

CodeWarrior™

Development Tools

MSL C++ Reference

Revised 030711



Metrowerks, the Metrowerks logo, and CodeWarrior are trademarks or registered trademarks of Metrowerks Corp. in the US and/or other countries. All other tradenames and trademarks are the property of their respective owners.

Copyright © Metrowerks Corporation. 2003. ALL RIGHTS RESERVED.

The reproduction and use of this document and related materials are governed by a license agreement media, it may be printed for non-commercial personal use only, in accordance with the license agreement related to the product associated with the documentation. Consult that license agreement before use or reproduction of any portion of this document. If you do not have a copy of the license agreement, contact your Metrowerks representative or call 800-377-5416 (if outside the US call +1-512-996-5300). Subject to the foregoing non-commercial personal use, no portion of this documentation may be reproduced or transmitted in any form or by any means, electronic or mechanical, without prior written permission from Metrowerks.

Metrowerks reserves the right to make changes to any product described or referred to in this document without further notice. Metrowerks makes no warranty, representation or guarantee regarding the merchantability or fitness of its products for any particular purpose, nor does Metrowerks assume any liability arising out of the application or use of any product described herein and specifically disclaims any and all liability. **Metrowerks software is not authorized for and has not been designed, tested, manufactured, or intended for use in developing applications where the failure, malfunction, or any inaccuracy of the application carries a risk of death, serious bodily injury, or damage to tangible property, including, but not limited to, use in factory control systems, medical devices or facilities, nuclear facilities, aircraft navigation or communication, emergency systems, or other applications with a similar degree of potential hazard.**

USE OF ALL SOFTWARE, DOCUMENTATION AND RELATED MATERIALS ARE SUBJECT TO THE METROWERKS END USER LICENSE AGREEMENT FOR SUCH PRODUCT.

How to Contact Metrowerks

Corporate Headquarters	Metrowerks Corporation 7700 West Parmer Lane Austin, TX 78729 U.S.A.
World Wide Web	http://www.metrowerks.com
Sales	Voice: 800-377-5416 Fax: 512-996-4910 Email: sales@metrowerks.com
Technical Support	Voice: 800-377-5416 Voice: 512-996-5300 Email: support@metrowerks.com

Table of Contents

1	Introduction	1
	About the MSL C++ Library Reference Manual	1
2	The C++ Library	5
	The MSL C++ Library Overview	5
	Definitions	5
	Additional Definitions	8
	Multi-Thread Safety	9
	MSL C++ Thread Safety Policy	9
	Methods of Descriptions	10
	Other Conventions	11
	Library-wide Requirements	12
	Library contents and organization	12
	Using the library	14
	Constraints on programs	15
	Conforming Implementations	17
	Reentrancy	17
	Restrictions On Exception Handling	17
3	Language Support Library	19
	The Language Support Library	19
	Types	19
	Implementation properties	20
	Numeric limits	20
	is_specialized	21
	min	21
	max	21
	digits	21
	is_signed	22
	is_integer	22
	is_exact	22

Table of Contents

radix	22
epsilon	23
round_error	23
min_exponent	23
min_exponent10	23
max_exponent	24
max_exponent10	24
has_infinity	24
has_quiet_NaN	24
has_signaling_NaN	24
has_denorm	25
has_denorm_loss	25
infinity	25
quiet_NaN	25
signaling_NaN	26
denorm_min	26
is_iec559	26
is_bounded	27
is_modulo.	27
traps.	27
tinyness_before.	27
round_style	28
Type float_round_style.	28
Type float_denorm_style	28
numeric_limits specializations.	29
C Library	29
Start and termination.	30
abort	30
atexit.	31
exit	31
Dynamic Memory Management	32
operator new.	32
operator delete	33

Array Forms	33
operator new[]	33
operator delete[]	34
Placement Forms	34
placement operator new	34
placement operator delete	34
Storage Allocation Errors.	34
Class Bad_alloc	35
Constructor	35
Assignment Operator	35
destructor	35
what	35
type new_handler	36
set_new_handler	36
Type identification	36
Class type_info.	37
operator==	37
operator!=.	37
before	37
name	38
Constructors	38
Assignment Operator	38
Class bad_cast	38
Constructors	38
Assignment Operator	39
what	39
Class bad_typeid	39
Constructors	39
Assignment Operator	39
what	40
Exception Handling	40
Class Exception	40
Constructors	40

Table of Contents

Assignment Operator	40
destructor	41
what	41
Violating Exception Specifications	41
Constructors	41
Assignment Operator	42
what	42
type unexpected_handler	42
set_unexpected	42
unexpected	43
Abnormal Termination	43
type terminate_handler	43
set_terminate	43
terminate	44
uncaught_exception	44
Other Runtime Support	44

4 Diagnostics Library 47

The Diagnostics library	47
Exception Classes	47
Class Logic_error	48
Class domain_error	48
Class Invalid_argument	48
Class Length_error	49
Class Out_of_range	49
Class Runtime_error	49
Class Range_error	50
Class Overflow_error	50
Class Underflow_error	50
Assertions	51
Error Numbers	51

5 General Utilities Libraries	53
The General Utilities Library	53
Requirements	53
Utility Components	56
Operators	56
operator!=	56
operator>	56
operator<=	56
operator>=	57
Pairs	57
Constructors	57
operator ==	57
operator <	58
make_pair	58
Function objects	58
Arithmetic operations	59
plus	59
minus	59
multiplies	60
divides	60
modulus	60
negate	61
Comparisons.	61
equal_to	61
not_equal_to.	62
greater	62
less	63
greater_equal	63
less_equal	64
Logical operations	64
logical_and	64
logical_or	65
logical_not	65

Table of Contents

Negators	65
Unary_negate	66
binary_negate	66
Binders	67
Template class binder1st	67
bind1st	67
Template class binder2nd	67
bind2nd	68
Adaptors for Pointers to Functions	68
pointer_to_unary_function	68
class pointer_to_binary_function	68
pointer_to_binary_function	69
Adaptors for Pointers to Members	69
mem_fun_t	69
mem_fun1_t	70
mem_fun	70
mem_fun_ref_t	71
mem_fun1_ref_t	71
mem_fun_ref	72
const_mem_fun_t	72
const_mem_fun1_t	73
const_mem_fun_ref_t	73
const_mem_fun1_ref_t	74
Memory	74
allocator members	74
address	74
allocate	75
deallocate	75
max_size	75
construct	75
destroy	76
allocator globals	76
operator==	76

operator!=	76
Raw storage iterator	77
Constructors	77
operator *	77
operator=	78
operator++	78
Temporary buffers	78
get_temporary_buffer	79
return_temporary_buffer	79
Specialized Algorithms	79
uninitialized_copy	80
uninitialized_fill	80
uninitialized_fill_n	81
Template Class Auto_ptr	81
auto_ptr constructors	83
operator =	84
destructor	84
Auto_ptr Members	84
operator*	84
operator->	85
get	85
release	85
reset	85
auto_ptr conversions	86
Conversion Constructor	86
operator auto_ptr_ref	86
operator auto_ptr	86
C Library	86
Date and Time	87
6 Strings Library	89
The Strings Library	89
Character traits	89

Table of Contents

Character Trait Definitions	90
Character Trait Requirements	90
assign	90
eq	91
lt	91
compare	91
length	91
find	92
move	92
copy	92
not_eof	92
to_char_type	93
to_int_type	93
eq_int_type	93
get_state	93
eof	94
Character Trait Type Definitions	94
struct char_traits<T>	94
String Classes	94
Class Basic_string	95
Constructors and Assignments	96
Constructors	96
Destructor	97
Assignment Operator	98
Assignment & Addition Operator basic_string	98
Iterator Support	98
begin	99
end	99
rbegin	99
rend	99
Capacity	100
size	100
length	100

max_size	100
resize	100
capacity	101
reserve	101
clear	101
empty	101
Element Access	102
operator[]	102
at	102
Modifiers	102
operator+=	102
append	103
assign	104
insert.	105
erase	106
replace	107
copy	108
swap	108
String Operations	109
c_str	109
data	109
get_allocator.	109
find	110
rfind	111
find_first_of	112
find_last_of	113
find_first_not_of	114
find_last_not_of	115
substr	115
compare	116
Non-Member Functions and Operators	117
operator+	117
operator==	118

Table of Contents

operator!=	119
operator<	120
operator>	121
operator<=	122
operator>=	123
swap	124
Inserters and extractors.	124
operator>>	124
operator<<	125
getline	125
Null Terminated Sequence Utilities	126
Character Support.	126
String Support	126
Input and Output Manipulations	127

7 Localization Library 129

The Localization library	129
Supported Locale Names	129
Strings and Characters in Locale Data Files	130
Locales	133
Class locale	133
Combined Locale Names	133
Locale Types	134
locale::Category	135
Locale::facet.	136
locale::id	137
Constructors	137
destructor	138
combine	138
name	138
operator ==	139
operator !=	139
operator ()	139

global	140
classic	140
Locale Globals	140
use_facet	140
has_facet	141
Convenience Interfaces	141
Character Classification	141
toupper	142
tolower	142
Standard Locale Categories	142
The Ctype Category	143
Template Class Ctype	143
is	144
scan_is	144
scan_not	144
toupper	145
tolower	145
widen	145
narrow	146
do_is	146
do_scan_is	146
do_scan_not	147
do_toupper	147
do_tolower	147
do_widen	147
do_narrow	148
Template class ctype_byname	148
Ctype Specializations	152
Constructor	153
destructor	153
Specialized Ctype Members	154
classic_table	154
Ctype_byname<char> Constructor	155

Table of Contents

Template Class <code>Codecvt</code>	155
<code>out</code>	156
<code>unshift</code>	156
<code>in</code>	156
<code>always_noconv</code>	157
<code>length</code>	157
<code>max_length</code>	157
<code>Codecvt</code> Virtual Functions	158
The Numeric Category	163
<code>get</code>	164
<code>put</code>	168
<code>Num_put</code> Virtual Functions	169
The Numeric Punctuation Facet	169
<code>decimal_point</code>	170
<code>thousands_sep</code>	170
<code>grouping</code>	170
<code>truename</code>	170
<code>falseename</code>	171
<code>numpunct</code> virtual functions	171
Template Class <code>Numpunct_byname</code>	172
<code>decimal_point</code>	173
<code>thousands_sep</code>	173
<code>grouping</code>	173
<code>false_name</code> and <code>true_name</code>	173
<code>Numeric_wide</code>	174
The Collate Category	176
<code>compare</code>	176
<code>transform</code>	176
<code>hash</code>	177
Collate Virtual Functions	177
Template Class <code>Collate_byname</code>	177
The Time Category	186
Template Class <code>Time_get_byname</code>	195

Template Class Time_put	195
Time_put Members	196
Time_put Virtual Functions	197
The Monetary Category	210
get	217
Money_get Virtual Functions	217
put	219
Money_put Virtual Functions	219
decimal_point	221
thousands_sep	221
grouping	221
curr_symbol	222
positive_sign	222
negative_sign	222
frac_digits	222
pos_format	223
neg_format	223
Moneypunct Virtual Functions	224
Extending moneypunct by derivation	225
decimal_point	228
thousands_sep	228
grouping	228
curr_symbol	229
positive_sign	229
negative_sign	229
frac_digits	230
pos_format / neg_format	230
The Message Retrieval Category	231
open	233
get	233
close	233
Messages Virtual Functions	234
MSL C++ implementation of messages	234

Table of Contents

Program-defined Facets	240
C Library Locales	240
8 Containers Library	241
The Containers library	241
Container Requirements	241
Sequences Requirements	242
Associative Containers Requirements.	243
Sequences	244
Template Class Deque	244
Constructors	245
assign	245
resize	245
insert.	246
erase	246
swap	247
Template Class List	247
Constructors	247
assign	248
resize	248
insert.	249
push_front	249
push_back.	249
erase	250
pop_front	250
pop_back	250
clear	250
splice	251
remove	251
remove_if	251
unique	252
merge	252
reverse	252

sort	253
swap	253
Container Adaptors	253
Template Class Queue	253
operator ==	254
operator <	254
Template Class Priority_queue.	254
Constructors	255
push	255
pop	255
Template Class Stack	256
Constructors	256
empty	256
size	256
top.	257
push	257
pop	257
Template Class Vector	257
Constructors	258
assign	258
capacity	259
resize	259
insert.	259
erase	260
swap	260
Class Vector<bool>	260
Associative Containers	260
Template Class Map.	261
Constructors	261
operator []	261
find	262
lower_bound.	262
upper_bound.	263

Table of Contents

equal_range	263
swap	264
Template Class Multimap	264
Constructors	265
find	265
lower_bound	266
equal_range	266
swap	267
Template Class Set	267
Constructors	268
swap	268
Template Class Multiset	268
Constructors	269
swap	269
Template Class Bitset	270
Template Class Bitset	270
Constructors	271
operator &=	271
operator =	272
operator ^=	272
operator <<=	272
operator >>=	272
Set	273
reset	273
operator ~	274
flip	274
to_ulong	274
to_string	275
count	275
size	275
operator ==	276
operator !=	276
test	276

any	276
none	277
operator <<	277
operator >>	277
operator &	278
operator 	278
operator ^	278
operator >>	279
operator <<	279
9 Iterators Library	281
The Iterators library	281
Requirements	282
Input Iterators	282
Output Iterators	282
Forward Iterators	282
Bidirectional Iterators	282
Random Access Iterators	283
Header <iterator>	283
Iterator Primitives	283
Iterator Traits	283
Basic Iterator	284
Standard Iterator Tags	284
advance	284
distance	285
Predefined Iterators	285
Reverse iterators	285
Constructors	286
base	286
Reverse_iterator operators	286
Insert Iterators	290
Constructors	290
Back_insert_iterator Operators	290

Table of Contents

back_inserter	291
Template Class Front_insert_iterator	291
Constructors	291
Front_insert_iterator operators.	292
front_inserter	292
Template Class Insert_iterator	293
Constructors	293
Insert_iterator Operators	293
inserter	294
Stream Iterators.	294
Template Class Istream_iterator	294
Constructors	294
destructor	295
Istream_iterator Operations.	295
Template Class Ostream_iterator	296
Constructors	296
destructor	296
Ostream_iterator Operators	297
Template Class Istreambuf_iterator	297
Constructors	297
Istreambuf_iterator Operators	298
equal	299
Template Class Ostreambuf_iterator	299
Constructors	299
Ostreambuf_iterator Operators.	299
failed.	300
10 Algorithms Library	301
The Algorithms Library	301
Header <algorithm>	301
Non-modifying Sequence Operations	301
for_each	302
find	302

find_if	302
find_end	303
find_first_of	304
adjacent_find	305
count	305
count_if	306
mismatch	306
equal	307
search	308
search_n	309
Mutating Sequence Operators	309
copy	310
copy_backward	310
swap	310
swap_ranges	311
iter_swap	311
transform	312
replace	313
replace_copy	313
replace_copy_if	314
fill	314
fill_n	315
generate	315
generate_n	316
remove	316
remove_if	316
remove_copy	317
remove_copy_if	317
unique	318
unique_copy	318
reverse	319
reverse_copy	319
rotate	320

Table of Contents

rotate_copy	320
random_shuffle.	321
partition	321
stable_partition.	322
Sorting And Related Operations	322
sort	323
stable_sort	324
partial_sort	325
partial_sort_copy	326
nth_element	327
lower_bound.	328
upper_bound.	329
equal_range	330
binary_search	331
merge	332
inplace_merge	333
includes	334
set_union	335
set_intersection.	336
set_difference	337
set_symmetric_difference	338
push_heap.	339
pop_heap	339
make_heap	340
sort_heap	341
min	341
max	342
min_element.	343
max_element	344
lexicographical_compare	345
next_permutation	346
prev_permutation	347
C library algorithms	347

bsearch	347
qsort	348
11 Numerics Library	351
The Numerics Library (clause 26)	351
Numeric type requirements	351
Numeric arrays	352
Template Class Valarray	352
Constructors	353
Destructor	353
Assignment Operator	354
operator[]	354
operator[]	355
valarray unary operators	355
Valarray Computed Assignment	356
size	358
sum	358
min	358
max	359
shift	359
cshift	359
apply	360
resize	360
Valarray Non-member Operations	360
Valarray Logical Operators	365
Non-member logical operations	367
Class slice	372
Constructors	372
start	373
size	373
stride	373
Template Class Slice_array	373
Constructors	374

Table of Contents

Assignment Operator	374
slice_array computed assignment	375
Slice_array Fill Function	375
Class Gslice	375
Constructors	376
start	376
size	376
stride	377
Template Class Gslice_array	377
Constructors	377
Assignment Operators	377
Gslice_array Computed Assignment	378
Fill Function	378
Template Class Mask_array	378
Constructors	379
Assignment Operators	379
Mask_array Computed Assignment	380
Mask_array Fill Function	380
Template Class Indirect_array	380
Constructors	381
Assignment Operators	381
Indirect_array Computed Assignment	382
indirect_array fill function	382
Generalized Numeric Operations	382
Header <numeric>	383
accumulate	383
inner_product	384
partial_sum	385
adjacent_difference	386
C Library	387
<cmath>	387
<cstdlib>	387

12 Complex Class	391
The Complex Class Library	391
<code>_MSL_CX_LIMITED_RANGE</code>	391
Header <complex>	392
Header <complex> forward declarations	392
Complex Specializations	392
Complex Template Class	392
Constructors and Assignments	393
Constructors	393
Complex Member Functions	393
<code>real</code>	394
<code>imag</code>	394
Complex Class Operators	394
Overloaded Operators and Functions	396
Overloaded Complex Operators	396
Complex Value Operations	401
<code>real</code>	402
<code>imag</code>	402
<code>abs</code>	402
<code>arg</code>	402
<code>norm</code>	403
<code>conj</code>	403
<code>polar</code>	403
Complex Transcendentals	404
<code>cos</code>	404
<code>cosh</code>	404
<code>exp</code>	405
<code>log</code>	405
<code>log10</code>	405
<code>pow</code>	406
<code>sin</code>	406
<code>sinh</code>	407
<code>sqrt</code>	407

Table of Contents

tan	407
tanh	408
13 Input and Output Library	409
The Input and Output Library	409
Input and Output Library Summary	409
Iostreams requirements	410
Definitions	410
Type requirements	411
Type SZ_T	411
14 Forward Declarations	413
The Streams and String Forward Declarations	413
Header <iosfwd>	413
Header <stringfwd>	414
15 Iostream Objects	417
The Standard Input and Output Stream Library	417
Header <iostream>	417
Stream Buffering	418
Narrow stream objects	418
istream cin	418
ostream cout	418
ostream cerr	419
ostream clog	419
Wide stream objects	419
wistream wcin	420
wostream wcout	420
wostream wcerr	420
wostream wlcout	421
16 Iostreams Base Classes	423
Input and Output Stream Base Library	423
Header <ios>	423

Template Class fpos	423
TypeDef Declarations	424
Class ios_base	424
Typedef Declarations	425
Class ios_base::failure	425
failure	425
failure::what	425
Type fmtflags	425
Type iostate	427
Type openmode	427
Type seekdir	428
Class Init	428
Class Init Constructor	428
Destructor	428
ios_base fmtflags state functions	429
flags	429
setf	432
unsetf	433
precision	434
width	435
ios_base locale functions	436
imbue	436
getloc	437
ios_base storage function	437
xalloc	437
iword	437
pword	438
register_callback	438
sync_with_stdio	439
ios_base Constructor	439
Ios_base Constructor	439
Destructor	439
Template class basic_ios	440

Table of Contents

basic_ios Constructor	440
Destructor.	440
Basic_ios Member Functions	441
tie	441
rdbuf.	443
imbue	444
fill.	445
copyfmt.	446
basic_ios iostate flags functions	446
operator bool	446
operator !	446
rdstate	447
clear	449
setstate	450
good	451
eof.	451
fail.	453
bad	454
exceptions.	455
ios_base manipulators	456
fmtflags manipulators	456
adjustfield manipulators	457
basefield manipulators	458
floatfield manipulators	458
Overloading Manipulators	459
17 Stream Buffers	461
The Stream Buffers Library	461
Stream buffer requirements	461
Class basic_streambuf<charT, traits>	462
basic_streambuf Constructor	463
Destructor.	463
basic_streambuf Public Member Functions	463

Locales	463
basic_streambuf::pubimbue	464
basic_streambuf::getloc	464
Buffer Management and Positioning	464
basic_streambuf::pubsetbuf	464
basic_streambuf::pubseekoff	466
basic_streambuf::pubseekpos	467
basic_streambuf::pubsync	469
Get Area	470
basic_streambuf::in_avail	470
basic_streambuf::snextc	470
basic_streambuf::sbumpc	471
basic_streambuf::sgetc	472
basic_streambuf::sgetn	473
Putback	473
basic_streambuf::sputback	473
basic_streambuf::sungetc	475
Put Area	475
basic_streambuf::sputc	475
basic_streambuf::sputn	476
basic_streambuf Protected Member Functions	477
basic_streambuf::eback	477
basic_streambuf::gptr	477
basic_streambuf::egptr	477
basic_streambuf::gbump	478
basic_streambuf::setg	478
Put Area Access	478
basic_streambuf::pbase	479
basic_streambuf::pptr	479
basic_streambuf::eptr	479
basic_streambuf::pbump	479
basic_streambuf::setp	480
basic_streambuf Virtual Functions	480

Table of Contents

basic_streambuf::imbue	480
Buffer Management and Positioning	481
basic_streambuf::setbuf	481
basic_streambuf::seekoff	481
basic_streambuf::seekpos	482
basic_streambuf::sync	482
Get Area	483
basic_streambuf::showmanc	483
basic_streambuf::xsgetn	483
basic_streambuf::underflow	484
basic_streambuf::uflow	484
Putback	485
basic_streambuf::pbackfail	485
Put Area	485
basic_streambuf::xsputn	485
basic_streambuf::overflow	486
18 Formatting Manipulators	487
The Formatting and Manipulators Library	487
Input Streams	487
Template class basic_istream	488
basic_istream Constructors	488
Destructor	488
Class basic_istream::sentry	489
Class basic_istream::sentry Constructor	490
Destructor	490
sentry::Operator bool	490
Formatted input functions	490
Arithmetic Extractors Operator >>	491
basic_istream extractor operator >>	492
Overloading Extractors:	495
Unformatted input functions	497
basic_istream::gcount	497

basic_istream::get	499
basic_istream::getline	501
basic_istream::ignore	503
basic_istream::peek	505
basic_istream::read	505
basic_istream::readsome	507
basic_istream::putback	509
basic_istream::unget	510
basic_istream::sync	512
basic_istream::tellg	513
basic_istream::seekg	514
Standard basic_istream manipulators	515
basic_ifstream::ws	516
basic_iostream Constructor	517
Destructor	518
Output streams	518
Template class basic_ostream	518
basic_ostream Constructor	519
Destructor	519
Class basic_ostream::sentry	520
Class basic_ostream::sentry Constructor	520
Destructor	521
sentry::Operator bool	521
Formatted output functions	521
Arithmetic Inserter Operator <<	522
basic_ostream::operator<<	524
Overloading Inserters	526
Unformatted output functions	527
basic_ostream::tellop	527
basic_ostream::seekp	528
basic_ostream::put	530
basic_ostream::write	530
basic_ostream::flush	532

Table of Contents

Standard basic_ostream manipulators	534
basic_ostream:: endl.	535
basic_ostream::ends	535
basic_ostream::flush.	536
Standard manipulators	539
Standard Manipulator Instantiations	539
resetiosflags	539
setiosflags.	540
:setbase	541
setfill.	542
setprecision	543
setw	544
Overloaded Manipulator	545
19 String Based Streams	549
The String Based Stream Library	549
Header <sstream>	549
Template class basic_stringbuf.	550
basic_stringbuf constructors.	550
Member functions	551
basic_stringbuf::str	551
Overridden virtual functions	553
basic_stringbuf::underflow	553
basic_stringbuf::pbackfail	553
basic_stringbuf::overflow.	554
basic_stringbuf::seekoff	554
basic_stringbuf::seekpos	555
Template class basic_istringstream	555
basic_istringstream Constructor	556
Member functions	557
basic_istringstream::rdbuf	557
basic_istringstream::str	558
Class basic_ostringstream.	559

basic_ostringstream Constructor	560
Member functions	561
basic_ostringstream::rdbuf	561
basic_ostringstream::str	563
Class basic_stringstream	564
basic_stringstream Constructor	565
Member functions	566
basic_stringstream::rdbuf	566
basic_stringstream::str	567
20 File Based Streams	569
The File Based Streams Library	569
Header <fstream>	569
File Streams Type Defines	569
Template class basic_filebuf	570
basic_filebuf Constructors	570
Destructor	571
Member functions	571
basic_filebuf::is_open	571
basic_filebuf::open	572
basic_filebuf::close	574
Overridden virtual functions	574
basic_filebuf::showmany	574
basic_filebuf::underflow	575
basic_filebuf::pbackfail	575
basic_filebuf::overflow	575
basic_filebuf::seekoff	576
basic_filebuf::seekpos	576
basic_filebuf::setbuf	577
basic_filebuf::sync	577
basic_filebuf::imbue	577
Template class basic_ifstream	578
basic_ifstream Constructor	578

Table of Contents

Member functions	579
basic_ifstream::rdbuf	580
basic_ifstream::is_open	581
basic_ifstream::open	581
basic_ifstream::close	583
Template class basic_ofstream	583
basic_ofstream Constructors	584
Member functions	585
basic_ofstream::rdbuf	585
basic_ofstream::is_open	587
basic_ofstream::open	587
basic_ofstream::close	589
Template class basic_fstream	589
basic_fstream Constructor	590
Member Functions	591
basic_fstream::rdbuf	591
basic_fstream::is_open	592
basic_fstream::open	593
basic_fstream::close	594
21 C Library Files	597
The C Library Files	597
22 Strstream	599
The Strstream Class Library (Annex D)	599
Header <strstream>	599
Strstreambuf Class	600
Strstreambuf constructors and Destructors	600
Constructors	600
Destructor	601
Strstreambuf Public Member Functions	602
freeze	602
pcount	603

str	604
Protected Virtual Member Functions	605
setbuf	605
seekoff	606
seekpos	606
underflow	607
pbackfail	607
overflow	608
Istrstream Class.	608
Constructors and Destructor.	609
Constructors	609
Destructor.	610
Public Member Functions	610
rdbuf.	610
str	611
Ostrstream Class	611
Constructors and Destructor.	612
Constructors	612
Destructor.	613
Public Member Functions	613
freeze	613
pcount	614
rdbuf.	615
str	616
Strstream Class	616
Strstream Types	616
Constructors and Destructor.	616
Constructors	617
Destructor.	617
Public Member Functions	617
freeze	617
pcount	618
rdbuf.	618

Table of Contents

str	618
23 Msl_mutex.h 621	
The Msl_mutex Support Library	621
Header <msl_mutex.h>.	621
Mutex	621
Constructors and Assignment Operator	622
Operator=	622
Destructor.	623
Lock	623
Unlock	623
Mutex_lock	624
Constructors and Assignment Operator	624
Operator=	624
Destructor.	625
24 Bitvector Class Library 627	
The bitvector Class Library	627
Bitvector types	630
iterators.	631
reference class members	631
Constructors, Destructors and Assignment Operators	632
constructors	632
Destructor.	633
Allocators	633
get_allocator.	633
max_size	633
Member Functions	634
size	634
empty	634
capacity	634
reserve	635
assign	635

begin	635
end	636
rbegin	636
rend	636
operator[]	637
at	637
resize	637
front	638
back	638
push_back	638
pop_back	639
insert	639
erase	639
clear	640
swap	640
flip	640
invariants	641
Operators	641
Operator ==	641
Operator <	641
Operator !=	642
Operator >	642
Operator >=	642
Operator <=	643
25 MSL_Utility	645
The MSL Utilities Library	645
The <msl_utility> Header	645
Basic Compile-Time Transformations	646
remove_const	646
remove_volatile	646
remove_cv	647
remove_pointer	647

Table of Contents

remove_reference	648
remove_bounds	648
remove_all	649
Type Query	649
is_same	649
CV Query	650
is_const	650
is_volatile	650
Type Classification	650
is_signed / is_unsigned	652
POD classification	653
Miscellaneous	654
compile_assert	654
array_size	655
can_derive_from	655
store_as - Container optimization	655
call_traits	657
is_empty	658
compressed_pair	658
alloc_ptr	660

26 MSL C++ Debug Mode	661
Overview of MSL C++ Debug Mode	661
Types of Errors Detected	661
How to Enable Debug Mode	661
Debug Mode Implementations	662
Debug Mode Containers	664
deque	665
list	666
string	667
vector	668
tree-based containers - map, multimap, set, multiset	669
cdequeue	669

slist	670
hash-based containers - map, multimap, set, multiset	671
Invariants	672
27 Hash Libraries	675
The Hash Containers Library	675
General Hash Issues	675
Introduction	675
Namespace Issues.	676
Fully Qualified Reference:	676
Namespace Alias	676
Using Declaration.	677
Using Directive.	678
Compatibility Headers	678
Constructors	679
Iterator Issues	680
Capacity	680
insert.	682
insert.	683
erase	684
Observers	684
Set Operations	685
Global Methods	685
Incompatibility with Previous versions Metrowerks Hash Containers	686
Hash_set	686
Introduction..	687
Hash_map	689
Introduction	689
Old Hashmap Headers	689
Hash_fun	692
28 Metrowerks::threads	695
Overview of Metrowerks Threads Library	695

Table of Contents

Mutex and Locks	696
Threads	701
Condition Variables	704
call_once	710
thread_specific_ptr	712
29 Mslconfig	715
C++ Switches, Flags and Defines	715
__CSTD	716
__Inhibit_Container_Optimization	716
__Inhibit_Optimize_RB_bit	717
__MSL_DEBUG	717
__msl_error	717
__MSL_ARRAY_AUTO_PTR	718
__MSL_CFILE_STREAM	718
__MSL_CPP__	718
__MSL_EXTENDED_BINDERS	718
__MSL_EXTENDED_PRECISION_OUTP	720
__MSL_FORCE_ENABLE_BOOL_SUPPORT	720
__MSL_FORCE_ENUMS_ALWAYS_INT	721
__MSL_IMP_EXP	722
__MSL_LONGLONG_SUPPORT__	722
__MSL_MINIMUM_NAMED_LOCALE	723
__MSL_MULTITHREAD	723
__MSL_NO_BOOL	723
__MSL_NO_CONSOLE_IO	724
__MSL_NO_CPP_NAMESPACE	724
__MSL_NO_EXCEPTIONS	724
__MSL_NO_EXPLICIT_FUNC_TEMPLATE_ARG	725
__MSL_NO_FILE_IO	725
__MSL_NO_IO	726
__MSL_NO_LOCALE	726
__MSL_NO_REFCOUNT_STRING	726

_MSL_NO_VECTOR_BOOL	726
_MSL_NO_WCHART	727
_MSL_NO_WCHART_LANG_SUPPORT	727
_MSL_NO_WCHART_C_SUPPORT	727
_MSL_NO_WCHART_CPP_SUPPORT	727
_MSL_POSIX_STREAM	728
_MSL_WIDE_FILENAME	728
_MSL_WFILEIO_AVAILABLE	728
_MSL_USE_AUTO_PTR_96	729
_STD	729

Table of Contents

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.

About the MSL C++ Library Reference Manual

This section describes each chapter in this manual. The various chapter's layout mirrors that of the ISO (the International Organization for Standardization) C++ Standard.

[“The MSL C++ Library Overview.”](#) of this manual describes the language support library that provides components that are required by certain parts of the C++ language, such as memory allocation and exception processing.

[“The Language Support Library.”](#) discusses the ANSI/ISO language support library.

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

[“The General Utilities Library.”](#) discusses the general utilities library, which includes components used by other library elements, such as predefined storage allocator for dynamic storage management.

[“The Strings Library.”](#) 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.

[“The Localization library.”](#) covers the localization components extend internationalization support for character classification, numeric, monetary, and date/time formatting and parsing among other things.

Introduction

About the MSL C++ Library Reference Manual

“[The Containers library](#),” 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.

“[The Iterators library](#)” discusses iterator classes.

“[The Algorithms Library](#),” discusses the algorithms library. This library provides sequence, sorting, and general numerics algorithms.

“[The Numerics Library \(clause 26\)](#),” discusses the numerics library. It describes numeric arrays, generalized numeric algorithms and facilities included from the ISO C library.

“[The Complex Class Library](#),” describes the components for complex number types

“[The Input and Output Library](#),” overviews the input and output class libraries.

“[The Streams and String Forward Declarations](#),” discusses the input and output streams forward declarations.

“[The Standard Input and Output Stream Library](#),” discusses the initialized input and output objects.

“[Input and Output Stream Base Library](#),” discusses the iostream_base class.

“[The Stream Buffers Library](#),” discusses the stream buffer classes.

“[The Formatting and Manipulators Library](#),” discusses the formatting and manipulator classes.

“[The String Based Stream Library](#),” discusses the string based stream classes.

“[The File Based Streams Library](#),” discusses the file based stream classes.

“[The C Library Files](#),” discusses the namespace C Library functions.

“[The Strstream Class Library \(Annex D\)](#),” discusses the non standard string stream classes.

“[The Msl_mutex Support Library](#),” discusses the mutex classes.

“[The bitvector Class Library](#)” discusses the bool vector class library.

“[The MSL Utilities Library](#)” utilities used for non standard headers.

“[Overview of MSL C++ Debug Mode](#)” describes the Metrowerks Standard C++ library debug mode facilities.

“[The Hash Containers Library](#)” nonstandard “hash” libraries.

“[C++ Switches, Flags and Defines](#)” is a chapter on the various flags that you can use to create a customized version of the MSL C++ Library

Introduction

About the MSL C++ Library Reference Manual

The C++ Library

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

The MSL C++ Library Overview

This section introduces you to the definitions, conventions, terminology, and other aspects of the MSL C++ library.

The chapter is constructed in the following sub sections and mirrors clause 17 of the ISO (the International Organization for Standardization) C++ Standard :

- [“Definitions”](#) standard C++ terminology
- [“Additional Definitions”](#) additional terminology
- [“Multi-Thread Safety”](#) multi-threaded policy
- [“Methods of Descriptions”](#) standard conventions
- [“Library-wide Requirements”](#) library requirements

Definitions

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

- [“Arbitrary-Positional Stream”](#)
- [“Character”](#)
- [“Character Sequences”](#)
- [“Comparison Function”](#)
- [“Component”](#)
- [“Default Behavior”](#)
- [“Handler Function”](#)
- [“Iostream Class Templates”](#)
- [“Modifier Function”](#)

- “[Object State](#)”
- “[Narrow-oriented Iostream Classes](#)”
- “[NTCTS](#)”
- “[Observer Function](#)”
- “[Replacement Function](#)”
- “[Required Behavior](#)”
- “[Repositional Stream](#)”
- “[Reserved Function](#)”
- “[Traits](#)”
- “[Wide-oriented Iostream Classes](#)”

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

Character

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

Character Sequences

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

Comparison Function

An operator function for equality or relational operators.

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 referred to as a component.

Default Behavior

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

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.

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.

Modifier Function

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

Object State

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

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.

NTCTS

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

Observer Function

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

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.

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.

Repositional Stream

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

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.

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.

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.

Additional Definitions

Metrowerks Standard Libraries have additional definitions.

- .“[Multi-Thread Safety](#)” 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 string class objects (via an internal refcount), then only one string 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.

MSL C++ Thread Safety Policy

MSL C++ is Level-1 thread safe. That is:

- It is safe to simultaneously call const and non-const methods from different threads to distinct objects.
- It is safe to simultaneously call const methods, and methods otherwise guaranteed not to alter the state of an object (or invalidate outstanding references and iterators of a container) from different threads to the same object.
- It is not safe for different threads to simultaneously access the same object when at least one thread calls non-const methods, or methods that invalidate outstanding references or iterators to the object. The programmer is responsible for using thread synchronization primitives (e.g. mutex) to avoid such situations.

Simultaneous use of allocators such as new and malloc are thread safe.

Simultaneous use of global objects such as cin and cout is not safe. The programmer is responsible for using thread synchronization primitives to avoid such situations. MSL C++ provides an extension to standard C++ (std::mutex) to aid in such code. For example:

Listing 2.1 MSL Mutex Example

```
#include <iostream>
#include <iomanip>
#include <mutex.h>

std::mutex cout_lock;
```

```
int main()
{
    cout_lock.lock();
    std::cout << "The number is " <<
    std::setw(5) << 20 << '\n';
    cout_lock.unlock();
}
```

Note that if only one thread is accessing a standard stream then no synchronization is necessary. For example, one could have one thread handling input from cin, and another thread handling output to cout, without worrying about mutex objects.

The thread safety of MSL C++ can be controlled by the flag

[_MSL_MULTITHREAD](#) in <mslconfig>. If you explicitly use std::mutex objects in your code, then they will become empty do-nothing objects when multi-threading is turned off (_MSL_MULTITHREAD is undefined). Thus the same source can be used in both single thread and multi-thread projects.

The _MSL_MULTITHREAD flag causes some mutex objects to be set up in the library internally to protect data that is not obviously shared. .

See [_MSL_MULTITHREAD](#) for a fuller description.

Methods of Descriptions

Conventions used to describe the C++ Standard Library.

Structure of each sub-clause

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

Table 2.1 Chapter Descriptions

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

Other Conventions

Some other terminology and conventions used in this reference are:

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.

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.
- The 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.

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.

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.

Functions within classes

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

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
```

Library-wide Requirements

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

- [“Library contents and organization”](#)
- [“Using the library”](#)
- [“Constraints on programs”](#)
- [“Conforming Implementations”](#)
- [“Reentrancy”](#)

Library contents and organization

The Metrowerks Standard Libraries are organized in the same fashion as the ANSI/ISO C++ Standard.

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 `namespace` nested within namespace std.

Headers

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

Table 2.2 MSL C++ Library headers:

C++	Headers	C++	Headers
<code><algorithm></code>	<code><bitset></code>	<code><complex></code>	<code><deque></code>
<code><exception></code>	<code><fstream></code>	<code><functional></code>	<code><iomanip></code>
<code><ios></code>	<code><iostream></code>	<code><iostream></code>	<code><istream></code>
<code><iterator></code>	<code><limits></code>	<code><list></code>	<code><locale></code>
<code><map></code>	<code><memory></code>	<code><new></code>	<code><numeric></code>
<code><ostream></code>	<code><queue></code>	<code><set></code>	<code><sstream></code>
<code><stack></code>	<code><stdexcept></code>	<code><streambuf></code>	<code><string></code>
<code><typeinfo></code>	<code><utility></code>	<code><valarray></code>	<code><vector></code>
C Style	Headers	C Style	Header
<code><cassert></code>	<code><cctype></code>	<code><cerrno></code>	<code><cfloat></code>
<code><ciso646></code>	<code><climits></code>	<code><clocale></code>	<code><cmath></code>
<code><csetjmp></code>	<code><csignal></code>	<code><cstdarg></code>	<code><cstddef></code>
<code><cstdio></code>	<code><cstdlib></code>	<code><cstring></code>	<code><ctime></code>
<code><cwchar></code>	<code><cwctype></code>		

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.

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
<cstdint>	Types
<limits>	Implementation properties
<cstdlib>	Start and termination
<new>	Dynamic memory management
<typeinfo>	Type identification
<exception>	Exception handling
<cstdarg>	Other runtime support

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

Using the library

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

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.

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.

Constraints on programs

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

Reserved Names

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

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

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

User code can safely use macros that are all uppercase characters and underscores, except for leading underscores. Library code will either be in namespace std or in namespace Metrowerks. Implementation details in namespace std will be prefixed by a double underscore or an underscore followed by an uppercase character. Implementation details in namespace Metrowerks are nested in a nested namespace, for example:

Metrowerks::details.

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.

Headers

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

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.

Replacement Functions

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

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

Listing 2.2 Dynamic Memory Allocators

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

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`

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.

Function arguments

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

Conforming Implementations

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

Reentrancy

In Metrowerks Standard Library, Multi-Threaded safety, as describe in the section [“Multi-Thread Safety”](#) is a driving force in the design of this efficient C++ library

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 C Style 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`.

Language Support Library

This chapter includes the implicit functions and objects generated during the execution of some C++ programs. It also contains the headers for those function, objects and defined types.

The Language Support Library

The chapter is constructed in the following sub sections and mirrors clause 18 of the ISO (the International Organization for Standardization) C++ Standard :

- “[Types](#)” covers predefined types
- “[Implementation properties](#)” covers implementation defined properties
- “[Start and termination](#)” covers functions used for starting and termination of a program
- “[Dynamic Memory Management](#)” covers operators used for dynamic allocation and release of memory.
- “[Type identification](#)” covers objects and functions used for runtime type identification.
- “[Exception Handling](#)” covers objects and functions used for exception handling and errors in exception handling.
- “[Other Runtime Support](#)” covers variations from standard C library support functions.

Types

The header <cstddef> contains the same types and definitions as the standard C stddef.h with the following changes as show in “[Header <cstddef>](#)”.

Table 3.1 Header <cstddef>

NULL	The macro NULL is an implementation-defined C++ constant value. MSL defines this as 0L.
offsetof	This macro accepts a restricted set of type arguments that shall be a POD structure or a POD union. The result of applying the offsetof macro to a field that is a static data member or a function member is undefined.
ptrdiff_t	No change from standard C. An signed integral type large enough to hold the difference between two pointers.
size_t	No change from standard C. An unsigned integral type large enough to hold the result of the sizeof operator.

Implementation properties

The headers <limits>, <climits>, and <cfloat> supply implementation dependent characteristics for fundamental types.

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 including floating point and integer types are provided.

- The member `is_specialized` shall be true for specializations of `numeric_limits`.
- Members declared static const in the `numeric_limits` template, specializations are usable as integral constant expressions.
- Non-fundamental standard types, do not have specializations.

`Numeric_limits` Static Members

All static members shall be provided but they do not need to be used.

is_specialized

The data member for distinguishing specializations. The default value is false.

```
static const bool is_specialized = false;
```

min

```
static T min() throw();
```

The maximum positive normalized value is returned.

max

The minimum finite value for floating point types with denormalization.

```
static T max() throw();
```

Remarks

The maximum positive normalized value is returned.

digits

Designates the number of non-signed digits that can be represented for integral types.
The number of radix digits in the mantissa for floating point types

```
static const int digits = 0;
```

is_signed

True if the number is signed.

```
static const bool is_signed = false;
```

is_integer

True if the number is an integer.

```
static const bool is_integer = false;
```

is_exact

True if the number is exact.

```
static const bool is_exact = false;
```

Remarks

All integer types are exact, but not all floating point types are exact.

radix

Specifies the base or radix of the exponent of a floating point type or base of an integral type.

```
static const int radix = 0;
```

epsilon

The difference between 1 and the least value greater than 1.

```
static T epsilon() throw();
```

round_error

A function to measure the rounding error.

```
static T round_error() throw();
```

Remarks

Returns the maximum rounding error.

min_exponent

Holds the minimum exponent so that the radix raised to one less than this would be normalized.

```
static const int min_exponent;
```

min_exponent10

Stores the minimum negative exponent that 10 raised to that power would be a normalized floating point type.

```
static const int min_exponent10 = 0;
```

max_exponent

The maximum positive integer so that the radix raised to the power one less than this is representable.

```
static const int max_exponent = 0;
```

max_exponent10

The maximum positive integer so that the 10 raised to this power is representable.

```
static const int max_exponent10 = 0;
```

has_infinity

True if is positive for infinity.

```
static const bool has_infinity = false;
```

has_quiet_NaN

True if the number has a quiet “Not a Number”.

```
static const bool has_quiet_NaN = false;
```

has_signaling_NaN

True if the number is a signaling “Not a Number”.

```
static const bool has_signaling_NaN = false;
```

has_denorm

Distinguishes if the floating point number has the ability to be denormalized.

```
static const float_denorm_style  
has_denorm = denorm_absent;
```

Remarks

The static variable has_denorm equals denorm_present if the type allows denormalized values or denorm_absent if the type does not and denorm_ineterminate if it is indeterminate

has_denorm_loss

Is true if there is a loss of accuracy because of a denormalization loss.

```
static const bool has_denorm_loss = false;
```

infinity

Determines a positive infinity.

```
static T infinity() throw();
```

Remarks

Returns a positive infinity if available.

quiet_NaN

Determines if there is a quiet “Not a Number”.

```
static T quiet_NaN() throw();
```

Remarks

Returns a quiet “Not a Number” if available.

signaling_NaN

Determines if there is a signaling “Not a Number”.

```
static T signaling_NaN() throw();
```

Remarks

Returns a signaling “Not a Number” if available.

denorm_min

Determines the minimum positive denormalized value.

```
static T denorm_min() throw();
```

Remarks

Returns the minimum positive denormalized value.

is_iec559

The values is true if and only if the type adheres to IEC 559 standard

```
static const bool is_iec559 = false;
```

is_bounded

The value is true if the set of values representable by the type is finite.

```
static const bool is_bounded = false;
```

Remarks

All predefined data types are bounded.

is_modulo

This value is 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.

```
static const bool is_modulo = false;
```

Remarks

This value is generally true for unsigned integral types and false for floating point types.

traps

The value is true if trapping is implemented for the type.

```
static const bool traps = false;
```

tinyness_before

This value is true if tinyness is detected before rounding.

```
static const bool tinyness_before = false;
```

round_style

This value is the rounding style as a type float_round_style.

```
static const float_round_style round_style = round_toward_zero;
```

Remarks

See Also [“Floating Point Rounding Styles”](#)

Type float_round_style

An enumerated type in std namespace used to determine the characteristics for rounding floating point numbers.

Table 3.2 Floating Point Rounding Styles

Enumerated Type	Value	Meaning
round_ineterminate	-1	The rounding is indeterminable
round_toward_zero	0	The rounding is toward zero
round_to_nearest	1	Round is to the nearest value
round_toward_infinity	2	The rounding is to infinity
round_toward_neg_infinity	3	The rounding is to negative infinity

Type float_denorm_style

The presence of denormalization is represented by the std namespace enumerated type float_denorm_style.

Table 3.3 Floating Point Denorm Styles

Enumerated Type	Value	Meaning
denorm_inde determinate	-1	Denormalization is indeterminable
denorm_absent	0	Denormalization is absent
denorm_present	1	Denormalization is present

numeric_limits specializations

All member have specializations but these values are not required to be meaningful. Any value that is not meaningful is set to 0 or `false`.

C Library

The contents of `<climits>` are the same as standard C's `limits.h` and the contents of `<cfloat>` are the same as standard C's `float.h`.

Table 3.4 Header <climits>

CHAR_BIT	CHAR_MAX	CHAR_MIN	INT_MAX
INT_MIN	LONG_MAX	LONG_MIN	MB_LEN_MAX
SCHAR_MAX	SCHAR_MIN	SHRT_MAX	SHRT_MIN
UCHAR_MAX	UINT_MAX	ULONG_MAX	USHRT_MAX

The header `<cfloat>` is the same as standard C `float.h`

Table 3.5 Header <cfloat>

DBL_DIG	DBL_EPSILON	DBL_MANT_DIG
DBL_MAX	DBL_MAX_10_EXP	DBL_MAX_EXP
DBL_MIN	DBL_MIN_10_EXP	DBL_MIN_EXP
FLT_DIG	FLT_EPSILON	FLT_MANT_DIG
FLT_MAX	FLT_MAX_10_EXP	FLT_MAX_EXP

Table 3.5 Header <cfloat>

FLT_MIN	FLT_MIN_10_EXP	FLT_MIN_EXP
FLT_RADIX	FLT_ROUNDS	LDBL_DIG
LDBL_EPSILON	LDBL_MANT_DIG	LDBL_MAX
LDBL_MAX_10_EXP	LDBL_MAX_EXP	LDBL_MIN
LDBL_MIN_10_EXP	LDBL_MIN_EXP	

Start and termination

The header <cstdlib> has the same functionality as the standard C header stdlib.h in regards to start and termination functions except for the functions and macros as described below.

Table 3.6 Start and Termination Differences

Macro	Value	Meaning
EXIT_FAILURE	1	This macro is used to signify a failed return
EXIT_SUCCESS	0	This macro is used to signify a successful return

The return from the `main` function is ignored on the Macintosh operating system and is return using the native event processing method on other operating systems.

abort

Terminates the Program with abnormal termination.

`abort(void)`

Remarks

The program is terminated without executing destructors for objects of automatic or static storage duration and without calling the functions passed to `atexit`.

atexit

The atexit function registers functions to be called when [exit](#) is called in normal program termination.

```
extern "C" int atexit(void (* f)(void))  
  
extern "C++" int atexit(void (* f)(void))
```

Remarks

If there is no handler for a thrown exception `terminate` is called. The registration of at least 32 functions is allowed.

- Functions registered with atexit are called in reverse order.
- A function registered with atexit before an object of static storage duration will not be called until the object's destruction.
- A function registered with atexit after an object of static storage duration is initialized will be called before the object's destruction.

The atexit() function returns zero if the registration succeeds, non zero if it fails.

exit

Terminates the program with normal cleanup actions.

```
exit(int status)
```

Remarks

The function exit() has additional behavior in order:

- Objects with static storage duration are destroyed and functions registered by calling atexit are called.
- Objects with static storage duration are destroyed in the reverse order of construction. If main contains no automatic objects control can be transferred to main if an exception thrown is caught in main.
- Functions registered with atexit are called
- All open C streams with unwritten buffered data are flushed, closed, including streams associated with cin and cout. All `tmpfile()` files are removed.

- Control is returned to the host environment.
If status is zero or EXIT_SUCCESS, a successful termination is returned to the host environment.
If status is EXIT_FAILURE, an unsuccessful termination is returned to the host environment.
Otherwise the status returned to the host environment is implementation-defined.

Dynamic Memory Management

The header <new> defines procedures for the management of dynamic allocation and error reporting of dynamic allocation errors.

Storage Allocation and Deallocation

This clause covers storage allocation and deallocation functions and error management.

Single Object Forms

Dynamic allocation and freeing of single object data types.

operator new

Dynamically allocates signable objects.

```
void* operator new (std::size_t size) throw(std::bad_alloc);  
void* operator new (std::size_t size,  
                    const std::nothrow_t&) throw();
```

Remarks

The nothrow version of new returns a `null` pointer on failure. The normal version throws a `bad_alloc` exception on error.

Returns a pointer to the allocated memory.

operator delete

Frees memory allocated with operator new.

```
void operator delete(void* ptr) throw();  
void operator delete(void* ptr, const std::nothrow_t&) throw();
```

Array Forms

Dynamic allocation and freeing of array based data types.

operator new[]

Used for dynamic allocation or array based data types.

```
void* operator new[](  
    std::size_t size) throw(std::bad_alloc);  
void* operator new[](  
    std::size_t size, const std::nothrow_t&) throw();
```

Remarks

The default operator new will throw an exception upon failure. The nothrow version will return NULL upon failure.

operator delete[]

Operator delete[] is used in conjunction with operator new[] for array allocations.

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

Placement Forms

Placement operators are reserved and may not be overloaded by a C++ program.

placement operator new

Allocates memory at a specific memory address.

```
void* operator new (std::size_t size, void* ptr) throw();  
void* operator new[] (std::size_t size, void* ptr) throw();
```

placement operator delete

The placement delete operators are used in conjunction with the corresponding placement new operators.

```
void operator delete (void* ptr, void*) throw();  
void operator delete[](void* ptr, void*) throw();
```

Storage Allocation Errors

C++ provides for various objects, functions and types for management of allocation errors.

Class Bad_alloc

A class used to report a failed memory allocation attempt.

Constructor

Constructs a `bad_alloc` object.

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

Assignment Operator

Assigns one `bad_alloc` object to another `bad_alloc` object.

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

destructor

Destroys the `bad_alloc` object.

```
virtual ~bad_alloc() throw();
```

what

An error message describing the allocation exception.

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

Remarks

Returns a null terminated byte string "bad_alloc".

type new_handler

The type of a handler function that is called by operator new or operator new[].

```
typedef void (*new_handler)();
```

Remarks

If new requires more memory allocation, the new_handler will:

- Allocate more memory and return.
- Throw an exception of type bad_alloc or bad_alloc derived class.
- Either call abort or exit.

set_new_handler

Sets the new handler function.

```
new_handler set_new_handler  
(new_handler new_p) throw();
```

Remarks

Returns zero on the first call and the previous new_handler upon further calls.

Type identification

The header <typeinfo> defines three types for type identification and type identification errors.

The three classes are:

- [Class type_info](#)
- [Class bad_cast](#)

- [Class bad_typeid](#)

Class type_info

Class type_info contains functions and operations to obtain information about a type.

operator==

Returns true if types are the same.

```
bool operator==(const type_info& rhs) const;
```

Remarks

Returns true if the objects are the type.

operator!=

Compares for inequality.

```
bool operator!=(const type_info& rhs) const;
```

Remarks

Returns true if the objects are not the same type.

before

Is true if this object precedes the argument in collation order.

```
bool before(const type_info& rhs) const;
```

Remarks

Returns true if *this precedes the argument the collation order.

name

Returns the name of the class.

```
const char* name() const;
```

Remarks

Returns the type name.

Constructors

Private constructor so copying to prevent copying of this object.

```
type_info(const type_info& rhs);
```

Assignment Operator

Private assignment to prevent copying of this object.

```
type_info& operator=(const type_info& rhs);
```

Class bad_cast

A class for exceptions thrown in runtime casting.

Constructors

Constructs an object of class bad_cast.

```
bad_cast() throw();
```

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

Assignment Operator

Copies an object of class bad_cast.

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

what

An error message describing the casting exception.

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

Remarks

Returns the null terminated byte string "bad_cast".

Class bad_typeid

Defines a type used for handling bad typeid exceptions.

Constructors

Constructs an object of class bad_typeid.

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

Assignment Operator

Copies a class bad_typeid object.

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

what

An error message describing the typeid exception.

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

Remarks

Returns the null terminated byte string "bad_typeid".

Exception Handling

The header <exception> defines types and procedures necessary for the handling of exceptions.

Class Exception

A base class for objects thrown as exceptions.

Constructors

Constructs and object of the exception class.

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

Assignment Operator

Copies an object of exception class.

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

destructor

Destroys an exception object.

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

what

An error message describing the exception.

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

Remarks

Returns the null terminated byte string "exception".

Violating Exception Specifications

Defines objects used for exception violations.

Class bad_exception

A type used for information and reporting of a bad exceptions.

Constructors

Constructs an object of class bad_exception.

```
bad_exception() throw();  
bad_exception(const bad_exception&) throw();
```

Assignment Operator

Copies an object of class `bad_exception`

```
bad_exception& operator=
(const bad_exception&) throw();
```

what

An error message describing the bad exception.

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

Remarks

Returns the null terminated byte string “`bad_exception`”.

type `unexpected_handler`

A type of handler called by the `unexpected` function.

```
typedef void (*unexpected_handler)();
```

Remarks

The `unexpected_handler` calls `terminate()`.

`set_unexpected`

Sets the unexpected handler function.

```
unexpected_handler set_unexpected
(unexpected_handler f) throw();
```

Returns the previous unexpected_handler.

unexpected

Called when a function ends by an exception not allowed in the specifications.

```
void unexpected();
```

Remarks

May be called directly by the program.

Abnormal Termination

Types and functions used for abnormal program termination.

type terminate_handler

A type of handler called by the function `terminate` when terminating an exception.

```
typedef void (*terminate_handler)();
```

Remarks

The `terminate_handler` calls `abort()`.

set_terminate

Sets the function for terminating an exception.

```
terminate_handler set_terminate  
(terminate_handler f) throw();
```

Remarks

The `terminate_handler` shall not be a null pointer.

The previous `terminate_handler` is returned.

terminate

A function called when exception handling is abandoned.

```
void terminate();
```

Remarks

Exception handling may be abandoned by the implementation or may be called directly by the program.

uncaught_exception

Determines an uncaught exception

```
bool uncaught_exception();
```

Remarks

Throwing an exception while `uncaught_exception` is true can result in a call of `terminate`.

Returns true if an exception is uncaught.

Other Runtime Support

The C++ headers `<cstdarg>`, `<csetjmp>`, `<ctime>`, `<csignal>` and `<cstdlib>` contain macros, types and functions that vary from the corresponding standard C headers.

Table 3.7 Header `<cstdarg>`

<code>va_arg</code>	A macro used in C++ Runtime support
<code>va_end</code>	A macro used in C++ Runtime support

Table 3.7 Header <cstdarg>

va_start	A macro used in C++ Runtime support
va_list	A type used in C++ Runtime support

If the second parameter of va_start is declared with a function, array, or reference type, or with a type for which there is no parameter, the behavior is undefined

Table 3.8 Header <csetjmp>

setjmp	A macro used in C++ Runtime support
jmp_buf	A type used in C++ Runtime support
longjmp	A function used in C++ Runtime support

The function longjmp is more restricted than in the standard C implementation.

Table 3.9 Header <ctime>

CLOCKS_PER_SEC	A macro used in C++ Runtime support
clock_t	A type used in C++ Runtime support
clock	A function used in C++ Runtime support

If a signal handler attempts to use exception handling the result is undefined.

Table 3.10 Header <csignal>

SIGABRT	A macro used in C++ Runtime support
SIGILL	A macro used in C++ Runtime support
SIGSEGV	A macro used in C++ Runtime support
SIG_DFL	A macro used in C++ Runtime support
SIG_IGN	A macro used in C++ Runtime support
SIGFPE	A macro used in C++ Runtime support
SIGINT	A macro used in C++ Runtime support
SIGTERM	A macro used in C++ Runtime support
SIG_ERR	A macro used in C++ Runtime support
sig_atomic_t	A macro used in C++ Runtime support

Table 3.10 Header <csignal>

raise	A type used in C++ Runtime support
signal	A function used in C++ Runtime support

NOTE All signal handlers should have C linkage.

Table 3.11 Header <cstdlib>

getenv	A function used in C++ Runtime support
system	A function used in C++ Runtime support

Diagnostics Library

This chapter describes objects and facilities used to report error conditions.

The Diagnostics library

The chapter is constructed in the following sub sections and mirrors clause 19 of the ISO (the International Organization for Standardization) C++ Standard :

- [“Exception Classes”](#)
- [“Assertions”](#)
- [“Error Numbers”](#)

Exception Classes

The library provides for exception classes for use with logic errors and runtime errors. Logic errors in theory can be predicted in advance while runtime errors can not. The header <stdexcept> predefines several types of exceptions for C++ error reporting.

There are nine exception classes.

- [“Class Logic_error”](#)
- [“Class domain_error”](#)
- [“Class Invalid_argument”](#)
- [“Class Length_error”](#)
- [“Class Out_of_range”](#)
- [“Class Runtime_error”](#)
- [“Class Range_error”](#)
- [“Class Overflow_error”](#)
- [“Class Underflow_error”](#)

Class Logic_error

The `logic_error` class is derived from the “[Class Exception](#),” and is used for exceptions that are detectable before program execution.

Constructors

```
logic_error(const string& what_arg);
```

Constructs an object of class `logic_error`. Initializes `exception::what` to the `what_arg` argument.

Class domain_error

A derived class of logic error the `domain_error` object is used for exceptions of domain errors.

Constructors

```
domain_error(const string& what_arg);
```

Constructs an object of `domain_error`. Initializes `exception::what` to the `what_arg` argument

Class Invalid_argument

A derived class of `logic_error` the `invalid_argument` is used for exceptions of invalid arguments.

Constructors

```
invalid_argument(const string& what_arg);
```

Constructs an object of class `invalid_argument`. Initializes `exception::what` to the `what_arg` argument

Class Length_error

A derived class of logic_error the length_error is use to report exceptions when an object exceeds allowed sizes.

Constructors

```
length_error(const string& what_arg);
```

Constructs an object of class length_error. Initializes exception::what to the what_arg argument

Class Out_of_range

A derived class of logic_error an object of out_of_range is used for exceptions for out of range errors.

Constructors

```
out_of_range(const string& what_arg);
```

Constructs an object of the class out_of_range. Initializes exception::what to the what_arg argument

Class Runtime_error

Derived from the ["Class Exception."](#) the runtime_error object is used to report errors detectable only during runtime.

Constructors

Constructs an object of the class runtime_error. Initializes exception::what to the what_arg argument

Class Range_error

Derived from the `runtime_error` class, an object of `range_error` is used for exceptions due to runtime out of range errors.

Constructors

```
runtime_error(const string& what_arg);
```

```
range_error(const string& what_arg);
```

Constructs an object of the class `range_error`. Initializes `exception::what` to the `what_arg` argument

Class Overflow_error

The `overflow_error` object is derived from the class `runtime_error` and is used to report arithmetical overflow errors.

Constructors

```
overflow_error(const string& what_arg);
```

Constructs an object of the class `overflow_error`. Initializes `exception::what` to the `what_arg` argument

Class Underflow_error

The class `underflow_error` is derived from the class `runtime_error` and is used to report the arithmetical underflow error.

Constructors

```
underflow_error(const string& what_arg);
```

Constructs an object of the class `underflow_error`. Initializes `exception::what` to the `what_arg` argument

Assertions

The header <cassert> provides for the assert macro and is used the same as the standard C header `assert.h`

Error Numbers

The header <cerrno> provides macros: EDOM ERANGE and errno to be used for domain and range errors reported by using the errno facility. The <cerrno> header is used the same as standard C header `errno.h`

General Utilities Libraries

This clause describes components used by other elements of the Standard C++ library. These components may also be used by C++ programs.

The General Utilities Library

The chapter is constructed in the following sub sections and mirrors clause 20 of the ISO (the International Organization for Standardization) C++ Standard :

- “[Requirements](#)”
- “[Utility Components](#)”
- “[Function objects](#)”
- “[Memory](#)”
- “[Date and Time](#)”

Requirements

This section describes the requirements for template arguments, types used to instantiate templates and storage allocators used as general utilities.

Equality Comparisons

The equality comparison operator is required. The `(==)` expression has a `bool` return type and specifies that for `x == y` and `y == z` that `x` will equal `z`. In addition the reciprocal is also true. That is, if `x == y` then `y` equals `x`. Also if `x == y` and `y == z` then `z` will be equal to `x`.

Less Than Comparison

A less than operator is required. The `(<)` expression has a `bool` return type and states that if `x < y` that `x` is less than `y` and that `y` is not less than `x`.

Copy Construction

A copy constructor for the general utilities library has the following requirements:

- If the copy constructor is `TYPE(t)` then the argument must be an equivalent of `TYPE`.
- If the copy constructor is `TYPE(const t)` then the argument must be the equivalent of `const TYPE`.
- `&T`, denotes the address of `T`.
- `&const T`, denotes the address of `const T`.

Default Construction

A default constructor is not necessary. However, some container class members may specify a default constructor as a default argument. In that case when a default constructor is used as a default argument there must be a default constructor defined.

Allocator Requirements

The general utilities library requirements include requirements for allocators. Allocators are objects that contain information about the container. This includes information concerning pointer types, the type of their difference, the size of objects in this allocation, also the memory allocation and deallocation information. All of the standard containers are parameterized in terms of allocators.

The allocator class includes the following members

Table 5.1 Allocator Members

Expression	Meaning
<code>pointer</code>	A pointer to a type
<code>const_pointer</code>	A pointer to a const type
<code>reference</code>	A reference of a type
<code>const_reference</code>	A reference to a const type
<code>value_type</code>	A type identical to the type
<code>size_type</code>	An unsigned integer that can represent the largest object in the allocator

Table 5.1 Allocator Members

Expression	Meaning
difference_type	A signed integer that can represent the difference between any two pointers in the allocator
rebind	The template member is effectively a typedef of the type to which the allocator is bound
address(type)	Returns the address of type
address(const type)	Returns the address of the const type
allocate(size)	Returns the allocation of size
allocate(size, address)	Returns the allocation of size at the address
max_size	The largest value that can be passed to allocate
Ax == Ay	Returns a bool true if the storage of each allocator can be deallocated by the other
Ax != Ay	Returns a bool true if the storage of each allocator can not be deallocated by the other
T()	Constructs an instance of type
T x(y)	x is constructed with the values of y

Allocator template parameters must meet additional requirements

- All instances of an allocator are interchangeable and compare equal to each other
- Members must meet the requirements in “[The Typedef Members Requirements](#)”

Implementation-defined allocators are allowed.

Table 5.2 The Typedef Members Requirements

Member	Type
pointer	T*
const_pointer	T const*
size_type	size_t
difference_type	ptrdiff_t

Utility Components

This sub-clause contains some basic template functions and classes that are used throughout the rest of the library.

Operators

The Standard C++ library provides general templated comparison operators that are based on operator== and operator<.

operator!=

This operator determines if the first argument is not equal to the second argument.

```
template <class T> bool operator!=(const T& x, const T& y);
```

operator>

This operator determines if the first argument is less than the second argument.

```
template <class T> bool operator>(const T& x, const T& y);
```

operator<=

This operator determines if the first argument is less than or equal to the second argument.

```
template <class T> bool operator<=(const T& x, const T& y);
```

operator>=

This operator determines if the first argument is greater than or equal to the second argument.

```
template <class T> bool operator>=(const T& x, const T& y);
```

Pairs

The utility library includes support for paired values.

Constructors

The pair class contains various constructors to fit each pairs needs.

```
pair();
```

Initializes its members as with default type constructors.

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

Initializes and does any implicit conversions if necessary.

operator ==

The pair equality operator returns true if each pair argument is equal to the other.

