer is returned.

See Also `basic_streambuf::pubseekoff()` and `pubseekpos()`.

Listing 18.15 Example of `basic_istream::seekg()` usage:

The file `MW Reference` contains:
`ABCDEFGHIJKLMNOPQRSTUVWXYZ`

```
#include <iostream>
#include <fstream>
#include <cstdlib>

int main()
{
    using namespace std;

    ifstream in("MW Reference");
    if(!in.is_open())
        {cout << "could not open file"; exit(1);}

    streampos spEnd, spStart, aCheck;
    spEnd = spStart = 5;

    in.seekg(spStart);
    aCheck = in.tellg();
    cout << "The offset at the start of the reading in bytes is "
         << aCheck.offset() << endl;

    char ch;
    while(spEnd != spStart+10)
    {
        in.get(ch);
        cout << ch;
        spEnd = in.tellg();
    }
}
```

27.6 Formatting And Manipulators

27.6.1 Input Streams

```
}

aCheck = in.tellg();
cout << "\nThe current position's offset in bytes now is "
    << aCheck.offset() << endl;
streamoff gSet = 0;
in.seekg(gSet, ios::beg);

aCheck = in.tellg();
cout << "The final position's offset in bytes now is "
    << aCheck.offset() << endl;

in.close();
return 0;
}
```

Result:

The offset at the start of the reading in bytes is 5

FGHIJKLMNO

The current position's offset in bytes now is 15

The final position's offset in bytes now is 0

27.6.1.4 Standard `basic_istream` manipulators

`basic_ifstream::ws`

Description To provide inline style formatting.

Prototype

```
template
    <class charT, class traits>
basic_istream<charT, traits> &ws
    (basic_istream<charT,traits>& is);
```

Remarks The `ws` manipulator skips whitespace characters in input.

Returns The `this` pointer.

Listing 18.16 Example of basic_istream:: manipulator ws usage:

The file MW Reference (where the number of blanks (and/or tabs) is unknown) contains:

a b c

```
#include <iostream>
#include <fstream>
#include <cstdlib>

int main()
{
    char * inFileName = "MW Reference";

    ifstream in(inFileName);
    if (!in.is_open())
    {cout << "Couldn't open for input\n"; exit(1);}

    char ch;
    in.unsetf(ios::skipws);

    cout << "Does not skip whitespace\n|";
    while (1)
    {
        in >> ch; // does not skip white spaces
        if (in.good())
            cout << ch;
        else break;
    }
    cout << "|\n\n";

    //reset file position
    in.clear();
    in.seekg(0, ios::beg);

    cout << "Does skip whitespace\n|";
    while (1)
    {
        in >> ws >> ch; // ignore white spaces

        if (in.good())
```

27.6 Formatting And Manipulators

27.6.1 Input Streams

```
        cout << ch;
    else break;
}
cout << "|" << endl;

in.close();
return(0);
}
```

Result:

Does not skip whitespace

| a b c|

Does skip whitespace

|abc|

27.6.1.4.1 basic_iostream Constructor

Constructor

Description Constructs an and destroy object of the class basic_iostream.

Prototype `explicit basic_iostream
(basic_streambuf<charT, traits>* (sb);`

Remarks Calls basic_istream(<charT, traits> (sb) and basic_ostream(charT, traits>* (sb). After it is constructed rdbuf() equals sb and gcount() equals zero.

Destructor

Prototype `virtual ~basic_iostream();`

Remarks Destroys an object of type basic_iostream.

27.6.2 Output streams

The include file `<ostream>` includes classes and types that provide output stream mechanisms.

The topics in this section are:

- [“27.6.2.1 Template class basic_ostream” on page 557](#)
- [“27.6.2.2 basic_ostream Constructor” on page 559](#)
- [“Class basic_ostream::sentry Constructor” on page 561](#)
- [“27.6.2.3 Class basic_ostream::sentry” on page 560](#)
- [“27.6.2.4 Formatted output functions” on page 562](#)
- [“27.6.2.4.1 Common requirements” on page 562](#)
- [“27.6.2.4.2 Arithmetic Inserter Operator <<” on page 562](#)
- [“27.6.2.4.3 basic_ostream::operator<<” on page 564](#)
- [“27.6.2.5 Unformatted output functions” on page 568](#)
- [“27.6.2.6 Standard basic_ostream manipulators” on page 575](#)

27.6.2.1 Template class basic_ostream

A class for stream output mechanisms.

Prototype

```
namespace std{
template
    <class charT, class traits = ios_traits<charT> >
class basic_ostream : virtual public basic_ios<charT, traits>{
public:
    // Types:
    typedef charTchar_type;
    typedef typename traits::int_type int_type;
    typedef typename traits::pos_type pos_type;
    typedef typename traits::off_type off_type;

    explicit basic_ostream (basic_streambuf<char_type, traits>*sb);
    virtual ~basic_ostream();
```

27.6 Formatting And Manipulators

27.6.2 Output streams

```
class sentry;

basic_ostream<charT, traits>& operator<<
    (basic_ostream<charT, traits>&(*pf)
    (basic_ostream<charT, traits>&));
basic_ostream<charT, traits>& operator<<
    (basic_ostream<charT, traits>&(*pf)
    (basic_ios<charT, traits>&));
basic_ostream<charT, traits>& operator<<
    (const char_type *s)
basic_ostream<charT, traits>& operator<<
    (char_type c)
basic_ostream<charT, traits>& operator<<
    (bool n)
basic_ostream<charT, traits>& operator<<
    (short n)
basic_ostream<charT, traits>& operator<<
    (unsigned short n)
basic_ostream<charT, traits>& operator<<
    (int n)
basic_ostream<charT, traits>& operator<<
    (unsigned int n)
basic_ostream<charT, traits>& operator<<
    (long n)
basic_ostream<charT, traits>& operator<<
    (unsigned long n)
basic_ostream<charT, traits>& operator<<
    (float f)
basic_ostream<charT, traits>& operator<<
    (double f)
basic_ostream<charT, traits>& operator<<
    (long double f)
basic_ostream<charT, traits>& operator<<
    (void p)
basic_ostream<charT, traits>& operator<<
    (basic_streambuf<char_type, traits>* sb);)

basic_ostream<charT, traits>& put(char_type c);

basic_ostream<charT, traits>& write
```

```

        (const char_type* s, streamsize n);

basic_ostream<charT, traits>& flush();

pos_type tellp();
basic_ostream<charT, traits>& seekp(pos_type);
basic_ostream<charT, traits>& seekp
    (off_type, ios_base::seekdir);
};
}

```

Remarks The `basic_ostream` class provides for output stream mechanisms for output stream classes. The `basic_ostream` class may be used as a independent class, as a base class for the `basic_ofstream` class or a user derived classes.

27.6.2.2 basic_ostream Constructor

Description To create and remove from memory `basic_ostream` object for stream output.

Prototype `explicit basic_ostream
(basic_streambuf<char_type, traits>*sb);`

Remarks The `basic_ostream` constructor constructs and initializes the base class object.

Destructor

Prototype `virtual ~basic_ostream();`

Remarks Removes a `basic_ostream` object from memory.

Listing 18.17 Example of `basic_ostream()` usage:

The MW Reference file contains originally
Metrowerks CodeWarrior "Software at Work"

27.6 Formatting And Manipulators

27.6.2 Output streams

```
#include <iostream>
#include <fstream>
#include <cstdlib>

char inFile[] = "MW Reference";

int main()
{
    using namespace std;

    ifstream inOut(inFile, ios::in | ios::out);
    if(!inOut.is_open())
        {cout << "Could not open file"; exit(1);}
    ostream Out(inOut.rdbuf());

    char str[] = "\nRegistered Trademark";

    inOut.rdbuf()->pubseekoff(0, ios::end);

    Out << str;

    inOut.close();

    return 0;
}
```

Result:
The File now reads:
Metrowerks CodeWarrior "Software at Work"
Registered Trademark

27.6.2.3 Class `basic_ostream::sentry`

Description A class for exception safe prefix and suffix operations.

Prototype `namespace std {
 template
 < class charT,`


```

        class traits = char_traits<charT> >
class basic_ostream<charT, traits>::sentry
{
    bool ok_;
public:
    explicit sentry
        (basic_ostream<charT,
         traits>& os,
         bool noskipws = false);
    ~sentry();
    operator bool() {return ok_;}
};

```

Class basic_ostream::sentry Constructor

Constructor

Description Prepare for formatted or unformatted output

Prototype `explicit sentry
(basic_ostream<charT, traits>& os);`

Remarks If after the operation `os.good()` is true `ok_ equals true` otherwise `ok_ equals false`. The constructor may call `set-state(failbit)` which may throw an exception.

Destructor

Prototype `~sentry();`

Remarks The destructor under normal circumstances will call `os.flush()`.

sentry::Operator bool

Description To return the value of the data member `ok_`.

Prototype `operator bool();`

27.6 Formatting And Manipulators

27.6.2 Output streams

Return Operator `bool` returns the value of `ok_`

27.6.2.4 Formatted output functions

Description Formatted output functions provide a manner of inserting for output specific data types.

27.6.2.4.1 Common requirements

Remarks The operations begins by calling `opfx()` and ends by calling `osfx()` then returning the value specified for the formatted output.

Some output maybe generated by converting the scalar data type to a NTBS (null terminated bit string) text.

If the function fails for any for any reason the function calls `set-state(failbit)`.

27.6.2.4.2 Arithmetic Inserter Operator <<

Description To provide formatted insertion of types into a stream.

Prototype

```
basic_ostream<charT, traits>& operator<<
    (short n)
basic_ostream<charT, traits>& operator<<
    (unsigned short n)
basic_ostream<charT, traits>& operator<<
    (int n)
basic_ostream<charT, traits>& operator<<
    (unsigned int n)
basic_ostream<charT, traits>& operator<<
    (long n)
basic_ostream<charT, traits>& operator<<
    (unsigned long n)
basic_ostream<charT, traits>& operator<<
    (float f)
basic_ostream<charT, traits>& operator<<
    (double f)
basic_ostream<charT, traits>& operator<<
```

(long double f)

Remarks Converts an arithmetical value. The formatted values are converted "as if" they had the same behavior of the `fprintf()` function

Returns The `this` pointer is returned

Table 18.2 Output states and stdio equivalents.

Output State	stdio equivalent
Integers	
(flags() & basefield) == oct	%o
(flags() & basefield) == hex	%x
(flags() & basefield) != 0	%X
Otherwise	
signed integral type	%d
unsigned integral type	%u
Floating Point Numbers	
(flags() & floatfield) == fixed	%f
(flags() & floatfield) == scientific	%e
(flags() & uppercase) != 0	%E
Otherwise	
(flags() & uppercase) != 0	%g %G
An integral type other than a char type	
(flags() & showpos) != 0	+
(flags() & showbase) != 0	#
A floating point type	
(flags() & showpos) != 0	+
(flags() & showpoint) != 0	#

27.6 Formatting And Manipulators

27.6.2 Output streams

For any conversion if `width()` is non-zero then a field width a conversion specification has the value of `width()`.

For any conversion if `(flags() and fixed) !=0` or if `precision() >0` the conversion specification is the value of `precision()`.

For any conversion padding behaves in the following manner.

Table 18.3 Conversion state and stcio equivalents.

State	Justification	stdio equivalent
<code>(flags() & adjustfield) == left</code>	left	space padding
<code>(flags() & adjustfield) == internal</code>	Internal	zero padding
Otherwise	right	space padding

Remarks The `ostream` insertion operators are overloaded to provide for insertion of most predefined types into and output stream. They return a reference to the basic stream object so they may be used in a chain of statements to input various types to the same stream.

Returns In most cases `*this` is returned unless failure in which case `setstate(failbit)` is called.

27.6.2.4.3 `basic_ostream::operator<<`

Prototype `basic_ostream<charT, traits>& operator<<
 (basic_ostream<charT, traits>&
 (*pf)(basic_ostream<charT, traits>&));`

Remarks Returns `pf(*this)`.
`basic_ostream<charT, traits>& operator<<
 (basic_ostream<charT, traits>&
 (*pf)(basic_ios<charT, traits>&));`

Remarks Calls `pf(*this)` return `*this`.
`basic_ostream<charT, traits>& operator<<
 (const char_type *s)`

```
basic_ostream<charT, traits>& operator<<
    (char_type c)
basic_ostream<charT, traits>& operator<<
    (bool n)
```

Remarks Behaves depending on how the boolalpha flag is set.

```
basic_ostream<charT, traits>& operator<<
    (void p)
```

Remarks Converts the pointer to void p as if the specifier was %p and returns *this.

```
basic_ostream<charT, traits>& operator<<
    (basic_streambuf>char_type, traits>* sb);)
```

Remarks If sb is null calls setstate(failbit) otherwise gets characters from sb and inserts them into *this until:

- end-of-file occurs.
- inserting into the stream fails.
- an exception is thrown.

If the operation fails calls setstate(failbit) or re-throws the exception, otherwise returns *this.

Remarks The formatted output functions insert the values into the appropriate argument type.

Return Most inserters (unless noted otherwise) return the this pointer.

Listing 18.18 Example of basic_ostream inserter usage:

```
#include <iostream>
#include <fstream>
#include <cstdlib>

char oFile[81] = "MW Reference";

int main()
{
    using namespace std;
```

27.6 Formatting And Manipulators

27.6.2 Output streams

```
ofstream out(oFile);

out << "float " << 33.33;
out << " double " << 3.16e+10;
out << " Integer " << 789;
out << " character " << 'C' << endl;
out.close();

cout << "float " << 33.33;
cout << "\ndouble " << 3.16e+10;
cout << "\nInteger " << 789;
cout << "\ncharacter " << 'C' << endl;

return 0;
}
```

Result:

Output: to MWReference

float 33.33 double 3.16e+10 Integer 789 character C

Output to console

float 33.33

double 3.16e+10

Integer 789

character C

Overloading Inserters

Description To provide specialized output mechanisms for an object.

Prototype Overloading inserter prototype

```
basic_ostream &operator<<
    (basic_ostream &stream, const manip<T>&)
{
    // procedures;
    return stream;
}
```

Remarks You may overload the inserter operator to tailor it to the specific needs of a particular class.

Returns The `this` pointer.

Listing 18.19 Example of overloaded inserter usage:

```
#include <iostream>
#include <string.h>
#include <iomanip>

class phonebook {
    friend ostream &operator<<
        (ostream &stream, phonebook o);
protected:
    char *name;
    int areacode;
    int exchange;
    int num;
public:
    phonebook(char *n, int a, int p, int nm) :
        areacode(a),
        exchange(p),
        num(nm),
        name(n) {}
};

int main()
{
    using namespace std;

    phonebook a("Sales", 800, 377, 5416);
    phonebook b("Voice", 512, 873, 4700);
    phonebook c("Fax", 512, 873, 4900);

    cout << a << b << c;

    return 0;
}
```

27.6 Formatting And Manipulators

27.6.2 Output streams

```
std::ostream &operator<<(std::ostream &stream, phonebook o)
{
    stream << o.name << " ";
    stream << "(" << o.areacode << ") ";
    stream << o.exchange << "-";
    stream << setfill('0') << setw(4)
        << o.num << "\n";
    return stream;
}
```

Result:

Sales (800) 377-5416

Voice (512) 873-4700

Fax (512) 873-4900

27.6.2.5 Unformatted output functions

Each unformatted output function begins by creating an object of the class `sentry`. The unformatted output functions are ended by destroying the `sentry` object and may return a value specified.

`basic_ostream::tellp`

Description To return the offset of the put pointer in an output stream.

Prototype `pos_type tellp();`

Returns If `fail()` returns `-1` else returns `rdbuf()->pubseekoff(0, cur, out)`.

See Also `basic_istream::tellg()`, `seekp(0)`.

Listing 18.20 Example of `basic_ostream::tellp()` usage.

see `basic_ostream::seekp()`.

basic_ostream::seekp

Description Randomly move to a position in an output stream.

Prototype `basic_ostream<charT, traits>& seekp(pos_type);`

Prototype `basic_ostream<charT, traits>& seekp
(off_type, iosbase::seekdir);`

Remarks The function `seekp` is overloaded to take a single argument of a `pos_type pos` that calls `rdbuf()->pubseekpos(pos)`. It is also overloaded to take two arguments an `off_type off` and `ios_base::seekdir type dir` that calls `rdbuf()->pubseekoff(off, dir)`.

Returns The this pointer.

See Also `basic_istream seekg()`, `tellp()`

Listing 18.21 Example of basic_ostream::seekp() usage.

