



Standard C++ Library Reference

Note!

Before using this information and the product it supports, read the information in "Notices" on page 415.

Edition Notice (September 2004)

© Copyright International Business Machines Corporation 1999, 2004. All rights reserved.

US Government Users Restricted Rights – Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

Contents

Chapter 1. C++ Library 1

Chapter 2. C++ Library Overview 5

Using C++ Library Headers	5
C++ Library Conventions	6
Iostreams Conventions	7
C++ Program Startup and Termination	7

Chapter 3. Characters 9

Character Sets	9
Character Sets and Locales	10
Escape Sequences	10
Numeric Escape Sequences	11
Trigraphs	11
Multibyte Characters	12
Wide-Character Encoding.	13

Chapter 4. Expressions 15

Chapter 5. Files and Streams 17

Text and Binary Streams	17
Byte and Wide Streams	18
Controlling Streams	19
Stream States	20

Chapter 6. Functions. 23

Chapter 7. Formatted Input 25

Scan Formats	25
Scan Functions	25
Scan Conversion Specifiers	26

Chapter 8. Formatted Output 31

Print Formats.	31
Print Functions	32
Print Conversion Specifiers	33

Chapter 9. STL Conventions 37

Iterator Conventions	37
Algorithm Conventions	38

Chapter 10. Containers. 41

Cont.	42
Cont::begin	44
Cont::clear.	44
Cont::const_iterator.	44
Cont::const_reference	44
Cont::const_reverse_iterator	44
Cont::difference_type	44
Cont::empty	44
Cont::end	45
Cont::erase	45
Cont::iterator	45

Cont::max_size	45
Cont::rbegin	45
Cont::reference	45
Cont::rend	45
Cont::reverse_iterator	46
Cont::size	46
Cont::size_type	46
Cont::swap	46
Cont::value_type.	46
operator!=	46
operator==	46
operator<	47
operator<=	47
operator>	47
operator>=	47
swap	47

Chapter 11. Preprocessing 49

Chapter 12. Standard C++ Library Header Files 51

<bitset>	54
bitset	54
operator<<	58
operator>>	59
<cassert>	59
<cctype>	59
<cerrno>	59
<cfloat>	59
<ciso646>	60
<climits>	60
<locale>	60
<cmath>	60
<complex>	61
abs	63
arg	63
complex	63
complex<double>	66
complex<float>	67
complex<long double>	67
conj	67
cos	67
cosh	67
exp	68
imag.	68
log	68
log10	68
norm	68
operator!=	68
operator*	68
operator+	69
operator-	69
operator/	69
operator<<	69
operator==	70

operator>>	70	noboolalpha	99
polar	70	noshowbase	99
pow	70	noshowpoint	99
real	71	noshowpos	99
sin	71	noskipws	100
sinh	71	nounitbuf	100
sqrt	71	nouppercase	100
__STD_COMPLEX	71	oct	100
tan	71	right	100
tanh	71	scientific	100
<csetjmp>	72	showbase	100
<csignal>	72	showpoint	100
<cstdarg>	72	showpos	100
<cstddef>	72	skipws	101
<cstdio>	72	streamoff	101
<cstdlib>	73	streampos	101
<cstring>	73	streamsize	101
<ctime>	73	unitbuf	101
<cwchar>	74	uppercase	101
<cwctype>	74	wios	101
<exception>	74	wstreampos	101
bad_exception	75	<iosfwd>	102
exception	75	<iostream>	103
set_terminate	75	cerr	104
set_unexpected	75	cin	104
terminate	75	clog	104
terminate_handler	75	cout	104
uncaught_exception	76	wcerr	104
unexpected	76	wcin	105
unexpected_handler	76	wclog	105
<fstream>	76	wcout	105
basic_filebuf	77	<istream>	105
basic_fstream	81	basic_iostream	106
basic_ifstream	82	basic_istream	106
basic_ofstream	83	iostream	112
filebuf	84	istream	112
fstream	84	operator>>	112
ifstream	84	wiostream	114
ofstream	85	wistream	114
wfstream	85	ws	114
wifstream	85	<limits>	114
wofstream	85	float_denorm_style	114
wfilebuf	85	float_round_style	115
<iomanip>	85	numeric_limits	115
resetiosflags	86	<locale>	119
setbase	86	codecvt	121
setfill	86	codecvt_base	125
setiosflags	86	codecvt_byname	125
setprecision	86	collate	126
setw	86	collate_byname	127
<ios>	86	ctype	127
basic_ios	88	ctype<char>	131
boolalpha	91	ctype_base	132
dec	92	ctype_byname	133
fixed	92	has_facet	133
fpos	92	isalnum	133
hex	93	isalpha	133
internal	94	iscntrl	133
ios	94	isdigit	133
ios_base	94	isgraph	134
left	99	islower	134

isprint	134	logic_error	184
ispunct	134	out_of_range	185
isspace	134	overflow_error	185
isupper	134	range_error	185
isxdigit	134	runtime_error	185
locale	134	underflow_error	185
messages	139	<streambuf>	185
messages_base	140	basic_streambuf	186
messages_byname	140	streambuf	194
money_base	141	wstreambuf	194
money_get	141	<string>	195
money_put	143	basic_string	197
moneypunct	145	char_traits	210
moneypunct_byname	149	char_traits<char>	213
num_get	149	char_traits<wchar_t>	213
num_put	152	getline	214
numpunct	156	operator+	214
numpunct_byname	158	operator!=	214
time_base	158	operator==	215
time_get	158	operator<	215
time_get_byname	162	operator<<	215
time_put	162	operator<=	216
time_put_byname	163	operator>	216
tolower	163	operator>=	216
toupper	164	operator>>	216
use_facet	164	string	217
<new>	164	swap	217
bad_alloc	164	wstring	217
new_handler	165	<strstream>	217
nothrow	165	strstreambuf	217
nothrow_t	165	istrstream	221
operator delete	165	ostrstream	222
operator delete[]	165	strstream	223
operator new	166	<typeinfo>	224
operator new[]	167	bad_cast	224
set_new_handler	167	bad_typeid	224
<ostream>	168	type_info	225
basic_ostream	169	<valarray>	225
endl	173	abs	229
ends	173	acos	229
flush	173	asin	229
operator<<	173	atan	230
ostream	176	atan2	230
wostream	176	cos	230
<sstream>	176	cosh	230
basic_stringbuf	176	exp	230
basic_istringstream	180	gslice	230
basic_ostringstream	181	gslice_array	231
basic_stringstream	182	indirect_array	232
istringstream	183	log	233
ostringstream	183	log10	233
stringbuf	183	mask_array	233
stringstream	183	operator!=	234
wstringstream	183	operator%	234
wostringstream	183	operator&	234
wstringbuf	183	operator&&	234
wstringstream	183	operator>	235
<stdexcept>	184	operator>=	235
domain_error	184	operator<	235
invalid_argument	184	operator<<	236
length_error	184		

operator<=	236
operator*	236
operator+	236
operator-	237
operator/	237
operator==	237
operator^	237
operator	238
operator	238
pow	238
sin	238
sinh	239
slice	239
slice_array	239
sqrt	240
tan	240
tanh	240
valarray	240
valarray<bool>	247

Chapter 13. Standard Template Library

C++ 249

<algorithm>	249
adjacent_find	253
binary_search	254
copy	254
copy_backward	254
count	254
count_if	255
equal	255
equal_range	255
fill	255
fill_n	256
find	256
find_end	256
find_first_of	256
find_if	257
for_each	257
generate	257
generate_n	257
includes	257
inplace_merge	258
iter_swap	258
lexicographical_compare	258
lower_bound	259
make_heap	259
max	259
max_element	259
merge	260
min	260
min_element	261
mismatch	261
next_permutation	261
nth_element	262
partial_sort	262
partial_sort_copy	262
partition	263
pop_heap	263
prev_permutation	263
push_heap	264
random_shuffle	264

remove	264
remove_copy	265
remove_copy_if	265
remove_if	265
replace	266
replace_copy	266
replace_copy_if	266
replace_if	266
reverse	267
reverse_copy	267
rotate	267
rotate_copy	267
search	267
search_n	268
set_difference	268
set_intersection	269
set_symmetric_difference	269
set_union	270
sort	270
sort_heap	271
stable_partition	271
stable_sort	271
swap	272
swap_ranges	272
transform	272
unique	272
unique_copy	273
upper_bound	273
<deque>	274
deque	274
operator!=	281
operator==	281
operator<	282
operator<=	282
operator>	282
operator>=	282
swap	282
<functional>	282
binary_function	285
binary_negate	285
bind1st	285
bind2nd	285
binder1st	286
binder2nd	286
const_mem_fun_t	286
const_mem_fun_ref_t	287
const_mem_fun1_t	287
const_mem_fun1_ref_t	287
divides	287
equal_to	287
greater	288
greater_equal	288
hash	288
less	288
less_equal	288
logical_and	288
logical_not	289
logical_or	289
mem_fun	289
mem_fun_ref	289
mem_fun_t	289

mem_fun_ref_t	290	operator>.	336
mem_fun1_t	290	operator>=	336
mem_fun1_ref_t	290	swap	336
minus	290	<memory>	336
modulus	290	allocator	337
multiplies	290	allocator<void>.	340
negate.	291	auto_ptr	340
not1	291	get_temporary_buffer.	343
not2	291	operator!=	343
not_equal_to	291	operator==	343
plus	291	raw_storage_iterator	343
pointer_to_binary_function	291	return_temporary_buffer	344
pointer_to_unary_function	291	uninitialized_copy.	345
ptr_fun	292	uninitialized_fill	345
unary_function	292	uninitialized_fill_n	345
unary_negate	292	<numeric>	345
<iterator>	293	accumulate	346
advance	294	adjacent_difference	346
back_insert_iterator	294	inner_product	346
back_inserter	296	partial_sum	347
bidirectional_iterator_tag	296	<queue>	347
distance	296	operator!=	348
forward_iterator_tag	296	operator==	348
front_insert_iterator	296	operator<.	348
front_inserter	297	operator<=	348
input_iterator_tag	298	operator>.	348
insert_iterator	298	operator>=	348
inserter	299	priority_queue	348
istream_iterator.	299	queue	351
istreambuf_iterator	300	<set>	353
iterator	302	multiset	354
iterator_traits	302	operator!=	360
operator!=	303	operator==	360
operator==	303	operator<.	360
operator<.	304	operator<=	361
operator<=	304	operator>.	361
operator>.	304	operator>=	361
operator>=	304	set	361
operator+.	304	swap	367
operator-.	304	<stack>	368
ostream_iterator	304	operator!=	368
ostreambuf_iterator	306	operator==	368
output_iterator_tag	307	operator<.	368
random_access_iterator_tag.	307	operator<=	369
reverse_iterator.	307	operator>.	369
<list>	310	operator>=	369
list	310	stack	369
operator!=	319	<unordered_map>.	371
operator==	319	unordered_map	371
operator<.	319	unordered_multimap	378
operator<=	319	<unordered_set>	386
operator>.	319	unordered_multiset	386
operator>=	319	unordered_set	393
swap	320	<utility>	400
<map>	320	make_pair	401
map	321	operator!=	401
multimap.	328	operator==	401
operator!=	335	operator<.	401
operator==	335	operator<=	402
operator<.	335	operator>.	402
operator<=	336	operator>=	402

pair	402
<vector>	403
operator!=	403
operator==	403
operator<.	404
operator<=	404
operator>.	404
operator>=	404

swap	404
vector	404
vector<bool, A>	411

Notices 415

References	417
Bug Reports	417

Chapter 1. C++ Library

The C++ library supplied by IBM and this manual are based on the Dinkum C++ Library and the *Dinkum C++ Library Reference*. Changes to this manual, based on changes to the C++ language standard, are indicated with a vertical bar (|).

Use of this Dinkum C++ Library Reference is subject to limitations. See the Copyright Notice (page 415) for detailed restrictions.

A C++ program can call on a large number of functions from the **Dinkum C++ Library**, a conforming implementation of the **Standard C++ library**. These functions perform essential services such as input and output. They also provide efficient implementations of frequently used operations. Numerous function and class definitions accompany these functions to help you to make better use of the library. Most of the information about the Standard C++ library can be found in the descriptions of the **C++ library headers (page 5)** that declare or define library entities for the program.

The Standard C++ library consists of 53 headers. Of these 53 headers, 13 constitute the **Standard Template Library**, or **STL**. These are indicated below with the notation (STL):

- <algorithm> (page 249) — (STL) for defining numerous templates that implement useful algorithms
- <bitset> (page 54) — for defining a template class that administers sets of bits
- <complex> (page 61) — for defining a template class that supports complex arithmetic
- <deque> (page 274) — (STL) for defining a template class that implements a deque container
- <exception> (page 74) — for defining several functions that control exception handling
- <fstream> (page 76) — for defining several iostreams template classes that manipulate external files
- <functional> (page 282) — (STL) for defining several templates that help construct predicates for the templates defined in <algorithm> (page 249) and <numeric> (page 345)
- <iomanip> (page 85) — for declaring several iostreams manipulators that take an argument
- <ios> (page 86) — for defining the template class that serves as the base for many iostreams classes
- <iosfwd> (page 102) — for declaring several iostreams template classes before they are necessarily defined
- <iostream> (page 103) — for declaring the iostreams objects that manipulate the standard streams
- <istream> (page 105) — for defining the template class that performs extractions
- <iterator> (page 293) — (STL) for defining several templates that help define and manipulate iterators
- <limits> (page 114) — for testing numeric type properties
- <list> (page 310) — (STL) for defining a template class that implements a list container
- <locale> (page 119) — for defining several classes and templates that control locale-specific behavior, as in the iostreams classes

`<map>` (page 320) — (STL) for defining template classes that implement associative containers that map keys to values
`<memory>` (page 336) — (STL) for defining several templates that allocate and free storage for various container classes
`<new>` (page 164) — for declaring several functions that allocate and free storage
`<numeric>` (page 345) — (STL) for defining several templates that implement useful numeric functions
`<ostream>` (page 168) — for defining the template class that performs insertions
`<queue>` (page 347) — (STL) for defining a template class that implements a queue container
`<set>` (page 353) — (STL) for defining template classes that implement associative containers
`<sstream>` (page 176) — for defining several iostreams template classes that manipulate string containers
`<stack>` (page 368) — (STL) for defining a template class that implements a stack container
`<stdexcept>` (page 184) — for defining several classes useful for reporting exceptions
`<streambuf>` (page 185) — for defining template classes that buffer iostreams operations
`<string>` (page 195) — for defining a template class that implements a string container
`<stringstream>` (page 217) — for defining several iostreams classes that manipulate in-memory character sequences
`<typeinfo>` (page 224) — for defining class `type_info`, the result of the `typeid` operator
`<unordered_map>` (page 371) — (STL) for defining template classes that implement unordered associative containers that map keys to values
`<unordered_set>` (page 386) — (STL) for defining template classes that implement unordered associative containers
`<utility>` (page 400) — (STL) for defining several templates of general utility
`<valarray>` (page 225) — for defining several classes and template classes that support value-oriented arrays
`<vector>` (page 403) — (STL) for defining a template class that implements a vector container

The Standard C++ library works in conjunction with the headers from the Standard C library. For information about the Standard C library, refer to the documentation that is supplied with the operating system.

Other information on the Standard C++ library includes:

C++ Library Overview (page 5) — how to use the Standard C++ library
Characters (page 9) — how to write character constants (page 9) and string literals (page 9), and how to convert between multibyte characters (page 12) and wide characters (page 13)
Files and Streams (page 17) — how to read and write data between the program and files (page 17)
Formatted Output (page 31) — how to generate text under control of a format string (page 31)
Formatted Input (page 25) — how to scan and parse text under control of a format string (page 31)
STL Conventions (page 37) — how to read the descriptions of STL (page 1)

template classes and functions

Containers (page 41) — how to use an arbitrary STL (page 1) container template class

A few special conventions are introduced into this document specifically for this particular **implementation** of the Standard C++ library. Because the C++ Standard (page 417) is still relatively new, not all implementations support all the features described here. Hence, this implementation introduces macros, or alternative declarations, where necessary to provide reasonable substitutes for the capabilities required by the C++ Standard.

Chapter 2. C++ Library Overview

Using C++ Library Headers (page 5) · C++ Library Conventions (page 6) ·
Iostreams Conventions (page 7) · Program Startup and Termination (page 7)

All C++ library entities are declared or defined in one or more standard headers. To make use of a library entity in a program, write an include directive (page 50) that names the relevant standard header. The Standard C++ library consists of 53 required headers. These 53 **C++ library headers** (along with the additional Standard C headers) constitute a **hosted implementation** of the C++ library:

<algorithm> (page 249), <bitset> (page 54), <cassert> (page 59), <cctype> (page 59), <cerrno> (page 59), <cfloat> (page 59), <ciso646> (page 60), <climits> (page 60), <locale> (page 60), <cmath> (page 60), <complex> (page 61), <csetjmp> (page 72), <csignal> (page 72), <cstdarg> (page 72), <cstddef> (page 72), <cstdio> (page 72), <cstdlib> (page 73), <cstring> (page 73), <ctime> (page 73), <cwchar> (page 74), <cwctype> (page 74), <deque> (page 274), <exception> (page 74), <fstream> (page 76), <functional> (page 282), <iomanip> (page 85), <ios> (page 86), <iosfwd> (page 102), <iostream> (page 103), <istream> (page 105), <iterator> (page 293), <limits> (page 114), <list> (page 310), <locale> (page 119), <map> (page 320), <memory> (page 336), <new> (page 164), <numeric> (page 345), <ostream> (page 168), <queue> (page 347), <set> (page 353), <sstream> (page 176), <stack> (page 368), <stdexcept> (page 184), <streambuf> (page 185), <string> (page 195), <strstream> (page 217), <typeinfo> (page 224), <unordered_map> (page 371), <unordered_set> (page 386), <utility> (page 400), <valarray> (page 225), and <vector> (page 403).

A **freestanding implementation** of the C++ library provides only a subset of these headers: <cstddef> (page 72), <cstdlib> (page 73) (declaring at least the functions abort, atexit, and exit), <exception> (page 74), <limits> (page 114), <new> (page 164), <typeinfo> (page 224), and <cstdarg> (page 72).

The C++ library headers have two broader subdivisions, iostreams (page 7) headers and STL (page 1) headers.

Using C++ Library Headers

You include the contents of a standard header by naming it in an *include* (page 50) directive, as in:

```
#include <iostream> /* include I/O facilities */
```

You can include the standard headers in any order, a standard header more than once, or two or more standard headers that define the same macro or the same type. Do not include a standard header within a declaration. Do not define macros that have the same names as keywords before you include a standard header.

A C++ library header includes any other C++ library headers it needs to define needed types. (Always include explicitly any C++ library headers needed in a translation unit, however, lest you guess wrong about its actual dependencies.) A Standard C header never includes another standard header. A standard header declares or defines only the entities described for it in this document.

Every function in the library is declared in a standard header. Unlike in Standard C, the standard header never provides a masking macro, with the same name as the function, that masks the function declaration and achieves the same effect.

All names other than `operator delete` and `operator new` in the C++ library headers are defined in the `std` namespace, or in a namespace nested within the `std` namespace. Including a C++ library header does *not* introduce any library names into the current namespace. You refer to the name `cin` (page 104), for example, as `std::cin`. Alternatively, you can write the declaration:

```
using namespace std;
```

which promotes all library names into the current namespace. If you write this declaration immediately after all *include* directives, you can otherwise ignore namespace considerations in the remainder of the translation unit. Note that macro names are not subject to the rules for nesting namespaces.

Note that the C Standard headers behave mostly as if they include no namespace declarations. If you include, for example, `<cstdlib>` (page 73), you should call `std::abort()` to cause abnormal termination, but if you include `<stdlib.h>`, you should call `abort()`. (The C++ Standard is intentionally vague on this topic, so you should stick with just the usages described here for maximum portability.)

Unless specifically indicated otherwise, you may not define names in the `std` namespace, or in a namespace nested within the `std` namespace.

C++ Library Conventions

The C++ library obeys much the same conventions as the Standard C library, plus a few more outlined here.

An implementation has certain latitude in how it declares types and functions in the C++ library:

- Names of functions in the Standard C library may have either **extern "C++"** or **extern "C"** linkage. Include the appropriate Standard C header rather than declare a library entity inline.
- A member function name in a library class may have additional function signatures over those listed in this document. You can be sure that a function call described here behaves as expected, but you cannot reliably take the address of a library member function. (The type may not be what you expect.)
- A library class may have undocumented (non-virtual) base classes. A class documented as derived from another class may, in fact, be derived from that class through other undocumented classes.
- A type defined as a synonym for some integer type may be the same as one of several different integer types.
- A **bitmask type** can be implemented as either an integer type or an enumeration. In either case, you can perform bitwise operations (such as AND and OR) on values of the same bitmask type. The *elements* A and B of a bitmask type are nonzero values such that A & B is zero.
- A library function that has no exception specification can throw an arbitrary exception, unless its definition clearly restricts such a possibility.

On the other hand, there are some restrictions you can count on:

- The Standard C library uses no masking macros. Only specific function signatures are reserved, not the names of the functions themselves.

- A library function name outside a class will *not* have additional, undocumented, function signatures. You can reliably take its address.
- Base classes and member functions described as virtual are assuredly virtual, while those described as non-virtual are assuredly non-virtual.
- Two types defined by the C++ library are always different unless this document explicitly suggests otherwise.
- Functions supplied by the library, including the default versions of replaceable functions (page 164), can throw *at most* those exceptions listed in any exception specification. No destructors supplied by the library throw exceptions. Functions in the Standard C library may propagate an exception, as when `qsort` calls a comparison function that throws an exception, but they do not otherwise throw exceptions.

Iostreams Conventions

The **iostreams** headers support conversions between text and encoded forms, and input and output to external files (page 17): `<fstream>` (page 76), `<iomanip>` (page 85), `<ios>` (page 86), `<iosfwd>` (page 102), `<iostream>` (page 103), `<istream>` (page 105), `<ostream>` (page 168), `<sstream>` (page 176), `<streambuf>` (page 185), and `<strstream>` (page 217).

The simplest use of iostreams requires only that you include the header `<iostream>`. You can then extract values from `cin` (page 104), to read the standard input. The rules for doing so are outlined in the description of the class `basic_istream` (page 106). You can also insert values to `cout` (page 104), to write the standard output. The rules for doing so are outlined in the description of the class `basic_ostream` (page 169). Format control common to both extractors and insertors is managed by the class `basic_ios` (page 88). Manipulating this format information in the guise of extracting and inserting objects is the province of several manipulators (page 85).

You can perform the same iostreams operations on files that you open by name, using the classes declared in `<fstream>`. To convert between iostreams and objects of class `basic_string` (page 197), use the classes declared in `<sstream>`. And to do the same with C strings, use the classes declared in `<strstream>`.

The remaining headers provide support services, typically of direct interest to only the most advanced users of the iostreams classes.

C++ Program Startup and Termination

A C++ program performs the same operations as does a C program at program startup and at program termination, plus a few more outlined here.

Before the target environment calls the function `main`, and after it stores any constant initial values you specify in all objects that have static duration, the program executes any remaining constructors for such static objects. The order of execution is not specified between translation units, but you can nevertheless assume that some iostreams (page 7) objects are properly initialized for use by these static constructors. These control text streams:

- **cin (page 104)** — for standard input
- **cout (page 104)** — for standard output
- **cerr (page 104)** — for unbuffered standard error output
- **clog (page 104)** — for buffered standard error output

You can also use these objects within the destructors called for static objects, during program termination.

As with C, returning from main or calling exit calls all functions registered with atexit in reverse order of registry. An exception thrown from such a registered function calls terminate().

Chapter 3. Characters

Character Sets (page 9) · Character Sets and Locales (page 10) · Escape Sequences (page 10) · Numeric Escape Sequences (page 11) · Trigraphs (page 11) · Multibyte Characters (page 12) · Wide-Character Encoding (page 13)

Characters play a central role in Standard C. You represent a C program as one or more **source files**. The translator reads a source file as a text stream consisting of characters that you can read when you display the stream on a terminal screen or produce hard copy with a printer. You often manipulate text when a C program executes. The program might produce a text stream that people can read, or it might read a text stream entered by someone typing at a keyboard or from a file modified using a text editor. This document describes the characters that you use to write C source files and that you manipulate as streams when executing C programs.

Character Sets

When you write a program, you express C source files as text lines (page 17) containing characters from the **source character set**. When a program executes in the **target environment**, it uses characters from the **target character set**. These character sets are related, but need not have the same encoding or all the same members.

Every character set contains a distinct code value for each character in the **basic C character set**. A character set can also contain additional characters with other code values. For example:

- The **character constant** 'x' becomes the value of the code for the character corresponding to x in the target character set.
- The **string literal** "xyz" becomes a sequence of character constants stored in successive bytes of memory, followed by a byte containing the value zero: {'x', 'y', 'z', '\0'}

A string literal is one way to specify a **null-terminated string**, an array of zero or more bytes followed by a byte containing the value zero.

Visible graphic characters in the basic C character set:

Form	Members
<i>letter</i>	A B C D E F G H I J K L M N O P Q R S T U V W X Y Z a b c d e f g h i j k l m n o p q r s t u v w x y z
<i>digit</i>	0 1 2 3 4 5 6 7 8 9
<i>underscore</i>	_
<i>punctuation</i>	! " # % & ' () * + , - . / : ; < = > ? [\] ^ { } ~

Additional graphic characters in the basic C character set:

Character	Meaning
<i>space</i>	leave blank space
<i>BEL</i>	signal an alert (BEL)

<i>BS</i>	<i>go back one position (BackSpace)</i>
<i>FF</i>	<i>go to top of page (Form Feed)</i>
<i>NL</i>	<i>go to start of next line (NewLine)</i>
<i>CR</i>	<i>go to start of this line (Carriage Return)</i>
<i>HT</i>	<i>go to next Horizontal Tab stop</i>
<i>VT</i>	<i>go to next Vertical Tab stop</i>

The code value zero is reserved for the **null character** which is always in the target character set. Code values for the basic C character set are positive when stored in an object of type *char*. Code values for the digits are contiguous, with increasing value. For example, '0' + 5 equals '5'. Code values for any two letters are *not* necessarily contiguous.

Character Sets and Locales

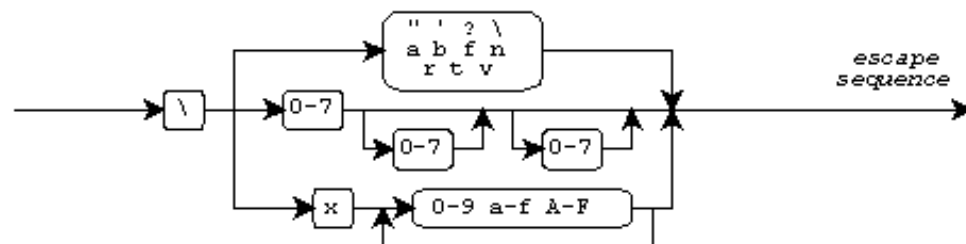
An implementation can support multiple locales, each with a different character set. A locale summarizes conventions peculiar to a given culture, such as how to format dates or how to sort names. To change locales and, therefore, target character sets while the program is running, use the function `setlocale`. The translator encodes character constants and string literals for the "C" locale, which is the locale in effect at program startup.

Escape Sequences

Within character constants and string literals, you can write a variety of **escape sequences**. Each escape sequence determines the code value for a single character. You use escape sequences to represent character codes:

- you cannot otherwise write (such as `\n`)
- that can be difficult to read properly (such as `\t`)
- that might change value in different target character sets (such as `\a`)
- that must not change in value among different target environments (such as `\0`)

An escape sequence takes the form shown in the diagram.



Mnemonic escape sequences help you remember the characters they represent:

Character	Escape Sequence
"	<code>\"</code>
'	<code>\'</code>
?	<code>\?</code>
\	<code>\\</code>
<i>BEL</i>	<code>\a</code>
<i>BS</i>	<code>\b</code>
<i>FF</i>	<code>\f</code>
<i>NL</i>	<code>\n</code>
<i>CR</i>	<code>\r</code>
<i>HT</i>	<code>\t</code>
<i>VT</i>	<code>\v</code>

Numeric Escape Sequences

You can also write **numeric escape sequences** using either octal or hexadecimal digits. An **octal escape sequence** takes one of the forms:

`\d` or `\dd` or `\ddd`

The escape sequence yields a code value that is the numeric value of the 1-, 2-, or 3-digit octal number following the backslash (`\`). Each *d* can be any digit in the range 0-7.

A **hexadecimal escape sequence** takes one of the forms:

`\xh` or `\xhh` or ...

The escape sequence yields a code value that is the numeric value of the arbitrary-length hexadecimal number following the backslash (`\`). Each *h* can be any decimal digit 0-9, or any of the letters a-f or A-F. The letters represent the digit values 10-15, where either a or A has the value 10.

A numeric escape sequence terminates with the first character that does not fit the digit pattern. Here are some examples:

- You can write the null character (page 10) as `'\0'`.
- You can write a newline character (*NL*) within a string literal by writing: `"hi\n"` which becomes the array `{ 'h', 'i', '\n', 0 }`
- You can write a string literal that begins with a specific numeric value: `"\3abc"` which becomes the array `{ 3, 'a', 'b', 'c', 0 }`
- You can write a string literal that contains the hexadecimal escape sequence `\xF` followed by the digit 3 by writing two string literals: `"\xF" "3"` which becomes the array `{ 0xF, '3', 0 }`

Trigraphs

A **trigraph** is a sequence of three characters that begins with two question marks (`??`). You use trigraphs to write C source files with a character set that does not contain convenient graphic representations for some punctuation characters. (The resultant C source file is not necessarily more readable, but it is unambiguous.)

The list of all **defined trigraphs** is:

Character	Trigraph
[??{
\	??/
]	??}
^	??^
{	??<
	??!
}	??>
~	??~
#	??=

These are the only trigraphs. The translator does not alter any other sequence that begins with two question marks.

For example, the expression statements:

```
printf("Case ??=3 is done??/n");
printf("You said what????/n");
```

are equivalent to:

```
printf("Case #3 is done\n");
printf("You said what??\n");
```

The translator replaces each trigraph with its equivalent single character representation in an early phase of translation (page 50). You can always treat a trigraph as a single source character.

Multibyte Characters

A source character set or target character set can also contain **multibyte characters** (sequences of one or more bytes). Each sequence represents a single character in the **extended character set**. You use multibyte characters to represent large sets of characters, such as Kanji. A multibyte character can be a one-byte sequence that is a character from the basic C character set (page 9), an additional one-byte sequence that is implementation defined, or an additional sequence of two or more bytes that is implementation defined.

Any multibyte encoding that contains sequences of two or more bytes depends, for its interpretation between bytes, on a **conversion state** determined by bytes earlier in the sequence of characters. In the **initial conversion state** if the byte immediately following matches one of the characters in the basic C character set, the byte must represent that character.

For example, the **EUC encoding** is a superset of ASCII. A byte value in the interval [0xA1, 0xFE] is the first of a two-byte sequence (whose second byte value is in the interval [0x80, 0xFF]). All other byte values are one-byte sequences. Since all members of the basic C character set (page 9) have byte values in the range [0x00, 0x7F] in ASCII, EUC meets the requirements for a multibyte encoding in Standard C. Such a sequence is *not* in the initial conversion state immediately after a byte value in the interval [0xA1, 0xFE]. It is ill-formed if a second byte value is not in the interval [0x80, 0xFF].

Multibyte characters can also have a **state-dependent encoding**. How you interpret a byte in such an encoding depends on a conversion state that involves both a **parse state**, as before, and a **shift state**, determined by bytes earlier in the sequence of characters. The **initial shift state**, at the beginning of a new multibyte character, is also the initial conversion state. A subsequent **shift sequence** can determine an **alternate shift state**, after which all byte sequences (including one-byte sequences) can have a different interpretation. A byte containing the value zero, however, always represents the null character (page 10). It cannot occur as any of the bytes of another multibyte character.

For example, the **JIS encoding** is another superset of ASCII. In the initial shift state, each byte represents a single character, except for two three-byte shift sequences:

- The three-byte sequence "\x1B\$B" shifts to two-byte mode. Subsequently, two successive bytes (both with values in the range [0x21, 0x7E]) constitute a single multibyte character.
- The three-byte sequence "\x1B(B" shifts back to the initial shift state.

JIS also meets the requirements for a multibyte encoding in Standard C. Such a sequence is *not* in the initial conversion state when partway through a three-byte shift sequence or when in two-byte mode.

(Amendment 1 adds the type `mbstate_t`, which describes an object that can store a conversion state. It also relaxes the above rules for generalized multibyte characters (page 18), which describe the encoding rules for a broad range of wide streams (page 18).)

You can write multibyte characters in C source text as part of a comment, a character constant, a string literal, or a filename in an *include* (page 50) directive. How such characters print is implementation defined. Each sequence of multibyte characters that you write must begin and end in the initial shift state. The program can also include multibyte characters in null-terminated (page 9) C strings used by several library functions, including the format strings (page 31) for `printf` and `scanf`. Each such character string must begin and end in the initial shift state.

Wide-Character Encoding

Each character in the extended character set also has an integer representation, called a **wide-character encoding**. Each extended character has a unique wide-character value. The value zero always corresponds to the **null wide character**. The type definition `wchar_t` specifies the integer type that represents wide characters.

You write a **wide-character constant** as `L'mbc'`, where `mbc` represents a single multibyte character. You write a **wide-character string literal** as `L"mbs"`, where `mbs` represents a sequence of zero or more multibyte characters. The wide-character string literal `L"xyz"` becomes a sequence of wide-character constants stored in successive bytes of memory, followed by a null wide character:
`{L'x', L'y', L'z', L'\0'}`

The following library functions help you convert between the multibyte and wide-character representations of extended characters: `btowc`, `mblen`, `mbrlen`, `mbrtowc`, `mbsrtowcs`, `mbstowcs`, `mbtowc`, `wcrtomb`, `wcsrtombs`, `wcstombs`, `wctob`, and `wctomb`.

The macro `MB_LEN_MAX` specifies the length of the longest possible multibyte sequence required to represent a single character defined by the implementation across supported locales. And the macro `MB_CUR_MAX` specifies the length of the longest possible multibyte sequence required to represent a single character defined for the current locale.

For example, the string literal (page 9) `"hello"` becomes an array of six *char*:

```
{ 'h', 'e', 'l', 'l', 'o', 0 }
```

while the wide-character string literal `L"hello"` becomes an array of six integers of type `wchar_t`:

```
{ L'h', L'e', L'l', L'l', L'o', 0 }
```

Chapter 4. Expressions

You write expressions to determine values, to alter values stored in objects, and to call functions that perform input and output. In fact, you express all computations in the program by writing expressions. The translator must evaluate some of the expressions you write to determine properties of the program. The translator or the target environment must evaluate other expressions prior to program startup to determine the initial values stored in objects with static duration. The program evaluates the remaining expressions when it executes.

This document describes briefly just those aspect of expressions most relevant to the use of the Standard C library:

An **address constant expression** specifies a value that has a pointer type and that the translator or target environment can determine prior to program startup.

A **constant expression** specifies a value that the translator or target environment can determine prior to program startup.

An **integer constant expression** specifies a value that has an integer type and that the translator can determine at the point in the program where you write the expression. (You cannot write a function call, assigning operator, or *comma* operator except as part of the operand of a *sizeof* (page 16) operator.) In addition, you must write only subexpressions that have integer type. You can, however, write a floating-point constant as the operand of an integer *type cast* operator.

An **lvalue expression** An lvalue expression designates an object that has an object type other than an array type. Hence, you can access the value stored in the object. A *modifiable* lvalue expression designates an object that has an object type other than an array type or a *const* type. Hence, you can alter the value stored in the object. You can also designate objects with an lvalue expression that has an array type or an incomplete type, but you can only take the address of such an expression.

Promoting occurs for an expression whose integer type is not one of the “computational” types. Except when it is the operand of the *sizeof* operator, an integer rvalue expression has one of four types: *int*, *unsigned int*, *long*, or *unsigned long*. When you write an expression in an rvalue context and the expression has an integer type that is not one of these types, the translator *promotes* its type to one of these. If all of the values representable in the original type are also representable as type *int*, then the promoted type is *int*. Otherwise, the promoted type is *unsigned int*. Thus, for *signed char*, *short*, and any *signed bitfield* type, the promoted type is *int*. For each of the remaining integer types (*char*, *unsigned char*, *unsigned short*, any plain *bitfield* type, or any *unsigned bitfield* type), the effect of these rules is to favor promoting to *int* wherever possible, but to promote to *unsigned int* if necessary to preserve the original value in all possible cases.

An **rvalue expression** is an expression whose value can be determined only when the program executes. The term also applies to expressions which *need not* be determined until program execution.

You use the **sizeof** operator, as in the expression `sizeof X` to determine the size in bytes of an object whose type is the type of `X`. The translator uses the expression you write for `X` only to determine a type; it is not evaluated.

A **void expression** has type *void*.

Chapter 5. Files and Streams

Text and Binary Streams (page 17) · Byte and Wide Streams (page 18) · Controlling Streams (page 19) · Stream States (page 20)

A program communicates with the target environment by reading and writing **files** (ordered sequences of bytes). A file can be, for example, a data set that you can read and write repeatedly (such as a disk file), a stream of bytes generated by a program (such as a pipeline), or a stream of bytes received from or sent to a peripheral device (such as the keyboard or display). The latter two are **interactive files**. Files are typically the principal means by which to interact with a program.

You manipulate all these kinds of files in much the same way — by calling library functions. You include the standard header `<stdio.h>` to declare most of these functions.

Before you can perform many of the operations on a file, the file must be **opened**. Opening a file associates it with a **stream**, a data structure within the Standard C library that glosses over many differences among files of various kinds. The library maintains the state of each stream in an object of type **FILE**.

The target environment opens three files prior to program startup. You can open a file by calling the library function `fopen` with two arguments. The first argument is a filename, a multibyte string that the target environment uses to identify which file you want to read or write. The second argument is a C string that specifies:

- whether you intend to read data from the file or write data to it or both
- whether you intend to generate new contents for the file (or create a file if it did not previously exist) or leave the existing contents in place
- whether writes to a file can alter existing contents or should only append bytes at the end of the file
- whether you want to manipulate a text stream (page 17) or a binary stream (page 18)

Once the file is successfully opened, you can then determine whether the stream is **byte oriented** (a **byte stream** (page 18)) or **wide oriented** (a **wide stream** (page 18)). Wide-oriented streams are supported only with Amendment 1. A stream is initially **unbound**. Calling certain functions to operate on the stream makes it byte oriented, while certain other functions make it wide oriented. Once established, a stream maintains its orientation until it is closed by a call to `fclose` or `freopen`.

Text and Binary Streams

A **text stream** consists of one or more **lines** of text that can be written to a text-oriented display so that they can be read. When reading from a text stream, the program reads an *NL* (newline) at the end of each line. When writing to a text stream, the program writes an *NL* to signal the end of a line. To match differing conventions among target environments for representing text in files, the library functions can alter the number and representations of characters transmitted between the program and a text stream.

Thus, positioning within a text stream is limited. You can obtain the current file-position indicator (page 19) by calling `fgetpos` or `ftell`. You can position a text

stream at a position obtained this way, or at the beginning or end of the stream, by calling `fsetpos` or `fseek`. Any other change of position might well be not supported.

For maximum portability, the program should not write:

- empty files
- *space* characters at the end of a line
- partial lines (by omitting the *NL* at the end of a file)
- characters other than the printable characters, *NL*, and *HT* (horizontal tab)

If you follow these rules, the sequence of characters you read from a text stream (either as byte or multibyte characters) will match the sequence of characters you wrote to the text stream when you created the file. Otherwise, the library functions can remove a file you create if the file is empty when you close it. Or they can alter or delete characters you write to the file.

A **binary stream** consists of one or more bytes of arbitrary information. You can write the value stored in an arbitrary object to a (byte-oriented) binary stream and read exactly what was stored in the object when you wrote it. The library functions do not alter the bytes you transmit between the program and a binary stream. They can, however, append an arbitrary number of null bytes to the file that you write with a binary stream. The program must deal with these additional null bytes at the end of any binary stream.

Thus, positioning within a binary stream is well defined, except for positioning relative to the end of the stream. You can obtain and alter the current file-position indicator (page 19) the same as for a text stream (page 17). Moreover, the offsets used by `ftell` and `fseek` count bytes from the beginning of the stream (which is byte zero), so integer arithmetic on these offsets yields predictable results.

A **byte stream** treats a file as a sequence of bytes. Within the program, the stream looks like the same sequence of bytes, except for the possible alterations described above.

Byte and Wide Streams

While a **byte stream** treats a file as a sequence of bytes, a **wide stream** treats a file as a sequence of **generalized multibyte characters**, which can have a broad range of encoding rules. (Text and binary files are still read and written as described above.) Within the program, the stream looks like the corresponding sequence of wide characters (page 13). Conversions between the two representations occur within the Standard C library. The conversion rules can, in principle, be altered by a call to `setlocale` that alters the category `LC_CTYPE`. Each wide stream determines its conversion rules at the time it becomes wide oriented, and retains these rules even if the category `LC_CTYPE` subsequently changes.

Positioning within a wide stream suffers the same limitations as for text streams (page 17). Moreover, the file-position indicator (page 19) may well have to deal with a state-dependent encoding (page 12). Typically, it includes both a byte offset within the stream and an object of type `mbstate_t`. Thus, the only reliable way to obtain a file position within a wide stream is by calling `fgetpos`, and the only reliable way to restore a position obtained this way is by calling `fsetpos`.

Controlling Streams

`fopen` returns the address of an object of type `FILE`. You use this address as the stream argument to several library functions to perform various operations on an open file. For a byte stream, all input takes place as if each character is read by calling `fgetc`, and all output takes place as if each character is written by calling `fputc`. For a wide stream (with Amendment 1), all input takes place as if each character is read by calling `fgetwc`, and all output takes place as if each character is written by calling `fputwc`.

You can **close** a file by calling `fclose`, after which the address of the `FILE` object is invalid.

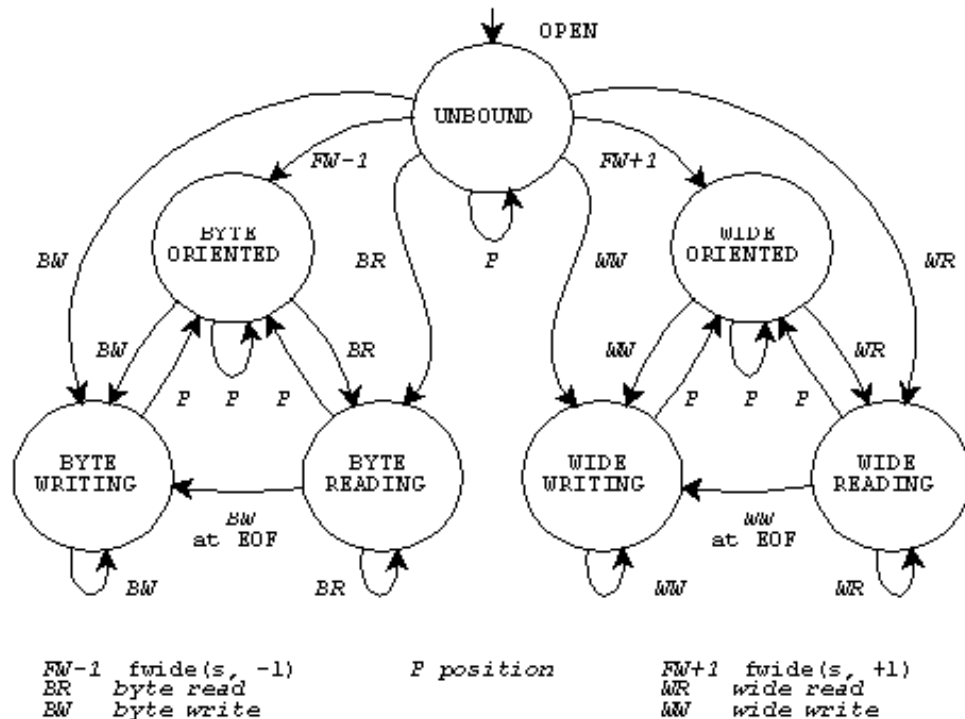
A `FILE` object stores the state of a stream, including:

- an **error indicator** — set nonzero by a function that encounters a read or write error
- an **end-of-file indicator** — set nonzero by a function that encounters the end of the file while reading
- a **file-position indicator** — specifies the next byte in the stream to read or write, if the file can support positioning requests
- a **stream state (page 20)** — specifies whether the stream will accept reads and/or writes and, with Amendment 1, whether the stream is unbound (page 17), byte oriented (page 17), or wide oriented (page 17)
- a **conversion state (page 12)** — remembers the state of any partly assembled or generated generalized multibyte character (page 18), as well as any shift state for the sequence of bytes in the file)
- a **file buffer** — specifies the address and size of an array object that library functions can use to improve the performance of read and write operations to the stream

Do not alter any value stored in a `FILE` object or in a file buffer that you specify for use with that object. You cannot copy a `FILE` object and portably use the address of the copy as a stream argument to a library function.

Stream States

The valid states, and state transitions, for a stream are shown in the diagram.



Each of the circles denotes a stable state. Each of the lines denotes a transition that can occur as the result of a function call that operates on the stream. Five groups of functions can cause state transitions.

Functions in the first three groups are declared in `<stdio.h>`:

- the **byte read functions** — `fgetc`, `fgets`, `fread`, `fscanf`, `getc`, `getchar`, `gets`, `scanf`, and `ungetc`
- the **byte write functions** — `fprintf`, `fputc`, `fputs`, `fwrite`, `printf`, `putc`, `putchar`, `puts`, `vfprintf`, and `vprintf`
- the **position functions** — `fflush`, `fseek`, `fsetpos`, and `rewind`

Functions in the remaining two groups are declared in `<wchar.h>`:

- the **wide read functions** — `fgetwc`, `fgetws`, `fwscanf`, `getwc`, `getwchar`, `ungetwc`, and `wscanf`
- the **wide write functions** — `fwprintf`, `fputwc`, `fputws`, `putwc`, `putwchar`, `vfwprintf`, `vprintf`, and `wprintf`

For the stream `s`, the call `fwide(s, 0)` is always valid and never causes a change of state. Any other call to `fwide`, or to any of the five groups of functions described above, causes the state transition shown in the state diagram. If no such transition is shown, the function call is invalid.

The state diagram shows how to establish the orientation of a stream:

- The call `fwide(s, -1)`, or to a byte read or byte write function, establishes the stream as byte oriented (page 17).

- The call `fwide(s, 1)`, or to a wide read or wide write function, establishes the stream as wide oriented (page 17).

The state diagram shows that you must call one of the position functions between most write and read operations:

- You cannot call a read function if the last operation on the stream was a write.
- You cannot call a write function if the last operation on the stream was a read, unless that read operation set the end-of-file indicator (page 19).

Finally, the state diagram shows that a position operation never *decreases* the number of valid function calls that can follow.

Chapter 6. Functions

You write functions to specify all the actions that a program performs when it executes. The type of a function tells you the type of result it returns (if any). It can also tell you the types of any arguments that the function expects when you call it from within an expression.

This document describes briefly just those aspect of functions most relevant to the use of the Standard C library:

Argument promotion occurs when the type of the function fails to provide any information about an argument. Promotion occurs if the function declaration is not a function prototype or if the argument is one of the unnamed arguments in a varying number of arguments. In this instance, the argument must be an rvalue expression (page 15). Hence:

- An integer argument type is promoted.
- An lvalue of type *array of T* becomes an rvalue of type *pointer to T*.
- A function designator of type *function returning T* becomes an rvalue of type *pointer to function returning T*.
- An argument of type *float* is converted to *double*.

A **do statement** executes a statement one or more times, while its test-context expression (page 24) has a nonzero value:

```
do
    statement
while (test);
```

An **expression statement** evaluates an expression in a side-effects context (page 24):

<code>printf("hello\n");</code>	call a function
<code>y = m * x + b;</code>	store a value
<code>++count;</code>	alter a stored value

A **for statement** executes a statement zero or more times, while the optional test-context expression (page 24) *test* has a nonzero value. You can also write two expressions, *se-1* and *se-2*, in a *for* statement that are each in a side-effects context (page 24):

```
for (se-1; test; se-2)
    statement
```

An **if statement** executes a statement only if the test-context expression (page 24) has a nonzero value:

```
if (test)
    statement
```

An **if-else statement** executes one of two statements, depending on whether the test-context expression (page 24) has a nonzero value:

```
if (test)
    statement-1
else
    statement-2
```

A **return statement** terminates execution of the function and transfers control to the expression that called the function. If you write the optional rvalue expression (page 15) within the *return* statement, the result must be assignment-compatible with the type returned by the function. The program converts the value of the expression to the type returned and returns it as the value of the function call:

```
return expression;
```

An expression that occurs in a **side-effects context** specifies no value and designates no object or function. Hence, it can have type *void*. You typically evaluate such an expression for its **side effects** — any change in the state of the program that occurs when evaluating an expression. Side effects occur when the program stores a value in an object, accesses a value from an object of *volatile* qualified type, or alters the state of a file.

A **switch statement** jumps to a place within a controlled statement, depending on the value of an integer expression:

```
switch (expr)
{
case val-1:
    stat-1;
    break;
case val-2:
    stat-2;           falls through to next
default:
    stat-n
}
```

In a **test-context expression** the value of an expression causes control to flow one way within the statement if the computed value is nonzero or another way if the computed value is zero. You can write only an expression that has a scalar rvalue result, because only scalars can be compared with zero.

A **while statement** executes a statement zero or more times, while the test-context expression has a nonzero value:

```
while (test)
    statement
```

Chapter 7. Formatted Input

Scan Formats (page 25) · Scan Functions (page 25) · Scan Conversion Specifiers (page 26)

Several library functions help you convert data values from text sequences that are generally readable by people to encoded internal representations. You provide a format string (page 31) as the value of the format argument to each of these functions, hence the term **formatted input**. The functions fall into two categories:

The **byte scan functions** (declared in `<stdio.h>`) convert sequences of type *char* to internal representations, and help you scan such sequences that you read: `fscanf`, `scanf`, and `sscanf`. For these functions, a format string is a multibyte string that begins and ends in the initial shift state (page 12).

The **wide scan functions** (declared in `<wchar.h>` and hence added with **Amendment 1**) convert sequences of type *wchar_t*, to internal representations, and help you scan such sequences that you read: `fwscanf`, `wscanf` and `swscanf`. For these functions, a format string is a wide-character string. In the descriptions that follow, a wide character *wc* from a format string or a stream is compared to a specific (byte) character *c* as if by evaluating the expression `wctob(wc) == c`.

Scan Formats

A format string has the same general **syntax (page 31)** for the scan functions as for the print functions (page 31): zero or more **conversion specifications (page 31)**, interspersed with literal text and **white space (page 31)**. For the scan functions, however, a conversion specification is one of the scan conversion specifications (page 25) described below.

A scan function scans the format string once from beginning to end to determine what conversions to perform. Every scan function accepts a varying number of arguments, either directly or under control of an argument of type *va_list*. Some scan conversion specifications in the format string use the next argument in the list. A scan function uses each successive argument no more than once. Trailing arguments can be left unused.

In the description that follows, the integer conversions (page 32) and floating-point conversions (page 32) are the same as for the print functions (page 31).

Scan Functions

For the scan functions, literal text in a format string must match the next characters to scan in the input text. White space in a format string must match the longest possible sequence of the next zero or more white-space characters in the input. Except for the scan conversion specifier (page 26) `%n` (which consumes no input), each **scan conversion specification** determines a pattern that one or more of the next characters in the input must match. And except for the scan conversion specifiers *c*, *n*, and *[]*, every match begins by skipping any white space characters in the input.

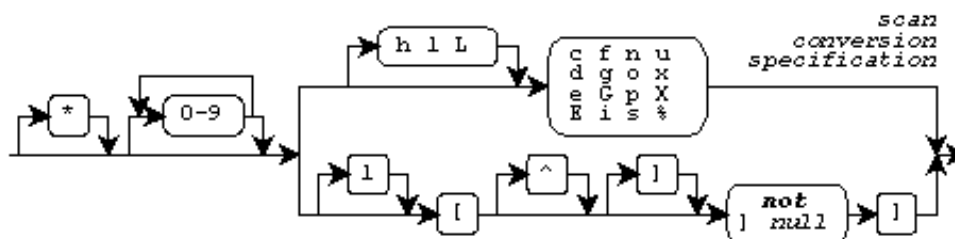
A scan function returns when:

- it reaches the terminating null in the format string
- it cannot obtain additional input characters to scan (**input failure**)
- a conversion fails (**matching failure**)

A scan function returns EOF if an input failure occurs before any conversion. Otherwise it returns the number of converted values stored. If one or more characters form a valid prefix but the conversion fails, the valid prefix is consumed before the scan function returns. Thus:

```
scanf("%i", &i)      consumes 0X from field 0XZ
scanf("%f", &f)      consumes 3.2E from field 3.2EZ
```

A scan conversion specification typically converts the matched input characters to a corresponding encoded value. The next argument value must be the address of an object. The conversion converts the encoded representation (as necessary) and stores its value in the object. A scan conversion specification has the format shown in the diagram.



Following the percent character (%) in the format string, you can write an asterisk (*) to indicate that the conversion should not store the converted value in an object.

Following any *, you can write a nonzero **field width** that specifies the maximum number of input characters to match for the conversion (not counting any white space that the pattern can first skip).

Scan Conversion Specifiers

Following any field width (page 26), you must write a one-character **scan conversion specifier**, either a one-character code or a scan set (page 28), possibly preceded by a one-character qualifier. Each combination determines the type required of the next argument (if any) and how the scan functions interpret the text sequence and converts it to an encoded value. The integer (page 32) and floating-point conversions (page 32) also determine what base to assume for the text representation. (The base is the base argument to the functions `strtol` and `strtoul`.) The following table lists all defined combinations and their properties.

Conversion Specifier	Argument Type	Conversion Function	Base
%c	char x[]		
%lc	wchar_t x[]		
%d	int *x	strtol	10
%hd	short *x	strtol	10
%ld	long *x	strtol	10
%e	float *x	strtod	10
%le	double *x	strtod	10
%Le	long double *x	strtod	10
%E	float *x	strtod	10
%lE	double *x	strtod	10
%LE	long double *x	strtod	10
%f	float *x	strtod	10

%lf	double *x	strtod	10
%Lf	long double *x	strtod	10
%g	float *x	strtod	10
%lg	double *x	strtod	10
%Lg	long double *x	strtod	10
%G	float *x	strtod	10
%LG	double *x	strtod	10
%LG	long double *x	strtod	10
%i	int *x	strtol	0
%hi	short *x	strtol	0
%li	long *x	strtol	0
%n	int *x		
%hn	short *x		
%ln	long *x		
%o	unsigned int *x	strtoul	8
%ho	unsigned short *x	strtoul	8
%lo	unsigned long *x	strtoul	8
%p	void **x		
%s	char x[]		
%ls	wchar_t x[]		
%u	unsigned int *x	strtoul	10
%hu	unsigned short *x	strtoul	10
%lu	unsigned long *x	strtoul	10
%x	unsigned int *x	strtoul	16
%hx	unsigned short *x	strtoul	16
%lx	unsigned long *x	strtoul	16
%X	unsigned int *x	strtoul	16
%hX	unsigned short *x	strtoul	16
%lX	unsigned long *x	strtoul	16
%[...]	char x[]		
%l[...]	wchar_t x[]		
%%	none		

The scan conversion specifier (or scan set (page 28)) determines any behavior not summarized in this table. In the following descriptions, examples follow each of the scan conversion specifiers. In each example, the function `sscanf` matches the **bold** characters.

You write `%c` to store the matched input characters in an array object. If you specify no field width *w*, then *w* has the value one. The match does not skip leading white space (page 31). Any sequence of *w* characters matches the conversion pattern.

```
sscanf("129E-2", "%c", &c)           stores '1'
sscanf("129E-2", "%2c", &c[0])       stores '1', '2'
```

For a wide stream (page 18), conversion occurs as if by repeatedly calling `wcrtomb`, beginning in the initial conversion state (page 12).

```
swscanf(L"129E-2", L"%c", &c)        stores '1'
```

You write `%lc` to store the matched input characters in an array object, with elements of type `wchar_t`. If you specify no field width *w*, then *w* has the value one. The match does not skip leading white space (page 31). Any sequence of *w* characters matches the conversion pattern. For a byte stream (page 18), conversion occurs as if by repeatedly calling `mbrtowc`, beginning in the initial conversion state (page 12).

```
sscanf("129E-2", "%lc", &c)           stores L'1'
sscanf("129E-2", "%2lc", &c)          stores L'1', L'2'
swscanf(L"129E-2", L"%lc", &c)        stores L'1'
```

You write `%d`, `%i`, `%o`, `%u`, `%x`, or `%X` to convert the matched input characters as a signed integer and store the result in an integer object.

```
sscanf("129E-2", "%o%d%x", &i, &j, &k) stores 10, 9, 14
```

You write `%e`, `%E`, `%f`, `%g`, or `%G` to convert the matched input characters as a signed fraction, with an optional exponent, and store the result in a floating-point object.

```
sscanf("129E-2", "%e", &f) stores 1.29
```

You write `%n` to store the number of characters matched (up to this point in the format) in an integer object. The match does not skip leading white space and does not match any input characters.

```
sscanf("129E-2", "%n", &i) stores 2
```

You write `%p` to convert the matched input characters as an external representation of a *pointer to void* and store the result in an object of type *pointer to void*. The input characters must match the form generated by the `%p` print conversion specification (page 32).

```
sscanf("129E-2", "%p", &p) stores, e.g. 0x129E
```

You write `%s` to store the matched input characters in an array object, followed by a terminating null character. If you do not specify a field width *w*, then *w* has a large value. Any sequence of up to *w* non white-space characters matches the conversion pattern.

```
sscanf("129E-2", "%s", &s[0]) stores "129E-2"
```

For a wide stream (page 18), conversion occurs as if by repeatedly calling `wcrtomb` beginning in the initial conversion state (page 12).

```
swscanf(L"129E-2", L"%s", &s[0]) stores L"129E-2"
```

You write `%ls` to store the matched input characters in an array object, with elements of type `wchar_t`, followed by a terminating null wide character. If you do not specify a field width *w*, then *w* has a large value. Any sequence of up to *w* non white-space characters matches the conversion pattern. For a byte stream (page 18), conversion occurs as if by repeatedly calling `mbrtowc`, beginning in the initial conversion state.

```
sscanf("129E-2", "%ls", &s[0]) stores L"129E-2"  
swscanf(L"129E-2", L"%ls", &s[0]) stores L"129E-2"
```

You write `%[` to store the matched input characters in an array object, followed by a terminating null character. If you do not specify a field width *w*, then *w* has a large value. The match does not skip leading white space. A sequence of up to *w* characters matches the conversion pattern in the **scan set** that follows. To complete the scan set, you follow the left bracket (`[`) in the conversion specification with a sequence of zero or more **match** characters, terminated by a right bracket (`]`).

If you do not write a caret (^) immediately after the `[`, then each input character must match *one* of the match characters. Otherwise, each input character must not match *any* of the match characters, which begin with the character following the ^. If you write a `]` immediately after the `[` or `[^`, then the `]` is the first match character, not the terminating `]`. If you write a minus (-) as other than the first or last match character, an implementation can give it special meaning. It usually indicates a range of characters, in conjunction with the characters immediately preceding or following, as in `0-9` for all the digits.) You cannot specify a null match character.

```
sscanf("129E-2", "[%4321]", &s[0]) stores "12"
```

For a wide stream (page 18), conversion occurs as if by repeatedly calling `wcrtomb`, beginning in the initial conversion state.

```
swscanf(L"129E-2", L"[54321]", &s[0]) stores "12"
```

You write `%I[` to store the matched input characters in an array object, with elements of type `wchar_t`, followed by a terminating null wide character. If you do not specify a field width *w*, then *w* has a large value. The match does not skip leading white space. A sequence of up to *w* characters matches the conversion pattern in the scan set (page 28) that follows.

For a byte stream (page 18), conversion occurs as if by repeatedly calling `mbrtowc`, beginning in the initial conversion state.

```
sscanf("129E-2", "[54321]", &s[0]) stores L"12"  
swscanf(L"129E-2", L"[54321]", &s[0]) stores L"12"
```

You write `%%` to match the percent character (%). The function does not store a value.

```
sscanf("% 0xA", "%% %i") stores 10
```

Chapter 8. Formatted Output

Print Formats (page 31) · Print Functions (page 32) · Print Conversion Specifiers (page 33)

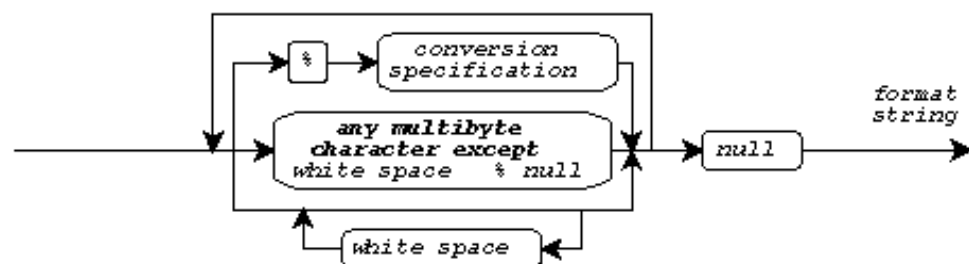
Several library functions help you convert data values from encoded internal representations to text sequences that are generally readable by people. You provide a format string (page 31) as the value of the format argument to each of these functions, hence the term **formatted output**. The functions fall into two categories.

The **byte print functions** (declared in `<stdio.h>`) convert internal representations to sequences of type *char*, and help you compose such sequences for display: `fprintf`, `printf`, `sprintf`, `vfprintf`, `vprintf`, and `vsprintf`. For these function, a format string is a multibyte string that begins and ends in the initial shift state (page 12).

The **wide print functions** (declared in `<wchar.h>` and hence added with **Amendment 1**) convert internal representations to sequences of type `wchar_t`, and help you compose such sequences for display: `fwprintf`, `swprintf`, `wprintf`, `vfwprintf`, `vswprintf`, and `vwprintf`. For these functions, a format string is a wide-character string. In the descriptions that follow, a wide character `wc` from a format string or a stream is compared to a specific (byte) character `c` as if by evaluating the expression `wctob(wc) == c`.

Print Formats

A **format string** has the same syntax for both the print functions and the scan functions (page 25), as shown in the diagram.



A format string consists of zero or more **conversion specifications** interspersed with literal text and **white space**. White space is a sequence of one or more characters `c` for which the call `isspace(c)` returns nonzero. (The characters defined as white space can change when you change the `LC_CTYPE` locale category.) For the print functions, a conversion specification is one of the print conversion specifications (page 32) described below.

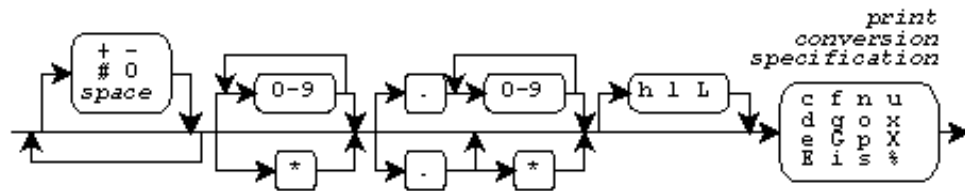
A print function scans the format string once from beginning to end to determine what conversions to perform. Every print function accepts a varying number of arguments, either directly or under control of an argument of type `va_list`. Some print conversion specifications in the format string use the next argument in the list. A print function uses each successive argument no more than once. Trailing arguments can be left unused.

In the description that follows:

- **integer conversions** are the **conversion specifiers** that end in d, i, o, u, x, or X
- **floating-point conversions** are the conversion specifiers that end in e, E, f, g, or G

Print Functions

For the print functions, literal text or white space (page 31) in a format string generates characters that match the characters in the format string. A **print conversion specification** typically generates characters by converting the next argument value to a corresponding text sequence. A print conversion specification has the format:



Following the percent character (%) in the format string, you can write zero or more **format flags**:

- - — to left-justify a conversion
- + — to generate a plus sign for signed values that are positive
- **space** — to generate a *space* for signed values that have neither a plus nor a minus sign
- # — to prefix 0 on an o conversion, to prefix 0x on an x conversion, to prefix 0X on an X conversion, or to generate a decimal point and fraction digits that are otherwise suppressed on a floating-point conversion
- 0 — to pad a conversion with leading zeros after any sign or prefix, in the absence of a minus (-) format flag or a specified precision

Following any format flags, you can write a **field width** that specifies the minimum number of characters to generate for the conversion. Unless altered by a format flag, the default behavior is to pad a short conversion on the left with *space* characters. If you write an asterisk (*) instead of a decimal number for a field width, then a print function takes the value of the next argument (which must be of type *int*) as the field width. If the argument value is negative, it supplies a - format flag and its magnitude is the field width.

Following any field width, you can write a dot (.) followed by a **precision** that specifies one of the following: the minimum number of digits to generate on an integer conversion; the number of fraction digits to generate on an e, E, or f conversion; the maximum number of significant digits to generate on a g or G conversion; or the maximum number of characters to generate from a C string on an s conversion.

If you write an * instead of a decimal number for a precision, a print function takes the value of the next argument (which must be of type *int*) as the precision. If the argument value is negative, the default precision applies. If you do not write either an * or a decimal number following the dot, the precision is zero.

Print Conversion Specifiers

Following any precision (page 32), you must write a one-character **print conversion specifier**, possibly preceded by a one-character qualifier. Each combination determines the type required of the next argument (if any) and how the library functions alter the argument value before converting it to a text sequence. The integer (page 32) and floating-point conversions (page 32) also determine what base to use for the text representation. If a conversion specifier requires a precision *p* and you do not provide one in the format, then the conversion specifier chooses a default value for the precision. The following table lists all defined combinations and their properties.

Conversion Specifier	Argument Type	Converted Value	Default Base	Pre-cision
%c	int x	(unsigned char)x		
%lc	wint_t x	wchar_t a[2] = {x}		
%d	int x	(int)x	10	1
%hd	int x	(short)x	10	1
%ld	long x	(long)x	10	1
%e	double x	(double)x	10	6
%Le	long double x	(long double)x	10	6
%E	double x	(double)x	10	6
%LE	long double x	(long double)x	10	6
%f	double x	(double)x	10	6
%Lf	long double x	(long double)x	10	6
%g	double x	(double)x	10	6
%Lg	long double x	(long double)x	10	6
%G	double x	(double)x	10	6
%LG	long double x	(long double)x	10	6
%i	int x	(int)x	10	1
%hi	int x	(short)x	10	1
%li	long x	(long)x	10	1
%n	int *x			
%hn	short *x			
%ln	long *x			
%o	int x	(unsigned int)x	8	1
%ho	int x	(unsigned short)x	8	1
%lo	long x	(unsigned long)x	8	1
%p	void *x	(void *)x		
%s	char x[]	x[0]...		large
%ls	wchar_t x[]	x[0]...		large
%u	int x	(unsigned int)x	10	1
%hu	int x	(unsigned short)x	10	1
%lu	long x	(unsigned long)x	10	1
%x	int x	(unsigned int)x	16	1
%hx	int x	(unsigned short)x	16	1
%lx	long x	(unsigned long)x	16	1
%X	int x	(unsigned int)x	16	1
%hX	int x	(unsigned short)x	16	1
%lX	long x	(unsigned long)x	16	1
%%	none	'%'		

The print conversion specifier determines any behavior not summarized in this table. In the following descriptions, *p* is the precision. Examples follow each of the print conversion specifiers. A single conversion can generate up to 509 characters.

You write %c to generate a single character from the converted value.

```
printf("%c", 'a')           generates a
printf("<%3c|%-3c>", 'a', 'b') generates < a|b >
```

For a wide stream (page 18), conversion of the character x occurs as if by calling btowc(x).

```
wprintf(L"%c", 'a')           generates btowc(a)
```

You write `%lc` to generate a single character from the converted value. Conversion of the character `x` occurs as if it is followed by a null character in an array of two elements of type `wchar_t` converted by the conversion specification `ls` (page 35).

```
printf("%lc", L'a')           generates a
wprintf(L"%lc", L'a')        generates L'a'
```

You write `%d`, `%i`, `%o`, `%u`, `%x`, or `%X` to generate a possibly signed integer representation. `%d` or `%i` specifies signed decimal representation, `%o` unsigned octal, `%u` unsigned decimal, `%x` unsigned hexadecimal using the digits 0-9 and a-f, and `%X` unsigned hexadecimal using the digits 0-9 and A-F. The conversion generates at least *p* digits to represent the converted value. If *p* is zero, a converted value of zero generates no digits.

```
printf("%d %o %x", 31, 31, 31) generates 31 37 1f
printf("%hu", 0xffff)          generates 65535
printf("%#X %d", 31, 31)       generates 0X1F +31
```

You write `%e` or `%E` to generate a signed fractional representation with an exponent. The generated text takes the form $\pm d.dddE\pm dd$, where \pm is either a plus or minus sign, *d* is a decimal digit, the dot (.) is the decimal point for the current locale, and *E* is either *e* (for `%e` conversion) or *E* (for `%E` conversion). The generated text has one integer digit, a decimal point if *p* is nonzero or if you specify the `#` format flag, *p* fraction digits, and at least two exponent digits. The result is rounded. The value zero has a zero exponent.

```
printf("%e", 31.4)           generates 3.140000e+01
printf("%.2E", 31.4)         generates 3.14E+01
```

You write `%f` to generate a signed fractional representation with no exponent. The generated text takes the form $\pm d.ddd$, where \pm is either a plus or minus sign, *d* is a decimal digit, and the dot (.) is the decimal point for the current locale. The generated text has at least one integer digit, a decimal point if *p* is nonzero or if you specify the `#` format flag, and *p* fraction digits. The result is rounded.

```
printf("%f", 31.4)           generates 31.400000
printf("%.0f %#.0f", 31.0, 31.0) generates 31 31.
```

You write `%g` or `%G` to generate a signed fractional representation with or without an exponent, as appropriate. For `%g` conversion, the generated text takes the same form as either `%e` or `%f` conversion. For `%G` conversion, it takes the same form as either `%E` or `%f` conversion. The precision *p* specifies the number of significant digits generated. (If *p* is zero, it is changed to 1.) If `%e` conversion would yield an exponent in the range $[-4, p)$, then `%f` conversion occurs instead. The generated text has no trailing zeros in any fraction and has a decimal point only if there are nonzero fraction digits, unless you specify the `#` format flag.

```
printf("%.6g", 31.4)         generates 31.4
printf("%.1g", 31.4)         generates 3.14e+01
```

You write `%n` to store the number of characters generated (up to this point in the format) in the object of type `int` whose address is the value of the next successive argument.

```
printf("abc%n", &x)          stores 3
```

You write `%p` to generate an external representation of a *pointer to void*. The conversion is implementation defined.

```
printf("%p", (void *)&x)     generates, e.g. F4C0
```

You write `%s` to generate a sequence of characters from the values stored in the argument C string.

<code>printf("%s", "hello")</code>	generates <code>hello</code>
<code>printf("%.2s", "hello")</code>	generates <code>he</code>

For a wide stream (page 18), conversion occurs as if by repeatedly calling `mbrtowc`, beginning in the initial conversion state (page 12). The conversion generates no more than *p* characters, up to but not including the terminating null character.

<code>wprintf(L"%s", "hello")</code>	generates <code>hello</code>
--------------------------------------	-------------------------------------

You write `%ls` to generate a sequence of characters from the values stored in the argument wide-character string. For a byte stream (page 18), conversion occurs as if by repeatedly calling `wcrtomb`, beginning in the initial conversion state (page 12), so long as complete multibyte characters can be generated. The conversion generates no more than *p* characters, up to but not including the terminating null character.

<code>printf("%ls", L"hello")</code>	generates <code>hello</code>
<code>wprintf(L "%.2s", L"hello")</code>	generates <code>he</code>

You write `%%` to generate the percent character (%).

<code>printf("%%")</code>	generates <code>%</code>
---------------------------	---------------------------------

Chapter 9. STL Conventions

The Standard Template Library (page 1), or STL (page 1), establishes uniform standards for the application of iterators (page 37) to STL containers (page 41) or other sequences that you define, by STL algorithms (page 38) or other functions that you define. This document summarizes many of the conventions used widely throughout the Standard Template Library.

Iterator Conventions

The STL facilities make widespread use of **iterators**, to mediate between the various algorithms and the sequences upon which they act. For brevity in the remainder of this document, the name of an iterator type (or its prefix) indicates the category of iterators required for that type. In order of increasing power, the categories are summarized here as:

- **OutIt** — An **output iterator** X can only have a value V stored indirect on it, after which it *must* be incremented before the next store, as in $(*X++ = V)$, $(*X = V, ++X)$, or $(*X = V, X++)$.
- **InIt** — An **input iterator** X can represent a singular value that indicates end-of-sequence. If an input iterator does not compare equal to its end-of-sequence value, it can have a value V accessed indirect on it any number of times, as in $(V = *X)$. To progress to the next value, or end-of-sequence, you increment it, as in $++X$, $X++$, or $(V = *X++)$. Once you increment *any* copy of an input iterator, none of the other copies can safely be compared, dereferenced, or incremented thereafter.
- **FwdIt** — A **forward iterator** X can take the place of an output iterator (for writing) or an input iterator (for reading). You can, however, read (via $V = *X$) what you just wrote (via $*X = V$) through a forward iterator. And you can make multiple copies of a forward iterator, each of which can be dereferenced and incremented independently.
- **BidIt** — A **bidirectional iterator** X can take the place of a forward iterator. You can, however, also decrement a bidirectional iterator, as in $-X$, $X--$, or $(V = *X--)$.
- **RanIt** — A **random-access iterator** X can take the place of a bidirectional iterator. You can also perform much the same integer arithmetic on a random-access iterator that you can on an object pointer. For N an integer object, you can write $x[N]$, $x + N$, $x - N$, and $N + X$.

Note that an object pointer can take the place of a random-access iterator, or any other for that matter. All iterators can be assigned or copied. They are assumed to be lightweight objects and hence are often passed and returned by value, not by reference. Note also that none of the operations described above can throw an exception, at least when performed on a valid iterator.

The hierarchy of iterator categories can be summarize by showing three sequences. For write-only access to a sequence, you can use any of:

```
output iterator
-> forward iterator
-> bidirectional iterator
-> random-access iterator
```

The right arrow means “can be replaced by.” So any algorithm that calls for an output iterator should work nicely with a forward iterator, for example, but *not* the other way around.

For read-only access to a sequence, you can use any of:

```
input iterator
    -> forward iterator
    -> bidirectional iterator
    -> random-access iterator
```

An input iterator is the weakest of all categories, in this case.

Finally, for read/write access to a sequence, you can use any of:

```
forward iterator
    -> bidirectional iterator
    -> random-access iterator
```

Remember that an object pointer can always serve as a random-access iterator. Hence, it can serve as any category of iterator, so long as it supports the proper read/write access to the sequence it designates.

An iterator *It* other than an object pointer must also define the member types required by the specialization `iterator_traits<It>`. Note that these requirements can be met by deriving *It* from the public base class `iterator`.

This “algebra” of iterators is fundamental to practically everything else in the Standard Template Library (page 1). It is important to understand the promises, and limitations, of each iterator category to see how iterators are used by containers and algorithms in STL.

Algorithm Conventions

The descriptions of the algorithm template functions employ several shorthand phrases:

- The phrase “**in the range [A, B)**” means the sequence of zero or more discrete values beginning with A up to but not including B. A range is valid only if B is **reachable** from A — you can store A in an object N (`N = A`), increment the object zero or more times (`++N`), and have the object compare equal to B after a finite number of increments (`N == B`).
- The phrase “**each N in the range [A, B)**” means that N begins with the value A and is incremented zero or more times until it equals the value B. The case `N == B` is *not* in the range.
- The phrase “**the lowest value of N in the range [A, B) such that X**” means that the condition X is determined for each N in the range [A, B) until the condition X is met.
- The phrase “**the highest value of N in the range [A, B) such that X**” usually means that X is determined for each N in the range [A, B). The function stores in K a copy of N each time the condition X is met. If any such store occurs, the function replaces the final value of N (which equals B) with the value of K. For a bidirectional or random-access iterator, however, it can also mean that N begins with the highest value in the range and is decremented over the range until the condition X is met.
- Expressions such as **X - Y**, where X and Y can be iterators other than random-access iterators, are intended in the mathematical sense. The function

does not necessarily evaluate operator- if it must determine such a value. The same is also true for expressions such as $X + N$ and $X - N$, where N is an integer type.

Several algorithms make use of a predicate, using operator==, that must impose an **equivalence relationship** on pairs of elements from a sequence. For all elements X , Y , and Z :

- $X == X$ is true.
- If $X == Y$ is true, then $Y == X$ is true.
- If $X == Y$ && $Y == Z$ is true, then $X == Z$ is true.

Several algorithms make use of a predicate that must impose a **strict weak ordering** on pairs of elements from a sequence. For the predicate $\text{pr}(X, Y)$:

- “strict” means that $\text{pr}(X, X)$ is false
- “weak” means that X and Y have an **equivalent ordering** if $!\text{pr}(X, Y) \ \&\& \ !\text{pr}(Y, X)$ ($X == Y$ need not be defined)
- “ordering” means that $\text{pr}(X, Y) \ \&\& \ \text{pr}(Y, Z)$ implies $\text{pr}(X, Z)$

Some of these algorithms implicitly use the predicate $X < Y$. Other predicates that typically satisfy the “strict weak ordering” requirement are $X > Y$, $\text{less}(X, Y)$, and $\text{greater}(X, Y)$. Note, however, that predicates such as $X \leq Y$ and $X \geq Y$ do *not* satisfy this requirement.

A sequence of elements designated by iterators in the range $[\text{first}, \text{last})$ is “a **sequence ordered by operator<**” if, for each N in the range $[0, \text{last} - \text{first})$ and for each M in the range $(N, \text{last} - \text{first})$ the predicate $!(*(\text{first} + M) < *(\text{first} + N))$ is true. (Note that the elements are sorted in *ascending* order.) The predicate function $\text{operator}<$, or any replacement for it, must not alter either of its operands. Moreover, it must impose a strict weak ordering (page 39) on the operands it compares.

A sequence of elements designated by iterators in the range $[\text{first}, \text{last})$ is “a **heap ordered by operator<**” if, for each N in the range $[1, \text{last} - \text{first})$ the predicate $!(\text{first} < *(\text{first} + N))$ is true. (The first element is the largest.) Its internal structure is otherwise known only to the template functions make_heap (page 259), pop_heap (page 263), and push_heap (page 264). As with an ordered sequence (page 39), the predicate function $\text{operator}<$, or any replacement for it, must not alter either of its operands, and it must impose a strict weak ordering (page 39) on the operands it compares.

Chapter 10. Containers

```
namespace std {
template<class T>
class Cont;

    // TEMPLATE FUNCTIONS
template<class T>
    bool operator==(
        const Cont<T>& lhs,
        const Cont<T>& rhs);
template<class T>
    bool operator!=(
        const Cont<T>& lhs,
        const Cont<T>& rhs);
template<class T>
    bool operator<(
        const Cont<T>& lhs,
        const Cont<T>& rhs);
template<class T>
    bool operator>(
        const Cont<T>& lhs,
        const Cont<T>& rhs);
template<class T>
    bool operator<=(
        const Cont<T>& lhs,
        const Cont<T>& rhs);
template<class T>
    bool operator>=(
        const Cont<T>& lhs,
        const Cont<T>& rhs);
template<class T>
    void swap(
        Cont<T>& lhs,
        Cont<T>& rhs);
};
```

A **container** is an STL (page 1) template class that manages a sequence of elements. Such elements can be of any object type that supplies a copy constructor, a destructor, and an assignment operator (all with sensible behavior, of course). The destructor may not throw an exception. This document describes the properties required of all such containers, in terms of a generic template class `Cont`. An actual container template class may have additional template parameters. It will certainly have additional member functions.

The STL template container classes are:

- deque (page 274)
- list (page 310)
- map (page 321)
- multimap (page 328)
- multiset (page 354)
- set (page 361)
- vector (page 404)

The Standard C++ library template class `basic_string` also meets the requirements for a template container class.

Cont

[begin](#) (page 44) · [clear](#) (page 44) · [const_iterator](#) (page 44) · [const_reference](#) (page 44) · [const_reverse_iterator](#) (page 44) · [difference_type](#) (page 44) · [empty](#) (page 44) · [end](#) (page 45) · [erase](#) (page 45) · [iterator](#) (page 45) · [max_size](#) (page 45) · [rbegin](#) (page 45) · [reference](#) (page 45) · [rend](#) (page 45) · [reverse_iterator](#) (page 46) · [size](#) (page 46) · [size_type](#) (page 46) · [swap](#) (page 46) · [value_type](#) (page 46)

```
template<class T> >
class Cont {
public:
    typedef T0 size_type;
    typedef T1 difference_type;
    typedef T2 reference;
    typedef T3 const_reference;
    typedef T4 value_type;
    typedef T5 iterator;
    typedef T6 const_iterator;
    typedef T7 reverse_iterator;
    typedef T8 const_reverse_iterator;
    iterator begin();
    const_iterator begin() const;
    iterator end();
    const_iterator end() const;
    reverse_iterator rbegin();
    const_reverse_iterator rbegin() const;
    reverse_iterator rend();
    const_reverse_iterator rend() const;
    size_type size() const;
    size_type max_size() const;
    bool empty() const;
    iterator erase(iterator it);
    iterator erase(iterator first, iterator last);
    void clear();
    void swap(Cont& x);
};
```

The template class describes an object that controls a varying-length sequence of elements, typically of type `T`. The sequence is stored in different ways, depending on the actual container.

A container constructor or member function may call the constructor `T(const T&)` or the function `T::operator=(const T&)`. If such a call throws an exception, the container object is obliged to maintain its integrity, and to rethrow any exception it catches. You can safely swap, assign to, erase, or destroy a container object after it throws one of these exceptions. In general, however, you cannot otherwise predict the state of the sequence controlled by the container object.

A few additional caveats:

- If the expression `~T()` throws an exception, the resulting state of the container object is undefined.
- If the container stores an allocator object `a1`, and `a1` throws an exception other than as a result of a call to `a1.allocate`, the resulting state of the container object is undefined.
- If the container stores a function object `comp`, to determine how to order the controlled sequence, and `comp` throws an exception of any kind, the resulting state of the container object is undefined.

The container classes defined by STL satisfy several additional requirements, as described in the following paragraphs.

Container template class `list` (page 310) provides deterministic, and useful, behavior even in the presence of the exceptions described above. For example, if an exception is thrown during the insertion of one or more elements, the container is left unaltered and the exception is rethrown.

For *all* the container classes defined by STL, if an exception is thrown during calls to the following member functions:

```
insert // single element inserted  
push_back  
push_front
```

the container is left unaltered and the exception is rethrown.

For *all* the container classes defined by STL, no exception is thrown during calls to the following member functions:

```
erase // single element erased  
pop_back  
pop_front
```

Moreover, no exception is thrown while copying an iterator returned by a member function.

The member function `swap` (page 46) makes additional promises for *all* container classes defined by STL:

- The member function throws an exception only if the container stores an allocator object `a1`, and `a1` throws an exception when copied.
- References, pointers, and iterators that designate elements of the controlled sequences being swapped remain valid.

An object of a container class defined by STL allocates and frees storage for the sequence it controls through a stored object of type `A`, which is typically a template parameter. Such an allocator object (page 337) must have the same external interface as an object of class `allocator` (page 337). In particular, `A` must be the same type as `A::rebind<value_type>::other`

For *all* container classes defined by STL, the member function:

```
A get_allocator() const;
```

returns a copy of the stored allocator object. Note that the stored allocator object is *not* copied when the container object is assigned. All constructors initialize the value stored in `allocator`, to `A()` if the constructor contains no allocator parameter.

According to the C++ Standard (page 417) a container class defined by STL can assume that:

- All objects of class `A` compare equal.
- Type `A::const_pointer` is the same as `const T *`.
- Type `A::const_reference` is the same as `const T&`.
- Type `A::pointer` is the same as `T *`.
- Type `A::reference` is the same as `T&`.

In this implementation (page 3), however, containers do *not* make such simplifying assumptions. Thus, they work properly with allocator objects that are more ambitious:

- All objects of class A need not compare equal. (You can maintain multiple pools of storage.)
- Type `A::const_pointer` need not be the same as `const T *`. (A pointer can be a class.)
- Type `A::pointer` need not be the same as `T *`. (A const pointer can be a class.)

Cont::begin

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

The member function returns an iterator that points at the first element of the sequence (or just beyond the end of an empty sequence).

Cont::clear

```
void clear();
```

The member function calls `erase(begin(), end())`.

Cont::const_iterator

```
typedef T6 const_iterator;
```

The type describes an object that can serve as a constant iterator for the controlled sequence. It is described here as a synonym for the unspecified type T6.

Cont::const_reference

```
typedef T3 const_reference;
```

The type describes an object that can serve as a constant reference to an element of the controlled sequence. It is described here as a synonym for the unspecified type T3 (typically `A::const_reference`).

Cont::const_reverse_iterator

```
typedef T8 const_reverse_iterator;
```

The type describes an object that can serve as a constant reverse iterator for the controlled sequence. It is described here as a synonym for the unspecified type T8 (typically `reverse_iterator <const_iterator>`).

Cont::difference_type

```
typedef T1 difference_type;
```

The signed integer type describes an object that can represent the difference between the addresses of any two elements in the controlled sequence. It is described here as a synonym for the unspecified type T1 (typically `A::difference_type`).

Cont::empty

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

The member function returns true for an empty controlled sequence.

Cont::end

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

The member function returns an iterator that points just beyond the end of the sequence.

Cont::erase

```
iterator erase(iterator it);  
iterator erase(iterator first, iterator last);
```

The first member function removes the element of the controlled sequence pointed to by `it`. The second member function removes the elements of the controlled sequence in the range `[first, last)`. Both return an iterator that designates the first element remaining beyond any elements removed, or `end()` if no such element exists.

The member functions never throw an exception.

Cont::iterator

```
typedef T5 iterator;
```

The type describes an object that can serve as an iterator for the controlled sequence. It is described here as a synonym for the unspecified type `T5`. An object of type `iterator` can be cast to an object of type `const_iterator` (page 44).

Cont::max_size

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

The member function returns the length of the longest sequence that the object can control, in constant time regardless of the length of the controlled sequence.

Cont::rbegin

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

The member function returns a reverse iterator that points just beyond the end of the controlled sequence. Hence, it designates the beginning of the reverse sequence.

Cont::reference

```
typedef T2 reference;
```

The type describes an object that can serve as a reference to an element of the controlled sequence. It is described here as a synonym for the unspecified type `T2` (typically `A::reference`). An object of type `reference` can be cast to an object of type `const_reference` (page 44).

Cont::rend

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

The member function returns a reverse iterator that points at the first element of the sequence (or just beyond the end of an empty sequence). Hence, it designates the end of the reverse sequence.

Cont::reverse_iterator

```
typedef T7 reverse_iterator;
```

The type describes an object that can serve as a reverse iterator for the controlled sequence. It is described here as a synonym for the unspecified type T7 (typically `reverse_iterator <iterator>`).

Cont::size

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

The member function returns the length of the controlled sequence, in constant time regardless of the length of the controlled sequence.

Cont::size_type

```
typedef T0 size_type;
```

The unsigned integer type describes an object that can represent the length of any controlled sequence. It is described here as a synonym for the unspecified type T0 (typically `A::size_type`).

Cont::swap

```
void swap(Cont& x);
```

The member function swaps the controlled sequences between `*this` and `x`. If `get_allocator() == x.get_allocator()`, it does so in constant time. Otherwise, it performs a number of element assignments and constructor calls proportional to the number of elements in the two controlled sequences.

Cont::value_type

```
typedef T4 value_type;
```

The type is a synonym for the template parameter T. It is described here as a synonym for the unspecified type T4 (typically `A::value_type`).

operator!=

```
template<class T>
bool operator!=(
    const Cont <T>& lhs,
    const Cont <T>& rhs);
```

The template function returns `!(lhs == rhs)`.

operator==

```
template<class T>
bool operator==(
    const Cont <T>& lhs,
    const Cont <T>& rhs);
```

The template function overloads `operator==` to compare two objects of template class `Cont` (page 42). The function returns `lhs.size() == rhs.size() && equal(lhs.begin(), lhs.end(), rhs.begin())`.

operator<

```
template<class T>
    bool operator<(
        const Cont <T>& lhs,
        const Cont <T>& rhs);
```

The template function overloads `operator<` to compare two objects of template class `Cont`. The function returns `lexicographical_compare(lhs.begin(), lhs.end(), rhs.begin(), rhs.end())`.

operator<=

```
template<class T>
    bool operator<=(
        const Cont <T>& lhs,
        const Cont <T>& rhs);
```

The template function returns `!(rhs < lhs)`.

operator>

```
template<class T>
    bool operator>(
        const Cont <T>& lhs,
        const Cont <T>& rhs);
```

The template function returns `rhs < lhs`.

operator>=

```
template<class T>
    bool operator>=(
        const Cont <T>& lhs,
        const Cont <T>& rhs);
```

The template function returns `!(lhs < rhs)`.

swap

```
template<class T>
    void swap(
        Cont <T>& lhs,
        Cont <T>& rhs);
```

The template function executes `lhs.swap (page 46) (rhs)`.

Portions derived from work copyright © 1994 by Hewlett-Packard Company. All rights reserved.

Chapter 11. Preprocessing

The translator processes each source file in a series of phases. **Preprocessing** constitutes the earliest phases, which produce a translation unit (page 50). Preprocessing treats a source file as a sequence of text lines (page 17). You can specify **directives** and **macros** that insert, delete, and alter source text.

This document describes briefly just those aspect of preprocessing most relevant to the use of the Standard C library:

The macro `__FILE__` expands to a string literal (page 9) that gives the remembered filename of the current source file. You can alter the value of this macro by writing a *line* directive.

The macro `__LINE__` expands to a decimal integer constant that gives the remembered line number within the current source file. You can alter the value of this macro by writing a *line* directive.