```
template <class T1, class T2>
```

```
bool operator==(const pair<T1, T2>& x, const pair<T1, T2>& y);
```

operator <

The pair less than operator returns true if the second pair argument is less than the first pair argument.

```
template <class T1, class T2> bool operator <  
    const pair<T1, T2>& x, const pair<T1, T2>& y);
```

make_pair

Makes a pair of the two arguments.

```
template <class T1, class T2>  
pair<T1, T2> make_pair(const T1& x, const T2& y);
```

Remarks

Returns a pair of the two arguments.

Function objects

Function objects have the operator() defined and used for more effective use of the library. When a pointer to a function would normally be passed to an algorithm function the library is specified to accept an object with operator() defined. The use of function objects with function templates increases the power and efficiency of the library

Struct Unary_function and Struct Binary_function classes are provided to simplify the typedef of the argument and result types.

NOTE

In order to manipulate function objects that take one or two arguments it is required that their function objects provide the defined types. If the function object takes one argument then argument_type and result_type are defined. If the function object

takes two arguments then the first_argument_type, second_argument_type, and result_type must be defined.

Arithmetic operations

The utility library provides function object classes with operator() defined for the arithmetic operations.

plus

Adds the first and the second and returns that sum.

```
template <class T> struct plus : binary_function<T,T,T> {  
    T operator()(const T& x, const T& y) const;  
};
```

Remarks

Returns x plus y.

minus

Subtracts the second from the first and returns the difference.

```
template <class T> struct minus : binary_function<T,T,T> {  
    T operator()(const T& x, const T& y) const;  
};
```

Remarks

Returns x minus y.

multiplies

Multiplies the first times the second and returns the resulting value.

```
template <class T> struct multiplies : binary_function<T,T,T> {  
    T operator()(const T& x, const T& y) const;  
};
```

Remarks

Returns x multiplied by y.

divides

Divides the first by the second and returns the resulting value.

```
template <class T> struct divides : binary_function<T,T,T> {  
    T operator()(const T& x, const T& y) const;  
};
```

Remarks

Returns x divided by y.

modulus

Determines the modulus of the first by the second argument and returns the result.

```
template <class T> struct modulus : binary_function<T,T,T> {  
    T operator()(const T& x, const T& y) const;  
};
```

Remarks

Returns x modulus y.

negate

This function returns the negative value of the argument.

```
template <class T> struct negate : unary_function<T,T> {  
    T operator()(const T& x) const;  
};
```

Remarks

Returns the negative of x.

Comparisons

The utility library provides function object classes with operator() defined for the comparison operations.

NOTE	For the greater, less, greater_equal and less_equal template classes specializations for pointers yield a total order.
-------------	--

equal_to

Returns true if the first argument is equal to the second argument.

```
template <class T> struct equal_to : binary_function<T,T,bool> {  
    bool operator()(const T& x, const T& y) const;  
};
```

Remarks

Returns true if x is equal to y.

not_equal_to

Returns true if the first argument is not equal to the second argument.

```
template <class T> struct not_equal_to :  
    binary_function<T,T,bool> {  
  
    bool operator()(const T& x, const T& y) const;  
  
};
```

Remarks

Returns true if x is not equal to y.

greater

Returns true if the first argument is greater than the second argument.

```
template <class T> struct greater : binary_function<T,T,bool> {  
  
    bool operator()(const T& x, const T& y) const;  
  
};
```

Remarks

Returns true if x is greater than y.

less

Returns true if the first argument is less than the second argument.

```
template <class T> struct less : binary_function<T,T,bool> {  
    bool operator()(const T& x, const T& y) const;  
};
```

Remarks

Returns true if x is less than y.

greater_equal

Returns true if the first argument is greater than or equal to the second argument.

```
template <class T> struct greater_equal :  
    binary_function<T,T,bool> {  
  
    bool operator()(const T& x, const T& y) const;  
};
```

Remarks

Returns true if x is greater than or equal to y.

less_equal

Returns true if the first argument is less than or equal to the second argument.

```
template <class T> struct less_equal :  
binary_function<T,T,bool> {  
bool operator()(const T& x, const T& y) const;  
};
```

Remarks

Returns true if x is less than or equal to y.

Logical operations

The utility library provides function object classes with operator() defined for the logical operations.

logical_and

Returns true if the first and the second argument are true.

```
template <class T> struct logical_and :  
binary_function<T,T,bool> {  
bool operator()(const T& x, const T& y) const;  
};
```

Remarks

Returns true if x and y are true.

logical_or

Returns true if the first or the second argument are true.

```
template <class T> struct logical_or :  
binary_function<T,T,bool> {  
bool operator()(const T& x, const T& y) const;  
};
```

Remarks

Returns true if the x or y are true.

logical_not

Returns true if the argument is zero

```
template <class T> struct logical_not : unary_function<T,bool> {  
bool operator()(const T& x) const;  
};
```

Remarks

Returns true if x is equal to zero.

Negators

The utility library provides negators `not1` and `not2` that return the complement of the unary or binary predicate. A predicate is an object that takes one or two arguments and returns something convertible to `bool`.

Unary_negate

In the template class `unary_negate` the operator() returns the compliment of the predicate argument.

not1

The template function `not1` returns the `unary_predicate` of the predicate argument.

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

Remarks

Returns true if pred is not true.

binary_negate

In the template class `binary_negate` the operator() returns the compliment of the predicate arguments.

not2

The template function `not2` returns the `binary_predicate` of the predicate arguments.

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

Remarks

Returns the compliment of the argument.

Binders

The binders classes, bind1st and bind2nd take a function object and a value and return a function object constructed out of the function bound to the value.

Template class binder1st

The binders class bind1st takes a function object and a value and return a function object constructed out of the function bound to the value.

Remarks

The constructor initializes the operation.

bind1st

Binds the first.

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

Remarks

Binds the operation to the first argument type.

Template class binder2nd

The binders class bind1st takes a function object and a value and return a function object constructed out of the function bound to the value.

Remarks

The constructor initializes the operation.

bind2nd

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

Remarks

Binds the operation to the second argument type.

Adaptors for Pointers to Functions

Adaptors for pointers to pointers to both unary and binary functions work with adaptors.

pointer_to_unary_function

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

Remarks

Returns a pointer for a unary function.

class pointer_to_binary_function

A class for pointer to binary binding.

pointer_to_binary_function

```
template <class Arg1, class Arg2, class Result>
pointer_to_binary_function<Arg1,Arg2,Result
ptr_fun(Result (* f)(Arg1, Arg2));
```

Remarks

Returns a pointer for a binary function.

Adaptors for Pointers to Members

Adaptors for pointer members are adaptors that allow you to call member functions for elements within a collection.

mem_fun_t

An adaptor for pointers to member functions.

```
template<class S, class T>
mem_fun_t<S,T,A> : public unary_function<T*, S>
explicit mem_fun(S (T::*p)());
```

Remarks

The constructor for `mem_fun_t` calls the member function it is initialized with a given pointer argument and an appropriate additional argument.

mem_fun1_t

A class for binding a member function.

```
template<class S, class T, class A>
class mem_fun1_t : public binary_function<T*,A, S>
explicit mem_fun1_t(S (T::*p) (A));
```

Remarks

The constructor for `mem_fun1_t` calls the member function it is initialized with given a pointer argument and an appropriate additional argument.

mem_fun

Member to function template functions.

```
template<class S, class T> mem_fun_t<S,T>
mem_fun(S (T::*f) ());
template<class S, class T, class A>
mem_fun(S (T::*f) (A));
```

Remarks

The function returns an object through which a function can be called.

mem_fun_ref_t

Member to function reference object.

```
template<class S, class T>

class mem_fun_ref_t : public unary_function<T, S>
explicit mem_fun_ref_t(S (T::*p)() );
```

Remarks

The function `mem_fun_ref_t` calls the member function it is initialized with given a reference argument.

mem_fun1_ref_t

Member to function reference object.

```
template<class S, class T, class A>

class mem_fun1_ref_t : public binary_function<T,A, S>
explicit mem_fun1_ref_t(S (T::*p)(A));
```

Remarks

The constructor for `mem_fun1_ref_t` calls the member function it is initialized with given a reference argument and an additional argument of the appropriate type.

mem_fun_ref

Template function for member to reference.

```
template<class S, class T> mem_fun_ref_t<S,T>
    mem_fun_ref(S (T::*f) (A)) ;

template<class S, class T, class A> mem_fun1_ref_t<S, T, A>
    mem_fun_ref(S (T::*f) (A)) ;
```

Remarks

The template function `mem_fun_ref` returns an object through which `X::f` can be called given a reference to an `X` followed by the argument required for `f`.

const_mem_fun_t

A constant member to function adaptor.

```
template<class S, class T> class const_mem_fun_t
: public unary_function<T*, S>
explicit const_mem_fun(S (T::*p) () const);
```

Remarks

Provides a constant member to function object.

The constructor for `const_mem_fun_t` calls the member function it is initialized with given a pointer argument.

const_mem_fun1_t

Provides a const to member function object.

```
template<class S, class T, class A> const_mem_fun1_t  
: public binary_function<T,A,S>  
explicit mem_fun1_t(S (T::*p)(A) const);
```

Remarks

The constructor for `const_mem_fun1_t` calls the member function it is initialized with given a pointer argument and an additional argument of the appropriate type.

const_mem_fun_ref_t

A constant member function reference adaptor.

```
template<class S, class T>  
class const_mem_fun_ref_t<S,T> : public unary_function<T,S>  
explicit const_mem_fun_ref_t( S (T::*p) () const);
```

Remarks

The template functions `mem_fun_ref` returns an object through which X::f can be called.

The constructor for `const_mem_fun_ref_t` calls the member function it is initialized with given a reference argument.

const_mem_fun1_ref_t

A constant member to function reference adaptor object.

```
template<class S, class T, class A>
class const_mem_fun1_ref_t<S,T>: public binary_function<T,A,S>
explicit const_mem_fun1_ref_t( S (T::*p) (A) const);
```

Remarks

The constructor for `const_mem_fun1_ref_t` calls the member function it is initialized with given a reference argument and an additional argument of the appropriate type.

The template functions `mem_fun_ref` returns an object through which `X::f` can be called

Memory

The header `<memory>` includes functions and classes for the allocation and deallocation of memory.

allocator members

Members of the allocator class.

address

Determine the address of the allocation.

```
pointer address(reference x) const;
const_pointer address(const_reference x) const;
```

Remarks

Returns the address of the allocation.

allocate

Create an allocation and return a pointer to it.

```
pointer allocate(size_type n, allocator<void>::const_pointer  
    hint=0);
```

Remarks

A pointer to the initial element of an array of storage.

Allocate throw a `bad_alloc` exception if the storage cannot be obtained.

deallocate

Remove an allocation from memory.

```
void deallocate(pointer p, size_type n);
```

Deallocates the storage referenced by p.

max_size

Determines the Maximum size for an allocation.

```
size_type max_size() const throw();
```

Remarks

Returns the largest size of memory that may be.

construct

Determines the Maximum size for an allocation.

```
void construct(pointer p, const_reference val);
```

Remarks

A pointer to the allocated memory is returned.

destroy

Destroys the memory allocated

```
void destroy(pointer p);
```

allocator globals

Provides globals operators in memory allocation.

operator==

Equality operator.

```
template <class T1, class T2> bool operator==  
(const allocator<T1>&,  
 const allocator<T2>&) throw();
```

Remarks

Returns true if the arguments are equal.

operator!=

Inequality operator

```
template <class T1, class T2> bool operator!=  
(const allocator<T1>&,  
 const allocator<T2>&) throw();
```

Remarks

Returns true if the arguments are not equal.

Raw storage iterator

A means of storing the results of un-initialized memory.

NOTE	The formal template parameter OutputIterator is required to have its operator* return an object for which operator& is defined and returns a pointer to T, and is also required to satisfy the requirements of an output iterator.
-------------	--

Constructors

A constructor for the `raw_storage_iterator` class.

```
raw_storage_iterator(OutputIterator x);
```

Remarks

Initializes the iterator.

operator *

A dereference operator.

```
raw_storage_iterator<OutputIterator,T>&
operator*();
```

Remarks

The dereference operator return `*this`.

operator=

The `raw_storage_iterator` assignment operator.

```
raw_storage_iterator<OutputIterator, T>&
operator=(const T& element);
```

Remarks

Constructs a value from element at the location to which the iterator points.

A reference to the iterator.

operator++

Post and Pre-increment operators for `raw_storage_iterator`.

```
raw_storage_iterator<OutputIterator, T>&
operator++();      // Pre-increment
raw_storage_iterator<OutputIterator, T>
operator++(int);   // Post-increment
```

Remarks

Returns the old value of the iterator.

Temporary buffers

Methods for storing and retrieving temporary allocations.

get_temporary_buffer

Retrieves a pointer to store temporary objects.

```
template <class T> pair<T*, ptrdiff_t>
    get_temporary_buffer(ptrdiff_t n);
```

Remarks

Returns an address for the buffer and its size or zero if unsuccessful.

return_temporary_buffer

Deallocation for the `get_temporary_buffer` procedure.

```
template <class T>
void return_temporary_buffer(T* p);
```

Remarks

The buffer must have been previously allocated by `get_temporary_buffer`.

Specialized Algorithms

Algorithm necessary to fulfill iterator requirements.

uninitialized_copy

An uninitialized copy.

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

Remarks

Returns a `ForwardIterator` to the result argument.

uninitialized_fill

An uninitialized fill.

```
template <class ForwardIterator, class T>  
  
void uninitialized_fill  
(ForwardIterator first,  
 ForwardIterator last, const T& x);
```

uninitialized_fill_n

An uninitialized fill with a size limit.

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

Template Class Auto_ptr

The auto_ptr class stores a pointer to an object obtained using new and deletes that object when it is destroyed. For example when a local allocation goes out of scope.

The template auto_ptr_ref holds a reference to an auto_ptr, and is used by the auto_ptr conversions. This allows auto_ptr objects to be passed to and returned from functions.

NOTE	An auto_ptr owns the object it holds a pointer to. When copying an auto_ptr the pointer transfers ownership to the destination.
-------------	---

If more than one auto_ptr owns the same object at the same time the behavior of the program is undefined.

See the example of using std::auto_ptr and extension version for arrays in [Listing 5.1](#)

This extension can be turned off by commenting out #define _MSL_ARRAY_AUTO_PTR in <mslconfig>. No recompile of the C++ lib is necessary, but do rebuild any precompiled headers when making this change.

The functionality provided by the extended std::auto_ptr is very similar to that provided by the newer Metrowerks::alloc_ptr found in <msl_utility>.

Listing 5.1 Using Auto_ptr

```
#include <iostream>  
#include <memory>  
  
using std::auto_ptr;  
using std::_Array;
```

General Utilities Libraries

Template Class Auto_ptr

```
struct A
{
    A() {std::cout << "construct A\n";}
    virtual ~A() {std::cout << "destruct A\n";}
};

struct B
: A
{
    B() {std::cout << "construct B\n";}
    virtual ~B() {std::cout << "destruct B\n";}
};

auto_ptr<B> source();
void sink_b(auto_ptr<B>);
void sink_a(auto_ptr<A>);

auto_ptr<B, _Array<B> > array_source();
void array_sink(auto_ptr<B, _Array<B> >);

auto_ptr<B>
source()
{
    return auto_ptr<B>(new B);
}

void
sink_b(auto_ptr<B>)
{
}

void
sink_a(auto_ptr<A>)
{
}

auto_ptr<B, _Array<B> >
array_source()
{
    return auto_ptr<B, _Array<B> >(new B [2]);
}

void
array_sink(auto_ptr<B, _Array<B> >)
```

```
{  
}  
  
int main()  
{  
    {  
        auto_ptr<B> b(new B);  
        auto_ptr<B> b2(b);  
        b = b2;  
        auto_ptr<B> b3(source());  
        auto_ptr<A> a(b);  
        a = b3;  
        b3 = source();  
        sink_b(source());  
        auto_ptr<A> a2(source());  
        a2 = source();  
        sink_a(source());  
    }  
    {  
        auto_ptr<B, _Array<B> > b(new B [2]);  
        auto_ptr<B, _Array<B> > b2(b);  
        b = b2;  
        auto_ptr<B, _Array<B> > b3(array_source());  
        b3 = array_source();  
        array_sink(array_source());  
        //      auto_ptr<A, _Array<A> > a(b3);      // Should not compile  
        //      a = b3;                                // Should not  
        compile  
    }  
}
```

auto_ptr constructors

Constructs an `auto_ptr` object.

```
explicit auto_ptr(X* p =0) throw();  
  
auto_ptr(auto_ptr& a) throw();  
  
template<class Y> auto_ptr(auto_ptr<Y>& a) throw();
```

operator =

An `auto_ptr` assignment operator.

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

Remarks

Returns the this pointer.

destructor

Destroys the `auto_ptr` object.

```
~auto_ptr() throw();
```

Auto_ptr Members

Member of the `auto_ptr` class.

operator*

The de-reference operator.

```
X& operator*() const throw();
```

Remarks

Returns what the dereferenced this pointer holds.

operator->(

The pointer dereference operator.

```
X* operator->() const throw();
```

Remarks

Returns what the dereferenced this pointer holds.

get

Gets the value that the pointer points to.

```
X* get() const throw();
```

Remarks

Returns what the dereferenced this pointer holds.

release

Releases the auto_ptr object.

```
X* release() throw();
```

Remarks

Returns what the dereferenced this pointer holds.

reset

Resets the auto_ptr to zero or another pointer.

```
void reset(X* p=0) throw();
```

auto_ptr conversions

Conversion functionality for the auto_ptr class for copying and converting.

Conversion Constructor

A conversion constructor.

```
auto_ptr(auto_ptr_ref<X> r) throw();
```

operator auto_ptr_ref

Provides a convert to lvalue process.

```
template<class Y> operator auto_ptr_ref<Y>() throw();
```

Remarks

Returns a reference that holds the this pointer.

operator auto_ptr

Releases the auto_ptr and returns the pointer held.

```
template<class Y> operator auto_ptr<Y>() throw();
```

Remarks

Returns the pointer held.

C Library

The MSL C++ memory libraries use the C library memory functions. See the MSL C Reference for <stdlib.h> functions calloc, malloc, free, realloc for more information.

Date and Time

The header <ctime> has the same contents as the Standard C library header <time.h> but within namespace std.

General Utilities Libraries

Date and Time

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.

The Strings Library

The chapter is constructed in the following sub sections and mirrors clause 21 of the ISO (the International Organization for Standardization) C++ Standard :

- [“Character traits”](#) defines types and facilities for character manipulations
- [“String Classes”](#) lists string and character structures and classes
- [“Class Basic_string”](#) defines facilities for character sequence manipulations.
- [“Null Terminated Sequence Utilities”](#) lists facilities for Null terminated character sequence strings.

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:

- [“Character Trait Definitions”](#)
- [“Character Trait Requirements”](#)
- [“Character Trait Type Definitions”](#)
- [“struct char_traits<T>”](#)

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).

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_type&);
```

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();
```

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
streampos	pos_type	stream position values
mbstate_t	state_type	file state values

struct `char_traits`<T>

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.

String Classes

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

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.

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:

- [“Constructors and Assignments”](#)
- [“Iterator Support”](#)
- [“Capacity”](#)
- [“Element Access”](#)
- [“Modifiers”](#)
- [“String Operations”](#)
- [“Non-Member Functions and Operators”](#)

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

The class `basic_string` can have either of two implementations:

- Refcounted.
- Non-refcounted.

The interface and functionality are identical with both implementations. The only difference is performance. Which performs best is dependent upon usage patterns in each application.

The refcounted implementation ships as the default.

NOTE To enable the non-refcounted implementation un-comment `#define _MSL_NO_REFCOUNT_STRING` in `<mslconfig>`. The C++ library and precompiled headers must be rebuilt after making this change.

Constructors and Assignments

Constructor, destructor and assignment operators and functions.

Constructors

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.

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

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());
```

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());
```

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());
```

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());
```

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());
```

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

Deallocates the memory referenced by the `basic_string` object.

```
~basic_string();
```

Assignment Operator

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

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

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

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

Used to assign a NCTCS to a string.

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

Used to assign a single char type to a string.

Assignment & Addition Operator *basic_string*

Appends the string *rhs* to the current string.

```
string& operator+= (const string& rhs);
```

```
string& operator+= (const charT* s);
```

```
string& operator+= (charT s);
```

Remarks

Both of the overloaded functions construct a string object from the input *s*, and append it to the current string.

The assignment operator returns the *this* pointer.

Iterator Support

Member functions for string iterator support.

begin

Returns an iterator to the first character in the string

```
iterator begin();  
const_iterator begin() const;
```

end

Returns an iterator that is past the end value.

```
iterator end();  
const_iterator end() const;
```

rbegin

Returns an iterator that is equivalent to

```
reverse_iterator(end()).  
reverse_iterator rbegin();  
const_reverse_iterator rbegin() const;
```

rend

Returns an iterator that is equivalent to

```
reverse_iterator(begin()).  
reverse_iterator rend();  
const_reverse_iterator rend() const;
```

Capacity

Member functions for determining a strings capacity.

size

Returns the size of the string.

```
size_type size() const;
```

length

Returns the length of the string

```
size_type length() const;
```

max_size

Returns the maximum size of the string.

```
size_type max_size() const;
```

resize

Resizes the string to size n.

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

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

Returns the memory storage capacity.

```
size_type capacity() const;
```

reserve

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

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

clear

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

```
void clear();
```

empty

Empties the string stored.

```
bool empty() const;
```

Remarks

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

Element Access

Member functions and operators for accessing individual string elements.

operator[]

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

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

at

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

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

Modifiers

Operators for appending a string.

operator+=

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

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

append

A function used to append to the end of a 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);
```

assign

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

```
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.

insert

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

```
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.

erase

Erases the string

```
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.

replace

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

```
basic_string replace 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);
```

Strings Library

Class Basic_string

```
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,

copy

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

```
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.

swap

Swaps one string for another.

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

String Operations

Member functions for sequences of character operations.

c_str

Returns the string as a Null terminated character type sequence.

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

data

Returns the string as an array without a Null terminator.

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

get_allocator

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

```
allocator_type get_allocator() const;
```

find

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

```
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;
```

Remarks

The found position or `npos` if not found.

rfind

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

```
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;
```

Remarks

The found position or npos if not found.

find_first_of

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

```
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;
```

Remarks

The found position or `npos` if not found.

find_last_of

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

```
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;
```

Remarks

The found position or `npos` if not found is returned.

find_first_not_of

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

```
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;
```

Remarks

The found position or `npos` if not found.

find_last_not_of

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

```
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;
```

Remarks

The found position or `npos` if not found.

substr

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

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

Remarks

May throw an exception,

compare

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

```
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;
```

Return

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.

Non-Member Functions and Operators

operator+

Appends one string to another.

```
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);
```

Remarks

The combined strings are returned.

operator==

Test for lexicographical equality.

```
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);
```

Return

True if the strings match otherwise false.

operator!=

Test for lexicographical inequality.

```
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);
```

Remarks

True if the strings do not match otherwise false.

operator<

Tests for a lexicographically less than condition.

```
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);
```

Remarks

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

operator>

Tests for a lexicographically greater than condition.

```
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);
```

Remarks

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

operator<=

Tests for a lexicographically less than or equal to condition.

```
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);
```

Remarks

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

operator>=

Tests for a lexicographically greater than or equal to condition.

```
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);
```

Remarks

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

swap

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

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

Inverters and extractors

Overloaded inverters and extractors for `basic_string` types.

operator>>

Overloaded `extractor` for stream input operations.

```
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<<

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

```
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

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

```
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.

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 Character support testing

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

String Support

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

Table 6.3 String support testing

<code><cstring.h></code>	<code><wchar.h></code>	<code><cstring.h></code>	<code><wchar.h></code>
<code>memchr</code>	<code>wmemchr</code>	<code>strerror</code>	
<code>memcmp</code>	<code>wmemcmp</code>	<code>strlen</code>	<code>wcslen</code>
<code>memcpy</code>	<code>wmemcpy</code>	<code>strncat</code>	<code>wcsncat</code>

Table 6.3 String support testing

<cstring.h>	<wchar.h>	<cstring.h>	<wchar.h>
memmove	wmemmove	strcmp	wcsncmp
memset	wmemset	strncpy	wcsncpy
strcat	wcscat	strpbrk	wcspbrk
strchr	wcschr	strrchr	wcsrchr
strcmp	wcsncmp	strspn	wcsspn
strcoll	wcscoll	strstr	wcsstr
strcpy	wcscpy	strtok	wcstok
strcspn	wcsccspn	strxfrm	wcsxfrm
mbstate_t	size_t	wint_t	
NULL		WCHAR_MAX	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	<cstdlib.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

Strings Library

Null Terminated Sequence Utilities

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

wchar.h	wchar.h	wchar.h	<cstdlib.h>
getwc	vfwprintf	wprintf	strtoul
getwchar	vwprintf	wscanf	wctomb
mbrlen	vswprintf		wcstombs

Localization Library

This chapter describes components that C++ library that may use for porting to different cultures.

The Localization library

Much of named locales is implementation defined behavior and is not portable between vendors. This document specifies the behavior of MSL C++. Other vendors may not provide this functionality, or may provide it in a different manner.

The chapter is constructed in the following sub sections and mirrors clause 22 of the ISO (the International Organization for Standardization) C++ Standard :

- [“Strings and Characters in Locale Data Files”](#)
- [“Locales”](#)
- [“Standard Locale Categories”](#)
- [“C Library Locales”](#)

Supported Locale Names

MSL C++ predefines only two names: “C” and “”. However, other names sent to the locale constructor are interpreted as file names containing data to create a named locale. So localizing your program is as easy as creating a data file specifying the desired behavior. The format for this data file is outlined below for each different facet.

A `locale` is a collection of `facets`. And a `facet` is a class that provides a certain behavior. The “C” `locale` contains the following facets:

- `ctype<char> & ctype<wchar_t>`
- `codecvt<char, char, mbstate_t> & codecvt<wchar_t, char, mbstate_t>`
- `num_get<char> & num_get<wchar_t>`
- `num_put<char> & num_put<wchar_t>`

- `numpunct<char> & numpunct<wchar_t>`
- `collate<char> & collate<wchar_t>`
- `time_get<char> & time_get<wchar_t>`
- `time_put<char> & time_put<wchar_t>`
- `money_get<char> & money_get<wchar_t>`
- `money_put<char> & money_put<wchar_t>`
- `moneypunct<char, bool> & moneypunct<wchar_t, bool>`
- `messages<char> & messages<wchar_t>`

A named `locale` replaces many of these facets with “`_byname`” versions, whose behavior can vary based on the name passed.

- `ctype_byname<char> & ctype_byname<wchar_t>`
- `codecvt_byname<char, char, mbstate_t> & codecvt_byname<wchar_t, char, mbstate_t>`
- `numpunct_byname<char> & numpunct_byname<wchar_t>`
- `collate_byname<char> & collate_byname<wchar_t>`
- `time_get_byname<char> & time_get_byname<wchar_t>`
- `time_put_byname<char> & time_put_byname<wchar_t>`
- `moneypunct_byname<char, bool> & moneypunct_byname<wchar_t, bool>`
- `messages_byname<char> & messages_byname<wchar_t>`

The behavior of each of these “`_byname`” facets can be specified with a data file. A single data file can contain data for all of the byname facets. That way, when you code:

```
locale myloc("MyLocale");
```

then the file “`MyLocale`” will be used for each “`_byname`” facet in `myloc`.

Strings and Characters in Locale Data Files

The named locale facility involves reading strings and characters from files. This document gives the details of the syntax used to enter strings and characters.

Character Syntax

Characters in a locale data file can in general appear quoted ('') or not. For example:

```
thousands_sep = ,  
thousands_sep = ', '
```

Both of the above statements set `thousands_sep` to a comma. Quotes might be necessary to disambiguate the intended character from ordinary whitespace. For example, to set the `thousands_sep` to a space character, quotes must be used:

```
thousands_sep = ' '
```

The whitespace appearing before and after the equal sign is not necessary and insignificant.

Escape sequences

The usual C escape sequences are recognized. For example, to set the `thousands_sep` to the single quote character, an escape sequence must be used:

```
thousands_sep = '\'
```

The recognized escape sequences are:

- `\n` - newline
- `\t` - horizontal tab
- `\v` - vertical tab
- `\b` - backspace
- `\r` - carriage return
- `\f` - form feed
- `\a` - alert
- `\\\` - `\`
- `\?` - `?`
- `\"` - `"`
- `\'` - `'`
- `\u \U` - universal character
- `\x` - hexadecimal character
- `\ooo` - octal character

The octal character may have from 1 to 3 octal digits (digits must be in the range [0, 7]. The parser will read as many digits as it can to interpret a valid octal number. For example:

```
\18
```

This is the character `'\1'` followed by the character `'8'`.

\17

But this is the single character '\17'.

The hexadecimal and universal character formats are all identical with each other, and have slightly relaxed syntax compared to the formats specified in the standard. The x (or u or U) is followed by zero to `sizeof(charT) *CHAR_BIT/4` hexadecimal digits. `charT` is `char` when reading narrow data, and `wchar_t` when reading wide data (even when reading wide data from a narrow file). On Macintosh and Windows this translates to 0 to 2 digits when reading a `char`, and from 0 to 4 digits when reading a `wchar_t`. Parsing the character is terminated when either the digit limit has been reached, or a non-hexadecimal digit has been reached. If there are 0 valid digits, then the character is read as '\0'. Example (assume a 8 bit `char` and 16 bit `wchar_t`):

`\x01234`

When reading narrow data this is the following sequence of 4 char's: '\1' '2' '3' '4'

The '\x01' is read as one character, but the following '2' is not included because a 8 bit `char` can only hold 2 hex digits.

When reading wide data the above example parses to the following two `wchar_t`'s:

`L'\x123' L'4'`

The '\x0123' is read as one `wchar_t`, but the following '4' is not included because a 16 bit `wchar_t` can only hold 4 hex digits.

Errors

If a character is expected, but an end of file occurs, then `failbit` is set. If a character is started with a single quote, and end of file occurs before the character within the quotes can be read, or if a closing quote is not found directly after the character, then `failbit` will be set. Depending on the context of when the character is being read, setting `failbit` may or may not cause a runtime error to be thrown.

String Syntax

Strings can be quoted or not (using ""). If the string contains white space, then it must be quoted. For example:

`Hi there!`

This would be parsed as two strings: "Hi" and "there!". But the following is one string:

"Hi there!"

If a string begins with quotes, but does not end with a quote (before end of file), then failbit will be set. This may nor may not cause a runtime error to be thrown (depending on the context).

Any of the escape sequences described under character syntax are allowed within strings. But within strings, single quotes do not delimit characters. Instead single quotes are just another character in the string. Note that you can use \' to place the string quote character within a string.

Locales

The header <locale> defines classes used to contain and manipulate information for a locale.

- [“Class locale”](#)
- [“Locale Globals”](#)
- [“Convenience Interfaces”](#)
- [“Character Classification”](#)

Class locale

The class locale contains a set of facets for locale implementation. These facets are as if they were and index and an interface at the same time.

Combined Locale Names

Two locale constructors can result in a new locale whose name is a combination of the names of two other locales:

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

If other has a name (and if one has a name in the case of the second constructor), then the resulting locale's name is composed from the two locales' names. A combined name locale has the format:

```
collate_name/ctype_name/monetary_name/numeric_name/time_name/
messages_name
```

Each name is the name of a locale from which that category of facets was copied.

The locale loc is created from two locales: other and one. The facets in the categories collate and numeric are taken from one. The rest of the facets are taken from other. The name of the resulting locale is:

```
one/other/other/one/other/other
```

The locale loc2 is created from the “C” locale and from loc (which already has a combined name). It takes only the monetary and collate facets from loc, and the rest from “C”:

```
one/C/other/C/C/C
```

Using this format, two locales can be compared by name, and if their names are equal, then they have the same facets.

Listing 7.1 Locale example usage:

```
#include <locale>
#include <iostream>

int main()
{
    using std::locale;
    locale loc(locale("other"), locale("one"),
               locale::collate | locale::numeric);
    std::cout << loc.name() << '\n';
    locale loc2(locale(), loc, locale::monetary |
               locale::collate);
    std::cout << loc2.name() << '\n';
}
```

Locale Types

This library contains various types specific for locale implementation.

locale::Category

An integral type used as a mask for all types.

```
typedef int category;
```

Each `locale` member function takes a `locale::category` argument based on a corresponding `facet`.

Table 7.1 Locale Category Facets

Category	Includes Facets
collate	<code>collate<char>, collate<wchar_t></code>
ctype	<code>ctype<char>, ctype<wchar_t>, codecvt<char,char,mbstate_t>, codecvt<wchar_t,char,mbstate_t></code>
messages	<code>messages<char>, messages<wchar_t></code>
monetary	<code>moneypunct<char>, moneypunct<wchar_t>, moneypunct<char,true>, moneypunct<wchar_t,true>, money_get<char>, money_get<wchar_t>, money_put<char>, money_put<wchar_t></code>
numeric	<code>numpunct<char>, numpunct<wchar_t>, num_get<char>, num_get<wchar_t>, num_put<char>, num_put<wchar_t></code>
time	<code>time_get<char>, time_get<wchar_t>, time_put<char>, time_put<wchar_t></code>

An implementation is included for each `facet` template member of a category.

Table 7.2 Required Instantiations

Category	Includes Facets
collate	<code>collate_byname<char>, collate_byname<wchar_t></code>
ctype	<code>ctype_byname<char>, ctype_byname<wchar_t></code>
messages	<code>messages_byname<char>, messages_byname<wchar_t></code>

Table 7.2 Required Instantiations

Category	Includes Facets
monetary	money_punct_byname<char, International>, money_punct_byname<wchar_t, International>, money_get<C, InputIterator>, money_put<C, OutputIterator>
numeric	num_punct_byname<char>, num_punct_byname<wchar_t>, num_get<C, InputIterator>, num_put<C, OutputIterator>
time	time_get<char, InputIterator>, time_get_byname<char, InputIterator>, time_get<wchar_t, OutputIterator>, time_get_byname<wchar_t, OutputIterator>, time_put<char, OutputIterator>, time_put_byname<char, OutputIterator>, time_put<wchar_t, OutputIterator>, time_put_byname<wchar_t, OutputIterator>

Locale::facet

The class `facet` is the base class for `locale` feature sets.

Listing 7.2 class locale:: facet synopsis

```
namespace std {
class locale::facet {
protected:
explicit facet(size_t refs = 0);
virtual ~facet();
private:
facet(const facet&); // not defined
void operator=(const facet&); // not defined };
}
```

locale::id

The class `locale::id` is used for an index for locale facet identification.

Listing 7.3 class `locale::id` synopsis

```
namespace std {
class locale::id {
public:
    id();
private:
    void operator=(const id&); // not defined
    id(const id&); // not defined };
}
```

Constructors

Constructs an object of `locale`.

```
locale() throw();

locale(const locale& other) throw();

explicit locale(const char* std_name);

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

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

locale(const locale& other, const locale& one, category cats);

locale("")
```

Remarks

The `""` `locale` will attempt to read the environment variable `LANG` and create a `locale` with the associated string. If `getenv("LANG")` returns null, then `"C"` is used. There is no data file associated with the `"C"` `locale`. The `"C"` `locale` is coded directly into MSL C++.

destructor

Removes a `locale` object.

```
~locale() throw();
```

Locale Members

Member functions of the class `locale`.

combine

Creates a copy of the `locale` except for the type `Facet` of the argument.

```
template <class Facet> locale combine(const locale& other);
```

Remarks

The newly created `locale` is returned.

name

Returns the name of the `locale`.

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

Remarks

Returns the name of the `locale` or “*” if there is none.

Locale Operators

The class `locale` has overloaded operators.

operator ==

The locale equality operator.

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

Remarks

The equality operator returns true if both arguments are the same locale.

operator !=

The locale non-equality operator

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

Remarks

The non-equality operator returns true if the locales are not the same.

operator ()

Compares two strings using `use_facet<collate>`.

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

Remarks

Returns true if the first argument is less than the second argument for ordering.

Locale Static Members

global

Installs a new global locale.

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

Remarks

Global returns the previous locale.

classic

Sets the locale to “C” locale equivalent to locale(“C”).

```
static const locale& classic();
```

Remarks

This function returns the “C” locale.

Locale Globals

Locale has two global functions.

use_facet

Retrieves a reference to a facet of a locale.

```
template <class Facet> const Facet& use_facet  
(const locale& loc);
```

Remarks

Throws a `bad_cast` exception if `has_facet` is false.

The function returns a facet reference to corresponding to its argument.

has_facet

Tests a locale to see if a facet is present

```
template <class Facet> bool has_facet  
(const locale& loc) throw();
```

Remarks

If a facet requested is present `has_facet` returns true.

Convenience Interfaces

Character classification functionality is provided for in the `locale` class.

Character Classification

In the character classification functions true is returned if the function evaluates to true.

Listing 7.4 Character Classification

```
template <class charT> bool isspace (charT c, const locale& loc);  
template <class charT> bool isprint (charT c, const locale& loc);  
template <class charT> bool iscntrl (charT c, const locale& loc);  
template <class charT> bool isupper (charT c, const locale& loc);  
template <class charT> bool islower (charT c, const locale& loc);  
template <class charT> bool isalpha (charT c, const locale& loc);  
template <class charT> bool isdigit (charT c, const locale& loc);  
template <class charT> bool ispunct (charT c, const locale& loc);  
template <class charT> bool isxdigit(charT c, const locale& loc);  
template <class charT> bool isalnum (charT c, const locale& loc);  
template <class charT> bool isgraph (charT c, const locale& loc);
```

Character Conversions

Character conversion functionality is provided for in the `locale` class.

toupper

Converts to upper case character using the locale specified.

```
template <class charT> charT toupper  
(charT c, const locale& loc) const;
```

Remarks

Returns the upper case character.

tolower

Converts to a lower case character using the locale specified.

```
template <class charT> charT tolower  
(charT c, const locale& loc) const;
```

Remarks

Returns the lower case character.

Standard Locale Categories

The standard provides for various locale categories for providing formatting and manipulation of data and streams.

- [“The Ctype Category”](#)
- [“The Numeric Category”](#)
- [“The Numeric Punctuation Facet”](#)
- [“The Collate Category”](#)

- “[The Time Category](#)”
- “[The Monetary Category](#)”
- “[The Message Retrieval Category](#)”
- “[Program-defined Facets](#)”

The Ctype Category

The type `ctype_base` provides for const enumerations.

Listing 7.5 Ctype Category

```
namespace std {
class ctype_base
{
public:
    enum mask
    {
        alpha    = 0x0001,
        blank   = 0x0002,
        cntrl   = 0x0004,
        digit   = 0x0008,
        graph   = 0x0010,
        lower   = 0x0020,
        print   = 0x0040,
        punct   = 0x0080,
        space   = 0x0100,
        upper   = 0x0200,
        xdigit  = 0x0400,
        alnum   = alpha | digit
    };
};
}
```

Template Class Ctype

The class `ctype` provides for character classifications.

is

An overloaded function that tests for or places a mask.

```
bool is(mask m, charT c) const;
```

Test if *c* matches the mask *m*.

Returns true if the char *c* matches mask.

```
const charT* is
```

```
(const charT* low, const charT* high,
```

```
mask* vec) const;
```

Fills between the low and high with the mask argument.

Returns the second argument.

scan_is

Scans the range for a mask value.

```
const charT* scan_is
```

```
(mask m, const charT* low, const charT* high) const;
```

Remarks

Returns a pointer to the first character in the range that matches the mask, or the high argument if there is no match.

scan_not

Scans the range for exclusion of the mask value.

```
const charT* scan_not(mask m,  
const charT* low, const charT* high) const;
```

Remarks

Returns a pointer to the first character in the range that does not match the mask, or the `high` argument if all characters match

toupper

Converts to a character or a range of characters to uppercase.

```
charT toupper(charT) const;  
const charT* toupper (charT* low, const charT* high) const;
```

Remarks

Returns the converted char if it exists.

tolower

Converts to a character or a range of characters to lowercase.

```
charT tolower(charT c) const;  
const charT* tolower(charT* low, const charT* high) const;
```

Remarks

Returns the converted char if it exists.

widen

Converts a `char` or range of `char` type to the `charT` type.

```
charT widen(char c) const;  
const char* widen  
(const char* low, const char* high, charT* to) const;
```

Remarks

The converted `charT` is returned.

narrow

Converts a `charT` or range of `charT` type to the `char` type.

```
char narrow(charT c, char default) const;  
const charT* narrow(const charT* low, const charT*, char default,
```

Remarks

The converted `char` is returned.

Ctype Virtual Functions

Virtual functions must be overloaded in the locale.

do_is

Implements the function `is`.

```
bool do_is (mask m, charT c) const;  
const charT* do_is  
(const charT* low, const charT* high, mask* vec) const;
```

do_scan_is

Implements the function `scan_is`.

```
const charT* do_scan_is(mask m,  
const charT* low, const charT* high) const;
```

do_scan_not

Implements the function `scan_not`.

```
const charT* do_scan_not(mask m,  
                          const charT* low, const charT* high) const;
```

do_toupper

Implements the function `toupper`.

```
charT do_toupper(charT c) const;  
const charT* do_toupper(charT* low, const charT* high) const;
```

do_tolower

Implements the function `tolower`.

```
charT do_tolower(charT c) const;  
const charT* do_tolower(charT* low, const charT* high) const;
```

do_widen

Implements the function `widen`.

```
charT do_widen(char c) const;  
const char* do_widen(const char* low, const char* high,  
                     charT* dest) const;
```

do_narrow

Implements the function `narrow`.

```
char do_narrow(charT c, char dfault) const;  
  
const charT* do_narrow(const charT* low, const charT* high,  
char dfault, char* dest) const;
```

Template class ctype_byname

The template class `ctype_byname` has several responsibilities:

- character classification
- conversion to upper/lower case
- conversion to/from char

Ctype_byname Constructor

```
explicit ctype_byname(const char*,  
size_t refs = 0);
```

The facet `ctype` has several responsibilities:

- character classification
- conversion to upper/lower case
- conversion to/from char

The first two of these items can be customized with `ctype_byname`. If you construct `ctype_byname` with a `const char*` that refers to a file, then that file is scanned by `ctype_byname`'s constructor for information to customize character classification, and case transformation tables.

```
ctype_byname<char> ct("en_US");  
// looks for the file "en_US"
```

If the file "en_US" exists, has `ctype` data in it, and there are no syntax errors in the data, then `ct` will behave as dictated by that data. If the file exists, but does not have `ctype` data in it, then the facet will behave as if it were constructed with "C". If the file has `ctype` data in it, but there is a syntax error in the data, or if the file does not exist, then a `std::runtime_error` is thrown.

For `ctype_byname<char>`, the ctype data section begins with:

```
$ctype_narrow
```

For `ctype_byname<wchar_t>`, the ctype data section begins with:

```
$ctype_wide
```

Classification

The classification table is created with one or more entries of the form:

```
ctype[character1 - character2] =
    ctype_classification |
    ctype_classification | ...
    ctype[character] = ctype_classification |
    ctype_classification | ...
```

where character, character1 and character2 are characters represented according to the rules for [“Strings and Characters in Locale Data Files”](#). The characters may appear as normal characters:

```
ctype[a - z]
ctype['a' - 'z']
```

or as octal, hexadecimal or universal:

```
ctype['\101']
ctype['\x41']
ctype['\u41']
```

The usual escape sequences are also recognized: `\n`, `\t`, `\a`, `\\\`, `\'` and so on.

On the right hand side of the equal sign, `ctype_classification` is one of:

- alpha
- blank
- cntrl
- digit
- graph
- lower
- print
- punct

- space
- upper
- xdigit

An `|` can be used to assign a character, or range of characters, more than one classification. These keywords correspond to the names of the enum `ctype_base::mask`, except that `alnum` is not present. To get `alnum` simply specify `"alpha | digit"`. The keyword `blank` is introduced, motivated by C99's `isblank` function.

Each of these keywords represent one bit in the `ctype_base::mask`. Thus for each entry into the `ctype` table, one must specify all attributes that apply. For example, in the "C" locale `a-z` are represented as:

```
ctype['a' - 'z'] =
    xdigit | lower | alpha | graph | print
```

Case Transformation

Case transformation is usually handled by a table that maps each character to itself, except for those characters being transformed - which are mapped to their transformed counterpart. For example, a lower case map might look like:

```
lower['a'] == 'a'
lower['A'] == 'a'
```

This is represented in the `ctype` data as two tables: `lower` and `upper`. You can start a map by first specifying that all characters map to themselves:

```
lower['\0' - '\xFF'] = '\0' - '\xFF'
```

You can then override a subrange in this table to specify that `'A' - 'Z'` maps to `'a' - 'z'`:

```
lower['A' - 'Z']      = 'a' - 'z'
```

These two statements have completely specified the lower case mapping for an 8 bit `char`. The upper case table is similar. For example, here is the specification for upper case mapping of a 16 bit `wchar_t` in the "C" locale:

```
upper['\0' - '\xFFFF'] = '\0' - '\xFFFF'
upper['a' - 'z']       = 'A' - 'Z'
```

Below is the complete "C" locale specification for both `ctype_byname<char>` and `ctype_byname<wchar_t>`. Note that a "C" data file does not actually exist. But if you provided a locale data file with this information in it, then the behavior would be the same as the "C" locale.

Listing 7.6 Example of “C” Locale

```
$ctype_narrow
ctype['\x00' - '\x08'] = cntrl
ctype['\x09']           = cntrl | space | blank
ctype['\x0A' - '\x0D'] = cntrl | space
ctype['\x0E' - '\x1F'] = cntrl
ctype['\x20']           = space | blank | print
ctype['\x21' - '\x2F'] = punct | graph | print
ctype['\x30' - '\x39'] = digit | xdigit | graph | print
ctype['\x3A' - '\x40'] = punct | graph | print
ctype['\x41' - '\x46'] = xdigit | upper | alpha | graph | print
ctype['\x47' - '\x5A'] = upper | alpha | graph | print
ctype['\x5B' - '\x60'] = punct | graph | print
ctype['\x61' - '\x66'] = xdigit | lower | alpha | graph | print
ctype['\x67' - '\x7A'] = lower | alpha | graph | print
ctype['\x7B' - '\x7E'] = punct | graph | print
ctype['\x7F']           = cntrl

lower['\0' - '\xFF'] = '\0' - '\xFF'
lower['A' - 'Z']     = 'a' - 'z'

upper['\0' - '\xFF'] = '\0' - '\xFF'
upper['a' - 'z']     = 'A' - 'Z'

$ctype_wide
ctype['\x00' - '\x08'] = cntrl
ctype['\x09']           = cntrl | space | blank
ctype['\x0A' - '\x0D'] = cntrl | space
ctype['\x0E' - '\x1F'] = cntrl
ctype['\x20']           = space | blank | print
ctype['\x21' - '\x2F'] = punct | graph | print
ctype['\x30' - '\x39'] = digit | xdigit | graph | print
ctype['\x3A' - '\x40'] = punct | graph | print
ctype['\x41' - '\x46'] = xdigit | upper | alpha | graph | print
ctype['\x47' - '\x5A'] = upper | alpha | graph | print
ctype['\x5B' - '\x60'] = punct | graph | print
ctype['\x61' - '\x66'] = xdigit | lower | alpha | graph | print
ctype['\x67' - '\x7A'] = lower | alpha | graph | print
ctype['\x7B' - '\x7E'] = punct | graph | print
ctype['\x7F']           = cntrl

lower['\0' - '\xFFFF'] = '\0' - '\xFFFF'
lower['A' - 'Z']       = 'a' - 'z'

upper['\0' - '\xFFFF'] = '\0' - '\xFFFF'
```

```
upper[ 'a' - 'z' ]           = 'A' - 'Z'
```

Ctype Specializations

The category `ctype` has various specializations to help localization.

The class `ctype<char>` has four protected data members:

- `const mask* __table_;`
- `const unsigned char* __lower_map_;`
- `const unsigned char* __upper_map_;`
- `bool __owns_;`

Each of the pointers refers to an array of length `ctype<char>::table_size`. The destructor `~ctype<char>()` will delete `__table_` if `__owns_` is true, but it will not delete `__lower_map_` and `__upper_map_`. The derived class destructor must take care of deleting these pointers if they are allocated on the heap (`ctype<char>` will not allocate these pointers). A derived class can set these pointers however it sees fit, and have `ctype<char>` implement all of the rest of the functionality.

The class `ctype<wchar_t>` has three protected data members:

```
Metrowerks::range_map<charT, ctype_base::mask> __table_;  
Metrowerks::range_map<charT, charT>           __lower_map_;  
Metrowerks::range_map<charT, charT>           __upper_map_;
```

The class `range_map` works much like the tables in `ctype<char>` except that they are sparse tables. This avoids having tables of length 0xFFFF. These tables map the first template parameter into the second.

Listing 7.7 The range_map interface

```
template <class T, class U>  
class range_map  
{  
public:  
    U operator[](const T& x) const;  
    void insert(const T& x1, const T& x2, const U& y1, const U& y2);  
    void insert(const T& x1, const T& x2, const U& y1);  
    void insert(const T& x1, const U& y1);  
    void clear();  
};
```

When constructed, the `range_map` implicitly holds a map of all `T` that map to `U()`. Use of the insert methods allows exceptions to that default mapping. For example, the first insert method maps the range `[x1 - x2]` into `[y1 - y2]`. The second insert method maps the `x`-range into a constant: `y1`. And the third insert method maps the single `T(x1)` into `U(y1)`. The method `clear()` brings the `range_map` back to the default setting: all `T` map into `U()`.

A class derived from `ctype<wchar_t>` can fill `__table_`, `__lower_map_` and `__upper_map_` as it sees fit, and allow the base class to query these tables. For an example see `ctype_byname<wchar_t>`.

Specialized Ctype Constructor and Destructor

Specialized `ctype<char>` and `ctype<wchar_t>` constructors and destructors.

Constructor

Constructs a ctype object.

```
explicit ctype  
(const mask* tbl = 0, bool del = false,  
size_t refs = 0);
```

destructor

Removes a ctype object.

```
~ctype();
```

Specialized Ctype Members

Listing 7.8 Several Ctype members are specialized in the standard library

```
Specialized ctype<char> and ctype<wchar_t> member functions.  
bool is(mask m, char c) const;  
const char* is(const char* low, const char* high,  
mask* vec) const;  
const char* scan_is(mask m,  
const char* low, const char* high) const;  
const char* scan_not(mask m,  
const char* low, const char* high) const;  
char toupper(char c) const;  
const char* toupper(char* low, const char* high) const;  
char tolower(char c) const;  
const char* tolower(char* low, const char* high) const;  
char widen(char c) const;  
const char* widen(const char* low, const char* high,  
char* to) const;  
char narrow(char c, char /*default*/) const;  
const char* narrow(const char* low, const char* high,  
char /*default*/, char* to) const;  
const mask* table() const throw();
```

Ctype<Char> Static Members

Specialized ctype<char> static members. are provided.

classic_table

Determines the classification of characters in the “C” locale.

```
static const mask* classic_table() throw();
```

Remarks

Returns to a table that represents the classification in a “C” locale.

Ctype<Char> Virtual Functions

Specialize ctype<char> virtual member functions are identical functionality to “[Ctype Virtual Functions.](#)”

Class ctype_byname<char>

A specialization of ctype_byname of type char.

Ctype_byname<char> Constructor

```
explicit ctype_byname(const char*,
                      size_t refs = 0);
```

The facet ctype has several responsibilities:

- character classification
- conversion to upper/lower case
- conversion to/from char

For a full and complete description of this facet specialization see “[Ctype_byname Constructor](#)” which list the process in greater detail.

Template Class Codecvt

A class used for converting one character encoded types to another. For example, from wide character to multibyte character sets.

Codecvt Members

Member functions of the codecvt class.

out

Convert internal representation to external.

```
result out(  
    stateT& state, const internT* from,  
    const internT* from_end, const internT*&  
    from_next, externT* to, externT* to_limit,  
    externT*& to_next) const;
```

unshift

Converts the shift state.

```
result unshift(stateT& state,  
    externT* to, externT* to_limit, externT*& to_next) const;
```

in

Converts external representation to internal.

```
result in(stateT& state, const externT* from,  
    const externT* from_end, const externT*&  
    from_next, internT* to, internT* to_limit,  
    internT*& to_next) const;
```

always_noconv

Determines if no conversion is ever done.

```
bool always_noconv() const throw();
```

Remarks

Returns true if no conversion will be done.

length

Determines the length between two points.

```
int length(stateT& state, const externT* from,  
           const externT* from_end, size_t max) const;
```

Remarks

The distance between two points is returned.

max_length

Determines the length necessary for conversion.

```
int max_length() const throw();
```

Remarks

The number of elements to convert from externT to internT is returned.

Codecvt Virtual Functions

Virtual functions for `codecvt` implementation.

```
result do_out(stateT& state, const internT* from,
    const internT* from_end,
    const internT*& from_next, externT* to,
    externT* to_limit, externT*& to_next) const;
```

Implements `out`.

The result is returned as a value as in [“Convert Result Values.”](#)

```
result do_in(stateT& state, const externT* from,
    const externT* from_end,
    const externT*& from_next, internT* to,
    internT* to_limit, internT*& to_next) const;
```

Implements `in`.

The result is returned as a value as in [“Convert Result Values.”](#)

```
result do_unshift(stateT& state,
    externT* to, externT* to_limit, externT*& to_next) const;
Implements unshift.
```

The result is returned as a value as in [“Convert Result Values.”](#)

```
int do_encoding() const throw();
```

Implements `encoding`.

```
bool do_always_noconv() const throw();
```

Implements `always_noconv`.

```
int do_length(stateT& state, const externT* from, const externT*
    from_end, size_t max) const;
```

Implements `length`.

```
int do_max_length() const throw();
```

Implements `max_length`.

Table 7.3 Convert Result Values

Value	Meaning
error	Encountered a <code>from_type</code> character it could not convert
noconv	No conversion was needed
ok	Completed the conversion
partial	Not all source characters converted

Template Class `Codecvt_byname`

The facet `codecvt` is responsible for translating internal characters (`wchar_t`) to/from external `char`'s in a file.

There are several techniques for representing a series of `wchar_t`'s with a series of `char`'s. The `codecvt_byname` facet can be used to select among several of the encodings. If you construct `codecvt_byname` with a `const char*` that refers to a file, then that file is scanned by `codecvt_byname`'s constructor for information to customize the encoding.

```
codecvt_byname<wchar_t, char, std::mbstate_t>
cvt("en_US");
```

If the file "en_US" exists, has `codecvt` data in it, and there are no syntax errors in the data, then `cvt` will behave as dictated by that data. If the file exists, but does not have `codecvt` data in it, then the facet will behave as if it were constructed with "C". If the file has `codecvt` data in it, but there is a syntax error in the data, or if the file does not exist, then a `std::runtime_error` is thrown.

For `codecvt_byname<char, char, mbstate_t>`, the `codecvt` data section begins with:

```
$codecvtnarrow
```

For `codecvt_byname<wchar_t, char, mbstate_t>`, the `codecvt` data section begins with:

```
$codecvt_wide
```

Although `$codecvt_narrow` is a valid data section, it really does not do anything. The `codecvt_byname<char, char, mbstate_t>` facet does not add any functionality beyond `codecvt<char, char, mbstate_t>`. This facet is a degenerate case of `noconv` (no conversion). This can be represented in the locale data file as:

```
$codecvt_narrow
noconv
```

The facet `codecvt_byname<wchar_t, char, mbstate_t>` is much more interesting. After the data section introduction (`$codecvt_wide`), one of these keywords can appear:

- noconv
- UCS-2
- JIS
- Shift-JIS
- EUC
- UTF-8

These keywords will be parsed as strings according to the rules for [“Strings and Characters in Locale Data Files”](#).

Codecvt_byname Keywords

These `Codecvt_byname` keywords will be parsed as strings according to the rules for entering strings in locale data files.

noconv

This conversion specifies that the base class should handle the conversion. The MSL C++ implementation of `codecvt<wchar_t, char, mbstate_t>` will I/O all bytes of the `wchar_t` in native byte order.

UCS-2

This encoding input and outputs the two lowest order bytes of the `wchar_t`, high byte first. For a big-endian, 16 bit `wchar_t` platform, this encoding is equivalent to `noconv`.

JIS

This is an early encoding used by the Japanese to represent a mixture of ASCII and a subset of Kanji.

Shift-JIS

Another early encoding used by the Japanese to represent a mixture of ASCII and a subset of Kanji.

EUC

Extended Unix Code.

UTF-8

A popular Unicode multibyte encoding. For example

```
$ codecvt_wide  
UTF-8
```

specifies that `codecvt_byname<wchar_t, char, mbstate_t>` will implement the UTF-8 encoding scheme. If this data is in a file called "en_US", then the following program can be used to output a `wchar_t` string in UTF-8 to a file:

Listing 7.9 Example of Writing a `wchar_t` String in utf-8 to a File:

```
#include <iostream>  
#include <iomanip>  
  
int main()  
{  
    std::locale loc("en_US");  
    std::wofstream out;  
    out.imbue(loc);  
    out.open("test.dat");  
    out << L"This is a test \x00DF";  
}
```

The binary contents of the file is (in hex):

54 68 69 73 20 69 73 20 61 20 74 65 73 74 20 C3 9F

Without the UTF-8 encoding, the default encoding will take over (all `wchar_t` bytes in native byte order):

```
#include <iostream>  
  
int main()
```

```
{  
    std::wofstream out("test.dat");  
    out << L"This is a test \x00DF";  
}
```

On a big-endian machine with a 2 byte wchar_t
the resulting file in hex is:

```
00 54 00 68 00 69 00 73 00 20 00 69 00 73 00 20  
00 61 00 20 00 74 00 65 00 73 00 74 00 20 00 DF
```

Extending codecvt by derivation

The facet codecvt can still be customized if you are on a platform that does not support a file system, or if you do not wish to use data files for other reasons. Naturally, you can derive from `codecvt` and override each of the virtual methods in a portable manner as specified by the C++ standard. Additionally you can take advantage of the MSL C++ specific classes used to implement `codecvt_byname`. There are five implementation specific facets that you can use in place of `codecvt` or `codecvt_byname` to get the behavior of one of the five encodings:

- `__ucs_2`
- `__jis`
- `__shift_jis`
- `__euc`
- `__utf_8`

These classes are templated simply on the internal character type (and should be instantiated with `wchar_t`). The external character type is implicitly `char`, and the state type is implicitly `mbstate_t`.

Note in “[An example use of __utf_8 is:](#)” that this locale (and `wofstream`) will have all of the facets of the current global locale except that its `codecvt<wchar_t, char, mbstate_t>` will use the UTF-8 encoding scheme. Thus the binary contents of the file is (in hex):

Listing 7.10 An example use of __utf_8 is:

```
#include <iostream>  
#include <fstream>  
  
int main()  
{  
    std::locale loc(std::locale(), new std::__utf_8<wchar_t>);  
}
```

```
    std::wofstream out;
    out.imbue(loc);
    out.open("test.dat");
    out << L"This is a test \x00DF";
}
```

Result

```
54 68 69 73 20 69 73 20 61 20 74 65 73 74 20 C3 9F
```

The Numeric Category

A class for numeric formatting and manipulation for locales.

Template Class Num_get

A class for formatted numeric input.

Num_get Members

The class num_get includes specific functions for parsing and formatting of numbers.

get

The function `get` is overloaded for un-formatted input.

```
iter_type get(iter_type in, iter_type end,
ios_base& str,ios_base::iostate& err,long& val) const;

iter_type get(iter_type in, iter_type end,
ios_base& str,ios_base::iostate& err,
unsigned short& val) const;

iter_type get(iter_type in, iter_type end,
ios_base& str,ios_base::iostate& err,unsigned int& val) const;

iter_type get(iter_type in, iter_type end,
ios_base& str,ios_base::iostate& err, unsigned long& val) const;

iter_type get(iter_type in, iter_type end,
ios_base& str,ios_base::iostate& err, short& val) const;

iter_type get(iter_type in, iter_type end,
ios_base& str,ios_base::iostate& err, double& val) const;

iter_type get(iter_type in, iter_type end,
ios_base& str,ios_base::iostate& err, long double& val) const;
```

```
iter_type get(iter_type in, iter_type end,  
ios_base& str, ios_base::iostate& err, void*& val) const;
```

Remarks

returns an iterator type.

Num_get Virtual Functions

Implements the relative versions of the `get` function

```
iter_type do_get(iter_type in, iter_type end,
ios_base& str, ios_base::iostate& err, long& val) const;

iter_type do_get(iter_type in, iter_type end,
ios_base& str, ios_base::iostate& err,
unsigned short& val) const;

iter_type do_get(iter_type in, iter_type end,
ios_base& str, ios_base::iostate& err,
nsigned int& val) const;

iter_type do_get(iter_type in, iter_type end,
ios_base& str, ios_base::iostate& err,
unsigned long& val) const;

iter_type do_get(iter_type in, iter_type end,
ios_base& str, ios_base::iostate& err,
float& val) const;

iter_type do_get(iter_type in, iter_type end,
ios_base& str, ios_base::iostate& err, double& val) const;

iter_type do_get(iter_type in, iter_type end,
ios_base& str, ios_base::iostate& err, long double& val) const;
```

```
iter_type do_get(iter_type in, iter_type end,
ios_base& str,ios_base::iostate& err, void*& val) const;

iiter_type do_get(iter_type in, iter_type end, ios_base& str,
ios_base::iostate& err, bool& val) const;
```

Remarks

Implements the relative versions of `get`.

Template Class Num_put

A class for formatted numeric output.

Num_put Members

The class `num_put` includes specific functions for parsing and formatting of numbers.

put

The function `put` is overloaded for un-formatted output.

```
iter_type put(iter_type out, ios_base& str,
char_type fill, bool val) const;
```

```
iter_type put(iter_type out, ios_base& str,
char_type fill, long val) const;
```

```
iter_type put(iter_type out, ios_base& str,
char_type fill,unsigned long val) const;
```

```
iter_type put(iter_type out, ios_base& str,
char_type fill, double val) const;
```

```
iter_type put(iter_type out, ios_base& str,
char_type fill,long double val) const;
```

```
iter_type put(iter_type out, ios_base& str,
char_type fill const void* val) const;
```

Num_put Virtual Functions

Implementation functions for `put`.

```
iter_type do_put(iter_type out, ios_base& str,
char_type fill, bool val) const;
iter_type do_put(iter_type out, ios_base& str,
char_type fill, long val) const;
iter_type do_put(iter_type out, ios_base& str,
char_type fill, unsigned long val) const;
iter_type do_put(iter_type out, ios_base& str,
char_type fill, double val) const;
iter_type do_put(iter_type out, ios_base& str,
char_type fill, long double val) const;
iter_type do_put(iter_type out, ios_base& str,
char_type fill, const void* val) const;
```

The Numeric Punctuation Facet

A facet for numeric punctuation in formatting and parsing.

Template Class Numpunct

A class for numeric punctuation conversion.

Numpunct Members

The template class `numpunct` provides various functions for punctuation localizations.

decimal_point

Determines the character used for a decimal point.

```
char_type decimal_point() const;
```

Remarks

Returns the character used for a decimal point.

thousands_sep

Determines the character used for a thousand separator.

```
char_type thousands_sep() const;
```

Remarks

Returns the character used for the thousand separator.

grouping

Describes the thousand separators.

```
string grouping() const;
```

Remarks

Returns a string describing the thousand separators.

truename

Determines the localization for “true”.

```
string_type truename() const;
```

Remarks

Returns a string describing the localization of the word “true”.

falsename

Determines the localization for “false”.

```
string_type falsename() const;
```

Remarks

Returns a string describing the localization of the word “false”.

numpunct virtual functions

Implementation of the public functions.

```
char_type do_decimal_point() const;
```

Implements decimal_point.

```
string_type do_thousands_sep() const;
```

Implements thousands_sep.

```
string do_grouping() const;
```

Implements grouping.

```
string_type do_truename() const;
```

Implements truename.

```
string_type do_falsename() const;
```

Implements falsename.

Template Class `Numpunct_byname`

The facet `numpunct` is responsible for specifying the punctuation used for parsing and formatting numeric quantities. You can specify the decimal point character, the thousands separator, the grouping, and the spelling of true and false. If you construct `numpunct_byname` with a `const char*` that refers to a file, then that file is scanned by `numpunct_byname`'s constructor for information to customize the encoding.