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

std::string motto = "Metrowerks CodeWarrior - Software at Work";

int main()
{
    using namespace std;

    ostringstream ostr(motto);
    streampos cur_pos, start_pos;

    cout << "The original array was :\n"
         << motto << "\n\n";
    // associate buffer
    stringbuf *strbuf(ostr.rdbuf());

    streamoff str_off = 10;
```

27.6 Formatting And Manipulators

27.6.2 Output streams

```
cur_pos = ostr.tellp();
cout << "The current position is "
      << cur_pos.offset()
      << " from the beginning\n";

ostr.seekp(str_off);

cur_pos = ostr.tellp();
cout << "The current position is "
      << cur_pos.offset()
      << " from the beginning\n";

strbuf->sputc('\0');

cout << "The stringbuf array is\n"
      << strbuf->str() << "\n\n";
cout << "The ostream array is still\n"
      << motto;

return 0;
}
```

Results:

The original array was :
Metrowerks CodeWarrior - Software at Work

The current position is 0 from the beginning
The current position is 10 from the beginning
The stringbuf array is
Metrowerks

The ostream array is still
Metrowerks CodeWarrior - Software at Work

basic_ostream::put

Description To place a single character in the output stream.

Prototype `basic_ostream<charT, traits>& put(char_type c);`

Remarks The unformatted function `put ()` inserts one character in the output stream. If the operation fails calls `setstate(badbit)`.

Returns The `this` pointer.

Listing 18.22 Example of `basic_ostream::put()` usage:

```
#include <iostream>

int main()
{
    using namespace std;

    char *str = "Metrowerks CodeWarrior \"Software at Work\"";
    while(*str)
    {
        cout.put(*str++);
    }
    return 0;
}
```

Result:

Metrowerks CodeWarrior "Software at Work"

`basic_ostream::write`

Description To insert a block of binary data into an output stream.

Prototype `basic_ostream<charT, traits>& write
(const char_type* s, streamsize n);`

Remarks The overloaded function `write ()` is used to insert a block of binary data into a stream. This function is can be used to write an object by casting that object as a `unsigned char` pointer. If the operation fails calls `setstate(badbit)`.

Returns A reference to `ostream`. (The `this` pointer.)

See Also `read ()`

27.6 Formatting And Manipulators

27.6.2 Output streams

Listing 18.23 Example of `basic_ostream::write()` usage:

```
#include <iostream>
#include <fstream>
#include <iomanip>
#include <cstdlib>
#include <cstring>

struct stock {
    char name[80];
    double price;
    long trades;
};

char *Exchange = "BBSE";
char *Company = "Big Bucks Inc.";

int main()
{
    using namespace std;

    stock Opening, Closing;

    strcpy(Opening.name, Company);
    Opening.price = 180.25;
    Opening.trades = 581300;

    // open file for output
    ofstream Market(Exchange,
        ios::out | ios::trunc | ios::binary);
    if(!Market.is_open())
    {cout << "can't open file for output"; exit(1);}

    Market.write((char*) &Opening, sizeof(stock));
    Market.close();

    // open file for input
    ifstream Market2(Exchange, ios::in | ios::binary);
    if(!Market2.is_open())
    {cout << "can't open file for input"; exit(2);}
```

```

Market2.read((char*) &Closing, sizeof(stock));

cout << Closing.name << "\n"
    << "The number of trades was: "
    << Closing.trades << '\n';
cout << fixed << setprecision(2)
    << "The closing price is: $"
    << Closing.price << endl;

Market2.close();

return 0;
}

```

Result:

Big Bucks Inc.

The number of trades was: 581300

The closing price is: \$180.25

basic_ostream::flush

Description To force the output buffer to release its contents.

Prototype `basic_ostream<charT, traits>& flush();`

Remarks The function `flush()` is an output only function in C++. You may use it for an immediate expulsion of the output buffer. This is useful when you have critical data or you need to ensure that a sequence of events occurs in a particular order. If the operation fails calls `set-state(badbit)`.

Returns The `this` pointer.

Listing 18.24 Example of `basic_ostream::flush()` usage:

```

#include <iostream>
#include <iomanip>
#include <ctime>

```

27.6 Formatting And Manipulators

27.6.2 Output streams

```
class stopwatch {
private:
    double begin, set, end;
public:
    stopwatch();
    ~stopwatch();
    void start();
    void stop();
};

stopwatch::stopwatch()
{
using namespace std;

    begin = (double) clock() / CLOCKS_PER_SEC;
    end    = 0.0;
    start();
    cout << "begin the timer: ";
}

stopwatch::~~stopwatch()
{
using namespace std;

    stop(); // set end
    cout << "\nThe Object lasted: ";
    cout << fixed << setprecision(2)
         << end - begin << " seconds \n";
}

// clock ticks divided by ticks per second
void stopwatch::start()
{
using namespace std;

    set = double(clock())/CLOCKS_PER_SEC;
}

void stopwatch::stop()
```

```
{
using namespace std;

    end = double(clock()/CLOCKS_PER_SEC);
}

void time_delay(unsigned short t);

int main()
{
using namespace std;

    stopwatch watch; // create object and initialize
    cout.flush(); // this flushes the buffer
    time_delay(5);
    return 0; // destructor called at return
}

//time delay function
void time_delay(unsigned short t)
{
using namespace std;

    time_t tStart, tEnd;
    time(&tStart);
    while(tStart + t > time(&tEnd)){};
}
```

Result:

Note: comment out the flush and both lines will display simultaneously at the end of the program.

begin the timer: < *immediate display then pause* >

begin the timer:

The Object lasted: 3.83 seconds

27.6.2.6 Standard basic_ostream manipulators

Description To provide an inline formatting mechanism.

27.6 Formatting And Manipulators

27.6.2 Output streams

basic_ostream::endl

Description To insert a newline and flush the output stream.

Prototype

```
template
    < class charT, class traits >
basic_ostream<charT, traits> & endl
    (basic_ostream<charT,traits>& os);
```

Remarks The manipulator `endl` takes no external arguments, but is placed in the stream. It inserts a newline character into the stream and flushes the output.

Returns A reference to `basic_ostream`. (The `this` pointer.)

See Also `ostream::operators`

basic_ostream::ends

Description To insert a NULL character.

Prototype

```
template
    < class charT, class traits >
basic_ostream<charT, traits> &ends
    (basic_ostream<charT,traits>& os);
```

Remarks The manipulator `ends`, takes no external arguments, but is placed in the stream. It inserts a NULL character into the stream, usually to terminate a string.

Returns A reference to `ostream`. (The `this` pointer)

NOTE: The `ostringstream` provides in-core character streams but must be null terminated by the user. The manipulator `ends` provides a null terminator.

Listing 18.25 Example of `basic_ostream::ends` usage:

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

int main()
{
    using namespace std;

    ostringstream out; // see note above
    out << "Ask the teacher anything\n";
    out << "OK, what is 2 + 2?\n";
    out << 2 << " plus " << 2 << " equals "
        << 4 << ends;

    cout << out.str();
    return 0;
}
```

Result:

```
Ask the teacher anything
OK, what is 2 + 2?
2 plus 2 equals 4?
```

`basic_ostream::flush`

Description To flush the stream for output.

Prototype `template<class charT, class traits>`
`basic_ostream<charT, traits> &`
`flush(basic_ostream<charT, traits> (os);`

Remarks The manipulator `flush`, takes no external arguments, but is placed in the stream. The manipulator `flush` will attempt to release an output buffer for immediate use without waiting for an external input.

Returns A reference to ostream. (The `this` pointer.)

27.6 Formatting And Manipulators

27.6.2 Output streams

See Also `ostream::flush()`

Listing 18.26 Example of `basic_ostream::flush` usage:

```
#include <iostream>
#include <iomanip>
#include <ctime>

class stopwatch {
private:
    double begin, set, end;
public:
    stopwatch();
    ~stopwatch();
    void start();
    void stop();
};

stopwatch::stopwatch()
{
    using namespace std;

    begin = (double) clock() / CLOCKS_PER_SEC;
    end    = 0.0;
    start();
    {
        begin = (double) clock() / CLOCKS_PER_SEC;
        end    = 0.0;
        start();
        cout << "begin time the timer: " << flush;
    }
}

stopwatch::~~stopwatch()
{
    using namespace std;

    stop(); // set end
    cout << "\nThe Object lasted: ";
    cout << fixed << setprecision(2)
```

```
        << end - begin << " seconds \n";
    }

    // clock ticks divided by ticks per second
    void stopwatch::start()
    {
        using namespace std;

        set = double(clock()/CLOCKS_PER_SEC);
    }

    void stopwatch::stop()
    {
        using namespace std;

        end = double(clock()/CLOCKS_PER_SEC);
    }

    void time_delay(unsigned short t);

    int main()
    {
        using namespace std;

        stopwatch watch; // create object and initialize
        time_delay(5);
        return 0; // destructor called at return
    }

    //time delay function
    void time_delay(unsigned short t)
    {
        using namespace std;

        time_t tStart, tEnd;
        time(&tStart);
        while(tStart + t > time(&tEnd)){};
    }
```

27.6 Formatting And Manipulators

27.6.3 Standard manipulators

Results:

Note: comment out the flush and both lines display simultaneously at the end of the program.

begin time the timer:

< short pause >

The Object lasted: 3.78 seconds

27.6.3 Standard manipulators

The include file `iomanip` defines a template classes and related functions for input and output manipulation.

Standard Manipulator Instantiations

Description To create a specific use instance of a template by replacing the parameterized elements with pre-defined types.

resetiosflags

Description To unset previously set formatting flags.

Prototypes `smanip resetiosflags(ios_base::fmtflags mask)`

Remarks Use the manipulator `resetiosflags` directly in a stream to reset any format flags to a previous condition. You would normally store the return value of `setf()` in order to achieve this task.

Returns A `smanip` type, that is an implementation defined type.

See Also `ios_base::setf()`, `ios_base::unsetf()`

Listing 18.27 Example of resetiosflags() usage:

```
#include <iostream>
#include <iomanip>
```

```
int main()
{
    using namespace std;

    double d = 2933.51;
    long flags;
    flags = ios::scientific | ios::showpos | ios::showpoint;

    cout << "Original: " << d << endl;
    cout << "Flags set: " << setiosflags(flags)
        << d << endl;
    cout << "Flags reset to original: "
        << resetiosflags(flags) << d << endl;

    return 0;
}
```

```
Result:
Original:  2933.51
Flags set:  +2.933510e+03
Flags reset to original:  2933.51
```

setiosflags

Description	Set the stream format flags.
Prototypes	<code>smanip setiosflags(ios_base::fmtflags mask)</code>
Remarks	Use the manipulator <code>setiosflags()</code> to set the input and output formatting flags directly in the stream.
Returns	A <code>smanip</code> type, that is an implementation defined type.
See Also	<code>ios_base::setf()</code> , <code>ios_base::unsetf()</code>

Listing 18.28 Example of `setiosflags()` usage:

```
See resetiosflags()
```

27.6 Formatting And Manipulators

27.6.3 Standard manipulators

:setbase

Description To set the numeric base of an output.

Prototypes `smanip setbase(int)`

Remarks The manipulator `setbase()` directly sets the numeric base of integral output to the stream. The arguments are in the form of 8, 10, 16, or 0. 8 octal, 10 decimal and 16 hexadecimal. Zero represents `ios::basefield`, a combination of all three.

Returns A `smanip` type, that is an implementation defined type.

See Also `ios_base::setf()`

Listing 18.29 Example of <omanip>::setbase usage:

```
#include <iostream>
#include <iomanip>

int main()
{
    using namespace std;

    cout << "Hexadecimal "
         << setbase(16) << 196 << '\n';
    cout << "Decimal " << setbase(10) << 196 << '\n';
    cout << "Octal " << setbase(8) << 196 << '\n';

    cout.setf(ios::hex, ios::oct | ios::hex);
    cout << "Reset to Hex " << 196 << '\n';
    cout << "Reset basefield setting "
         << setbase(0) << 196 << endl;

    return 0;
}
```

Result:

Hexadecimal c4

Decimal 196

Octal 304
Reset to Hex c4
Reset basefield setting 196

setfill

Description	To specify the characters to used to insert in unused spaces in the output.
Prototypes	<code>smanip setfill(int c)</code>
Remarks	Use the manipulator <code>setfill()</code> directly in the output to fill blank spaces with character <code>c</code> .
Returns	A <code>smanip</code> type, that is an implementation defined type.
See Also	<code>basic_ios::fill</code>

Listing 18.30 Example of `basic_ios::setfill()` usage:

```
#include <iostream>
#include <iomanip>

int main()
{
    using namespace std;

    cout.width(8);
    cout << setfill('*') << "Hi!" << "\n";
    char fill = cout.fill();
    cout << "The filler is a " << fill << endl;

    return 0;
}
```

Result:
Hi!*****
The filler is a *

27.6 Formatting And Manipulators

27.6.3 Standard manipulators

setprecision

Description Set and return the current format precision.

Prototypes `smanip<int> setprecision(int)`

Remarks Use the manipulator `setprecision()` directly in the output stream with floating point numbers to limit the number of digits. You may use `setprecision()` with scientific or non-scientific floating point numbers.

With the flag `ios::floatfield` set the number in `precision` refers to the total number of significant digits generated. If the settings are for either `ios::scientific` or `ios::fixed` then the precision refers to the number of digits after the decimal place.

NOTE: This means that `ios::scientific` will have one more significant digit than `ios::floatfield`, and `ios::fixed` will have a varying number of digits.

Returns A `smanip` type, that is an implementation defined type.

See Also `ios_base::setf()`, `ios_base::precision()`

Listing 18.31 Example of `<omanip>::setprecision()` usage:

```
#include <iostream>
#include <iomanip>

int main()
{
    using namespace std;

    cout << "Original: " << 321.123456 << endl;
    cout << "Precision set: " << setprecision(8)
        << 321.123456 << endl;
    return 0;
}
```

Result:
Original: 321.123
Precision set: 321.12346

setw

Description To set the width of the output field.

Prototypes `smanip<int> setw(int)`

Remarks Use the manipulator `setw()` directly in a stream to set the field size for output.

Returns A pointer to ostream

See Also `ios_base::width()`

Listing 18.32 Example of <omanip>::setw() usage:

```
#include <iostream>
#include <iomanip>

int main()
{
    using namespace std;

    cout << setw(8)
         << setfill('*')
         << "Hi!" << endl;
    return 0;
}
```

Result:
Hi!*****

27.6 Formatting And Manipulators

27.6.3 Standard manipulators

Overloaded Manipulator

Description To store a function pointer and object type for input.

Prototype Overloaded input manipulator for `int` type.

```
istream &imanic_name  
    (istream &stream, type param)  
{  
    // body of code  
    return stream;  
}
```

Overloaded output manipulator for `int` type.

```
ostream &omanip_name  
    (ostream &stream, type param)  
{  
    // body of code  
    return stream;  
}
```

For other input/output types

```
smanip<type> manip_name(type param)  
{  
    return smanip<type> (manip_name, param);  
}
```

Remarks Use an overloaded manipulator to provide special and unique input handling characteristics for your class.

Returns A pointer to stream object.

Listing 18.33 Example of overloaded manipulator usage:

```
#include <iostream>  
#include <iomanip>  
#include <cstring>  
#include <cstdlib>  
#include <cctype>
```

```
char buffer[80];
```

27.6 Formatting And Manipulators

27.6.3 Standard manipulators

```
char *Password = "Metrowerks";

char *StrUpr(char * str);

std::omanip<char *> verify(char *check);
std::istream &verify_implement(std::istream &stream, char *check);

int main()
{
    using namespace std;

    cin >> verify(StrUpr>Password));
    cout << "Log in was Completed ! \n";

    return 0;
}

std::omanip<char *> verify(char *check)
{
    return std::omanip <char *> (verify_implement, check);
}

std::istream &verify_implement(std::istream &stream, char *check)
{
    using namespace std;

    short attempts = 3;

    do {
        cout << "Enter password: ";
        stream >> buffer;

        StrUpr(buffer);
        if (! strcmp(check, buffer)) return stream;
        cout << "\a\a";
        attempts--;
    } while(attempts > 0);

    cout << "All Tries failed \n";
    exit(1);
}
```

27.6 Formatting And Manipulators

27.6.3 Standard manipulators

```
    return stream;
}

char *StrUpr(char * str)
{
    char *p = str;    // dupe string
    while(*p) *p++ = std::toupper(*p);
    return str;
}
```

Result:

Enter password: <codewarrior>

Enter password: <mw>

Enter password: <metrowerks>

Log in was Completed !



27.7 String Based Streams

This chapter discusses string-based streams in the standard C++ library.

Overview of String Based Stream Classes

There are four template classes and 6 various types defined in the header `<sstream>` that are used to associate stream buffers with objects of class `basic_string`.

The sections in this chapter are:

- [“Header `<sstream>`” on page 589](#)
- [“27.7.1 Template class `basic_stringbuf`.” on page 590](#)
- [“27.7.2 Template class `basic_istream`” on page 597](#)
- [“27.7.3 Class `basic_stringstream`” on page 607](#)

Header `<sstream>`

Overview The header `<sstream>` includes classes and typed that associate stream buffers with string objects for input and output manipulations.

Prototype

```
namespace std{
    template
        <class charT, class traits = char_traits<charT> >
        class basic_stringbuf;
    typedef basic_stringbuf<char>stringbuf;
```

27.7 String Based Streams

27.7.1 Template class *basic_stringbuf*.

```
typedef basic_strngbuf<wchar>wstringbuf;

template
    <class charT, class traits = char_traits<charT> >
class basic_istreamream;
typedef basic_istreamream<char> istreamstream;
typedef basic_istreamream<wchar> wistreamstream;

template
    <class charT, class traits = char_traits<charT> >
class basic_ostreamstream;
typedef basic_ostreamstream<char> ostreamstream;
typedef basic_ostreamstream<wchar> wostreamstream;
};
}
```

Remarks The class `basic_string` is discussed in previous chapters.

27.7.1 Template class *basic_stringbuf*.

Overview The template class `basic_stringbuf` is derived from `basic_streambuf` is use to associate both input and output streams with an object of class `basic_string`.

The other topics in this section are:

- [“27.7.1.1 basic_stringbuf constructors” on page 591](#)
- [“27.7.1.2 Member functions” on page 593](#)
- [“27.7.1.3 Overridden virtual functions” on page 594](#)

Prototype

```
template
    <class charT, class traits = char_traits<charT> >
class basic_stringbuf: public basic_streambuf<charT, traits> > {
public:

    typedef charT char_type;
    typedef typename traits::int_type int_type;
```

```
typedef typename traits::pos_type pos_type;
typedef typename traits::off_type off_type;

explicit basic_stringbuf
    (ios_base::openmode which = ios_base::in | ios_base::out);
explicit basic_stringbuf
    (const basic_string<char_type> &str,
     ios_base::openmode which = ios_base::in | ios_base::out);

basic_string<char_type> str() const;
void str(const basic_string<char_type>&s);

protected
virtual int_type underflow();
virtual int_type pbackfail
    (int_type c = traits::eof());
virtual int_type overflow
    (int_type c = traits::eof());

virtual pos_type seekoff
    (off_type off,
     ios_base::seekdir way,
     ios_base::openmode which = ios_base::in | ios_base::out);
virtual pos_type seekpos
    (pos_type sp,
     ios_base::openmode which = ios_base::in | ios_base::out);

private:
ios_base::openmode mode; exposition only
};
}
```

Remarks The class `basic_stringbuf` is derived from `basic_streambuf` to associate a stream with a `basic_string` object for in-core memory character manipulations.