A *define* **directive** defines a name as a macro. Following the directive name `define`, you write one of two forms:

- a name *not* immediately followed by a left parenthesis, followed by any sequence of preprocessing tokens — to define a macro without parameters
- a name immediately followed by a left parenthesis with *no* intervening white space, followed by zero or more distinct *parameter names* separated by commas, followed by a right parenthesis, followed by any sequence of preprocessing tokens — to define a macro with as many parameters as names that you write inside the parentheses

You can selectively skip groups of lines within source files by writing an *if* **directive**, or one of the other **conditional directives**, *ifdef* or *ifndef*. You follow the conditional directive by the first group of lines that you want to selectively skip. Zero or more *elif* directives follow this first group of lines, each followed by a group of lines that you want to selectively skip. An optional *else* directive follows all groups of lines controlled by *elif* directives, followed by the last group of lines you want to selectively skip. The last group of lines ends with an *endif* directive.

At most one group of lines is retained in the translation unit — the one immediately preceded by a directive whose if expression (page 49) has a nonzero value. For the directive:

```
#ifdef X
```

this expression is defined (X), and for the directive:

```
#ifndef X
```

this expression is !defined (X).

An **if expression** is a conditional expression that the preprocessor evaluates. You can write only integer constant expressions (page 15), with the following additional considerations:

- The expression `defined X`, or `defined (X)`, is replaced by 1 if X is defined as a macro, otherwise 0.

- You cannot write the *sizeof* (page 16) or *type cast* operators. (The translator expands all macro names, then replaces each remaining name with 0, before it recognizes keywords.)
- The translator may be able to represent a broader range of integers than the target environment.
- The translator represents type *int* the same as *long*, and *unsigned int* the same as *unsigned long*.
- The translator can translate character constants to a set of code values different from the set for the target environment.

An **include directive** includes the contents of a standard header or another source file in a translation unit. The contents of the specified standard header or source file replace the *include* directive. Following the directive name *include*, write one of the following:

- a standard header name between angle brackets
- a filename between double quotes
- any other form that expands to one of the two previous forms after macro replacement

A **line directive** alters the source line number and filename used by the predefined macros `__FILE__` (page 49) and `__LINE__`. Following the directive name *line*, write one of the following:

- a decimal integer (giving the new line number of the line following)
- a decimal integer as before, followed by a string literal (giving the new line number and the new source filename)
- any other form that expands to one of the two previous forms after macro replacement

Preprocessing translates each source file in a series of distinct **phases**. The first few phases of translation: terminate each line with a newline character (*NL*), convert trigraphs to their single-character equivalents, and concatenate each line ending in a backslash (`\`) with the line following. Later phases process include directives (page 50), expand macros, and so on to produce a **translation unit**. The translator combines separate translation units, with contributions as needed from the Standard C library, at **link time**, to form the executable **program**.

An **undef directive** removes a macro definition. You might want to remove a macro definition so that you can define it differently with a *define* directive or to unmask any other meaning given to the name. The name whose definition you want to remove follows the directive name *undef*. If the name is not currently defined as a macro, the *undef* directive has no effect.




Chapter 12. Standard C++ Library Header Files

The Standard C++ Library is composed of eight special-purpose libraries:

- The Language Support Library
- The Diagnostics Library
- The General Utilities Library
- The Standard String Templates
- Localization Classes and Templates
- The Containers, Iterators and Algorithms Libraries (the Standard Template Library)
- The Standard Numerics Library
- The Standard Input/Output Library
- C++ Headers for the Standard C Library



The Language Support Library

The Language Support Library defines types and functions that will be used implicitly by C++ programs that employ such C++ language features as operators new and delete, exception handling and runtime type information (RTTI).

Standard C++ header	Equivalent in previous versions
<exception> (page 74)	 <stdexcept.h>  no equivalent
<limits> (page 114)	no equivalent
<new> (page 164)	<new.h>
<typeinfo> (page 224)	<typeinfo.h>  no equivalent

The Diagnostics Library

The Diagnostics Library is used to detect and report error conditions in C++ programs.

Standard C++ header	Equivalent in previous versions
<stdexcept> (page 184)	 <stdexcept.h>  no equivalent

The General Utilities Library

The General Utilities Library is used by other components of the Standard C++ Library, especially the Containers, Iterators and Algorithms Libraries (the Standard Template Library).

Standard C++ header	Equivalent in previous versions
<utility> (page 400)	no equivalent
<functional> (page 282)	no equivalent

<code><memory></code> (page 336)	no equivalent
--	---------------

The Standard String Templates

The Strings Library is a facility for the manipulation of character sequences.

Standard C++ header	Equivalent in previous versions
<code><string></code> (page 195)	no equivalent

Localization Classes and Templates

The Localization Library permits a C++ program to address the cultural differences of its various users.

Standard C++ header	Equivalent in previous versions
<code><locale></code> (page 119)	no equivalent

The Containers, Iterators and Algorithms Libraries (the Standard Template Library)

The Standard Template Library (STL) is a facility for the management and manipulation of collections of objects.

Standard C++ header	Equivalent in previous versions
<code><algorithm></code> (page 249)	no equivalent
<code><bitset></code> (page 54)	no equivalent
<code><deque></code> (page 274)	no equivalent
<code><iterator></code> (page 293)	no equivalent
<code><list></code> (page 310)	no equivalent
<code><map></code> (page 320)	no equivalent
<code><queue></code> (page 353)	no equivalent
<code><set></code> (page 353)	no equivalent
<code><stack></code> (page 368)	no equivalent
<code><unordered_map></code> (page 371)	no equivalent
<code><unordered set></code> (page 386)	no equivalent
<code><vector></code> (page 403)	no equivalent

The Standard Numerics Library

The Numerics Library is a facility for performing seminumerical operations.

Users who require library facilities for complex arithmetic but wish to maintain compatibility with older compilers may use the compatibility complex numbers library whose types are defined in the non-standard header file `<complex.h>`. Although the header files `<complex>` and `<complex.h>` are similar in purpose, they are mutually incompatible.

Standard C++ header	Equivalent in previous versions
<code><complex></code> (page 61)	no equivalent
<code><numeric></code> (page 345)	no equivalent
<code><valarray></code> (page 225)	no equivalent

The Standard Input/Output Library

The standard iostreams library differs from the compatibility iostreams in a number of important respects. To maintain compatibility between such a product and VisualAge C++ Version 5.0 or z/OS C/C++ Version 1.2, use instead the compatibility iostreams library.

Standard C++ header	Equivalent in previous versions
<fstream> (page 76)	no equivalent
<iomanip> (page 85)	no equivalent
<ios> (page 86)	no equivalent
<iosfwd> (page 102)	no equivalent
<iostream> (page 103)	no equivalent
<istream> (page 105)	no equivalent
<ostream> (page 168)	no equivalent
<streambuf> (page 185)	no equivalent
<sstream> (page 176)	no equivalent

C++ Headers for the Standard C Library

The C International Standard specifies 18 headers which must be provided by a conforming hosted implementation. The name of each of these headers is of the form *name.h*. The C++ Standard Library includes the C Standard Library and, hence, includes these 18 headers. Additionally, for each of the 18 headers specified by the C International Standard, the C++ standard specifies a corresponding header that is functionally equivalent to its C library counterpart, but which locates all of the declarations that it contains within the std namespace. The name of each of these C++ headers is of the form *cname*, where *name* is the string that results when the ".h" extension is removed from the name of the equivalent C Standard Library header. For example, the header files <stdlib.h> and <cstdlib> are both provided by the C++ Standard Library and are equivalent in function, with the exception that all declarations in <cstdlib> are located within the std namespace.

Standard C++ Header	Corresponding Standard C & C++ Header
<cassert> (page 59)	<assert.h>
<cctype> (page 59)	<ctype.h>
<cerrno> (page 59)	<errno.h>
<cfloat> (page 59)	<float.h>
<ciso646> (page 60)	<iso646.h>
<climits> (page 60)	<limits.h>
<clocale> (page 60)	<locale.h>
<cmath> (page 60)	<math.h>
<csetjmp> (page 72)	<setjmp.h>
<csignal> (page 72)	<signal.h>
<cstdarg> (page 72)	<stdarg.h>
<cstddef> (page 72)	<stddef.h>
<cstdio> (page 72)	<stdio.h>
<cstdlib> (page 73)	<stdlib.h>

<cstring> (page 73)	<string.h>
<ctime> (page 73)	<time.h>
<wchar> (page 74)	<wchar.h>
<cwctype> (page 74)	<wctype.h>

<bitset>

```

namespace std {
template<size_t N>
    class bitset;

    // TEMPLATE FUNCTIONS
template<class E, class T, size_t N>
    basic_istream<E, T>&
        operator>>(basic_istream<E, T>& is,
                    bitset<N>& x);
template<class E, class T, size_t N>
    basic_ostream<E, T>&
        operator<<(basic_ostream<E, T>& os,
                    const bitset<N>& x);
};

```

Include the standard header **<bitset>** to define the template class **bitset** and two supporting templates.

bitset

any (page 55) · at (page 55) · **bitset** (page 55) · **bitset_size** (page 56) · **count** (page 56) · **element_type** (page 56) · **flip** (page 56) · **none** (page 56) · **operator!=** (page 56) · **operator&=** (page 56) · **operator<<** (page 56) · **operator<=** (page 56) · **operator==** (page 56) · **operator>>** (page 57) · **operator>=** (page 57) · **operator[]** (page 57) · **operator^=** (page 57) · **operator|=** (page 57) · **operator~** (page 57) · **reference** (page 57) · **reset** (page 58) · **set** (page 58) · **size** (page 58) · **test** (page 58) · **to_string** (page 58) · **to_ulong** (page 58)

```

template<size_t N>
    class bitset {
public:
    typedef bool element_type;
    class reference;
    bitset();
    bitset(unsigned long val);
    template<class E, class T, class A>
        explicit bitset(const basic_string<E, T, A>& str,
                        typename basic_string<E, T, A>::size_type
                            pos = 0,
                        typename basic_string<E, T, A>::size_type
                            n = basic_string<E, T, A>::npos);
    bitset<N>& operator&=(const bitset<N>& rhs);
    bitset<N>& operator|=(const bitset<N>& rhs);
    bitset<N>& operator^=(const bitset<N>& rhs);
    bitset<N>& operator<=(const bitset<N>& pos);
    bitset<N>& operator>=(const bitset<N>& pos);
    bitset<N>& set();
    bitset<N>& set(size_t pos, bool val = true);
    bitset<N>& reset();
    bitset<N>& reset(size_t pos);
    bitset<N>& flip();
    bitset<N>& flip(size_t pos);
    reference operator[](size_t pos);
    bool operator[](size_t pos) const;
    reference at(size_t pos);
};

```

```

bool at(size_t pos) const;
unsigned long to_ulong() const;
template<class E, class T, class A>
    basic_string<E, T, A> to_string() const;
size_t count() const;
size_t size() const;
bool operator==(const bitset<N>& rhs) const;
bool operator!=(const bitset<N>& rhs) const;
bool test(size_t pos) const;
bool any() const;
bool none() const;
bitset<N> operator<<(size_t pos) const;
bitset<N> operator>>(size_t pos) const;
bitset<N> operator~();
static const size_t bitset_size = N;
};

```

The template class describes an object that stores a sequence of N bits. A bit is **set** if its value is 1, **reset** if its value is 0. To **flip** a bit is to change its value from 1 to 0 or from 0 to 1. When converting between an object of class `bitset<N>` and an object of some integral type, bit position j corresponds to the bit value $1 \ll j$. The integral value corresponding to two or more bits is the sum of their bit values.

bitset::any

```
bool any() const;
```

The member function returns true if any bit is set in the bit sequence.

bitset::at

```
bool at(size_type pos) const;
reference at(size_type pos);
```

The member function returns an object of class `reference` (page 57), which designates the bit at position `pos`, if the object can be modified. Otherwise, it returns the value of the bit at position `pos` in the bit sequence. If that position is invalid, the function throws an object of class `out_of_range` (page 185).

bitset::bitset

```

bitset();
bitset(unsigned long val);
template<class E, class T, class A>
    explicit bitset(const basic_string<E, T, A>& str,
        typename basic_string<E, T, A>::size_type
            pos = 0,
        typename basic_string<E, T, A>::size_type
            n = basic_string<E, T, A>::npos);

```

The first constructor resets all bits in the bit sequence. The second constructor sets only those bits at position j for which `val & 1 << j` is nonzero.

The third constructor determines the initial bit values from elements of a string determined from `str`. If `str.size() < pos`, the constructor throws an object of class `out_of_range` (page 185). Otherwise, the effective length of the string `rlen` is the smaller of `n` and `str.size() - pos`. If any of the `rlen` elements beginning at position `pos` is other than 0 or 1, the constructor throws an object of class `invalid_argument` (page 184). Otherwise, the constructor sets only those bits at position j for which the element at position `pos + j` is 1.

bitset::bitset_size

```
static const size_t bitset_size = N;
```

The const static member is initialized to the template parameter N.

bitset::count

```
size_t count() const;
```

The member function returns the number of bits set in the bit sequence.

bitset::element_type

```
typedef bool element_type;
```

The type is a synonym for bool.

bitset::flip

```
bitset<N>& flip();  
bitset<N>& flip(size_t pos);
```

The first member function flips all bits in the bit sequence, then returns **this*. The second member function throws `out_of_range` (page 185) if `size() <= pos`. Otherwise, it flips the bit at position `pos`, then returns **this*.

bitset::none

```
bool none() const;
```

The member function returns true if none of the bits are set in the bit sequence.

bitset::operator!=

```
bool operator !=(const bitset<N>& rhs) const;
```

The member operator function returns true only if the bit sequence stored in **this* differs from the one stored in *rhs*.

bitset::operator&=

```
bitset<N>& operator&=(const bitset<N>& rhs);
```

The member operator function replaces each element of the bit sequence stored in **this* with the logical AND of its previous value and the corresponding bit in *rhs*. The function returns **this*.

bitset::operator<<

```
bitset<N> operator<<(const bitset<N>& pos);
```

The member operator function returns `bitset(*this) <= pos`.

bitset::operator<=

```
bitset<N>& operator<=(const bitset<N>& pos);
```

The member operator function replaces each element of the bit sequence stored in **this* with the element `pos` positions earlier in the sequence. If no such earlier element exists, the function clears the bit. The function returns **this*.

bitset::operator==

```
bool operator ==(const bitset<N>& rhs) const;
```


The member operator function returns true only if the bit sequence stored in **this* is the same as the one stored in *rhs*.

bitset::operator>>

```
bitset<N> operator>>(const bitset<N>& pos);
```

The member operator function returns `bitset(*this) >>= (page 57) pos`.

bitset::operator>>=

```
bitset<N>& operator>>=(const bitset<N>& pos);
```

The member function replaces each element of the bit sequence stored in **this* with the element *pos* positions later in the sequence. If no such later element exists, the function clears the bit. The function returns **this*.

bitset::operator[]

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

The member function returns an object of class `reference`, which designates the bit at position *pos*, if the object can be modified. Otherwise, it returns the value of the bit at position *pos* in the bit sequence. If that position is invalid, the behavior is undefined.

bitset::operator^=

```
bitset<N>& operator^=(const bitset<N>& rhs);
```

The member operator function replaces each element of the bit sequence stored in **this* with the logical EXCLUSIVE OR of its previous value and the corresponding bit in *rhs*. The function returns **this*.

bitset::operator|=

```
bitset<N>& operator|=(const bitset<N>& rhs);
```

The member operator function replaces each element of the bit sequence stored in **this* with the logical OR of its previous value and the corresponding bit in *rhs*. The function returns **this*.

bitset::operator~

```
bitset<N> operator~();
```

The member operator function returns `bitset(*this).flip()`.

bitset::reference

```
class reference {  
public:  
    reference& operator=(bool b);  
    reference& operator=(const reference& x);  
    bool operator~() const;  
    operator bool() const;  
    reference& flip();  
};
```

The member class describes an object that designates an individual bit within the bit sequence. Thus, for *b* an object of type `bool`, *x* and *y* objects of type `bitset<N>`, and *i* and *j* valid positions within such an object, the member functions of class `reference` ensure that (in order):

- `x[i] = b` stores *b* at bit position *i* in *x*

- `x[i] = y[j]` stores the value of the bit `y[j]` at bit position `i` in `x`
- `b = ~x[i]` stores the flipped value of the bit `x[i]` in `b`
- `b = x[i]` stores the value of the bit `x[i]` in `b`
- `x[i].flip()` stores the flipped value of the bit `x[i]` back at bit position `i` in `x`

bitset::reset

```
bitset<N>& reset();
bitset<N>& reset(size_t pos);
```

The first member function resets (or clears) all bits in the bit sequence, then returns `*this`. The second member function throws `out_of_range` if `size() <= pos`. Otherwise, it resets the bit at position `pos`, then returns `*this`.

bitset::set

```
bitset<N>& set();
bitset<N>& set(size_t pos, bool val = true);
```

The first member function sets all bits in the bit sequence, then returns `*this`. The second member function throws `out_of_range` if `size() <= pos`. Otherwise, it stores `val` in the bit at position `pos`, then returns `*this`.

bitset::size

```
size_t size() const;
```

The member function returns `N`.

bitset::test

```
bool test(size_t pos, bool val = true);
```

The member function throws `out_of_range` (page 185) if `size() <= pos`. Otherwise, it returns `true` only if the bit at position `pos` is set.

bitset::to_string

```
template<class E, class T, class A>
    basic_string<E, T, A> to_string() const;
```

The member function constructs `str`, an object of class `basic_string<E, T, A>`. For each bit in the bit sequence, the function appends 1 if the bit is set, otherwise 0. The *last* element appended to `str` corresponds to bit position zero. The function returns `str`.

bitset::to_ulong

```
unsigned long to_ulong() const;
```

The member function throws `overflow_error` (page 185) if any bit in the bit sequence has a bit value that cannot be represented as a value of type *unsigned long*. Otherwise, it returns the sum of the bit values in the bit sequence.

operator<<

```
template<class E, class T, size_t N>
    basic_ostream<E, T>&
        operator<<(basic_ostream<E, T>& os,
                    const bitset<N>& x);
```

The template function overloads `operator<<` to insert a text representation of the bit sequence in `os`. It effectively executes `os << x.to_string<E, T, allocator<E>>(),` then returns `os`.

operator>>

```
template<class E, class T, size_t N>
    basic_istream<E, T>&
        operator>>(basic_istream<E, T>& is,
                    bitset<N>& x);
```

The template function overloads `operator>>` to store in `x` the value `bitset(str)`, where `str` is an object of type `basic_string<E, T, allocator<E>>&` extracted from `is`. The function extracts elements and appends them to `str` until:

- `N` elements have been extracted and stored
- end-of-file occurs on the input sequence
- the next input element is neither 0 nor 1, in which case the input element is not extracted

If the function stores no characters in `str`, it calls `is.setstate(ios_base::failbit)`. In any case, it returns `is`.

<cassert>

```
#include <cassert.h>
```

Include the standard header `<cassert>` to effectively include the standard header `<assert.h>`.

<cctype>

```
#include <cctype.h>

namespace std {
    using ::isalnum; using ::isalpha; using ::iscntrl;
    using ::isdigit; using ::isgraph; using ::islower;
    using ::isprint; using ::ispunct; using ::isspace;
    using ::isupper; using ::isxdigit; using ::tolower;
    using ::toupper;
};
```

Include the standard header **<cctype>** to effectively include the standard header `<cctype.h>` within the `std` namespace (page 6).

<cerrno>

```
#include <errno.h>
```

Include the standard header `<cerrno>` to effectively include the standard header `<errno.h>`.

<cfloat>

```
#include <float.h>
```

Include the standard header `<cfloat>` to effectively include the standard header `<float.h>`.

<ciso646>

```
#include <iso646.h>
```

Include the standard header **<ciso646>** to effectively include the standard header `<iso646.h>`.

<climits>

```
#include <limits.h>
```

Include the standard header **<climits>** to effectively include the standard header `<limits.h>`.

<locale>

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

```
namespace std {  
    using ::lconv; using ::localeconv; using ::setlocale;  
};
```

Include the standard header **<locale>** to effectively include the standard header `<locale.h>` within the std namespace (page 6).

<cmath>

```
#include <math.h>
```

```
namespace std {  
    using ::abs; using ::acos; using ::asin;  
    using ::atan; using ::atan2; using ::ceil;  
    using ::cos; using ::cosh; using ::exp;  
    using ::fabs; using ::floor; using ::fmod;  
    using ::frexp; using ::ldexp; using ::log;  
    using ::log10; using ::modf; using ::pow;  
    using ::sin; using ::sinh; using ::sqrt;  
    using ::tan; using ::tanh;  
  
    using ::acosf; using ::asinf;  
    using ::atanf; using ::atan2f; using ::ceilf;  
    using ::cosf; using ::coshf; using ::expf;  
    using ::fabsf; using ::floorf; using ::fmodf;  
    using ::frexpf; using ::ldexpf; using ::logf;  
    using ::log10f; using ::modff; using ::powf;  
    using ::sinf; using ::sinhf; using ::sqrtf;  
    using ::tanf; using ::tanhf;  
  
    using ::acosl; using ::asinxl;  
    using ::atanl; using ::atan2l; using ::ceil;  
    using ::cosl; using ::coshl; using ::expl;  
    using ::fabsl; using ::floorl; using ::fmodl;  
    using ::frexpl; using ::ldexpl; using ::logl;  
    using ::log10l; using ::modfl; using ::powl;  
    using ::sinl; using ::sinhl; using ::sqrtl;  
    using ::tanl; using ::tanhl;  
};
```

Include the standard header **<cmath>** to effectively include the standard header `<math.h>` within the std namespace (page 6).

<complex>

abs (page 63) · arg (page 63) · complex (page 63) · complex<double> (page 66) · complex<float> (page 67) · complex<long double> (page 67) · conj (page 67) · cos (page 67) · cosh (page 67) · exp (page 68) · imag (page 68) · log (page 68) · log10 (page 68) · norm (page 68) · operator!= (page 68) · operator* (page 68) · operator+ (page 69) · operator- (page 69) · operator/ (page 69) · operator<< (page 69) · operator== (page 70) · operator>> (page 70) · polar (page 70) · pow (page 70) · real (page 71) · sin (page 71) · sinh (page 71) · sqrt (page 71) · tan (page 71) · tanh (page 71) · __STD_COMPLEX (page 71)

```
namespace std {
#define __STD_COMPLEX

    // TEMPLATE CLASSES
    template<class T>
        class complex;
    template<>
        class complex<float>;
    template<>
        class complex<double>;
    template<>
        class complex<long double>;

    // TEMPLATE FUNCTIONS
    template<class T>
        complex<T> operator+(const complex<T>& lhs,
                               const complex<T>& rhs);
    template<class T>
        complex<T> operator+(const complex<T>& lhs,
                               const T& rhs);
    template<class T>
        complex<T> operator+(const T& lhs,
                               const complex<T>& rhs);
    template<class T>
        complex<T> operator-(const complex<T>& lhs,
                               const complex<T>& rhs);
    template<class T>
        complex<T> operator-(const complex<T>& lhs,
                               const T& rhs);
    template<class T>
        complex<T> operator-(const T& lhs,
                               const complex<T>& rhs);
    template<class T>
        complex<T> operator*(const complex<T>& lhs,
                               const complex<T>& rhs);
    template<class T>
        complex<T> operator*(const complex<T>& lhs,
                               const T& rhs);
    template<class T>
        complex<T> operator*(const T& lhs,
                               const complex<T>& rhs);
    template<class T>
        complex<T> operator/(const complex<T>& lhs,
                               const complex<T>& rhs);
    template<class T>
        complex<T> operator/(const complex<T>& lhs,
                               const T& rhs);
    template<class T>
        complex<T> operator/(const T& lhs,
                               const complex<T>& rhs);
    template<class T>
        complex<T> operator+(const complex<T>& lhs);
    template<class T>
        complex<T> operator-(const complex<T>& lhs);
    template<class T>
```

```

        bool operator==(const complex<T>& lhs,
            const complex<T>& rhs);
template<class T>
    bool operator==(const complex<T>& lhs,
        const T& rhs);
template<class T>
    bool operator==(const T& lhs,
        const complex<T>& rhs);
template<class T>
    bool operator!=(const complex<T>& lhs,
        const complex<T>& rhs);
template<class T>
    bool operator!=(const complex<T>& lhs,
        const T& rhs);
template<class T>
    bool operator!=(const T& lhs,
        const complex<T>& rhs);
template<class U, class E, class T>
    basic_istream<E, T>&
        operator>>(basic_istream<E, T>& is,
        complex<U>& x);
template<class U, class E, class T>
    basic_ostream<E, T>&
        operator<<(basic_ostream<E, T>& os,
        const complex<U>& x);
template<class T>
    T real(const complex<T>& x);
template<class T>
    T imag(const complex<T>& x);
template<class T>
    T abs(const complex<T>& x);
template<class T>
    T arg(const complex<T>& x);
template<class T>
    T norm(const complex<T>& x);
template<class T>
    complex<T> conj(const complex<T>& x);
template<class T>
    complex<T> polar(const T& rho, const T& theta = 0);
template<class T>
    complex<T> cos(const complex<T>& x);
template<class T>
    complex<T> cosh(const complex<T>& x);
template<class T>
    complex<T> exp(const complex<T>& x);
template<class T>
    complex<T> log(const complex<T>& x);
template<class T>
    complex<T> log10(const complex<T>& x);
template<class T>
    complex<T> pow(const complex<T>& x, int y);
template<class T>
    complex<T> pow(const complex<T>& x, const T& y);
template<class T>
    complex<T> pow(const complex<T>& x,
        const complex<T>& y);
template<class T>
    complex<T> pow(const T& x, const complex<T>& y);
template<class T>
    complex<T> sin(const complex<T>& x);
template<class T>
    complex<T> sinh(const complex<T>& x);
template<class T>
    complex<T> sqrt(const complex<T>& x);
};

```

```
template<class T>
    T abs(const complex<T>& x);
```

arg

```
template<class T>
    T arg(const complex<T>& x);
```

complex

Chapter 12. Standard C++ Library Header Files 63

```

friend bool
    operator!=(const complex<T>& lhs, const T& rhs);
friend bool
    operator!=(const T& lhs, const complex<T>& rhs);
};

```

The template class describes an object that stores two objects of type **T**, one that represents the real part of a complex number and one that represents the imaginary part. An object of class **T**:

- has a public default constructor, destructor, copy constructor, and assignment operator — with conventional behavior
- can be assigned integer or floating-point values, or type cast to such values — with conventional behavior
- defines the arithmetic operators and math functions, as needed, that are defined for the floating-point types — with conventional behavior

In particular, no subtle differences may exist between copy construction and default construction followed by assignment. And none of the operations on objects of class **T** may throw exceptions.

Explicit specializations of template class **complex** exist for the three floating-point types. In this implementation (page 3), a value of any other type **T** is type cast to *double* for actual calculations, with the *double* result assigned back to the stored object of type **T**.

complex::complex

```

complex(const T& re = 0, const T& im = 0);
template<class U>
    complex(const complex<U>& x);

```

The first constructor initializes the stored real part to *re* and the stored imaginary part to *im*. The second constructor initializes the stored real part to *x.real()* and the stored imaginary part to *x.imag()*.

In this implementation (page 3), if a translator does not support member template functions, the template:

```

template<class U>
    complex(const complex<U>& x);

```

is replaced by:

```

complex(const complex& x);

```

which is the copy constructor.

complex::imag

```

T imag() const;

```

The member function returns the stored imaginary part.

complex::operator*=

```

template<class U>
    complex& operator*=(const complex<U>& rhs);
complex& operator*=(const T& rhs);

```

The first member function replaces the stored real and imaginary parts with those corresponding to the complex product of **this* and *rhs*. It then returns **this*.

The second member function multiplies both the stored real part and the stored imaginary part with rhs. It then returns *this.

In this implementation (page 3), if a translator does not support member template functions, the template:

```
template<class U>
    complex& operator==(const complex<U>& rhs);
```

is replaced by:

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

complex::operator+=

```
template<class U>
    complex& operator+=(const complex<U>& rhs);
complex& operator+=(const T& rhs);
```

The first member function replaces the stored real and imaginary parts with those corresponding to the complex sum of *this and rhs. It then returns *this.

The second member function adds rhs to the stored real part. It then returns *this.

In this implementation (page 3), if a translator does not support member template functions, the template:

```
template<class U>
    complex& operator+=(const complex<U>& rhs);
```

is replaced by:

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

complex::operator-=

```
template<class U>
    complex& operator-(const complex<U>& rhs);
complex& operator-(const T& rhs);
```

The first member function replaces the stored real and imaginary parts with those corresponding to the complex difference of *this and rhs. It then returns *this.

The second member function subtracts rhs from the stored real part. It then returns *this.

In this implementation (page 3), if a translator does not support member template functions, the template:

```
template<class U>
    complex& operator-(const complex<U>& rhs);
```

is replaced by:

```
complex& operator-(const complex& rhs);
```

complex::operator/=

```
template<class U>
    complex& operator/=(const complex<U>& rhs);
complex& operator/=(const T& rhs);
```

The first member function replaces the stored real and imaginary parts with those corresponding to the complex quotient of *this and rhs. It then returns *this.

The second member function multiplies both the stored real part and the stored imaginary part with `rhs`. It then returns `*this`.

In this implementation (page 3), if a translator does not support member template functions, the template:

```
template<class U>
    complex& operator/(const complex<U>& rhs);
```

is replaced by:

```
complex& operator/(const complex& rhs);
```

complex::operator=

```
template<class U>
    complex& operator=(const complex<U>& rhs);
complex& operator=(const T& rhs);
```

The first member function replaces the stored real part with `rhs.real()` and the stored imaginary part with `rhs.imag()`. It then returns `*this`.

The second member function replaces the stored real part with `rhs` and the stored imaginary part with zero. It then returns `*this`.

In this implementation (page 3), if a translator does not support member template functions, the template:

```
template<class U>
    complex& operator=(const complex<U>& rhs);
```

is replaced by:

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

which is the default assignment operator.

complex::real

```
T real() const;
```

The member function returns the stored real part.

complex::value_type

```
typedef T value_type;
```

The type is a synonym for the template parameter `T`.

complex<double>

```
template<>
    class complex<double> {
public:
    complex(double re = 0, double im = 0);
    complex(const complex<float>& x);
    explicit complex(const complex<long double>& x);
    // rest same as template class complex
    };
```

The explicitly specialized template class describes an object that stores two objects of type **double**, one that represents the real part of a complex number and one that represents the imaginary part. The explicit specialization differs only in the constructors it defines. The first constructor initializes the stored real part to `re` and

the stored imaginary part to `im`. The remaining two constructors initialize the stored real part to `x.real()` and the stored imaginary part to `x.imag()`.

complex<float>

```
template<>
class complex<float> {
public:
    complex(float re = 0, float im = 0);
    explicit complex(const complex<double>& x);
    explicit complex(const complex<long double>& x);
    // rest same as template class complex
};
```

The explicitly specialized template class describes an object that stores two objects of type **float**, one that represents the real part of a complex number and one that represents the imaginary part. The explicit specialization differs only in the constructors it defines. The first constructor initializes the stored real part to `re` and the stored imaginary part to `im`. The remaining two constructors initialize the stored real part to `x.real()` and the stored imaginary part to `x.imag()`.

complex<long double>

```
template<>
class complex<long double> {
public:
    complex(long double re = 0, long double im = 0);
    complex(const complex<float>& x);
    complex(const complex<double>& x);
    // rest same as template class complex
};
```

The explicitly specialized template class describes an object that stores two objects of type **long double**, one that represents the real part of a complex number and one that represents the imaginary part. The explicit specialization differs only in the constructors it defines. The first constructor initializes the stored real part to `re` and the stored imaginary part to `im`. The remaining two constructors initialize the stored real part to `x.real()` and the stored imaginary part to `x.imag()`.

conj

```
template<class T>
complex<T> conj(const complex<T>& x);
```

The function returns the conjugate of `x`.

cos

```
template<class T>
complex<T> cos(const complex<T>& x);
```

The function returns the cosine of `x`.

cosh

```
template<class T>
complex<T> cosh(const complex<T>& x);
```

The function returns the hyperbolic cosine of `x`.

exp

```
template<class T>
    complex<T> exp(const complex<T>& x);
```

The function returns the exponential of x.

imag

```
template<class T>
    T imag(const complex<T>& x);
```

The function returns the imaginary part of x.

log

```
template<class T>
    complex<T> log(const complex<T>& x);
```

The function returns the logarithm of x. The branch cuts are along the negative real axis.

log10

```
template<class T>
    complex<T> log10(const complex<T>& x);
```

The function returns the base 10 logarithm of x. The branch cuts are along the negative real axis.

norm

```
template<class T>
    T norm(const complex<T>& x);
```

The function returns the squared magnitude of x.

operator!=

```
template<class T>
    bool operator!=(const complex<T>& lhs,
                    const complex<T>& rhs);
template<class T>
    bool operator!=(const complex<T>& lhs,
                    const T& rhs);
template<class T>
    bool operator!=(const T& lhs,
                    const complex<T>& rhs);
```

The operators each return true only if `real(lhs) != real(rhs) || imag(lhs) != imag(rhs)`.

operator*

```
template<class T>
    complex<T> operator*(const complex<T>& lhs,
                        const complex<T>& rhs);
template<class T>
    complex<T> operator*(const complex<T>& lhs,
                        const T& rhs);
template<class T>
    complex<T> operator*(const T& lhs,
                        const complex<T>& rhs);
```

The operators each convert both operands to the return type, then return the complex product of the converted lhs and rhs.

operator+

```
template<class T>
    complex<T> operator+(const complex<T>& lhs,
        const complex<T>& rhs);
template<class T>
    complex<T> operator+(const complex<T>& lhs,
        const T& rhs);
template<class T>
    complex<T> operator+(const T& lhs,
        const complex<T>& rhs);
template<class T>
    complex<T> operator+(const complex<T>& lhs);
```

The binary operators each convert both operands to the return type, then return the complex sum of the converted lhs and rhs.

The unary operator returns lhs.

operator-

```
template<class T>
    complex<T> operator-(const complex<T>& lhs,
        const complex<T>& rhs);
template<class T>
    complex<T> operator-(const complex<T>& lhs,
        const T& rhs);
template<class T>
    complex<T> operator-(const T& lhs,
        const complex<T>& rhs);
template<class T>
    complex<T> operator-(const complex<T>& lhs);
```

The binary operators each convert both operands to the return type, then return the complex difference of the converted lhs and rhs.

The unary operator returns a value whose real part is $-\text{real}(\text{lhs})$ and whose imaginary part is $-\text{imag}(\text{lhs})$.

operator/

```
template<class T>
    complex<T> operator/(const complex<T>& lhs,
        const complex<T>& rhs);
template<class T>
    complex<T> operator/(const complex<T>& lhs,
        const T& rhs);
template<class T>
    complex<T> operator/(const T& lhs,
        const complex<T>& rhs);
```

The operators each convert both operands to the return type, then return the complex quotient of the converted lhs and rhs.

operator<<

```
template<class U, class E, class T>
    basic_ostream<E, T>&
        operator<<(basic_ostream<E, T>& os,
        const complex<U>& x);
```

The template function inserts the complex value `x` in the output stream `os`, effectively by executing:

```
basic_ostringstream<E, T> ostr;
ostr.flags(os.flags());
ostr.imbue(os.imbue());
ostr.precision(os.precision());
ostr << '(' << real(x) << ','
    << imag(x) << ')';
os << ostr.str().c_str();
```

Thus, if `os.width()` is greater than zero, any padding occurs either before or after the parenthesized pair of values, which itself contains no padding. The function returns `os`.

operator==

```
template<class T>
    bool operator==(const complex<T>& lhs,
                    const complex<T>& rhs);
template<class T>
    bool operator==(const complex<T>& lhs,
                    const T& rhs);
template<class T>
    bool operator==(const T& lhs,
                    const complex<T>& rhs);
```

The operators each return true only if `real(lhs) == real(rhs) && imag(lhs) == imag(rhs)`.

operator>>

```
template<class U, class E, class T>
    basic_istream<E, T>&
        operator>>(basic_istream<E, T>& is,
                    complex<U>& x);
```

The template function attempts to extract a complex value from the input stream `is`, effectively by executing:

```
is >> ch && ch == '('
    && is >> re >> ch && ch == ','
    && is >> im >> ch && ch == ')'
```

Here, `ch` is an object of type `E`, and `re` and `im` are objects of type `U`.

If the result of this expression is true, the function stores `re` in the real part and `im` in the imaginary part of `x`. In any event, the function returns `is`.

polar

```
template<class T>
    complex<T> polar(const T& rho,
                    const T& theta = 0);
```

The function returns the complex value whose magnitude is `rho` and whose phase angle is `theta`.

pow

```
template<class T>
    complex<T> pow(const complex<T>& x, int y);
template<class T>
    complex<T> pow(const complex<T>& x,
```

```

        const T& y);
template<class T>
    complex<T> pow(const complex<T>& x,
        const complex<T>& y);
template<class T>
    complex<T> pow(const T& x,
        const complex<T>& y);

```

The functions each effectively convert both operands to the return type, then return the converted x to the power y . The branch cut for x is along the negative real axis.

real

```

template<class T>
    T real(const complex<T>& x);

```

The function returns the real part of x .

sin

```

template<class T>
    complex<T> sin(const complex<T>& x);

```

The function returns the sine of x .

sinh

```

template<class T>
    complex<T> sinh(const complex<T>& x);

```

The function returns the hyperbolic sine of x .

sqrt

```

template<class T>
    complex<T> sqrt(const complex<T>& x);

```

The function returns the square root of x , with phase angle in the half-open interval $(-\pi/2, \pi/2]$. The branch cuts are along the negative real axis.

__STD_COMPLEX

```

#define __STD_COMPLEX

```

The macro is defined, with an unspecified expansion, to indicate compliance with the specifications of this header.

tan

```

template<class T>
    complex<T> tan(const complex<T>& x);

```

The function returns the tangent of x .

tanh

```

template<class T>
    complex<T> tanh(const complex<T>& x);

```

The function returns the hyperbolic tangent of x .

<csetjmp>

```
#include <setjmp.h>

namespace std {
    using ::jmp_buf; using ::longjmp;
};
```

Include the standard header **<csetjmp>** to effectively include the standard header **<setjmp.h>** within the std namespace (page 6).

<csignal>

```
#include <signal.h>

namespace std {
    using ::sig_atomic_t; using ::raise; using ::signal;
};
```

Include the standard header **<csignal>** to effectively include the standard header **<signal.h>** within the std namespace (page 6).

<cstdarg>

```
#include <stdarg.h>

namespace std {
    using ::va_list;
};
```

Include the standard header **<cstdarg>** to effectively include the standard header **<stdarg.h>** within the std namespace (page 6).

<cstddef>

```
#include <stddef.h>

namespace std {
    using ::ptrdiff_t; using ::size_t;
};
```

Include the standard header **<cstddef>** to effectively include the standard header **<stddef.h>** within the std namespace (page 6).

<cstdio>

```
#include <stdio.h>

namespace std {
    using ::size_t; using ::fpos_t; using ::FILE;
    using ::clearerr; using ::fclose; using ::feof;
    using ::ferror; using ::fflush; using ::fgetc;
    using ::fgetpos; using ::fgets; using ::fopen;
    using ::fprintf; using ::fputc; using ::fputs;
    using ::fread; using ::freopen; using ::fscanf;
    using ::fseek; using ::fsetpos; using ::ftell;
    using ::fwrite; using ::gets; using ::perror;
    using ::printf; using ::puts; using ::remove;
    using ::rename; using ::rewind; using ::scanf;
    using ::setbuf; using ::setvbuf; using ::sprintf;
```



```
using ::sscanf; using ::tmpfile; using ::tmpnam;
using ::ungetc; using ::vfprintf; using ::vprintf;
using ::vsprintf;
};
```

Include the standard header **<cstdio>** to effectively include the standard header **<stdio.h>** within the std namespace (page 6).

<cstdlib>

```
#include <stdlib.h>

namespace std {
using ::size_t; using ::div_t; using ::ldiv_t;
using ::abort; using ::abs; using ::atexit;
using ::atof; using ::atoi; using ::atol;
using ::bsearch; using ::calloc; using ::div;
using ::exit; using ::free; using ::getenv;
using ::labs; using ::ldiv; using ::malloc;
using ::mblen; using ::mbstowcs; using ::mbtowc;
using ::qsort; using ::rand; using ::realloc;
using ::srand; using ::strtod; using ::strtol;
using ::strtoul; using ::system;
using ::wcstombs; using ::wctomb;
};
```

Include the standard header **<cstdlib>** to effectively include the standard header **<stdlib.h>** within the std namespace (page 6).

<cstring>

```
#include <string.h>

namespace std {
using ::size_t; using ::memcmp; using ::memcpy;
using ::memmove; using ::memset; using ::strcat;
using ::strcmp; using ::strcoll; using ::strcpy;
using ::strcspn; using ::strerror; using ::strlen;
using ::strncat; using ::strncmp; using ::strncpy;
using ::strspn; using ::strtok; using ::strxfrm;
};
```

Include the standard header **<cstring>** to effectively include the standard header **<string.h>** within the std namespace (page 6).

<ctime>

```
#include <time.h>

namespace std {
using ::clock_t; using ::size_t;
using ::time_t; using ::tm;
using ::asctime; using ::clock; using ::ctime;
using ::difftime; using ::gmtime; using ::localtime;
using ::mktime; using ::strftime; using ::time;
};
```

Include the standard header **<ctime>** to effectively include the standard header **<time.h>** within the std namespace (page 6).

<cwchar>

```
#include <wchar.h>

namespace std {
    using ::mbstate_t; using ::size_t; using ::wint_t;
    using ::fgetwc; using ::fgetws; using ::fputwc;
    using ::fputws; using ::fwide; using ::fwprintf;
    using ::fwscanf; using ::getwc; using ::getwchar;
    using ::mbrlen; using ::mbrtowc; using ::mbsrtowcs;
    using ::mbsinit; using ::putwc; using ::putwchar;
    using ::swprintf; using ::swscanf; using ::ungetwc;
    using ::vfwprintf; using ::vswprintf; using ::vwprintf;
    using ::wcrctomb; using ::wprintf; using ::wscanf;
    using ::wcsrtombs; using ::wcstol; using ::wcscat;
    using ::wcschr; using ::wcscmp; using ::wcscoll;
    using ::wcscpy; using ::wcscspn; using ::wcslen;
    using ::wcsncat; using ::wcsncmp; using ::wcsncpy;
    using ::wcpbrk; using ::wcsrchr; using ::wcssp;
    using ::wcsstr; using ::wcstok; using ::wcsxfrm;
    using ::wmemchr; using ::wmemcmp; using ::wmemcpy;
    using ::wmemmove; using ::wmemset; using ::wcsftime;
};
```

Include the standard header **<cwchar>** to effectively include the standard header **<wchar.h>** within the std namespace (page 6).

<cwctype>

```
#include <wctype.h>

namespace std {
    using ::wint_t; using ::wctrans_t; using ::wctype_t;
    using ::iswalnum; using ::iswalpha; using ::iswcntrl;
    using ::iswctype; using ::iswdigit; using ::iswgraph;
    using ::iswlower; using ::iswprint; using ::iswpunct;
    using ::iswspace; using ::iswupper; using ::iswxdigit;
    using ::towctrans; using ::tolower; using ::toupper;
    using ::wctrans; using ::wctype;
};
```

Include the standard header **<cwctype>** to effectively include the standard header **<wctype.h>** within the std namespace (page 6).

<exception>

```
namespace std {
    class exception;
    class bad_exception;

    // FUNCTIONS
    typedef void (*terminate_handler)();
    typedef void (*unexpected_handler)();
    terminate_handler
        set_terminate(terminate_handler ph) throw();
    unexpected_handler
        set_unexpected(unexpected_handler ph) throw();
    void terminate();
    void unexpected();
    bool uncaught_exception();
};
```

Include the standard header **<exception>** to define several types and functions related to the handling of exceptions.

bad_exception

```
class bad_exception : public exception {  
};
```

The class describes an exception that can be thrown from an unexpected handler (page 76). The value returned by `what()` is an implementation-defined C string. None of the member functions throw any exceptions.

exception

```
class exception {  
public:  
    exception() throw();  
    exception(const exception& rhs) throw();  
    exception& operator=(const exception& rhs) throw();  
    virtual ~exception() throw();  
    virtual const char *what() const throw();  
};
```

The class serves as the base class for all exceptions thrown by certain expressions and by the Standard C++ library. The C string value returned by `what()` is left unspecified by the default constructor, but may be defined by the constructors for certain derived classes as an implementation-defined C string.

None of the member functions throw any exceptions.

set_terminate

```
terminate_handler  
    set_terminate(terminate_handler ph) throw();
```

The function establishes a new terminate handler (page 75) as the function `*ph`. Thus, `ph` must not be a null pointer. The function returns the address of the previous terminate handler.

set_unexpected

```
unexpected_handler  
    set_unexpected(unexpected_handler ph) throw();
```

The function establishes a new unexpected handler (page 76) as the function `*ph`. Thus, `ph` must not be a null pointer. The function returns the address of the previous unexpected handler.

terminate

```
void terminate();
```

The function calls a **terminate handler**, a function of type `void ()`. If `terminate` is called directly by the program, the terminate handler is the one most recently set by a call to `set_terminate` (page 75). If `terminate` is called for any of several other reasons during evaluation of a throw expression, the terminate handler is the one in effect immediately after evaluating the throw expression.

A terminate handler may not return to its caller. At program startup, the terminate handler is a function that calls `abort()`.

terminate_handler

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

The type describes a pointer to a function suitable for use as a terminate handler (page 75).

uncaught_exception

```
bool uncaught_exception();
```

The function returns true only if a thrown exception is being currently processed. Specifically, it returns true after completing evaluation of a throw expression and before completing initialization of the exception declaration in the matching handler or calling `unexpected` (page 76) as a result of the throw expression.

unexpected

```
void unexpected();
```

The function calls an **unexpected handler**, a function of type `void ()`. If `unexpected` is called directly by the program, the unexpected handler is the one most recently set by a call to `set_unexpected` (page 75). If `unexpected` is called when control leaves a function by a thrown exception of a type not permitted by an **exception specification** for the function, as in:

```
void f() throw()      // function may throw no exceptions
    {throw "bad"; }   // throw calls unexpected()
```

the unexpected handler is the one in effect immediately after evaluating the throw expression.

An unexpected handler may not return to its caller. It may terminate execution by:

- throwing an object of a type listed in the exception specification (or an object of any type if the unexpected handler is called directly by the program)
- throwing an object of type `bad_exception`
- calling `terminate()`, `abort()`, or `exit(int)`

At program startup, the unexpected handler is a function that calls `terminate()`.

unexpected_handler

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

The type describes a pointer to a function suitable for use as an unexpected handler.

<fstream>

```
namespace std {
template<class E, class T = char_traits<E> >
    class basic_filebuf;
typedef basic_filebuf<char> filebuf;
typedef basic_filebuf<wchar_t> wfilebuf;
template<class E, class T = char_traits<E> >
    class basic_ifstream;
typedef basic_ifstream<char> ifstream;
typedef basic_ifstream<wchar_t> wifstream;
template<class E, class T = char_traits<E> >
    class basic_ofstream;
typedef basic_ofstream<char> ofstream;
typedef basic_ofstream<wchar_t> wofstream;
template<class E, class T = char_traits<E> >
```

```

class basic_fstream;
typedef basic_fstream<char> fstream;
typedef basic_fstream<wchar_t> wfstream;
};

```

Include the iostreams (page 7) standard header **<fstream>** to define several classes that support iostreams operations on sequences stored in external files (page 17).

basic_filebuf

```

template <class E, class T = char_traits<E> >
class basic_filebuf : public basic_streambuf<E, T> {
public:
    typedef typename basic_streambuf<E, T>::char_type
        char_type;
    typedef typename basic_streambuf<E, T>::traits_type
        traits_type;
    typedef typename basic_streambuf<E, T>::int_type
        int_type;
    typedef typename basic_streambuf<E, T>::pos_type
        pos_type;
    typedef typename basic_streambuf<E, T>::off_type
        off_type;
    basic_filebuf();
    bool is_open() const;
    basic_filebuf *open(const char *s,
        ios_base::openmode mode);
    basic_filebuf *close();
protected:
    virtual pos_type seekoff(off_type off,
        ios_base::seekdir way,
        ios_base::openmode which =
            ios_base::in | ios_base::out);
    virtual pos_type seekpos(pos_type pos,
        ios_base::openmode which =
            ios_base::in | ios_base::out);
    virtual int_type underflow();
    virtual int_type pbackfail(int_type c =
        traits_type::eof());
    virtual int_type overflow(int_type c =
        traits_type::eof());
    virtual int sync();
    virtual basic_streambuf<E, T>
        *setbuf(E *s, streamsize n);
};

```

The template class describes a **stream buffer (page 187)** that controls the transmission of elements of type E, whose character traits (page 211) are determined by the class T, to and from a sequence of elements stored in an external file (page 17).

An object of class **basic_filebuf<E, T>** stores a **file pointer**, which designates the FILE object that controls the **stream (page 17)** associated with an open (page 17) file. It also stores pointers to two **file conversion facets (page 79)** for use by the protected member functions **overflow** (page 79) and **underflow** (page 81).

basic_filebuf::basic_filebuf

```
basic_filebuf();
```

The constructor stores a null pointer in all the pointers controlling the input buffer (page 187) and the output buffer (page 187). It also stores a null pointer in the file pointer (page 77).

basic_filebuf::char_type

```
typedef E char_type;
```

The type is a synonym for the template parameter E.

basic_filebuf::close

```
basic_filebuf *close();
```

The member function returns a null pointer if the file pointer (page 77) `fp` is a null pointer. Otherwise, it calls `fclose(fp)`. If that function returns a nonzero value, the function returns a null pointer. Otherwise, it returns `this` to indicate that the file was successfully closed (page 19).

For a wide stream, if any insertions have occurred since the stream was opened, or since the last call to `streampos`, the function calls `overflow()`. It also inserts any sequence needed to restore the initial conversion state (page 12), by using the file conversion facet (page 79) `fac` to call `fac.unshift` as needed. Each element `x` of type *char* thus produced is written to the associated stream designated by the file pointer `fp` as if by successive calls of the form `fputc(x, fp)`. If the call to `fac.unshift` or any write fails, the function does not succeed.

basic_filebuf::int_type

```
typedef typename traits_type::int_type int_type;
```

The type is a synonym for `traits_type::int_type`.

basic_filebuf::is_open

```
bool is_open();
```

The member function returns true if the file pointer is not a null pointer.

basic_filebuf::off_type

```
typedef typename traits_type::off_type off_type;
```

The type is a synonym for `traits_type::off_type`.

basic_filebuf::open

```
basic_filebuf *open(const char *s,  
    ios_base::openmode mode);
```

The member function endeavors to open the file with filename `s`, by calling `fopen(s, strmode)`. Here `strmode` is determined from `mode` & `~(ate & | binary)`:

- `ios_base::in` becomes "r" (open existing file for reading).
- `ios_base::out` or `ios_base::out | ios_base::trunc` becomes "w" (truncate existing file or create for writing).
- `ios_base::out | app` becomes "a" (open existing file for appending all writes).
- `ios_base::in | ios_base::out` becomes "r+" (open existing file for reading and writing).
- `ios_base::in | ios_base::out | ios_base::trunc` becomes "w+" (truncate existing file or create for reading and writing).
- `ios_base::in | ios_base::out | ios_base::app` becomes "a+" (open existing file for reading and for appending all writes).

If `mode` & `ios_base::binary` is nonzero, the function appends `b` to `strmode` to open a binary stream (page 18) instead of a text stream (page 17). It then stores the

value returned by `fopen` in the file pointer (page 77) `fp`. If mode & `ios_base::ate` is nonzero and the file pointer is not a null pointer, the function calls `fseek(fp, 0, SEEK_END)` to position the stream at end-of-file. If that positioning operation fails, the function calls `close(fp)` and stores a null pointer in the file pointer.

If the file pointer is not a null pointer, the function determines the **file conversion facet**: `use_facet<codecvt<E, char, traits_type::state_type>>(getloc())`, for use by underflow and overflow.

If the file pointer is a null pointer, the function returns a null pointer. Otherwise, it returns this.

basic_filebuf::overflow

```
virtual int_type overflow(int_type c =  
    traits_type::eof());
```

If `c != traits_type::eof()`, the protected virtual member function endeavors to insert the element `traits_type::to_char_type(c)` into the output buffer (page 187). It can do so in various ways:

- If a write position (page 188) is available, it can store the element into the write position and increment the next pointer for the output buffer.
- It can make a write position available by allocating new or additional storage for the output buffer.
- It can convert any pending output in the output buffer, followed by `c`, by using the file conversion facet (page 79) `fac` to call `fac.out` as needed. Each element `x` of type `char` thus produced is written to the associated stream designated by the file pointer `fp` as if by successive calls of the form `fputc(x, fp)`. If any conversion or write fails, the function does not succeed.

If the function cannot succeed, it returns `traits_type::eof()`. Otherwise, it returns `traits_type::not_eof(c)`.

basic_filebuf::pbackfail

```
virtual int_type pbackfail(int_type c =  
    traits_type::eof());
```

The protected virtual member function endeavors to put back an element into the input buffer (page 187), then make it the current element (pointed to by the next pointer). If `c == traits_type::eof()`, the element to push back is effectively the one already in the stream before the current element. Otherwise, that element is replaced by `x = traits_type::to_char_type(c)`. The function can put back an element in various ways:

- If a putback position (page 188) is available, and the element stored there compares equal to `x`, it can simply decrement the next pointer for the input buffer.
- If the function can make a putback position available, it can do so, set the next pointer to point at that position, and store `x` in that position.
- If the function can push back an element onto the input stream, it can do so, such as by calling `ungetc` for an element of type `char`.

If the function cannot succeed, it returns `traits_type::eof()`. Otherwise, it returns `traits_type::not_eof(c)`.

basic_filebuf::pos_type

```
typedef typename traits_type::pos_type pos_type;
```

The type is a synonym for `traits_type::pos_type`.

basic_filebuf::seekoff

```
virtual pos_type seekoff(off_type off,  
    ios_base::seekdir way,  
    ios_base::openmode which =  
        ios_base::in | ios_base::out);
```

The protected virtual member function endeavors to alter the current positions for the controlled streams. For an object of class `basic_filebuf<E, T>`, a stream position can be represented by an object of type `fpos_t`, which stores an offset and any state information needed to parse a wide stream (page 18). Offset zero designates the first element of the stream. (An object of type `pos_type` (page 190) stores at least an `fpos_t` object.)

For a file opened for both reading and writing, both the input and output streams are positioned in tandem. To switch (page 20) between inserting and extracting, you must call either `pubseekoff` (page 190) or `pubseekpos` (page 191). Calls to `pubseekoff` (and hence to `seekoff`) have various limitations for text streams (page 17), binary streams (page 18), and wide streams (page 18).

If the file pointer (page 77) `fp` is a null pointer, the function fails. Otherwise, it endeavors to alter the stream position by calling `fseek(fp, off, way)`. If that function succeeds and the resultant position `fposn` can be determined by calling `fgetpos(fp, &fposn)`, the function succeeds. If the function succeeds, it returns a value of type `pos_type` containing `fposn`. Otherwise, it returns an invalid stream position.

basic_filebuf::seekpos

```
virtual pos_type seekpos(pos_type pos,  
    ios_base::openmode which =  
        ios_base::in | ios_base::out);
```

The protected virtual member function endeavors to alter the current positions for the controlled streams. For an object of class `basic_filebuf<E, T>`, a stream position can be represented by an object of type `fpos_t`, which stores an offset and any state information needed to parse a wide stream (page 18). Offset zero designates the first element of the stream. (An object of type `pos_type` (page 190) stores at least an `fpos_t` object.)

For a file opened for both reading and writing, both the input and output streams are positioned in tandem. To switch (page 20) between inserting and extracting, you must call either `pubseekoff` (page 190) or `pubseekpos` (page 191). Calls to `pubseekoff` (and hence to `seekoff`) have various limitations for text streams (page 17), binary streams (page 18), and wide streams (page 18).

For a wide stream, if any insertions have occurred since the stream was opened, or since the last call to `streampos`, the function calls `overflow()`. It also inserts any sequence needed to restore the initial conversion state (page 12), by using the file conversion facet (page 79) `fac` to call `fac.unshift` as needed. Each element `x` of type `char` thus produced is written to the associated stream designated by the file pointer `fp` as if by successive calls of the form `fputc(x, fp)`. If the call to `fac.unshift` or any write fails, the function does not succeed.

If the file pointer (page 77) `fp` is a null pointer, the function fails. Otherwise, it endeavors to alter the stream position by calling `fsetpos(fp, &fposn)`, where `fposn` is the `fpos_t` object stored in `pos`. If that function succeeds, the function returns `pos`. Otherwise, it returns an invalid stream position.

basic_filebuf::setbuf

```
virtual basic_streambuf<E, T>
    *setbuf(E *s, streamsize n);
```

The protected member function returns zero if the file pointer (page 77) `fp` is a null pointer. Otherwise, it calls `setvbuf(fp, (char *)s, _IOFBF, n * sizeof(E))` to offer the array of `n` elements beginning at `s` as a buffer for the stream. If that function returns a nonzero value, the function returns a null pointer. Otherwise, it returns this to signal success.

basic_filebuf::sync

```
int sync();
```

The protected member function returns zero if the file pointer (page 77) `fp` is a null pointer. Otherwise, it returns zero only if calls to both `overflow()` and `fflush(fp)` succeed in flushing any pending output to the stream.

basic_filebuf::traits_type

```
typedef T traits_type;
```

The type is a synonym for the template parameter `T`.

basic_filebuf::underflow

```
virtual int_type underflow();
```

The protected virtual member function endeavors to extract the current element `c` from the input stream, and return the element as `traits_type::to_int_type(c)`. It can do so in various ways:

- If a read position (page 188) is available, it takes `c` as the element stored in the read position and advances the next pointer for the input buffer (page 187).
- It can read one or more elements of type *char*, as if by successive calls of the form `fgetc(fp)`, and convert them to an element `c` of type `E` by using the file conversion facet (page 79) `fac` to call `fac.in` as needed. If any read or conversion fails, the function does not succeed.

If the function cannot succeed, it returns `traits_type::eof()`. Otherwise, it returns `c`, converted as described above.

basic_fstream

```
template <class E, class T = char_traits<E> >
class basic_fstream : public basic_iostream<E, T> {
public:
    basic_fstream();
    explicit basic_fstream(const char *s,
        ios_base::openmode mode =
            ios_base::in | ios_base::out);
    basic_filebuf<E, T> *rdbuf() const;
    bool is_open() const;
    void open(const char *s,
        ios_base::openmode mode =
            ios_base::in | ios_base::out);
    void close();
};
```

The template class describes an object that controls insertion and extraction of elements and encoded objects using a stream buffer (page 187) of class `basic_filebuf<E, T>`, with elements of type `E`, whose character traits (page 211) are determined by the class `T`. The object stores an object of class `basic_filebuf<E, T>`.

basic_fstream::basic_fstream

```
basic_fstream();
explicit basic_fstream(const char *s,
    ios_base::openmode mode =
        ios_base::in | ios_base::out);
```

The first constructor initializes the base class by calling `basic_istream(sb)`, where `sb` is the stored object of class `basic_filebuf<E, T>`. It also initializes `sb` by calling `basic_filebuf<E, T>()`.

The second constructor initializes the base class by calling `basic_istream(sb)`. It also initializes `sb` by calling `basic_filebuf<E, T>()`, then `sb.open(s, mode)`. If the latter function returns a null pointer, the constructor calls `setstate(failbit)`.

basic_fstream::close

```
void close();
```

The member function calls `rdbuf()->close()`.

basic_fstream::is_open

```
bool is_open();
```

The member function returns `rdbuf()->is_open()`.

basic_fstream::open

```
void open(const char *s,
    ios_base::openmode mode =
        ios_base::in | ios_base::out);
```

The member function calls `rdbuf()->open(s, mode)`. If that function returns a null pointer, the function calls `setstate(failbit)`.

basic_fstream::rdbuf

```
basic_filebuf<E, T> *rdbuf() const
```

The member function returns the address of the stored stream buffer, of type pointer to `basic_filebuf<E, T>`.

basic_ifstream

```
template <class E, class T = char_traits<E> >
    class basic_ifstream : public basic_istream<E, T> {
public:
    basic_filebuf<E, T> *rdbuf() const;
    basic_ifstream();
    explicit basic_ifstream(const char *s,
        ios_base::openmode mode = ios_base::in);
    bool is_open() const;
    void open(const char *s,
        ios_base::openmode mode = ios_base::in);
    void close();
};
```

The template class describes an object that controls extraction of elements and encoded objects from a stream buffer of class `basic_filebuf<E, T>`, with elements

of type E, whose character traits (page 211) are determined by the class T. The object stores an object of class `basic_filebuf<E, T>`.

basic_ifstream::basic_ifstream

```
basic_ifstream();  
explicit basic_ifstream(const char *s,  
    ios_base::openmode mode = ios_base::in);
```

The first constructor initializes the base class by calling `basic_istream(sb)`, where `sb` is the stored object of class `basic_filebuf<E, T>`. It also initializes `sb` by calling `basic_filebuf<E, T>()`.

The second constructor initializes the base class by calling `basic_istream(sb)`. It also initializes `sb` by calling `basic_filebuf<E, T>()`, then `sb.open(s, mode | ios_base::in)`. If the latter function returns a null pointer, the constructor calls `setstate(failbit)`.

basic_ifstream::close

```
void close();
```

The member function calls `rdbuf()->close()`.

basic_ifstream::is_open

```
bool is_open();
```

The member function returns `rdbuf()->is_open()`.

basic_ifstream::open

```
void open(const char *s,  
    ios_base::openmode mode = ios_base::in);
```

The member function calls `rdbuf()->open(s, mode | ios_base::in)`. If that function returns a null pointer, the function calls `setstate(failbit)`.

basic_ifstream::rdbuf

```
basic_filebuf<E, T> *rdbuf() const
```

The member function returns the address of the stored stream buffer.

basic_ofstream

```
template <class E, class T = char_traits<E> >  
    class basic_ofstream : public basic_ostream<E, T> {  
public:  
    basic_filebuf<E, T> *rdbuf() const;  
    basic_ofstream();  
    explicit basic_ofstream(const char *s,  
        ios_base::openmode mode = ios_base::out);  
    bool is_open() const;  
    void open(const char *s,  
        ios_base::openmode mode = ios_base::out);  
    void close();  
};
```

The template class describes an object that controls insertion of elements and encoded objects into a stream buffer of class `basic_filebuf<E, T>`, with elements of type E, whose character traits (page 211) are determined by the class T. The object stores an object of class `basic_filebuf<E, T>`.

basic_ofstream::basic_ofstream

```
basic_ofstream();  
explicit basic_ofstream(const char *s,  
    ios_base::openmode which = ios_base::out);
```

The first constructor initializes the base class by calling `basic_ostream(sb)`, where `sb` is the stored object of class `basic_filebuf<E, T>`. It also initializes `sb` by calling `basic_filebuf<E, T>()`.

The second constructor initializes the base class by calling `basic_ostream(sb)`. It also initializes `sb` by calling `basic_filebuf<E, T>()`, then `sb.open(s, mode | ios_base::out)`. If the latter function returns a null pointer, the constructor calls `setstate(failbit)`.

basic_ofstream::close

```
void close();
```

The member function calls `rdbuf()->close()`.

basic_ofstream::is_open

```
bool is_open();
```

The member function returns `rdbuf()->is_open()`.

basic_ofstream::open

```
void open(const char *s,  
    ios_base::openmode mode = ios_base::out);
```

The member function calls `rdbuf()->open(s, mode | ios_base::out)`. If that function returns a null pointer, the function calls `setstate(failbit)`.

basic_ofstream::rdbuf

```
basic_filebuf<E, T> *rdbuf() const
```

The member function returns the address of the stored stream buffer.

filebuf

```
typedef basic_filebuf<char, char_traits<char> > filebuf;
```

The type is a synonym for template class `basic_filebuf` (page 77), specialized for elements of type `char` with default character traits (page 211).

fstream

```
typedef basic_fstream<char, char_traits<char> > fstream;
```

The type is a synonym for template class `basic_fstream` (page 81), specialized for elements of type `char` with default character traits (page 211).

ifstream

```
typedef basic_ifstream<char, char_traits<char> > ifstream;
```

The type is a synonym for template class `basic_ifstream` (page 82), specialized for elements of type `char` with default character traits (page 211).

ofstream

```
typedef basic_ofstream<char, char_traits<char> >
    ofstream;
```

The type is a synonym for template class `basic_ofstream` (page 83), specialized for elements of type `char` with default character traits (page 211).

wfstream

```
typedef basic_fstream<wchar_t, char_traits<wchar_t> >
    wfstream;
```

The type is a synonym for template class `basic_fstream` (page 81), specialized for elements of type `wchar_t` with default character traits (page 211).

wifstream

```
typedef basic_ifstream<wchar_t, char_traits<wchar_t> >
    wifstream;
```

The type is a synonym for template class `basic_ifstream` (page 82), specialized for elements of type `wchar_t` with default character traits (page 211).

wofstream

```
typedef basic_ofstream<wchar_t, char_traits<wchar_t> >
    wofstream;
```

The type is a synonym for template class `basic_ofstream` (page 83), specialized for elements of type `wchar_t` with default character traits (page 211).

wfilebuf

```
typedef basic_filebuf<wchar_t, char_traits<wchar_t> >
    wfilebuf;
```

The type is a synonym for template class `basic_filebuf` (page 77), specialized for elements of type `wchar_t` with default character traits (page 211).

<iomanip>

```
namespace std {
    T1 resetiosflags(ios_base::fmtflags mask);
    T2 setiosflags(ios_base::fmtflags mask);
    T3 setbase(int base);
    template<class E>
        T4 setfill(E c);
    T5 setprecision(streamsize n);
    T6 setw(streamsize n);
};
```

Include the `iostreams` (page 7) standard header **<iomanip>** to define several manipulators (page 87) that each take a single argument. Each of these manipulators returns an unspecified type, called T1 through T6 here, that overloads both `basic_istream<E, T>::operator>>` and `basic_ostream<E, T>::operator<<`. Thus, you can write extractors and inserters such as:

```
cin >> setbase(8);
cout << setbase(8);
```

resetiosflags

T1 **resetiosflags**(ios_base::fmtflags mask);

The manipulator returns an object that, when extracted from or inserted into the stream *str*, calls *str.setf*(ios_base::fmtflags(), mask), then returns *str*.

setbase

T3 **setbase**(int base);

The manipulator returns an object that, when extracted from or inserted into the stream *str*, calls *str.setf*(mask, ios_base::basefield), then returns *str*. Here, mask is determined as follows:

- If base is 8, then mask is ios_base::oct
- If base is 10, then mask is ios_base::dec
- If base is 16, then mask is ios_base::hex
- If base is any other value, then mask is ios_base::fmtflags(0)

setfill

```
template<class E>
T4 setfill(E fillch);
```

The template manipulator returns an object that, when extracted from or inserted into the stream *str*, calls *str.fill*(fillch), then returns *str*. The type *E* must be the same as the element type for the stream *str*.

setiosflags

T2 **setiosflags**(ios_base::fmtflags mask);

The manipulator returns an object that, when extracted from or inserted into the stream *str*, calls *str.setf*(mask), then returns *str*.

setprecision

T5 **setprecision**(streamsize prec);

The manipulator returns an object that, when extracted from or inserted into the stream *str*, calls *str.precision*(prec), then returns *str*.

setw

T6 **setw**(streamsize wide);

The manipulator returns an object that, when extracted from or inserted into the stream *str*, calls *str.width*(wide), then returns *str*.

<ios>

basic_ios (page 88) · fpos (page 92) · ios (page 94) · ios_base (page 94) · streamoff (page 101) · streampos (page 101) · streamsize (page 101) · wios (page 101) · wstreampos (page 101)

boolalpha (page 91) · dec (page 92) · fixed (page 92) · hex (page 93) · internal (page 94) · left (page 99) · noboolalpha (page 99) · noshowbase (page 99) · noshowpoint (page 99) · noshowpos (page 99) · noskipws (page 100) · nounitbuf

(page 100) · nouppercase (page 100) · oct (page 100) · right (page 100) · scientific (page 100) · showbase (page 100) · showpoint (page 100) · showpos (page 100) · skipws (page 101) · unitbuf (page 101) · uppercase (page 101)

```
namespace std {
    typedef T1 streamoff;
    typedef T2 streamsize;
    class ios_base;

    // TEMPLATE CLASSES
    template <class E, class T = char_traits<E> >
        class basic_ios;
    typedef basic_ios<char, char_traits<char> > ios;
    typedef basic_ios<wchar_t, char_traits<wchar_t> >
        wios;
    template <class St>
        class fpos;
    typedef fpos<mbstate_t> streampos;
    typedef fpos<mbstate_t> wstreampos;

    // MANIPULATORS
    ios_base& boolalpha(ios_base& str);
    ios_base& noboolalpha(ios_base& str);
    ios_base& showbase(ios_base& str);
    ios_base& noshowbase(ios_base& str);
    ios_base& showpoint(ios_base& str);
    ios_base& noshowpoint(ios_base& str);
    ios_base& showpos(ios_base& str);
    ios_base& noshowpos(ios_base& str);
    ios_base& skipws(ios_base& str);
    ios_base& noskipws(ios_base& str);
    ios_base& unitbuf(ios_base& str);
    ios_base& nounitbuf(ios_base& str);
    ios_base& uppercase(ios_base& str);
    ios_base& nouppercase(ios_base& str);
    ios_base& internal(ios_base& str);
    ios_base& left(ios_base& str);
    ios_base& right(ios_base& str);
    ios_base& dec(ios_base& str);
    ios_base& hex(ios_base& str);
    ios_base& oct(ios_base& str);
    ios_base& fixed(ios_base& str);
    ios_base& scientific(ios_base& str);
};
```

Include the iostreams (page 7) standard header **<ios>** to define several types and functions basic to the operation of iostreams. (This header is typically included for you by another of the iostreams headers. You seldom have occasion to include it directly.)

A large group of functions are **manipulators**. A manipulator declared in **<ios>** alters the values stored in its argument object of class **ios_base** (page 94). Other manipulators perform actions on streams controlled by objects of a type derived from this class, such as a specialization of one of the template classes **basic_istream** (page 106) or **basic_ostream** (page 169). For example, **noskipws(str)** clears the format flag **ios_base::skipws** in the object **str**, which might be of one of these types.

You can also call a manipulator by inserting it into an output stream or extracting it from an input stream, thanks to some special machinery supplied in the classes derived from **ios_base**. For example:

```
istr >> noskipws;
```

calls `noskipws(istr)`.

basic_ios

`bad` (page 89) · `basic_ios` (page 89) · `char_type` (page 89) · `clear` (page 89) · `copyfmt` (page 89) · `eof` (page 89) · `exceptions` (page 89) · `init` (page 90) · `fail` (page 90) · `good` (page 90) · `imbue` (page 90) · `init` (page 90) · `int_type` (page 90) · `narrow` (page 90) · `off_type` (page 90) · `operator!` (page 91) · `operator void *` (page 91) · `pos_type` (page 91) · `rdbuf` (page 91) · `rdstate` (page 91) · `setstate` (page 91) · `tie` (page 91) · `traits_type` (page 91) · `widen` (page 91)

```
template <class E, class T = char_traits<E> >
    class basic_ios : public ios_base {
public:
    typedef E char_type;
    typedef T traits_type;
    typedef typename T::int_type int_type;
    typedef typename T::pos_type pos_type;
    typedef typename T::off_type off_type;
    explicit basic_ios(basic_streambuf<E, T> *sb);
    virtual ~basic_ios();
    operator void *() const;
    bool operator!() const;
    iostate rdstate() const;
    void clear(iostate state = goodbit);
    void setstate(iostate state);
    bool good() const;
    bool eof() const;
    bool fail() const;
    bool bad() const;
    iostate exceptions() const;
    iostate exceptions(iostate except);
    basic_ios& copyfmt(const basic_ios& rhs);
    locale imbue(const locale& loc);
    char_type widen(char ch);
    char narrow(char_type ch, char dflt);
    char_type fill() const;
    char_type fill(char_type ch);
    basic_ostream<E, T> *tie() const;
    basic_ostream<E, T> *tie(basic_ostream<E, T> *str);
    basic_streambuf<E, T> *rdbuf() const;
    basic_streambuf<E, T>
        *rdbuf(basic_streambuf<E, T> *sb);
    E widen(char ch);
    char narrow(E ch, char dflt);
protected:
    void init(basic_streambuf<E, T> *sb);
    basic_ios();
    basic_ios(const facet&);    // not defined
    void operator=(const facet&) // not defined
    };
```

The template class describes the storage and member functions common to both input streams (of template class `basic_istream` (page 106)) and output streams (of template class `basic_ostream` (page 169)) that depend on the template parameters. (The class `ios_base` (page 94) describes what is common and *not* dependent on template parameters.) An object of class `basic_ios<E, T>` helps control a stream with elements of type `E`, whose character traits (page 211) are determined by the class `T`.

An object of class `basic_ios<E, T>` stores:

- a **tie pointer** to an object of type `basic_ostream<E, T>`
- a **stream buffer pointer** to an object of type `basic_streambuf<E, T>`

- formatting information (page 95)
- stream state information (page 95) in a base object of type `ios_base` (page 94)
- a **fill character** in an object of type *char_type*

basic_ios::bad

```
bool bad() const;
```

The member function returns true if `rdstate()` & `badbit` is nonzero.

basic_ios::basic_ios

```
explicit basic_ios(basic_streambuf<E, T> *sb);  
basic_ios();
```

The first constructor initializes its member objects by calling `init(sb)`. The second (protected) constructor leaves its member objects uninitialized. A later call to `init` *must* initialize the object before it can be safely destroyed.

basic_ios::char_type

```
typedef E char_type;
```

The type is a synonym for the template parameter `E`.

basic_ios::clear

```
void clear(iostate state = goodbit);
```

The member function replaces the stored stream state information (page 95) with `state | (rdbuf() != 0 ? goodbit : badbit)`. If `state & exceptions()` is nonzero, it then throws an object of class `failure` (page 95).

basic_ios::copyfmt

```
basic_ios& copyfmt(const basic_ios& rhs);
```

The member function reports the callback event (page 95) `erase_event` (page 95). It then copies from `rhs` into `*this` the fill character (page 89), the tie pointer (page 88), and the formatting information (page 95). Before altering the exception mask (page 95), it reports the callback event `copyfmt_event` (page 95). If, after the copy is complete, `state & exceptions()` is nonzero, the function effectively calls `clear` (page 89) with the argument `rdstate()`. It returns `*this`.

basic_ios::eof

```
bool eof() const;
```

The member function returns true if `rdstate()` & `eofbit` is nonzero.

basic_ios::exceptions

```
iostate exceptions() const;  
iostate exceptions(iostate except);
```

The first member function returns the stored exception mask (page 95). The second member function stores `except` in the exception mask and returns its previous stored value. Note that storing a new exception mask can throw an exception just like the call `clear(rdstate())`.

basic_ios::fail

```
bool fail() const;
```

The member function returns true if `rdstate()` & `failbit` is nonzero.

basic_ios::fill

```
char_type fill() const;  
char_type fill(char_type ch);
```

The first member function returns the stored fill character (page 89). The second member function stores `ch` in the fill character and returns its previous stored value.

basic_ios::good

```
bool good() const;
```

The member function returns true if `rdstate() == goodbit` (no state flags are set).

basic_ios::imbue

```
locale imbue(const locale& loc);
```

If `rdbuf` (page 91) is not a null pointer, the member function calls `rdbuf()->pubimbue(loc)`. In any case, it returns `ios_base::imbue(loc)`.

basic_ios::init

```
void init(basic_streambuf<E, T> *sb);
```

The member function stores values in all member objects, so that:

- `rdbuf()` returns `sb`
- `tie()` returns a null pointer
- `rdstate()` returns `goodbit` if `sb` is nonzero; otherwise, it returns `badbit`
- `exceptions()` returns `goodbit`
- `flags()` returns `skipws | dec`
- `width()` returns zero
- `precision()` returns 6
- `fill()` returns the space character
- `getloc()` returns `locale::classic()`
- `word` returns zero and `pword` returns a null pointer for all argument value

basic_ios::int_type

```
typedef typename T::int_type int_type;
```

The type is a synonym for `T::int_type`.

basic_ios::narrow

```
char narrow(char_type ch, char dflt);
```

The member function returns `use_facet<ctype<E>>(getloc()).narrow(ch, dflt)`.

basic_ios::off_type

```
typedef typename T::off_type off_type;
```

The type is a synonym for `T::off_type`.

basic_ios::operator void *
operator void *() const;

The operator returns a null pointer only if fail().

basic_ios::operator!
bool **operator!**() const;

The operator returns fail().

basic_ios::pos_type
typedef typename T::pos_type **pos_type**;

The type is a synonym for T::pos_type.

basic_ios::rdbuf
basic_streambuf<E, T> ***rdbuf**() const;
basic_streambuf<E, T> ***rdbuf**(basic_streambuf<E, T> *sb);

The first member function returns the stored stream buffer pointer.

The second member function stores sb in the stored stream buffer pointer and returns the previously stored value.

basic_ios::rdstate
iostate **rdstate**() const;

The member function returns the stored stream state information.

basic_ios::setstate
void **setstate**(iostate state);

The member function effectively calls clear(state | rdstate()).

basic_ios::tie
basic_ostream<E, T> ***tie**() const;
basic_ostream<E, T> ***tie**(basic_ostream<E, T> *str);

The first member function returns the stored tie pointer (page 88). The second member function stores str in the tie pointer and returns its previous stored value.

basic_ios::traits_type
typedef T **traits_type**;

The type is a synonym for the template parameter T.

basic_ios::widen
char_type **widen**(char ch);

The member function returns use_facet< ctype<E> >(getloc()). widen(ch).

boolalpha

ios_base& **boolalpha**(ios_base& str);

The manipulator effectively calls str.setf(ios_base:: boolalpha), then returns str.

dec

```
ios_base& dec(ios_base& str);
```

The manipulator effectively calls `str.setf(ios_base::dec, ios_base::basefield)`, then returns `str`.

fixed

```
ios_base& fixed(ios_base& str);
```

The manipulator effectively calls `str.setf(ios_base::fixed, ios_base::floatfield)`, then returns `str`.

fpos

```
template <class St>
class fpos {
public:
    fpos(streamoff off);
    explicit fpos(St state);
    St state() const;
    void state(St state);
    operator streamoff() const;
    streamoff operator-(const fpos& rhs) const;
    fpos& operator+=(streamoff off);
    fpos& operator-=(streamoff off);
    fpos operator+(streamoff off) const;
    fpos operator-(streamoff off) const;
    bool operator==(const fpos& rhs) const;
    bool operator!=(const fpos& rhs) const;
};
```

The template class describes an object that can store all the information needed to restore an arbitrary file-position indicator (page 19) within any stream. An object of class `fpos<St>` effectively stores at least two member objects:

- a byte offset, of type `streamoff` (page 101)
- a conversion state, for use by an object of class `basic_filebuf`, of type `St`, typically `mbstate_t`

It can also store an arbitrary file position, for use by an object of class `basic_filebuf` (page 77), of type `fpos_t`. For an environment with limited file size, however, `streamoff` and `fpos_t` may sometimes be used interchangeably. And for an environment with no streams that have a state-dependent encoding (page 12), `mbstate_t` may actually be unused. So the number of member objects stored may vary.

fpos::fpos

```
fpos(streamoff off);
explicit fpos(St state);
```

The first constructor stores the offset `off`, relative to the beginning of file and in the initial conversion state (page 12) (if that matters). If `off` is -1, the resulting object represents an invalid stream position.

The second constructor stores a zero offset and the object state.

fpos::operator!=

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

The member function returns `!(*this == rhs)`.

fpos::operator+

```
fpos operator+(streamoff off) const;
```

The member function returns `fpos(*this) += off`.

fpos::operator+=

```
fpos& operator+=(streamoff off);
```

The member function adds `off` to the stored offset member object, then returns `*this`. For positioning within a file, the result is generally valid only for binary streams (page 18) that do not have a state-dependent encoding (page 12).

fpos::operator-

```
streamoff operator-(const fpos& rhs) const;  
fpos operator-(streamoff off) const;
```

The first member function returns `(streamoff)*this - (streamoff)rhs`. The second member function returns `fpos(*this) -= off`.

fpos::operator-=

```
fpos& operator-=(streamoff off);
```

The member function returns `fpos(*this) -= off`. For positioning within a file, the result is generally valid only for binary streams (page 18) that do not have a state-dependent encoding (page 12).

fpos::operator==

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

The member function returns `(streamoff)*this == (streamoff)rhs`.

fpos::operator streamoff

```
operator streamoff() const;
```

The member function returns the stored offset member object, plus any additional offset stored as part of the `fpos_t` member object.

fpos::state

```
St state() const;  
void state(St state);
```

The first member function returns the value stored in the `St` member object. The second member function stores `state` in the `St` member object.

hex

```
ios_base& hex(ios_base& str);
```

The manipulator effectively calls `str.setf(ios_base::hex, ios_base::basefield)`, then returns `str`.

internal

```
ios_base& internal(ios_base& str);
```

The manipulator effectively calls `str.setf(ios_base::internal, ios_base::adjustfield)`, then returns `str`.

ios

```
typedef basic_ios<char, char_traits<char> > ios;
```

The type is a synonym for template class `basic_ios` (page 88), specialized for elements of type *char* with default character traits (page 211).

ios_base

`event` (page 95) · `event_callback` (page 95) · `failure` (page 95) · `flags` (page 95) · `fmtflags` (page 96) · `getloc` (page 96) · `imbue` (page 96) · `Init` (page 97) · `ios_base` (page 97) · `iostate` (page 97) · `iword` (page 97) · `openmode` (page 97) · `precision` (page 98) · `pword` (page 98) · `register_callback` (page 98) · `seekdir` (page 98) · `setf` (page 98) · `sync_with_stdio` (page 98) · `unsetf` (page 99) · `width` (page 99) · `xalloc` (page 99)

```
class ios_base {
public:
    class failure;
    typedef T1 fmtflags;
    static const fmtflags boolalpha, dec, fixed, hex,
        internal, left, oct, right, scientific,
        showbase, showpoint, showpos, skipws, unitbuf,
        uppercase, adjustfield, basefield, floatfield;
    typedef T2 iostate;
    static const iostate badbit, eofbit, failbit,
        goodbit;
    typedef T3 openmode;
    static const openmode app, ate, binary, in, out,
        trunc;
    typedef T4 seekdir;
    static const seekdir beg, cur, end;
    typedef T5 event;
    static const event copyfmt_event, erase_event,
        copyfmt_event;
    class Init;
    fmtflags flags() const;
    fmtflags flags(fmtflags fmtfl);
    fmtflags setf(fmtflags fmtfl);
    fmtflags setf(fmtflags fmtfl, fmtflags mask);
    void unsetf(fmtflags mask);
    streamsize precision() const;
    streamsize precision(streamsize prec);
    streamsize width() const;
    streamsize width(streamsize wide);
    locale imbue(const locale& loc);
    locale getloc() const;
    static int xalloc();
    long& iword(int idx);
    void*& pword(int idx);
    typedef void* (event_callback(event ev,
        ios_base& ios, int idx);
    void register_callback(event_callback pfn, int idx);
    static bool sync_with_stdio(bool sync = true);
protected:
    ios_base();
```

```
private:
    ios_base(const ios_base&);
    ios_base& operator=(const ios_base&);
};
```

The class describes the storage and member functions common to both input and output streams that does not depend on the template parameters. (The template class `basic_ios` (page 88) describes what is common and *is* dependent on template parameters.)

An object of class `ios_base` stores **formatting information**, which consists of:

- **format flags** in an object of type `fmtflags` (page 96)
- an **exception mask** in an object of type `iostate` (page 97)
- a **field width** in an object of type `int`
- a **display precision** in an object of type `int`
- a **locale object** (page 135) in an object of type `locale` (page 134)
- two **extensible arrays**, with elements of type `long` and `void` pointer

An object of class `ios_base` also stores **stream state information**, in an object of type `iostate` (page 97), and a **callback stack**.

`ios_base::event`

```
typedef T5 event;
static const event copyfmt_event, erase_event,
    imbue_event;
```

The type is an enumerated type `T5` that describes an object that can store the **callback event** used as an argument to a function registered with `register_callback` (page 98). The distinct event values are:

- `copyfmt_event`, to identify a callback that occurs near the end of a call to `copyfmt`, just before the exception mask is copied.
- `erase_event`, to identify a callback that occurs at the beginning of a call to `copyfmt`, or at the beginning of a call to the destructor for `*this`.
- `imbue_event`, to identify a callback that occurs at the end of a call to `imbue` (page 96), just before the function returns.

`ios_base::event_callback`

```
typedef void *(event_callback(event ev,
    ios_base& ios, int idx));
```

The type describes a pointer to a function that can be registered with `register_callback` (page 98). Such a function must not throw an exception.

`ios_base::failure`

```
class failure : public exception {
public:
    explicit failure(const string& what_arg) {
    };
};
```

The member class serves as the base class for all exceptions thrown by the member function `clear` (page 89) in template class `basic_ios` (page 88). The value returned by `what()` is `what_arg.data()`.

`ios_base::flags`

```
fmtflags flags() const;
fmtflags flags(fmtflags fmtfl);
```

The first member function returns the stored format flags (page 95). The second member function stores `fmtfl` in the format flags and returns its previous stored value.

ios_base::fmtflags

```
typedef T1 fmtflags;  
static const fmtflags boolalpha, dec, fixed, hex,  
    internal, left, oct, right, scientific,  
    showbase, showpoint, showpos, skipws, unitbuf,  
    uppercase, adjustfield, basefield, floatfield;
```

The type is a bitmask type (page 6) `T1` that describes an object that can store format flags. The distinct flag values (elements) are:

- `boolalpha`, to insert or extract objects of type *bool* as names (such as `true` and `false`) rather than as numeric values
- `dec`, to insert or extract integer values in decimal format
- `fixed`, to insert floating-point values in fixed-point format (with no exponent field)
- `hex`, to insert or extract integer values in hexadecimal format
- `internal`, to pad to a field width (page 95) as needed by inserting fill characters (page 89) at a point internal to a generated numeric field
- `left`, to pad to a field width (page 95) as needed by inserting fill characters (page 89) at the end of a generated field (left justification)
- `oct`, to insert or extract integer values in octal format
- `right`, to pad to a field width (page 95) as needed by inserting fill characters (page 89) at the beginning of a generated field (right justification)
- `scientific`, to insert floating-point values in scientific format (with an exponent field)
- `showbase`, to insert a prefix that reveals the base of a generated integer field
- `showpoint`, to insert a decimal point unconditionally in a generated floating-point field
- `showpos`, to insert a plus sign in a non-negative generated numeric field
- `skipws`, to skip leading white space (page 31) before certain extractions
- `unitbuf`, to flush output after each insertion
- `uppercase`, to insert uppercase equivalents of lowercase letters in certain insertions

In addition, several useful values are:

- `adjustfield`, `internal` | `left` | `right`
- `basefield`, `dec` | `hex` | `oct`
- `floatfield`, `fixed` | `scientific`

ios_base::getloc

```
locale getloc() const;
```

The member function returns the stored locale object.

ios_base::imbue

```
locale imbue(const locale& loc);
```

The member function stores `loc` in the locale object, then reports the callback event (page 95) `imbue_event` (page 95). It returns the previous stored value.

ios_base::Init

```
class Init {  
};
```

The nested class describes an object whose construction ensures that the standard iostreams objects are properly constructed (page 104), even before the execution of a constructor for an arbitrary static object.

ios_base::ios_base

```
ios_base();
```

The (protected) constructor does nothing. A later call to `basic_ios::init` *must* initialize the object before it can be safely destroyed. Thus, the only safe use for class `ios_base` is as a base class for template class `basic_ios` (page 88).

ios_base::iostate

```
typedef T2 iostate;  
static const iostate badbit, eofbit, failbit, goodbit;
```

The type is a bitmask type (page 6) `T2` that describes an object that can store stream state information (page 95). The distinct flag values (elements) are:

- `badbit`, to record a loss of integrity of the stream buffer
- `eofbit`, to record end-of-file while extracting from a stream
- `failbit`, to record a failure to extract a valid field from a stream

In addition, a useful value is:

- `goodbit`, no bits set

ios_base::iword

```
long& iword(int idx);
```

The member function returns a reference to element `idx` of the extensible array (page 95) with elements of type *long*. All elements are effectively present and initially store the value zero. The returned reference is invalid after the next call to `iword` for the object, after the object is altered by a call to `basic_ios::copyfmt`, or after the object is destroyed.

If `idx` is negative, or if unique storage is unavailable for the element, the function calls `setstate(badbit)` and returns a reference that might not be unique.

To obtain a unique index, for use across all objects of type `ios_base`, call `xalloc` (page 99).

ios_base::openmode

```
typedef T3 openmode;  
static const openmode app, ate, binary, in, out, trunc;
```

The type is a bitmask type (page 6) `T3` that describes an object that can store the **opening mode** for several iostreams objects. The distinct flag values (elements) are:

- `app`, to seek to the end of a stream before each insertion
- `ate`, to seek to the end of a stream when its controlling object is first created
- `binary`, to read a file as a binary stream (page 18), rather than as a text stream (page 17)
- `in`, to permit extraction from a stream

- `out`, to permit insertion to a stream
- `trunc`, to truncate an existing file when its controlling object is first created

ios_base::precision

```
streamsize precision() const;
streamsize precision(streamsize prec);
```

The first member function returns the stored display precision (page 95). The second member function stores `prec` in the display precision and returns its previous stored value.

ios_base::pword

```
void *& pword(int idx);
```

The member function returns a reference to element `idx` of the extensible array (page 95) with elements of type *void* pointer. All elements are effectively present and initially store the null pointer. The returned reference is invalid after the next call to `pword` for the object, after the object is altered by a call to `basic_ios::copyfmt`, or after the object is destroyed.

If `idx` is negative, or if unique storage is unavailable for the element, the function calls `setstate(badbit)` and returns a reference that might not be unique.