```
numpunct_byname<char> np( "en_US" );
```

If the file "en_US" exists, has `numpunct` data in it, and there are no syntax errors in the data, then `np` will behave as dictated by that data. If the file exists, but does not have `numpunct` data in it, then the facet will behave as if it were constructed with "C". If the file has `numpunct` data in it, but there is a syntax error in the data, or if the file does not exist, then a `std::runtime_error` is thrown.

For `numpunct_byname<char>`, the `numpunct` data section begins with:

```
$numeric_narrow
```

For `numpunct_byname<wchar_t>`, the `numpunct` data section begins with:

```
$numeric_wide
```

The syntax for both the narrow and wide data sections is the same. There are five keywords that allow you to specify the different parts of the `numpunct` data:

1. ["decimal_point"](#)
2. ["thousands_sep"](#)
3. ["grouping"](#)
4. ["false_name and true_name"](#)

You enter data with one of these keywords, followed by an equal sign '=' , and then the data. You can specify any or all of the 5 keywords. Data not specified will default to that of the "C" locale. The first two keywords (decimal_point and thousands_sep) have character data associated with them. See the rules for "[Character Syntax](#)" for details. The last three keywords have string data associated with them. See the rules for "[String Syntax](#)".

decimal_point

The decimal point data is a single character, as in:

```
decimal_point = '.'
```

thousands_sep

The character to be used for the thousands separator is specified with `thousands_sep`, as in:

```
thousands_sep = ',', '
```

grouping

The grouping string specifies the number of digits to group, going from right to left. For example, the grouping: 321 means that the number 12345789 would be printed as in:

```
1,2,3,4,56,789
```

The above grouping string can be specified as:

```
grouping = 321
```

A grouping string of “0” or “” means: do not group.

false_name and true_name

The names of false and true can be specified with `false_name` and `true_name`. For example:

```
false_name = "no way"  
true_name   = sure
```

Numeric_wide

For \$numeric_wide, wide characters can be represented with the hex or universal format (e.g. "\u64D0").

Listing 7.11 Example of Numeric_wide use

Given the data file:

```
$numeric_narrow  
decimal_point = ','  
thousands_sep = '.'  
grouping = 32  
false_name = nope  
true_name = sure
```

```
#include <sstream>  
#include <iostream>  
#include <iomanip>  
  
int main()  
{  
    std::locale loc("my_loc");  
    std::cout.imbue(loc);  
    std::istringstream in("1.23.456 nope 1.23.456,789");  
    in.imbue(loc);  
    in >> std::boolalpha;  
    long i;  
    bool b;  
    double d;  
    in >> i >> b >> d;  
    std::cout << i << '\n'  
        << std::boolalpha << !b << '\n'  
        << std::fixed << d;  
}
```

The output is:
1.23.456
sure

1.23.456,789000

Extending numpunct by derivation

It is easy enough to derive from numpunct and override the virtual functions in a portable manner. But numpunct also has a non-standard protected interface that you can take advantage of if you wish. There are five protected data members:

```
char_type __decimal_point_;
char_type __thousands_sep_;
string     __grouping_;
string_type __truename_;
string_type __falsename_;
```

A derived class could set these data members in its constructor to whatever is appropriate, and thus not need to override the virtual methods.

Listing 7.12 Example of numpunct<char>

```
struct mypunct: public std::numpunct<char>
{
    mypunct();
};

mypunct::mypunct()
{
    __decimal_point_ = ',';
    __thousands_sep_ = '.';
    __grouping_ = "\3\2";
    __falsename_ = "nope";
    __truename_ = "sure";
}

int main()
{
    std::locale loc(std::locale(), new mypunct);
    std::cout.imbue(loc);
    // ...
}
```

The Collate Category

The Template class collate used for the comparison and manipulation of strings.

Collate Members

Member functions used for comparison and hashing of strings.

compare

Lexicographical comparison of strings.

```
int compare(const charT* low1, const charT* high1,  
const charT* low2, const charT* high2) const;
```

Remarks

A value of 1 is returned if the first is lexicographically greater than the second. A value of negative 1 is returned if the second is greater than the first. A value of zero is returned if the strings are the same.

transform

Provides a string object to be compared to other transformed strings.

```
string_type transform  
(const charT* low, const charT* high) const;
```

Remarks

The `transform` member function is used for comparison of a series of strings.

Returns a string for comparison.

hash

Determines the hash value for the string.

```
long hash(const charT* low, const charT* high) const;
```

Remarks

Returns the hash value of the string

Collate Virtual Functions

Localized implementation functions for public collate member functions.

```
int do_compare  
(const charT* low1, const charT* high1,  
const charT* low2, const charT* high2) const;  
Implements compare.  
  
string_type do_transform(const charT* low, const charT* high)  
const;  
Implements transform  
  
long do_hash(const charT* low, const charT* high) const;  
Implements hash.
```

Template Class Collate_byname

The facet collate is responsible for specifying the sorting rules used for sorting strings. The base class collate does a simple lexical comparison on the binary values in the string. collate_byname can perform much more complex comparisons that are based on the Unicode sorting algorithm. If you construct collate_byname with a `const char*` that refers to a file, then that file is scanned by `collate_byname`'s constructor for information to customize the collation rules.

```
collate_byname<char> col("en_US");
```

If the file "en_US" exists, has collate data in it, and there are no syntax errors in the data, then col will behave as dictated by that data. If the file exists, but does not have collate data in it, then the facet will behave as if it were constructed with "C". If the file has collate data in it, but there is a syntax error in the data, or if the file does not exist, then a `std::runtime_error` is thrown.

Collate Data Section

For `collate_byname<char>`, the collate data section begins with:

```
$collate_narrow
```

For `collate_byname<wchar_t>`, the collate data section begins with:

```
$collate_wide
```

The syntax for both the narrow and wide data sections is the same. The data consists of a single string that has a syntax very similar to Java's RuleBasedCollator class. This syntax is designed to provide a level three sorting key consistent with the sorting algorithm specified by the Unicode collation algorithm.

Rule Format

The collation string rule is composed of a list of collation rules, where each rule is of three forms:

```
< modifier >
< relation > < text-argument >
< reset >      < text-argument >
```

Text-Argument:

A text-argument is any sequence of characters, excluding special characters (that is, common whitespace characters and rule syntax characters). If those characters are desired, you can put them in single quotes (e.g. ampersand => '&').

Modifier:

There is a single modifier which is used to specify that all accents (secondary differences) are backwards.

'@': Indicates that accents are sorted backwards, as in French.

Relation:

The relations are the following:

- '<': Greater, as a letter difference (primary)
- ';' : Greater, as an accent difference (secondary)
- ', ': Greater, as a case difference (tertiary)
- '=' : Equal

Reset:

There is a single reset which is used primarily for expansions, but which can also be used to add a modification at the end of a set of rules.

- '&': Indicates that the next rule follows the position to where the reset text-argument would be sorted.

Relational

The relational allow you to specify the relative ordering of characters. For example, the following string expresses that 'a' is less than 'b' which is less than 'c':

"< a < b < c"

For the time being, just accept that a string should start with '<'. That rule will be both relaxed and explained later.

Many languages (including English) consider 'a' < 'A', but only as a tertiary difference. And such minor differences are not considered significant unless more important differences are found to be equal. For example consider the strings:

- aa
- Aa
- ab

Since 'a' < 'A', then "aa" < "Aa". But "Aa" < "ab" because the difference between the second characters 'a' and 'b' is more important than the difference between the first characters 'A' and 'a'. This type of relationship can be expressed in the collation rule with:

"< a, A < b, B < c, C"

This says that 'a' is less than 'A' by a tertiary difference, and then 'b' and 'B' are greater than 'a' and 'A' by a primary difference (and similarly for 'c' and 'C').

Accents are usually considered secondary differences. For example, lower case e with an acute accent might be considered to be greater than lower case e, but only by a secondary difference. This can be represented with a semicolon like:

"... < e, E ; é, É < ..."

Note that characters can be entered in hexadecimal or universal format. They can also be quoted with single quotes (for example 'a'). If it is ambiguous whether a character is a command or a text argument, adding quotes specifies that it is a text argument.

Characters not present in a rule are implicitly ordered after all characters that do appear in a rule.

French collation

Normally primary, secondary and tertiary differences are considered left to right. But in French, secondary differences are considered right to left. This can be specified in the rule string by starting it with '@':

"@ ... < e, E ; é, É < ..."

Contraction

Some languages sort groups of letters as a single character. Consider the two strings: "acha" and "acia". In English they are sorted as just shown. But Spanish requires "ch" to be considered a single character that is sorted after 'c' and before 'd'. Thus the order in Spanish is reversed relative to English (that is "acia" < "acha"). This can be specified like:

"... < c < ch < d ..."

Taking case into account, you can expand this idea to:

"... < c, C < ch, CH, Ch, CH < d, D ..."

Expansion

Some languages expand a single character into multiple characters for sorting purposes. For example in English the ligature 'æ' might be sorted as 'a' followed by 'e'. To represent this in a rule, the reset character (&) is used. The idea is to reset the

current sorting key to an already entered value, and create multiple entries for the ligature. For example:

```
"... < a < b < c < d < e ... < z & a = æ & e = æ ..."
```

This rule resets the sort key to that of 'a', and then enters 'æ'. Then resets the sort key to that of 'e' and enters 'æ' again. This rule says that 'æ' is exactly equivalent to 'a' followed by 'e'. Alternatively ';' could have been used instead of '='. This would have made "æ" less than "æ" but only by a secondary difference.

Ignorable Characters

Characters in the rule before the first '<' are ignorable. That is they are not considered during the primary sorting at all (it is as if they aren't even there). Accents and punctuation are often marked as ignorable, but given a non-ignorable secondary or tertiary weight. For example, the default Java rule starts out with:

```
"= '\u200B'=\u200C=\u200D=\u200E=\u200F ...
"; '\u0020'; '\u00A0'..."
```

This completely ignores the first five characters (formatting control), and ignores except for secondary differences the next two characters (spacing characters).

This is why all example rules up till now started with '<' (so that none of the characters would be ignorable).

In the [“Example of locale sorting.”](#) notice how the space character was entered using quotes to disambiguate it from insignificant white space. The program below creates a vector of strings and sorts them both by “binary order” (just using string's operator <), and by the custom rule above using a locale as the sorting key.

Listing 7.13 Example of locale sorting

Assume the file "my_loc" has the following data in it:

```
$collate_narrow
"; - = ' '
< a, A < b, B < c, C
< ch, cH, Ch, CH
< d, D < e, E < f, F
< g, G < h, H < i, I
< j, J < k, K < l, L
< ll, lL, Ll, LL
< m, M < n, N < o, O
< p, P < q, Q < r, R
< s, S < t, T < u, U
< v, V < w, W < x, X
```

< Y, Y < z, Z"

```
#include <iostream>
#include <algorithm>
#include <vector>
#include <string>
#include <iostream>

int main()
{
    std::vector<std::string> v;
    v.push_back("aaaaaaaaB");
    v.push_back("aaaaaaaaA");
    v.push_back("AaaaaaaB");
    v.push_back("AaaaaaaA");
    v.push_back("blackbird");
    v.push_back("black-bird");
    v.push_back("black bird");
    v.push_back("blackbirds");
    v.push_back("acia");
    v.push_back("acha");
    std::ostream_iterator<std::string> out(std::cout, "\n");
    std::cout << "Binary order:\n\n";
    std::sort(v.begin(), v.end());
    std::copy(v.begin(), v.end(), out);
    std::cout << '\n';
    std::locale loc("my_loc");
    std::sort(v.begin(), v.end(), loc);
    std::cout << "Customized order:\n\n";
    std::copy(v.begin(), v.end(), out);
    std::cout << '\n';
}
```

The output is:
Binary order:

AaaaaaaA
AaaaaaaB
aaaaaaaaA
aaaaaaaaB
acha
acia
black bird
black-bird

blackbird

blackbirds

Customized order:

aaaaaaA
AaaaaaA
aaaaaaB
AaaaaaB
acia
acha
blackbird
black-bird
black bird
blackbirds

Extending collate by derivation

The behavior of collate can still be customized if you are on a platform that does not support a file system, or if you do not wish to use data files for other reasons.

Naturally, you can derive from collate and override each of the virtual methods in a portable manner as specified by the C++ standard. Additionally you can take advantage of the MSL C++ specific protected interface of collate_byname if you wish (to make your job easier if portability is not a concern).

The class collate_byname has one protected data member:

```
__collation_rule<charT> rule_;
```

Listing 7.14 The class std::__collation_rule interface:

```
template <class charT>
class __collation_rule
{
    struct value
    {
        charT primary;
        charT secondary;
        charT tertiary;
        ;
    };

public:
    struct entry
        : value
```

```
{  
    unsigned char length;  
};  
  
__collation_rule();  
explicit __collation_rule(const basic_string<charT>& rule);  
void set_rule(const basic_string<charT>& rule);  
entry operator()(const charT* low,  
                  const charT* high, int& state) const;  
bool is_french() const;  
bool empty() const;  
};
```

Most of this interface is to support `collate_byname`. If you simply derive from `collate_byname`, set the rule with a string, and let `collate_byname` do all the work, then there is really very little you have to know about `__collation_rule`.

A `__collation_rule` can be empty (contain no rule). In that case `collate_byname` will use `collate`'s sorting rule. This is also the case if `collate_byname` is constructed with "C". And once constructed, `__collation_rule`'s rule can be set or changed with `set_rule`. That is all you need to know to take advantage of all this horsepower!

Listing 7.15 Example of a `__collation_rule`:

```
#include <iostream>  
#include <locale>  
#include <string>  
  
struct my_collate  
    : public std::collate_byname<char>  
{  
    my_collate();  
};  
  
my_collate::my_collate()  
    : std::collate_byname<char>("C")  
{  
    rule_.set_rule("< a = A < b = B < c = C  
                  < d = D < e = E < f = F"  
                  "< g = G < h = H < i = I"  
                  "< j = J < k = K < l = L"  
                  "< m = M < n = N < o = O"  
                  "< p = P < q = Q < r = R"  
                  "< s = S < t = T < u = U"
```

```

        "< v = V < w = W < x = X"
        "< y = Y < z = Z");
}

int main()
{
    std::locale loc(std::locale(), new my_collate);
    std::string s1("Arnold");
    std::string s2("arnold");
    if (loc(s1, s2))
        std::cout << s1 << " < " << s2 << '\n';
    else if (loc(s2, s1))
        std::cout << s1 << " > " << s2 << '\n';
    else
        std::cout << s1 << " == " << s2 << '\n';
}

```

The custom facet `my_collate` derives from `std::collate_byname<char>` and sets the rule in its constructor. That's all it has to do. For this example, a case-insensitive rule has been constructed. The output of this program is:

Arnold == arnold

Alternatively, you could use `my_collate` directly (this is exactly what MSL C++'s locale does):

Listing 7.16 Example of custom facet `my_collate`:

```

int main()
{
    my_collate col;
    std::string s1("Arnold");
    std::string s2("arnold");
    switch (col.compare(s1.data(), s1.data()+s1.size(),
                        s2.data(), s2.data()+s2.size()))
    {
        case -1:
            std::cout << s1 << " < " << s2 << '\n';
            break;
        case 0:
            std::cout << s1 << " == " << s2 << '\n';
            break;
        case 1:
            std::cout << s1 << " > " << s2 << '\n';
            break;
    }
}

```

```
}
```

The output of this program is also:
Arnold == arnold

The Time Category

The facets `time_get` and `time_put` are conceptually simple: they are used to parse and format dates and times in a culturally sensitive manner. But as is not uncommon, there can be a lot of details. And for the most part, the standard is quiet about the details, leaving much of the behavior of these facets in the “implementation defined” category. Therefore this document not only discusses how to extend and customize the time facets, but it also explains much of the default behavior as well.

Time_get Members

The facet `time_get` has 6 member functions:

- `date_order`
- `get_time`
- `get_date`
- `get_weekday`
- `get_monthname`
- `get_year`

```
dateorder date_order() const;
```

Determines how the date, month and year are ordered.

Returns an enumeration representing the date, month, year order. Returns zero if it is un-ordered.

```
iter_type get_time  
(iter_type s, iter_type end, ios_base& str,  
ios_base::iostate& err, tm* t) const;
```

Determines the localized time.

Returns an iterator immediately beyond the last character recognized as a valid time.

```
iter_type get_date  
(iter_type s, iter_type end, ios_base& str,  
ios_base::iostate& err, tm* t) const;
```

Determines the localized date.

Returns an iterator immediately beyond the last character recognized as a valid date.

```
iter_type get_weekday  
(iter_type s, iter_type end, ios_base& str,  
ios_base::iostate& err, tm* t) const;
```

Determines the localized weekday.

Returns an iterator immediately beyond the last character recognized as a valid weekday.

```
iter_type get_monthname  
(iter_type s, iter_type end, ios_base& str,  
ios_base::iostate& err, tm* t) const;
```

Determines the localized month name.

Returns an iterator immediately beyond the last character recognized as a valid month name.

```
iter_type get_year(iter_type s, iter_type end,  
ios_base& str, ios_base::iostate& err,  
tm* t) const;
```

Determines the localized year.

Returns an iterator immediately beyond the last character recognized as a valid year.

Time_get Virtual Functions

The facet time_get has 6 protected virtual members:

- do_date_order
- do_get_time
- do_get_date
- do_get_weekday
- do_get_monthname
- do_get_year

```
dateorder do_date_order() const;
```

The method `do_date_order` returns `no_order`. This result can be changed via derivation.

```
iter_type do_get_time(iter_type s, iter_type end,
ios_base& str, ios_base::iostate& err,
tm* t) const;
```

The method `do_get_time` parses time with the format:

```
"%H:%M:%S"

iter_type do_get_date
(iter_type s, iter_type end, ios_base& str,
ios_base::iostate& err, tm* t) const;
```

The method `do_get_date` parses a date with the format:

```
"%A %B %d %T %Y"
```

This format string can be changed via the named locale facility, or by derivation.

```
iter_type do_get_weekday
(iter_type s, iter_type end, ios_base& str,
ios_base::iostate& err, tm* t) const;
```

The method `do_get_weekday` parses with the format:

```
"%A"
```

Although the format string can only be changed by derivation, the names of the weekdays themselves can be changed via the named locale facility or by derivation.

```
iter_type do_get_monthname
(iter_type s, iter_type end, ios_base& str,
ios_base::iostate& err, tm* t) const;
```

The method `do_get_monthname` parses with the format:

```
"%B"
```

Although the format string can only be changed by derivation, the names of the months themselves can be changed via the named locale facility or by derivation.

```
iter_type do_get_year
(iter_type s, iter_type end, ios_base& str,
ios_base::iostate& err, tm* t) const;
```

The method `do_get_year` parses a year with the format:

"%Y"

This behavior can only be changed by derivation.

The details of what these formats mean can be found in the [“Format/Parsing Table”](#).

In addition to the above mentioned protected methods, MSL C++ provides a non-standard, non-virtual protected method:

```
iter_type __do_parse(iter_type in, iter_type end,
ios_base& str, ios_base::iostate& err,
const basic_string<charT>& pattern, tm* t) const;
```

This method takes the parameters typical of the standard methods, but adds the pattern parameter of type `basic_string`. The pattern is a general string governed by the rules outlined in the section [“Format Parsing”](#). Derived classes can make use of this method to parse patterns not offered by `time_get`.

Listing 7.17 Derived classes example:

```
template <class charT, class InputIterator>
typename my_time_get<charT, InputIterator>::iter_type
my_time_get<charT, InputIterator>::do_get_date_time(
    iter_type in, iter_type end, std::ios_base& str,
    std::ios_base::iostate& err, std::tm* t) const
{
    const std::ctype<charT>& ct = std::use_facet<std::ctype<charT>>
        (str.getloc());
    return __do_parse(in, end, str, err, ct.widen("%c"), t);
}
```

Format Parsing

These commands follow largely from the C90 and C99 standards. However a major difference here is that most of the commands have meaning for parsing as well as formatting, whereas the C standard only uses these commands for formatting. The pattern string consists of zero or more conversion specifiers and ordinary characters (`char` or `wchar_t`). A conversion specifier consists of a `%` character, possibly followed by an `E` or `O` modifier character (described below), followed by a character that determines the behavior of the conversion specifier. Ordinary characters (non-conversion specifiers) must appear in the source string during parsing in the appropriate place or failbit gets set. On formatting, ordinary characters are sent to the output stream unmodified.

The `E` modifier can appear on any conversion specifier. But it is ignored for both parsing and formatting.

The `O` modifier can appear on any conversion specifier. It is ignored for parsing, but effects the following conversion specifiers on output by not inserting leading zeroes: `%C`, `%d`, `%D`, `%F`, `%g`, `%H`, `%I`, `%j`, `%m`, `%M`, `%S`, `%U`, `%V`, `%W`, `%y`

Table 7.4 Format/Parsing Table

Modifier	Parse	Format
<code>%a</code>	Reads one of the locale's weekday names. The name can either be the full name, or the abbreviated name. Case is significant. On successful parsing of one of the weekday names, sets <code>tm_wday</code> , otherwise sets <code>failbit</code> . For parsing, this format is identical to <code>%A</code> .	Outputs the locale's abbreviated weekday name as specified by <code>tm_wday</code> . The "C" locale's abbreviated weekday names are: Sun, Mon, Tue, Wed, Thu, Fri, Sat.
<code>%A</code>	For parsing, this format is identical to <code>%a</code> .	Outputs the locale's full weekday name as specified by <code>tm_wday</code> . The "C" locale's full weekday names are: Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday.
<code>%b</code>	Reads one of the locale's month names. The name can either be the full name, or the abbreviated name. Case is significant. On successful parsing of one of the month names, sets <code>tm_mon</code> , otherwise sets <code>failbit</code> . For parsing, this format is identical to <code>%B</code> .	Outputs the locale's abbreviated month name as specified by <code>tm_mon</code> . The "C" locale's abbreviated month names are: Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, Dec.
<code>%B</code>	For parsing, this format is identical to <code>%b</code> .	Outputs the locale's full month name as specified by <code>tm_mon</code> . The "C" locale's full month names are: January, February, March, April, May, June, July, August, September, October, November, December.
<code>%c</code>	Reads the date-and-time as specified by the current locale. The "C" locale specification is " <code>%A %B %d %T %Y</code> ". On successful parsing this sets <code>tm_wday</code> , <code>tm_mon</code> , <code>tm_mday</code> , <code>tm_sec</code> , <code>tm_min</code> , <code>tm_hour</code> and <code>tm_year</code> . If the entire pattern is not successfully parsed, then no <code>tm</code> members are set and <code>failbit</code> is set.	Outputs the locale's date-and-time. The "C" locale's date-and-time format is " <code>%A %B %d %T %Y</code> ". This information is specified by <code>tm_wday</code> , <code>tm_mon</code> , <code>tm_mday</code> , <code>tm_sec</code> , <code>tm_min</code> , <code>tm_hour</code> and <code>tm_year</code> .
<code>%C</code>	This is not a valid parse format. If <code>%C</code> is used in a parse pattern, a <code>runtime_error</code> is thrown.	Outputs the current year divided by 100. Single digit results will be pre-appended with '0' unless the <code>O</code> modifier is used.

Table 7.4 Format/Parsing Table

Modifier	Parse	Format
%d	Reads the day of the month. The result must be in the range [1, 31] else failbit will be set. Upon successful parsing tm_mday is set. For parsing, this format is identical to %e.	Outputs the day of the month as specified by tm_mday. Single digit results will be pre-appended with '0' unless the O modifier is used.
%D	Is equivalent to "%m/%d/%y".	Is equivalent to "%m/%d/%y". If the O modifier is used, is equivalent to "%Om/%Od/%y".
%e	Reads the day of the month. The result must be in the range [1, 31] else failbit will be set. Upon successful parsing tm_mday is set. For parsing, this format is identical to %d.	Outputs the day of the month as specified by tm_mday. Single digit results will be pre-appended with a space.
%F	Is equivalent to "%Y-%m-%d" (the ISO 8601 date format).	Is equivalent to "%Y-%m-%d". If the O modifier is used, is equivalent to "%Y-%Om-%Od".
%g	This is not a valid parse format. If %g is used in a parse pattern, a runtime_error is thrown.	Outputs the last 2 digits of the ISO 8601 week-based year . Single digit results will be pre-appended with '0' unless the O modifier is used. Specified by tm_year, tm_wday and tm_yday.
%G	This is not a valid parse format. If %G is used in a parse pattern, a runtime_error is thrown.	Outputs the ISO 8601 week-based year . Specified by tm_year, tm_wday and tm_yday.
%h	Is equivalent to %b.	Is equivalent to %b.
%H	Reads the hour (24-hour clock) as a decimal number. The result must be in the range [0, 23] else failbit will be set. Upon successful parsing tm_hour is set.	Outputs the hour (24-hour clock) as specified by tm_hour. Single digit results will be pre-appended with '0' unless the O modifier is used.
%I	Reads the hour (12-hour clock) as a decimal number. The result must be in the range [1, 12] else failbit will be set. Upon successful parsing tm_hour is set. This format is usually used with %p to specify am/pm. If a %p is not parsed with the %I, am is assumed.	Outputs the hour (12-hour clock) as specified by tm_hour. Single digit results will be pre-appended with '0' unless the O modifier is used.
%j	This is not a valid parse format. If %j is used in a parse pattern, a runtime_error is thrown.	Outputs the day of the year as specified by tm_yday in the range [001, 366]. If the O modifier is used, leading zeroes are suppressed.

Table 7.4 Format/Parsing Table

Modifier	Parse	Format
%m	Reads the month as a decimal number. The result must be in the range [1, 12] else failbit will be set. Upon successful parsing tm_mon is set.	Outputs the month as specified by tm_mon as a decimal number in the range [1, 12]. Single digit results will be pre-appended with '0' unless the O modifier is used.
%M	Reads the minute as a decimal number. The result must be in the range [0, 59] else failbit will be set. Upon successful parsing tm_min is set.	Outputs the minute as specified by tm_min as a decimal number in the range [0, 59]. Single digit results will be pre-appended with '0' unless the O modifier is used.
%n	Is equivalent to '\n'. A newline must appear in the source string at this position else failbit will be set.	Is equivalent to '\n'. A newline is output.
%p	Reads the locale's designation for am or pm. If neither of these strings are parsed then failbit will be set. A successful read will modify tm_hour, but only if %l is successfully parsed in the same parse pattern.	Outputs the locale's designation for am or pm, depending upon the value of tm_hour. The "C" locale's designations are am and pm.
%r	Reads the 12-hour time as specified by the current locale. The "C" locale specification is "%l:%M:%S %p". On successful parsing this sets tm_hour, tm_min, and tm_sec. If the entire pattern is not successfully parsed, then no tm members are set and failbit is set.	Outputs the locale's 12-hour time. The "C" locale's date-and-time format is "%l:%M:%S %p". This information is specified by tm_hour, tm_min, and tm_sec.
%R	Is equivalent to "%H:%M".	Is equivalent to "%H:%M". If the O modifier is used, is equivalent to "%OH:%M".
%S	: Reads the second as a decimal number. The result must be in the range [0, 60] else failbit will be set. Upon successful parsing tm_sec is set.	Outputs the second as specified by tm_sec as a decimal number in the range [0, 60]. Single digit results will be pre-appended with '0' unless the O modifier is used.
%t	Is equivalent to '\t'. A tab must appear in the source string at this position else failbit will be set.	Is equivalent to '\t'. A tab is output.
%T	Is equivalent to "%H:%M:%S".	Is equivalent to "%H:%M:%S". If the O modifier is used, is equivalent to "%OH:%M:%S".

Table 7.4 Format/Parsing Table

Modifier	Parse	Format
%u	Reads the ISO 8601 weekday as a decimal number [1, 7], where Monday is 1. If the result is outside the range [1, 7] failbit will be set. Upon successful parsing tm_wday is set.	Outputs tm_wday as the ISO 8601 weekday in the range [1, 7] where Monday is 1.
%U	This is not a valid parse format. If %U is used in a parse pattern, a runtime_error is thrown.	Outputs the week number of the year (the first Sunday as the first day of week 1) as a decimal number in the range [00, 53] using tm_year, tm_wday and tm_yday. If the O modifier is used, any leading zero is suppressed.
%V	This is not a valid parse format. If %V is used in a parse pattern, a runtime_error is thrown.	Outputs the ISO 8601 week-based year week number in the range [01, 53]. Specified by tm_year, tm_wday and tm_yday. If the O modifier is used, any leading zero is suppressed.
%w	Reads the weekday as a decimal number [0, 6], where Sunday is 0. If the result is outside the range [0, 6] failbit will be set. Upon successful parsing tm_wday is set.	Outputs tm_wday as the weekday in the range [0, 6] where Sunday is 0.
%W	This is not a valid parse format. If %W is used in a parse pattern, a runtime_error is thrown.	Outputs the week number in the range [00, 53]. Specified by tm_year, tm_wday and tm_yday. The first Monday as the first day of week 1. If the O modifier is used, any leading zero is suppressed.
%x	Reads the date as specified by the current locale. The “C” locale specification is “%A %B %d %Y”. On successful parsing this sets tm_wday, tm_mon, tm_mday, and tm_year. If the entire pattern is not successfully parsed, then no tm members are set and failbit is set.	Outputs the locale's date. The “C” locale's date format is “%A %B %d %Y”. This information is specified by tm_wday, tm_mon, tm_mday, and tm_year.
%X	Reads the time as specified by the current locale. The “C” locale specification is “%H:%M:%S”. On successful parsing this sets tm_hour, tm_min, and tm_sec. If the entire pattern is not successfully parsed, then no tm members are set and failbit is set.	Outputs the locale's time. The “C” locale's time format is “%H:%M:%S”. This information is specified by tm_hour, tm_min, and tm_sec.
%y	Reads the year as a 2 digit number. The century is specified by the locale. The “C” locale specification is 20 (the 21st century). On successful parsing this sets tm_year. If the year is not successfully parsed, then tm_year is not set and failbit is set.	Outputs the last two digits of tm_year. Single digit results will be pre-appended with '0' unless the O modifier is used.

Table 7.4 Format/Parsing Table

Modifier	Parse	Format
%Y	Reads the year. On successful parsing this sets tm_year. If the year is not successfully parsed, then tm_year is not set and failbit is set.	Outputs the year as specified by tm_year. (e.g. 2001)
%z	Reads the offset from UTC in the ISO 8601 format "-0430" (meaning 4 hours 30 minutes behind UTC, west of Greenwich). Two strings are accepted according to the current locale, one indicating Daylight Savings Time is not in effect, the other indicating it is in effect. Depending upon which string is read, tm_isdst will be set to 0 or 1. If the locale's designations for these strings are zero length, then no parsing is done and tm_isdst is set to -1. If the locale has non-empty strings for the UTC offset and neither is successfully parsed, failbit is set.	Outputs the UTC offset according to the current locale and the setting of tm_isdst (if non-negative). The "C" locale's designation for these strings is "" (an empty string).
%Z	: Reads the time zone name. Two strings are accepted according to the current locale, one indicating Daylight Savings Time is not in effect, the other indicating it is in effect. Depending upon which string is read, tm_isdst will be set to 0 or 1. If the locale's designations for these strings are zero length, then no parsing is done and tm_isdst is set to -1. If the locale has non-empty strings for the time zone names and neither is successfully parsed, failbit is set.	Outputs the time zone according to the current locale and the setting of tm_isdst (if non-negative). The "C" locale's designation for these strings is "" (an empty string).
%%	A % must appear in the source string at this position else failbit will be set	A % is output.
% followed by a space	One or more white space characters are parsed in this position. White space is determined by the locale's ctype facet. If at least one white space character does not exist in this position, then failbit is set.	A space (' ') for output.

ISO 8601 week-based year

The %g, %G, and %V give values according to the ISO 8601 week-based year. In this system, weeks begin on a Monday and week 1 of the year is the week that includes January 4th, which is also the week that includes the first Thursday of the year, and is also the first week that contains at least four days in the year. If the first Monday of

January is the 2nd, 3rd, or 4th, the preceding days are part of the last week of the preceding year; thus, for Saturday 2nd January 1999, %G is replaced by 1998 and %v is replaced by 53. If December 29th, 30th, or 31st is a Monday, it and any following days are part of week 1 of the following year. Thus, for Tuesday 30th December 1997, %G is replaced by 1998 and %v is replaced by 1.

Template Class Time_get_byname

A class used for locale time manipulations.

Listing 7.18 Template class time_get_byname

```
namespace std {
template <class charT,
class InputIterator = istreambuf_iterator<charT> >
class time_get_byname
    : public time_get<charT, InputIterator>
{
public:

    typedef time_base::dateorder dateorder;
    typedef InputIterator iter_type;

    explicit time_get_byname(const char* std_name, size_t refs = 0);

protected:
    virtual ~time_get_byname() ;
};

}
```

Template Class Time_put

The time_put facet format details are described in the listing [“Format/Parsing Table”](#).

Listing 7.19 Template Class Time_put Synopsis

```
namespace std {
template <class charT, class OutputIterator =
ostreambuf_iterator<charT> > class time_put
    : public locale::facet
{
public:
```

```
typedef charT           char_type;
typedef OutputIterator  iter_type;

explicit time_put(size_t refs = 0);

iter_type put(iter_type out,
    ios_base& str, char_type fill, const tm* tmb,
const charT* pattern, const charT* pat_end) const;
iter_type put(iter_type out, ios_base& str, char_type fill,
const tm* tmb, char format, char modifier = 0) const;

static locale::id id;

protected:
    virtual ~time_put();
    virtual iter_type do_put(iter_type out,
        ios_base& str, char_type fill, const tm* tmb,
        char format, char modifier) const;
};

}
```

Time_put Members

The class time_put has one member function.

```
iter_type put(iter_type s, ios_base& str,
char_type fill, const tm* t, const charT* pattern, const charT*
pat_end) const;

iter_type put(iter_type s, ios_base& str,
char_type fill, const tm* t, char format,
char modifier = 0) const;
```

Remarks

Formats a localized time.

Returns an iterator immediately beyond the last character.

Time_put Virtual Functions

The class time_put has one virtual member function.

```
iter_type do_put(iter_type s, ios_base&,
char_type fill, const tm* t, char format,
char modifier) const;
```

Remarks

Implements the public member function `put`.

Template Class Time_put_byname Synopsis

```
namespace std {
template <class charT, class OutputIterator =
ostreambuf_iterator<charT> >
class time_put_byname
    : public time_put<charT, OutputIterator>
{
public:
    typedef charT char_type;
    typedef OutputIterator iter_type;

    explicit time_put_byname(const char* std_name, size_t refs = 0);
protected:
    virtual ~time_put_byname();
};
```

Extending The Behavior Of The Time Facets

The time facets can easily be extended and customized for many different cultures. To stay portable one can derive from `time_get` and `time_put` and re-implement the behavior described above. Or one could take advantage of the MSL C++ implementation of these classes and build upon the existing functionality quite easily. Specifically you can easily alter the following data in the MSL time facets:

- The abbreviations of the weekday names
- The full weekday names

- The abbreviations of the month names
- The full month names
- The date-and-time format pattern (what %c will expand to)
- The date format pattern (what %x will expand to)
- The time format pattern (what %X will expand to)
- The 12 hour time format pattern (what %r will expand to)
- The strings used for AM/PM
- The strings used for the UTC offset
- The strings used for time zone names
- The default century to be used when parsing %y

Extending locale by using named locale facilities

The easiest way to specify the locale specific data is to use the named locale facilities. When you create a named locale with a string that refers to a locale data file, the time facets parse that data file for time facet data.

```
locale loc("my_locale");
```

The narrow file “my_locale” can hold time data for both narrow and wide time facets. Wide characters and strings can be represented in the narrow file using hexadecimal or universal format (e.g. '\u06BD'). Narrow time data starts with the keyword:

```
$time_narrow
```

And wide time data starts with the keyword:

```
$time_wide
```

Otherwise, the format for the time data is identical for the narrow and wide data.

There are twelve keywords that allow you to enter the time facet data:

1. abrev_weekday
2. weekday
3. abrev_monthname
4. monthname
5. date_time
6. am_pm
7. time_12hour

8. date
9. time
10. time_zone
11. utc_offset
12. default_century

You enter data with one of these keywords, followed by an equal sign '=', and then the data. You can specify any or all of the 12 keywords in any order. Data not specified will default to that of the "C" locale.

NOTE See "[String Syntax](#)" for syntax details.

abrev_weekday

This keyword allows you to enter the abbreviations for the weekday names. There must be seven strings that follow this keyword, corresponding to Sun through Sat. The "C" designation is:

```
abrev_weekday = Sun Mon Tue Wed Thu Fri Sat
```

weekday

This keyword allows you to enter the full weekday names. There must be seven strings that follow this keyword, corresponding to Sunday through Saturday. The "C" designation is:

```
weekday = Sunday Monday Tuesday Wednesday Thursday Friday  
Saturday
```

abrev_monthname

This keyword allows you to enter the abbreviations for the month names. There must be twelve strings that follow this keyword, corresponding to Jan through Dec. The "C" designation is:

```
abrev_monthname = Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov  
Dec
```

monthname

This keyword allows you to enter the full month names. There must be twelve strings that follow this keyword, corresponding to January through December. The “C” designation is:

```
monthname =  
    January February March April May June July  
    August September October November December
```

date_time

This keyword allows you to enter the parsing/formatting string to be used when %c is encountered. The “C” locale has:

```
date_time = "%A %B %d %T %Y"
```

The date_time string must not contain %c, else an infinite recursion will occur.

am_pm

This keyword allows you to enter the two strings that designate AM and PM. The “C” locale specifies:

```
am_pm = am pm
```

time_12hour

This keyword allows you to enter the parsing/formatting string to be used when %r is encountered. The “C” locale has:

```
time_12hour = "%I:%M:%S %p"
```

The time_12hour string must not contain %r, else an infinite recursion will occur.

date

This keyword allows you to enter the parsing/formatting string to be used when %x is encountered. The “C” locale has:

```
date = "%A %B %d %Y"
```

The date string must not contain %x, else an infinite recursion will occur.

time

This keyword allows you to enter the parsing/formatting string to be used when %X is encountered. The “C” locale has:

```
time = "%H:%M:%S"
```

The time string must not contain %X, else an infinite recursion will occur.

time_zone

This keyword allows you to enter two strings that designate the names of the locale's time zones: the first being the name for the time zone when Daylight Savings Time is not in effect, and the second name for when it is. The “C” locale has:

```
time_zone = " " "
```

This means that time zone information is not available in the “C” locale.

utc_offset

This keyword allows you to enter two strings that designate the UTC offsets of the locale's time zones: the first being the offset for the time zone when Daylight Savings Time is not in effect, and the second string for when it is. The “C” locale has:

```
utc_offset = " " "
```

This means that UTC offset information is not available in the “C” locale.

default_century

This keyword allows you to enter the default century which is used to create the correct year when parsing the %y format. This format parses a number and then computes the year by adding it to 100*default_century. The “C” locale has:

```
default_century = 20
```

Assume a Date class. The I/O for the Date class can be written using time_get and time_put in a portable manner. The input operator might look like:

Listing 7.20 Date Class Example Use

```
template<class charT, class traits>
std::basic_istream<charT, traits>&
operator >>(std::basic_istream<charT, traits>& is, Date& item)
{
    typename std::basic_istream<charT, traits>::sentry ok(is);
    if (ok)
```

```
{  
    std::ios_base::iostate err = std::ios_base::goodbit;  
    try  
    {  
        const std::time_get<charT>& tg =  
            std::use_facet<std::time_get<charT>>  
            (is.getloc());  
        std::tm t;  
        tg.get_date(is, 0, is, err, &t);  
        if (!(err & std::ios_base::failbit))  
            item = Date(t.tm_mon+1, t.tm_mday, t.tm_year+1900);  
    }  
    catch (...)  
    {  
        err |= std::ios_base::badbit | std::ios_base::failbit;  
    }  
    is.setstate(err);  
}  
    return is;  
}
```

The code extracts the time_get facet from the istream's locale and uses its get_date method to fill a tm. If the extraction was successful, then the data is transferred from the tm into the Date class.

Listing 7.21 The output method

```
template<class charT, class traits>  
std::basic_ostream<charT, traits>&  
operator <<(std::basic_ostream<charT, traits>& os, const Date& item)  
{  
    std::basic_ostream<charT, traits>::sentry ok(os);  
    if (ok)  
    {  
        bool failed;  
        try  
        {  
            const std::time_put<charT>& tp =  
                std::use_facet<std::time_put<charT>>  
                (os.getloc());  
            std::tm t;  
            t.tm_mday = item.day();  
            t.tm_mon = item.month() - 1;  
            t.tm_year = item.year() - 1900;  
            t.tm_wday = item.dayOfWeek();  
        }
```

```
    charT pattern[2] = {'%', 'x'};
    failed = tp.put(os, os, os.fill(), &t, pattern,
                    pattern+2).failed();
}
catch (...)
{
    failed = true;
}
if (failed)
    os.setstate(std::ios_base::failbit |
                std::ios_base::badbit);
}
return os;
}
```

After extracting the time_put facet from the ostream's locale, you transfer data from your Date class into the tm (or the Date class could simply export a tm). Then the put method is called with the tm and using the pattern "%x". There are several good things about the Date's I/O methods:

- They are written in portable standard C++.
- They are culturally sensitive since they use the locale's time facets.
- They can handle narrow or wide streams.
- The streams can be in memory (e.g. stringstream) or file based streams (fstream)
- For wide file streams, routing is automatically going through a codecvt that could (for example) be using something like UTF-8 to convert to/from the external file.
- They are relatively simple considering the tremendous flexibility involved.

With the Date's I/O done, the rest of the example is very easy. A French locale can be created with the following data in a file named "French":

```
$time_narrow
date = "%A, le %d %B %Y"
weekday =
    dimanche lundi mardi mercredi jeudi vendredi samedi
abrev_weekday =
    dim lun mar mer jeu ven sam
monthname = j
    janvier février mars avril mai juin juillet août
    septembre octobre novembre décembre
abrev_monthname =
    jan fév mar avr mai juin juil aoû sep oct nov déc
```

Now a program can read and write Date's in both English and French (and the Date class is completely ignorant of both languages).

Listing 7.22 Example of dates in English and French

```
#include <locale>
#include <iostream>
#include <sstream>
#include "Date.h"

int
main()
{
    std::istringstream in("Saturday February 24 2001");
    Date today;
    in >> today;
    std::cout.imbue(std::locale("French"));
    std::cout << "En Paris, c'est " << today << '\n';
    std::cout.imbue(std::locale("US"));
    std::cout << "But in New York it is " << today << '\n';
}
```

This program reads in a Date using the “C” locale from an `istringstream`. Then `cout` is imbued with “French” and the same Date is written out. And finally the same stream is imbued again with a “US” locale and the same Date is written out again. The output is:

En Paris, c'est samedi, le 24 février 2001

But in New York it is Saturday February 24 2001

For this example the "US" locale was implemented with an empty file. This was possible since the relevant parts of the "US" locale coincide with the "C" locale.

Extending by derivation

The behavior of the time facets can still be customized if you are on a platform that does not support a file system, or if you do not wish to use data files for other reasons. Naturally, you can derive from `time_get` and `time_put` and override each of the virtual methods in a portable manner as specified by the C++ standard. Additionally you can take advantage of the MSL C++ implementation if you wish (to make your job easier if portability is not a concern).

The central theme of the MSL time facets design is a non-standard facet class called `std::timepunct`:

Listing 7.23 Template Class Timepunct Synopsis

```
template <class charT>
class timepunct
    : public locale::facet
{
public:
    typedef charT           char_type;
    typedef basic_string<charT> string_type;

    explicit timepunct(size_t refs = 0);

    const string_type& abrev_weekday(int wday) const
        {return __weekday_names_[7+wday];}
    const string_type& weekday(int wday) const
        {return __weekday_names_[wday];}
    const string_type& abrev_monthname(int mon) const
        {return __month_names_[12+mon];}
    const string_type& monthname(int mon) const {
        return __month_names_[mon];}
    const string_type& date_time() const
        {return __date_time_;}
    const string_type& am_pm(int hour) const
        {return __am_pm_[hour/12];}
    const string_type& time_12hour() const
```

```
{           return __12hr_time_; }
const string_type& date() const
{return __date__;}
const string_type& time() const
{return __time__;}
const string_type& time_zone(int isdst) const
{return __time_zone_[isdst];}
const string_type& utc_offset(int isdst) const
{return __utc_offset_[bool(isdst)];}
int           default_century() const
{return __default_century__;}

static locale::id id;

protected:
virtual ~timepunct() {}

string_type __weekday_names_[14];
string_type __month_names_[24];
string_type __am_pm_[2];
string_type __date_time__;
string_type __date__;
string_type __time__;
string_type __12hr_time__;
string_type __time_zone_[2];
string_type __utc_offset_[2];
int           __default_century__;
};

};
```

This class is analogous to `numpunct` and `moneypunct`. It holds all of the configurable data. The facets `time_get` and `time_put` refer to `timepunct` for the data and then behave accordingly. All of the data in `timepunct` is protected so that the constructor of a derived facet can set this data however it sees fit. The `timepunct` facet will set this data according to the “C” locale.

Both the full weekday names and the abbreviated weekday names are stored in `__weekday_names__`. The full names occupy the first seven elements of the array, and the abbreviated names get the last seven slots. Similarly for `__month_names__`.

The `__am_pm__` member holds the strings that represent AM and PM, in that order.

The `__date_time__` member holds the formatting/parsing string for the date-and-time. This is the member that gets queried when `%c` comes up. Do not put `%c` in this string or an infinite recursion will occur. The default for this string is “`%A %B %d %T %Y`”.

The `__date_` member holds the formatting/parsing string for the date. This is the member that gets queried when `%x` comes up. Do not put `%x` in this string or an infinite recursion will occur. The default for this string is “`%A %B %d %Y`”.

The `__time_` member holds the formatting/parsing string for the time. This is the member that gets queried when `%x` comes up. Do not put `%x` in this string or an infinite recursion will occur. The default for this string is “`%H:%M:%S`”.

The `__12hr_time_` member holds the formatting/parsing string for the 12-hour-time. This is the member that gets queried when `%r` comes up. Do not put `%r` in this string or an infinite recursion will occur. The default for this string is “`%I:%M:%S %p`”.

The `__time_zone_` member contains two strings. The first is the name of the time zone when Daylight Savings Time is not in effect. The second string is the name of the time zone when Daylight Savings Time is in effect. These can be used to parse or format the `tm_isdst` member of a `tm`. These strings may be empty (as they are in the “C” locale) which means that time zone information is not available.

The `__utc_offset_` member contains two strings. The first represents the UTC offset when Daylight Savings Time is not in effect. The second string is the offset when Daylight Savings Time is in effect. These can be used to parse or format the `tm_isdst` member of a `tm`. These strings may be empty (as they are in the “C” locale) which means that UTC offset information is not available.

The final member, `__default_century_` is an int representing the default century to assume when parsing a two digit year with `%y`. The value 19 represents the 1900's, 20 represent's the 2000's, etc. The default is 20.

It is a simple matter to derive from `timepunct` and set these data members to whatever you see fit.

Timepunct_byname

You can use `timepunct_byname` to get the effects of a named locale for time facets instead of using a named locale:

The `time_get_byname` and `time_put_byname` facets do not add any functionality over `time_get` and `time_put`.

Listing 7.24 Using Timepunct_byname

```
#include <locale>
#include <iostream>
#include <sstream>
#include "Date.h"
```

```
int
main()
{
    std::istringstream in("Saturday February 24 2001");
    Date today;
    in >> today;
    std::cout.imbue(std::locale(std::locale(),
        new std::timepunct_byname<char>("French")));
    std::cout << "En Paris, c'est " << today << '\n';
    std::cout.imbue(std::locale(std::locale(),
        new std::timepunct_byname<char>("US")));
    std::cout << "But in New York it is " << today << '\n';
}
```

This has the exact same effect as the named locale example.

But the timepunct_byname example still uses the files “French” and “US”. Below is an example timepunct derived class that avoids files but still captures the functionality of the above examples.

Listing 7.25 Example Timepunct Facet Use

```
// The first job is to create a facet derived from timepunct
// that stores the desired data in the timepunct:

class FrenchTimepunct
    : public std::timepunct<char>
{
public:
    FrenchTimepunct();
};

FrenchTimepunct::FrenchTimepunct()
{
    __date_ = "%A, le %d %B %Y";
    __weekday_names_[0] = "dimanche";
    __weekday_names_[1] = "lundi";
    __weekday_names_[2] = "mardi";
    __weekday_names_[3] = "mercredi";
    __weekday_names_[4] = "jeudi";
    __weekday_names_[5] = "vendredi";
    __weekday_names_[6] = "samedi";
    __weekday_names_[7] = "dim";
    __weekday_names_[8] = "lun";
    __weekday_names_[9] = "mar";
```

```
__weekday_names_[10] = "mer";
__weekday_names_[11] = "jeu";
__weekday_names_[12] = "ven";
__weekday_names_[13] = "sam";
__month_names_[0] = "janvier";
__month_names_[1] = "février";
__month_names_[2] = "mars";
__month_names_[3] = "avril";
__month_names_[4] = "mai";
__month_names_[5] = "juin";
__month_names_[6] = "juillet";
__month_names_[7] = "août";
__month_names_[8] = "septembre";
__month_names_[9] = "octobre";
__month_names_[10] = "novembre";
__month_names_[11] = "décembre";
__month_names_[12] = "jan";
__month_names_[13] = "fév";
__month_names_[14] = "mar";
__month_names_[15] = "avr";
__month_names_[16] = "mai";
__month_names_[17] = "juin";
__month_names_[18] = "juil";
__month_names_[19] = "aoû";
__month_names_[20] = "sep";
__month_names_[21] = "oct";
__month_names_[22] = "nov";
__month_names_[23] = "déc";
}

//Though tedious, the job is quite simple.
//Next simply use your facet:

int main()
{
    std::istringstream in("Saturday February 24 2001");
    Date today;
    in >> today;
    std::cout.imbue(std::locale(std::locale(),
        new FrenchTimepunct));
    std::cout << "En Paris, c'est " << today << '\n';
    std::cout.imbue(std::locale::classic());
    std::cout << "But in New York it is " << today << '\n';
}
```

Here we have explicitly asked for the classic locale, instead of the “US” locale since the two are the same (but executing `classic()` does not involve file I/O). Using the global locale (`locale()`) instead of `classic()` would have been equally fine in this example.

The Monetary Category

There are five standard money classes:

- class `money_base`;
- template <class `charT`, class `InputIterator` = `istreambuf_iterator<charT>` > class `money_get`;
- template <class `charT`, class `OutputIterator` = `ostreambuf_iterator<charT>` > class `money_put`;
- template <class `charT`, bool `International` = false> class `moneypunct`;
- template <class `charT`, bool `International` = false> class `moneypunct_byname`;

The first of these (`money_base`) is not a facet, but the remaining four are. The `money_base` class is responsible only for specifying pattern components that will be used to specify how monetary values are parsed and formatted (currency symbol first or last, etc.).

The facets `money_get` and `money_put` are responsible for parsing and formatting respectively. Though their behavior is made up of virtual methods, and thus can be overridden via derivation, it will be exceedingly rare for you to feel the need to do so. Like the numeric facets, the real customization capability comes with the “punct” classes: `moneypunct` and `moneypunct_byname`.

A user-defined Money class (there will be an example later on) can use `money_get` and `money_put` in defining its I/O, and remain completely ignorant of whether it is dealing with francs or pounds. Instead clients of Money will imbue a stream with a locale that specifies this information. On I/O the facets `money_get` and `money_put` query `moneypunct` (or `moneypunct_byname`) for the appropriate locale-specific data. The Money class can remain blissfully ignorant of cultural specifics, and at the same time, serve all cultures!

A sample Money class

The very reason that we can design a Money class before we know the details of `moneypunct` customization is because the Money class can remain completely ignorant of this customization. This Money class is meant only to demonstrate I/O. Therefore it is as simple as possible. We begin with a simple struct:

Listing 7.26 A example demonstration of input and output

```

struct Money
{
    long double amount_;
};

// The I/O methods for this class follow a fairly standard formula,
// but reference the money facets to do the real work:
template<class charT, class traits>
std::basic_istream<charT,traits>&
operator >>(std::basic_istream<charT,traits>& is, Money& item)
{
    typename std::basic_istream<charT,traits>::sentry ok(is);
    if (ok)
    {
        std::ios_base::iostate err = std::ios_base::goodbit;
        try
        {
            const std::money_get<charT>& mg =
                std::use_facet<std::money_get<charT> > (is.getloc());
            mg.get(is, 0, false, is, err, item.amount_);
        }
        catch (...)
        {
            err |= std::ios_base::badbit | std::ios_base::failbit;
        }
        is.setstate(err);
    }
    return is;
}

template<class charT, class traits>
std::basic_ostream<charT, traits>&
operator <<(std::basic_ostream<charT, traits>& os,
            const Money& item)
{
    std::basic_ostream<charT, traits>::sentry ok(os);
    if (ok)
    {
        bool failed;
        try
        {
            const std::money_put<charT>& mp =
                std::use_facet<std::money_put<charT> > (os.getloc());
            failed = mp.put(os, false, os, os.fill(),
                            item.amount_).failed();
        }
    }
}

```

```
    }
    catch (...)
    {
        failed = true;
    }
    if (failed)
        os.setstate(std::ios_base::failbit |
                    std::ios_base::badbit);
}
return os;
}
```

The extraction operator (`>>`) obtains a reference to `money_get` from the stream's locale, and then simply uses its `get` method to parse directly into `Money`'s `amount_`. The insertion operator (`<<`) does the same thing with `money_put` and its `put` method. These methods are extremely flexible, as all of the formatting details (save one) are saved in the stream's locale. That one detail is whether we are dealing a local currency format, or an international currency format. The above methods hard wire this decision to "local" by specifying `false` in the `get` and `put` calls. The `moneypunct` facet can store data for both of these formats. An example difference between an international format and a local format is the currency symbol. The US local currency symbol is "\$", but the international US currency symbol is "USD".

For completeness, we extend this example to allow client code to choose between local and international formats via a stream manipulator. See Matt Austern's excellent C/C++ Users Journal article: The Standard Librarian: User-Defined Format Flags for a complete discussion of the technique used here.

To support the manipulators, our simplistic `Money` struct is expanded in the following code example.

Listing 7.27 Example of manipulator support.

```
struct Money
{
    enum format {local, international};

    static void set_format(std::ios_base& s, format f)
        {flag(s) = f;}
    static format get_format(std::ios_base& s)
        {return static_cast<format>(flag(s));}
    static long& flag(std::ios_base& s);

    long double amount_;
```

```
};
```

An enum has been added to specify local or international format. But this enum is only defined within the Money class. There is no format data member within Money. That information will be stored in a stream by clients of Money. To aid in this effort, three static methods have been added: set_format, get_format and flag. The first two methods simply call flag which has the job of reading and writing the format information to the stream. Although flag is where the real work is going on, its definition is surprisingly simple.

Listing 7.28 Money class flag

```
long&
Money::flag(std::ios_base& s)
{
    static int n = std::ios_base::xalloc();
    return s.iword(n);
}
```

As described in Austern's C/C++ User Journal article, flag uses the stream's xalloc facility to reserve an area of storage which will be the same location in all streams. And then it uses iword to obtain a reference to that storage for a particular stream. Now it is easier to see how set_format and get_format are simply writing and reading a long associated with the stream s.

To round out this manipulator facility we need the manipulators themselves to allow client code to write statements like:

```
in >> international >> money;
out << local << money << '\n';
```

These are easily accomplished with a pair of namespace scope methods:

Listing 7.29 Money class manipulators

```
template<class charT, class traits>
std::basic_ios<charT, traits>&
local(std::basic_ios<charT, traits>& s)
{
    Money::set_format(s, Money::local);
    return s;
}
```

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

```
std::basic_ios<charT, traits>&
international(std::basic_ios<charT, traits>& s)
{
    Money::set_format(s, Money::international);
    return s;
}
```

And finally, we need to modify the Money inserter and extractor methods to read this information out of the stream, instead of just blindly specifying `false` (`local`) in the `get` and `put` methods.

Listing 7.30 Money class inserters and extractors

```
template<class charT, class traits>
std::basic_istream<charT, traits>&
operator >>(std::basic_istream<charT, traits>& is, Money& item)
{
    typename std::basic_istream<charT, traits>::sentry ok(is);
    if (ok)
    {
        std::ios_base::iostate err = std::ios_base::goodbit;
        try
        {
            const std::money_get<charT>& mg =
                std::use_facet<std::money_get<charT>>(is.getloc());
            mg.get(is, 0, Money::get_format(is) ==
                   Money::international, is, err, item.amount_);
        } catch (...)
        {
            err |= std::ios_base::badbit |
                std::ios_base::failbit;
        }
        is.setstate(err);
    }
    return is;
}

template<class charT, class traits>
std::basic_ostream<charT, traits>&
operator <<(std::basic_ostream<charT, traits>& os,
           const Money& item)
{
    std::basic_ostream<charT, traits>::sentry ok(os);
    if (ok)
    {
```

```

        bool failed;
        try
        {
            const std::money_put<charT>& mp =
                std::use_facet<std::money_put<charT>>(os.getloc());
            failed = mp.put(os, Money::get_format(os) ==
                            Money::international, os, os.fill(),
                            item.amount_).failed();
        }
        catch (...)
        {
            failed = true;
        }
        if (failed)
            os.setstate(std::ios_base::failbit |
                        std::ios_base::badbit);
    }
    return os;
}

```

Because we gave the enum `Money::local` the value 0, this has the effect of making `local` the default format for a stream.

We now have a simple `Money` class that is capable of culturally sensitive input and output, complete with local and international manipulators! To motivate the following sections on how to customize `moneypunct` data. Below is sample code that uses our `Money` class, along with the named locale facility:

Listing 7.31 Example of using a money class

```

int main()
{
    std::istringstream in("USD (1,234,567.89)");
    Money money;
    in >> international >> money;
    std::cout << std::showbase << local << money << '\n';
    std::cout << international << money << '\n';
    std::cout.imbue(std::locale("Norwegian"));
    std::cout << local << money << '\n';
    std::cout << international << money << '\n';
}

```

And the output is:
\$-1,234,567.89

```
USD (1,234,567.89)
-1 234 567,89 kr
NOK (1 234 567,89)
```

Template Class Money_get

The template class `Money_get` is used for `locale` monetary input routines.

Listing 7.32 Template Class Money_get Synopsis

```
namespace std {
template <class charT,
class InputIterator = istreambuf_iterator<charT> >
class money_get : public locale::facet {
public:
typedef charT char_type;
typedef InputIterator iter_type;
typedef basic_string<charT> string_type;
explicit money_get(size_t refs = 0);
iter_type get(iter_type s, iter_type end, bool intl,
ios_base& f, ios_base::iostate& err,
long double& units) const;
iter_type get(iter_type s, iter_type end, bool intl,
ios_base& f, ios_base::iostate& err,
string_type& digits) const;
static locale::id id;
protected:
~money_get(); //virtual
virtual iter_type do_get(iter_type, iter_type, bool, ios_base&,
ios_base::iostate& err, long double& units) const;
virtual iter_type do_get(iter_type, iter_type, bool, ios_base&,
ios_base::iostate& err, string_type& digits) const;
};
```

Money_get Members

Localized member functions for inputting monetary values.

get

Inputs a localized monetary value.

```
iter_type get(iter_type s, iter_type end,
    bool intl, ios_base& f, ios_base::iostate& err,
    long double& quant) const;

iter_type get( s, iter_type end, bool intl,
    ios_base& f, ios_base::iostate& err, string_type& quant) const;
```

Remarks

Returns an iterator immediately beyond the last character recognized as a valid monetary quantity.

Money_get Virtual Functions

Implementation functions for localization of the `money_get` public member functions.

```
iter_type do_get(iter_type s, iter_type end,
    bool intl, ios_base& str, ios_base::iostate& err,
    long double& units) const;

iter_type do_get(iter_type s, iter_type end,
    bool intl, ios_base& str, ios_base::iostate& err
    string_type& digits) const;
```

Remarks

Implements a localized monetary `get` function.

Template Class Money_put

The template class `money_put` is used for `locale` monetary output routines.

Listing 7.33 Template Class Money_put Synopsis

```
namespace std {
template <class charT,
class OutputIterator = ostreambuf_iterator<charT> >
class money_put : public locale::facet {
public:
typedef charT char_type;
typedef OutputIterator iter_type;
typedef basic_string<charT> string_type;
explicit money_put(size_t refs = 0);
iter_type put(iter_type s, bool intl, ios_base& f,
char_type fill, long double units) const;
iter_type put(iter_type s, bool intl, ios_base& f,
char_type fill, const string_type& digits) const;
static locale::id id;
protected:
~money_put(); //virtual
virtual iter_type
do_put(iter_type, bool, ios_base&, char_type fill,
long double units) const;
virtual iter_type
do_put(iter_type, bool, ios_base&, char_type fill,
const string_type& digits) const;
};
```

Money_put Members

Localized member functions for outputting monetary values.

put

Outputs a localized monetary value.

```
iter_type put(iter_type s, bool intl, ios_base& f,  
char_type fill, long double quant) const;  
  
iter_type put(iter_type s, bool intl, ios_base& f,  
char_type fill, const string_type& quant) const;
```

Remarks

Returns an iterator immediately beyond the last character recognized as a valid monetary quantity.

Money_put Virtual Functions

Implementation functions for localization of the `money_put` public member functions.

```
iter_type do_put(iter_type s, bool intl,  
ios_base& str, char_type fill,  
long double units) const;  
  
iter_type do_put(iter_type s, bool intl,  
ios_base& str, char_type fill,  
const string_type& digits) const;
```

Remarks

Implements a localized put function.

Class MoneyPunct

An object used for localization of monetary punctuation.

Listing 7.34 Template Class `Moneypunct` Synopsis

```
namespace std {
class money_base {
public:
enum part { none, space, symbol, sign, value };
struct pattern { char field[4]; };
};

template <class charT, bool International = false>
class moneypunct : public locale::facet, public money_base {
public:
typedef charT char_type;
typedef basic_string<charT> string_type;
explicit moneypunct(size_t refs = 0);
charT decimal_point() const;
charT thousands_sep() const;
string grouping() const;
string_type curr_symbol() const;
string_type positive_sign() const;
string_type negative_sign() const;
int frac_digits() const;
pattern pos_format() const;
pattern neg_format() const;
static locale::id id;
static const bool intl = International;
protected:
~moneypunct(); //virtual
virtual charT do_decimal_point() const;
virtual charT do_thousands_sep() const;
virtual string do_grouping() const;
virtual string_type do_curr_symbol() const;
virtual string_type do_positive_sign() const;
virtual string_type do_negative_sign() const;
virtual int do_frac_digits() const;
virtual pattern do_pos_format() const;
virtual pattern do_neg_format() const;
};
}
```

Moneypunct Members

Member functions to determine the punctuation used for monetary formatting.

decimal_point

Determines what character to use as a decimal point.

```
charT decimal_point() const;
```

Remarks

Returns a char to be used as a decimal point.

thousands_sep

Determines which character to use for a thousandths separator.

```
charT thousands_sep() const;
```

Remarks

The character to be used for the thousands separator is specified with thousands_sep.

Returns the character to use for a thousandths separator.

grouping

Determines a string that determines the grouping of thousands.

```
string grouping() const;
```

Remarks

The grouping string specifies the number of digits to group, going from right to left.

Returns the string that determines the grouping of thousands.

curr_symbol

Determines a string of the localized currency symbol.

```
string_type curr_symbol() const;
```

Remarks

Returns the string of the localized currency symbol.

positive_sign

Determines a string of the localized positive sign.

```
string_type positive_sign() const;
```

Remarks

Returns the string of the localized positive sign.

negative_sign

Determines a string of the localized negative sign.

```
string_type negative_sign() const;
```

Remarks

Returns the string of the localized negative sign.

frac_digits

Determines a string of the localized fractional digits.

```
int frac_digits() const;
```

Remarks

Returns the string of the localized fractional digits.

pos_format

Determines the format of the localized non-negative values.

```
pattern pos_format() const;
```

Remarks

These keywords allow you to enter the format for both positive and negative values. There are 5 keywords to specify a format:

- none
- space
- symbol
- sign
- value

A monetary format is a sequence of four of these keywords. Each value: symbol, sign, value, and either space or none appears exactly once. The value none, if present, is not first; the value space, if present, is neither first nor last. The behavior of breaking any of these rules is undefined. The default pattern for positive values, and for local and international formats is:

```
pos_format = symbol sign none value
```

Returns the format of the localized non-negative values.

neg_format

Determines the format of the localized non-negative values.

```
pattern neg_format() const;
```

Remarks

These keywords allow you to enter the format for both positive and negative values. There are 5 keywords to specify a format:

none

space

symbol

sign

value

A monetary format is a sequence of four of these keywords. Each value : symbol, sign, value, and either space or none appears exactly once. The value none, if present, is not first; the value space, if present, is neither first nor last. The behavior of breaking any of these rules is undefined. The default pattern for negative values, and for local and international formats is:

```
neg_format = symbol sign none value
```

Returns the format of the localized non-negative values.

Moneypunct Virtual Functions

Virtual functions that implement the localized public member functions.

```
charT do_decimal_point() const;
```

Implements decimal_point.

```
charT do_thousands_sep() const;
```

Implements thousands_sep.

```
string do_grouping() const;
```

Implements grouping.

```
string_type do_curr_symbol() const;
```

Implements cur_symbol.

```
string_type do_positive_sign() const;
```

Implements `positive_sign`.

```
string_type do_negative_sign() const;
```

```
do_negative_sign()
```

Implements `negative_sign`.

```
int do_frac_digits() const;
```

Implements `frac_digits`.

```
pattern do_pos_format() const;
```

Implements `pos_format`.