27.7.1.1 `basic_stringbuf` constructors

The `basic_stringbuf` has two constructors:

27.7 String Based Streams

27.7.1 Template class *basic_stringbuf*.

- `explicit basic_stringbuf(ios_base::openmode);`
- `explicit basic_stringbuf(const basic_string, ios_base::openmode);`

Constructor

Description To create a string buffer for characters for input/output.

Prototype

```
explicit basic_stringbuf
    (ios_base::openmode which =
     ios_base::in | ios_base::out);
explicit basic_stringbuf
    (const basic_string <char_type> &str,
     ios_base::openmode which =
     ios_base::in | ios_base::out);
```

Remarks The `basic_stringbuf` constructor is used to create an object usually as an intermediate storage object for input and output. The overloaded constructor is used to determine the input or output attributes of the `basic_string` object when it is created.

No array object is allocated.

Listing 19.1 Example of `basic_stringbuf::basic_stringbuf()` usage:

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

const int size = 100;

int main()
{
    using namespace std;

    stringbuf strbuf;
    strbuf.pubsetbuf('\0', size);
    strbuf.sputn("ABCDE",50);

    char ch;
    // look ahead at the next character
```



```
ch =strbuf.snextc();
cout << ch;
    // get pointer was not returned after peeking
ch = strbuf.snextc();
cout << ch;

return 0;
}
```

Result:
BC

27.7.1.2 Member functions

The class `basic_stringbuf` has one member functions:

- `str()`

`basic_stringbuf::str`

Description To return the `basic_string` object stored in the buffer.

Prototype `basic_string<char_type> str() const;`

Remarks The function `str()` freezes the buffer then returns a `basic_string` object.

`void str(const basic_string<char_type>&s);`

Remarks The function `str()` assigns the value of the `basic_string` object to the argument `'s'` if successful.

Returns If successful a `basic_string` object.

Listing 19.2 Example of `basic_stringbuf::str()` usage:

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

char CW[] = "Metrowerks CodeWarrior";
char AW[] = " - \"Software at Work\"";
```

27.7 String Based Streams

27.7.1 Template class *basic_stringbuf*.

```
int main()
{
    using namespace std;

    string buf;
    stringbuf strbuf(buf, ios::out);

    int size;
    size = strlen(CW);
    strbuf.sputn(CW, size);
    size = strlen(AW);
    strbuf.sputn(AW, size);

    cout << strbuf.str();

    return 0;
}
```

Result

Metrowerks CodeWarrior - "Software at Work"

27.7.1.3 Overridden virtual functions

The base class `basic_streambuf` has several virtual functions that are to be overloaded by derived classes. The are:

- `underflow()`
- `pbackfail()`
- `overflow()`
- `seekoff()`
- `seekpos()`

`basic_stringbuf::underflow`

Description To show an underflow condition and not increment the get pointer.

Prototype `virtual int_type underflow();`

Remarks The function `underflow` overrides the `basic_streambuf` virtual function.

Returns The first character of the pending sequence and does not increment the get pointer. If the position is `null` returns `traits::eof()` to indicate failure.

See Also `basic_streambuf::underflow()`

`basic_stringbuf::pbackfail`

Description To show a failure in a put back operation.

Prototype

```
virtual int_type pbackfail
(int_type c = traits::eof());
```

Remarks The function `pbackfail` overrides the `basic_streambuf` virtual function.

Returns The function `pbackfail()` is only called when a put back operation really has failed and returns `traits::eof`. If success occurs the return is undefined.

See Also `basic_streambuf::pbackfail()`

`basic_stringbuf::overflow`

Description Consumes the pending characters of an output sequence.

Prototype

```
virtual int_type overflow
(int_type c = traits::eof());
```

Remarks The function `overflow` overrides the `basic_streambuf` virtual function.

Returns The function returns `traits::eof()` for failure or some unspecified result to indicate success.

27.7 String Based Streams

27.7.1 Template class *basic_stringbuf*.

See Also `basic_streambuf::overflow()`

`basic_stringbuf::seekoff`

Description To return an offset of the current pointer in an input or output streams.

Prototype

```
virtual pos_type seekoff
    (off_type off,
     ios_base::seekdir way,
     ios_base::openmode which =
     ios_base::in | ios_base::out);
```

Remarks The function `seekoff` overrides the `basic_streambuf` virtual function.

Returns A `pos_type` value, which is an invalid stream position.

See Also `basic_streambuf::seekoff()`

`basic_stringbuf::seekpos`

Description To alter an input or output stream position.

Prototype

```
virtual pos_type seekpos
    (pos_type sp,
     ios_base::openmode which =
     ios_base::in | ios_base::out);
```

Remarks The function `seekoff` overrides the `basic_streambuf` virtual function.

Returns A `pos_type` value, which is an invalid stream position.

See Also `basic_streambuf::seekoff()`

27.7.2 Template class `basic_istream`

Overview The template class `basic_istream` is derived from `basic_istream` and is used to associate input streams with an object of class `basic_string`.

The prototype is listed below. The other topics in this section are:

- [“27.7.2.1 `basic_istream` constructors” on page 598](#)
- [“27.7.2.2 Member functions” on page 599](#)

Prototype

```
namespace std {
    template
        <class charT, class traits = char_traits<charT> >
        class basic_istream : public basic_istream<charT, traits>
        {
        public:
            typedef charT char_type;
            typedef typename traits::int_type int_type;
            typedef typename traits::pos_type pos_type;
            typedef typename traits::off_type off_type;

            explicit basic_istream
                (ios_base::openmode which = ios_base::in);
            explicit basic_istream
                (const basic_string<charT> &str,
                 ios_base::openmode which = ios_base::in);

            basic_stringbuf<charT, traits>* rdbuf() const;

            basic_string<charT> str() const;
            void str(const basic_string<charT> &s);

        private:
            basic_stringbuf<charT, traits> sb; exposition only
        };
    }
```

27.7 String Based Streams

27.7.2 Template class *basic_istream*

Remarks The class `basic_istream` uses an object of type `basic_stringbuf` to control the associated storage.

See Also `basic_ostringstream`, `basic_string`,
`basic_stringstream`, `basic_filebuf`.

27.7.2.1 `basic_istream` constructors

The class `basic_istream` has two constructors.

- `basic_istream`
 (`ios_base::openmode`)
- `basic_istream`
 (`const basic_string`, `ios_base::openmode`)

Constructor

Description The `basic_istream` constructors create a `basic_stringstream` object and initialize the `basic_streambuf` object.

Prototype

```
explicit basic_istream  
    (ios_base::openmode which = ios_base::in);  
explicit basic_istream  
    (const basic_string<charT> &str,  
     ios_base::openmode which = ios_base::in);
```

Remarks The `basic_istream` constructor is overloaded to accept a
an object of class `basic_string` for input.

See Also `basic_ostringstream`, `basic_stringstream`

Listing 19.3 Example of `basic_istream::basic_istream()` usage

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

```
int main()
```

```
{
using namespace std;

    string sBuffer = "3 12.3 line";
    int num = 0;
    double flt = 0;
    char szArr[20] = "\0";

    istream Paragraph(sBuffer, ios::in);
    Paragraph >> num;
    Paragraph >> flt;
    Paragraph >> szArr;

    cout << num << " " << flt << " "
         << szArr << endl;

    return 0;
}
```

Result
3 12.3 line

27.7.2.2 Member functions

The class `basic_istream` has two member functions

- `rdbuf()`
- `str()`

`basic_istream::rdbuf`

Description To retrieve a pointer to the stream buffer.

Prototype `basic_stringbuf<charT, traits>* rdbuf() const;`

Remarks To manipulate a stream for random access or synchronization it is necessary to retrieve a pointer to the streams buffer. The function `rdbuf()` allows you to retrieve this pointer.

27.7 String Based Streams

27.7.2 Template class *basic_istream*

Returns A pointer to an object of type `basic_stringbuf` **sb** is returned by the `rdbuf` function.

See Also `basic_ostringstream::rdbuf()` `basic_ios::rdbuf()`
`basic_stringstream::rdbuf()`

Listing 19.4 Example of `basic_istream::rdbuf()` usage.

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

std::string buf = "Metrowerks CodeWarrior - \"Software at work\"";
char words[50];

int main()
{
    using namespace std;

    istringstream ist(buf);
    istream in(ist.rdbuf());
    in.seekg(25);

    in.get(words, 50);
    cout << words;

    return 0
}
```

Result
"Software at work"

basic_istream::str

Description To return the `basic_string` object stored in the buffer.

Prototype `basic_string<charT> str() const;`
`void str(const basic_string<charT> &s);`

Remarks	The function <code>str()</code> freezes the buffer then returns a <code>basic_string</code> object.
Returns	If successful a <code>basic_string</code> object.
See Also	<code>basic_streambuf::str()</code> , <code>basic_ostringstream.str()</code> <code>basic_stringstream::str()</code>

Listing 19.5 Example of `basic_istream::str()` usage.

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

std::string buf = "Metrowerks CodeWarrior - \"Software at Work\"";

int main()
{
    using namespace std;

    istringstream istr(buf);
    cout << istr.str();
    return 0;
}
```

Result:
Metrowerks CodeWarrior - "Software at Work"

27.7.2.3 Class `basic_ostringstream`

Overview The template class `basic_ostringstream` is derived from `basic_ostream` is use to associate output streams with an object of class `basic_string`.

The prototype is listed below. The other topics in this section are:

- 27.7.2.4 `basic_ostringstream` constructors
- 27.7.2.5 Member functions

27.7 String Based Streams

27.7.2.3 Class *basic_ostringstream*

Prototype

```
namespace std {
    template
    <class charT, class traits = char_traits<charT> >
    class basic_ostringstream : public basic_ostream<charT, traits>{

    public:
        typedef charT char_type;
        typedef typename traits::int_type int_type;
        typedef typename traits::pos_type pos_type;
        typedef typename traits::off_type off_type;

        explicit basic_ostringstream
            (ios_base::openmode which = ios_base::out);
        explicit basic_ostringstream
            (const basic_string<charT> &str,
             ios_base::openmode which = ios_base::out);

        basic_stringbuf<charT, traits>* rdbuf() const;

        basic_strng<charT> str() const;
        void str(const basic_string<charT> &s);

    private:
        basic_stringbuf<charT,traits> sb; exposition only
    };
}
```

Remarks The class `basic_ostringstream` uses an object of type `basic_stringbuf` to control the associated storage.

See Also `basic_istringstream`, `basic_string`, `basic_stringstream`, `basic_filebuf`.

27.7.2.4 `basic_ostringstream` constructors.

The class `basic_ostringstream` has two constructors

- `basic_ostringstream`
(`ios_base::openmode`)
- `basic_ostringstream`
(`const basic_string`, `ios_base::openmode`)

Constructor

Description The `basic_ostringstream` constructors create a `basic_stringstream` object and initialize the `basic_streambuf` object.

Prototype `explicit basic_ostringstream`
(`ios_base::openmode` `which` = `ios_base::out`);
`explicit basic_ostringstream`
(`const basic_string<charT>` &`str`,
`ios_base::openmode` `which` = `ios_base::out`);

Remarks The `basic_stringstream` constructor is overloaded to accept an object of class `basic_string` for output.

See Also `basic_istringstream`, `basic_stringstream`

Listing 19.6 Example of `basic_ostringstream::basic_ostringstream()` usage

The file MW Reference contains
Metrowerks CodeWarrior - "Software at Work"
Registered Trademark

```
#include <iostream>
#include <fstream>
#include <sstream>
#include <cstdlib>

int main()
{
    using namespace std;

    ifstream in("MW Reference");
    if(!in.is_open())
        {cout << "can't open file for input"; exit(1);}
}
```

27.7 String Based Streams

27.7.2.3 Class *basic_ostringstream*

```
ostringstream Paragraph;
char ch = '\0';

while((ch = in.get()) != EOF)
{
    Paragraph << ch;
}

cout << Paragraph.str();

in.close();
return 0;
}
```

Result:

Metrowerks CodeWarrior - "Software at Work"
Registered Trademark

27.7.2.5 Member functions

The class `basic_ostringstream` has two member functions:

- `rdbuf()`
- `str()`

`basic_ostringstream::rdbuf`

Description	To retrieve a pointer to the stream buffer.
Prototype	<code>basic_stringbuf<charT, traits>* rdbuf() const;</code>
Remarks	To manipulate a stream for random access or synchronization it is necessary to retrieve a pointer to the streams buffer. The function <code>rdbuf()</code> allows you to retrieve this pointer.
Returns	A pointer to an object of type <code>basic_stringbuf</code> sb is returned by the <code>rdbuf</code> function.

See Also `basic_ostringstream::rdbuf()` `basic_ios::rdbuf()`
 `basic_stringstream::rdbuf()`

Listing 19.7 example of `basic_ostringstream::rdbuf()` usage

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

std::string motto = "Metrowerks CodeWarrior - \"Software at
Work\"";

int main()
{
    using namespace std;

    ostringstream ostr(motto);
    streampos cur_pos(0), start_pos(0);

    cout << "The original array was :\n"
         << motto << "\n\n";
    // associate buffer
    stringbuf *strbuf(ostr.rdbuf());

    streamoff str_off = 10;
    cur_pos = ostr.tellp();
    cout << "The current position is "
         << static_cast<streamoff>(cur_pos);
         << " from the beginning\n";

    ostr.seekp(str_off);

    cur_pos = ostr.tellp();
    cout << "The current position is "
         << static_cast<streamoff>(cur_pos);
         << " from the beginning\n";

    strbuf->sputc('\0');

    cout << "The stringbuf array is\n"
```

27.7 String Based Streams

27.7.2.3 Class *basic_ostringstream*

```
<< strbuf->str() << "\n\n";
cout << "The ostringstream array is still\n"
<< motto;

return 0;
}
```

Results:

The original array was :

Metrowerks CodeWarrior - "Software at Work"

The current position is 0 from the beginning

The current position is 10 from the beginning

The stringbuf array is

Metrowerks

Metrowerks CodeWarrior - "Software at Work"

basic_ostringstream::str

Description	To return the <code>basic_string</code> object stored in the buffer.
Prototype	<pre>basic_string<charT> str() const; void str(const basic_string<charT> &s);</pre>
Remarks	The function <code>str()</code> freezes the buffer then returns a <code>basic_string</code> object.
Returns	If successful a <code>basic_string</code> object.
See Also	<code>basic_streambuf::str()</code> , <code>basic_istringstream.str()</code> <code>basic_stringstream::str()</code>

Listing 19.8 Example of `basic_ostringstream::str()` usage.

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

int main()
```

```
{
using namespace std;

    ostream out;
    out << "Ask the teacher anything\n";
    out << "OK, what is 2 + 2?\n";
    out << 2 << " plus " << 2 << " equals "
        << 4 << ends;

    cout << out.str();
    return 0;
}
```

Result:

Ask the teacher anything

OK, what is 2 + 2?

2 plus 2 equals 4?

27.7.3 Class basic_stringstream

Overview The template class `basic_stringstream` is derived from `basic_istream` is use to associate input and output streams with an object of class `basic_string`.

The prototype is listed below. The other topics in this section are:

- 27.7.3.4 `basic_stringstream` constructors
- 27.7.3.5 Member functions

Prototype

```
namespace std {
    template
        <class charT, class traits = char_traits<charT> >
    class basic_stringstream : public basic_istream<charT, traits>
    {
    public:
        typedef charT char_type;
        typedef typename traits::int_type int_type;
    }
```

27.7 String Based Streams

27.7.3 Class *basic_stringstream*

```
typedef typename traits::pos_type pos_type;
typedef typename traits::off_type off_type;

explicit basic_stringstream
(ios_base::openmode which = ios_base::out | ios_base::out);
explicit basic_stringstream
(const basic_string<charT> &str,
 ios_base::openmode which = ios_base::out | ios_base::out);

basic_stringbuf<charT, traits>* rdbuf() const;

basic_string<charT> str() const;
void str(const basic_string<charT> &s);

private:
basic_stringbuf<charT, traits> sb; exposition only
};
}
```

Remarks The class `basic_stringstream` uses an object of type `basic_stringbuf` to control the associated storage.

See Also `basic_istringstream`, `basic_string`,
`basic_stringstream`, `basic_filebuf`

27.7.3.4 `basic_stringstream` constructors

The class `basic_stringstream` has two constructors:

- `explicit basic_stringstream (ios_base::openmode)`
- `explicit basic_stringstream (const basic_string, ios_base::openmode)`

Constructor

Description The `basic_stringstream` constructors create a `basic_stringstream` object and initialize the `basic_streambuf` object.

Prototype `explicit basic_stringstream
 (ios_base::openmode which =
 ios_base::out | ios_base::out);`
`explicit basic_stringstream
 (const basic_string<charT> &str,
 ios_base::openmode which =
 ios_base::out | ios_base::out);`

Remarks The `basic_stringstream` constructor is overloaded to accept a an object of class `basic_string` for input or output.