To obtain a unique index, for use across all objects of type `ios_base`, call `xalloc` (page 99).

ios_base::register_callback

```
void register_callback(event_callback pfn, int idx);
```

The member function pushes the pair `{pfn, idx}` onto the stored callback stack (page 95). When a callback event (page 95) `ev` is reported, the functions are called, in reverse order of registry, by the expression `(*pfn)(ev, *this, idx)`.

ios_base::seekdir

```
typedef T4 seekdir;
static const seekdir beg, cur, end;
```

The type is an enumerated type `T4` that describes an object that can store the **seek mode** used as an argument to the member functions of several `iostreams` classes. The distinct flag values are:

- `beg`, to seek (alter the current read or write position) relative to the beginning of a sequence (array, stream, or file)
- `cur`, to seek relative to the current position within a sequence
- `end`, to seek relative to the end of a sequence

ios_base::setf

```
void setf(fmtflags mask);
fmtflags setf(fmtflags fmtfl, fmtflags mask);
```

The first member function effectively calls `flags(mask | flags())` (set selected bits), then returns the previous format flags (page 95). The second member function effectively calls `flags(mask & ~fmtfl, flags() & ~mask)` (replace selected bits under a mask), then returns the previous format flags.

ios_base::sync_with_stdio

```
static bool sync_with_stdio(bool sync = true);
```

The static member function stores a **stdio sync flag**, which is initially true. When true, this flag ensures that operations on the same file are properly synchronized between the `iostreams` (page 7) functions and those defined in the Standard C library. Otherwise, synchronization may or may not be guaranteed, but performance may be improved. The function stores sync in the stdio sync flag and returns its previous stored value. You can call it reliably only before performing any operations on the standard streams.

ios_base::unsetf

```
void unsetf(fmtflags mask);
```

The member function effectively calls `flags(~mask & flags())` (clear selected bits).

ios_base::width

```
streamsize width() const;  
streamsize width(streamsize wide);
```

The first member function returns the stored field width (page 95). The second member function stores wide in the field width and returns its previous stored value.

ios_base::xalloc

```
static int xalloc();
```

The static member function returns a stored static value, which it increments on each call. You can use the return value as a unique index argument when calling the member functions `isword` (page 97) or `pword` (page 98).

left

```
ios_base& left(ios_base& str);
```

The manipulator effectively calls `str.setf(ios_base::left, ios_base::adjustfield)`, then returns `str`.

noboolalpha

```
ios_base& noboolalpha(ios_base& str);
```

The manipulator effectively calls `str.unsetf(ios_base::boolalpha)`, then returns `str`.

noshowbase

```
ios_base& noshowbase(ios_base& str);
```

The manipulator effectively calls `str.unsetf(ios_base::showbase)`, then returns `str`.

noshowpoint

```
ios_base& noshowpoint(ios_base& str);
```

The manipulator effectively calls `str.unsetf(ios_base::showpoint)`, then returns `str`.

noshowpos

```
ios_base& noshowpos(ios_base& str);
```

The manipulator effectively calls `str.unsetf(ios_base::showpos)`, then returns `str`.

noskipws

```
ios_base& noskipws(ios_base& str);
```

The manipulator effectively calls `str.unsetf(ios_base::skipws)`, then returns `str`.

nounitbuf

```
ios_base& nounitbuf(ios_base& str);
```

The manipulator effectively calls `str.unsetf(ios_base::unitbuf)`, then returns `str`.

nouppercase

```
ios_base& nouppercase(ios_base& str);
```

The manipulator effectively calls `str.unsetf(ios_base::uppercase)`, then returns `str`.

oct

```
ios_base& oct(ios_base& str);
```

The manipulator effectively calls `str.setf(ios_base::oct, ios_base::basefield)`, then returns `str`.

right

```
ios_base& right(ios_base& str);
```

The manipulator effectively calls `str.setf(ios_base::right, ios_base::adjustfield)`, then returns `str`.

scientific

```
ios_base& scientific(ios_base& str);
```

The manipulator effectively calls `str.setf(ios_base::scientific, ios_base::floatfield)`, then returns `str`.

showbase

```
ios_base& showbase(ios_base& str);
```

The manipulator effectively calls `str.setf(ios_base::showbase)`, then returns `str`.

showpoint

```
ios_base& showpoint(ios_base& str);
```

The manipulator effectively calls `str.setf(ios_base::showpoint)`, then returns `str`.

showpos

```
ios_base& showpos(ios_base& str);
```

The manipulator effectively calls `str.setf(ios_base::showpos)`, then returns `str`.

skipws

```
ios_base& skipws(ios_base& str);
```

The manipulator effectively calls `str.setf(ios_base::skipws)`, then returns `str`.

streamoff

```
typedef T1 streamoff;
```

The type is a signed integer type `T1` that describes an object that can store a byte offset involved in various stream positioning operations. Its representation has at least 32 value bits. It is *not* necessarily large enough to represent an arbitrary byte position within a stream. The value `streamoff(-1)` generally indicates an erroneous offset.

streampos

```
typedef fpos<mbstate_t> streampos;
```

The type is a synonym for `fpos<mbstate_t>`.

streamsize

```
typedef T2 streamsize;
```

The type is a signed integer type `T3` that describes an object that can store a count of the number of elements involved in various stream operations. Its representation has at least 16 bits. It is *not* necessarily large enough to represent an arbitrary byte position within a stream.

unitbuf

```
ios_base& unitbuf(ios_base& str);
```

The manipulator effectively calls `str.setf(ios_base::unitbuf)`, then returns `str`.

uppercase

```
ios_base& uppercase(ios_base& str);
```

The manipulator effectively calls `str.setf(ios_base::uppercase)`, then returns `str`.

wios

```
typedef basic_ios<wchar_t, char_traits<wchar_t> > wios;
```

The type is a synonym for template class `basic_ios` (page 88), specialized for elements of type `wchar_t` with default character traits (page 211).

wstreampos

```
typedef fpos<mbstate_t> wstreampos;
```

The type is a synonym for `fpos<wmbstate_t>`.

<iosfwd>

```
namespace std {
    typedef T1 streamoff;
    typedef T2 streamsize;
    typedef fpos streampos;

    // TEMPLATE CLASSES
    template<class E>
        class char_traits;
    class char_traits<char>;
    class char_traits<wchar_t>;
    template<class E, class T = char_traits<E> >
        class basic_ios;
    template<class E, class T = char_traits<E> >
        class istreambuf_iterator;
    template<class E, class T = char_traits<E> >
        class ostreambuf_iterator;
    template<class E, class T = char_traits<E> >
        class basic_streambuf;
    template<class E, class T = char_traits<E> >
        class basic_istream;
    template<class E, class T = char_traits<E> >
        class basic_ostream;
    template<class E, class T = char_traits<E> >
        class basic_iostream;
    template<class E, class T = char_traits<E> >
        class basic_stringbuf;
    template<class E, class T = char_traits<E> >
        class basic_istreamstream;
    template<class E, class T = char_traits<E> >
        class basic_ostreamstream;
    template<class E, class T = char_traits<E> >
        class basic_stringstream;
    template<class E, class T = char_traits<E> >
        class basic_filebuf;
    template<class E, class T = char_traits<E> >
        class basic_ifstream;
    template<class E, class T = char_traits<E> >
        class basic_ofstream;
    template<class E, class T = char_traits<E> >
        class basic_fstream;

    // char TYPE DEFINITIONS
    typedef basic_ios<char, char_traits<char> > ios;
    typedef basic_streambuf<char, char_traits<char> >
        streambuf;
    typedef basic_istream<char, char_traits<char> >
        istream;
    typedef basic_ostream<char, char_traits<char> >
        ostream;
    typedef basic_iostream<char, char_traits<char> >
        iostream;
    typedef basic_stringbuf<char, char_traits<char> >
        stringbuf;
    typedef basic_istreamstream<char, char_traits<char> >
        istreamstream;
    typedef basic_ostreamstream<char, char_traits<char> >
        ostreamstream;
    typedef basic_stringstream<char, char_traits<char> >
        stringstream;
    typedef basic_filebuf<char, char_traits<char> >
        filebuf;
    typedef basic_ifstream<char, char_traits<char> >
        ifstream;
    typedef basic_ofstream<char, char_traits<char> >
        ofstream;
```

```

typedef basic_fstream<char, char_traits<char> >
    fstream;

    // wchar_t TYPE DEFINITIONS
typedef basic_ios<wchar_t, char_traits<wchar_t> > wios;
typedef basic_streambuf<wchar_t, char_traits<wchar_t> >
    wstreambuf;
typedef basic_istream<wchar_t, char_traits<wchar_t> >
    wistream;
typedef basic_ostream<wchar_t, char_traits<wchar_t> >
    wostream;
typedef basic_iostream<wchar_t, char_traits<wchar_t> >
    wiostream;
typedef basic_stringbuf<wchar_t, char_traits<wchar_t> >
    wstringbuf;
typedef basic_istreamstream<wchar_t,
    char_traits<wchar_t> > wistreamstream;
typedef basic_ostreamstream<wchar_t,
    char_traits<wchar_t> > wostreamstream;
typedef basic_stringstream<wchar_t,
    char_traits<wchar_t> > wstringstream;
typedef basic_filebuf<wchar_t, char_traits<wchar_t> >
    wfilebuf;
typedef basic_ifstream<wchar_t, char_traits<wchar_t> >
    wifstream;
typedef basic_ofstream<wchar_t, char_traits<wchar_t> >
    wofstream;
typedef basic_fstream<wchar_t, char_traits<wchar_t> >
    wfstream;
};

```

Include the iostreams (page 7) standard header **<iosfwd>** to declare forward references to several template classes used throughout iostreams. All such template classes are defined in other standard headers. You include this header explicitly only when you need one of the above declarations, but not its definition.

<iostream>

```

namespace std {
extern istream cin;
extern ostream cout;
extern ostream cerr;
extern ostream clog;

extern wistream wcin;
extern wostream wcout;
extern wostream wcerr;
extern wostream wclog;
};

```

Include the iostreams (page 7) standard header **<iostream>** to declare objects that control reading from and writing to the standard streams. This is often the *only* header you need include to perform input and output from a C++ program.

The objects fall into two groups:

- cin (page 104), cout (page 104), cerr (page 104), and clog (page 104) are byte oriented (page 17), performing conventional byte-at-a-time transfers
- wcin (page 105), wcout (page 105), wcerr (page 104), and wclog (page 105) are wide oriented (page 17), translating to and from the wide characters (page 13) that the program manipulates internally

Once you perform certain operations (page 20) on a stream, such as the standard input, you cannot perform operations of a different orientation on the same stream. Hence, a program cannot operate interchangeably on both `cin` and `wcin`, for example.

All the objects declared in this header share a peculiar property — you can assume they are **constructed** before any static objects you define, in a translation unit that includes `<iostreams>`. Equally, you can assume that these objects are **not destroyed** before the destructors for any such static objects you define. (The output streams are, however, flushed during program termination.) Hence, you can safely read from or write to the standard streams prior to program startup and after program termination.

This guarantee is *not* universal, however. A static constructor may call a function in another translation unit. The called function cannot assume that the objects declared in this header have been constructed, given the uncertain order in which translation units participate in static construction. To use these objects in such a context, you must first construct an object of class `ios_base::Init` (page 97), as in:

```
#include <iostream>
void marker()
{    // called by some constructor
    ios_base::Init unused_name;
    cout << "called fun" << endl;
}
```

cerr

```
extern ostream cerr;
```

The object controls unbuffered insertions to the standard error output as a byte stream (page 18). Once the object is constructed, the expression `cerr.flags() & unitbuf` is nonzero.

cin

```
extern istream cin;
```

The object controls extractions from the standard input as a byte stream (page 18). Once the object is constructed, the call `cin.tie()` returns `&cout`.

clog

```
extern ostream clog;
```

The object controls buffered insertions to the standard error output as a byte stream (page 18).

cout

```
extern ostream cout;
```

The object controls insertions to the standard output as a byte stream (page 18).

wcerr

```
extern wostream wcerr;
```


The object controls unbuffered insertions to the standard error output as a wide stream (page 18). Once the object is constructed, the expression `wcerr.flags() & unitbuf` is nonzero.

wcin

```
extern wistream wcin;
```

The object controls extractions from the standard input as a wide stream (page 18). Once the object is constructed, the call `wcin.tie()` returns `&wcout`.

wclog

```
extern wostream wclog;
```

The object controls buffered insertions to the standard error output as a wide stream.

wcout

```
extern wostream wcout;
```

The object controls insertions to the standard output as a wide stream (page 18).

<istream>

```
namespace std {
    template<class E, class T = char_traits<E> >
        class basic_istream;
    typedef basic_istream<char, char_traits<char> >
        istream;
    typedef basic_istream<wchar_t, char_traits<wchar_t> >
        wistream;
    template<class E, class T = char_traits<E> >
        class basic_iostream;
    typedef basic_iostream<char, char_traits<char> >
        iostream;
    typedef basic_iostream<wchar_t, char_traits<wchar_t> >
        wiostream;

    // EXTRACTORS
    template<class E, class T>
        basic_istream<E, T>&
        operator>>(basic_istream<E, T>& is, E *s);
    template<class E, class T>
        basic_istream<E, T>&
        operator>>(basic_istream<E, T>& is, E& c);
    template<class T>
        basic_istream<char, T>&
        operator>>(basic_istream<char, T>& is,
            signed char *s);
    template<class T>
        basic_istream<char, T>&
        operator>>(basic_istream<char, T>& is,
            signed char& c);
    template<class T>
        basic_istream<char, T>&
        operator>>(basic_istream<char, T>& is,
            unsigned char *s);
    template<class T>
        basic_istream<char, T>&
        operator>>(basic_istream<char, T>& is,
            unsigned char& c);
}
```

```

// MANIPULATORS
template class<E, T>
    basic_istream<E, T>& ws(basic_istream<E, T>& is);
};

```

Include the iostreams (page 7) standard header **<iostream>** to define template class `basic_istream` (page 106), which mediates extractions for the iostreams, and the template class `basic_iostream` (page 106), which mediates both insertions and extractions. The header also defines a related manipulator (page 87). (This header is typically included for you by another of the iostreams headers. You seldom have occasion to include it directly.)

basic_iostream

```

template <class E, class T = char_traits<E> >
    class basic_iostream : public basic_istream<E, T>,
        public basic_ostream<E, T> {
public:
    explicit basic_iostream(basic_streambuf<E, T>& *sb);
    virtual ~basic_iostream();
};

```

The template class describes an object that controls insertions, through its base object `basic_ostream<E, T>` (page 169), and extractions, through its base object `basic_istream<E, T>`. The two objects share a common virtual base object `basic_ios<E, T>`. They also manage a common stream buffer (page 187), with elements of type `E`, whose character traits (page 211) are determined by the class `T`. The constructor initializes its base objects via `basic_istream(sb)` and `basic_ostream(sb)`.

basic_istream

basic_istream (page 108) · **gcount** (page 108) · **get** (page 108) · **getline** (page 109) · **ignore** (page 109) · **ipfx** (page 109) · **isfx** (page 110) · **operator>>** (page 110) · **peek** (page 111) · **putback** (page 111) · **read** (page 111) · **readsome** (page 111) · **seekg** (page 111) · **sentry** (page 112) · **sync** (page 112) · **tellg** (page 112) · **unget** (page 112)

```

template <class E, class T = char_traits<E> >
    class basic_istream
        : virtual public basic_ios<E, T> {
public:
    typedef typename basic_ios<E, T>::char_type char_type;
    typedef typename basic_ios<E, T>::traits_type traits_type;
    typedef typename basic_ios<E, T>::int_type int_type;
    typedef typename basic_ios<E, T>::pos_type pos_type;
    typedef typename basic_ios<E, T>::off_type off_type;
    explicit basic_istream(basic_streambuf<E, T> *sb);
    class sentry;
    virtual ~istream();
    bool ipfx(bool noskip = false);
    void isfx();
    basic_istream& operator>>(
        basic_istream& (*pf)(basic_istream&));
    basic_istream& operator>>(
        ios_base& (*pf)(ios_base&));
    basic_istream& operator>>(
        basic_ios<E, T>& (*pf)(basic_ios<E, T>&));
    basic_istream& operator>>(
        basic_streambuf<E, T> *sb);
    basic_istream& operator>>(bool& n);
    basic_istream& operator>>(short& n);
    basic_istream& operator>>(unsigned short& n);
    basic_istream& operator>>(int& n);
    basic_istream& operator>>(unsigned int& n);

```

```

basic_istream& operator>>(long& n);
basic_istream& operator>>(unsigned long& n);
basic_istream& operator>>(void *& n);
basic_istream& operator>>(float& n);
basic_istream& operator>>(double& n);
basic_istream& operator>>(long double& n);
streamsize gcount() const;
int_type get();
basic_istream& get(char_type& c);
basic_istream& get(char_type *s, streamsize n);
basic_istream&
    get(char_type *s, streamsize n, char_type delim);
basic_istream&
    get(basic_streambuf<char_type, T> *sb);
basic_istream&
    get(basic_streambuf<E, T> *sb, char_type delim);
basic_istream& getline(char_type *s, streamsize n);
basic_istream& getline(char_type *s, streamsize n,
    char_type delim);
basic_istream& ignore(streamsize n = 1,
    int_type delim = traits_type::eof());
int_type peek();
basic_istream& read(char_type *s, streamsize n);
streamsize readsome(char_type *s, streamsize n);
basic_istream& putback(char_type c);
basic_istream& unget();
pos_type tellg();
basic_istream& seekg(pos_type pos);
basic_istream& seekg(off_type off,
    ios_base::seek_dir way);
int sync();
};

```

The template class describes an object that controls extraction of elements and encoded objects from a stream buffer (page 187) with elements of type E, also known as `char_type` (page 89), whose character traits (page 211) are determined by the class T, also known as `traits_type` (page 91).

Most of the member functions that overload `operator>>` (page 110) are **formatted input functions**. They follow the pattern:

```

iosstate state = goodbit;
const sentry ok(*this);
if (ok)
    {try
        {<extract elements and convert
         accumulate flags in state
         store a successful conversion>}
        catch (...)
            {try
                {setstate(badbit); }
            catch (...)
                {}
            if ((exceptions() & badbit) != 0)
                throw; }}
setstate(state);
return (*this);

```

Many other member functions are **unformatted input functions**. They follow the pattern:

```

iosstate state = goodbit;
count = 0;    // the value returned by gcount
const sentry ok(*this, true);
if (ok)
    {try

```

```

        {<extract elements and deliver
         count extracted elements in count
         accumulate flags in state>}
    catch (...)
    {try
        {setstate(badbit); }
    catch (...)
    {}
        if ((exceptions() & badbit) != 0)
            throw; }}
    setstate(state);

```

Both groups of functions call `setstate eofbit` if they encounter end-of-file while extracting elements.

An object of class `basic_istream<E, T>` stores:

- a virtual public base object of class **`basic_ios<E, T>`**
- an **extraction count** for the last unformatted input operation (called `count` in the code above)

`basic_istream::basic_istream`

```
explicit basic_istream(basic_streambuf<E, T> *sb);
```

The constructor initializes the base class by calling `init(sb)`. It also stores zero in the extraction count (page 108).

`basic_istream::gcount`

```
streamsize gcount() const;
```

The member function returns the extraction count (page 108).

`basic_istream::get`

```
int_type get();
basic_istream& get(char_type& c);
basic_istream& get(char_type *s, streamsize n);
basic_istream& get(char_type *s, streamsize n,
    char_type delim);
basic_istream& get(basic_streambuf<E, T> *sb);
basic_istream& get(basic_streambuf<E, T> *sb,
    char_type delim);

```

The first of these unformatted input functions (page 107) extracts an element, if possible, as if by returning `rddbuf()->sbumpc()`. Otherwise, it returns `traits_type::eof()`. If the function extracts no element, it calls `setstate(failbit)`.

The second function extracts the `int_type` (page 90) element `x` the same way. If `x` compares equal to `traits_type::eof(x)`, the function calls `setstate(failbit)`. Otherwise, it stores `traits_type::to_char_type(x)` in `c`. The function returns `*this`.

The third function returns `get(s, n, widen('\n'))`.

The fourth function extracts up to `n - 1` elements and stores them in the array beginning at `s`. It always stores `char_type()` after any extracted elements it stores. In order of testing, extraction stops:

1. at end of file
2. after the function extracts an element that compares equal to `delim`, in which case the element is put back to the controlled sequence

3. after the function extracts $n - 1$ elements

If the function extracts no elements, it calls `setstate(failbit)`. In any case, it returns `*this`.

The fifth function returns `get(sb, widen('\n'))`.

The sixth function extracts elements and inserts them in `sb`. Extraction stops on end-of-file or on an element that compares equal to `delim` (which is not extracted). It also stops, without extracting the element in question, if an insertion fails or throws an exception (which is caught but not rethrown). If the function extracts no elements, it calls `setstate(failbit)`. In any case, the function returns `*this`.

basic_istream::getline

```
basic_istream& getline(char_type *s, streamsize n);
basic_istream& getline(char_type *s, streamsize n,
    char_type delim);
```

The first of these unformatted input functions (page 107) returns `getline(s, n, widen('\n'))`.

The second function extracts up to $n - 1$ elements and stores them in the array beginning at `s`. It always stores `char_type()` after any extracted elements it stores. In order of testing, extraction stops:

1. at end of file
2. after the function extracts an element that compares equal to `delim`, in which case the element is neither put back nor appended to the controlled sequence
3. after the function extracts $n - 1$ elements

If the function extracts no elements or $n - 1$ elements, it calls `setstate(failbit)`. In any case, it returns `*this`.

basic_istream::ignore

```
basic_istream& ignore(streamsize n = 1,
    int_type delim = traits_type::eof());
```

The unformatted input function (page 107) extracts up to n elements and discards them. If n equals `numeric_limits<int>::max()`, however, it is taken as arbitrarily large. Extraction stops early on end-of-file or on an element `x` such that `traits_type::to_int_type(x)` compares equal to `delim` (which is also extracted). The function returns `*this`.

basic_istream::ipfx

```
bool ipfx(bool noskip = false);
```

The member function prepares for formatted (page 107) or unformatted (page 107) input. If `good()` is true, the function:

- calls `tie->flush()` if `tie()` is not a null pointer
- effectively calls `ws(*this)` if `flags() & skipws` is nonzero

If, after any such preparation, `good()` is false, the function calls `setstate(failbit)`. In any case, the function returns `good()`.

You should not call `ipfx` directly. It is called as needed by an object of class `sentry` (page 112).

basic_istream::isfx

```
void isfx();
```

The member function has no official duties, but an implementation may depend on a call to `isfx` by a formatted (page 107) or unformatted (page 107) input function to tidy up after an extraction. You should not call `isfx` directly. It is called as needed by an object of class `sentry` (page 112).

basic_istream::operator>>

```
basic_istream& operator>>(
    basic_istream& (*pf)(basic_istream&));
basic_istream& operator>>(
    ios_base& (*pf)(ios_base&));
basic_istream& operator>>(
    basic_ios<E, T>& (*pf)(basic_ios<E, T>&));
basic_istream& operator>>(
    basic_streambuf<E, T> *sb);
basic_istream& operator>>(bool& n);
basic_istream& operator>>(short& n);
basic_istream& operator>>(unsigned short& n);
basic_istream& operator>>(int& n);
basic_istream& operator>>(unsigned int& n);
basic_istream& operator>>(long& n);
basic_istream& operator>>(unsigned long& n);
basic_istream& operator>>(void *& n);
basic_istream& operator>>(float& n);
basic_istream& operator>>(double& n);
basic_istream& operator>>(long double& n);
```

The first member function ensures that an expression of the form `istr >> ws` calls `ws(istr)`, then returns `*this`. The second and third functions ensure that other manipulators (page 87), such as `hex` (page 93) behave similarly. The remaining functions constitute the formatted input functions (page 107).

The function:

```
basic_istream& operator>>(
    basic_streambuf<E, T> *sb);
```

extracts elements, if `sb` is not a null pointer, and inserts them in `sb`. Extraction stops on end-of-file. It also stops, without extracting the element in question, if an insertion fails or throws an exception (which is caught but not rethrown). If the function extracts no elements, it calls `setstate(failbit)`. In any case, the function returns `*this`.

The function:

```
basic_istream& operator>>(bool& n);
```

extracts a field and converts it to a boolean value by calling `use_facet<num_get<E, InIt>>(getloc()).get(InIt(rdbuf()), Init(0), *this, getloc(), n)`. Here, `InIt` is defined as `istreambuf_iterator<E, T>`. The function returns `*this`.

The functions:

```
basic_istream& operator>>(short& n);
basic_istream& operator>>(unsigned short& n);
basic_istream& operator>>(int& n);
basic_istream& operator>>(unsigned int& n);
basic_istream& operator>>(long& n);
basic_istream& operator>>(unsigned long& n);
basic_istream& operator>>(void *& n);
```

each extract a field and convert it to a numeric value by calling `use_facet<num_get<E, InIt>(getloc()). get(InIt(rdbuf()), Init(0), *this, getloc(), x)`. Here, `InIt` is defined as `istreambuf_iterator<E, T>`, and `x` has type *long*, *unsigned long*, or *void ** as needed.

If the converted value cannot be represented as the type of `n`, the function calls `setstate(failbit)`. In any case, the function returns `*this`.

The functions:

```
basic_istream& operator>>(float& n);
basic_istream& operator>>(double& n);
basic_istream& operator>>(long double& n);
```

each extract a field and convert it to a numeric value by calling `use_facet<num_get<E, InIt>(getloc()). get(InIt(rdbuf()), Init(0), *this, getloc(), x)`. Here, `InIt` is defined as `istreambuf_iterator<E, T>`, and `x` has type *double* or *long double* as needed.

If the converted value cannot be represented as the type of `n`, the function calls `setstate(failbit)`. In any case, it returns `*this`.

basic_istream::peek

```
int_type peek();
```

The unformatted input function (page 107) extracts an element, if possible, as if by returning `rdbuf()->sgetc()`. Otherwise, it returns `traits_type::eof()`.

basic_istream::putback

```
basic_istream& putback(char_type c);
```

The unformatted input function (page 107) puts back `c`, if possible, as if by calling `rdbuf()->sputbackc()`. If `rdbuf()` is a null pointer, or if the call to `sputbackc` returns `traits_type::eof()`, the function calls `setstate(badbit)`. In any case, it returns `*this`.

basic_istream::read

```
basic_istream& read(char_type *s, streamsize n);
```

The unformatted input function (page 107) extracts up to `n` elements and stores them in the array beginning at `s`. Extraction stops early on end-of-file, in which case the function calls `setstate(failbit)`. In any case, it returns `*this`.

basic_istream::readsome

```
streamsize readsome(char_type *s, streamsize n);
```

The member function extracts up to `n` elements and stores them in the array beginning at `s`. If `rdbuf()` is a null pointer, the function calls `setstate(failbit)`. Otherwise, it assigns the value of `rdbuf()->in_avail()` to `N`. if `N < 0`, the function calls `setstate(eofbit)`. Otherwise, it replaces the value stored in `N` with the smaller of `n` and `N`, then calls `read(s, N)`. In any case, the function returns `gcount()`.

basic_istream::seekg

```
basic_istream& seekg(pos_type pos);
basic_istream& seekg(off_type off,
    ios_base::seek_dir way);
```

If `fail()` is false, the first member function calls `rdbuf()->pubseekpos(pos)`. If `fail()` is false, the second function calls `rdbuf()->pubseekoff(off, way)`. Both functions return `*this`.

basic_istream::sentry

```
class sentry {
public:
    explicit sentry(basic_istream& is,
        bool noskip = false);
    operator bool() const;
};
```

The nested class describes an object whose declaration structures the formatted input functions (page 107) and the unformatted input functions (page 107). The constructor effectively calls `is.ipfx(noskip)` and stores the return value. `operator bool()` delivers this return value. The destructor effectively calls `is.isfx()`.

basic_istream::sync

```
int sync();
```

If `rdbuf()` is a null pointer, the function returns -1. Otherwise, it calls `rdbuf()->pubsync()`. If that returns -1, the function calls `setstate(badbit)` and returns -1. Otherwise, the function returns zero.

basic_istream::tellg

```
pos_type tellg();
```

If `fail()` is false, the member function returns `rdbuf()->pubseekoff(0, cur, in)`. Otherwise, it returns `pos_type(-1)`.

basic_istream::unget

```
basic_istream& unget();
```

The unformatted input function (page 107) puts back the previous element in the stream, if possible, as if by calling `rdbuf()->sungetc()`. If `rdbuf()` is a null pointer, or if the call to `sungetc` returns `traits_type::eof()`, the function calls `setstate(badbit)`. In any case, it returns `*this`.

iostream

```
typedef basic_istream<char, char_traits<char> > iostream;
```

The type is a synonym for template class `basic_istream` (page 106), specialized for elements of type `char` with default character traits (page 211).

istream

```
typedef basic_istream<char, char_traits<char> > istream;
```

The type is a synonym for template class `basic_istream` (page 106), specialized for elements of type `char` with default character traits (page 211).

operator>>

```
template<class E, class T>
    basic_istream<E, T>&
        operator>>(basic_istream<E, T>& is, E *s);
template<class E, class T>
    basic_istream<E, T>&
```



```

        operator>>(basic_istream<E, T>& is, E& c);
template<class T>
    basic_istream<char, T>&
        operator>>(basic_istream<char, T>& is,
            signed char *s);
template<class T>
    basic_istream<char, T>&
        operator>>(basic_istream<char, T>& is,
            signed char& c);
template<class T>
    basic_istream<char, T>&
        operator>>(basic_istream<char, T>& is,
            unsigned char *s);
template<class T>
    basic_istream<char, T>&
        operator>>(basic_istream<char, T>& is,
            unsigned char& c);

```

The template function:

```

template<class E, class T>
    basic_istream<E, T>&
        operator>>(basic_istream<E, T>& is, E *s);

```

extracts up to $n - 1$ elements and stores them in the array beginning at s . If $is.width()$ is greater than zero, n is $is.width()$; otherwise it is the largest array of E that can be declared. The function always stores $E()$ after any extracted elements it stores. Extraction stops early on end-of-file or on any element (which is not extracted) that would be discarded by ws (page 114). If the function extracts no elements, it calls $is.setstate(failbit)$. In any case, it calls $is.width(0)$ and returns is .

The template function:

```

template<class E, class T>
    basic_istream<E, T>&
        operator>>(basic_istream<E, T>& is, char& c);

```

extracts an element, if possible, and stores it in c . Otherwise, it calls $is.setstate(failbit)$. In any case, it returns is .

The template function:

```

template<class T>
    basic_istream<char, T>&
        operator>>(basic_istream<char, T>& is,
            signed char *s);

```

returns $is >> (\text{char} *)s$.

The template function:

```

template<class T>
    basic_istream<char, T>&
        operator>>(basic_istream<char, T>& is,
            signed char& c);

```

returns $is >> (\text{char}\&)c$.

The template function:

```

template<class T>
    basic_istream<char, T>&
        operator>>(basic_istream<char, T>& is,
            unsigned char *s);

```

returns `is >> (char *)s`.

The template function:

```
template<class T>
    basic_istream<char, T>&
        operator>>(basic_istream<char, T>& is,
                    unsigned char& c);
```

returns `is >> (char&)c`.

wiostream

```
typedef basic_istream<wchar_t, char_traits<wchar_t> >
    wiostream;
```

The type is a synonym for template class `basic_istream` (page 106), specialized for elements of type `wchar_t` with default character traits (page 211).

wistream

```
typedef basic_istream<wchar_t, char_traits<wchar_t> >
    wistream;
```

The type is a synonym for template class `basic_istream` (page 106), specialized for elements of type `wchar_t` with default character traits (page 211).

WS

```
template class<E, T>
    basic_istream<E, T>& ws(basic_istream<E, T>& is);
```

The manipulator extracts and discards any elements `x` for which `use_facet<ctype<E> >(getloc()).is(ctype<E>::space, x)` is true.

The function calls `setstate(eofbit)` if it encounters end-of-file while extracting elements. It returns `is`.

<limits>

```
namespace std {
    enum float_denorm_style;
    enum float_round_style;
    template<class T>
        class numeric_limits;
};
```

Include the standard header `<limits>` to define the template class `numeric_limits`. Explicit specializations of this class describe many arithmetic properties of the scalar types (other than pointers).

float_denorm_style

```
enum float_denorm_style {
    denorm_indeterminate = -1,
    denorm_absent = 0,
    denorm_present = 1
};
```

The enumeration describes the various methods that an implementation can choose for representing a denormalized floating-point value — one too small to represent as a normalized value:

- **denorm_indeterminate** — presence or absence of denormalized forms cannot be determined at translation time
- **denorm_absent** — denormalized forms are absent
- **denorm_present** — denormalized forms are present

float_round_style

```
enum float_round_style {  
    round_indeterminate = -1,  
    round_toward_zero = 0,  
    round_to_nearest = 1,  
    round_toward_infinity = 2,  
    round_toward_neg_infinity = 3  
};
```

The enumeration describes the various methods that an implementation can choose for rounding a floating-point value to an integer value:

- **round_indeterminate** — rounding method cannot be determined
- **round_toward_zero** — round toward zero
- **round_to_nearest** — round to nearest integer
- **round_toward_infinity** — round away from zero
- **round_toward_neg_infinity** — round to more negative integer

numeric_limits

```
template<class T>  
class numeric_limits {  
public:  
    static const float_denorm_style has_denorm  
        = denorm_absent;  
    static const bool has_denorm_loss = false;  
    static const bool has_infinity = false;  
    static const bool has_quiet_NaN = false;  
    static const bool has_signaling_NaN = false;  
    static const bool is_bounded = false;  
    static const bool is_exact = false;  
    static const bool is_iec559 = false;  
    static const bool is_integer = false;  
    static const bool is_modulo = false;  
    static const bool is_signed = false;  
    static const bool is_specialized = false;  
    static const bool tinyness_before = false;  
    static const bool traps = false;  
    static const float_round_style round_style =  
        round_toward_zero;  
    static const int digits = 0;  
    static const int digits10 = 0;  
    static const int max_exponent = 0;  
    static const int max_exponent10 = 0;  
    static const int min_exponent = 0;  
    static const int min_exponent10 = 0;  
    static const int radix = 0;  
    static T denorm_min() throw();  
    static T epsilon() throw();  
    static T infinity() throw();  
    static T max() throw();  
    static T min() throw();
```

```
static T quiet_NaN() throw();
static T round_error() throw();
static T signaling_NaN() throw();
};
```

The template class describes many arithmetic properties of its parameter type `T`. The header defines explicit specializations for the types `wchar_t`, `bool`, `char`, *signed char*, *unsigned char*, *short*, *unsigned short*, *int*, *unsigned int*, *long*, *unsigned long*, *float*, *double*, and *long double*. For all these explicit specializations, the member `is_specialized` is true, and all relevant members have meaningful values. The program can supply additional explicit specializations.

For an arbitrary specialization, *no* members have meaningful values. A member object that does not have a meaningful value stores zero (or false) and a member function that does not return a meaningful value returns `T(0)`.

numeric_limits::denorm_min

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

The function returns the minimum value for the type (which is the same as `min()` if `has_denorm` is not equal to `denorm_present`).

numeric_limits::digits

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

The member stores the number of radix (page 119) digits that the type can represent without change (which is the number of bits other than any sign bit for a predefined integer type, or the number of mantissa digits for a predefined floating-point type).

numeric_limits::digits10

```
static const int digits10 = 0;
```

The member stores the number of decimal digits that the type can represent without change.

numeric_limits::epsilon

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

The function returns the difference between 1 and the smallest value greater than 1 that is representable for the type (which is the value `FLT_EPSILON` for type *float*).

numeric_limits::has_denorm

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

The member stores `denorm_present` (page 115) for a floating-point type that has denormalized values (effectively a variable number of exponent bits).

numeric_limits::has_denorm_loss

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

The member stores true for a type that determines whether a value has lost accuracy because it is delivered as a denormalized result (too small to represent as a normalized value) or because it is inexact (not the same as a result not subject to limitations of exponent range and precision), an option with IEC 559 (page 117) floating-point representations that can affect some results.

numeric_limits::has_infinity

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

The member stores true for a type that has a representation for positive infinity. True if `is_iec559` (page 117) is true.

numeric_limits::has_quiet_NaN

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

The member stores true for a type that has a representation for a **quiet NaN**, an encoding that is “Not a Number” which does not signal its presence in an expression. True if `is_iec559` (page 117) is true.

numeric_limits::has_signaling_NaN

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

The member stores true for a type that has a representation for a **signaling NaN**, an encoding that is “Not a Number” which signals its presence in an expression by reporting an exception. True if `is_iec559` (page 117) is true.

numeric_limits::infinity

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

The function returns the representation of positive infinity for the type. The return value is meaningful only if `has_infinity` (page 117) is true.

numeric_limits::is_bounded

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

The member stores true for a type that has a bounded set of representable values (which is the case for all predefined types).

numeric_limits::is_exact

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

The member stores true for a type that has exact representations for all its values (which is the case for all predefined integer types). A fixed-point or rational representation is also considered exact, but not a floating-point representation.

numeric_limits::is_iec559

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

The member stores true for a type that has a representation conforming to **IEC 559**, an international standard for representing floating-point values (also known as **IEEE 754** in the USA).

numeric_limits::is_integer

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

The member stores true for a type that has an integer representation (which is the case for all predefined integer types).

numeric_limits::is_modulo

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

The member stores true for a type that has a **modulo representation**, where all results are reduced modulo some value (which is the case for all predefined unsigned integer types).

numeric_limits::is_signed

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

The member stores true for a type that has a signed representation (which is the case for all predefined floating-point and signed integer types).

numeric_limits::is_specialized

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

The member stores true for a type that has an explicit specialization defined for template class `numeric_limits` (page 115) (which is the case for all scalar types other than pointers).

numeric_limits::max

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

The function returns the maximum finite value for the type (which is `INT_MAX` for type *int* and `FLT_MAX` for type *float*). The return value is meaningful if `is_bounded` (page 117) is true.

numeric_limits::max_exponent

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

The member stores the maximum positive integer such that the type can represent as a finite value radix (page 119) raised to that power (which is the value `FLT_MAX_EXP` for type *float*). Meaningful only for floating-point types.

numeric_limits::max_exponent10

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

The member stores the maximum positive integer such that the type can represent as a finite value 10 raised to that power (which is the value `FLT_MAX_10_EXP` for type *float*). Meaningful only for floating-point types.

numeric_limits::min

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

The function returns the minimum normalized value for the type (which is `INT_MIN` for type *int* and `FLT_MIN` for type *float*). The return value is meaningful if `is_bounded` (page 117) is true or `is_bounded` is false and `is_signed` (page 118) is false.

numeric_limits::min_exponent

```
static const int min_exponent = 0;
```

The member stores the minimum negative integer such that the type can represent as a normalized value radix (page 119) raised to that power (which is the value `FLT_MIN_EXP` for type *float*). Meaningful only for floating-point types.

numeric_limits::min_exponent10

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

The member stores the minimum negative integer such that the type can represent as a normalized value 10 raised to that power (which is the value `FLT_MIN_10_EXP` for type *float*). Meaningful only for floating-point types.

numeric_limits::quiet_NaN

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

The function returns a representation of a quiet NaN (page 117) for the type. The return value is meaningful only if `has_quiet_NaN` (page 117) is true.

numeric_limits::radix

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

The member stores the base of the representation for the type (which is 2 for the predefined integer types, and the base to which the exponent is raised, or `FLT_RADIX`, for the predefined floating-point types).

numeric_limits::round_error

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

The function returns the maximum rounding error for the type.

numeric_limits::round_style

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

The member stores a value that describes the various methods that an implementation can choose for rounding a floating-point value to an integer value.

numeric_limits::signaling_NaN

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

The function returns a representation of a signaling NaN (page 117) for the type. The return value is meaningful only if `has_signaling_NaN` (page 117) is true.

numeric_limits::tinyness_before

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

The member stores true for a type that determines whether a value is “tiny” (too small to represent as a normalized value) before rounding, an option with IEC 559 (page 117) floating-point representations that can affect some results.

numeric_limits::traps

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

The member stores true for a type that generates some kind of signal to report certain arithmetic exceptions.

<locale>

`codecvt` (page 121) · `codecvt_base` (page 125) · `codecvt_byname` (page 125) · `collate` (page 126) · `collate_byname` (page 127) · `ctype` (page 127) · `ctype<char>` (page 131) · `ctype_base` (page 132) · `ctype_byname` (page 133) · `has_facet` (page 133) · `locale` (page 134) · `messages` (page 139) · `messages_base` (page 140) · `messages_byname` (page 140) · `money_base` (page 141) · `money_get` (page 141) · `money_put` (page 143) · `moneypunct` (page 145) · `moneypunct_byname` (page 149) · `num_get` (page 149) · `num_put` (page 152) · `numpunct` (page 156) · `numpunct_byname` (page 158)

· **time_base** (page 158) · **time_get** (page 158) · **time_get_byname** (page 162) ·
time_put (page 162) · **time_put_byname** (page 163) · **use_facet** (page 164)

isalnum (page 133) · **isalpha** (page 133) · **iscntrl** (page 133) · **isdigit** (page 133) ·
isgraph (page 134) · **islower** (page 134) · **isprint** (page 134) · **ispunct** (page 134) ·
isspace (page 134) · **isupper** (page 134) · **isxdigit** (page 134) · **tolower** (page 163) ·
toupper (page 164)

```
namespace std {
    class locale;
    class ctype_base;
    template<class E>
        class ctype;
    template<>
        class ctype<char>;
    template<class E>
        class ctype_byname;
    class codecvt_base;
    template<class From, class To, class State>
        class codecvt;
    template<class From, class To, class State>
        class codecvt_byname;
    template<class E, class InIt>
        class num_get;
    template<class E, class OutIt>
        class num_put;
    template<class E>
        class numpunct;
    template<class E>
        class numpunct_byname;
    template<class E>
        class collate;
    template<class E>
        class collate_byname;
    class time_base;
    template<class E, class InIt>
        class time_get;
    template<class E, class InIt>
        class time_get_byname;
    template<class E, class OutIt>
        class time_put;
    template<class E, class OutIt>
        class time_put_byname;
    class money_base;
    template<class E, bool Intl, class InIt>
        class money_get;
    template<class E, bool Intl, class OutIt>
        class money_put;
    template<class E, bool Intl>
        class moneypunct;
    template<class E, bool Intl>
        class moneypunct_byname;
    class messages_base;
    template<class E>
        class messages;
    template<class E>
        class messages_byname;

    // TEMPLATE FUNCTIONS
    template<class Facet>
        bool has_facet(const locale& loc);
    template<class Facet>
        const Facet& use_facet(const locale& loc);
    template<class E>
        bool isspace(E c, const locale& loc) const;
    template<class E>
```



```

        bool isprint(E c, const locale& loc) const;
template<class E>
        bool isctrl(E c, const locale& loc) const;
template<class E>
        bool isupper(E c, const locale& loc) const;
template<class E>
        bool islower(E c, const locale& loc) const;
template<class E>
        bool isalpha(E c, const locale& loc) const;
template<class E>
        bool isdigit(E c, const locale& loc) const;
template<class E>
        bool ispunct(E c, const locale& loc) const;
template<class E>
        bool isxdigit(E c, const locale& loc) const;
template<class E>
        bool isalnum(E c, const locale& loc) const;
template<class E>
        bool isgraph(E c, const locale& loc) const;
template<class E>
        E toupper(E c, const locale& loc) const;
template<class E>
        E tolower(E c, const locale& loc) const;
};

```

Include the standard header **<locale>** to define a host of template classes and functions that encapsulate and manipulate locales.

codecvt

```

template<class From, class To, class State>
class codecvt
    : public locale::facet, codecvt_base {
public:
    typedef From intern_type;
    typedef To extern_type;
    typedef State state_type;
    explicit codecvt(size_t refs = 0);
    result in(State& state,
        const To *first1, const To *last1,
        const To *next1,
        From *first2, From *last2, From *next2);
    result out(State& state,
        const From *first1, const From *last1,
        const From *next1,
        To *first2, To *last2, To *next2);
    result unshift(State& state,
        To *first2, To *last2, To *next2);
    bool always_noconv() const throw();
    int max_length() const throw();
    int length(State& state,
        const To *first1, const To *last1,
        size_t _N2) const throw();
    int encoding() const throw();
    static locale::id id;
protected:
    ~codecvt();
    virtual result do_in(State& state,
        const To *first1, const To *last1,
        const To *next1,
        From *first2, From *last2, From *next2);
    virtual result do_out(State& state,
        const From *first1, const From *last1,
        const From *next1,
        To *first2, To *last2, To *next2);
    virtual result do_unshift(State& state,

```

```

        To *first2, To *last2, To *next2);
virtual bool do_always_noconv() const throw();
virtual int do_max_length() const throw();
virtual int do_encoding() const throw();
virtual int do_length(State& state,
        const To *first1, const To *last1,
        size_t len2) const throw();
};

```

The template class describes an object that can serve as a locale facet (page 135), to control conversions between a sequence of values of type `From` and a sequence of values of type `To`. The class `State` characterizes the transformation — and an object of class `State` stores any necessary state information during a conversion.

As with any locale facet, the static object `id` has an initial stored value of zero. The first attempt to access its stored value stores a unique positive value in `id`.

The template versions of `do_in` and `do_out` always return `codecvt_base::noconv`. The Standard C++ library defines an explicit specialization, however, that is more useful:

```

template<>
    codecvt<wchar_t, char, mbstate_t>

```

which converts between `wchar_t` and *char* sequences.

codecvt::always_noconv

```

bool always_noconv() const throw();

```

The member function returns `do_always_noconv()`.

codecvt::codecvt

```

explicit codecvt(size_t refs = 0);

```

The constructor initializes its `locale::facet` base object with `locale::facet(refs)`.

codecvt::do_always_noconv

```

virtual bool do_always_noconv() const throw();

```

The protected virtual member function returns true only if every call to `do_in` (page 122) or `do_out` (page 123) returns `noconv` (page 125). The template version always returns true.

codecvt::do_encoding

```

virtual int do_encoding() const throw();

```

The protected virtual member function returns:

- -1, if the encoding of sequences of type `extern_type` is state dependent
- 0, if the encoding involves sequences of varying lengths
- *n*, if the encoding involves only sequences of length *n*

codecvt::do_in

```

virtual result do_in(State state&,
        const To *first1, const To *last1, const To *next1,
        From *first2, From *last2, From *next2);

```

The protected virtual member function endeavors to convert the source sequence at [`first1`, `last1`) to a destination sequence that it stores within [`first2`, `last2`). It

always stores in `next1` a pointer to the first unconverted element in the source sequence, and it always stores in `next2` a pointer to the first unaltered element in the destination sequence.

state must represent the initial conversion state (page 12) at the beginning of a new source sequence. The function alters its stored value, as needed, to reflect the current state of a successful conversion. Its stored value is otherwise unspecified.

The function returns:

- `codecvt_base::error` if the source sequence is ill formed
- `codecvt_base::noconv` if the function performs no conversion
- `codecvt_base::ok` if the conversion succeeds
- `codecvt_base::partial` if the source is insufficient, or if the destination is not large enough, for the conversion to succeed

The template version always returns `noconv`.

codecvt::do_length

```
virtual int do_length(State state&,
    const To *first1, const To *last1,
    size_t len2) const throw();
```

The protected virtual member function effectively calls `do_in(state, first1, last1, next1, buf, buf + len2, next2)` for some buffer `buf` and pointers `next1` and `next2`, then returns `next2 - buf`. (Thus, it counts the maximum number of conversions, not greater than `len2`, defined by the source sequence at `[first1, last1)`.)

The template version always returns the lesser of `last1 - first1` and `len2`.

codecvt::do_max_length

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

The protected virtual member function returns the largest permissible value that can be returned by `do_length(first1, last1, 1)`, for arbitrary valid values of `first1` and `last1`. (Thus, it is roughly analogous to the macro `MB_CUR_MAX`, at least when `To` is type `char`.)

The template version always returns 1.

codecvt::do_out

```
virtual result do_out(State state&,
    const From *first1, const From *last1,
    const From *next1,
    To *first2, To *last2, To *next2);
```

The protected virtual member function endeavors to convert the source sequence at `[first1, last1)` to a destination sequence that it stores within `[first2, last2)`. It always stores in `next1` a pointer to the first unconverted element in the source sequence, and it always stores in `next2` a pointer to the first unaltered element in the destination sequence.

state must represent the initial conversion state (page 12) at the beginning of a new source sequence. The function alters its stored value, as needed, to reflect the current state of a successful conversion. Its stored value is otherwise unspecified.

The function returns:

- `codecvt_base::error` if the source sequence is ill formed
- `codecvt_base::noconv` if the function performs no conversion
- `codecvt_base::ok` if the conversion succeeds
- `codecvt_base::partial` if the source is insufficient, or if the destination is not large enough, for the conversion to succeed

The template version always returns `noconv`.

codecvt::do_unshift

```
virtual result do_unshift(State state&,
    To *first2, To *last2, To *next2);
```

The protected virtual member function endeavors to convert the source element `From(0)` to a destination sequence that it stores within `[first2, last2)`, except for the terminating element `To(0)`. It always stores in `next2` a pointer to the first unaltered element in the destination sequence.

`state` must represent the initial conversion state (page 12) at the beginning of a new source sequence. The function alters its stored value, as needed, to reflect the current state of a successful conversion. Typically, converting the source element `From(0)` leaves the current state in the initial conversion state.

The function returns:

- `codecvt_base::error` if `state` represents an invalid state
- `codecvt_base::noconv` if the function performs no conversion
- `codecvt_base::ok` if the conversion succeeds
- `codecvt_base::partial` if the destination is not large enough for the conversion to succeed

The template version always returns `noconv`.

codecvt::extern_type

```
typedef To extern_type;
```

The type is a synonym for the template parameter `To`.

codecvt::in

```
result in(State state&,
    const To *first1, const To *last1, const To *next1,
    From *first2, From *last2, From *next2);
```

The member function returns `do_in(state, first1, last1, next1, first2, last2, next2)`.

codecvt::intern_type

```
typedef From intern_type;
```

The type is a synonym for the template parameter `From`.

codecvt::length

```
int length(State state&,
    const To *first1, const To *last1,
    size_t len2) const throw();
```

The member function returns `do_length(first1, last1, len2)`.

codecvt::encoding

```
int encoding() const throw();
```

The member function returns `do_encoding()`.

codecvt::max_length

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

The member function returns `do_max_length()`.

codecvt::out

```
result out(State state&,
           const From *first1, const From *last1,
           const From *next1,
           To *first2, To *last2, To *next2);
```

The member function returns `do_out(state, first1, last1, next1, first2, last2, next2)`.

codecvt::state_type

```
typedef State state_type;
```

The type is a synonym for the template parameter `State`.

codecvt::unshift

```
result unshift(State state&,
               To *first2, To *last2, To *next2);
```

The member function returns `do_unshift(state, first2, last2, next2)`.

codecvt_base

```
class codecvt_base {
public:
    enum result {ok, partial, error, noconv};
};
```

The class describes an enumeration common to all specializations of template class `codecvt` (page 121). The enumeration **result** describes the possible return values from `do_in` (page 122) or `do_out` (page 123):

- **error** if the source sequence is ill formed
- **noconv** if the function performs no conversion
- **ok** if the conversion succeeds
- **partial** if the destination is not large enough for the conversion to succeed

codecvt_byname

```
template<class From, class To, class State>
class codecvt_byname
    : public codecvt<From, To, State> {
public:
    explicit codecvt_byname(const char *s,
                           size_t refs = 0);
protected:
    ~codecvt_byname();
};
```

The template class describes an object that can serve as a locale facet (page 135) of type `codecvt<From, To, State>`. Its behavior is determined by the named (page 136) locale `s`. The constructor initializes its base object with `codecvt<From, To, State>(refs)`.

collate

```
template<class E>
    class collate : public locale::facet {
public:
    typedef E char_type;
    typedef basic_string<E> string_type;
    explicit collate(size_t refs = 0);
    int compare(const E *first1, const E *last1,
                const E *first2, const E *last2) const;
    string_type transform(const E *first,
                           const E *last) const;
    long hash(const E *first, const E *last) const;
    static locale::id id;
protected:
    ~collate();
    virtual int
        do_compare(const E *first1, const E *last1,
                    const E *first2, const E *last2) const;
    virtual string_type do_transform(const E *first,
                                       const E *last) const;
    virtual long do_hash(const E *first,
                          const E *last) const;
    };

```

The template class describes an object that can serve as a locale facet (page 135), to control comparisons of sequences of type `E`.

As with any locale facet, the static object `id` has an initial stored value of zero. The first attempt to access its stored value stores a unique positive value in `id`.

collate::char_type

```
typedef E char_type;
```

The type is a synonym for the template parameter `E`.

collate::collate

```
explicit collate(size_t refs = 0);
```

The constructor initializes its base object with `locale::facet(refs)`.

collate::compare

```
int compare(const E *first1, const E *last1,
            const E *first2, const E *last2) const;
```

The member function returns `do_compare(first1, last1, first2, last2)`.

collate::do_compare

```
virtual int do_compare(const E *first1, const E *last1,
                       const E *first2, const E *last2) const;
```

The protected virtual member function compares the sequence at `[first1, last1)` with the sequence at `[first2, last2)`. It compares values by applying `operator<` between pairs of corresponding elements of type `E`. The first sequence compares less if it has the smaller element in the earliest unequal pair in the sequences, or if no unequal pairs exist but the first sequence is shorter.

If the first sequence compares less than the second sequence, the function returns -1. If the second sequence compares less, the function returns +1. Otherwise, the function returns zero.

collate::do_hash

```
virtual long do_hash(const E *first,  
                     const E *last) const;
```

The protected virtual member function returns an integer derived from the values of the elements in the sequence [first, last). Such a **hash** value can be useful, for example, in distributing sequences pseudo randomly across an array of lists.

collate::do_transform

```
virtual string_type do_transform(const E *first,  
                                 const E *last) const;
```

The protected virtual member function returns an object of class `string_type` (page 127) whose controlled sequence is a copy of the sequence [first, last). If a class derived from `collate<E>` overrides `do_compare` (page 126), it should also override `do_transform` to match. Put simply, two transformed strings should yield the same result, when passed to `collate::compare`, that you would get from passing the untransformed strings to `compare` in the derived class.

collate::hash

```
long hash(const E *first, const E *last) const;
```

The member function returns `do_hash(first, last)`.

collate::string_type

```
typedef basic_string<E> string_type;
```

The type describes a specialization of template class `basic_string` (page 197) whose objects can store copies of the source sequence.

collate::transform

```
string_type transform(const E *first,  
                     const E *last) const;
```

The member function returns `do_transform(first, last)`.

collate_byname

```
template<class E>  
    class collate_byname : public collate<E> {  
public:  
    explicit collate_byname(const char *s,  
                           size_t refs = 0);  
protected:  
    ~collate_byname();  
    };  
};
```

The template class describes an object that can serve as a locale facet (page 135) of type `collate<E>`. Its behavior is determined by the named (page 136) locale `s`. The constructor initializes its base object with `collate<E>(refs)`.

ctype

`char_type` (page 129) · `ctype` (page 129) · `do_is` (page 129) · `do_narrow` (page 129) · `do_scan_is` (page 129) · `do_scan_not` (page 129) · `do_tolower` (page 130) ·

do_toupper (page 130) · **do_widen** (page 130) · **is** (page 130) · **narrow** (page 130) · **scan_is** (page 130) · **scan_not** (page 130) · **tolower** (page 131) · **toupper** (page 131) · **widen** (page 131)

```
template<class E>
class ctype
    : public locale::facet, public ctype_base {
public:
    typedef E char_type;
    explicit ctype(size_t refs = 0);
    bool is(mask msk, E ch) const;
    const E *is(const E *first, const E *last,
        mask *dst) const;
    const E *scan_is(mask msk, const E *first,
        const E *last) const;
    const E *scan_not(mask msk, const E *first,
        const E *last) const;
    E toupper(E ch) const;
    const E *toupper(E *first, E *last) const;
    E tolower(E ch) const;
    const E *tolower(E *first, E *last) const;
    E widen(char ch) const;
    const char *widen(char *first, char *last,
        E *dst) const;
    char narrow(E ch, char dflt) const;
    const E *narrow(const E *first, const E *last,
        char dflt, char *dst) const;
    static locale::id id;
protected:
    ~ctype();
    virtual bool do_is(mask msk, E ch) const;
    virtual const E *do_is(const E *first, const E *last,
        mask *dst) const;
    virtual const E *do_scan_is(mask msk, const E *first,
        const E *last) const;
    virtual const E *do_scan_not(mask msk, const E *first,
        const E *last) const;
    virtual E do_toupper(E ch) const;
    virtual const E *do_toupper(E *first, E *last) const;
    virtual E do_tolower(E ch) const;
    virtual const E *do_tolower(E *first, E *last) const;
    virtual E do_widen(char ch) const;
    virtual const char *do_widen(char *first, char *last,
        E *dst) const;
    virtual char do_narrow(E ch, char dflt) const;
    virtual const E *do_narrow(const E *first,
        const E *last, char dflt, char *dst) const;
};
```

The template class describes an object that can serve as a locale facet (page 135), to characterize various properties of a “character” (element) of type E. Such a facet also converts between sequences of E elements and sequences of *char*.

As with any locale facet, the static object **id** has an initial stored value of zero. The first attempt to access its stored value stores a unique positive value in **id**.

The Standard C++ library defines two explicit specializations of this template class:

- `ctype<char>` (page 131), an explicit specialization whose differences are described separately
- `ctype<wchar_t>`, which treats elements as wide characters (page 13)

In this implementation (page 3), other specializations of template class `ctype<E>`:

- convert a value *ch* of type E to a value of type *char* with the expression `(char)ch`

- convert a value *c* of type *char* to a value of type *E* with the expression *E(c)*

All other operations are performed on *char* values the same as for the explicit specialization `ctype<char>`.

ctype::char_type

```
typedef E char_type;
```

The type is a synonym for the template parameter *E*.

ctype::ctype

```
explicit ctype(size_t refs = 0);
```

The constructor initializes its `locale::facet` base object with `locale::facet(refs)`.

ctype::do_is

```
virtual bool do_is(mask msk, E ch) const;
virtual const E *do_is(const E *first, const E *last,
    mask *dst) const;
```

The first protected member template function returns true if `MASK(ch) & msk` is nonzero, where `MASK(ch)` designates the mapping between an element value *ch* and its classification mask, of type *mask* (page 132). The name `MASK` is purely symbolic here; it is not defined by the template class. For an object of class `ctype<char>` (page 131), the mapping is `tab[(unsigned char)(char)ch]`, where `tab` is the stored pointer to the `ctype` mask table (page 132).

The second protected member template function stores in `dst[I]` the value `MASK(first[I]) & msk`, where *I* ranges over the interval `[0, last - first)`.

ctype::do_narrow

```
virtual char do_narrow(E ch, char dflt) const;
virtual const E *do_narrow(const E *first, const E *last,
    char dflt, char *dst) const;
```

The first protected member template function returns `(char)ch`, or `dflt` if that expression is undefined.

The second protected member template function stores in `dst[I]` the value `do_narrow(first[I], dflt)`, for *I* in the interval `[0, last - first)`.

ctype::do_scan_is

```
virtual const E *do_scan_is(mask msk, const E *first,
    const E *last) const;
```

The protected member function returns the smallest pointer *p* in the range `[first, last)` for which `do_is(msk, *p)` is true. If no such value exists, the function returns `last`.

ctype::do_scan_not

```
virtual const E *do_scan_not(mask msk, const E *first,
    const E *last) const;
```

The protected member function returns the smallest pointer *p* in the range `[first, last)` for which `do_is(msk, *p)` is false. If no such value exists, the function returns `last`.

ctype::do_tolower

```
virtual E do_tolower(E ch) const;  
virtual const E *do_tolower(E *first, E *last) const;
```

The first protected member template function returns the lowercase character corresponding to *ch*, if such a character exists. Otherwise, it returns *ch*.

The second protected member template function replaces each element *first[I]*, for *I* in the interval $[0, \text{last} - \text{first})$, with *do_tolower*(*first[I]*).

ctype::do_toupper

```
virtual E do_toupper(E ch) const;  
virtual const E *do_toupper(E *first, E *last) const;
```

The first protected member template function returns the uppercase character corresponding to *ch*, if such a character exists. Otherwise, it returns *ch*.

The second protected member template function replaces each element *first[I]*, for *I* in the interval $[0, \text{last} - \text{first})$, with *do_toupper*(*first[I]*).

ctype::do_widen

```
virtual E do_widen(char ch) const;  
virtual const char *do_widen(char *first, char *last,  
                             E *dst) const;
```

The first protected member template function returns *E*(*ch*).

The second protected member template function stores in *dst[I]* the value *do_widen*(*first[I]*), for *I* in the interval $[0, \text{last} - \text{first})$.

ctype::is

```
bool is(mask msk, E ch) const;  
const E *is(const E *first, const E *last,  
            mask *dst) const;
```

The first member function returns *do_is*(*msk*, *ch*). The second member function returns *do_is*(*first*, *last*, *dst*).

ctype::narrow

```
char narrow(E ch, char dflt) const;  
const E *narrow(const E *first, const E *last,  
                char dflt, char *dst) const;
```

The first member function returns *do_narrow*(*ch*, *dflt*). The second member function returns *do_narrow*(*first*, *last*, *dflt*, *dst*).

ctype::scan_is

```
const E *scan_is(mask msk, const E *first,  
                 const E *last) const;
```

The member function returns *do_scan_is*(*msk*, *first*, *last*).

ctype::scan_not

```
const E *scan_not(mask msk, const E *first,  
                  const E *last) const;
```

The member function returns *do_scan_not*(*msk*, *first*, *last*).

ctype::tolower

```
E tolower(E ch) const;
const E *tolower(E *first, E *last) const;
```

The first member function returns `do_tolower(ch)`. The second member function returns `do_tolower(first, last)`.

ctype::toupper

```
E toupper(E ch) const;
const E *toupper(E *first, E *last) const;
```

The first member function returns `do_toupper(ch)`. The second member function returns `do_toupper(first, last)`.

ctype::widen

```
E widen(char ch) const;
const char *widen(char *first, char *last, E *dst) const;
```

The first member function returns `do_widen(ch)`. The second member function returns `do_widen(first, last, dst)`.

ctype<char>

```
template<>
class ctype<char>
: public locale::facet, public ctype_base {
public:
    typedef char char_type;
    explicit ctype(const mask *tab = 0, bool del = false,
        size_t refs = 0);
    bool is(mask msk, char ch) const;
    const char *is(const char *first, const char *last,
        mask *dst) const;
    const char *scan_is(mask msk,
        const char *first, const char *last) const;
    const char *scan_not(mask msk,
        const char *first, const char *last) const;
    char toupper(char ch) const;
    const char *toupper(char *first, char *last) const;
    char tolower(char ch) const;
    const char *tolower(char *first, char *last) const;
    char widen(char ch) const;
    const char *widen(char *first, char *last,
        char *dst) const;
    char narrow(char ch, char dflt) const;
    const char *narrow(const char *first,
        const char *last, char dflt, char *dst) const;
    static locale::id id;
protected:
    ~ctype();
    virtual char do_toupper(char ch) const;
    virtual const char *do_toupper(char *first,
        char *last) const;
    virtual char do_tolower(char ch) const;
    virtual const char *do_tolower(char *first,
        char *last) const;
    virtual char do_widen(char ch) const;
    virtual const char *do_widen(char *first, char *last,
        char *dst) const;
    virtual char do_narrow(char ch, char dflt) const;
    virtual const char *do_narrow(const char *first,
        const char *last, char dflt, char *dst) const;
```

```

const mask *table() const throw();
static const mask *classic_table() const throw();
static const size_t table_size;
};

```

The class is an explicit specialization of template class `ctype` (page 127) for type `char`. Hence, it describes an object that can serve as a locale facet (page 135), to characterize various properties of a “character” (element) of type `char`. The explicit specialization differs from the template class in several ways:

- An object of class `ctype<char>` stores a pointer to the first element of a **ctype mask table**, an array of `UCHAR_MAX + 1` elements of type `ctype_base::mask`. It also stores a boolean object that indicates whether the array should be deleted when the `ctype<E>` object is destroyed.
- Its sole public constructor lets you specify `tab`, the `ctype` mask table, and `del`, the boolean object that is true if the array should be deleted when the `ctype<char>` object is destroyed — as well as the usual reference-count parameter `refs`.
- The protected member function `table()` returns the stored `ctype` mask table.
- The static member object `table_size` specifies the minimum number of elements in a `ctype` mask table.
- The protected static member function `classic_table()` returns the `ctype` mask table appropriate to the “C” locale.
- There are no protected virtual member functions `do_is` (page 129), `do_scan_is` (page 129), or `do_scan_not` (page 129). The corresponding public member functions perform the equivalent operations themselves.
- The member functions `do_narrow` (page 129) and `do_widen` (page 130) simply copy elements unaltered.

ctype_base

```

class ctype_base {
public:
    enum mask;
    static const mask space, print, cntrl,
        upper, lower, digit, punct, xdigit,
        alpha, alnum, graph;
};

```

The class serves as a base class for facets of template class `ctype` (page 127). It defines just the enumerated type **mask** and several constants of this type. Each of the constants characterizes a different way to classify characters, as defined by the functions with similar names declared in the header `<ctype.h>`. The constants are:

- **space** (function `isspace`)
- **print** (function `isprint`)
- **cntrl** (function `iscntrl`)
- **upper** (function `isupper`)
- **lower** (function `islower`)
- **digit** (function `isdigit`)
- **punct** (function `ispunct`)
- **xdigit** (function `isxdigit`)
- **alpha** (function `isalpha`)
- **alnum** (function `isalnum`)
- **graph** (function `isgraph`)

You can characterize a combination of classifications by ORing these constants. In particular, it is always true that `alnum == (alpha | digit)` and `graph == (alnum | punct)`.

ctype_byname

```
template<class E>
    class ctype_byname : public ctype<E> {
public:
    explicit ctype_byname(const char *s,
        size_t refs = 0);
protected:
    ~ctype_byname();
    };
```

The template class describes an object that can serve as a locale facet (page 135) of type `ctype<E>`. Its behavior is determined by the named (page 136) locale `s`. The constructor initializes its base object with `ctype<E>(refs)` (or the equivalent for base class `ctype<char>` (page 131)).

has_facet

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

The template function returns true if a locale facet (page 135) of class `Facet` is listed within the locale object (page 135) `loc`.

isalnum

```
template<class E>
    bool isalnum(E c, const locale& loc) const;
```

The template function returns `use_facet< ctype<E> >(loc). is(ctype<E>:: alnum, c)`.

isalpha

```
template<class E>
    bool isalpha(E c, const locale& loc) const;
```

The template function returns `use_facet< ctype<E> >(loc). is(ctype<E>:: alpha, c)`.

iscntrl

```
template<class E>
    bool iscntrl(E c, const locale& loc) const;
```

The template function returns `use_facet< ctype<E> >(loc). is(ctype<E>:: cntrl, c)`.

isdigit

```
template<class E>
    bool isdigit(E c, const locale& loc) const;
```

The template function returns `use_facet< ctype<E> >(loc). is(ctype<E>:: digit, c)`.

isgraph

```
template<class E>
    bool isgraph(E c, const locale& loc) const;
```

The template function returns `use_facet< ctype<E> >(loc). is(ctype<E>:: graph, c)`.

islower

```
template<class E>
    bool islower(E c, const locale& loc) const;
```

The template function returns `use_facet< ctype<E> >(loc). is(ctype<E>:: lower, c)`.

isprint

```
template<class E>
    bool isprint(E c, const locale& loc) const;
```

The template function returns `use_facet< ctype<E> >(loc). is(ctype<E>:: print, c)`.

ispunct

```
template<class E>
    bool ispunct(E c, const locale& loc) const;
```

The template function returns `use_facet< ctype<E> >(loc). is(ctype<E>:: punct, c)`.

isspace

```
template<class E>
    bool isspace(E c, const locale& loc) const;
```

The template function returns `use_facet< ctype<E> >(loc). is(ctype<E>:: space, c)`.

isupper

```
template<class E>
    bool isupper(E c, const locale& loc) const;
```

The template function returns `use_facet< ctype<E> >(loc). is(ctype<E>:: upper, c)`.

isxdigit

```
template<class E>
    bool isxdigit(E c, const locale& loc) const;
```

The template function returns `use_facet< ctype<E> >(loc). is(ctype<E>:: xdigit, c)`.

locale

`category` (page 136) · `classic` (page 137) · `combine` (page 137) · `facet` (page 137) · `global` (page 137) · `id` (page 138) · `locale` (page 138) · `name` (page 138) · `operator!=` (page 138) · `operator()` (page 138) · `operator==` (page 139)

```

class locale {
public:
    class facet;
    class id;
    typedef int category;
    static const category none, collate, ctype, monetary,
        numeric, time, messages, all;
    locale();
    explicit locale(const char *s);
    locale(const locale& x, const locale& y,
        category cat);
    locale(const locale& x, const char *s, category cat);
    template<class Facet>
        locale(const locale& x, Facet *fac);
    template<class Facet>
        locale combine(const locale& x) const;
    template<class E, class T, class A>
        bool operator()(const basic_string<E, T, A>& lhs,
            const basic_string<E, T, A>& rhs) const;
    string name() const;
    bool operator==(const locale& x) const;
    bool operator!=(const locale& x) const;
    static locale global(const locale& x);
    static const locale& classic();
};

```

The class describes a **locale object** that encapsulates a locale. It represents culture-specific information as a list of **facets**. A facet is a pointer to an object of a class derived from class `facet` (page 137) that has a public object of the form:

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

You can define an open-ended set of these facets. You can also construct a locale object that designates an arbitrary number of facets.

Predefined groups of these facets represent the locale categories traditionally managed in the Standard C library by the function `setlocale`.

Category **collate** (page 137) (`LC_COLLATE`) includes the facets:

```
collate<char>
collate<wchar_t>
```

Category **ctype** (page 137) (`LC_CTYPE`) includes the facets:

```
ctype<char>
ctype<wchar_t>
codecvt<char, char, mbstate_t>
codecvt<wchar_t, char, mbstate_t>
```

Category **monetary** (page 137) (`LC_MONETARY`) includes the facets:

```
moneypunct<char, false>
moneypunct<wchar_t, false>
moneypunct<char, true>
moneypunct<wchar_t, true>
money_get<char, istreambuf_iterator<char> >
money_get<wchar_t, istreambuf_iterator<wchar_t> >
money_put<char, ostreambuf_iterator<char> >
money_put<wchar_t, ostreambuf_iterator<wchar_t> >
```

Category **numeric** (page 137) (`LC_NUMERIC`) includes the facets:

```

num_get<char, istreambuf_iterator<char> >
num_get<wchar_t, istreambuf_iterator<wchar_t> >
num_put<char, ostreambuf_iterator<char> >
num_put<wchar_t, ostreambuf_iterator<wchar_t> >
numpunct<char>
numpunct<wchar_t>

```

Category **time** (page 137) (LC_TIME) includes the facets:

```

time_get<char, istreambuf_iterator<char> >
time_get<wchar_t, istreambuf_iterator<wchar_t> >
time_put<char, ostreambuf_iterator<char> >
time_put<wchar_t, ostreambuf_iterator<wchar_t> >

```

Category **messages** (page 137) [sic] (LC_MESSAGE) includes the facets:

```

messages<char>
messages<wchar_t>

```

(The last category is required by Posix, but not the C Standard.)

Some of these predefined facets are used by the `iostreams` (page 7) classes, to control the conversion of numeric values to and from text sequences.

An object of class `locale` also stores a **locale name** as an object of class `string` (page 217). Using an invalid locale name to construct a locale facet (page 135) or a locale object throws an object of class `runtime_error` (page 185). The stored locale name is `"*"` if the locale object cannot be certain that a C-style locale corresponds exactly to that represented by the object. Otherwise, you can establish a matching locale within the Standard C library, for the locale object `x`, by calling `setlocale(LC_ALL, x.name.c_str())`.

In this implementation (page 3), you can also call the static member function:

```
static locale empty();
```

to construct a locale object that has no facets. It is also a **transparent locale** — if the template functions `has_facet` (page 133) and `use_facet` (page 164) cannot find the requested facet in a transparent locale, they consult first the global locale (page 137) and then, if that is transparent, the classic locale (page 137). Thus, you can write:

```
cout.imbue(locale::empty());
```

Subsequent insertions to `cout` (page 104) are mediated by the current state of the global locale. You can even write:

```

locale loc(locale::empty(), locale::classic(),
           locale::numeric);
cout.imbue(loc);

```

Numeric formatting rules for subsequent insertions to `cout` remain the same as in the C locale, even as the global locale supplies changing rules for inserting dates and monetary amounts.

locale::category

```

typedef int category;
static const category none, collate, ctype, monetary,
                    numeric, time, messages, all;

```

The type is a synonym for `int`, so that it can represent any of the C locale categories. It can also represent a group of constants local to class `locale`:

- **none**, corresponding to none of the the C categories
- **collate**, corresponding to the C category LC_COLLATE
- **ctype**, corresponding to the C category LC_CTYPE
- **monetary**, corresponding to the C category LC_MONETARY
- **numeric**, corresponding to the C category LC_NUMERIC
- **time**, corresponding to the C category LC_TIME
- **messages**, corresponding to the Posix category LC_MESSAGE
- **all**, corresponding to the C union of all categories LC_ALL

You can represent an arbitrary group of categories by ORing these constants, as in `monetary | time`.

locale::classic

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

The static member function returns a locale object that represents the **classic locale**, which behaves the same as the C locale within the Standard C library.

locale::combine

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

The member function returns a locale object that replaces in (or adds to) `*this` the facet `Facet` listed in `x`.

locale::facet

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

The member class serves as the base class for all locale facets (page 135). Note that you can neither copy nor assign an object of class `facet`. You can construct and destroy objects derived from class `locale::facet`, but not objects of the base class proper. Typically, you construct an object `myfac` derived from `facet` when you construct a locale, as in:

```
locale loc(locale::classic(), new myfac);
```

In such cases, the constructor for the base class `facet` should have a zero `refs` argument. When the object is no longer needed, it is deleted. Thus, you supply a nonzero `refs` argument only in those rare cases where you take responsibility for the lifetime of the object.

locale::global

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

The static member function stores a copy of `x` as the **global locale**. It also calls `setlocale(LC_ALL, x.name. c_str())`, to establishing a matching locale within the Standard C library. The function then returns the previous global locale. At program startup, the global locale is the same as the classic locale (page 137).

locale::id

```
class id {
protected:
    id();
private:
    id(const id&)           // not defined
    void operator=(const id&) // not defined
};
```

The member class describes the static member object required by each unique locale facet (page 135). Note that you can neither copy nor assign an object of class `id`.

locale::locale

```
locale();
explicit locale(const char *s);
locale(const locale& x, const locale& y,
       category cat);
locale(const locale& x, const char *s, category cat);
template<class Facet>
    locale(const locale& x, Facet *fac);
```

The first constructor initializes the object to match the global locale (page 137). The second constructor initializes all the locale categories to have behavior consistent with the locale name (page 136) `s`. The remaining constructors copy `x`, with the exceptions noted:

```
locale(const locale& x, const locale& y,
       category cat);
```

replaces from `y` those facets corresponding to a category `c` for which `c & cat` is nonzero.

```
locale(const locale& x, const char *s, category cat);
```

replaces from `locale(s, all)` those facets corresponding to a category `c` for which `c & cat` is nonzero.

```
template<class Facet>
    locale(const locale& x, Facet *fac);
```

replaces in (or adds to) `x` the facet `fac`, if `fac` is not a null pointer.

If a locale name `s` is a null pointer or otherwise invalid, the function throws `runtime_error` (page 185).

locale::name

```
string name() const;
```

The member function returns the stored locale name (page 136).

locale::operator!=

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

The member function returns `!(*this == x)`.

locale::operator()

```
template<class E, class T, class A>
    bool operator()(const basic_string<E, T, A>& lhs,
                   const basic_string<E, T, A>& rhs);
```

The member function effectively executes:

```
const collate<E>& fac = use_fac<collate<E> >(*this);
return (fac.compare(lhs.begin(), lhs.end(),
    rhs.begin(), rhs.end()) < 0);
```

Thus, you can use a locale object as a function object (page 285).

locale::operator==

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

The member function returns true only if **this* and *x* are copies of the same locale or have the same name (other than "*").

messages

```
template<class E>
class messages
    : public locale::facet, public messages_base {
public:
    typedef E char_type;
    typedef basic_string<E> string_type;
    explicit messages(size_t refs = 0);
    catalog open(const string& name,
        const locale& loc) const;
    string_type get(catalog cat, int set, int msg,
        const string_type& dflt) const;
    void close(catalog cat) const;
    static locale::id id;
protected:
    ~messages();
    virtual catalog do_open(const string& name,
        const locale& loc) const;
    virtual string_type do_get(catalog cat, int set,
        int msg, const string_type& dflt) const;
    virtual void do_close(catalog cat) const;
};
```

The template class describes an object that can serve as a locale facet (page 135), to characterize various properties of a **message catalog** that can supply messages represented as sequences of elements of type *E*.

As with any locale facet, the static object **id** has an initial stored value of zero. The first attempt to access its stored value stores a unique positive value in **id**.

messages::char_type

```
typedef E char_type;
```

The type is a synonym for the template parameter *E*.

messages::close

```
void close(catalog cat) const;
```

The member function calls `do_close(cat);`.

messages::do_close

```
virtual void do_close(catalog cat) const;
```

The protected member function closes the message catalog (page 139) *cat*, which must have been opened by an earlier call to `do_open` (page 140).

messages::do_get

```
virtual string_type do_get(catalog cat, int set, int msg,  
    const string_type& dflt) const;
```

The protected member function endeavors to obtain a message sequence from the message catalog (page 139) *cat*. It may make use of *set*, *msg*, and *dflt* in doing so. It returns a copy of *dflt* on failure. Otherwise, it returns a copy of the specified message sequence.

messages::do_open

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

The protected member function endeavors to open a message catalog (page 139) whose name is *name*. It may make use of the locale *loc* in doing so. It returns a value that compares less than zero on failure. Otherwise, the returned value can be used as the first argument on a later call to *get* (page 140). It should in any case be used as the argument on a later call to *close* (page 139).

messages::get

```
string_type get(catalog cat, int set, int msg,  
    const string_type& dflt) const;
```

The member function returns *do_get*(*cat*, *set*, *msg*, *dflt*);.

messages::messages

```
explicit messages(size_t refs = 0);
```

The constructor initializes its base object with *locale::facet*(*refs*).

messages::open

```
catalog open(const string& name,  
    const locale& loc) const;
```

The member function returns *do_open*(*name*, *loc*);.

messages::string_type

```
typedef basic_string<E> string_type;
```

The type describes a specialization of template class *basic_string* (page 197) whose objects can store copies of the message sequences.

messages_base

```
class messages_base {  
    typedef int catalog;  
};
```

The class describes a type common to all specializations of template class *messages* (page 139). The type **catalog** is a synonym for type *int* that describes the possible return values from *messages::do_open*.

messages_byname

```
template<class E>  
    class messages_byname : public messages<E> {  
public:  
    explicit messages_byname(const char *s,
```

```

        size_t refs = 0);
protected:
    ~messages_byname();
};

```

The template class describes an object that can serve as a locale facet of type `messages<E>`. Its behavior is determined by the named locale `s`. The constructor initializes its base object with `messages<E>(refs)`.

money_base

```

class money_base {
    enum part {none, sign, space, symbol, value};
    struct pattern {
        char field[4];
    };
};

```

The class describes an enumeration and a structure common to all specializations of template class `money_punct` (page 145). The enumeration `part` describes the possible values in elements of the array `field` in the structure `pattern`. The values of `part` are:

- **none** to match zero or more spaces or generate nothing
- **sign** to match or generate a positive or negative sign
- **space** to match zero or more spaces or generate a space
- **symbol** to match or generate a currency symbol
- **value** to match or generate a monetary value

money_get

```

template<class E,
        class InIt = istreambuf_iterator<E> >
class money_get : public locale::facet {
public:
    typedef E char_type;
    typedef InIt iter_type;
    typedef basic_string<E> string_type;
    explicit money_get(size_t refs = 0);
    iter_type get(iter_type first, iter_type last,
        bool intl, ios_base& x, ios_base::iostate& st,
        long double& val) const;
    iter_type get(iter_type first, iter_type last,
        bool intl, ios_base& x, ios_base::iostate& st,
        string_type& val) const;
    static locale::id id;
protected:
    ~money_get();
    virtual iter_type do_get(iter_type first,
        iter_type last, bool intl, ios_base& x,
        ios_base::iostate& st, string_type& val) const;
    virtual iter_type do_get(iter_type first,
        iter_type last, bool intl, ios_base& x,
        ios_base::iostate& st, long double& val) const;
};

```

The template class describes an object that can serve as a locale facet, to control conversions of sequences of type `E` to monetary values.

As with any locale facet, the static object `id` has an initial stored value of zero. The first attempt to access its stored value stores a unique positive value in `id`.

money_get::char_type

typedef E **char_type**;

The type is a synonym for the template parameter E.

money_get::do_get

```
virtual iter_type do_get(iter_type first, iter_type last,  
    bool intl, ios_base& x, ios_base::iostate& st,  
    string_type& val) const;  
virtual iter_type do_get(iter_type first, iter_type last,  
    bool intl, ios_base& x, ios_base::iostate& st,  
    long double& val) const;
```

The first virtual protected member function endeavors to match sequential elements beginning at *first* in the sequence [*first*, *last*) until it has recognized a complete, nonempty **monetary input field**. If successful, it converts this field to a sequence of one or more decimal digits, optionally preceded by a minus sign (-), to represent the amount and stores the result in the *string_type* (page 143) object *val*. It returns an iterator designating the first element beyond the monetary input field. Otherwise, the function stores an empty sequence in *val* and sets *ios_base::failbit* in *st*. It returns an iterator designating the first element beyond any prefix of a valid monetary input field. In either case, if the return value equals *last*, the function sets *ios_base::eofbit* in *st*.

The second virtual protected member function behaves the same as the first, except that if successful it converts the optionally-signed digit sequence to a value of type *long double* and stores that value in *val*.

The format of a monetary input field is determined by the locale facet (page 135) *fac* returned by the (effective) call `use_facet <moneypunct<E, intl> >(x.getloc())`. Specifically:

- *fac.neg_format()* determines the order in which components of the field occur.
- *fac.curr_symbol()* determines the sequence of elements that constitutes a currency symbol.
- *fac.positive_sign()* determines the sequence of elements that constitutes a positive sign.
- *fac.negative_sign()* determines the sequence of elements that constitutes a negative sign.
- *fac.grouping()* determines how digits are grouped to the left of any decimal point.
- *fac.thousands_sep()* determines the element that separates groups of digits to the left of any decimal point.
- *fac.decimal_point()* determines the element that separates the integer digits from the fraction digits.
- *fac.frac_digits()* determines the number of significant fraction digits to the right of any decimal point.

If the sign string (*fac.negative_sign* or *fac.positive_sign*) has more than one element, only the first element is matched where the element equal to **money_base::sign (page 141)** appears in the format pattern (*fac.neg_format*). Any remaining elements are matched at the end of the monetary input field. If neither string has a first element that matches the next element in the monetary input field, the sign string is taken as empty and the sign is positive.

If `x.flags() & showbase` is nonzero, the string `fac.curr_symbol` *must* match where the element equal to **`money_base::symbol`** appears in the format pattern. Otherwise, if `money_base::symbol` occurs at the end of the format pattern, and if no elements of the sign string remain to be matched, the currency symbol is *not* matched. Otherwise, the currency symbol is *optionally* matched.

If no instances of `fac.thousands_sep()` occur in the value portion of the monetary input field (where the element equal to **`money_base::value`** appears in the format pattern), no grouping constraint is imposed. Otherwise, any grouping constraints imposed by `fac.grouping()` is enforced. Note that the resulting digit sequence represents an integer whose low-order `fac.frac_digits()` decimal digits are considered to the right of the decimal point.

Arbitrary white space (page 31) is matched where the element equal to **`money_base::space`** appears in the format pattern, if it appears other than at the end of the format pattern. Otherwise, no internal white space is matched. An element `c` is considered white space if `use_facet<ctype<E>>(x.getloc()).is(ctype_base::space, c)` is true.

money_get::get

```
iter_type get(iter_type first, iter_type last,
              bool intl, ios_base& x, ios_base::iostate& st,
              long double& val) const;
iter_type get(iter_type first, iter_type last,
              bool intl, ios_base& x, ios_base::iostate& st,
              string_type& val) const;
```

Both member functions return `do_get(first, last, intl, x, st, val)`.

money_get::iter_type

```
typedef InIt iter_type;
```

The type is a synonym for the template parameter `InIt`.

money_get::money_get

```
explicit money_get(size_t refs = 0);
```

The constructor initializes its base object with `locale::facet(refs)`.

money_get::string_type

```
typedef basic_string<E> string_type;
```

The type describes a specialization of template class `basic_string` (page 197) whose objects can store sequences of elements from the source sequence.

money_put

```
template<class E,
        class OutIt = ostreambuf_iterator<E> >
class money_put : public locale::facet {
public:
    typedef E char_type;
    typedef OutIt iter_type;
    typedef basic_string<E> string_type;
    explicit money_put(size_t refs = 0);
    iter_type put(iter_type next, bool intl, ios_base& x,
                 E fill, long double& val) const;
    iter_type put(iter_type next, bool intl, ios_base& x,
                 E fill, string_type& val) const;
    static locale::id id;
```

```
protected:
    ~money_put();
    virtual iter_type do_put(iter_type next, bool intl,
        ios_base& x, E fill, string_type& val) const;
    virtual iter_type do_put(iter_type next, bool intl,
        ios_base& x, E fill, long double& val) const;
};
```

The template class describes an object that can serve as a locale facet (page 135), to control conversions of monetary values to sequences of type E.

As with any locale facet, the static object **id** has an initial stored value of zero. The first attempt to access its stored value stores a unique positive value in **id**.

money_put::char_type

```
typedef E char_type;
```

The type is a synonym for the template parameter E.

money_put::do_put

```
virtual iter_type do_put(iter_type next, bool intl,
    ios_base& x, E fill, string_type& val) const;
virtual iter_type do_put(iter_type next, bool intl,
    ios_base& x, E fill, long double& val) const;
```

The first virtual protected member function generates sequential elements beginning at next to produce a **monetary output field** from the string_type (page 145) object val. The sequence controlled by val must begin with one or more decimal digits, optionally preceded by a minus sign (-), which represents the amount. The function returns an iterator designating the first element beyond the generated monetary output field.

The second virtual protected member function behaves the same as the first, except that it effectively first converts val to a sequence of decimal digits, optionally preceded by a minus sign, then converts that sequence as above.

The format of a monetary output field is determined by the locale facet (page 135) fac returned by the (effective) call use_facet <moneypunct<E, intl> >(x.getloc()). Specifically:

- fac.pos_format() determines the order in which components of the field are generated for a non-negative value.
- fac.neg_format() determines the order in which components of the field are generated for a negative value.
- fac.curr_symbol() determines the sequence of elements to generate for a currency symbol.
- fac.positive_sign() determines the sequence of elements to generate for a positive sign.
- fac.negative_sign() determines the sequence of elements to generate for a negative sign.
- fac.grouping() determines how digits are grouped to the left of any decimal point.
- fac.thousands_sep() determines the element that separates groups of digits to the left of any decimal point.
- fac.decimal_point() determines the element that separates the integer digits from any fraction digits.

- `fac.frac_digits()` determines the number of significant fraction digits to the right of any decimal point.

If the sign string (`fac.negative_sign` or `fac.positive_sign`) has more than one element, only the first element is generated where the element equal to **`money_base::sign`** appears in the format pattern (`fac.neg_format` or `fac.pos_format`). Any remaining elements are generated at the end of the monetary output field.

If `x.flags()` & `showbase` is nonzero, the string `fac.curr_symbol` is generated where the element equal to **`money_base::symbol`** appears in the format pattern. Otherwise, no currency symbol is generated.

If no grouping constraints are imposed by `fac.grouping()` (its first element has the value `CHAR_MAX`) then no instances of `fac.thousands_sep()` are generated in the value portion of the monetary output field (where the element equal to **`money_base::value`** appears in the format pattern). If `fac.frac_digits()` is zero, then no instance of `fac.decimal_point()` is generated after the decimal digits. Otherwise, the resulting monetary output field places the low-order `fac.frac_digits()` decimal digits to the right of the decimal point.

Padding (page 154) occurs as for any numeric output field, except that if `x.flags()` & `x.internal` is nonzero, any internal padding is generated where the element equal to **`money_base::space`** appears in the format pattern, if it does appear. Otherwise, internal padding occurs before the generated sequence. The padding character is `fill`.

The function calls `x.width(0)` to reset the field width to zero.

`money_put::put`

```
iter_type put(iter_type next, bool intl, ios_base& x,
              E fill, long double& val) const;
iter_type put(iter_type next, bool intl, ios_base& x,
              E fill, string_type& val) const;
```

Both member functions return `do_put(next, intl, x, fill, val)`.

`money_put::iter_type`

```
typedef InIt iter_type;
```

The type is a synonym for the template parameter `OutIt`.

`money_put::money_put`

```
explicit money_put(size_t refs = 0);
```

The constructor initializes its base object with `locale::facet(refs)`.

`money_put::string_type`

```
typedef basic_string<E> string_type;
```

The type describes a specialization of template class `basic_string` (page 197) whose objects can store sequences of elements from the source sequence.

`moneypunct`

`char_type` (page 146) · `curr_symbol` (page 146) · `decimal_point` (page 146) ·
`do_curr_symbol` (page 147) · `do_decimal_point` (page 147) · `do_frac_digits` (page

147) · do_grouping (page 147) · do_neg_format (page 147) · do_negative_sign (page 147) · do_pos_format (page 147) · do_positive_sign (page 148) · do_thousands_sep (page 148) · frac_digits (page 148) · grouping (page 148) · moneypunct (page 148) · neg_format (page 148) · negative_sign (page 148) · pos_format (page 148) · positive_sign (page 148) · string_type (page 148) · thousands_sep (page 149)

```
template<class E, bool Intl>
    class moneypunct
        : public locale::facet, public money_base {
public:
    typedef E char_type;
    typedef basic_string<E> string_type;
    explicit moneypunct(size_t refs = 0);
    E decimal_point() const;
    E 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 const bool intl = Intl;
    static locale::id id;
protected:
    ~moneypunct();
    virtual E do_decimal_point() const;
    virtual E 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;
};
```

The template class describes an object that can serve as a locale facet (page 135), to describe the sequences of type E used to represent a **monetary input field** (page 142) or a **monetary output field** (page 144). If the template parameter Intl is true, international conventions are observed.

As with any locale facet, the static object **id** has an initial stored value of zero. The first attempt to access its stored value stores a unique positive value in id.