```
pattern do_neg_format() const;
```

Implements `neg_format`.

Extending `moneypunct` by derivation

It is easy enough to derive from `moneypunct` and override the virtual functions in a portable manner. But `moneypunct` also has a non-standard protected interface that you can take advantage of if you wish. There are nine protected data members:

```
charT      __decimal_point_;
charT      __thousands_sep_;
string     __grouping_;
string_type __cur_symbol_;
string_type __positive_sign_;
string_type __negative_sign_;
int        __frac_digits_;
pattern    __pos_format_;
pattern    __neg_format_;
```

A derived class could set these data members in its constructor to whatever is appropriate, and thus not need to override the virtual methods.

Listing 7.35 Extending `Moneypunct` by derivation

```
struct mypunct
    : public std::moneypunct<char, false>
{
```

```
    mypunct();
};

mypunct::mypunct()
{
    __decimal_point_ = ',';
    __thousands_sep_ = ' ';
    __cur_symbol_ = "kr";
    __pos_format_.field[0] = __neg_format_.field[0] = char(sign);
    __pos_format_.field[1] = __neg_format_.field[1] = char(value);
    __pos_format_.field[2] = __neg_format_.field[2] = char(space);
    __pos_format_.field[3] = __neg_format_.field[3] = char(symbol);
}

int
main()
{
    std::locale loc(std::locale(), new mypunct);
    std::cout.imbue(loc);
    // ...
}
```

Indeed, this is just what `moneypunct_byname` does after reading the appropriate data from a locale data file.

Template Class `Moneypunct_byname`

A template class for implementation of the `moneypunct` template class.

Listing 7.36 Template Class `Moneypunct_byname` Synopsis

```
namespace std {
template <class charT, bool Intl = false>
class moneypunct_byname : public moneypunct<charT, Intl> {
public:
    typedef money_base::pattern pattern;
    typedef basic_string<charT> string_type;
    explicit moneypunct_byname(const char*, size_t refs = 0);
protected:
    ~moneypunct_byname(); // virtual
    virtual charT do_decimal_point() const;
    virtual charT do_thousands_sep() const;
    virtual string do_grouping() const;
    virtual string_type do_curr_symbol() const;
```

```
virtual string_type do_positive_sign() const;
virtual string_type do_negative_sign() const;
virtual int do_frac_digits() const;
virtual pattern do_pos_format() const;
virtual pattern do_neg_format() const;
};

}
```

When a named locale is created:

```
std::locale my_loc("MyLocale");
```

this places the facet `moneypunct_byname ("MyLocale")` in the locale. The `moneypunct_byname` constructor considers the name it is constructed with as the name of a data file which may or may not contain `moneypunct` data. There are 4 keywords that mark the beginning of `moneypunct` data in a locale data file.

- `$money_local_narrow`
- `$money_international_narrow`
- `$money_local_wide`
- `$money_international_wide`

These data sections can appear in any order in the locale data file. And they are all optional. Any data not specified defaults to that of the "C" locale. Wide characters and strings can be represented in the narrow locale data file using hexadecimal or universal format (for example, '\u06BD'). See the rules for "[Strings and Characters in Locale Data Files](#)" for more syntax details.

Data file syntax

The syntax for entering `moneypunct` data is the same under all four keywords. There are 9 keywords that can be used within a `$money_XXX` data section to specify `moneypunct` data. The keywords can appear in any order and they are all optional.

- `decimal_point`
- `thousands_sep`
- `grouping`
- `curr_symbol`
- `positive_sign`
- `negative_sign`
- `frac_digits`

- pos_format
- neg_format

Each of these keywords is followed by an equal sign (=) and then the appropriate data (described below).

decimal_point

The decimal point data is a single character, as in:

```
decimal_point = '.'
```

Remarks

The default decimal point is ','

thousands_sep

The character to be used for the thousands separator is specified with thousands_sep, as in:

```
thousands_sep = ',',
```

Remarks

The default thousands separator is ','

grouping

The grouping string specifies the number of digits to group, going from right to left.

Remarks

For example, the grouping: 321 means that the number 12345789 would be printed as in:

```
1,2,3,4,56,789
```

The above grouping string can be specified as:

```
grouping = 321
```

A grouping string of "0" or "" means: don't group. The default grouping string is "3".

curr_symbol

The currency symbol is specified as a string by curr_symbol, as in:

```
curr_symbol = $
```

It is customary for international currency symbols to be four characters long, but this is not enforced by the `locale` facility. The default local currency symbols is "\$". The default international currency symbol is "USD".

positive_sign

The string to be used for the positive sign is specified by positive_sign. Many locales set this as the empty string, as in:

```
positive_sign = ""
```

Remarks

The default positive sign is the empty string.

negative_sign

The negative sign data is a string specified by negative_sign, as in:

```
negative_sign = ()
```

Remarks

The precise rules for how to treat signs that are longer than one character are laid out in the standard. Suffice it to say that this will typically enclose a negative value in parentheses.

The default negative sign for local formats is " - ", and for international formats is " () ".

frac_digits

The number of digits to appear after the decimal point is specified by `frac_digits`, as in:

```
frac_digits = 2
```

Remarks

The default value is 2.

pos_format / neg_format

These keywords allow you to enter the format for both positive and negative values.

Remarks

There are 5 keywords to specify a format:

none

space

symbol

sign

value

A monetary format is a sequence of four of these keywords. Each value: `symbol`, `sign`, `value`, and either `space` or `none` appears exactly once. The value `none`, if present, is not first; the value `space`, if present, is neither first nor last. The behavior of breaking any of these rules is undefined.

The default pattern for positive and negative values, and for local and international formats is:

```
pos_format = symbol sign none value  
neg_format = symbol sign none value
```

Notice that in the "[Example Data file.](#)" not all of the fields have been specified because the default values for these fields were already correct. On the other hand, it does not hurt to specify default data to improve (human) readability in the data file.

Listing 7.37 Example Data file

To have the example code run correctly, we need a file named "Norwegian" containing the following data:

```
$money_local_narrow  
decimal_point = ','  
thousands_sep = ''  
curr_symbol = kr  
pos_format = sign value space symbol  
neg_format = sign value space symbol  
  
$money_international_narrow  
decimal_point = ','  
thousands_sep = ''  
curr_symbol = "NOK "
```

The Message Retrieval Category

The messages facet is the least specified facet in the C++ standard. Just about everything having to do with messages is implementation defined.

Listing 7.38 Template Class Messages Synopsis

```
namespace std {  
class messages_base  
{  
public:  
    typedef int catalog;  
};  
  
template <class charT>  
class messages
```

```
: public locale::facet,
    public messages_base
{
public:
    typedef charT char_type;
    typedef basic_string<charT> string_type;

    explicit messages(size_t refs = 0);

    catalog open(const basic_string<char>& fn,
                 const locale& loc) const;
    string_type get(catalog c, int set, int msgid,
                    const string_type& dfault) const;
    void close(catalog c) const;

    static locale::id id;

protected:
    virtual ~messages();
    virtual catalog do_open(const basic_string<char>& fn,
                           const locale& loc) const;
    virtual string_type do_get(catalog c, int set, int msgid,
                               const string_type& dfault) const;
    virtual void do_close(catalog c) const;
};

}
```

The intent is that you can use this class to read messages from a catalog. There may be multiple sets of messages in a catalog. And each message set can have any number of int/string pairs. But beyond that, the standard is quiet.

Does the string fn in open refer to a file? If so, what is the format of the set/msgid/string data to be read in from the file? There is also a messages_byname class that derives from messages. What functionality does messages_byname add over messages?

Unfortunately the answers to all of these questions are implementation defined. This document seeks to answer those questions. Please remember that applications depending on these answers will probably not be portable to other implementations of the standard C++ library.

Messages Members

Public member functions for catalog message retrieval.

open

Opens a message catalog for reading

```
catalog open(const basic_string<char>& name,  
const locale& loc) const;
```

Remarks

Returns a value that may be passed to get to retrieve a message from a message catalog.

get

Retrieves a message from a message catalog.

```
string_type get(catalog cat, int set, int msgid,  
const string_type& default) const;
```

Remarks

Returns the message in the form of a string.

close

Closes a message catalog.

```
void close(catalog cat) const;
```

Messages Virtual Functions

Virtual functions used to localize the public member functions.

```
catalog do_open(const basic_string<char>& name,
const locale& loc) const;
Implements open.

string_type do_get(catalog cat, int set, int msgid,
const string_type& default) const;
Implements get.

void do_close(catalog cat) const;
Implements close.
```

MSL C++ implementation of messages

The Metrowerks standard C++ library has a custom implementation of messages.

```
Example code to open a catalog:
typedef std::messages<char> Msg;
const Msg& ct = std::use_facet<Msg>(std::locale::classic());
Msg::catalog cat = ct.open("my_messages", std::locale::classic());
if (cat < 0)
{
    std::cout << "Can't open message file\n";
    std::exit(1);
}
```

The first line simply type defines `messages<char>` for easier reading or typing. The second line extracts the messages facet from the “C” locale. The third line instructs the messages facet to look for a file named “`my_messages`” and read message set data out of it using the classic (“C”) locale (one could specify a locale with a specialized codecvt facet for reading the data file). If the file is not found, the open method returns -1. The facet `messages<char>` reads data from a narrow file (`ifstream`). The facet `messages<wchar_t>` reads data from a wide file (`wifstream`).

The messages data file can contain zero or more message data sets of the format:

- \$set setid
- msgid message
- msgid message
- msgid message
- ...

The keyword `$set` begins a message data set. The `set id` is the set number. It can be any int. Set `id`'s do not need to be contiguous. But the `set id` must be unique among the sets in this catalog.

The `msgid` is the message `id` number. It can be any int. Message `id`'s do not need to be contiguous. But the message `id` must be unique among the messages in this set.

The message is an optionally quoted (") string that is the message for this `setid` and `msgid`. If the message contains white space, it must be quoted. The message can have characters represented escape sequences using the hexadecimal or universal format. For example (see also "[String Syntax](#)"):

```
"\u0048\u0069\u0020\u0054\u0068\u0065\u0072\u0065\u0021"
```

The message data set terminates when the data is not of the form

```
msgid message
```

Thus, there are no syntax errors in this data. Instead, a syntax error is simply interpreted as the end of the data set. The catalog file can contain data other than message data sets. The messages facet will scan the file until it encounters `$set` `setid`.

Listing 7.39 Example of message facet

An example message data file might contain:

```
$set 1
1 "First Message"
2 "Error in foo"
3 Baboo
4 "\u0048\u0069\u0020\u0054\u0068\u0065\u0072\u0065\u0021"

$set 2
1 Ok
2 Cancel
```

A program that uses messages to read and output this file follows:

```
#include <locale>
#include <iostream>
```

```
int main()
{
    typedef std::messages<char> Msg;
    const Msg& ct = std::use_facet<Msg>(std::locale::classic());
    Msg::catalog cat = ct.open("my_messages",
        std::locale::classic());
    if (cat < 0)
    {
        std::cout << "Can't open message file\n";
        return 1;
    }
    std::string eof("no more messages");
    for (int set = 1; set <= 2; ++set)
    {
        std::cout << "set " << set << "\n\n";
        for (int msgid = 1; msgid < 10; ++msgid)
        {
            std::string msg = ct.get(cat, set, msgid, eof);
            if (msg == eof)
                break;
            std::cout << msgid << "\t" << msg << '\n';
        }
        std::cout << '\n';
    }
    ct.close(cat);
}
```

The output of this program is:
set 1

```
1 First Message
2 Error in foo
3 Baboo
4 Hi There!
```

set 2

```
1 Ok
2 Cancel
```

Template Class `Messages_byname` Synopsis

The class `messages_byname` adds no functionality over `messages`. The `const char*` that it is constructed with is ignored. To localize messages for a specific culture, either open a different catalog (file), or have different sets in a catalog represent messages for different cultures.

Listing 7.40 Template Class `Messages_byname` Synopsis

```
namespace std {
template <class charT>
class messages_byname : public messages<charT> {
public:
    typedef messages_base::catalog catalog;
    typedef basic_string<charT> string_type;
    explicit messages_byname(const char*, size_t refs = 0);
protected:
    ~messages_byname(); // virtual
    virtual catalog do_open(const basic_string<char>&, const locale&) const;
    virtual string_type do_get(catalog, int set, int msgid,
        const string_type& dfault) const;
    virtual void do_close(catalog) const;
};
```

Extending `messages` by derivation

If you are on a platform without file support, or you do not want to use files for messages for other reasons, you may derive from `messages` and override the virtual methods as described by the standard. Additionally you can take advantage of the MSL C++ specific protected interface of `messages` if you wish (to make your job easier if portability is not a concern).

The `messages` facet has the non-virtual protected member:

```
string_type& __set(catalog c, int set, int msgid);
```

You can use this to place the quadruple (`c`, `set`, `msgid`, `string`) into `messages'` database. The constructor of the derived facet can fill the database using multiple calls to `__set`. Below is an example of such a class. This example also overrides `do_open` to double check that the catalog name is a valid name, and then return the proper catalog number. And `do_close` is also overridden to do nothing. The `messages` destructor will reclaim all of the memory used by its database:

The main program (client code) in the [“Example of extending message by derivation.”](#) is nearly identical to the previous example. Here we simply create and use the customized messages facet. Alternatively we could have created a locale and installed this facet into it. And then extracted the facet back out of the locale using `use_facet` as in the first example.

Listing 7.41 Example of extending message by derivation

```
#include <locale>
#include <iostream>
#include <string>
#include <map>

class MyMessages
    : public std::messages<char>
{
public:
    MyMessages();
protected:
    virtual catalog do_open(const std::string& fn,
                           const std::locale&) const;
    virtual void    do_close(catalog) const {};
private:
    std::map<std::string, catalog> catalogs_;
};

MyMessages::MyMessages()
{
    catalogs_[ "my_messages" ] = 1;
    __set(1, 1, 1) = "set 1: first message";
    __set(1, 1, 2) = "set 1: second message";
    __set(1, 1, 3) = "set 1: third message";
    __set(1, 2, 1) = "set 2: first message";
    __set(1, 2, 2) = "set 2: second message";
    __set(1, 2, 3) = "set 2: third message";
}

MyMessages::catalog
MyMessages::do_open(const std::string& fn, const std::locale&) const
{
    std::map<std::string, catalog>::const_iterator i =
        catalogs_.find(fn);
    if (i == catalogs_.end())
        return -1;
    return i->second;
```

```
}

int main()
{
    typedef MyMessages Msg;
    Msg ct;
    Msg::catalog cat = ct.open("my_messages",
        std::locale::classic());
    if (cat < 0)
    {
        std::cout << "Can't open message file\n";
        return 1;
    }
    std::string eof("no more messages");
    for (int set = 1; set <= 2; ++set)
    {
        std::cout << "set " << set << "\n\n";
        for (int msgid = 1; msgid < 10; ++msgid)
        {
            std::string msg = ct.get(cat, set, msgid, eof);
            if (msg == eof)
                break;
            std::cout << msgid << "\t" << msg << '\n';
        }
        std::cout << '\n';
    }
    ct.close(cat);
}
```

The output of this program is:
set 1

```
1   set 1: first message
2   set 1: second message
3   set 1: third message
```

set 2

```
1   set 2: first message
2   set 2: second message
3   set 2: third message
```

Program-defined Facets

A C++ program may add its own locales to be added to and used the same as the built in facets. To do this derive a class from `locale::facet` with the static member `static locale::id id`.

C Library Locales

The C++ header `<clocale>` are the same as the C header `locale` but in standard namespace.

Table 7.5 Header `<clocale>` Synopsis

Type	Name(s)	Name(s)
Macro	LC_ALL	LC_COLLATE
Macro	LC_CTYPE	LC_MONETARY
Macro	LC_NUMERIC	LC_TIME
Macro	NULL	
Struct	lconv	
Function	localeconv	setlocale

Containers Library

Containers are used to store and manipulate collections of information.

The Containers library

The chapter is constructed in the following sub sections and mirrors clause 23 of the ISO (the International Organization for Standardization) C++ Standard :

- [“Container Requirements”](#)
- [“Sequences”](#)
- [“Associative Containers”](#)
- [“Template Class Bitset”](#)

Container Requirements

Container objects store other objects and control the allocation and de-allocation of those objects. There are five classes implementing these requirements.

- [“Template Class Deque”](#)
- [“Template Class List”](#)
- [“Container Adaptors”](#)
- [“Template Class Vector”](#)
- [“Class Vector<bool>”](#)

All containers must meet basic requirements.

The swap(), equal() and lexicographical_compare() algorithms are defined in the algorithm library for more information see [“The Algorithms Library”](#).

The member function `size()` returns the number of elements in a container.

The member function `begin()` returns an iterator to the first element and `end()` returns an iterator to the last element.

If `begin()` equals `end()` the container is empty.

Copy constructors for container types copy and allocator argument from their first parameter. All other constructors take an Allocator reference argument.

The member function `get_allocator()` returns a copy of the Allocator object used in construction of the container.

If an iterator type of the container is bi-directional or a random access iterator the container is reversible.

Unless specified containers meet these requirements.

If an exception is thrown by an `insert()` function while inserting a single element, that function has no effects.

If an exception is thrown by a `push_back()` or `push_front()` function, that function has no effects.

The member functions `erase()`, `pop_back()` or `pop_front()` do not throw an exception.

None of the copy constructors or assignment operators of a returned iterator throw an exception.

The member function `swap()` does not throw an exception. Except if an exception is thrown by the copy constructor or assignment operator of the container's compare object.

The member function `swap()` does not invalidate any references, pointers, or iterators referring to the elements of the containers being swapped.

Sequences Requirements

A sequence is a kind of container that organizes a finite set of objects, all of the same type, into a strictly linear arrangement.

The Library includes three kinds of sequence containers `vector`, `lists`, `deque` and `adaptors` classes

Additional Requirements

The iterator returned from `a.erase(q)` points to the element immediately following `q` prior to the element being erased.

If no prior element exists for `a.erase` then `a.end()` is returned.

- The previous conditions are true for `a.erase(q1, q2)` as well.

For every sequence defined in this clause the constructor

```
template <class InputIterator>
X(InputIterator f, InputIterator l,
    const Allocator& a = Allocator())
```

- shall have the same effect as:

```
X(static_cast<typename X::size_type>(f),
static_cast<typename X::value_type>(l), a)
```

- if `InputIterator` is an integral type.

Member functions in the forms:

```
template <class InputIterator>
rt fx1(iterator p, InputIterator f, InputIterator l);
template <class InputIterator>
rt fx2(InputIterator f, InputIterator l);
template <class InputIterator>
rt fx3(iterator i1, iterator i2, InputIterator f, InputIterator l);
```

- shall have the same effect, respectively, as:

```
fx1(p, static_cast<typename X::size_type>(f),
static_cast<typename X::value_type>(l));
fx2(static_cast<typename X::size_type>(f),
static_cast<typename X::value_type>(l));
fx3(i1, i2, static_cast<typename X::size_type>(f),
static_cast<typename X::value_type>(l));
```

- if `InputIterator` is an integral type.

The member function `at()` provides bounds-checked access to container elements.

The member function `at()` throws `out_of_range` if `n >= a.size()`.

Associative Containers Requirements

Associative containers provide an ability for optimized retrieval of data based on keys.

Associative container are parameterized on Key and an ordering relation. Furthermore, map and multimap associate an arbitrary type T with the key.

The phrase “equivalence of keys” means the equivalence relation imposed by the comparison and not the `operator ==` on keys.

An associative container supports both unique keys as well as support for equivalent keys.

- The classes set and map support unique keys.
- The classes multiset and multimap support equivalent keys.

An iterator of an associative container must be of the bidirectional iterator category.

The insert members shall not affect the validity of iterators.

Iterators of associative containers iterate through the containers in the non-descending order of keys where non-descending is defined by the comparison that was used to construct them.

Sequences

The sequence libraries consist of several headers.

- [“Template Class Deque”](#)
- [“Template Class List”](#)
- [“Container Adaptors”](#)
- [“Template Class Vector”](#)
- [“Class Vector<bool>”](#)

Template Class Deque

A deque is a kind of sequence that supports random access iterators. The deque class also supports insert and erase operations at the beginning middle or the end. However, deque is especially optimized for pushing and popping elements at the beginning and end.

A deque satisfies all of the requirements of a container and of a reversible container as well as of a sequence.

Constructors

The deque constructor creates an object of the class deque.

```
explicit deque(const Allocator& = Allocator());  
  
explicit deque(size_type n, const T& value = T(),  
const Allocator& = Allocator());  
  
template <class InputIterator>  
deque(InputIterator first, InputIterator last,  
const Allocator& = Allocator());
```

assign

The assign function is overloaded to allow various types to be assigned to a deque.

```
template <class InputIterator>  
  
void assign (InputIterator first, InputIterator last);  
  
void assign(size_type n, const T& t);
```

Deque Capacity

The class deque has one member function to resize the deque.

resize

This function resizes the deque.

```
void resize(size_type sz, T c = T());
```

Deque Modifiers

The deque class has member functions to modify the deque.

insert

The insert function is overloaded to insert a value into deque.

```
iterator insert(iterator position, const T& x);  
  
void insert  
  
(iterator position, size_type n, const T& x);  
  
template <class InputIterator>  
  
void insert  
  
(iterator position, InputIterator first,  
  
InputIterator last);
```

erase

An overloaded function that allows the removal of a value at a position.

```
iterator erase(iterator position);  
  
iterator erase(iterator first, iterator last);
```

Remarks

An iterator to the position erased.

Deque Specialized Algorithms

Deque has one specialize swap function.

swap

Swaps the element at one position with another.

```
template <class T, class Allocator>
void swap (deque<T,Allocator>& x, deque<T,Allocator>& y);
```

Template Class List

A list is a sequence that supports bidirectional iterators and allows insert and erase operations anywhere within the sequence.

In a list fast random access to list elements is not supported.

A list satisfies all of the requirements of a container as well as those of a reversible container and of a sequence except for `operator[]` and the member function `at` which are not included.

Constructors

The overloaded list constructors create objects of type list.

```
explicit list(const Allocator& = Allocator());
explicit list(size_type n, const T& value = T(),
const Allocator& = Allocator());

template <class InputIterator>
list(InputIterator first, InputIterator last,
const Allocator& = Allocator());
```

assign

The overloaded assign function allows values to be assigned to a list after construction.

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

void assign(size_type n, const T& t);
```

List Capacity

The list class provides for one member function to resize the list.

resize

Resizes the list.

```
void resize(size_type sz, T c = T());
```

List Modifiers

The list class has several overloaded functions to allow modification of the list object.

insert

The insert member function insert a value at a position.

```
iterator insert(iterator position, const T& x);  
  
void insert(iterator position, size_type n, const T& x);  
template <class InputIterator>  
  
void insert  
(iterator position, InputIterator first,InputIterator last);
```

push_front

The push_front member function pushes a value at the front of the list.

```
void push_front(const T& x);
```

push_back

The push_back member function pushes a value onto the end of the list.

```
void push_back(const T& x);
```

erase

The erase member function removes a value at a position or range.

```
iterator erase(iterator position);  
iterator erase(iterator first, iterator last);
```

Remarks

Returns an iterator to the last position.

pop_front

The pop_front member function removes a value from the top of the list.

```
void pop_front();
```

pop_back

The pop_back member function removes a value from the end of the list.

```
void pop_back();
```

clear

Clears a list by removing all elements.

```
void clear(); .
```

List Operations

The list class provides for operations to manipulate the list.

splice

Moves an element or a range of elements in front of a position specified.

```
void splice  
  (iterator position, list<T,Allocator>& x);  
  
void splice  
  (iterator position, list<T,Allocator>& x, iterator i);  
  
void splice  
  (iterator position, list<T,Allocator>& x,  
   iterator first, iterator last);
```

remove

Removes all element with a value.

```
void remove(const T& value);
```

remove_if

Removes all element for which the predicate is true.

```
template <class Predicate>  
void remove_if(Predicate pred);
```

unique

Removes duplicates of consecutive elements.

```
void unique();
```

```
template <class BinaryPredicate>  
void unique(BinaryPredicate binary_pred);
```

merge

Moves sorted elements into a list according to the compare argument.

```
void merge(list<T,Allocator>& x);
```

```
template <class Compare>  
void merge(list<T,Allocator>& x, Compare comp);
```

reverse

Reverses the order of the list.

```
void reverse();
```

sort

Sorts a list according to the Compare function or by less than value for the parameterless version.

```
void sort();
```

```
template <class Compare> void sort(Compare comp);
```

List Specialized Algorithms

The list class provides a swapping function.

swap

Changes the position of the first argument with the second argument.

```
template <class T, class Allocator>
```

```
void swap (list<T,Allocator>& x, list<T,Allocator>& y);
```

Container Adaptors

Container adaptors take a Container template parameter so that the container is copied into the Container member of each adaptor.

Template Class Queue

Any of the sequence types supporting operations front(), back(), push_back() and pop_front() can be used to instantiate queue.

operator ==

A user supplied operator for the queue class that compares the queue's data member.

```
bool operator ==
```

Remarks

Returns true if the data members are equal.

operator <

A user supplied operator for the queue class that compares the queue's data member.

```
bool operator <
```

Remarks

Returns true if the data member is less than the compared queue.

Template Class Priority_queue

You can instantiate any `priority_queue` with any sequence that has random access iterator and supporting operations `front()`, `push_back()` and `pop_back()`.

Instantiation of a `priority_queue` requires supplying a function or function object for making the priority comparisons.

Constructors

Creates an object of type priority_queue.

```
priority_queue(const Compare& x = Compare(),
               const Container& y = Container());
template <class InputIterator>
priority_queue
(InputIterator first, InputIterator last,
 const Compare& x = Compare(),
 const Container& y = Container());
```

priority_queue members

The class priority_queue provides public member functions for manipulation the priority_queue.

push

Inserts an element into the priority_queue.

```
void push(const value_type& x);
```

pop

Removes an element from a priority_queue.

```
void pop();
```

Template Class Stack

A stack class may be instantiated by any sequence supporting operations `back()`, `push_back()` and `pop_back()`.

Public Member Functions

Constructors

Creates an object of type stack with a container object.

```
explicit stack(const Container& = Container());
```

empty

Signifies when the stack is empty

```
bool empty() const;
```

Remarks

Returns true if there are no elements in the stack.

size

Gives the number of elements in a stack.

```
size_type size() const;
```

Remarks

Returns the number of elements in a stack.

top

Gives the top element in the stack.

```
value_type& top()  
const value_type& top() const
```

Remarks

Returns the value at the top of the stack.

push

Puts a value onto a stack.

```
void push(const value_type& x) { c.push_back(x); }
```

pop

Removes an element from a stack.

```
void pop()
```

Template Class Vector

A vector is a kind of sequence container that supports random access iterators. You can use insert and erase operations at the end and in the middle but at the end is faster.

A vector satisfies all of the requirements of a container and of a reversible container and of a sequence. It also satisfies most of the optional sequence requirements with the exceptions being `push_front` and `pop_front` member functions.

Constructors

The vector class provides overloaded constructors for creation of a vector object.

```
vector(const Allocator& = Allocator());  
  
explicit vector(size_type n, const T& value = T(),  
const Allocator& = Allocator());  
  
template <class InputIterator>  
vector(InputIterator first, InputIterator last,  
const Allocator& = Allocator());  
  
vector(const vector<T,Allocator>& x);
```

assign

The member function assign allows you to assign values to an already created object.

```
template <class InputIterator>  
void assign  
(InputIterator first, InputIterator last);  
void assign(size_type n, const T& t);
```

capacity

Tells the maximum number of elements the vector can hold.

```
size_type capacity() const;
```

Remarks

Returns the maximum number of elements the vector can hold.

resize

Resizes a vector if a second argument is give the elements are filled with that value.

```
void resize(size_type sz, T c = T());
```

Vector Modifiers

The vector class provides various member functions for vector data manipulation.

insert

The member function insert inserts a value or a range of values at a set position.

```
iterator insert(iterator position, const T& x);  
  
void insert(iterator position, size_type n, const T& x);  
  
template <class InputIterator> void insert  
(iterator position, InputIterator first, InputIterator last);
```

erase

Removes elements at a position or for a range.

```
iterator erase(iterator position);
```

```
iterator erase(iterator first, iterator last);
```

Vector Specialized Algorithms

The vector class provides for a specialized swap function.

swap

Swaps the data of one argument with the other argument.

```
template <class T, class Allocator> void swap  
(vector<T,Allocator>& x, vector<T,Allocator>& y);
```

Class Vector<bool>

A specialized vector for `bool` elements is provided to optimize allocated space.

A MSL bitvecotr class is available for efficient `bool` vecotr manipulations. Refer to [“The bitvector Class Library”](#) for more information.

Associative Containers

The associative container library consists of four template container classes.

- [“Template Class Map”](#)
- [“Template Class Multimap”](#)
- [“Template Class Set”](#)
- [“Template Class Multiset”](#)

Template Class Map

The map class is an associative container that supports unique keys and provides for retrieval of values of another type `T` based on the keys. The map template class supports bidirectional iterators.

The template class map satisfies all of the requirements of a normal container and those of a reversible container, as well as an associative container.

A map also provides operations for unique keys.

Constructors

The map class provides an overloaded constructor for creating an object of type map.

```
explicit map(const Compare& comp = Compare(),
const Allocator& = Allocator());

template <class InputIterator> map (InputIterator first,
InputIterator last, const Compare& comp = Compare(),
const Allocator& = Allocator());
```

Map Element Access

The map class includes an element access operator.

operator []

Access an indexed element.

```
T& operator[] (const key_type& x);
```

Remarks

Returns the value at the position indicated.

Map Operations

The map class includes member functions for map operations.

find

Finds an element based upon a key.

```
iterator find(const key_type& x);
```

```
const_iterator find(const key_type& x) const;
```

Remarks

Returns the position where the element is found.

lower_bound

Finds the first position where an element based upon a key would be inserted.

```
iterator lower_bound(const key_type& x);
```

```
const_iterator lower_bound(const key_type& x) const;
```

Remarks

Returns the first position where an element would be inserted.

upper_bound

Finds the last position where an element based upon a key would be inserted.

```
iterator upper_bound(const key_type& x);
```

```
const_iterator upper_bound(const key_type &x) const;
```

Remarks

Returns the last position where an element would be inserted.

equal_range

Finds both the first and last position in a range where an element based upon a key would be inserted.

```
pair<iterator, iterator> equal_range (const key_type &x);
```

```
pair<const_iterator, const_iterator> equal_range  
(const key_type& x) const;
```

Remarks

Returns a pair of elements representing a range for insertion.

Map Specialized Algorithms

The map class provides for a method to swap elements.

swap

Swaps the first argument with the second argument.

```
template <class Key, class T, class Compare, class Allocator>
void swap
(map<Key, T, Compare, Allocator>& x,
map<Key, T, Compare, Allocator>& y);
```

Template Class Multimap

A `multimap` container supports equivalent keys that may contain multiple copies of the same key value. Multimap provides for fast retrieval of values of another type based on the keys.

Multimap supports bidirectional iterators.

The `multimap` satisfies all of the requirements of a container, reversible container and associative containers.

Multimap supports the `a_eq` operations but not the `a_uniq` operations.

For a `multimap<Key, T>` the `key_type` is `Key` and the `value_type` is `pair<const Key, T>`

Constructors

The multimap constructor is overloaded for creation of a multimap object.

```
explicit multimap  
(const Compare& comp = Compare(),  
const Allocator& = Allocator());  
  
template <class InputIterator> multimap  
(InputIterator first, InputIterator last,  
const Compare& comp = Compare(),  
const Allocator& = Allocator()0;
```

Multimap Operations

The multimap class includes member functions for manipulation of multimap data.

find

Finds a value based upon a key argument.

```
iterator find(const key_type &x);  
  
const_iterator find(const key_type& x) const;
```

Remarks

Returns the position where the element is at.

lower_bound

Finds the first position where an element based upon a key would be inserted.

```
iterator lower_bound (const key_type& x);
```

```
const_iterator lower_bound (const key_type& x) const;
```

Remarks

Returns the position where an element was found.

equal_range

Finds the first and last positions where a range of elements based upon a key would be inserted.

```
pair<iterator, iterator> equal_range  
(const key_type& x);
```

```
pair<const_iterator, const_iterator> equal_range  
(const key_type& x) const;
```

Remarks

Returns a pair object that represents the first and last position where a range is found.

Multimap Specialized Algorithms

The multimap class provides a specialized function for swapping elements.

swap

Swaps the first argument for the last argument.

```
template <class Key, class T, class Compare, class Allocator>
void swap
(multimap<Key,T,Compare,Allocator>& x,
multimap<Key,T,Compare,Allocator>& y);
```

Template Class Set

The template class `set` is a container that supports unique keys and provides for fast retrieval of the keys themselves.

Set supports bidirectional iterators.

The class `set` satisfies all of the requirements of a container, a reversible container and an associative container.

A set supports the `a_uniq` operations but not the `a_eq` operations.

Constructors

The set class includes overloaded constructors for creation of a set object.

```
explicit set

(const Compare& comp = Compare(),
const Allocator& = Allocator());

template <class InputIterator> set

(InputIterator first, last,
const Compare& comp = Compare(),
const Allocator& = Allocator());
```

Set Specialized Algorithms

The set class specializes the swap function.

swap

Swaps the first argument with the second argument.

```
template <class Key, class Compare, class Allocator>

void swap

(set<Key, Compare, Allocator>& x,
set<Key, Compare, Allocator>& y);
```

Template Class Multiset

The template class multiset is an associative container that supports equivalent keys and retrieval of the keys themselves.

Multiset supports bidirectional iterators.

The multiset satisfies all of the requirements of a container, reversible container and an associative container.

A multiset supports the `a_eq` operations but not the `a_uniq` operations.

Constructors

The multiset class includes overloaded constructors for creation of a multiset object.

```
explicit multiset  
(const Compare& comp = Compare(),  
const Allocator& = Allocator());  
  
template <class InputIterator> multiset  
(InputIterator first, last, const Compare& comp = Compare(),  
const Allocator& = Allocator());
```

Multiset Specialized Algorithms

The multiset class provides a specialized swap function.

swap

Swaps the first argument with the second argument.

```
template <class Key, class Compare, class Allocator>  
void swap  
(multiset<Key, Compare, Allocator>& x,  
multiset<Key, Compare, Allocator>& y);
```

Template Class Bitset

The `bitset` header defines a template class and related procedures for representing and manipulating fixed-size sequences of bits.

Template Class Bitset

The template class `bitset` can store a sequence consisting of a fixed number of bits.

In the `bitset` class each bit represents either the value zero (`reset`) or one (`set`), there is no negative position. You can `toggle` a bit to change the value.

When converting between an object of class `bitset` and an integral value, the integral value corresponding to two or more bits is the sum of their bit values.

The `bitset` functions can report three kinds of errors as exceptions.

- An `invalid_argument` exception
- An `out_of_range` error exception
- An `overflow_error` exception

See “[Exception Classes](#).” for more information on exception classes.

Constructors

The bitset class includes overloaded constructors for creation of a bitset object.

```
bitset();  
  
bitset(unsigned long val);  
  
template <class charT, class traits, class Allocator>  
explicit bitset  
(const basic_string<charT, traits, Allocator>& str,  
typename basic_string  
<charT, traits, Allocator>::size_type pos = 0,  
typename basic_string<charT, traits,  
Allocator>::size_type n = basic_string  
<charT, traits, Allocator>::npos);
```

Bitset Members

The bitset class provides various member operators.

operator &=

A bitwise “and equal” operator.

```
bitset<N>& operator&=(const bitset<N>& rhs);
```

Remarks

Returns the result of the “and equals” operation.

operator !=

A bitwise “not equal” operator.

```
bitset<N>& operator|=(const bitset<N>& rhs);
```

Remarks

Returns the result of the “not equals” operation.

operator ^=

A bitwise “exclusive or equals” operator.

```
bitset<N>& operator^=(const bitset<N>& rhs);
```

Remarks

Returns the result of the “exclusive or equals” operation.

operator <=>

A bitwise “left shift equals” operator.

```
bitset<N>& operator <<=(size_t pos);
```

Remarks

Returns the result of the “left shift equals” operation.

operator >=>

A bitwise “right shift equals” operator.

```
bitset<N>& operator>>=(size_t pos);
```

Remarks

Returns the result of the “right shifts equals” operation.

Set

Sets all the bits or a single bit to a value.

```
bitset<N>& set();  
bitset<N>& set(size_t pos, int val = 1);
```

Remarks

For the function with no parameters sets all the bits to true. For the overloaded function with just a position argument sets that bit to true. For the function with both a position and a value sets the bit at that position to the value.

Returns the altered bitset.

reset

Sets the bits to false.

```
bitset<N>& reset();  
bitset<N>& reset(size_t pos);
```

Remarks

The reset function without any arguments sets all the bits to false. The reset function with an argument sets the bit at that position to false.

Returns the modified bitset.

operator ~

Toggles all bits in the bitset.

```
bitset<N> operator~() const;
```

Remarks

Returns the modified bitset.

flip

Toggles all the bits in the bitset.

```
bitset<N>& flip();  
bitset<N>& flip(size_t pos);
```

Remarks

Returns the modified bitset.

to_ulong

Gives the value as an unsigned long.

```
unsigned long to_ulong() const;
```

Remarks

Returns the unsigned long value that the bitset represents.

to_string

Gives the string as zero and ones that the bitset represents.

```
template <class charT, class traits, class Allocator>  
basic_string<charT, traits, Allocator> to_string() const;
```

Remarks

Returns a string that the bitset represents.

count

Tells the number of bits that are true.

```
size_t count() const;
```

Remarks

Returns the number of set bits.

size

Tells the size of the bitset as the number of bits.

```
size_t size() const;
```

Remarks

Returns the size of the bitset.

operator ==

The equality operator.

```
bool operator==(const bitset<N>& rhs) const;
```

Remarks

Returns true if the argument is equal to the right side bitset.

operator !=

The inequality operator.

```
bool operator!=(const bitset<N>& rhs) const;
```

Remarks

Returns true if the argument is not equal to the right side bitset.

test

Test if a bit at a position is set.

```
bool test(size_t pos) const;
```

Remarks

Returns true if the bit at the position is true.

any

Tests if all bits are set to true.

```
bool any() const;
```

Remarks

Returns true if any bits in the bitset are true.

none

Tests if all bits are set to false.

```
bool none() const;
```

Remarks

Returns true if all bits are false.

operator <<

Shifts the bitset to the left a number of positions.

```
bitset<N> operator<<(size_t pos) const;
```

Remarks

Returns the modified bitset.

operator >>

Shifts the bitset to the right a number of positions.

```
bitset<N> operator>>(size_t pos) const;
```

Remarks

Returns the modified bitset.

Bitset Operators

Bitwise operators are included in the bitset class.

operator &

A bitwise `and` operator.

```
bitset<N> operator&(const bitset<N>& lhs, const bitset<N>& rhs);
```

Remarks

Returns the modified bitset.

operator |

A bitwise `or` operator.

```
bitset<N> operator|(const bitset<N>& lhs, const bitset<N>& rhs);
```

Remarks

Returns the modified bitset.

operator ^

A bitwise `exclusive or` operator.

```
bitset<N> operator^(const bitset<N>& lhs, const bitset<N>& rhs);
```

Remarks

Returns the modified bitset.

operator >>

An extractor operator for a bitset input.

```
template <class charT, class traits, size_t N>
basic_istream<charT, traits>& operator>>
(basic_istream<charT, traits>& is, bitset<N>& x);
```

Remarks

Returns the bitset.

operator <<

An inserter operator for a bitset output.

```
template <class charT, class traits, size_t N>
basic_ostream<charT, traits>& operator<<
(basic_ostream<charT, traits>& os, const bitset<N>& x);
```

Remarks

Returns the bitset.

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.

The Iterators library

This chapter describes the components used in C++ programs to perform iterations for container classes, streams and stream buffers.

The chapter is constructed in the following sub sections and mirrors clause 24 of the ISO (the International Organization for Standardization) C++ Standard ::

- “[Requirements](#)”
 - “[Input Iterators](#)”
 - “[Output Iterators](#)”
 - “[Forward Iterators](#)”
 - “[Bidirectional Iterators](#)”
 - “[Random Access Iterators](#)”
- “[Header <iterator>](#)”
- “[Iterator Primitives](#)”
 - “[Iterator Traits](#)”
 - “[Basic Iterator](#)”
 - “[Standard Iterator Tags](#)”
- “[Predefined Iterators](#)”
 - “[Reverse iterators](#)”
 - “[Insert Iterators](#)”
- “[Stream Iterators](#)”
 - “[Template Class Istream_iterator](#)”

- [“Template Class Ostream iterator”](#)
- [“Template Class Istreambuf iterator”](#)
- [“Template Class Ostreambuf iterator”](#)

Requirements

Iterators are a generalized pointer that allow the C++ program to work with various containers in a unified manner.

All iterators allow the dereference into a value type.

Since iterators are an abstraction of a pointer all functions that work with regular pointers work equally with regular pointers.

Input Iterators

There are requirements for input iterators, this manual, does not attempt to list them all.

Algorithms on input iterators should never attempt to pass through the same iterator more than once.

Output Iterators

There are requirements for output iterators, this manual, does not attempt to list them all.

An output iterator is assignable.

Forward Iterators

Forward iterators meet all the requirements of input and output iterators.

There are requirements for forward iterators, this manual, does not attempt to list them all.

Bidirectional Iterators

Bidirectional iterators meet the requirements of forward iterators.

There are requirements for forward iterators, this manual, does not attempt to list them all.

Random Access Iterators

Random access iterators meet the requirements of bidirectional iterators.

There are requirements for forward iterators, this manual, does not attempt to list them all.

Header <iterator>

The header iterator includes classes, types and functions used to allow the C++ program to work with various containers in a unified manner.

Iterator Primitives

The library provides several classes and functions to simplify the task of defining iterators:.

Iterator Traits

To implement algorithms only in terms of iterators, it is often necessary to determine the value and difference types for a particular iterator type. Therefore, it is required that if `iterator` is the type of an iterator, then the types

```
iterator_traits<Iterator>::difference_type  
iterator_traits<Iterator>::value_type  
iterator_traits<Iterator>::iterator_category
```

are defined as the iterator's difference type, value type and iterator category, respectively.

In the case of an output iterator, the types

```
iterator_traits<Iterator>::difference_type  
iterator_traits<Iterator>::value_type
```

defined as void.

The template `iterator_traits<Iterator>` is specialized for pointers and for pointers to const

Basic Iterator

The iterator template may be used as a base class for new iterators.

Standard Iterator Tags

The standard library includes category tag classes which are used as compile time tags for algorithm selection. These tags are used to determine the best iterator argument at compile time. These tags are:

- `input_iterator_tag`
- `output_iterator_tag`
- `forward_iterator_tag`,
- `bidirectional_iterator_tag`
- `random_access_iterator_tag`

Iterator Operations

Since only random access iterators provide plus and minus operators, the library provides two template functions for this functionality.

advance

Increments or decrements iterators.

```
template <class InputIterator, class Distance>
void advance(InputIterator& i, Distance n);
```

distance

Provides a means to determine the number of increments or decrements necessary to get from the beginning to the end.

```
template<class InputIterator>
typename iterator_traits<InputIterator>::difference_type distance
(InputIterator first, InputIterator last);
```

Remarks

The distance from last must be reachable from first.

The the number of increments from first to last.

Predefined Iterators

The standard provides for two basic predefined iterators.

- [“Reverse iterators”](#)
- [“Insert Iterators”](#)

Reverse iterators

Both bidirectional and random access iterators have corresponding reverse iterator adaptors that they iterate through.

Template Class Reverse_iterator

A reverse_iterator must meet the requirements of a bidirectional iterator.

Reverse_iterator Requirements

Additional requirements may be necessary if random access operators are referenced in a way that requires instantiation.

Constructors

Creates an instance of a reverse_iterator object.

```
explicit reverse_iterator(Iterator x);  
  
template <class U> reverse_iterator  
(const reverse_iterator<U> &u);
```

base

The base operator is used for conversion.

```
Iterator base() const; // explicit
```

Remarks

The current iterator is returned.

Reverse_iterator operators

The common operators are provided for reverse_iterators.

Operator *

```
reference operator*() const;
```

A reference iterator is returned.

A pointer to the dereferenced iterator is returned.

Operator ->

```
pointer operator ->() const;
```

Operator ++

```
reverse_iterator& operator++();  
reverse_iterator operator++(int);
```

The this pointer is returned.

Operator --

```
reverse_iterator& operator--();  
reverse_iterator operator--(int);
```

The this pointer is returned.

Operator +

```
reverse_iterator operator+  
(typename reverse_iterator<Iterator>::difference_type n) const;
```

The reverse_iterator representing the result of the operation is returned.

Operator +=

```
reverse_iterator& operator+=  
(typename reverse_iterator<Iterator>::difference_type n);
```

The reverse_iterator representing the result of the operation is returned.

Operator -

```
iterator operator-  
(typename reverse_iterator<Iterator>::difference_type n) const;
```

The reverse_iterator representing the result of the operation is returned.

Operator -=

```
reverse_iterator& operator-=  
(typename reverse_iterator<Iterator>  
::difference_type n);
```

The reverse_iterator representing the result of the operation is returned.

Operator []

```
reference operator[]  
(typename reverse_iterator<Iterator>::difference_type n) const;
```

An element access reference is returned.

Operator ==

```
template <class Iterator> bool operator==  
(const reverse_iterator<Iterator>& x,  
const reverse_iterator<Iterator>& y);
```

A bool true value is returned if the iterators are equal.

Operator <

```
template <class Iterator> bool operator<  
(const reverse_iterator<Iterator>& x,  
const reverse_iterator<Iterator>& y);
```

A bool true value is returned if the first iterator is less than the second.

Operator !=

```
template <class Iterator> bool operator!=  
(const reverse_iterator<Iterator>& x,  
const reverse_iterator<Iterator>& y);
```

A bool true value is returned if the first iterator is not equal to the second.

Operator >

```
template <class Iterator> bool operator>  
(const reverse_iterator<Iterator>& x,  
const reverse_iterator<Iterator>& y);
```

A bool true value is returned if the first iterator is greater than the second.

Operator >=

```
template <class Iterator> bool operator>=
(const reverse_iterator<Iterator>& x,
const reverse_iterator<Iterator>& y);
```

The reverse_iterator representing the result of the operation is returned.

Operator <=

```
template <class Iterator> bool operator<=
(const reverse_iterator<Iterator>& x,
const reverse_iterator<Iterator>& y);
```

The reverse_iterator representing the result of the operation is returned.

Operator -

```
template <class Iterator>
typename reverse_iterator<Iterator>
::difference_type operator-
(const reverse_iterator<Iterator>& x,
const reverse_iterator<Iterator>& y);
```

The reverse_iterator representing the result of the operation is returned.

Operator +

```
template <class Iterator>
reverse_iterator<Iterator> operator+
(typename reverse_iterator<Iterator>
::difference_type n,
const reverse_iterator<Iterator>& x);
```

The reverse_iterator representing the result of the operation is returned.

Insert Iterators

Insert iterators, are provided to make it possible to deal with insertion in the same way as writing into an array.

Class Back_insert_iterator

A back_insert_iterator inserts at the back.

Constructors

Constructs a back_insert_iterator object.

```
explicit back_insert_iterator(Container& x);
```

Operator =

An operator is provided for copying a const_reference value.

```
back_insert_iterator<Container>& operator=  
(typename Container::const_reference value);
```

Remarks

A reference to the copied back_insert_iterator is returned.

Back_insert_iterator Operators

Several standard operators are provided for Back_insert_iterator.

Operator *

```
back_insert_iterator<Container>& operator*();
```

The dereference iterator is returned.

Operator ++

```
back_insert_iterator<Container>& operator++();  
back_insert_iterator<Container> operator++(int);  
The incremented iterator is returned.
```

back_inserter

Provides a means to get the back iterator.

```
template <class Container> back_insert_iterator<Container>  
back_inserter  
(Container& x);
```

Remarks

The `back_insert_iterator` is returned.

Template Class Front_insert_iterator

A `front_insert_iterator` inserts at the front.

Constructors

Creates a `front_insert_iterator` object.

```
explicit front_insert_iterator(Container& x);
```

Operator =

Assigns a value to an already created assignment operator.

```
front_insert_iterator<Container>& operator=  
(typename Container::const_reference value);
```

Remarks

A front_insert_iterator copy of the const_reference value is returned.

Front_insert_iterator operators

Several common operators are provided for the `front_insert_iterator` class.

Operator *

```
front_insert_iterator<Container>& operator*();
```

A this pointer is returned.

Operator ++

```
front_insert_iterator<Container>& operator++();
```

```
front_insert_iterator<Container> operator++(int);
```

Remarks

A post or pre increment operator.

The this pointer is returned.

front_inserter

Provides a means to get the front iterator.

```
template <class Container>
front_insert_iterator<Container>
front_inserter(Container& x);
```

Remarks

The front_insert_iterator is returned.

Template Class `Insert_iterator`

A bidirectional insertion iterator.

Constructors

Creates an instance of an `insert_iterator` object.

```
insert_iterator  
(Container& x, typename Container::iterator i);
```

operator =

An operator for assignment of a `const_reference` value.

```
insert_iterator<Container>& operator=  
(typename Container::const_reference value);
```

Remarks

Returns a copy of the `insert_iterator`.

Insert_iterator Operators

Various operators are provided for an `insert_iterator`.

Operator *

```
insert_iterator<Container>& operator*();
```

The dereferenced iterator is returned.

Operator ++

```
insert_iterator<Container>& operator++();
```

```
insert_iterator<Container>& operator++(int);
```

The `this` pointer is returned.

inserter

Provides a means to get the iterator.

```
template <class Container, class Inserter>
insert_iterator<Container> inserter
(Container& x, Inserter i);
```

Remarks

The inserter iterator is returned.

Stream Iterators

Input and output iterators are provided to make it possible for algorithmic templates to work directly with input and output streams.

Template Class `Istream_iterator`

An `istream_iterator` reads (using `operator>>`) successive elements from the input stream. It reads after it is constructed, and every time the increment operator is used.

If an end of stream is reached the iterator returns false.

Since `istream` iterators are not assignable `istream` iterators can only be used for one pass algorithms.

Constructors

Creates and object of an `istream_iterator` object.

```
istream_iterator();
istream_iterator(istream_type& s);
istream_iterator
(const istream_iterator<T, charT, traits, Distance>& x);
```

The parameterless iterator is the only legal constructor for an `end` condition.

destructor

Removes an instance of an `istream_iterator`.

```
~istream_iterator();
```

Istream_iterator Operations

Various operators are provided for an `istream_iterator`.

Operator *

```
const T& operator*() const;
```

A dereferenced iterator is returned.

Operator ->

```
const T* operator->() const;
```

The address of a dereferenced iterator is returned.

Operator ++

```
istream_iterator <T,charT,traits,Distance>& operator++();
```

```
istream_iterator <T,charT,traits,Distance>& operator++(int);
```

The this pointer is returned.

Operator ==

```
template <class T, class charT,  
class traits, class Distance> bool operator==  
(const istream_iterator<T,charT, traits,  
Distance> & x, const istream_iterator  
<T,charT,traits,Distance> & y);
```

A bool true value is retuned if the arguments ate the same.

Template Class Ostream_iterator

The ostream_iterator writes (using `operator<<`) successive elements onto the output stream.

Constructors

Creates and instance of an `ostream_iterator` object.

```
ostream_iterator(ostream_type& s);  
ostream_iterator(ostream_type& s, const charT* delimiter);  
ostream_iterator(const ostream_iterator& x);
```

Operator =

```
ostream_iterator& operator=(const T& value);
```

Returns a value to an ostream iterator.

destructor

Removes and instance of an `ostream_iterator` object.

```
~ostream_iterator();
```

Ostream_iterator Operators

Various operators are provided in the ostream_iterator class.

Operator *

```
ostream_iterator& operator*();
```

The dereference iterator is returned.

Operator ++

```
ostream_iterator& operator++();
```

```
ostream_iterator& operator++(int);
```

The this pointer is returned.

Template Class istreambuf_iterator

The istreambuf_iterator reads successive characters from the istreambuf object for which it was constructed.

An ostream_iterator can only be used for a one pass algorithm.

Constructors

An overloaded constructor is provided for creation of an istreambuf_iterator object.

```
istreambuf_iterator() throw();  
istreambuf_iterator(basic_istream<charT,traits>& s) throw();  
istreambuf_iterator(basic_streambuf<charT,traits>* s) throw();  
istreambuf_iterator(const proxy& p) throw();
```

Istreambuf_iterator Operators

Various operators are provided for the istreambuf_iterator class.

Operator *

```
charT operator*() const
```

A dereferenced character type is returned.

Operator ++

```
istreambuf_iterator<charT, traits>&
istreambuf_iterator<charT, traits>::operator++();
```

The this pointer is returned.

Operator ==

```
template <class charT, class traits>
bool operator==
(const istreambuf_iterator<charT, traits>& a,
const istreambuf_iterator<charT, traits>& b);
```

True is returned if the arguments are equal.

Operator !=

```
template <class charT, class traits>
bool operator!=
(const istreambuf_iterator<charT, traits>& a,
const istreambuf_iterator<charT, traits>& b);
```

True is returned if the arguments are not equal.

equal

An equality comparison.

```
bool equal(istreambuf_iterator<charT,traits>& b);
```

Remarks

True is returned if the arguments are equal.

Template Class Ostreambuf_iterator

The `ostreambuf_iterator` writes successive characters to the `ostreambuf` object for which it was constructed.

Constructors

The constructor is overloaded for creation of an `ostreambuf_iterator` object.

```
ostreambuf_iterator(ostream_type& s) throw();  
ostreambuf_iterator(streambuf_type* s) throw();
```

Operator =

```
ostreambuf_iterator<charT,traits>& operator=(charT c);
```

The result of the assignment is returned.

Ostreambuf_iterator Operators

Operator *

```
ostreambuf_iterator<charT,traits>& operator*();
```

The dereferenced `ostreambuf_iterator` is returned.

Operator ++

```
ostreambuf_iterator<charT,traits>& operator++();  
ostreambuf_iterator<charT,traits>& operator++(int);
```

The this pointer is returned.

failed

Reports a failure in writing.

```
bool failed() const throw();
```

Remarks

The bool false value is returned if a write failure occurs.

Algorithms Library

This chapter discusses the algorithms library. These algorithms cover sequences, sorting, and numerics.

The Algorithms Library

The standard provides for various algorithms that a C++ program may use to perform algorithmic operations on containers and other sequences.

The chapter is constructed in the following sub sections and mirrors clause 25 of the ISO (the International Organization for Standardization) C++ Standard :

- [“Non-modifying Sequence Operations”](#)
- [“Mutating Sequence Operators”](#)
- [“Sorting And Related Operations”](#)
- [“C library algorithms”](#)

Header <algorithm>

The header algorithm provides classes, types and functions for use with the standard C++ libraries.

The standard algorithms can work with program defined data structures, as long as these data structures have iterator types satisfying the assumptions on the algorithms.

The names of the parameters used in this chapter reflect their usage.

A predicate parameter is used for a function object that returns a value testable as true. The binary predicate parameter takes two arguments.

Non-modifying Sequence Operations

Various algorithms are provided which do not modify the original object.

for_each

The function `for_each` is used to perform an operation for each element.

```
template<class InputIterator, class Function>
Function for_each
(InputIterator first, InputIterator last, Function f);
```

Remarks

The function `f` is returned.

find

The function `find` searches for the first element that contains the value passed.

```
template<class InputIterator, class T>
InputIterator find
(InputIterator first, InputIterator last, const T& value);
```

Remarks

Returns the type passed.

find_if

The function `find_if` searches for the first element that matches the criteria passed by the predicate.

```
template<class InputIterator, class Predicate>
InputIterator find_if
(InputIterator first, InputIterator last, Predicate pred);
```

Remarks

Returns the iterator of the matched value.

find_end

The function `find_end` searches for the last occurrence of a value.

```
template<class ForwardIterator1,
         class ForwardIterator2>
ForwardIterator1 find_end
(ForwardIterator1 first1, ForwardIterator1 last1,
 ForwardIterator2 first2, ForwardIterator2 last2);
```

```
template<class ForwardIterator1,
         class ForwardIterator2, class BinaryPredicate>
ForwardIterator1 find_end
(ForwardIterator1 first1, ForwardIterator1 last1,
 ForwardIterator2 first2, ForwardIterator2 last2,
 BinaryPredicate pred);
```

Remarks

Returns the iterator to the last value or the `last1` argument if none is found.

find_first_of

The function `find_first_of` searches for the first occurrence of a value.

```
template<class ForwardIterator1,
         class ForwardIterator2>
ForwardIterator1 find_first_of
(ForwardIterator1 first1, ForwardIterator1 last1,
 ForwardIterator2 first2, ForwardIterator2 last2);

template<class ForwardIterator1,
         class ForwardIterator2, class BinaryPredicate>
ForwardIterator1 find_first_of
(ForwardIterator1 first1, ForwardIterator1 last1,
 ForwardIterator2 first2,
 ForwardIterator2 last2, BinaryPredicate pred);
```

Remarks

Returns the iterator to the last value or the `last1` argument if none is found.

adjacent_find

The function `adjacent_find` is used to search for two adjacent elements that are equal or equal according to the predicate argument.

```
template<class ForwardIterator>

ForwardIterator adjacent_find

(ForwardIterator first, ForwardIterator last);
```

```
template<class ForwardIterator, class BinaryPredicate>

ForwardIterator adjacent_find

(ForwardIterator first, ForwardIterator last,

BinaryPredicate pred);
```

Remarks

Returns the iterator to the first occurrence found or to `last` if no occurrence is found.

count

The function `count` is used to find the number of elements.

```
template <class InputIterator, class T>

typename iterator_traits

<InputIterator>::difference_type count

(InputIterator first, InputIterator last, const T& value);
```

Remarks

Returns the number of elements (iterators) as an `iterator_traits` `difference_type`.

count_if

The function `count_if` is used to find the number of elements that match the criteria.

```
template <class InputIterator, class Predicate>
typename iterator_traits
<InputIterator>::difference_type count_if
(InputIterator first, InputIterator last, Predicate pred);
```

Remarks

Returns the number of elements (iterators) as an `iterator_traits` `difference_type`.

mismatch

The function `mismatch` is used to find sequences that are not the same or differ according to the predicate criteria.

```
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 pred);
```

Remarks

Returns a pair<iterator> that represent the beginning element and the range. If no mismatch is found the end and the corresponding range element is returned.

equal

The function equal is used to determine if a range two ranges are equal.

```
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 pred);
```

Remarks

A bool true is returned if the values are equal or meet the criteria of the predicate.

search

The function `search` is used to search for the first octanes of a sub-range that meet the criteria.

```
template<class ForwardIterator1, class ForwardIterator2>
ForwardIterator1 search
(ForwardIterator1 first1, ForwardIterator1 last1,
ForwardIterator2 first2, ForwardIterator2 last2);

template<class ForwardIterator1,
class ForwardIterator2, class BinaryPredicate>
ForwardIterator1 search
(ForwardIterator1 first1, ForwardIterator1 last1,
ForwardIterator2 first2, ForwardIterator2 last2,
BinaryPredicate pred);
```

Remarks

An iterator to the first occurrence is returned or `last1` is returned if no criteria is met.

search_n

The function `search_n` is used to search for a number of consecutive elements with the same properties.

```
template<class ForwardIterator, class Size, class T>
ForwardIterator search_n
(ForwardIterator first, ForwardIterator last,
Size count, const T& value);

template<class ForwardIterator,
class Size, class T, class BinaryPredicate>
ForwardIterator search_n
(ForwardIterator first, ForwardIterator last, Size count,
const T& value, BinaryPredicate pred);
```

Remarks

An iterator to the first occurrence is returned or `last1` is returned if no criteria is met.

Mutating Sequence Operators

Various algorithms are provided that are used to modify the original object.

copy

The function `copy` is used to copy a range.

```
template<class InputIterator, class OutputIterator>
OutputIterator copy(InputIterator first,
InputIterator last, OutputIterator result);
```

Remarks

The position of the last copied element is returned.

copy_backward

The function `copy_backwards` is used to copy a range starting with the last element.

```
template<class BidirectionalIterator1,
class BidirectionalIterator2>
BidirectionalIterator2 copy_backward
(BidirectionalIterator1 first, BidirectionalIterator1 last,
BidirectionalIterator2 result);
```

Remarks

The position of the last copied element is returned.

swap

The function `swap` is used to exchange values from two locations.

```
template<class T> void swap(T& a, T& b);
```

Remarks

There is no return.

swap_ranges

The function `swap_ranges` is used swap elements of two ranges.

```
template<class ForwardIterator1, class ForwardIterator2>
ForwardIterator2 swap_ranges
(ForwardIterator1 first1, ForwardIterator1 last1,
ForwardIterator2 first2);
```

Remarks

The position of the last swapped element is returned.

iter_swap

The function `iter_swap` is used to exchange two values pointed to by iterators.

```
template<class ForwardIterator1, class ForwardIterator2>
void iter_swap(ForwardIterator1 a, ForwardIterator2 b);
```

Remarks

There is no return.

transform

The function `transform` is used to modify and copy elements of two ranges.

```
template<class InputIterator,
         class OutputIterator, class UnaryOperation>
OutputIterator transform
(InputIterator first, InputIterator last,
 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

The position of the last transformed element is returned.

replace

The function `replace` is used to replace an element with another element of different value.

```
template<class ForwardIterator, class T>
void replace
(ForwardIterator first, ForwardIterator last,
const T& old_value, const T& new_value);

template<class ForwardIterator, class Predicate, class T>
void replace_if
(ForwardIterator first, ForwardIterator last,
Predicate pred, const T& new_value);
```

Remarks

There is no return.

replace_copy

The function `replace_copy` is used to replace specific elements while copying an entire range.

```
template<class InputIterator, class OutputIterator, class T>
OutputIterator replace_copy
(InputIterator first, InputIterator last,
OutputIterator result,
const T& old_value, const T& new_value);
```

Remarks

The position of the last copied element is returned.

replace_copy_if

The function `replace_copy_if` is used to replace specific elements that match certain criteria while copying the entire range.

```
template<class Iterator,
         class OutputIterator, class Predicate, class T>
OutputIterator replace_copy_if
(Iterator first, Iterator last,
OutputIterator result, Predicate pred, const T& new_value);
```

Remarks

The position of the last copied element is returned.

fill

The function `fill` is used to fill a range with values.

```
template<class ForwardIterator, class T>
void fill
(ForwardIterator first, ForwardIterator last, const T& value);
```

Remarks

There is no return value.

fill_n

The function `fill_n` is used to fill a number of elements with a specified value.

```
template<class OutputIterator,  
        class Size, class T>  
void fill_n  
(OutputIterator first, Size n, const T& value);
```

Remarks

There is no return value.

generate

The function `generate` is used to replace elements with the result of an operation.

```
template<class ForwardIterator, class Generator>  
void generate  
(ForwardIterator first, ForwardIterator last, Generator gen);
```

Remarks

There is no return value.

generate_n

The function `generate_n` is used to replace a number of elements with the result of an operation.

```
template<class OutputIterator, class Size, class Generator>
void generate_n
(OutputIterator first, Size n, Generator gen);
```

Remarks

There is no return value.

remove

The function `remove` is used to remove elements with a specified value.

```
template<class ForwardIterator, class T>
ForwardIterator remove
(ForwardIterator first, ForwardIterator last,const T& value);
```

Remarks

The end of the resulting range is returned.

remove_if

The function `remove_if` is used to remove elements using a specified criteria.

```
template<class ForwardIterator, class Predicate>
ForwardIterator remove_if
(ForwardIterator first, ForwardIterator last,Predicate pred);
```

Remarks

The end of the resulting range is returned.

remove_copy

The function `remove_copy` is used remove elements that do not match a value during a copy.

```
template<class InputIterator, class OutputIterator, class T>
OutputIterator remove_copy
(InputIterator first, InputIterator last,
OutputIterator result, const T& value);
```

Remarks

The end of the resulting range is returned.

remove_copy_if

The function `remove_copy_if` is used to remove elements that do not match a criteria while doing a copy.

```
template<class InputIterator,
class OutputIterator, class Predicate>
OutputIterator remove_copy_if
(InputIterator first, InputIterator last,
OutputIterator result, Predicate pred);
```

Remarks

The end of the resulting range is returned.

unique

The function `unique` is used remove all adjacent duplicates.

```
template<class ForwardIterator>

ForwardIterator unique

(ForwardIterator first, ForwardIterator last);

template<class ForwardIterator, class BinaryPredicate>

ForwardIterator unique (ForwardIterator first,
ForwardIterator last, BinaryPredicate pred);
```

Remarks

The end of the resulting range is returned.

unique_copy

The function `unique_copy` is used to remove adjacent duplicates while copying.

```
template<class InputIterator, class OutputIterator>

OutputIterator unique_copy

(InputIterator first, InputIterator last,
OutputIterator result);

template<class InputIterator,
class OutputIterator, class BinaryPredicate>

OutputIterator unique_copy

(InputIterator first, InputIterator last,
OutputIterator result, BinaryPredicate pred);
```

Remarks

The end of the resulting range is returned.

reverse

The function `reverse` is used to reverse a sequence.

```
template<class BidirectionalIterator>
void reverse
(BidirectionalIterator first,BidirectionalIterator last);
```

Remarks

No value is returned.

reverse_copy

The function `reverse_copy` is used to copy the elements while reversing their order.

```
template<class BidirectionalIterator, class OutputIterator>
OutputIterator reverse_copy
(BidirectionalIterator first,BidirectionalIterator last,
OutputIterator result);
```

Remarks

The position of the last copied element is returned.

rotate

The function `rotate` is used to rotate the elements within a sequence.

```
template<class ForwardIterator>
void rotate
(ForwardIterator first, ForwardIterator middle,
ForwardIterator last);
```

Remarks

There is no return value.

rotate_copy

The function `rotate_copy` is used to copy a sequence with a rotated order.