See Also `basic_ostringstream`, `basic_istringstream`

Listing 19.9 Example of `basic_stringstream::basic_stringstream()` usage

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

char buf[50] = "ABCD 22 33.33";
char words[50];

int main()
{
    using namespace std;

    stringstream iost;

    char word[20];
    long num;
    double real;

    iost << buf;
    iost >> word;
    iost >> num;
    iost >> real;

    cout << word << " "
         << num << " "
         << real << endl;
```

27.7 String Based Streams

27.7.3 Class *basic_stringstream*

```
    return 0;
}
```

Result

ABCD 22 33.33

27.7.3.5 Member functions

The class `basic_stringstream` has two member functions:

- `rdbuf()`
- `str()`

`basic_stringstream::rdbuf`

Description	To retrieve a pointer to the stream buffer.
Prototype	<code>basic_stringbuf<charT, traits>* rdbuf() const;</code>
Remarks	To manipulate a stream for random access or synchronization it is necessary to retrieve a pointer to the streams buffer. The function <code>rdbuf()</code> allows you to retrieve this pointer.
Returns	A pointer to an object of type <code>basic_stringbuf</code> sb is returned by the <code>rdbuf</code> function.
See Also	<code>basic_ostringstream::rdbuf()</code> <code>basic_ios::rdbuf()</code> <code>basic_stringstream::rdbuf()</code>

Listing 19.10 Example of `basic_stringstream::rdbuf()` usage

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

std::string buf = "Metrowerks CodeWarrior - \"Software at Work\"";
char words[50];

int main()
```

```
{
using namespace std;

    stringstream ist(buf, ios::in);
    istream in(ist.rdbuf());
    in.seekg(25);

    in.get(words,50);
    cout << words;

    return 0;
}
```

Result
"Software at Work"

`basic_stringstream::str`

- | | |
|--------------------|-----------------------------------------------------------------------------------------------------------------------------|
| Description | To return the <code>basic_string</code> object stored in the buffer. |
| Prototype | <code>basic_strng<charT> str() const;</code>
<code>void str(const basic_string<charT> &s);</code> |
| Remarks | The function <code>str()</code> freezes the buffer then returns a <code>basic_string</code> object. |
| Returns | If successful a <code>basic_string</code> object. |
| See Also | <code>basic_streambuf::str()</code> , <code>basic_ostringstream.str()</code>

<code>basic_istringstream::str()</code> |

Listing 19.11 Example of `basic_stringstream::str()` usage

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

std::string buf = "Metrowerks CodeWarrior - \"Software at Work\"";
char words[50];
```

27.7 String Based Streams

27.7.3 Class *basic_stringstream*

```
int main()
{
using namespace std;

    stringstream iost(buf, ios::in);

    cout << iost.str();

    return 0;
}
```

Result

Metrowerks CodeWarrior - "Software at Work"



27.8 File Based Streams

Association of stream buffers with files for file reading and writing.

Overview of File Based Streams

The sections in this chapter are:

- [“Header <fstream>” on page 613](#)
- [“27.8.1 File streams” on page 613](#)
- [“27.8.1.1 Template class basic_filebuf” on page 614](#)
- [“27.8.1.5 Template class basic_ifstream” on page 622](#)
- [“27.8.1.8 Template class basic_ofstream” on page 629](#)
- [“27.8.1.11 Template class basic_fstream” on page 636](#)

Header <fstream>

Description The header <fstream> defines template classes and types to assist in reading and writing of files.

27.8.1 File streams

Prototype

```
namespace std{
template
    <class charT, class traits = ios_traits<charT> >
class basic_filebuf;
typedef basic_filebuf<char> filebuf;
```

27.8 File Based Streams

27.8.1.1 Template class *basic_filebuf*

```
typedef basic_filebuf<wchar_t> wfilebuf;

template
<class charT, class traits = ios_traits<charT> >
class basic_ifstream;
typedef basic_ifstream<char> ifstream;
typedef basic_ifstream<wchar_t> wifstream;

template
<class charT, class traits = ios_traits<charT> >
class basic_ofstream;
typedef basic_ofstream<char> ofstream;
typedef basic_ofstream<wchar_t> wofstream;
}
```

Remarks A FILE refers to the type FILE as defined in the Standard C Library and provides an external input or output stream with the underlying type of char or byte. A stream is a sequence of char or bytes.

27.8.1.1 Template class *basic_filebuf*

Description A class to provide for input and output file stream buffering mechanisms.

The prototype is listed below. Other topics in this section are:

- [“27.8.1.2 basic_filebuf Constructors” on page 616](#)
- [“27.8.1.3 Member functions” on page 616](#)
- [“27.8.1.4 Overridden virtual functions” on page 619](#)

Prototype

```
namespace std{
    template
    <class charT, class traits = ios_traits<charT> >
    class basic_filebuf : public basic_streambuf <charT, traits>
    {
    public:
```

```
typedef charT char_type;
typedef typename traits::int_type int_type;
typedef typename traits::pos_type pos_type;
typedef typename traits::off_type off_type;

basic_filebuf();
virtual ~basic_filebuf();

bool is_open() const;
basic_filebuf<charT, traits>* open
    (const char* c, ios_base::openmode mode);
basic_filebuf<charT, traits>* close();

protected:

virtual int showmanyc();

virtual int_type underflow();
virtual int_type pbackfail
    (int_type c = traits::eof());
virtual int_type overflow
    (int_type c = traits::eof());

virtual basic_streambuf<charT traits>* setbuf
    (char_type* s, streamsize n);

virtual pos_type seekoff
    (off_type off,
     ios_base::seekdir way,
     ios_base::in | ios_base::out);
virtual pos_type seekpos
    (pos_type sp,
     ios_base::openmode which,
     ios_base::in | ios_base::out);

virtual int sync();
virtual void imbue(const locale& loc);
```

27.8 File Based Streams

27.8.1.1 Template class *basic_filebuf*

```
};  
}
```

Remarks The `filebuf` class is derived from the `streambuf` class and provides a buffer for file output and or input.

27.8.1.2 `basic_filebuf` Constructors

Default Constructor

Description To construct and initialize a `filebuf` object.

Prototype `basic_filebuf()`

Remarks The constructor opens a `basic_filebuf` object and initializes it with `basic_streambuf<charT, traits>()` and if successful `is_open()` is `false`.

Destructor

Description To remove the `basic_filebuf` object from memory.

Prototype `virtual ~basic_filebuf();`

Listing 20.1 For example of `basic_filebuf::basic_filebuf()` usage:

See `basic_filebuf::open()`.

27.8.1.3 Member functions

`basic_filebuf::is_open`

Description Test to ensure `filebuf` stream is open for reading or writing.

Prototype `bool is_open() const`

Remarks Use the function `is_open()` for a `filebuf` stream to ensure it is open before attempting to do any input or output operation on the stream.

Returns True if stream is available and open.

Listing 20.2 For example of `basic_filebuf::is_open()` usage

See: `basic_filebuf::basic_filebuf`

`basic_filebuf::open`

Description Open a `basic_filebuf` object and associate it with a file.

Prototype `basic_filebuf<charT, traits>* open
(const char* c,
ios_base::openmode mode);`

Remarks You would use the function `open()` to open a `filebuf` object and associate it with a file. You may use `open()` to reopen a buffer and associate it if the object was closed but not destroyed.

WARNING! If an attempt is made to open a file in an inappropriate file opening mode, the file will not open and a test for the object will not give false, therefore use the function `is_open()` to check for file openings.

Table 20.1 Legal `basic_filebuf` file opening modes

Opening Modes	stdio equivalent
Input Only	
<code>ios::in</code>	"r"
<code>ios::binary ios::in</code>	"rb"
Output only	

27.8 File Based Streams

27.8.1.1 Template class *basic_filebuf*

Opening Modes	stdio equivalent
ios::out	"w"
ios::binary ios::out	"wb"
ios::out ios::trunc	"w"
ios::binary ios::out ios::trunc	"wb"
ios::out ios::app	"a"
Input and Output	
ios::in ios::out	"r+"
ios::binary ios::in ios::out	"r+b"
ios::in ios::out ios::trunc	"w+"
ios::binary ios::in ios::out ios::trunc	"w+b"
ios::binary ios::out ios::app	"ab"

Returns If successful the this pointer is returned, if `is_open()` equals true then a null pointer is returned.

Listing 20.3 Example of `filebuf::open()` usage:

The file MW Reference before operation contained:
Metrowerks CodeWarrior "Software at Work"

```
#include <fstream>
#include <cstdlib>

char inFile[] = "MW Reference";

int main(){
using namespace std;

    filebuf in;
    in.open(inFile, ios::out | ios::app);
    if(!in.is_open())
        {cout << "could not open file"; exit(1);}
    char str[] = "\n\tregistered trademark";
```

```
in.sputn(str, strlen(str));  
  
in.close();  
return 0;  
}
```

Result:

The file MW Reference now contains:
Metrowerks CodeWarrior "Software at Work"
registered trademark

basic_filebuf::close

Description To close a filebuf stream without destroying it.

Prototype `basic_filebuf<charT, traits>* close();`

Remarks The function `close()` would remove the stream from memory but will not remove the filebuf object. You may re-open a filebuf stream that was closed using the `close()` function.

Returns The `this` pointer with success otherwise a null pointer.

Listing 20.4 For example of `basic_filebuf::close()` usage

See `basic_filebuf::open()`

27.8.1.4 Overridden virtual functions

basic_filebuf::showmanyc

Description Overrides `basic_streambuf::showmanyc()`.

Prototype `virtual int showmanyc();`

Remarks Behaves the same as `basic_sreambuf::showmanyc()`.

27.8 File Based Streams

27.8.1.1 Template class *basic_filebuf*

basic_filebuf::underflow

- Description** Overrides `basic_streambuf::underflow()`;
- Prototype** `virtual int_type underflow();`
- Remarks** Behaves the same as `basic_streambuf::underflow` with the specialization that a sequence of characters is read as if they were read from a file into an internal buffer.

basic_filebuf::pbackfail

- Description** Overrides `basic_streambuf::pbackfail()`.
- Prototype** `virtual int_type pbackfail
(int_type c = traits::eof());`
- Remarks** This function puts back the characters designated by `c` to the input sequence if possible.
- Returns** `traits::eof()` if failure and returns either the character put back or `traits::not_eof(c)` for success.

basic_filebuf::overflow

- Description** Overrides `basic_streambuf::overflow()`
- Prototype** `virtual int_type overflow
(int_type c = traits::eof());`
- Remarks** Behaves the same as `basic_streambuf<charT, traits>::overflow(c)` except the behavior of consuming characters is performed by conversion.
- Returns** `traits::eof()` with failure.

basic_filebuf::seekoff

Description	Overrides <code>basic_streambuf::seekoff()</code>
Prototype	<pre>virtual pos_type seekoff (off_type off, ios_base::seekdir way, ios_base::in ios_base::out);</pre>
Remarks	Sets the offset position of the stream as if using the C standard library function <code>fseek(file, off, whence)</code> .
Returns	Seekoff function returns a newly formed <code>pos_type</code> object which contains all information needed to determine the current position if successful. An invalid stream position if it fails.

basic_filebuf::seekpos

Description	Overrides <code>basic_streambuf::seekpos()</code>
Prototype	<pre>virtual pos_type seekpos (pos_type sp, ios_base::openmode which, ios_base::in ios_base::out);</pre>
Remarks	Description undefined in standard at the time of writing.
Returns	Seekpos function returns a newly formed <code>pos_type</code> object which contains all information needed to determine the current position if successful. An invalid stream position if it fails.

basic_filebuf::setbuf

Description	Overrides <code>basic_streambuf::setbuf()</code>
Prototype	<pre>virtual basic_streambuf<charT traits>* setbuf (char_type* s, streamsize n);</pre>

27.8 File Based Streams

27.8.1.5 Template class *basic_ifstream*

Remarks Description undefined in standard at the time of writing.

basic_filebuf::sync

Description Overrides basic_streambuf::sync

Prototype `virtual int sync();`

Remarks Description undefined in standard at the time of writing.

basic_filebuf::imbue

Description Overrides basic_streambuf::imbue

Prototype `virtual void imbue(const locale& loc);`

Remarks Description undefined in standard at the time of writing.

27.8.1.5 Template class **basic_ifstream**

A class to provide for input file stream mechanisms.

The prototype is listed below. Other topics in this section are:

- [“27.8.1.6 basic_ifstream Constructor” on page 623](#)
- [“27.8.1.7 Member functions” on page 625](#)

Prototype

```
namespace std{
    template
        <class charT, class traits = ios_traits<charT> > {
    class basic_ifstream : public basic_istream<charT, traits>
    {
    public:
        typedef charT char_type;
        typedef typename traits::int_type int_type;
        typedef typename traits::pos_type pos_type;
        typedef typename traits::off_type off_type;
```

```
basic_ifstream();
explicit basic_ifstream
    (const char *s, openmode mode = in);

basic_filebuf<charT, traits>* rdbuf() const;
bool is_open();
void open(const char* s, openmode mode = in);
void close();

private:
basic_filebuf<charT, traits> sb; exposition only
};
}
```

NOTE: If the `basic_ifstream` supports reading from file. It uses a `basic_filebuf` object to control the sequence. That object is represented here as `basic_filebuf sb`.

Remarks The `basic_ifstream` provides mechanisms specifically for input file streams.

27.8.1.6 `basic_ifstream` Constructor

Default Constructor and Overloaded Constructor

Description Create a file stream for input.

Prototype

```
basic_ifstream();
explicit basic_ifstream
    (const char *s, openmode mode = in);
```

Remarks The constructor creates a stream for file input; it is overloaded to either create and initialize when called or to simply create a class and be opened using the `open()` member function. The default opening mode is `ios::in`. See `basic_filebuf::open()` for valid open mode settings.

27.8 File Based Streams

27.8.1.5 Template class *basic_ifstream*

NOTE: See `basic_ifstream::open` for legal opening modes.

See also `basic_ifstream::open()` for overloaded form usage.

Listing 20.5 Example of `basic_ifstream::basic_ifstream()` constructor usage:

The MW Reference file contains:
Metrowerks CodeWarrior "Software at Work"

```
#include <iostream>
#include <fstream>
#include <cstdlib>

char inFile[] = "MW Reference";

int main()
{
    using namespace std;

    ifstream in(inFile, ios::in);
    if(!in.is_open())
        {cout << "can't open input file"; exit(1);}

    char c = '\0';
    while(in.good())
    {
        if(c) cout << c;
        in.get(c);
    }

    in.close();
    return 0;
}
```

Result:
Metrowerks CodeWarrior "Software at Work"

27.8.1.7 Member functions

basic_ifstream::rdbuf

Description	The <code>rdbuf()</code> function retrieves a pointer to a <code>filebuf</code> type buffer.
Prototype	<code>basic_filebuf<charT, traits>* rdbuf() const;</code>
Remarks	In order to manipulate for random access or use an <code>ifstream</code> stream for both input and output you need to manipulate the base buffer. The function <code>rdbuf()</code> returns a pointer to this buffer for manipulation.
Returns	A pointer to type <code>basic_filebuf</code> .

Listing 20.6 Example of `basic_ifstream::rdbuf()` usage:

The MW Reference file contains originally
Metrowerks CodeWarrior "Software at Work"

```
#include <iostream>
#include <fstream>
#include <cstdlib>

char inFile[] = "MW Reference";

int main()
{
    using namespace std;

    ifstream inOut(inFile, ios::in | ios::out);
    if(!inOut.is_open())
        {cout << "Could not open file"; exit(1);}

    ostream Out(inOut.rdbuf());

    char str[] = "\n\tRegistered Trademark";

    inOut.rdbuf()->pubseekoff(0, ios::end);
```

27.8 File Based Streams

27.8.1.5 Template class *basic_ifstream*

```
Out << str;

inOut.close();

return 0;
}
```

Result:

The File now reads:

Metrowerks CodeWarrior "Software at Work"
Registered Trademark

basic_ifstream::is_open

Description	Test for open stream.
Prototype	<code>bool is_open() const</code>
Remarks	Use <code>is_open()</code> to test that a stream is indeed open and ready for input from the file.
Returns	True if file is open.

Listing 20.7 For example of `basic_ifstream::is_open()` usage

See `basic_ifstream::basic_ifstream()`

basic_ifstream::open

Description	Open is used to open a file or reopen a file after closing it.
Prototype	<code>void open(const char* s, openmode mode = in);</code>
Remarks	The default open mode is <code>ios::in</code> , but can be one of several modes. (see below) A stream is opened and prepared for input or output as selected.

Returns No return

Table 20.2 17.4.1.1.4 Legal basic_ifstream file opening modes

Opening Modes	stdio equivalent
Input Only	
ios:: in	"r"
ios:: binary ios::in	"rb"
Input and Output	
ios::in ios::out	"r+"
ios::binary ios::in ios::out	"r+b"
ios:: in ios::out ios::trunc	"w+"
ios::binary ios::in ios::out ios::trunc	"w+b"
ios::binary ios:: out ios::app	"ab"

NOTE: If an attempt is made to open a file in an inappropriate file opening mode, the file will not open and a test for the object will not give false, therefore use the function `is_open()` to check for file openings

Listing 20.8 Example of basic_ifstream::open() usage:

The MW Reference file contains:
Metrowerks CodeWarrior "Software at Work"

```
#include <iostream>
#include <fstream>
#include <cstdlib>

char inFile[] = "MW Reference";

int main()
{
using namespace std;
```

27.8 File Based Streams

27.8.1.5 Template class *basic_ifstream*

```
ifstream in;
in.open(inFile);
if(!in.is_open())
    {cout << "can't open input file"; exit(1);}

char c = NULL;
while((c = in.get()) != EOF)
{
    cout << c;
}

in.close();
return 0;
}
```

Result:

Metrowerks CodeWarrior "Software at Work"

basic_ifstream::close

Description Closes the file stream.

Prototype void close();

Remarks The close() function closes the stream for operation but does not destroy the ifstream object so it may be re-opened at a later time. If the function fails calls setstate(failbit) which may throw an exception.

Returns: No return.

Listing 20.9 Example of basic_ifstream::close() usage:

See basic_ifstream::basic_ifstream()

27.8.1.8 Template class *basic_ofstream*

Description A class to provide for output file stream mechanisms.

The prototype is listed below. Other topics in this section are:

- [“27.8.1.9 *basic_ofstream* constructor” on page 630](#)
- [“27.8.1.10 Member functions” on page 631](#)

Prototype

```
namespace std{
    template
        <class charT, class traits = ios_traits<charT> >
        class basic_ofstream : public basic_ostream<charT, traits>
        {
        public:
            typedef charT char_type;
            typedef typename traits::int_type int_type;
            typedef typename traits::pos_type pos_type;
            typedef typename traits::off_type off_type;

            basic_ofstream();
            explicit basic_ofstream
                (const char *s, openmode mode = out | trunc);

            basic_filebuf<charT, traits>* rdbuf() const;
            bool is_open();
            void open(const char* s, openmode mode = out);
            void close();

        private:
            basic_filebuf<charT, traits> sb; exposition only
        };
}
```

27.8 File Based Streams

27.8.1.8 Template class *basic_ofstream*

NOTE: The `basic_ofstream` supports writing to file. It uses a `basic_filebuf` object to control the sequence. That object is represented here as `basic_filebuf sb`.

Remarks The `basic_ofstream` class provides for mechanisms specific to output file streams.

27.8.1.9 `basic_ofstream` constructor

Default and Overloaded Constructors

Description To create a file stream object for output.

Prototype

```
basic_ofstream();  
explicit basic_ofstream  
    (const char *s, openmode mode = out | trunc);
```

Remarks The class `basic_ofstream` creates an object for handling file output. It may be opened later using the `ofstream::open()` member function. It may also be associated with a file when the object is declared. The default open mode is `ios::out`.

NOTE: There are only certain valid file opening modes for an `ofstream` object see `basic_ofstream::open()` for a list of valid opening modes.

Listing 20.10 Example of `basic_ofstream::ofstream()` usage:

Before the operation the file MW Reference may or may not exist.