The const static object **intl** stores the value of the template parameter Intl.

moneypunct::char_type

```
typedef E char_type;
```

The type is a synonym for the template parameter E.

moneypunct::curr_symbol

```
string_type curr_symbol() const;
```

The member function returns do_curr_symbol().

moneypunct::decimal_point

```
E decimal_point() const;
```

The member function returns do_decimal_point().

money_punct::do_curr_symbol

string_type **do_curr_symbol**() const;

The protected virtual member function returns a locale-specific sequence of elements to use as a currency symbol.

money_punct::do_decimal_point

E **do_decimal_point**() const;

The protected virtual member function returns a locale-specific element to use as a decimal-point.

money_punct::do_frac_digits

int **do_frac_digits**() const;

The protected virtual member function returns a locale-specific count of the number of digits to display to the right of any decimal point.

money_punct::do_grouping

string **do_grouping**() const;

The protected virtual member function returns a locale-specific rule for determining how digits are grouped to the left of any decimal point. The encoding is the same as for `lconv::grouping`.

money_punct::do_neg_format

pattern **do_neg_format**() const;

The protected virtual member function returns a locale-specific rule for determining how to generate a monetary output field (page 144) for a negative amount. Each of the four elements of `pattern::field` can have the values:

- **none** (page 141) to match zero or more spaces or generate nothing
- **sign** (page 141) to match or generate a positive or negative sign
- **space** (page 141) to match zero or more spaces or generate a space
- **symbol** (page 141) to match or generate a currency symbol
- **value** (page 141) to match or generate a monetary value

Components of a monetary output field are generated (and components of a monetary input field (page 142) are matched) in the order in which these elements appear in `pattern::field`. Each of the values `sign`, `symbol`, `value`, and either `none` or `space` must appear exactly once. The value `none` must not appear first. The value `space` must not appear first or last. If `Intl` is true, the order is `symbol`, `sign`, `none`, then `value`.

The template version of `money_punct<E, Intl>` returns `{money_base::symbol, money_base::sign, money_base::value, money_base::none}`.

money_punct::do_negative_sign

string_type **do_negative_sign**() const;

The protected virtual member function returns a locale-specific sequence of elements to use as a negative sign.

money_punct::do_pos_format

pattern **do_pos_format**() const;

The protected virtual member function returns a locale-specific rule for determining how to generate a monetary output field (page 144) for a positive amount. (It also determines how to match the components of a monetary input field (page 142).) The encoding is the same as for `do_neg_format` (page 147).

The template version of `moneypunct<E, Intl>` returns `{money_base::symbol, money_base::sign, money_base::value, money_base::none}`.

moneypunct::do_positive_sign

`string_type do_positive_sign() const;`

The protected virtual member function returns a locale-specific sequence of elements to use as a positive sign.

moneypunct::do_thousands_sep

`E do_thousands_sep() const;`

The protected virtual member function returns a locale-specific element to use as a group separator to the left of any decimal point.

moneypunct::frac_digits

`int frac_digits() const;`

The member function returns `do_frac_digits()`.

moneypunct::grouping

`string grouping() const;`

The member function returns `do_grouping()`.

moneypunct::moneypunct

`explicit moneypunct(size_t refs = 0);`

The constructor initializes its base object with `locale::facet(refs)`.

moneypunct::neg_format

`pattern neg_format() const;`

The member function returns `do_neg_format()`.

moneypunct::negative_sign

`string_type negative_sign() const;`

The member function returns `do_negative_sign()`.

moneypunct::pos_format

`pattern pos_format() const;`

The member function returns `do_pos_format()`.

moneypunct::positive_sign

`string_type positive_sign() const;`

The member function returns `do_positive_sign()`.

moneypunct::string_type

`typedef basic_string<E> string_type;`

The type describes a specialization of template class `basic_string` (page 197) whose objects can store copies of the punctuation sequences.

moneypunct::thousands_sep

`E thousands_sep() const;`

The member function returns `do_thousands_sep()`.

moneypunct_byname

```
template<class E, bool Intl>
class moneypunct_byname
    : public moneypunct<E, Intl> {
public:
    explicit moneypunct_byname(const char *s,
                               size_t refs = 0);
protected:
    ~moneypunct_byname();
};
```

The template class describes an object that can serve as a locale facet of type `moneypunct<E, Intl>`. Its behavior is determined by the named (page 136) locale `s`. The constructor initializes its base object with `moneypunct<E, Intl>(refs)`.

num_get

```
template<class E, class InIt = istreambuf_iterator<E> >
class num_get : public locale::facet {
public:
    typedef E char_type;
    typedef InIt iter_type;
    explicit num_get(size_t refs = 0);
    iter_type get(iter_type first, iter_type last,
                  ios_base& x, ios_base::iostate& st,
                  long& val) const;
    iter_type get(iter_type first, iter_type last,
                  ios_base& x, ios_base::iostate& st,
                  unsigned long& val) const;
    iter_type get(iter_type first, iter_type last,
                  ios_base& x, ios_base::iostate& st,
                  double& val) const;
    iter_type get(iter_type first, iter_type last,
                  ios_base& x, ios_base::iostate& st,
                  long double& val) const;
    iter_type get(iter_type first, iter_type last,
                  ios_base& x, ios_base::iostate& st,
                  void *&val) const;
    iter_type get(iter_type first, iter_type last,
                  ios_base& x, ios_base::iostate& st,
                  bool& val) const;
    static locale::id id;
protected:
    ~num_get();
    virtual iter_type
        do_get(iter_type first, iter_type last,
               ios_base& x, ios_base::iostate& st,
               long& val) const;
    virtual iter_type
        do_get(iter_type first, iter_type last,
               ios_base& x, ios_base::iostate& st,
               unsigned long& val) const;
    virtual iter_type
        do_get(iter_type first, iter_type last,
               ios_base& x, ios_base::iostate& st,
               double& val) const;
```

```

virtual iter_type
    do_get(iter_type first, iter_type last,
           ios_base& x, ios_base::iostate& st,
           long double& val) const;
virtual iter_type
    do_get(iter_type first, iter_type last,
           ios_base& x, ios_base::iostate& st,
           void *& val) const;
virtual iter_type
    do_get(iter_type first, iter_type last,
           ios_base& x, ios_base::iostate& st,
           bool& val) const;
};

```

The template class describes an object that can serve as a locale facet (page 135), to control conversions of sequences of type E to numeric values.

As with any locale facet, the static object **id** has an initial stored value of zero. The first attempt to access its stored value stores a unique positive value in **id**.

num_get::char_type

typedef E **char_type**;

The type is a synonym for the template parameter E.

num_get::do_get

```

virtual iter_type do_get(iter_type first, iter_type last,
                          ios_base& x, ios_base::iostate& st,
                          long& val) const;
virtual iter_type do_get(iter_type first, iter_type last,
                          ios_base& x, ios_base::iostate& st,
                          unsigned long& val) const;
virtual iter_type do_get(iter_type first, iter_type last,
                          ios_base& x, ios_base::iostate& st,
                          double& val) const;
virtual iter_type do_get(iter_type first, iter_type last,
                          ios_base& x, ios_base::iostate& st,
                          long double& val) const;
virtual iter_type do_get(iter_type first, iter_type last,
                          ios_base& x, ios_base::iostate& st,
                          void *& val) const;
virtual iter_type do_get(iter_type first, iter_type last,
                          ios_base& x, ios_base::iostate& st,
                          bool& val) const;

```

The first virtual protected member function endeavors to match sequential elements beginning at **first** in the sequence [**first**, **last**) until it has recognized a complete, nonempty **integer input field**. If successful, it converts this field to its equivalent value as type *long*, and stores the result in **val**. It returns an iterator designating the first element beyond the numeric input field. Otherwise, the function stores nothing in **val** and sets **ios_base::failbit** in **st**. It returns an iterator designating the first element beyond any prefix of a valid integer input field. In either case, if the return value equals **last**, the function sets **ios_base::eofbit** in **st**.

The integer input field is converted by the same rules used by the scan functions (page 25) for matching and converting a series of *char* elements from a file. (Each such *char* element is assumed to map to an equivalent element of type E by a simple, one-to-one, mapping.) The equivalent scan conversion specification (page 25) is determined as follows:

- If `x.flags() & ios_base::basefield == ios_base::oct`, the conversion specification is `lo`.
- If `x.flags() & ios_base::basefield == ios_base::hex`, the conversion specification is `lx`.
- If `x.flags() & ios_base::basefield == 0`, the conversion specification is `li`.
- Otherwise, the conversion specification is `ld`.

The format of an integer input field is further determined by the locale facet (page 135) `fac` returned by the call `use_facet<E>(x.getloc())`. Specifically:

- `fac.grouping()` determines how digits are grouped to the left of any decimal point
- `fac.thousands_sep()` determines the sequence that separates groups of digits to the left of any decimal point

If no instances of `fac.thousands_sep()` occur in the numeric input field, no grouping constraint is imposed. Otherwise, any grouping constraints imposed by `fac.grouping()` is enforced and separators are removed before the scan conversion occurs.

The second virtual protected member function:

```
virtual iter_type do_get(iter_type first, iter_type last,
    ios_base& x, ios_base::iostate& st,
    unsigned long& val) const;
```

behaves the same as the first, except that it replaces a conversion specification of `ld` with `lu`. If successful it converts the numeric input field to a value of type *unsigned long* and stores that value in `val`.

The third virtual protected member function:

```
virtual iter_type do_get(iter_type first, iter_type last,
    ios_base& x, ios_base::iostate& st,
    double& val) const;
```

behaves the same as the first, except that it endeavors to match a complete, nonempty **floating-point input field**. `fac.decimal_point()` determines the sequence that separates the integer digits from the fraction digits. The equivalent scan conversion specifier is `lf`.

The fourth virtual protected member function:

```
virtual iter_type do_get(iter_type first, iter_type last,
    ios_base& x, ios_base::iostate& st,
    long double& val) const;
```

behaves the same the third, except that the equivalent scan conversion specifier is `Lf`.

The fifth virtual protected member function:

```
virtual iter_type do_get(iter_type first, iter_type last,
    ios_base& x, ios_base::iostate& st,
    void *& val) const;
```

behaves the same the first, except that the equivalent scan conversion specifier is `p`.

The sixth virtual protected member function:

```
virtual iter_type do_get(iter_type first, iter_type last,
    ios_base& x, ios_base::iostate& st,
    bool& val) const;
```

behaves the same as the first, except that it endeavors to match a complete, nonempty **boolean input field**. If successful it converts the boolean input field to a value of type `bool` and stores that value in `val`.

A boolean input field takes one of two forms. If `x.flags() & ios_base::boolalpha` is false, it is the same as an integer input field, except that the converted value must be either 0 (for false) or 1 (for true). Otherwise, the sequence must match either `fac.falsename()` (for false), or `fac.truename()` (for true).

num_get::get

```
iter_type get(iter_type first, iter_type last,
    ios_base& x, ios_base::iostate& st,
    long& val) const;
iter_type get(iter_type first, iter_type last,
    ios_base& x, ios_base::iostate& st,
    unsigned long& val) const;
iter_type get(iter_type first, iter_type last,
    ios_base& x, ios_base::iostate& st,
    double& val) const;
iter_type get(iter_type first, iter_type last,
    ios_base& x, ios_base::iostate& st,
    long double& val) const;
iter_type get(iter_type first, iter_type last,
    ios_base& x, ios_base::iostate& st,
    void *& val) const;
iter_type get(iter_type first, iter_type last,
    ios_base& x, ios_base::iostate& st,
    bool& val) const;
```

All member functions return `do_get(first, last, x, st, val)`.

num_get::iter_type

```
typedef InIt iter_type;
```

The type is a synonym for the template parameter `InIt`.

num_get::num_get

```
explicit num_get(size_t refs = 0);
```

The constructor initializes its base object with `locale::facet(refs)`.

num_put

```
template<class E, class OutIt = ostreambuf_iterator<E> >
    class num_put : public locale::facet {
public:
    typedef E char_type;
    typedef OutIt iter_type;
    explicit num_put(size_t refs = 0);
    iter_type put(iter_type next, ios_base& x,
        E fill, long val) const;
    iter_type put(iter_type next, ios_base& x,
        E fill, unsigned long val) const;
    iter_type put(iter_type next, ios_base& x,
        E fill, double val) const;
    iter_type put(iter_type next, ios_base& x,
        E fill, long double val) const;
    iter_type put(iter_type next, ios_base& x,
```



```

        E fill, const void *val) const;
    iter_type put(iter_type next, ios_base& x,
        E fill, bool val) const;
    static locale::id id;
protected:
    ~num_put();
    virtual iter_type do_put(iter_type next, ios_base& x,
        E fill, long val) const;
    virtual iter_type do_put(iter_type next, ios_base& x,
        E fill, unsigned long val) const;
    virtual iter_type do_put(iter_type next, ios_base& x,
        E fill, double val) const;
    virtual iter_type do_put(iter_type next, ios_base& x,
        E fill, long double val) const;
    virtual iter_type do_put(iter_type next, ios_base& x,
        E fill, const void *val) const;
    virtual iter_type do_put(iter_type next, ios_base& x,
        E fill, bool val) const;
};

```

The template class describes an object that can serve as a locale facet (page 135), to control conversions of numeric values to sequences of type E.

As with any locale facet, the static object **id** has an initial stored value of zero. The first attempt to access its stored value stores a unique positive value in **id**.

num_put::char_type

typedef E **char_type**;

The type is a synonym for the template parameter E.

num_put::do_put

```

virtual iter_type do_put(iter_type next, ios_base& x,
    E fill, long val) const;
virtual iter_type do_put(iter_type next, ios_base& x,
    E fill, unsigned long val) const;
virtual iter_type do_put(iter_type next, ios_base& x,
    E fill, double val) const;
virtual iter_type do_put(iter_type next, ios_base& x,
    E fill, long double val) const;
virtual iter_type do_put(iter_type next, ios_base& x,
    E fill, const void *val) const;
virtual iter_type do_put(iter_type next, ios_base& x,
    E fill, bool val) const;

```

The first virtual protected member function generates sequential elements beginning at next to produce an **integer output field** from the value of val. The function returns an iterator designating the next place to insert an element beyond the generated integer output field.

The integer output field is generated by the same rules used by the print functions (page 32) for generating a series of *char* elements to a file. (Each such *char* element is assumed to map to an equivalent element of type E by a simple, one-to-one, mapping.) Where a print function pads a field with either spaces or the digit 0, however, **do_put** instead uses **fill**. The equivalent print conversion specification (page 32) is determined as follows:

- If **x.flags() & ios_base::basefield == ios_base::oct**, the conversion specification is **lo**.
- If **x.flags() & ios_base::basefield == ios_base::hex**, the conversion specification is **lx**.

- Otherwise, the conversion specification is `ld`.

If `x.width()` is nonzero, a field width of this value is prepended. The function then calls `x.width(0)` to reset the field width to zero.

Padding occurs only if the minimum number of elements `N` required to specify the output field is less than `x.width()`. Such padding consists of a sequence of `N - width()` copies of `fill`. Padding then occurs as follows:

- If `x.flags() & ios_base::adjustfield == ios_base::left`, the flag `-` is prepended. (Padding occurs after the generated text.)
- If `x.flags() & ios_base::adjustfield == ios_base::internal`, the flag `0` is prepended. (For a numeric output field, padding occurs where the print functions pad with `0`.)
- Otherwise, no additional flag is prepended. (Padding occurs before the generated sequence.)

Finally:

- If `x.flags() & ios_base::showpos` is nonzero, the flag `+` is prepended to the conversion specification.
- If `x.flags() & ios_base::showbase` is nonzero, the flag `#` is prepended to the conversion specification.

The format of an integer output field is further determined by the locale facet (page 135) `fac` returned by the call `use_facet<num_punct>(x.getloc())`. Specifically:

- `fac.grouping()` determines how digits are grouped to the left of any decimal point
- `fac.thousands_sep()` determines the sequence that separates groups of digits to the left of any decimal point

If no grouping constraints are imposed by `fac.grouping()` (its first element has the value `CHAR_MAX`) then no instances of `fac.thousands_sep()` are generated in the output field. Otherwise, separators are inserted after the print conversion occurs.

The second virtual protected member function:

```
virtual iter_type do_put(iter_type next, ios_base& x,
    E fill, unsigned long val) const;
```

behaves the same as the first, except that it replaces a conversion specification of `ld` with `lu`.

The third virtual protected member function:

```
virtual iter_type do_put(iter_type next, ios_base& x,
    E fill, double val) const;
```

behaves the same as the first, except that it produces a **floating-point output field** from the value of `val`. `fac.decimal_point()` determines the sequence that separates the integer digits from the fraction digits. The equivalent print conversion specification is determined as follows:

- If `x.flags() & ios_base::floatfield == ios_base::fixed`, the conversion specification is `lf`.
- If `x.flags() & ios_base::floatfield == ios_base::scientific`, the conversion specification is `le`. If `x.flags() & ios_base::uppercase` is nonzero, `e` is replaced with `E`.

- Otherwise, the conversion specification is `lg`. If `x.flags() & ios_base::uppercase` is nonzero, `g` is replaced with `G`.

If `x.flags() & ios_base::fixed` is nonzero, or if `x.precision()` is greater than zero, a precision with the value `x.precision()` is prepended to the conversion specification. Any padding (page 154) behaves the same as for an integer output field. The padding character is `fill`. Finally:

- If `x.flags() & ios_base::showpos` is nonzero, the flag `+` is prepended to the conversion specification.
- If `x.flags() & ios_base::showpoint` is nonzero, the flag `#` is prepended to the conversion specification.

The fourth virtual protected member function:

```
virtual iter_type do_put(iter_type next, ios_base& x,
    E fill, long double val) const;
```

behaves the same the third, except that the qualifier `l` in the conversion specification is replaced with `L`.

The fifth virtual protected member function:

```
virtual iter_type do_put(iter_type next, ios_base& x,
    E fill, const void *val) const;
```

behaves the same the first, except that the conversion specification is `p`, plus any qualifier needed to specify padding.

The sixth virtual protected member function:

```
virtual iter_type do_put(iter_type next, ios_base& x,
    E fill, bool val) const;
```

behaves the same as the first, except that it generates a **boolean output field** from `val`.

A boolean output field takes one of two forms. If `x.flags() & ios_base::boolalpha` is false, the generated sequence is either `0` (for false) or `1` (for true). Otherwise, the generated sequence is either `fac.falsename()` (for false), or `fac.truename()` (for true).

num_put::put

```
iter_type put(iter_type next, ios_base& x,
    E fill, long val) const;
iter_type put(iter_type next, ios_base& x,
    E fill, unsigned long val) const;
iter_type put(iter_type iter_type next, ios_base& x,
    E fill, double val) const;
iter_type put(iter_type next, ios_base& x,
    E fill, long double val) const;
iter_type put(iter_type next, ios_base& x,
    E fill, const void *val) const;
iter_type put(iter_type next, ios_base& x,
    E fill, bool val) const;
```

All member functions return `do_put(next, x, fill, val)`.

num_put::iter_type

typedef InIt **iter_type**;

The type is a synonym for the template parameter OutIt.

num_put::num_put

explicit **num_put**(size_t refs = 0);

The constructor initializes its base object with `locale::facet(refs)`.

num_punct

char_type (page 156) · **decimal_point** (page 156) · **do_decimal_point** (page 156) · **do_falsename** (page 157) · **do_grouping** (page 157) · **do_truename** (page 157) · **do_thousands_sep** (page 157) · **falsename** (page 157) · **grouping** (page 157) · **num_punct** (page 157) · **string_type** (page 157) · **thousands_sep** (page 157) · **truename** (page 157)

```
template<class E, class num_punct : public locale::facet {
public:
    typedef E char_type;
    typedef basic_string<E> string_type;
    explicit num_punct(size_t refs = 0);
    E decimal_point() const;
    E thousands_sep() const;
    string grouping() const;
    string_type truename() const;
    string_type falsename() const;
    static locale::id id;
protected:
    ~num_punct();
    virtual E do_decimal_point() const;
    virtual E do_thousands_sep() const;
    virtual string do_grouping() const;
    virtual string_type do_truename() const;
    virtual string_type do_falsename() const;
};
```

The template class describes an object that can serve as a locale facet (page 135), to describe the sequences of type E used to represent the input fields matched by `num_get` (page 149) or the output fields generated by `num_get` (page 149).

As with any locale facet, the static object **id** has an initial stored value of zero. The first attempt to access its stored value stores a unique positive value in **id**.

num_punct::char_type

typedef E **char_type**;

The type is a synonym for the template parameter E.

num_punct::decimal_point

E **decimal_point**() const;

The member function returns `do_decimal_point()`.

num_punct::do_decimal_point

E **do_decimal_point**() const;

The protected virtual member function returns a locale-specific element to use as a decimal-point.

numpunct::do_falsename

`string_type do_falsename() const;`

The protected virtual member function returns a locale-specific sequence to use as a text representation of the value false.

numpunct::do_grouping

`string do_grouping() const;`

The protected virtual member function returns a locale-specific rule for determining how digits are grouped to the left of any decimal point. The encoding is the same as for `lconv::grouping`.

numpunct::do_thousands_sep

`E do_thousands_sep() const;`

The protected virtual member function returns a locale-specific element to use as a group separator to the left of any decimal point.

numpunct::do_truename

`string_type do_truename() const;`

The protected virtual member function returns a locale-specific sequence to use as a text representation of the value true.

numpunct::falsename

`string_type falsename() const;`

The member function returns `do_falsename()`.

numpunct::grouping

`string grouping() const;`

The member function returns `do_grouping()`.

numpunct::numpunct

`explicit numpunct(size_t refs = 0);`

The constructor initializes its base object with `locale::facet(refs)`.

numpunct::string_type

`typedef basic_string<E> string_type;`

The type describes a specialization of template class `basic_string` (page 197) whose objects can store copies of the punctuation sequences.

numpunct::thousands_sep

`E thousands_sep() const;`

The member function returns `do_thousands_sep()`.

numpunct::truename

`string_type falsename() const;`

The member function returns `do_truename()`.

numpunct_byname

```
template<class E>
class numpunct_byname : public numpunct<E> {
public:
    explicit numpunct_byname(const char *s,
                             size_t refs = 0);
protected:
    ~numpunct_byname();
};
```

The template class describes an object that can serve as a locale facet of type `numpunct<E>`. Its behavior is determined by the named (page 136) locale `s`. The constructor initializes its base object with `numpunct<E>(refs)`.

time_base

```
class time_base {
public:
    enum dateorder {no_order, dmy, mdy, ymd, ydm};
};
```

The class serves as a base class for facets of template class `time_get` (page 158). It defines just the enumerated type `dateorder` and several constants of this type. Each of the constants characterizes a different way to order the components of a date. The constants are:

- **no_order** specifies no particular order.
- **dmy** specifies the order day, month, then year, as in 2 December 1979.
- **mdy** specifies the order month, day, then year, as in December 2, 1979.
- **ymd** specifies the order year, month, then day, as in 1979/12/2.
- **ydm** specifies the order year, day, then month, as in 1979: 2 Dec.

time_get

```
template<class E, class InIt = istreambuf_iterator<E> >
class time_get : public locale::facet {
public:
    typedef E char_type;
    typedef InIt iter_type;
    explicit time_get(size_t refs = 0);
    dateorder date_order() const;
    iter_type get_time(iter_type first, iter_type last,
                       ios_base& x, ios_base::iostate& st, tm *pt) const;
    iter_type get_date(iter_type first, iter_type last,
                       ios_base& x, ios_base::iostate& st, tm *pt) const;
    iter_type get_weekday(iter_type first, iter_type last,
                           ios_base& x, ios_base::iostate& st, tm *pt) const;
    iter_type get_month(iter_type first, iter_type last,
                        ios_base& x, ios_base::iostate& st, tm *pt) const;
    iter_type get_year(iter_type first, iter_type last,
                       ios_base& x, ios_base::iostate& st, tm *pt) const;
    static locale::id id;
protected:
    ~time_get();
    virtual dateorder do_date_order() const;
    virtual iter_type
        do_get_time(iter_type first, iter_type last,
                    ios_base& x, ios_base::iostate& st, tm *pt) const;
    virtual iter_type
        do_get_date(iter_type first, iter_type last,
                    ios_base& x, ios_base::iostate& st, tm *pt) const;
    virtual iter_type
        do_get_weekday(iter_type first, iter_type last,
```

```

        ios_base& x, ios_base::iostate& st, tm *pt) const;
virtual iter_type
    do_get_month(iter_type first, iter_type last,
        ios_base& x, ios_base::iostate& st, tm *pt) const;
virtual iter_type
    do_get_year(iter_type first, iter_type last,
        ios_base& x, ios_base::iostate& st, tm *pt) const;
};

```

The template class describes an object that can serve as a locale facet (page 135), to control conversions of sequences of type `E` to time values.

As with any locale facet, the static object `id` has an initial stored value of zero. The first attempt to access its stored value stores a unique positive value in `id`.

time_get::char_type

```
typedef E char_type;
```

The type is a synonym for the template parameter `E`.

time_get::date_order

```
dateorder date_order() const;
```

The member function returns `date_order()`.

time_get::do_date_order

```
virtual dateorder do_date_order() const;
```

The virtual protected member function returns a value of type `time_base::dateorder`, which describes the order in which date components are matched by `do_get_date` (page 159). In this implementation (page 3), the value is `time_base::mdy`, corresponding to dates of the form December 2, 1979.

time_get::do_get_date

```
virtual iter_type
    do_get_date(iter_type first, iter_type last,
        ios_base& x, ios_base::iostate& st, tm *pt) const;
```

The virtual protected member function endeavors to match sequential elements beginning at `first` in the sequence `[first, last)` until it has recognized a complete, nonempty **date input field**. If successful, it converts this field to its equivalent value as the components `tm::tm_mon`, `tm::tm_day`, and `tm::tm_year`, and stores the results in `pt->tm_mon`, `pt->tm_day` and `pt->tm_year`, respectively. It returns an iterator designating the first element beyond the date input field. Otherwise, the function sets `ios_base::failbit` in `st`. It returns an iterator designating the first element beyond any prefix of a valid date input field. In either case, if the return value equals `last`, the function sets `ios_base::eofbit` in `st`.

In this implementation (page 3), the date input field has the form `MMM DD, YYYY`, where:

- `MMM` is matched by calling `get_month` (page 161), giving the month.
- `DD` is a sequence of decimal digits whose corresponding numeric value must be in the range `[1, 31]`, giving the day of the month.
- `YYYY` is matched by calling `get_year` (page 161), giving the year.
- The literal spaces and commas must match corresponding elements in the input sequence.

time_get::do_get_month

```
virtual iter_type  
    do_get_month(iter_type first, iter_type last,  
        ios_base& x, ios_base::iostate& st, tm *pt) const;
```

The virtual protected member function endeavors to match sequential elements beginning at `first` in the sequence `[first, last)` until it has recognized a complete, nonempty **month input field**. If successful, it converts this field to its equivalent value as the component `tm::tm_mon`, and stores the result in `pt->tm_mon`. It returns an iterator designating the first element beyond the month input field. Otherwise, the function sets `ios_base::failbit` in `st`. It returns an iterator designating the first element beyond any prefix of a valid month input field. In either case, if the return value equals `last`, the function sets `ios_base::eofbit` in `st`.

The month input field is a sequence that matches the longest of a set of locale-specific sequences, such as: Jan, January, Feb, February, etc. The converted value is the number of months since January.

time_get::do_get_time

```
virtual iter_type  
    do_get_time(iter_type first, iter_type last,  
        ios_base& x, ios_base::iostate& st, tm *pt) const;
```

The virtual protected member function endeavors to match sequential elements beginning at `first` in the sequence `[first, last)` until it has recognized a complete, nonempty **time input field**. If successful, it converts this field to its equivalent value as the components `tm::tm_hour`, `tm::tm_min`, and `tm::tm_sec`, and stores the results in `pt->tm_hour`, `pt->tm_min` and `pt->tm_sec`, respectively. It returns an iterator designating the first element beyond the time input field. Otherwise, the function sets `ios_base::failbit` in `st`. It returns an iterator designating the first element beyond any prefix of a valid time input field. In either case, if the return value equals `last`, the function sets `ios_base::eofbit` in `st`.

In this implementation (page 3), the time input field has the form `HH:MM:SS`, where:

- `HH` is a sequence of decimal digits whose corresponding numeric value must be in the range `[0, 24)`, giving the hour of the day.
- `MM` is a sequence of decimal digits whose corresponding numeric value must be in the range `[0, 60)`, giving the minutes past the hour.
- `SS` is a sequence of decimal digits whose corresponding numeric value must be in the range `[0, 60)`, giving the seconds past the minute.
- The literal colons must match corresponding elements in the input sequence.

time_get::do_get_weekday

```
virtual iter_type  
    do_get_weekday(iter_type first, iter_type last,  
        ios_base& x, ios_base::iostate& st, tm *pt) const;
```

The virtual protected member function endeavors to match sequential elements beginning at `first` in the sequence `[first, last)` until it has recognized a complete, nonempty **weekday input field**. If successful, it converts this field to its equivalent value as the component `tm::tm_wday`, and stores the result in `pt->tm_wday`. It returns an iterator designating the first element beyond the weekday input field. Otherwise, the function sets `ios_base::failbit` in `st`. It

returns an iterator designating the first element beyond any prefix of a valid weekday input field. In either case, if the return value equals last, the function sets `ios_base::eofbit` in `st`.

The weekday input field is a sequence that matches the longest of a set of locale-specific sequences, such as: Sun, Sunday, Mon, Monday, etc. The converted value is the number of days since Sunday.

time_get::do_get_year

```
virtual iter_type
do_get_year(iter_type first, iter_type last,
ios_base& x, ios_base::iostate& st, tm *pt) const;
```

The virtual protected member function endeavors to match sequential elements beginning at `first` in the sequence `[first, last)` until it has recognized a complete, nonempty **year input field**. If successful, it converts this field to its equivalent value as the component `tm::tm_year`, and stores the result in `pt->tm_year`. It returns an iterator designating the first element beyond the year input field. Otherwise, the function sets `ios_base::failbit` in `st`. It returns an iterator designating the first element beyond any prefix of a valid year input field. In either case, if the return value equals last, the function sets `ios_base::eofbit` in `st`.

The year input field is a sequence of decimal digits whose corresponding numeric value must be in the range `[1900, 2036)`. The stored value is this value minus 1900. In this implementation (page 3), a numeric value in the range `[0, 136)` is also permissible. It is stored unchanged.

time_get::get_date

```
iter_type get_date(iter_type first, iter_type last,
ios_base& x, ios_base::iostate& st, tm *pt) const;
```

The member function returns `do_get_date(first, last, x, st, pt)`.

time_get::get_month

```
iter_type get_month(iter_type first, iter_type last,
ios_base& x, ios_base::iostate& st, tm *pt) const;
```

The member function returns `do_get_month(first, last, x, st, pt)`.

time_get::get_time

```
iter_type get_time(iter_type first, iter_type last,
ios_base& x, ios_base::iostate& st, tm *pt) const;
```

The member function returns `do_get_time(first, last, x, st, pt)`.

time_get::get_weekday

```
iter_type get_weekday(iter_type first, iter_type last,
ios_base& x, ios_base::iostate& st, tm *pt) const;
```

The member function returns `do_get_weekday(first, last, x, st, pt)`.

time_get::get_year

```
iter_type get_year(iter_type first, iter_type last,
ios_base& x, ios_base::iostate& st, tm *pt) const;
```

The member function returns `do_get_year(first, last, x, st, pt)`.

time_get::iter_type

typedef InIt **iter_type**;

The type is a synonym for the template parameter InIt.

time_get::time_get

explicit **time_get**(size_t refs = 0);

The constructor initializes its base object with `locale::facet(refs)`.

time_get_byname

```
template<class E, class InIt>
    class time_get_byname : public time_get<E, InIt> {
public:
    explicit time_get_byname(const char *s,
                             size_t refs = 0);
protected:
    ~time_get_byname();
};
```

The template class describes an object that can serve as a locale facet (page 135) of type `time_get<E, InIt>`. Its behavior is determined by the named (page 136) locale `s`. The constructor initializes its base object with `time_get<E, InIt>(refs)`.

time_put

```
template<class E, class OutIt = ostreambuf_iterator<E> >
    class time_put : public locale::facet {
public:
    typedef E char_type;
    typedef OutIt iter_type;
    explicit time_put(size_t refs = 0);
    iter_type put(iter_type next, ios_base& x,
                  char_type fill, const tm *pt, char fmt, char mod = 0) const;
    iter_type put(iter_type next, ios_base& x,
                  char_type fill, const tm *pt, const E *first, const E *last) const;
    static locale::id id;
protected:
    ~time_put();
    virtual iter_type do_put(iter_type next, ios_base& x,
                              char_type fill, const tm *pt, char fmt, char mod = 0) const;
};
```

The template class describes an object that can serve as a locale facet (page 135), to control conversions of time values to sequences of type `E`.

As with any locale facet, the static object `id` has an initial stored value of zero. The first attempt to access its stored value stores a unique positive value in `id`.

time_put::char_type

typedef E **char_type**;

The type is a synonym for the template parameter `E`.

time_put::do_put

```
virtual iter_type do_put(iter_type next, ios_base& x,
                          char_type fill, const tm *pt, char fmt, char mod = 0) const;
```

The virtual protected member function generates sequential elements beginning at next from time values stored in the object *pt, of type tm. The function returns an iterator designating the next place to insert an element beyond the generated output.

The output is generated by the same rules used by strftime, with a last argument of pt, for generating a series of *char* elements into an array. (Each such *char* element is assumed to map to an equivalent element of type E by a simple, one-to-one, mapping.) If mod equals zero, the effective format is "%F", where F equals fmt. Otherwise, the effective format is "%MF", where M equals mod.

The parameter fill is not used.

time_put::put

```
iter_type put(iter_type next, ios_base& x,
              char_type fill, const tm *pt, char fmt, char mod = 0) const;
iter_type put(iter_type next, ios_base& x,
              char_type fill, const tm *pt, const E *first, const E *last) const;
```

The first member function returns do_put(next, x, fill, pt, fmt, mod). The second member function copies to *next++ any element in the interval [first, last) other than a percent (%). For a percent followed by a character C in the interval [first, last), the function instead evaluates next = do_put(next, x, fill, pt, C, 0) and skips past C. If, however, C is a qualifier character from the set EQQ#, followed by a character C2 in the interval [first, last), the function instead evaluates next = do_put(next, x, fill, pt, C2, C) and skips past C2.

time_put::iter_type

```
typedef InIt iter_type;
```

The type is a synonym for the template parameter OutIt.

time_put::time_put

```
explicit time_put(size_t refs = 0);
```

The constructor initializes its base object with locale::facet(refs).

time_put_byname

```
template<class E, class OutIt>
class time_put_byname : public time_put<E, OutIt> {
public:
    explicit time_put_byname(const char *s,
                             size_t refs = 0);
protected:
    ~time_put_byname();
};
```

The template class describes an object that can serve as a locale facet of type time_put<E, OutIt>. Its behavior is determined by the named (page 136) locale s. The constructor initializes its base object with time_put<E, OutIt>(refs).

tolower

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

The template function returns use_facet< ctype<E> >(loc). tolower(c).

toupper

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

The template function returns `use_facet< ctype<E> >(loc). toupper(c)`.

use_facet

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

The template function returns a reference to the locale facet of class Facet listed within the locale object (page 135) loc. If no such object is listed, the function throws an object of class bad_cast (page 224).

<new>

```
namespace std {
typedef void (*new_handler)();
class bad_alloc;
class nothrow_t;
extern const nothrow_t nothrow;

    // FUNCTIONS
new_handler set_new_handler(new_handler ph) throw();
};

    // OPERATORS -- NOT IN NAMESPACE std
void operator delete(void *p) throw();
void operator delete(void *, void *) throw();
void operator delete(void *p,
    const std::nothrow_t&) throw();
void operator delete[](void *p) throw();
void operator delete[](void *, void *) throw();
void operator delete[](void *p,
    const std::nothrow_t&) throw();
void *operator new(std::size_t n)
    throw(std::bad_alloc);
void *operator new(std::size_t n,
    const std::nothrow_t&) throw();
void *operator new(std::size_t n, void *p) throw();
void *operator new[](std::size_t n)
    throw(std::bad_alloc);
void *operator new[](std::size_t n,
    const std::nothrow_t&) throw();
void *operator new[](std::size_t n, void *p) throw();
```

Include the standard header `<new>` to define several types and functions that control allocation and freeing of storage under program control.

Some of the functions declared in this header are **replaceable**. The implementation supplies a default version, whose behavior is described in this document. A program can, however, define a function with the same signature to replace the default version at link time. The replacement version must satisfy the requirements described in this document.

bad_alloc

```
class bad_alloc : public exception {
};
```

The class describes an exception thrown to indicate that an allocation request did not succeed. The value returned by `what()` is an implementation-defined C string. None of the member functions throw any exceptions.

new_handler

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

The type points to a function suitable for use as a new handler (page 166).

nothrow

```
extern const nothrow_t nothrow;
```

The object is used as a function argument to match the parameter type `nothrow_t` (page 165).

nothrow_t

```
class nothrow_t {};
```

The class is used as a function parameter to `operator new` to indicate that the function should return a null pointer to report an allocation failure, rather than throw an exception.

operator delete

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

The first function is called by a **delete expression** to render the value of `p` invalid. The program can define a function with this function signature that replaces (page 164) the default version defined by the Standard C++ library. The required behavior is to accept a value of `p` that is null or that was returned by an earlier call to `operator new(size_t)`.

The default behavior for a null value of `p` is to do nothing. Any other value of `p` must be a value returned earlier by a call as described above. The default behavior for such a non-null value of `p` is to reclaim storage allocated by the earlier call. It is unspecified under what conditions part or all of such reclaimed storage is allocated by a subsequent call to `operator new(size_t)`, or to any of `calloc(size_t)`, `malloc(size_t)`, or `realloc(void*, size_t)`.

The second function is called by a **placement delete expression** corresponding to a new expression of the form `new(std::size_t)`. It does nothing.

The third function is called by a placement delete expression corresponding to a new expression of the form `new(std::size_t, const std::nothrow_t&)`. It calls `delete(p)`.

operator delete[]

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

The first function is called by a **delete[] expression** to render the value of *p* invalid. The program can define a function with this function signature that replaces (page 164) the default version defined by the Standard C++ library.

The required behavior is to accept a value of *p* that is null or that was returned by an earlier call to `operator new[](size_t)`.

The default behavior for a null value of *p* is to do nothing. Any other value of *ptr* must be a value returned earlier by a call as described above. The default behavior for such a non-null value of *p* is to reclaim storage allocated by the earlier call. It is unspecified under what conditions part or all of such reclaimed storage is allocated by a subsequent call to `operator new(size_t)`, or to any of `calloc(size_t)`, `malloc(size_t)`, or `realloc(void*, size_t)`.

The second function is called by a **placement delete[] expression** corresponding to a `new[]` expression of the form `new[](std::size_t)`. It does nothing.

The third function is called by a placement delete expression corresponding to a `new[]` expression of the form `new[](std::size_t, const std::nothrow_t&)`. It calls `delete[](p)`.

operator new

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

The first function is called by a **new expression** to allocate *n* bytes of storage suitably aligned to represent any object of that size. The program can define a function with this function signature that replaces (page 164) the default version defined by the Standard C++ library.

The required behavior is to return a non-null pointer only if storage can be allocated as requested. Each such allocation yields a pointer to storage disjoint from any other allocated storage. The order and contiguity of storage allocated by successive calls is unspecified. The initial stored value is unspecified. The returned pointer points to the start (lowest byte address) of the allocated storage. If *n* is zero, the value returned does not compare equal to any other value returned by the function.

The default behavior is to execute a loop. Within the loop, the function first attempts to allocate the requested storage. Whether the attempt involves a call to `malloc(size_t)` is unspecified. If the attempt is successful, the function returns a pointer to the allocated storage. Otherwise, the function calls the designated new handler. If the called function returns, the loop repeats. The loop terminates when an attempt to allocate the requested storage is successful or when a called function does not return.

The required behavior of a **new handler** is to perform one of the following operations:

- make more storage available for allocation and then return
- call either `abort()` or `exit(int)`
- throw an object of type `bad_alloc`

The default behavior of a new handler is to throw an object of type `bad_alloc`. A null pointer designates the default new handler.

The order and contiguity of storage allocated by successive calls to `operator new(size_t)` is unspecified, as are the initial values stored there.

The second function:

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

is called by a placement new expression to allocate `n` bytes of storage suitably aligned to represent any object of that size. The program can define a function with this function signature that replaces (page 164) the default version defined by the Standard C++ library.

The default behavior is to return `operator new(n)` if that function succeeds. Otherwise, it returns a null pointer.

The third function:

```
void *operator new(std::size_t n, void *p) throw();
```

is called by a **placement new expression**, of the form `new (args) T`. Here, `args` consists of a single object pointer. The function returns `p`.

operator new[]

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

The first function is called by a **new[] expression** to allocate `n` bytes of storage suitably aligned to represent any array object of that size or smaller. The program can define a function with this function signature that replaces (page 164) the default version defined by the Standard C++ library.

The required behavior is the same as for `operator new(size_t)`. The default behavior is to return `operator new(n)`.

The second function is called by a placement `new[]` expression to allocate `n` bytes of storage suitably aligned to represent any array object of that size. The program can define a function with this function signature that replaces (page 164) the default version defined by the Standard C++ library.

The default behavior is to return `operator new(n)` if that function succeeds. Otherwise, it returns a null pointer.

The third function is called by a **placement new[] expression**, of the form `new (args) T[N]`. Here, `args` consists of a single object pointer. The function returns `p`.

set_new_handler

```
new_handler set_new_handler(new_handler ph) throw();
```

The function stores `ph` in a static new handler (page 166) pointer that it maintains, then returns the value previously stored in the pointer. The new handler is used by `operator new(size_t)`.

<ostream>

```
namespace std {
template<class E, class T = char_traits<E> >
    class basic_ostream;
typedef basic_ostream<char, char_traits<char> >
    ostream;
typedef basic_ostream<wchar_t, char_traits<wchar_t> >
    wostream;

    // INSERTERS
template<class E, class T>
    basic_ostream<E, T>&
        operator<<(basic_ostream<E, T>& os,
            const E *s);
template<class E, class T>
    basic_ostream<E, T>&
        operator<<(basic_ostream<E, T>& os,
            E c);
template<class E, class T>
    basic_ostream<E, T>&
        operator<<(basic_ostream<E, T>& os,
            const char *s);
template<class E, class T>
    basic_ostream<E, T>&
        operator<<(basic_ostream<E, T>& os,
            char c);
template<class T>
    basic_ostream<char, T>&
        operator<<(basic_ostream<char, T>& os,
            const char *s);
template<class T>
    basic_ostream<char, T>&
        operator<<(basic_ostream<char, T>& os,
            char c);
template<class T>
    basic_ostream<char, T>&
        operator<<(basic_ostream<char, T>& os,
            const signed char *s);
template<class T>
    basic_ostream<char, T>&
        operator<<(basic_ostream<char, T>& os,
            signed char c);
template<class T>
    basic_ostream<char, T>&
        operator<<(basic_ostream<char, T>& os,
            const unsigned char *s);
template<class T>
    basic_ostream<char, T>&
        operator<<(basic_ostream<char, T>& os,
            unsigned char c);

    // MANIPULATORS
template<class E, T>
    basic_ostream<E, T>&
        endl(basic_ostream<E, T>& os);
template<class E, T>
    basic_ostream<E, T>&
        ends(basic_ostream<E, T>& os);
```



```
template class<E, T>
    basic_ostream<E, T>&
        flush(basic_ostream<E, T>& os);
};
```

Include the iostreams (page 7) standard header **<ostream>** to define template class `basic_ostream` (page 169), which mediates insertions for the iostreams. The header also defines several related manipulators (page 87). (This header is typically included for you by another of the iostreams headers. You seldom have occasion to include it directly.)

basic_ostream

basic_ostream (page 170) · **flush** (page 170) · **operator<<** (page 170) · **opfx** (page 172) · **osfx** (page 172) · **put** (page 172) · **seekp** (page 172) · **sentry** (page 172) · **tellp** (page 172) · **write** (page 172)

```
template <class E, class T = char_traits<E> >
    class basic_ostream
        : virtual public basic_ios<E, T> {
public:
    typedef typename basic_ios<E, T>::char_type char_type;
    typedef typename basic_ios<E, T>::traits_type traits_type;
    typedef typename basic_ios<E, T>::int_type int_type;
    typedef typename basic_ios<E, T>::pos_type pos_type;
    typedef typename basic_ios<E, T>::off_type off_type;
    explicit basic_ostream(basic_streambuf<E, T> *sb);
    class sentry;
    virtual ~ostream();
    bool opfx();
    void osfx();
    basic_ostream& operator<<(
        basic_ostream& (*pf)(basic_ostream&));
    basic_ostream& operator<<(
        ios_base& (*pf)(ios_base&));
    basic_ostream& operator<<(
        basic_ios<E, T>& (*pf)(basic_ios<E, T>&));
    basic_ostream& operator<<(
        basic_streambuf<E, T> *sb);
    basic_ostream& operator<<(bool n);
    basic_ostream& operator<<(short n);
    basic_ostream& operator<<(unsigned short n);
    basic_ostream& operator<<(int n);
    basic_ostream& operator<<(unsigned int n);
    basic_ostream& operator<<(long n);
    basic_ostream& operator<<(unsigned long n);
    basic_ostream& operator<<(float n);
    basic_ostream& operator<<(double n);
    basic_ostream& operator<<(long double n);
    basic_ostream& operator<<(const void *n);
    basic_ostream& put(char_type c);
    basic_ostream& write(char_type *s, streamsize n);
    basic_ostream& flush();
    pos_type tellp();
    basic_ostream& seekp(pos_type pos);
    basic_ostream& seekp(off_type off,
        ios_base::seek_dir way);
};
```

The template class describes an object that controls insertion of elements and encoded objects into a stream buffer (page 187) with elements of type `E`, also known as `char_type` (page 89), whose character traits (page 211) are determined by the class `T`, also known as `traits_type` (page 91).

Most of the member functions that overload operator<< (page 170) are **formatted output functions**. They follow the pattern:

```
iostate state = goodbit;
const sentry ok(*this);
if (ok)
    {try
      {<convert and insert elements
       accumulate flags in state>}
     catch (...)
      {try
       {setstate(badbit); }
      }
    }
catch (...)
    {}
    if ((exceptions() & badbit) != 0)
        throw; }}
width(0);    // except for operator<<(E)
setstate(state);
return (*this);
```

Two other member functions are **unformatted output functions**. They follow the pattern:

```
iostate state = goodbit;
const sentry ok(*this);
if (!ok)
    state |= badbit;
else
    {try
      {<obtain and insert elements
       accumulate flags in state>}
     catch (...)
      {try
       {setstate(badbit); }
      }
    }
catch (...)
    {}
    if ((exceptions() & badbit) != 0)
        throw; }}
setstate(state);
return (*this);
```

Both groups of functions call setstate(badbit) if they encounter a failure while inserting elements.

An object of class basic_istream<E, T> stores only a virtual public base object of class **basic_ios<E, T>** (page 88)

basic_ostream::basic_ostream

```
explicit basic_ostream(basic_streambuf<E, T> *sb);
```

The constructor initializes the base class by calling init(sb).

basic_ostream::flush

```
basic_ostream& flush();
```

If rdbuf() is not a null pointer, the function calls rdbuf()->pubsync(). If that returns -1, the function calls setstate(badbit). It returns *this.

basic_ostream::operator<<

```
basic_ostream& operator<<(
    basic_ostream& (*pf)(basic_ostream&));
basic_ostream& operator<<(
    ios_base& (*pf)(ios_base&));
basic_ostream& operator<<(
```

```

        basic_ios<E, T>& (*pf)(basic_ios<E, T>&));
basic_ostream& operator<<(
    basic_streambuf<E, T> *sb);
basic_ostream& operator<<(bool n);
basic_ostream& operator<<(short n);
basic_ostream& operator<<(unsigned short n);
basic_ostream& operator<<(int n);
basic_ostream& operator<<(unsigned int n);
basic_ostream& operator<<(long n);
basic_ostream& operator<<(unsigned long n);
basic_ostream& operator<<(float n);
basic_ostream& operator<<(double n);
basic_ostream& operator<<(long double n);
basic_ostream& operator<<(const void *n);

```

The first member function ensures that an expression of the form `ostr << endl` calls `endl(ostr)`, then returns `*this`. The second and third functions ensure that other manipulators (page 87), such as `hex` (page 93) behave similarly. The remaining functions are all formatted output functions (page 170).

The function:

```

basic_ostream& operator<<(
    basic_streambuf<E, T> *sb);

```

extracts elements from `sb`, if `sb` is not a null pointer, and inserts them. Extraction stops on end-of-file, or if an extraction throws an exception (which is rethrown). It also stops, without extracting the element in question, if an insertion fails. If the function inserts no elements, or if an extraction throws an exception, the function calls `setstate(failbit)`. In any case, the function returns `*this`.

The function:

```

basic_ostream& operator<<(bool n);

```

converts `n` to a boolean field and inserts it by calling `use_facet<num_put<E, OutIt>(getloc()). put(OutIt(rdbuf()), *this, getloc(), n)`. Here, `OutIt` is defined as `ostreambuf_iterator<E, T>`. The function returns `*this`.

The functions:

```

basic_ostream& operator<<(short n);
basic_ostream& operator<<(unsigned short n);
basic_ostream& operator<<(int n);
basic_ostream& operator<<(unsigned int n);
basic_ostream& operator<<(long n);
basic_ostream& operator<<(unsigned long n);
basic_ostream& operator<<(const void *n);

```

each convert `n` to a numeric field and insert it by calling `use_facet<num_put<E, OutIt>(getloc()). put(OutIt(rdbuf()), *this, getloc(), n)`. Here, `OutIt` is defined as `ostreambuf_iterator<E, T>`.

The function returns `*this`.

The functions:

```

basic_ostream& operator<<(float n);
basic_ostream& operator<<(double n);
basic_ostream& operator<<(long double n);

```

each convert *n* to a numeric field and insert it by calling `use_facet<num_put<E, OutIt>(getloc()). put(OutIt(rdbuf()), *this, getloc(), n)`. Here, `OutIt` is defined as `ostreambuf_iterator<E, T>`. The function returns `*this`.

basic_ostream::opfx

```
bool opfx();
```

If `good()` is true, and `tie()` is not a null pointer, the member function calls `tie->flush()`. It returns `good()`.

You should not call `opfx` directly. It is called as needed by an object of class `sentry` (page 172).

basic_ostream::osfx

```
void osfx();
```

If `flags()` & `unitbuf` is nonzero, the member function calls `flush()`. You should not call `osfx` directly. It is called as needed by an object of class `sentry`.

basic_ostream::put

```
basic_ostream& put(char_type c);
```

The unformatted output function (page 170) inserts the element *c*. It returns `*this`.

basic_ostream::seekp

```
basic_ostream& seekp(pos_type pos);  
basic_ostream& seekp(off_type off,  
    ios_base::seek_dir way);
```

If `fail()` is false, the first member function calls `rdbuf()-> pubseekpos(pos)`. If `fail()` is false, the second function calls `rdbuf()-> pubseekoff(off, way)`. Both functions return `*this`.

basic_ostream::sentry

```
class sentry {  
public:  
    explicit sentry(basic_ostream<E, T>& os);  
    operator bool() const;  
private:  
    sentry(const sentry&); // not defined  
    sentry& operator=(const sentry&); // not defined  
};
```

The nested class describes an object whose declaration structures the formatted output functions (page 170) and the unformatted output functions (page 170). The constructor effectively calls `os.opfx()` and stores the return value. `operator bool()` delivers this return value. The destructor effectively calls `os.osfx()`, but only if `uncaught_exception()` returns false.

basic_ostream::tellp

```
pos_type tellp();
```

If `fail()` is false, the member function returns `rdbuf()-> pubseekoff(0, cur, in)`. Otherwise, it returns `pos_type(-1)`.

basic_ostream::write

```
basic_ostream& write(const char_type *s, streamsize n);
```

The unformatted output function (page 170) inserts the sequence of *n* elements beginning at *s*.

endl

```
template class<E, T>
    basic_ostream<E, T> endl(basic_ostream<E, T>& os);
```

The manipulator calls `os.put(os.widen('\n'))`, then calls `os.flush()`. It returns `os`.

ends

```
template class<E, T>
    basic_ostream<E, T> ends(basic_ostream<E, T>& os);
```

The manipulator calls `os.put(E('\0'))`. It returns `os`.

flush

```
template class<E, T>
    basic_ostream<E, T> flush(basic_ostream<E, T>& os);
```

The manipulator calls `os.flush()`. It returns `os`.

operator<<

```
template<class E, class T>
    basic_ostream<E, T>&
        operator<<(basic_ostream<E, T>& os,
            const E *s);
template<class E, class T>
    basic_ostream<E, T>&
        operator<<(basic_ostream<E, T>& os,
            E c);
template<class E, class T>
    basic_ostream<E, T>&
        operator<<(basic_ostream<E, T>& os,
            const char *s);
template<class E, class T>
    basic_ostream<E, T>&
        operator<<(basic_ostream<E, T>& os,
            char c);
template<class T>
    basic_ostream<char, T>&
        operator<<(basic_ostream<char, T>& os,
            const char *s);
template<class T>
    basic_ostream<char, T>&
        operator<<(basic_ostream<char, T>& os,
            char c);
template<class T>
    basic_ostream<char, T>&
        operator<<(basic_ostream<char, T>& os,
            const signed char *s);
template<class T>
    basic_ostream<char, T>&
        operator<<(basic_ostream<char, T>& os,
            signed char c);
template<class T>
    basic_ostream<char, T>&
        operator<<(basic_ostream<char, T>& os,
            const unsigned char *s);
```

```
template<class T>
    basic_ostream<char, T>&
        operator<<(basic_ostream<char, T>& os,
            unsigned char c);
```

The template function:

```
template<class E, class T>
    basic_ostream<E, T>&
        operator<<(basic_ostream<E, T>& os,
            const E *s);
```

is a formatted output functions (page 170) that determines the length $n = \text{traits_type::length}(s)$ of the sequence beginning at s , and inserts the sequence. If $n < \text{os.width}()$, then the function also inserts a repetition of $\text{os.width}() - n$ fill characters (page 89). The repetition precedes the sequence if $(\text{os.flags}() \& \text{adjustfield}) \neq \text{left}$. Otherwise, the repetition follows the sequence. The function returns os .

The template function:

```
template<class E, class T>
    basic_ostream<E, T>&
        operator<<(basic_ostream<E, T>& os,
            E c);
```

inserts the element c . If $1 < \text{os.width}()$, then the function also inserts a repetition of $\text{os.width}() - 1$ fill characters (page 89). The repetition precedes the sequence if $(\text{os.flags}() \& \text{adjustfield}) \neq \text{left}$. Otherwise, the repetition follows the sequence. It returns os .

The template function:

```
template<class E, class T>
    basic_ostream<E, T>&
        operator<<(basic_ostream<E, T>& os,
            const char *s);
```

behaves the same as:

```
template<class E, class T>
    basic_ostream<E, T>&
        operator<<(basic_ostream<E, T>& os,
            const E *s);
```

except that each element c of the sequence beginning at s is converted to an object of type E by calling $\text{os.put}(\text{os.widen}(c))$.

The template function:

```
template<class E, class T>
    basic_ostream<E, T>&
        operator<<(basic_ostream<E, T>& os,
            char c);
```

behaves the same as:

```
template<class E, class T>
    basic_ostream<E, T>&
        operator<<(basic_ostream<E, T>& os,
            E c);
```

except that c is converted to an object of type E by calling $\text{os.put}(\text{os.widen}(c))$.

The template function:

```
template<class T>
    basic_ostream<char, T>&
        operator<<(basic_ostream<char, T>& os,
                    const char *s);
```

behaves the same as:

```
template<class E, class T>
    basic_ostream<E, T>&
        operator<<(basic_ostream<E, T>& os,
                    const E *s);
```

(It does not have to widen the elements before inserting them.)

The template function:

```
template<class T>
    basic_ostream<char, T>&
        operator<<(basic_ostream<char, T>& os,
                    char c);
```

behaves the same as:

```
template<class E, class T>
    basic_ostream<E, T>&
        operator<<(basic_ostream<E, T>& os,
                    E c);
```

(It does not have to widen c before inserting it.)

The template function:

```
template<class T>
    basic_ostream<char, T>&
        operator<<(basic_ostream<char, T>& os,
                    const signed char *s);
```

returns os << (const char *)s.

The template function:

```
template<class T>
    basic_ostream<char, T>&
        operator<<(basic_ostream<char, T>& os,
                    signed char c);
```

returns os << (char)c.

The template function:

```
template<class T>
    basic_ostream<char, T>&
        operator<<(basic_ostream<char, T>& os,
                    const unsigned char *s);
```

returns os << (const char *)s.

The template function:

```
template<class T>
    basic_ostream<char, T>&
        operator<<(basic_ostream<char, T>& os,
                    unsigned char c);
```

returns os << (char)c.

ostream

```
typedef basic_ostream<char, char_traits<char> > ostream;
```

The type is a synonym for template class `basic_ostream` (page 169), specialized for elements of type `char` with default character traits (page 211).

wostream

```
typedef basic_ostream<wchar_t, char_traits<wchar_t> >  
wostream;
```

The type is a synonym for template class `basic_ostream` (page 169), specialized for elements of type `wchar_t` with default character traits (page 211).

<sstream>

```
namespace std {  
    template<class E,  
        class T = char_traits<E>,  
        class A = allocator<E> >  
        class basic_stringbuf;  
    typedef basic_stringbuf<char> stringbuf;  
    typedef basic_stringbuf<wchar_t> wstringbuf;  
    template<class E,  
        class T = char_traits<E>,  
        class A = allocator<E> >  
        class basic_istreamstream;  
    typedef basic_istreamstream<char> istreamstream;  
    typedef basic_istreamstream<wchar_t> wistreamstream;  
    template<class E,  
        class T = char_traits<E>,  
        class A = allocator<E> >  
        class basic_ostreamstream;  
    typedef basic_ostreamstream<char> ostreamstream;  
    typedef basic_ostreamstream<wchar_t> woostreamstream;  
    template<class E,  
        class T = char_traits<E>,  
        class A = allocator<E> >  
        class basic_stringstream;  
    typedef basic_stringstream<char> stringstream;  
    typedef basic_stringstream<wchar_t> wstringstream;  
};
```

Include the iostreams (page 7) standard header `<sstream>` to define several template classes that support iostreams operations on sequences stored in an allocated array object. Such sequences are easily converted to and from objects of template class `basic_string` (page 197).

basic_stringbuf

```
template <class E,  
    class T = char_traits<E>,  
    class A = allocator<E> >  
    class basic_stringbuf  
    : public basic_streambuf<E, T> {  
public:  
    typedef typename basic_streambuf<E, T>::char_type  
        char_type;  
    typedef typename basic_streambuf<E, T>::traits_type  
        traits_type;  
    typedef typename basic_streambuf<E, T>::int_type  
        int_type;  
    typedef typename basic_streambuf<E, T>::pos_type
```



```

        pos_type;
typedef typename basic_streambuf<E, T>::off_type
        off_type;
basic_stringbuf(ios_base::openmode mode =
        ios_base::in | ios_base::out);
basic_stringbuf(basic_string<E, T, A>& x,
        ios_base::openmode mode =
        ios_base::in | ios_base::out);
basic_string<E, T, A> str() const;
void str(basic_string<E, T, A>& x);
protected:
    virtual pos_type seekoff(off_type off,
        ios_base::seekdir way,
        ios_base::openmode mode =
        ios_base::in | ios_base::out);
    virtual pos_type seekpos(pos_type sp,
        ios_base::openmode mode =
        ios_base::in | ios_base::out);
    virtual int_type underflow();
    virtual int_type pbackfail(int_type c =
        traits_type::eof());
    virtual int_type overflow(int_type c =
        traits_type::eof());
};

```

The template class describes a **stream buffer** (page 187) that controls the transmission of elements of type E, whose character traits (page 211) are determined by the class T, to and from a sequence of elements stored in an array object. The object is allocated, extended, and freed as necessary to accommodate changes in the sequence.

An object of class `basic_stringbuf<E, T, A>` stores a copy of the `ios_base::openmode` argument from its constructor as its **stringbuf mode** mode:

- If mode & `ios_base::in` is nonzero, the input buffer (page 187) is accessible.
- If mode & `ios_base::out` is nonzero, the output buffer (page 187) is accessible.

basic_stringbuf::basic_stringbuf

```

basic_stringbuf(ios_base::openmode mode =
        ios_base::in | ios_base::out);
basic_stringbuf(basic_string<E, T, A>& x,
        ios_base::openmode mode =
        ios_base::in | ios_base::out);

```

The first constructor stores a null pointer in all the pointers controlling the input buffer (page 187) and the output buffer (page 187). It also stores mode as the stringbuf mode (page 177).

The second constructor allocates a copy of the sequence controlled by the string object x. If mode & `ios_base::in` is nonzero, it sets the input buffer to begin reading at the start of the sequence. If mode & `ios_base::out` is nonzero, it sets the output buffer to begin writing at the start of the sequence. It also stores mode as the stringbuf mode (page 177).

basic_stringbuf::char_type

```

typedef E char_type;

```

The type is a synonym for the template parameter E.

basic_stringbuf::int_type

```
typedef typename traits_type::int_type int_type;
```

The type is a synonym for `traits_type::int_type`.

basic_stringbuf::off_type

```
typedef typename traits_type::off_type off_type;
```

The type is a synonym for `traits_type::off_type`.

basic_stringbuf::overflow

```
virtual int_type overflow(int_type c =  
    traits_type::eof());
```

If `c` does not compare equal to `traits_type::eof()`, the protected virtual member function endeavors to insert the element `traits_type::to_char_type(c)` into the output buffer (page 187). It can do so in various ways:

- If a write position (page 188) is available, it can store the element into the write position and increment the next pointer for the output buffer.
- It can make a write position available by allocating new or additional storage for the output buffer. (Extending the output buffer this way also extends any associated input buffer (page 187).)

If the function cannot succeed, it returns `traits_type::eof()`. Otherwise, it returns `traits_type::not_eof(c)`.

basic_stringbuf::pbackfail

```
virtual int_type pbackfail(int_type c =  
    traits_type::eof());
```

The protected virtual member function endeavors to put back an element into the input buffer (page 187), then make it the current element (pointed to by the next pointer). If `c` compares equal to `traits_type::eof()`, the element to push back is effectively the one already in the stream before the current element. Otherwise, that element is replaced by `x = traits_type::to_char_type(c)`. The function can put back an element in various ways:

- If a putback position (page 188) is available, and the element stored there compares equal to `x`, it can simply decrement the next pointer for the input buffer.
- If a putback position is available, and if the stringbuf mode (page 177) permits the sequence to be altered (mode & `ios_base::out` is nonzero), it can store `x` into the putback position and decrement the next pointer for the input buffer.

If the function cannot succeed, it returns `traits_type::eof()`. Otherwise, it returns `traits_type::not_eof(c)`.

basic_stringbuf::pos_type

```
typedef typename traits_type::pos_type pos_type;
```

The type is a synonym for `traits_type::pos_type`.

basic_stringbuf::seekoff

```
virtual pos_type seekoff(off_type off,  
    ios_base::seekdir way,  
    ios_base::openmode mode =  
        ios_base::in | ios_base::out);
```

The protected virtual member function endeavors to alter the current positions for the controlled streams. For an object of class `basic_stringbuf<E, T, A>`, a stream position consists purely of a stream offset. Offset zero designates the first element of the controlled sequence.

The new position is determined as follows:

- If `way == ios_base::beg`, the new position is the beginning of the stream plus off.
- If `way == ios_base::cur`, the new position is the current stream position plus off.
- If `way == ios_base::end`, the new position is the end of the stream plus off.

If `mode & ios_base::in` is nonzero, the function alters the next position to read in the input buffer. If `mode & ios_base::out` is nonzero, the function alters the next position to write in the output buffer. For a stream to be affected, its buffer must exist. For a positioning operation to succeed, the resulting stream position must lie within the controlled sequence. If the function affects both stream positions, `way` must be `ios_base::beg` or `ios_base::end` and both streams are positioned at the same element. Otherwise (or if neither position is affected) the positioning operation fails.

If the function succeeds in altering the stream position(s), it returns the resultant stream position. Otherwise, it fails and returns an invalid stream position.

`basic_stringbuf::seekpos`

```
virtual pos_type seekpos(pos_type sp,  
    ios_base::openmode mode =  
        ios_base::in | ios_base::out);
```

The protected virtual member function endeavors to alter the current positions for the controlled streams. For an object of class `basic_stringbuf<E, T, A>`, a stream position consists purely of a stream offset. Offset zero designates the first element of the controlled sequence. The new position is determined by `sp`.

If `mode & ios_base::in` is nonzero, the function alters the next position to read in the input buffer. If `mode & ios_base::out` is nonzero, the function alters the next position to write in the output buffer. For a stream to be affected, its buffer must exist. For a positioning operation to succeed, the resulting stream position must lie within the controlled sequence. Otherwise (or if neither position is affected) the positioning operation fails.

If the function succeeds in altering the stream position(s), it returns the resultant stream position. Otherwise, it fails and returns an invalid stream position.

`basic_stringbuf::str`

```
basic_string<E, T, A> str() const;  
void str(basic_string<E, T, A>& x);
```

The first member function returns an object of class `basic_string<E, T, A>`, whose controlled sequence is a copy of the sequence controlled by `*this`. The sequence copied depends on the stored stringbuf mode (page 177) mode:

- If `mode & ios_base::out` is nonzero and an output buffer exists, the sequence is the entire output buffer (`epptr()` - `pbase()` elements beginning with `pbase()`).

- Otherwise, if `mode & ios_base::in` is nonzero and an input buffer exists, the sequence is the entire input buffer (`egptr() - eback()` elements beginning with `eback()`).
- Otherwise, the copied sequence is empty.

The second member function deallocates any sequence currently controlled by `*this`. It then allocates a copy of the sequence controlled by `x`. If `mode & ios_base::in` is nonzero, it sets the input buffer to begin reading at the beginning of the sequence. If `mode & ios_base::out` is nonzero, it sets the output buffer to begin writing at the beginning of the sequence.

basic_stringbuf::traits_type

```
typedef T traits_type;
```

The type is a synonym for the template parameter `T`.

basic_stringbuf::underflow

```
virtual int_type underflow();
```

The protected virtual member function endeavors to extract the current element `c` from the input buffer, then advance the current stream position, and return the element as `traits_type::to_int_type(c)`. It can do so in only one way: If a read position (page 188) is available, it takes `c` as the element stored in the read position and advances the next pointer for the input buffer.

If the function cannot succeed, it returns `traits_type::eof()`. Otherwise, it returns the current element in the input stream, converted as described above.

basic_istream

```
template <class E,
          class T = char_traits<E>,
          class A = allocator<E> >
class basic_istream
    : public basic_istream<E, T> {
public:
    explicit basic_istream(
        ios_base::openmode mode = ios_base::in);
    explicit basic_istream(
        const basic_string<E, T, A>& x,
        ios_base::openmode mode = ios_base::in);
    basic_stringbuf<E, T, A> *rdbuf() const;
    basic_string<E, T, A>& str();
    void str(const basic_string<E, T, A>& x);
};
```

The template class describes an object that controls extraction of elements and encoded objects from a stream buffer of class `basic_stringbuf<E, T, A>`, with elements of type `E`, whose character traits (page 211) are determined by the class `T`, and whose elements are allocated by an allocator of class `A`. The object stores an object of class `basic_stringbuf<E, T, A>`.

basic_istream::basic_istream

```
explicit basic_istream(
    ios_base::openmode mode = ios_base::in);
explicit basic_istream(
    const basic_string<E, T, A>& x,
    ios_base::openmode mode = ios_base::in);
```

The first constructor initializes the base class by calling `basic_istream(sb)`, where `sb` is the stored object of class `basic_stringbuf<E, T, A>`. It also initializes `sb` by calling `basic_stringbuf<E, T, A>(mode | ios_base::in)`.

The second constructor initializes the base class by calling `basic_istream(sb)`. It also initializes `sb` by calling `basic_stringbuf<E, T, A>(x, mode | ios_base::in)`.

basic_istream::rdbuf

```
basic_stringbuf<E, T, A> *rdbuf() const
```

The member function returns the address of the stored stream buffer, of type pointer to `basic_stringbuf<E, T, A>`.

basic_istream::str

```
basic_string<E, T, A> str() const;
void str(basic_string<E, T, A>& x);
```

The first member function returns `rdbuf()->str()`. The second member function calls `rdbuf()->str(x)`.

basic_ostringstream

```
template <class E,
          class T = char_traits<E>,
          class A = allocator<E> >
class basic_ostringstream
    : public basic_ostream<E, T> {
public:
    explicit basic_ostringstream(
        ios_base::openmode mode = ios_base::out);
    explicit basic_ostringstream(
        const basic_string<E, T, A>& x,
        ios_base::openmode mode = ios_base::out);
    basic_stringbuf<E, T, A> *rdbuf() const;
    basic_string<E, T, A> str();
    void str(const basic_string<E, T, A>& x);
};
```

The template class describes an object that controls insertion of elements and encoded objects into a stream buffer of class `basic_stringbuf<E, T, A>`, with elements of type `E`, whose character traits (page 211) are determined by the class `T`, and whose elements are allocated by an allocator of class `A`. The object stores an object of class `basic_stringbuf<E, T, A>`.

basic_ostringstream::basic_ostringstream

```
explicit basic_ostringstream(
    ios_base::openmode mode = ios_base::out);
explicit basic_ostringstream(
    const basic_string<E, T, A>& x,
    ios_base::openmode mode = ios_base::out);
```

The first constructor initializes the base class by calling `basic_ostream(sb)`, where `sb` is the stored object of class `basic_stringbuf<E, T, A>`. It also initializes `sb` by calling `basic_stringbuf<E, T, A>(mode | ios_base::out)`.

The second constructor initializes the base class by calling `basic_ostream(sb)`. It also initializes `sb` by calling `basic_stringbuf<E, T, A>(x, mode | ios_base::out)`.

basic_ostringstream::rdbuf

```
basic_stringbuf<E, T, A> *rdbuf() const
```

The member function returns the address of the stored stream buffer, of type pointer to `basic_stringbuf<E, T, A>`.

basic_ostringstream::str

```
basic_string<E, T, A> str() const;
void str(basic_string<E, T, A>& x);
```

The first member function returns `rdbuf()->str()`. The second member function calls `rdbuf()->str(x)`.

basic_stringstream

```
template <class E,
          class T = char_traits<E>,
          class A = allocator<E> >
class basic_stringstream
    : public basic_istream<E, T> {
public:
    explicit basic_stringstream(
        ios_base::openmode mode =
            ios_base::in | ios_base::out);
    explicit basic_stringstream(
        const basic_string<E, T, A>& x,
        ios_base::openmode mode =
            ios_base::in | ios_base::out);
    basic_stringbuf<E, T, A> *rdbuf() const;
    basic_string<E, T, A>& str();
    void str(const basic_string<E, T, A>& x);
};
```

The template class describes an object that controls insertion and extraction of elements and encoded objects using a stream buffer of class `basic_stringbuf<E, T, A>`, with elements of type `E`, whose character traits (page 211) are determined by the class `T`, and whose elements are allocated by an allocator of class `A`. The object stores an object of class `basic_stringbuf<E, T, A>`.

basic_stringstream::basic_stringstream

```
explicit basic_stringstream(
    ios_base::openmode mode =
        ios_base::in | ios_base::out);
explicit basic_stringstream(
    const basic_string<E, T, A>& x,
    ios_base::openmode mode =
        ios_base::in | ios_base::out);
```

The first constructor initializes the base class by calling `basic_istream(sb)`, where `sb` is the stored object of class `basic_stringbuf<E, T, A>`. It also initializes `sb` by calling `basic_stringbuf<E, T, A>(mode)`.

The second constructor initializes the base class by calling `basic_ostream(sb)`. It also initializes `sb` by calling `basic_stringbuf<E, T, A>(x, mode)`.

basic_stringstream::rdbuf

```
basic_stringbuf<E, T, A> *rdbuf() const
```

The member function returns the address of the stored stream buffer, of type pointer to `basic_stringbuf<E, T, A>`.

basic_stringstream::str

```
basic_string<E, T, A> str() const;
void str(basic_string<E, T, A>& x);
```

The first member function returns `rdbuf()->str()`. The second member function calls `rdbuf()->str(x)`.

istringstream

```
typedef basic_istringstream<char> istringstream;
```

The type is a synonym for template class `basic_istringstream` (page 180), specialized for elements of type *char*.

ostringstream

```
typedef basic_ostringstream<char> ostringstream;
```

The type is a synonym for template class `basic_ostringstream` (page 181), specialized for elements of type *char*.

stringbuf

```
typedef basic_stringbuf<char> stringbuf;
```

The type is a synonym for template class `basic_stringbuf` (page 176), specialized for elements of type *char*.

stringstream

```
typedef basic_stringstream<char> stringstream;
```

The type is a synonym for template class `basic_stringstream`, specialized for elements of type *char*.

wistringstream

```
typedef basic_istringstream<wchar_t> wistringstream;
```

The type is a synonym for template class `basic_istringstream`, specialized for elements of type *wchar_t*.

wostringstream

```
typedef basic_ostringstream<wchar_t> wostringstream;
```

The type is a synonym for template class `basic_ostringstream`, specialized for elements of type *wchar_t*.

wstringbuf

```
typedef basic_stringbuf<wchar_t> wstringbuf;
```

The type is a synonym for template class `basic_stringbuf`, specialized for elements of type *wchar_t*.

wstringstream

```
typedef basic_stringstream<wchar_t> wstringstream;
```

The type is a synonym for template class `basic_stringstream`, specialized for elements of type *wchar_t*.

<stdexcept>

```
namespace std {  
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;  
};
```

Include the standard header **<stdexcept>** to define several classes used for reporting exceptions. The classes form a derivation hierarchy, as indicated by the indenting above, all derived from class **exception** (page 75).

domain_error

```
class domain_error : public logic_error {  
public:  
    domain_error(const string& what_arg);  
};
```

The class serves as the base class for all exceptions thrown to report a domain error. The value returned by **what()** is a copy of **what_arg.data()**.

invalid_argument

```
class invalid_argument : public logic_error {  
public:  
    invalid_argument(const string& what_arg);  
};
```

The class serves as the base class for all exceptions thrown to report an invalid argument. The value returned by **what()** is a copy of **what_arg.data()**.

length_error

```
class length_error : public logic_error {  
public:  
    length_error(const string& what_arg);  
};
```

The class serves as the base class for all exceptions thrown to report an attempt to generate an object too long to be specified. The value returned by **what()** is a copy of **what_arg.data()**.

logic_error

```
class logic_error : public exception {  
public:  
    logic_error(const string& what_arg);  
};
```

The class serves as the base class for all exceptions thrown to report errors presumably detectable before the program executes, such as violations of logical preconditions. The value returned by **what()** is a copy of **what_arg.data()**.

out_of_range

```
class out_of_range : public logic_error {
public:
    out_of_range(const string& what_arg);
};
```

The class serves as the base class for all exceptions thrown to report an argument that is out of its valid range. The value returned by `what()` is a copy of `what_arg.data()`.

overflow_error

```
class overflow_error : public runtime_error {
public:
    overflow_error(const string& what_arg);
};
```

The class serves as the base class for all exceptions thrown to report an arithmetic overflow. The value returned by `what()` is a copy of `what_arg.data()`.

range_error

```
class range_error : public runtime_error {
public:
    range_error(const string& what_arg);
};
```

The class serves as the base class for all exceptions thrown to report a range error. The value returned by `what()` is a copy of `what_arg.data()`.

runtime_error

```
class runtime_error : public exception {
public:
    runtime_error(const string& what_arg);
};
```

The class serves as the base class for all exceptions thrown to report errors presumably detectable only when the program executes. The value returned by `what()` is a copy of `what_arg.data()`.

underflow_error

```
class underflow_error : public runtime_error {
public:
    underflow_error(const string& what_arg);
};
```

The class serves as the base class for all exceptions thrown to report an arithmetic underflow. The value returned by `what()` is a copy of `what_arg.data()`.

<streambuf>

```
namespace std {
template<class E, class T = char_traits<E> >
    class basic_streambuf;
typedef basic_streambuf<char, char_traits<char> >
    streambuf;
typedef basic_streambuf<wchar_t,
    char_traits<wchar_t> > wstreambuf;
};
```

Include the `istream` (page 7) standard header `<streambuf>` to define template class `basic_streambuf` (page 186), which is basic to the operation of the `istream` classes. (This header is typically included for you by another of the `istream` headers. You seldom have occasion to include it directly.)

`basic_streambuf`

`basic_streambuf` (page 188) · `char_type` (page 188) · `eback` (page 188) · `egptr` (page 188) · `epptr` (page 188) · `gbump` (page 188) · `getloc` (page 189) · `gptr` (page 189) · `imbue` (page 189) · `in_avail` (page 189) · `int_type` (page 189) · `off_type` (page 189) · `overflow` (page 189) · `pbackfail` (page 189) · `pbase` (page 190) · `pbump` (page 190) · `pos_type` (page 190) · `pptr` (page 190) · `pubimbue` (page 190) · `pubseekoff` (page 190) · `pubseekpos` (page 191) · `pubsetbuf` (page 191) · `pubsync` (page 191) · `sbumpc` (page 191) · `seekoff` (page 191) · `seekpos` (page 191) · `setbuf` (page 192) · `setg` (page 192) · `setp` (page 192) · `sgetc` (page 192) · `sgetn` (page 192) · `showmanyc` (page 192) · `snextc` (page 192) · `sputbackc` (page 192) · `sputc` (page 193) · `sputn` (page 193) · `stoss` (page 193) · `sungetc` (page 193) · `sync` (page 193) · `traits_type` (page 193) · `uflow` (page 193) · `underflow` (page 194) · `xsgetn` (page 194) · `xspn` (page 194)

```
template <class E, class T = char_traits<E> >
    class basic_streambuf {
public:
    typedef E char_type;
    typedef T traits_type;
    typedef typename traits_type::int_type int_type;
    typedef typename traits_type::pos_type pos_type;
    typedef typename traits_type::off_type off_type;
    virtual ~streambuf();
    locale pubimbue(const locale& loc);
    locale getloc() const;
    basic_streambuf *pubsetbuf(char_type *s,
        streamsize n);
    pos_type pubseekoff(off_type off,
        ios_base::seekdir way,
        ios_base::openmode which =
            ios_base::in | ios_base::out);
    pos_type pubseekpos(pos_type sp,
        ios_base::openmode which =
            ios_base::in | ios_base::out);
    int pubsync();
    streamsize in_avail();
    int_type snextc();
    int_type sbumpc();
    int_type sgetc();
    void stoss(); // OPTIONAL
    streamsize sgetn(char_type *s, streamsize n);
    int_type sputbackc(char_type c);
    int_type sungetc();
    int_type sputc(char_type c);
    streamsize sputn(const char_type *s, streamsize n);
protected:
    basic_streambuf();
    char_type *eback() const;
    char_type *gptr() const;
    char_type *egptr() const;
    void gbump(int n);
    void setg(char_type *gbeg,
        char_type *gnext, char_type *gend);
    char_type *pbase() const;
    char_type *pptr() const;
    char_type *epptr() const;
    void pbump(int n);
    void setp(char_type *pbeg, char_type *pend);
    virtual void imbue(const locale &loc);
```

```

virtual basic_streambuf *setbuf(char_type *s,
    streamsize n);
virtual pos_type seekoff(off_type off,
    ios_base::seekdir way,
    ios_base::openmode which =
        ios_base::in | ios_base::out);
virtual pos_type seekpos(pos_type sp,
    ios_base::openmode which =
        ios_base::in | ios_base::out);
virtual int sync();
virtual streamsize showmanyc();
virtual streamsize xsgetn(char_type *s,
    streamsize n);
virtual int_type underflow();
virtual int_type uflow();
virtual int_type pbackfail(int_type c =
    traits_type::eof());
virtual streamsize xsputn(const char_type *s,
    streamsize n);
virtual int_type overflow(int_type c =
    traits_type::eof());
};

```

The template class describes an abstract base class for deriving a **stream buffer**, which controls the transmission of elements to and from a specific representation of a stream. An object of class `basic_streambuf` helps control a stream with elements of type `T`, also known as `char_type` (page 188), whose character traits (page 211) are determined by the class `char_traits` (page 210), also known as `traits_type` (page 193).

Every stream buffer conceptually controls two independent streams, in fact, one for extractions (input) and one for insertions (output). A specific representation may, however, make either or both of these streams inaccessible. It typically maintains some relationship between the two streams. What you insert into the output stream of a `basic_stringbuf<E, T>` object, for example, is what you later extract from its input stream. And when you position one stream of a `basic_filebuf<E, T>` (page 77) object, you position the other stream in tandem.

The public interface to template class `basic_streambuf` (page 186) supplies the operations common to all stream buffers, however specialized. The protected interface supplies the operations needed for a specific representation of a stream to do its work. The protected virtual member functions let you tailor the behavior of a derived stream buffer for a specific representation of a stream. Each of the derived stream buffers in this library describes how it specializes the behavior of its protected virtual member functions. Documented here is the **default behavior** for the base class, which is often to do nothing.

The remaining protected member functions control copying to and from any storage supplied to buffer transmissions to and from streams. An **input buffer**, for example, is characterized by:

- `eback()` (page 188), a pointer to the beginning of the buffer
- `gptr()` (page 189), a pointer to the next element to read
- `egptr()` (page 188), a pointer just past the end of the buffer

Similarly, an **output buffer** is characterized by:

- `pbase()` (page 190), a pointer to the beginning of the buffer
- `pptr()` (page 190), a pointer to the next element to write
- `epptr()` (page 188), a pointer just past the end of the buffer

For any buffer, the protocol is:

- If the next pointer is null, no buffer exists. Otherwise, all three pointers point into the same sequence. (They can be safely compared for order.)
- For an output buffer, if the next pointer compares less than the end pointer, you can store an element at the **write position** designated by the next pointer.
- For an input buffer, if the next pointer compares less than the end pointer, you can read an element at the **read position** designated by the next pointer.
- For an input buffer, if the beginning pointer compares less than the next pointer, you can put back an element at the **putback position** designated by the decremented next pointer.

Any protected virtual member functions you write for a class derived from `basic_streambuf<E, T>` must cooperate in maintaining this protocol.

An object of class `basic_streambuf<E, T>` stores the six pointers described above. It also stores a **locale object** (page 135) in an object of type `locale` (page 134) for potential use by a derived stream buffer.

`basic_streambuf::basic_streambuf`

```
basic_streambuf();
```

The protected constructor stores a null pointer in all the pointers controlling the input buffer (page 187) and the output buffer (page 187). It also stores `locale::classic()` in the locale object (page 135).

`basic_streambuf::char_type`

```
typedef E char_type;
```

The type is a synonym for the template parameter `E`.

`basic_streambuf::eback`

```
char_type *eback() const;
```

The member function returns a pointer to the beginning of the input buffer (page 187).

`basic_streambuf::egptr`

```
char_type *egptr() const;
```

The member function returns a pointer just past the end of the input buffer (page 187).

`basic_streambuf::epptr`

```
char_type *epptr() const;
```

The member function returns a pointer just past the end of the output buffer (page 187).

`basic_streambuf::gbump`

```
void gbump(int n);
```

The member function adds `n` to the next pointer for the input buffer (page 187).

basic_streambuf::getloc

```
locale getloc() const;
```

The member function returns the stored locale object.

basic_streambuf::gptr

```
char_type *gptr() const;
```

The member function returns a pointer to the next element of the input buffer (page 187).

basic_streambuf::imbue

```
virtual void imbue(const locale &loc);
```

The default behavior is to do nothing.

basic_streambuf::in_avail

```
streamsize in_avail();
```

If a read position (page 188) is available, the member function returns `egptr() - gptr()`. Otherwise, it returns `showmanyc()`.

basic_streambuf::int_type

```
typedef typename traits_type::int_type int_type;
```

The type is a synonym for `traits_type::int_type`.

basic_streambuf::off_type

```
typedef typename traits_type::off_type off_type;
```

The type is a synonym for `traits_type::off_type`.

basic_streambuf::overflow

```
virtual int_type overflow(int_type c =  
    traits_type::eof());
```

If `c` does not compare equal to `traits_type::eof()`, the protected virtual member function endeavors to insert the element `traits_type::to_char_type(c)` into the output stream. It can do so in various ways:

- If a write position (page 188) is available, it can store the element into the write position and increment the next pointer for the output buffer (page 187).
- It can make a write position available by allocating new or additional storage for the output buffer.
- It can make a write position available by writing out, to some external destination, some or all of the elements between the beginning and next pointers for the output buffer.

If the function cannot succeed, it returns `traits_type::eof()` or throws an exception. Otherwise, it returns `traits_type::not_eof(c)`. The default behavior is to return `traits_type::eof()`.

basic_streambuf::pbackfail

```
virtual int_type pbackfail(int_type c =  
    traits_type::eof());
```

The protected virtual member function endeavors to put back an element into the input stream, then make it the current element (pointed to by the next pointer). If `c` compares equal to `traits_type::eof()`, the element to push back is effectively the one already in the stream before the current element. Otherwise, that element is replaced by `traits_type::to_char_type(c)`. The function can put back an element in various ways:

- If a putback position (page 188) is available, it can store the element into the putback position and decrement the next pointer for the input buffer (page 187).
- It can make a putback position available by allocating new or additional storage for the input buffer.
- For a stream buffer with common input and output streams, it can make a putback position available by writing out, to some external destination, some or all of the elements between the beginning and next pointers for the output buffer.

If the function cannot succeed, it returns `traits_type::eof()` or throws an exception. Otherwise, it returns some other value. The default behavior is to return `traits_type::eof()`.

basic_streambuf::pbase

`char_type *pbase() const;`

The member function returns a pointer to the beginning of the output buffer (page 187).

basic_streambuf::pbump

`void pbump(int n);`

The member function adds `n` to the next pointer for the output buffer (page 187).

basic_streambuf::pos_type

`typedef typename traits_type::pos_type pos_type;`

The type is a synonym for `traits_type::pos_type`.

basic_streambuf::pptr

`char_type *pptr() const;`

The member function returns a pointer to the next element of the output buffer.

basic_streambuf::pubimbue

`locale pubimbue(const locale& loc);`

The member function stores `loc` in the locale object, calls `imbue()`, then returns the previous value stored in the locale object.

basic_streambuf::pubseekoff

```
pos_type pubseekoff(off_type off,
                    ios_base::seekdir way,
                    ios_base::openmode which =
                    ios_base::in | ios_base::out);
```

The member function returns `seekoff(off, way, which)`.

basic_streambuf::pubseekpos

```
pos_type pubseekpos(pos_type sp,  
                    ios_base::openmode which =  
                    ios_base::in | ios_base::out);
```

The member function returns `seekpos(sp, which)`.

basic_streambuf::pubsetbuf

```
basic_streambuf *pubsetbuf(char_type *s, streamsize n);
```

The member function returns `stbuf(s, n)`.

basic_streambuf::pubsync

```
int pubsync();
```

The member function returns `sync()`.

basic_streambuf::sbumpc

```
int_type sbumpc();
```

If a read position (page 188) is available, the member function returns `traits_type::to_int_type(*gptr())` and increments the next pointer for the input buffer. Otherwise, it returns `uflow()`.

basic_streambuf::seekoff

```
virtual pos_type seekoff(off_type off,  
                        ios_base::seekdir way,  
                        ios_base::openmode which =  
                        ios_base::in | ios_base::out);
```

The protected virtual member function endeavors to alter the current positions for the controlled streams. The new position is determined as follows:

- If `way == ios_base::beg`, the new position is the beginning of the stream plus `off`.
- If `way == ios_base::cur`, the new position is the current stream position plus `off`.
- If `way == ios_base::end`, the new position is the end of the stream plus `off`.

Typically, if `which & ios_base::in` is nonzero, the input stream is affected, and if `which & ios_base::out` is nonzero, the output stream is affected. Actual use of this parameter varies among derived stream buffers, however.

If the function succeeds in altering the stream position(s), it returns the resultant stream position (or one of them). Otherwise, it returns an invalid stream position. The default behavior is to return an invalid stream position.

basic_streambuf::seekpos

```
virtual pos_type seekpos(pos_type sp,  
                        ios_base::openmode which =  
                        ios_base::in | ios_base::out);
```

The protected virtual member function endeavors to alter the current positions for the controlled streams. The new position is `sp`.

Typically, if `which & ios_base::in` is nonzero, the input stream is affected, and if `which & ios_base::out` is nonzero, the output stream is affected. Actual use of this parameter varies among derived stream buffers, however.

If the function succeeds in altering the stream position(s), it returns the resultant stream position (or one of them). Otherwise, it returns an invalid stream position. The default behavior is to return an invalid stream position.

basic_streambuf::setbuf

```
virtual basic_streambuf *setbuf(char_type *s,  
                                streamsize n);
```

The protected virtual member function performs an operation peculiar to each derived stream buffer. (See, for example, `basic_filebuf` (page 77).) The default behavior is to return this.

basic_streambuf::setg

```
void setg(char_type *gbeg, char_type *gnext,  
          char_type *gend);
```

The member function stores `gbeg` in the beginning pointer, `gnext` in the next pointer, and `gend` in the end pointer for the input buffer (page 187).

basic_streambuf::setp

```
void setp(char_type *pbeg, char_type *pend);
```

The member function stores `pbeg` in the beginning pointer, `pbeg` in the next pointer, and `pend` in the end pointer for the output buffer (page 187).

basic_streambuf::sgetc

```
int_type sgetc();
```

If a read position (page 188) is available, the member function returns `traits_type::to_int_type(*gptr())`. Otherwise, it returns `underflow()`.

basic_streambuf::sgetn

```
streamsize sgetn(char_type *s, streamsize n);
```

The member function returns `xsgetn(s, n)`.

basic_streambuf::showmanyc

```
virtual streamsize showmanyc();
```

The protected virtual member function returns a count of the number of characters that can be extracted from the input stream with no fear that the program will suffer an indefinite wait. The default behavior is to return zero.

basic_streambuf::snextc

```
int_type snextc();
```

The member function calls `sbumpc()` and, if that function returns `traits_type::eof()`, returns `traits_type::eof()`. Otherwise, it returns `sgetc()`.

basic_streambuf::sputbackc

```
int_type sputbackc(char_type c);
```

If a putback position (page 188) is available and `c` compares equal to the character stored in that position, the member function decrements the next pointer for the input buffer and returns `ch`, which is the value `traits_type::to_int_type(c)`. Otherwise, it returns `pbackfail(ch)`.

basic_streambuf::sputc

```
int_type sputc(char_type c);
```

If a write position (page 188) is available, the member function stores *c* in the write position, increments the next pointer for the output buffer, and returns *ch*, which is the value `traits_type::to_int_type(c)`. Otherwise, it returns `overflow(ch)`.

basic_streambuf::sputn

```
streamsize sputn(const char_type *s, streamsize n);
```

The member function returns `xputn(s, n)`.

basic_streambuf::stosscc

```
void stosscc(); // OPTIONAL
```

The member function calls `sbumpc()`. Note that an implementation is not required to supply this member function.

basic_streambuf::sungetc

```
int_type sungetc();
```

If a putback position (page 188) is available, the member function decrements the next pointer for the input buffer and returns `traits_type::to_int_type(*gptr())`. Otherwise it returns `pbackfail()`.

basic_streambuf::sync

```
virtual int sync();
```

The protected virtual member function endeavors to synchronize the controlled streams with any associated external streams. Typically, this involves writing out any elements between the beginning and next pointers for the output buffer. It does *not* involve putting back any elements between the next and end pointers for the input buffer. If the function cannot succeed, it returns -1. The default behavior is to return zero.

basic_streambuf::traits_type

```
typedef T traits_type;
```

The type is a synonym for the template parameter *T*.

basic_streambuf::uflow

```
virtual int_type uflow();
```

The protected virtual member function endeavors to extract the current element *c* from the input stream, then advance the current stream position, and return the element as `traits_type::to_int_type(c)`. It can do so in various ways:

- If a read position (page 188) is available, it takes *c* as the element stored in the read position and advances the next pointer for the input buffer.
- It can read an element directly, from some external source, and deliver it as the value *c*.
- For a stream buffer with common input and output streams, it can make a read position available by writing out, to some external destination, some or all of the elements between the beginning and next pointers for the output buffer. Or it can allocate new or additional storage for the input buffer. The function then reads in, from some external source, one or more elements.