```
template<class ForwardIterator, class OutputIterator>
OutputIterator rotate_copy
(ForwardIterator first, ForwardIterator middle,
ForwardIterator last, OutputIterator result);
```

Remarks

The position of the last copied element is returned.

random_shuffle

The function `random_shuffle` is used to exchange the order of the elements in a random fashion.

```
template<class RandomAccessIterator>

void random_shuffle

(RandomAccessIterator first,RandomAccessIterator last);

template<class RandomAccessIterator,
         class RandomNumberGenerator>

void random_shuffle

(RandomAccessIterator first, RandomAccessIterator last,
RandomNumberGenerator& rand);
```

Remarks

No value is returned.

partition

The function `partition` is used to change the order of the elements so that the elements that meet the criteria are first in order.

```
template<class BidirectionalIterator, class Predicate>

BidirectionalIterator partition

(BidirectionalIterator first,
BidirectionalIterator last, Predicate pred);
```

Remarks

Returns an iterator to the first position where the predicate argument is false.

stable_partition

The function `stable_partition` is used to change the order of the elements so that the elements meet the criteria are first in order. The relative original order is preserved.

```
template<class BidirectionalIterator, class Predicate>
BidirectionalIterator stable_partition
(BidirectionalIterator first,
BidirectionalIterator last, Predicate pred);
```

Remarks

Returns an iterator to the first position where the predicate argument is false.

Sorting And Related Operations

All of the sorting functions have two versions:, one that takes a function object for comparison and one that uses the less than operator.

sort

The function `sort` is used sorts the range according to the criteria.

```
template<class RandomAccessIterator>
void sort
(RandomAccessIterator first,RandomAccessIterator last);

template<class RandomAccessIterator,
class Compare>
void sort(RandomAccessIterator first,
RandomAccessIterator last, Compare comp);
```

Remarks

There is no return value.

stable_sort

The function `stable_sort` is used to sort the range but preserves the original order for equal elements.

```
template<class RandomAccessIterator>
void stable_sort
(RandomAccessIterator first, RandomAccessIterator last);

template<class RandomAccessIterator, class Compare>
void stable_sort
(RandomAccessIterator first,
RandomAccessIterator last, Compare comp);
```

Remarks

There is no return value.

partial_sort

The function `partial_sort` is used to sort a sub-range leaving the rest unsorted.

```
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

There is no return value.

partial_sort_copy

The function `partial_sort_copy` is used to copy a partially sorted sequence.

```
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

The position at the end of the copied elements is returned.

nth_element

The function `nth_element` is used to sort based upon a specified position.

```
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

There is no value returned.

lower_bound

The function `lower_bound` is used to find the first position that an element may be inserted without changing the order.

```
template<class ForwardIterator, class T>
ForwardIterator lower_bound
(ForwardIterator first, ForwardIterator last, const T& value);

template<class ForwardIterator, class T, class Compare>
ForwardIterator lower_bound
(ForwardIterator first, ForwardIterator last,
const T& value, Compare comp);
```

Remarks

The position where the element can be inserted is returned.

upper_bound

The function `upper_bound` is used to find the last position that an element may be inserted without changing the order.

```
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

The position where the element can be inserted is returned.

equal_range

The function `equal_range` is used to find the range as a pair where an element can be inserted without altering the order.

```
template<class ForwardIterator, class T>
pair<ForwardIterator, ForwardIterator> equal_range
(ForwardIterator first, ForwardIterator last, const T& value);

template<class ForwardIterator, class T, class Compare>
pair<ForwardIterator, ForwardIterator> equal_range
(ForwardIterator first, ForwardIterator last,
const T& value, Compare comp);
```

Remarks

The range as a pair<> where the element can be inserted is returned.

binary_search

The function `binary_search` is used to see if a value is present in a range or that a value meets a criteria within that range.

```
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

The bool value true is met if any element meets the criteria.

merge

The function `merge` is used to combine two sorted ranges.

```
template<class InputIterator1,
         class InputIterator2, class OutputIterator>
OutputIterator merge
(InputIterator1 first1, InputIterator1 last1,
 InputIterator2 first2, InputIterator2 last2,
 OutputIterator result);

template<class InputIterator1, class InputIterator2,
         class OutputIterator, class Compare>
OutputIterator merge
(InputIterator1 first1, InputIterator1 last1,
 InputIterator2 first2, InputIterator2 last2,
 OutputIterator result, Compare comp);
```

Return

The position of the first element not overwritten is returned.

inplace_merge

The function `replacement` is used to merge consecutive sequences to the first for a concatenation.

```
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);
```

Remarks

There is no value returned.

includes

The function `includes` is used to determine if every element meets a specified criteria.

```
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);
```

Remarks

The bool value true is retuned if all values match or false if one or more does not meet the criteria.

set_union

The function `set_union` is used to process the sorted union of two ranges.

```
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);
```

Remarks

The end of the constructed range is returned.

set_intersection

The function `set_intersection` is used to process the intersection of two ranges.

```
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 end of the constructed range is returned.

set_difference

The function `set_difference` is used to process all of the elements of one range that are not part of another range.

```
template<class InputIterator1,
         class InputIterator2, class OutputIterator>
OutputIterator set_difference
(InputIterator1 first1, 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 end of the constructed range is returned.

set_symmetric_difference

The function `set_symmetric_difference` is used to process all of the elements that are in only one of two ranges.

```
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);
```

Remarks

The end of the constructed range is returned.

push_heap

The function `push_heap` is used to add an element to a heap.

```
template<class RandomAccessIterator>

void push_heap

(RandomAccessIterator first, RandomAccessIterator last);

template<class RandomAccessIterator, class Compare>

void push_heap

(RandomAccessIterator first,
RandomAccessIterator last, Compare comp);
```

Remarks

There is no value returned.

pop_heap

The function `pop_heap` is used to remove an element from a heap.

```
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

There is no value returned.

make_heap

The function `make_heap` is used to convert a range into a heap.

```
template<class RandomAccessIterator>
void make_heap(
    RandomAccessIterator first, RandomAccessIterator last);
```

```
template<class RandomAccessIterator, class Compare>
void make_heap(
    RandomAccessIterator first, RandomAccessIterator last,
    Compare comp);
```

Remarks

There is no value returned.

sort_heap

The function `sort_heap` is used to sort a heap.

```
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

Note that this result is not stable

There is no value returned.

min

The function `min` is used to determine the lesser of two objects by value or based upon a comparison.

```
template<class T>
const T& min (const T& a, const T& b);

template<class T, class Compare>
const T& min(const T& a, const T& b, Compare comp);
```

Remarks

The lesser of the two objects is returned.

max

The function `max` is used to determine the greater of two objects by value or based upon a comparison.

```
template<class T>
const T& max (const T& a, const T& b);

template<class T, class Compare>
const T& max(const T& a, const T& b, Compare comp);
```

Remarks

The greater of the two objects is returned.

min_element

The function `min_element` is used to determine the lesser element within a range based upon a value or a comparison.

```
template<class ForwardIterator>

ForwardIterator min_element

(ForwardIterator first, ForwardIterator last);

template<class ForwardIterator, class Compare>

ForwardIterator min_element

(ForwardIterator first, ForwardIterator last,

Compare comp);
```

Remarks

The position of the element is returned.

max_element

The function `max_element` is used to determine the greater element within a range based upon a value or a comparison.

```
template<class ForwardIterator>
ForwardIterator max_element
(ForwardIterator first, ForwardIterator last);

template<class ForwardIterator, class Compare>
ForwardIterator max_element
(ForwardIterator first, ForwardIterator last,
Compare comp);
```

Remarks

The position of the element is returned.

lexicographical_compare

The function `lexicographical_compare` is used to determine if a range is lexicographically less than another.

```
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

Returns true if the first argument is less than the second and false for all other conditions.

next_permutation

The function `next_permutation` is used to sort in an ascending order based upon lexicographical criteria.

```
template<class BidirectionalIterator>
bool next_permutation
(BidirectionalIterator first, BidirectionalIterator last);
```

```
template<class BidirectionalIterator, class Compare>
bool next_permutation
(BidirectionalIterator first,
BidirectionalIterator last, Compare comp);
```

Remarks

Returns true if all elements have been sorted.

prev_permutation

The function `prev_permutation` is used to sort in an descending order based upon lexicographical criteria.

```
template<class BidirectionalIterator>
bool prev_permutation
(BidirectionalIterator first, BidirectionalIterator last);
```

```
template<class BidirectionalIterator, class Compare>
bool prev_permutation
(BidirectionalIterator first,
BidirectionalIterator last, Compare comp);
```

Remarks

Returns true if all elements have been sorted.

C library algorithms

The C++ header <cstdlib> provides two variations from the standard C header stdlib.h for searching and sorting.

bsearch

The function signature of `bsearch`

```
bsearch(const void *, const void *, size_t,
size_t,    int (*) (const void *, const void *));
```

is replaced by

```
extern "C" void *bsearch  
(const void * key, const void * base,  
size_t nmemb, size_t size,  
int (* compar)(const void *, const void *));
```

and

```
extern "C++" void *bsearch  
(const void * key, const void * base,  
size_t nmemb, size_t size,  
int (* compar)(const void *, const void *));
```

qsort

The function signature of qsort

```
qsort(void *, size_t, size_t,  
int (*)(const void *, const void *));
```

is replaced by

```
extern "C" void qsort  
void* base, size_t nmemb, size_t size,  
int (* compar)(const void*, const void*));
```

and

```
extern "C++" void qsort  
(void* base, size_t nmemb, size_t size,  
int (* compar)(const void*, const void*));
```


Algorithms Library

Header <algorithm>

Numerics Library

This chapter is a reference guide to the ANSI/ISO standard Numeric classes which are used to perform the semi-numerical operations.

The Numerics Library (clause 26)

The chapter is constructed in the following sub sections and mirrors clause 26 of the ISO (the International Organization for Standardization) C++ Standard :

- “[Numeric type requirements](#)”
- “[The Complex Class Library](#)” is a separate chapter
- “[Numeric arrays](#)”
- “[Generalized Numeric Operations](#)”
- “[C Library](#)”

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 only with a type `TYPE` that satisfies the following requirements:

`TYPE` is not an abstract class (it has no pure virtual member functions);

- `TYPE` is not a reference type;
- `TYPE` is not cv-qualified;
- If `TYPE` is a class, it has a public default constructor;
- If `TYPE` is a class, it has a public copy constructor with the signature
`TYPE::TYPE(const TYPE&)`
- If `TYPE` is a class, it has a public destructor;
- If `TYPE` is a class, it has a public assignment operator whose signature is either

```
TYPE&    TYPE::operator=(const TYPE&)
```

or

```
TYPE&    TYPE::operator=(TYPE)
```

- If `TYPE` is a class, the assignment operator, copy and default constructors, and destructor shall correspond to each other as far as initialization of raw storage using the default constructor, followed by assignment, is the 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.
- If `TYPE` is a class, it shall not overload unary `operator&`.

If an operation on `TYPE` throws an exception then the effects are undefined.

Specific classes member functions or general functions may have other restrictions.

Numeric arrays

The numeric array library consists of several classes and non member operators for the manipulation of array objects.

- [“Template Class Valarray”](#)
- [“Valarray Non-member Operations”](#)
- [“Class slice”](#)
- [“Template Class Slice_array”](#)
- [“Class Gslice”](#)
- [“Template Class Gslice_array”](#)
- [“Template Class Mask_array”](#)
- [“Template Class Indirect_array”](#)

Template Class Valarray

The template class valarray is a single direction smart array with element indexing beginning with the zero element.

Constructors

The class `valarray` provides overloaded constructors to create an object of `valarray` in several manners.

```
valarray();  
  
explicit valarray(size_t);  
  
valarray(const T&, size_t);  
  
valarray(const T*, size_t);  
  
valarray(const valarray<T>&);  
  
valarray(const slice_array<T>&);  
  
valarray(const gslice_array<T>&);  
  
valarray(const mask_array<T>&);  
  
valarray(const indirect_array<T>&);
```

Destructor

Removes a `valarray` object from memory.

```
~valarray();
```

Assignment Operator

The `valarray` class provides for various means of assignment to an already created object.

```
valarray<T>& operator=(const valarray<T>&);

valarray<T>& operator=(const T&);

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

A `valarray` object is returned.

valarray element access

An index operator is provided for single element access of `valarray` objects.

operator[]

This operator provide element access for read and write operations.

```
T operator[](size_t) const;

T& operator[](size_t);
```

Remarks

A type is returned.

valarray subset operations

An index operator is provided for subset array access.

operator[]

The index operator is specialized for subset access to allow both read and write operations.

```
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>&);
```

Remarks

The return corresponds to the index type.

valarray unary operators

The `valarray` class provides operators for array manipulation.

Operator +

```
valarray<T> operator+() const;
```

Returns a valarray sum of $x+y$;

Operator -

```
valarray<T> operator-() const;
```

Returns a valarray result of $x-y$;

Operator ~

```
valarray<T> operator~() const;
```

Returns a valarray result of $x \sim y$;

Operator !

```
valarray<bool> operator!() const;
```

Returns a bool valarray of $\neg x$;

Valarray Computed Assignment

The valarray class provides for a means of compound assignment and math operation. A valarray object is returned.

Operator *=

```
valarray<T>& operator*=(const valarray<T>&);
```

```
valarray<T>& operator*=(const T&);
```

Returns a valarray result of $x * y$;

Operator /=

```
valarray<T>& operator/=(const valarray<T>&);
```

```
valarray<T>& operator/=(const T&);
```

Returns a valarray result of x / y ;

Operator %=

```
valarray<T>& operator%=(const valarray<T>&);
```

```
valarray<T>& operator%=(const T&);
```

Returns a valarray result of $x \% y$;

Operator +=

```
valarray<T>& operator+= (const valarray<T>&);  
valarray<T>& operator+= (const T&);  
Returns a valarray result of x+=y;
```

Operator -=

```
valarray<T>& operator-= (const valarray<T>&);  
valarray<T>& operator-= (const T&);  
Returns a valarray result of x-=y;
```

Operator ^=

```
valarray<T>& operator^= (const valarray<T>&);  
valarray<T>& operator^= (const T&);  
Returns a valarray result of x^=y;
```

Operator &=

```
valarray<T>& operator&= (const valarray<T>&);  
valarray<T>& operator&= (const T&);  
Returns a valarray result of x&=y;
```

Operator |=

```
valarray<T>& operator|= (const valarray<T>&);  
valarray<T>& operator|= (const T&);  
Returns a valarray result of x!=y;
```

Operator <<=

```
valarray<T>& operator<<=(const valarray<T>&);  
valarray<T>& operator<<=(const T&);  
Returns a valarray result of x<<=y;
```

Operator >>-

```
valarray<T>& operator>>=(const valarray<T>&);
```

```
valarray<T>& operator>>=(const T&);
```

Returns a valarray result of `x>>=y;`

Valarray Member Functions

The `valarray` class provides member functions for array information.

size

Tells the size of the array.

```
size_t size() const;
```

Remarks

Returns the size of the array.

sum

Tells the sum of the array elements.

```
T sum() const;
```

Remarks

Returns the sum of the array elements.

min

Tells the smallest element of an array.

```
T min() const;
```

Remarks

Returns the smallest element in an array.

max

T max() const;

Remarks

Returns the largest element in an array.

shift

Returns a new array where the elements have been shifted a set amount.

valarray<T> shift(int n) const;

Remarks

Returns the modified array.

cshift

A cyclical shift of an array.

valarray<T> cshift(int n) const;

Remarks

Returns the modified array.

apply

Processes the elements of an array.

```
valarray<T> apply(T func(T)) const;  
valarray<T> apply(T func(const T&)) const;
```

Remarks

This function “applies” the function specified to all the elements of an array.

Return the modified array.

resize

Resizes an array and initializes the elements

```
void resize(size_t sz, T c = T());
```

Remarks

If no object is provided the array is initialized with the default constructor.

Valarray Non-member Operations

Non-member operators are provided for manipulation of arrays.

Valarray Binary Operators

Non-member valarray operators are provided for the manipulation of arrays.

```
template<class T> valarray<T> operator*
(const valarray<T>&, const valarray<T>&);
```

```
template<class T> valarray<T> operator/
(const valarray<T>&, const valarray<T>&);
```

```
template<class T> valarray<T> operator%
(const valarray<T>&, const valarray<T>&);
```

```
template<class T> valarray<T> operator+
(const valarray<T>&, const valarray<T>&);
```

```
Template<class T> valarray<T> operator-
(const valarray<T>&, const valarray<T>&);
```

```
template<class T> valarray<T> operator^
(const valarray<T>&, const valarray<T>&);
```

```
template<class T> valarray<T> operator&
(const valarray<T>&, const valarray<T>&);
```

```
template<class T> valarray<T> operator|
(const valarray<T>&, const valarray<T>&);
```

```
template<class T> valarray<T> operator<<
(const valarray<T>&, const valarray<T>&);

template<class T> valarray<T> operator>>
(const valarray<T>&, const valarray<T>&);

template<class T> valarray<T> operator*
(const valarray<T>&, const T&);

template<class T> valarray<T> operator*
(const T&, const valarray<T>&);

template<class T> valarray<T> operator/
(const valarray<T>&, const T&);

template<class T> valarray<T> operator/
(const T&, const valarray<T>&);

template<class T> valarray<T> operator%
(const valarray<T>&, const T&);

template<class T> valarray<T> operator%
(const T&, const valarray<T>&);
```

```
template<class T> valarray<T> operator+
(const valarray<T>&, const T&);

template<class T> valarray<T> operator+
(const T&, const valarray<T>&);

template<class T> valarray<T> operator-
(const valarray<T>&, const T&);

template<class T> valarray<T> operator-
(const T&, const valarray<T>&);

template<class T> valarray<T> operator^
(const valarray<T>&, const T&);

template<class T> valarray<T> operator^
(const T&, const valarray<T>&);

template<class T> valarray<T> operator&
(const valarray<T>&, const T&);

template<class T> valarray<T> operator&
(const T&, const valarray<T>&);

template<class T> valarray<T> operator|
```

```
(const valarray<T>&, const T&);  
  
template<class T> valarray<T> operator|  
(const T&, const valarray<T>&);  
  
template<class T> valarray<T> operator<<  
(const valarray<T>&, const T&);  
  
template<class T> valarray<T> operator<<  
(const T&, const valarray<T>&);  
  
template<class T> valarray<T> operator>>  
(const valarray<T>&, const T&);  
  
template<class T> valarray<T> operator>>  
(const T&, const valarray<T>&);
```

Remarks

Each operator returns an array whose length is equal to the lengths of the argument arrays and initialized with the result of applying the operator.

Valarray Logical Operators

The `valarray` class provides logical operators for the comparison of like arrays.

```
template<class T> valarray<bool> operator==  
(const valarray<T>&, const valarray<T>&);  
  
template<class T> valarray<bool> operator!=  
(const valarray<T>&, const valarray<T>&);  
  
template<class T> valarray<bool> operator<  
(const valarray<T>&, const valarray<T>&);  
  
template<class T> valarray<bool> operator>  
(const valarray<T>&, const valarray<T>&);  
  
template<class T> valarray<bool> operator<=  
(const valarray<T>&, const valarray<T>&);  
  
template<class T> valarray<bool> operator>=  
(const valarray<T>&, const valarray<T>&);  
  
template<class T> valarray<bool> operator&&  
(const valarray<T>&, const valarray<T>&);  
  
template<class T> valarray<bool> operator||
```

```
(const valarray<T>&, const valarray<T>&) ;
```

Remarks

All of the logical operators returns a bool array whose length is equal to the length of the array arguments. The elements of the returned array are initialized with a boolean result of the match.

Non-member logical operations

Non-member logical operators are provided to allow for variations of order of the operation.

```
template<class T> valarray<bool> operator==  
(const valarray&, const T&);  
  
template<class T> valarray<bool> operator==  
(const T&, const valarray&);  
  
template<class T> valarray<bool> operator!=  
(const valarray&, const T&);  
  
template<class T> valarray<bool> operator!=  
(const T&, const valarray&);  
  
template<class T> valarray<bool> operator<  
(const valarray&, const T&);  
  
template<class T> valarray<bool> operator<  
(const T&, const valarray&);  
  
template<class T> valarray<bool> operator>  
(const valarray&, const T&);  
  
template<class T> valarray<bool> operator>
```

```
(const T&, const valarray&);  
  
template<class T> valarray<bool> operator<=  
(const valarray&, const T&);  
  
template<class T> valarray<bool> operator<=  
(const T&, const valarray&);  
  
template<class T> valarray<bool> operator>=  
(const valarray&, const T&);  
  
template<class T> valarray<bool> operator>=  
(const T&, const valarray&);  
  
template<class T> valarray<bool> operator&&  
(const valarray<T>&, const T&);  
  
template<class T> valarray<bool> operator&&  
(const T&, const valarray<T>&);  
  
template<class T> valarray<bool> operator||  
(const valarray<T>&, const T&);  
  
template<class T> valarray<bool> operator||  
(const T&, const valarray<T>&);
```

Remarks

The result of these operations is bool array whose length is equal to the length of the array argument. Each element of the returned array is the result of a logical match.

valarray transcendentals

Trigonometric and exponential functions are provided for the `valarray` classes.

```
template<class T> valarray<T> abs  
(const valarray<T>&);
```

```
template<class T> valarray<T> acos  
(const valarray<T>&);
```

```
template<class T> valarray<T> asin  
(const valarray<T>&);
```

```
template<class T> valarray<T> atan  
(const valarray<T>&);
```

```
template<class T> valarray<T> atan2  
(const valarray<T>&, const valarray<T>&);
```

```
template<class T> valarray<T> atan2  
(const valarray<T>&, const T&);  
template<class T> valarray<T> atan2  
(const T&, const valarray<T>&);
```

```
template<class T> valarray<T> cos  
(const valarray<T>&);
```

```
template<class T> valarray<T> cosh  
(const valarray<T>&);
```

```
template<class T> valarray<T> exp  
(const valarray<T>&);
```

```
template<class T> valarray<T> log  
(const valarray<T>&);
```

```
template<class T> valarray<T> log10  
(const valarray<T>&);
```

```
template<class T> valarray<T> pow  
(const valarray<T>&, const valarray<T>&);
```

```
template<class T> valarray<T> pow  
(const valarray<T>&, const T&);
```

```
template<class T> valarray<T> pow  
(const T&, const valarray<T>&);
```

```
template<class T> valarray<T> sin  
(const valarray<T>&);
```

```
template<class T> valarray<T> sinh
```

```
(const valarray<T>&);  
  
template<class T> valarray<T> sqrt  
(const valarray<T>&);  
  
template<class T> valarray<T> tan  
(const valarray<T>&);  
  
template<class T> valarray<T> tanh  
(const valarray<T>&);
```

Remarks

A `valarray` object is returned with the individual elements initialized with the result of the corresponding operation.

Class slice

A `slice` is a set of indices that have three properties, a starting index, the number of elements and the distance between the elements.

Constructors

A constructor is overloaded to initialize an object with values or without values.

```
slice();  
  
slice(size_t start, size_t length, size_t stride);  
  
slice(const slice&);
```

slice access functions

The `slice` class has three member functions.

start

`Start` tells start is the position where the slice starts.

```
size_t start() const;
```

Remarks

The starting position is returned.

size

`Size` tells the size of the slice.

```
size_t size() const;
```

Remarks

The size of the slice is returned by the `size` member function.

stride

The distance between elements is given by the `stride` function.

```
size_t stride() const;
```

Remarks

The distance between each element is returned by `stride`.

Template Class Slice_array

The `slice_array` class is a helper class used by the slice subscript operator.

Constructors

Constructs a slice_array object.

```
private:  
    slice_array();  
    slice_array(const slice_array&);
```

Assignment Operator

The assignment operator allows for the initialization of a slice_array after construction.

```
void operator=(const valarray<T>&) const;  
slice_array& operator=(const slice_array&);
```

slice_array computed assignment

Several compound assignment operators are provided.

```
void operator*= (const valarray<T>&) const;
void operator/= (const valarray<T>&) const;
void operator%= (const valarray<T>&) const;
void operator+= (const valarray<T>&) const;
void operator-= (const valarray<T>&) const;
void operator^= (const valarray<T>&) const;
void operator&= (const valarray<T>&) const;
void operator|= (const valarray<T>&) const;
void operator<=>(const valarray<T>&) const;
void operator>=>(const valarray<T>&) const;
```

Remarks

There is no return for the compound operators.

Slice_array Fill Function

An assignment operation is provided to fill individual elements of the array.

```
void operator=(const T&);
```

Remarks

No value is returned.

Class Gslice

A general slice class is provided for multidimensional arrays.

Constructors

An overloaded constructor is provided for the creation of a `gslice` object.

```
gslice();  
  
gslice(size_t start, const valarray<size_t>& lengths,  
       const valarray<size_t>& strides);  
  
gslice(const gslice&);
```

Gslice Access Functions

The `gslice` class provides for access to the start, size and stride of the slice class.

start

The start function give the starting position.

```
size_t start() const;
```

Remarks

The starting position of the `gslice` is returned.

size

The size function returns the number of elements.

```
valarray<size_t> size() const;
```

Remarks

The number of elements as a `valarray` is returned.

stride

The stride function tells the size of each element.

```
valarray<size_t> stride() const;
```

Remarks

The size of the element as a valarray is returned.

Template Class Gslice_array

The `gslice_array` class is a helper class used by the `gslice` subscript operator.

Constructors

An overloaded constructor is provided for the creation of a `gslice_array` object.

```
gslice_array();  
gslice_array(const gslice_array&);
```

Assignment Operators

An assignment operator is provided for initializing a `gslice_array` after it has been created.

```
void operator=(const valarray<T>&) const;  
gslice_array& operator=(const gslice_array&);
```

Remarks

A copy of the modified `gslice_array` is returned for the second assignment operator.

Gslice_array Computed Assignment

Several compound assignment operators are provided.

```
void operator*= (const valarray<T>&) const;
void operator/= (const valarray<T>&) const;
void operator%= (const valarray<T>&) const;
void operator+= (const valarray<T>&) const;
void operator-= (const valarray<T>&) const;
void operator^= (const valarray<T>&) const;
void operator&= (const valarray<T>&) const;
void operator|= (const valarray<T>&) const;
void operator<<=(const valarray<T>&) const;
void operator>>=(const valarray<T>&) const;
```

Remarks

No return is given for the compound operators.

Fill Function

An assignment operation is provided to fill individual elements of the array.

```
void operator=(const T&);
```

Remarks

There is no return for the fill function.

Template Class Mask_array

The `mask_array` class is a helper class used by the mask subscript operator.

Constructors

An overloaded constructor is provided for creating a `mask_array` object.

```
private:  
    mask_array();  
    mask_array(const mask_array&);
```

Assignment Operators

An overloaded assignment operator is provided for assigning values to a `mask_array` after construction.

```
void operator=(const valarray<T>&) const;  
mask_array& operator=(const mask_array&);
```

Remarks

The copy assignment operator returns a `mask_array` reference.

Mask_array Computed Assignment

Several compound assignment operators are provided.

```
void operator*= (const valarray<T>&) const;
void operator/= (const valarray<T>&) const;
void operator%= (const valarray<T>&) const;
void operator+= (const valarray<T>&) const;
void operator-= (const valarray<T>&) const;
void operator^= (const valarray<T>&) const;
void operator&= (const valarray<T>&) const;
void operator|= (const valarray<T>&) const;
void operator<<=(const valarray<T>&) const;
void operator>>=(const valarray<T>&) const;
```

Remarks

There is no return value for the compound assignment operators.

Mask_array Fill Function

An assignment operation is provided to fill individual elements of the array.

```
void operator = (const T&);
```

Remarks

There is no return for the fill function.

Template Class Indirect_array

The `indirect_array` class is a helper class used by the indirect subscript operator.

This template is a helper template used by the indirect subscript operator
`indirect_array<T> valarray<T>::operator[](const valarray<size_t>&).`

It has reference semantics to a subset of an array specified by an `indirect_array`.

Constructors

An overloaded constructor is provided for creating a `indirect_array` object.

```
indirect_array();  
indirect_array(const indirect_array&);
```

Assignment Operators

An overloaded assignment operator is provided for assigning values to a `indirect_array` after construction.

```
void operator=(const valarray<T>&) const;  
indirect_array& operator=(const indirect_array&);
```

Remarks

The copy assignment operator returns a `indirect_array` reference.

Indirect_array Computed Assignment

Several compound assignment operators are provided.

```
void operator*= (const valarray<T>&) const;
void operator/= (const valarray<T>&) const;
void operator%= (const valarray<T>&) const;
void operator+= (const valarray<T>&) const;
void operator-= (const valarray<T>&) const;
void operator^= (const valarray<T>&) const;
void operator&= (const valarray<T>&) const;
void operator|= (const valarray<T>&) const;
void operator<<=(const valarray<T>&) const;
void operator>>=(const valarray<T>&) const;
```

Remarks

There is no return value for the compound assignment operators.

indirect_array fill function

An assignment operation is provided to fill individual elements of the array.

```
void operator=(const T&);
```

Remarks

There is no return for the fill function.

Generalized Numeric Operations

The standard library provides general algorithms for numeric processing.

Header <numeric>

The header <numeric> includes template functions for generalize numeric processing.

accumulate

Accumulate the sum of a sequence.

```
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

The sum of the values in a range or the some of the values after being processed by an operation is returned.

inner_product

Computes and returns the value of a product of the values in a range.

```
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);
```

Remarks

The value of the product starting with an initial value in a range is returned. In the function with the operation argument it is the product after the operation is performed.

partial_sum

Computes the partial sum of a sequence of numbers.

```
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);
```

The first computes the partial sum and sends it to the output iterator argument.

```
x, y, z
x, x+y, y+z.
```

The second form computes according to the operational argument and sends it to the output iterator argument. For example if the operational argument was a multiplication operation

```
x, y, z
x, x*y, y*z
```

Remarks

The range as the result plus the last minus the first.

adjacent_difference

Computed the adjacent difference in a sequence of numbers.

```
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);
```

The first computes the adjacent difference and sends it to the output iterator argument.

```
x, y, z  
x, y-x, z-y.
```

The second form computes according to the operational argument and sends it to the output iterator argument. For example if the operational argument was a division operation

```
x, y, z  
x, y/x, z/y
```

Remarks

The range as the result plus the last minus the first.

C Library

The standard provides for the math functions included in the standard C library with some overloading for various types.

<cmath>

The contents of the <cmath> headers is the same as the Standard C library headers <math.h> with the addition to the double versions of the math functions in <cmath>, C++ adds float and long double overloaded versions of some functions, with the same semantics.

<cstdlib>

The contents of the <cstdlib> headers is the same as the Standard C library headers <stdlib.h>. In addition to the int versions of certain math functions in <cstdlib>, C++ adds long overloaded versions of some functions, with the same semantics.

Listing 11.1 The Added C++ Signatures in Cstdlib and Cmath

```
long double abs (long double);  
long double acos (long double);  
long double asin (long double);  
long double atan (long double);  
long double atan2(long double, long double);  
long double ceil (long double);  
long double cos (long double);  
long double cosh (long double);  
long double exp (long double);  
long double fabs (long double);  
long double floor(long double);  
long double fmod (long double, long double);  
long double frexp(long double, int*);  
long double ldexp(long double, int);  
long double log (long double);  
long double log10(long double);  
long double modf (long double, long double*);  
long double pow (long double, long double);  
long double pow (long double, int);  
long double sin (long double);  
long double sinh (long double);  
long double sqrt (long double);  
long double tan (long double);  
long double tanh (long double);
```

```
float abs (float);  
float acos (float);  
float asin (float);  
float atan (float);  
float atan2(float, float);  
float ceil (float);  
float cos (float);  
float cosh (float);  
float exp (float);  
float fabs (float);  
float floor(float);  
float fmod (float, float);  
float frexp(float, int*);  
float ldexp(float, int);  
float log (float);  
float log10(float);  
float modf (float, float*);  
float pow (float, float);  
float pow (float, int);  
float sin (float);  
float sinh (float);  
float sqrt (float);  
float tan (float);  
float tanh (float);
```

```
double abs(double);  
double pow(double, int);
```

Complex Class

The header <complex> defines a template class, and facilities for representing and manipulating complex numbers.

The Complex Class Library

The header <complex> defines classes, operators, and functions for representing and manipulating complex numbers

The chapter is constructed in the following sub sections and mirrors clause 26.2 of the ISO (the International Organization for Standardization) C++ Standard :

- [“Header <complex>.”](#) shows the complex header class declarations
- [“Complex Specializations.”](#) lists the float, double and long double specializations
- [“Complex Template Class.”](#) 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.

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

Header <complex> forward declarations

The complex class has forward declarations.

- template<class T> class complex;
- template<> class complex<float>;
- template<> class complex<double>;
- template<> class complex<long double>;

Complex Specializations

The standard specialize the template complex class for float, double and long double types.

Complex Template Class

The template class complex contains Cartesian components `real` and `imag` for a complex number.

The complex class consists of:

- “[Constructors and Assignments](#),”
- “[Complex Member Functions](#),”
- “[Complex Class Operators](#),”

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.

Constructors and Assignments

Constructor, destructor and assignment operators and functions.

Constructors

Construct an object of a complex class.

```
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

An assignment operator for complex classes.

```
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](#)”
- “[imag](#)”

real

Retrieves the real component.

```
T real() const;
```

imag

Retrieves the imag component.

```
T imag() const;
```

Complex Class Operators

Several assignment operators are overloaded for the complex class manipulations.

- [“operator +=”](#)
- [“operator -=”](#)
- [“operator *=”](#)
- [“operator /=”](#)

operator +=

Adds and assigns to a complex class.

```
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.

The `this` pointer is returned.

operator -=

Subtracts and assigns from a complex class.

```
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.

The `this` pointer is returned.

operator *=

Multiplies by and assigns to a complex class.

```
complex<T>& operator*=(const T&);  
  
template<class X> complex<T>& operator*=  
(const complex<X>&);
```

Remarks

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.

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.

The `this` pointer is returned.

operator /=

Divides by and assigns to a complex class.

```
complex<T>& operator/=(const T&);  
  
template<class X> complex<T>& operator/=  
(const complex<X>&);
```

Remarks

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.

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.

The `this` pointer is returned.

Overloaded Operators and Functions

There are several non member functions and overloaded operators in the complex class library.

[“Overloaded Complex Operators”](#)

[“Complex Value Operations”](#)

[“Complex Transcendentals”](#)

Overloaded Complex Operators

The overloaded complex operators consists of:

- [“operator +”](#)
- [“operator -”](#)
- [“operator /”](#)
- [“operator !=”](#)
- [“operator >>”](#)
- [“operator <<”](#)

operator +

Adds to the complex class.

```
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.

The complex class after the addition.

operator -

Subtracts from the complex class.

```
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.

The complex class after the Subtraction.

operator *

Multiplies the complex class.

```
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.

The complex class after the multiplication.

operator /

Divides from the complex class.

```
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.

The complex class after the division.

operator ==

A boolean equality comparison.

```
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 !=

A boolean non equality comparison.

```
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 >>

Extracts a complex type from a stream.

```
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 <<

Inserts a complex number into a stream.

```
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](#)”
- “[imag](#)”
- “[abs](#)”
- “[arg](#)”
- “[norm](#)”
- “[conj](#)”
- “[polar](#)”

Complex Class
Complex Template Class

real

Retrieves the real component of a complex class.

```
template<class T> T real(const complex<T>&);
```

Remarks

Returns the real component of the argument.

imag

Retrieves the imaginary component of a complex class.

```
template<class T> T imag(const complex<T>&);
```

Remarks

Returns the imaginary component of the argument.

abs

Determines the absolute value of a complex class.

```
template<class T> T abs(const complex<T>&);
```

Remarks

Returns the absolute value of the complex class argument.

arg

Determines the phase angle.

```
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

Determines the squared magnitude.

```
template<class T> T norm(const complex<T>&);
```

Remarks

The squared magnitude of the complex class.

conj

Determines the complex conjugate.

```
template<class T> complex<T> conj(const complex<T>&);
```

Remarks

Returns the complex conjugate of the complex class argument.

polar

Determines the polar coordinates.

```
template<class T>
complex<T> polar(const T&, const T&);
```

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](#)”
- “[cosh](#)”
- “[exp](#)”
- “[log](#)”
- “[log10](#)”
- “[pow](#)”
- “[sin](#)”
- “[sinh](#)”
- “[sqrt](#)”
- “[tan](#)”
- “[tanh](#)”

cos

Determines the cosine.

```
template<class T> complex<T> cos (const complex<T>&);
```

Remarks

Returns the cosine of the complex class argument.

cosh

Determines the hyperbolic cosine.

```
template<class T> complex<T> cosh (const complex<T>&);
```

Remarks

Returns the cosine of the complex class argument.

exp

Determines the exponential.

```
template<class T> complex<T> exp (const complex<T>&);
```

Remarks

Returns the base exponential of the complex class argument.

log

Determines the natural base logarithm.

```
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, $\text{imag}(\log(cpx))$, is π .

log10

Determines the logarithm to base ten.

```
template<class T>  
complex<T> log10(const complex<T>&);
```

Remarks

Returns the logarithm base(10) of the argument cpx defined as $\log(cpx) / \log(10)$.

Complex Class

Complex Template Class

pow

Raises the complex class to a set power.

```
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

Determines the sine.

```
template<class T>  
complex<T> sin (const complex<T>&);
```

Remarks

Returns the sine of the complex class argument.

sinh

Determines the hyperbolic sine.

```
template<class T>  
  
complex<T> sinh (const complex<T>&);
```

Remarks

Returns the hyperbolic sine of the complex class argument.

sqrt

Determines the square root.

```
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

Determines the tangent.

```
template<class T>  
  
complex<T> tan (const complex<T>&);
```

Remarks

Returns the tangent of the complex class argument.

tanh

Determines the hyperbolic tangent.

```
template<class T>  
complex<T> tanh (const complex<T>&);
```

Remarks

Returns the hyperbolic tangent of the complex class argument.

Input and Output Library

A listing of the set of components that C++ programs may use to perform input/output operations.

The Input and Output Library

The chapter is constructed in the following sub sections and mirrors clause 27.1 of the ISO (the International Organization for Standardization) C++ Standard :

- [“Input and Output Library Summary”](#)
- [“Iostreams requirements”](#)

Input and Output Library Summary

This library includes the headers.

Listing 13.1 Input/Output Library Summary

Include	Purpose
<iostreamfwd>	Forward declarations
<iostream>	Standard iostream objects
<ios>	Iostream base classes
<streambuf>	Stream buffers
<iostream>	Formatting and manipulators
<ostream>	Output streams
<iomanip>	Input and output manipulators
<sstream>	String streams
<cstdlib>	Standard C utilities

Listing 13.1 Input/Output Library Summary

Include	Purpose
<fstream>	File Streams
<cstdio>	Standard C input and output support
<cwchar>	Standard C wide characters support

iostreams requirements

No requirements library has been defined.

Topics in this section are:

- [“Definitions”](#)
- [“Type requirements”](#)
- [“Type SZ_T”](#)

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 - 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 previously encountered position. An arbitrary-positional stream can integral position within the length of the stream.

Type requirements

Several types are required by the standards, they are consolidated in strings (chapter 21.)

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.

Forward Declarations

The header `<iostreamfwd>` is used for forward declarations of template classes.

The non-standard header `<stringfwd>` is used for forward declarations of string class objects.

The Streams and String 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.

The chapter mirrors clause 27.2 of the ISO (the International Organization for Standardization) C++ Standard :

Header `<iostfwd>`

The header `<iostfwd>` is used for forward declarations of template classes.

Remarks

The template class `basic_ios<charT, traits>` serves as a base class for class `basic_istream` and `basic_ostream`.

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`.

Forward Declarations

Header `<stringfwd>`

Header `<stringfwd>`

This non-standard header can be used to forward declare `basic_string` (much like `<iostream>` forward declares streams). There is also a `<stringfwd.h>` that forward declares `basic_string` and places it into the global namespace.

NOTE

The header `<stringfwd>` is a non standard header.

Listing 14.1 Header `<stringfwd>` Synopsis

```
namespace std { // Optional
template <class T>    class allocator;
template<class charT>   struct char_traits;
template <class charT, class traits, class Allocator>
class basic_string;

typedef basic_string <char, char_traits<char>, allocator<char> >
string;
typedef basic_string
<wchar_t, char_traits<wchar_t>, allocator<wchar_t> > wstring;
}
```

Including `<stringfwd>` allows you to use a string object.

Listing 14.2 Example of `<stringfwd>` Inclusion of `std::string`

```
#include <stringfwd>
class MyClass
{
...
    std::string* my_string_ptr;
};
```

The headers `<stringfwd.h>` and `<string>` can be used in combination to place `string` into the global namespace, much like is done with other `<name.h>` headers. The header `<string.h>` does not work because that is a standard C header.

Listing 14.3 Example of Stringfwd usage

```
#include <stringfwd.h>
#include <string>
```

```
int main()
{
    string a("Hi");    // no std:: required
    return 0;
}
```

Forward Declarations

Header <stringfwd>

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 <cstdio>.

The Standard Input and Output Stream Library

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.

The chapter is constructed in the following sub sections and mirrors clause 27.3 of the ISO (the International Organization for Standardization) C++ Standard :

- [Narrow stream objects](#)
- [Wide stream objects](#)

Header <iostream>

The header <iostream> declares standard input and output objects in namespace std.

Listing 15.1 Iostream input and output objects

```
extern istream cin;
extern ostream cout;
extern ostream cerr;
extern ostream clog;
extern wistream wcin;
extern wostream wcout;
extern wostream cerr;
extern wostream wclog;
```

Stream Buffering

All streams are buffered (by default) except `cerr` and `wcerr`.

NOTE You can change the buffering characteristic of a stream with:

```
cout.setf(ios_base::unitbuf);  
or  
cerr.unsetf(ios_base::unitbuf);
```

Narrow stream objects

Narrow stream objects provide unbuffered input and output associated with standard input and output declared in `<cstdio>`.

istream cin

An unbuffered input stream.

```
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`.

Returns an `istream` object;

ostream cout

An unbuffered output stream.

```
ostream cout;
```

Remarks

The object `cout` controls output to an unbuffered stream buffer associated with `stdout` declared in `<cstdio>`.

An ostream object;

ostream cerr

Controls output to an unbuffered stream.

```
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.

An ostream object;

ostream clog

Controls output to a stream buffer.

```
ostream clog;
```

Remarks

The object `clog` controls output to a stream buffer associated with `cerr` declared in `<cstdio>`.

An ostream object;

Wide stream objects

Wide stream objects provide unbuffered input and output associated with standard input and output declared in `<cstdio>`.

iostream Objects

Header <iostream>

wistream wcin

An unbuffered wide input stream.

```
wistream wcin;
```

Remarks

The object `win` controls input from an unbuffered wide stream buffer associated with `stdin` declared in `<cstdio>`. After `wcin` is initialized `win.tie()` returns `wout`.

A wistream object;

wostream wcout

An unbuffered wide output stream.

```
wostream wcout;
```

Remarks

The object `cout` controls output to an unbuffered wide stream buffer associated with `stdout` declared in `<cstdio>`.

A wostream object;

wostream wcerr

Controls output to an unbuffered wide stream.

```
wostream wcerr;
```

Remarks

The object `werr` controls output to an unbuffered wide stream buffer associated with `stderr` declared in `<cstdio>`. After `werr` is initialized, `werr.flags()` and `unitbuf` is nonzero.

A `wostream` object;

wostream wlog

Controls output to a wide stream buffer.

```
wostream wlog;
```

Remarks

The object `wlog` controls output to a wide stream buffer associated with `cerr` declared in `<cstdio>`.

A `wostream` object

iostream Objects

Header <iostream>

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.

Input and Output Stream Base Library

The chapter is constructed in the following sub sections and mirrors clause 27.4 of the ISO (the International Organization for Standardization) C++ Standard :

- [“Header `<ios>`”](#)
- [“Typedef Declarations”](#)
- [“Class `ios_base`”](#)
- [“Template class `basic_ios`”](#)
- [“`ios_base` manipulators”](#)

Header `<ios>`

The header file `<ios>` provides for implementation of stream objects for standard input and output.

Template Class `fpos`

The template class `fpos<stateT>` is a class used for specifying file position information. The template parameter corresponds to the type needed to hold state information in a multi-byte sequence (typically `mbstate_t` from `<cwchar>`). `fpos` is essentially a wrapper for whatever mechanisms are necessary to hold a stream position (and multi-byte state). In fact the standard stream position typedefs are defined in terms of `fpos`:

```
typedef fpos<mbstate_t> streampos;
typedef fpos<mbstate_t> wstreampos;
```

The template class `fpos` is typically used in the `istream` and `ostream` classes in calls involving file position such as `tellg`, `tellp`, `seekg` and `seekp`. Though in these classes the `fpos` is typedef'd to `pos_type`, and can be changed to a custom implementation by specifying a traits class in the stream's template parameters.

TypeDef Declarations

The following typedef's are defined in the class `ios_base`.

```
typedef long streamoff;
typedef long streamsize;
```

Class `ios_base`

A base class for input and output stream mechanisms

The prototype is listed below. Additional topics in this section are:

- [“Typedef Declarations”](#)
- [“Class `ios_base::failure`”](#)
- [“`failure`”](#)
- [“Type `fmtflags`”](#)
- [“Type `iostate`”](#)
- [“Type `openmode`”](#)
- [“Type `seekdir`”](#)
- [“Class `Init`”](#)
- [“Class `Init Constructor`”](#)
- [“`ios_base fmtflags state functions`”](#)
- [“`ios_base locale functions`”](#)
- [“`ios_base storage function`”](#)
- [“`ios_base Constructor`”](#)

The `ios_base` class is a base class and includes many enumerations and mechanisms necessary for input and output operations.

Typedef Declarations

No types are specified in the current standards.

Class ios_base::failure

Defines a base class for types of object thrown as exceptions.

failure

Construct a class failure.

```
explicit failure(const string& msg);
```

Remarks

The function `failure()` construct a class failure initializing with `exception(msg)`.

failure::what

To return the exception message.

```
const char *what() const;
```

Remarks

The function `what()` is use to deliver the `msg.str()`.

Returns the message with which the exception was created.

Type fmtflags

An enumeration used to set various formatting flags for reading and writing of streams.

Iostreams Base Classes

Class *ios_base*

Table 16.1 Format Flags Enumerations

Flag	Effects when set
boolalpha	insert or extract bool type in alphabetic form
dec	decimal output
fixed	when set shows floating point numbers in normal manner, six decimal places is default
hex	hexadecimal output
oct	octal output
left	left justified
right	right justified
internal	pad a field between signs or base characters
scientific	show scientific notation for floating point numbers
showbase	shows the bases numeric values
showpoint	shows the decimal point and trailing zeros
showpos	shows the leading plus sign for positive numbers
skipws	skip leading white spaces with input
unitbuf	buffer the output and flush after insertion operation
uppercase	show the scientific notation, x or o in uppercase

Table 16.2 Format flag field constants

Constants	Allowable values
adjustfield	left right internal
basefield	dec oct hex
floatfield	scientific fixed

Listing 16.1 Example of *ios* format flags usage

see `basic_ios::setf()` and `basic_ios::unsetf()`

Type iostate

An enumeration that is used to define the various states of a stream.

Table 16.3 Enumeration iostate

Flags	Usage
goodbit	True when all of badbit, eofbit and failbit are false.
badbit	True when the stream is in an irrecoverable error state (such as failure due to lack of memory)
failbit	True when a read or a write has failed for any reason (This can happen for example when the input read a character while attempting to read an integer.)
eofbit	True when the end of the stream has been detected. Note that eofbit can be set during a read, and yet the read may still succeed (failbit not set). (This can happen for example when an integer is the last character in a file.) note: see variance from AT&T standard

For an example of ios iostate flags usage refer to `basic_ios::setstate()` and `basic_ios::rdstate()`

Type openmode

An enumeration that is used to specify various file opening modes.

Table 16.4 Enumeration openmode

Mode	Definition
app	Start the read or write at end of the file
ate	Start the read or write immediately at the end
binary	binary file
in	Start the read at end of the stream
out	Start the write at the beginning of the stream
trunc	Start the read or write at the beginning of the stream

Type seekdir

An enumeration to position a pointer to a specific place in a file stream.

Table 16.5 Enumeration seekdir

Enumeration	Position
beg	Begging of stream
cur	Current position of stream
end	End of stream

For an example of ios seekdir usage refer to `streambuf::pubseekoff`

Class Init

An object that associates `<iostream>` object buffers with standard stream declared in `<cstdio>`.

Class Init Constructor

To construct an object of class Init;

```
Init();
```

Remarks

The default 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

```
~Init();
```

Remarks

The destructor subtracts one from init_cnt and if the result is one calls cout.flush(), cerr.flush() and clog.flush().

ios_base fmtflags state functions

To set the state of the ios_base format flags.

flags

To alter formatting flags using a mask.

```
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.

The fmtflags type before alterations.

See ios enumerators for a list of fmtflags.

See Also:

setiosflags() and resetiosflags()

Listing 16.2 Example of flags() usage:

```
#include <iostream>  
  
// showf() displays flag settings  
void showf();  
  
int main()  
{  
    using namespace std;  
    showf(); // show format flags
```

Iostreams Base Classes

Class ios_base

```
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",
    "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
```

iostreams Base Classes

Class ios_base

setf

Set the stream format flags.

```
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.

type `basic_ios::fmtflags` is returned.

Listing 16.3 Example of setf() usage:

```
#include <iostream>

int main()
{
using namespace std;

    double d = 10.01;

    cout.setf(ios::showpos | ios::showpoint);
    cout << d << endl;
    cout.setf(ios::showpoint, ios::showpos | ios::showpoint);
    cout << d << endl;

    return 0;
}
```

Result:

```
+10.01
10.01
```

unsetf

To un-set previously set formatting flags.

```
void unsetf(fmtflags)
```

Remarks

Use the `unsetf()` function to reset any format flags to a previous condition. You would normally store the return value of `setf()` in order to achieve this task.

There is no return.

Listing 16.4 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

iostreams Base Classes

Class ios_base

precision

Set and return the current format precision.

```
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.

This means that `ios::scientific` will have one more significant digit than `ios::floatfield`, and `ios::fixed` will have a varying number of digits.

The current value set.

See Also

```
setprecision()
```

Listing 16.5 Example of precision() usage:

```
#include <iostream>  
#include <cmath>  
  
const double pi = 4 * std::atan(1.0);  
  
int main()  
{  
    using namespace std;  
  
    double TenPi = 10*pi;  
  
    cout.precision(5);  
    cout.unsetf(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

To set the width of the output field.

```
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.

Width is the one and only modifier that is not sticky and needs to be reset with each use. Width is reset to `width(0)` after each use.

The previous width setting is returned.

Listing 16.6 Example of width() usage:

```
#include <iostream>

int main()
{
using namespace std;

    int width;
```

Iostreams Base Classes

Class *ios_base*

```
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!*****
```

ios_base locale functions

Sets the locale for input output operations.

imbue

Stores a value representing the locale.

```
locale imbue(const locale loc);
```

Remarks

The precondition of the argument loc is equal to getloc().

The previous value of `getloc()`.

getloc

Determined the imbued locale for input output operations.

```
locale getloc() const;
```

Remarks

The global C++ locale if no locale has been imbued. Otherwise it returns the locale of the input and output operations.

ios_base storage function

To allocate storage pointers.

xalloc

Allocation function.

```
static int xalloc()
```

Remarks

Returns index++.

iword

Allocate an array of `int` and store a pointer.

Remarks

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.

iostreams Base Classes

Class ios_base

After a subsequent call to iword() for the same object the return value may be invalid.

Returns `iarray[idx]`

pword

Allocate an array of pointers.

```
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].

After a subsequent call to pword() for the same object the return value may be invalid.

Returns `parray[idx]`.

register_callback

Registers functions when an event occurs.

```
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`. Functions 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.

Identical pairs are not merged and a function registered twice will be called twice.

sync_with_stdio

Synchronizes stream input output with 'C' input and output functions.

```
static bool sync_with_stdio(bool sync = true);
```

Remarks

Is not supported in the Metrowerks Standard Library.

Always returns `true` indicating that the MSLstreams are always synchronized with the C streams.

ios_base Constructor

ios_base Constructor

Construct and destruct an object of class `ios_base`

`protected:`

```
ios_base();
```

Remarks

The `ios_base` constructor is protected so it may only be derived from. If the values of the `ios_base` members are undermined.

Destructor

```
~ios_base();
```

Remarks

Calls registered callbacks and destroys an object of class `ios_base`.

Template class `basic_ios`

A template class for input and output streams.

The prototype is listed below. Additional topics in this section are:

- [“basic_ios Constructor”](#)
- [“Basic_ios Member Functions”](#)
- [“basic_ios iostate flags functions”](#)

The `basic_ios` template class is a base class and includes many enumerations and mechanisms necessary for input and output operations.

basic_ios Constructor

Construct an object of class `basic_ios` and assign values.

```
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

```
virtual ~basic_ios();
```

Remarks

Destroys an object of type `basic_ios`.

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()
iarray	a null pointer
parray	a null pointer

Basic_ios Member Functions

Member functions of the class `basic_ios`.

tie

To tie an `ostream` to the calling stream.

```
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. The `tie()` function with an argument ties the new object to the `ostream` and returns a pointer if any from the first. The post-condition of `tie()` function that takes the argument `tiestr` is that `tiestr` is equal to `tie()`;

A pointer to type `ostream` that is or previously was tied, or zero if there was none.

Listing 16.7 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

To retrieve a pointer to the stream buffer.

```
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 post-condition of `sb` is equal to `rdbuf()`.

Returns a pointer to `basic_streambuf` object.

Listing 16.8 Example of `rdbuf()` usage:

```
#include <iostream>

struct address {
    int number;
    char street[40];
} addbook;

int main()
```

iostreams Base Classes

Template class basic_ios

```
{  
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

Stores a value representing the locale.

```
locale imbue(const locale& rhs);
```

Remarks

The function `imbue()` calls `ios_base::imbue()` and `rdbuf->pubimbue()`.

Returns the current locale.

fill

To insert characters into the stream's unused spaces.

```
char_type fill() const  
char_type fill(char_type)
```

Remarks

Use `fill(char_type)` in output to fill blank spaces with a character. The function `fill()` is overloaded to return the current filler without altering it.

Returns the current character being used as a filler.

See Also

`manipulator setfill()`

Listing 16.9 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 *

iostreams Base Classes

Template class basic_ios

copyfmt

Copies a basic_ios object.

```
basic_ios& copyfmt(const basic_ios& rhs);
```

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 `iword` arrays are copied not the pointers themselves.

Returns the `this` pointer.

basic_ios iostate flags functions

To set flags pertaining to the state of the input and output streams.

operator bool

A `bool` operator.

```
operator bool() const;
```

Remarks

Returns `!fail()`.

operator !

A `bool not` operator.

```
bool operator !();
```

Remarks

Returns `fail()`.

rdstate

To retrieve the state of the current formatting flags.

```
iostate 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 `iostate` used in `ios::setstate()`

See Also

```
ios::setstate()
```

Listing 16.10 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)
```

iostreams Base Classes

Template class basic_ios

```
{  
    // simulate a bad bit  
    if(count++ == 12) in.setstate(ios::badbit);  
    status(in);  
}  
  
status(in);  
in.close();  
return 0;  
}  
  
void status(ifstream &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  
Fatal I/O Error
```

clear

Clears `iostate` field.

```
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. Post-condition of `clear` is the argument is equal to `rdstate()`.

If `rdstate()` and `exceptions() != 0` an exception is thrown.

No value is returned.

Listing 16.11 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)
```

iostreams Base Classes

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  
Non-Fatal I/O Error
```

setstate

To set the state of the format flags.

```
void setstate(iostate state) throw(failure);
```

Remarks

Calls `clear(rdstate() | state)` and may throw and exception.

There is no return value.

For an example of `setstate()` usage refer to `ios::rdstate()`

good

To test for the lack of error bits being set.

```
bool good() const;
```

Remarks

Use the function `good()` to test for the lack of error bits being set.

Returns true if `rdstate() == 0`.

For an example of `good()` usage refer to `basic_ios::bad()`

eof

To test for the eofbit setting.

```
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.

If `eofbit` is set in `rdstate()` true is returned.

Listing 16.12 Example of `eof()` usage:

MW Reference is simply a one line text document

```
ABCDEFGHIJKLMNPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz
```

```
#include <iostream>
```

Iostreams Base Classes

Template class basic_ios

```
#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

To test for stream reading failure from any cause.

```
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 16.13 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);
}

return 0;
}
```

Result:

```
float 33.33 double 3.16e+10 integer 789 character C
```

iostreams Base Classes

Template class basic_ios

bad

To test for fatal I/O error.

```
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 16.14 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(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 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

exceptions

To handle basic_ios exceptions.

```
iostate exceptions() const;

void exceptions(iostate except);
```

Remarks

The function `exceptions()` determines what elements in `rdstate()` cause exceptions to be thrown. The overloaded `exceptions(iostate)` calls `clear(rdstate())` and leaves the argument `except` equal to `exceptions()`.

Returns a mask that determines what elements are set in rdstate().

ios_base manipulators

To provide an in line input and output formatting mechanism.

The topics in this section are:

- “[fmtflags manipulators](#)”
- “[adjustfield manipulators](#)”
- “[basefield manipulators](#)”
- “[floatfield manipulators](#)”

fmtflags manipulators

To provide an in line input and output numerical formatting mechanism.

Remarks

Manipulators are used in the stream to alter the formatting of the stream.

A reference to an object of type `ios_base` is returned to the stream. (The `this` pointer.)

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& showpoint (ios_base&)</code>	show decimal point
<code>ios_base& noshowpoint(ios_base&)</code>	do not show decimal point

Table 16.7 Prototype of ios_base manipulators

Manipulator	Definition
ios_base& showpos(ios_base&)	show the positive sign
ios_base& noshowpos(ios_base&)	do not show positive sign
ios_base& skipws(ios_base&)	input only skip white spaces
ios_base& noskipws(ios_base&)	input only no skip white spaces
ios_base& uppercase(ios_base&)	show scientific in uppercase
ios_base& nouppercase (ios_base&)	do not show scientific in uppercase
ios_base& unitbuf (ios_base::unitbuf)	set the unitbuf flag
ios_base& nounitbuf (ios_base::unitbuf)	unset the unitbuf flag

adjustfield manipulators

To provide an in line input and output orientation formatting mechanism.

Remarks

Manipulators are used in the stream to alter the formatting of the stream.

A reference to an object of type `ios_base` is returned to the stream. (The `this` pointer.)

Table 16.8 Adjustfield manipulators

Manipulator	Definition
ios_base& internal(ios_base&)	fill between indicator and value
ios_base& left(ios_base&)	left justify in a field
ios_base& right(ios_base&)	right justify in a field

basefield manipulators

To provide an in line input and output numerical formatting mechanism.

Remarks

Manipulators are used in the stream to alter the formatting of the stream.

A reference to an object of type `ios_base` is returned to the stream. (The `this` pointer.)

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

floatfield manipulators

To provide an in line input and output numerical formatting mechanism.

Remarks

Manipulators are used in the stream to alter the formatting of the stream.

A reference to an object of type `ios_base` is returned to the stream. (The `this` pointer.)

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

Listing 16.15 Example of manipulator usage:

```
#include <iostream>
#include <iomanip>

int main()
```

```
{  
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

To provide an in line formatting mechanism. The basic template for parameterless manipulators is shown in [“Basic parameterless manipulator”](#)

Listing 16.16 Basic parameterless manipulator

```
ostream &manip-name(ostream &stream)  
{  
    // coding  
    return stream;  
}
```

Remarks

Use overloaded manipulators to provide specific and unique formatting methods relative to one class.

A reference to `ostream`. (Usually the `this` pointer.)

See Also

`<iomanip>` for manipulators with parameters

Listing 16.17 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

Stream Buffers

The header `<streambuf>` defines types that control input and output to character sequences.

The Stream Buffers Library

The chapter is constructed in the following sub sections and mirrors clause 27.5 of the ISO (the International Organization for Standardization) C++ Standard :

- [“Header <streambuf>”](#)
- [“Stream buffer requirements”](#)
- [“Class basic_streambuf<charT, traits>”](#)

Listing 17.1 Header <streambuf>

```
namespace std {
template <class charT, class traits = char_traits<charT> >
    class basic_streambuf;
typedef basic_streambuf<char> streambuf;
typedef basic_streambuf<wchar_t> wstreambuf;
}
```

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
- The sequences can support operations to or from associated sequences.

Stream Buffers

Class `basic_streambuf<charT, traits>`

- 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`)

Class `basic_streambuf<charT, traits>`

The prototype is listed below. Additional topics in this section are:

- [“basic_streambuf Constructor”](#)
- [“basic_streambuf Public Member Functions”](#)
- [“Locales”](#)
- [“Buffer Management and Positioning”](#)
- [“Get Area”](#)
- [“Putback”](#)
- [“Put Area”](#)
- [“basic_streambuf Protected Member Functions”](#)
- [“Get Area Access”](#)
- [“Put Area Access”](#)
- [“basic_streambuf Virtual Functions”](#)
- [“Locales”](#)
- [“Buffer Management and Positioning”](#)
- [“Get Area”](#)
- [“Putback”](#)
- [“Put Area”](#)

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.

basic_streambuf Constructor

The default constructor constructs an object of type `basic_streambuf`.

`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

```
virtual ~basic_streambuf();
```

Remarks

Removes the object from memory.

basic_streambuf Public Member Functions

The public member functions allow access to member functions from derived classes.

Locales

Locales are used for encapsulation and manipulation of information to a particular locale.

Stream Buffers

Class *basic_streambuf<charT, traits>*

basic_streambuf::pubimbue

To set the locale.

```
locale pubimbue(const locale &loc);
```

Remarks

The function pubimbue calls imbue(loc).

Returns the previous value of `getloc()`.

basic_streambuf::getloc

To get the locale.

```
locale getloc() const;
```

Remarks

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.

Buffer Management and Positioning

Functions used to manipulate the buffer and the input and output positioning pointers.

basic_streambuf::pubsetbuf

To set an allocation after construction.

```
basic_streambuf<char_type, traits> *pubsetbuf  
(char_type* s, streamsize n);
```

Remarks

The first argument is used in another function by a `filebuf` derived class. See `setbuf()`. The second argument is used to set the size of a dynamic allocated buffer.

Returns a pointer to `basic_streambuf<char_type, traits>` via `setbuf(s, n)`.

Listing 17.2 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

Stream Buffers

Class `basic_streambuf<charT, traits>`

basic_streambuf::pubseekoff

Determines the position of the get pointer.

```
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.

Returns a `pos_type` via `seekoff(off, way, which)`

See Also

`pubseekpos()`

Listing 17.3 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

Determine and move to a desired offset.

```
pos_type pubseekpos  
(pos_type sp,  
ios_base::openmode which = ios::in | ios::out);
```

Remarks

The function `pubseekpos()` is used to move to a desired offset using a type `pos_type`, which holds all necessary information.

Returns a `pos_type` via `seekpos(sb, which)`

See Also

`pubseekoff()`, `seekoff()`

Listing 17.4 Example of streambuf::pubseekpos() usage:

The file MW Reference contains: ABCDEFGHIJKLMNOPQRSTUVWXYZ

```
#include <iostream>
#include <fstream>
#include <cstdlib>
```

Stream Buffers

Class `basic_streambuf<charT, traits>`

```
int main()
{
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

To synchronize the `streambuf` object with its input/output.

```
int pubsync();
```

Remarks

The function `pubsync()` will attempt to synchronize the `streambuf` input and output.

Returns zero if successful or EOF if not via `sync()`.

Listing 17.5 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
Enter your street name: Donley Drive
```

Stream Buffers

Class basic_streambuf<charT, traits>

Your address is: 2201 Donley Drive

Get Area

Public functions for retrieving input from a buffer.

basic_streambuf::in_avail

To test for availability of input stream.

```
streamsize in_avail();
```

Remarks

If a read is permitted returns size of stream as a type `streamsize`.

basic_streambuf::snextc

To retrieve the next character in a stream.

```
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.

If `sbumpc` returns `traits::eof` returns that, otherwise returns `sgetc()`.

Listing 17.6 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

To move the get pointer.

```
int_type sbumpc();
```

Remarks

The function `sbumpc()` moves the get pointer one element when called.

Return

The value of the character at the `get` pointer. It returns `uflow()` if it fails to move the pointer.

See Also

`sgetc()`

Stream Buffers

Class `basic_streambuf<charT, traits>`

Listing 17.7 Example of `streambuf::sbumpc()` usage:

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

const int size = 100;
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

To extract a character from the stream.

```
int_type sgetc();
```

Remarks

The function `sgetc()` extracts a single character, without moving the get pointer.

A `int_type` type at the `get` pointer if available otherwise returns `underflow()`.

For an example of `streambuf::sgetc()` usage refer to `streambuf::sbumpc()`

basic_streambuf::sgetn

To extract a series of characters from the stream.

```
streamsize sgetn(char_type *s, streamsize n);
```

Remarks

The public member function `sgetn()` is used to extract a series of characters from the stream buffer. After the operation, the `get` pointer references the character following the last character extracted.

Returns a `streamsize` type as returned from the function `xsgetn(s,n)`.

For an example of `streambuf::sgetn()` usage refer to `pubsetbuf()`

Putback

Public functions to return a value to a stream.

basic_streambuf::sputback

To put a character back into the stream.