```
#include <iostream>  
#include <fstream>  
#include <cstdlib>
```

```
char outFile[] = "MW Reference";
```

```
int main()
{
    using namespace std;

    ofstream out(outFile);
    if(!out.is_open())
        {cout << "file not opened"; exit(1);}

    out << "This is an annotated reference that "
        << "contains a description\n"
        << "of the Working ANSI C++ Standard "
        << "Library and other\nfacilities of "
        << "the Metrowerks Standard Library. ";

    out.close();
    return 0;
}
```

Result:

This is an annotated reference that contains a description
of the Working ANSI C++ Standard Library and other
facilities of the Metrowerks Standard Library.

27.8.1.10 Member functions

`basic_ofstream::rdbuf`

Description	To retrieve a pointer to the stream buffer.
Prototype	<code>basic_filebuf<charT, traits>* rdbuf() const;</code>
Remarks	In order to manipulate a stream for random access or other operations you must use the streams base buffer. The member function <code>rdbuf()</code> is used to return a pointer to this buffer.
Returns	A pointer to <code>filebuf</code> type.

27.8 File Based Streams

27.8.1.8 Template class *basic_ofstream*

Listing 20.11 Example of `basic_ofstream::rdbuf()` usage:

The file MW Reference before the operation contains:
This is an annotated reference that contains a description
of the Working ANSI C++ Standard Library and other
facilities of the Metrowerks Standard Library

```
#include <iostream>
#include <fstream>
#include <cstdlib>

char outFile[] = "MW Reference";

int main()
{
    using namespace std;

    ofstream out(outFile, ios::in | ios::out);
    if(!out.is_open())
        {cout << "could not open file for output"; exit(1);}
    istream inOut(out.rdbuf());

    char ch;
    while((ch = inOut.get()) != EOF)
    {
        cout.put(ch);
    }

    out << "\nAnd so it goes...";

    out.close();

    return 0;
}
```

Result:

This is an annotated reference that contains a description
of the Working ANSI C++ Standard Library and other
facilities of the Metrowerks Standard Library.

This is an annotated reference that contains a description

of the Working ANSI C++ Standard Library and other facilities of the Metrowerks Standard Library.
And so it goes...

`basic_ofstream::is_open`

Description	To test whether the file was opened.
Prototype	<code>bool is_open();</code>
Remarks	The <code>is_open()</code> function is used to check that a file stream was indeed opened and ready for output. You should always test with this function after using the constructor or the <code>open()</code> function to open a stream.
Returns	True if file stream is open and available for output.

Listing 20.12 For example of `basic_ofstream::is_open()` usage

See `basic_ofstream::ofstream()`

`basic_ofstream::open`

Description	To open or re-open a file stream for output.
Prototype	<code>void open(const char* s, openmode mode = out);</code>
Remarks	The function <code>open()</code> opens a file stream for output. The default mode is <code>ios::out</code> , but may be any valid open mode (see below.) If failure occurs <code>open()</code> calls <code>setstate(failbit)</code> which may throw an exception.
Returns	No return

27.8 File Based Streams

27.8.1.8 Template class *basic_ofstream*

Table 20.3 Legal *basic_ofstream* file opening modes.

Opening Modes	stdio equivalent
Output only	
<code>ios::out</code>	<code>"w"</code>
<code>ios::binary ios::out</code>	<code>"wb"</code>
<code>ios::out ios::trunc</code>	<code>"w"</code>
<code>ios::binary ios::out ios::trunc</code>	<code>"wb"</code>
<code>ios::out ios::app</code>	<code>"a"</code>
Input and Output	
<code>ios::in ios::out</code>	<code>"r+"</code>
<code>ios::binary ios::in ios::out</code>	<code>"rb+"</code>
<code>ios::in ios::out ios::trunc</code>	<code>"w+"</code>
<code>ios::binary ios::in ios::out ios::trunc</code>	<code>"wb+"</code>
<code>ios::binary ios::out ios::app</code>	<code>"ab"</code>

NOTE: If an attempt is made to open a file in an inappropriate file opening mode, the file will not open and a test for the object will not give false, therefore use the function `is_open()` to check for file openings.

Listing 20.13 Example of *basic_ofstream::open()* usage:

Before operation, the file MW Reference contained:
Chapter One

```
#include <iostream>
#include <fstream>
#include <cstdlib>

char outFile[] = "MW Reference";
```

```
int main()
{
using namespace std;

    ofstream out;
    out.open(outFile, ios::out | ios::app);
    if(!out.is_open())
        {cout << "file not opened"; exit(1);}

    out << "\nThis is an annotated reference that "
        << "contains a description\n"
        << "of the Working ANSI C++ Standard "
        << "Library and other\nfacilities of "
        << "the Metrowerks Standard Library.";

    out.close();
    return 0;
}
```

Result:

After the operation MW Reference contained
Chapter One

This is an annotated reference that contains a description
of the Working ANSI C++ Standard Library and other
facilities of the Metrowerks Standard Library.

basic_ofstream::close

Description The member function closes the stream but does not destroy it.

Prototype `void close();`

Remarks Use the function `close()` to close a stream. It may be re-opened at a later time using the member function `open()`. If failure occurs `open()` calls `setstate(failbit)` which may throw an exception.

Returns No return.

27.8 File Based Streams

27.8.1.11 Template class *basic_fstream*

Listing 20.14 For example of `basic_ofstream::close()` usage.

```
basic_ofstream()
```

27.8.1.11 Template class *basic_fstream*

A template class for the association of a file for input and output

The prototype is listed below. The other topic in this section is:

- [“27.8.1.12 *basic_fstream* Constructor” on page 637](#)
- [“27.8.1.13 Member Functions” on page 638](#)

Prototype

```
namespace std {
    template
        <class charT, class traits=ios_traits<charT> >
        class basic_fstream : public basic_iostream<charT, traits>
        {
        public:
            typedef charT char_type;
            typedef typename traits::int_type int_type;
            typedef typename traits::pos_type pos_type;
            typedef typename traits::off_type off_type;

            basic_fstream();
            explicit basic_fstream
                (const char *s,
                 ios_base::openmode = ios_base::in | ios_base::out);

            basic_filebuf<charT, traits>* rdbuf() const;

            bool is_open();
            void open
                (const char* s,
                 ios_base::openmode = ios_base::in | ios_base::out);
            void close();

        private:
```

```
basic_filebuf<charT, traits> sb; exposition only
};
}
```

Remarks The template class `basic_fstream` is used for both reading and writing from files.

NOTE: The `basic_fstream` supports writing to file. It uses a `basic_filebuf` object to control the sequence. That object is represented here as `basic_filebuf sb`.

27.8.1.12 `basic_fstream` Constructor

Default and Overloaded Constructor

Description To construct an object of `basic_ifstream` for input and output operations.

Prototypes

```
basic_fstream();
explicit basic_fstream
    (const char *s,
     ios_base::openmode =
     ios_base::in | ios_base::out);
```

Remarks The `basic_fstream` class is derived from `basic_iostream` and that and a `basic_filebuf` object are initialized at construction.

Listing 20.15 Example of `basic_fstream::basic_fstream()` usage

The MW Reference file contains originally
Metrowerks CodeWarrior "Software at Work"

```
#include <iostream>
#include <fstream>
#include <cstdlib>
```

```
char inFile[] = "MW Reference";
```

27.8 File Based Streams

27.8.1.11 Template class *basic_fstream*

```
int main()
{
    using namespace std;

    fstream inOut(inFile, ios::in | ios::out);
    if(!inOut.is_open())
        {cout << "Could not open file"; exit(1);}

    char str[] = "\n\tRegistered Trademark";

    char ch;
    while((ch = inOut.get())!= EOF)
    {
        cout << ch;
    }
    inOut.clear();
    inOut << str;
    inOut.close();

    return 0;
}
```

Result:

Metrowerks CodeWarrior "Software at Work"

The File now reads:

Metrowerks CodeWarrior "Software at Work"

Registered Trademark

27.8.1.13 Member Functions

basic_fstream::rdbuf

Description The `rdbuf()` function retrieves a pointer to a `filebuf` type buffer.

Prototype `basic_filebuf<charT, traits>* rdbuf() const;`

Remarks In order to manipulate for random access or use of an `fstream` stream you may need to manipulate the base buffer. The function `rdbuf()` returns a pointer to this buffer for manipulation.

Returns A pointer to type `basic_filebuf`.

Listing 20.16 Example of `basic_fstream::rdbuf()` usage

The MW Reference file contains originally
Metrowerks CodeWarrior "Software at Work"

```
#include <iostream>
#include <fstream>
#include <cstdlib>

char inFile[] = "MW Reference";

int main()
{
    using namespace std;

    fstream inOut;
    inOut.open(inFile, ios::in | ios::out);
    if(!inOut.is_open())
        {cout << "Could not open file"; exit(1);}

    char str[] = "\n\tRegistered Trademark";

    inOut.rdbuf()->pubseekoff(0,ios::end);
    inOut << str;
    inOut.close();

    return 0;
}
```

Result:
The File now reads:
Metrowerks CodeWarrior "Software at Work"
Registered Trademark

27.8 File Based Streams

27.8.1.11 Template class *basic_fstream*

basic_fstream::is_open

Description Test to ensure `basic_fstream` file is open and available for reading or writing.

Prototype `bool is_open() const`

Remarks Use the function `is_open()` for a `basic_fstream` file to ensure it is open before attempting to do any input or output operation on a file.

Returns True if a file is available and open.

For an example, see [“Example of basic_fstream::basic_fstream\(\) usage” on page 637](#).

basic_fstream::open

Description To open or re-open a file stream for input or output.

Prototypes

```
void open
    (const char* s,
     ios_base::openmode =
     ios_base::in | ios_base::out);
```

Remarks You would use the function `open()` to open a `basic_fstream` object and associate it with a file. You may use `open()` to reopen a file and associate it if the object was closed but not destroyed.

WARNING! If an attempt is made to open a file in an inappropriate file opening mode, the file will not open and a test for the object will not give false, therefore use the function `is_open()` to check for file openings.

Table 20.4 **Legal file opening modes**

Opening Modes	stdio equivalent
Input Only	
<code>ios::in</code>	"r"
<code>ios::binary ios::in</code>	"rb"
Output only	
<code>ios::out</code>	"w"
<code>ios::binary ios::out</code>	"wb"
<code>ios::out ios::trunc</code>	"w"
<code>ios::binary ios::out ios::trunc</code>	"wb"
<code>ios::out ios::app</code>	"a"
Input and Output	
<code>ios::in ios::out</code>	"r+"
<code>ios::binary ios::in ios::out</code>	"r+b"
<code>ios::in ios::out ios::trunc</code>	"w+"
<code>ios::binary ios::in ios::out ios::trunc</code>	"w+b"
<code>ios::binary ios::out ios::app</code>	"ab"

Returns No return.

For an example, see [“Example of basic_fstream::rdbuf\(\) usage” on page 639](#).

basic_fstream::close

Description The member function closes the stream but does not destroy it.

Prototype `void close();`

27.8 File Based Streams

27.8.1.11 Template class *basic_fstream*

Remarks Use the function `close()` to close a stream. It may be re-opened at a later time using the member function `open()`. If failure occurs `open()` calls `setstate(failbit)` which may throw an exception.

Returns No return.

For an example, see [“Example of `basic_fstream::basic_fstream\(\)` usage” on page 637](#).



C Library files

The header `<cstdio>` contains the C++ implementation of the Standard C Headers.

27.8.2 C Library files

`<cstdio>` Macros

Macros:

BUFSIZ	EOF	FILENAME_MAX
FOPEN_MAX	L_tmpnam	NULL
SEEK_CUR	SEEK_END	SEEK_SET
stderr	stdin	stdout
TMP_MAX	_IOFBF	_IOLBF
_IONBF		

`<cstdio>` Types

Types:

FILE	fpos_t	size_t
------	--------	--------

<stdio> Functions

Functions:

clearerr	fclose	feof
ferror	fflush	fgetc
fgetpos	fgets	fopen
fprintf	fputc	fputs
fread	freopen	fscanf
fseek	fsetpos	ftell
fwrite	getc	getchar
gets	perror	printf
putc	putchar	puts
remove	rename	rewind
scanf	setbuf	setvbuf
sprintf	scanf	tmpnam
ugetc	vprintf	vfprintf
vsprintf	tmpfile	



Annex D Strstream

The header `<strstream>` defines streambuf derived classes that allow for the formatting and storage of character array based buffers, as well as their input and output.

Overview of Strstream Classes

The sections in this chapter are:

- [“Strstreambuf Class” on page 648](#), a base class for strstream classes
 - [“Strstreambuf constructors and Destructors” on page 650](#)
 - [“Strstreambuf Public Member Functions” on page 651](#)
 - [“Protected Virtual Member Functions” on page 654](#)
- [“Istrstream Class” on page 657](#), a strstream class for input
 - [“Constructors and Destructor” on page 658](#)
 - [“Public Member Functions” on page 659](#)
- [“Ostrstream Class” on page 660](#), a strstream class for output
 - [“Constructors and Destructor” on page 661](#)
 - [“Public Member Functions” on page 663](#)
- [“Strstream Class” on page 666](#), a class for input and output
 - [“Constructors and Destructor” on page 667](#)
 - [“Public Member Functions” on page 667](#)

Header `<strstream>`

The include file `strstream` includes three classes, for in memory character array based stream input and output.

Annex D Strstream

Header <strstream>

Listing 22.1 Class declarations for header <strstream>

```
class strstreambuf
{
public:
    explicit strstreambuf(streamsize alsize_arg = 0);
    strstreambuf(void* (*palloc_arg)(size_t),
        void (*pfree_arg)(void*));
    strstreambuf(char* gnext_arg, streamsize n, char* pbeg_arg = 0);
    strstreambuf(const char* gnext_arg, streamsize n);
    strstreambuf(signed char* gnext_arg, streamsize n,
        signed char* pbeg_arg = 0);
    strstreambuf(const signed char* gnext_arg, streamsize n);
    strstreambuf(unsigned char* gnext_arg, streamsize n,
        unsigned char* pbeg_arg = 0);
    strstreambuf(const unsigned char* gnext_arg, streamsize n);
    virtual ~strstreambuf();
    void freeze(bool freezefl = true);
    char* str();
    int pcount() const;
protected:
    virtual int_type overflow (int_type c = EOF);
    virtual int_type pbackfail(int_type c = EOF);
    virtual int_type underflow();
    virtual pos_type seekoff(off_type off, ios_base::seekdir way,
        ios_base::openmode which = ios_base::in | ios_base::out);
    virtual pos_type seekpos(pos_type sp,
        ios_base::openmode which = ios_base::in | ios_base::out);
    virtual streambuf* setbuf(char* s, streamsize n);
private:
    typedef unsigned char strstate;
    static const strstate allocated = 1 << 0;
    static const strstate constant = 1 << 1;
    static const strstate dynamic = 1 << 2;
    static const strstate frozen = 1 << 3;
    static const streamsize default_alsize = 128;
    streamsize alsize_;
    void* (*palloc_)(size_t);
    void (*pfree_)(void*);
```

```
    strstate strmode_;

    void init(char* gnext_arg, streamsize n, char* pbeg_arg = 0);
};
```

```
class istrstream : public basic_istream<char>
{
public:
    explicit istrstream(const char* s);
    explicit istrstream(char* s);
    istrstream(const char* s, streamsize n);
    istrstream(char* s, streamsize n);
    virtual ~istrstream();
    strstreambuf* rdbuf() const;
    char* str();
private:
    strstreambuf strbuf_;
};
```

```
class ostrstream : public basic_ostream<char>
{
public:
    ostrstream();
    ostrstream(char* s, int n, ios_base::openmode mode =
ios_base::out);
    virtual ~ostrstream();
    strstreambuf* rdbuf() const;
    void freeze(bool freezefl = true);
    char* str();
    int pcount() const;
private:
    strstreambuf strbuf_;
};
```

```
class strstream : public basic_iostream<char>
{
public:
    // Types
```

Annex D Strstream

Strstreambuf Class

```
typedef char                                     char_type;
typedef typename char_traits<char>::int_type int_type;
typedef typename char_traits<char>::pos_type pos_type;
typedef typename char_traits<char>::off_type off_type;
// constructors/destructor
strstream();
strstream(char* s, int n, ios_base::openmode mode =
ios_base::in|ios_base::out);
virtual ~strstream();
// Members:
strstreambuf* rdbuf() const;
void freeze(bool freezefl = true);
int pcount() const;
char* str();
private:
    strstreambuf strbuf_;
};
```

Strstreambuf Class

The class `strstreambuf` is derived from `streambuf` to associate a stream with an in memory character array.

The `strstreambuf` class includes virtual protected and public member functions

- [“freeze” on page 651](#), freezes the buffer
- [“pcount” on page 652](#), determines the buffer size
- [“str” on page 653](#), returns a string
- [“setbuf” on page 654](#), a virtual function to set the buffer
- [“seekoff” on page 655](#), a virtual function for stream offset
- [“seekpos” on page 655](#), a virtual function for stream position
- [“underflow” on page 655](#), a virtual function for input error
- [“pbackfail” on page 656](#), a virtual function for put back error
- [“overflow” on page 656](#), a virtual function for output error

Listing 22.2 The strstreambuf class declaration.