If the function cannot succeed, it returns `traits_type::eof()`, or throws an exception. Otherwise, it returns the current element `c` in the input stream, converted as described above, and advances the next pointer for the input buffer. The default behavior is to call `underflow()` and, if that function returns `traits_type::eof()`, to return `traits_type::eof()`. Otherwise, the function returns the current element `c` in the input stream, converted as described above, and advances the next pointer for the input buffer.

basic_streambuf::underflow

```
virtual int_type underflow();
```

The protected virtual member function endeavors to extract the current element `c` from the input stream, without advancing the current stream position, and return it as `traits_type::to_int_type(c)`. It can do so in various ways:

- If a read position (page 188) is available, `c` is the element stored in the read position.
- It can make a read position available by allocating new or additional storage for the input buffer, then reading in, from some external source, one or more elements.

If the function cannot succeed, it returns `traits_type::eof()`, or throws an exception. Otherwise, it returns the current element in the input stream, converted as described above. The default behavior is to return `traits_type::eof()`.

basic_streambuf::xsgetn

```
virtual streamsize xsgetn(char_type *s, streamsize n);
```

The protected virtual member function extracts up to `n` elements from the input stream, as if by repeated calls to `sgetc` (page 191), and stores them in the array beginning at `s`. It returns the number of elements actually extracted.

basic_streambuf::xspn

```
virtual streamsize xspn(const char_type *s,
    streamsize n);
```

The protected virtual member function inserts up to `n` elements into the output stream, as if by repeated calls to `sputc` (page 193), from the array beginning at `s`. It returns the number of elements actually inserted.

streambuf

```
typedef basic_streambuf<char, char_traits<char> >
    streambuf;
```

The type is a synonym for template class `basic_streambuf`, specialized for elements of type `char` with default character traits (page 211).

wstreambuf

```
typedef basic_streambuf<wchar_t, char_traits<wchar_t> >
    wstreambuf;
```

The type is a synonym for template class `basic_streambuf`, specialized for elements of type `wchar_t` with default character traits (page 211).

<string>

basic_string (page 197) · **char_traits** (page 210) · **char_traits<char>** (page 213) · **char_traits<wchar_t>** (page 213) · **getline** (page 214) · **operator+** (page 214) · **operator!=** (page 214) · **operator==** (page 215) · **operator<** (page 215) · **operator<<** (page 215) · **operator<=** (page 216) · **operator>** (page 216) · **operator>=** (page 216) · **operator>>** (page 216) · **string** (page 217) · **swap** (page 217) · **wstring** (page 217)

```
namespace std {
template<class E>
    class char_traits;
template<>
    class char_traits<char>;
template<>
    class char_traits<wchar_t>;
template<class E,
    class T = char_traits<E>,
    class A = allocator<E> >
    class basic_string;
typedef basic_string<char> string;
typedef basic_string<wchar_t> wstring;

    // TEMPLATE FUNCTIONS
template<class E, class T, class A>
    basic_string<E, T, A> operator+(
        const basic_string<E, T, A>& lhs,
        const basic_string<E, T, A>& rhs);
template<class E, class T, class A>
    basic_string<E, T, A> operator+(
        const basic_string<E, T, A>& lhs,
        const E *rhs);
template<class E, class T, class A>
    basic_string<E, T, A> operator+(
        const basic_string<E, T, A>& lhs,
        const basic_string<E, T, A>& rhs);
template<class E, class T, class A>
    basic_string<E, T, A> operator+(
        const E *lhs,
        const basic_string<E, T, A>& rhs);
template<class E, class T, class A>
    basic_string<E, T, A> operator+(
        const basic_string<E, T, A>& lhs,
        const E *rhs);
template<class E, class T, class A>
    basic_string<E, T, A> operator+(
        const E *lhs,
        const basic_string<E, T, A>& rhs);
template<class E, class T, class A>
    basic_string<E, T, A> operator+(
        const E *lhs,
        const basic_string<E, T, A>& rhs);
template<class E, class T, class A>
    bool operator==(
        const basic_string<E, T, A>& lhs,
        const basic_string<E, T, A>& rhs);
template<class E, class T, class A>
    bool operator==(
        const basic_string<E, T, A>& lhs,
        const E *rhs);
template<class E, class T, class A>
    bool operator==(
        const E *lhs,
        const basic_string<E, T, A>& rhs);
template<class E, class T, class A>
    bool operator!=(
        const basic_string<E, T, A>& lhs,
        const basic_string<E, T, A>& rhs);
template<class E, class T, class A>
    bool operator!=(
        const basic_string<E, T, A>& lhs,
        const E *rhs);
template<class E, class T, class A>
    bool operator!=(
        const E *lhs,
```

```

        const basic_string<E, T, A>& rhs);
template<class E, class T, class A>
    bool operator<(
        const basic_string<E, T, A>& lhs,
        const basic_string<E, T, A>& rhs);
template<class E, class T, class A>
    bool operator<(
        const basic_string<E, T, A>& lhs,
        const E *rhs);
template<class E, class T, class A>
    bool operator<(
        const E *lhs,
        const basic_string<E, T, A>& rhs);
template<class E, class T, class A>
    bool operator>(
        const basic_string<E, T, A>& lhs,
        const basic_string<E, T, A>& rhs);
template<class E, class T, class A>
    bool operator>(
        const basic_string<E, T, A>& lhs,
        const E *rhs);
template<class E, class T, class A>
    bool operator>(
        const E *lhs,
        const basic_string<E, T, A>& rhs);
template<class E, class T, class A>
    bool operator<=(
        const basic_string<E, T, A>& lhs,
        const basic_string<E, T, A>& rhs);
template<class E, class T, class A>
    bool operator<=(
        const basic_string<E, T, A>& lhs,
        const E *rhs);
template<class E, class T, class A>
    bool operator<=(
        const E *lhs,
        const basic_string<E, T, A>& rhs);
template<class E, class T, class A>
    bool operator>=(
        const basic_string<E, T, A>& lhs,
        const basic_string<E, T, A>& rhs);
template<class E, class T, class A>
    bool operator>=(
        const basic_string<E, T, A>& lhs,
        const E *rhs);
template<class E, class T, class A>
    bool operator>=(
        const E *lhs,
        const basic_string<E, T, A>& rhs);
template<class E, class T, class A>
    void swap(
        basic_string<E, T, A>& lhs,
        basic_string<E, T, A>& rhs);
template<class E, class T, class A>
    basic_ostream<E>& operator<<(
        basic_ostream<E>& os,
        const basic_string<E, T, A>& str);
template<class E, class T, class A>
    basic_istream<E>& operator>>(
        basic_istream<E>& is,
        basic_string<E, T, A>& str);
template<class E, class T, class A>
    basic_istream<E, T>& getline(
        basic_istream<E, T>& is,
        basic_string<E, T, A>& str);
template<class E, class T, class A>
    basic_istream<E, T>& getline(

```

```

        basic_istream<E, T>& is,
        basic_string<E, T, A>& str,
        E delim);
};

```

Include the standard header **<string>** to define the container (page 41) template class **basic_string** (page 197) and various supporting templates.

basic_string

basic_string (page 202) · **allocator_type** (page 201) · **append** (page 201) · **assign** (page 202) · **at** (page 202) · **begin** (page 203) · **c_str** (page 203) · **capacity** (page 203) · **clear** (page 203) · **compare** (page 203) · **const_iterator** (page 204) · **const_pointer** (page 204) · **const_reference** (page 204) · **const_reverse_iterator** (page 204) · **copy** (page 204) · **data** (page 204) · **difference_type** (page 204) · **empty** (page 205) · **end** (page 205) · **erase** (page 205) · **find** (page 205) · **find_first_not_of** (page 205) · **find_first_of** (page 206) · **find_last_not_of** (page 206) · **find_last_of** (page 206) · **get_allocator** (page 206) · **insert** (page 206) · **iterator** (page 207) · **length** (page 207) · **max_size** (page 207) · **npos** (page 207) · **operator+=** (page 207) · **operator=** (page 207) · **operator[]** (page 208) · **pointer** (page 208) · **push_back** (page 208) · **rbegin** (page 208) · **reference** (page 208) · **rend** (page 208) · **replace** (page 208) · **reserve** (page 209) · **resize** (page 209) · **reverse_iterator** (page 209) · **rfind** (page 209) · **size** (page 210) · **size_type** (page 210) · **substr** (page 210) · **swap** (page 210) · **traits_type** (page 210) · **value_type** (page 210)

```

template<class E,
        class T = char_traits<E>,
        class A = allocator<T> >
class basic_string {
public:
    typedef T traits_type;
    typedef A allocator_type;
    typedef T0 iterator;
    typedef T1 const_iterator;
    typedef T2 size_type;
    typedef T3 difference_type;
    typedef reverse_iterator<const_iterator>
        const_reverse_iterator;
    typedef reverse_iterator<iterator>
        reverse_iterator;
    typedef typename allocator_type::pointer
        pointer;
    typedef typename allocator_type::const_pointer
        const_pointer;
    typedef typename allocator_type::reference
        reference;
    typedef typename allocator_type::const_reference
        const_reference;
    typedef typename allocator_type::value_type
        value_type;
    static const size_type npos = -1;
    basic_string();
    explicit basic_string(const allocator_type& al);
    basic_string(const basic_string& rhs);
    basic_string(const basic_string& rhs, size_type pos,
        size_type n = npos);
    basic_string(const basic_string& rhs, size_type pos,
        size_type n, const allocator_type& al);
    basic_string(const value_type *s, size_type n);
    basic_string(const value_type *s, size_type n,
        const allocator_type& al);
    basic_string(const value_type *s);
    basic_string(const value_type *s,
        const allocator_type& al);

```

```

basic_string(size_type n, value_type c);
basic_string(size_type n, value_type c,
    const allocator_type& al);
template <class InIt>
    basic_string(InIt first, InIt last);
template <class InIt>
    basic_string(InIt first, InIt last,
        const allocator_type& al);
basic_string& operator=(const basic_string& rhs);
basic_string& operator=(const value_type *s);
basic_string& operator=(value_type c);
iterator begin();
const_iterator begin() const;
iterator end();
const_iterator end() const;
reverse_iterator rbegin();
const_reverse_iterator rbegin() const;
reverse_iterator rend();
const_reverse_iterator rend() const;
const_reference at(size_type pos) const;
reference at(size_type pos);
const_reference operator[](size_type pos) const;
reference operator[](size_type pos);
void push_back(value_type c);
const value_type *c_str() const;
const value_type *data() const;
size_type length() const;
size_type size() const;
size_type max_size() const;
void resize(size_type n, value_type c = value_type());
size_type capacity() const;
void reserve(size_type n = 0);
bool empty() const;
basic_string& operator+=(const basic_string& rhs);
basic_string& operator+=(const value_type *s);
basic_string& operator+=(value_type c);

basic_string& append(const basic_string& str);
basic_string& append(const basic_string& str,
    size_type pos, size_type n);
basic_string& append(const value_type *s,
    size_type n);
basic_string& append(const value_type *s);
basic_string& append(size_type n, value_type c);
template<class InIt>
    basic_string& append(InIt first, InIt last);
basic_string& assign(const basic_string& str);
basic_string& assign(const basic_string& str,
    size_type pos, size_type n);
basic_string& assign(const value_type *s,
    size_type n);
basic_string& assign(const value_type *s);
basic_string& assign(size_type n, value_type c);
template<class InIt>
    basic_string& assign(InIt first, InIt last);
basic_string& insert(size_type p0,
    const basic_string& str);
basic_string& insert(size_type p0,
    const basic_string& str, size_type pos,
    size_type n);
basic_string& insert(size_type p0,
    const value_type *s, size_type n);
basic_string& insert(size_type p0,
    const value_type *s);
basic_string& insert(size_type p0,
    size_type n, value_type c);
iterator insert(iterator it,
    value_type c = value_type());

```

```

void insert(iterator it, size_type n, value_type c);
template<class InIt>
    void insert(iterator it,
        InIt first, InIt last);
basic_string& erase(size_type p0 = 0,
    size_type n = npos);
iterator erase(iterator it);
iterator erase(iterator first, iterator last);
void clear();
basic_string& replace(size_type p0, size_type n0,
    const basic_string& str);
basic_string& replace(size_type p0, size_type n0,
    const basic_string& str, size_type pos,
    size_type n);
basic_string& replace(size_type p0, size_type n0,
    const value_type *s, size_type n);
basic_string& replace(size_type p0, size_type n0,
    const value_type *s);
basic_string& replace(size_type p0, size_type n0,
    size_type n, value_type c);
basic_string& replace(iterator first0, iterator last0,
    const basic_string& str);
basic_string& replace(iterator first0, iterator last0,
    const value_type *s, size_type n);
basic_string& replace(iterator first0, iterator last0,
    const value_type *s);
basic_string& replace(iterator first0, iterator last0,
    size_type n, value_type c);
template<class InIt>
    basic_string&
        replace(iterator first0, iterator last0,
            InIt first, InIt last);
size_type copy(value_type *s, size_type n,
    size_type pos = 0) const;
void swap(basic_string& str);
size_type find(const basic_string& str,
    size_type pos = 0) const;
size_type find(const value_type *s, size_type pos,
    size_type n) const;
size_type find(const value_type *s,
    size_type pos = 0) const;
size_type find(value_type c, size_type pos = 0) const;
size_type rfind(const basic_string& str,
    size_type pos = npos) const;
size_type rfind(const value_type *s, size_type pos,
    size_type n = npos) const;
size_type rfind(const value_type *s,
    size_type pos = npos) const;
size_type rfind(value_type c,
    size_type pos = npos) const;
size_type find_first_of(const basic_string& str,
    size_type pos = 0) const;
size_type find_first_of(const value_type *s,
    size_type pos, size_type n) const;
size_type find_first_of(const value_type *s,
    size_type pos = 0) const;
size_type find_first_of(value_type c,
    size_type pos = 0) const;
size_type find_last_of(const basic_string& str,
    size_type pos = npos) const;
size_type find_last_of(const value_type *s,
    size_type pos, size_type n = npos) const;
size_type find_last_of(const value_type *s,
    size_type pos = npos) const;
size_type find_last_of(value_type c,
    size_type pos = npos) const;
size_type find_first_not_of(const basic_string& str,

```

```

        size_type pos = 0) const;
size_type find_first_not_of(const value_type *s,
    size_type pos, size_type n) const;
size_type find_first_not_of(const value_type *s,
    size_type pos = 0) const;
size_type find_first_not_of(value_type c,
    size_type pos = 0) const;
size_type find_last_not_of(const basic_string& str,
    size_type pos = npos) const;
size_type find_last_not_of(const value_type *s,
    size_type pos, size_type n) const;
size_type find_last_not_of(const value_type *s,
    size_type pos = npos) const;
size_type find_last_not_of(value_type c,
    size_type pos = npos) const;
basic_string substr(size_type pos = 0,
    size_type n = npos) const;
int compare(const basic_string& str) const;
int compare(size_type p0, size_type n0,
    const basic_string& str);
int compare(size_type p0, size_type n0,
    const basic_string& str, size_type pos,
        size_type n);
int compare(const value_type *s) const;
int compare(size_type p0, size_type n0,
    const value_type *s) const;
int compare(size_type p0, size_type n0,
    const value_type *s, size_type pos) const;
allocator_type get_allocator() const;
};

```

The template class describes an object that controls a varying-length sequence of elements of type `E`, also known as `value_type` (page 210). Such an element type must not require explicit construction or destruction, and it must be suitable for use as the `E` parameter to `basic_istream` (page 106) or `basic_ostream` (page 169). (A “plain old data structure,” or **POD**, from C generally meets this criterion.) The Standard C++ library provides two specializations of this template class, with the type definitions `string` (page 217), for elements of type `char`, and `wstring` (page 217), for elements of type `wchar_t`.

Various important properties of the elements in a `basic_string` specialization are described by the class `T`, also known as `traits_type` (page 210). A class that specifies these character traits (page 211) must have the same external interface as an object of template class `char_traits` (page 210).

The object allocates and frees storage for the sequence it controls through a stored allocator object (page 337) of class `A`, also known as `allocator_type` (page 201). Such an allocator object must have the same external interface as an object of template class `allocator` (page 337). (Class `char_traits` (page 210) has no provision for alternate addressing schemes, such as might be required to implement a far heap (page 338).) Note that the stored allocator object is *not* copied when the container object is assigned.

The sequences controlled by an object of template class `basic_string` are usually called **strings**. These objects should not be confused, however, with the null-terminated C strings used throughout the Standard C++ library.

Many member functions require an **operand sequence** of elements. You can specify such an operand sequence several ways:

- `c` — one element with value `c`

- `n, c` — a repetition of `n` elements each with value `c`
- `s` — a null-terminated sequence (such as a C string, for `E` of type *char*) beginning at `s` (which must not be a null pointer), where the terminating element is the value `value_type()` and is not part of the operand sequence
- `s, n` — a sequence of `n` elements beginning at `s` (which must not be a null pointer)
- `str` — the sequence specified by the `basic_string` object `str`
- `str, pos, n` — the substring of the `basic_string` object `str` with up to `n` elements (or through the end of the string, whichever comes first) beginning at position `pos`
- `first, last` — a sequence of elements delimited by the iterators `first` and `last`, in the range `[first, last)`, which must *not* overlap the sequence controlled by the string object whose member function is being called

If a **position argument** (such as `pos` above) is beyond the end of the string on a call to a `basic_string` member function, the function reports an **out-of-range error** by throwing an object of class `out_of_range` (page 185).

If a function is asked to generate a sequence longer than `max_size()` elements, the function reports a **length error** by throwing an object of class `length_error` (page 184).

References, pointers, and iterators that designate elements of the controlled sequence can become invalid after any call to a function that alters the controlled sequence, or after the first call to the non-const member functions at (page 202), `begin` (page 203), `end` (page 205), `operator[]` (page 208), `rbegin` (page 208), or `rend` (page 208). (The idea is to permit multiple strings to share the same representation until one string becomes a candidate for change, at which point that string makes a private copy of the representation, using a discipline called **copy on write**.)

`basic_string::allocator_type`

```
typedef A allocator_type;
```

The type is a synonym for the template parameter `A`.

`basic_string::append`

```
basic_string& append(const value_type *s);
basic_string& append(const value_type *s,
                    size_type n);
basic_string& append(const basic_string& str,
                    size_type pos, size_type n);
basic_string& append(const basic_string& str);
basic_string& append(size_type n, value_type c);
template<class InIt>
    basic_string& append(InIt first, InIt last);
```

If `InIt` is an integer type, the template member function behaves the same as `append((size_type)first, (value_type)last)`. Otherwise, the member functions each append the operand sequence (page 200) to the end of the sequence controlled by `*this`, then return `*this`.

In this implementation (page 3), if a translator does not support member template functions, the template:

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

is replaced by:

```
basic_string& append(const_pointer first,
                    const_pointer last);
```

basic_string::assign

```
basic_string& assign(const value_type *s);
basic_string& assign(const value_type *s,
                    size_type n);
basic_string& assign(const basic_string& str,
                    size_type pos, size_type n);
basic_string& assign(const basic_string& str);
basic_string& assign(size_type n, value_type c);
template<class InIt>
    basic_string& assign(InIt first, InIt last);
```

If InIt is an integer type, the template member function behaves the same as `assign((size_type)first, (value_type)last)`. Otherwise, the member functions each replace the sequence controlled by `*this` with the operand sequence (page 200), then return `*this`.

In this implementation (page 3), if a translator does not support member template functions, the template:

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

is replaced by:

```
basic_string& assign(const_pointer first,
                    const_pointer last);
```

basic_string::at

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

The member functions each return a reference to the element of the controlled sequence at position `pos`, or report an out-of-range error (page 201).

basic_string::basic_string

```
basic_string(const value_type *s);
basic_string(const value_type *s,
            const allocator_type& al);
basic_string(const value_type *s, size_type n);
basic_string(const value_type *s, size_type n,
            const allocator_type& al);
basic_string(const basic_string& rhs);
basic_string(const basic_string& rhs, size_type pos,
            size_type n = npos);
basic_string(const basic_string& rhs, size_type pos,
            size_type n, const allocator_type& al);
basic_string(size_type n, value_type c);
basic_string(size_type n, value_type c,
            const allocator_type& al);
basic_string();
explicit basic_string(const allocator_type& al);
template <class InIt>
    basic_string(InIt first, InIt last);
template <class InIt>
    basic_string(InIt first, InIt last, const allocator_type& al);
```

All constructors store an allocator object (page 337) and initialize the controlled sequence. The allocator object is the argument `al`, if present. For the copy constructor, it is `x.get_allocator()`. Otherwise, it is `A()`.

The controlled sequence is initialized to a copy of the operand sequence (page 200) specified by the remaining operands. A constructor with no operand sequence specifies an empty initial controlled sequence. If `InIt` is an integer type in a template constructor, the operand sequence `first`, `last` behaves the same as `(size_type)first`, `(value_type)last`.

In this implementation (page 3), if a translator does not support member template functions, the templates:

```
template <class InIt>
    basic_string(InIt first, InIt last);
template <class InIt>
    basic_string(InIt first, InIt last,
                  const allocator_type& al);
```

are replaced by:

```
basic_string(const_pointer first, const_pointer last);
basic_string(const_pointer first, const_pointer last,
              const allocator_type& al);
```

basic_string::begin

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

The member functions each return a random-access iterator that points at the first element of the sequence (or just beyond the end of an empty sequence).

basic_string::c_str

```
const value_type *c_str() const;
```

The member function returns a pointer to a non-modifiable C string constructed by adding a terminating null element (`value_type()`) to the controlled sequence. Calling any non-const member function for `*this` can invalidate the pointer.

basic_string::capacity

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

The member function returns the storage currently allocated to hold the controlled sequence, a value at least as large as `size()`.

basic_string::clear

```
void clear();
```

The member function calls `erase(begin(), end())`.

basic_string::compare

```
int compare(const basic_string& str) const;
int compare(size_type p0, size_type n0,
            const basic_string& str);
int compare(size_type p0, size_type n0,
            const basic_string& str, size_type pos, size_type n);
int compare(const value_type *s) const;
int compare(size_type p0, size_type n0,
            const value_type *s) const;
int compare(size_type p0, size_type n0,
            const value_type *s, size_type pos) const;
```

The member functions each compare up to `n0` elements of the controlled sequence beginning with position `p0`, or the entire controlled sequence if these arguments are not supplied, to the operand sequence (page 200). Each function returns:

- a negative value if the first differing element in the controlled sequence compares less than the corresponding element in the operand sequence (as determined by `traits_type::compare`), or if the two have a common prefix but the operand sequence is longer
- zero if the two compare equal element by element and are the same length
- a positive value otherwise

basic_string::const_iterator

```
typedef T1 const_iterator;
```

The type describes an object that can serve as a constant random-access iterator for the controlled sequence. It is described here as a synonym for the implementation-defined type `T1`.

basic_string::const_pointer

```
typedef typename allocator_type::const_pointer  
const_pointer;
```

The type is a synonym for `allocator_type::const_pointer`.

basic_string::const_reference

```
typedef typename allocator_type::const_reference  
const_reference;
```

The type is a synonym for `allocator_type::const_reference`.

basic_string::const_reverse_iterator

```
typedef reverse_iterator<const_iterator>  
const_reverse_iterator;
```

The type describes an object that can serve as a constant reverse iterator for the controlled sequence.

basic_string::copy

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

The member function copies up to `n` elements from the controlled sequence, beginning at position `pos`, to the array of `value_type` beginning at `s`. It returns the number of elements actually copied.

basic_string::data

```
const value_type *data() const;
```

The member function returns a pointer to the first element of the sequence (or, for an empty sequence, a non-null pointer that cannot be dereferenced).

basic_string::difference_type

```
typedef T3 difference_type;
```

The signed integer type describes an object that can represent the difference between the addresses of any two elements in the controlled sequence. It is described here as a synonym for the implementation-defined type `T3`.

basic_string::empty

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

The member function returns true for an empty controlled sequence.

basic_string::end

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

The member functions each return a random-access iterator that points just beyond the end of the sequence.

basic_string::erase

```
iterator erase(iterator first, iterator last);  
iterator erase(iterator it);  
basic_string& erase(size_type p0 = 0,  
    size_type n = npos);
```

The first member function removes the elements of the controlled sequence in the range [first, last). The second member function removes the element of the controlled sequence pointed to by it. Both return an iterator that designates the first element remaining beyond any elements removed, or end() if no such element exists.

The third member function removes up to n elements of the controlled sequence beginning at position p0, then returns *this.

basic_string::find

```
size_type find(value_type c, size_type pos = 0) const;  
size_type find(const value_type *s,  
    size_type pos = 0) const;  
size_type find(const value_type *s, size_type pos,  
    size_type n) const;  
size_type find(const basic_string& str,  
    size_type pos = 0) const;
```

The member functions each find the first (lowest beginning position) subsequence in the controlled sequence, beginning on or after position pos, that matches the operand sequence (page 200) specified by the remaining operands. If it succeeds, it returns the position where the matching subsequence begins. Otherwise, the function returns npos (page 207).

basic_string::find_first_not_of

```
size_type find_first_not_of(value_type c,  
    size_type pos = 0) const;  
size_type find_first_not_of(const value_type *s,  
    size_type pos = 0) const;  
size_type find_first_not_of(const value_type *s,  
    size_type pos, size_type n) const;  
size_type find_first_not_of(const basic_string& str,  
    size_type pos = 0) const;
```

The member functions each find the first (lowest position) element of the controlled sequence, at or after position pos, that matches *none* of the elements in the operand sequence (page 200) specified by the remaining operands. If it succeeds, it returns the position. Otherwise, the function returns npos (page 207).

basic_string::find_first_of

```
size_type find_first_of(value_type c,  
    size_type pos = 0) const;  
size_type find_first_of(const value_type *s,  
    size_type pos = 0) const;  
size_type find_first_of(const value_type *s,  
    size_type pos, size_type n) const;  
size_type find_first_of(const basic_string& str,  
    size_type pos = 0) const;
```

The member functions each find the first (lowest position) element of the controlled sequence, at or after position *pos*, that matches *any* of the elements in the operand sequence (page 200) specified by the remaining operands. If it succeeds, it returns the position. Otherwise, the function returns *npos* (page 207).

basic_string::find_last_not_of

```
size_type find_last_not_of(value_type c,  
    size_type pos = npos) const;  
size_type find_last_not_of(const value_type *s,  
    size_type pos = npos) const;  
size_type find_last_not_of(const value_type *s,  
    size_type pos, size_type n) const;  
size_type find_last_not_of(const basic_string& str,  
    size_type pos = npos) const;
```

The member functions each find the last (highest position) element of the controlled sequence, at or before position *pos*, that matches *none* of the elements in the operand sequence (page 200) specified by the remaining operands. If it succeeds, it returns the position. Otherwise, the function returns *npos* (page 207).

basic_string::find_last_of

```
size_type find_last_of(value_type c,  
    size_type pos = npos) const;  
size_type find_last_of(const value_type *s,  
    size_type pos = npos) const;  
size_type find_last_of(const value_type *s,  
    size_type pos, size_type n = npos) const;  
size_type find_last_of(const basic_string& str,  
    size_type pos = npos) const;
```

The member functions each find the last (highest position) element of the controlled sequence, at or before position *pos*, that matches *any* of the elements in the operand sequence (page 200) specified by the remaining operands. If it succeeds, it returns the position. Otherwise, the function returns *npos* (page 207).

basic_string::get_allocator

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

The member function returns the stored allocator object (page 337).

basic_string::insert

```
basic_string& insert(size_type p0, const value_type *s);  
basic_string& insert(size_type p0, const value_type *s,  
    size_type n);  
basic_string& insert(size_type p0,  
    const basic_string& str);  
basic_string& insert(size_type p0,  
    const basic_string& str, size_type pos, size_type n);  
basic_string& insert(size_type p0,  
    size_type n, value_type c);  
iterator insert(iterator it,
```

```

        value_type c = value_type();
template<class InIt>
    void insert(iterator it, InIt first, InIt last);
void insert(iterator it, size_type n, value_type c);

```

The member functions each insert, before position `p0` or before the element pointed to by `it` in the controlled sequence, the operand sequence (page 200) specified by the remaining operands. A function that returns a value returns `*this`. If `InIt` is an integer type in the template member function, the operand sequence `first, last` behaves the same as `(size_type)first, (value_type)last`.

In this implementation (page 3), if a translator does not support member template functions, the template:

```

template<class InIt>
    void insert(iterator it, InIt first, InIt last);

```

is replaced by:

```

void insert(iterator it,
            const_pointer first, const_pointer last);

```

basic_string::iterator

```
typedef T0 iterator;
```

The type describes an object that can serve as a random-access iterator for the controlled sequence. It is described here as a synonym for the implementation-defined type `T0`.

basic_string::length

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

The member function returns the length of the controlled sequence (same as `size()`).

basic_string::max_size

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

The member function returns the length of the longest sequence that the object can control.

basic_string::npos

```
static const size_type npos = -1;
```

The constant is the largest representable value of type `size_type`. It is assuredly larger than `max_size()`, hence it serves as either a very large value or as a special code.

basic_string::operator+=

```

basic_string& operator+=(value_type c);
basic_string& operator+=(const value_type *s);
basic_string& operator+=(const basic_string& rhs);

```

The operators each append the operand sequence (page 200) to the end of the sequence controlled by `*this`, then return `*this`.

basic_string::operator=

```

basic_string& operator=(value_type c);
basic_string& operator=(const value_type *s);
basic_string& operator=(const basic_string& rhs);

```

The operators each replace the sequence controlled by `*this` with the operand sequence (page 200), then return `*this`.

basic_string::operator[]

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

The member functions each return a reference to the element of the controlled sequence at position `pos`. If that position is invalid, the behavior is undefined.

basic_string::pointer

```
typedef typename allocator_type::pointer  
    pointer;
```

The type is a synonym for `allocator_type::pointer`.

basic_string::push_back

```
void push_back(value_type c);
```

The member function effectively calls `insert(end(), c)`.

basic_string::rbegin

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

The member function returns a reverse iterator that points just beyond the end of the controlled sequence. Hence, it designates the beginning of the reverse sequence.

basic_string::reference

```
typedef typename allocator_type::reference  
    reference;
```

The type is a synonym for `allocator_type::reference`.

basic_string::rend

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

The member functions each return a reverse iterator that points at the first element of the sequence (or just beyond the end of an empty sequence). Hence, the function designates the end of the reverse sequence.

basic_string::replace

```
basic_string& replace(size_type p0, size_type n0,  
    const value_type *s);  
basic_string& replace(size_type p0, size_type n0,  
    const value_type *s, size_type n);  
basic_string& replace(size_type p0, size_type n0,  
    const basic_string& str);  
basic_string& replace(size_type p0, size_type n0,  
    const basic_string& str, size_type pos, size_type n);  
basic_string& replace(size_type p0, size_type n0,  
    size_type n, value_type c);  
basic_string& replace(iterator first0, iterator last0,  
    const value_type *s);  
basic_string& replace(iterator first0, iterator last0,  
    const value_type *s, size_type n);  
basic_string& replace(iterator first0, iterator last0,  
    const basic_string& str);
```



```

basic_string& replace(iterator first0, iterator last0,
    size_type n, value_type c);
template<class InIt>
basic_string&
    replace(iterator first0, iterator last0,
        InIt first, InIt last);

```

The member functions each replace up to `n0` elements of the controlled sequence beginning with position `p0`, or the elements of the controlled sequence beginning with the one pointed to by `first`, up to but not including `last`. The replacement is the operand sequence (page 200) specified by the remaining operands. The function then returns `*this`. If `InIt` is an integer type in the template member function, the operand sequence `first`, `last` behaves the same as `(size_type)first`, `(value_type)last`.

In this implementation (page 3), if a translator does not support member template functions, the template:

```

template<class InIt>
basic_string& replace(iterator first0, iterator last0,
    InIt first, InIt last);

```

is replaced by:

```

basic_string& replace(iterator first0, iterator last0,
    const_pointer first, const_pointer last);

```

basic_string::reserve

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

The member function ensures that `capacity()` henceforth returns at least `n`.

basic_string::resize

```
void resize(size_type n, value_type c = value_type());
```

The member function ensures that `size()` henceforth returns `n`. If it must make the controlled sequence longer, it appends elements with value `c`. To make the controlled sequence shorter, the member function effectively calls `erase(begin() + n, end())`.

basic_string::reverse_iterator

```

typedef reverse_iterator<iterator>
    reverse_iterator;

```

The type describes an object that can serve as a reverse iterator for the controlled sequence.

basic_string::rfind

```

size_type rfind(value_type c, size_type pos = npos) const;
size_type rfind(const value_type *s,
    size_type pos = npos) const;
size_type rfind(const value_type *s,
    size_type pos, size_type n = npos) const;
size_type rfind(const basic_string& str,
    size_type pos = npos) const;

```

The member functions each find the last (highest beginning position) subsequence in the controlled sequence, beginning on or before position `pos`, that matches the operand sequence (page 200) specified by the remaining operands. If it succeeds, the function returns the position where the matching subsequence begins. Otherwise, it returns `npos` (page 207).

basic_string::size

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

The member function returns the length of the controlled sequence.

basic_string::size_type

```
typedef T2 size_type;
```

The unsigned integer type describes an object that can represent the length of any controlled sequence. It is described here as a synonym for the implementation-defined type T2.

basic_string::substr

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

The member function returns an object whose controlled sequence is a copy of up to n elements of the controlled sequence beginning at position pos.

basic_string::swap

```
void swap(basic_string& str);
```

The member function swaps the controlled sequences between *this and str. If get_allocator() == str.get_allocator(), it does so in constant time, it throws no exceptions, and it invalidates no references, pointers, or iterators that designate elements in the two controlled sequences. Otherwise, it performs a number of element assignments and constructor calls proportional to the number of elements in the two controlled sequences.

basic_string::traits_type

```
typedef T traits_type;
```

The type is a synonym for the template parameter T.

basic_string::value_type

```
typedef typename allocator_type::value_type  
value_type;
```

The type is a synonym for allocator_type::value_type.

char_traits

```
template<class E>  
    class char_traits {  
public:  
    typedef E char_type;  
    typedef T1 int_type;  
    typedef T2 pos_type;  
    typedef T3 off_type;  
    typedef T4 state_type;  
    static void assign(char_type& x, const char_type& y);  
    static char_type *assign(char_type *x, size_t n,  
                             char_type y);  
    static bool eq(const char_type& x,  
                   const char_type& y);  
    static bool lt(const char_type& x,  
                   const char_type& y);  
    static int compare(const char_type *x,  
                       const char_type *y, size_t n);  
    static size_t length(const char_type *x);
```

```

static char_type *copy(char_type *x,
    const char_type *y, size_t n);
static char_type *move(char_type *x,
    const char_type *y, size_t n);
static const char_type *find(const char_type *x,
    size_t n, const char_type& y);
static char_type to_char_type(const int_type& ch);
static int_type to_int_type(const char_type& c);
static bool eq_int_type(const int_type& ch1,
    const int_type& ch2);
static int_type eof();
static int_type not_eof(const int_type& ch);
};

```

The template class describes various **character traits** for type E. The template class `basic_string` (page 197) as well as several `iostreams` template classes, including `basic_ios` (page 88), use this information to manipulate elements of type E. Such an element type must not require explicit construction or destruction. It must supply a default constructor, a copy constructor, and an assignment operator, with the expected semantics. A bitwise copy must have the same effect as an assignment.

Not all parts of the Standard C++ Library rely completely upon the member functions of `char_traits<E>` to manipulate an element. Specifically, formatted input functions (page 107) and formatted output functions (page 170) make use of the following additional operations, also with the expected semantics:

- `operator==(E)` and `operator!=(E)` to compare elements
- `(char)ch` to convert an element `ch` to its corresponding single-byte character code, or `'\0'` if no such code exists
- `(E)c` to convert a `char` value `c` to its corresponding character code of type E

None of the member functions of class `char_traits` may throw exceptions.

char_traits::assign

```

static void assign(char_type& x, const char_type& y);
static char_type *assign(char_type *x, size_t n,
    char_type y);

```

The first static member function assigns `y` to `x`. The second static member function assigns `y` to each element `x[N]` for `N` in the range `[0, N)`.

char_traits::char_type

```

typedef E char_type;

```

The type is a synonym for the template parameter E.

char_traits::compare

```

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

```

The static member function compares the sequence of length `n` beginning at `x` to the sequence of the same length beginning at `y`. The function returns:

- a negative value if the first differing element in `x` (as determined by `eq` (page 212)) compares less than the corresponding element in `y` (as determined by `lt` (page 212))
- zero if the two compare equal element by element
- a positive value otherwise

char_traits::copy

```
static char_type *copy(char_type *x, const char_type *y,  
    size_t n);
```

The static member function copies the sequence of *n* elements beginning at *y* to the array beginning at *x*, then returns *x*. The source and destination must not overlap.

char_traits::eof

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

The static member function returns a value that represents end-of-file (such as EOF or WEOF). If the value is also representable as type *E*, it must correspond to no *valid* value of that type.

char_traits::eq

```
static bool eq(const char_type& x, const char_type& y);
```

The static member function returns true if *x* compares equal to *y*.

char_traits::eq_int_type

```
static bool eq_int_type(const int_type& ch1,  
    const int_type& ch2);
```

The static member function returns true if *ch1* == *ch2*.

char_traits::find

```
static const char_type *find(const char_type *x,  
    size_t n, const char_type& y);
```

The static member function determines the lowest *N* in the range $[0, n)$ for which *eq*(*x*[*N*], *y*) is true. If successful, it returns *x* + *N*. Otherwise, it returns a null pointer.

char_traits::int_type

```
typedef T1 int_type;
```

The type is (typically) an integer type *T1* that describes an object that can represent any element of the controlled sequence as well as the value returned by *eof*(). It must be possible to type cast a value of type *E* to *int_type* then back to *E* without altering the original value.

char_traits::length

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

The static member function returns the number of elements *N* in the sequence beginning at *x* up to but not including the element *x*[*N*] which compares equal to *char_type*().

char_traits::lt

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

The static member function returns true if *x* compares less than *y*.

char_traits::move

```
static char_type *move(char_type *x, const char_type *y,  
    size_t n);
```

The static member function copies the sequence of *n* elements beginning at *y* to the array beginning at *x*, then returns *x*. The source and destination may overlap.

char_traits::not_eof

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

If `!eq_int_type (page 212) (eof (page 212) (), ch)`, the static member function returns `ch`. Otherwise, it returns a value other than `eof()`.

char_traits::off_type

```
typedef T3 off_type;
```

The type is a signed integer type *T3* that describes an object that can store a byte offset involved in various stream positioning operations. It is typically a synonym for `streamoff` (page 101), but in any case it has essentially the same properties as that type.

char_traits::pos_type

```
typedef T2 pos_type;
```

The type is an opaque type *T2* that describes an object that can store all the information needed to restore an arbitrary file-position indicator (page 19) within a stream. It is typically a synonym for `streampos` (page 101), but in any case it has essentially the same properties as that type.

char_traits::state_type

```
typedef T4 state_type;
```

The type is an opaque type *T4* that describes an object that can represent a conversion state (page 12). It is typically a synonym for `mbstate_t`, but in any case it has essentially the same properties as that type.

char_traits::to_char_type

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

The static member function returns `ch`, represented as type *E*. A value of `ch` that cannot be so represented yields an unspecified result.

char_traits::to_int_type

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

The static member function returns `ch`, represented as type `int_type`. It should always be true that `to_char_type(to_int_type(c)) == c` for any value of `c`.

char_traits<char>

```
template<>
class char_traits<char>;
```

The class is an explicit specialization of template class `char_traits` (page 210) for elements of type *char*, (so that it can take advantage of library functions that manipulate objects of this type).

char_traits<wchar_t>

```
template<>
class char_traits<wchar_t>;
```

The class is an explicit specialization of template class `char_traits` (page 210) for elements of type `wchar_t` (so that it can take advantage of library functions that manipulate objects of this type).

getline

```
template<class E, class T, class A>
    basic_istream<E, T>& getline(
        basic_istream<E, T>& is,
        basic_string<E, T, A>& str);
template<class E, class T, class A>
    basic_istream<E, T>& getline(
        basic_istream<E, T>& is,
        basic_string<E, T, A>& str,
        E delim);
```

The first function returns `getline(is, str, is.widen('\n'))`.

The second function replaces the sequence controlled by `str` with a sequence of elements extracted from the stream `is`. In order of testing, extraction stops:

1. at end of file
2. after the function extracts an element that compares equal to `delim`, in which case the element is neither put back nor appended to the controlled sequence
3. after the function extracts `str.max_size()` elements, in which case the function calls `setstate(ios_base::failbit)`.

If the function extracts no elements, it calls `setstate(failbit)`. In any case, it returns `*this`.

operator+

```
template<class E, class T, class A>
    basic_string<E, T, A> operator+(
        const basic_string<E, T, A>& lhs,
        const basic_string<E, T, A>& rhs);
template<class E, class T, class A>
    basic_string<E, T, A> operator+(
        const basic_string<E, T, A>& lhs,
        const E *rhs);
template<class E, class T, class A>
    basic_string<E, T, A> operator+(
        const basic_string<E, T, A>& lhs,
        E rhs);
template<class E, class T, class A>
    basic_string<E, T, A> operator+(
        const E *lhs,
        const basic_string<E, T, A>& rhs);
template<class E, class T, class A>
    basic_string<E, T, A> operator+(
        E lhs,
        const basic_string<E, T, A>& rhs);
```

The functions each overload `operator+` to concatenate two objects of template class `basic_string` (page 197). All effectively return `basic_string<E, T, A>(lhs).append(rhs)`.

operator!=

```
template<class E, class T, class A>
    bool operator!=(
        const basic_string<E, T, A>& lhs,
        const basic_string<E, T, A>& rhs);
```

```

template<class E, class T, class A>
    bool operator!=(
        const basic_string<E, T, A>& lhs,
        const E *rhs);
template<class E, class T, class A>
    bool operator!=(
        const E *lhs,
        const basic_string<E, T, A>& rhs);

```

The template functions each overload operator!= to compare two objects of template class basic_string (page 197). All effectively return basic_string<E, T, A>(lhs).compare(rhs) != 0.

operator==

```

template<class E, class T, class A>
    bool operator==(
        const basic_string<E, T, A>& lhs,
        const basic_string<E, T, A>& rhs);
template<class E, class T, class A>
    bool operator==(
        const basic_string<E, T, A>& lhs,
        const E *rhs);
template<class E, class T, class A>
    bool operator==(
        const E *lhs,
        const basic_string<E, T, A>& rhs);

```

The template functions each overload operator== to compare two objects of template class basic_string (page 197). All effectively return basic_string<E, T, A>(lhs).compare(rhs) == 0.

operator<

```

template<class E, class T, class A>
    bool operator<(
        const basic_string<E, T, A>& lhs,
        const basic_string<E, T, A>& rhs);
template<class E, class T, class A>
    bool operator<(
        const basic_string<E, T, A>& lhs,
        const E *rhs);
template<class E, class T, class A>
    bool operator<(
        const E *lhs,
        const basic_string<E, T, A>& rhs);

```

The template functions each overload operator< to compare two objects of template class basic_string (page 197). All effectively return basic_string<E, T, A>(lhs).compare(rhs) < 0.

operator<<

```

template<class E, class T, class A>
    basic_ostream<E, T>& operator<<(
        basic_ostream<E, T>& os,
        const basic_string<E, T, A>& str);

```

The template function overloads operator<< to insert an object str of template class basic_string (page 197) into the stream os. The function effectively returns os.write(str.c_str(), str.size()).

operator<=

```
template<class E, class T, class A>
    bool operator<=(
        const basic_string<E, T, A>& lhs,
        const basic_string<E, T, A>& rhs);
template<class E, class T, class A>
    bool operator<=(
        const basic_string<E, T, A>& lhs,
        const E *rhs);
template<class E, class T, class A>
    bool operator<=(
        const E *lhs,
        const basic_string<E, T, A>& rhs);
```

The template functions each overload operator<= to compare two objects of template class basic_string (page 197). All effectively return basic_string<E, T, A>(lhs).compare(rhs) <= 0.

operator>

```
template<class E, class T, class A>
    bool operator>(
        const basic_string<E, T, A>& lhs,
        const basic_string<E, T, A>& rhs);
template<class E, class T, class A>
    bool operator>(
        const basic_string<E, T, A>& lhs,
        const E *rhs);
template<class E, class T, class A>
    bool operator>(
        const E *lhs,
        const basic_string<E, T, A>& rhs);
```

The template functions each overload operator> to compare two objects of template class basic_string (page 197). All effectively return basic_string<E, T, A>(lhs).compare(rhs) > 0.

operator>=

```
template<class E, class T, class A>
    bool operator>=(
        const basic_string<E, T, A>& lhs,
        const basic_string<E, T, A>& rhs);
template<class E, class T, class A>
    bool operator>=(
        const basic_string<E, T, A>& lhs,
        const E *rhs);
template<class E, class T, class A>
    bool operator>=(
        const E *lhs,
        const basic_string<E, T, A>& rhs);
```

The template functions each overload operator>= to compare two objects of template class basic_string (page 197). All effectively return basic_string<E, T, A>(lhs).compare(rhs) >= 0.

operator>>

```
template<class E, class T, class A>
    basic_istream<E, T>& operator>>(
        basic_istream<E, T>& is,
        const basic_string<E, T, A>& str);
```


The template function overloads operator>> to replace the sequence controlled by `str` with a sequence of elements extracted from the stream `is`. Extraction stops:

- at end of file
- after the function extracts `is.width()` elements, if that value is nonzero
- after the function extracts `is.max_size()` elements
- after the function extracts an element `c` for which `use_facet< ctype<E> >(getloc()). is(ctype<E>::space, c)` is true, in which case the character is put back

If the function extracts no elements, it calls `setstate(ios_base::failbit)`. In any case, it calls `is.width(0)` and returns `*this`.

string

```
typedef basic_string<char> string;
```

The type describes a specialization of template class `basic_string` specialized for elements of type `char`.

swap

```
template<class T, class A>  
void swap(  
    basic_string<E, T, A>& lhs,  
    basic_string<E, T, A>& rhs);
```

The template function executes `lhs.swap(rhs)`.

wstring

```
typedef basic_string<wchar_t> wstring;
```

The type describes a specialization of template class `basic_string` for elements of type `wchar_t`.

<sstream>

```
namespace std {  
class stringstreambuf;  
class istringstream;  
class ostringstream;  
class stringstream;  
};
```

Include the `iostreams` (page 7) standard header `<sstream>` to define several classes that support `iostreams` operations on sequences stored in an allocated array of `char` object. Such sequences are easily converted to and from C strings.

stringstreambuf

```
class stringstreambuf : public streambuf {  
public:  
    explicit stringstreambuf(streamsize n = 0);  
    stringstreambuf(void (*palloc)(size_t),  
        void (*pfree)(void *));  
    stringstreambuf(char *gp, streamsize n,  
        char *pp = 0);  
    stringstreambuf(signed char *gp, streamsize n,  
        signed char *pp = 0);  
    stringstreambuf(unsigned char *gp, streamsize n,
```

```

        unsigned char *pp = 0);
strstreambuf(const char *gp, streamsize n);
strstreambuf(const signed char *gp, streamsize n);
strstreambuf(const unsigned char *gp, streamsize n);
void freeze(bool frz = true);
char *str();
streamsize pcount();
protected:
    virtual streampos seekoff(streamoff off,
        ios_base::seekdir way,
        ios_base::openmode which =
            ios_base::in | ios_base::out);
    virtual streampos seekpos(streampos sp,
        ios_base::openmode which =
            ios_base::in | ios_base::out);
    virtual int underflow();
    virtual int pbackfail(int c = EOF);
    virtual int overflow(int c = EOF);
};

```

The class describes a **stream buffer (page 187)** that controls the transmission of elements to and from a sequence of elements stored in a *char* array object. Depending on how it is constructed, the object can be allocated, extended, and freed as necessary to accommodate changes in the sequence.

An object of class `strstreambuf` stores several bits of mode information as its **strstreambuf mode**. These bits indicate whether the controlled sequence:

- has been **allocated**, and hence needs to be freed eventually
- is **modifiable**
- is **extendable** by reallocating storage
- has been **frozen** and hence needs to be unfrozen before the object is destroyed, or freed (if allocated) by an agency other than the object

A controlled sequence that is frozen cannot be modified or extended, regardless of the state of these separate mode bits.

The object also stores pointers to two functions that control **strstreambuf allocation**. If these are null pointers, the object devises its own method of allocating and freeing storage for the controlled sequence.

strstreambuf::freeze

```
void freeze(bool frz = true);
```

If `frz` is true, the function alters the stored `strstreambuf` mode (page 218) to make the controlled sequence frozen. Otherwise, it makes the controlled sequence not frozen.

strstreambuf::pcount

```
streamsize pcount();
```

The member function returns a count of the number of elements written to the controlled sequence. Specifically, if `pptr()` is a null pointer, the function returns zero. Otherwise, it returns `pptr() - pbase()`.

strstreambuf::overflow

```
virtual int overflow(int c = EOF);
```

If `c != EOF`, the protected virtual member function endeavors to insert the element `(char)c` into the output buffer (page 187). It can do so in various ways:

- If a write position (page 188) is available, it can store the element into the write position and increment the next pointer for the output buffer.
- If the stored `strstreambuf` mode (page 218) says the controlled sequence is modifiable, extendable, and not frozen, the function can make a write position available by allocating new for the output buffer. (Extending the output buffer this way also extends any associated input buffer (page 187).)

If the function cannot succeed, it returns `EOF`. Otherwise, if `c == EOF` it returns some value other than `EOF`. Otherwise, it returns `c`.

`strstreambuf::pbackfail`

```
virtual int pbackfail(int c = EOF);
```

The protected virtual member function endeavors to put back an element into the input buffer (page 187), then make it the current element (pointed to by the next pointer).

If `c = EOF`, the element to push back is effectively the one already in the stream before the current element. Otherwise, that element is replaced by `x = (char)c`. The function can put back an element in various ways:

- If a putback position (page 188) is available, and the element stored there compares equal to `x`, it can simply decrement the next pointer for the input buffer.
- If a putback position is available, and if the `strstreambuf` mode (page 218) says the controlled sequence is modifiable, the function can store `x` into the putback position and decrement the next pointer for the input buffer.

If the function cannot succeed, it returns `EOF`. Otherwise, if `c == EOF` it returns some value other than `EOF`. Otherwise, it returns `c`.

`strstreambuf::seekoff`

```
virtual streampos seekoff(streamoff off,  
    ios_base::seekdir way,  
    ios_base::openmode which =  
        ios_base::in | ios_base::out);
```

The protected virtual member function endeavors to alter the current positions for the controlled streams. For an object of class `strstreambuf`, a stream position consists purely of a stream offset. Offset zero designates the first element of the controlled sequence.

The new position is determined as follows:

- If `way == ios_base::beg`, the new position is the beginning of the stream plus `off`.
- If `way == ios_base::cur`, the new position is the current stream position plus `off`.
- If `way == ios_base::end`, the new position is the end of the stream plus `off`.

If `which & ios_base::in` is nonzero and the input buffer exist, the function alters the next position to read in the input buffer (page 187). If `which & ios_base::out` is also nonzero, `way != ios_base::cur`, and the output buffer exists, the function also sets the next position to write to match the next position to read.

Otherwise, if which & ios_base::out is nonzero and the output buffer exists, the function alters the next position to write in the output buffer (page 187). Otherwise, the positioning operation fails. For a positioning operation to succeed, the resulting stream position must lie within the controlled sequence.

If the function succeeds in altering the stream position(s), it returns the resultant stream position. Otherwise, it fails and returns an invalid stream position.

strstreambuf::seekpos

```
virtual streampos seekpos(streampos sp,
    ios_base::openmode which =
        ios_base::in | ios_base::out);
```

The protected virtual member function endeavors to alter the current positions for the controlled streams. For an object of class strstreambuf, a stream position consists purely of a stream offset. Offset zero designates the first element of the controlled sequence. The new position is determined by sp.

If which & ios_base::in is nonzero and the input buffer exists, the function alters the next position to read in the input buffer (page 187). (If which & ios_base::out is nonzero and the output buffer exists, the function also sets the next position to write to match the next position to read.) Otherwise, if which & ios_base::out is nonzero and the output buffer exists, the function alters the next position to write in the output buffer (page 187). Otherwise, the positioning operation fails. For a positioning operation to succeed, the resulting stream position must lie within the controlled sequence.

If the function succeeds in altering the stream position(s), it returns the resultant stream position. Otherwise, it fails and returns an invalid stream position.

strstreambuf::str

```
char *str();
```

The member function calls freeze(), then returns a pointer to the beginning of the controlled sequence. (Note that no terminating null element exists, unless you insert one explicitly.)

strstreambuf::strstreambuf

```
explicit strstreambuf(streamsize n = 0);
strstreambuf(void (*palloc)(size_t),
    void (*pfree)(void *));
strstreambuf(char *gp, streamsize n,
    char *pp = 0);
strstreambuf(signed char *gp, streamsize n,
    signed char *pp = 0);
strstreambuf(unsigned char *gp, streamsize n,
    unsigned char *pp = 0);
strstreambuf(const char *gp, streamsize n);
strstreambuf(const signed char *gp, streamsize n);
strstreambuf(const unsigned char *gp, streamsize n);
```

The first constructor stores a null pointer in all the pointers controlling the input buffer (page 187), the output buffer (page 187), and strstreambuf allocation (page 218). It sets the stored strstreambuf mode (page 218) to make the controlled sequence modifiable and extendable.

The second constructor behaves much as the first, except that it stores `palloc` as the pointer to the function to call to allocate storage, and `pfree` as the pointer to the function to call to free that storage.

The three constructors:

```
strstreambuf(char *gp, streamsize n,  
              char *pp = 0);  
strstreambuf(signed char *gp, streamsize n,  
              signed char *pp = 0);  
strstreambuf(unsigned char *gp, streamsize n,  
              unsigned char *pp = 0);
```

also behave much as the first, except that `gp` designates the array object used to hold the controlled sequence. (Hence, it must not be a null pointer.) The number of elements `N` in the array is determined as follows:

- If $(n > 0)$, then `N` is `n`.
- If $(n == 0)$, then `N` is `strlen((const char *)gp)`.
- If $(n < 0)$, then `N` is `INT_MAX`.

If `pp` is a null pointer, the function establishes just an input buffer, by executing:

```
setg(gp, gp, gp + N);
```

Otherwise, it establishes both input and output buffers, by executing:

```
setg(gp, gp, pp);  
setp(pp, gp + N);
```

In this case, `pp` must be in the interval `[gp, gp + N]`.

Finally, the three constructors:

```
strstreambuf(const char *gp, streamsize n);  
strstreambuf(const signed char *gp, streamsize n);  
strstreambuf(const unsigned char *gp, streamsize n);
```

all behave the same as:

```
streambuf((char *)gp, n);
```

except that the stored mode makes the controlled sequence neither modifiable nor extendable.

strstreambuf::underflow

```
virtual int underflow();
```

The protected virtual member function endeavors to extract the current element `c` from the input buffer (page 187), then advance the current stream position, and return the element as `(int)(unsigned char)c`. It can do so in only one way: If a read position (page 188) is available, it takes `c` as the element stored in the read position and advances the next pointer for the input buffer.

If the function cannot succeed, it returns EOF. Otherwise, it returns the current element in the input stream, converted as described above.

istream

```
class istream : public istream {  
public:  
    explicit istream(const char *s);  
    explicit istream(char *s);
```

```

istream(const char *s, streamsize n);
istream(char *s, streamsize n);
strstreambuf *rdbuf() const;
char *str();
};

```

The class describes an object that controls extraction of elements and encoded objects from a stream buffer (page 187) of class `strstreambuf` (page 217). The object stores an object of class `strstreambuf`.

istream::istream

```

explicit istream(const char *s);
explicit istream(char *s);
istream(const char *s, streamsize n);
istream(char *s, streamsize n);

```

All the constructors initialize the base class by calling `istream(sb)`, where `sb` is the stored object of class `strstreambuf`. The first two constructors also initialize `sb` by calling `strstreambuf((const char *)s, 0)`. The remaining two constructors instead call `strstreambuf((const char *)s, n)`.

istream::rdbuf

```
strstreambuf *rdbuf() const
```

The member function returns the address of the stored stream buffer, of type pointer to `strstreambuf` (page 217).

istream::str

```
char *str();
```

The member function returns `rdbuf()->str()`.

ostream

```

class ostream : public ostream {
public:
    ostream();
    ostream(char *s, streamsize n,
            ios_base::openmode mode = ios_base::out);
    strstreambuf *rdbuf() const;
    void freeze(bool frz = true);
    char *str();
    streamsize pcount() const;
};

```

The class describes an object that controls insertion of elements and encoded objects into a stream buffer of class `strstreambuf`. The object stores an object of class `strstreambuf`.

ostream::freeze

```
void freeze(bool frz = true)
```

The member function calls `rdbuf()->freeze(frz)`.

ostream::ostream

```

ostream();
ostream(char *s, streamsize n,
        ios_base::openmode mode = ios_base::out);

```

Both constructors initialize the base class by calling `ostream(sb)`, where `sb` is the stored object of class `strstreambuf`. The first constructor also initializes `sb` by calling `strstreambuf()`. The second constructor initializes the base class one of two ways:

- If `mode & ios_base::app == 0`, then `s` must designate the first element of an array of `n` elements, and the constructor calls `strstreambuf(s, n, s)`.
- Otherwise, `s` must designate the first element of an array of `n` elements that contains a C string whose first element is designated by `s`, and the constructor calls `strstreambuf(s, n, s + strlen(s))`.

ostream::pcount

`streamsize pcount() const;`

The member function returns `rdbuf()-> pcount()`.

ostream::rdbuf

`strstreambuf *rdbuf() const`

The member function returns the address of the stored stream buffer, of type pointer to `strstreambuf` (page 217).

ostream::str

`char *str();`

The member function returns `rdbuf()-> str()`.

strstream

```
class strstream : public ostream {
public:
    strstream();
    strstream(char *s, streamsize n,
               ios_base::openmode mode =
                   ios_base::in | ios_base::out);
    strstreambuf *rdbuf() const;
    void freeze(bool frz = true);
    char *str();
    streamsize pcount() const;
};
```

The class describes an object that controls insertion and extraction of elements and encoded objects using a stream buffer (page 187) of class `strstreambuf` (page 217). The object stores an object of class `strstreambuf`.

strstream::freeze

`void freeze(bool frz = true)`

The member function calls `rdbuf()-> freeze(zfrz)`.

strstream::pcount

`streamsize pcount() const;`

The member function returns `rdbuf()-> pcount()`.

strstream::strstream

```
strstream();
strstream(char *s, streamsize n,
           ios_base::openmode mode =
               ios_base::in | ios_base::out);
```

Both constructors initialize the base class by calling `streambuf(sb)`, where `sb` is the stored object of class `strstreambuf` (page 217). The first constructor also initializes `sb` by calling `strstreambuf()`. The second constructor initializes the base class one of two ways:

- If `mode & ios_base::app == 0`, then `s` must designate the first element of an array of `n` elements, and the constructor calls `strstreambuf(s, n, s)`.
- Otherwise, `s` must designate the first element of an array of `n` elements that contains a C string whose first element is designated by `s`, and the constructor calls `strstreambuf(s, n, s + strlen(s))`.

strstream::rdbuf

`strstreambuf *rdbuf() const`

The member function returns the address of the stored stream buffer, of type pointer to `strstreambuf` (page 217).

strstream::str

`char *str();`

The member function returns `rdbuf()->str()`.

<typeinfo>

```
namespace std {  
class type_info;  
class bad_cast;  
class bad_typeid;  
};
```

Include the standard header **<typeinfo>** to define several types associated with the type-identification operator **typeid**, which yields information about both static and dynamic types.

bad_cast

```
class bad_cast : public exception {  
};
```

The class describes an exception thrown to indicate that a **dynamic cast** expression, of the form:

```
dynamic_cast<type>(expression)
```

generated a null pointer to initialize a reference. The value returned by `what()` is an implementation-defined C string. None of the member functions throw any exceptions.

bad_typeid

```
class bad_typeid : public exception {  
};
```

The class describes an exception thrown to indicate that a **typeid** (page 224) operator encountered a null pointer. The value returned by `what()` is an implementation-defined C string. None of the member functions throw any exceptions.

type_info

```
class type_info {
public:
    virtual ~type_info();
    bool operator==(const type_info& rhs) const;
    bool operator!=(const type_info& rhs) const;
    bool before(const type_info& rhs) const;
    const char *name() const;
private:
    type_info(const type_info& rhs);
    type_info& operator=(const type_info& rhs);
};
```

The class describes type information generated within the program by the implementation. Objects of this class effectively store a pointer to a **name** for the type, and an encoded value suitable for comparing two types for equality or **collating order**. The names, encoded values, and collating order for types are all unspecified and may differ between program executions.

An expression of the form typeid x is the *only* way to construct a (temporary) typeinfo object. The class has only a private copy constructor. Since the assignment operator is also private, you cannot copy or assign objects of class typeinfo either.

type_info::operator!=

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

The function returns !(*this == rhs).

type_info::operator==

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

The function returns a nonzero value if *this and rhs represent the same type.

type_info::before

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

The function returns a nonzero value if *this precedes rhs in the collating order for types.

type_info::name

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

The function returns a C string which specifies the name of the type.

<valarray>

gslice (page 230) · gslice_array (page 231) · indirect_array (page 232) · mask_array (page 233) · slice (page 239) · slice_array (page 239) · valarray (page 240) · valarray<bool> (page 247)

abs (page 229) · acos (page 229) · asin (page 229) · atan (page 230) · atan2 (page 230) · cos (page 230) · cosh (page 230) · exp (page 230) · log (page 233) · log10 (page 233) · operator!= (page 234) · operator% (page 234) · operator& (page 234) · operator&& (page 234) · operator> (page 235) · operator>> (page 235) · operator>= (page 235) · operator< (page 235) · operator<< (page 236) · operator<= (page 236) · operator* (page 236) · operator+ (page 236) · operator- (page 237) · operator/ (page

237) · **operator==** (page 237) · **operator^** (page 237) · **operator|** (page 238) · **operator||** (page 238) · **pow** (page 238) · **sin** (page 238) · **sinh** (page 239) · **sqrt** (page 240) · **tan** (page 240) · **tanh** (page 240)

```
namespace std {
class slice;
class gslice;

    // TEMPLATE CLASSES
template<class T>
    class valarray;
template<class T>
    class slice_array;
template<class T>
    class gslice_array;
template<class T>
    class mask_array;
template<class T>
    class indirect_array;

    // TEMPLATE FUNCTIONS
template<class T>
    valarray<T> operator*(const valarray<T>& x,
        const valarray<T>& y);
template<class T>
    valarray<T> operator*(const valarray<T> x,
        const T& y);
template<class T>
    valarray<T> operator*(const T& x,
        const valarray<T>& y);
template<class T>
    valarray<T> operator/(const valarray<T>& x,
        const valarray<T>& y);
template<class T>
    valarray<T> operator/(const valarray<T> x,
        const T& y);
template<class T>
    valarray<T> operator/(const T& x,
        const valarray<T>& y);
template<class T>
    valarray<T> operator%(const valarray<T>& x,
        const valarray<T>& y);
template<class T>
    valarray<T> operator%(const valarray<T> x,
        const T& y);
template<class T>
    valarray<T> operator%(const T& x,
        const valarray<T>& y);
template<class T>
    valarray<T> operator+(const valarray<T>& x,
        const valarray<T>& y);
template<class T>
    valarray<T> operator+(const valarray<T> x,
        const T& y);
template<class T>
    valarray<T> operator+(const T& x,
        const valarray<T>& y);
template<class T>
    valarray<T> operator-(const valarray<T>& x,
        const valarray<T>& y);
template<class T>
    valarray<T> operator-(const valarray<T> x,
        const T& y);
template<class T>
    valarray<T> operator-(const T& x,
        const valarray<T>& y);
template<class T>
```

```

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

```

```

        valarray<bool> operator==(const valarray<T> x,
            const T& y);
template<class T>
    valarray<bool> operator==(const T& x,
        const valarray<T>& y);
template<class T>
    valarray<bool> operator!=(const valarray<T>& x,
        const valarray<T> y);
template<class T>
    valarray<bool> operator!=(const valarray<T> x,
        const T& y);
template<class T>
    valarray<bool> operator!=(const T& x,
        const valarray<T> y);
template<class T>
    valarray<bool> operator<(const valarray<T>& x,
        const valarray<T> y);
template<class T>
    valarray<bool> operator<(const valarray<T> x,
        const T& y);
template<class T>
    valarray<bool> operator<(const T& x,
        const valarray<T> y);
template<class T>
    valarray<bool> operator>=(const valarray<T>& x,
        const valarray<T> y);
template<class T>
    valarray<bool> operator>=(const valarray<T> x,
        const T& y);
template<class T>
    valarray<bool> operator>=(const T& x,
        const valarray<T> y);
template<class T>
    valarray<bool> operator>(const valarray<T>& x,
        const valarray<T> y);
template<class T>
    valarray<bool> operator>(const valarray<T> x,
        const T& y);
template<class T>
    valarray<bool> operator>(const T& x,
        const valarray<T> y);
template<class T>
    valarray<bool> operator<=(const valarray<T>& x,
        const valarray<T> y);
template<class T>
    valarray<bool> operator<=(const valarray<T> x,
        const T& y);
template<class T>
    valarray<bool> operator<=(const T& x,
        const valarray<T> y);
template<class T>
    valarray<T> abs(const valarray<T>& x);
template<class T>
    valarray<T> acos(const valarray<T>& x);
template<class T>
    valarray<T> asin(const valarray<T>& x);
template<class T>
    valarray<T> atan(const valarray<T>& x);
template<class T>
    valarray<T> atan2(const valarray<T>& x,
        const valarray<T> y);
template<class T>
    valarray<T> atan2(const valarray<T> x, const T& y);
template<class T>
    valarray<T> atan2(const T& x, const valarray<T>& y);
template<class T>
    valarray<T> cos(const valarray<T>& x);

```

```

template<class T>
    valarray<T> cosh(const valarray<T>& x);
template<class T>
    valarray<T> exp(const valarray<T>& x);
template<class T>
    valarray<T> log(const valarray<T>& x);
template<class T>
    valarray<T> log10(const valarray<T>& x);
template<class T>
    valarray<T> pow(const valarray<T>& x,
        const valarray<T>& y);
template<class T>
    valarray<T> pow(const valarray<T> x, const T& y);
template<class T>
    valarray<T> pow(const T& x, const valarray<T>& y);
template<class T>
    valarray<T> sin(const valarray<T>& x);
template<class T>
    valarray<T> sinh(const valarray<T>& x);
template<class T>
    valarray<T> sqrt(const valarray<T>& x);
template<class T>
    valarray<T> tan(const valarray<T>& x);
template<class T>
    valarray<T> tanh(const valarray<T>& x);
};

```

Include the standard header **<valarray>** to define the template class `valarray` (page 240) and numerous supporting template classes and functions. These template classes and functions are permitted unusual latitude, in the interest of improved performance. Specifically, any function returning `valarray<T>` may return an object of some other type `T'`. In that case, any function that accepts one or more arguments of type `valarray<T>` must have overloads that accept arbitrary combinations of those arguments, each replaced with an argument of type `T'`. (Put simply, the only way you can detect such a substitution is to go looking for it.)

abs

```

template<class T>
    valarray<T> abs(const valarray<T>& x);

```

The template function returns an object of class `valarray<T>`, each of whose elements `I` is the absolute value of `x[I]`.

acos

```

template<class T>
    valarray<T> acos(const valarray<T>& x);

```

The template function returns an object of class `valarray<T>`, each of whose elements `I` is the arccosine of `x[I]`.

asin

```

template<class T>
    valarray<T> asin(const valarray<T>& x);

```

The template function returns an object of class `valarray<T>`, each of whose elements `I` is the arcsine of `x[I]`.

atan

```
template<class T>
    valarray<T> atan(const valarray<T>& x);
```

The template function returns an object of class `valarray<T>`, each of whose elements `I` is the arctangent of `x[I]`.

atan2

```
template<class T>
    valarray<T> atan2(const valarray<T>& x,
                      const valarray<T>& y);
template<class T>
    valarray<T> atan2(const valarray<T> x, const T& y);
template<class T>
    valarray<T> atan2(const T& x, const valarray<T>& y);
```

The first template function returns an object of class `valarray<T>`, each of whose elements `I` is the arctangent of `x[I] / y[I]`. The second template function stores in element `I` the arctangent of `x[I] / y`. The third template function stores in element `I` the arctangent of `x / y[I]`.

cos

```
template<class T>
    valarray<T> cos(const valarray<T>& x);
```

The template function returns an object of class `valarray<T>`, each of whose elements `I` is the cosine of `x[I]`.

cosh

```
template<class T>
    valarray<T> cosh(const valarray<T>& x);
```

The template function returns an object of class `valarray<T>`, each of whose elements `I` is the hyperbolic cosine of `x[I]`.

exp

```
template<class T>
    valarray<T> exp(const valarray<T>& x);
```

The template function returns an object of class `valarray<T>`, each of whose elements `I` is the exponential of `x[I]`.

gslice

```
class gslice {
public:
    gslice();
    gslice(size_t st,
          const valarray<size_t> len,
          const valarray<size_t> str);
    size_t start() const;
    const valarray<size_t> size() const;
    const valarray<size_t> stride() const;
};
```

The class stores the parameters that characterize a `gslice_array` (page 231) when an object of class `gslice` appears as a subscript for an object of class `valarray<T>`. The stored values include:

- a **starting index**
- a **length vector** of class `valarray<size_t>`
- a **stride vector** of class `valarray<size_t>`

The two vectors must have the same length.

gslice::gslice

```
gslice();
gslice(size_t st,
        const valarray<size_t> len,
        const valarray<size_t> str);
```

The default constructor stores zero for the starting index, and zero-length vectors for the length and stride vectors. The second constructor stores `st` for the starting index, `len` for the length vector, and `str` for the stride vector.

gslice::size

```
const valarray<size_t> size() const;
```

The member function returns the stored length vector.

gslice::start

```
size_t start() const;
```

The member function returns the stored starting index.

gslice::stride

```
const valarray<size_t> stride() const;
```

The member function returns the stored stride vector.

gslice_array

```
template<class T>
class gslice_array {
public:
    typedef T value_type;
    void operator=(const valarray<T> x) const;
    void operator=(const T& x);
    void operator*=(const valarray<T> x) const;
    void operator/=(const valarray<T> x) const;
    void operator%=(const valarray<T> x) const;
    void operator+=(const valarray<T> x) const;
    void operator-= (const valarray<T> x) const;
    void operator^=(const valarray<T> x) const;
    void operator&=(const valarray<T> x) const;
    void operator|=(const valarray<T> x) const;
    void operator<<=(const valarray<T> x) const;
    void operator>>=(const valarray<T> x) const;
private:
    void gslice_array(); // not defined
    void gslice_array(
        const gslice_array&); // not defined
    gslice_array& operator=(
        const gslice_array&); // not defined
};
```

The class describes an object that stores a reference to an object `x` of class `valarray<T>`, along with an object `gs` of class `gslice` which describes the sequence of elements to select from the `valarray<T>` object.

You construct a `gslice_array<T>` object only by writing an expression of the form `x[gs]`. The member functions of class `gslice_array` then behave like the corresponding function signatures defined for `valarray<T>`, except that only the sequence of selected elements is affected.

The sequence is determined as follows. For a length vector `gs.size()` of length `N`, construct the index vector `valarray<size_t> idx(0, N)`. This designates the initial element of the sequence, whose index `k` within `x` is given by the mapping:

```
k = start;
for (size_t i = 0; i < gs.size()[i]; ++i)
    k += idx[i] * gs.stride()[i];
```

The successor to an index vector value is given by:

```
for (size_t i = N; 0 < i--; )
    if (++idx[i] < gs.size()[i])
        break;
    else
        idx[i] = 0;
```

For example:

```
const size_t lv[] = {2, 3};
const size_t dv[] = {7, 2};
const valarray<size_t> len(lv, 2), str(dv, 2);
// x[gslice(3, len, str)] selects elements with
// indices 3, 5, 7, 10, 12, 14
```

indirect_array

```
template<class T>
class indirect_array {
public:
    typedef T value_type;
    void operator=(const valarray<T> x) const;
    void operator=(const T& x);
    void operator*=(const valarray<T> x) const;
    void operator/=(const valarray<T> x) const;
    void operator%=(const valarray<T> x) const;
    void operator+=(const valarray<T> x) const;
    void operator-=(const valarray<T> x) const;
    void operator^=(const valarray<T> x) const;
    void operator&=(const valarray<T> x) const;
    void operator|=(const valarray<T> x) const;
    void operator<<=(const valarray<T> x) const;
    void operator>>=(const valarray<T> x) const;
private:
private:
    void indirect_array(); // not defined
    void indirect_array(
        const indirect_array&); // not defined
    indirect_array& operator=(
        const indirect_array&); // not defined
};
```

The class describes an object that stores a reference to an object `x` of class `valarray<T>`, along with an object `xa` of class `valarray<size_t>` which describes the sequence of elements to select from the `valarray<T>` object.

You construct an `indirect_array<T>` object only by writing an expression of the form `x[xa]`. The member functions of class `indirect_array` then behave like the corresponding function signatures defined for `valarray<T>`, except that only the sequence of selected elements is affected.

The sequence consists of `xa.size()` elements, where element `i` becomes the index `xa[i]` within `x`. For example:

```
const size_t vi[] = {7, 5, 2, 3, 8};
// x[valarray<size_t>(vi, 5)] selects elements with
// indices 7, 5, 2, 3, 8
```

log

```
template<class T>
    valarray<T> log(const valarray<T>& x);
```

The template function returns an object of class `valarray<T>`, each of whose elements `I` is the natural logarithm of `x[I]`.

log10

```
template<class T>
    valarray<T> log10(const valarray<T>& x);
```

The template function returns an object of class `valarray<T>`, each of whose elements `I` is the base-10 logarithm of `x[I]`.

mask_array

```
template<class T>
    class mask_array {
public:
    typedef T value_type;
    void operator=(const valarray<T> x) const;
    void operator=(const T& x);
    void operator*=(const valarray<T> x) const;
    void operator/=(const valarray<T> x) const;
    void operator%=(const valarray<T> x) const;
    void operator+=(const valarray<T> x) const;
    void operator-= (const valarray<T> x) const;
    void operator^=(const valarray<T> x) const;
    void operator&=(const valarray<T> x) const;
    void operator|=(const valarray<T> x) const;
    void operator<<=(const valarray<T> x) const;
    void operator>>=(const valarray<T> x) const;
private:
    void mask_array(); // not defined
    void mask_array(
        const mask_array&); // not defined
    gslice_array& operator=(
        const mask_array&); // not defined
    };
```

The class describes an object that stores a reference to an object `x` of class `valarray<T>`, along with an object `ba` of class `valarray<bool>` which describes the sequence of elements to select from the `valarray<T>` object.

You construct a `mask_array<T>` object only by writing an expression of the form `x[xa]`. The member functions of class `mask_array` then behave like the corresponding function signatures defined for `valarray<T>`, except that only the sequence of selected elements is affected.

The sequence consists of at most `ba.size()` elements. An element `j` is included only if `ba[j]` is true. Thus, there are as many elements in the sequence as there are true elements in `ba`. If `i` is the index of the lowest true element in `ba`, then `x[i]` is element zero in the selected sequence. For example:

```
const bool vb[] = {false, false, true, true, false, true};
// x[valarray<bool>(vb, 56] selects elements with
// indices 2, 3, 5
```

operator!=

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

The first template operator returns an object of class `valarray<bool>` (page 247), each of whose elements `I` is `x[I] != y[I]`. The second template operator stores in element `I` `x[I] != y`. The third template operator stores in element `I` `x != y[I]`.

operator%

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

The first template operator returns an object of class `valarray<T>`, each of whose elements `I` is `x[I] % y[I]`. The second template operator stores in element `I` `x[I] % y`. The third template operator stores in element `I` `x % y[I]`.

operator&

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

The first template operator returns an object of class `valarray<T>`, each of whose elements `I` is `x[I] & y[I]`. The second template operator stores in element `I` `x[I] & y`. The third template operator stores in element `I` `x & y[I]`.

operator&&

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

```

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

```

The first template operator returns an object of class `valarray<bool>`, each of whose elements `I` is `x[I] && y[I]`. The second template operator stores in element `I` `x[I] && y`. The third template operator stores in element `I` `x && y[I]`.

operator>

```

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

```

The first template operator returns an object of class `valarray<bool>` (page 247), each of whose elements `I` is `x[I] > y[I]`. The second template operator stores in element `I` `x[I] > y`. The third template operator stores in element `I` `x > y[I]`.

operator>>

```

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

```

The first template operator returns an object of class `valarray<T>`, each of whose elements `I` is `x[I] >> y[I]`. The second template operator stores in element `I` `x[I] >> y`. The third template operator stores in element `I` `x >> y[I]`.

operator>=

```

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

```

The first template operator returns an object of class `valarray<bool>` (page 247), each of whose elements `I` is `x[I] >= y[I]`. The second template operator stores in element `I` `x[I] >= y`. The third template operator stores in element `I` `x >= y[I]`.

operator<

```

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

```

```

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

```

The first template operator returns an object of class `valarray<bool>` (page 247), each of whose elements `I` is `x[I] < y[I]`. The second template operator stores in element `I` `x[I] < y`. The third template operator stores in element `I` `x < y[I]`.

operator<<

```

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

```

The first template operator returns an object of class `valarray<T>`, each of whose elements `I` is `x[I] << y[I]`. The second template operator stores in element `I` `x[I] << y`. The third template operator stores in element `I` `x << y[I]`.

operator<=

```

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

```

The first template operator returns an object of class `valarray<bool>` (page 247), each of whose elements `I` is `x[I] <= y[I]`. The second template operator stores in element `I` `x[I] <= y`. The third template operator stores in element `I` `x <= y[I]`.

operator*

```

template<class T>
        valarray<T> operator*(const valarray<T>& x,
        const valarray<T>& y);
template<class T>
        valarray<T> operator*(const valarray<T> x,
        const T& y);
template<class T>
        valarray<T> operator*(const T& x,
        const valarray<T>& y);

```

The first template operator returns an object of class `valarray<T>`, each of whose elements `I` is `x[I] * y[I]`. The second template operator stores in element `I` `x[I] * y`. The third template operator stores in element `I` `x * y[I]`.

operator+

```

template<class T>
        valarray<T> operator+(const valarray<T>& x,
        const valarray<T>& y);
template<class T>
        valarray<T> operator+(const valarray<T> x,

```

```

        const T& y);
template<class T>
    valarray<T> operator+(const T& x,
        const valarray<T>& y);

```

The first template operator returns an object of class `valarray<T>`, each of whose elements I is $x[I] + y[I]$. The second template operator stores in element I $x[I] + y$. The third template operator stores in element I $x + y[I]$.

operator-

```

template<class T>
    valarray<T> operator-(const valarray<T>& x,
        const valarray<T>& y);
template<class T>
    valarray<T> operator-(const valarray<T> x,
        const T& y);
template<class T>
    valarray<T> operator-(const T& x,
        const valarray<T>& y);

```

The first template operator returns an object of class `valarray<T>`, each of whose elements I is $x[I] - y[I]$. The second template operator stores in element I $x[I] - y$. The third template operator stores in element I $x - y[I]$.

operator/

```

template<class T>
    valarray<T> operator/(const valarray<T>& x,
        const valarray<T>& y);
template<class T>
    valarray<T> operator/(const valarray<T> x,
        const T& y);
template<class T>
    valarray<T> operator/(const T& x,
        const valarray<T>& y);

```

The first template operator returns an object of class `valarray<T>`, each of whose elements I is $x[I] / y[I]$. The second template operator stores in element I $x[I] / y$. The third template operator stores in element I $x / y[I]$.

operator==

```

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

```

The first template operator returns an object of class `valarray<bool>` (page 247), each of whose elements I is $x[I] == y[I]$. The second template operator stores in element I $x[I] == y$. The third template operator stores in element I $x == y[I]$.

operator^

```

template<class T>
    valarray<T> operator^(const valarray<T>& x,
        const valarray<T>& y);
template<class T>
    valarray<T> operator^(const valarray<T> x,

```

```

        const T& y);
template<class T>
    valarray<T> operator^(const T& x,
        const valarray<T>& y);

```

The first template operator returns an object of class `valarray<T>`, each of whose elements I is $x[I] \wedge y[I]$. The second template operator stores in element I $x[I] \wedge y$. The third template operator stores in element I $x \wedge y[I]$.

operator|

```

template<class T>
    valarray<T> operator|(const valarray<T>& x,
        const valarray<T>& y);
template<class T>
    valarray<T> operator|(const valarray<T> x,
        const T& y);
template<class T>
    valarray<T> operator|(const T& x,
        const valarray<T>& y);

```

The first template operator returns an object of class `valarray<T>`, each of whose elements I is $x[I] \mid y[I]$. The second template operator stores in element I $x[I] \mid y$. The third template operator stores in element I $x \mid y[I]$.

operator||

```

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

```

The first template operator returns an object of class `valarray<bool>`, each of whose elements I is $x[I] \mid\mid y[I]$. The second template operator stores in element I $x[I] \mid\mid y$. The third template operator stores in element I $x \mid\mid y[I]$.

pow

```

template<class T>
    valarray<T> pow(const valarray<T>& x,
        const valarray<T>& y);
template<class T>
    valarray<T> pow(const valarray<T> x, const T& y);
template<class T>
    valarray<T> pow(const T& x, const valarray<T>& y);

```

The first template function returns an object of class `valarray<T>`, each of whose elements I is $x[I]$ raised to the $y[I]$ power. The second template function stores in element I $x[I]$ raised to the y power. The third template function stores in element I x raised to the $y[I]$ power.

sin

```

template<class T>
    valarray<T> sin(const valarray<T>& x);

```

The template function returns an object of class `valarray<T>`, each of whose elements `I` is the sine of `x[I]`.

sinh

```
template<class T>
    valarray<T> sinh(const valarray<T>& x);
```

The template function returns an object of class `valarray<T>`, each of whose elements `I` is the hyperbolic sine of `x[I]`.

slice

```
class slice {
public:
    slice();
    slice(size_t st, size_t len, size_t str);
    size_t start() const;
    size_t size() const;
    size_t stride() const;
};
```

The class stores the parameters that characterize a `slice_array` (page 239) when an object of class `slice` appears as a subscript for an object of class `valarray<T>`. The stored values include:

- a **starting index**
- a **total length**
- a **stride**, or distance between subsequent indices

slice::slice

```
slice();
slice(size_t st,
      const valarray<size_t> len, const valarray<size_t> str);
```

The default constructor stores zeros for the starting index, total length, and stride. The second constructor stores `st` for the starting index, `len` for the total length, and `str` for the stride.

slice::size

```
size_t size() const;
```

The member function returns the stored total length.

slice::start

```
size_t start() const;
```

The member function returns the stored starting index.

slice::stride

```
size_t stride() const;
```

The member function returns the stored stride.

slice_array

```
template<class T>
    class slice_array {
public:
    typedef T value_type;
```

```

void operator=(const valarray<T> x) const;
void operator=(const T& x);
void operator*=(const valarray<T> x) const;
void operator/=(const valarray<T> x) const;
void operator%=(const valarray<T> x) const;
void operator+=(const valarray<T> x) const;
void operator-=(const valarray<T> x) const;
void operator^=(const valarray<T> x) const;
void operator&=(const valarray<T> x) const;
void operator|=(const valarray<T> x) const;
void operator<=(const valarray<T> x) const;
void operator>=(const valarray<T> x) const;
private:
void slice_array(); // not defined
void slice_array(
    const slice_array&); // not defined
slice_array& operator=(
    const slice_array&); // not defined
};

```

The class describes an object that stores a reference to an object `x` of class `valarray<T>`, along with an object `sl` of class `slice` (page 239) which describes the sequence of elements to select from the `valarray<T>` object.

You construct a `slice_array<T>` object only by writing an expression of the form `x[sl]` (page 244). The member functions of class `slice_array` then behave like the corresponding function signatures defined for `valarray<T>`, except that only the sequence of selected elements is affected.

The sequence consists of `sl.size()` elements, where element `i` becomes the index `sl.start() + i * sl.stride()` within `x`. For example:

```

// x[slice(2, 5, 3)] selects elements with
// indices 2, 5, 8, 11, 14

```

sqrt

```

template<class T>
valarray<T> sqrt(const valarray<T>& x);

```

The template function returns an object of class `valarray<T>`, each of whose elements `I` is the square root of `x[I]`.

tan

```

template<class T>
valarray<T> tan(const valarray<T>& x);

```

The template function returns an object of class `valarray<T>`, each of whose elements `I` is the tangent of `x[I]`.

tanh

```

template<class T>
valarray<T> tanh(const valarray<T>& x);

```

The template function returns an object of class `valarray<T>`, each of whose elements `I` is the hyperbolic tangent of `x[I]`.

valarray

`apply` (page 242) · `cshift` (page 242) · `max` (page 242) · `min` (page 242) · `operator T`
`*` (page 243) · `operator!` (page 243) · `operator%=>` (page 243) · `operator&=>` (page

243) · operator>>= (page 243) · operator<<= (page 243) · operator*= (page 243) · operator+ (page 243) · operator+= (page 243) · operator- (page 244) · operator-= (page 244) · operator/= (page 244) · operator= (page 244) · operator[] (page 244) · operator^= (page 246) · operator|= (page 246) · operator~ (page 246) · resize (page 246) · shift (page 246) · size (page 247) · sum (page 247) · valarray (page 247) · value_type (page 247)

```
template<class T>
    class valarray {
public:
    typedef T value_type;
    valarray();
    explicit valarray(size_t n);
    valarray(const T& val, size_t n);
    valarray(const T *p, size_t n);
    valarray(const slice_array<T>& sa);
    valarray(const gslice_array<T>& ga);
    valarray(const mask_array<T>& ma);
    valarray(const indirect_array<T>& ia);
    valarray<T>& operator=(const valarray<T>& va);
    valarray<T>& operator=(const T& x);
    valarray<T>& operator=(const slice_array<T>& sa);
    valarray<T>& operator=(const gslice_array<T>& ga);
    valarray<T>& operator=(const mask_array<T>& ma);
    valarray<T>& operator=(const indirect_array<T>& ia);
    T operator[](size_t n) const;
    T& operator[](size_t n);
    valarray<T> operator[](slice sa) const;
    slice_array<T> operator[](slice sa);
    valarray<T> operator[](const gslice& ga) const;
    gslice_array<T> operator[](const gslice& ga);
    valarray<T>
        operator[](const valarray<bool>& ba) const;
    mask_array<T> operator[](const valarray<bool>& ba);
    valarray<T>
        operator[](const valarray<size_t>& xa) const;
    indirect_array<T>
        operator[](const valarray<size_t>& xa);
    valarray<T> operator+();
    valarray<T> operator-();
    valarray<T> operator~();
    valarray<bool> operator!();
    valarray<T>& operator*=(const valarray<T>& x);
    valarray<T>& operator*=(const T& x);
    valarray<T>& operator/=(const valarray<T>& x);
    valarray<T>& operator/=(const T& x);
    valarray<T>& operator%=(const valarray<T>& x);
    valarray<T>& operator%=(const T& x);
    valarray<T>& operator+=(const valarray<T>& x);
    valarray<T>& operator+=(const T& x);
    valarray<T>& operator-=(const valarray<T>& x);
    valarray<T>& operator-=(const T& x);
    valarray<T>& operator^=(const valarray<T>& x);
    valarray<T>& operator^=(const T& x);
    valarray<T>& operator&=(const valarray<T>& x);
    valarray<T>& operator&=(const T& x);
    valarray<T>& operator|=(const valarray<T>& x);
    valarray<T>& operator|=(const T& x);
    valarray<T>& operator<<=(const valarray<T>& x);
    valarray<T>& operator<<=(const T& x);
    valarray<T>& operator>>=(const valarray<T>& x);
    valarray<T>& operator>>=(const T& x);
    operator T *();
    operator const T *() const;
    size_t size() const;
    T sum() const;
    T max() const;
```

```

T min() const;
valarray<T> shift(int n) const;
valarray<T> cshift(int n) const;
valarray<T> apply(T fn(T)) const;
valarray<T> apply(T fn(const T&)) const;
void resize(size_t n);
void resize(size_t n, const T& c);
};

```

The template class describes an object that controls a varying-length sequence of elements of type `T`. The sequence is stored as an array of `T`. It differs from template class `vector` (page 404) in two important ways:

- It defines numerous arithmetic operations between corresponding elements of `valarray<T>` objects of the same type and length, such as $x = \cos(y) + \sin(z)$.
- It defines a variety of interesting ways to subscript a `valarray<T>` object, by overloading operator[] (page 244).