```
int_type sputback(char_type c);
```

Remarks

The function `sputbackc()` 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.

Stream Buffers

Class `basic_streambuf<charT, traits>`

If successful returns a pointer to the `get` pointer as an `int_type` otherwise returns `pbackfail(c)`.

Listing 17.8 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:
AxB
xBcDEF

basic_streambuf::sungetc

To restore a character extracted.

```
int_type sungetc();
```

Remarks

The function `sungetc()` restores the previously extracted character. After the operation, the `get` pointer references the last character extracted.

If successful returns a pointer to the `get` pointer as an `int_type` otherwise returns `pbackfail(c)`.

For an example of `streambuf::sungetc()` usage refer to `streambuf::sputbackc()`

Put Area

Public functions for inputting characters into a buffer.

basic_streambuf::sputc

To insert a character in the stream.

```
int_type sputc(char_type c);
```

Stream Buffers

Class `basic_streambuf<charT, traits>`

Remarks

The function `sputc()` inserts a character into the stream. After the operation, the get pointer references the character following the last character extracted.

If successful returns `c` as an `int_type` otherwise returns `overflow(c)`.

Listing 17.9 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

To insert a series of characters into a stream.

```
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.

Returns a `streamsize` type returned from a call to `xputn(s, n)`.

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.

Get Area Access

Member functions for extracting information from a stream.

basic_streambuf::eback

Retrieve the beginning pointer for stream input.

```
char_type* eback() const;
```

Remarks

Returns the beginning pointer.

basic_streambuf::gptr

Retrieve the next pointer for stream input.

```
char_type* gptr() const;
```

Remarks

Returns the next pointer.

basic_streambuf::egptr

Retrieve the end pointer for stream input.

```
char_type* egptr() const;
```

Stream Buffers

Class *basic_streambuf<charT, traits>*

Remarks

Returns the end pointer.

basic_streambuf::gbump

Advances the next pointer for stream input.

```
void gbump(int n);
```

Remarks

The function `gbump()` advances the input pointer by the value of the `int n` argument.

basic_streambuf::setg

To set the beginning, next and end pointers.

```
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()`.

Put Area Access

Protected member functions for stream output sequences.

basic_streambuf::pbase

To retrieve the beginning pointer for stream output.

```
char_type* pbase() const;
```

Remarks

Returns the beginning pointer.

basic_streambuf::pptr

To retrieve the next pointer for stream output.

```
char_type* pptr() const;
```

Remarks

Returns the next pointer.

basic_streambuf::eptr

To retrieve the end pointer for stream output.

```
char_type* eptr() const;
```

Remarks

Returns the end pointer.

basic_streambuf::pbump

To advance the next pointer for stream output.

```
void pbump(int n);
```

Stream Buffers

Class `basic_streambuf<charT, traits>`

Remarks

The function `pbump()` advances the `next` pointer by the value of the `int` argument `n`.

basic_streambuf::setp

To set the values for the beginning, next and end pointers.

```
void setp  
    (char_type* pbeg,  
     char_type* pend);
```

Remarks

After the call to `setp()`, `pbeg` equals `pbase()`, `pbeg` equals `pptr()` and `pend` equals `eptr()`.

basic_streambuf Virtual Functions

The virtual functions in `basic_streambuf` class are to be overloaded in any derived class.

Locales

To get and set the stream locale. These functions should be overridden in derived classes.

basic_streambuf::imbue

To change any translations base on locale.

```
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.

Buffer Management and Positioning

Virtual functions for positioning and manipulating the stream buffer. These functions should be overridden in derived classes.

basic_streambuf::setbuf

To set a buffer for stream input and output sequences.

```
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.

Returns the `this` pointer.

basic_streambuf::seekoff

To return an offset of the current pointer in an input or output streams.

```
virtual pos_type seekoff  
    (off_type off,  
     ios_base::seekdir way,  
     ios_base::openmode which = ios::in | ios::out);
```

Stream Buffers

Class basic_streambuf<charT, traits>

Remarks

The function `seekoff()` is overridden in `basic_stringbuf` and `basic_filebuf` classes.

Returns a `pos_type` value, which is an invalid stream position.

basic_streambuf::seekpos

To alter an input or output stream position.

```
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.

Returns a `pos_type` value, which is an invalid stream position.

basic_streambuf::sync

To synchronize the controlled sequences in arrays.

```
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.

Returns zero if successful and -1 if failure occurs.

Get Area

Virtual functions for extracting information from an input stream buffer. These functions should be overridden in derived classes.

basic_streambuf::showmanc

Shows how many characters in an input stream

```
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.

Returns zero for normal behavior and negative or positive one.

basic_streambuf::xsgetn

To read a number of characters from an input stream buffer.

```
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.

Returns the number of characters read.

Stream Buffers

Class basic_streambuf<charT, traits>

basic_streambuf::underflow

To show an underflow condition and not increment the get pointer.

```
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.

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.

basic_streambuf::uflow

To show a underflow condition for a single character and increment the get pointer.

```
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.

Calls `underflow()` and if `traits::eof` is not returned returns the integer value of the get pointer and increments the next pointer for input.

Putback

Virtual functions for replacing data to a stream. These functions should be overridden in derived classes.

basic_streambuf::pbackfail

To show a failure in a put back operation.

```
virtual int_type pbackfail  
(int_type c = traits::eof());
```

Remarks

The resulting conditions are the same as the function `underflow()`.

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.

Put Area

Virtual function for inserting data into an output stream buffer. These functions should be overridden in derived classes.

basic_streambuf::xsputn

Write a number of characters to an output buffer.

```
virtual streamsize xsputn  
(const char_type *s, streamsize n);
```

Remarks

The function `xsputn()` writes to the output character by using repeated calls to `sputc(c)`. Write stops when `n` characters have been written or EOF is encountered.

Returns the number of characters written in a type `streamsize`.

Stream Buffers

Class `basic_streambuf<charT, traits>`

basic_streambuf::overflow

Consumes the pending characters of an output sequence.

```
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.

The function returns `traits::eof()` for failure or some unspecified result to indicate success.

Formatting Manipulators

This chapter discusses formatting and manipulators in the input/output library.

The Formatting and Manipulators Library

There are three headers—`<iostream>`, `<ostream>`, and `<iomanip>`—that contain stream formatting and manipulator routines and implementations.

The chapter is constructed in the following sub sections and mirrors clause 27.6 of the ISO (the International Organization for Standardization) C++ Standard :

- [“Headers”](#)
- [“Input Streams”](#)
- [“Output streams”](#)
- [“Standard manipulators”](#)

Headers

This section lists the header for `istream`, `ostream`, and `iomanip`.

- Header `<iostream>` for input streams
- Header `<ostream>` for output streams
- Header `<iomanip>` for input and output manipulation

Input Streams

The header `<iostream>` controls input from a stream buffer.

The topics in this section are:

- [“Template class `basic_istream`”](#)
- [“`basic_istream` Constructors”](#)
- [“Class `basic_istream::sentry`”](#)

-
- “[Formatted input functions](#)”
 - “[Common requirements](#)”
 - “[Arithmetic Extractors Operator >>](#)”
 - “[basic_istream extractor operator >>](#)”
 - “[Unformatted input functions](#)”
 - “[Standard basic_istream manipulators](#)”
 - “[basic_iostream Constructor](#)”

Template class **basic_istream**

A class that defines several functions for stream input mechanisms from a controlled stream buffer.

The `basic_istream` class is derived from the `basic_ios` class and provides many functions for input operations.

basic_istream Constructors

Creates an `basic_istream` object.

```
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

Destroy the `basic_istream` object.

```
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

Class `basic_istream::sentry`

A class for exception safe prefix and suffix operations.

Class basic_istream::sentry Constructor

Prepare for formatted or unformatted input

```
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 `setstate(failbit)` which may throw an exception.

Destructor

Destroys a sentry object.

```
~sentry();
```

Remarks

The destructor has no effects.

sentry::Operator bool

To return the value of the data member `ok_`.

```
operator bool();
```

Remarks

Operator `bool` returns the value of `ok_`

Formatted input functions

Formatted function provide mechanisms for input operations of specific types.

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().

Arithmetic Extractors Operator >>

Extractors that provide formatted arithmetic input operation. Extracts a short integer value and stores it in `n`.

```
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);
```

Remarks

The Arithmetic extractors extract a specific type from the input stream and store it in the address provided

Table 18.1 States and stdio equivalents

state	stdio equivalent
(flags() & basefield) == oct	%o
(flags() & basefield) == hex	%x

Table 18.1 States and stdio equivalents

state	stdio equivalent
(flags() & basefield) != 0	%x
(flags() & basefield) == 0	%i
Otherwise	
signed integral type	%d
unsigned integral type	%u

basic_istream extractor operator >>

Extracts characters or sequences of characters and converts if necessary to numerical data.

```
basic_istream<charT, traits>& operator >>
```

```
basic_istream<charT, traits>& (*pf)
```

```
(basic_istream<charT, traits>&) )
```

Returns pf(*this).

```
basic_istream<charT, traits>& operator >>
```

```
(basic_ios<charT, traits>& (*pf) (basic_ios<charT, traits>&) )
```

Calls pf(*this) then returns *this.

```
basic_istream<charT, traits>& operator >>(char_type *s);
```

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);
```

Extracts a single character and stores it in c if possible otherwise call setstate(failbit).

```
basic_istream<charT, traits>& operator >>(void*& p);
```

Converts a pointer to void and stores it in `p`.

```
basic_istream<charT, traits>& operator >>
(basic_streambuf<char_type, traits>* sb);
```

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.

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.

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;

ifstream in(ioFile);
if(!in.is_open())
{cout << "cannot open file for input"; exit(1);}

char type[20];
```

Formatting Manipulators

Input Streams

```
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:

To provide custom formatted data retrieval.

```
extractor prototype

Basic_istream &operator >>(basic_istream &s, const imanip<T>&)

{      // procedures

    return s;

}
```

Remarks

You may overload the extractor operator to tailor the specific needs of a particular class.

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;
    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;
```

Formatting Manipulators

Input Streams

```
    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
```

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

To obtain the number of bytes read.

```
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.

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;
};
```

Formatting Manipulators

Input Streams

```
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

Overloaded functions to retrieve a `char` or a `char` sequence from an input stream.

```
int_type get();
```

Extracts a character if available and returns that value. Else, calls `setstate(failbit)` and returns `eof()`.

```
basic_istream<charT, traits>& get(char_type& c);
```

Extracts a character and assigns it to `c` if possible else calls `setstate(failbit)`.

```
basic_istream<charT, traits>& get(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.

Formatting Manipulators

Input Streams

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_streampbuf<char_type,
traits>& sb, char_type delim = traits::newline());
```

Extracts characters and assigns them to the `basic_streampbuf` object `sb` if possible or else it calls `setstate(failbit)`. Extraction stops if...

an insertion fails

`end-of-file` is encountered.

an exception is thrown

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

["basic_istream::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>  
  
const int size = 100;  
char buf[size];  
  
int main()  
{  
using namespace std;  
  
    cout << " Enter your name: ";  
    cin.get(buf, size);  
    cout << buf;  
  
    return 0;  
}
```

Result:

```
Enter your name: Metrowerks CodeWarrior <enter>  
Metrowerks CodeWarrior
```

basic_istream::getline

To obtain a delimiter terminated character sequence from an input stream.

```
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.

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";
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

To extract and discard a number of characters.

```
basic_istream<charT, traits> & ignore
(steamsize n = 1, int_type delim = traits::eof());
```

Remarks

The function `ignore()` will extract and discard characters until
A limit is met (the first argument)

Formatting Manipulators

Input Streams

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()`;

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
/* 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 */
```

basic_istream::peek

To view at the next character to be extracted.

```
int_type peek();
```

Remarks

The function `peek()` allows you to look ahead at the next character in a stream to be extracted without extracting it.

If `good()` is false returns `traits::eof()` else returns the value of the next character in the stream.

See Also

Example of `basic_istream::peek()` usage see [“basic_istream::ignore”](#)

basic_istream::read

To obtain a block of binary data from and input stream.

```
basic_istream<charT, traits>& read
(char_type* s, streamsize n);
```

Remarks

The function `read()` will attempt to extract a block of binary data until the following conditions are met.

A limit of `n` number of characters are stored.

`end-of-file` is encountered on the input (in which case `setstate(failbit)` is called).

Return

The `this` pointer is returned.

See Also

[“basic_ostream::write”](#)

Listing 18.8 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)
<< "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

Extracts characters and stores them in an array.

```
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.

Return

The number of characters extracted.

Listing 18.9 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

To replace a previously extracted character.

```
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.

Return

The `this` pointer is returned.

See Also

[“basic_istream::unget”](#)

Listing 18.10 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;
```

Formatting Manipulators

Input Streams

```
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

To replace a previously extracted character.

```
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.

The `this` pointer is returned.

See Also

[“basic_istream::putback”](#), [“basic_istream::ignore”](#)

Listing 18.11 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);
```

```
#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:

Formatting Manipulators

Input Streams

```
char ch;                      to save char
    /* comment will remain */
    read until failure
while((ch = in.get()) != EOF) cout.put(ch);
```

basic_istream::sync

To synchronize input and output

```
int sync();
```

Remarks

This function attempts to make the input source consistent with the stream being extracted.

If `rdbuf() ->pubsync()` returns -1 `setstate(badbit)` is called and `traits::eof` is returned.

Return

If `rdbuf()` is Null returns -1 otherwise returns zero.

Listing 18.12 Example of basic_istream::sync() usage:

The file MW Reference contains:

This function 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())
{
    in.get(str, 10, EOF);
    cout <<str;
}
return 0;
}
```

Result:

This function attempts to make the input source
consistent with the stream being extracted.

--

Metrowerks CodeWarrior "Software at Work"

basic_istream::tellg

To determine the offset of the get pointer in a stream

```
pos_type tellg();
```

Remarks

The function tellg calls rdbuf()->pubseekoff(0, cur, in).

The current offset as a pos_type if successful else returns -1.

See Also

[basic_streambuf::pubseekoff\(\)](#)

Example of basic_istream::tellg() usage see [“basic_istream::seekg”](#)

basic_istream::seekg

To move to a variable position in a stream.

```
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.

The `this` pointer is returned.

See Also

`basic_streambuf::pubseekoff()` and `pubseekpos()`.

Listing 18.13 Example of basic_istream::seekg() usage:

The file MW Reference contains:

```
ABCDEFGHIJKLMNPQRSTUVWXYZ  
#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);}  
  
// note streampos is typedef in iosfwd  
streampos spEnd(5), spStart(5);  
  
in.seekg(spStart);  
streampos aCheck = in.tellg();
```

```
cout << "The offset at the start of the reading in bytes is "
    << aCheck << endl;

char ch;
while(spEnd != spStart+10)
{
    in.get(ch);
    cout << ch;
    spEnd = in.tellg();
}

aCheck = in.tellg();
cout << "\nThe current position's offset in bytes now is "
    << aCheck << endl;
streamoff gSet = 0;
in.seekg(gSet, ios::beg);

aCheck = in.tellg();
cout << "The final position's offset in bytes now is "
    << aCheck << endl;

in.close();
return 0;
}
```

Result:

```
The offset at the start of the reading in bytes is 5
FGHIJKLMNOP
The current position's offset in bytes now is 15
The final position's offset in bytes now is 0
```

Standard basic_istream manipulators

The istream class provides several manipulators for input streams.

Formatting Manipulators

Input Streams

basic_ifstream::ws

To provide inline style formatting.

```
template<class charT, class traits>
basic_istream<charT, traits> &ws
(basic_istream<charT, traits>& is);
```

Remarks

The ws manipulator skips whitespace characters in input.

The this pointer is returned.

Listing 18.14 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())
        cout << ch;
    else break;
}
cout << " | " << endl;

in.close();
return(0);
}
```

Result:

```
Does not skip whitespace
|           a           b     c |
```

```
Does skip whitespace
|abc|
```

basic_iostream Constructor

Constructor

Constructs an and destroy object of the class basic_iostream.

```
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

```
virtual ~basic_iostream();
```

Remarks

Destroys an object of type `basic_iostream`.

Output streams

The include file `<ostream>` includes classes and types that provide output stream mechanisms.

The topics in this section are:

- [“Template class basic_ostream”](#)
- [“basic_ostream Constructor”](#)
- [“Class basic_ostream::sentry Constructor”](#)
- [“Class basic_ostream::sentry”](#)
- [“Formatted output functions”](#)
- [“Common requirements”](#)
- [“Arithmetic Inserter Operator <<”](#)
- [“basic_ostream::operator<<”](#)
- [“Unformatted output functions”](#)
- [“Standard basic_ostream manipulators”](#)

Template class `basic_ostream`

A class for stream output mechanisms.

The `basic_ostream` class provides for output stream mechanisms for output stream classes. The `basic_ostream` class may be used as an independent class, as a base class for the `basic_ofstream` class or a user derived classes.

basic_ostream Constructor

To create and remove from memory `basic_ostream` object for stream output.

```
explicit basic_ostream(basic_streambuf<char_type, traits>*sb);
```

Remarks

The `basic_ostream` constructor constructs and initializes the base class object.

Destructor

```
virtual ~basic_ostream();
```

Remarks

Removes a `basic_ostream` object from memory.

Listing 18.15 Example of `basic_ostream()` 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())
```

Formatting Manipulators

Output streams

```
{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

Class `basic_ostream::sentry`

A class for exception safe prefix and suffix operations.

Class `basic_ostream::sentry` Constructor

Prepare for formatted or unformatted output

```
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 `setstate(failbit)` which may throw an exception.

Destructor

```
~sentry();
```

Remarks

The destructor under normal circumstances will call `os.flush()`.

sentry::Operator bool

To return the value of the data member `ok_`.

```
operator bool();
```

Remarks

Operator `bool` returns the value of `ok_`

Formatted output functions

Formatted output functions provide a manner of inserting for output specific data types.

Common requirements

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 `setstate(failbit)`.

Formatting Manipulators

Output streams

Arithmetic Inserter Operator <<

To provide formatted insertion of types into a stream.

```
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

In most cases *this is returned unless failure in which case `setstate(failbit)` is called.

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

Table 18.2 Output states and stdio equivalents.

Output State	stdio equivalent
Floating Point Numbers	
(flags() & floatfield) == fixed	%f
(flags() & floatfield) == scientific (flags() & uppercase) != 0	%e %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	+ #

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 stdio equivalents.

State	Justification	stdio equivalent
(flags()& adjustfield) == left	left	space padding
(flags() & adjustfield) == internal	Internal	zero padding
Otherwise	right	space padding

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.

basic_ostream::operator<<

```
basic_ostream<charT, traits>& operator<<
(basic_ostream<charT, traits>&
(*pf)(basic_ostream<charT, traits>&)) ;
```

Returns pf(*this).

```
basic_ostream<charT, traits>& operator<<
(basic_ostream<charT, traits>&
(*pf)(basic_ios<charT, traits>&)) ;
```

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)
```

Behaves depending on how the boolalpha flag is set.

```
basic_ostream<charT, traits>& operator<<(void p)
```

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); )
```

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.

Most `inserters` (unless noted otherwise) return the `this` pointer.

Listing 18.16 Example of basic_ostream inserter usage:

```
#include <iostream>
#include <fstream>
#include <cstdlib>

char oFile[81] = "MW Reference";

int main()
{
using namespace std;

    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

To provide specialized output mechanisms for an object.

Overloading inserter prototype

```
basic_ostream &operator<<  
(basic_ostream &stream, const omanip<T>&) {  
    // procedures;  
    return stream;  
}
```

Remarks

You may overload the inserter operator to tailor it to the specific needs of a particular class.

The `this` pointer is returned.

Listing 18.17 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;
}

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
```

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

To return the offset of the put pointer in an output stream.

```
pos_type tellp();
```

Formatting Manipulators

Output streams

Return

If fail() returns -1 else returns rdbuf()->pubseekoff(0, cur, out).

See Also

[“basic_istream::tellg”](#), [“basic_ostream::seekp”](#)

Example of basic_ostream::tellp() usage see[“basic_ostream::seekp”](#)

basic_ostream::seekp

Randomly move to a position in an output stream.

```
basic_ostream<charT, traits>& seekp(pos_type);  
  
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”](#), [“basic_ostream::tellp”](#)

Listing 18.18 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;
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 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
```

```
The ostringstream array is still
Metrowerks CodeWarrior - Software at Work
```

Formatting Manipulators

Output streams

basic_ostream::put

To place a single character in the output stream.

```
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)`.

The `this` pointer is returned.

Listing 18.19 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

To insert a block of binary data into an output stream.

```
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 can be used to write an object by casting that object as a `unsigned char` pointer. If the operation fails calls `setstate(badbit)`.

A reference to `ostream`. (The `this` pointer) is returned.

See Also

[“basic_istream::read”](#)

Listing 18.20 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));
```

Formatting Manipulators

Output streams

```
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

To force the output buffer to release its contents.

```
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 `setstate(badbit)`.

The `this` pointer is returned.

Note that in the “[Example of basic_ostream::flush\(\) usage:](#)” if you comment out the flush that both lines will display simultaneously at the end of the program.

Listing 18.21 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();
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);
```

Formatting Manipulators

Output streams

```
}

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:

```
begin the timer: < immediate display then pause >
begin the timer:
The Object lasted: 3.83 seconds
```

Standard basic_ostream manipulators

The ostream class provides an inline formatting mechanism.

basic_ostream:: endl

To insert a newline and flush the output stream.

```
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.

A reference to basic_ostream. (The this pointer) is returned.

See Also

[“basic_ostream::operator<<”](#)

basic_ostream::ends

To insert a NULL character.

```
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.

A reference to ostream. (The this pointer) is returned.

The ostringstream provides in-core character streams but must be null terminated by the user. The manipulator ends provides a null terminator.

Formatting Manipulators

Output streams

Listing 18.22 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

To flush the stream for output.

```
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.

A reference to `ostream` (the `this` pointer) is returned.

Note in the “[Example of basic_ostream:: flush usage:](#)” comment out the flush and both lines will display simultaneously at the end of the program.

See Also

[“basic_ostream::flush”](#)

Listing 18.23 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)
```

Formatting Manipulators

Output streams

```
    << 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)){};
}
```

Results:

```
begin time the timer:
< short pause >
The Object lasted: 3.78 seconds
```

Standard manipulators

The include file `iomanip` defines a template classes and related functions for input and output manipulation.

Standard Manipulator Instantiations

To create a specific use instance of a template by replacing the parameterized elements with pre-defined types.

resetiosflags

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.

A smanip type, that is an implementation defined type is returned.

See Also

`ios_base::setf()`, `ios_base::unsetf()`

Listing 18.24 Example of `resetiosflags()` usage:

```
#include <iostream>
#include <iomanip>

int main()
{
using namespace std;

    double d = 2933.51;
    long flags;
```

Formatting Manipulators

Standard manipulators

```
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

Set the stream format flags.

Prototypes

```
smanip setiosflags(ios_base::fmtflags mask)
```

Remarks

Use the manipulator `setiosflags()` to set the input and output formatting flags directly in the stream.

A smanip type, that is an implementation defined type is returned.

See Also

`ios_base::setf()`, `ios_base::unsetf()`

For example of `setiosflags()` usage see “[resetiosflags](#)”

:setbase

To set the numeric base of an output.

```
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 18.25 Example of 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
```

Formatting Manipulators

Standard manipulators

```
Octal 304
Reset to Hex c4
Reset basefield setting 196
```

setfill

To specify the characters to used to insert in unused spaces in the output.

```
smanip setfill(int c)
```

Remarks

Use the manipulator `setfill()` directly in the output to fill blank spaces with character `c`.

Returns a `smanip` type, that is an implementation defined type.

See Also

```
basic_ios::fill
```

Listing 18.26 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 *

setprecision

Set and return the current format precision.

```
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.

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 18.27 Example of 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

Formatting Manipulators

Standard manipulators

Precision set: 321.12346

setw

To set the width of the output field.

```
smanip<int> setw(int)
```

Remarks

Use the manipulator `setw()` directly in a stream to set the field size for output.

A pointer to `ostream` is returned.

See Also

```
ios_base::width()
```

Listing 18.28 Example of `setw()` usage:

```
#include <iostream>
#include <iomanip>

int main()
{
using namespace std;

    cout << setw(8)
        << setfill('*')
        << "Hi!" << endl;
    return 0;
}
```

Result:

```
Hi!*****
```

Overloaded Manipulator

To store a function pointer and object type for input.

Overloaded input manipulator for `int` type.

```
istream &imanip_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> mainip_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 18.29 Example of overloaded manipulator usage:

```
#include <iostream>  
#include <cstring>  
#include <cstdlib>  
#include <cctype>  
  
char buffer[80];
```

Formatting Manipulators

Standard manipulators

```
char *Password = "Metrowerks";

struct verify
{
    explicit verify(char* check) : check_(check) {}
    char* check_;
};

char *StrUpr(char * str);
std::istream& operator >> (std::istream& stream, const verify& v);

int main()
{
using namespace std;

    cin >> verify(StrUpr>Password));
    cout << "Log in was Completed ! \n";

    return 0;
}

std::istream& operator >> (std::istream& stream, const verify& v)
{
using namespace std;

    short attempts = 3;

    do {
        cout << "Enter password: ";
        stream >> buffer;

        StrUpr(buffer);
        if (!strcmp(v.check_, buffer)) return stream;
        cout << "\a\aa";
        attempts--;
    } while(attempts > 0);

    cout << "All Tries failed \n";
    exit(1);
    return stream;
}

char *StrUpr(char * str)
{
    char *p = str; // dupe string
```

```
    while(*p) *p++ = static_cast<char>(std::toupper(*p));
    return str;
}
```

Result:

```
Enter password: <codewarrior>
Enter password: <mw>
Enter password: <metrowerks>
Log in was Completed !
```

Formatting Manipulators

Standard manipulators

String Based Streams

This chapter discusses string-based streams in the standard C++ library.

The String Based Stream Library

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 chapter is constructed in the following sub sections and mirrors clause 27.7 of the ISO (the International Organization for Standardization) C++ Standard :

- “[Header `<sstream>`](#)”
- “[Template class `basic_stringbuf`](#).”
- “[Template class `basic_istringstream`](#)”
- “[Class `basic_stringstream`](#)”

Header `<sstream>`

Overview

The header `<sstream>` includes classes and typed that associate stream buffers with string objects for input and output manipulations.

NOTE The class `basic_string` is discussed in previous chapters.

String Based Streams

Template class basic_stringbuf.

Template class basic_stringbuf.

Overview

The template class `basic_stringbuf` is derived from `basic_streambuf` to associate both input and output streams with an object of class `basic_string`.

The other topics in this section are:

- [“basic_stringbuf constructors”](#)
- [“Member functions”](#)
- [“Overridden virtual functions”](#)

The class `basic_stringbuf` is derived from `basic_streambuf` to associate a stream with a `basic_string` object for in-core memory character manipulations.

basic_stringbuf constructors

The `basic_stringbuf` has two constructors to create a string buffer for characters for input/output.

```
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

Member functions

The class `basic_stringbuf` has one member functions:

basic_stringbuf::str

To return or clear the `basic_string` object stored in the buffer.

```
basic_string<char_type> str() const;  
void str(const basic_string<char_type>&s);
```

Remarks

The function `str()` freezes the buffer then returns a `basic_string` object.

String Based Streams

Template class `basic_stringbuf`.

The function `str(const string s)` assigns the value of the `string` 's' to the `stringbuf` object.

The no argument version returns a `basic_string` if successful. The function with an argument has no return.

Listing 19.2 Example of `basic_stringbuf::str()` usage:

```
#include <iostream>
#include <sstream>
#include <cstring>

char CW[] = "Metrowerks CodeWarrior";
char AW[] = " - \"Software at Work\""

int main()
{
using namespace std;

    string buf;
    stringbuf strbuf(buf, ios::in | ios::out);

    int size;
    size = strlen(CW);
    strbuf.sputn(CW, size);
    size = strlen(AW);
    strbuf.sputn(AW, size);
    cout << strbuf.str() << endl;

    // Clear the buffer then fill it with
    // new information and then display it
    string clrBuf = "";
    string ANewLine = "We Listen we Act";

    strbuf.str(clrBuf);
    strbuf.sputn(ANewLine.c_str(), ANewLine.size());

    cout << strbuf.str() << endl;
    return 0;
}
```

Results

Metrowerks CodeWarrior - "Software at Work"
We Listen we Act

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

To show an underflow condition and not increment the get pointer.

```
virtual int_type underflow();
```

Remarks

The function `underflow` overrides the `basic_streambuf` virtual function.

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

To show a failure in a put back operation.

```
virtual int_type pbackfail  
(int_type c = traits::eof());
```

Remarks

The function `pbackfail` overrides the `basic_streambuf` virtual function.

String Based Streams

Template class basic_stringbuf.

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

Consumes the pending characters of an output sequence.

```
virtual int_type overflow  
(int_type c = traits::eof());
```

Remarks

The function `overflow` overrides the `basic_streambuf` virtual function.

The function returns `traits::eof()` for failure or some unspecified result to indicate success.

See Also

`basic_streambuf::overflow()`

basic_stringbuf::seekoff

To return an offset of the current pointer in an input or output streams.

```
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.

A `pos_type` value, which is an invalid stream position is returned.

See Also

`basic_streambuf::seekoff()`

basic_stringbuf::seekpos

To alter an input or output stream position.

```
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.

A `pos_type` value, which is an invalid stream position is returned.

See Also

`basic_streambuf::seekoff()`

Template class basic_istringstream

The template class `basic_istringstream` 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:

- [“basic_istringstream Constructor”](#)
- [“Member functions”](#)

```
namespace std {
```

String Based Streams

Template class basic_istringstream

template

The class `basic_istringstream` uses an object of type `basic_stringbuf` to control the associated storage.

See Also

["Class basic_ostringstream"](#)

["Class basic_stringstream"](#)

basic_istringstream Constructor

The `basic_istringstream` constructors create a `basic_stringstream` object and initialize the `basic_streampbuf` object.

```
explicit basic_istringstream  
    (ios_base::openmode which = ios_base::in);  
  
explicit basic_istringstream  
    (const basic_string<charT> &str,  
     ios_base::openmode which = ios_base::in);
```

Remarks

The `basic_istringstream` 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_istringstream::basic_istringstream()` 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";

istringstream Paragraph(sBuffer, ios::in);
Paragraph >> num;
Paragraph >> flt;
Paragraph >> szArr;

cout << num << " " << flt << " "
<< szArr << endl;

return 0;
}
```

```
Result
3 12.3 line
```

Member functions

The class `basic_istringstream` has two member functions

basic_istringstream::rdbuf

To retrieve a pointer to the stream buffer.

```
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.

A pointer to an object of type `basic_stringbuf` `sb` is returned by the `rdbuf` function.

String Based Streams

Template class basic_istringstream

See Also

```
basic_ostringstream::rdbuf()  
basic_ios::rdbuf()  
basic_stringstream::rdbuf()
```

Listing 19.4 Example of basic_istringstream::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_istringstream::str

To return or assign the `basic_string` object stored in the buffer.

```
basic_string<charT> str() const;  
void str(const basic_string<charT> &s);
```

Remarks

The function `str()` freezes the buffer then returns a `basic_string` object.

The function `str(const string s)` assigns the value of the `string` 's' to the `stringbuf` object.

The no argument version returns a `basic_string` if successful. The function with an argument has no return.

See Also

`basic_streambuf::str()`
`basic_ostringstream.str()`
`basic_stringstream::str()`

Listing 19.5 Example of `basic_istringstream::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"
```

Class `basic_ostringstream`

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:

- `basic_ostringstream` constructors

String Based Streams

Class `basic_ostringstream`

- Member functions

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`.

basic_ostringstream Constructor

The `basic_ostringstream` constructors create a `basic_stringstream` object and initialize the `basic_streambuf` object.

```
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 a 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);}

    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

Member functions

The class `basic_ostringstream` has two member functions:

basic_ostringstream::rdbuf

To retrieve a pointer to the stream buffer.

```
basic_stringbuf<charT, traits>* rdbuf() const;
```

String Based Streams

Class basic_ostreamstream

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.

A pointer to an object of type `basic_stringbuf sb` is returned by the `rdbuf` function.

See Also

`basic_ostream::rdbuf()`
`basic_ios::rdbuf()`
`basic_stringstream::rdbuf()`

Listing 19.7 example of `basic_ostream::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"
    << 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

To return or assign the `basic_string` object stored in the buffer.

```
basic_string<charT> str() const;
void str(const basic_string<charT> &s);
```

Remarks

The function `str()` freezes the buffer then returns a `basic_string` object.

The function `str(const string s)` assigns the value of the `string 's'` to the `stringbuf` object.

The no argument version returns a `basic_string` if successful. The function with an argument has no return.

See Also

`basic_streambuf::str()`, `basic_istringstream.str()`
`basic_stringstream::str()`

Listing 19.8 Example of `basic_ostringstream::str()` usage.

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

int main()
{
using namespace std;

ostringstream 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?

Class `basic_stringstream`

The template class `basic_stringstream` is derived from `basic_iostream` is used to associate input and output streams with an object of class `basic_string`.

The class `basic_stringstream` uses an object of type `basic_stringbuf` to control the associated storage.

See Also

["Template class `basic_istringstream`"](#) ["Class `basic_ostringstream`"](#)

basic_stringstream Constructor

The `basic_stringstream` constructors create a `basic_stringstream` object and initialize the `basic_streambuf` object.

```
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;
```

String Based Streams

Class `basic_stringstream`

```
iost << buf;
iost >> word;
iost >> num;
iost >> real;

cout << word << " "
    << num << " "
    << real << endl;

return 0;
}
```

Result

ABCD 22 33.33

Member functions

The class `basic_stringstream` has two member functions:

basic_stringstream::rdbuf

To retrieve a pointer to the stream buffer.

```
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.

A pointer to an object of type `basic_stringbuf` `sb` is returned by the `rdbuf` function.

See Also

["Template class basic_istringstream"](#)
["Class basic_ostringstream"](#)

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

To return or assign the `basic_string` object stored in the buffer.

```
basic_string<charT> str() const;  
  
void str(const basic_string<charT> &s);
```

Remarks

The function `str()` freezes the buffer then returns a `basic_string` object.

The function `str(const string s)` assigns the value of the `string` 's' to the `stringbuf` object.

String Based Streams

Class basic_stringstream

The no argument version returns a `basic_string` if successful. The function with an argument has no return.

See Also

`basic_streambuf::str()`
`basic_ostringstream.str()`
`basic_istringstream::str()`

Listing 19.11 Example of `basic_stringstream::str()` usage

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

std::string buf = "Metrowerks CodeWarrior - \"Software at Work\" ";
char words[50];

int main()
{
using namespace std;

stringstream iost(buf, ios::in);

cout << iost.str();

return 0;
}
```

Result

Metrowerks CodeWarrior - "Software at Work"

File Based Streams

Association of stream buffers with files for file reading and writing.

The File Based Streams Library

The chapter is constructed in the following sub sections and mirrors clause 27.8 of the ISO (the International Organization for Standardization) C++ Standard :

- [“Header <fstream>”](#)
- [“File Streams Type Defines”](#)
- [“Template class basic_filebuf”](#)
- [“Template class basic_ifstream”](#)
- [“Template class basic_ofstream”](#)
- [“Template class basic_fstream”](#)

Header <fstream>

The header <fstream> defines template classes and types to assist in reading and writing of files.

File Streams Type Defines

- `typedef basic_filebuf<char> filebuf;`
- `typedef basic_filebuf<wchar_t> wfilebuf;`
- `typedef basic_ifstream<char> ifstream;`
- `typedef basic_ifstream<wchar_t> wifstream;`
- `typedef basic_ofstream<char> ofstream;`
- `typedef basic_ofstream<wchar_t> wofstream;`

File Based Streams

Template class basic_filebuf

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.

Template class basic_filebuf

A class to provide for input and output file stream buffering mechanisms.

The prototype is listed below. Other topics in this section are:

- [“basic_filebuf Constructors”](#)
- [“Member functions”](#)
- [“Overridden virtual functions”](#)

The filebuf class is derived from the streambuf class and provides a buffer for file output and or input.

basic_filebuf Constructors

To construct and initialize a filebuf object.

```
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.

Listing 20.1 For example of basic_filebuf::basic_filebuf() usage:

The file MW Reference before operation contains.
Metrowerks CodeWarrior "Software at Work"

```
#include <iostream>
#include <fstream>
#include <cstdio>
#include <cstring>

char inFile[ ] = "MW Reference";
```

```
int main()
{
using namespace std;

FILE *fp = fopen( inFile, "a+" );
filebuf in(fp);
if( !in.is_open() )
    { cout << "could not open file"; exit(1); }
char str[] = "\n\ttrademark";
in.sputn(str, strlen(str));

in.close();
return 0;
}
```

Result:
The file MW Reference now contains:
Metrowerks CodeWarrior "Software at Work"
trademark

Destructor

To remove the `basic_filebuf` object from memory.

```
virtual ~basic_filebuf();
```

Member functions

The class `basic_filebuf` provides several functions for file buffer manipulations.

basic_filebuf::is_open

Test to ensure filebuf stream is open for reading or writing.

```
bool is_open() const
```

File Based Streams

Template class basic_filebuf

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.

See Also

For example of `basic_filebuf::is_open()` usage see `basic_filebuf::basic_filebuf`

basic_filebuf::open

Open a `basic_filebuf` object and associate it with a file.

```
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.

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.

If successful the `this` pointer is returned, if `is_open()` equals true then a null pointer is returned.

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	

Table 20.1 Legal basic_filebuf file opening modes

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"

Listing 20.2 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;
}
```

File Based Streams

Template class basic_filebuf

Result:

The file MW Reference now contains:
Metrowerks CodeWarrior "Software at Work"
registered trademark

basic_filebuf::close

To close a `filebuf` stream without destroying it.

```
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.

The `this` pointer is returned with success otherwise a null pointer.

See Also

For example of `basic_filebuf::close()` usage see `basic_filebuf::open()`

Overridden virtual functions

basic_filebuf::showmanyC

Overrides `basic_streambuf::showmanyC()`.

```
virtual int showmanyC();
```

Remarks

Behaves the same as `basic_streambuf::showmanyC()`.

basic_filebuf::underflow

Overrides `basic_streambuf::underflow()`;

```
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

Overrides `basic_streambuf::pbackfail()`.

```
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

Overrides `basic_streambuf::overflow()`

```
virtual int_type overflow  
(int_type c = traits::eof());
```

File Based Streams

Template class basic_filebuf

Remarks

Behaves the same as `basic_strreambuf<charT, traits>::overflow(c)` except the behavior of consuming characters is performed by conversion.

Returns `traits::eof()` with failure.

basic_filebuf::seekoff

Overrides `basic_streambuf::seekoff()`

```
virtual pos_type seekoff  
(off_type off,  
ios_base::seekdir way,  
ios_base::in | ios_base::out);
```

Remarks

Sets the offset position of the stream as if using the C standard library function `fseek(file, off, whence)`.

Seekoff function returns a newly formed `pos_type` object which contains all information needed to determine the current position if successful. An invalid stream position if it fails.

basic_filebuf::seekpos

Overrides `basic_streambuf::seekpos()`

```
virtual pos_type seekpos  
(pos_type sp,  
ios_base::openmode which,  
ios_base::in | ios_base::out);
```

Remarks

Description undefined in standard at the time of writing.

Seekpos function returns a newly formed `pos_type` object which contains all information needed to determine the current position if successful. An invalid stream position if it fails.

basic_filebuf::setbuf

Overrides `basic_streambuf::setbuf()`

```
virtual basic_streambuf<charT traits>* setbuf  
(char_type* s, streamsize n);
```

Remarks

Description undefined in standard at the time of writing.

basic_filebuf::sync

Overrides `basic_streambuf::sync`

```
virtual int sync();
```

Remarks

Description undefined in standard at the time of writing.

basic_filebuf::imbue

Overrides `basic_streambuf::imbue`

```
virtual void imbue(const locale& loc);
```

File Based Streams

Template class basic_ifstream

Remarks

Description undefined in standard at the time of writing.

Template class basic_ifstream

A class to provide for input file stream mechanisms.

The prototype is listed below. Other topics in this section are:

- [“basic_ifstream Constructor”](#)
- [“Member functions”](#)

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.

The basic_ifstream provides mechanisms specifically for input file streams.

basic_ifstream Constructor

Create a file stream for input.

```
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.

See `basic_ifstream::open` for legal opening modes.

See also

`basic_ifstream::open()` for overloaded form usage.

Listing 20.3 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"

Member functions

The ifstream class has several public member functions for stream manipulations.

File Based Streams

Template class basic_ifstream

basic_ifstream::rdbuf

The `rdbuf()` function retrieves a pointer to a `filebuf` type buffer.

```
basic_filebuf<charT, traits>* rdbuf() const;
```

Remarks

In order to manipulate for random access or use an `ifstream` stream for both input and output you 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 20.4 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);

Out << str;

inOut.close();

return 0;
```

}

Result:

The File now reads:

Metrowerks CodeWarrior "Software at Work"

Registered Trademark

basic_ifstream::is_open

Test for open stream.

```
bool is_open() const
```

Remarks

Use `is_open()` to test that a stream is indeed open and ready for input from the file.

Returns true if file is open.

See Also

For example of `basic_ifstream::is_open()` usage see
`basic_ifstream::basic_ifstream()`

basic_ifstream::open

Open is used to open a file or reopen a file after closing it.

```
void open(const char* s, openmode mode = in);
```

Remarks

The default open mode is `ios::in`, but can be one of several modes. (see below) A stream is opened and prepared for input or output as selected.

There is no return

File Based Streams

Template class `basic_ifstream`

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 20.2 Legal basic_ifstream file opening modes

Opening Modes	stdio equivalent
Input Only	
<code>ios:: in</code>	"r"
<code>ios:: binary ios::in</code>	"rb"
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"

Listing 20.5 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;

    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

Closes the file stream.

```
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.

There is no return.

See Also

For example of `basic_ifstream::close()` usage See
`basic_ifstream::basic_ifstream()`

Template class basic_ofstream

A class to provide for output file stream mechanisms.

The prototype is listed below. Other topics in this section are:

- ["basic_ofstream Constructors"](#)
- ["Member functions"](#)

File Based Streams

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`.

The `basic_ofstream` class provides for mechanisms specific to output file streams.

basic_ofstream Constructors

To create a file stream object for output.

```
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`.

There are only certain valid file opening modes for an `ofstream` object see `basic_ofstream::open()` for a list of valid opening modes.

Listing 20.6 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.

Member functions

The `ofstream` class provides public member functions for output stream manipulation.

`basic_ofstream::rdbuf`

To retrieve a pointer to the stream buffer.

```
basic_filebuf<charT, traits>* rdbuf() const;
```

Remarks

In order to manipulate a stream for random access or other operations you must use the streams base buffer. The member function `rdbuf()` is used to return a pointer to this buffer.

A pointer to `filebuf` type is returned.

Listing 20.7 Example of `basic_ofstream::rdbuf()` usage:

The file MW Reference before the operation contains:
This is an annotated reference that contains a description

File Based Streams

Template class basic_ofstream

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

To test whether the file was opened.

```
bool is_open();
```

Remarks

The `is_open()` 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 `open()` function to open a stream.

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.

Returns `true` if file stream is open and available for output.

See Also

For example of `basic_ofstream::is_open()` usage see
`basic_ofstream::ofstream()`

basic_ofstream::open

To open or re-open a file stream for output.

```
void open(const char* s, openmode mode = out);
```

Remarks

The function `open()` opens a file stream for output. The default mode is `ios::out`, but may be any valid open mode (see below.) If failure occurs `open()` calls `setstate(failbit)` which may throw an exception.

There is no return

File Based Streams

Template class basic_ofstream

Table 20.3 Legal basic_ofstream file opening modes.

Opening Modes	stdio equivalent
Output only	
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"

Listing 20.8 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

The member function closes the stream but does not destroy it.

```
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.

There is no return.

See Also

For example of `basic_ofstream::close()` usage see `basic_ofstream()`.

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:

- ["basic_fstream Constructor"](#)
- ["Member Functions"](#)

File Based Streams

Template class basic_fstream

The template class `basic_fstream` is used for both reading and writing from files.

NOTE	The <code>basic_fstream</code> supports writing to file. It uses a <code>basic_filebuf</code> object to control the sequence. That object is represented here as <code>basic_filebuf sb</code> .
-------------	--

basic_fstream Constructor

To construct an object of `basic_ifstream` for input and output operations.

```
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 20.9 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";  
  
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
```

Member Functions

The fstream class provided public member functions for input and output stream manipulations.

basic_fstream::rdbuf

The `rdbuf()` function retrieves a pointer to a `filebuf` type buffer.

```
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.

A pointer to type `basic_filebuf` is returned.

File Based Streams

Template class basic_fstream

Listing 20.10 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

basic_fstream::is_open

Test to ensure `basic_fstream` file is open and available for reading or writing.

```
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.

See Also

For an example, see [“Example of basic_fstream::basic_fstream\(\) usage”](#).

basic_fstream::open

To open or re-open a file stream for input or output.

```
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.

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.

There is no return value.

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	

File Based Streams

Template class basic_fstream

Table 20.4 Legal file opening modes

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"

See Also

For an example, see [“Example of basic_fstream::rdbuf\(\) usage”](#).

basic_fstream::close

The member function closes the stream but does not destroy it.

```
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.

There is no return value.

See Also

For an example, see [“Example of basic_fstream:: basic_fstream\(\) usage”](#).

File Based Streams

Template class basic_fstream

C Library Files

The header <cstdio> contains the C++ implementation of the Standard C Headers.

The C Library Files

The chapter is constructed in the following sub sections and mirrors clause 27.9 of the ISO (the International Organization for Standardization) C++ Standard :

Table 21.1 <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		

Table 21.2 <cstdio> Types

Types:		
FILE	fpos_t	size_t

Table 21.3 <cstdio> Functions

Functions:		
clearerr	fclose	feof

Table 21.3 <cstdio> Functions

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	

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.

The Strstream Class Library (Annex D)

The chapter is constructed in the following sub sections and mirrors annex D of the ISO (the International Organization for Standardization) C++ Standard :

- [“Strstreambuf Class,”](#) a base class for strstream classes
 - [“Strstreambuf constructors and Destructors”](#)
 - [“Strstreambuf Public Member Functions”](#)
 - [“Protected Virtual Member Functions”](#)
- [“Istrstream Class,”](#) a strstream class for input
 - [“Constructors and Destructor”](#)
 - [“Public Member Functions”](#)
- [“Ostrstream Class,”](#) a strstream class for output
 - [“Constructors and Destructor”](#)
 - [“Public Member Functions”](#)
- [“Strstream Class,”](#) a class for input and output
 - [“Constructors and Destructor”](#)
 - [“Public Member Functions”](#)

Header `<strstream>`

The include file strstream includes three classes, for in memory character array based stream input and output.

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](#),” freezes the buffer
- “[pcount](#),” determines the buffer size
- “[str](#),” returns a string
- “[setbuf](#),” a virtual function to set the buffer
- “[seekoff](#),” a virtual function for stream offset
- “[seekpos](#),” a virtual function for stream position
- “[underflow](#),” a virtual function for input error
- “[pbackfail](#),” a virtual function for put back error
- “[overflow](#),” a virtual function for output error

NOTE

The template class `streambuf` is an abstract class for deriving various stream buffers whose objects control input and output sequences.

Strstreambuf constructors and Destructors

Special constructors and destructors are included for the `strstreambuf` class.

Constructors

Constructs an object of type `streambuf`.

```
explicit strstreambuf(streamsize alsiz_arg = 0);  
  
strstreambuf(void* (*palloc_arg)(size_t),  
void (*pfree_arg)(void*));
```

Dynamic constructors

```
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);
```

Character array constructors

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

To destroy a strstreambuf object.

```
virtual ~strstreambuf();
```

Remarks

Removes the object from memory.

Strstreambuf Public Member Functions

The public member functions allow access to member functions from derived classes.

freeze

To freeze the allocation of strstreambuf.

```
void    freeze(bool freezefl = true);
```

Remarks

The function freeze() stops allocation if the strstreambuf object is using dynamic allocation and prevents the destructor from freeing the allocation. The function freeze(0) when used with zero as an argument releases the freeze to allow for destruction.

There is no return

Listing 22.1 Example of strstreambuf::freeze() usage:

```
#include <iostream>
#include <strstream>
#include <string.h>

const int size = 100;

int main()
{
    // 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

To determine the effective length of the buffer,

```
int      pcount() const;
```

Remarks

The function pcount() is used to determine the offset of the next character position from the beginning of the buffer.

A null terminated character array is returned.

For an example of strstreambuf::pcount() usage refer to strstreambuf::freeze

Strstream

Strstreambuf Class

str

To return the char array stored in the buffer.

```
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.

A null terminated character array is returned.

Listing 22.2 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());

    // 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 ostrstream array is now\n" << buf;

    return 0;
}
```

Result:

The original array was
Metrowerks CodeWarrior - Software at Work

The strstreambuf array is
Metrowerks

The ostrstream 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

To set a buffer for stream input and output sequences.

```
virtual streambuf* setbuf(char* s, streamsize n);
```

Remarks

The function `setbuf()` is overridden in `strstream` classes.

The `this` pointer is returned.

Strstream

Strstreambuf Class

seekoff

To return an offset of the current pointer in an input or output streams.

```
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.

A `pos_type` value, which is an invalid stream position is returned.

seekpos

To alter an input or output stream position.

```
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.

A `pos_type` value, which is an invalid stream position is returned.

underflow

To show an underflow condition and not increment the get pointer.

```
vvirtual int_type underflow();
```

Remarks

The virtual function `underflow()` is called when a character is not available for input.

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.

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.

pbackfail

To show a failure in a put back operation.

```
virtual int_type pbackfail(int_type c = EOF);
```

Remarks

The resulting conditions are the same as the function `underflow()`.

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

Consumes the pending characters of an output sequence.

```
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.

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,](#)” to create and remove an `istrstream` object
- “[rdbuf,](#)” to access the buffer
- “[str,](#)” returns the buffer

Constructors and Destructor

The `istrstream` class has an overloaded constructor.

Constructors

Create an array based stream for input

```
explicit istrstream(const char* s);  
explicit istrstream(char* s);  
istrstream(const char* s, streamsize n);  
istrstream(char* s, streamsize n);
```

Remarks

The `istrstream` 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 22.3 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;
```

Strstream

Istrstream Class

```
    return 0;  
}
```

Result:

```
double is 3.21  
string is array  
int is 321
```

Destructor

To destroy an `istrstream` object.

```
virtual ~istrstream();
```

Remarks

The `istrstream` destructor removes the `istrstream` object from memory.

Public Member Functions

There are two public member functions.

rdbuf

Returns a pointer to the `strstreambuf`

```
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.

Returns a pointer to `strstreambuf`.

For an example of `istrstream::rdbuf()` usage refer to `strstreambuf::str()`

str

Returns a pointer to the stored array.

```
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.

A null terminated char array is returned.

Listing 22.4 Example of istrstream::str() usage.

```
#include <iostream>
#include <strstream>

const int size = 100;
char buf[size] = "Metrowerks CodeWarrior - Software at Work";

int main()
{
    istrstream 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”](#)
- [“freeze”](#)
- [“pcount”](#)

Strstream

Ostrstream Class

- “[rdbuf](#)”
- “[str](#)”

Constructors and Destructor

The `ostrstream` class has an overloaded constructor.

Constructors

Creates a stream and associates it with a char array for output.

```
ostrstream();  
  
ostrstream(char* s, int n,  
ios_base::openmode mode = ios_base::out);
```

Remarks

The `ostrstream` array is overloaded for association a pre allocated array or for dynamic allocation.

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 22.5 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

Destroys an ostrstream object.

```
virtual ~ostrstream();
```

Remarks

A ostrstream destructor removes the ostrstream object from memory.

Public Member Functions

The `ostrstream` class has four public member functions.

freeze

Freezes the dynamic allocation or destruction of a buffer.

```
void freeze(bool freezefl = true);
```

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.

Returns a pointer to `strstreambuf`.

Listing 22.6 Example of ostrstream freeze() usage.

```
#include <iostream>
#include <strstream>
```

Strstream

Ostrstream Class

```
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;
}
```

Result:

```
the size of the array so far is 14 characters
he final size of the array is 24 characters
Metowerks 1234 Software
```

pcount

Determines the number of bytes offset of the current stream position to the beginning of the array.

```
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.

Returns an int_type that is the length of the array.

Listing 22.7 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
Metowerks 1234
```

rdbuf

To retrieve a pointer to the streams buffer.

```
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.

Returns a pointer to strstreambuf.

For an example of ostrstream rdbuf() usage refer to streambuf::pubseekoff()

Strstream

Strstream Class

str

Returns a pointer to a character array.

```
char* str();
```

Remarks

The function str() freezes any dynamic allocation.

Returns a null terminated character array.

For an example of ostrstream str() usage refer to 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”](#)
- [“freeze”](#)
- [“pcount”](#)
- [“rdbuf”](#)
- [“str”](#)

Strstream Types

The `strstream` class type defines a `char_type`, `int_type`, `pos_type` and `off_type`, for stream positioning and storage.

Constructors and Destructor

Specialized constructors and destructors are provided.

Constructors

Creates a stream and associates it with a char array for input and output.

```
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

Destroys a `strstream` object.

```
virtual ~strstream();
```

Remarks

Removes the `strstream` object from memory.

Public Member Functions

The class `strstream` has four public member functions.

freeze

Freezes the dynamic allocation or destruction of a buffer.

```
void freeze(bool freezefl = true);
```

Remarks

The function `freeze` stops dynamic allocation of a buffer.

Strstream*Strstream Class*

pcount

Determines the number of bytes offset of the current stream position to the beginning of the array.

```
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.

Returns an int_type that is the length of the array.

rdbuf

Retrieves a pointer to the streams buffer

```
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.

Returns a pointer to strstreambuf.

str

Returns a pointer to a character array.

```
char* str();
```

Remarks

The function str() freezes any dynamic allocation.

Returns a null terminated character array.

Strstream

Strstream Class

Msl_mutex.h

This chapter is a reference guide to the mutex support library.

The Msl_mutex Support Library

The mutex “mutual exclusion” support library provides the mechanism for implementing thread-safe applications.

It is important to remember that <mutex.h> does not attempt to be a complete thread library. It merely implements just enough for the C++ library’s implementation. Programs with needs greater than this should use the facilities of the native operating system.

The mutex support library consists of the following classes:

- [“Mutex.”](#) a class for implementing a basic mutual exclusion area.
- [“Mutex_lock.”](#) an exception safe execution of the mutex object.

Header <msl_mutex.h>

The header mutex.h is used for mutual exclusion to provide thread-safe applications.

Mutex

The mutex class provides for the basic mutual exclusion mechanism.

Listing 23.1 Class Mutex Synopsis

```
namespace Metrowerks{
class mutex
{
public:
    mutex();
    ~mutex();
```

```
void lock();
void unlock();

private:
    mutex(const mutex&);           // Not defined
    mutex& operator=(const mutex&); // Not defined
};
```

See Also

For example of using the mutex class see:[“Example of mutex_lock.”](#)

Constructors and Assignment Operator

Initialize the mutex object.

```
mutex ();
private:
    mutex(const mutex&);
```

Remarks

A default and a copy constructor are defined.

The copy constructor is declared private and not defined to prevent the `mutex` object from being copied.

Operator=

The assignment operator for the Mutex Class.

```
private:
    mutex& operator=(const mutex&);
```

Remarks

The assignment operator is declared private and not defined to prevent the `mutex` object from being copied.

Destructor

Used for implicit mutex destruction.

```
~mutex () ;
```

Remarks

Destroys the `mutex` object.

Public Member Functions

Public members that provide for mutual exclusion.

Lock

Locks the object into exclusion.

```
void lock () ;
```

Remarks

If locked the object is prevented from being used.

See Also

For example of using `mutex::lock()` see: "[Example of mutex lock.](#)"

Unlock

Unlocks the exclusion.

```
void unlock () ;
```

Remarks

If unlocked the object is allowed to be reused.

Mutex_lock

The class mutex_lock is used so that a mutex object won't accidentally be left locked after an exception is thrown.

Listing 23.2 Class mutex_lock synopsis

```
class mutex_lock {  
public:  
    explicit mutex_lock(mutex& m);  
    ~mutex_lock() ;  
private:  
    mutex& m_;  
    mutex_lock(const mutex_lock&);  
    mutex_lock& operator=(const mutex_lock&);  
};
```

Constructors and Assignment Operator

Initialize the mutex_lock object.

```
inline explicit  
  
mutex_lock(mutex& m) : m_(m) m_.lock();}  
  
private:  
  
    mutex_lock(const mutex_lock&);
```

Remarks

The copy constructor is declared private and not defined to prevent the mutex_lock object from being copied.

Operator=

The assignment operator for mutex_lock.

```
mutex_lock& operator=(const mutex_lock&);
```

Remarks

The assignment operator is declared private and not defined to prevent the `mutex_lock` object from being copied.

Destructor

Destroys the `mutex_lock` object.

```
mutex_lock() {m_.unlock();}
```

Remarks

Destroys the `mutex_lock` object.

Listing 23.3 Example of mutex_lock

```
class A
{
public:
    void foo();
private:
    mutex mut_;
};

void A::foo()
{
    mut_.lock();
    // do some stuff
    mut_.unlock();    // not exception safe!
}
```

If "do some stuff" should throw an exception then `mut_` will remain locked with no good way to unlock it. So you should instead use:

```
void A::foo()
{
    mutex_lock lock(mut_);
    // do some stuff
}
```

Msl_mutex.h

Header <msl_mutex.h>

Bitvector Class Library

The header <bitvector> defines a template class, and facilities for representing and manipulating complex numbers.

The bitvector Class Library

The MSL C++ Library includes a bitvector class for manipulation of dynamic arrays of bool elements.

This class has the functionality of `std::vector<bool>`. The class has been completely rewritten and optimized to move around a word of bits at a time. Several of the `std::algorithms` can now be used with bitvector iterators and still be optimized. Namely, `copy`, `copybackward`, `fill_n`, `fill`, `equal` are all optimized to work with bitvector's iterators. `vector<bool>` is now implemented in terms of `Metrowerks::bitvector`.

As in past releases you can turn off the `vector<bool>` specialization `_MSL_NO_VECTOR_BOOL`. However, even if you turn off the `vector<bool>` specialization, `Metrowerks::bitvector` is still available in the extension header <bitvector>.

The bitvector class consists of

- [“Bitvector types”](#)
- [“Constructors, Destructors and Assignment Operators”](#)
- [“Allocators”](#)
- [“Member Functions”](#)
- [“Operators”](#)

The bitvecor behaves in the same manner for bool types as a vector but in a much more effecient manner.

Listing 24.1 Class bitvector synopsis

```
namespace Metrowerks{
```

Bitvector Class Library

The bitvector Class Library

```
template <class Allocator>
class bitvector<Allocator = std::allocator<bool> >
{
public:
//  types:
    typedef bool           const_reference;
    typedef typename Allocator::size_type   size_type;
    typedef typename Allocator::difference_type difference_type;
    typedef bool           value_type;
    typedef Allocator      allocator_type;
    typedef typename Allocator<unsigned int>::pointer     pointer;
    typedef typename Allocator<unsigned int>::const_pointer const_pointer

    class iterator;          // random access
    class const_iterator;    // random access
    typedef std::reverse_iterator<iterator>
        reverse_iterator;
    typedef std::reverse_iterator<const_iterator>
        const_reverse_iterator;

    class reference
    {
public:
        typedef bitvector container_type;
        ~reference();
        operator bool() const;
        reference& operator=(const bool x);
        reference& operator=(const reference& x);
        void flip();           // flips the bit
    };

//  construct/copy/destroy:
    explicit bitvector(const Allocator& a = Allocator());
    explicit bitvector(size_type n, bool value = false,
                      const Allocator& a = Allocator());
    template <class InputIterator>
    bitvector(InputIterator first, InputIterator last,
              const Allocator& a = Allocator());
    bitvector(const bitvector& x);
    bitvector& operator=(const bitvector& x);
    ~bitvector();

    allocator_type get_allocator() const;
```

```
size_type max_size() const;

size_type size() const;
bool empty() const;
size_type capacity() const;
void reserve(size_type n);

void assign(size_type n, bool x);
template <class InputIterator>
    void assign(InputIterator first, InputIterator last);

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;

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

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

void resize(size_type sz, bool c = false);

reference front();
const_reference front() const;
reference back();
const_reference back() const;

void push_back(bool x);
void pop_back();

iterator insert(iterator position, bool x);
void insert(iterator position, size_type n, bool x);
template <class InputIterator>
void insert(iterator position, InputIterator first,
            InputIterator last);

iterator erase(iterator position);
iterator erase(iterator first, iterator last);
```

```
void clear();
void swap(bitvector&);
void flip();
bool invariants() const;
};

template <class A> bool operator==
    (const bitvector<A>& x, const bitvector<A>& y);
template <class A> bool operator<
    (const bitvector<A>& x, const bitvector<A>& y);
template <class A> bool operator!=
    (const bitvector<A>& x, const bitvector<A>& y);
template <class A> bool operator>
    (const bitvector<A>& x, const bitvector<A>& y);
template <class A> bool operator>=
    (const bitvector<A>& x, const bitvector<A>& y);
template <class A> bool operator<=
    (const bitvector<A>& x, const bitvector<A>& y);
template <class A> void swap(bitvector<A>& x, bitvector<A>& y);
}
```

Bitvector types

The bitvector class provides several predefined types for bool dynamic arrays.

Listing 24.2 Class bitvector predefined types

```
bool const_reference;
typename Allocator::size_type size_type;
typename Allocator::difference_type difference_type;
bool value_type;
Allocator allocator_type;
typename Allocator<unsigned int>::pointer pointer;
typename Allocator<unsigned int>::const_pointer const_pointer
```

iterators

Iterators for random access and reverse iteration.

```
class iterator;  
  
class const_iterator;  
  
random access  
  
typedef std::reverse_iterator<iterator> reverse_iterator;  
  
typedef std::reverse_iterator<const_iterator>  
  
const_reverse_iterator;
```

reference class members

The bitvector includes the reference class. Included in this are one type, and member functions and operators.

Type container_type

```
typedef bitvector container_type;
```

Operator =

Assignment operator.

```
reference& operator=(const bool x);  
  
reference& operator=(const reference& x);
```

Destructor

Destroys the reference object.

```
~reference();
```

Operator ()

Grouping operator

```
operator bool() const;
```

flip

Flips the bit.

```
void flip();
```

Constructors, Destructors and Assignment Operators

Constructors, destructors and assignment operators are provided for the bitvector class library. To create destroy and assign bitvector objects.

constructors

Creates a bitvecotr object.

```
explicit bitvector(const Allocator& a = Allocator());
```

```
explicit bitvector  
(size_type n, bool value = false,  
const Allocator& a = Allocator());
```

```
template <class InputIterator> bitvector  
(InputIterator first, InputIterator last,  
const Allocator& a = Allocator());
```

The copy constructor

```
bitvector(const bitvector& x);  
bitvector& operator=(const bitvector& x);
```

The assignment operator

Destructor

Destroys the bitvector object.

```
~bitvector();
```

Allocators

Allocator functions are provided in the bitvector class

get_allocator

Gets an allocator.

```
allocator_type get_allocator() const;
```

Remarks

Behaves for bit vectors in the same way as its `vector<bool>` counterpart.

max_size

Sets the maximum size for the vector

```
size_type max_size() const;
```

Remarks

Behaves in the same manner for bool types as the `vector<bool>` counterpart but in a much more efficient manner.

Member Functions

The bitvector class provides public member functions for bitvector manipulations.

size

Determines the size of the bitvector.

```
size_type size() const;
```

Remarks

Behaves in the same manner for bool types as the `vector<bool>` counterpart but in a much more efficient manner.

empty

Remove all bitvecotr elements

```
bool empty() const;
```

Remarks

Behaves for bit vectors in the same way as its `vector<bool>` counterpart.

capacity

Determines the capacity of the bitvector.

```
size_type capacity() const;
```

Remarks

Behaves in the same manner for bool types as the `vector<bool>` counterpart but in a much more efficient manner.

reserve

Reserves space in the bitvector.

```
void reserve(size_type n);
```

Remarks

Behaves in the same manner for bool types as the `vector<bool>` counterpart but in a much more effecient manner.

assign

Assigns a bool value of size or another bitvector.

```
void assign(size_type n, bool x);  
  
template <class InputIterator> void assign  
(InputIterator first, InputIterator last);
```

Remarks

Behaves in the same manner for bool types as the `vector<bool>` counterpart but in a much more effecient manner.

begin

The beginning element of the bitvector.

```
iterator begin();  
  
const_iterator begin() const;
```

Remarks

Behaves in the same manner for bool types as the `vector<bool>` counterpart but in a much more effecient manner.

end

The end element of a bitvector

```
iterator end();  
const_iterator end() const;
```

Remarks

Behaves in the same manner for bool types as the vector<bool> counterpart but in a much more effecient manner.

rbegin

The reverse beginning element of a bitvector.

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

Remarks

Behaves in the same manner for bool types as the vector<bool> counterpart but in a much more effecient manner.

rend

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

Remarks

Behaves in the same manner for bool types as the vector<bool> counterpart but in a much more effecient manner.

operator[]

The element operator to use a bitvector like an array.

```
reference operator[](size_type n);  
const_reference operator[](size_type n) const;
```

Remarks

Behaves in the same manner for bool types as the vector<bool> counterpart but in a much more effecient manner.

at

Sets the position of the bitvector to an element.