```
class strstreambuf
: public streambuf
{
public:
    explicit strstreambuf(streamsize alsize_arg = 0);
    strstreambuf(void* (*palloc_arg)(size_t),
        void (*pfree_arg)(void*));
    strstreambuf(char* gnext_arg, streamsize n, char* pbeg_arg = 0);
    strstreambuf(const char* gnext_arg, streamsize n);
    strstreambuf(signed char* gnext_arg, streamsize n,
        signed char* pbeg_arg = 0);
    strstreambuf(const signed char* gnext_arg, streamsize n);
    strstreambuf(unsigned char* gnext_arg, streamsize n,
        unsigned char* pbeg_arg = 0);
    strstreambuf(const unsigned char* gnext_arg, streamsize n);
    virtual ~strstreambuf();
    void freeze(bool freezefl = true);
    char* str();
    int pcount() const;
protected:
    virtual int_type overflow (int_type c = EOF);
    virtual int_type pbackfail(int_type c = EOF);
    virtual int_type underflow();
    virtual pos_type seekoff(off_type off, ios_base::seekdir way,
        ios_base::openmode which = ios_base::in | ios_base::out);
    virtual pos_type seekpos(pos_type sp,
        ios_base::openmode which = ios_base::in | ios_base::out);
    virtual streambuf* setbuf(char* s, streamsize n);
private:
    typedef unsigned char strstate;
    static const strstate allocated = 1 << 0;
    static const strstate constant = 1 << 1;
    static const strstate dynamic = 1 << 2;
    static const strstate frozen = 1 << 3;
    static const streamsize default_alsize = 128;
    streamsize alsize_;
    void* (*palloc_)(size_t);
    void (*pfree_)(void*);
```

Annex D Strstream

Strstreambuf Class

```
strstate strmode_;
```

```
void init(char* gnext_arg, streamsize n, char* pbeg_arg = 0);  
};
```

Remarks The template class `streambuf` is an abstract class for deriving various stream buffers whose objects control input and output sequences.

Strstreambuf constructors and Destructors

Default Constructor and Overloaded Constructors

Description Construct and destruct an object of type `streambuf`.

Dynamic Prototypes `explicit strstreambuf(streamsize alsize_arg = 0);`
`strstreambuf(void* (*palloc_arg)(size_t),`
`void (*pfree_arg)(void*));`

Character Array Prototypes `strstreambuf(char* gnext_arg, streamsize n,`
`char* pbeg_arg = 0);`
`strstreambuf(const char* gnext_arg, streamsize n);`
`strstreambuf(signed char* gnext_arg,`
`streamsize n, signed char* pbeg_arg = 0);`
`strstreambuf(const signed char* gnext_arg,`
`streamsize n);`
`strstreambuf(unsigned char* gnext_arg,`
`streamsize n, unsigned char* pbeg_arg = 0);`
`strstreambuf(const unsigned char* gnext_arg,`
`streamsize n);`

Remarks The constructor sets all pointer member objects to null pointers.

The `strstreambuf` object is used usually for a intermediate storage object for input and output. The overloaded constructor that is used determines the attributes of the array object when it is created. These might be allocated, or dynamic and are stored in a bitmask type. The first two constructors listed allow for dynamic allocation. The

constructors with character array arguments will use that character array for a buffer.

Destructor

Description	To destroy a strstreambuf object.
Prototype	<code>virtual ~strstreambuf();</code>
Remarks	Removes the object from memory.

Strstreambuf Public Member Functions

Description	The public member functions allow access to member functions from derived classes.
--------------------	------------------------------------------------------------------------------------

freeze

Description	To freeze the allocation of strstreambuf.
Prototype	<code>void freeze(bool freezeFl = true);</code>
Remarks	The function <code>freeze()</code> stops allocation if the strstreambuf object is using dynamic allocation and prevents the destructor from freeing the allocation. The function <code>freeze(0)</code> when used with zero as an argument releases the freeze to allow for destruction.
Return	No return

Listing 22.3 Example of strstreambuf::freeze() usage:

```
#include <iostream>
#include <strstream>
#include <string.h>

const int size = 100;

int main()
{
```

Annex D Strstream

Strstreambuf Class

```
        // dynamic allocation minimum allocation 100
    strstreambuf strbuf(size);

        // add a string and get size
    strbuf.sputn( "Metrowerks ", strlen("Metrowerks "));
    cout << "The size of the stream is: "
        << strbuf.pcount() << endl;

        // add a string and get size
    strbuf.sputn( "CodeWarrior", strlen("CodeWarrior"));
    cout << "The size of the stream is: "
        << strbuf.pcount() << endl;

    strbuf.sputc('\0');    // null terminate for output

        // now freeze for no more growth
    strbuf.freeze();
        // try to add more
    strbuf.sputn( " -- Software at Work --",
        strlen(" -- Software at Work --"));

    cout << "The size of the stream is: "
        << strbuf.pcount() << endl;
    cout << "The buffer contains:\n"
        << strbuf.str() << endl;
    return 0;
}
```

Result:

```
The size of the stream is: 11
The size of the stream is: 22
The size of the stream is: 23
The buffer contains:
Metrowerks CodeWarrior
```

pcount

Description To determine the effective length of the buffer,

Prototype `int pcount() const;`

Remarks The function `pcount()` is used to determine the offset of the next character position from the beginning of the buffer.

Return A null terminated character array.

Listing 22.4 Example of `strstreambuf::pcount()` usage.

See: `strstreambuf::freeze`

str

Description To return the char array stored in the buffer.

Prototype `char* str();`

Remarks The function `str()` freezes the buffer and appends a null character then returns the array. The user is responsible for destruction of any dynamically allocated buffer.

Return A null terminated character array.

Listing 22.5 Example of `strstreambuf::str()` usage

```
#include <iostream>
#include <strstream>

const int size = 100;
char buf[size];
char arr[size] = "Metrowerks CodeWarrior - Software at Work";

int main()
{
    ostrstream ostr(buf, size);
    ostr << arr;

    // associate buffer
    strstreambuf *strbuf(ostr.rdbuf());
}
```

Annex D Strstream

Strstreambuf Class

```
// do some manipulations
strbuf->pubseekoff(10,ios::beg);
strbuf->sputc('\0');
strbuf->pubseekoff(0, ios::beg);

cout << "The original array was\n" << arr << "\n\n";
cout << "The strstreambuf array is\n"
    << strbuf->str() << "\n\n";
cout << "The ostream array is now\n" << buf;

return 0;
}
```

Result:

The original array was
Metrowerks CodeWarrior - Software at Work

The strstreambuf array is
Metrowerks

The ostream array is now
Metrowerks

Protected Virtual Member Functions

Protected member functions that are overridden for stream buffer manipulations by the `strstream` class and derived classes from it.

setbuf

Description To set a buffer for stream input and output sequences.

Prototype `virtual streambuf* setbuf(char* s, streamsize n);`

Remarks The function `setbuf()` is overridden in `strstream` classes.

Return The `this` pointer.

seekoff

Description To return an offset of the current pointer in an input or output streams.

Prototype

```
virtual pos_type seekoff(  
    off_type off,  
    ios_base::seekdir way,  
    ios_base::openmode  
    which = ios_base::in | ios_base::out);
```

Remarks The function `seekoff()` is overridden in `strstream` classes.

Return A `pos_type` value, which is an invalid stream position.

seekpos

Description To alter an input or output stream position.

Prototype

```
virtual pos_type seekpos(  
    pos_type sp,  
    ios_base::openmode  
    which = ios_base::in | ios_base::out);
```

Remarks The function `seekpos()` is overridden in `strstream` classes.

Return A `pos_type` value, which is an invalid stream position.

underflow

Description To show an underflow condition and not increment the get pointer.

Prototype

```
virtual int_type underflow();
```

Remarks The virtual function `underflow()` is called when a character is not available for input.

There are many constraints for `underflow()`.

Annex D Strstream

Strstreambuf Class

- The pending sequence of characters is a concatenation of end pointer minus the get pointer plus some sequence of characters to be read from input.
- The result character if the sequence is not empty the first character in the sequence or the next character in the sequence.
- The backup sequence if the beginning pointer is null, the sequence is empty, otherwise the sequence is the get pointer minus the beginning pointer.

Return The first character of the pending sequence and does not increment the get pointer. If the position is null returns `traits::eof()` to indicate failure.

pbackfail

Description To show a failure in a put back operation.

Prototype `virtual int_type pbackfail(int_type c = EOF);`

Remarks The resulting conditions are the same as the function `underflow()`.

Return The function `pbackfail()` is only called when a put back operation really has failed and returns `traits::eof`. If success occurs the return is undefined.

overflow

Description Consumes the pending characters of an output sequence.

Prototype `virtual int_type overflow (int_type c = EOF);`

Remarks The pending sequence is defined as the concatenation of the put pointer minus the beginning pointer plus either the sequence of characters or an empty sequence, unless the beginning pointer is null in which case the pending sequence is an empty sequence.

This function is called by `sputc()` and `sputn()` when the buffer is not large enough to hold the output sequence.

Overriding this function requires that:

- When overridden by a derived class how characters are consumed must be specified.
- After the overflow either the beginning pointer must be null or the beginning and put pointer must both be set to the same non-null value.
- The function may fail if appending characters to an output stream fails or failure to set the previous requirement occurs.

Return The function returns `traits::eof()` for failure or some unspecified result to indicate success.

Istrstream Class

The class `istrstream` is used to create and associate a stream with an array for input.

The `istrstream` class includes the following facilities

- [“Constructors and Destructor” on page 658](#), to create and remove an `istrstream` object
- [“rdbuf” on page 659](#), to access the buffer
- [“str” on page 660](#), returns the buffer

Listing 22.6 The `istrstream` class declaration

```
class istrstream : public basic_istream<char>
{
public:
    explicit istrstream(const char* s);
    explicit istrstream(char* s);
    istrstream(const char* s, streamsize n);
    istrstream(char* s, streamsize n);
    virtual ~istrstream();
    strstreambuf* rdbuf() const;
    char* str();
```

Annex D Strstream

Istrstream Class

```
private:
    strstreambuf strbuf_;
};
```

Constructors and Destructor

The `istrstream` class has an overloaded constructor.

Constructors

Description	Create an array based stream for input
Prototype	<pre>explicit istrstream(const char* s); explicit istrstream(char* s); istrstream(const char* s, streamsize n); istrstream(char* s, streamsize n);</pre>
Remarks	The <code>istrstream</code> constructor is overloaded to accept a dynamic or pre-allocated character based array for input. It is also overloaded to limit the size of the allocation to prevent accidental overflow

Listing 22.7 Example of usage.

```
#include <iostream>
#include <strstream>

char buf[100] = "double 3.21 string array int 321";

int main()
{
    char arr[4][20];
    double d;
    long i;

    istrstream istr(buf);
    istr >> arr[0] >> d >> arr[1] >> arr[2] >> arr[3] >> i;

    cout << arr[0] << " is " << d << "\n"
```

```
<< arr[1] << " is " << arr[2] << "\n"
<< arr[3] << " is " << i << endl;

return 0;
}
```

Result:
double is 3.21
string is array
int is 321

Destructor

Description To destroy an `istrstream` object.

Prototype `virtual ~istrstream();`

Remarks The `istrstream` destructor removes the `istrstream` object from memory.

Public Member Functions

There are two public member functions.

rdbuf

Description Returns a pointer to the `strstreambuf`

Prototype `strstreambuf* rdbuf() const;`

Remarks To manipulate a stream for random access or synchronization it is necessary to retrieve a pointer to the stream's buffer. The function `rdbuf()` allows you to retrieve this pointer.

Return A pointer to `strstreambuf`.

Annex D Strstream

Ostrstream Class

Listing 22.8 Example of `istream::rdbuf()` usage.

See: `strstreambuf::str()`

str

Description Return a pointer to the stored array.

Prototype `char* str();`

Remarks The function `str()` freezes and terminates the character array stored in the buffer with a null character. It then returns the null terminated character array.

Return A null terminated char array

Listing 22.9 Example of `istream::str()` usage.

```
#include <iostream>
#include <strstream>

const int size = 100;
char buf[size] = "Metrowerks CodeWarrior - Software at Work";

int main()
{
    istream istr(buf, size);
    cout << istr.str();
    return 0;
}
```

Result:
Metrowerks CodeWarrior - Software at Work

Ostrstream Class

The class `strstreambuf` is derived from `streambuf` to associate a stream with an array buffer for output.

The `ostrstream` class includes the following facilities

- [“Constructors and Destructor” on page 661](#)
- [“freeze” on page 663](#)
- [“pcount” on page 664](#)
- [“rdbuf” on page 665](#)
- [“str” on page 665](#)

Listing 22.10 The `ostrstream` class declaration

```
class ostrstream : public basic_ostream<char>
{
public:
    ostrstream();
    ostrstream(char* s, int n, ios_base::openmode mode =
ios_base::out);
    virtual ~ostrstream();
    strstreambuf* rdbuf() const;
    void freeze(bool freezefl = true);
    char* str();
    int pcount() const;
private:
    strstreambuf strbuf_;
};
```

Constructors and Destructor

The `ostrstream` class has an overloaded constructor.

Constructors

Description Creates a stream and associates it with a char array for output.

Prototype `ostrstream();`
 `ostrstream(char* s, int n,`
 `ios_base::openmode mode = ios_base::out);`

Annex D Strstream

Ostrstream Class

Remarks The `ostrstream` array is overloaded for association a pre allocated array or for dynamic allocation.

NOTE: When using an `ostrstream` object the user must supply a null character for termination. When storing a string which is already null terminated that null terminator is stripped off to allow for appending.

Listing 22.11 Example of `ostrstream` usage.

```
#include <iostream>
#include <strstream>

int main()
{
    ostrstream out;
    out << "Ask the teacher anything you want to know" << ends;

    istream inOut(out.rdbuf() );

    char c;
    while( inOut.get(c) ) cout.put(c);

    return 0;
}
```

Result:
Ask the teacher anything you want to know

Destructor

Description Destroys an `ostrstream` object.

Prototype `virtual ~ostrstream();`

Remarks A `ostrstream` destructor removes the `ostrstream` object from memory.

Public Member Functions

The `ostream` class has four public member functions.

freeze

Description	Freezes the dynamic allocation or destruction of a buffer.
Prototype	<code>void freeze(bool freezefl = true);</code>
Remarks	To manipulate a stream for random access or synchronization it is necessary to retrieve a pointer to the streams buffer. The function <code>rdbuf()</code> allows you to retrieve this pointer.
Return	A pointer to <code>strstreambuf</code> .

Listing 22.12 Example of `ostrstream` `freeze()` usage.

```
#include <iostream>
#include <strstream>

int main()
{
    ostrstream out;
    out << "Metowerks " << 1234;
    out << "the size of the array so far is "
        << out.pcount() << " characters \n";

    out << " Software" << '\0';
    out.freeze();    // freezes so no more growth can occur

    out << " at work" << ends;
    out << "the final size of the array  is "
        << out.pcount() << " characters \n";

    cout << out.str() << endl;

    return 0;
}
```

Annex D Strstream

Ostrstream Class

Result:

the size of the array so far is 14 characters
the final size of the array is 24 characters
Metowerks 1234 Software

pcount

Description Determines the number of bytes offset of the current stream position to the beginning of the array.

Prototype `int pcount() const;`

Remarks The function `pcount()` is used to determine the offset of the array. This may not equal to the number of characters inserted due to possible positioning operations.

Return An `int_type`, the length of the array.

Listing 22.13 Example of `ostrstream pcount()` usage.