An object of class `T`:

- has a public default constructor, destructor, copy constructor, and assignment operator — with conventional behavior
- defines the arithmetic operators and math functions, as needed, that are defined for the floating-point types — with conventional behavior

In particular, no subtle differences may exist between copy construction and default construction followed by assignment. And none of the operations on objects of class `T` may throw exceptions.

valarray::apply

```

valarray<T> apply(T fn(T)) const;
valarray<T> apply(T fn(const T&)) const;

```

The member function returns an object of class `valarray<T>`, of length `size()`, each of whose elements `I` is `fn((*this)[I])`.

valarray::cshift

```

valarray<T> cshift(int n) const;

```

The member function returns an object of class `valarray<T>`, of length `size()`, each of whose elements `I` is `(*this)[(I + n) % size()]`. Thus, if element zero is taken as the leftmost element, a positive value of `n` shifts the elements circularly left `n` places.

valarray::max

```

T max() const;

```

The member function returns the value of the largest element of `*this`, which must have nonzero length. If the length is greater than one, it compares values by applying operator< between pairs of corresponding elements of class `T`.

valarray::min

```

T min() const;

```

The member function returns the value of the smallest element of `*this`, which must have nonzero length. If the length is greater than one, it compares values by applying operator< between pairs of elements of class `T`.

valarray::operator T *

```
operator T *();  
operator const T *() const;
```

Both member functions return a pointer to the first element of the controlled array, which must have at least one element.

valarray::operator!

```
valarray<bool> operator!();
```

The member operator returns an object of class `valarray<bool>`, of length `size()`, each of whose elements `I` is `!(*this)[I]`.

valarray::operator%=>

```
valarray<T>& operator%=(const valarray<T>& x);  
valarray<T>& operator%=(const T& x);
```

The member operator replaces each element `I` of `*this` with `(*this)[I] % x[I]`. It returns `*this`.

valarray::operator&=>

```
valarray<T>& operator&=(const valarray<T>& x);  
valarray<T>& operator&=(const T& x);
```

The member operator replaces each element `I` of `*this` with `(*this)[I] & x[I]`. It returns `*this`.

valarray::operator>>=>

```
valarray<T>& operator>>=(const valarray<T>& x);  
valarray<T>& operator>>=(const T& x);
```

The member operator replaces each element `I` of `*this` with `(*this)[I] >> x[I]`. It returns `*this`.

valarray::operator<<=>

```
valarray<T>& operator<<=(const valarray<T>& x);  
valarray<T>& operator<<=(const T& x);
```

The member operator replaces each element `I` of `*this` with `(*this)[I] << x[I]`. It returns `*this`.

valarray::operator*=>

```
valarray<T>& operator*=(const valarray<T>& x);  
valarray<T>& operator*=(const T& x);
```

The member operator replaces each element `I` of `*this` with `(*this)[I] * x[I]`. It returns `*this`.

valarray::operator+>

```
valarray<T> operator+();
```

The member operator returns an object of class `valarray<T>`, of length `size()`, each of whose elements `I` is `(*this)[I]`.

valarray::operator+=>

```
valarray<T>& operator+=(const valarray<T>& x);  
valarray<T>& operator+=(const T& x);
```

The member operator replaces each element I of $*this$ with $(*this)[I] + x[I]$. It returns $*this$.

valarray::operator-

```
valarray<T> operator-();
```

The member operator returns an object of class `valarray<T>`, of length `size()`, each of whose elements I is $-(*this)[I]$.

valarray::operator-=

```
valarray<T>& operator-=(const valarray<T>& x);  
valarray<T>& operator-=(const T& x);
```

The member operator replaces each element I of $*this$ with $(*this)[I] - x[I]$. It returns $*this$.

valarray::operator/=

```
valarray<T>& operator/=(const valarray<T>& x);  
valarray<T>& operator/=(const T& x);
```

The member operator replaces each element I of $*this$ with $(*this)[I] / x[I]$. It returns $*this$.

valarray::operator=

```
valarray<T>& operator=(const valarray<T>& va);  
    valarray<T>& operator=(const T& x);  
valarray<T>& operator=(const slice_array<T>& sa);  
valarray<T>& operator=(const gslice_array<T>& ga);  
valarray<T>& operator=(const mask_array<T>& ma);  
valarray<T>& operator=(const indirect_array<T>& ia);
```

The first member operator replaces the controlled sequence with a copy of the sequence controlled by `va`. The second member operator replaces each element of the controlled sequence with a copy of `x`. The remaining member operators replace those elements of the controlled sequence selected by their arguments, which are generated only by `operator[]` (page 244). If the value of a member in the replacement controlled sequence depends on a member in the initial controlled sequence, the result is undefined.

If the length of the controlled sequence changes, the result is generally undefined. In this implementation (page 3), however, the effect is merely to invalidate any pointers or references to elements in the controlled sequence.

valarray::operator[]

```
T& operator[](size_t n);  
slice_array<T> operator[](slice sa);  
gslice_array<T> operator[](const gslice& ga);  
mask_array<T> operator[](const valarray<bool>& ba);  
indirect_array<T> operator[](const valarray<size_t>& xa);  
  
T operator[](size_t n) const;  
valarray<T> operator[](slice sa) const;  
valarray<T> operator[](const gslice& ga) const;  
valarray<T> operator[](const valarray<bool>& ba) const;  
valarray<T> operator[](const valarray<size_t>& xa) const;
```

The member operator is overloaded to provide several ways to select sequences of elements from among those controlled by $*this$. The first group of five member operators work in conjunction with various overloads of `operator=` (page 244) (and

other assigning operators) to allow selective replacement (slicing) of the controlled sequence. The selected elements must exist.

The first member operator selects element *n*. For example:

```
valarray<char> v0("abcdefghijklmnop", 16);
v0[3] = 'A';
// v0 == valarray<char>("abcAefghijklmnop", 16)
```

The second member operator selects those elements of the controlled sequence designated by *sa*. For example:

```
valarray<char> v0("abcdefghijklmnop", 16);
valarray<char> v1("ABCDE", 5);
v0[slice(2, 5, 3)] = v1;
// v0 == valarray<char>("abAdeBgHcJkDmnEp", 16)
```

The third member operator selects those elements of the controlled sequence designated by *ga*. For example:

```
valarray<char> v0("abcdefghijklmnop", 16);
valarray<char> v1("ABCDEF", 6);
const size_t lv[] = {2, 3};
const size_t dv[] = {7, 2};
const valarray<size_t> len(lv, 2), str(dv, 2);
v0[gslice(3, len, str)] = v1;
// v0 == valarray<char>("abcAeBgCijDlEnFp", 16)
```

The fourth member operator selects those elements of the controlled sequence designated by *ba*. For example:

```
valarray<char> v0("abcdefghijklmnop", 16);
valarray<char> v1("ABC", 3);
const bool vb[] = {false, false, true, true, false, true};
v0[valarray<bool>(vb, 6)] = v1;
// v0 == valarray<char>("abABeCghijklmnop", 16)
```

The fifth member operator selects those elements of the controlled sequence designated by *ia*. For example:

```
valarray<char> v0("abcdefghijklmnop", 16);
valarray<char> v1("ABCDE", 5);
const size_t vi[] = {7, 5, 2, 3, 8};
v0[valarray<size_t>(vi, 5)] = v1;
// v0 == valarray<char>("abCDeBgAEjklmnop", 16)
```

The second group of five member operators each construct an object that represents the value(s) selected. The selected elements must exist.

The sixth member operator returns the value of element *n*. For example:

```
valarray<char> v0("abcdefghijklmnop", 16);
// v0[3] returns 'd'
```

The seventh member operator returns an object of class `valarray<T>` containing those elements of the controlled sequence designated by *sa*. For example:

```
valarray<char> v0("abcdefghijklmnop", 16);
// v0[slice(2, 5, 3)] returns valarray<char>("cfilo", 5)
```

The eighth member operator selects those elements of the controlled sequence designated by *ga*. For example:

```

valarray<char> v0("abcdefghijklmnop", 16);
const size_t lv[] = {2, 3};
const size_t dv[] = {7, 2};
const valarray<size_t> len(lv, 2), str(dv, 2);
// v0[gslice(3, len, str)] returns
//   valarray<char>("dfhkmo", 6)

```

The ninth member operator selects those elements of the controlled sequence designated by *ba*. For example:

```

valarray<char> v0("abcdefghijklmnop", 16);
const bool vb[] = {false, false, true, true, false, true};
// v0[valarray<bool>(vb, 6)] returns
//   valarray<char>("cdf", 3)

```

The last member operator selects those elements of the controlled sequence designated by *ia*. For example:

```

valarray<char> v0("abcdefghijklmnop", 16);
const size_t vi[] = {7, 5, 2, 3, 8};
// v0[valarray<size_t>(vi, 5)] returns
//   valarray<char>("hfcdi", 5)

```

valarray::operator^=

```

valarray<T>& operator^=(const valarray<T>& x);
valarray<T>& operator^=(const T& x);

```

The member operator replaces each element *I* of **this* with $(*this)[I] \wedge x[I]$. It returns **this*.

valarray::operator|=

```

valarray<T>& operator|=(const valarray<T>& x);
valarray<T>& operator|=(const T& x);

```

The member operator replaces each element *I* of **this* with $(*this)[I] \mid x[I]$. It returns **this*.

valarray::operator~

```

valarray<T> operator~();

```

The member operator returns an object of class `valarray<T>`, of length `size()`, each of whose elements *I* is $\sim(*this)[I]$.

valarray::resize

```

void resize(size_t n);
void resize(size_t n, const T& c);

```

The member functions both ensure that `size()` henceforth returns *n*. If it must make the controlled sequence longer, the first member function appends elements with value `T()`, while the second member function appends elements with value *c*. To make the controlled sequence shorter, both member functions remove and delete elements with subscripts in the range $[n, size())$. Any pointers or references to elements in the controlled sequence are invalidated.

valarray::shift

```

valarray<T> shift(int n) const;

```

The member function returns an object of class `valarray<T>`, of length `size()`, each of whose elements `I` is either `(*this)[I + n]`, if `I + n` is a valid subscript, or `T()`. Thus, if element zero is taken as the leftmost element, a positive value of `n` shifts the elements left `n` places, with zero fill.

valarray::size

`size_t size() const;`

The member function returns the number of elements in the array.

valarray::sum

`T sum() const;`

The member function returns the sum of all elements of `*this`, which must have nonzero length. If the length is greater than one, it adds values to the sum by applying operator`+=` between pairs of elements of class `T`.

valarray::valarray

```
valarray();  
explicit valarray(size_t n);  
valarray(const T& val, size_t n);  
valarray(const T *p, size_t n);  
valarray(const slice_array<T>& sa);  
valarray(const gslice_array<T>& ga);  
valarray(const mask_array<T>& ma);  
valarray(const indirect_array<T>& ia);
```

The first (default) constructor initializes the object to an empty array. The next three constructors each initialize the object to an array of `n` elements as follows:

- For `explicit valarray(size_t n)`, each element is initialized with the default constructor.
- For `valarray(const T& val, size_t n)`, each element is initialized with `val`.
- For `valarray(const T *p, size_t n)`, the element at position `I` is initialized with `p[I]`.

Each of the remaining constructors initializes the object to a `valarray<T>` object determined by the argument.

valarray::value_type

`typedef T value_type;`

The type is a synonym for the template parameter `T`.

valarray<bool>

```
class valarray<bool>
```

The type is a specialization of template class `valarray` (page 240), for elements of type `bool`.

Chapter 13. Standard Template Library C++

<algorithm>

adjacent_find (page 253) · binary_search (page 254) · copy (page 254) ·
copy_backward (page 254) · count (page 254) · count_if (page 255) · equal (page
255) · equal_range (page 255) · fill (page 255) · fill_n (page 256) · find (page 256) ·
find_end (page 256) · find_first_of (page 256) · find_if (page 257) · for_each (page
257) · generate (page 257) · generate_n (page 257) · includes (page 257) ·
inplace_merge (page 258) · iter_swap (page 258) · lexicographical_compare (page
258) · lower_bound (page 259) · make_heap (page 259) · max (page 259) ·
max_element (page 259) · merge (page 260) · min (page 260) · min_element (page
261) · mismatch (page 261) · next_permutation (page 261) · nth_element (page
262) · partial_sort (page 262) · partial_sort_copy (page 262) · partition (page 263) ·
pop_heap (page 263) · prev_permutation (page 263) · push_heap (page 264) ·
random_shuffle (page 264) · remove (page 264) · remove_copy (page 265) ·
remove_copy_if (page 265) · remove_if (page 265) · replace (page 266) ·
replace_copy (page 266) · replace_copy_if (page 266) · replace_if (page 266) ·
reverse (page 267) · reverse_copy (page 267) · rotate (page 267) · rotate_copy (page
267) · search (page 267) · search_n (page 268) · set_difference (page 268) ·
set_intersection (page 269) · set_symmetric_difference (page 269) · set_union
(page 270) · sort (page 270) · sort_heap (page 271) · stable_partition (page 271) ·
stable_sort (page 271) · swap (page 272) · swap_ranges (page 272) · transform
(page 272) · unique (page 272) · unique_copy (page 273) · upper_bound (page
273)

```
namespace std {  
    template<class InIt, class Fun>  
        Fun for_each(InIt first, InIt last, Fun f);  
    template<class InIt, class T>  
        InIt find(InIt first, InIt last, const T& val);  
    template<class InIt, class Pred>  
        InIt find_if(InIt first, InIt last, Pred pr);  
    template<class FwdIt1, class FwdIt2>  
        FwdIt1 find_end(FwdIt1 first1, FwdIt1 last1,  
            FwdIt2 first2, FwdIt2 last2);  
    template<class FwdIt1, class FwdIt2, class Pred>  
        FwdIt1 find_end(FwdIt1 first1, FwdIt1 last1,  
            FwdIt2 first2, FwdIt2 last2, Pred pr);  
    template<class FwdIt1, class FwdIt2>  
        FwdIt1 find_first_of(FwdIt1 first1, FwdIt1 last1,  
            FwdIt2 first2, FwdIt2 last2);  
    template<class FwdIt1, class FwdIt2, class Pred>  
        FwdIt1 find_first_of(FwdIt1 first1, FwdIt1 last1,  
            FwdIt2 first2, FwdIt2 last2, Pred pr);  
    template<class FwdIt>  
        FwdIt adjacent_find(FwdIt first, FwdIt last);  
    template<class FwdIt, class Pred>  
        FwdIt adjacent_find(FwdIt first, FwdIt last, Pred pr);  
    template<class InIt, class T, class Dist>  
        typename iterator_traits<InIt>::difference_type  
            count(InIt first, InIt last,  
                const T& val);  
    template<class InIt, class Pred, class Dist>  
        typename iterator_traits<InIt>::difference_type  
            count_if(InIt first, InIt last,  
                Pred pr);  
    template<class InIt1, class InIt2>  
        pair<InIt1, InIt2> mismatch(InIt1 first, InIt1 last,
```

```

        InIt2 x);
template<class InIt1, class InIt2, class Pred>
    pair<InIt1, InIt2> mismatch(InIt1 first, InIt1 last,
        InIt2 x, Pred pr);
template<class InIt1, class InIt2>
    bool equal(InIt1 first, InIt1 last, InIt2 x);
template<class InIt1, class InIt2, class Pred>
    bool equal(InIt1 first, InIt1 last, InIt2 x, Pred pr);
template<class FwdIt1, class FwdIt2>
    FwdIt1 search(FwdIt1 first1, FwdIt1 last1,
        FwdIt2 first2, FwdIt2 last2);
template<class FwdIt1, class FwdIt2, class Pred>
    FwdIt1 search(FwdIt1 first1, FwdIt1 last1,
        FwdIt2 first2, FwdIt2 last2, Pred pr);
template<class FwdIt, class Dist, class T>
    FwdIt search_n(FwdIt first, FwdIt last,
        Dist n, const T& val);
template<class FwdIt, class Dist, class T, class Pred>
    FwdIt search_n(FwdIt first, FwdIt last,
        Dist n, const T& val, Pred pr);
template<class InIt, class OutIt>
    OutIt copy(InIt first, InIt last, OutIt x);
template<class BidIt1, class BidIt2>
    BidIt2 copy_backward(BidIt1 first, BidIt1 last,
        BidIt2 x);
template<class T>
    void swap(T& x, T& y);
template<class FwdIt1, class FwdIt2>
    FwdIt2 swap_ranges(FwdIt1 first, FwdIt1 last,
        FwdIt2 x);
template<class FwdIt1, class FwdIt2>
    void iter_swap(FwdIt1 x, FwdIt2 y);
template<class InIt, class OutIt, class Unop>
    OutIt transform(InIt first, InIt last, OutIt x,
        Unop uop);
template<class InIt1, class InIt2, class OutIt,
    class Binop>
    OutIt transform(InIt1 first1, InIt1 last1,
        InIt2 first2, OutIt x, Binop bop);
template<class FwdIt, class T>
    void replace(FwdIt first, FwdIt last,
        const T& vold, const T& vnew);
template<class FwdIt, class Pred, class T>
    void replace_if(FwdIt first, FwdIt last,
        Pred pr, const T& val);
template<class InIt, class OutIt, class T>
    OutIt replace_copy(InIt first, InIt last, OutIt x,
        const T& vold, const T& vnew);
template<class InIt, class OutIt, class Pred, class T>
    OutIt replace_copy_if(InIt first, InIt last, OutIt x,
        Pred pr, const T& val);
template<class FwdIt, class T>
    void fill(FwdIt first, FwdIt last, const T& x);
template<class OutIt, class Size, class T>
    void fill_n(OutIt first, Size n, const T& x);
template<class FwdIt, class Gen>
    void generate(FwdIt first, FwdIt last, Gen g);
template<class OutIt, class Pred, class Gen>
    void generate_n(OutIt first, Dist n, Gen g);
template<class FwdIt, class T>
    FwdIt remove(FwdIt first, FwdIt last, const T& val);
template<class FwdIt, class Pred>
    FwdIt remove_if(FwdIt first, FwdIt last, Pred pr);
template<class InIt, class OutIt, class T>
    OutIt remove_copy(InIt first, InIt last, OutIt x,
        const T& val);
template<class InIt, class OutIt, class Pred>

```

```

        OutIt remove_copy_if(InIt first, InIt last, OutIt x,
                               Pred pr);
template<class FwdIt>
    FwdIt unique(FwdIt first, FwdIt last);
template<class FwdIt, class Pred>
    FwdIt unique(FwdIt first, FwdIt last, Pred pr);
template<class InIt, class OutIt>
    OutIt unique_copy(InIt first, InIt last, OutIt x);
template<class InIt, class OutIt, class Pred>
    OutIt unique_copy(InIt first, InIt last, OutIt x,
                       Pred pr);
template<class BidIt>
    void reverse(BidIt first, BidIt last);
template<class BidIt, class OutIt>
    OutIt reverse_copy(BidIt first, BidIt last, OutIt x);
template<class FwdIt>
    void rotate(FwdIt first, FwdIt middle, FwdIt last);
template<class FwdIt, class OutIt>
    OutIt rotate_copy(FwdIt first, FwdIt middle,
                       FwdIt last, OutIt x);
template<class RanIt>
    void random_shuffle(RanIt first, RanIt last);
template<class RanIt, class Fun>
    void random_shuffle(RanIt first, RanIt last, Fun& f);
template<class BidIt, class Pred>
    BidIt partition(BidIt first, BidIt last, Pred pr);
template<class BidIt, class Pred>
    BidIt stable_partition(BidIt first, BidIt last,
                             Pred pr);
template<class RanIt>
    void sort(RanIt first, RanIt last);
template<class RanIt, class Pred>
    void sort(RanIt first, RanIt last, Pred pr);
template<class BidIt>
    void stable_sort(BidIt first, BidIt last);
template<class BidIt, class Pred>
    void stable_sort(BidIt first, BidIt last, Pred pr);
template<class RanIt>
    void partial_sort(RanIt first, RanIt middle,
                       RanIt last);
template<class RanIt, class Pred>
    void partial_sort(RanIt first, RanIt middle,
                       RanIt last, Pred pr);
template<class InIt, class RanIt>
    RanIt partial_sort_copy(InIt first1, InIt last1,
                             RanIt first2, RanIt last2);
template<class InIt, class RanIt, class Pred>
    RanIt partial_sort_copy(InIt first1, InIt last1,
                             RanIt first2, RanIt last2, Pred pr);
template<class RanIt>
    void nth_element(RanIt first, RanIt nth, RanIt last);
template<class RanIt, class Pred>
    void nth_element(RanIt first, RanIt nth, RanIt last,
                       Pred pr);
template<class FwdIt, class T>
    FwdIt lower_bound(FwdIt first, FwdIt last,
                       const T& val);
template<class FwdIt, class T, class Pred>
    FwdIt lower_bound(FwdIt first, FwdIt last,
                       const T& val, Pred pr);
template<class FwdIt, class T>
    FwdIt upper_bound(FwdIt first, FwdIt last,
                       const T& val);
template<class FwdIt, class T, class Pred>
    FwdIt upper_bound(FwdIt first, FwdIt last,
                       const T& val, Pred pr);
template<class FwdIt, class T>

```

```

    pair<FwdIt, FwdIt> equal_range(FwdIt first,
        FwdIt last, const T& val);
template<class FwdIt, class T, class Pred>
    pair<FwdIt, FwdIt> equal_range(FwdIt first,
        FwdIt last, const T& val, Pred pr);
template<class FwdIt, class T>
    bool binary_search(FwdIt first, FwdIt last,
        const T& val);
template<class FwdIt, class T, class Pred>
    bool binary_search(FwdIt first, FwdIt last,
        const T& val, Pred pr);
template<class InIt1, class InIt2, class OutIt>
    OutIt merge(InIt1 first1, InIt1 last1,
        InIt2 first2, InIt2 last2, OutIt x);
template<class InIt1, class InIt2, class OutIt,
    class Pred>
    OutIt merge(InIt1 first1, InIt1 last1,
        InIt2 first2, InIt2 last2, OutIt x, Pred pr);
template<class BidIt>
    void inplace_merge(BidIt first, BidIt middle,
        BidIt last);
template<class BidIt, class Pred>
    void inplace_merge(BidIt first, BidIt middle,
        BidIt last, Pred pr);
template<class InIt1, class InIt2>
    bool includes(InIt1 first1, InIt1 last1,
        InIt2 first2, InIt2 last2);
template<class InIt1, class InIt2, class Pred>
    bool includes(InIt1 first1, InIt1 last1,
        InIt2 first2, InIt2 last2, Pred pr);
template<class InIt1, class InIt2, class OutIt>
    OutIt set_union(InIt1 first1, InIt1 last1,
        InIt2 first2, InIt2 last2, OutIt x);
template<class InIt1, class InIt2, class OutIt,
    class Pred>
    OutIt set_union(InIt1 first1, InIt1 last1,
        InIt2 first2, InIt2 last2, OutIt x, Pred pr);
template<class InIt1, class InIt2, class OutIt>
    OutIt set_intersection(InIt1 first1, InIt1 last1,
        InIt2 first2, InIt2 last2, OutIt x);
template<class InIt1, class InIt2, class OutIt,
    class Pred>
    OutIt set_intersection(InIt1 first1, InIt1 last1,
        InIt2 first2, InIt2 last2, OutIt x, Pred pr);
template<class InIt1, class InIt2, class OutIt>
    OutIt set_difference(InIt1 first1, InIt1 last1,
        InIt2 first2, InIt2 last2, OutIt x);
template<class InIt1, class InIt2, class OutIt,
    class Pred>
    OutIt set_difference(InIt1 first1, InIt1 last1,
        InIt2 first2, InIt2 last2, OutIt x, Pred pr);
template<class InIt1, class InIt2, class OutIt>
    OutIt set_symmetric_difference(InIt1 first1,
        InIt1 last1, InIt2 first2, InIt2 last2, OutIt x);
template<class InIt1, class InIt2, class OutIt,
    class Pred>
    OutIt set_symmetric_difference(InIt1 first1,
        InIt1 last1, InIt2 first2, InIt2 last2, OutIt x,
        Pred pr);
template<class RanIt>
    void push_heap(RanIt first, RanIt last);
template<class RanIt, class Pred>
    void push_heap(RanIt first, RanIt last, Pred pr);
template<class RanIt>
    void pop_heap(RanIt first, RanIt last);
template<class RanIt, class Pred>
    void pop_heap(RanIt first, RanIt last, Pred pr);

```

```

template<class RanIt>
    void make_heap(RanIt first, RanIt last);
template<class RanIt, class Pred>
    void make_heap(RanIt first, RanIt last, Pred pr);
template<class RanIt>
    void sort_heap(RanIt first, RanIt last);
template<class RanIt, class Pred>
    void sort_heap(RanIt first, RanIt last, Pred pr);
template<class T>
    const T& max(const T& x, const T& y);
template<class T, class Pred>
    const T& max(const T& x, const T& y, Pred pr);
template<class T>
    const T& min(const T& x, const T& y);
template<class T, class Pred>
    const T& min(const T& x, const T& y, Pred pr);
template<class FwdIt>
    FwdIt max_element(FwdIt first, FwdIt last);
template<class FwdIt, class Pred>
    FwdIt max_element(FwdIt first, FwdIt last, Pred pr);
template<class FwdIt>
    FwdIt min_element(FwdIt first, FwdIt last);
template<class FwdIt, class Pred>
    FwdIt min_element(FwdIt first, FwdIt last, Pred pr);
template<class InIt1, class InIt2>
    bool lexicographical_compare(InIt1 first1,
        InIt1 last1, InIt2 first2, InIt2 last2);
template<class InIt1, class InIt2, class Pred>
    bool lexicographical_compare(InIt1 first1,
        InIt1 last1, InIt2 first2, InIt2 last2, Pred pr);
template<class BidIt>
    bool next_permutation(BidIt first, BidIt last);
template<class BidIt, class Pred>
    bool next_permutation(BidIt first, BidIt last,
        Pred pr);
template<class BidIt>
    bool prev_permutation(BidIt first, BidIt last);
template<class BidIt, class Pred>
    bool prev_permutation(BidIt first, BidIt last,
        Pred pr);
};

```

Include the STL (page 1) standard header **<algorithm>** to define numerous template functions that perform useful algorithms. The descriptions that follow make extensive use of common template parameter names (or prefixes) to indicate the least powerful category of iterator permitted as an actual argument type:

- **OutIt** (page 37) — to indicate an output iterator
- **InIt** (page 37) — to indicate an input iterator
- **FwdIt** (page 37) — to indicate a forward iterator
- **BidIt** (page 37) — to indicate a bidirectional iterator
- **RanIt** (page 37) — to indicate a random-access iterator

The descriptions of these templates employ a number of conventions (page 38) common to all algorithms.

adjacent_find

```

template<class FwdIt>
    FwdIt adjacent_find(FwdIt first, FwdIt last);
template<class FwdIt, class Pred>
    FwdIt adjacent_find(FwdIt first, FwdIt last, Pred pr);

```

The first template function determines the lowest N in the range $[0, \text{last} - \text{first})$ for which $N + 1 \neq \text{last} - \text{first}$ and the predicate $\text{pr}(*(\text{first} + N)) == *(\text{first} + N + 1)$ is true. Here, `operator==` must impose an equivalence relationship (page 39) between its operands. It then returns $\text{first} + N$. If no such value exists, the function returns `last`. It evaluates the predicate exactly $N + 1$ times.

The second template function behaves the same, except that the predicate is `pr(*(\text{first} + N), *(\text{first} + N + 1))`.

binary_search

```
template<class FwdIt, class T>
    bool binary_search(FwdIt first, FwdIt last,
        const T& val);
template<class FwdIt, class T, class Pred>
    bool binary_search(FwdIt first, FwdIt last,
        const T& val, Pred pr);
```

The first template function determines whether a value of N exists in the range $[0, \text{last} - \text{first})$ for which $*(\text{first} + N)$ has equivalent ordering (page 39) to `val`, where the elements designated by iterators in the range $[\text{first}, \text{last})$ form a sequence ordered by (page 39) `operator<`. If so, the function returns `true`. If no such value exists, it returns `false`.

If `FwdIt` is a random-access iterator type, the function evaluates the ordering predicate $X < Y$ at most $\text{ceil}(\log(\text{last} - \text{first})) + 2$ times. Otherwise, the function evaluates the predicate a number of times proportional to $\text{last} - \text{first}$.

The second template function behaves the same, except that it replaces `operator<(X, Y)` with `pr(X, Y)`.

copy

```
template<class InIt, class OutIt>
    OutIt copy(InIt first, InIt last, OutIt x);
```

The template function evaluates $*(x + N) = *(\text{first} + N)$ once for each N in the range $[0, \text{last} - \text{first})$, for strictly increasing values of N beginning with the lowest value. It then returns $x + N$. If x and `first` designate regions of storage, x must not be in the range $[\text{first}, \text{last})$.

copy_backward

```
template<class BidIt1, class BidIt2>
    BidIt2 copy_backward(BidIt1 first, BidIt1 last,
        BidIt2 x);
```

The template function evaluates $*(x - N - 1) = *(\text{last} - N - 1)$ once for each N in the range $[0, \text{last} - \text{first})$, for strictly decreasing values of N beginning with the highest value. It then returns $x - (\text{last} - \text{first})$. If x and `first` designate regions of storage, x must not be in the range $[\text{first}, \text{last})$.

count

```
template<class InIt, class T>
    typename iterator_traits<InIt>::difference_type
    count(InIt first, InIt last, const T& val);
```

The template function sets a count n to zero. It then executes `++n` for each N in the range $[0, \text{last} - \text{first})$ for which the predicate $*(\text{first} + N) == \text{val}$ is true.

Here, `operator==` must impose an equivalence relationship (page 39) between its operands. The function returns `n`. It evaluates the predicate exactly `last - first` times.

count_if

```
template<class InIt, class Pred, class Dist>
    typename iterator_traits<InIt>::difference_type
    count_if(InIt first, InIt last,
            Pred pr);
```

The template function sets a count `n` to zero. It then executes `++n` for each `N` in the range `[0, last - first)` for which the predicate `pr(*(first + N))` is true. The function returns `n`. It evaluates the predicate exactly `last - first` times.

equal

```
template<class InIt1, class InIt2>
    bool equal(InIt1 first, InIt1 last, InIt2 x);
template<class InIt1, class InIt2, class Pred>
    bool equal(InIt1 first, InIt1 last, InIt2 x, Pred pr);
```

The first template function returns true only if, for each `N` in the range `[0, last1 - first1)`, the predicate `*(first1 + N) == *(first2 + N)` is true. Here, `operator==` must impose an equivalence relationship (page 39) between its operands. The function evaluates the predicate at most once for each `N`.

The second template function behaves the same, except that the predicate is `pr(*(first1 + N), *(first2 + N))`.

equal_range

```
template<class FwdIt, class T>
    pair<FwdIt, FwdIt> equal_range(FwdIt first,
                                   FwdIt last, const T& val);
template<class FwdIt, class T, class Pred>
    pair<FwdIt, FwdIt> equal_range(FwdIt first,
                                   FwdIt last, const T& val, Pred pr);
```

The first template function effectively returns `pair(lower_bound(first, last, val), upper_bound(first, last, val))`, where the elements designated by iterators in the range `[first, last)` form a sequence ordered by (page 39) `operator<`. Thus, the function determines the largest range of positions over which `val` can be inserted in the sequence and still preserve its ordering.

If `FwdIt` is a random-access iterator type, the function evaluates the ordering predicate `X < Y` at most `ceil(2 * log(last - first)) + 1`. Otherwise, the function evaluates the predicate a number of times proportional to `last - first`.

The second template function behaves the same, except that it replaces `operator<(X, Y)` with `pr(X, Y)`.

fill

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

The template function evaluates `*(first + N) = x` once for each `N` in the range `[0, last - first)`.

fill_n

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

The template function evaluates $*(first + N) = x$ once for each N in the range $[0, n)$.

find

```
template<class InIt, class T>
InIt find(InIt first, InIt last, const T& val);
```

The template function determines the lowest value of N in the range $[0, last - first)$ for which the predicate $*(first + N) == val$ is true. Here, `operator==` must impose an equivalence relationship (page 39) between its operands. It then returns $first + N$. If no such value exists, the function returns `last`. It evaluates the predicate at most once for each N .

find_end

```
template<class FwdIt1, class FwdIt2>
FwdIt1 find_end(FwdIt1 first1, FwdIt1 last1,
                FwdIt2 first2, FwdIt2 last2);
template<class FwdIt1, class FwdIt2, class Pred>
FwdIt1 find_end(FwdIt1 first1, FwdIt1 last1,
                FwdIt2 first2, FwdIt2 last2, Pred pr);
```

The first template function determines the highest value of N in the range $[0, last1 - first1 - (last2 - first2))$ such that for each M in the range $[0, last2 - first2)$, the predicate $*(first1 + N + M) == *(first2 + N + M)$ is true. Here, `operator==` must impose an equivalence relationship (page 39) between its operands. It then returns $first1 + N$. If no such value exists, the function returns `last1`. It evaluates the predicate at most $(last2 - first2) * (last1 - first1 - (last2 - first2) + 1)$ times.

The second template function behaves the same, except that the predicate is `pr(*(first1 + N + M), *(first2 + N + M))`.

find_first_of

```
template<class FwdIt1, class FwdIt2>
FwdIt1 find_first_of(FwdIt1 first1, FwdIt1 last1,
                    FwdIt2 first2, FwdIt2 last2);
template<class FwdIt1, class FwdIt2, class Pred>
FwdIt1 find_first_of(FwdIt1 first1, FwdIt1 last1,
                    FwdIt2 first2, FwdIt2 last2, Pred pr);
```

The first template function determines the lowest value of N in the range $[0, last1 - first1)$ such that for some M in the range $[0, last2 - first2)$, the predicate $*(first1 + N) == *(first2 + M)$ is true. Here, `operator==` must impose an equivalence relationship (page 39) between its operands. It then returns $first1 + N$. If no such value exists, the function returns `last1`. It evaluates the predicate at most $(last1 - first1) * (last2 - first2)$ times.

The second template function behaves the same, except that the predicate is `pr(*(first1 + N), *(first2 + M))`.

find_if

```
template<class InIt, class Pred>
InIt find_if(InIt first, InIt last, Pred pr);
```

The template function determines the lowest value of N in the range $[0, \text{last} - \text{first})$ for which the predicate $\text{pred}*(\text{first} + N)$ is true. It then returns $\text{first} + N$. If no such value exists, the function returns last . It evaluates the predicate at most once for each N .

for_each

```
template<class InIt, class Fun>
Fun for_each(InIt first, InIt last, Fun f);
```

The template function evaluates $f*(\text{first} + N)$ once for each N in the range $[0, \text{last} - \text{first})$. It then returns f .

generate

```
template<class FwdIt, class Gen>
void generate(FwdIt first, FwdIt last, Gen g);
```

The template function evaluates $*(\text{first} + N) = g()$ once for each N in the range $[0, \text{last} - \text{first})$.

generate_n

```
template<class OutIt, class Pred, class Gen>
void generate_n(OutIt first, Dist n, Gen g);
```

The template function evaluates $*(\text{first} + N) = g()$ once for each N in the range $[0, n)$.

includes

```
template<class InIt1, class InIt2>
bool includes(InIt1 first1, InIt1 last1,
              InIt2 first2, InIt2 last2);
template<class InIt1, class InIt2, class Pred>
bool includes(InIt1 first1, InIt1 last1,
              InIt2 first2, InIt2 last2, Pred pr);
```

The first template function determines whether a value of N exists in the range $[0, \text{last2} - \text{first2})$ such that, for each M in the range $[0, \text{last1} - \text{first1})$, $*(\text{first} + M)$ and $*(\text{first} + N)$ do not have equivalent ordering (page 39), where the elements designated by iterators in the ranges $[\text{first1}, \text{last1})$ and $[\text{first2}, \text{last2})$ each form a sequence ordered by (page 39) operator $<$. If so, the function returns false. If no such value exists, it returns true. Thus, the function determines whether the ordered sequence designated by iterators in the range $[\text{first2}, \text{last2})$ all have equivalent ordering with some element designated by iterators in the range $[\text{first1}, \text{last1})$.

The function evaluates the predicate at most $2 * ((\text{last1} - \text{first1}) + (\text{last2} - \text{first2})) - 1$ times.

The second template function behaves the same, except that it replaces operator $<(X, Y)$ with $\text{pr}(X, Y)$.

inplace_merge

```
template<class BidIt>
    void inplace_merge(BidIt first, BidIt middle,
        BidIt last);
template<class BidIt, class Pred>
    void inplace_merge(BidIt first, BidIt middle,
        BidIt last, Pred pr);
```

The first template function reorders the sequences designated by iterators in the ranges `[first, middle)` and `[middle, last)`, each ordered by (page 39) `operator<`, to form a merged sequence of length `last - first` beginning at `first` also ordered by `operator<`. The merge occurs without altering the relative order of elements within either original sequence. Moreover, for any two elements from different original sequences that have equivalent ordering (page 39), the element from the ordered range `[first, middle)` precedes the other.

The function evaluates the ordering predicate $X < Y$ at most $\text{ceil}((\text{last} - \text{first}) * \log(\text{last} - \text{first}))$ times. (Given enough temporary storage, it can evaluate the predicate at most $(\text{last} - \text{first}) - 1$ times.)

The second template function behaves the same, except that it replaces `operator<(X, Y)` with `pr(X, Y)`.

iter_swap

```
template<class FwdIt1, class FwdIt2>
    void iter_swap(FwdIt1 x, FwdIt2 y);
```

The template function leaves the value originally stored in `*y` subsequently stored in `*x`, and the value originally stored in `*x` subsequently stored in `*y`.

lexicographical_compare

```
template<class InIt1, class InIt2>
    bool lexicographical_compare(InIt1 first1,
        InIt1 last1, InIt2 first2, InIt2 last2);
template<class InIt1, class InIt2, class Pred>
    bool lexicographical_compare(InIt1 first1,
        InIt1 last1, InIt2 first2, InIt2 last2, Pred pr);
```

The first template function determines K , the number of elements to compare as the smaller of `last1 - first1` and `last2 - first2`. It then determines the lowest value of N in the range $[0, K)$ for which `*(first1 + N)` and `*(first2 + N)` do not have equivalent ordering (page 39). If no such value exists, the function returns true only if $K < (\text{last2} - \text{first2})$. Otherwise, it returns true only if `*(first1 + N) < *(first2 + N)`. Thus, the function returns true only if the sequence designated by iterators in the range `[first1, last1)` is lexicographically less than the other sequence.

The function evaluates the ordering predicate $X < Y$ at most $2 * K$ times.

The second template function behaves the same, except that it replaces `operator<(X, Y)` with `pr(X, Y)`.

lower_bound

```
template<class FwdIt, class T>
    FwdIt lower_bound(FwdIt first, FwdIt last,
        const T& val);
template<class FwdIt, class T, class Pred>
    FwdIt lower_bound(FwdIt first, FwdIt last,
        const T& val, Pred pr);
```

The first template function determines the highest value of N in the range $(0, \text{last} - \text{first}]$ such that, for each M in the range $[0, N)$ the predicate $*(\text{first} + M) < \text{val}$ is true, where the elements designated by iterators in the range $[\text{first}, \text{last})$ form a sequence ordered by (page 39) `operator<`. It then returns $\text{first} + N$. Thus, the function determines the lowest position before which `val` can be inserted in the sequence and still preserve its ordering.

If `FwdIt` is a random-access iterator type, the function evaluates the ordering predicate $X < Y$ at most $\text{ceil}(\log(\text{last} - \text{first})) + 1$ times. Otherwise, the function evaluates the predicate a number of times proportional to $\text{last} - \text{first}$.

The second template function behaves the same, except that it replaces `operator<(X, Y)` with `pr(X, Y)`.

make_heap

```
template<class RanIt>
    void make_heap(RanIt first, RanIt last);
template<class RanIt, class Pred>
    void make_heap(RanIt first, RanIt last, Pred pr);
```

The first template function reorders the sequence designated by iterators in the range $[\text{first}, \text{last})$ to form a heap ordered by (page 39) `operator<`.

The function evaluates the ordering predicate $X < Y$ at most $3 * (\text{last} - \text{first})$ times.

The second template function behaves the same, except that it replaces `operator<(X, Y)` with `pr(X, Y)`.

max

```
template<class T>
    const T& max(const T& x, const T& y);
template<class T, class Pred>
    const T& max(const T& x, const T& y, Pred pr);
```

The first template function returns y if $x < y$. Otherwise it returns x . `T` need supply only a single-argument constructor and a destructor.

The second template function behaves the same, except that it replaces `operator<(X, Y)` with `pr(X, Y)`.

max_element

```
template<class FwdIt>
    FwdIt max_element(FwdIt first, FwdIt last);
template<class FwdIt, class Pred>
    FwdIt max_element(FwdIt first, FwdIt last, Pred pr);
```

The first template function determines the lowest value of N in the range $[0, \text{last} - \text{first})$ such that, for each M in the range $[0, \text{last} - \text{first})$ the predicate $\text{*(first + N)} < \text{*(first + M)}$ is false. It then returns $\text{first} + N$. Thus, the function determines the lowest position that contains the largest value in the sequence.

The function evaluates the ordering predicate $X < Y$ exactly $\max((\text{last} - \text{first}) - 1, 0)$ times.

The second template function behaves the same, except that it replaces $\text{operator}<(X, Y)$ with $\text{pr}(X, Y)$.

merge

```
template<class InIt1, class InIt2, class OutIt>
    OutIt merge(InIt1 first1, InIt1 last1,
                InIt2 first2, InIt2 last2, OutIt x);
template<class InIt1, class InIt2, class OutIt,
        class Pred>
    OutIt merge(InIt1 first1, InIt1 last1,
                InIt2 first2, InIt2 last2, OutIt x, Pred pr);
```

The first template function determines K , the number of elements to copy as $(\text{last1} - \text{first1}) + (\text{last2} - \text{first2})$. It then alternately copies two sequences, designated by iterators in the ranges $[\text{first1}, \text{last1})$ and $[\text{first2}, \text{last2})$ and each ordered by (page 39) $\text{operator}<$, to form a merged sequence of length K beginning at x , also ordered by $\text{operator}<$. The function then returns $x + K$.

The merge occurs without altering the relative order of elements within either sequence. Moreover, for any two elements from different sequences that have equivalent ordering (page 39), the element from the ordered range $[\text{first1}, \text{last1})$ precedes the other. Thus, the function merges two ordered sequences to form another ordered sequence.

If x and first1 designate regions of storage, the range $[x, x + K)$ must not overlap the range $[\text{first1}, \text{last1})$. If x and first2 designate regions of storage, the range $[x, x + K)$ must not overlap the range $[\text{first2}, \text{last2})$. The function evaluates the ordering predicate $X < Y$ at most $K - 1$ times.

The second template function behaves the same, except that it replaces $\text{operator}<(X, Y)$ with $\text{pr}(X, Y)$.

min

```
template<class T>
    const T& min(const T& x, const T& y);
template<class T, class Pred>
    const T& min(const T& x, const T& y, Pred pr);
```

The first template function returns y if $y < x$. Otherwise it returns x . T need supply only a single-argument constructor and a destructor.

The second template function behaves the same, except that it replaces $\text{operator}<(X, Y)$ with $\text{pr}(X, Y)$.

min_element

```
template<class FwdIt>
    FwdIt min_element(FwdIt first, FwdIt last);
template<class FwdIt, class Pred>
    FwdIt min_element(FwdIt first, FwdIt last, Pred pr);
```

The first template function determines the lowest value of N in the range $[0, \text{last} - \text{first})$ such that, for each M in the range $[0, \text{last} - \text{first})$ the predicate $*(\text{first} + M) < *(\text{first} + N)$ is false. It then returns $\text{first} + N$. Thus, the function determines the lowest position that contains the smallest value in the sequence.

The function evaluates the ordering predicate $X < Y$ exactly $\max((\text{last} - \text{first}) - 1, 0)$ times.

The second template function behaves the same, except that it replaces $\text{operator}<(X, Y)$ with $\text{pr}(X, Y)$.

mismatch

```
template<class InIt1, class InIt2>
    pair<InIt1, InIt2> mismatch(InIt1 first, InIt1 last,
                                InIt2 x);
template<class InIt1, class InIt2, class Pred>
    pair<InIt1, InIt2> mismatch(InIt1 first, InIt1 last,
                                InIt2 x, Pred pr);
```

The first template function determines the lowest value of N in the range $[0, \text{last1} - \text{first1})$ for which the predicate $!(*(\text{first1} + N) == *(\text{first2} + N))$ is true. Here, $\text{operator}==$ must impose an equivalence relationship (page 39) between its operands. It then returns $\text{pair}(\text{first1} + N, \text{first2} + N)$. If no such value exists, N has the value $\text{last1} - \text{first1}$. The function evaluates the predicate at most once for each N .

The second template function behaves the same, except that the predicate is $\text{pr}(*(\text{first1} + N), *(\text{first2} + N))$.

next_permutation

```
template<class BidIt>
    bool next_permutation(BidIt first, BidIt last);
template<class BidIt, class Pred>
    bool next_permutation(BidIt first, BidIt last,
                            Pred pr);
```

The first template function determines a repeating sequence of permutations, whose initial permutation occurs when the sequence designated by iterators in the range $[\text{first}, \text{last})$ is ordered by (page 39) $\text{operator}<$. (The elements are sorted in *ascending* order.) It then reorders the elements in the sequence, by evaluating $\text{swap}(X, Y)$ for the elements X and Y zero or more times, to form the next permutation. The function returns true only if the resulting sequence is not the initial permutation. Otherwise, the resultant sequence is the one next larger lexicographically than the original sequence. No two elements may have equivalent ordering (page 39).

The function evaluates $\text{swap}(X, Y)$ at most $(\text{last} - \text{first}) / 2$.

The second template function behaves the same, except that it replaces $\text{operator}<(X, Y)$ with $\text{pr}(X, Y)$.

nth_element

```
template<class RanIt>
    void nth_element(RanIt first, RanIt nth, RanIt last);
template<class RanIt, class Pred>
    void nth_element(RanIt first, RanIt nth, RanIt last,
        Pred pr);
```

The first template function reorders the sequence designated by iterators in the range `[first, last)` such that for each `N` in the range `[0, nth - first)` and for each `M` in the range `[nth - first, last - first)` the predicate `!(*(first + M) < *(first + N))` is true. Moreover, for `N` equal to `nth - first` and for each `M` in the range `(nth - first, last - first)` the predicate `!(*(first + M) < *(first + N))` is true. Thus, if `nth != last` the element `*nth` is in its proper position if elements of the entire sequence were sorted in *ascending* order, ordered by (page 39) `operator<`. Any elements before this one belong before it in the sort sequence, and any elements after it belong after it.

The function evaluates the ordering predicate `X < Y` a number of times proportional to `last - first`, on average.

The second template function behaves the same, except that it replaces `operator<(X, Y)` with `pr(X, Y)`.

partial_sort

```
template<class RanIt>
    void partial_sort(RanIt first, RanIt middle,
        RanIt last);
template<class RanIt, class Pred>
    void partial_sort(RanIt first, RanIt middle,
        RanIt last, Pred pr);
```

The first template function reorders the sequence designated by iterators in the range `[first, last)` such that for each `N` in the range `[0, middle - first)` and for each `M` in the range `(N, last - first)` the predicate `!(*(first + M) < *(first + N))` is true. Thus, the smallest `middle - first` elements of the entire sequence are sorted in *ascending* order, ordered by (page 39) `operator<`. The order of the remaining elements is otherwise unspecified.

The function evaluates the ordering predicate `X < Y` at most `ceil((last - first) * log(middle - first))` times.

The second template function behaves the same, except that it replaces `operator<(X, Y)` with `pr(X, Y)`.

partial_sort_copy

```
template<class InIt, class RanIt>
    RanIt partial_sort_copy(InIt first1, InIt last1,
        RanIt first2, RanIt last2);
template<class InIt, class RanIt, class Pred>
    RanIt partial_sort_copy(InIt first1, InIt last1,
        RanIt first2, RanIt last2, Pred pr);
```

The first template function determines `K`, the number of elements to copy as the smaller of `last1 - first1` and `last2 - first2`. It then copies and reorders `K` of the sequence designated by iterators in the range `[first1, last1)` such that the `K` elements copied to `first2` are ordered by (page 39) `operator<`. Moreover, for each `N` in the range `[0, K)` and for each `M` in the range `(0, last1 - first1)`

corresponding to an uncopied element, the predicate `!(*(first2 + M) < *(first1 + N))` is true. Thus, the smallest K elements of the entire sequence designated by iterators in the range `[first1, last1)` are copied and sorted in *ascending* order to the range `[first2, first2 + K)`.

The function evaluates the ordering predicate $X < Y$ at most $\text{ceil}((\text{last} - \text{first}) * \log(K))$ times.

The second template function behaves the same, except that it replaces `operator<(X, Y)` with `pr(X, Y)`.

partition

```
template<class BidIt, class Pred>
    BidIt partition(BidIt first, BidIt last, Pred pr);
```

The template function reorders the sequence designated by iterators in the range `[first, last)` and determines the value K such that for each N in the range $[0, K)$ the predicate `pr(*(first + N))` is true, and for each N in the range $[K, \text{last} - \text{first})$ the predicate `pr(*(first + N))` is false. The function then returns `first + K`.

The predicate must not alter its operand. The function evaluates `pr(*(first + N))` exactly `last - first` times, and swaps at most $(\text{last} - \text{first}) / 2$ pairs of elements.

pop_heap

```
template<class RanIt>
    void pop_heap(RanIt first, RanIt last);
template<class RanIt, class Pred>
    void pop_heap(RanIt first, RanIt last, Pred pr);
```

The first template function reorders the sequence designated by iterators in the range `[first, last)` to form a new heap, ordered by (page 39) `operator<` and designated by iterators in the range `[first, last - 1)`, leaving the original element at `*first` subsequently at `*(last - 1)`. The original sequence must designate an existing heap, also ordered by `operator<`. Thus, `first != last` must be true and `*(last - 1)` is the element to remove from (pop off) the heap.

The function evaluates the ordering predicate $X < Y$ at most $\text{ceil}(2 * \log(\text{last} - \text{first}))$ times.

The second template function behaves the same, except that it replaces `operator<(X, Y)` with `pr(X, Y)`.

prev_permutation

```
template<class BidIt>
    bool prev_permutation(BidIt first, BidIt last);
template<class BidIt, class Pred>
    bool prev_permutation(BidIt first, BidIt last,
                          Pred pr);
```

The first template function determines a repeating sequence of permutations, whose initial permutation occurs when the sequence designated by iterators in the range `[first, last)` is the *reverse* of one ordered by (page 39) `operator<`. (The elements are sorted in *descending* order.) It then reorders the elements in the sequence, by evaluating `swap(X, Y)` for the elements X and Y zero or more times, to

form the next permutation. The function returns true only if the resulting sequence is not the initial permutation. Otherwise, the resultant sequence is the one next smaller lexicographically than the original sequence. No two elements may have equivalent ordering (page 39).

The function evaluates `swap(X, Y)` at most $(\text{last} - \text{first}) / 2$.

The second template function behaves the same, except that it replaces `operator<(X, Y)` with `pr(X, Y)`.

push_heap

```
template<class RanIt>
    void push_heap(RanIt first, RanIt last);
template<class RanIt, class Pred>
    void push_heap(RanIt first, RanIt last, Pred pr);
```

The first template function reorders the sequence designated by iterators in the range `[first, last)` to form a new heap ordered by (page 39) `operator<`. Iterators in the range `[first, last - 1)` must designate an existing heap, also ordered by `operator<`. Thus, `first != last` must be true and `*(last - 1)` is the element to add to (push on) the heap.

The function evaluates the ordering predicate `X < Y` at most $\text{ceil}(\log(\text{last} - \text{first}))$ times.

The second template function behaves the same, except that it replaces `operator<(X, Y)` with `pr(X, Y)`.

random_shuffle

```
template<class RanIt>
    void random_shuffle(RanIt first, RanIt last);
template<class RanIt, class Fun>
    void random_shuffle(RanIt first, RanIt last, Fun& f);
```

The first template function evaluates `swap(*(first + N), *(first + M))` once for each `N` in the range `[1, last - first)`, where `M` is a value from some uniform random distribution over the range `[0, N)`. Thus, the function randomly shuffles the order of elements in the sequence.

The second template function behaves the same, except that `M` is `(Dist)f((Dist)N)`, where `Dist` is a type convertible to `iterator_traits (page 302):: difference_type` (page 303).

remove

```
template<class FwdIt, class T>
    FwdIt remove(FwdIt first, FwdIt last, const T& val);
```

The template function effectively assigns `first` to `X`, then executes the statement:

```
if (!(*(first + N) == val))
    *X++ = *(first + N);
```

once for each `N` in the range `[0, last - first)`. Here, `operator==` must impose an equivalence relationship (page 39) between its operands. It then returns `X`. Thus, the function removes from the sequence all elements for which the predicate

`*(first + N) == val` is true, without altering the relative order of remaining elements, and returns the iterator value that designates the end of the revised sequence.

remove_copy

```
template<class InIt, class OutIt, class T>
    OutIt remove_copy(InIt first, InIt last, OutIt x,
                       const T& val);
```

The template function effectively executes the statement:

```
if (!(*(first + N) == val))
    *x++ = *(first + N);
```

once for each `N` in the range `[0, last - first)`. Here, `operator==` must impose an equivalence relationship (page 39) between its operands. It then returns `x`. Thus, the function removes from the sequence all elements for which the predicate `*(first + N) == val` is true, without altering the relative order of remaining elements, and returns the iterator value that designates the end of the revised sequence.

If `x` and `first` designate regions of storage, the range `[x, x + (last - first))` must not overlap the range `[first, last)`.

remove_copy_if

```
template<class InIt, class OutIt, class Pred>
    OutIt remove_copy_if(InIt first, InIt last, OutIt x,
                          Pred pr);
```

The template function effectively executes the statement:

```
if (!pr(*(first + N)))
    *x++ = *(first + N);
```

once for each `N` in the range `[0, last - first)`. It then returns `x`. Thus, the function removes from the sequence all elements for which the predicate `pr(*(first + N))` is true, without altering the relative order of remaining elements, and returns the iterator value that designates the end of the revised sequence.

If `x` and `first` designate regions of storage, the range `[x, x + (last - first))` must not overlap the range `[first, last)`.

remove_if

```
template<class FwdIt, class Pred>
    FwdIt remove_if(FwdIt first, FwdIt last, Pred pr);
```

The template function effectively assigns `first` to `X`, then executes the statement:

```
if (!pr(*(first + N)))
    *X++ = *(first + N);
```

once for each `N` in the range `[0, last - first)`. It then returns `X`. Thus, the function removes from the sequence all elements for which the predicate `pr(*(first + N))` is true, without altering the relative order of remaining elements, and returns the iterator value that designates the end of the revised sequence.

replace

```
template<caass FwdIt, class T>
void replace(FwdIt first, FwdIt last,
             const T& vold, const T& vnew);
```

The template function executes the statement:

```
if (*(first + N) == vold)
    *(first + N) = vnew;
```

once for each N in the range [0, last - first). Here, operator== must impose an equivalence relationship (page 39) between its operands.

replace_copy

```
template<class InIt, class OutIt, class T>
OutIt replace_copy(InIt first, InIt last, OutIt x,
                  const T& vold, const T& vnew);
```

The template function executes the statement:

```
if (*(first + N) == vold)
    *(x + N) = vnew;
else
    *(x + N) = *(first + N)
```

once for each N in the range [0, last - first). Here, operator== must impose an equivalence relationship (page 39) between its operands.

If x and first designate regions of storage, the range [x, x + (last - first)) must not overlap the range [first, last).

replace_copy_if

```
template<class InIt, class OutIt, class Pred, class T>
OutIt replace_copy_if(InIt first, InIt last, OutIt x,
                     Pred pr, const T& val);
```

The template function executes the statement:

```
if (pr(*(first + N)))
    *(x + N) = val;
else
    *(x + N) = *(first + N)
```

once for each N in the range [0, last - first).

If x and first designate regions of storage, the range [x, x + (last - first)) must not overlap the range [first, last).

replace_if

```
template<class FwdIt, class Pred, class T>
void replace_if(FwdIt first, FwdIt last,
               Pred pr, const T& val);
```

The template function executes the statement:

```
if (pr(*(first + N)))
    *(first + N) = val;
```

once for each N in the range [0, last - first).

reverse

```
template<class BidIt>
void reverse(BidIt first, BidIt last);
```

The template function evaluates `swap(*(first + N), *(last - 1 - N))` once for each `N` in the range `[0, (last - first) / 2)`. Thus, the function reverses the order of elements in the sequence.

reverse_copy

```
template<class BidIt, class OutIt>
OutIt reverse_copy(BidIt first, BidIt last, OutIt x);
```

The template function evaluates `*(x + N) = *(last - 1 - N)` once for each `N` in the range `[0, last - first)`. It then returns `x + (last - first)`. Thus, the function reverses the order of elements in the sequence that it copies.

If `x` and `first` designate regions of storage, the range `[x, x + (last - first))` must not overlap the range `[first, last)`.

rotate

```
template<class FwdIt>
void rotate(FwdIt first, FwdIt middle, FwdIt last);
```

The template function leaves the value originally stored in `*(first + (N + (middle - last)) % (last - first))` subsequently stored in `*(first + N)` for each `N` in the range `[0, last - first)`. Thus, if a “left” shift by one element leaves the element originally stored in `*(first + (N + 1) % (last - first))` subsequently stored in `*(first + N)`, then the function can be said to rotate the sequence either left by `middle - first` elements or right by `last - middle` elements. Both `[first, middle)` and `[middle, last)` must be valid ranges. The function swaps at most `last - first` pairs of elements.

rotate_copy

```
template<class FwdIt, class OutIt>
OutIt rotate_copy(FwdIt first, FwdIt middle,
                  FwdIt last, OutIt x);
```

The template function evaluates `*(x + N) = *(first + (N + (middle - first)) % (last - first))` once for each `N` in the range `[0, last - first)`. Thus, if a “left” shift by one element leaves the element originally stored in `*(first + (N + 1) % (last - first))` subsequently stored in `*(first + N)`, then the function can be said to rotate the sequence either left by `middle - first` elements or right by `last - middle` elements as it copies. Both `[first, middle)` and `[middle, last)` must be valid ranges.

If `x` and `first` designate regions of storage, the range `[x, x + (last - first))` must not overlap the range `[first, last)`.

search

```
template<class FwdIt1, class FwdIt2>
FwdIt1 search(FwdIt1 first1, FwdIt1 last1,
              FwdIt2 first2, FwdIt2 last2);
template<class FwdIt1, class FwdIt2, class Pred>
FwdIt1 search(FwdIt1 first1, FwdIt1 last1,
              FwdIt2 first2, FwdIt2 last2, Pred pr);
```

The first template function determines the lowest value of N in the range $[0, (last1 - first1) - (last2 - first2))$ such that for each M in the range $[0, last2 - first2)$, the predicate $*(first1 + N + M) == *(first2 + M)$ is true. Here, `operator==` must impose an equivalence relationship (page 39) between its operands. It then returns $first1 + N$. If no such value exists, the function returns $last1$. It evaluates the predicate at most $(last2 - first2) * (last1 - first1)$ times.

The second template function behaves the same, except that the predicate is `pr(*(first1 + N + M), *(first2 + M))`.

search_n

```
template<class FwdIt, class Dist, class T>
    FwdIt search_n(FwdIt first, FwdIt last,
                  Dist n, const T& val);
template<class FwdIt, class Dist, class T, class Pred>
    FwdIt search_n(FwdIt first, FwdIt last,
                  Dist n, const T& val, Pred pr);
```

The first template function determines the lowest value of N in the range $[0, (last - first) - n)$ such that for each M in the range $[0, n)$, the predicate $*(first + N + M) == val$ is true. Here, `operator==` must impose an equivalence relationship (page 39) between its operands. It then returns $first + N$. If no such value exists, the function returns $last$. It evaluates the predicate at most $n * (last - first)$ times.

The second template function behaves the same, except that the predicate is `pr(*(first + N + M), val)`.

set_difference

```
template<class InIt1, class InIt2, class OutIt>
    OutIt set_difference(InIt1 first1, InIt1 last1,
                       InIt2 first2, InIt2 last2, OutIt x);
template<class InIt1, class InIt2, class OutIt,
        class Pred>
    OutIt set_difference(InIt1 first1, InIt1 last1,
                       InIt2 first2, InIt2 last2, OutIt x, Pred pr);
```

The first template function alternately copies values from two sequences designated by iterators in the ranges $[first1, last1)$ and $[first2, last2)$, both ordered by (page 39) `operator<`, to form a merged sequence of length K beginning at x , also ordered by `operator<`. The function then returns $x + K$.

The merge occurs without altering the relative order of elements within either sequence. Moreover, for two elements from different sequences that have equivalent ordering (page 39) that would otherwise be copied to adjacent elements, the function copies only the element from the ordered range $[first1, last1)$ and skips the other. An element from one sequence that has equivalent ordering with no element from the other sequence is copied from the ordered range $[first1, last1)$ and skipped from the other. Thus, the function merges two ordered sequences to form another ordered sequence that is effectively the difference of two sets.

If x and $first1$ designate regions of storage, the range $[x, x + K)$ must not overlap the range $[first1, last1)$. If x and $first2$ designate regions of storage,

the range $[x, x + K)$ must not overlap the range $[first2, last2)$. The function evaluates the ordering predicate $X < Y$ at most $2 * ((last1 - first1) + (last2 - first2)) - 1$ times.

The second template function behaves the same, except that it replaces `operator<(X, Y)` with `pr(X, Y)`.

set_intersection

```
template<class InIt1, class InIt2, class OutIt>
    OutIt set_intersection(InIt1 first1, InIt1 last1,
                          InIt2 first2, InIt2 last2, OutIt x);
template<class InIt1, class InIt2, class OutIt,
        class Pred>
    OutIt set_intersection(InIt1 first1, InIt1 last1,
                          InIt2 first2, InIt2 last2, OutIt x, Pred pr);
```

The first template function alternately copies values from two sequences designated by iterators in the ranges $[first1, last1)$ and $[first2, last2)$, both ordered by (page 39) `operator<`, to form a merged sequence of length K beginning at x , also ordered by `operator<`. The function then returns $x + K$.

The merge occurs without altering the relative order of elements within either sequence. Moreover, for two elements from different sequences that have equivalent ordering (page 39) that would otherwise be copied to adjacent elements, the function copies only the element from the ordered range $[first1, last1)$ and skips the other. An element from one sequence that has equivalent ordering with no element from the other sequence is also skipped. Thus, the function merges two ordered sequences to form another ordered sequence that is effectively the intersection of two sets.

If x and $first1$ designate regions of storage, the range $[x, x + K)$ must not overlap the range $[first1, last1)$. If x and $first2$ designate regions of storage, the range $[x, x + K)$ must not overlap the range $[first2, last2)$. The function evaluates the ordering predicate $X < Y$ at most $2 * ((last1 - first1) + (last2 - first2)) - 1$ times.

The second template function behaves the same, except that it replaces `operator<(X, Y)` with `pr(X, Y)`.

set_symmetric_difference

```
template<class InIt1, class InIt2, class OutIt>
    OutIt set_symmetric_difference(InIt1 first1,
                                   InIt1 last1, InIt2 first2, InIt2 last2, OutIt x);
template<class InIt1, class InIt2, class OutIt,
        class Pred>
    OutIt set_symmetric_difference(InIt1 first1,
                                   InIt1 last1, InIt2 first2, InIt2 last2, OutIt x,
                                   Pred pr);
```

The first template function alternately copies values from two sequences designated by iterators in the ranges $[first1, last1)$ and $[first2, last2)$, both ordered by (page 39) `operator<`, to form a merged sequence of length K beginning at x , also ordered by `operator<`. The function then returns $x + K$.

The merge occurs without altering the relative order of elements within either sequence. Moreover, for two elements from different sequences that have equivalent ordering (page 39) that would otherwise be copied to adjacent elements,

the function copies neither element. An element from one sequence that has equivalent ordering with no element from the other sequence is copied. Thus, the function merges two ordered sequences to form another ordered sequence that is effectively the symmetric difference of two sets.

If x and first1 designate regions of storage, the range $[x, x + K)$ must not overlap the range $[\text{first1}, \text{last1})$. If x and first2 designate regions of storage, the range $[x, x + K)$ must not overlap the range $[\text{first2}, \text{last2})$. The function evaluates the ordering predicate $X < Y$ at most $2 * ((\text{last1} - \text{first1}) + (\text{last2} - \text{first2})) - 1$ times.

The second template function behaves the same, except that it replaces $\text{operator}<(X, Y)$ with $\text{pr}(X, Y)$.

set_union

```
template<class InIt1, class InIt2, class OutIt>
    OutIt set_union(InIt1 first1, InIt1 last1,
                    InIt2 first2, InIt2 last2, OutIt x);
template<class InIt1, class InIt2, class OutIt,
        class Pred>
    OutIt set_union(InIt1 first1, InIt1 last1,
                    InIt2 first2, InIt2 last2, OutIt x, Pred pr);
```

The first template function alternately copies values from two sequences designated by iterators in the ranges $[\text{first1}, \text{last1})$ and $[\text{first2}, \text{last2})$, both ordered by (page 39) $\text{operator}<$, to form a merged sequence of length K beginning at x , also ordered by $\text{operator}<$. The function then returns $x + K$.

The merge occurs without altering the relative order of elements within either sequence. Moreover, for two elements from different sequences that have equivalent ordering (page 39) that would otherwise be copied to adjacent elements, the function copies only the element from the ordered range $[\text{first1}, \text{last1})$ and skips the other. Thus, the function merges two ordered sequences to form another ordered sequence that is effectively the union of two sets.

If x and first1 designate regions of storage, the range $[x, x + K)$ must not overlap the range $[\text{first1}, \text{last1})$. If x and first2 designate regions of storage, the range $[x, x + K)$ must not overlap the range $[\text{first2}, \text{last2})$. The function evaluates the ordering predicate $X < Y$ at most $2 * ((\text{last1} - \text{first1}) + (\text{last2} - \text{first2})) - 1$ times.

The second template function behaves the same, except that it replaces $\text{operator}<(X, Y)$ with $\text{pr}(X, Y)$.

sort

```
template<class RanIt>
    void sort(RanIt first, RanIt last);
template<class RanIt, class Pred>
    void sort(RanIt first, RanIt last, Pred pr);
```

The first template function reorders the sequence designated by iterators in the range $[\text{first}, \text{last})$ to form a sequence ordered by (page 39) $\text{operator}<$. Thus, the elements are sorted in *ascending* order.

The function evaluates the ordering predicate $X < Y$ at most $\text{ceil}((\text{last} - \text{first}) * \log(\text{last} - \text{first}))$ times.

The second template function behaves the same, except that it replaces `operator<(X, Y)` with `pr(X, Y)`.

sort_heap

```
template<class RanIt>
    void sort_heap(RanIt first, RanIt last);
template<class RanIt, class Pred>
    void sort_heap(RanIt first, RanIt last, Pred pr);
```

The first template function reorders the sequence designated by iterators in the range `[first, last)` to form a sequence that is ordered by (page 39) `operator<`. The original sequence must designate a heap, also ordered by (page 39) `operator<`. Thus, the elements are sorted in *ascending* order.

The function evaluates the ordering predicate $X < Y$ at most $\text{ceil}((\text{last} - \text{first}) * \log(\text{last} - \text{first}))$ times.

The second template function behaves the same, except that it replaces `operator<(X, Y)` with `pr(X, Y)`.

stable_partition

```
template<class BidIt, class Pred>
    BidIt stable_partition(BidIt first, BidIt last,
                           Pred pr);
```

The template function reorders the sequence designated by iterators in the range `[first, last)` and determines the value K such that for each N in the range $[0, K)$ the predicate `pr(*(first + N))` is true, and for each N in the range $[K, \text{last} - \text{first})$ the predicate `pr(*(first + N))` is false. It does so without altering the relative order of either the elements designated by indexes in the range $[0, K)$ or the elements designated by indexes in the range $[K, \text{last} - \text{first})$. The function then returns `first + K`.

The predicate must not alter its operand. The function evaluates `pr(*(first + N))` exactly `last - first` times, and swaps at most $\text{ceil}((\text{last} - \text{first}) * \log(\text{last} - \text{first}))$ pairs of elements. (Given enough temporary storage, it can replace the swaps with at most $2 * (\text{last} - \text{first})$ assignments.)

stable_sort

```
template<class BidIt>
    void stable_sort(BidIt first, BidIt last);
template<class BidIt, class Pred>
    void stable_sort(BidIt first, BidIt last, Pred pr);
```

The first template function reorders the sequence designated by iterators in the range `[first, last)` to form a sequence ordered by (page 39) `operator<`. It does so without altering the relative order of elements that have equivalent ordering (page 39). Thus, the elements are sorted in *ascending* order.

The function evaluates the ordering predicate $X < Y$ at most $\text{ceil}((\text{last} - \text{first}) * (\log(\text{last} - \text{first}))^2)$ times. (Given enough temporary storage, it can evaluate the predicate at most $\text{ceil}((\text{last} - \text{first}) * \log(\text{last} - \text{first}))$ times.)

The second template function behaves the same, except that it replaces `operator<(X, Y)` with `pr(X, Y)`.

swap

```
template<class T>
void swap(T& x, T& y);
```

The template function leaves the value originally stored in *y* subsequently stored in *x*, and the value originally stored in *x* subsequently stored in *y*.

swap_ranges

```
template<class FwdIt1, class FwdIt2>
FwdIt2 swap_ranges(FwdIt1 first, FwdIt1 last,
FwdIt2 x);
```

The template function evaluates `swap(*(first + N), *(x + N))` once for each *N* in the range `[0, last - first)`. It then returns `x + (last - first)`. If *x* and *first* designate regions of storage, the range `[x, x + (last - first))` must not overlap the range `[first, last)`.

transform

```
template<class InIt, class OutIt, class Unop>
OutIt transform(InIt first, InIt last, OutIt x,
Unop uop);
template<class InIt1, class InIt2, class OutIt,
class Binop>
OutIt transform(InIt1 first1, InIt1 last1,
InIt2 first2, OutIt x, Binop bop);
```

The first template function evaluates `*(x + N) = uop(*(first + N))` once for each *N* in the range `[0, last - first)`. It then returns `x + (last - first)`. The call `uop(*(first + N))` must not alter `*(first + N)`.

The second template function evaluates `*(x + N) = bop(*(first1 + N), *(first2 + N))` once for each *N* in the range `[0, last1 - first1)`. It then returns `x + (last1 - first1)`. The call `bop(*(first1 + N), *(first2 + N))` must not alter either `*(first1 + N)` or `*(first2 + N)`.

unique

```
template<class FwdIt>
FwdIt unique(FwdIt first, FwdIt last);
template<class FwdIt, class Pred>
FwdIt unique(FwdIt first, FwdIt last, Pred pr);
```

The first template function effectively assigns *first* to *X*, then executes the statement:

```
if (N == 0 || !(*(first + N) == V))
    V = *(first + N), *X++ = V;
```

once for each *N* in the range `[0, last - first)`. It then returns *X*. Thus, the function repeatedly removes from the sequence the second of a pair of elements for which the predicate `*(first + N) == *(first + N - 1)` is true, until only the first of a sequence of equal elements survives. Here, operator`==` must impose an equivalence relationship (page 39) between its operands. It does so without altering the relative order of remaining elements, and returns the iterator value that designates the end of the revised sequence. The function evaluates the predicate at most `last - first` times.

The second template function behaves the same, except that it executes the statement:

```
if (N == 0 || !pr(*(first + N), V))
    V = *(first + N), *X++ = V;
```

unique_copy

```
template<class InIt, class OutIt>
    OutIt unique_copy(InIt first, InIt last, OutIt x);
template<class InIt, class OutIt, class Pred>
    OutIt unique_copy(InIt first, InIt last, OutIt x,
        Pred pr);
```

The first template function effectively executes the statement:

```
if (N == 0 || !(*(first + N) == V))
    V = *(first + N), *x++ = V;
```

once for each N in the range [0, last - first). It then returns x. Thus, the function repeatedly removes from the sequence it copies the second of a pair of elements for which the predicate $*(first + N) == *(first + N - 1)$ is true, until only the first of a sequence of equal elements survives. Here, `operator==` must impose an equivalence relationship (page 39) between its operands. It does so without altering the relative order of remaining elements, and returns the iterator value that designates the end of the copied sequence.

If x and first designate regions of storage, the range [x, x + (last - first)) must not overlap the range [first, last).

The second template function behaves the same, except that it executes the statement:

```
if (N == 0 || !pr(*(first + N), V))
    V = *(first + N), *x++ = V;
```

upper_bound

```
template<class FwdIt, class T>
    FwdIt upper_bound(FwdIt first, FwdIt last,
        const T& val);
template<class FwdIt, class T, class Pred>
    FwdIt upper_bound(FwdIt first, FwdIt last,
        const T& val, Pred pr);
```

The first template function determines the highest value of N in the range (0, last - first] such that, for each M in the range [0, N) the predicate $!(val < *(first + M))$ is true, where the elements designated by iterators in the range [first, last) form a sequence ordered by (page 39) `operator<`. It then returns first + N. Thus, the function determines the highest position before which val can be inserted in the sequence and still preserve its ordering.

If FwdIt is a random-access iterator type, the function evaluates the ordering predicate $X < Y$ at most $\text{ceil}(\log(\text{last} - \text{first})) + 1$ times. Otherwise, the function evaluates the predicate a number of times proportional to last - first.

The second template function behaves the same, except that it replaces `operator<(X, Y)` with `pr(X, Y)`.

<deque>

```
namespace std {
template<class T, class A>
    class deque;

    // TEMPLATE FUNCTIONS
template<class T, class A>
    bool operator==(
        const deque<T, A>& lhs,
        const deque<T, A>& rhs);
template<class T, class A>
    bool operator!=(
        const deque<T, A>& lhs,
        const deque<T, A>& rhs);
template<class T, class A>
    bool operator<(
        const deque<T, A>& lhs,
        const deque<T, A>& rhs);
template<class T, class A>
    bool operator>(
        const deque<T, A>& lhs,
        const deque<T, A>& rhs);
template<class T, class A>
    bool operator<=(
        const deque<T, A>& lhs,
        const deque<T, A>& rhs);
template<class T, class A>
    bool operator>=(
        const deque<T, A>& lhs,
        const deque<T, A>& rhs);
template<class T, class A>
    void swap(
        deque<T, A>& lhs,
        deque<T, A>& rhs);
};
```

Include the STL (page 1) standard header **<deque>** to define the container (page 41) template class **deque** and several supporting templates.

deque

allocator_type (page 276) · **assign** (page 276) · **at** (page 276) · **back** (page 276) · **begin** (page 276) · **clear** (page 277) · **const_iterator** (page 277) · **const_pointer** (page 277) · **const_reference** (page 277) · **const_reverse_iterator** (page 277) · **deque** (page 277) · **difference_type** (page 277) · **empty** (page 278) · **end** (page 278) · **erase** (page 278) · **front** (page 278) · **get_allocator** (page 278) · **insert** (page 278) · **iterator** (page 279) · **max_size** (page 279) · **operator[]** (page 279) · **pointer** (page 279) · **pop_back** (page 279) · **pop_front** (page 280) · **push_back** (page 280) · **push_front** (page 280) · **rbegin** (page 280) · **reference** (page 280) · **rend** (page 280) · **resize** (page 280) · **reverse_iterator** (page 281) · **size** (page 281) · **size_type** (page 281) · **swap** (page 281) · **value_type** (page 281)

```
template<class T, class A = allocator<T> >
    class deque {
public:
    typedef A allocator_type;
    typedef typename A::pointer pointer;
    typedef typename A::const_pointer const_pointer;
    typedef typename A::reference reference;
    typedef typename A::const_reference const_reference;
    typedef typename A::value_type value_type;
    typedef T0 iterator;
    typedef T1 const_iterator;
    typedef T2 size_type;
```

```

typedef T3 difference_type;
typedef reverse_iterator<const_iterator>
    const_reverse_iterator;
typedef reverse_iterator<iterator>
    reverse_iterator;
deque();
explicit deque(const A& a1);
explicit deque(size_type n);
deque(size_type n, const T& v);
deque(size_type n, const T& v,
    const A& a1);
deque(const deque& x);
template<class InIt>
    deque(InIt first, InIt last);
template<class InIt>
    deque(InIt first, InIt last, const A& a1);
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;
void resize(size_type n);
void resize(size_type n, T x);
size_type size() const;
size_type max_size() const;
bool empty() const;
A get_allocator() const;
reference at(size_type pos);
const_reference at(size_type pos) const;
reference operator[](size_type pos);
const_reference operator[](size_type pos);
reference front();
const_reference front() const;
reference back();
const_reference back() const;
void push_front(const T& x);
void pop_front();
void push_back(const T& x);
void pop_back();
template<class InIt>
    void assign(InIt first, InIt last);
void assign(size_type n, const T& x);
iterator insert(iterator it, const T& x);
void insert(iterator it, size_type n, const T& x);
template<class InIt>
    void insert(iterator it, InIt first, InIt last);
iterator erase(iterator it);
iterator erase(iterator first, iterator last);
void clear();
void swap(deque& x);
};

```

The template class describes an object that controls a varying-length sequence of elements of type `T`. The sequence is represented in a way that permits insertion and removal of an element at either end with a single element copy (constant time). Such operations in the middle of the sequence require element copies and assignments proportional to the number of elements in the sequence (linear time).

The object allocates and frees storage for the sequence it controls through a stored allocator object (page 337) of class `A`. Such an allocator object must have the same external interface as an object of template class `allocator` (page 337). Note that the stored allocator object is *not* copied when the container object is assigned.

Deque reallocation occurs when a member function must insert or erase elements of the controlled sequence:

- If an element is inserted into an empty sequence, or if an element is erased to leave an empty sequence, then iterators earlier returned by `begin()` and `end()` become **invalid**.
- If an element is inserted at `first()`, then all iterators but no references, that designate existing elements become invalid.
- If an element is inserted at `end()`, then `end()` and all iterators, but no references, that designate existing elements become invalid.
- If an element is erased at `first()`, only that iterator and references to the erased element become invalid.
- If an element is erased at `last() - 1`, only that iterator, `last()`, and references to the erased element become invalid.
- Otherwise, inserting or erasing an element invalidates all iterators and references.

deque::allocator_type

```
typedef A allocator_type;
```

The type is a synonym for the template parameter A.

deque::assign

```
template<class InIt>
    void assign(InIt first, InIt last);
void assign(size_type n, const T& x);
```

If `InIt` is an integer type, the first member function behaves the same as `assign((size_type)first, (T)last)`. Otherwise, the first member function replaces the sequence controlled by `*this` with the sequence `[first, last)`, which must *not* overlap the initial controlled sequence. The second member function replaces the sequence controlled by `*this` with a repetition of `n` elements of value `x`.

deque::at

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

The member function returns a reference to the element of the controlled sequence at position `pos`. If that position is invalid, the function throws an object of class `out_of_range`.

deque::back

```
reference back();
const_reference back() const;
```

The member function returns a reference to the last element of the controlled sequence, which must be non-empty.

deque::begin

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

The member function returns a random-access iterator that points at the first element of the sequence (or just beyond the end of an empty sequence).

deque::clear

```
void clear();
```

The member function calls `erase(begin(), end())`.

deque::const_iterator

```
typedef T1 const_iterator;
```

The type describes an object that can serve as a constant random-access iterator for the controlled sequence. It is described here as a synonym for the implementation-defined type `T1`.

deque::const_pointer

```
typedef typename A::const_pointer const_pointer;
```

The type describes an object that can serve as a constant pointer to an element of the controlled sequence.

deque::const_reference

```
typedef typename A::const_reference const_reference;
```

The type describes an object that can serve as a constant reference to an element of the controlled sequence.

deque::const_reverse_iterator

```
typedef reverse_iterator<const_iterator>
    const_reverse_iterator;
```

The type describes an object that can serve as a constant reverse random-access iterator for the controlled sequence.

deque::deque

```
deque();
explicit deque(const A& a1);
explicit deque(size_type n);
deque(size_type n, const T& v);
deque(size_type n, const T& v,
      const A& a1);
deque(const deque& x);
template<class InIt>
    deque(InIt first, InIt last);
template<class InIt>
    deque(InIt first, InIt last, const A& a1);
```

All constructors store an allocator object (page 337) and initialize the controlled sequence. The allocator object is the argument `a1`, if present. For the copy constructor, it is `x.get_allocator()`. Otherwise, it is `A()`.

The first two constructors specify an empty initial controlled sequence. The third constructor specifies a repetition of `n` elements of value `T()`. The fourth and fifth constructors specify a repetition of `n` elements of value `x`. The sixth constructor specifies a copy of the sequence controlled by `x`. If `InIt` is an integer type, the last two constructors specify a repetition of `(size_type)first` elements of value `(T)last`. Otherwise, the last two constructors specify the sequence `[first, last)`.

deque::difference_type

```
typedef T3 difference_type;
```

The signed integer type describes an object that can represent the difference between the addresses of any two elements in the controlled sequence. It is described here as a synonym for the implementation-defined type T3.

deque::empty

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

The member function returns true for an empty controlled sequence.

deque::end

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

The member function returns a random-access iterator that points just beyond the end of the sequence.

deque::erase

```
iterator erase(iterator it);  
iterator erase(iterator first, iterator last);
```

The first member function removes the element of the controlled sequence pointed to by it. The second member function removes the elements of the controlled sequence in the range [first, last). Both return an iterator that designates the first element remaining beyond any elements removed, or end() if no such element exists.

Removing N elements causes N destructor calls and an assignment for each of the elements between the insertion point and the nearer end of the sequence. Removing an element at either end invalidates (page 279) only iterators and references that designate the erased elements. Otherwise, erasing an element invalidates all iterators and references.

The member functions never throw an exception.

deque::front

```
reference front();  
const_reference front() const;
```

The member function returns a reference to the first element of the controlled sequence, which must be non-empty.

deque::get_allocator

```
A get_allocator() const;
```

The member function returns the stored allocator object (page 337).

deque::insert

```
iterator insert(iterator it, const T& x);  
void insert(iterator it, size_type n, const T& x);  
template<class InIt>  
    void insert(iterator it, InIt first, InIt last);
```

Each of the member functions inserts, before the element pointed to by it in the controlled sequence, a sequence specified by the remaining operands. The first member function inserts a single element with value x and returns an iterator that points to the newly inserted element. The second member function inserts a repetition of n elements of value x.

If `InIt` is an integer type, the last member function behaves the same as `insert(it, (size_type)first, (T)last)`. Otherwise, the last member function inserts the sequence `[first, last)`, which must *not* overlap the initial controlled sequence.

When inserting a single element, the number of element copies is linear in the number of elements between the insertion point and the nearer end of the sequence. When inserting a single element at either end of the sequence, the amortized number of element copies is constant. When inserting `N` elements, the number of element copies is linear in `N` plus the number of elements between the insertion point and the nearer end of the sequence — except when the template member is specialized for `InIt` an input or forward iterator, which behaves like `N` single insertions. Inserting an element at either end invalidates (page 276) all iterators, but no references, that designate existing elements. Otherwise, inserting an element invalidates all iterators and references.

If an exception is thrown during the insertion of a single element, the container is left unaltered and the exception is rethrown. If an exception is thrown during the insertion of multiple elements, and the exception is not thrown while copying an element, the container is left unaltered and the exception is rethrown.

deque::iterator

```
typedef T0 iterator;
```

The type describes an object that can serve as a random-access iterator for the controlled sequence. It is described here as a synonym for the implementation-defined type `T0`.

deque::max_size

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

The member function returns the length of the longest sequence that the object can control.

deque::operator[]

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

The member function returns a reference to the element of the controlled sequence at position `pos`. If that position is invalid, the behavior is undefined.

deque::pointer

```
typedef typename A::pointer pointer;
```

The type describes an object that can serve as a pointer to an element of the controlled sequence.

deque::pop_back

```
void pop_back();
```

The member function removes the last element of the controlled sequence, which must be non-empty. Removing the element invalidates (page 276) only iterators and references that designate the erased element.

The member function never throws an exception.

deque::pop_front

```
void pop_front();
```

The member function removes the first element of the controlled sequence, which must be non-empty. Removing the element invalidates (page 276) only iterators and references that designate the erased element.

The member function never throws an exception.

deque::push_back

```
void push_back(const T& x);
```

The member function inserts an element with value *x* at the end of the controlled sequence. Inserting the element invalidates (page 276) all iterators, but no references, to existing elements.

If an exception is thrown, the container is left unaltered and the exception is rethrown.

deque::push_front

```
void push_front(const T& x);
```

The member function inserts an element with value *x* at the beginning of the controlled sequence. Inserting the element invalidates (page 276) all iterators, but no references, to existing elements.

If an exception is thrown, the container is left unaltered and the exception is rethrown.

deque::rbegin

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

The member function returns a reverse iterator that points just beyond the end of the controlled sequence. Hence, it designates the beginning of the reverse sequence.

deque::reference

```
typedef typename A::reference reference;
```

The type describes an object that can serve as a reference to an element of the controlled sequence.

deque::rend

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

The member function returns a reverse iterator that points at the first element of the sequence (or just beyond the end of an empty sequence). Hence, it designates the end of the reverse sequence.

deque::resize

```
void resize(size_type n);  
void resize(size_type n, T x);
```

The member functions both ensure that `size()` henceforth returns *n*. If it must make the controlled sequence longer, the first member function appends elements

with value `T()`, while the second member function appends elements with value `x`. To make the controlled sequence shorter, both member functions call `erase(begin() + n, end())`.

deque::reverse_iterator

```
typedef reverse_iterator<iterator>
    reverse_iterator;
```

The type describes an object that can serve as a reverse random-access iterator for the controlled sequence.

deque::size

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

The member function returns the length of the controlled sequence.

deque::size_type

```
typedef T2 size_type;
```

The unsigned integer type describes an object that can represent the length of any controlled sequence. It is described here as a synonym for the implementation-defined type `T2`.

deque::swap

```
void swap(deque& x);
```

The member function swaps the controlled sequences between `*this` and `x`. If `get_allocator() == x.get_allocator()`, it does so in constant time, it throws no exceptions, and it invalidates no references, pointers, or iterators that designate elements in the two controlled sequences. Otherwise, it performs a number of element assignments and constructor calls proportional to the number of elements in the two controlled sequences.

deque::value_type

```
typedef typename A::value_type value_type;
```

The type is a synonym for the template parameter `T`.

operator!=

```
template<class T, class A>
    bool operator!=(
        const deque<T, A>& lhs,
        const deque<T, A>& rhs);
```

The template function returns `!(lhs == rhs)`.

operator==

```
template<class T, class A>
    bool operator==(
        const deque<T, A>& lhs,
        const deque<T, A>& rhs);
```

The template function overloads `operator==` to compare two objects of template class `deque` (page 274). The function returns `lhs.size() == rhs.size() && equal(lhs.begin(), lhs.end(), rhs.begin())`.

operator<

```
template<class T, class A>
bool operator<(  
    const deque<T, A>& lhs,  
    const deque<T, A>& rhs);
```

The template function overloads operator< to compare two objects of template class deque. The function returns `lexicographical_compare(lhs.begin(), lhs.end(), rhs.begin(), rhs.end())`.

operator<=

```
template<class T, class A>
bool operator<=(  
    const deque<T, A>& lhs,  
    const deque<T, A>& rhs);
```

The template function returns `!(rhs < lhs)`.

operator>

```
template<class T, class A>
bool operator>(  
    const deque<T, A>& lhs,  
    const deque<T, A>& rhs);
```

The template function returns `rhs < lhs`.

operator>=

```
template<class T, class A>
bool operator>=(  
    const deque<T, A>& lhs,  
    const deque<T, A>& rhs);
```

The template function returns `!(lhs < rhs)`.

swap

```
template<class T, class A>
void swap(  
    deque<T, A>& lhs,  
    deque<T, A>& rhs);
```

The template function executes `lhs.swap (page 281)(rhs)`.

Portions derived from work copyright © 1994 by Hewlett-Packard Company. All rights reserved.