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

Remarks

Behaves in the same manner for bool types as the vector<bool> counterpart but in a much more effecient manner.

resize

Resizes a bitvector object.

```
void resize(size_type sz, bool c = false);
```

Remarks

Behaves in the same manner for bool types as the vector<bool> counterpart but in a much more effecient manner.

front

Gets the front element.

```
reference front();  
const_reference front() const;
```

Remarks

Behaves in the same manner for bool types as the vector<bool> counterpart but in a much more effecient manner.

back

Gets the back element.

```
reference back();  
const_reference back() const;
```

Remarks

Behaves in the same manner for bool types as the vector<bool> counterpart but in a much more effecient manner.

push_back

Push an element off the back of the bitvector.

```
void push_back(bool x);
```

Remarks

Behaves in the same manner for bool types as the vector<bool> counterpart but in a much more effecient manner.

pop_back

Pops the element off of the back of the bitvector.

```
void pop_back();
```

Remarks

Behaves for bit vectors in the same way as its `vector<bool>` counterpart.

insert

Inserts an element into the bit vector at a position.

```
iterator insert(iterator position, bool x);  
  
void insert (iterator position, size_type n, bool x);  
  
template <class InputIterator> void insert  
(iterator position, InputIterator first, InputIterator last);
```

Remarks

Behaves in the same manner for bool types as the `vector<bool>` counterpart but in a much more effecient manner.

erase

Erases an bitvector from a start to an end position.

```
iterator erase(iterator position);  
  
iterator erase(iterator first, iterator last)
```

Remarks

Behaves in the same manner for bool types as the `vector<bool>` counterpart but in a much more effecient manner.

clear

Clears a bitvector of all elements.

```
void clear();
```

Remarks

Behaves in the same manner for bool types as the vector<bool> counterpart but in a much more effecient manner.

swap

The swap function for bitvector types.

```
template <class A> void swap(bitvector<A>& x, bitvector<A>& y);  
void swap(bitvector&);
```

Remarks

Behaves in the same manner for bool types as the vector<bool> counterpart but in a much more effecient manner.

flip

Flips the bitvector bits.

```
void flip();
```

Remarks

Behaves in the same manner for bool types as the vector<bool> counterpart but in a much more effecient manner.

invariants

An invariant function for bool vectors.

```
bool invariants() const;
```

Remarks

Behaves in the same manner for bool types as the vector<bool> counterpart but in a much more effecient manner.

Operators

The bitvector library provides overloaded operators for bool vector manipulations.

Operator ==

The equal to operator for bitvector types.

```
template <class A> bool operator==  
(const bitvector<A>& x, const bitvector<A>& y);
```

Remarks

Behaves in the same manner for bool types as the vector<bool> counterpart but in a much more effecient manner.

Operator <

The less than operator for bitvector types.

```
template <class A> bool operator<  
(const bitvector<A>& x, const bitvector<A>& y);
```

Remarks

Behaves in the same manner for bool types as the vector<bool> counterpart but in a much more effecient manner.

Operator !=

The not-equal operator for bitvector types.

```
template <class A> bool operator!=  
  (const bitvector<A>& x, const bitvector<A>& y);
```

Remarks

Behaves in the same manner for bool types as the vector<bool> counterpart but in a much more effecient manner.

Operator >

The greater than operator for bitvector types.

```
template <class A> bool operator>  
  (const bitvector<A>& x, const bitvector<A>& y);
```

Remarks

Behaves in the same manner for bool types as the vector<bool> counterpart but in a much more effecient manner.

Operator >=

The greater than or equal to operator for bitvector types

```
template <class A> bool operator>=  
  (const bitvector<A>& x, const bitvector<A>& y);
```

Remarks

Behaves in the same manner for bool types as the vector<bool> counterpart but in a much more efficient manner.

Operator <=

The less than or equal to operator for bitvector types.

```
template <class A> bool operator<=
    (const bitvector<A>& x, const bitvector<A>& y);
```

Remarks

Behaves in the same manner for bool types as the vector<bool> counterpart but in a much more efficient manner.

Bitvector Class Library

The bitvector Class Library

MSL_Utility

This chapter is a reference guide to the General utility support in the Metrowerks standard libraries.

The MSL Utilities Library

This chapter consists of utilities for support of non standard headers.

- [“Basic Compile-Time Transformations”](#)
- [“Type Query”](#)
- [“CV Query”](#)
- [“Type Classification”](#)
- [“POD classification”](#)
- [“Miscellaneous”](#)

The <msl_utility> Header

The purpose of this header is to offer a collection of non-standard utilities collected under the namespace Metrowerks. These utilities are of a fundamental nature, and are typically used in other utilities, rather than top level code. Example usage assumes that a using declaration or directive has been previously issued.

NOTE

This header is non-standard. The classes herein are offered as extensions to the C++ standard. They are marked as such by the namespace Metrowerks.

Concepts and ideas co-developed on Boost

<http://www.boost.org/>

Basic Compile-Time Transformations

A collection of templated strut types which can be used for simple compile-time transformations of types.

remove_const

Will remove the top level const (if present) from a type.

```
typedef typename  
remove_const<T>::type non_const_type;
```

Remarks

The resulting “non_const_type” will be the same as the input type T, except that if T is const qualified, that constant qualification will be removed.

Listing 25.1 Example of remove_const

```
typedef typename remove_const <const int>::type Int;  
Int has type int.
```

remove_volatile

Will remove the top level volatile (if present) from a type.

```
typedef typename  
remove_volatile<T>::type non_volatile_type;
```

Remarks

The resulting “non_volatile_type” will be the same as the input type T, except that if T is volatile qualified, that volatile qualification will be removed.

Listing 25.2 Example of remove_volatile

```
typedef typename remove_volatile <volatile int>::type Int;  
Int has type int.
```

remove_cv

Will remove the top level qualifiers (const and/or volatile, if present) from a type.

```
typedef typename  
remove_cv<T>::type non_qualified_type;
```

Remarks

The resulting “non_qualified_type” will be the same as the input type T, except that if T is cv qualified, the qualifiers will be removed.

Listing 25.3 Example of remove_cv

```
typedef typename remove_cv <const int>::type Int;  
Int has type int.
```

remove_pointer

If given a pointer, returns the type being pointed to. If given a non-pointer type, simply returns the input.

```
typedef typename  
remove_pointer<T>::type pointed_to_type;
```

Listing 25.4 Example of remove_pointer

```
typedef typename  
remove_pointer<const int*volatile*const>::type IntPtr;  
typedef typename remove_pointer<IntPtr>::type Int;
```

MSL_Utility

The <msl_utility> Header

IntPtr will have type type const int*volatile. Int will have the type const int.

remove_reference

If given a reference, returns the type being referenced. If given a non-reference, simply returns the input.

```
typedef typename  
remove_reference<T>::type referenced_type;
```

Listing 25.5 Example of remove_reference

```
typedef typename remove_reference<int&>::type Int;  
typedef typename remove_reference<const int&>::type ConstInt;  
Int has the type int, and ConstInt has the type const int.
```

remove_bounds

If given an array type, will return the type of an element in the array. If given a non-array type, simply returns the input.

```
typedef typename remove_bounds<T>::type Element;
```

Listing 25.6 Example of remove_bounds

```
typedef int IntArray[4];  
typedef typename remove_bounds<IntArray>::type Int;  
Int has the type int.
```

remove_all

This transformation will recursively remove cv qualifiers, pointers, references and array bounds until the type is a fundamental type, enum, union, class or member pointer.

```
typedef typename remove_all<T>::type fundamental_type;
```

Listing 25.7 Example of remove_all

```
typedef const int** Array[4];
typedef typename remove_all<Array*&>::type Int;
Int has the type int.
```

Type Query

The following structs perform basic queries on one or more types and return a bool value.

is_same

This struct can be used to tell if two types are the same type or not.

```
bool b = is_same<T, U>::value;
```

Listing 25.8 Example of is_same

```
bool b = is_same<const int, int>::value;
The resulting value is false. int and const int are two distinct types.
```

CV Query

is_const

Returns true if type has a top level const qualifier, else false.

```
bool b = is_const<T>::value;
```

Listing 25.9 Example of is_const

```
bool b = is_const<const int>::value;  
The resulting value is true.
```

is_volatile

Returns true if type has a top level volatile qualifier, else false.

```
bool b = is_volatile<T>::value;
```

Listing 25.10 Example of is_volatile

```
bool b = is_volatile<const int>::value;  
The resulting value is false.
```

Type Classification

The following structs implement classification as defined by section 3.9 in the C++ standard. All types can be classified into one of ten basic categories:

1. integral
2. floating
3. void
4. pointer

5. member pointer
6. reference
7. array
8. enum
9. union
10. class

Top level cv qualifications do not affect type classification. For example, both const int and int are considered to be of integral type.

```
bool b = is_XXX<T>::value;  
where XXX    is one of the ten basic categories.
```

1. is_integral
2. is_floating
3. is_void
4. is_pointer
5. is_member_pointer
6. is_reference
7. is_array
8. is_enum
9. is_union
10. is_class

Remarks

Now MyEnum will be correctly classified as an enum instead of a class (via the is_enum struct). You do not need to worry about providing this specialization unless you are explicitly using the is_enum query and wanting your enumeration to answer to it correctly.

There are also five “super” categories that are made up of combinations of the ten basic categories:

1. is_arithmetic - is_integral or is_floating
2. is_fundamental - is_arithmetic or is_void

-
3. `is_scalar` - `is_arithmetic` or `is_pointer` or `is_member_pointer` or `is_enum`
 4. `is_compound` - not `is_fundamental`
 5. `is_object` - anything but a void or reference type

The classifications: `is_enum` and `is_union` do not currently work automatically. Enumerations and unions will be mistakenly classified as class type. This can be corrected on a case by case basis by specializing `is_enum_imp` or `is_union_imp`. These specializations are in the `Metrowerks::details` namespace.

`is_extension` is also provided for those types that we provide as an extension to the C++ standard. `is_extension<T>::value` will be false for all types except for long long and unsigned long long.

`has_extension` is a modified form of `is_extension` that answers to true if a type either is an extension or contains an extension.

Listing 25.11 Example of `is_integral`

```
bool b = is_integral<volatile int>::value;
The value of b is true.
```

Listing 25.12 Example of `Metrowerks::details` namespace

```
enum MyEnum {zero, one, two};

template <>
struct Metrowerks::details::is_enum_imp<MyEnum>
{
    static const bool value = true;
}
```

Listing 25.13 Example of `is_extension` and `has_extension`

```
is_extension<long long*&>::value;      // false
has_extension<long long*&>::value;      // true
```

is_signed / is_unsigned

These structs only work on arithmetic types. The type must be constructable by an int and be less-than comparable.

Remarks

In the “[Example of is_signed and is_unsigned use](#)” bool tests as unsigned

Listing 25.14 Example of is_signed and is_unsigned use

```
bool b1 = is_signed<char>::value;
bool b2 = is_unsigned<char>::value;
```

POD classification

Four structs classify types as to whether or not they have trivial special members as defined in section 12 of the C++ standard:

- has_trivial_default_ctor
- has_trivial_copy_ctor
- has_trivial_assignment
- has_trivial_dtor

This library will answer correctly for non-class types. But user defined class types will always answer false to any of these queries. If you create a class with trivial special members, and you want that class to be able to take advantage of any optimizations that might arise from the assumption of trivial special members, you can specialize these structs:

Note that in the “[Example of specialized structs](#)” these specializations need not worry about cv qualifications. The higher level has_trival_XXX structs do that for you.

Finally there is an is_POD struct that will answer true if a type answers true on all four of the above queries.

Listing 25.15 Example of specialized structs

```
template <>

struct Metrowerks::details::class_has_trivial_default_ctor<MyClass>
{ static const bool value = true; };

template <>
struct Metrowerks::details::class_has_trivial_copy_ctor<MyClass>
{ static const bool value = true; };

template <>
```

MSL_Utility

The <msl_utility> Header

```
struct Metrowerks::details::class_has_trivial_assignment<MyClass>
    {static const bool value = true;};

template <>
struct Metrowerks::details::class_has_trivial_dtor<MyClass>
    {static const bool value = true;};
```

Miscellaneous

Miscellaneous utility functions are included in the MSL Utilities library.

compile_assert

This is a compile time assert. This is a very basic version of this idea. Can be used to test assumptions at compile time.

Listing 25.16 Example of compile_assert use

```
#include <msl_utility>

template <class T>
T
foo(const T& t)
{
    Metrowerks::compile_assert<sizeof(T) >= sizeof(int)>
T_Must_Be_At_Least_As_Big_As_int;
    //...
    return t;
}

int main()
{
    int i;
    foo(i); // ok
    char c;
    foo(c); // Error      : illegal use of incomplete struct/union/class
                    //                      'Metrowerks::compile_assert<0>'
```

array_size

Given an array type, you can get the size of the array with array_size.

Listing 25.17 Example usage of array_size

```
typedef int Array[10];
size_t n = array_size<Array>::value;

n has the value of 10.
```

can_derive_from

A simple union of class type and union types.

```
bool b = can_derive_from<T>::value;
```

store_as - Container optimization

Starting with Pro 4.1 the standard sequences vector, deque and list implemented the “void* optimization” as described in section 13.5 of Stroustrup’s 3rd. In a nutshell, this optimization allows all Container<T*> to share an implementation with Container<void*> for the purpose of reducing template code bloat.

Remarks

Starting with Pro 6 containers will take this idea further and with more flexibility. Using store_as, one can specify what types can be stored as alternate types in the containers. For example, a vector<unsigned char> can be implemented as a vector<char>, thus saving on the instantiation of vector<unsigned char>. Only vector uses store_as in Pro 6, but other containers will have this capability in future releases.

```
template <> struct store_as<Type_to_be_optimized>
{typedef Implementaiton_Type type;};
```

For example, to specify that a container`<long>` be implemented in terms of a container`<unsigned long>`:

```
template <> struct store_as<long>
{
    typedef unsigned long type;
};
```

If a specialization of `store_as` does not appear for a type, then that type will be implemented as itself in containers.

This header has a default table of `store_as` specializations suitable for your platform. But if for some reason you are not happy with the shipping configuration, you can alter the behavior here. Additionally, this optimization can be turned off by #defining `_Inhibit_Container_Optimization` in `<mslconfig>`, or in a prefix file.

The requirements for a type to appear in a `store_as` specialization are:

It must have a trivial copy constructor, assignment operator and destructor.

Its default constructor must do nothing but cause all bytes in the object to be zeroed (if this constructor is used).

The two types in a `store_as` specialization must have the same `sizeof`.

User defined types can also take advantage of this optimization if they meet the above requirements (recommended only for those that really know what they are doing). Client code must declare that the user defined type is a POD, and then create a `store_as` entry. For example, here is a struct that will share its vector implementation with `vector<unsigned long>`:

Note that this is strictly a code size optimization. Functionality does not change. And it only saves on code size if you already have to use a `vector<unsigned long>` for some other reason (or if you are using a vector that is stored as a `vector<unsigned long>`).

Listing 25.18 A struct that shares its vector implementation

```
#include <msl_utility>
#include <vector>

struct A
{
    int data_;
};

template <> struct Metrowerks::is_POD<A>
{
    static const bool value = true;
};
```

```
template <> struct Metrowerks::store_as<A>
    {typedef unsigned long type;};

int main()
{
    std::vector<A> a(5);
// Shares implementation with vector<unsigned long>
}
```

call_traits

This struct is a collection of type definitions that ease coding of template classes when the template parameter may be a non-array object, an array, or a reference. The type definitions specify how to pass a type into a function, and how to pass it back out either by value, reference or const reference. The interface is:

```
call_traits<T>::value_type  
call_traits<T>::reference  
call_traits<T>::const_reference  
call_traits<T>::param_type
```

Remarks

The first three types are suggestions on how to return a type from a function by value, reference or const reference. The fourth type is a suggestion on how to pass a type into a method.

The call_traits struct is most useful in avoiding references to a reference which are currently illegal in C++. Another use is in helping to decay array-type parameters into pointers. In general, use of call_traits is limited to advanced techniques, and will not require specializations of call_traits to be made. For example uses of call_traits see compressed_pair. For an example specialization see alloc_ptr.

is_empty

Answers true if the type is a class or union that has no data members, otherwise answers false. This is a key struct for determining if the space for an “empty” object can be optimized away or not.

```
bool b = is_empty<T>::value;
```

compressed_pair

Like std::pair, but attempts to optimize away the space for either the first or second template parameter if the type is "empty". And instead of the members being accessible via the public data members first and second, they are accessible via member methods first() and second(). compressed_pair handles reference types as well as other types thanks to the call_traits template. This is a good example to study if you're wanting to see how to take advantage of either call_traits or is_empty. To see an example of how compressed_pair is used see alloc_ptr.

Remarks

Use of the single argument constructors will fail at compile time (ambiguous call) if first_type and second_type are the same type.

The swap specialization will call swap on each member if and only if its size has not been optimized away. The call to swap on each member will look both in std, and in the member's namespace for the appropriate swap specialization. Thus clients of compressed_pair need not put swap specializations into namespace std.

A good use of compressed_pair is in the implementation of a container that must store a function object. Function objects are typically zero-sized classes, but are also allowed to be ordinary function pointers. If the function object is a zero-sized class, then the container can optimize its space away by using it as a base class. But if the function object instantiates to a function pointer, it can not be used as a base class. By putting the function object into a compressed_pair, the container implementor need not worry whether it will instantiate to a class or function pointer.

MyContainer1 uses a zero-sized Compare object. On a 32 bit machine, the sizeof MyContainer1 will be 4 bytes as the space for Compare is optimized away by compressed_pair. But MyContainer2 instantiates Compare with an ordinary

function pointer which can't be optimized away. Thus the sizeof MyContainer2 is 8 bytes.

Listing 25.19 Example of compressed_pair

```
#include <iostream>
#include <functional>
#include <msl_utility>

template <class T, class Compare>
class MyContainer
{
public:
    explicit MyContainer(const Compare& c = Compare()) : data_(0, c) {}

    T*                  pointer()                  {return
data_.first();}

    const T*            pointer() const           {return data_.first();}

    Compare&           compare()                {return
data_.second();}

    const Compare&     compare() const           {return data_.second();}

    void               swap(MyContainer& y) {data_.swap(y.data_);}

private:
    Metrowerks::compressed_pair<T*, Compare> data_;
};

int main()
{
    typedef MyContainer<int, std::less<int> >      MyContainer1;
    typedef MyContainer<int, bool (*)(int, int)> MyContainer2;
    std::cout << sizeof(MyContainer1) << '\n';
    std::cout << sizeof(MyContainer2) << '\n';
}
```

alloc_ptr

An extension of std::auto_ptr. alloc_ptr will do everything that auto_ptr will do with the same syntax. Additionally alloc_ptr will deal with array new/delete:

```
alloc_ptr<int, array_deleter<int> > a(new int[4]);  
// Ok, destructor will use delete[]
```

Remarks

By adding the array_deleter<T> template parameter you can enable alloc_ptr to correctly handle pointers to arrays of elements.

alloc_ptr will also work with allocators which adhere to the standard interface. This comes in very handy if you are writing a container that is templated on an allocator type. You can instantiate an alloc_ptr to work with an allocator with:

```
alloc_ptr<T, Allocator<T>, typename Allocator<T>::size_type> a;
```

The third parameter can be omitted if the allocator is always going to allocate and deallocate items one at a time (e.g. node based containers).

alloc_ptr takes full advantage of compressed_pair so that it is as efficient as std::auto_ptr. The sizeof(alloc_ptr<int>) is only one word. Additionally alloc_ptr will work with a reference to an allocator instead of an allocator (thanks to call_traits). This is extremely useful in the implementation of node based containers.

This is essentially the std::auto_ptr interface with a few twists to accommodate allocators and size parameters.

MSL C++ Debug Mode

This chapter describes the MSL Debug Mode for code diagnostics.

Overview of MSL C++ Debug Mode

The STL portion of MSL C++ has a debug mode that can be used to diagnose common mistakes in code that uses the MSL C++ containers and their iterators. When an error is detected, a `std::logic_error` is thrown with an appropriate error message.

Types of Errors Detected

Given a container (such as `vector`), the following errors are detected in MSL Debug mode:

- Incrementing an iterator beyond `end()`.
- Decrementing an iterator before `begin()`.
- Dereferencing an iterator that is not dereferenceable.
- Any use of an invalid iterator besides assigning a valid value to it.
- Passing an iterator to a container method when that iterator does not point into that container.
- Comparison of two iterators that don't point into the same container.

How to Enable Debug Mode

To enable MSL C++ Debug mode simply uncomment this line in the MSL Configuration header `<mslconfig>` See "[C++ Switches, Flags and Defines](#)" for more information.

```
#define _MSL_DEBUG
```

Alternatively you can `#define _MSL_DEBUG` in a prefix file. Either way, you must rebuild your C++ library after flipping this switch. Convenience projects are provided under MSL(MSL_Build_Projects)/ to make this task easier. After rebuilding the C++

library, rebuild your application and run it. If there are any errors, a std::logic_error will be thrown. If exceptions are disabled, then instead the error function __msl_error(const char*) is called. This function can be defined by client code. There are some sample implementations in <mslconfig>. The default simply calls fprintf and abort.

Debug Mode Implementations

The debug facilities are available for the standard containers as well as the Metrowerks extension containers:

- “[deque](#)”
- “[list](#)”
- “[string](#)”
- “[vector](#)”
- “[tree-based containers - map, multimap, set, multiset](#)”
- “[cdeque](#)”
- “[slist](#)”
- “[hash-based containers - map, multimap, set, multiset](#)”

Each container has methods that will invalidate some or all outstanding iterators. If those iterators are invalidated, then their use (except for assigning a new valid iterator) will generate an error. An iterator is considered invalidated if it no longer points into the container, or if the container's method silently causes the iterator to point to a new element within the container. Some methods (such as swap, or list::splice) will transfer ownership of outstanding iterators from one container to another, but otherwise leave them valid.

In this “[Example of dereference at end:](#)” the iterator i is incremented to the end of the vector and then dereferenced and assigned through. In release mode this is undefined behavior and may overwrite other important information in your application. However in debug mode this example prints out:

```
MSL DEBUG: dereferenced invalid iterator
```

Listing 26.1 Example of dereference at end:

```
#include <iostream>
#include <vector>
#include <stdexcept>

int main()
```

```
{  
    try  
    {  
        std::vector<int> v(10);  
        std::vector<int>::iterator i = v.begin() + 9;  
        *i = 9; // ok  
        ++i; // ok  
        *i = 10; // error  
    } catch (std::exception& e)  
    {  
        std::cerr << e.what() << '\n';  
    }  
    catch (...)  
    {  
        std::cerr << "Unknown exception caught\n";  
    }  
}
```

In the [“Example of iterator/list mismatch:”](#) an iterator is initialized to point into the first list. But then this iterator is mistakenly used to erase an element from a second list. This is normally undefined behavior. In debug mode this example prints out:

```
MSL DEBUG: invalid iterator given to list
```

Listing 26.2 Example of iterator/list mismatch:

```
#include <iostream>  
#include <list>  
#include <stdexcept>  
  
int main()  
{  
    try  
    {  
        std::list<int> l1(10), l2(10);  
        std::list<int>::iterator i = l1.begin();  
        l2.erase(i); // error  
    }  
    catch (std::exception& e)  
    {  
        std::cerr << e.what() << '\n';  
    }  
    catch (...)  
    {  
        std::cerr << "Unknown exception caught\n";  
    }  
}
```

```
}
```

In the “[Example of use of invalidated iterator:](#)” the push_back method on deque invalidates all iterators. When the loop goes to increment i, it is operating on an invalidated iterator. This is normally undefined behavior. In debug mode this example prints out:

```
MSL DEBUG: increment end or invalid iterator
```

Listing 26.3 Example of use of invalidated iterator:

```
#include <iostream>
#include <deque>
#include <stdexcept>

int main()
{
    try
    {
        std::deque<int> d(10);
        std::deque<int>::iterator i = d.begin(), e = d.end();
        for (; i != e; ++i)
            d.push_back(0);
    }
    catch (std::exception& e)
    {
        std::cerr << e.what() << '\n';
    }
    catch (...)
    {
        std::cerr << "Unknown exception caught\n";
    }
}
```

Debug Mode Containers

The list below documents when iterators are invalidated for each container, and for each method in that container:

deque

Various functions are included for debugging the `deque` class.

assign

All assign methods (including operator=) invalidate all iterators.

push_front/back

Invalidates all iterators.

pop_front/back

Only the iterators to the erased elements are invalidated.

insert

All iterators are invalidated.

erase

If erasing at either end, only iterators to elements erased are invalidated, else all iterators are invalidated.

resize

If the size increases, all iterators are invalidated. Else only iterators to the erased elements are invalidated.

clear

Invalidates all iterators.

swap

Iterators remain valid, but they now point into the swapped container.

Remarks

The index operator is range checked just like the `at()` method.

list

Various functions are included for debugging the `list` class.

assign

All assign methods (including operator=) invalidate all iterators.

push_front/back

No iterators are invalidated.

pop_front/back

Only the iterators to the erased elements are invalidated.

insert

No iterators are invalidated.

erase

Only the iterators to the erased elements are invalidated.

resize

Only the iterators to the erased elements are invalidated.

clear

Invalidates all iterators.

swap

Iterators remain valid, but they now point into the swapped container.

splice, merge

Iterators remain valid, but iterators into the argument list now point into this.

string

Various functions are included for debugging the `string` class.

assign

All assign methods (including operator=) invalidate all iterators.

push_back

If capacity is not exceeded no iterators are invalidated, else all iterators are invalidated.

pop_back

Only the iterators to the erased element is invalidated.

insert

If capacity is not exceeded iterators to elements beyond the insertion point are invalidated, else all iterators are invalidated.

erase

Iterators to elements at and beyond the erased elements are invalidated.

resize

If capacity is exceeded all iterators are invalidated, else iterators to any erased elements are invalidated.

clear

Invalidates all iterators.

swap

Iterators remain valid, but they now point into the swapped container.

Remarks

The index operator is range checked just like the `at()` method.

vector

Various functions are included for debugging the `vector` class.

assign

All assign methods (including operator=) invalidate all iterators.

push_back

If capacity is not exceeded no iterators are invalidated, else all iterators are invalidated.

pop_back

Only the iterators to the erased element is invalidated.

insert

If capacity is not exceeded iterators to elements beyond the insertion point are invalidated, else all iterators are invalidated.

erase

Iterators to elements at and beyond the erased elements are invalidated.

resize

If capacity is exceeded all iterators are invalidated, else iterators to any erased elements are invalidated.

clear

Invalidates all iterators.

swap

Iterators remain valid, but they now point into the swapped container.

Remarks

The index operator is range checked just like the `at()` method.

tree-based containers - map, multimap, set, multiset

Various functions are included for debugging the free-based container classes `map`, `multimap`, `set` and `multiset` classes.

assign

Invalidates all iterators.

insert

No iterators are invalidated.

erase

Only the iterators to the erased elements are invalidated.

clear

Invalidates all iterators.

swap

Iterators remain valid, but they now point into the swapped container.

cdeque

Various functions are included for debugging the `cdeque` class.

assign

All assign methods (including operator=) invalidate all iterators.

push_front/back

If capacity exceeded invalidates all iterators, else no iterators are invalidated.

pop_front/back

Only the iterators to the erased elements are invalidated.

insert

If capacity exceeded or if insert position is not at the front or back, invalidates all iterators, else no iterators are invalidated.

erase

If erasing at either end, only iterators to elements erased are invalidated, else all iterators are invalidated.

resize

If capacity exceeded invalidates all iterators, else iterators to any erased elements are invalidated.

clear

Invalidates all iterators.

swap

Iterators remain valid, but they now point into the swapped container.

Remarks

The index operator is range checked just like the at() method.

slist

Various functions are included for debugging the `slist` class.

assign

All assign methods (including operator=) invalidate all iterators.

push_front/back

No iterators are invalidated.

pop_front/back

Only the iterators to the erased elements are invalidated.

insert

No iterators are invalidated.

erase

Only the iterators to the erased elements are invalidated.

resize

Only the iterators to the erased elements are invalidated.

clear

Invalidates all iterators.

swap

Iterators remain valid, but they now point into the swapped container.

splice, splice_after, merge

Iterators remain valid, but iterators into the argument list now point into this.

Remarks

Incrementing end() is not an error, it gives you begin().

hash-based containers - map, multimap, set, multiset

Various funstions are included for debugging the hash based map, mulitpmmap, set and multiset classes.

assign

Invalidates all iterators.

insert

If `load_factor()` attempts to grow larger than `load_factor_limit()`, then the table is rehashed which invalidates all iterators, else no iterators are invalidated.

erase

Only the iterators to the erased elements are invalidated.

clear

Invalidates all iterators.

swap

Iterators remain valid, but they now point into the swapped container.

Invariants

In addition to the iterator checking described above, each container (except `string`) has a new member method:

```
bool invariants() const;
```

This method can be called at any time to assess the container's class invariants. If the method returns false, then the container has somehow become corrupted and there is a bug (most likely in client code, but anything is possible). If the method returns true, then no errors have been detected. This can easily be used in debug code like:

Listing 26.4 Example of invariant debugging

```
#include <vector>
#include <cassert>

int main()
{
    int iarray[4];
    std::vector<int> v(10);
    assert(v.invariants());
    for (int i = 0; i <= 4; ++i)
        iarray[i] = 0;
```

```
    assert(v.invariants());
}
```

The for loop indexing over `iarray` goes one element too far and steps on the vector. The assert after the loop detects that the vector has been compromised and fires.

Be warned that the `invariants` method for some containers can have a significant computational expense ($O(N)$), so this method is not advised for release code (nor are any of the debug facilities).

Hash Libraries

This chapter is a reference guide to the hash support in the Metrowerks standard libraries.

The Hash Containers Library

This chapter on Metrowerks implementation of hashes is made up of.

- [“General Hash Issues”](#)
- [“Hash_set”](#)
- [“Hash_map”](#)
- [“Hash_fun”](#)

A separate chapter [“The MSL Utilities Library”](#) is also useful when understanding the methodology.

General Hash Issues

This document reflects issues that are common to `hash_set`, `hash_multiset`, `hash_map` and `hash_multimap`. Rather than repeat each of these issue for each of the four hash containers, they are discussed here once and for all.

Introduction

These classes are analogous to `std::set`, `std::multiset`, `std::map` and `std::multimap`, but are based on a hash table. The design and implementation of these classes has the following goals:

- High CPU performance
- Minimum memory usage
- Ease of use
- Control over hashing details

- Backward compatibility with previous Metrowerks hash containers
- Compatibility with hash containers supplied by SGI and Microsoft

Not all of these goals can be simultaneously met. For example, optimizations often require a trade-off between size and speed. “Ease of use” can pull the design in opposite directions from “control over details”. And it is not possible to be 100% compatible with two or more other implementations, when they are not compatible among themselves. Nevertheless, thought and concessions have been made toward all of these goals.

Namespace Issues

These classes are a Metrowerks extension to the standard C++ library. So they have been implemented within the namespace Metrowerks. There are several techniques available for accessing these classes:

Fully Qualified Reference:

One technique is to fully qualify each use of a Metrowerks extension with the full namespace. For example:

Listing 27.1 Qualified Reference

```
#include <hash_set>

int main()
{
    Metrowerks::hash_set<int> a;
```

Namespace Alias

“Metrowerks” is quite a long name and can get tiresome continually typing. But it is not likely to conflict with other library's namespaces. You can easily shorten the Metrowerks namespace while still retaining the protection of namespaces through the

use of an alias. For example, here is how to refer to the Metrowerks namespace as “mw”:

Listing 27.2 Namespace Alias

```
#include <hash_map>

namespace mw = Metrowerks;

int main()
{
    mw::hash_map<int, int> a;
}
```

The short name “mw” is much more likely to conflict with other's libraries, but as the implementor of your code you can choose your aliases such that there is no conflict.

Using Declaration

Using declarations can bring individual names into the current namespace. They can be used either at namespace scope (outside of functions) or at function scope (inside of functions). Here is an example use of a using declaration at namespace scope:

Listing 27.3 Namespace Scope

```
#include <hash_set>

using Metrowerks::hash_multiset;

int main()
{
    hash_multiset<int> a;
}
```

Remarks

Anywhere below the using declaration, hash_set can be referred to without the use of the Metrowerks qualifier.

Using Directive

Using directives will import every name in one namespace into another. These can be used to essentially “turn off” namespaces so that you don't have to deal with them. They can be used at namespace scope, or to limit their effect, can also be used at function scope. For example:

Listing 27.4 Function Scope

```
#include <hash_map>

int main()
{
    using namespace Metrowerks;
    hash_multimap<int, int> a;
}
```

Remarks

In the above example, any name in the Metrowerks namespace can be used in main without qualification.

Compatibility Headers

Most headers with the name <name> have an associated compatibility header <name.h>. These compatibility headers simply issue using declarations for all of the names they contain. Here is an example use:

Listing 27.5 Using Declarations for Names

```
#include <hash_set.h>
#include <hash_map.h>

int main()
{
    hash_set<int> a;
    hash_map<int, int> b;
```

Constructors

Each hash container has a constructor which takes the following arguments, with the following defaults:

```
size_type num_buckets = 0  
const key_hasher& hash = key_hasher()  
const key_compare& comp = key_compare()  
float load_factor_limit = 2  
float growth_factor = 4  
const allocator_type& a = allocator_type()
```

Remarks

Since all arguments have defaults, the constructor serves as a default constructor. It is also declared explicit to inhibit implicit conversions from the first argument: `size_type`. The first argument is a way to specify the initial number of buckets. This was chosen as the first parameter in order to remain compatible both with previous versions of Metrowerks hash containers, as well as the SGI hash containers.

The second and third parameters allow client code to initialize the hash and compare function objects if necessary. This will typically only be necessary if ordinary function pointers are being used. When function objects are used, the default constructed function object is often sufficient.

The fourth and fifth parameters allow you to set the initial values of `load_factor_limit` and `growth_factor`. Details on how these parameters interact with the `size()` and `bucket_count()` of the container can be found in the capacity section.

A second constructor also exists that accepts templated input iterators for constructing a hash container from a range. After the pair of iterators, the 6 parameters from the first constructor follow in the same order, and with the same defaults.

Iterator Issues

The hash iterators are of the forward type. You can increment them via prefix or postfix `++`, but you can not decrement them. This is compatible with our previous implementation of the hash containers, and with the hash containers provided by SGI. But the hash iterators provided by Microsoft are bidirectional. Code that takes advantage of the decrement operators offered by Microsoft will fail at compile time in the Metrowerks implementation.

Remarks

Forward iterators were chosen over bidirectional iterators to save on memory consumption. Bidirectional iterators would add an additional word of memory to each entry in the hash container. Furthermore a hash container is an unordered collection of elements. This “unorder” can even change as elements are added to the hash container. The ability to iterate an unordered collection in reverse order has a diminished value.

Iterators are invalidated when the number of buckets in the hash container change. This means that iteration over a container while adding elements must be done with extra care (see Capacity for more details). Despite that iterators are invalidated in this fashion, pointers and references into the hash container are never invalidated except when the referenced element is removed from the container.

Capacity

`empty`, `size` and `max_size` have semantics identical with that described for standard containers.

Remarks

The load factor of a hash container is the number of elements divided by the number of buckets:

$$\text{load_factor} = \frac{\text{size}()}{\text{bucket_count}()}$$

During the life time of a container, the load factor is at all times less than or equal to the load factor limit:

```
size()  
----- <= load_factor_limit()  
bucket_count()
```

This is a class invariant. When both size() and bucket_count() are zero, the load_factor is interpreted to be zero. size() can not be greater than zero if bucket_count() is zero. Client code can directly or indirectly alter size(), bucket_count() and load_factor_limit(). But at all times, bucket_count() may be adjusted so that the class invariant is not compromised.

If client code increases size() via methods such as insert such that the invariant is about to be violated, bucket_count() will be increased by growth_factor().

If client code decreases size() via methods such as erase, the invariant can not be violated.

If client code increases load_factor_limit(), the invariant can not be violated.

If client code decreases load_factor_limit() to the point that the invariant would be violated, then bucket_count() will be increased just enough to satisfy the invariant.

If client code increases bucket_count(), the invariant can not be violated.

If client code decreases bucket_count() to the point that the invariant would be violated, then bucket_count() will be decreased only to the minimum amount such that the invariant will not be violated.

The final item in the bulleted list results amounts to a “shrink to fit” statement.

```
myhash.bucket_count(0); // shrink to fit
```

The above statement will reduce the bucket count to the point that the load_factor() is just at or below the load_factor_limit().

```
bucket_count()
```

returns the current number of buckets in the container.

bucket_count(size_type num_buckets) sets the number of buckets to the first prime number that is equal to or greater than num_buckets, subject to the class invariant described above. It returns the actual number of buckets that were set. This is a relatively expensive operation as all items in the container must be rehashed into the new container. This routine is analogous to vector's reserve. But it does not reserve space for a number of elements. Instead it sets the number

of buckets which in turn reserves space for elements, subject to the setting of `load_factor_limit()`.

```
load_factor()
    returns size()/bucket_count() as a float.
load_factor_limit()
    returns the current load_factor_limit.
```

`load_factor_limit(float lf)` sets the load factor limit. If the new load factor limit is less than the current load factor limit, the number of buckets may be increased if the new load factor limit would violate the class invariant as described above.

You can completely block the automatic change of `bucket_count` with:

```
myhash.load_factor_limit(INFINITY);
```

This may be important if you are wanting outstanding iterators to not be invalidated while inserting items into the container. The argument to `load_factor_limit` must be positive, else an exception of type `std::out_of_range` is thrown.

The `growth_factor` functions will read and set the `growth_factor`. When setting, the new growth factor must be greater than 1 else an exception of type `std::out_of_range` is thrown.

The `collision(const_iterator)` method will count the number of items in the same bucket with the referred to item. This may be helpful in diagnosing a poor hash distribution.

insert

Insert For Unique Hashed Containers

`hash_set` and `hash_map`

have the following insert method:

```
std::pair<iterator, bool>
insert(const value_type& x);
```

Remarks

If `x` does not already exist in the container, it will be inserted. The returned iterator will point to the newly inserted `x`, and the `bool` will be true. If `x` already

exists in the container, the container is unchanged. The returned iterator will point to the element that is equal to `x`, and the `bool` will be false.

```
iterator insert(iterator, const value_type& x);
```

Operates just like the version taking only a `value_type`. The iterator argument is ignored. It is only present for compatibility with standard containers.

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

Inserts those elements in `[first, last)` that don't already exist in the container.

insert

The `insert` for multi-hashed containers functions `hash_multiset` and `hash_multimap` have the following insert methods.

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

Remarks

In the first `insert` prototype `x` is inserted into the container and an iterator pointing to the newly inserted value is returned. If values equal to `x` already exist in the container, then the new element is inserted after all other equal elements. This ordering is stable throughout the lifetime of the container.

In the second prototype `insert` first checks to see if `*p` is equivalent to `x` according to the `compare` function. If it is, then `x` is inserted before `p`. If not then `x` is inserted as if the `insert` without an iterator was used. An iterator is returned which points to the newly inserted element.

The final `insert` prototype inserts `[first, last)` into the container. Equal elements will be ordered according to which was inserted first.

erase

Erases has items at the position or selected items.

```
void erase(iterator position);  
size_type erase(const key_type& x);  
void erase(iterator first, iterator last);
```

Remarks

The first `erase` function erases the item pointed to by position from the container. The second erases all items in the container that compare equal to `x`. and returns the number of elements erased. The third `erase` erases the range [first, last) from the container.

```
swap(hash_set& y);  
Swaps the contents of *this with y in constant time.  
clear();  
Erases all elements from the container.
```

Observers

Miscellaneous functions used in the hash implementation.

```
get_allocator() const;  
Returns the allocator the hash container was constructed with.  
  
key_comp() const  
Returns the comparison function the hash container was constructed with.  
  
value_comp() const  
Returns the comparison function used in the underlying hash table. For hash_set and hash_multiset, this is the same as key_comp().  
  
key_hash()  
Returns the hash function used in the underlying hash table.
```

Returns the hash function the hash container was constructed with.

`value_hash()`

Returns the hash function used in the underlying hash table. For `hash_set` and `hash_multiset`, this is the same as `key_hash()`.

Set Operations

Miscellaneous hash set utility functions.

find

`iterator find(const key_type& x) const;`

Returns an iterator to the first element in the container that is equal to `x`, or if `x` is not in the container, returns `end()`.

count

`count(const key_type& x) const`

Returns the number of elements in the container equal to `x`.

equal_range

`std::pair<iterator, iterator> equal_range(const key_type& x);`

Returns a pair of iterators indicating a range in the container such that all elements in the range are equal to `x`. If no elements equal to `x` are in the container, an empty range is returned.

Global Methods

Global has functions.

swap

`swap(x, y)`

Hash Libraries

Hash_set

Same semantics as `x.swap(y)`.

operator ==

```
operator == (x, y)
```

Returns true if `x` and `y` contain the same elements in the same order. To accomplish this they most likely must have the same number of buckets as well.

operator !=

```
operator != (x, y)
```

Returns `!(x == y)`

Incompatibility with Previous versions Metrowerks Hash Containers

The current hash containers are very compatible with previous versions except for a few methods:

You can no longer compare two hash containers with the ordering operators: `<`, `<=`, `>`, `>=`. Since hash containers are unordered sets of items, such comparisons have little meaning.

`lower_bound` is no longer supported. Use `find` instead if you expect the item to be in the container. If not in the container, `find` will return `end()`. As there is no ordering, finding the position which an item could be inserted before has no meaning in a hash container.

`upper_bound` is no longer supported. Again because of the fact that hash containers are unordered, `upper_bound` has questionable semantics.

Despite the lack of `lower_bound` and `upper_bound`, `equal_range` is supported. In a pinch, `equal_range().first` suffices for `lower_bound`, and `equal_range().second` suffices for `upper_bound`.

Hash_set

This header contains two classes:

- `hash_set`
- `hash_multiset`.

`hash_set` is a container that holds an unordered set of items, and no two items in the container can compare equal. `hash_multiset` permits duplicate entries. Also see the General Hash Issues Introduction.

NOTE This header is non-standard. The classes herein are offered as extensions to the C++ standard. They are marked as such by the namespace Metrowerks.

Introduction.

These containers are in the namespace Metrowerks. See Namespace Issues for details and hints about how to best take advantage of this fact.

`hash_set` and `hash_multiset` are largely compatible with previous versions of these classes which appeared in namespace `std`. But see Incompatibility for a short list of incompatibilities.

Old HashSet Headers

Previous versions of CodeWarrior placed `hash_set` and `hash_multiset` in the headers `<hashset.h>` and `<hashmset.h>` respectively. These headers are still available, but should be used only for transition purposes. They will disappear in a future release. These headers import the contents of `<hash_set>` into the `std` namespace (as previous versions of `hash_(multi)set` were implemented in `std`).

Listing 27.6 Old HashSet Headers

```
#include <hashset.h>

int main()
{
    std::hash_set<int> a;
}
```

Template Parameters

Both `hash_set` and `hash_multiset` have the following template parameters and defaults:

Listing 27.7 Template Parameters and Defaults

```
template <class T, class Hash = hash<T>, class Compare =
    std::equal_to<T>, class Allocator = std::allocator<T> >
class hash_(multi)set;
```

The first parameter is the type of element the set is to contain. It can be almost any type, but must be copyable.

The second parameter is the hash function used to look up elements. It defaults to the hash function in `<hash_fun>`. Client code can use `hash<T>` as is, specialize it, or supply completely different hash function objects or hash function pointers. The hash function must accept a `T`, and return a `size_t`.

The third parameter is the comparison function which defaults to `std::equal_to<T>`. This function should have equality semantics. A specific requirement is that if two keys compare equal according to `Compare`, then they must also produce the same result when processed by `Hash`.

The fourth and final parameter is the allocator, which defaults to `std::allocator<T>`. The same comments and requirements that appear in the standard for allocators apply here as well.

Nested Types

`hash_set` and `hash_multiset` define a host of nested types similar to standard containers. Several noteworthy points:

- `key_type` and `value_type` are the same type and represent the type of element stored.
- `key_hasher` and `value_hasher` are the same type and represent the hash function.
- `key_compare` and `value_compare` are the same type and represent the comparison function.
- `iterator` and `const_iterator` are the same type and have semantics common to a forward `const_iterator`.

Iterator Issues

See Iterator Issues that are common to all hash containers.

Iterators of `hash_set` and `hash_multiset` are not mutable. They act as `const_iterators`. One can cast away the `const` qualification of references returned by iterators, but if the element is modified such that the hash function now has a different value, the behavior is undefined.

See Capacity for details on how to control the number of buckets.

hash_set

hash_set is a container based on a hash table that supports fast find, insert and erase. The elements in a hash_set are unordered. A hash_set does not allow multiple entries of equivalent elements.

Hash_map

The hash_map is a container that holds an unordered set of key-value pairs, and no two keys in the container can compare equal. hash_multimap permits duplicate entries. Also see the General Hash Issues Introduction.

This header contains two classes:

- `hash_map`
- `hash_multimap`

NOTE This header is non-standard. The classes herein are offered as extensions to the C++ standard. They are marked as such by the namespace Metrowerks.

Introduction

These containers are in the namespace Metrowerks. See Namespace Issues for details and hints about how to best take advantage of this fact.

hash_map and hash_multimap are largely compatible with previous versions of these classes which appeared in namespace std. But see Incompatibility for a short list of incompatibilities.

Old Hashmap Headers

Previous versions of CodeWarrior placed `hash_map` and `hash_multimap` in the headers `<hashmap.h>` and `<hashmmap.h>` respectively. These headers are still available, but should be used only for transition purposes. They will disappear in a future release. These headers import the contents of `<hash_map>` into the std namespace (as previous versions of `hash_(multi)map` were implemented in std.

Hash Libraries

Hash_map

Listing 27.8 Old Hashmap Headers

```
#include <hashmap.h>

int main()
{
    std::hash_map<int, int> a;
}
```

Template Parameters

Both `hash_map` and `hash_multimap` have the following template parameters and defaults:

Listing 27.9 Hashmap Template Parameters

```
template <class Key, class T, class Hash = hash<Key>,
          class Compare = std::equal_to<Key>,
          class Allocator = std::allocator<std::pair<const Key, T>>>
class hash_(multi)map;
```

The first parameter is the type of key the map is to contain. It can be almost any type, but must be copyable.

The second parameter is the type of the value that will be associated with each key. It can be almost any type, but must be copyable.

The third parameter is the hash function used to look up elements. It defaults to the `hash` function in `<hash_fun>`. Client code can use `hash<Key>` as is, specialize it, or supply completely different hash function objects or hash function pointers. The hash function must accept a `Key`, and return a `size_t`.

The fourth parameter is the comparison function which defaults to `std::equal_to<Key>`. This function should have equality semantics. A specific requirement is that if two keys compare equal according to `Compare`, then they must also produce the same result when processed by `Hash`.

The fifth and final parameter is the allocator, which defaults to `std::allocator<std::pair<const Key, T>>`. The same comments and requirements that appear in the standard for allocators apply here as well.

Nested Types

`hash_map` and `hash_multimap` define a host of nested types similar to standard containers. Several noteworthy points:

- `key_type` and `value_type` are not the same type. `value_type` is a pair<const `Key`, `T`>.
- `key_hasher` and `value_hasher` are not the same type. `key_hasher` is the template parameter `Hash`. `value_hasher` is a nested type which converts `key_hasher` into a function which accepts a `value_type`.

– `value_hasher` has the public typedef's

```
typedef value_type argument_type;  
typedef size_type result_type;
```

This qualifies it as a `std::unary_function` (as defined in `<functional>`) and so could be used where other functionals are used.

– `value_hasher` has these public member functions:

```
size_type operator()(const value_type& x) const;  
size_type operator()(const key_type& x) const;
```

These simply return the result of `key_hasher`, but with the first operator extracting the `key_type` from the `value_type` before passing the `key_type` on to `key_hasher`.

- `Key_compare` and `value_compare` are not the same type. `key_compare` is the template parameter `Compare`. `value_compare` is a nested type which converts `key_compare` into a function which accepts a `value_type`.

– `value_compare` has the public typedef's

```
typedef value_type first_argument_type;  
typedef value_type second_argument_type;  
typedef bool result_type;
```

This qualifies it as a `std::binary_function` (as defined in `<functional>`) and so could be used where other functionals are used.

– `value_compare` has these public member functions:

```
bool operator()(const value_type& x,
                 const value_type& y) const;
bool operator()(const key_type& x,
                 const value_type& y) const;
bool operator()(const value_type& x,
                 const key_type& y) const;
```

These pass their arguments on to key_compare, extracting the key_type from value_type when necessary.

Iterator Issues

See Iterator Issues that are common to all hash containers.

See Capacity for details on how to control the number of buckets.

Element Access

```
mapped_type& operator[](const key_type& x);
```

If the key x already exists in the container, returns a reference to the mapped_type associated with that key. If the key x does not already exist in the container, inserts a new entry: (x, mapped_type()), and returns a reference to the newly created, default constructed mapped_type.

Hash_fun

<hash_fun> declares a templated struct which serves as a function object named hash. This is the default hash function for all hash containers. As supplied, hash works for integral types, basic_string types, and char* types (c-strings).

NOTE	This header is non-standard. The classes herein are offered as extensions to the C++ standard. They are marked as such by the namespace Metrowerks.
-------------	---

Client code can specialize hash to work for other types. For example:

Alternatively, client code can simply supply customized hash functions to the hash containers via the template parameters.

The returned `size_t` should be as evenly distributed as possible in the range `[0, numeric_limits<size_t>::max()]`. Logic in the hash containers will take care of folding this output into the range of the current number of buckets.

Hash Libraries

Hash_fun

Metrowerks::threads

This chapter is a reference guide to the threads support in the Metrowerks standard libraries.

Overview of Metrowerks Threads Library

If you're already familiar with boost::threads, then you'll be very comfortable with Metrowerks::threads. The interface closely follows the boost library. There are some minor differences.

The biggest difference is that the library is part of MSL C++, and lives in namespace Metrowerks. The entire package can be accessed via `<msl_thread>`. It is essentially a fairly thin C++ wrapper over a sub-set of Posix-threads. But it also can run on top of Apple's MP tasks (for PEF applications). And there is also a "single thread" version where most of the code just does nothing. It is there to ease porting multithreaded code to a single threaded environment. But be aware that your multithreaded logic may or may not translate into a working single threaded application (especially if you deal with condition variables).

The threads library currently has these configuration flags:

Table 28.1 Threads Configuration Flags

Flag	Effects
<code>_MSL_SINGLE_THREAD</code>	A do-nothing standin
<code>_MSL_USE_PTHREADS</code>	Poxsix-Threads
<code>_MSL_USE_MP_TASKS</code>	Apple Carbon MP-tasks
<code>_MSL_USE_WINTHREADS</code>	Windows threads

MSL C++ will automatically configure itself based on how `_MSL_THREADS` is set. However you can override the automatic configuration simply by setting it yourself in your prefix file or preprocessor preference panel. You must recompile the C++ library to have the same setting.

You can now create a runtime check to make sure your MSL C++ is compiled with consistent settings:

```
#include <msl_utility>
int main()
{
    check(Metrowerks::msl_settings());
}
```

This program will assert if it finds anything inconsistent between itself and the way MSL C++ was compiled.

Mutex and Locks

Metrowerks::threads has 6 types of mutexes.

- mutex
- try_mutex
- timed_mutex
- recursive_mutex
- recursive_timed_mutex

Listing 28.1 Mutex synopsis

```
class mutex
{
public:
    typedef /* details */ scoped_lock;

    mutex();
    ~mutex();
};

class try_mutex
{
public:
    typedef /* details */ scoped_lock;
    typedef /* details */ scoped_try_lock;

    try_mutex();
    ~try_mutex();
};
```

```
class timed_mutex
{
public:
    typedef /* details */ scoped_lock;
    typedef /* details */ scoped_try_lock;
    typedef /* details */ scoped_timed_lock;

    timed_mutex();
    ~timed_mutex();
};

class recursive_mutex
{
public:
    typedef /* details */ scoped_lock;

    recursive_mutex();
    ~recursive_mutex();
};

class recursive_try_mutex
{
public:
    typedef /* details */ scoped_lock;
    typedef /* details */ scoped_try_lock;

    recursive_try_mutex();
    ~recursive_try_mutex();
};

class recursive_timed_mutex
{
public:
    typedef /* details */ scoped_lock;
    typedef /* details */ scoped_try_lock;
    typedef /* details */ scoped_timed_lock;

    recursive_timed_mutex();
    ~recursive_timed_mutex();
};
```

Note that each mutex type has only a default constructor and destructor. It is not copyable, and it does not even have lock and unlock functions. You access this functionality via one of the nested types:

- scoped_lock
- scoped_try_lock
- scoped_timed_lock

Listing 28.2 A scoped_lock

```
template <typename Mutex>
class scoped_lock
{
public:
    typedef Mutex mutex_type;

    explicit scoped_lock(mutex_type& m);
    scoped_lock(mutex_type& m, bool lock_it);
    ~scoped_lock();

    void lock();
    void unlock();
    bool locked() const;
    operator int bool_type::* () const;
};
```

You can use the `scoped_lock` to lock and unlock the associated `mutex`, and test whether it is locked or not (the operator `bool_type` is just a safe way to test the `lock` in an if statement like you might a pointer), for example:

```
if (my_lock) ...
```

Normally you won't use any of the `scoped_lock`'s members except its constructor and destructor. These lock and unlock the `mutex` respectively.

Listing 28.3 Example of lock and unlock usage

```
#include <msl_thread>

Metrowerks::mutex foo_mut;

void foo()
{
    Metrowerks::mutex::scoped_lock lock(foo_mut);
    // only one thread can enter here at a time
```

```
} // foo_mut is implicitly unlocked here, no matter how foo returns
```

In single thread mode, the above example compiles, and the lock simply doesn't do anything. If you expect `foo()` to call itself, or to call another function which will lock the same `mutex` (before `foo` releases `foo_mut`), then you should use a recursive `mutex`.

A `mutex` can conveniently be a class member, which can then be used to lock various member functions on entry. But recall that your class copy constructor will need to create a fresh `mutex` when copying, as the `mutex` itself can not be copied (or assigned to).

In some cases you want to lock the `mutex` only if you don't have to wait for it. If it is unlocked, you lock it, else your thread can do something else. Use `scoped_try_lock` for this application. Note that not all `mutex` types support `scoped_try_lock` (have it as a nested type). The `scoped_try_lock` looks just like `scoped_lock` but adds this member function `bool try_lock()`,

Listing 28.4 Example of try_lock() usage

```
#include <msl_thread>

Metrowerks::try_mutex foo_mut;

void foo()
{
    Metrowerks::try_mutex::scoped_try_lock lock(foo_mut, false);
    if (lock.try_lock())
    {
        // got the lock
    }
    else
    {
        // do something else
    }
}
```

In the above example, the second parameter in the constructor tells the lock to not lock the `mutex` upon construction (else you might have to wait).

Sometimes you are willing to wait for a `mutex` lock, but only for so long, and then you want to give up. `scoped_timed_lock` is the proper lock for this situation. It looks just like a `scoped_lock` but adds two members:

```
bool timed_lock(const universal_time& unv_time);  
bool timed_lock(const elapsed_time& elps_time);
```

These let you specify the amount of time you're willing to wait, either in terms of an absolute time (`universal_time`), or in terms of an interval from the current time (`elapsed_time`).

Listing 28.5 Example of `timed_lock()`

```
Metrowerks::timed_mutex foo_mut;  
  
void foo()  
{  
    Metrowerks::timed_mutex::scoped_timed_lock lock(foo_mut, false);  
    Metrowerks::elapsed_time time_out(1, 500000000);  
    if (lock.timed_lock(time_out))  
    {  
        // got the lock  
    }  
    else  
    {  
        // do something else  
    }  
}
```

This specifies that the thread should quit trying for the lock after 1.5 seconds. Both `elapsed_time` and `universal_time` are simple structs with `sec_` and `nsec_` exposed data members representing seconds and nanoseconds. In the case of `universal_time`, this is the number of seconds and nanoseconds since midnight Jan. 1, 1970. The `universal_time` default constructor returns the current time. So the above example could have also been written as in "["Alternate example of `timed_lock\(\)` usage"](#)".

Listing 28.6 Alternate example of `timed_lock()` usage

```
void foo()  
{  
    Metrowerks::timed_mutex::scoped_timed_lock lock(foo_mut, false);  
    Metrowerks::elapsed_time time_out(1, 500000000);  
    Metrowerks::universal_time now;  
    if (lock.timed_lock(now + time_out))  
    {  
        // got the lock  
    }
```

```
else
{
    // do something else
}
}
```

In general you can add and subtract and compare `universal_time` and `elapsed_time` as makes sense.

In single thread mode, all locks will lock their mutexes and return immediately (times are ignored). However, if you try to lock a locked mutex, or unlock an unlocked mutex, then an exception of type `Metrowerks::lock_error` (derived from `std::exception`) will be thrown (even in single thread mode).

Threads

The class `Metrowerks::thread` represents a thread of execution.

Listing 28.7 Class thread synopsis

```
class thread
{
public:
    thread();
    explicit thread(const std::tr1::function<void ()>& f);
    explicit thread(void (*f)());

    ~thread();

    bool operator==(const thread& rhs) const;
    bool operator!=(const thread& rhs) const;

    void join();

    static void sleep(const universal_time& unv_time);
    static void sleep(const elapsed_time& elps_time);
    static void yield();
};
```

A default constructed `thread` object represents the current thread. You can create a new thread of execution by passing a general function object, or a simple function pointer. In either case, the function must take no parameters and return `void`. When a thread destructs, it "detaches" the thread of execution (to use Posix-threads terminology).

Once this happens, the thread is independent. You will no longer be able to refer to it, and it will clean up after itself when it terminates. But should main terminate before the thread does, the program ends anyway. You can have one thread wait on another with the `join()` member function.

Listing 28.8 Example of join() function

```
#include <msl_thread>
#include <iostream>

void do_something()
{
    std::cout << "Thread 1!\n";
}

int main()
{
    Metrowerks::thread t1(do_something);
    t1.join();
}
```

In the above example, `main` will wait for (`join` with) `t1`. Note that global objects like `std::cout` must be protected if more than one thread is going to access it. You must do this work yourself.

Listing 28.9 Example of protecting threads

```
#include <msl_thread>
#include <iostream>

Metrowerks::mutex cout_mutex;

void do_something()
{
    Metrowerks::mutex::scoped_lock lock(cout_mutex);
    std::cout << "Thread 1!\n";
}

void do_something_else()
{
    Metrowerks::mutex::scoped_lock lock(cout_mutex);
    std::cout << "Thread 2!\n";
}
```

```
int main()
{
    std::cout << "Main\n";
    Metrowerks::thread t1(do_something);
    Metrowerks::thread t2(do_something_else);
    t1.join();
    t2.join();
}
```

In this example, each thread locks `cout_mutex` before using `cout`. `main()` didn't have to lock `cout` because no other threads started until after `main()` was done with `cout`.

You can also have threads sleep, but using a `mutex` and/or a condition variable (described in [“Condition Variables”](#)) is almost always a better solution. Similarly for `thread::yield` which is really just a convenience function for calling `sleep` with `elapsed_time(0)`.

In single thread mode, creating a thread is equivalent to a synchronous function call (though not nearly as efficient).

If you have multiple threads to create, you can create a `Metrowerks::thread_group`.

Listing 28.10 Example of `thread_group`

```
class thread_group
{
public:
    thread_group();
    ~thread_group();

    const thread* create_thread(const thread::func_type& f);
    void join_all();
};
```

The main feature of `thread_group` is that it makes it very easy to join with all of the threads.

Listing 28.11 Example of joining threads

```
int main()
{
    std::cout << "Main\n";
    Metrowerks::thread_group g;
```

```
g.create_thread(do_something);
g.create_thread(do_something_else);
g.join_all();
}
```

Condition Variables

A condition variable is a way for two threads to signal each other based on some predicate, such as a queue being empty or full. This is represented by `Metrowerks::condition`.

Listing 28.12 Metrowerks::condition class synopsis

```
class condition
{
public:
condition();
~condition();

void notify_one();
void notify_all();

template <typename ScopedLock> void wait(ScopedLock& lock);

template <typename ScopedLock, typename Predicate>
void wait(ScopedLock& lock, Predicate pred);

template <typename ScopedLock>
bool timed_wait(ScopedLock& lock,
const universal_time& unv_time);

template <typename ScopedLock, typename Predicate>
bool timed_wait(ScopedLock& lock,
const universal_time& unv_time, Predicate pred);

template <typename ScopedLock, typename Predicate>
bool timed_wait(ScopedLock& lock,
const elapsed_time& elps_time, Predicate pred);
};
```

Note that `condition` is not copyable nor assignable.

A condition allows one thread to pass a locked lock to the condition's wait function. The current thread then atomically unlocks the locks and goes to sleep. It will stay asleep until another thread calls this condition's `notify_one()` or `notify_all()` member function. The original thread will then atomically awake and lock the lock.

The difference between `notify_one` and `notify_all` is that the former notifies only one thread waiting on the condition, whereas the latter notifies all threads waiting on the condition.

When using the variation of the wait function without the predicate, it is important that you recheck the predicate (data) you were waiting for when the wait returns. You can not assume that whatever it is that you were wanting to be true is now true. This is most easily done by calling the wait within a while loop:

```
Metrowerks::condition cond;  
...  
Metrowerks::mutex::scoped_lock lock(some_mutex);  
while (I_need_more_data)  
    cond.wait(lock);
```

It is up to some other thread to make `I_need_more_data` false, and it will likely need to lock `some_mutex` in order to do it. When it does, it should execute one of:

```
cond.notify_one();  
or  
cond.notify_all();
```

It must also unlock `some_mutex` to allow the other thread's wait to return. But it does not matter whether `some_mutex` gets unlocked before or after the notification call. Once the original wakes from the wait, then the signal is satisfied. Should it wait again, then another thread will have to renotify it.

If it is more convenient, you can pass a predicate to the wait function, which will then do the while loop for you. Note that there are also several timed waits if you want to limit the sleep time (which can be thought of as an additional "condition" on the system clock).

[“Example of condition usage”](#) is a full example of condition usage. One thread puts stuff into a queue while another thread reads stuff back out of the other end.

Listing 28.13 Example of condition usage

```
#include <iostream>  
#include <queue>
```

```
#include <msl_thread>

class unbounded_queue
{
public:
    typedef Metrowerks::mutex Mutex;
    typedef Mutex::scoped_lock Lock;

    void send (int m);
    int receive();

private:
    std::queue<int> the_queue_;
    Metrowerks::condition queue_is_empty_so_;
    Mutex mut_;
};

void
unbounded_queue::send (int m)
{
    Lock lock(mut_);
    the_queue_.push(m);
    std::cout << "sent: " << m << '\n'
    if (the_queue_.size() == 1)
        queue_is_empty_so_.notify_one();
}

int
unbounded_queue::receive()
{
    Lock lock(mut_);
    while (the_queue_.empty())
        queue_is_empty_so_.wait(lock);
    int i = the_queue_.front();
    std::cout << "received: " << i << '\n'
    the_queue_.pop();
    return i;
}

unbounded_queue buf;

void sender()
{
    int n = 0;
    while (n < 1000)
```

```
{  
    buf.send(n);  
    ++n;  
}  
buf.send(-1);  
}  
  
void receiver()  
{  
    int n;  
    do  
    {  
        n = buf.receive();  
    } while (n >= 0);  
}  
  
int main()  
{  
    Metrowerks::thread send(sender);  
    Metrowerks::thread receive(receiver);  
    send.join();  
    receive.join();  
}
```

In the above example one thread continually sends data to a `std::queue`, while another thread reads data out of the queue. The reader thread must wait if the queue is empty, and the sender thread must notify the reader thread (to wake up) if the queue changes from empty to non-empty.

An interesting exercise is to transform the above example into a "bounded queue". That is, there is nothing from stopping the above example's queue from sending all of the data before the receiver thread wakes up and starts consuming it. ["Example of queue limitation"](#) is an example if you wanted to limit the above queue to a certain number of elements (like 20).