```
#include <iostream>
#include <strstream>

int main()
{
    ostrstream out;
    out << "Metowerks " << 1234 << ends;
    out << "the size of the array so far is "
        << out.pcount() << " characters \n";

    out << " Software at work" << ends;
    out << "the final size of the array is "
        << out.pcount() << " characters \n";

    cout << out.str() << endl;

    return 0;
}
```

Result:
the size of the array so far is 15 characters
the final size of the array is 33 characters
Metoworks 1234

rdbuf

- Description** To retrieve a pointer to the streams buffer.
- Prototype** `strstreambuf* rdbuf() const;`
- Remarks** To manipulate a stream for random access or synchronization it is necessary to retrieve a pointer to the streams buffer. The function `rdbuf()` allows you to retrieve this pointer.
- Return** A pointer to `strstreambuf`.

Listing 22.14 Example of ostrstream rdbuf() usage.

See: `streambuf::pubseekoff()`

str

- Description** Returns a pointer to a character array.
- Prototype** `char* str();`
- Remarks** The function `str()` freezes any dynamic allocation.
- Return** A null terminated character array.

Listing 22.15 Example of ostrstream str() usage.

See `ostrstream::freeze()`,

Strstream Class

The class `strstream` is derived from `streambuf` to associate a stream with an array buffer for output

The `strstream` class includes the following facilities

- [“Constructors and Destructor” on page 667](#)
- [“freeze” on page 667](#)
- [“pcount” on page 668](#)
- [“rdbuf” on page 668](#)
- [“str” on page 668](#)

Listing 22.16 The `strstream` class declaration

```
class strstream : public basic_ostream<char>
{
public:
    // Types
    typedef char                                char_type;
    typedef typename char_traits<char>::int_type int_type;
    typedef typename char_traits<char>::pos_type pos_type;
    typedef typename char_traits<char>::off_type off_type;
    // constructors/destructor
    strstream();
    strstream(char* s, int n, ios_base::openmode mode =
ios_base::in|ios_base::out);
    virtual ~strstream();
    // Members:
    strstreambuf* rdbuf() const;
    void freeze(bool freeze fl = true);
    int pcount() const;
    char* str();
private:
    strstreambuf strbuf_;
};
```

Strstream Types

The `strstream` class typedefines a `char_type`, `int_type`, `pos_type` and `off_type`, for stream positioning and storage.

Constructors and Destructor

Constructors

Description Creates a stream and associates it with a char array for input and output.

Prototype

```
strstream();  
strstream(char* s, int n, ios_base::openmode mode =  
    ios_base::in|ios_base::out);
```

Remarks The `strstream` array is overloaded for association a pre allocated array or for dynamic allocation

Destructor

Description Destroys a `ststream` object.

Prototype

```
virtual ~strstream();
```

Remarks Removes the `strstream` object from memory.

Public Member Functions

The class `strstream` has four public member functions.

freeze

Description Freezes the dynamic allocation or destruction of a buffer.

Prototype

```
void freeze(bool freezefl = true);
```

Remarks The function `freeze` stops dynamic allocation of a buffer.

Annex D Strstream

Strstream Class

pcount

Description	Determines the number of bytes offset of the current stream position to the beginning of the array.
Prototype	<code>int pcount() const;</code>
Remarks	The function <code>pcount()</code> is used to determine the offset of the array. This may not equal to the number of characters inserted due to possible positioning operations.
Return	An <code>int_type</code> , the length of the array.

rdbuf

Description	Retrieves a pointer to the streams buffer
Prototype	<code>strstreambuf* rdbuf() const;</code>
Remarks	To manipulate a stream for random access or synchronization it is necessary to retrieve a pointer to the streams buffer. The function <code>rdbuf()</code> allows you to retrieve this pointer.
Return	A pointer to <code>strstreambuf</code> .

str

Description	Returns a pointer to a character array.
Prototype	<code>char* str();</code>
Remarks	The function <code>str()</code> freezes any dynamic allocation.
Return	A null terminated character array.



Mutex.h

This chapter is a reference guide to the mutex support library.

Overview of Mutex Support Library

The mutex “mutual exclusion” support library provides the mechanism for implementing thread-safe applications. This mechanism consists of software locks that prevent more than one thread having simultaneous access to critical areas of code.

The mutex support library consists of the following classes:

- [“null_mutex” on page 672](#), a class for marking a future mutual exclusion area during program development
- [“mutex” on page 673](#), a class for implementing a basic mutual exclusion area
- [“mutex_block” on page 675](#), a template class for ensuring mutual exclusion for a complete block of code
- [“mutex_arith” on page 676](#), a template class for ensuring mutual exclusion during the evaluation of arithmetic operations with side effects
- [“rw_mutex” on page 680](#), a class for ensuring mutual exclusion during read and write operations
- [“mutex_rec” on page 682](#), a template class for protecting data records against simultaneous access by more than one thread

Header <mutex.h>

Description The header <mutex.h> defines

Mutex.h

Header <mutex.h>

Prototype

```
namespace std{
class null_mutex {
public:
    inline null_mutex ();
    inline ~null_mutex ();
    inline int remove () const;
    inline int acquire (int = 0) const;
    inline int release () const;
};

class mutex {
private:
    mutex_type    _lock;
public:
    inline mutex ();
    inline ~mutex ();
    inline int remove () const;
    inline int acquire (int = 0) const;
    inline int release () const;
    bool try_lock ();
};

template <class MUTEX>
class mutex_block {
public:
    inline mutex_block (const MUTEX& m, int type = 0);
    inline ~mutex_block ();
    inline int remove () const;
    inline int release () const;
    inline int acquire (int type = 0) const;
private:
    const MUTEX&    _lock;
};

template <class TYPE, class MUTEX>
class mutex_arith {
private:
    TYPE    _count;
```

```
    MUTEX _lock;
public:
    inline mutex_arith ();
    inline mutex_arith (TYPE i);
    inline TYPE operator++ ();
    inline TYPE operator++ (int);
    inline TYPE operator+= (const TYPE i);
    inline TYPE operator-- ();
    inline TYPE operator-- (int);
    inline TYPE operator-= (const TYPE i);
    inline bool operator== (const TYPE i);
    inline bool operator!= (const TYPE i);
    inline bool operator>= (const TYPE i);
    inline bool operator> (const TYPE i);
    inline bool operator<= (const TYPE i);
    inline bool operator< (const TYPE i);
    inline void operator= (const TYPE i);
    inline operator TYPE ();
};

class rw_mutex {
private:
    mutex_type _lock;
public:
    inline rw_mutex ();
    inline ~rw_mutex ();
    inline int remove () const;
    inline int acquire (int) const;
    inline int release () const;
    bool try_rdlock ();
    bool try_wrlock ();
};

template <class MUTEX = mutex>
class mutex_rec {
public:
    mutex_rec();
    int acquire(int type = 0) const;
    int release() const;
    int remove () const;
```

Mutex.h

mutex

```
private:
    thread_id get_tid () const;
    MUTEX lock;
    thread_id thr_id;
    mutex_arith<unsigned int,mutex> nest_count
};
```

mutex

Description The mutex provides a method for defining during program development a code sequence that will need to be protected against simultaneous access by more than one thread. The mutex provides no actual protection but serves as a marker that can be replaced by actual mutex objects later in the development process.

Prototype

```
class mutex {
public:
    inline mutex ();
    inline ~mutex ();
    inline int remove () const;
    inline int acquire (int = 0) const;
    inline int release () const;
};
```

Constructor and Destructor

Constructor

Description Constructs a mutex object.

Prototype inline mutex () {

Destructor

Description Removes a mutex object.

Prototype inline ~mutex () {}

Public Member Functions

remove

Description Explicit null_mutex destruction.

Prototype `inline int remove () const { return 0; }`

acquire

Description Acquire the _lock. (Wait if necessary).

Prototype `inline int acquire (int = 0) const`

Returns Null

release

Description Release the _lock.

Prototype `inline int release () const;`

Returns Null

mutex

Description This provides for the basic mutual exclusion mechanism.

Prototype

```
class mutex {  
private:  
    mutex_type _lock;  
public:  
    inline mutex ();  
    inline ~mutex ();  
    inline int remove () const;  
    inline int acquire (int = 0) const;  
    inline int release () const;
```

```
        bool try_lock ();  
};
```

Constructor and Destructor

Constructor

Description Initialize the mutex.

Prototype `inline mutex ();`

Destructor

Description Implicit mutex destruction.

Prototype `inline ~mutex ();`

Public Member Functions

remove

Description Explicit mutex destruction.

Prototype `inline int remove () const;`

acquire

Description Acquire the `_lock`. (Wait if necessary).

Prototype `inline int acquire (int = 0) const;`

release

Description Release the `_lock`.

Prototype `inline int release () const;`

try_lock

Description Attempt to lock the mutex, but don't wait.

Prototype `bool try_lock ();`

Return True if successful.

mutex_block

Description This template provides the mutex mechanism for a complete block of code.

Prototype

```
template <class MUTEX>
class mutex_block {
public:
    inline mutex_block (const MUTEX& m, int type =
0);
    inline ~mutex_block ();
    inline int remove () const;
    inline int release () const;
    inline int acquire (int type = 0) const;
private:
    const MUTEX& _lock;
};
```

Constructor and Destructors

Constructor

Description Implicit lock acquisition

Prototype `inline mutex_block (const MUTEX& m, int type = 0);`

Destructor

Description Implicit lock release

Prototype `inline ~mutex_block ();`

Public Member Functions

remove

Description Explicit lock release

Prototype `inline int remove () const`

Return `_lock.remove();`

release

Description Explicit lock release

Prototype `inline int release () const {`

Return `_lock.release ();`

acquire

Description Explicit lock acquisition

Prototype `inline int acquire (int type = 0) const;`

Return `_lock.acquire(type);`

mutex_arith

Description This template provides inline mutual exclusion for arithmetic operations that will protect against multiple thread access during the evaluation of an expression with side effects.

Prototype `template <class TYPE, class MUTEX>
class mutex_arith {
private:`

```
    TYPE    _count;
    MUTEX   _lock;
public:
    inline mutex_arith ();
    inline mutex_arith (TYPE i);
    inline TYPE operator++ ();
    inline TYPE operator++ (int);
    inline TYPE operator+= (const TYPE i);
    inline TYPE operator-- ();
    inline TYPE operator-- (int);
    inline TYPE operator-= (const TYPE i);
    inline bool operator== (const TYPE i);
    inline bool operator!= (const TYPE i);
    inline bool operator>= (const TYPE i);
    inline bool operator> (const TYPE i);
    inline bool operator<= (const TYPE i);
    inline bool operator< (const TYPE i);
    inline void operator= (const TYPE i);
    inline operator TYPE ();
};
```

Constructors

Description Initialize the mutex_arith object.

Prototype inline mutex_arith ()

Remarks Initialize _count to 0

Prototype inline mutex_arith (TYPE i)

Remarks Initialize _count to i.

Public Member Operators

Prefix operator++

Description Increment _count by unit.

Prototype inline TYPE operator++ ();

Postfix operator++

Description Increment `_count` by unit.

Prototype `inline TYPE operator++ (int);`

operator+=

Description Increment `_count` by `i`.

Prototype `inline TYPE operator+= (const TYPE i)`

Prefix operator--

Description Decrement `_count` by unit.

Prototype `inline TYPE operator-- ()`

Postfix operator--

Description Decrement `_count` by unit.

Prototype `inline TYPE operator-- (int)`

operator-=

Description Decrement `_count` by `i`.

Prototype `inline TYPE operator-= (const TYPE i);`

Return `_count -= i;`

operator==

Description Compare `_count` with `i`.

Prototype `inline bool operator== (const TYPE i);`

Return `_count == i;`

Remarks

operator!=

Description Compare `_count` with `i`.

Prototype `inline bool operator!= (const TYPE i)`

Return `_count != i;`

operator>=

Description Check if `_count >= i`.

Prototype `inline bool operator>= (const TYPE i);`

Return `_count >= i;`

operator>

Description Check if `_count > i`.

Prototype `inline bool operator> (const TYPE i);`

Return `_count > i;`

operator<=

Description Check if `_count <= i`.

Prototype `inline bool operator<= (const TYPE i);`

Return `_count <= i;`

operator<

Description Check if `_count < i`.

Prototype `inline bool operator< (const TYPE i);`

Return `_count < i;`

operator=

Description Assign `i` to `count_`

Prototype `inline void operator= (const TYPE i);`

operator TYPE

Description Return `count_`.

Prototype `inline operator TYPE ();`

Return `_count`

rw_mutex

Description This template provides mutual exclusion for input and output operations.

Prototype

```
class rw_mutex {
private:
    mutex_type _lock;
public:
    inline rw_mutex ();
    inline ~rw_mutex ();
    inline int remove () const;
    inline int acquire (int) const;
    inline int release () const;
    bool try_rdlock ();
```



```
        bool try_wrlock ();  
};
```

Constructor and Destructor

Constructor

Description Initialize the readers/writer mutex.

Prototype `inline rw_mutex ();`

Destructor

Description Implicit mutex destruction.

Prototype `inline ~rw_mutex ();`

Public Member Functions

remove

Description Explicit `rw_mutex` destruction.

Prototype `inline int remove () const;`

acquire

Description Acquire the `_lock`. Wait if necessary.

Prototype `inline int acquire (int) const;`

release

Description Release the `_lock`.

Prototype `inline int release () const;`

try_rdlock

Description Attempt to lock the mutex for reading, but don't wait.

Prototype `bool try_rdlock ();`

Return True if successful.

try_wrlock

Description Attempt to lock the mutex for writing, but don't wait.

Prototype `bool try_wrlock ();`

Return True if successful.

mutex_rec

Description This template provides mutual exclusion mechanism for protecting against simultaneous access to a data record.

Prototype