<functional>

`binary_function (page 285) · binary_negate (page 285) · binder1st (page 286) · binder2nd (page 286) · const_mem_fun_t (page 286) · const_mem_fun_ref_t (page 287) · const_mem_fun1_t (page 287) · const_mem_fun1_ref_t (page 287) · divides (page 287) · equal_to (page 287) · greater (page 288) · greater_equal (page 288) · hash (page 288) · less (page 288) · less_equal (page 288) · logical_and (page 288) · logical_not (page 289) · logical_or (page 289) · mem_fun_t (page 289) · mem_fun_ref_t (page 290) · mem_fun1_t (page 290) · mem_fun1_ref_t (page 290) · minus (page 290) · modulus (page 290) · multiplies (page 290) · negate (page 291)`

· **not_equal_to** (page 291) · **plus** (page 291) · **pointer_to_binary_function** (page 291) · **pointer_to_unary_function** (page 291) · **unary_function** (page 292) · **unary_negate** (page 292)

bind1st (page 285) · **bind2nd** (page 285) · **mem_fun** (page 289) · **mem_fun_ref** (page 289) · **not1** (page 291) · **not2** (page 291) · **ptr_fun** (page 292)

```
namespace std {
    template<class Arg, class Result>
        struct unary_function;
    template<class Arg1, class Arg2, class Result>
        struct binary_function;
    template<class T>
        struct plus;
    template<class T>
        struct minus;
    template<class T>
        struct multiplies;
    template<class T>
        struct divides;
    template<class T>
        struct modulus;
    template<class T>
        struct negate;
    template<class T>
        struct equal_to;
    template<class T>
        struct not_equal_to;
    template<class T>
        struct greater;
    template<class T>
        struct less;
    template<class T>
        struct greater_equal;
    template<class T>
        struct less_equal;
    template<class T>
        struct logical_and;
    template<class T>
        struct logical_or;
    template<class T>
        struct logical_not;
    template<class Pred>
        struct unary_negate;
    template<class Pred>
        struct binary_negate;
    template<class Pred>
        class binder1st;
    template<class Pred>
        class binder2nd;
    template<class Arg, class Result>
        class pointer_to_unary_function;
    template<class Arg1, class Arg2, class Result>
        class pointer_to_binary_function;
    template<class R, class T>
        struct mem_fun_t;
    template<class R, class T, class A>
        struct mem_fun1_t;
    template<class R, class T>
        struct const_mem_fun_t;
    template<class R, class T, class A>
        struct const_mem_fun1_t;
    template<class R, class T>
        struct mem_fun_ref_t;
    template<class R, class T, class A>
        struct mem_fun1_ref_t;
    template<class R, class T>
```

```

    struct const_mem_fun_ref_t;
template<class R, class T, class A>
    struct const_mem_fun1_ref_t;

    // TEMPLATE FUNCTIONS
template<class Pred>
    unary_negate<Pred> not1(const Pred& pr);
template<class Pred>
    binary_negate<Pred> not2(const Pred& pr);
template<class Pred, class T>
    binder1st<Pred> bind1st(const Pred& pr, const T& x);
template<class Pred, class T>
    binder2nd<Pred> bind2nd(const Pred& pr, const T& x);
template<class Arg, class Result>
    pointer_to_unary_function<Arg, Result>
        ptr_fun(Result (*)(Arg));
template<class Arg1, class Arg2, class Result>
    pointer_to_binary_function<Arg1, Arg2, Result>
        ptr_fun(Result (*)(Arg1, Arg2));
template<class R, class T>
    mem_fun_t<R, T> mem_fun(R (T::*pm)());
template<class R, class T, class A>
    mem_fun1_t<R, T, A> mem_fun(R (T::*pm)(A arg));
template<class R, class T>
    const_mem_fun_t<R, T> mem_fun(R (T::*pm)() const);
template<class R, class T, class A>
    const_mem_fun1_t<R, T, A> mem_fun(R (T::*pm)(A arg) const);
template<class R, class T>
    mem_fun_ref_t<R, T> mem_fun_ref(R (T::*pm)());
template<class R, class T, class A>
    mem_fun1_ref_t<R, T, A>
        mem_fun_ref(R (T::*pm)(A arg));
template<class R, class T>
    const_mem_fun_ref_t<R, T> mem_fun_ref(R (T::*pm)() const);
template<class R, class T, class A>
    const_mem_fun1_ref_t<R, T, A>
        mem_fun_ref(R (T::*pm)(A arg) const);
}

#ifdef _VACPP_TR1

namespace tr1 {

template <class _Ty>
    struct hash;

template <> struct hash<bool>;
template <> struct hash<char>;
template <> struct hash<signed char>;
template <> struct hash<unsigned char>;
template <> struct hash<wchar_t>;
template <> struct hash<short>;
template <> struct hash<unsigned short>;
template <> struct hash<int>;
template <> struct hash<unsigned int>;
template <> struct hash<long>;
template <> struct hash<unsigned long>;
template <> struct hash<long long>;
template <> struct hash<unsigned long long>;
template <> struct hash<float>;
template <> struct hash<double>;
template <> struct hash<long double>;
template <class _T> struct hash<_T *>;

template <class _CharT, class _Traits, class _Alloc>
    struct hash<basic_string<_CharT, _Traits, _Alloc> >;

```

```

}

#endif /* def _VACPP_TR1 */

```

Include the STL (page 1) standard header **<functional>** to define several templates that help construct **function objects**, objects of a type that defines operator(). A function object can thus be a function pointer, but in the more general case the object can store additional information that can be used during a function call.

binary_function

```

template<class Arg1, class Arg2, class Result>
struct binary_function {
    typedef Arg1 first_argument_type;
    typedef Arg2 second_argument_type;
    typedef Result result_type;
};

```

The template class serves as a base for classes that define a member function of the form:

```

result_type operator()(const first_argument_type&,
    const second_argument_type&) const

```

Hence, all such **binary functions** can refer to their first argument type as **first_argument_type**, their second argument type as **second_argument_type**, and their return type as **result_type**.

binary_negate

```

template<class Pred>
class binary_negate
    : public binary_function<
        typename Pred::first_argument_type,
        typename Pred::second_argument_type, bool> {
public:
    explicit binary_negate(const Pred& pr);
    bool operator()(
        const typename Pred::first_argument_type& x,
        const typename Pred::second_argument_type& y) const;
};

```

The template class stores a copy of pr, which must be a binary function (page 285) object. It defines its member function **operator()** as returning !pr(x, y).

bind1st

```

template<class Pred, class T>
binder1st<Pred> bind1st(const Pred& pr, const T& x);

```

The function returns binder1st<Pred>(pr, typename Pred::first_argument_type(x)).

bind2nd

```

template<class Pred, class T>
binder2nd<Pred> bind2nd(const Pred& pr, const T& y);

```

The function returns binder2nd<Pred>(pr, typename Pred::second_argument_type(y)).

binder1st

```
template<class Pred>
class binder1st
    : public unary_function<
        typename Pred::second_argument_type,
        typename Pred::result_type> {
public:
    typedef typename Pred::second_argument_type argument_type;
    typedef typename Pred::result_type result_type;
    binder1st(const Pred& pr,
        const typename Pred::first_argument_type& x);
    result_type operator()(const argument_type& y) const;
protected:
    Pred op;
    typename Pred::first_argument_type value;
};
```

The template class stores a copy of *pr*, which must be a binary function (page 285) object, in **op**, and a copy of *x* in **value**. It defines its member function **operator()** as returning *op*(*value*, *y*).

binder2nd

```
template<class Pred>
class binder2nd
    : public unary_function<
        typename Pred::first_argument_type,
        typename Pred::result_type> {
public:
    typedef typename Pred::first_argument_type argument_type;
    typedef typename Pred::result_type result_type;
    binder2nd(const Pred& pr,
        const typename Pred::second_argument_type& y);
    result_type operator()(const argument_type& x) const;
protected:
    Pred op;
    typename Pred::second_argument_type value;
};
```

The template class stores a copy of *pr*, which must be a binary function (page 285) object, in **op**, and a copy of *y* in **value**. It defines its member function **operator()** as returning *op*(*x*, *value*).

const_mem_fun_t

```
template<class R, class T>
struct const_mem_fun_t
    : public unary_function<T *, R> {
    explicit const_mem_fun_t(R (T::*pm)() const);
    R operator()(const T *p) const;
};
```

The template class stores a copy of *pm*, which must be a pointer to a member function of class *T*, in a private member object. It defines its member function **operator()** as returning *(p->*pm)()* const.

const_mem_fun_ref_t

```
template<class R, class T>
struct const_mem_fun_ref_t
: public unary_function<T, R> {
    explicit const_mem_fun_t(R (T::*pm)() const);
    R operator()(const T& x) const;
};
```

The template class stores a copy of `pm`, which must be a pointer to a member function of class `T`, in a private member object. It defines its member function **operator()** as returning `(x.*pm)() const`.

const_mem_fun1_t

```
template<class R, class T, class A>
struct const_mem_fun1_t
: public binary_function<T *, A, R> {
    explicit const_mem_fun1_t(R (T::*pm)(A) const);
    R operator()(const T *p, A arg) const;
};
```

The template class stores a copy of `pm`, which must be a pointer to a member function of class `T`, in a private member object. It defines its member function **operator()** as returning `(p->*pm)(arg) const`.

const_mem_fun1_ref_t

```
template<class R, class T, class A>
struct const_mem_fun1_ref_t
: public binary_function<T, A, R> {
    explicit const_mem_fun1_ref_t(R (T::*pm)(A) const);
    R operator()(const T& x, A arg) const;
};
```

The template class stores a copy of `pm`, which must be a pointer to a member function of class `T`, in a private member object. It defines its member function **operator()** as returning `(x.*pm)(arg) const`.

divides

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

The template class defines its member function as returning `x / y`.

equal_to

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

The template class defines its member function as returning `x == y`.

greater

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

The template class defines its member function as returning $x > y$. The member function defines a total ordering (page 401), even if T is an object pointer type.

greater_equal

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

The template class defines its member function as returning $x \geq y$. The member function defines a total ordering (page 401), even if T is an object pointer type.

hash

```
namespace tr1 {
template <class T>
    struct hash : public std::unary_function<T, std::size_t>
    {
        std::size_t operator()(T val) const;
    };
}
```

The template class is used as the default hash function by the *hashed associative containers*. Its member function **operator()** returns an unspecified value, but equal arguments yield the same result.

less

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

The template class defines its member function as returning $x < y$. The member function defines a total ordering (page 401), even if T is an object pointer type.

less_equal

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

The template class defines its member function as returning $x \leq y$. The member function defines a total ordering (page 401), even if T is an object pointer type.

logical_and

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


The template class defines its member function as returning `x && y`.

logical_not

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

The template class defines its member function as returning `!x`.

logical_or

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

The template class defines its member function as returning `x || y`.

mem_fun

```
template<class R, class T>
    mem_fun_t<R, T> mem_fun(R (T::*pm)());
template<class R, class T, class A>
    mem_fun1_t<R, T, A> mem_fun(R (T::*pm)(A));
template<class R, class T>
    const_mem_fun_t<R, T> mem_fun(R (T::*pm)() const);
template<class R, class T, class A>
    const_mem_fun1_t<R, T, A> mem_fun(R (T::*pm)(A) const);
```

The template function returns `pm` cast to the return type.

mem_fun_ref

```
template<class R, class T>
    mem_fun_ref_t<R, T> mem_fun_ref(R (T::*pm)());
template<class R, class T, class A>
    mem_fun1_ref_t<R, T, A> mem_fun_ref(R (T::*pm)(A));
template<class R, class T>
    const_mem_fun_ref_t<R, T> mem_fun_ref(R (T::*pm)() const);
template<class R, class T, class A>
    const_mem_fun1_ref_t<R, T, A> mem_fun_ref(R (T::*pm)(A) const);
```

The template function returns `pm` cast to the return type.

mem_fun_t

```
template<class R, class T>
struct mem_fun_t : public unary_function<T *, R> {
    explicit mem_fun_t(R (T::*pm)());
    R operator()(T *p) const;
};
```

The template class stores a copy of `pm`, which must be a pointer to a member function of class `T`, in a private member object. It defines its member function **operator()** as returning `(p->*pm)()`.

mem_fun_ref_t

```
template<class R, class T>
struct mem_fun_ref_t
    : public unary_function<T, R> {
    explicit mem_fun_ref_t(R (T::*pm)());
    R operator()(T& x) const;
};
```

The template class stores a copy of `pm`, which must be a pointer to a member function of class `T`, in a private member object. It defines its member function `operator()` as returning `(x.*pm)()`.

mem_fun1_t

```
template<class R, class T, class A>
struct mem_fun1_t
    : public binary_function<T *, A, R> {
    explicit mem_fun1_t(R (T::*pm)(A));
    R operator()(T *p, A arg) const;
};
```

The template class stores a copy of `pm`, which must be a pointer to a member function of class `T`, in a private member object. It defines its member function `operator()` as returning `(p->*pm)(arg)`.

mem_fun1_ref_t

```
template<class R, class T, class A>
struct mem_fun1_ref_t
    : public binary_function<T, A, R> {
    explicit mem_fun1_ref_t(R (T::*pm)(A));
    R operator()(T& x, A arg) const;
};
```

The template class stores a copy of `pm`, which must be a pointer to a member function of class `T`, in a private member object. It defines its member function `operator()` as returning `(x.*pm)(arg)`.

minus

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

The template class defines its member function as returning `x - y`.

modulus

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

The template class defines its member function as returning `x % y`.

multiplies

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

The template class defines its member function as returning $x * y$.

negate

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

The template class defines its member function as returning $-x$.

not1

```
template<class Pred>
unary_negate<Pred> not1(const Pred& pr);
```

The template function returns `unary_negate<Pred>(pr)`.

not2

```
template<class Pred>
binary_negate<Pred> not2(const Pred& pr);
```

The template function returns `binary_negate<Pred>(pr)`.

not_equal_to

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

The template class defines its member function as returning $x != y$.

plus

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

The template class defines its member function as returning $x + y$.

pointer_to_binary_function

```
template<class Arg1, class Arg2, class Result>
class pointer_to_binary_function
    : public binary_function<Arg1, Arg2, Result> {
public:
    explicit pointer_to_binary_function(
        Result (*pf)(Arg1, Arg2));
    Result operator()(const Arg1 x, const Arg2 y) const;
};
```

The template class stores a copy of `pf`. It defines its member function `operator()` as returning `(*pf)(x, y)`.

pointer_to_unary_function

```
template<class Arg, class Result>
class pointer_to_unary_function
    : public unary_function<Arg, Result> {
```

```

public:
    explicit pointer_to_unary_function(
        Result (*pf)(Arg));
    Result operator()(const Arg x) const;
};

```

The template class stores a copy of pf. It defines its member function **operator()** as returning (*pf)(x).

ptr_fun

```

template<class Arg, class Result>
    pointer_to_unary_function<Arg, Result>
        ptr_fun(Result (*pf)(Arg));
template<class Arg1, class Arg2, class Result>
    pointer_to_binary_function<Arg1, Arg2, Result>
        ptr_fun(Result (*pf)(Arg1, Arg2));

```

The first template function returns **pointer_to_unary_function**<Arg, Result>(pf).

The second template function returns **pointer_to_binary_function**<Arg1, Arg2, Result>(pf).

unary_function

```

template<class Arg, class Result>
    struct unary_function {
        typedef Arg argument_type;
        typedef Result result_type;
    };

```

The template class serves as a base for classes that define a member function of the form:

```
result_type operator()(const argument_type&) const
```

Hence, all such **unary functions** can refer to their sole argument type as **argument_type** and their return type as **result_type**.

unary_negate

```

template<class Pred>
    class unary_negate
        : public unary_function<
            typename Pred::argument_type,
            bool> {
public:
    explicit unary_negate(const Pred& pr);
    bool operator()(
        const typename Pred::argument_type& x) const;
};

```

The template class stores a copy of pr, which must be a unary function (page 292) object. It defines its member function **operator()** as returning !pr(x).

Portions derived from work copyright © 1994 by Hewlett-Packard Company. All rights reserved.

<iterator>

[advance](#) (page 294) · [back_insert_iterator](#) (page 294) · [back_inserter](#) (page 296) · [bidirectional_iterator_tag](#) (page 296) · [distance](#) (page 296) · [forward_iterator_tag](#) (page 296) · [front_insert_iterator](#) (page 296) · [front_inserter](#) (page 297) · [input_iterator_tag](#) (page 298) · [insert_iterator](#) (page 298) · [inserter](#) (page 299) · [istream_iterator](#) (page 299) · [istreambuf_iterator](#) (page 300) · [iterator](#) (page 302) · [iterator_traits](#) (page 302) · [operator!=](#) (page 303) · [operator==](#) (page 303) · [operator<](#) (page 304) · [operator<=](#) (page 304) · [operator>](#) (page 304) · [operator>=](#) (page 304) · [operator+](#) (page 304) · [operator-](#) (page 304) · [ostream_iterator](#) (page 304) · [ostreambuf_iterator](#) (page 306) · [output_iterator_tag](#) (page 307) · [random_access_iterator_tag](#) (page 307) · [reverse_iterator](#) (page 307)

```
namespace std {
    struct input_iterator_tag;
    struct output_iterator_tag;
    struct forward_iterator_tag;
    struct bidirectional_iterator_tag;
    struct random_access_iterator_tag;

    // TEMPLATE CLASSES
    template<class C, class T, class Dist,
             class Pt, class Rt>
        struct iterator;
    template<class It>
        struct iterator_traits;
    template<class T>
        struct iterator_traits<T *>
    template<class RanIt>
        class reverse_iterator;
    template<class Cont>
        class back_insert_iterator;
    template<class Cont>
        class front_insert_iterator;
    template<class Cont>
        class insert_iterator;
    template<class U, class E, class T, class Dist>
        class istream_iterator;
    template<class U, class E, class T>
        class ostream_iterator;
    template<class E, class T>
        class istreambuf_iterator;
    template<class E, class T>
        class ostreambuf_iterator;

    // TEMPLATE FUNCTIONS
    template<class RanIt>
        bool operator==(
            const reverse_iterator<RanIt>& lhs,
            const reverse_iterator<RanIt>& rhs);
    template<class U, class E, class T, class Dist>
        bool operator==(
            const istream_iterator<U, E, T, Dist>& lhs,
            const istream_iterator<U, E, T, Dist>& rhs);
    template<class E, class T>
        bool operator==(
            const istreambuf_iterator<E, T>& lhs,
            const istreambuf_iterator<E, T>& rhs);
    template<class RanIt>
        bool operator!=(
            const reverse_iterator<RanIt>& lhs,
            const reverse_iterator<RanIt>& rhs);
    template<class U, class E, class T, class Dist>
        bool operator!=(
            const istream_iterator<U, E, T, Dist>& lhs,
```

```

        const istream_iterator<U, E, T, Dist>& rhs);
template<class E, class T>
    bool operator!=(
        const istreambuf_iterator<E, T>& lhs,
        const istreambuf_iterator<E, T>& rhs);
template<class RanIt>
    bool operator<(
        const reverse_iterator<RanIt>& lhs,
        const reverse_iterator<RanIt>& rhs);
template<class RanIt>
    bool operator>(
        const reverse_iterator<RanIt>& lhs,
        const reverse_iterator<RanIt>& rhs);
template<class RanIt>
    bool operator<=(
        const reverse_iterator<RanIt>& lhs,
        const reverse_iterator<RanIt>& rhs);
template<class RanIt>
    bool operator>=(
        const reverse_iterator<RanIt>& lhs,
        const reverse_iterator<RanIt>& rhs);
template<class RanIt>
    Dist operator-(
        const reverse_iterator<RanIt>& lhs,
        const reverse_iterator<RanIt>& rhs);
template<class RanIt>
    reverse_iterator<RanIt> operator+(
        Dist n,
        const reverse_iterator<RanIt>& rhs);
template<class Cont>
    back_insert_iterator<Cont> back_inserter(Cont& x);
template<class Cont>
    front_insert_iterator<Cont> front_inserter(Cont& x);
template<class Cont, class Iter>
    insert_iterator<Cont> inserter(Cont& x, Iter it);
template<class InIt, class Dist>
    void advance(InIt& it, Dist n);
template<class InIt, class Dist>
    iterator_traits<InIt>::difference_type
    distance(InIt first, InIt last);
};

```

Include the STL (page 1) standard header **<iterator>** to define a number of classes, template classes, and template functions that aid in the declaration and manipulation of iterators.

advance

```

template<class InIt, class Dist>
    void advance(InIt& it, Dist n);

```

The template function effectively advances it by incrementing it n times. If InIt is a random-access iterator type, the function evaluates the expression `it += n`. Otherwise, it performs each increment by evaluating `++it`. If InIt is an input or forward iterator type, n must not be negative.

back_insert_iterator

```

template<class Cont>
    class back_insert_iterator
        : public iterator<output_iterator_tag,
            void, void, void, void> {
public:
    typedef Cont container_type;
    typedef typename Cont::reference reference;

```

```

typedef typename Cont::value_type value_type;
explicit back_insert_iterator(Cont& x);
back_insert_iterator&
    operator=(typename Cont::const_reference val);
back_insert_iterator& operator*();
back_insert_iterator& operator++();
back_insert_iterator operator++(int);
protected:
    Cont *container;
};

```

The template class describes an output iterator object. It inserts elements into a container of type **Cont**, which it accesses via the protected pointer object it stores called **container**. The container must define:

- the member type **const_reference**, which is the type of a constant reference to an element of the sequence controlled by the container
- the member type **reference**, which is the type of a reference to an element of the sequence controlled by the container
- the member type **value_type**, which is the type of an element of the sequence controlled by the container
- the member function **push_back(value_type c)**, which appends a new element with value *c* to the end of the sequence

back_insert_iterator::back_insert_iterator

```
explicit back_insert_iterator(Cont& x);
```

The constructor initializes *container* (page 295) with *&x*.

back_insert_iterator::container_type

```
typedef Cont container_type;
```

The type is a synonym for the template parameter *Cont*.

back_insert_iterator::operator*

```
back_insert_iterator& operator*();
```

The member function returns **this*.

back_insert_iterator::operator++

```
back_insert_iterator& operator++();
back_insert_iterator operator++(int);
```

The member functions both return **this*.

back_insert_iterator::operator=

```
back_insert_iterator&
    operator=(typename Cont::const_reference val);
```

The member function evaluates *container*. *push_back(val)*, then returns **this*.

back_insert_iterator::reference

```
typedef typename Cont::reference reference;
```

The type describes a reference to an element of the sequence controlled by the associated container.

back_insert_iterator::value_type

```
typedef typename Cont::value_type value_type;
```

The type describes the elements of the sequence controlled by the associated container.

back_inserter

```
template<class Cont>
    back_insert_iterator<Cont> back_inserter(Cont& x);
```

The template function returns `back_insert_iterator<Cont>(x)`.

bidirectional_iterator_tag

```
struct bidirectional_iterator_tag
    : public forward_iterator_tag {
};
```

The type is the same as `iterator<It>::iterator_category` when `It` describes an object that can serve as a bidirectional iterator.

distance

```
template<class Init, class Dist>
    typename iterator_traits<Init>::difference_type
    distance(Init first, Init last);
```

The template function sets a count `n` to zero. It then effectively advances `first` and increments `n` until `first == last`. If `Init` is a random-access iterator type, the function evaluates the expression `n += last - first`. Otherwise, it performs each iterator increment by evaluating `++first`.

forward_iterator_tag

```
struct forward_iterator_tag
    : public input_iterator_tag {
};
```

The type is the same as `iterator<It>::iterator_category` when `It` describes an object that can serve as a forward iterator.

front_insert_iterator

```
template<class Cont>
    class front_insert_iterator
        : public iterator<output_iterator_tag,
            void, void, void, void> {
public:
    typedef Cont container_type;
    typedef typename Cont::reference reference;
    typedef typename Cont::value_type value_type;
    explicit front_insert_iterator(Cont& x);
    front_insert_iterator&
        operator=(typename Cont::const_reference val);
    front_insert_iterator& operator*();
    front_insert_iterator& operator++();
    front_insert_iterator operator++(int);
protected:
    Cont *container;
};
```


The template class describes an output iterator object. It inserts elements into a container of type **Cont**, which it accesses via the protected pointer object it stores called **container**. The container must define:

- the member type **const_reference**, which is the type of a constant reference to an element of the sequence controlled by the container
- the member type **reference**, which is the type of a reference to an element of the sequence controlled by the container
- the member type **value_type**, which is the type of an element of the sequence controlled by the container
- the member function **push_front(value_type c)**, which prepends a new element with value *c* to the beginning of the sequence

front_insert_iterator::container_type

```
typedef Cont container_type;
```

The type is a synonym for the template parameter *Cont*.

front_insert_iterator::front_insert_iterator

```
explicit front_insert_iterator(Cont& x);
```

The constructor initializes *container* (page 297) with *&x*.

front_insert_iterator::operator*

```
front_insert_iterator& operator*();
```

The member function returns **this*.

front_insert_iterator::operator++

```
front_insert_iterator& operator++();  
front_insert_iterator operator++(int);
```

The member functions both return **this*.

front_insert_iterator::operator=

```
front_insert_iterator&  
    operator=(typename Cont::const_reference val);
```

The member function evaluates *container.push_front(val)*, then returns **this*.

front_insert_iterator::reference

```
typedef typename Cont::reference reference;
```

The type describes a reference to an element of the sequence controlled by the associated container.

front_insert_iterator::value_type

```
typedef typename Cont::value_type value_type;
```

The type describes the elements of the sequence controlled by the associated container.

front_inserter

```
template<class Cont>  
    front_insert_iterator<Cont> front_inserter(Cont& x);
```

The template function returns *front_insert_iterator<Cont>(x)*.

input_iterator_tag

```
struct input_iterator_tag {  
};
```

The type is the same as `iterator<It>::iterator_category` when `It` describes an object that can serve as an input iterator.

insert_iterator

```
template<class Cont>  
class insert_iterator  
    : public iterator<output_iterator_tag,  
        void, void, void, void> {  
public:  
    typedef Cont container_type;  
    typedef typename Cont::reference reference;  
    typedef typename Cont::value_type value_type;  
    insert_iterator(Cont& x,  
        typename Cont::iterator it);  
    insert_iterator&  
        operator=(typename Cont::const_reference val);  
    insert_iterator& operator*();  
    insert_iterator& operator++();  
    insert_iterator& operator++(int);  
protected:  
    Cont *container;  
    typename Cont::iterator iter;  
};
```

The template class describes an output iterator object. It inserts elements into a container of type **Cont**, which it accesses via the protected pointer object it stores called **container**. It also stores the protected iterator object, of class `Cont::iterator`, called **iter**. The container must define:

- the member type **const_reference**, which is the type of a constant reference to an element of the sequence controlled by the container
- the member type **iterator**, which is the type of an iterator for the container
- the member type **reference**, which is the type of a reference to an element of the sequence controlled by the container
- the member type **value_type**, which is the type of an element of the sequence controlled by the container
- the member function **insert(iterator it, value_type c)**, which inserts a new element with value `c` immediately before the element designated by `it` in the controlled sequence, then returns an iterator that designates the inserted element

insert_iterator::container_type

```
typedef Cont container_type;
```

The type is a synonym for the template parameter `Cont`.

insert_iterator::insert_iterator

```
insert_iterator(Cont& x,  
    typename Cont::iterator it);
```

The constructor initializes `container` (page 298) with `&x`, and `iter` (page 298) with `it`.

insert_iterator::operator*

```
insert_iterator& operator*();
```

The member function returns `*this`.

insert_iterator::operator++

```
insert_iterator& operator++();  
insert_iterator& operator++(int);
```

The member functions both return `*this`.

insert_iterator::operator=

```
insert_iterator&  
operator=(typename Cont::const_reference val);
```

The member function evaluates `iter = container.insert(iter, val)`, then returns `*this`.

insert_iterator::reference

```
typedef typename Cont::reference reference;
```

The type describes a reference to an element of the sequence controlled by the associated container.

insert_iterator::value_type

```
typedef typename Cont::value_type value_type;
```

The type describes the elements of the sequence controlled by the associated container.

inserter

```
template<class Cont, class Iter>  
insert_iterator<Cont> inserter(Cont& x, Iter it);
```

The template function returns `insert_iterator<Cont>(x, it)`.

istream_iterator

```
template<class U, class E = char,  
        class T = char_traits<E>,  
        class Dist = ptrdiff_t>  
class istream_iterator  
    : public iterator<input_iterator_tag,  
                    U, Dist, U *, U> {  
public:  
    typedef E char_type;  
    typedef T traits_type;  
    typedef basic_istream<E, T> istream_type;  
    istream_iterator();  
    istream_iterator(istream_type& is);  
    const U& operator*() const;  
    const U *operator->() const;  
    istream_iterator<U, E, T, Dist>& operator++();  
    istream_iterator<U, E, T, Dist> operator++(int);  
};
```

The template class describes an input iterator object. It extracts objects of class **U** from an **input stream**, which it accesses via an object it stores, of type pointer to `basic_istream<E, T>`. After constructing or incrementing an object of class

`istream_iterator` with a non-null stored pointer, the object attempts to extract and store an object of type `U` from the associated input stream. If the extraction fails, the object effectively replaces the stored pointer with a null pointer (thus making an end-of-sequence indicator).

`istream_iterator::char_type`

```
typedef E char_type;
```

The type is a synonym for the template parameter `E`.

`istream_iterator::istream_iterator`

```
istream_iterator();  
istream_iterator(istream_type& is);
```

The first constructor initializes the input stream pointer with a null pointer. The second constructor initializes the input stream pointer with `&is`, then attempts to extract and store an object of type `U`.

`istream_iterator::istream_type`

```
typedef basic_istream<E, T> istream_type;
```

The type is a synonym for `basic_istream<E, T>`.

`istream_iterator::operator*`

```
const U& operator*() const;
```

The operator returns the stored object of type `U`.

`istream_iterator::operator->`

```
const U *operator->() const;
```

The operator returns `&***this`.

`istream_iterator::operator++`

```
istream_iterator<U, E, T, Dist>& operator++();  
istream_iterator<U, E, T, Dist> operator++(int);
```

The first operator attempts to extract and store an object of type `U` from the associated input stream. The second operator makes a copy of the object, increments the object, then returns the copy.

`istream_iterator::traits_type`

```
typedef T traits_type;
```

The type is a synonym for the template parameter `T`.

`istreambuf_iterator`

```
template<class E, class T = char_traits<E> >  
class istreambuf_iterator  
    : public iterator<input_iterator_tag,  
        E, typename T::off_type, E *, E> {  
public:  
    typedef E char_type;  
    typedef T traits_type;  
    typedef typename T::int_type int_type;  
    typedef basic_streambuf<E, T> streambuf_type;  
    typedef basic_istream<E, T> istream_type;  
    istreambuf_iterator(streambuf_type *sb = 0) throw();
```

```

istreambuf_iterator(istream_type& is) throw();
const E& operator*() const;
const E *operator->();
istreambuf_iterator& operator++();
istreambuf_iterator operator++(int);
bool equal(const istreambuf_iterator& rhs) const;
};

```

The template class describes an input iterator object. It extracts elements of class **E** from an **input stream buffer**, which it accesses via an object it stores, of type pointer to `basic_streambuf<E, T>`. After constructing or incrementing an object of class `istreambuf_iterator` with a non-null stored pointer, the object effectively attempts to extract and store an object of type **E** from the associated input stream. (The extraction may be delayed, however, until the object is actually dereferenced or copied.) If the extraction fails, the object effectively replaces the stored pointer with a null pointer (thus making an end-of-sequence indicator).

istreambuf_iterator::char_type

```
typedef E char_type;
```

The type is a synonym for the template parameter **E**.

istreambuf_iterator::equal

```
bool equal(const istreambuf_iterator& rhs) const;
```

The member function returns true only if the stored stream buffer pointers for the object and **rhs** are both null pointers or are both non-null pointers.

istreambuf_iterator::int_type

```
typedef typename T::int_type int_type;
```

The type is a synonym for `T::int_type`.

istreambuf_iterator::istream_type

```
typedef basic_istream<E, T> istream_type;
```

The type is a synonym for `basic_istream<E, T>`.

istreambuf_iterator::istreambuf_iterator

```
istreambuf_iterator(streambuf_type *sb = 0) throw();
istreambuf_iterator(istream_type& is) throw();
```

The first constructor initializes the input stream-buffer pointer with **sb**. The second constructor initializes the input stream-buffer pointer with `is.rdbuf()`, then (eventually) attempts to extract and store an object of type **E**.

istreambuf_iterator::operator*

```
const E& operator*() const;
```

The operator returns the stored object of type **E**.

istreambuf_iterator::operator++

```
istreambuf_iterator& operator++();
istreambuf_iterator operator++(int);
```

The first operator (eventually) attempts to extract and store an object of type **E** from the associated input stream. The second operator makes a copy of the object, increments the object, then returns the copy.

istreambuf_iterator::operator->

```
const E *operator->() const;
```

The operator returns `&*this`.

istreambuf_iterator::streambuf_type

```
typedef basic_streambuf<E, T> streambuf_type;
```

The type is a synonym for `basic_streambuf<E, T>`.

istreambuf_iterator::traits_type

```
typedef T traits_type;
```

The type is a synonym for the template parameter `T`.

iterator

```
template<class C, class T, class Dist = ptrdiff_t
        class Pt = T *, class Rt = T&>
    struct iterator {
        typedef C iterator_category;
        typedef T value_type;
        typedef Dist difference_type;
        typedef Pt pointer;
        typedef Rt reference;
    };
```

The template class serves as a base type for all iterators. It defines the member types `iterator_category`, (a synonym for the template parameter `C`), `value_type` (a synonym for the template parameter `T`), `difference_type` (a synonym for the template parameter `Dist`), `pointer` (a synonym for the template parameter `Pt`), and `reference` (a synonym for the template parameter `T`).

Note that `value_type` should *not* be a constant type even if `pointer` points at an object of `const` type and `reference` designates an object of `const` type.

iterator_traits

```
template<class It>
    struct iterator_traits {
        typedef typename It::iterator_category iterator_category;
        typedef typename It::value_type value_type;
        typedef typename It::difference_type difference_type;
        typedef typename It::pointer pointer;
        typedef typename It::reference reference;
    };
template<class T>
    struct iterator_traits<T*> {
        typedef random_access_iterator_tag iterator_category;
        typedef T value_type;
        typedef ptrdiff_t difference_type;
        typedef T *pointer;
        typedef T& reference;
    };
template<class T>
    struct iterator_traits<const T*> {
        typedef random_access_iterator_tag iterator_category;
        typedef T value_type;
        typedef ptrdiff_t difference_type;
        typedef const T *pointer;
        typedef const T& reference;
    };
```

The template class determines several critical types associated with the iterator type `It`. It defines the member types `iterator_category` (a synonym for `It::iterator_category`), `value_type` (a synonym for `It::value_type`), `difference_type` (a synonym for `It::difference_type`), `pointer` (a synonym for `It::pointer`), and `reference` (a synonym for `It::reference`).

The partial specializations determine the critical types associated with an object pointer type `T *`. In this implementation (page 3), you can also use several template functions that do not make use of partial specialization:

```
template<class C, class T, class Dist>
    C _Iter_cat(const iterator<C, T, Dist>&);
template<class T>
    random_access_iterator_tag _Iter_cat(const T *);

template<class C, class T, class Dist>
    T * _Val_type(const iterator<C, T, Dist>&);
template<class T>
    T * _Val_type(const T *);

template<class C, class T, class Dist>
    Dist * _Dist_type(const iterator<C, T, Dist>&);
template<class T>
    ptrdiff_t * _Dist_type(const T *);
```

which determine several of the same types a bit more indirectly. You use these functions as arguments on a function call. Their sole purpose is to supply a useful template class parameter to the called function.

operator!=

```
template<class RanIt>
    bool operator!=(
        const reverse_iterator<RanIt>& lhs,
        const reverse_iterator<RanIt>& rhs);
template<class U, class E, class T, class Dist>
    bool operator!=(
        const istream_iterator<U, E, T, Dist>& lhs,
        const istream_iterator<U, E, T, Dist>& rhs);
template<class E, class T>
    bool operator!=(
        const istreambuf_iterator<E, T>& lhs,
        const istreambuf_iterator<E, T>& rhs);
```

The template operator returns `!(lhs == rhs)`.

operator==

```
template<class RanIt>
    bool operator==(
        const reverse_iterator<RanIt>& lhs,
        const reverse_iterator<RanIt>& rhs);
template<class U, class E, class T, class Dist>
    bool operator==(
        const istream_iterator<U, E, T, Dist>& lhs,
        const istream_iterator<U, E, T, Dist>& rhs);
template<class E, class T>
    bool operator==(
        const istreambuf_iterator<E, T>& lhs,
        const istreambuf_iterator<E, T>& rhs);
```

The first template operator returns true only if `lhs.current == rhs.current`. The second template operator returns true only if both `lhs` and `rhs` store the same stream pointer. The third template operator returns `lhs.equal(rhs)`.

operator<

```
template<class RanIt>
bool operator<(
    const reverse_iterator<RanIt>& lhs,
    const reverse_iterator<RanIt>& rhs);
```

The template operator returns `rhs.current < lhs.current` [sic].

operator<=

```
template<class RanIt>
bool operator<=(
    const reverse_iterator<RanIt>& lhs,
    const reverse_iterator<RanIt>& rhs);
```

The template operator returns `!(rhs < lhs)`.

operator>

```
template<class RanIt>
bool operator>(
    const reverse_iterator<RanIt>& lhs,
    const reverse_iterator<RanIt>& rhs);
```

The template operator returns `rhs < lhs`.

operator>=

```
template<class RanIt>
bool operator>=(
    const reverse_iterator<RanIt>& lhs,
    const reverse_iterator<RanIt>& rhs);
```

The template operator returns `!(lhs < rhs)`.

operator+

```
template<class RanIt>
reverse_iterator<RanIt> operator+(Dist n,
    const reverse_iterator<RanIt>& rhs);
```

The template operator returns `rhs + n`.

operator-

```
template<class RanIt>
Dist operator-(
    const reverse_iterator<RanIt>& lhs,
    const reverse_iterator<RanIt>& rhs);
```

The template operator returns `rhs.current - lhs.current` [sic].

ostream_iterator

```
template<class U, class E = char,
        class T = char_traits<E> >
class ostream_iterator
    : public iterator<output_iterator_tag,
        void, void, void, void> {
public:
    typedef U value_type;
    typedef E char_type;
    typedef T traits_type;
```



```

typedef basic_ostream<E, T> ostream_type;
ostream_iterator(ostream_type& os);
ostream_iterator(ostream_type& os, const E *delim);
ostream_iterator<U, E, T>& operator=(const U& val);
ostream_iterator<U, E, T>& operator*();
ostream_iterator<U, E, T>& operator++();
ostream_iterator<U, E, T> operator++(int);
};

```

The template class describes an output iterator object. It inserts objects of class **U** into an **output stream**, which it accesses via an object it stores, of type pointer to `basic_ostream<E, T>`. It also stores a pointer to a **delimiter string**, a null-terminated string of elements of type `E`, which is appended after each insertion. (Note that the string itself is *not* copied by the constructor.

ostream_iterator::char_type

```
typedef E char_type;
```

The type is a synonym for the template parameter `E`.

ostream_iterator::operator*

```
ostream_iterator<U, E, T>& operator*();
```

The operator returns `*this`.

ostream_iterator::operator++

```
ostream_iterator<U, E, T>& operator++();
ostream_iterator<U, E, T> operator++(int);
```

The operators both return `*this`.

ostream_iterator::operator=

```
ostream_iterator<U, E, T>& operator=(const U& val);
```

The operator inserts `val` into the output stream associated with the object, then returns `*this`.

ostream_iterator::ostream_iterator

```
ostream_iterator(ostream_type& os);
ostream_iterator(ostream_type& os, const E *delim);
```

The first constructor initializes the output stream pointer with `&os`. The delimiter string pointer designates an empty string. The second constructor initializes the output stream pointer with `&os` and the delimiter string pointer with `delim`.

ostream_iterator::ostream_type

```
typedef basic_ostream<E, T> ostream_type;
```

The type is a synonym for `basic_ostream<E, T>`.

ostream_iterator::traits_type

```
typedef T traits_type;
```

The type is a synonym for the template parameter `T`.

ostream_iterator::value_type

typedef U **value_type**;

The type is a synonym for the template parameter U.

ostreambuf_iterator

```
template<class E, class T = char_traits<E> >
    class ostreambuf_iterator
        : public iterator<output_iterator_tag,
            void, void, void, void> {
public:
    typedef E char_type;
    typedef T traits_type;
    typedef basic_streambuf<E, T> streambuf_type;
    typedef basic_ostream<E, T> ostream_type;
    ostreambuf_iterator(streambuf_type *sb) throw();
    ostreambuf_iterator(ostream_type& os) throw();
    ostreambuf_iterator& operator=(E x);
    ostreambuf_iterator& operator*();
    ostreambuf_iterator& operator++();
    T1 operator++(int);
    bool failed() const throw();
};
```

The template class describes an output iterator object. It inserts elements of class **E** into an **output stream buffer**, which it accesses via an object it stores, of type pointer to `basic_streambuf<E, T>`.

ostreambuf_iterator::char_type

typedef E **char_type**;

The type is a synonym for the template parameter E.

ostreambuf_iterator::failed

bool **failed**() const throw();

The member function returns true only if no insertion into the output stream buffer has earlier failed.

ostreambuf_iterator::operator*

ostreambuf_iterator& **operator***();

The operator returns `*this`.

ostreambuf_iterator::operator++

ostreambuf_iterator& **operator**++();
T1 **operator**++(int);

The first operator returns `*this`. The second operator returns an object of some type T1 that can be converted to `ostreambuf_iterator<E, T>`.

ostreambuf_iterator::operator=

ostreambuf_iterator& **operator**=(E x);

The operator inserts x into the associated stream buffer, then returns `*this`.

ostreambuf_iterator::ostream_type

```
typedef basic_ostream<E, T> ostream_type;
```

The type is a synonym for `basic_ostream<E, T>`.

ostreambuf_iterator::ostreambuf_iterator

```
ostreambuf_iterator(streambuf_type *sb) throw();
```

```
ostreambuf_iterator(ostream_type& os) throw();
```

The first constructor initializes the output stream-buffer pointer with `sb`. The second constructor initializes the output stream-buffer pointer with `os.rdbuf()`. (The stored pointer must not be a null pointer.)

ostreambuf_iterator::streambuf_type

```
typedef basic_streambuf<E, T> streambuf_type;
```

The type is a synonym for `basic_streambuf<E, T>`.

ostreambuf_iterator::traits_type

```
typedef T traits_type;
```

The type is a synonym for the template parameter `T`.

output_iterator_tag

```
struct output_iterator_tag {  
    };
```

The type is the same as `iterator<It>::iterator_category` when `It` describes an object that can serve as a output iterator.

random_access_iterator_tag

```
struct random_access_iterator_tag  
    : public bidirectional_iterator_tag {  
    };
```

The type is the same as `iterator<It>::iterator_category` when `It` describes an object that can serve as a random-access iterator.

reverse_iterator

```
template<class RanIt>  
    class reverse_iterator : public iterator<  
        typename iterator_traits<RanIt>::iterator_category,  
        typename iterator_traits<RanIt>::value_type,  
        typename iterator_traits<RanIt>::difference_type,  
        typename iterator_traits<RanIt>::pointer,  
        typename iterator_traits<RanIt>::reference> {  
    typedef typename iterator_traits<RanIt>::difference_type  
        Dist;  
    typedef typename iterator_traits<RanIt>::pointer  
        Ptr;  
    typedef typename iterator_traits<RanIt>::reference  
        Ref;  
public:  
    typedef RanIt iterator_type;  
    reverse_iterator();  
    explicit reverse_iterator(RanIt x);  
    template<class U>  
        reverse_iterator(const reverse_iterator<U>& x);
```

```

RanIt base() const;
Ref operator*() const;
Ptr operator->() const;
reverse_iterator& operator++();
reverse_iterator operator++(int);
reverse_iterator& operator--();
reverse_iterator operator--();
reverse_iterator& operator+=(Dist n);
reverse_iterator operator+(Dist n) const;
reverse_iterator& operator-=(Dist n);
reverse_iterator operator-(Dist n) const;
Ref operator[](Dist n) const;
protected:
    RanIt current;
};

```

The template class describes an object that behaves like a random-access iterator, only in reverse. It stores a random-access iterator of type **RanIt** in the protected object **current**. Incrementing the object *x* of type `reverse_iterator` decrements *x.current*, and decrementing *x* increments *x.current*. Moreover, the expression **x* evaluates to **(current - 1)*, of type **Ref**. Typically, **Ref** is type **T&**.

Thus, you can use an object of class `reverse_iterator` to access in reverse order a sequence that is traversed in order by a random-access iterator.

Several STL containers (page 41) specialize `reverse_iterator` for **RanIt** a bidirectional iterator. In these cases, you must not call any of the member functions `operator+=`, `operator+`, `operator-=`, `operator-`, or `operator[]`.

reverse_iterator::base

```
RanIt base() const;
```

The member function returns `current` (page 308).

reverse_iterator::iterator_type

```
typedef RanIt iterator_type;
```

The type is a synonym for the template parameter **RanIt**.

reverse_iterator::operator*

```
Ref operator*() const;
```

The operator returns **(current - 1)*.

reverse_iterator::operator+

```
reverse_iterator operator+(Dist n) const;
```

The operator returns `reverse_iterator(*this) += n`.

reverse_iterator::operator++

```
reverse_iterator& operator++();
reverse_iterator operator++(int);
```

The first (preincrement) operator evaluates `-current`. then returns **this*.

The second (postincrement) operator makes a copy of **this*, evaluates `-current`, then returns the copy.

reverse_iterator::operator+=
reverse_iterator& **operator+=(Dist n);**

The operator evaluates current - n. then returns *this.

reverse_iterator::operator-
reverse_iterator **operator-(Dist n) const;**

The operator returns reverse_iterator(*this) -= n.

reverse_iterator::operator—
reverse_iterator& **operator--();**
reverse_iterator **operator--();**

The first (predecrement) operator evaluates ++current. then returns *this.

The second (postdecrement) operator makes a copy of *this, evaluates ++current, then returns the copy.

reverse_iterator::operator-=
reverse_iterator& **operator-=(Dist n);**

The operator evaluates current + n. then returns *this.

reverse_iterator::operator->
Ptr **operator->() const;**

The operator returns &**this.

reverse_iterator::operator[]
Ref **operator[](Dist n) const;**

The operator returns *(*this + n).

reverse_iterator::pointer
typedef Ptr **pointer;**

The type is a synonym for the template parameter Ref.

reverse_iterator::reference
typedef Ref **reference;**

The type is a synonym for the template parameter Ref.

reverse_iterator::reverse_iterator
reverse_iterator();
explicit **reverse_iterator(RanIt x);**
template<class U>
 reverse_iterator (page 309)(const reverse_iterator<U>& x);

The first constructor initializes current (page 308) with its default constructor. The second constructor initializes current with x.current.

The template constructor initializes current with x.base (page 308) ().

<list>

```
namespace std {
template<class T, class A>
    class list;

    // TEMPLATE FUNCTIONS
template<class T, class A>
    bool operator==(
        const list<T, A>& lhs,
        const list<T, A>& rhs);
template<class T, class A>
    bool operator!=(
        const list<T, A>& lhs,
        const list<T, A>& rhs);
template<class T, class A>
    bool operator<(
        const list<T, A>& lhs,
        const list<T, A>& rhs);
template<class T, class A>
    bool operator>(
        const list<T, A>& lhs,
        const list<T, A>& rhs);
template<class T, class A>
    bool operator<=(
        const list<T, A>& lhs,
        const list<T, A>& rhs);
template<class T, class A>
    bool operator>=(
        const list<T, A>& lhs,
        const list<T, A>& rhs);
template<class T, class A>
    void swap(
        list<T, A>& lhs,
        list<T, A>& rhs);
};
```

Include the STL (page 1) standard header **<list>** to define the container (page 41) template class **list** and several supporting templates.

list

allocator_type (page 312) · **assign** (page 312) · **back** (page 312) · **begin** (page 312) · **clear** (page 312) · **const_iterator** (page 313) · **const_pointer** (page 313) · **const_reference** (page 313) · **const_reverse_iterator** (page 313) · **difference_type** (page 313) · **empty** (page 313) · **end** (page 313) · **erase** (page 313) · **front** (page 314) · **get_allocator** (page 314) · **insert** (page 314) · **iterator** (page 314) · **list** (page 314) · **max_size** (page 315) · **merge** (page 315) · **pointer** (page 315) · **pop_back** (page 315) · **pop_front** (page 316) · **push_back** (page 316) · **push_front** (page 316) · **rbegin** (page 316) · **reference** (page 316) · **remove** (page 316) · **remove_if** (page 316) · **rend** (page 317) · **resize** (page 317) · **reverse** (page 317) · **reverse_iterator** (page 317) · **size** (page 317) · **size_type** (page 317) · **sort** (page 317) · **splice** (page 318) · **swap** (page 318) · **unique** (page 318) · **value_type** (page 319)

```
template<class T, class A = allocator<T> >
    class list {
public:
    typedef A allocator_type;
    typedef typename A::pointer pointer;
    typedef typename A::const_pointer
        const_pointer;
    typedef typename A::reference reference;
    typedef typename A::const_reference const_reference;
    typedef typename A::value_type value_type;
    typedef T0 iterator;
```

```

typedef T1 const_iterator;
typedef T2 size_type;
typedef T3 difference_type;
typedef reverse_iterator<const_iterator>
    const_reverse_iterator;
typedef reverse_iterator<iterator>
    reverse_iterator;
list();
explicit list(const A& a);
explicit list(size_type n);
list(size_type n, const T& v);
list(size_type n, const T& v, const A& a);
list(const list& x);
template<class InIt>
    list(InIt first, InIt last);
template<class InIt>
    list(InIt first, InIt last, const A& a);
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;
void resize(size_type n);
void resize(size_type n, T x);
size_type size() const;
size_type max_size() const;
bool empty() const;
A get_allocator() const;
reference front();
const_reference front() const;
reference back();
const_reference back() const;
void push_front(const T& x);
void pop_front();
void push_back(const T& x);
void pop_back();
template<class InIt>
    void assign(InIt first, InIt last);
void assign(size_type n, const T& x);
iterator insert(iterator it, const T& x);
void insert(iterator it, size_type n, const T& x);
template<class InIt>
    void insert(iterator it, InIt first, InIt last);
iterator erase(iterator it);
iterator erase(iterator first, iterator last);
void clear();
void swap(list& x);
void splice(iterator it, list& x);
void splice(iterator it, list& x, iterator first);
void splice(iterator it, list& x, iterator first,
    iterator last);
void remove(const T& x);
template<class Pred>
    void remove_if(Pred pr);
void unique();
template<class Pred>
    void unique(Pred pr);
void merge(list& x);
template<class Pred>
    void merge(list& x, Pred pr);
void sort();

```

```
template<class Pred>
    void sort(Pred pr);
    void reverse();
};
```

The template class describes an object that controls a varying-length sequence of elements of type T. The sequence is stored as a bidirectional linked list of elements, each containing a member of type T.

The object allocates and frees storage for the sequence it controls through a stored allocator object (page 337) of class A. Such an allocator object must have the same external interface as an object of template class allocator (page 337). Note that the stored allocator object is *not* copied when the container object is assigned.

List reallocation occurs when a member function must insert or erase elements of the controlled sequence. In all such cases, only iterators or references that point at erased portions of the controlled sequence become **invalid**.

All additions to the controlled sequence occur as if by calls to insert (page 314), which is the only member function that calls the constructor T(const T&). If such an expression throws an exception, the container object inserts no new elements and rethrows the exception. Thus, an object of template class list is left in a known state when such exceptions occur.

list::allocator_type

```
typedef A allocator_type;
```

The type is a synonym for the template parameter A.

list::assign

```
template<class InIt>
    void assign(InIt first, InIt last);
    void assign(size_type n, const T& x);
```

If InIt is an integer type, the first member function behaves the same as assign((size_type)first, (T)last). Otherwise, the first member function replaces the sequence controlled by *this with the sequence [first, last), which must *not* overlap the initial controlled sequence. The second member function replaces the sequence controlled by *this with a repetition of n elements of value x.

list::back

```
reference back();
const_reference back() const;
```

The member function returns a reference to the last element of the controlled sequence, which must be non-empty.

list::begin

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

The member function returns a bidirectional iterator that points at the first element of the sequence (or just beyond the end of an empty sequence).

list::clear

```
void clear();
```

The member function calls erase(begin(), end()).

list::const_iterator

```
typedef T1 const_iterator;
```

The type describes an object that can serve as a constant bidirectional iterator for the controlled sequence. It is described here as a synonym for the implementation-defined type T1.

list::const_pointer

```
typedef typename A::const_pointer  
const_pointer;
```

The type describes an object that can serve as a constant pointer to an element of the controlled sequence.

list::const_reference

```
typedef typename A::const_reference const_reference;
```

The type describes an object that can serve as a constant reference to an element of the controlled sequence.

list::const_reverse_iterator

```
typedef reverse_iterator<const_iterator>  
const_reverse_iterator;
```

The type describes an object that can serve as a constant reverse bidirectional iterator for the controlled sequence.

list::difference_type

```
typedef T3 difference_type;
```

The signed integer type describes an object that can represent the difference between the addresses of any two elements in the controlled sequence. It is described here as a synonym for the implementation-defined type T3.

list::empty

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

The member function returns true for an empty controlled sequence.

list::end

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

The member function returns a bidirectional iterator that points just beyond the end of the sequence.

list::erase

```
iterator erase(iterator it);  
iterator erase(iterator first, iterator last);
```

The first member function removes the element of the controlled sequence pointed to by it. The second member function removes the elements of the controlled sequence in the range [first, last). Both return an iterator that designates the first element remaining beyond any elements removed, or end() if no such element exists.

Erasing *N* elements causes *N* destructor calls. No reallocation (page 312) occurs, so iterators and references become invalid (page 312) only for the erased elements.

The member functions never throw an exception.

list::front

```
reference front();  
const_reference front() const;
```

The member function returns a reference to the first element of the controlled sequence, which must be non-empty.

list::get_allocator

```
A get_allocator() const;
```

The member function returns the stored allocator object (page 337).

list::insert

```
iterator insert(iterator it, const T& x);  
void insert(iterator it, size_type n, const T& x);  
template<class InIt>  
    void insert(iterator it, InIt first, InIt last);
```

Each of the member functions inserts, before the element pointed to by *it* in the controlled sequence, a sequence specified by the remaining operands. The first member function inserts a single element with value *x* and returns an iterator that points to the newly inserted element. The second member function inserts a repetition of *n* elements of value *x*.

If *InIt* is an integer type, the last member function behaves the same as `insert(it, (size_type)first, (T)last)`. Otherwise, the last member function inserts the sequence [*first*, *last*), which must *not* overlap the initial controlled sequence.

Inserting *N* elements causes *N* constructor calls. No reallocation (page 312) occurs, so no iterators or references become invalid (page 312).

If an exception is thrown during the insertion of one or more elements, the container is left unaltered and the exception is rethrown.

list::iterator

```
typedef T0 iterator;
```

The type describes an object that can serve as a bidirectional iterator for the controlled sequence. It is described here as a synonym for the implementation-defined type *T0*.

list::list

```
list();  
explicit list(const A& a);  
explicit list(size_type n);  
list(size_type n, const T& v);  
list(size_type n, const T& v,  
    const A& a);  
list(const list& x);  
template<class InIt>  
    list(InIt first, InIt last);  
template<class InIt>  
    list(InIt first, InIt last, const A& a);
```

All constructors store an allocator object (page 337) and initialize the controlled sequence. The allocator object is the argument `al`, if present. For the copy constructor, it is `x.get_allocator()`. Otherwise, it is `A()`.

The first two constructors specify an empty initial controlled sequence. The third constructor specifies a repetition of `n` elements of value `T()`. The fourth and fifth constructors specify a repetition of `n` elements of value `x`. The sixth constructor specifies a copy of the sequence controlled by `x`. If `InIt` is an integer type, the last two constructors specify a repetition of `(size_type)first` elements of value `(T)last`. Otherwise, the last two constructors specify the sequence `[first, last)`. None of the constructors perform any interim reallocations (page 312).

list::max_size

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

The member function returns the length of the longest sequence that the object can control.

list::merge

```
void merge(list& x);
template<class Pred>
    void merge(list& x, Pred pr);
```

Both member functions remove all elements from the sequence controlled by `x` and insert them in the controlled sequence. Both sequences must be ordered by (page 39) the same predicate, described below. The resulting sequence is also ordered by that predicate.

For the iterators `Pi` and `Pj` designating elements at positions `i` and `j`, the first member function imposes the order `!(*Pj < *Pi)` whenever `i < j`. (The elements are sorted in *ascending* order.) The second member function imposes the order `!pr(*Pj, *Pi)` whenever `i < j`.

No pairs of elements in the original controlled sequence are reversed in the resulting controlled sequence. If a pair of elements in the resulting controlled sequence compares equal (`!(*Pi < *Pj) && !(*Pj < *Pi)`), an element from the original controlled sequence appears before an element from the sequence controlled by `x`.

An exception occurs only if `pr` throws an exception. In that case, the controlled sequence is left in unspecified order and the exception is rethrown.

list::pointer

```
typedef typename A::pointer pointer;
```

The type describes an object that can serve as a pointer to an element of the controlled sequence.

list::pop_back

```
void pop_back();
```

The member function removes the last element of the controlled sequence, which must be non-empty.

The member function never throws an exception.

list::pop_front

```
void pop_front();
```

The member function removes the first element of the controlled sequence, which must be non-empty.

The member function never throws an exception.

list::push_back

```
void push_back(const T& x);
```

The member function inserts an element with value *x* at the end of the controlled sequence.

If an exception is thrown, the container is left unaltered and the exception is rethrown.

list::push_front

```
void push_front(const T& x);
```

The member function inserts an element with value *x* at the beginning of the controlled sequence.

If an exception is thrown, the container is left unaltered and the exception is rethrown.

list::rbegin

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

The member function returns a reverse bidirectional iterator that points just beyond the end of the controlled sequence. Hence, it designates the beginning of the reverse sequence.

list::reference

```
typedef typename A::reference reference;
```

The type describes an object that can serve as a reference to an element of the controlled sequence.

list::remove

```
void remove(const T& x);
```

The member function removes from the controlled sequence all elements, designated by the iterator *P*, for which **P* == *x*.

The member function never throws an exception.

list::remove_if

```
template<class Pred>  
void remove_if(Pred pr);
```

The member function removes from the controlled sequence all elements, designated by the iterator *P*, for which *pr*(**P*) is true.

An exception occurs only if *pr* throws an exception. In that case, the controlled sequence is left in an unspecified state and the exception is rethrown.

list::rend

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

The member function returns a reverse bidirectional iterator that points at the first element of the sequence (or just beyond the end of an empty sequence). Hence, it designates the end of the reverse sequence.

list::resize

```
void resize(size_type n);  
void resize(size_type n, T x);
```

The member functions both ensure that `size()` henceforth returns `n`. If it must make the controlled sequence longer, the first member function appends elements with value `T()`, while the second member function appends elements with value `x`. To make the controlled sequence shorter, both member functions call `erase(begin() + n, end())`.

list::reverse

```
void reverse();
```

The member function reverses the order in which elements appear in the controlled sequence.

list::reverse_iterator

```
typedef reverse_iterator<iterator>  
    reverse_iterator;
```

The type describes an object that can serve as a reverse bidirectional iterator for the controlled sequence.

list::size

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

The member function returns the length of the controlled sequence.

list::size_type

```
typedef T2 size_type;
```

The unsigned integer type describes an object that can represent the length of any controlled sequence. It is described here as a synonym for the implementation-defined type `T2`.

list::sort

```
void sort();  
template<class Pred>  
    void sort(Pred pr);
```

Both member functions order the elements in the controlled sequence by a predicate, described below.

For the iterators `Pi` and `Pj` designating elements at positions `i` and `j`, the first member function imposes the order `!(*Pj < *Pi)` whenever `i < j`. (The elements are sorted in *ascending* order.) The member template function imposes the order `!pr(*Pj, *Pi)` whenever `i < j`. No pairs of elements in the original controlled sequence are reversed in the resulting controlled sequence.

An exception occurs only if `pr` throws an exception. In that case, the controlled sequence is left in unspecified order and the exception is rethrown.

list::splice

```
void splice(iterator it, list& x);  
void splice(iterator it, list& x, iterator first);  
void splice(iterator it, list& x, iterator first,  
            iterator last);
```

The first member function inserts the sequence controlled by `x` before the element in the controlled sequence pointed to by `it`. It also removes all elements from `x`. (`&x` must not equal `this`.)

The second member function removes the element pointed to by `first` in the sequence controlled by `x` and inserts it before the element in the controlled sequence pointed to by `it`. (If `it == first` || `it == ++first`, no change occurs.)

The third member function inserts the subrange designated by `[first, last)` from the sequence controlled by `x` before the element in the controlled sequence pointed to by `it`. It also removes the original subrange from the sequence controlled by `x`. (If `&x == this`, the range `[first, last)` must not include the element pointed to by `it`.)

If the third member function inserts `N` elements, and `&x != this`, an object of class `iterator` (page 314) is incremented `N` times. For all `splice` member functions, If `get_allocator() == str.get_allocator()`, no exception occurs. Otherwise, a copy and a destructor call also occur for each inserted element.

In all cases, only iterators or references that point at spliced elements become **invalid**.

list::swap

```
void swap(list& x);
```

The member function swaps the controlled sequences between `*this` and `x`. If `get_allocator() == x.get_allocator()`, it does so in constant time, it throws no exceptions, and it invalidates no references, pointers, or iterators that designate elements in the two controlled sequences. Otherwise, it performs a number of element assignments and constructor calls proportional to the number of elements in the two controlled sequences.

list::unique

```
void unique();  
template<class Pred>  
    void unique(Pred pr);
```

The first member function removes from the controlled sequence every element that compares equal to its preceding element. For the iterators `Pi` and `Pj` designating elements at positions `i` and `j`, the second member function removes every element for which `i + 1 == j` && `pr(*Pi, *Pj)`.

For a controlled sequence of length `N` (> 0), the predicate `pr(*Pi, *Pj)` is evaluated `N - 1` times.

An exception occurs only if `pr` throws an exception. In that case, the controlled sequence is left in an unspecified state and the exception is rethrown.

list::value_type

```
typedef typename A::value_type value_type;
```

The type is a synonym for the template parameter T.

operator!=

```
template<class T, class A>
    bool operator!=(
        const list <T, A>& lhs,
        const list <T, A>& rhs);
```

The template function returns `!(lhs == rhs)`.

operator==

```
template<class T, class A>
    bool operator==(
        const list <T, A>& lhs,
        const list <T, A>& rhs);
```

The template function overloads `operator==` to compare two objects of template class `list` (page 310). The function returns `lhs.size() == rhs.size() && equal(lhs.begin(), lhs.end(), rhs.begin())`.

operator<

```
template<class T, class A>
    bool operator<(
        const list <T, A>& lhs,
        const list <T, A>& rhs);
```

The template function overloads `operator<` to compare two objects of template class `list`. The function returns `lexicographical_compare(lhs.begin(), lhs.end(), rhs.begin(), rhs.end())`.

operator<=

```
template<class T, class A>
    bool operator<=(
        const list <T, A>& lhs,
        const list <T, A>& rhs);
```

The template function returns `!(rhs < lhs)`.

operator>

```
template<class T, class A>
    bool operator>(
        const list <T, A>& lhs,
        const list <T, A>& rhs);
```

The template function returns `rhs < lhs`.

operator>=

```
template<class T, class A>
    bool operator>=(
        const list <T, A>& lhs,
        const list <T, A>& rhs);
```

The template function returns `!(lhs < rhs)`.

swap

```
template<class T, class A>
void swap(
    list<T, A>& lhs,
    list<T, A>& rhs);
```

The template function executes `lhs.swap (page 318)(rhs)`.

Portions derived from work copyright © 1994 by Hewlett-Packard Company. All rights reserved.

<map>

```
namespace std {
template<class Key, class T, class Pred, class A>
    class map;
template<class Key, class T, class Pred, class A>
    class multimap;

    // TEMPLATE FUNCTIONS
template<class Key, class T, class Pred, class A>
    bool operator==(
        const map<Key, T, Pred, A>& lhs,
        const map<Key, T, Pred, A>& rhs);
template<class Key, class T, class Pred, class A>
    bool operator==(
        const multimap<Key, T, Pred, A>& lhs,
        const multimap<Key, T, Pred, A>& rhs);
template<class Key, class T, class Pred, class A>
    bool operator!=(
        const map<Key, T, Pred, A>& lhs,
        const map<Key, T, Pred, A>& rhs);
template<class Key, class T, class Pred, class A>
    bool operator!=(
        const multimap<Key, T, Pred, A>& lhs,
        const multimap<Key, T, Pred, A>& rhs);
template<class Key, class T, class Pred, class A>
    bool operator<(
        const map<Key, T, Pred, A>& lhs,
        const map<Key, T, Pred, A>& rhs);
template<class Key, class T, class Pred, class A>
    bool operator<(
        const multimap<Key, T, Pred, A>& lhs,
        const multimap<Key, T, Pred, A>& rhs);
template<class Key, class T, class Pred, class A>
    bool operator>(
        const map<Key, T, Pred, A>& lhs,
        const map<Key, T, Pred, A>& rhs);
template<class Key, class T, class Pred, class A>
    bool operator>(
        const multimap<Key, T, Pred, A>& lhs,
        const multimap<Key, T, Pred, A>& rhs);
template<class Key, class T, class Pred, class A>
    bool operator<=(
        const map<Key, T, Pred, A>& lhs,
        const map<Key, T, Pred, A>& rhs);
template<class Key, class T, class Pred, class A>
    bool operator<=(
        const multimap<Key, T, Pred, A>& lhs,
        const multimap<Key, T, Pred, A>& rhs);
template<class Key, class T, class Pred, class A>
    bool operator>=(
        const map<Key, T, Pred, A>& lhs,
        const map<Key, T, Pred, A>& rhs);
template<class Key, class T, class Pred, class A>
```



```

    bool operator>=(
        const multimap<Key, T, Pred, A>& lhs,
        const multimap<Key, T, Pred, A>& rhs);
template<class Key, class T, class Pred, class A>
void swap(
    map<Key, T, Pred, A>& lhs,
    map<Key, T, Pred, A>& rhs);
template<class Key, class T, class Pred, class A>
void swap(
    multimap<Key, T, Pred, A>& lhs,
    multimap<Key, T, Pred, A>& rhs);
};

```

Include the STL (page 1) standard header `<map>` to define the container (page 41) template classes `map` and `multimap`, and their supporting templates.

map

allocator_type (page 323) · begin (page 323) · clear (page 323) · const_iterator (page 323) · const_pointer (page 323) · const_reference (page 323) · const_reverse_iterator (page 323) · count (page 323) · difference_type (page 323) · empty (page 324) · end (page 324) · equal_range (page 324) · erase (page 324) · find (page 324) · get_allocator (page 324) · insert (page 324) · iterator (page 325) · key_comp (page 325) · key_compare (page 325) · key_type (page 325) · lower_bound (page 325) · map (page 325) · mapped_type (page 326) · max_size (page 326) · operator[] (page 326) · pointer (page 326) · rbegin (page 326) · reference (page 326) · rend (page 327) · reverse_iterator (page 327) · size (page 327) · size_type (page 327) · swap (page 327) · upper_bound (page 327) · value_comp (page 327) · value_compare (page 328) · value_type (page 328)

```

template<class Key, class T, class Pred = less<Key>,
        class A = allocator<pair<const Key, T> > >
class map {
public:
    typedef Key key_type;
    typedef T mapped_type;
    typedef Pred key_compare;
    typedef A allocator_type;
    typedef pair<const Key, T> value_type;
    class value_compare;
    typedef A::pointer pointer;
    typedef A::const_pointer const_pointer;
    typedef A::reference reference;
    typedef A::const_reference const_reference;
    typedef T0 iterator;
    typedef T1 const_iterator;
    typedef T2 size_type;
    typedef T3 difference_type;
    typedef reverse_iterator<const_iterator>
        const_reverse_iterator;
    typedef reverse_iterator<iterator> reverse_iterator;
    map();
    explicit map(const Pred& comp);
    map(const Pred& comp, const A& al);
    map(const map& x);
    template<class InIt>
        map(InIt first, InIt last);
    template<class InIt>
        map(InIt first, InIt last,
            const Pred& comp);
    template<class InIt>
        map(InIt first, InIt last,
            const Pred& comp, const A& al);
    iterator begin();
    const_iterator begin() const;

```

```

iterator end();
const_iterator end() const;
reverse_iterator rbegin();
const_reverse_iterator rbegin() const;
reverse_iterator rend();
const_reverse_iterator rend() const;
size_type size() const;
size_type max_size() const;
bool empty() const;
A get_allocator() const;
mapped_type operator[](const Key& key);
pair<iterator, bool> insert(const value_type& x);
iterator insert(iterator it, const value_type& x);
template<class InIt>
    void insert(InIt first, InIt last);
iterator erase(iterator it);
iterator erase(iterator first, iterator last);
size_type erase(const Key& key);
void clear();
void swap(map& x);
key_compare key_comp() const;
value_compare value_comp() const;
iterator find(const Key& key);
const_iterator find(const Key& key) const;
size_type count(const Key& key) const;
iterator lower_bound(const Key& key);
const_iterator lower_bound(const Key& key) const;
iterator upper_bound(const Key& key);
const_iterator upper_bound(const Key& key) const;
pair<iterator, iterator> equal_range(const Key& key);
pair<const_iterator, const_iterator>
    equal_range(const Key& key) const;
};

```

The template class describes an object that controls a varying-length sequence of elements of type `pair<const Key, T>`. The sequence is ordered by (page 39) the predicate `Pred`. The first element of each pair is the **sort key** and the second is its associated **value**. The sequence is represented in a way that permits lookup, insertion, and removal of an arbitrary element with a number of operations proportional to the logarithm of the number of elements in the sequence (logarithmic time). Moreover, inserting an element invalidates no iterators, and removing an element invalidates only those iterators which point at the removed element.

The object orders the sequence it controls by calling a stored **function object** of type `Pred`. You access this stored object by calling the member function `key_comp()`. Such a function object must impose a total ordering (page 401) on sort keys of type `Key`. For any element `x` that precedes `y` in the sequence, `key_comp()(y.first, x.first)` is false. (For the default function object `less<Key>`, sort keys never decrease in value.) Unlike template class `multimap` (page 328), an object of template class `map` ensures that `key_comp()(x.first, y.first)` is true. (Each key is unique.)

The object allocates and frees storage for the sequence it controls through a stored allocator object (page 337) of class `A`. Such an allocator object must have the same external interface as an object of template class `allocator` (page 337). Note that the stored allocator object is *not* copied when the container object is assigned.

map::allocator_type

```
typedef A allocator_type;
```

The type is a synonym for the template parameter A.

map::begin

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

The member function returns a bidirectional iterator that points at the first element of the sequence (or just beyond the end of an empty sequence).

map::clear

```
void clear();
```

The member function calls `erase(begin(), end())`.

map::const_iterator

```
typedef T1 const_iterator;
```

The type describes an object that can serve as a constant bidirectional iterator for the controlled sequence. It is described here as a synonym for the implementation-defined type T1.

map::const_pointer

```
typedef A::const_pointer const_pointer;
```

The type describes an object that can serve as a constant pointer to an element of the controlled sequence.

map::const_reference

```
typedef A::const_reference const_reference;
```

The type describes an object that can serve as a constant reference to an element of the controlled sequence.

map::const_reverse_iterator

```
typedef reverse_iterator<const_iterator>  
    const_reverse_iterator;
```

The type describes an object that can serve as a constant reverse bidirectional iterator for the controlled sequence.

map::count

```
size_type count(const Key& key) const;
```

The member function returns the number of elements x in the range `[lower_bound(key), upper_bound(key))`.

map::difference_type

```
typedef T3 difference_type;
```

The signed integer type describes an object that can represent the difference between the addresses of any two elements in the controlled sequence. It is described here as a synonym for the implementation-defined type T3.

map::empty

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

The member function returns true for an empty controlled sequence.

map::end

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

The member function returns a bidirectional iterator that points just beyond the end of the sequence.

map::equal_range

```
pair<iterator, iterator> equal_range(const Key& key);  
pair<const_iterator, const_iterator>  
    equal_range(const Key& key) const;
```

The member function returns a pair of iterators `x` such that `x.first == lower_bound(key)` and `x.second == upper_bound(key)`.

map::erase

```
iterator erase(iterator it);  
iterator erase(iterator first, iterator last);  
size_type erase(const Key& key);
```

The first member function removes the element of the controlled sequence pointed to by `it`. The second member function removes the elements in the interval `[first, last)`. Both return an iterator that designates the first element remaining beyond any elements removed, or `end()` if no such element exists.

The third member function removes the elements with sort keys in the range `[lower_bound(key), upper_bound(key))`. It returns the number of elements it removes.

The member functions never throw an exception.

map::find

```
iterator find(const Key& key);  
const_iterator find(const Key& key) const;
```

The member function returns an iterator that designates the earliest element in the controlled sequence whose sort key has equivalent ordering (page 39) to `key`. If no such element exists, the function returns `end()`.

map::get_allocator

```
A get_allocator() const;
```

The member function returns the stored allocator object (page 337).

map::insert

```
pair<iterator, bool> insert(const value_type& x);  
iterator insert(iterator it, const value_type& x);  
template<class InIt>  
    void insert(InIt first, InIt last);
```

The first member function determines whether an element `y` exists in the sequence whose key has equivalent ordering (page 39) to that of `x`. If not, it creates such an

element *y* and initializes it with *x*. The function then determines the iterator *it* that designates *y*. If an insertion occurred, the function returns `pair(it, true)`. Otherwise, it returns `pair(it, false)`.

The second member function returns `insert(x)`, using *it* as a starting place within the controlled sequence to search for the insertion point. (Insertion can occur in amortized constant time, instead of logarithmic time, if the insertion point immediately follows *it*.) The third member function inserts the sequence of element values, for each *it* in the range `[first, last)`, by calling `insert(*it)`.

If an exception is thrown during the insertion of a single element, the container is left unaltered and the exception is rethrown. If an exception is thrown during the insertion of multiple elements, the container is left in a stable but unspecified state and the exception is rethrown.

map::iterator

```
typedef T0 iterator;
```

The type describes an object that can serve as a bidirectional iterator for the controlled sequence. It is described here as a synonym for the implementation-defined type *T0*.

map::key_comp

```
key_compare key_comp() const;
```

The member function returns the stored function object that determines the order of elements in the controlled sequence. The stored object defines the member function:

```
bool operator()(const Key& x, const Key& y);
```

which returns true if *x* strictly precedes *y* in the sort order.

map::key_compare

```
typedef Pred key_compare;
```

The type describes a function object that can compare two sort keys to determine the relative order of two elements in the controlled sequence.

map::key_type

```
typedef Key key_type;
```

The type describes the sort key object stored in each element of the controlled sequence.

map::lower_bound

```
iterator lower_bound(const Key& key);  
const_iterator lower_bound(const Key& key) const;
```

The member function returns an iterator that designates the earliest element *x* in the controlled sequence for which `key_comp()(x, key)` is false.

If no such element exists, the function returns `end()`.

map::map

```
map();  
explicit map(const Pred& comp);  
map(const Pred& comp, const A& a1);
```

```

map(const map& x);
template<class Init>
    map(Init first, Init last);
template<class Init>
    map(Init first, Init last,
        const Pred& comp);
template<class Init>
    map(Init first, Init last,
        const Pred& comp, const A& a1);

```

All constructors store an allocator object (page 337) and initialize the controlled sequence. The allocator object is the argument `a1`, if present. For the copy constructor, it is `x.get_allocator()`. Otherwise, it is `A()`.

All constructors also store a function object that can later be returned by calling `key_comp()`. The function object is the argument `comp`, if present. For the copy constructor, it is `x.key_comp()`. Otherwise, it is `Pred()`.

The first three constructors specify an empty initial controlled sequence. The fourth constructor specifies a copy of the sequence controlled by `x`. The last three constructors specify the sequence of element values `[first, last)`.

map::mapped_type

```
typedef T mapped_type;
```

The type is a synonym for the template parameter `T`.

map::max_size

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

The member function returns the length of the longest sequence that the object can control.

map::operator[]

```
T& operator[] (const Key& key);
```

The member function determines the iterator `it` as the return value of `insert(value_type(key, T()))`. (It inserts an element with the specified key if no such element exists.) It then returns a reference to `(*it).second`.

map::pointer

```
typedef A::pointer pointer;
```

The type describes an object that can serve as a pointer to an element of the controlled sequence.

map::rbegin

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

The member function returns a reverse bidirectional iterator that points just beyond the end of the controlled sequence. Hence, it designates the beginning of the reverse sequence.

map::reference

```
typedef A::reference reference;
```

The type describes an object that can serve as a reference to an element of the controlled sequence.

map::rend

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

The member function returns a reverse bidirectional iterator that points at the first element of the sequence (or just beyond the end of an empty sequence). Hence, it designates the end of the reverse sequence.

map::reverse_iterator

```
typedef reverse_iterator<iterator> reverse_iterator;
```

The type describes an object that can serve as a reverse bidirectional iterator for the controlled sequence.

map::size

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

The member function returns the length of the controlled sequence.

map::size_type

```
typedef T2 size_type;
```

The unsigned integer type describes an object that can represent the length of any controlled sequence. It is described here as a synonym for the implementation-defined type T2.

map::swap

```
void swap(map& x);
```

The member function swaps the controlled sequences between *this and x. If `get_allocator() == x.get_allocator()`, it does so in constant time, it throws an exception only as a result of copying the stored function object of type `Pred`, and it invalidates no references, pointers, or iterators that designate elements in the two controlled sequences. Otherwise, it performs a number of element assignments and constructor calls proportional to the number of elements in the two controlled sequences.

map::upper_bound

```
iterator upper_bound(const Key& key);  
const_iterator upper_bound(const Key& key) const;
```

The member function returns an iterator that designates the earliest element x in the controlled sequence for which `key_comp()(key, x.first)` is true.

If no such element exists, the function returns `end()`.

map::value_comp

```
value_compare value_comp() const;
```

The member function returns a function object that determines the order of elements in the controlled sequence.

map::value_compare

```
class value_compare
: public binary_function<value_type, value_type,
    bool> {
public:
    bool operator()(const value_type& x,
        const value_type& y) const
    {return (comp(x.first, y.first)); }
protected:
    value_compare(key_compare pr)
        : comp(pr) {}
    key_compare comp;
};
```

The type describes a function object that can compare the sort keys in two elements to determine their relative order in the controlled sequence. The function object stores an object **comp** of type `key_compare`. The member function **operator()** uses this object to compare the sort-key components of two element.

map::value_type

```
typedef pair (page 402)<const Key, T> value_type;
```

The type describes an element of the controlled sequence.

multimap

`allocator_type` (page 330) · `begin` (page 330) · `clear` (page 330) · `const_iterator` (page 330) · `const_pointer` (page 330) · `const_reference` (page 330) · `const_reverse_iterator` (page 330) · `count` (page 330) · `difference_type` (page 330) · `empty` (page 331) · `end` (page 331) · `equal_range` (page 331) · `erase` (page 331) · `find` (page 331) · `get_allocator` (page 331) · `insert` (page 331) · `iterator` (page 332) · `key_comp` (page 332) · `key_compare` (page 332) · `key_type` (page 332) · `lower_bound` (page 332) · `mapped_type` (page 332) · `max_size` (page 333) · `multimap` (page 333) · `rbegin` (page 333) · `reference` (page 333) · `rend` (page 333) · `reverse_iterator` (page 334) · `size` (page 334) · `size_type` (page 334) · `swap` (page 334) · `upper_bound` (page 334) · `value_comp` (page 334) · `value_compare` (page 334) · `value_type` (page 335)

```
template<class Key, class T, class Pred = less<Key>,
    class A = allocator<pair<const Key, T> > >
class multimap {
public:
    typedef Key key_type;
    typedef T mapped_type;
    typedef Pred key_compare;
    typedef A allocator_type;
    typedef pair<const Key, T> value_type;
    class value_compare;
    typedef A::reference reference;
    typedef A::const_reference const_reference;
    typedef T0 iterator;
    typedef T1 const_iterator;
    typedef T2 size_type;
    typedef T3 difference_type;
    typedef reverse_iterator<const_iterator>
        const_reverse_iterator;
    typedef reverse_iterator<iterator> reverse_iterator;
    multimap();
    explicit multimap(const Pred& comp);
    multimap(const Pred& comp, const A& al);
    multimap(const multimap& x);
    template<class InIt>
        multimap(InIt first, InIt last);
```



```

template<class InIt>
    multimap(InIt first, InIt last,
             const Pred& comp);
template<class InIt>
    multimap(InIt first, InIt last,
             const Pred& comp, const A& a1);
iterator begin();
const_iterator begin() const;
iterator end();
const_iterator end() const;
reverse_iterator rbegin();
const_reverse_iterator rbegin() const;
reverse_iterator rend();
const_reverse_iterator rend() const;
size_type size() const;
size_type max_size() const;
bool empty() const;
A get_allocator() const;
iterator insert(const value_type& x);
iterator insert(iterator it, const value_type& x);
template<class InIt>
    void insert(InIt first, InIt last);
iterator erase(iterator it);
iterator erase(iterator first, iterator last);
size_type erase(const Key& key);
void clear();
void swap(multimap& x);
key_compare key_comp() const;
value_compare value_comp() const;
iterator find(const Key& key);
const_iterator find(const Key& key) const;
size_type count(const Key& key) const;
iterator lower_bound(const Key& key);
const_iterator lower_bound(const Key& key) const;
iterator upper_bound(const Key& key);
const_iterator upper_bound(const Key& key) const;
pair<iterator, iterator> equal_range(const Key& key);
pair<const_iterator, const_iterator>
    equal_range(const Key& key) const;
};

```

The template class describes an object that controls a varying-length sequence of elements of type `pair<const Key, T>`. The sequence is ordered by (page 39) the predicate `Pred`. The first element of each pair is the **sort key** and the second is its associated **value**. The sequence is represented in a way that permits lookup, insertion, and removal of an arbitrary element with a number of operations proportional to the logarithm of the number of elements in the sequence (logarithmic time). Moreover, inserting an element invalidates no iterators, and removing an element invalidates only those iterators which point at the removed element.

The object orders the sequence it controls by calling a stored **function object** of type `Pred`. You access this stored object by calling the member function `key_comp()`. Such a function object must impose a total ordering (page 401) on sort keys of type `Key`. For any element `x` that precedes `y` in the sequence, `key_comp()(y.first, x.first)` is false. (For the default function object `less<Key>`, sort keys never decrease in value.) Unlike template class `map` (page 321), an object of template class `multimap` does not ensure that `key_comp()(x.first, y.first)` is true. (Keys need not be unique.)

The object allocates and frees storage for the sequence it controls through a stored allocator object (page 337) of class `A`. Such an allocator object must have the same

external interface as an object of template class allocator (page 337). Note that the stored allocator object is *not* copied when the container object is assigned.

multimap::allocator_type

```
typedef A allocator_type;
```

The type is a synonym for the template parameter A.

multimap::begin

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

The member function returns a bidirectional iterator that points at the first element of the sequence (or just beyond the end of an empty sequence).

multimap::clear

```
void clear();
```

The member function calls `erase(begin(), end())`.

multimap::const_iterator

```
typedef T1 const_iterator;
```

The type describes an object that can serve as a constant bidirectional iterator for the controlled sequence. It is described here as a synonym for the implementation-defined type T1.

multimap::const_pointer

```
typedef A::const_pointer const_pointer;
```

The type describes an object that can serve as a constant pointer to an element of the controlled sequence.

multimap::const_reference

```
typedef A::const_reference const_reference;
```

The type describes an object that can serve as a constant reference to an element of the controlled sequence.

multimap::const_reverse_iterator

```
typedef reverse_iterator<const_iterator>  
    const_reverse_iterator;
```

The type describes an object that can serve as a constant reverse bidirectional iterator for the controlled sequence.

multimap::count

```
size_type count(const Key& key) const;
```

The member function returns the number of elements x in the range `[lower_bound(key), upper_bound(key))`.

multimap::difference_type

```
typedef T3 difference_type;
```

The signed integer type describes an object that can represent the difference between the addresses of any two elements in the controlled sequence. It is described here as a synonym for the implementation-defined type T3.

multimap::empty

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

The member function returns true for an empty controlled sequence.

multimap::end

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

The member function returns a bidirectional iterator that points just beyond the end of the sequence.

multimap::equal_range

```
pair<iterator, iterator> equal_range(const Key& key);  
pair<const_iterator, const_iterator>  
    equal_range(const Key& key) const;
```

The member function returns a pair of iterators *x* such that *x.first* == *lower_bound*(key) and *x.second* == *upper_bound*(key).

multimap::erase

```
iterator erase(iterator it);  
iterator erase(iterator first, iterator last);  
size_type erase(const Key& key);
```

The first member function removes the element of the controlled sequence pointed to by *it*. The second member function removes the elements in the range [*first*, *last*). Both return an iterator that designates the first element remaining beyond any elements removed, or *end*() if no such element exists.

The third member removes the elements with sort keys in the range [*lower_bound*(key), *upper_bound*(key)). It returns the number of elements it removes.

The member functions never throw an exception.

multimap::find

```
iterator find(const Key& key);  
const_iterator find(const Key& key) const;
```

The member function returns an iterator that designates the earliest element in the controlled sequence whose sort key has equivalent ordering (page 39) to key. If no such element exists, the function returns *end*().

multimap::get_allocator

```
A get_allocator() const;
```

The member function returns the stored allocator object (page 337).

multimap::insert

```
iterator insert(const value_type& x);  
iterator insert(iterator it, const value_type& x);  
template<class InIt>  
    void insert(InIt first, InIt last);
```

The first member function inserts the element *x* in the controlled sequence, then returns the iterator that designates the inserted element. The second member function returns `insert(x)`, using *it* as a starting place within the controlled sequence to search for the insertion point. (Insertion can occur in amortized constant time, instead of logarithmic time, if the insertion point immediately follows it.) The third member function inserts the sequence of element values, for each *it* in the range `[first, last)`, by calling `insert(*it)`.

If an exception is thrown during the insertion of a single element, the container is left unaltered and the exception is rethrown. If an exception is thrown during the insertion of multiple elements, the container is left in a stable but unspecified state and the exception is rethrown.

`multimap::iterator`

```
typedef T0 iterator;
```

The type describes an object that can serve as a bidirectional iterator for the controlled sequence. It is described here as a synonym for the implementation-defined type *T0*.

`multimap::key_comp`

```
key_compare key_comp() const;
```

The member function returns the stored function object that determines the order of elements in the controlled sequence. The stored object defines the member function:

```
bool operator()(const Key& x, const Key& y);
```

which returns true if *x* strictly precedes *y* in the sort order.

`multimap::key_compare`

```
typedef Pred key_compare;
```

The type describes a function object that can compare two sort keys to determine the relative order of two elements in the controlled sequence.

`multimap::key_type`

```
typedef Key key_type;
```

The type describes the sort key object stored in each element of the controlled sequence.

`multimap::lower_bound`

```
iterator lower_bound(const Key& key);  
const_iterator lower_bound(const Key& key) const;
```

The member function returns an iterator that designates the earliest element *x* in the controlled sequence for which `key_comp()(x, key)` is false.

If no such element exists, the function returns `end()`.

`multimap::mapped_type`

```
typedef T mapped_type;
```

The type is a synonym for the template parameter *T*.

multimap::max_size

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

The member function returns the length of the longest sequence that the object can control.

multimap::multimap

```
multimap();  
explicit multimap(const Pred& comp);  
multimap(const Pred& comp, const A& al);  
multimap(const multimap& x);  
template<class InIt>  
    multimap(InIt first, InIt last);  
template<class InIt>  
    multimap(InIt first, InIt last,  
            const Pred& comp);  
template<class InIt>  
    multimap(InIt first, InIt last,  
            const Pred& comp, const A& al);
```

All constructors store an allocator object (page 337) and initialize the controlled sequence. The allocator object is the argument `al`, if present. For the copy constructor, it is `x.get_allocator()` (page 331)(). Otherwise, it is `A()`.

All constructors also store a function object that can later be returned by calling `key_comp()`. The function object is the argument `comp`, if present. For the copy constructor, it is `x.key_comp()` (page 332)(). Otherwise, it is `Pred()`.

The first three constructors specify an empty initial controlled sequence. The fourth constructor specifies a copy of the sequence controlled by `x`. The last three constructors specify the sequence of element values [`first`, `last`).

multimap::pointer

```
typedef A::pointer pointer;
```

The type describes an object that can serve as a pointer to an element of the controlled sequence.

multimap::rbegin

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

The member function returns a reverse bidirectional iterator that points just beyond the end of the controlled sequence. Hence, it designates the beginning of the reverse sequence.

multimap::reference

```
typedef A::reference reference;
```

The type describes an object that can serve as a reference to an element of the controlled sequence.

multimap::rend

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

The member function returns a reverse bidirectional iterator that points at the first element of the sequence (or just beyond the end of an empty sequence). Hence, it designates the end of the reverse sequence.

multimap::reverse_iterator

```
typedef reverse_iterator<iterator> reverse_iterator;
```

The type describes an object that can serve as a reverse bidirectional iterator for the controlled sequence.

multimap::size

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

The member function returns the length of the controlled sequence.

multimap::size_type

```
typedef T2 size_type;
```

The unsigned integer type describes an object that can represent the length of any controlled sequence. It is described here as a synonym for the implementation-defined type T2.

multimap::swap

```
void swap(multimap& x);
```

The member function swaps the controlled sequences between *this and x. If `get_allocator() == x.get_allocator()`, it does so in constant time, it throws an exception only as a result of copying the stored function object of type Pred, and it invalidates no references, pointers, or iterators that designate elements in the two controlled sequences. Otherwise, it performs a number of element assignments and constructor calls proportional to the number of elements in the two controlled sequences.

multimap::upper_bound

```
iterator upper_bound(const Key& key);  
const_iterator upper_bound(const Key& key) const;
```

The member function returns an iterator that designates the earliest element x in the controlled sequence for which `key_comp()(key, x.first)` is true.

If no such element exists, the function returns `end()`.

multimap::value_comp

```
value_compare value_comp() const;
```

The member function returns a function object that determines the order of elements in the controlled sequence.

multimap::value_compare

```
class value_compare  
    : public binary_function (page 285)<value_type, value_type,  
        bool> {  
public:  
    bool operator()(const value_type& x,  
        const value_type& y) const  
        {return (comp(x.first, x.second)); }  
protected:
```

```

value_compare(key_compare pr)
    : comp(pr) {}
key_compare comp;
};

```

The type describes a function object that can compare the sort keys in two elements to determine their relative order in the controlled sequence. The function object stores an object **comp** of type `key_compare`. The member function **operator()** uses this object to compare the sort-key components of two element.

multimap::value_type

```
typedef pair (page 402)<const Key, T> value_type;
```

The type describes an element of the controlled sequence.

operator!=

```

template<class Key, class T, class Pred, class A>
bool operator!=(
    const map <Key, T, Pred, A>& lhs,
    const map <Key, T, Pred, A>& rhs);
template<class Key, class T, class Pred, class A>
bool operator!=(
    const multimap <Key, T, Pred, A>& lhs,
    const multimap <Key, T, Pred, A>& rhs);

```

The template function returns `!(lhs == rhs)`.

operator==

```

template<class Key, class T, class Pred, class A>
bool operator==(
    const map <Key, T, Pred, A>& lhs,
    const map <Key, T, Pred, A>& rhs);
template<class Key, class T, class Pred, class A>
bool operator==(
    const multimap <Key, T, Pred, A>& lhs,
    const multimap <Key, T, Pred, A>& rhs);

```

The first template function overloads `operator==` to compare two objects of template class `multimap` (page 328). The second template function overloads `operator==` to compare two objects of template class `multimap` (page 328). Both functions return `lhs.size() == rhs.size() && equal(lhs.begin(), lhs.end(), rhs.begin(), rhs.end())`.

operator<

```

template<class Key, class T, class Pred, class A>
bool operator<(
    const map <Key, T, Pred, A>& lhs,
    const map <Key, T, Pred, A>& rhs);
template<class Key, class T, class Pred, class A>
bool operator<(
    const multimap <Key, T, Pred, A>& lhs,
    const multimap <Key, T, Pred, A>& rhs);

```

The first template function overloads `operator<` to compare two objects of template class `multimap` (page 328). The second template function overloads `operator<` to compare two objects of template class `multimap` (page 328). Both functions return `lexicographical_compare(lhs.begin(), lhs.end(), rhs.begin(), rhs.end())`.

operator<=

```
template<class Key, class T, class Pred, class A>
    bool operator<=(
        const map <Key, T, Pred, A>& lhs,
        const map <Key, T, Pred, A>& rhs);
template<class Key, class T, class Pred, class A>
    bool operator<=(
        const multimap <Key, T, Pred, A>& lhs,
        const multimap <Key, T, Pred, A>& rhs);
```

The template function returns `!(rhs < lhs)`.

operator>

```
template<class Key, class T, class Pred, class A>
    bool operator>(
        const map <Key, T, Pred, A>& lhs,
        const map <Key, T, Pred, A>& rhs);
template<class Key, class T, class Pred, class A>
    bool operator>(
        const multimap <Key, T, Pred, A>& lhs,
        const multimap <Key, T, Pred, A>& rhs);
```

The template function returns `rhs < lhs`.

operator>=

```
template<class Key, class T, class Pred, class A>
    bool operator>=(
        const map <Key, T, Pred, A>& lhs,
        const map <Key, T, Pred, A>& rhs);
template<class Key, class T, class Pred, class A>
    bool operator!=(
        const multimap <Key, T, Pred, A>& lhs,
        const multimap <Key, T, Pred, A>& rhs);
```

The template function returns `!(lhs < rhs)`.

swap

```
template<class Key, class T, class Pred, class A>
    void swap(
        map <Key, T, Pred, A>& lhs,
        map <Key, T, Pred, A>& rhs);
template<class Key, class T, class Pred, class A>
    void swap(
        multimap <Key, T, Pred, A>& lhs,
        multimap <Key, T, Pred, A>& rhs);
```

The template function executes `lhs.swap (page 327)(rhs)`.