Listing 28.14 Example of queue limitation

```
#include <iostream>  
#include <cdeque>  
#include <msl_thread>  
  
class bounded_queue  
{  
public:  
    typedef Metrowerks::mutex Mutex;
```

Metrowerks::threads

Overview of Metrowerks Threads Library

```
typedef Mutex::scoped_lock Lock;
typedef Metrowerks::cdeque<int> Queue;

bounded_queue(int max) {the_queue_.reserve((unsigned)max);}

void send (int m);
int receive();

private:
    Queue the_queue_;
    Metrowerks::condition queue_is_empty_so_;
    Metrowerks::condition queue_is_full_so_;
    Mutex mut_;
};

template <class C>
struct container_not_full
{
    container_not_full(const C& c) : c_(c) {}
    bool operator()() const {return c_.size() != c_.capacity();}
private:
    const C& c_;
};

template <class C>
struct container_not_empty
{
    container_not_empty(const C& c) : c_(c) {}
    bool operator()() const {return !c_.empty();}
private:
    const C& c_;
};

void
bounded_queue::send (int m)
{
    Lock lock(mut_);
    queue_is_full_so_.wait(lock,
        container_not_full<Queue>(the_queue_));
    the_queue_.push_back(m);
    std::cout << "sent: " << m << '\n';

    if (the_queue_.size() == 1)
        queue_is_empty_so_.notify_one();
}
```

```
int
bounded_queue::receive()
{
    Lock lock(mut_);
    queue_is_empty_so_.wait(lock,
        container_not_empty<Queue>(the_queue_));
    int i = the_queue_.front();
    std::cout << "received: " << i << '\n'

    if (the_queue_.size() == the_queue_.capacity())
        queue_is_full_so_.notify_one();
    the_queue_.pop_front();
    return i;
}

bounded_queue buf(20);

void sender()
{
    int n = 0;
    while (n < 1000)
    {
        buf.send(n);
        ++n;
    }
    buf.send(-1);
}

void receiver()
{
    int n;
    do
    {
        n = buf.receive();
    } while (n >= 0);
}

int main()
{
    Metrowerks::thread send(sender);
    Metrowerks::thread receive(receiver);
    send.join();
    receive.join();
}
```

The above example actually demonstrates more than was advertised. Not only does it limit the queue length to 20, it also introduces a non-std container (`Metrowerks::cdeque`) which easily enables the monitoring of maximum queue length. It also demonstrates how more than one condition can be associated with a mutex. And furthermore, it uses the predicate versions of the wait statements so that explicit while loops are not necessary for the waits. Note that the predicates are negated: the wait will loop until the predicate is true.

Condition variables are fairly dangerous in single threaded code. They will compile and do nothing. But note that you may loop forever waiting for a predicate that won't change:

```
while (the_queue.empty())
    queue_not_empty.wait(lk);
```

If `the_queue.empty()` is true then this is just an infinite loop in single thread mode. There is no other thread that is going to make the predicate false.

call_once

Every once in a while, you need to make sure a function is called exactly once. This is useful for initialization code for example. The concept is similar to a local static, but local statics are not thread safe. It is possible two threads might try to construct a local static at once, before the initialization flag gets set.

Listing 28.15 Example two threads constructing a static

```
Metrowerks::mutex&
get_mutex()
{
    static Metrowerks::mutex mut; // ??!!!!
    return mut;
}
```

If more than one thread can call `get_mutex()` for the first time, at the same time, then it is possible that two threads may try to construct `mut` (and this would be bad). There are a couple of ways to deal with this problem.

You could make `mut` a global. But that may give you an undefined order of construction among global objects that is unacceptable for your application's start up code.

You could call `get_mutex()` once before you create any threads:

```
int main()
{
    get_mutex(); // just initialize the local static
}
```

Now it is safe to call `get_mutex()` from multiple threads as the construction step is already done.

Simple, but a little ugly. And you may not have control over `main` (what if you're writing a library?).

Enter `Metrowerks::call_once`. You can use `call_once` to ensure that only one thread calls `get_mutex` for the first time. The prototype for `call_once` looks like:

```
void call_once(void (*func)(), once_flag& flag);
```

`Metrowerks::once_flag` is the type of flag that you must initialize (at link time) to the macro: `_MSL_THREAD_ONCE_INIT`.

If `call_once` is called with such a flag, it will atomically execute the function, and set the flag to some other value. All other threads attempting to call `call_once` will block until the first call returns. Later threads calling into `call_once` with the same flag will return without doing anything. Here is how you could use it to "initialize" `get_mutex()`.

Listing 28.16 Example of initializing using `get_mutex()`

```
Metrowerks::mutex&
get_mutex_Impl()
{
    static Metrowerks::mutex mut;
    return mut;
}

void init_get_mutex()
{
    get_mutex_Impl();
}

Metrowerks::once_flag init_get_mutex_flag = _MSL_THREAD_ONCE_INIT;

Metrowerks::mutex&
get_mutex()
{
    Metrowerks::call_once(init_get_mutex, init_get_mutex_flag);
    return get_mutex_Impl();
```

}

The first thread into `get_mutex` will also go into `call_once` while blocking other threads from getting past that point. It then constructs the static mutex at its leisure. Once it returns, then threads can have unfettered access to the fully constructed static mutex.

`call_once` works identically in single thread mode.

thread_specific_ptr

This is a way to create "thread specific data". For example, you could create a "global" variable that is global to all functions, but local to each thread that access it. For example, `errno` is often implemented this way.

`Metrowerks::thread_specific_ptr` is a templated smart pointer that you can pass a new pointer to. It will associate that pointer with whatever thread passed it in (via its reset function). Other threads won't see that pointer. They will see `NULL` until they pass in their own heap-based data. The smart pointer will take care of releasing the heap data when the thread exits.

Listing 28.17 Class `thread_specific_ptr` synopsis

```
template <typename T>
class thread_specific_ptr
{
public:
    thread_specific_ptr();
    ~thread_specific_ptr();

    T* get() const;
    T* operator->() const {return get();}
    T& operator*() const {return *get();}
    T* release();
    void reset(T* p = 0);
};
```

You can have as many `thread_specific_ptr`'s as you want, and pointing to whatever type you desire. The `thread_specific_ptr` is not copyable or assignable, but you can assign a pointer to it.

Listing 28.18 Example of assigning a pointer

```
thread_specific_ptr<int> my_data;  
...  
my_data.reset(new int(3));
```

From then on, the thread that called reset can access that data like:

```
std::cout << *my_data;  
*my_data = 4;  
// etc.
```

You can release the memory with `my_data.release()`. This transfers pointer ownership back to you, so you must then delete the pointer. But you need not call `release` just to prevent memory leaks. `thread_specific_ptr` will automatically delete its data. And you can put in a new pointer by calling `reset` again. `thread_specific_ptr` will make sure the original pointer gets properly deleted. Do not use the array form of `new` with `thread_specific_ptr`. It will be using `delete` to free your pointer.

Listing 28.19 Example of freeing a pointer

```
#include <iostream>  
#include <msl_thread>  
  
Metrowerks::thread_specific_ptr<int> value;  
  
void increment()  
{  
    ++*value;  
}  
  
Metrowerks::mutex cout_mutex;  
  
void thread_proc()  
{  
    value.reset(new int(0));  
    for (int i = 0; i < 1000; ++i)  
        increment();  
    Metrowerks::mutex::scoped_lock lock(cout_mutex);  
    std::cout << *value << '\n';  
}  
  
int main()
```

Metrowerks::threads

Overview of Metrowerks Threads Library

```
{  
    Metrowerks::thread_group threads;  
    for (int i = 0; i < 5; ++i)  
        threads.create_thread(&thread_proc);  
    thread_proc();  
    threads.join_all();  
}
```

Should print out

```
1000  
1000  
1000  
1000  
1000  
1000
```

Once for main, and once for the five threads. Note how no locking is necessary in accessing the "global" `thread_specific_ptr`. It is as if each thread has its own local copy of this global.

Mslconfig

The MSL header <mslconfig> contains a description of the macros and defines that are used as switches or flags in the MSL C++ library.

C++ Switches, Flags and Defines

The MSL C++ library has various flags that may be set to customize the library to users specifications these include:

- “[CSTD](#)”
- “[Inhibit Container Optimization](#)”
- “[Inhibit Optimize RB bit](#)”
- “[MSL_DEBUG](#)”
- “[msl_error](#)”
- “[MSL_ARRAY_AUTO_PTR](#)”
- “[MSL_CFILE_STREAM](#)”
- “[MSL_CPP](#)”
- “[MSL_EXTENDED_BINDERS](#)”
- “[MSL_EXTENDED_PRECISION_OUTP](#)”
- “[MSL_FORCE_ENABLE_BOOL_SUPPORT](#)”
- “[MSL_FORCE_ENUMS_ALWAYS_INT](#)”
- “[MSL_IMP_EXP](#)”
- “[MSL_LONGLONG_SUPPORT](#)”
- “[MSL_MINIMUM_NAMED_LOCALE](#)”
- “[MSL_MULTITHREAD](#)”
- “[MSL_NO_BOOL](#)”
- “[MSL_NO_CONSOLE_IO](#)”
- “[MSL_NO_CPP_NAMESPACE](#)”

- “[MSL_NO_EXCEPTIONS](#)”
- “[MSL_NO_EXPLICIT_FUNC_TEMPLATE_ARG](#)”
- “[MSL_NO_FILE_IO](#)”
- “[MSL_NO_IO](#)”
- “[MSL_NO_LOCALE](#)”
- “[MSL_NO_REFCOUNT_STRING](#)”
- “[MSL_NO_VECTOR_BOOL](#)”
- “[MSL_NO_WCHART](#)”
- “[MSL_NO_WCHART_LANG_SUPPORT](#)”
- “[MSL_NO_WCHART_C_SUPPORT](#)”
- “[MSL_NO_WCHART_CPP_SUPPORT](#)”
- “[MSL_POSIX_STREAM](#)”
- “[MSL_WIDE_FILENAME](#)”
- “[MSL_WFILEIO_AVAILABLE](#)”
- “[MSL_USE_AUTO_PTR_96](#)”
- “[STD](#)”

CSTD

The `_CSTD` macro evaluates to `::std` if the MSL C library is compiled in the `std` namespace, and to nothing if the MSL C library is compiled in the global namespace.

`_STD` and `_CSTD` are meant to prefix C++ and C objects in such a way that you don't have to care whether or not the object is in `std` or not. For example:

`_STD::cout`, or `_CSTD::size_t`.

Inhibit Container Optimization

If this flag is defined it will disable pointer specializations in the containers. This may make debugging easier.

You must recompile the C++ lib when flipping this switch.

_Inhibit_Optimize_RB_bit

Normally the red/black tree used to implement the associative containers has a space optimization that compacts the red/black flag with the parent pointer in each node (saving one word per entry). By defining this flag, the optimization is turned off, and the red/black flag will be stored as an enum in each node of the tree.

_MSL_DEBUG

This switch when enabled and the library rebuilt will put MSL Standard C++ library into debug mode. For full information see “[Overview of MSL C++ Debug Mode](#)”.

You must recompile the C++ lib when flipping this switch.

__msl_error

This feature is included for those wishing to use the C++ lib with exceptions turned off. In the past, with exceptions turned off, the lib would call fprintf and abort upon an exceptional condition. Now you can configure what will happen in such a case by filling out the definition of `__msl_error()`. Two example definitions are given. One example will call fprintf and abort, the other will do nothing at all.

_MSL_ARRAY_AUTO_PTR

When defined auto_ptr can be used to hold pointers to memory obtained with the array form of new. The syntax looks like:

```
auto_ptr<string, _Array<string> >  
pString(new string[3]);  
pString.get()[0] = "pear";  
pString.get()[1] = "peach";  
pString.get()[2] = "apple";
```

Without the _Array tag, auto_ptr behaves in a standard fashion. This extension to the standard is not quite conforming as it can be detected through the use of template template arguments.

This extension can be disabled by not defining _MSL_ARRAY_AUTO_PTR.

_MSL_CFILE_STREAM

Set when the file system does not support wide character streams.

_MSL_CPP

Evaluates to an integer value which represents the C++ lib's current version number.

This value is best when read in hexadecimal format

_MSL_EXTENDED_BINDERS

Defining this flag adds defaulted template parameters to binder1st and binder2nd. This allows client code to alter the type of the value that is stored. This is especially useful

when you want the binder to store the value by const reference instead of by value to save on an expensive copy construction.

Listing 29.1 For example:

```
#include <string>
#include <functional>
#include <algorithm>

struct A
{
public:
    A(int data = 0) : data_(data) {}
    friend bool operator < (const A& x, const A& y) {return x < y;}
private:
    int data_;
    A(const A&);
};

int main()
{
using namespace std;
A a[5];
A* i = find_if(a, a+5, binder2nd<less<A>>(less<A>(), A(5)));
}
```

This causes the compile-time error, because binder2nd is attempting to store a copy of A(5). But with `_MSL_EXTENDED_BINDERS` you can request that binder2nd store a const A& to A(5).

```
A* i = find_if(a, a+5,
    binder2nd<less<A>, const A&>(less<A>(), A(5)));
```

This may be valuable when A is expensive to copy.

This also allows for the use of polymorphic operators by specifying reference types for the operator.

This extension to the standard is detectable with template template parameters so it can be disabled by not defining `_MSL_EXTENDED_BINDERS`.

_MSL_EXTENDED_PRECISION_OUTP

When defined this allows the output of floating point output to be printed with precision greater than DECIMAL_DIG. With this option, an exact binary to decimal conversion can be performed (by bumping precision high enough).

The cost is about 5-6Kb in code size.

You must recompile the C++ lib when flipping this switch.

_MSL_FORCE_ENABLE_BOOL_SUPPORT

This tri-state flag has the following properties

- If not defined, then the C++ library and headers will react to the settings in the language preferences panel (as in the past).
- If the flag is set to zero, then the C++ lib/header will force “Enable bool support” to be off while processing the header (and then reset at the end of the header).
- If the flag is set to one, then the C++ library and header will force “Enable bool support” to be on while processing the header (and then reset at the end of the header).

If _MSL_FORCE_ENABLE_BOOL_SUPPORT is defined, the C++ library will internally ignore the “Enable bool support” setting in the application's language preferences panel, despite the fact that most of the C++ library is compiled into the application (since it is in headers) instead of into the binary C++ library.

The purpose of this flag is (when defined) to avoid having to recompile the C++ library when “Enable bool” support is changed in the language preferences panel.

With _MSL_FORCE_ENABLE_BOOL_SUPPORT defined to one, std::methods will continue to have a real bool in their signature, even when bool support is turned off in the application. But the user won't be able to form a bool (or a true/false). The user won't be able to:

```
bool b = std::ios_base::sync_with_stdio(false);  
// error: undefined bool and false
```

but this will work:

```
unsigned char b =  
std::ios_base::sync_with_stdio(0);
```

And the C++ lib will link instead of getting the ctype link error.

Changing this flag will require a recompile of the C++ library.

_MSL_FORCE_ENUMS_ALWAYS_INT

This tri-state flag has the following properties

- If not defined, then the C++ library and headers will react to the settings in the language preferences panel (as in the past).
- If the flag is set to 0, then the C++ lib/header will force “Enums always int” to be off while processing the header (and then reset at the end of the header).
- If the flag is set to 1, then the C++ library and header will force “Enums always int” to be on while processing the header (and then reset at the end of the header).

If `_MSL_FORCE_ENUMS_ALWAYS_INT` is defined, the C++ library will internally ignore the “Enums always int” setting in the application’s language preferences, despite the fact that most of the C++ library is compiled into the application (since it is in headers) instead of into the binary C++ library.

The purpose of this flag is (when defined) to avoid having to recompile the C++ lib when “Enums always int” is changed in the language preferences panel.

For example, with `_MSL_FORCE_ENUMS_ALWAYS_INT` defined to zero, and if the user turns “enums always int” on in his language preferences panel, then any enums the user creates himself will have an underlying `int` type.

This can be exposed by printing out the `sizeof(the enum)` which will be four. However if the user prints out the `sizeof(a std::enum)`, then the size will be one(because all `std::enums` fit into 8 bits) despite the `enums_always_int` setting.

Changing this flags will require a recompile of the C++ library.

_MSL_IMP_EXP

The C, C++, SIOUX and runtime shared libraries have all been combined into one shared library locating under the appropriate OS support folder.

The exports files (.exp) have been removed. The prototypes of objects exported by the shared lib are decorated with a macro:

`_MSL_IMP_EXP_xxx`

or

`_MSL_IMP_EXP_xxx`

//where xxx is the library designation.

which can be defined to `__declspec(dllimport)`.

This replaces the functionality of the.exp/.def files. Additionally, the C, C++, SIOUX and runtimes can be imported separately by defining the following 4 macros differently:

`_MSL_IMP_EXP_C`

`_MSL_IMP_EXP_CPP`

`_MSL_IMP_EXP_SIOUX`

`_MSL_IMP_EXP_RUNTIME`

Define these macros to nothing if you don't want to import from the associated lib, otherwise they will pick up the definition of `_MSL_IMP_EXP`.

There is a header <UseDLLPrefix.h> that can be used as a prefix file to ease the use of the shared lib. It is set up to import all 4 sections.

There is a problem with non-const static data members of templated classes when used in a shared lib. Unfortunately <locale> is full of such objects.

Therefore you should also define `_MSL_NO_LOCALE` which turns off locale support when using the C++ lib as a shared lib. This is done for you in <UseDLLPrefix.h>. See "[“_MSL_NO_LOCALE”](#) for more details.

_MSL_LONGLONG_SUPPORT

When defined, C++ supports long long and unsigned long long integral types.
Recompile the C++ lib when flipping this switch.

_MSL_MINIMUM_NAMED_LOCALE

When defined, turns off all of the named locale stuff except for "C" and "" (which will be the same as "C"). This reduces both lib size and functionality, but only if you are already using named locales. If your code does not explicitly use named locales, this flag has no effect.

_MSL_MULTITHREAD

The thread safety of MSL C++ can be controlled by the flag `_MSL_MULTITHREAD`.

If you explicitly use `std::mutex` objects in your code, then they will become empty do-nothing objects when multi-threading is turned off (`_MSL_MULTITHREAD` is undefined). Thus the same source can be used in both single thread and multi-thread projects.

The `_MSL_MULTITHREAD` flag causes some mutex objects to be set up in the library internally to protect data that is not obviously shared. For example, `std::basic_string` is refcounted. It is possible that two threads might each have their own `basic_string`, and that `basic_string` might share data among threads under the covers via the refcount mechanism. Therefore `basic_string` protects its refcount with a mutex object so that client code (even multi-threaded client code) can not detect that a refcounting implementation is in use.

See "[Multi-Thread Safety](#)" for a full description of Metrowerks Standard Library multi-threading safety policy.

_MSL_NO_BOOL

If defined then `bool` will not be treated as a built-in type by the library. Instead it will be a `typedef` to `unsigned char` (with suitable values for true and false as well). If `_MSL_FORCE_ENABLE_BOOL_SUPPORT` is not defined then this flag will set itself according to the "Enable bool support" switch in the language preferences panel.

The C++ lib must be recompiled when flipping this switch.

When `_MSL_NO_BOOL` is defined, `vector<bool>` will really be a `vector<unsigned char>`, thus it will take up more space and not have flip methods. Also there will not be any traits specializations for `bool` (i.e. `numeric_limits`).

_MSL_NO_CONSOLE_IO

This flag allows you to turn off console support while keeping memory mapped streams (`stringstream`) functional.

See Also

[“_MSL_NO_FILE_IO”](#)

_MSL_NO_CPP_NAMESPACE

If defined then the C++ lib will be defined in the global namespace.

You must recompile the C++ lib when flipping this switch.

_MSL_NO_EXCEPTIONS

If defined then the C++ lib will not throw an exception in an exceptional condition. Instead `void __msl_error(const char*)`; will be called. You may edit this inline in `<mslconfig>` to do whatever is desired. Sample implementations of `__msl_error` are provided in `<mslconfig>`.

Remarks

The operator new (which is in the runtime libraries) is not affected by this flag.

This flag detects the language preferences panel “Enable C++ exceptions” and defines itself if this option is not on.

The C++ lib must be recompiled when changing this flag (including if the language preference panel is changed).

_MSL_NO_EXPLICIT_FUNC_TEMPLATE_ARG

When defined assumes that the compiler does not support calling function templates with explicit template arguments.

On Windows, when ARM Conformance is selected in the language preferences panel, then this switch is automatically turned on. The Windows compiler goes into a MS compatible mode with ARM on.

This mode does not support explicit function template arguments.

In this mode, the signatures of has_facet and use_facet change.

You must recompile the C++ lib when flipping this switch.

Listing 29.2 Example of _MSL_NO_EXPLICIT_FUNC_TEMPLATE_ARG usage:

Standard setting:

```
template <class Facet>
    const Facet& use_facet(const locale& loc);
template <class Facet>
    bool has_facet(const locale& loc) throw();
```

_MSL_NO_EXPLICIT_FUNC_TEMPLATE_ARG setting.

```
template <class Facet>
    const Facet& use_facet(const locale& loc, Facet* );
template <class Facet>
    bool has_facet(const locale& loc, Facet* ) throw();
```

_MSL_NO_FILE_IO

This flag allows you to turn off file support while keeping memory mapped streams (stringstream) functional.

See Also

[“_MSL_NO_CONSOLE_IO”](#)

_MSL_NO_IO

If this flag is defined the C++ will not support any I/O (not even stringstream).

_MSL_NO_LOCALE

When this flag is defined, locale support is stripped from the library. This has tremendous code size benefits.

All C++ I/O will implicitly use the “C” locale. You may not create locales or facets, and you may not call the imbue method on a stream. But otherwise all streams are completely functional.

The C++ lib must be recompiled when flipping this switch.

_MSL_NO_REFCOUNT_STRING

The flag `_MSL_NO_REFCOUNT_STRING` is deprecated and will have no effect (it is harmless). This rewrite has higher performance and lower code size compared to previous releases. The string class is insensitive to the setting of `_MSL_MULTITHREAD`.

_MSL_NO_VECTOR_BOOL

If this flag is defined it will disable the standard `vector<bool>` partial specialization. You can still instantiate `vector<bool>`, but it will not have the space optimization of one `bool` per bit.

There is no need to recompile the C++ lib when flipping this switch, but you should remake any precompiled headers you might be using.

_MSL_NO_WCHART

This flag has been replaced by three new flags,

[" _MSL_NO_WCHART_LANG_SUPPORT"](#).

[" _MSL_NO_WCHART_C_SUPPORT"](#). and

[" _MSL_NO_WCHART_CPP_SUPPORT"](#).

_MSL_NO_WCHART_LANG_SUPPORT

This flag is set if the compiler does not recognize wchar_t as a separate data type (no wchar_t support in the language preference panel). The C++ lib will still continue to support wide character functions. wchar_t will be typedef'd to another built-in type.

The C++ library must be recompiled when turning this switch on (but need not be recompiled when turning it off).

_MSL_NO_WCHART_C_SUPPORT

This flag is set if the underlying C lib does not support wide character functions. This should not be set when using MSL C.

The C++ library must be recompiled when turning this switch on (but need not be recompiled when turning it off).

_MSL_NO_WCHART_CPP_SUPPORT

This flag can be set if wide character support is not desired in the C++ lib. Setting this flag can cut the size of the I/O part of the C++ lib in half.

The C++ library must be recompiled when turning this switch on (but need not be recompiled when turning it off).

_MSL_POSIX_STREAM

Set when a POSIX based library is being used as the underlying C runtime library.

_MSL_WIDE_FILENAME

If the flag `_MSL_WIDE_FILENAME` is defined, then the file stream classes support wide character filenames (null terminated arrays of `const wchart_t*`). Each stream class has an overloaded constructor, and an overloaded open member taking the `const wchar_t`. If the underlying system supports wide filenames, MSL C++ will pass the `wchar_t` straight through without any locale encoding.

Thus the interpretation of the wide filename is done by the OS, not by the C++ library. If the underlying system does not support wide filenames, the open will fail at runtime.

By default `_MSL_WIDE_FILENAME` is not defined as these signatures are not standard.

Turning on this flag does not require a recompile of MSL C++.

When MSL C is not being used as the underlying C library, and when the file stream is implemented in terms of `FILE*` (see “[_MSL_CFILE_STREAM](#)”), the system is said to not support wide filenames and the open will fail at runtime.

For example, wide filenames are not supported when using the BSD C library on Apple's Mach-O platform.

When using Posix as the underlying implementation (see “[_MSL_POSIX_STREAM](#)”), wide filenames are supported if the Posix library comes from the MSL Extras Library (in which case the [“_MSL_WFILEIO_AVAILABLE”](#) flag must be on). Wide filenames are also supported if using the BSD Posix on Apple's Mach-O platform.

_MSL_WFILEIO_AVAILABLE

Set when a wide character file name is available for a file name.

_MSL_USE_AUTO_PTR_96

Defining this flag will disable the standard auto_ptr and enable the version of auto_ptr that appeared in the Dec.'96 CD2.

_STD

This macro evaluates to `::std` if the C++ lib is compiled in the std namespace, and to nothing if the C++ lib is compiled in the global namespace.

See Also

[“CSTD”](#)

Mslconfig

C++ Switches, Flags and Defines

Index

map 261

Symbols

<cassert> 51
<cerrno> 51
<functional
 negate> 61
 __MSL_CPP__ 718
 __msl_error 717
 __MSL_LONGLONG_SUPPORT__ 722
 _CSTD 716
 _Inhibit_Container_Optimization 716
 _Inhibit_Optimize_RB_bit 717
 __MSL_ARRAY_AUTO_PTR 718
 __MSL_CFILE_STREAM 718
 __MSL_CX_LIMITED_RANGE 391
 __MSL_DEBUG 717
 __MSL_EXTENDED_BINDERS 718
 __MSL_EXTENDED_PRECISION_OUTP 720
 __MSL_FORCE_ENABLE_BOOL_SUPPORT 720
 __MSL_FORCE_ENUMS_ALWAYS_INT 721
 __MSL_IMP_EXP 722
 __MSL_IMP_EXP_C 722
 __MSL_IMP_EXP_CPP 722
 __MSL_IMP_EXP_RUNTIME 722
 __MSL_IMP_EXP_SIOUX 722
 __MSL_MINIMUM_NAMED_LOCALE 723
 __MSL_NO_BOOL 723
 __MSL_NO_CONSOLE_IO 724
 __MSL_NO_CPP_NAMESPACE 724
 __MSL_NO_EXCEPTIONS 724
 __MSL_NO_EXPLICIT_FUNC_TEMPLATE_ARG 72
 5
 __MSL_NO_FILE_IO 725
 __MSL_NO_IO 726
 __MSL_NO_LOCALE 726
 __MSL_NO_REFCOUNT_STRING 726
 __MSL_NO_VECTOR_BOOL 726
 __MSL_NO_WCHART 727
 __MSL_NO_WCHART_C_SUPPORT 727
 __MSL_NO_WCHART_CPP_SUPPORT 727
 __MSL_NO_WCHART_LANG_SUPPORT 727

__MSL_POSIX_STREAM 728
__MSL_USE_AUTO_PTR_96 729
__MSL_WFILEIO_AVAILABLE 728
__STD 729

A

Abnormal Termination 43
abort
 Numeric_limits 30
abs 402
access 354
 valarray 354
Accumulate 383
Adaptors for Pointers to Functions 68
Adaptors for pointers to functions
 Functional 68
Adaptors for Pointers to Members 69
Adaptors for pointers to members
 Functional 69
address 74
adjacent_difference 386
Adjacent_find
 algorithm 305
Advance 284
Algorithm 301
adjacent_find 305
binary_search 331
copy 310
copy_backward 310
count 305
count_if 306
equal 307
equal_range 330
fill 314
fill_n 315
find 302
find_end 303
find_first_of 304
find_if 302
for_each 302
generate 315
generate_n 316
includes 334
inplace_merge 333

```
iter_swap 311
lexicographical_compare 345
lower_bound 328
make_heap 340
max 342
max_element 344
merge 332
min 341
min_element 343
mismatch 306
next_permutation 346
nth_element 327
partial_sort 325
partial_sort_copy 326
partition 321
pop_heap 339
prev_permutation 347
push_heap 339
random_shuffle 321
remove 316
remove_copy 317
remove_copy_if 317
remove_if 316
replace 313
replace_copy_if 314
reverse 319
reverse_copy 319
rotate 320
rotate_copy 320
search 308
search_n 309
set_difference 337
set_intersection 336
set_symetric_difference 338
set_union 335
sort 323
sort_heap 341
stable_partition 322
stable_sort 324
swap 310
swap_ranges 311
transform 312
unique 318
unique_copy 318
upper_bound 329
Algorithms Library 301–348
allocate 75
allocator globals 76
allocator members 74
Allocator requirements 54
Always_noconv
    codecvt 157
Any
    bitset 276
Apply
    valarray 360
Arbitrary-Positional Stream 6
arg 402
Arithmetic operations
    Functional 59
assert.h 51
Assertions 51
Assign
    deque 245
    list 248
    vector 258
assign 90
Assignment Operator
    bad_alloc 35
    bad_cast 39
    bad_typeid 39
    complex 393
    type_info 38
Assignment operator
    auto_ptr 84
    bad_exception 42
    exception 40
    gslice_array 377
    mask_array 379
    slice_array 374, 375
Assignment operators
    indirect_array 381
Associative Containers 260
Associative Containers Requirements 243
atexit
    Numeric_limits 31
auto_ptr 86
    destructor 84
Auto_ptr 81, 86
    Assignment operator 84
    Constructors 83
    Members 84
    Operator = 84
Auto_ptr conversions 86
Auto_ptr_ref 86
```

B

Back_insert_iterator
 back_inserter 291
 constructors 290
 operator = 290
 operators 290

Back_inserter
 back_insert_iterator 291

bad 454

bad_alloc
 assignment operator 35
 constructors 35
 destructor 35
 what 35

Bad_cast
 assignment operator 39
 constructor 38
 what 39

Bad_exception
 assignment operator 42
 constructor 41
 what 42

Bad_typeid
 assignment operator 39
 constructor 39
 what 40

Base
 reverse_iterator 286

Basic Iterator 284

basic_filebuf 570
 close 574
 constructors 570
 destructor 571
 imbue 577
 is_open 571
 open 572
 Open Modes 572
 overflow 575
 pbackfail 575
 seekoff 576
 seekpos 576
 setbuf 577
 showmany 574
 sync 577
 underflow 575

basic_fstream 589
 close 594
 constructor 590

 is_open 592
 open 593
 Open Modes 593
 rdbuf 591

 basic_ifstream 578
 close 583
 constructor 578
 is_open 581
 open 581
 Open Modes 582
 rdbuf 580

 basic_ios 440
 bad 454
 clear 449
 constructors 440
 copyfmt 446
 eof 451
 exceptions 455
 fail 453
 fill 445
 good 451
 imbue 444
 Operator ! 446
 Operator bool 446
 rdbuf 443
 rdstate 447
 setstate 450
 tie 441

 basic_iostream 517
 constructor 517
 destructor 518

 basic_istream 488
 constructors 488
 destructor 488
 extractors, arithmetic 491
 extractors, characters 492
 gcount 497
 get 499
 getline 501
 ignore 503
 peek 505
 putback 509
 read 505
 readsome 507
 seekg 514
 sentry 489
 sync 512
 tellg 513
 unget 510

```
ws 516
basic_istringstream 555
    constructors 556
    rdbuf 557
    str 558
basic_ofstream 583
    close 589
    constructors 584
    is_open 587
    open 587
    Open Modes 588
    rdbuf 585
basic_ostream 518
    constructor 519
    destructor 519
    endl 535
    ends 535
    flush 532
    flush,flush 536
    Inserters, arithmetic 522
    Inserters, characters 524
    put 530
    resetiosflags 539
    seekp 528
    sentry 520
    setbase 541
    setfill 542
    setiosflags 540
    setprecision 543
    setw 544
    tellp 527
    write 530
basic_ostringstream 559
    constructors 560
    rdbuf 561
    str 563
basic_streambuf 462
    constructors 463, 550
    destructor 463
    eback 477
    egptr 477
    eptr 479
    gbump 478
    getloc 464
    gptr 477
    imbue 480
    in_avail 470
    Locales 463
    overflow 486, 554
    pbackfail 485, 553
    pbase 479
    pbump 479
    pptr 479
    pubseekoff 466
    pubseekpos 467
    pubsetbuf 464
    pubsync 469
    pubuimbue 464
    sbumpc 471
    seekoff 481, 554
    seekpos 482, 555
    setbuf 481
    setg 478
    setp 480
    sgetc 472
    sgetn 473
    showmanc 483
    snextc 470
    sputback 473
    sputc 475
    sputn 476
    str 551
    sungetc 475
    sync 482
    uflow 484
    underflow 484, 553
    xsgetn 483
    xsputn 485
basic_string
    append 103
    assign 104
    assignment operator 98
    at 102
    begin 99
    c_str 109
    capacity 100, 101
    clear 101
    compare 116
    Constructors 96
    copy 108
    data 109
    destructor 97
    Element Access 102
    empty 101
    end 99
    erase 106
    extractor 124
    find 110
```

```
find_first_not_of 114, 115
find_first_of 112
find_last_of 113
get_allocator 109
getline 125
insert 105
inserter 125
Inserters and extractors 124
iterator support 98
max_size 100
Modifiers 102
Non-Member Functions and Operators 117
Null Terminated Sequence Utilites 126
operator 120, 122, 125
operator!= 119
operator+ 117
operator+= 102
operator== 118
operator> 121
operator>= 123
operator>> 124
rbegin 99
rend 99
replace 107
reserve 101
rfind 111
size 100
String Operations 109
substr 115
swap 108
basic_stringbuf 550
basic_stringstream 564
    constructors 565
    rdbuf 566
    str 567
before
    type_info 37
Bidirectional Iterators 282
Binary_function 58
Binary_negate 66
Binary_search
    algorithm 331
bind1st
    Functional 67
bind2nd 68
binder1st
    Functional 67
binder2nd
Functional 67
Binders
Functional 67
Bitset 270
    any 276
    constructors 271
    count 275
    flip 274
    none 277
    operator 277, 279
    operator != 272, 276
    operator & 278
    operator &= 271
    operator <=> 272
    operator == 276
    operator >> 277, 279
    operator >>= 272
    operator ^ 278
    operator ^= 272
    operator | 278
    Operator ~ 274
    reset 273
    set 273
    size 275
    test 276
    to_string 275
    to_ulong 274
bitvector 627
    assign 635
    at 637
    back 638
    begin 635
    capacity 634
    clear 640
    constructors 632
    Destructor 633
    empty 634
    end 636
    erase 639
    flip 640
    front 638
    get_allocator 633
    insert 639
    invariants 641
    iterators 631
    max_size 633
    Operator 641, 643
    Operator != 642
    Operator == 641
```

Operator > 642
Operator >= 642
pop_back 639
rbegin 636
reference class members 631
rend 636
reserve 635
resize 637
size 634
swap 640
types 630

Bitvector Class Library 627–643
boolalpha 456
Bsearch 347
Buffer management 464
Buffering 418

C

C Library Files 597–598
C Library Locales 240
C++ Library 5–17
Capacity
 vector 259
Category
 Locale 135
cerr 419
char_type 94
Character 6
character 90
Character Classification
 locale 141
character container type 90
Character Conversions
 locale 142
Character Sequences 6
Character Trait Definitions 90, 94
Character traits definitions 90
Class
 Back_insert_iterator 290
 basic_filebuf 570
 basic_fstream 589
 basic_ifstream 578
 basic_ios 440
 basic_iostream 517
 basic_istream 488
 sentry 489
 basic_istringstream 555
 basic_ofstream 583
 basic_ostream 518
 sentry 520
 basic_ostringstream 559
 basic_streambuf 462
 basic_stringbuf 550
 basic_stringstream 564
 Bitset 270
 complex 392
 Deque 244
 fpos 423
 Front_insert_iterator 291
 gslice 375
 gslice_array 377
 indirect_array 380
 Insert_iterator 293
 ios_base 424
 failure 425
 Init 428
 Istream_iterator 294
 Istreambuf_iterator 297
 list 247
 Map 261
 mask_array 378
 Multimap 264
 Multiset 268
 Ostream_iterator 296
 Ostreambuf_iterator 299
 Priority_queue 254
 Queue 253
 Reverse_iterator 285
 Set 267
 Stack 256
 Vector 257, 260
class
 Mutex_lock 624
Class Auto_ptr 81
Class bad_alloc 35
Class bad_cast 38
Class bad_typeid 39
Class Ctype
 locale 143
Class Ctype_byname 155
Class ctype_byname
 locale 148
Class exception 40
class mutex 621
Class slice 372

Class Slice_array 373
Class type_info 37
Classic
 locale 140
Classic_table
 ctype 154
Clear
 list 250
clear 449
clog 419
Close
 messages 233
close
 basic_filebuf 574
 basic_fstream 594
 basic_ifstream 583
 basic_ofstream 589
Cmath 387
Codecvt
 Virtual Functions 158
Codevtc
 always_noconv 157
 in 156
 length 157
 max_length 157
 out 156
 unshift 156
Collate
 compare 176
 hash 177
 member functions 176
 transform 176
 Virtual Functions 177
Collate Category 176
Combine
 locale 138
Compare
 collate 176
compare 91
Comparison Function 6
Comparisons
 Functional 61
complex 392
 abs 402
 arg 402
 conj 403
 constructor 393
 cos 404
 cosh 404
 exp 405
 imag 394, 402
 log 405
 log10 405
 norm 403
 operator 401
 operator - 398
 operator != 400
 operator * 399
 operator *= 395
 operator + 397
 operator += 394
 operator / 399
 operator /= 396
 operator -= 395
 operator = 393
 operator == 400
 operator >> 401
 polar 403
 pow 406
 real 394, 402
 sin 406
 sinh 407
 sqrt 407
 tan 407
 tanh 408
Complex Class Library 391–408
Component 6
Conforming Implementations 17
conj 403
const_mem_fun_ref_t 73
const_mem_fun_t
 template function 72
const_mem_fun1_ref_t
 template class 74
const_mem_fun1_t 73
Constraints on programs 15
Constructors
 insert_iterator 293
construct 75
Constructor
 list 247
 locale 137
 mutex_lock 624
Constructor,ctype_byname 148
Constructors 96
 Auto_ptr 83

back_insert_iterator 290
bad_alloc 35
bad_cast 38
bad_exception 41
bad_typeid 39
basic_filebuf 570
basic_fstream 590
basic_ifstream 578
basic_ios 440
basic_iostream 517
basic_istream 488
basic_istringstream 556
basic_ofstream 584
basic_ostream 519
basic_ostringstream 560
basic_streambuf 463, 550
basic_stringstream 565
bitset 271
ctype 153
deque 245
domain_error 48
exceptions 40
failure, ios_base 425
front_insert_iterator 291
gslice 376
gslice_array 377
indirect_array 381
invalid_argument 48
ios_base 439
istream_iterator 294
istreambuf_iterator 297
istrstream 609
length_error 49
logic_error 48
map 261
mask_array 379
multimap 265
multiset 269
ostream_iterator 296
ostreambuf_iterator 299
ostrstream 612
out_of_range 49
overflow_error 50
pair 57
priority_queue 255
range_error 50
raw_storage_iterator 77
reverse_iterator 286
runtime_error 49
sentry, basic_istream 490
sentry, basic_ostream 520
set 268
slice 372
slice_array 374
stack 256
strstream 617
strstreambuf 600
type_info 38
underflow_error 50
valarray 353
vector 258
Container adaptors 253
Container Requirements 241
Containers Library 241–279
Conversion Constructor 86
Copy
 algorithm 310
copy 92
Copy construction 54
Copy_backward
 algorithm 310
copyfmt 446
cos 404
cosh 404
Count
 algorithm 305
 bitset 275
Count_if
 algorithm 306
cout 418
Cshift
 valarray 359
cstdio
 Functions 597
 Macros 597
 Types 597
Cstdlib 387
Ctype
 classic_table 154
 constructors 153
 destructor 153
Ctype Category
 locale 143
Ctype Specializations
 locale 152
Ctype_byname,constructor 148

Curr_symbol	strstreambuf 601
moneypunct 222	Diagnostics Library 47–51
D	digits
Numeric_limits 21	Date and Time functions 87
Distance 285	date_order 186
divides	deallocate 75
functional 60	Debug Mode Implementations 662
do_date_order 188	dec 458
do_get_date 188	Decimal_point
do_get_monthname 188	moneypunct 221
do_get_time 188	numpunct 170
do_get_weekday 188	Default Behavior 7
do_get_year 188	Default construction 54
Do_is	Delete 33
locale 146	denorm_min
Do_narrow	Numeric_limits 26
locale 148	Deque 244
do_put_time_put 197	assign 245
Do_scan_is	constructors 245
locale 146	erase 246
Do_scan_not	insert 246
locale 147	resize 245
Do_tolower	swap 247
locale 147	destroy 76
Do_toupper	Destructor
locale 147	auto_ptr 84
Do_widen	bad_alloc 35
locale 147	ctype 153
Domain_error	exception 41
constructor 48	istream_iterator 295
Dynamic memory management	istrstream 610
32	mutex_lock 625
E	ostream_iterator 296
Eback	ostrstream 613
egptr	strstream 617
Empty	valarray 353
stack 256	Destructors
endl	basic_filebuf 571
ends	basic_ios 440
eof	basic_iostream 518
eptr	basic_istream 488
epsilon	basic_ostream 519
Numeric_limits 23	basic_streambuf 463
eq	Init, ios_base 428
eq_int_type	ios_base 439
Equal	sentry, basic_istream 490
	sentry, basic_ostream 521

- algorithm 307
- istreambuf_iterator 299
- Equal_range**
 - algorithm 330
 - map 263
 - multimap 266
- equal_to**
 - functional 61
- Equality Comparisons** 53
- Erase**
 - deque 246
 - list 250
 - vector 260
- errno.h 51
- Error numbers** 51
- Exception**
 - assignment operator 40
 - destructor 41
 - what 41
- Exception classes** 47
- Exception handling** 40
- Exceptions**
 - constructor 40
- exceptions**
 - basic_ios 455
- exit**
 - Numeric_limits 31
- exp** 405
- External "C" Linkage** 15
- Extractors**
 - basic_istream, arithmetic 491
 - basic_istream, characters 492
 - overloading 495

- F**
- fail** 453
- Failed**
 - ostreambuf_iterator 300
- Falsename**
 - numpunct 171
- File Based Streams** 569–595
- Fill**
 - algorithm 314
- fill** 445
- Fill_n**
 - algorithm 315
- Find**
 - algorithm 302
 - map 262
 - multimap 265
 - find 92
 - Find_end**
 - algorithm 303
 - Find_first_of**
 - algorithm 304
 - Find_if**
 - algorithm 302
 - fixed 458
 - flags 429
 - Flip**
 - bitset 274
 - float_denorm_style**
 - Numeric_limits 28
 - float_round_style**
 - Numeric_limits 28
 - flush 532
 - fmtflags 425
 - For_each**
 - algorithm 302
 - Formatting and Manipulators** 487–547
 - Forward Declarations** 413–415
 - Forward Iterators** 282
 - Fpos** 423
 - Frac_digits**
 - moneypunct 222
 - Freestanding Implementations** 14
 - freeze**
 - osstringstream 613
 - stringstream 617
 - strstreambuf 602
 - Front_insert_iterator**
 - constructor 291
 - front_inserter 292
 - operator = 291
 - operators 292
 - Front_inserter**
 - front_insert_iterator 292
 - fstream** 569
 - Functional** 58
 - Adaptors for pointers to functions 68
 - Adaptors for pointers to members 69
 - Arithmetic operations 59
 - bind1st 67
 - bind2nd 68

```
binder1st 67
binder2nd 67
Binders 67
Comparisons 61
Logical operations 64
mem_fun_t 69
mem_fun1_t 70
Negators 65
pointer_to_binary_function 68, 69
pointer_to_unary_function 68
functional
    divides 60
    equal_to 61
    greater 62
    greater_equal 63
    less 63
    less_equal 64
    logical_and 64
    logical_not 65
    logical_or 65
    minus 59
    modulus 60
    multiplies 60
    not_equal_to 62
    plus 59
getloc
    basic_streambuf 464
    ios_base 437
Global
    locale 140
good 451
gptr 477
greater
    functional 62
greater_equal
    functional 63
Grouping
    moneypunct 221
    numpunct 170
Gslice 375
    constructors 376
    size 376
    start 376
    stride 377
Gslice_array 377
    assignment operations 378
    assignment operator 377
    constructors 377
    fill operator 378
```

G

```
gbump 478
gcount 497
General Utilites Library 53–87
Generate
    algorithm 315
Generate_n
    algorithm 316
Get
    messages 233
    money_get 217
    num_get 164
get 85, 499
get_date 187
get_monthname 187
get_state 93
get_temporary_buffer 79
get_time 186
get_weekday 187
get_year 187
getline 501
```

H

```
Handler Function 7
has_denorm
    Numeric_limits 25
has_denorm_loss
    Numeric_limits 25
Has_facet
    locale 141
has_infinity
    Numeric_limits 24
has_quiet_NaN
    Numeric_limits 24
has_signaling_NaN
    Numeric_limits 24
Hash
    collate 177
    count 685
    equal_range 685
    erase 684
    find 685
    insert 682
    operator != 686
```

```
operator == 686
swap 685
Hash Libraries 675–693
Headers 94
    algorithm 301
    cmath 387
    cstdlib 387
    fstream 569
    functional 58
    ios 423
    iosfwd 413
    iostream 417
    istream 487
    iterator 283
    msl_mutex.h 621
    numeric 383
    streambuf 461
    stringfwd 414
    strstr 599
    utility 56
hex 458

I
I/O Library Summary 409
ignore 503
imag 402
    complex 394
imbue
    basic_filebuf 577
    basic_ios 444
    basic_streambuf 480
    iosbase 436
In
    codecvt 156
in_avail 470
Includes
    algorithm 334
Indirect_array 380
    assignment operations 382
    assignment operator 381
    constructors 381
indirect_array
    fill operator 382
infinity
    Numeric_limits 25
Inner_product 384
Inplace_merge
    algorithm 333
Input and Output Library 409–411
Input iterators 282
Insert
    deque 246
    list 249
    vector 259
Insert Iterators 290
Insert_iterator
    constructors 293
    inserter 294
    operator * 293
    operator = 293
Inserter
    insert_iterator 294
Inserters
    basic_ostream, arithmetic 522
    basic_ostream, characters 524
    overloading 526
int_type 94
internal 457
Introduction 1–3
Invalid_argument 48
    constructor 48
ios 423
ios_base 424
    constructors 439
    failure 425
        constructor 425
        what 425
flags 429
fmtflags 425
getloc 437
imbue 436
Init 428
    destructor 428
iostate 427
iword 437
Open Modes 427
precision 434
pword 438
register_callback 438
seekdir 428
setf 432
sync_with_stdio 439
unsetf 433
width 435
xalloc 437
iosfwd 413
```

iostate 427
iostream 417
Iostream Base Class 423–460
Iostream Class Templates 7
Iostream Objects 417–421
Iostreams Definitions 410
Iostreams requirements 410
Is
 locale 144
is_bounded
 Numeric_limits 27
is_exact
 Numeric_limits 22
is_iec559
 Numeric_limits 26
is_integer
 Numeric_limits 22
is_modulo
 Numeric_limits 27
is_open
 basic_filebuf 571
 basic_fstream 592
 basic_ifstream 581
 basic_ofstream 587
is_signed
 Numeric_limits 22
is_specialized
 Numeric_limits 21
istream 487
Istream_iterator
 constructors 294
 destructor 295
 operations 295
Istreambuf_iterator
 constructor 297
 equal 299
 operators 298
istrstream 608
 constructor 609
 destructor 610
 rdbuf 610
 str 611
Iter_swap
 algorithm 311
Iterator 283
 advance 284
 distance 285
Iterator Primitives 283
Iterator Traits 283
Iterators
 basic 284
 bidirectional 282
 forward 282
 input 282
 insert iterators 290
 Operation 284
 output 282
 predefined 285
 Random Access 283
 requirements 282
 reverse 285
Iterators Library 281–300
iword 437

L

Language Support Library 19–46
Leading Underscores 15
left 457
Length
 codecvt 157
length 91
Length_error 49
 constructor 49
less
 functional 63
Less than comparison 53
less_equal
 functional 64
Lexicographical_compare
 algorithm 345
Library-wide Requirements 12
Linkage 14
List 247
 assign 248
 clear 250
 constructor 247
 erase 250
 insert 249
 merge 252
 pop_back 250
 pop_front 250
 push_back 249
 push_front 249
 remove 251

```
resize 248
reverse 252
sort 253
splice 251
swap 253
unique 252
Locale
    facet 136
    category 135
    character classification 141
    character conversions 142
    class ctype 143
    class type_byname 148
    classic 140
    combine 138
    constructor 137
    ctype category 143
    ctype specializations 152
    do_is 146
    do_narrow 148
    do_scan_is 146
    do_scan_not 147
    do_tolower 147
    do_toupper 147
    do_widen 147
    global 140
    has_facet 141
    is 144
    name 138
    narrow 146
    operator != 139
    operator () 139
    Operator == 139
    scan_is 144
    scan_not 144
    tolower 142, 145
    toupper 142, 145
    use_facet 140
    widen 145
locale
    id 137
Locale Names
    combined 133
Locale Types 134
Locales
    basic_streambuf 463
Localization Library 129–240
Lock
    mutex 623
log 405
log10 405
Logic_error 48
    constructor 48
Logical operations
    Functional 64
logical_and
    functional 64
logical_not
    functional 65
logical_or
    functional 65
Lower_bound
    algorithm 328
    map 262
    multimap 266
lt 91
```

M

```
Make_heap
    algorithm 340
Make_pair
    pair 58
Manipulator
    Overloading 545
    scientific 458
Manipulators
    adjustfield 457
    basefield 458
    boolalpha 456
    dec 458
    endl 535
    ends 535
    fixed 458
    floatfield 458
    flush 536
    fmtflags 456
    hex 458
    Instantiations 539
    internal 457
    ios_base 456
    left 457
    noboolalpha 456
    noshowbase 456
    noshowpoint 456
    noshowpos 457
```

```

noskipws 457
nounitbuf 457
nouppercase 457
oct 458
overloaded 459
right 457
showbase 456
showpoint 456
showpos 457
skipws 457
uppercase 457
ws 516

Map 261
    constructor 261
    equal_range 263
    find 262
    lower_bound 262
    swap 264
    upper_bound 263

Mask_array 378
    Assignment operations 380
    assignment operator 379
    constructors 379
    fill operator 380

Max
    algorithm 342
    valarray 359

max
    Numeric_limits 21

Max_element
    algorithm 344

max_exponent
    Numeric_limits 24

max_exponent10
    Numeric_limits 24

Max_length
    codecvt 157

max_size 75

mem_fun 70
mem_fun_ref 72
mem_fun_ref_t 71
mem_fun_t
    Functional 69
mem_fun1_ref_ 71
mem_fun1_t
    Functional 70

Memory
    address 74

allocate 75
allocator globals 76
auto_ptr conversions 86
construct 75
deallocate 75
destroy 76
get 85
get_temporary_buffer 79
max_size 75
operator
    auto_ptr 86
operator* 77
operator auto_ptr_ref 86
operator!= 76
operator* 84
operator== 76
operator-> 85
raw_storage_iterator 77
    constructor 77
release 85
reset 85
return_temporary_buffer 79
Specialized Algorithms 79
uninitialized_copy 80
uninitialized_fill 80

Merge
    algorithm 332
    list 252

Message Retrieval Category 231

Messages
    close 233
    get 233
    open 233

Min
    algorithm 341
    valarray 358

min
    Numeric_limits 21

Min_element
    algorithm 343

min_exponent
    Numeric_limits 23

min_exponent10
    Numeric_limits 23

minus
    functional 59

Mismatch
    algorithm 306

```

Modifier Function 7
modulus
 functional 60
Monetary Category 210
Money_get
 get 217
money_get
 Members 216
Money_put
 put 219
Moneypunct
 curr_symbol 222
 decimal_point 221
 frac_digits 222
 grouping 221
 negative_sign 222
 pos_format 223
 positive_sign 222
 thousands_sep 221
move 92
MSL C++ Debug Mode 661
MSL Debug Mode 661–673
Msl_mutex.h 621–625
msl_mutex.h 621
Msl.Utility 645–660, 695–??
Mslconfig 715–729
Multimap 264
 constructors 265
 equal_range 266
 find 265
 lower_bound 266
 swap 267
multiplies
 functional 60
Multiset 268
 constructor 269
 swap 269
Mutex 621
 lock 623
 operator = 622
 Public Member Functions 623
 unlock 623
mutex
 Constructor 622
 Destructor 623
Mutex_lock 624
 constructor 624
 destructor 625
operator = 624
N
Name
 locale 138
name
 type_info 38
Narrow
 locale 146
Narrow-oriented Iostream Classes 7
neg_format 223
negate
 61
Negative_sign
 moneypunct 222
Negators
 Functional 65
New 32
new_handler 36
Next_permutation
 algorithm 346
noboolalpha 456
None
 bitset 277
Non-member functions
 valarray 360
norm 403
noshowpoint 456
noshowpos 457
noskipws 457
not_eof 92
not_equal_to
 functional 62
not1 66
not2 66
nounitbuf 457
nouppercase 457
NTCTS 7, 90
Nth_element
 algorithm 327
Num_get
 get 164
num_get
 Virtual Functions 166
Num_put
 put 168

```
num_put
    Members 167
    Virtual Functions 169
Numeric Category 163
Numeric limits 20
Numeric Punctuation Facet 169
Numeric_limits
    abort 30
    atexit 31
    denorm_min 26
    digits 21
    epsilon 23
    exit 31
    float_denorm_style 28
    float_round_style 28
    has_denorm 25
    has_denorm_loss 25
    has_infinity 24
    has_quiet_NaN 24
    has_signaling_NaN 24
    infinity 25
    is_bounded 27
    is_exact 22
    is_ie559 26
    is_integer 22
    is_modulo 27
    is_signed 22
    is_specialized 21
    max 21
    max_exponent 24
    max_exponent10 24
    min 21
    min_exponent 23
    min_exponent10 23
    quiet_NaN 25
    radix 22
    round_error 23
    round_style 28
    signaling_NaN 26
    Static Members 20
    tinyness_before 27
    traps 27
Numerics Library 351–390
Numpunct
    decimal_point 170
    falsename 171
    grouping 170
    thousands_sep 170
    truename 170
```

O

Object State 7
Observer Function 7
oct 458
off_type 94
Open
 basic_filebuf 572
 basic_fstream 593
 basic_ifstream 581
 basic_ofstream 587
 messages 233
Open Modes
 basic_filebuf 572
 basic_fstream 593
 basic_ifstream 582
 basic_ofstream 588
 ios_base 427
Operator 56, 56, 58, 254, 272, 277, 279, 401
!= 56
*
 Memory 77
> 56
>= 57
delete 33
 placement 34
new 32
 placement 34
Operator -
 complex 398
Operator !
 basic_ios 446
Operator !=
 bitset 276
 complex 400
 locale 139
Operator &
 bitset 278
Operator &=br/> bitset 271
Operator ()
 locale 139
Operator * 77, 84
 complex 399
 insert_iterator 293
Operator *=
 complex 395
Operator +

```

complex 397
Operator += complex 394
Operator / complex 399
Operator /= complex 396
Operator -= complex 395
Operator = complex 393
    front_insert_iterator 291
    insert_iterator 293
Operator ==
    bitset 276
    complex 400
    locale 139
    pair 57
    queue 254
Operator >>
    bitset 277, 279
    complex 401
Operator >>=
    bitset 272
Operator ^
    bitset 278
Operator ^=
    bitset 272
Operator |
    bitset 278
Operator |=
    bitset 272
Operator ~
    bitset 274
Operator bool
    basic_ios 446
    sentry, basic_istream 490
    sentry, basic_ostream 521
Operator!=
    Memory 76
    type_info 37
    utility 56
Operator()
    Functional 68
Operator++ 78
Operator= 78
    Auto_ptr 84
    mutex 622
mutex_lock 624
Operator==
    Memory 76
    type_info 37
Operator-> 85
Operator>
    utility 56
Operator>=
    utility 57
Operators
    back_insert_iterator 290
    reverse_iterator 286
    Utility 56
ostream cerr 419
ostream clog 419
ostream cout 418
Ostream_iterator
    constructors 296
    destructor 296
Ostream_iterator Operations 297
Ostreambuf_iterator
    constructor 299
    failed 300
Ostreambuf_iterator Operations 299
ostrstream 611
    constructor 612
    destructor 613
    freeze 613
    pcount 614
    rdbuf 615
    str 616
Other Conventions 11
Other Runtime Support 44
Out
    codecvt 156
Out_of_range 49
    constructor 49
Output Iterators 282
overflow 486, 608
    basic_filebuf 575
    basic_streambuf 554
Overflow_error 50
    constructor 50
Overloaded
    manipulators 459
Overloading
    Extractors 495

```

Inserters	526
Manipulator	545
P	
Pair	
Constructors	57
make_pair	58
Operator $\langle\!\rangle$	58
Operator ==	57
Utility	57
Partial_sort	
algorithm	325
Partial_sort_copy	
algorithm	326
partial_sum	385
Partition	
algorithm	321
pbackfail	485, 607
basic_filebuf	575
basic_streambuf	553
pbase	479
p bump	479
pcount	
ostrstream	614
strstream	618
strstreambuf	603
peek	505
Placement Operator Delete	34
Placement Operator New	34
plus	
functional	59
pointer_to_binary_function	
Functional	68, 69
pointer_to_unary_function	
Functional	68
polar	403
Pop	
priority_queue	255
stack	257
Pop_back	
list	250
Pop_front	
list	250
Pop_heap	
algorithm	339
Pos_format	
moneypunct	223
pos_type	94
Positive_sign	
moneypunct	222
pow	406
ptr	479
precision	434
Predefined Iterators	285
Predicate	
not1	66
not2	66
Prev_permutation	
algorithm	347
Priority_queue	254
constructors	255
pop	255
push	255
Program-defined Facets	240
pubimbue	464
pubseekoff	466
pubseekpos	467
pubsetbuf	464
pubsync	469
Push	
priority_queue	255
stack	257
Push_back	
list	249
Push_front	
list	249
Push_heap	
algorithm	339
Put	
money_put	219
num_put	168
put	530
put_time_put	196
putback	509
pword	438
Q	
Qsort	348
Queue	253
operator	254
operator ==	254
quiet_NaN	
Numeric_limits	25

R

radix
 Numeric_limits 22

Random Access Iterators 283

Random_shuffle
 algorithm 321

Range_error 50
 constructor 50

Raw storage iterator 77

Raw_storage_iterator
 constructor 77
 operator = 78
 operator++ 78

raw_storage_iterator 78

rdbuf 443
 basic_fstream 591
 basic_ifstream 580
 basic_istringstream 557
 basic_ofstream 585
 basic_ostringstream 561
 basic_stringstream 566
 istrstream 610
 ostrstream 615
 strstream 618

rdstate 447

read 505

readsome 507

real 402
 complex 394

Reentrancy 17

register_callback 438

release 85

Remove
 algorithm 316
 list 251

Remove_copy
 algorithm 317

Remove_copy_if
 algorithm 317

Remove_if
 algorithm 316

Replace
 algorithm 313

Replace_copy
 algorithm 313

Replace_copy_if
 algorithm 314

Replacement Function 8

Replacement Functions 16

Repositional Stream 8

Required Behavior 8

Reserved Function 8

Reserved Names 15

Reset
 bitset 273
 reset 85
 resetiosflags 539

Resize
 deque 245
 list 248
 valarray 360
 vector 259

Restrictions On Exception Handling 17

return_temporary_buffer 79

Reverse
 algorithm 319
 list 252

Reverse iterators 285

Reverse_copy
 algorithm 319

Reverse_iterator
 base 286
 constructor 286
 operators 286

right 457

Rotate
 algorithm 320

Rotate_copy
 algorithm 320

round_error
 Numeric_limits 23

round_style
 Numeric_limits 28

Runtime_error 49
 constructor 49

S

sbumc 471

Scan_is
 locale 144

Scan_not
 locale 144

scientific 458

Search

```
algorithm 308
Search_n
    algorithm 309
seekdir 428
seekg 514
seekoff 481
    basic_filebuf 576
    basic_streambuf 554
    strstreambuf 606
seekp 528
seekpos 482
    basic_filebuf 576
    basic_streambuf 555
    strstreambuf 606
sentry 489, 520
    constructor
        basic_istream 490
        basic_ostream 520
    destructor
        basic_istream 490
        basic_ostream 521
Operator bool
    basic_istream 490
    basic_ostream 521
Sequences 244
Sequences Requirements 242
Set 267
    bitset 273
    constructors 268
    swap 268
Set_difference
    algorithm 337
Set_intersection
    algorithm 336
Set_symmetric_difference
    algorithm 338
Set_union
    algorithm 335
setbase 541
setbuf 481, 577
    strstreambuf 605
setf 432
setfill 542
setg 478
setiosflags 540
setp 480
setprecision 543
setstate 450
setw 544
sgetc 472
sgetn 473
Shift
    valarray 359
showmanc 483
showmanyC
    basic_filebuf 574
showpoint 456
showpos 457
Slice_array 373
signaling_NaN
    Numeric_limits 26
sin 406
sinh 407
Size
    bitset 275
    gslice 376
    slice 373
    stack 256
    valarray 358
skipws 457
Slice
    constructors 372
    size 373
    start 373
    stride 373
Slice_array
    assignment operations 375
    assignment operator 374
    constructor 374
    fill operator 375
snextc 470
Sort
    algorithm 323
    list 253
Sort_heap
    algorithm 341
Specialized Ctype members 154
Splice
    list 251
    sputback 473
    sputc 475
    sputn 476
    sqrt 407
Stable_partition
```

- algorithm 322
- Stable_sort
 - algorithm 324
- Stack 256
 - constructors 256
 - empty 256
 - pop 257
 - push 257
 - size 256
 - top 257
- Standard Iterator Tags 284
- Standard Locale Categories 142
- Start
 - gslice 376
 - slice 373
- state_type 94
- Static Members
 - Numeric_limits 20
- str
 - basic_istringstream 558
 - basic_ostringstream 563
 - basic_streambuf 551
 - basic_stringstream 567
 - istrstream 611
 - ostrstream 616
 - strstream 618
 - strstreambuf 604
- Stream
 - buffering 418
- Stream Buffers 461–486
- Stream Iterators 294
- streambuf 461
- Stride
 - gslice 377
 - slice 373
- string 94
- String Based Streams 549–568
- Stringfwd 414
- Strings Library 89–128
- Strstream 599–619
- strstream 599, 612
 - constructor 600, 617
 - destructor 617
 - freeze 617
 - pcount 618
 - rdbuf 618
 - str 618
- strstreambuf 600
 - freeze 602
 - pcount 603
 - seekoff 606
 - seekpos 606
 - setbuf 605
 - str 604
 - stiostream
 - overflow 608
 - strstream
 - pbackfail 607
 - underflow 607
- strtreambuf
 - destructor 601
- struct char_traits 94
- subset 355
 - valarray 355
- Sum
 - valarray 358
- sungetc 475
- Supported locale names 129
- Swap
 - algorithm 310
 - deque 247
 - list 253
 - map 264
 - multimap 267
 - multiset 269
 - set 268
 - vector 260
- swap
 - basic_string 124
- Swap_ranges
 - algorithm 311
- sync 512
 - basic_filebuf 577
 - basic_streambuf 482
- sync_with_stdio
 - ios_base 439

T

- tan 407
- tanh 408
- tellg 513
- tellp 527
- Template Class Codecvt 155
- Template Class Codecvt_byname 159
- Template Class Collate 176

Template Class Collate_byname 177
Template Class Messages 231
Template Class Messages_byname 237
Template Class Money_get 216
Template Class Money_put 218
Template Class Moneypunct 220
Template Class Moneypunct_byname 226
Template Class Num_get 163
Template Class Num_put 167
Template Class Numpunct 169
Template Class Numpunct_byname 172
Template Class Time_get_byname 195
Template Class Time_put 195
Template Class Time_put_byname 197
terminate 44
terminate_handler 43
Test
 bitset 276
Thousands_sep
 moneypunct 221
 numpunct 170
tie 441
Time Category 186
Time_put
 Virtual functions 197
time_put,do_put 197
timeput,put 196
tinyness_before
 Numeric_limits 27
to_char_type 93
to_int_type 93
To_string
 bitset 275
To_ulong
 bitset 274
Tolower
 locale 142, 145
Top
 stack 257
Toupper
 locale 142, 145
Traits 8
traits 90
Transform
 collate 176
Translation Units 14
traps
 Numeric_limits 27
Truename
 numpunct 170
Ttransform
 algorithm 312
Type identification 36
Type_info
 assignment operator 38
 before 37
 constructor 38
 name 38
 operator != 37
 operator== 37

U

uflow 484
Unary operators
 valarray 355
Unary_function 58
Unary_negate 66
uncaught_exception 44
underflow 484
 basic_filebuf 575
 basic_streambuf 553
 strstreambuf 607
Underflow_error 50
 constructor 50
unexpected 43
unexpected_handler 42
unget 510
uninitialized_copy 80
uninitialized_fill 80
Unique
 algorithm 318
 list 252
Unique_copy
 algorithm 318
Unlock
 mutex 623
unsetf 433
Unshift
 codecvt 156
Upper_bound
 algorithm 329
 map 263
uppercase 457

Use_facet
 locale 140
Using the library 14
Utility 56
 Operator 56
 Operator!= 56
 Operator> 56
 Operator>= 57
 Operators 56
 Pair 57

what
 bad_alloc 35

Widen
 locale 145

Wide-oriented Iostream Classes 8

width 435

wistream wcin 420

wostream wcerr 420

wostream wclog 421

wostream wcout 420

write 530

ws 516

V

Valarray
 apply 360
 assignment operations 356
 binary operators 361
 constructors 353
 cshift 359
 destructor 353
 logical operators 365
 max 359
 min 358
 non-member functions 360
 resize 360
 shift 359
 size 358
 sum 358
 transcendentals 370
 unary operators 355

Vector 257, 260
 assign 258
 capacity 259
 constructors 258
 erase 260
 insert 259
 resize 259
 swap 260

W

wcerr 420
wcin 420
wclog 421
wcout 420

What
 bad_cast 39
 bad_exception 42
 bad_typeid 40
 exception 41

X

xalloc 437
xsgetn 483
xsputn 485