```
template <class MUTEX = mutex>
class mutex_rec {
public:
    mutex_rec();
    int acquire(int type = 0) const;
    int release() const;
    int remove() const;
private:
    thread_id get_tid () const;
    MUTEX lock;
    thread_id thr_id;
    mutex_arith<unsigned int,mutex> nest_count;
};
```

Constructor

Description Creates a mutex_rec class.

Prototype `mutex_rec()`

Public Members Functions

acquire

Description Acquire the _lock. Wait if necessary.

Prototype `int acquire(int type = 0) const`

release

Description Release the _lock.

Prototype `int release() const`

remove

Description Explicit mutex_rec destruction.

Prototype `int remove () const`

Mutex.h

mutex_rec

Index

Symbols

`_MSL_CX_LIMITED_RANGE` 419

A

`abs` 434

algorithms

categories 315

generalized numeric 366

accumulate 366

in-place and copying versions 315

mutating sequence 324

copy 324

partition 339

random_shuffle 339

replace 328

reverse 337

rotate 337

swap 326, 327

unique 335

non mutating sequence 317

count 320

equal 321

find 317

for_each 317

mismatch 321

search 323

numeric processing 366

sorting 341

binary_search 346

nth_element 345

with predicate parameters 316

Allocator object information 82

Arbitrary-Positional Stream 44

`arg` 434

`assign` 106

Assignment Operator

complex 427

Associative Containers

Basic Design and Organization 197

Types and Member Functions 198

`auto_ptr` destructor 101

B

`bad` 484

`basic_filebuf` 614

close 619

constructors 616

destructor 616

imbue 622

is_open 616

open 617

Open Modes 617

overflow 620

pbackfail 620

seekoff 621

seekpos 621

setbuf 621

showmanyc 619

sync 622

underflow 620

`basic_fstream` 636

close 641

constructor 637

is_open 640

open 640

Open Modes 641

rdbuf 638

`basic_ifstream` 622

close 628

constructor 623

is_open 626

open 626

Open Modes 627

rdbuf 625

`basic_ios` 468

`bad` 484

`clear` 478

constructors 470

`copyfmt` 475

`eof` 481

exceptions 486

`fail` 483

`fill` 474

`good` 481

`imbue` 474

`operator !` 476

`operator bool` 476

`rdbuf` 473

Index

- rdstate 476
- setstate 480
- basic_iostream 556
 - constructor 556
 - destructor 556
- basic_istream 523
 - constructors 526
 - destructor 526
 - extractors, arithmetic 529
 - extractors, characters 530
 - gcount 535
 - get 537
 - getline 540
 - ignore 542
 - peek 544
 - putback 547
 - read 544
 - readsome 546
 - seekg 552
 - sentry 527
 - sync 551
 - tellg 552
 - unget 549
 - ws 554
- basic_istreamstream 597
 - constructors 598
 - rdbuf 599
 - str 600
- basic_ofstream 629
 - close 635
 - constructors 630
 - is_open 633
 - open 633
 - Open Modes 634
 - rdbuf 631
- basic_ostream 557
 - constructor 559
 - destructor 559
 - endl 576
 - ends 576
 - flush 573
 - flush,flush 577
 - Inserters, arithmetic 562
 - Inserters, characters 564
 - put 570
 - resetiosflags 580
 - seekp 569
 - sentry 560
 - setbase 582
 - setfill 583
 - setiosflags 581
 - setprecision 584
 - setw 585
 - tellp 568
 - write 571
- basic_ostringstream 601
 - constructors 603
 - rdbuf 604
 - str 606
- basic_streambuf 494
 - constructors 497, 591
 - destructor 497
 - eback 511
 - egptr 511
 - eptr 513
 - gbump 512
 - getloc 498
 - gptr 511
 - imbue 514
 - in_avail 504
 - Locales 498
 - overflow 519, 595
 - pbackfail 518, 595
 - pbase 512
 - pbump 513
 - pptr 513
 - pubseekoff 500
 - pubseekpos 501
 - pubsetbuf 499
 - pubsync 503
 - pubuimbue 498
 - sbumpc 505
 - seekoff 515, 596
 - seekpos 515, 596
 - setbuf 514
 - setg 512
 - setp 513
 - sgetc 506
 - sgetn 507
 - showmanic 516
 - snextc 504
 - sputback 507
 - sputc 509
 - sputn 510

-
- str 593
 - sungetc 509
 - sync 515
 - uflow 517
 - underflow 516, 594
 - xsgetn 516
 - xspn 518
 - basic_string
 - append 127
 - assign 128
 - assignment operator 123
 - at 127
 - begin 124
 - c_str 131
 - capacity 125, 126
 - clear 126
 - compare 135
 - Constructors 122
 - copy 131
 - data 131
 - destructor 123
 - Element Access 126
 - empty 126
 - end 124
 - erase 129
 - extractor 142
 - find 132
 - find_first_not_of 134
 - find_first_of 133
 - find_last_of 133
 - get_allocator 131
 - getline 142
 - insert 128
 - inserter 142
 - Inserters and extractors 142
 - iterator support 124
 - max_size 125
 - Modifiers 127
 - Non-Member Functions and Operators 136
 - Null Terminated Sequence Utilites 143
 - operator 138, 140, 142
 - operator!= 138
 - operator+ 136
 - operator+= 127
 - operator== 137
 - operator> 139
 - operator>= 141
 - operator<< 142
 - rbegin 124
 - rend 125
 - replace 129
 - reserve 126
 - rfind 132
 - size 125
 - String Operations 131
 - substr 135
 - swap 131
 - basic_stringbuf 590
 - basic_stringstream 607
 - constructors 608
 - rdbuf 610
 - str 611
 - boolalpha 487
 - Buffer management 498
- ## C
- C Library files 643
 - cerr 446
 - char_type 107
 - Character 44
 - character 106
 - character container type 106
 - Character Sequences 44
 - Character Trait Definitions 106, 107
 - Character traits definitions 106
 - Class
 - basic_filebuf 614
 - basic_fstream 636
 - basic_ifstream 622
 - basic_ios 468
 - basic_iostream 556
 - basic_istream 523
 - sentry 527
 - basic_istream 597
 - basic_ofstream 629
 - basic_ostream 557
 - sentry 560
 - basic_ostringstream 601
 - basic_streambuf 494
 - basic_stringbuf 590
 - basic_stringstream 607
 - complex 426

Index

- ios_base 451
 - failure 453
 - Init 457
- clear 478
- clog 447
- close
 - basic_filebuf 619
 - basic_fstream 641
 - basic_ifstream 628
 - basic_ofstream 635
- compare 106
- Comparison Function 44
- Comparison Operations
 - Deque 217
 - List 226
 - Map 257
 - Multimap 268
 - Set 237
 - Vector 208
- complex 426, 430
 - abs 434
 - arg 434
 - conj 434
 - constructor 426
 - cos 435
 - cosh 436
 - exp 436
 - imag 427, 433
 - log 436
 - log10 436
 - norm 434
 - operator 433
 - operator != 432
 - operator * 431
 - operator *= 429
 - operator + 430
 - operator += 428
 - operator / 431
 - operator /= 429
 - operator -= 428
 - operator = 427
 - operator == 432
 - operator >> 432
 - polar 435
 - pow 437
 - real 427, 433
 - sin 437
 - sinh 437
 - sqrt 437
 - tan 438
 - tanh 438
- Complexity of
 - Deque Insertion 221
 - List Insertion 232
 - Vector Insertion 212
- Component 45
- Conforming Implementations 54
- conj 434
- Constraints on programs 52
- Constructors 121
 - basic_filebuf 616
 - basic_fstream 637
 - basic_ifstream 623
 - basic_ios 470
 - basic_iostream 556
 - basic_istream 526
 - basic_istreamstringstream 598
 - basic_ofstream 630
 - basic_ostream 559
 - basic_ostreamstringstream 603
 - basic_streambuf 497, 591
 - basic_stringstream 608
 - failure, ios_base 454
 - ios_base 468
 - istream 658
 - list 225
 - Map 256
 - Multimap 267
 - ostream 661
 - sentry, basic_istream 528
 - sentry, basic_ostream 561
 - stringstream 667
 - stringstreambuf 650
- Container Classes
 - Common Members of 189
- copy 107
- copyfmt 475
- cos 435
- cosh 436
- cout 446
- cstdio
 - Functions 644

Macros 643
Types 643

D

dec 489
Default Behavior 45
Deque
 Comparison Operations 217
 Complexity of Deque Insertion 221
 Constructors 215
 Element Access Member Functions 217
 Insert Member Functions 219
 Notes on Deque Erase Member Functions 222
 Related Functions 215
Destructor
 istream 659
 ostream 662
 stringstream 667
Destructors
 basic_filebuf 616
 basic_ios 470
 basic_istream 556
 basic_istream 526
 basic_ostream 559
 basic_streambuf 497
 deque 215
 Init, ios_base 458
 ios_base 468
 list 225
 Map 256
 Multimap 267
 Multiset 246
 sentry, basic_istream 528
 sentry, basic_ostream 561
 strstreambuf 651
 Vector 206

E

eback 511
egptr 511
endl 576
ends 576
eof 107, 481
epptr 513
eq 106

eq_int_type 107
exceptions
 basic_ios 486
exp 436
External "C" Linkage 53
Extractors
 basic_istream, arithmetic 529
 basic_istream, characters 530
 overloading 533

F

fail 483
fill 474
find 106
fixed 489
flags 458
flush 573
fmtflags 454
Freestanding Implementations 51
freeze
 ostream 663
 stringstream 667
 stringstreambuf 651
fstream 613
Function Objects
 Introduction to Function Objects 90

G

gbump 512
gcount 535
get 537
get_state 107
getline 540
getloc
 basic_streambuf 498
 ios_base 466
good 481
gptr 511

H

Handler Function 45
Headers 109
 fstream 613
 ios 449

Index

- iosfwd 443
- iostream 445
- istream 521
- mutex.h 669
- streambuf 493
- strstream 645
- heap operations 357
- hex 489

I

- I/O Library Summary 439
- ignore 542
- imag 433
 - complex 427
- imbue
 - basic_filebuf 622
 - basic_ios 474
 - basic_streambuf 514
 - iosbase 465
- in_avail 504
- Inserters
 - basic_ostream, arithmetic 562
 - basic_ostream, characters 564
 - overloading 566
- int_type 107
- internal 488
- ios 449
- ios_base 451
 - constructors 468
 - failure 453
 - constructor 454
 - what 454
 - flags 458
 - fmtflags 454
 - getloc 466
 - imbue 465
 - Init 457
 - destructor 458
 - iostate 456
 - iword 466
 - Open Modes 456
 - precision 463
 - pword 467
 - register_callback 467
 - seekdir 457

- setf 461
- sync_with_stdio 468
- unsetf 462
- width 464
- xalloc 466
- iosfwd 443
- iostate 456
- iostream 445
- Iostream Class Templates 45
- Iostreams Definitions 440
- Iostreams requirements 440
- is_open
 - basic_filebuf 616
 - basic_fstream 640
 - basic_ifstream 626
 - basic_ofstream 633
- istream 521
- istream win 447
- istrstream 657
 - constructor 658
 - destructor 659
 - rdbuf 659
 - str 660
- iword 466

L

- Leading Underscores 53
- left 488
- length 106
- Library-wide Requirements 49
- Linkage 52
- List
 - Comparison Operations 226
 - Erase Member Functions 229
 - Notes on list erase member functions 232
 - Related Functions 225
- Locales
 - basic_streambuf 498
- log 436
- log10 436
- lt 106

M

- Manipulator

-
- Overloading 586
 - scientific 489
 - Manipulators
 - adjustfield 488
 - basefield 488
 - boolalpha 487
 - dec 489
 - endl 576
 - ends 576
 - fixed 489
 - floatfield 489
 - flush 577
 - fmtflags 486
 - hex 489
 - Instantiations 580
 - internal 488
 - ios_base 486
 - left 488
 - noboolalpha 487
 - noshowbase 487
 - noshowpoint 487
 - noshowpos 487
 - noskipws 487
 - nounitbuf 488
 - nouppercase 487
 - oct 489
 - overloaded 490
 - right 488
 - showbase 487
 - showpoint 487
 - showpos 487
 - skipws 487
 - uppercase 487
 - ws 554
 - Map
 - Comparison Operations 257
 - Constructors 256
 - Destructors 256
 - Element Access Member Functions 258
 - Erase Member Functions 261
 - Insert Member Functions 260
 - Related Functions 256
 - Special Operations 261
 - Memory Model Information 82
 - Modifier Function 45
 - move 106
 - mtx.h 669
 - Multimap
 - Comparison Operations 268
 - Constructors 267
 - Destructors 267
 - Element Access Member Functions 269
 - Erase Member Functions 271
 - Insert Member Functions 270
 - Related Functions 267
 - Special Operations 272
 - Multiset
 - Comparison Operations 247
 - Element Access Member Functions 248
 - Erase Member Functions 250
 - Insert Member Functions 249
 - Special Operations 251
 - mutex 673
 - acquire 674
 - Constructor 674
 - Destructor 674
 - release 674
 - remove 674
 - try_lock 675
 - mutex_arith 676
 - Constructor 677
 - operator 679, 680
 - operator TYPE 680
 - operator!= 679
 - operator+= 678
 - operator-= 678
 - operator= 680
 - operator== 678
 - operator> 679
 - operator>= 679
 - Postfix operator-- 678
 - Postfix operator++ 678
 - Prefix operator-- 678
 - Prefix operator++ 677
 - mutex_block 675
 - acquire 676
 - Constructor 675
 - Destructor 675
 - release 676
 - remove 676
 - mutex_rec 682
 - acquire 683

Index

- Constructor 683
- release 683
- remove 683

N

- Narrow-oriented Iostream Classes 45
- noboolalpha 487
- norm 434
- noshowpoint 487
- noshowpos 487
- noskipws 487
- not_eof 107
- nounitbuf 488
- nouppercase 487
- NTCTS 46, 106
- null_mutex 672
 - acquire 673
 - Constructor 672
 - Destructor 672
 - release 673
 - remove 673

O

- ObjectState 45
- Observer Function 46
- oct 489
- off_type 107
- open
 - basic_filebuf 617
 - basic_fstream 640
 - basic_ifstream 626
 - basic_ofstream 633
- Open Modes
 - basic_filebuf 617
 - basic_fstream 641
 - basic_ifstream 627
 - basic_ofstream 634
 - ios_base 456
- operator 433
- operator - 430
 - complex 430
- Operator !
 - basic_ios 476
- operator !=
 - complex 432
- operator *
 - complex 431
- operator *=
 - complex 429
- operator +
 - complex 430
- operator +=
 - complex 428
- operator /
 - complex 431
- operator /=
 - complex 429
- operator -=
 - complex 428
- operator =
 - complex 427
- operator ==
 - complex 432
- operator >>
 - complex 432
- Operator bool
 - basic_ios 476
 - sentry, basic_istream 528
 - sentry, basic_ostream 561
- ostream cerr 446
- ostream clog 447
- ostream cout 446
- ostream wout 447
- ostrstream 660
 - constructor 661
 - destructor 662
 - freeze 663
 - pcount 664
 - rdbuf 665
 - str 665
- Other Conventions 48
- overflow 519, 656
 - basic_filebuf 620
 - basic_streambuf 595
- Overloaded
 - manipulators 490
- Overloading
 - Extractors 533
 - Inserters 566

Manipulator 586

P

pbackfail 518, 656
 basic_filebuf 620
 basic_streambuf 595

pbase 512

pbump 513

pcount

 ostrstream 664

 strstream 668

 strstreambuf 652

peek 544

pointing 272

polar 435

pos_type 108

pow 437

pptr 513

precision 463

predicate parameters 316

predicates

 binary 316

pubimbue 498

pubseekoff 500

pubseekpos 501

pubsetbuf 499

pubsync 503

put 570

putback 547

pword 467

R

rdbuf 473

 basic_fstream 638

 basic_ifstream 625

 basic_istream 599

 basic_ofstream 631

 basic_ostringstream 604

 basic_stringstream 610

 istrstream 659

 ostrstream 665

 strstream 668

rdstate 476

read 544

readsome 546

real 433

 complex 427

register_callback 467

Replacement Function 46

Replacement Functions 53

Repositional Stream 46

Required Behavior 46

Reserved Function 46

Reserved Names 53

resetiosflags 580

Restrictions On Exception Handling 54

right 488

rw_mutex 680

 acquire 681

 Constructor 681

 Destructor 681

 release 681

 remove 681

 try_rdlock 682

 try_wrlock 682

S

sbumc 505

scientific 489

seekdir 457

seekg 552

seekoff 515

 basic_filebuf 621

 basic_streambuf 596

 strstreambuf 655

seekp 569

seekpos 515

 basic_filebuf 621

 basic_streambuf 596

 strstreambuf 655

sentry 527, 560

 constructor

 basic_istream 528

 basic_ostream 561

 destructor

 basic_istream 528

 basic_ostream 561

 Operator bool

Index

- basic_istream 528
 - basic_ostream 561
 - Set
 - Comparison Operations 237
 - Constructors 236
 - Element Access Member Functions 238
 - Erase member Functions 240
 - Insert Member Functions 239
 - Special Set Operations 241
 - setbase 582
 - setbuf 514, 621
 - strstreambuf 654
 - setf 461
 - setfill 583
 - setg 512
 - setiosflags 581
 - setp 513
 - setprecision 584
 - setstate 480
 - setw 585
 - sgetc 506
 - sgetn 507
 - Shallow Copies 195
 - showmanc 516
 - showmanyc
 - basic_filebuf 619
 - showpoint 487
 - showpos 487
 - sin 437
 - sinh 437
 - skipws 487
 - snextc 504
 - sorting and related algorithms 341
 - sputback 507
 - sputc 509
 - sputn 510
 - sqrt 437
 - state_type 108
 - str
 - basic_istream 600
 - basic_ostream 606
 - basic_streambuf 593
 - basic_stringstream 611
 - istream 660
 - ostream 665
 - strstream 668
 - strstreambuf 653
 - Stream Iterators
 - Introduction to Stream Iterators 291
 - streambuf 493
 - string 109
 - strstream 645, 661
 - constructor 650, 667
 - destructor 667
 - freeze 667
 - pcount 668
 - rdbuf 668
 - str 668
 - strstreambuf 648
 - freeze 651
 - pcount 652
 - seekof 655
 - seekpos 655
 - setbuf 654
 - str 653
 - stream
 - overflow 656
 - pbackfail 656
 - underflow 655
 - strtreambuf
 - destructor 651
 - struct char_traits 108
 - sungetc 509
 - swap
 - basic_string 141
 - sxputn 518
 - sync 551
 - basic_filebuf 622
 - basic_streambuf 515
 - sync_with_stdio
 - ios_base 468
- ## T
- tan 438
 - tanh 438
 - tellg 552
 - tellp 568
 - to_char_type 107
 - to_int_type 107

Traits 46
traits 106
Translation Units 52

U

uflow 517
underflow 516
 basic_filebuf 620
 basic_streambuf 594
 strstreambuf 655
unget 549
unsetf 462
uppercase 487
Using the library 52

V

vector
 Comparison Operations 208
 Constructors 206
 Element Access Member Functions 208

Erase Member Functions 211
Insert Member Functions 210
Type Definitions 204

W

werr 448
Wide-oriented Iostream Classes 47
width 464
win 447
wlog 448
wostream werr 448
wostream wlog 448
wout 447
write 571
ws 554

X

xalloc 466
xsgetn 516

CodeWarrior

MSL C++ Reference

Credits

writing lead: Ron Liechty

other writers: Portions copyright © 1995 by Modena Software Inc.

engineering: Vicki Scott, Howard Hinnant and all other Libraries team members

frontline warriors: MPTP testers



Guide to CodeWarrior Documentation

CodeWarrior documentation is modular, like the underlying tools. There are manuals for the core tools, languages, libraries, and targets. The exact documentation provided with any CodeWarrior product is tailored to the tools included with the product. Your product will not have every manual listed here. However, you will probably have additional manuals (not listed here) for utilities or other software specific to your product.

Core Documentation	
IDE User Guide	How to use the CodeWarrior IDE
Debugger User Guide	How to use the CodeWarrior debugger
CodeWarrior Core Tutorials	Step-by-step introduction to IDE components
Language/Compiler Documentation	
C Compilers Reference	Information on the C/C++ front-end compiler
Pascal Compilers Reference	Information on the Pascal front-end compiler
Error Reference	Comprehensive list of compiler/linker error messages, with many fixes
Pascal Language Reference	The Metrowerks implementation of ANS Pascal
Assembler Guide	Stand-alone assembler syntax
Command-Line Tools Reference	Command-line options for Mac OS and Be compilers
Plugin API Manual	The CodeWarrior plugin compiler/linker API
Library Documentation	
MSL C Reference	Function reference for the Metrowerks ANSI standard C library
MSL C++ Reference	Function reference for the Metrowerks ANSI standard C++ library
Pascal Library Reference	Function reference for the Metrowerks ANS Pascal library
MFC Reference	Reference for the Microsoft Foundation Classes for Win32
Win32 SDK Reference	Microsoft's Reference for the Win32 API
The PowerPlant Book	Introductory guide to the Metrowerks application framework for Mac OS
PowerPlant Advanced Topics	Advanced topics in PowerPlant programming for Mac OS
Targeting Manuals	
Targeting BeOS	How to use CodeWarrior to program for BeOS
Targeting Java VM	How to use CodeWarrior to program for the Java Virtual Machine
Targeting Mac OS	How to use CodeWarrior to program for Mac OS
Targeting MIPS	How to use CodeWarrior to program for MIPS embedded processors
Targeting NEC V810/830	How to use CodeWarrior to program for NEC V810/830 processors
Targeting Net Yaroze	How to use CodeWarrior to program for Net Yaroze game console
Targeting Nucleus	How to use CodeWarrior to program for the Nucleus RTOS
Targeting OS-9	How to use CodeWarrior to program for the OS-9 RTOS
Targeting Palm OS	How to use CodeWarrior to program for PalmPilot
Targeting PlayStation OS	How to use CodeWarrior to program for the PlayStation game console
Targeting PowerPC Embedded Systems	How to use CodeWarrior to program for PPC embedded processors
Targeting VxWorks	How to use CodeWarrior to program for the VxWorks RTOS
Targeting Win32	How to use CodeWarrior to program for Windows