Portions derived from work copyright © 1994 by Hewlett-Packard Company. All rights reserved.

<memory>

```
namespace std {
    template<class T>
        class allocator;
    template<>
        class allocator<void>;
    template<class FwdIt, class T>
        class raw_storage_iterator;
```



```

template<class T>
class auto_ptr;

    // TEMPLATE OPERATORS
template<class T>
    bool operator==(allocator<T>& lhs,
                    allocator<T>& rhs);
template<class T>
    bool operator!=(allocator<T>& lhs,
                    allocator<T>& rhs);

    // TEMPLATE FUNCTIONS
template<class T>
    pair<T *, ptrdiff_t>
        get_temporary_buffer(ptrdiff_t n);
template<class T>
    void return_temporary_buffer(T *p);
template<class InIt, class FwdIt>
    FwdIt uninitialized_copy(InIt first, InIt last,
                             FwdIt result);
template<class FwdIt, class T>
    void uninitialized_fill(FwdIt first, FwdIt last,
                             const T& x);
template<class FwdIt, class Size, class T>
    void uninitialized_fill_n(FwdIt first, Size n,
                             const T& x);
};

```

Include the STL (page 1) standard header **<memory>** to define a class, an operator, and several templates that help allocate and free objects.

allocator

```

template<class T>
class allocator {
    typedef size_t size_type;
    typedef ptrdiff_t difference_type;
    typedef T *pointer;
    typedef const T *const_pointer;
    typedef T& reference;
    typedef const T& const_reference;
    typedef T value_type;
    pointer address(reference x) const;
    const_pointer address(const_reference x) const;
    template<class U>
        struct rebind;
    allocator();
    template<class U>
        allocator(const allocator<U>& x);
    template<class U>
        allocator& operator=(const allocator<U>& x);
    template<class U>
        pointer allocate(size_type n, const U *hint = 0);
    void deallocate(pointer p, size_type n);
    void construct(pointer p, const T& val);
    void destroy(pointer p);
    size_type max_size() const;
};

```

The template class describes an object that manages storage allocation and freeing for arrays of objects of type T. An object of class **allocator** is the default **allocator object** specified in the constructors for several container template classes in the Standard C++ library.

Template class `allocator` supplies several type definitions that are rather pedestrian. They hardly seem worth defining. But another class with the same members might choose more interesting alternatives. Constructing a container with an allocator object of such a class gives individual control over allocation and freeing of elements controlled by that container.

For example, an allocator object might allocate storage on a **private heap**. Or it might allocate storage on a **far heap**, requiring nonstandard pointers to access the allocated objects. Or it might specify, through the type definitions it supplies, that elements be accessed through special **accessor objects** that manage **shared memory**, or perform automatic **garbage collection**. Hence, a class that allocates storage using an allocator object should use these types religiously for declaring pointer and reference objects (as do the containers in the Standard C++ library).

Thus, an allocator defines the types (among others):

- `pointer` (page 339) — behaves like a pointer to `T`
- `const_pointer` (page 338) — behaves like a const pointer to `T`
- `reference` (page 340) — behaves like a reference to `T`
- `const_reference` (page 339) — behaves like a const reference to `T`

These types specify the form that pointers and references must take for allocated elements. (`allocator::pointer` is not necessarily the same as `T *` for all allocator objects, even though it has this obvious definition for class `allocator`.)

`allocator::address`

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

The member functions return the address of `x`, in the form that pointers must take for allocated elements.

`allocator::allocate`

```
template<class U>
    pointer allocate(size_type n, const U *hint = 0);
```

The member function allocates storage for an array of `n` elements of type `T`, by calling operator `new(n)`. It returns a pointer to the allocated object. The `hint` argument helps some allocators in improving locality of reference — a valid choice is the address of an object earlier allocated by the same allocator object, and not yet deallocated. To supply no hint, use a null pointer argument instead.

`allocator::allocator`

```
allocator();
template<class U>
    allocator(const allocator<U>& x);
```

The constructor does nothing. In general, however, an allocator object constructed from another allocator object should compare equal to it (and hence permit intermixing of object allocation and freeing between the two allocator objects).

`allocator::const_pointer`

```
typedef const T *pointer;
```

The pointer type describes an object `p` that can designate, via the expression `*p`, any const object that an object of template class `allocator` can allocate.

allocator::const_reference

```
typedef const T& const_reference;
```

The reference type describes an object *x* that can designate any const object that an object of template class `allocator` can allocate.

allocator::construct

```
void construct(pointer p, const T& val);
```

The member function constructs an object of type *T* at *p* by evaluating the placement new expression `new ((void *)p) T(val)`.

allocator::deallocate

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

The member function frees storage for the array of *n* objects of type *T* beginning at *p*, by calling `operator delete(p)`. The pointer *p* must have been earlier returned by a call to `allocate` (page 338) for an `allocator` object that compares equal to `*this`, allocating an array object of the same size and type. `deallocate` never throws an exception.

allocator::destroy

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

The member function destroys the object designated by *p*, by calling the destructor `p->T::~~T()`.

allocator::difference_type

```
typedef ptrdiff_t difference_type;
```

The signed integer type describes an object that can represent the difference between the addresses of any two elements in a sequence that an object of template class `allocator` can allocate.

allocator::max_size

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

The member function returns the length of the longest sequence of elements of type *T* that an object of class `allocator` *might* be able to allocate.

allocator::operator=

```
template<class U>  
    allocator& operator=(const allocator<U>& x);
```

The template assignment operator does nothing. In general, however, an `allocator` object assigned to another `allocator` object should compare equal to it (and hence permit intermixing of object allocation and freeing between the two `allocator` objects).

allocator::pointer

```
typedef T *pointer;
```

The pointer type describes an object *p* that can designate, via the expression `*p`, any object that an object of template class `allocator` can allocate.

allocator::rebind

```
template<class U>
    struct rebind {
        typedef allocator<U> other;
    };
```

The member template class defines the type **other**. Its sole purpose is to provide the type name `allocator<U>` given the type name `allocator<T>`.

For example, given an allocator object `a1` of type `A`, you can allocate an object of type `U` with the expression:

```
A::rebind<U>::other(a1).allocate(1, (U *)0)
```

Or, you can simply name its pointer type by writing the type:

```
A::rebind<U>::other::pointer
```

allocator::reference

```
typedef T& reference;
```

The reference type describes an object `x` that can designate any object that an object of template class `allocator` can allocate.

allocator::size_type

```
typedef size_t size_type;
```

The unsigned integer type describes an object that can represent the length of any sequence that an object of template class `allocator` can allocate.

allocator::value_type

```
typedef T value_type;
```

The type is a synonym for the template parameter `T`.

allocator<void>

```
template<>
    class allocator<void> {
        typedef void *pointer;
        typedef const void *const_pointer;
        typedef void value_type;
        template<class U>
            struct rebind;
        allocator();
        template<class U>
            allocator(const allocator<U>);
        template<class U>
            allocator<void> &operator=(const allocator<U>);
    };

```

The class explicitly specializes template class `allocator` (page 337) for type `void`. Its constructors and assignment operator behave the same as for the template class, but it defines only the types `const_pointer` (page 338), `pointer` (page 339), `value_type` (page 340), and the nested template class `rebind` (page 340).

auto_ptr

```
template<U>
    struct auto_ptr_ref;

template<class T>

```

```

class auto_ptr {
public:
    typedef T element_type;
    explicit auto_ptr(T *p = 0) throw();
    auto_ptr(auto_ptr<T>& rhs) throw();
    template<class U>
        auto_ptr(auto_ptr<U>& rhs) throw();
    auto_ptr(auto_ptr_ref<T> rhs) throw();
    ~auto_ptr();
    template<class U>
        operator auto_ptr<U>() throw();
    template<class U>
        operator auto_ptr_ref<U>() throw();
    template<class U>
        auto_ptr<T>& operator=(auto_ptr<U>& rhs) throw();
    auto_ptr<T>& operator=(auto_ptr<T>& rhs) throw();
    auto_ptr<T>& operator=(auto_ptr_ref<T>& rhs) throw();
    T& operator*() const throw();
    T *operator->() const throw();
    T *get() const throw();
    T *release() const throw();
    void reset(T *p = 0);
};

```

The class describes an object that stores a pointer to an allocated object of type T. The stored pointer must either be null or designate an object allocated by a new expression. An object constructed with a non-null pointer owns the pointer. It transfers ownership if its stored value is assigned to another object. (It replaces the stored value after a transfer with a null pointer.) The destructor for `auto_ptr<T>` deletes the allocated object if it owns it. Hence, an object of class `auto_ptr<T>` ensures that an allocated object is automatically deleted when control leaves a block, even via a thrown exception. You should not construct two `auto_ptr<T>` objects that own the same object.

You can pass an `auto_ptr<T>` object by value as an argument to a function call. You can return such an object by value as well. (Both operations depend on the implicit construction of intermediate objects of class `auto_ptr_ref<U>`, by various subtle conversion rules.) You cannot, however, reliably manage a sequence of `auto_ptr<T>` objects with an STL container (page 41).

auto_ptr::auto_ptr

```

explicit auto_ptr(T *p = 0) throw();
auto_ptr(auto_ptr<T>& rhs) throw();
auto_ptr(auto_ptr_ref<T> rhs) throw();
template<class U>
    auto_ptr(auto_ptr<U>& rhs) throw();

```

The first constructor stores p as the pointer to the allocated object. The second constructor transfers ownership of the pointer stored in rhs, by storing rhs.release(). in the constructed object. The third constructor behaves the same as the second, except that it stores rhs.ref.release(), where ref is the reference stored in rhs.

The template constructor behaves the same as the second constructor, provided that a pointer to U can be implicitly converted to a pointer to T.

auto_ptr_ref

```

template<U>
    struct auto_ptr_ref {
        auto_ptr_ref(auto_ptr<U>& rhs);
    };

```

A helper class that describes an object that stores a reference to an object of class `auto_ptr<T>`.

auto_ptr::~~auto_ptr
`~auto_ptr();`

The destructor evaluates the expression `delete q`.

auto_ptr::element_type
`typedef T element_type;`

The type is a synonym for the template parameter `T`.

auto_ptr::get
`T *get() const throw();`

The member function returns the stored pointer.

auto_ptr::operator=
`template<class U>
 auto_ptr<T>& operator=(auto_ptr<U>& rhs) throw();
 auto_ptr<T>& operator=(auto_ptr<>& rhs) throw();
 auto_ptr<T>& operator=(auto_ptr_ref<>& rhs) throw();`

The assignment evaluates the expression `delete q`, but only if the stored pointer value `q` changes as a result of the assignment. It then transfers ownership of the pointer stored in `rhs`, by storing `rhs.release()` in `*this`. The function returns `*this`.

auto_ptr::operator*
`T& operator*() const throw();`

The indirection operator returns `*get()`. Hence, the stored pointer must not be null.

auto_ptr::operator->
`T *operator->() const throw();`

The selection operator returns `get()`, so that the expression `a1->m` behaves the same as `(a1.get())->m`, where `a1` is an object of class `auto_ptr<T>`. Hence, the stored pointer must not be null, and `T` must be a class, structure, or union type with a member `m`.

auto_ptr::operator auto_ptr<U>
`template<class U>
 operator auto_ptr<U>() throw();`

The type cast operator returns `auto_ptr<U>(*this)`.

auto_ptr::operator auto_ptr_ref<U>
`template<class U>
 operator auto_ptr_ref<U>() throw();`

The type cast operator returns `auto_ptr_ref<U>(*this)`.

auto_ptr::release
`T *release() throw();`

The member replaces the stored pointer with a null pointer and returns the previously stored pointer.

auto_ptr::reset

```
void reset(T *p = 0);
```

The member function evaluates the expression `delete q`, but only if the stored pointer value `q` changes as a result of function call. It then replaces the stored pointer with `p`.

get_temporary_buffer

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

The template function allocates storage for a sequence of at most `n` elements of type `T`, from an unspecified source (which may well be the standard heap used by operator `new`). It returns a value `pr`, of type `pair<T *, ptrdiff_t>`. If the function allocates storage, `pr.first` designates the allocated storage and `pr.second` is the number of elements in the longest sequence the storage can hold. Otherwise, `pr.first` is a null pointer.

In this implementation (page 3), if a translator does not support member template functions, the template:

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

is replaced by:

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

operator!=

```
template<class T>
    bool operator!=(allocator<T>& lhs,
                    allocator<T>& rhs);
```

The template operator returns false.

operator==

```
template<class T>
    bool operator==(allocator<T>& lhs,
                    allocator<T>& rhs);
```

The template operator returns true. (Two allocator objects should compare equal only if an object allocated through one can be deallocated through the other. If the value of one object is determined from another by assignment or by construction, the two object should compare equal.)

raw_storage_iterator

```
template<class FwdIt, class T>
    class raw_storage_iterator
        : public iterator<output_iterator_tag,
                        void, void, void, void> {
    public:
```

```

typedef FwdIt iter_type;
typedef T element_type;
explicit raw_storage_iterator(FwdIt it);
raw_storage_iterator<FwdIt, T>& operator*();
raw_storage_iterator<FwdIt, T>&
    operator=(const T& val);
raw_storage_iterator<FwdIt, T>& operator++();
raw_storage_iterator<FwdIt, T> operator++(int);
};

```

The class describes an output iterator that constructs objects of type T in the sequence it generates. An object of class `raw_storage_iterator<FwdIt, T>` accesses storage through a forward iterator object, of class `FwdIt`, that you specify when you construct the object. For an object `it` of class `FwdIt`, the expression `&*it` must designate unconstructed storage for the next object (of type T) in the generated sequence.

raw_storage_iterator::element_type

```
typedef T element_type;
```

The type is a synonym for the template parameter T.

raw_storage_iterator::iter_type

```
typedef FwdIt iter_type;
```

The type is a synonym for the template parameter `FwdIt`.

raw_storage_iterator::operator*

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

The indirection operator returns `*this` (so that `operator=(const T&)` can perform the actual store in an expression such as `*x = val`).

raw_storage_iterator::operator=

```
raw_storage_iterator<FwdIt, T>& operator=(const T& val);
```

The assignment operator constructs the next object in the output sequence using the stored iterator value `it`, by evaluating the placement new expression `new ((void *)&*it) T(val)`. The function returns `*this`.

raw_storage_iterator::operator++

```
raw_storage_iterator<FwdIt, T>& operator++();
raw_storage_iterator<FwdIt, T> operator++(int);
```

The first (preincrement) operator increments the stored output iterator object, then returns `*this`.

The second (postincrement) operator makes a copy of `*this`, increments the stored output iterator object, then returns the copy.

raw_storage_iterator::raw_storage_iterator

```
explicit raw_storage_iterator(FwdIt it);
```

The constructor stores `it` as the output iterator object.

return_temporary_buffer

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


The template function frees the storage designated by `p`, which must be earlier allocated by a call to `get_temporary_buffer` (page 343).

uninitialized_copy

```
template<class InIt, class FwdIt>
FwdIt uninitialized_copy(InIt first, InIt last,
                          FwdIt result);
```

The template function effectively executes:

```
while (first != last)
    new ((void *)&*result++) U(*first++);
return first;
```

where `U` is `iterator_traits<InIt>::value_type`, unless the code throws an exception. In that case, all constructed objects are destroyed and the exception is rethrown.

uninitialized_fill

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

The template function effectively executes:

```
while (first != last)
    new ((void *)&*first++) U(x);
```

where `U` is `iterator_traits<FwdIt>::value_type`, unless the code throws an exception. In that case, all constructed objects are destroyed and the exception is rethrown.

uninitialized_fill_n

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

The template function effectively executes:

```
while (0 < n--)
    new ((void *)&*first++) U(x);
```

where `U` is `iterator_traits<FwdIt>::value_type`, unless the code throws an exception. In that case, all constructed objects are destroyed and the exception is rethrown.

Portions derived from work copyright © 1994 by Hewlett-Packard Company. All rights reserved.

<numeric>

```
namespace std {
template<class InIt, class T>
    T accumulate(InIt first, InIt last, T val);
template<class InIt, class T, class Pred>
    T accumulate(InIt first, InIt last, T val, Pred pr);
template<class InIt1, class InIt2, class T>
    T inner_product(InIt1 first1, InIt1 last1,
                     InIt2 first2, T val);
template<class InIt1, class InIt2, class T,
```

```

        class Pred1, class Pred2>
        T inner_product(InIt1 first1, InIt1 last1,
            Init2 first2, T val, Pred1 pr1, Pred2 pr2);
template<class InIt, class OutIt>
    OutIt partial_sum(InIt first, InIt last,
        OutIt result);
template<class InIt, class OutIt, class Pred>
    OutIt partial_sum(InIt first, InIt last,
        OutIt result, Pred pr);
template<class InIt, class OutIt>
    OutIt adjacent_difference(InIt first, InIt last,
        OutIt result);
template<class InIt, class OutIt, class Pred>
    OutIt adjacent_difference(InIt first, InIt last,
        OutIt result, Pred pr);
};

```

Include the STL (page 1) standard header `<numeric>` to define several template functions useful for computing numeric values. The descriptions of these templates employ a number of conventions (page 38) common to all algorithms.

accumulate

```

template<class InIt, class T>
    T accumulate(InIt first, InIt last, T val);
template<class InIt, class T, class Pred>
    T accumulate(InIt first, InIt last, T val, Pred pr);

```

The first template function repeatedly replaces `val` with `val + *I`, for each value of the `InIt` iterator `I` in the interval `[first, last)`. It then returns `val`.

The second template function repeatedly replaces `val` with `pr(val, *I)`, for each value of the `InIt` iterator `I` in the interval `[first, last)`. It then returns `val`.

adjacent_difference

```

template<class InIt, class OutIt>
    OutIt adjacent_difference(InIt first, InIt last,
        OutIt result);
template<class InIt, class OutIt, class Pred>
    OutIt adjacent_difference(InIt first, InIt last,
        OutIt result, Pred pr);

```

The first template function stores successive values beginning at `result`, for each value of the `InIt` iterator `I` in the interval `[first, last)`. The first value `val` stored (if any) is `*I`. Each subsequent value stored is `*I - val`, and `val` is replaced by `*I`. The function returns `result` incremented `last - first` times.

The second template function stores successive values beginning at `result`, for each value of the `InIt` iterator `I` in the interval `[first, last)`. The first value `val` stored (if any) is `*I`. Each subsequent value stored is `pr(*I, val)`, and `val` is replaced by `*I`. The function returns `result` incremented `last - first` times.

inner_product

```

template<class InIt1, class InIt2, class T>
    T inner_product(InIt1 first1, InIt1 last1,
        Init2 first2, T val);
template<class InIt1, class InIt2, class T,
    class Pred1, class Pred2>
    T inner_product(InIt1 first1, InIt1 last1,
        Init2 first2, T val, Pred1 pr1, Pred2 pr2);

```

The first template function repeatedly replaces `val` with `val + (*I1 * *I2)`, for each value of the `InIt1` iterator `I1` in the interval `[first1, last2)`. In each case, the `InIt2` iterator `I2` equals `first2 + (I1 - first1)`. The function returns `val`.

The first template function repeatedly replaces `val` with `pr1(val, pr2(*I1, *I2))`, for each value of the `InIt1` iterator `I1` in the interval `[first1, last2)`. In each case, the `InIt2` iterator `I2` equals `first2 + (I1 - first1)`. The function returns `val`.

partial_sum

```
template<class InIt, class OutIt>
    OutIt partial_sum(InIt first, InIt last,
        OutIt result);
template<class InIt, class OutIt, class Pred>
    OutIt partial_sum(InIt first, InIt last,
        OutIt result, Pred pr);
```

The first template function stores successive values beginning at `result`, for each value of the `InIt` iterator `I` in the interval `[first, last)`. The first value `val` stored (if any) is `*I`. Each subsequent value `val` stored is `val + *I`. The function returns `result` incremented `last - first` times.

The second template function stores successive values beginning at `result`, for each value of the `InIt` iterator `I` in the interval `[first, last)`. The first value `val` stored (if any) is `*I`. Each subsequent value `val` stored is `pr(val, *I)`. The function returns `result` incremented `last - first` times.

Portions derived from work copyright © 1994 by Hewlett-Packard Company. All rights reserved.

<queue>

```
namespace std {
    template<class T, class Cont>
        class queue;
    template<class T, class Cont, class Pred>
        class priority_queue;

    // TEMPLATE FUNCTIONS
    template<class T, class Cont>
        bool operator==(const queue<T, Cont>& lhs,
            const queue<T, Cont>& rhs);
    template<class T, class Cont>
        bool operator!=(const queue<T, Cont>& lhs,
            const queue<T, Cont>& rhs);
    template<class T, class Cont>
        bool operator<(const queue<T, Cont>& lhs,
            const queue<T, Cont>& rhs);
    template<class T, class Cont>
        bool operator>(const queue<T, Cont>& lhs,
            const queue<T, Cont>& rhs);
    template<class T, class Cont>
        bool operator<=(const queue<T, Cont>& lhs,
            const queue<T, Cont>& rhs);
    template<class T, class Cont>
        bool operator>=(const queue<T, Cont>& lhs,
            const queue<T, Cont>& rhs);
};
```

Include the STL (page 1) standard header **<queue>** to define the template classes `priority_queue` and `queue`, and several supporting templates.

operator!=

```
template<class T, class Cont>
bool operator!=(const queue <T, Cont>& lhs,
               const queue <T, Cont>& rhs);
```

The template function returns `!(lhs == rhs)`.

operator==

```
template<class T, class Cont>
bool operator==(const queue <T, Cont>& lhs,
               const queue <T, Cont>& rhs);
```

The template function overloads `operator==` to compare two objects of template class `queue` (page 351). The function returns `lhs.c (page 351) == rhs.c`.

operator<

```
template<class T, class Cont>
bool operator<(const queue <T, Cont>& lhs,
              const queue <T, Cont>& rhs);
```

The template function overloads `operator<` to compare two objects of template class `queue` (page 351). The function returns `lhs.c (page 351) < rhs.c`.

operator<=

```
template<class T, class Cont>
bool operator<=(const queue <T, Cont>& lhs,
               const queue <T, Cont>& rhs);
```

The template function returns `!(rhs < lhs)`.

operator>

```
template<class T, class Cont>
bool operator>(const queue <T, Cont>& lhs,
               const queue <T, Cont>& rhs);
```

The template function returns `rhs < lhs`.

operator>=

```
template<class T, class Cont>
bool operator>=(const queue <T, Cont>& lhs,
               const queue <T, Cont>& rhs);
```

The template function returns `!(lhs < rhs)`.

priority_queue

```
template<class T,
        class Cont = vector<T>,
        class Pred = less<typename Cont::value_type> >
class priority_queue {
public:
    typedef Cont container_type;
    typedef typename Cont::value_type value_type;
    typedef typename Cont::size_type size_type;
    priority_queue();
    explicit priority_queue(const Pred& pr);
    priority_queue(const Pred& pr,
```

```

        const container_type& cont);
priority_queue(const priority_queue& x);
template<class InIt>
    priority_queue(InIt first, InIt last);
template<class InIt>
    priority_queue(InIt first, InIt last,
        const Pred& pr);
template<class InIt>
    priority_queue(InIt first, InIt last,
        const Pred& pr, const container_type& cont);
bool empty() const;
size_type size() const;
const value_type& top() const;
void push(const value_type& x);
void pop();
protected:
    Cont c;
    Pred comp;
};

```

The template class describes an object that controls a varying-length sequence of elements. The object allocates and frees storage for the sequence it controls through a protected object named **c**, of class **Cont**. The type **T** of elements in the controlled sequence must match **value_type** (page 351).

The sequence is ordered using a protected object named **comp**. After each insertion or removal of the top element (at position zero), for the iterators **P0** and **Pi** designating elements at positions 0 and *i*, **comp(*P0, *Pi)** is false. (For the default template parameter **less<typename Cont::value_type>** the top element of the sequence compares largest, or highest priority.)

An object of class **Cont** must supply random-access iterators and several public members defined the same as for **deque** (page 274) and **vector** (page 404) (both of which are suitable candidates for class **Cont**). The required members are:

```

typedef T value_type;
typedef T0 size_type;
typedef T1 iterator;
Cont();
template<class InIt>
    Cont(InIt first, InIt last);
template<class InIt>
    void insert(iterator it, InIt first, InIt last);
iterator begin();
iterator end();
bool empty() const;
size_type size() const;
const value_type& front() const;
void push_back(const value_type& x);
void pop_back();

```

Here, **T0** and **T1** are unspecified types that meet the stated requirements.

priority_queue::container_type

```
typedef typename Cont::container_type container_type;
```

The type is a synonym for the template parameter **Cont**.

priority_queue::empty

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

The member function returns true for an empty controlled sequence.

priority_queue::pop

```
void pop();
```

The member function removes the first element of the controlled sequence, which must be non-empty, then reorders it.

priority_queue::priority_queue

```
priority_queue();  
explicit priority_queue(const Pred& pr);  
priority_queue(const Pred& pr,  
               const container_type& cont);  
priority_queue(const priority_queue& x);  
template<class InIt>  
  priority_queue(InIt first, InIt last);  
template<class InIt>  
  priority_queue(InIt first, InIt last,  
                const Pred& pr);  
template<class InIt>  
  priority_queue(InIt first, InIt last,  
                const Pred& pr, const container_type& cont);
```

All constructors with an argument `cont` initialize the stored object with `c(cont)`. The remaining constructors initialize the stored object with `c`, to specify an empty initial controlled sequence. The last three constructors then call `c.insert(c.end(), first, last)`.

All constructors also store a function object in `comp` (page 349). The function object `pr` is the argument `pr`, if present. For the copy constructor, it is `x.comp`. Otherwise, it is `Pred()`.

A non-empty initial controlled sequence is then ordered by calling `make_heap(c.begin(), c.end(), comp)`.

priority_queue::push

```
void push(const T& x);
```

The member function inserts an element with value `x` at the end of the controlled sequence, then reorders it.

priority_queue::size

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

The member function returns the length of the controlled sequence.

priority_queue::size_type

```
typedef typename Cont::size_type size_type;
```

The type is a synonym for `Cont::size_type`.

priority_queue::top

```
const value_type& top() const;
```

The member function returns a reference to the first (highest priority) element of the controlled sequence, which must be non-empty.

priority_queue::value_type

typedef typename Cont::value_type **value_type**;

The type is a synonym for Cont::value_type.

queue

```
template<class T,
        class Cont = deque<T> >
class queue {
public:
    typedef Cont container_type;
    typedef typename Cont::value_type value_type;
    typedef typename Cont::size_type size_type;
    queue();
    explicit queue(const container_type& cont);
    bool empty() const;
    size_type size() const;
    value_type& back();
    const value_type& back() const;
    value_type& front();
    const value_type& front() const;
    void push(const value_type& x);
    void pop();
protected:
    Cont c;
};
```

The template class describes an object that controls a varying-length sequence of elements. The object allocates and frees storage for the sequence it controls through a protected object named **c**, of class Cont. The type T of elements in the controlled sequence must match value_type (page 352).

An object of class Cont must supply several public members defined the same as for deque (page 274) and list (page 310) (both of which are suitable candidates for class Cont). The required members are:

```
typedef T value_type;
typedef T0 size_type;
Cont();
bool empty() const;
size_type size() const;
value_type& front();
const value_type& front() const;
value_type& back();
const value_type& back() const;
void push_back(const value_type& x);
void pop_front();
bool operator==(const Cont& X) const;
bool operator!=(const Cont& X) const;
bool operator<(const Cont& X) const;
bool operator>(const Cont& X) const;
bool operator<=(const Cont& X) const;
bool operator>=(const Cont& X) const;
```

Here, T0 is an unspecified type that meets the stated requirements.

queue::back

```
value_type& back();
const value_type& back() const;
```

The member function returns a reference to the last element of the controlled sequence, which must be non-empty.

queue::container_type

```
typedef Cont container_type;
```

The type is a synonym for the template parameter Cont.

queue::empty

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

The member function returns true for an empty controlled sequence.

queue::front

```
value_type& front();  
const value_type& front() const;
```

The member function returns a reference to the first element of the controlled sequence, which must be non-empty.

queue::pop

```
void pop();
```

The member function removes the last element of the controlled sequence, which must be non-empty.

queue::push

```
void push(const T& x);
```

The member function inserts an element with value x at the end of the controlled sequence.

queue::queue

```
queue();  
explicit queue(const container_type& cont);
```

The first constructor initializes the stored object with `c()`, to specify an empty initial controlled sequence. The second constructor initializes the stored object with `c(cont)`, to specify an initial controlled sequence that is a copy of the sequence controlled by cont.

queue::size

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

The member function returns the length of the controlled sequence.

queue::size_type

```
typedef typename Cont::size_type size_type;
```

The type is a synonym for `Cont::size_type`.

queue::value_type

```
typedef typename Cont::value_type value_type;
```

The type is a synonym for `Cont::value_type`.

Portions derived from work copyright © 1994 by Hewlett-Packard Company. All rights reserved.

<set>

```
namespace std {
template<class Key, class Pred, class A>
    class set;
template<class Key, class Pred, class A>
    class multiset;

    // TEMPLATE FUNCTIONS
template<class Key, class Pred, class A>
    bool operator==(
        const set<Key, Pred, A>& lhs,
        const set<Key, Pred, A>& rhs);
template<class Key, class Pred, class A>
    bool operator==(
        const multiset<Key, Pred, A>& lhs,
        const multiset<Key, Pred, A>& rhs);
template<class Key, class Pred, class A>
    bool operator!=(
        const set<Key, Pred, A>& lhs,
        const set<Key, Pred, A>& rhs);
template<class Key, class Pred, class A>
    bool operator!=(
        const multiset<Key, Pred, A>& lhs,
        const multiset<Key, Pred, A>& rhs);
template<class Key, class Pred, class A>
    bool operator<(
        const set<Key, Pred, A>& lhs,
        const set<Key, Pred, A>& rhs);
template<class Key, class Pred, class A>
    bool operator<(
        const multiset<Key, Pred, A>& lhs,
        const multiset<Key, Pred, A>& rhs);
template<class Key, class Pred, class A>
    bool operator>(
        const set<Key, Pred, A>& lhs,
        const set<Key, Pred, A>& rhs);
template<class Key, class Pred, class A>
    bool operator>(
        const multiset<Key, Pred, A>& lhs,
        const multiset<Key, Pred, A>& rhs);
template<class Key, class Pred, class A>
    bool operator<=(
        const set<Key, Pred, A>& lhs,
        const set<Key, Pred, A>& rhs);
template<class Key, class Pred, class A>
    bool operator<=(
        const multiset<Key, Pred, A>& lhs,
        const multiset<Key, Pred, A>& rhs);
template<class Key, class Pred, class A>
    bool operator>=(
        const set<Key, Pred, A>& lhs,
        const set<Key, Pred, A>& rhs);
template<class Key, class Pred, class A>
    bool operator>=(
        const multiset<Key, Pred, A>& lhs,
        const multiset<Key, Pred, A>& rhs);
template<class Key, class Pred, class A>
    void swap(
        set<Key, Pred, A>& lhs,
        set<Key, Pred, A>& rhs);
template<class Key, class Pred, class A>
    void swap(
        multiset<Key, Pred, A>& lhs,
        multiset<Key, Pred, A>& rhs);
};
```

Include the STL (page 1) standard header `<set>` to define the container (page 41) template classes `set` and `multiset`, and their supporting templates.

multiset

`allocator_type` (page 355) · `begin` (page 355) · `clear` (page 355) · `const_iterator` (page 355) · `const_pointer` (page 355) · `const_reference` (page 356) · `const_reverse_iterator` (page 356) · `count` (page 356) · `difference_type` (page 356) · `empty` (page 356) · `end` (page 356) · `equal_range` (page 356) · `erase` (page 356) · `find` (page 357) · `get_allocator` (page 357) · `insert` (page 357) · `iterator` (page 357) · `key_comp` (page 357) · `key_compare` (page 357) · `key_type` (page 358) · `lower_bound` (page 358) · `max_size` (page 358) · `multiset` (page 358) · `pointer` (page 358) · `rbegin` (page 358) · `reference` (page 359) · `rend` (page 359) · `reverse_iterator` (page 359) · `size` (page 359) · `size_type` (page 359) · `swap` (page 359) · `upper_bound` (page 359) · `value_comp` (page 359) · `value_compare` (page 360) · `value_type` (page 360)

```
template<class Key, class Pred = less<Key>,
        class A = allocator<Key> >
class multiset {
public:
    typedef Key key_type;
    typedef Pred key_compare;
    typedef Key value_type;
    typedef Pred value_compare;
    typedef A allocator_type;
    typedef A::pointer pointer;
    typedef A::const_pointer const_pointer;
    typedef A::reference reference;
    typedef A::const_reference const_reference;
    typedef T0 iterator;
    typedef T1 const_iterator;
    typedef T2 size_type;
    typedef T3 difference_type;
    typedef reverse_iterator<const_iterator>
        const_reverse_iterator;
    typedef reverse_iterator<iterator> reverse_iterator;
    multiset();
    explicit multiset(const Pred& comp);
    multiset(const Pred& comp, const A& al);
    multiset(const multiset& x);
    template<class InIt>
        multiset(InIt first, InIt last);
    template<class InIt>
        multiset(InIt first, InIt last,
            const Pred& comp);
    template<class InIt>
        multiset(InIt first, InIt last,
            const Pred& comp, const A& al);
    const_iterator begin() const;
    const_iterator end() const;
    const_reverse_iterator rbegin() const;
    const_reverse_iterator rend() const;
    size_type size() const;
    size_type max_size() const;
    bool empty() const;
    A get_allocator() const;
    iterator insert(const value_type& x);
    iterator insert(iterator it, const value_type& x);
    template<class InIt>
        void insert(InIt first, InIt last);
    iterator erase(iterator it);
    iterator erase(iterator first, iterator last);
    size_type erase(const Key& key);
    void clear();
```

```

void swap(multiset& x);
key_compare key_comp() const;
value_compare value_comp() const;
const_iterator find(const Key& key) const;
size_type count(const Key& key) const;
const_iterator lower_bound(const Key& key) const;
const_iterator upper_bound(const Key& key) const;
pair<const_iterator, const_iterator>
    equal_range(const Key& key) const;
};

```

The template class describes an object that controls a varying-length sequence of elements of type `const Key`. The sequence is ordered by (page 39) the predicate `Pred`. Each element serves as both a **sort key** and a **value**. The sequence is represented in a way that permits lookup, insertion, and removal of an arbitrary element with a number of operations proportional to the logarithm of the number of elements in the sequence (logarithmic time). Moreover, inserting an element invalidates no iterators, and removing an element invalidates only those iterators which point at the removed element.

The object orders the sequence it controls by calling a stored **function object** of type `Pred`. You access this stored object by calling the member function `key_comp()`. Such a function object must impose a total ordering (page 401) on sort keys of type `Key`. For any element `x` that precedes `y` in the sequence, `key_comp()(y, x)` is false. (For the default function object `less<Key>`, sort keys never decrease in value.) Unlike template class `set` (page 361), an object of template class `multiset` does not ensure that `key_comp()(x, y)` is true. (Keys need not be unique.)

The object allocates and frees storage for the sequence it controls through a stored allocator object (page 337) of class `A`. Such an allocator object must have the same external interface as an object of template class `allocator` (page 337). Note that the stored allocator object is *not* copied when the container object is assigned.

multiset::allocator_type

```
typedef A allocator_type;
```

The type is a synonym for the template parameter `A`.

multiset::begin

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

The member function returns a bidirectional iterator that points at the first element of the sequence (or just beyond the end of an empty sequence).

multiset::clear

```
void clear();
```

The member function calls `erase(begin(), iset::end())`.

multiset::const_iterator

```
typedef T1 const_iterator;
```

The type describes an object that can serve as a constant bidirectional iterator for the controlled sequence. It is described here as a synonym for the implementation-defined type `T1`.

multiset::const_pointer

```
typedef A::const_pointer const_pointer;
```

The type describes an object that can serve as a constant pointer to an element of the controlled sequence.

multiset::const_reference

```
typedef A::const_reference const_reference;
```

The type describes an object that can serve as a constant reference to an element of the controlled sequence.

multiset::const_reverse_iterator

```
typedef reverse_iterator<const_iterator>  
const_reverse_iterator;
```

The type describes an object that can serve as a constant reverse bidirectional iterator for the controlled sequence.

multiset::count

```
size_type count(const Key& key) const;
```

The member function returns the number of elements *x* in the range `[lower_bound(key), upper_bound(key))`.

multiset::difference_type

```
typedef T3 difference_type;
```

The signed integer type describes an object that can represent the difference between the addresses of any two elements in the controlled sequence. It is described here as a synonym for the implementation-defined type *T3*.

multiset::empty

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

The member function returns true for an empty controlled sequence.

multiset::end

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

The member function returns a bidirectional iterator that points just beyond the end of the sequence.

multiset::equal_range

```
pair<const_iterator, const_iterator>  
equal_range(const Key& key) const;
```

The member function returns a pair of iterators *x* such that `x.first == lower_bound(key)` and `x.second == upper_bound(key)`.

multiset::erase

```
iterator erase(iterator it);  
iterator erase(iterator first, iterator last);  
size_type erase(const Key& key);
```

The first member function removes the element of the controlled sequence pointed to by *it*. The second member function removes the elements in the range `[first, last)`. Both return an iterator that designates the first element remaining beyond any elements removed, or `end()` if no such element exists.

The third member removes the elements with sort keys in the range `[lower_bound(key), upper_bound(key))`. It returns the number of elements it removes.

The member functions never throw an exception.

multiset::find

```
const_iterator find(const Key& key) const;
```

The member function returns an iterator that designates the earliest element in the controlled sequence whose sort key has equivalent ordering (page 39) to key. If no such element exists, the function returns `end()`.

multiset::get_allocator

```
A get_allocator() const;
```

The member function returns the stored allocator object (page 337).

multiset::insert

```
iterator insert(const value_type& x);  
iterator insert(iterator it, const value_type& x);  
template<class InIt>  
    void insert(InIt first, InIt last);
```

The first member function inserts the element `x` in the controlled sequence, then returns the iterator that designates the inserted element. The second member function returns `insert(x)`, using it as a starting place within the controlled sequence to search for the insertion point. (Insertion can occur in amortized constant time, instead of logarithmic time, if the insertion point immediately follows it.) The third member function inserts the sequence of element values, for each it in the range `[first, last)`, by calling `insert(*it)`.

If an exception is thrown during the insertion of a single element, the container is left unaltered and the exception is rethrown. If an exception is thrown during the insertion of multiple elements, the container is left in a stable but unspecified state and the exception is rethrown.

multiset::iterator

```
typedef T0 iterator;
```

The type describes an object that can serve as a bidirectional iterator for the controlled sequence. It is described here as a synonym for the implementation-defined type `T0`.

multiset::key_comp

```
key_compare key_comp() const;
```

The member function returns the stored function object that determines the order of elements in the controlled sequence. The stored object defines the member function:

```
bool operator(const Key& x, const Key& y);
```

which returns true if `x` strictly precedes `y` in the sort order.

multiset::key_compare

```
typedef Pred key_compare;
```

The type describes a function object that can compare two sort keys to determine the relative order of two elements in the controlled sequence.

multiset::key_type

```
typedef Key key_type;
```

The type describes the sort key object which constitutes each element of the controlled sequence.

multiset::lower_bound

```
const_iterator lower_bound(const Key& key) const;
```

The member function returns an iterator that designates the earliest element *x* in the controlled sequence for which `key_comp()(x, key)` is false.

If no such element exists, the function returns `end()`.

multiset::max_size

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

The member function returns the length of the longest sequence that the object can control.

multiset::multiset

```
multiset();  
explicit multiset(const Pred& comp);  
multiset(const Pred& comp, const A& al);  
multiset(const multiset& x);  
template<class InIt>  
    multiset(InIt first, InIt last);  
template<class InIt>  
    multiset(InIt first, InIt last,  
            const Pred& comp);  
template<class InIt>  
    multiset(InIt first, InIt last,  
            const Pred& comp, const A& al);
```

All constructors store an allocator object (page 337) and initialize the controlled sequence. The allocator object is the argument *al*, if present. For the copy constructor, it is `x.get_allocator()`. Otherwise, it is `A()`.

All constructors also store a function object that can later be returned by calling `key_comp()`. The function object is the argument *comp*, if present. For the copy constructor, it is `x.key_comp()`. Otherwise, it is `Pred()`.

The first three constructors specify an empty initial controlled sequence. The fourth constructor specifies a copy of the sequence controlled by *x*. The last three constructors specify the sequence of element values [*first*, *last*).

multiset::pointer

```
typedef A::pointer pointer;
```

The type describes an object that can serve as a pointer to an element of the controlled sequence.

multiset::rbegin

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

The member function returns a reverse bidirectional iterator that points just beyond the end of the controlled sequence. Hence, it designates the beginning of the reverse sequence.

multiset::reference

```
typedef A::reference reference;
```

The type describes an object that can serve as a reference to an element of the controlled sequence.

multiset::rend

```
const_reverse_iterator rend() const;
```

The member function returns a reverse bidirectional iterator that points at the first element of the sequence (or just beyond the end of an empty sequence). Hence, it designates the end of the reverse sequence.

multiset::reverse_iterator

```
typedef reverse_iterator<iterator> reverse_iterator;
```

The type describes an object that can serve as a reverse bidirectional iterator for the controlled sequence.

multiset::size

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

The member function returns the length of the controlled sequence.

multiset::size_type

```
typedef T2 size_type;
```

The unsigned integer type describes an object that can represent the length of any controlled sequence. It is described here as a synonym for the implementation-defined type T2.

multiset::swap

```
void swap(multiset& x);
```

The member function swaps the controlled sequences between *this and x. If `get_allocator() == x.get_allocator()`, it does so in constant time, it throws an exception only as a result of copying the stored function object of type Pred, and it invalidates no references, pointers, or iterators that designate elements in the two controlled sequences. Otherwise, it performs a number of element assignments and constructor calls proportional to the number of elements in the two controlled sequences.

multiset::upper_bound

```
const_iterator upper_bound(const Key& key) const;
```

The member function returns an iterator that designates the earliest element x in the controlled sequence for which `key_comp()(key, x)` is true.

If no such element exists, the function returns `end()`.

multiset::value_comp

```
value_compare value_comp() const;
```

The member function returns a function object that determines the order of elements in the controlled sequence.

multiset::value_compare

typedef Pred **value_compare**;

The type describes a function object that can compare two elements as sort keys to determine their relative order in the controlled sequence.

multiset::value_type

typedef Key **value_type**;

The type describes an element of the controlled sequence.

operator!=

```
template<class Key, class Pred, class A>
    bool operator!=(
        const set <Key, Pred, A>& lhs,
        const set <Key, Pred, A>& rhs);
template<class Key, class Pred, class A>
    bool operator!=(
        const multiset <Key, Pred, A>& lhs,
        const multiset <Key, Pred, A>& rhs);
```

The template function returns `!(lhs == rhs)`.

operator==

```
template<class Key, class Pred, class A>
    bool operator==(
        const set <Key, Pred, A>& lhs,
        const set <Key, Pred, A>& rhs);
template<class Key, class Pred, class A>
    bool operator==(
        const multiset <Key, Pred, A>& lhs,
        const multiset <Key, Pred, A>& rhs);
```

The first template function overloads `operator==` to compare two objects of template class `multiset` (page 354). The second template function overloads `operator==` to compare two objects of template class `multiset` (page 354). Both functions return `lhs.size() == rhs.size() && equal(lhs.begin(), lhs.end(), rhs.begin())`.

operator<

```
template<class Key, class Pred, class A>
    bool operator<(
        const set <Key, Pred, A>& lhs,
        const set <Key, Pred, A>& rhs);
template<class Key, class Pred, class A>
    bool operator<(
        const multiset <Key, Pred, A>& lhs,
        const multiset <Key, Pred, A>& rhs);
```

The first template function overloads `operator<` to compare two objects of template class `multiset` (page 354). The second template function overloads `operator<` to compare two objects of template class `multiset` (page 354). Both functions return `lexicographical_compare(lhs.begin(), lhs.end(), rhs.begin(), rhs.end())`.

operator<=

```
template<class Key, class Pred, class A>
    bool operator<=(
        const set <Key, Pred, A>& lhs,
        const set <Key, Pred, A>& rhs);
template<class Key, class Pred, class A>
    bool operator<=(
        const multiset <Key, Pred, A>& lhs,
        const multiset <Key, Pred, A>& rhs);
```

The template function returns `!(rhs < lhs)`.

operator>

```
template<class Key, class Pred, class A>
    bool operator>(
        const set <Key, Pred, A>& lhs,
        const set <Key, Pred, A>& rhs);
template<class Key, class Pred, class A>
    bool operator>(
        const multiset <Key, Pred, A>& lhs,
        const multiset <Key, Pred, A>& rhs);
```

The template function returns `rhs < lhs`.

operator>=

```
template<class Key, class Pred, class A>
    bool operator>=(
        const set <Key, Pred, A>& lhs,
        const set <Key, Pred, A>& rhs);
template<class Key, class Pred, class A>
    bool operator>=(
        const multiset <Key, Pred, A>& lhs,
        const multiset <Key, Pred, A>& rhs);
```

The template function returns `!(lhs < rhs)`.

set

`allocator_type` (page 363) · `begin` (page 363) · `clear` (page 363) · `const_iterator` (page 363) · `const_pointer` (page 363) · `const_reference` (page 363) · `const_reverse_iterator` (page 363) · `count` (page 363) · `difference_type` (page 363) · `empty` (page 364) · `end` (page 364) · `equal_range` (page 364) · `erase` (page 364) · `find` (page 364) · `get_allocator` (page 364) · `insert` (page 364) · `iterator` (page 365) · `key_comp` (page 365) · `key_compare` (page 365) · `key_type` (page 365) · `lower_bound` (page 365) · `max_size` (page 365) · `pointer` (page 366) · `rbegin` (page 366) · `reference` (page 366) · `rend` (page 366) · `reverse_iterator` (page 366) · `set` (page 366) · `size` (page 367) · `size_type` (page 367) · `swap` (page 367) · `upper_bound` (page 367) · `value_comp` (page 367) · `value_compare` (page 367) · `value_type` (page 367)

```
template<class Key, class Pred = less<Key>,
        class A = allocator<Key> >
    class set {
    public:
        typedef Key key_type;
        typedef Pred key_compare;
        typedef Key value_type;
        typedef Pred value_compare;
        typedef A allocator_type;
        typedef A::pointer pointer;
        typedef A::const_pointer const_pointer;
```

```

typedef A::reference reference;
typedef A::const_reference const_reference;
typedef T0 iterator;
typedef T1 const_iterator;
typedef T2 size_type;
typedef T3 difference_type;
typedef reverse_iterator<const_iterator>
    const_reverse_iterator;
typedef reverse_iterator<iterator> reverse_iterator;
set();
explicit set(const Pred& comp);
set(const Pred& comp, const A& al);
set(const set& x);
template<class InIt>
    set(InIt first, InIt last);
template<class InIt>
    set(InIt first, InIt last,
        const Pred& comp);
template<class InIt>
    set(InIt first, InIt last,
        const Pred& comp, const A& al);
const_iterator begin() const;
const_iterator end() const;
const_reverse_iterator rbegin() const;
const_reverse_iterator rend() const;
size_type size() const;
size_type max_size() const;
bool empty() const;
A get_allocator() const;
pair<iterator, bool> insert(const value_type& x);
iterator insert(iterator it, const value_type& x);
template<class InIt>
    void insert(InIt first, InIt last);
iterator erase(iterator it);
iterator erase(iterator first, iterator last);
size_type erase(const Key& key);
void clear();
void swap(set& x);
key_compare key_comp() const;
value_compare value_comp() const;
const_iterator find(const Key& key) const;
size_type count(const Key& key) const;
const_iterator lower_bound(const Key& key) const;
const_iterator upper_bound(const Key& key) const;
pair<const_iterator, const_iterator>
    equal_range(const Key& key) const;
};

```

The template class describes an object that controls a varying-length sequence of elements of type `const Key`. The sequence is ordered by (page 39) the predicate `Pred`. Each element serves as both a **sort key** and a **value**. The sequence is represented in a way that permits lookup, insertion, and removal of an arbitrary element with a number of operations proportional to the logarithm of the number of elements in the sequence (logarithmic time). Moreover, inserting an element invalidates no iterators, and removing an element invalidates only those iterators which point at the removed element.

The object orders the sequence it controls by calling a stored **function object** of type `Pred`. You access this stored object by calling the member function `key_comp()`. Such a function object must impose a total ordering (page 401) on sort keys of type `Key`. For any element `x` that precedes `y` in the sequence, `key_comp()(y, x)` is false. (For the default function object `less<Key>`, sort keys never decrease in value.) Unlike template class `multiset` (page 354), an object of template class `set` ensures that `key_comp()(x, y)` is true. (Each key is unique.)

The object allocates and frees storage for the sequence it controls through a stored allocator object (page 337) of class A. Such an allocator object must have the same external interface as an object of template class allocator (page 337). Note that the stored allocator object is *not* copied when the container object is assigned.

set::allocator_type

```
typedef A allocator_type;
```

The type is a synonym for the template parameter A.

set::begin

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

The member function returns a bidirectional iterator that points at the first element of the sequence (or just beyond the end of an empty sequence).

set::clear

```
void clear();
```

The member function calls `erase(begin(), end())`.

set::const_iterator

```
typedef T1 const_iterator;
```

The type describes an object that can serve as a constant bidirectional iterator for the controlled sequence. It is described here as a synonym for the implementation-defined type T1.

set::const_pointer

```
typedef A::const_pointer const_pointer;
```

The type describes an object that can serve as a constant pointer to an element of the controlled sequence.

set::const_reference

```
typedef A::const_reference const_reference;
```

The type describes an object that can serve as a constant reference to an element of the controlled sequence.

set::const_reverse_iterator

```
typedef reverse_iterator<const_iterator>  
    const_reverse_iterator;
```

The type describes an object that can serve as a constant reverse bidirectional iterator for the controlled sequence.

set::count

```
size_type count(const Key& key) const;
```

The member function returns the number of elements x in the range `[lower_bound(key), upper_bound(key))`.

set::difference_type

```
typedef T3 difference_type;
```

The signed integer type describes an object that can represent the difference between the addresses of any two elements in the controlled sequence. It is described here as a synonym for the implementation-defined type T3.

set::empty

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

The member function returns true for an empty controlled sequence.

set::end

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

The member function returns a bidirectional iterator that points just beyond the end of the sequence.

set::equal_range

```
pair<const_iterator, const_iterator>  
    equal_range(const Key& key) const;
```

The member function returns a pair of iterators x such that x.first == lower_bound(key) and x.second == upper_bound(key).

set::erase

```
iterator erase(iterator it);  
iterator erase(iterator first, iterator last);  
size_type erase(const Key& key);
```

The first member function removes the element of the controlled sequence pointed to by it. The second member function removes the elements in the range [first, last). Both return an iterator that designates the first element remaining beyond any elements removed, or end() if no such element exists.

The third member removes the elements with sort keys in the range [lower_bound(key), upper_bound(key)). It returns the number of elements it removes.

The member functions never throw an exception.

set::find

```
const_iterator find(const Key& key) const;
```

The member function returns an iterator that designates the earliest element in the controlled sequence whose sort key has equivalent ordering (page 39) to key. If no such element exists, the function returns end().

set::get_allocator

```
A get_allocator() const;
```

The member function returns the stored allocator object.

set::insert

```
pair<iterator, bool> insert(const value_type& x);  
iterator insert(iterator it, const value_type& x);  
template<class InIt>  
    void insert(InIt first, InIt last);
```

The first member function determines whether an element y exists in the sequence whose key has equivalent ordering (page 39) to that of x. If not, it creates such an

element *y* and initializes it with *x*. The function then determines the iterator *it* that designates *y*. If an insertion occurred, the function returns `pair(it, true)`. Otherwise, it returns `pair(it, false)`.

The second member function returns `insert(x)`, using *it* as a starting place within the controlled sequence to search for the insertion point. (Insertion can occur in amortized constant time, instead of logarithmic time, if the insertion point immediately follows *it*.) The third member function inserts the sequence of element values, for each *it* in the range `[first, last)`, by calling `insert(*it)`.

If an exception is thrown during the insertion of a single element, the container is left unaltered and the exception is rethrown. If an exception is thrown during the insertion of multiple elements, the container is left in a stable but unspecified state and the exception is rethrown.

set::iterator

```
typedef T0 iterator;
```

The type describes an object that can serve as a bidirectional iterator for the controlled sequence. It is described here as a synonym for the implementation-defined type `T0`.

set::key_comp

```
key_compare key_comp() const;
```

The member function returns the stored function object that determines the order of elements in the controlled sequence. The stored object defines the member function:

```
bool operator(const Key& x, const Key& y);
```

which returns true if *x* strictly precedes *y* in the sort order.

set::key_compare

```
typedef Pred key_compare;
```

The type describes a function object that can compare two sort keys to determine the relative order of two elements in the controlled sequence.

set::key_type

```
typedef Key key_type;
```

The type describes the sort key object which constitutes each element of the controlled sequence.

set::lower_bound

```
const_iterator lower_bound(const Key& key) const;
```

The member function returns an iterator that designates the earliest element *x* in the controlled sequence for which `key_comp()(x, key)` is false.

If no such element exists, the function returns `end()`.

set::max_size

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

The member function returns the length of the longest sequence that the object can control.

set::pointer

```
typedef A::const_pointer pointer;
```

The type describes an object that can serve as a pointer to an element of the controlled sequence.

set::rbegin

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

The member function returns a reverse bidirectional iterator that points just beyond the end of the controlled sequence. Hence, it designates the beginning of the reverse sequence.

set::reference

```
typedef A::const_reference reference;
```

The type describes an object that can serve as a reference to an element of the controlled sequence.

set::rend

```
const_reverse_iterator rend() const;
```

The member function returns a reverse bidirectional iterator that points at the first element of the sequence (or just beyond the end of an empty sequence). Hence, it designates the end of the reverse sequence.

set::reverse_iterator

```
typedef reverse_iterator<iterator> reverse_iterator;
```

The type describes an object that can serve as a reverse bidirectional iterator for the controlled sequence.

set::set

```
set();  
explicit set(const Pred& comp);  
set(const Pred& comp, const A& a1);  
set(const set& x);  
template<class InIt>  
    set(InIt first, InIt last);  
template<class InIt>  
    set(InIt first, InIt last,  
        const Pred& comp);  
template<class InIt>  
    set(InIt first, InIt last,  
        const Pred& comp, const A& a1);
```

All constructors store an allocator object (page 337) and initialize the controlled sequence. The allocator object is the argument `a1`, if present. For the copy constructor, it is `x.get_allocator()`. Otherwise, it is `A()`.

All constructors also store a function object that can later be returned by calling `key_comp()`. The function object is the argument `comp`, if present. For the copy constructor, it is `x.key_comp()`. Otherwise, it is `Pred()`.

The first three constructors specify an empty initial controlled sequence. The fourth constructor specifies a copy of the sequence controlled by `x`. The last three constructors specify the sequence of element values [`first`, `last`).

set::size

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

The member function returns the length of the controlled sequence.

set::size_type

```
typedef T2 size_type;
```

The unsigned integer type describes an object that can represent the length of any controlled sequence. It is described here as a synonym for the implementation-defined type `T2`.

set::swap

```
void swap(set& x);
```

The member function swaps the controlled sequences between `*this` and `x`. If `get_allocator() == x.get_allocator()`, it does so in constant time, it throws an exception only as a result of copying the stored function object of type `Pred`, and it invalidates no references, pointers, or iterators that designate elements in the two controlled sequences. Otherwise, it performs a number of element assignments and constructor calls proportional to the number of elements in the two controlled sequences.

set::upper_bound

```
const_iterator upper_bound(const Key& key) const;
```

The member function returns an iterator that designates the earliest element `x` in the controlled sequence for which `key_comp()(key, x)` is true.

If no such element exists, the function returns `end()`.

set::value_comp

```
value_compare value_comp() const;
```

The member function returns a function object that determines the order of elements in the controlled sequence.

set::value_compare

```
typedef Pred value_compare;
```

The type describes a function object that can compare two elements as sort keys to determine their relative order in the controlled sequence.

set::value_type

```
typedef Key value_type;
```

The type describes an element of the controlled sequence.

swap

```
template<class Key, class Pred, class A>
void swap(
    multiset <Key, Pred, A>& lhs,
    multiset <Key, Pred, A>& rhs);
```

```
template<class Key, class Pred, class A>
void swap(
    set <Key, Pred, A>& lhs,
    set <Key, Pred, A>& rhs);
```

The template function executes `lhs.swap(rhs)`.

Portions derived from work copyright © 1994 by Hewlett-Packard Company. All rights reserved.

<stack>

```
namespace std {
template<class T, class Cont>
    class stack;

    // TEMPLATE FUNCTIONS
template<class T, class Cont>
    bool operator==(const stack<T, Cont>& lhs,
        const stack<T, Cont>&);
template<class T, class Cont>
    bool operator!=(const stack<T, Cont>& lhs,
        const stack<T, Cont>&);
template<class T, class Cont>
    bool operator<(const stack<T, Cont>& lhs,
        const stack<T, Cont>&);
template<class T, class Cont>
    bool operator>(const stack<T, Cont>& lhs,
        const stack<T, Cont>&);
template<class T, class Cont>
    bool operator<=(const stack<T, Cont>& lhs,
        const stack<T, Cont>&);
template<class T, class Cont>
    bool operator>=(const stack<T, Cont>& lhs,
        const stack<T, Cont>&);
};
```

Include the STL (page 1) standard header **<stack>** to define the template class `stack` and two supporting templates.

operator!=

```
template<class T, class Cont>
    bool operator!=(const stack <T, Cont>& lhs,
        const stack <T, Cont>& rhs);
```

The template function returns `!(lhs == rhs)`.

operator==

```
template<class T, class Cont>
    bool operator==(const stack <T, Cont>& lhs,
        const stack <T, Cont>& rhs);
```

The template function overloads `operator==` to compare two objects of template class `stack` (page 369). The function returns `lhs.c == rhs.c`.

operator<

```
template<class T, class Cont>
    bool operator<(const stack <T, Cont>& lhs,
        const stack <T, Cont>& rhs);
```


The template function overloads `operator<` to compare two objects of template class `stack` (page 369). The function returns `lhs.c < rhs.c`.

operator<=

```
template<class T, class Cont>
bool operator<=(const stack <T, Cont>& lhs,
               const stack <T, Cont>& rhs);
```

The template function returns `!(rhs < lhs)`.

operator>

```
template<class T, class Cont>
bool operator>(const stack <T, Cont>& lhs,
               const stack <T, Cont>& rhs);
```

The template function returns `rhs < lhs`.

operator>=

```
template<class T, class Cont>
bool operator>=(const stack <T, Cont>& lhs,
                const stack <T, Cont>& rhs);
```

The template function returns `!(lhs < rhs)`.

stack

```
template<class T,
        class Cont = deque<T> >
class stack {
public:
    typedef Cont container_type;
    typedef typename Cont::value_type value_type;
    typedef typename Cont::size_type size_type;
    stack();
    explicit stack(const container_type& cont);
    bool empty() const;
    size_type size() const;
    value_type& top();
    const value_type& top() const;
    void push(const value_type& x);
    void pop();
protected:
    Cont c;
};
```

The template class describes an object that controls a varying-length sequence of elements. The object allocates and frees storage for the sequence it controls through a protected object named `c`, of class `Cont`. The type `T` of elements in the controlled sequence must match `value_type` (page 371).

An object of class `Cont` must supply several public members defined the same as for `deque` (page 274), `list` (page 310), and `vector` (page 404) (all of which are suitable candidates for class `Cont`). The required members are:

```
typedef T value_type;
typedef T0 size_type;
Cont();
bool empty() const;
size_type size() const;
value_type& back();
const value_type& back() const;
```

```

    void push_back(const value_type& x);
    void pop_back();
    bool operator==(const Cont& X) const;
    bool operator!=(const Cont& X) const;
    bool operator<(const Cont& X) const;
    bool operator>(const Cont& X) const;
    bool operator<=(const Cont& X) const;
    bool operator>=(const Cont& X) const;

```

Here, `T0` is an unspecified type that meets the stated requirements.

stack::container_type

```
typedef Cont container_type;
```

The type is a synonym for the template parameter `Cont`.

stack::empty

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

The member function returns true for an empty controlled sequence.

stack::pop

```
void pop();
```

The member function removes the last element of the controlled sequence, which must be non-empty.

stack::push

```
void push(const T& x);
```

The member function inserts an element with value `x` at the end of the controlled sequence.

stack::size

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

The member function returns the length of the controlled sequence.

stack::size_type

```
typedef typename Cont::size_type size_type;
```

The type is a synonym for `Cont::size_type`.

stack::stack

```

stack();
explicit stack(const container_type& cont);

```

The first constructor initializes the stored object with `c()`, to specify an empty initial controlled sequence. The second constructor initializes the stored object with `c(cont)`, to specify an initial controlled sequence that is a copy of the sequence controlled by `cont`.

stack::top

```

value_type& top();
const value_type& top() const;

```

The member function returns a reference to the last element of the controlled sequence, which must be non-empty.

stack::value_type

typedef typename Cont::value_type **value_type**;

The type is a synonym for Cont::value_type.

Portions derived from work copyright © 1994 by Hewlett-Packard Company. All rights reserved.

<unordered_map>

Note: To enable this header file, you must define the macro `_VACPP_TR1`.

```
namespace std {
namespace tr1 {
    template <class Key,
               class T,
               class Hash = hash<Key>,
               class Pred = std::equal_to<Key>,
               class Alloc = std::allocator<std::pair<const Key, T> > >
        class unordered_map;

    template <class Key,
               class T,
               class Hash = hash<Key>,
               class Pred = std::equal_to<Key>,
               class Alloc = std::allocator<std::pair<const Key, T> > >
        class unordered_multimap;
}
}
```

Include the STL (page 1) standard header **<unordered_map>** to define the container (page 41) template classes `unordered_map` and `unordered_multimap`, and their supporting templates.

unordered_map

`allocator_type` (page 373) · `begin` (page 373) · `bucket` (page 373) · `bucket_count` (page 374) · `bucket_size` (page 374) · `clear` (page 374) · `const_iterator` (page 374) · `const_local_iterator` (page 374) · `const_pointer` (page 374) · `const_reference` (page 374) · `count` (page 374) · `difference_type` (page 374) · `empty` (page 375) · `end` (page 375) · `equal_range` (page 375) · `erase` (page 375) · `find` (page 375) · `get_allocator` (page 375) · `hash_function` (page 375) · `hasher` (page 375) · `insert` (page 376) · `iterator` (page 376) · `key_eq` (page 376) · `key_equal` (page 376) · `key_type` (page 376) · `load_factor` (page 376) · `local_iterator` (page 376) · `mapped_type` (page 377) · `max_bucket_count` (page 377) · `max_load_factor` (page 377) · `max_size` (page 377) · `operator[]` (page 377) · `pointer` (page 377) · `reference` (page 377) · `rehash` (page 377) · `size` (page 378) · `size_type` (page 378) · `swap` (page 378) · `unordered_map` (page 378) · `value_type` (page 378)

```
namespace std {
namespace tr1 {
    template <class Key,
               class T,
               class Hash = hash<Key>,
               class Pred = std::equal_to<Key>,
               class A= std::allocator<std::pair<const Key, T> > >
        class unordered_map
        {
        public:
            // types
            typedef Key key_type;
            typedef std::pair<const Key, T> value_type;
```

```

typedef T mapped_type;
typedef Hash hasher;
typedef Pred key_equal;
typedef A allocator_type;
typedef typename A::pointer pointer;
typedef typename A::const_pointer const_pointer;
typedef typename A::reference reference;
typedef typename A::const_reference const_reference;
typedef T0 size_type;
typedef T1 difference_type;

typedef T2 iterator;
typedef T3 const_iterator;
typedef T4 local_iterator;
typedef T5 const_local_iterator;

// construct/destroy/copy
explicit unordered_map(size_type n = 3,
                      const hasher& hf = hasher(),
                      const key_equal& eql = key_equal(),
                      const allocator_type& a = allocator_type());
template <class InputIterator>
    unordered_map(InputIterator f, InputIterator l,
                  size_type n = 3,
                  const hasher& hf = hasher(),
                  const key_equal& eql = key_equal(),
                  const allocator_type& a = allocator_type());
unordered_map(const unordered_map&);
~unordered_map();
unordered_map& operator=(const unordered_map&);
A get_allocator() const;

// size and capacity
bool empty() const;
size_type size() const;
size_type max_size() const;

// iterators
iterator begin();
const_iterator begin() const;
iterator end();
const_iterator end() const;

// modifiers
std::pair<iterator, bool> insert(const value_type& obj);
iterator insert(const iterator hint, const value_type& obj);
template <class InputIterator>
    void insert(InputIterator first, InputIterator last);

void erase(const_iterator position);
size_type erase(const key_type& k);
void erase(const_iterator first, const_iterator last);
void clear();

void swap(unordered_map&);

// observers
hasher hash_function() const;
key_equal key_eq() const;

// lookup
iterator find(const key_type& k);
const_iterator find(const key_type& k) const;
size_type count(const key_type& k) const;
std::pair<iterator, iterator>
    equal_range(const key_type& k);
std::pair<const_iterator, const_iterator>

```

```

        equal_range(const key_type& k) const;

mapped_type& operator[](const key_type& k);

// bucket interface
size_type bucket_count() const;
size_type max_bucket_count() const;
size_type bucket_size(size_type n) const;
size_type bucket(const key_type& k) const;
local_iterator begin(size_type n);
const_local_iterator begin(size_type n) const;
local_iterator end(size_type n);
const_local_iterator end(size_type n) const;

// hash policy
float load_factor() const;
float max_load_factor() const;
void max_load_factor(float z);
void rehash(size_type n);
};

template <class Key, class T, class Hash, class Pred, class Alloc>
void swap(const unordered_map<Key, T, Hash, Pred, Alloc>& x,
          const unordered_map<Key, T, Hash, Pred, Alloc>& y);
}
}

```

The template class describes an object that controls a varying-length sequence of elements of type `pair<const Key, T>`. The sequence is unordered. The first element of each pair is the **sort key** and the second is its associated **value**. If you have an optimal hash function, the number of operations performed during lookup, insertion, and removal of an arbitrary element does not depend on the number of elements in the sequence. Moreover, inserting an element invalidates no iterators, and removing an element invalidates only those iterators which point at the removed element.

The object allocates and frees storage for the sequence it controls through a stored allocator object (page 337) of class `A`. Such an allocator object must have the same external interface as an object of template class `allocator` (page 337). Note that the stored allocator object is *not* copied when the container object is assigned.

unordered_map::allocator_type

```
typedef A allocator_type;
```

The type is a synonym for the template parameter `A`.

unordered_map::begin

```

iterator begin();
const_iterator begin() const;

local_iterator begin(size_type n);
const_local_iterator begin(size_type n) const;

```

The member function returns a bidirectional iterator that points at the first element of the sequence (or just beyond the end of an empty sequence).

unordered_map::bucket

```
size_type bucket(const key_type& k) const;
```

The member function returns the index of the bucket that contains the specified key.

unordered_map::bucket_count

```
size_type bucket_count() const;
```

The member function returns the number of buckets that the unordered map contains.

unordered_map::bucket_size

```
size_type bucket_size(size_type n) const;
```

The member function returns the number of elements in the nth bucket.

unordered_map::clear

```
void clear();
```

The member function calls erase(begin(), end()).

unordered_map::const_iterator

```
typedef T3 const_iterator;
```

The type describes an object that can serve as a constant bidirectional iterator for the controlled sequence. It is described here as a synonym for the implementation-defined type T3.

unordered_map::const_local_iterator

```
typedef T5 const_local_iterator;
```

The type describes an object that can serve as a constant bidirectional iterator for the controlled sequence. It is described here as a synonym for the implementation-defined type T5.

unordered_map::const_pointer

```
typedef typename A::const_pointer const_pointer;
```

The type describes an object that can serve as a constant pointer to an element of the controlled sequence.

unordered_map::const_reference

```
typedef typename A::const_reference const_reference;
```

The type describes an object that can serve as a constant reference to an element of the controlled sequence.

unordered_map::count

```
size_type count(const key_type& k) const;
```

The member function returns the number of elements in the map that have a key equivalent to k, based on the key_eq function.

unordered_map::difference_type

```
typedef T1 difference_type;
```

The signed integer type describes an object that can represent the difference between the addresses of any two elements in the controlled sequence. It is described here as a synonym for the implementation-defined type T1.

unordered_map::empty

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

The member function returns true for an empty controlled sequence.

unordered_map::end

```
const_iterator end() const;  
iterator end();  
local_iterator end(size_type n);  
const_local_iterator end(size_type n) const;
```

The member function returns a bidirectional iterator that points just beyond the end of the sequence.

unordered_map::equal_range

```
std::pair<iterator, iterator>  
    equal_range(const key_type& k);  
std::pair<const_iterator, const_iterator>  
    equal_range(const key_type& k) const;
```

The member function returns a range that contains all of the elements with the specified key. It returns `make_pair(end(), end())` if no such elements exist.

unordered_map::erase

```
void erase(const_iterator position);  
size_type erase(const key_type& k);  
void erase(const_iterator first, const_iterator last);
```

The first member function removes the element of the controlled sequence pointed to by it. The second member function removes the elements in the interval `[first, last)`. Both return an iterator that designates the first element remaining beyond any elements removed, or `end()` if no such element exists.

The third member function removes all the elements in the map. It returns the number of elements it removes.

The member functions never throw an exception.

unordered_map::find

```
iterator      find(const key_type& k);  
const_iterator find(const key_type& k) const;
```

The member function returns an iterator that designates the earliest element in the controlled sequence whose sort key has equivalent ordering (page 39) to key. If no such element exists, the function returns `end()`.

unordered_map::get_allocator

```
A get_allocator() const;
```

The member function returns the stored allocator object (page 337).

unordered_map::hasher

```
typedef Hash hasher;
```

The type returns a value of type `std::size_t`.

unordered_map::hash_function

```
hasher hash_function() const;
```

The member function returns the hash function that was used to construct the map.

unordered_map::insert

```
std::pair<iterator, bool> insert(const value_type& obj);  
iterator insert(const_iterator hint, const value_type& obj);  
template <class InputIterator>  
    void insert(InputIterator first, InputIterator last);
```

The first member function determines whether an element *y* exists in the sequence whose key has equivalent ordering (page 39) to that of *obj*. If not, it creates such an element *y* and initializes it with *obj*. The function then determines the iterator *it* that designates *y*. If an insertion occurred, the function returns `pair(it, true)`. Otherwise, it returns `pair(it, false)`.

The second member function returns `insert(obj)`, using *it* as a starting place within the controlled sequence to search for the insertion point. The third member function inserts the sequence of element values, for each *it* in the range `[first, last)`, by calling `insert(*it)`.

If an exception is thrown during the insertion of a single element, the container is left unaltered and the exception is rethrown. If an exception is thrown during the insertion of multiple elements, the container is left in a stable but unspecified state and the exception is rethrown.

unordered_map::iterator

```
typedef T2 iterator;
```

The type describes an object that can serve as a bidirectional iterator for the controlled sequence. It is described here as a synonym for the implementation-defined type `T2`.

unordered_map::key_eq

```
key_equal key_eq() const;
```

The member function returns the key equality function that was used to create the map.

unordered_map::key_equal

```
typedef Pred key_equal;
```

The type returns an equivalence relation.

unordered_map::key_type

```
typedef Key key_type;
```

The type describes the sort key object stored in each element of the controlled sequence.

unordered_map::load_factor

```
float load_factor() const;
```

The member function returns the average number of elements per bucket.

unordered_map::local_iterator

```
typedef T4 local_iterator;
```


The type describes an object that can serve as a constant bidirectional iterator for the controlled sequence. It is described here as a synonym for the implementation-defined type T4.

unordered_map::mapped_type

```
typedef T mapped_type;
```

The type is a synonym for the template parameter T.

unordered_map::max_bucket_count

```
size_type max_bucket_count() const;
```

The member function returns the maximum number of buckets that the unordered map can contain.

unordered_map::max_load_factor

```
float max_load_factor() const;  
void max_load_factor(float z);
```

The first member function returns the maximum value that the container attempts to maintain for the load factor (the average number of elements per bucket). If the load factor increases beyond this value, the container creates more buckets.

The second member function sets the maximum load factor. If this value is less than the current load factor, the unordered map is rehashed.

unordered_map::max_size

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

The member function returns the length of the longest sequence that the object can control.

unordered_map::operator[]

```
mapped_type& operator[](const key_type& k);
```

The member function determines the iterator it as the return value of insert(value_type(k, T())). (It inserts an element with the specified key if no such element exists.) It then returns a reference to (*it).second.

unordered_map::pointer

```
typedef typename A::pointer pointer;
```

The type describes an object that can serve as a pointer to an element of the controlled sequence.

unordered_map::reference

```
typedef typename A::reference reference;
```

The type describes an object that can serve as a reference to an element of the controlled sequence.

unordered_map::rehash

```
void rehash(size_type n);
```

The member function rehashes the unordered map, ensuring that it contains at least n buckets.

unordered_map::size

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

The member function returns the length of the controlled sequence.

unordered_map::size_type

```
typedef T0 size_type;
```

The unsigned integer type describes an object that can represent the length of any controlled sequence. It is described here as a synonym for the implementation-defined type T0.

unordered_map::swap

```
void swap(unordered_map& x);
```

The member function swaps the controlled sequences between *this and x. If `get_allocator() == x.get_allocator()`, it does so in constant time, it throws an exception only as a result of copying the stored function object of type Pred, and it invalidates no references, pointers, or iterators that designate elements in the two controlled sequences. Otherwise, it performs a number of element assignments and constructor calls proportional to the number of elements in the two controlled sequences.

unordered_map::unordered_map

```
explicit unordered_map(size_type n = 3,  
    const hasher& hf = hasher(),  
    const key_equal& eql = key_equal(),  
    const allocator_type& a = allocator_type();  
template <class InputIterator>  
    unordered_map(InputIterator f, InputIterator l,  
        size_type n = 3,  
        const hasher& hf = hasher(),  
        const key_equal& eql = key_equal(),  
        const allocator_type& a = allocator_type();  
unordered_map(const unordered_map&);  
~unordered_map();  
unordered_map& operator=(const unordered_map&);
```

All constructors store an allocator object (page 337) and initialize the controlled sequence. The allocator object is the argument a1, if present. For the copy constructor, it is `x.get_allocator()`. Otherwise, it is `A()`.

The first three constructors specify an empty initial controlled sequence. The fourth constructor specifies a copy of the sequence controlled by x. The last three constructors specify the sequence of element values [first, last).

unordered_map::value_type

```
typedef std::pair<const Key, T> value_type;
```

The type describes an element of the controlled sequence.

unordered_multimap

```
allocator_type (page 380) · begin (page 381) · bucket (page 381) · bucket_count  
(page 381) · bucket_size (page 381) · clear (page 381) · const_iterator (page 381) ·  
const_local_iterator (page 381) · const_pointer (page 381) · const_reference (page  
381) · count (page 382) · difference_type (page 382) · empty (page 382) · end (page  
382) · equal_range (page 382) · erase (page 382) · find (page 382) · get_allocator  
(page 383) · hash_function (page 383) · hasher (page 383) · insert (page 383) ·
```

iterator (page 383) · key_eq (page 383) · key_equal (page 383) · key_type (page 384) · load_factor (page 384) · local_iterator (page 384) · mapped_type (page 384) · max_bucket_count (page 384) · max_load_factor (page 384) · max_size (page 384) · reference (page 384) · rehash (page 385) · size (page 385) · size_type (page 385) · swap (page 385) · unordered_multimap (page 385) · value_type (page 385)

```
namespace std {
namespace tr1 {
    template <class Key,
               class T,
               class Hash = hash<Key>,
               class Pred = std::equal_to<Key>,
               class A= std::allocator<std::pair<const Key, T> > >
    class unordered_multimap
    {
    public:
        // types
        typedef Key                               key_type;
        typedef std::pair<const Key, T>            value_type;
        typedef T                                  mapped_type;
        typedef Hash                              hasher;
        typedef Pred                              key_equal;
        typedef A                                  allocator_type;
        typedef typename A::pointer               pointer;
        typedef typename A::const_pointer         const_pointer;
        typedef typename A::reference             reference;
        typedef typename A::const_reference       const_reference;
        typedef T0                                size_type;
        typedef T1                                difference_type;

        typedef T2                                iterator;
        typedef T3                                const_iterator;
        typedef T4                                local_iterator;
        typedef T5                                const_local_iterator;

        // construct/destroy/copy
        explicit unordered_multimap(size_type n = 3,
                                     const hasher& hf = hasher(),
                                     const key_equal& eql = key_equal(),
                                     const allocator_type& a = allocator_type());

        template <class InputIterator>
        unordered_multimap(InputIterator f, InputIterator l,
                           size_type n = 3,
                           const hasher& hf = hasher(),
                           const key_equal& eql = key_equal(),
                           const allocator_type& a = allocator_type());

        unordered_multimap(const unordered_multimap&);
        ~unordered_multimap();
        unordered_multimap& operator=(const unordered_multimap&);
        A get_allocator() const;

        // size and capacity
        bool empty() const;
        size_type size() const;
        size_type max_size() const;

        // iterators
        iterator begin();
        const_iterator begin() const;
        iterator end();
        const_iterator end() const;

        // modifiers
        iterator insert(const value_type& obj);
        iterator insert(const iterator hint, const value_type& obj);
        template <class InputIterator>
        void insert(InputIterator first, InputIterator last);
```

```

void erase(const_iterator position);
size_type erase(const key_type& k);
void erase(const_iterator first, const_iterator last);
void clear();

void swap(unordered_multimap&);

// observers
hasher hash_function() const;
key_equal key_eq() const;

// lookup
iterator find(const key_type& k);
const_iterator find(const key_type& k) const;
size_type count(const key_type& k) const;
std::pair<iterator, iterator>
equal_range(const key_type& k);
std::pair<const_iterator, const_iterator>
equal_range(const key_type& k) const;

// bucket interface
size_type bucket_count() const;
size_type max_bucket_count() const;
size_type bucket_size(size_type n) const;
size_type bucket(const key_type& k) const;
local_iterator begin(size_type n);
const_local_iterator begin(size_type n) const;
local_iterator end(size_type n);
const_local_iterator end(size_type n) const;

// hash policy
float load_factor() const;
float max_load_factor() const;
void max_load_factor(float z);
void rehash(size_type n);
};

template <class Key, class T, class Hash, class Pred, class Alloc>
void swap(const unordered_multimap<Key, T, Hash, Pred, Alloc>& x,
         const unordered_multimap<Key, T, Hash, Pred, Alloc>& y);
}
}

```

The template class describes an object that controls a varying-length sequence of elements of type `pair<const Key, T>`. The sequence is unordered. The first element of each pair is the **sort key** and the second is its associated **value**. If you have an optimal hash function, the number of operations performed during lookup, insertion, and removal of an arbitrary element does not depend on the number of elements in the sequence. Moreover, inserting an element invalidates no iterators, and removing an element invalidates only those iterators which point at the removed element.

The object allocates and frees storage for the sequence it controls through a stored allocator object (page 337) of class A. Such an allocator object must have the same external interface as an object of template class `allocator` (page 337). Note that the stored allocator object is *not* copied when the container object is assigned.

unordered_multimap::allocator_type

```
typedef A allocator_type;
```

The type is a synonym for the template parameter A.

unordered_multimap::begin

```
iterator begin();  
const_iterator begin() const;  
local_iterator begin(size_type n);  
const_local_iterator begin(size_type n) const;
```

The member function returns a bidirectional iterator that points at the first element of the sequence (or just beyond the end of an empty sequence).

unordered_multimap::bucket

```
size_type bucket(const key_type& k) const;
```

The member function returns the index of the bucket that contains the specified key.

unordered_multimap::bucket_count

```
size_type bucket_count() const;
```

The member function returns the number of buckets that the unordered multimap contains.

unordered_multimap::bucket_size

```
size_type bucket_size(size_type n) const;
```

The member function returns the number of elements in the nth bucket.

unordered_multimap::clear

```
void clear();
```

The member function calls `erase(begin(), end())`.

unordered_multimap::const_iterator

```
typedef T3 const_iterator;
```

The type describes an object that can serve as a constant bidirectional iterator for the controlled sequence. It is described here as a synonym for the implementation-defined type T3.

unordered_multimap::const_local_iterator

```
typedef T5 const_local_iterator;
```

The type describes an object that can serve as a constant bidirectional iterator for the controlled sequence. It is described here as a synonym for the implementation-defined type T5.

unordered_multimap::const_pointer

```
typedef typename A::const_pointer const_pointer;
```

The type describes an object that can serve as a constant pointer to an element of the controlled sequence.

unordered_multimap::const_reference

```
typedef typename A::const_reference const_reference;
```

The type describes an object that can serve as a constant reference to an element of the controlled sequence.

unordered_multimap::count

```
size_type count(const key_type& k) const;
```

The member function returns the number of elements in the multimap that have a key equivalent to k, based on the key_eq function.

unordered_multimap::difference_type

```
typedef T1 difference_type;
```

The signed integer type describes an object that can represent the difference between the addresses of any two elements in the controlled sequence. It is described here as a synonym for the implementation-defined type T1.

unordered_multimap::empty

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

The member function returns true for an empty controlled sequence.

unordered_multimap::end

```
const_iterator end() const;  
iterator end();  
local_iterator end(size_type n);  
const_local_iterator end(size_type n) const;
```

The member function returns a bidirectional iterator that points just beyond the end of the sequence.

unordered_multimap::equal_range

```
std::pair<iterator, iterator>  
    equal_range(const key_type& k);  
std::pair<const_iterator, const_iterator>  
    equal_range(const key_type& k) const;
```

The member function returns a range that contains all of the elements with the specified key. It returns make_pair(end(), end()) if no such elements exist.

unordered_multimap::erase

```
void erase(const_iterator position);  
size_type erase(const key_type& k);  
void erase(const_iterator first, const_iterator last);
```

The first member function removes the element of the controlled sequence pointed to by it. The second member function removes the elements in the range [first, last). Both return an iterator that designates the first element remaining beyond any elements removed, or end() if no such element exists.

The third member removes all the elements in the multimap. It returns the number of elements it removes.

The member functions never throw an exception.

unordered_multimap::find

```
iterator      find(const key_type& k);  
const_iterator find(const key_type& k) const;
```

The member function returns an iterator that designates the earliest element in the controlled sequence whose sort key has equivalent ordering (page 39) to key. If no such element exists, the function returns end().

unordered_multimap::get_allocator

A **get_allocator()** const;

The member function returns the stored allocator object (page 337).

unordered_multimap::hasher

typedef Hash **hasher**;

The type returns a value of type std::size_t.

unordered_multimap::hash_function

hasher hash_function() const;

The member function returns the hash function that was used to construct the multimap.

unordered_multimap::insert

```
iterator insert(const value_type& obj);  
iterator insert(const_iterator hint, const value_type& obj);  
template <class InputIterator>  
void insert(InputIterator first, InputIterator last);
```

The first member function inserts the element obj in the controlled sequence, then returns the iterator that designates the inserted element. The second member function returns insert(obj), using it as a starting place within the controlled sequence to search for the insertion point. The third member function inserts the sequence of element values, for each it in the range [first, last), by calling insert(*it).

If an exception is thrown during the insertion of a single element, the container is left unaltered and the exception is rethrown. If an exception is thrown during the insertion of multiple elements, the container is left in a stable but unspecified state and the exception is rethrown.

unordered_multimap::iterator

typedef T2 **iterator**;

The type describes an object that can serve as a bidirectional iterator for the controlled sequence. It is described here as a synonym for the implementation-defined type T2.

unordered_multimap::key_eq

key_equal **key_eq()** const;

The member function returns the key equality function that was used to create the multimap.

unordered_multimap::key_equal

typedef Pred **key_equal**;

The type returns an equivalence relation.

unordered_multimap::key_type

```
typedef Key key_type;
```

The type describes the sort key object stored in each element of the controlled sequence.

unordered_multimap::load_factor

```
float load_factor() const;
```

The member function returns the average number of elements per bucket.

unordered_multimap::local_iterator

```
typedef T4 local_iterator;
```

The type describes an object that can serve as a constant bidirectional iterator for the controlled sequence. It is described here as a synonym for the implementation-defined type T4.

unordered_multimap::mapped_type

```
typedef T mapped_type;
```

The type is a synonym for the template parameter T.

unordered_multimap::max_bucket_count

```
size_type max_bucket_count() const;
```

The member function returns the maximum number of buckets that the unordered multimap can contain.

unordered_multimap::max_load_factor

```
float max_load_factor() const;  
void max_load_factor(float z);
```

The first member function returns the maximum value that the container attempts to maintain for the load factor (the average number of elements per bucket). If the load factor increases beyond this value, the container creates more buckets.

The second member function sets the maximum load factor. If this value is less than the current load factor, the unordered multimap is rehashed.

unordered_multimap::max_size

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

The member function returns the length of the longest sequence that the object can control.

unordered_multimap::pointer

```
typedef typename A::pointer pointer;
```

The type describes an object that can serve as a pointer to an element of the controlled sequence.

unordered_multimap::reference

```
typedef typename A::reference reference;
```

The type describes an object that can serve as a reference to an element of the controlled sequence.

unordered_multimap::rehash

```
void rehash(size_type n);
```

The member function rehashes the unordered multimap, ensuring that it contains at least n buckets.

unordered_multimap::size

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

The member function returns the length of the controlled sequence.

unordered_multimap::size_type

```
typedef T0 size_type;
```

The unsigned integer type describes an object that can represent the length of any controlled sequence. It is described here as a synonym for the implementation-defined type T0.

unordered_multimap::swap

```
void swap(unordered_multimap& x);
```

The member function swaps the controlled sequences between *this and x. If `get_allocator() == x.get_allocator()`, it does so in constant time, it throws an exception only as a result of copying the stored function object of type Pred, and it invalidates no references, pointers, or iterators that designate elements in the two controlled sequences. Otherwise, it performs a number of element assignments and constructor calls proportional to the number of elements in the two controlled sequences.

unordered_multimap::unordered_multimap

```
explicit unordered_multimap(size_type n = 3,  
    const hasher& hf = hasher(),  
    const key_equal& eql = key_equal(),  
    const allocator_type& a = allocator_type();  
template <class InputIterator>  
    unordered_multimap(InputIterator f, InputIterator l,  
        size_type n = 3,  
        const hasher& hf = hasher(),  
        const key_equal& eql = key_equal(),  
        const allocator_type& a = allocator_type();  
unordered_multimap(const unordered_multimap&);  
~unordered_multimap();  
unordered_multimap& operator=(const unordered_multimap&);
```

All constructors store an allocator object (page 337) and initialize the controlled sequence. The allocator object is the argument a1, if present. For the copy constructor, it is `x.get_allocator` (page 383)(). Otherwise, it is A().

The first three constructors specify an empty initial controlled sequence. The fourth constructor specifies a copy of the sequence controlled by x. The last three constructors specify the sequence of element values [first, last).

unordered_multimap::value_type

```
typedef std::pair<const Key, T> value_type;
```

The type describes an element of the controlled sequence.

<unordered_set>

Note: To enable this header file, you must define the macro `_VACPP_TR1`.

```
namespace std {
namespace tr1 {
    template <class Value,
               class Hash = hash<Value>,
               class Pred = std::equal_to<Value>,
               class Alloc = std::allocator<Value> >
        class unordered_set;

    template <class Value,
               class Hash = hash<Value>,
               class Pred = std::equal_to<Value>,
               class Alloc = std::allocator<Value> >
        class unordered_multiset;
}
}
```

Include the STL (page 1) standard header `<unordered_set>` to define the container (page 41) template classes `unordered_set` and `unordered_multiset`, and their supporting templates.

unordered_multiset

`allocator_type` (page 388) · `begin` (page 388) · `bucket` (page 388) · `bucket_count` (page 388) · `bucket_size` (page 388) · `clear` (page 389) · `const_iterator` (page 389) · `const_local_iterator` (page 389) · `const_pointer` (page 389) · `const_reference` (page 389) · `count` (page 389) · `difference_type` (page 389) · `empty` (page 389) · `end` (page 389) · `equal_range` (page 390) · `erase` (page 390) · `find` (page 390) · `get_allocator` (page 390) · `hash_function` (page 390) · `hasher` (page 390) · `insert` (page 390) · `iterator` (page 391) · `key_eq` (page 391) · `key_equal` (page 391) · `key_type` (page 391) · `load_factor` (page 391) · `local_iterator` (page 391) · `max_bucket_count` (page 391) · `max_load_factor` (page 391) · `max_size` (page 392) · `pointer` (page 392) · `reference` (page 392) · `rehash` (page 392) · `size` (page 392) · `size_type` (page 392) · `swap` (page 392) · `unordered_multiset` (page 393) · `value_type` (page 393)

```
namespace std {
namespace tr1 {
    template <class Value,
               class Hash = hash<Value>,
               class Pred = std::equal_to<Value>,
               class A= std::allocator<Value> >
        class unordered_multiset
        {
        public:
            // types
            typedef Value          key_type;
            typedef Value          value_type;
            typedef Hash           hasher;
            typedef Pred           key_equal;
            typedef A              allocator_type;
            typedef typename A::pointer pointer;
            typedef typename A::const_pointer const_pointer;
            typedef typename A::reference reference;
            typedef typename A::const_reference const_reference;
            typedef T0             size_type;
            typedef T1             difference_type;

            typedef T2             iterator;
            typedef T3             const_iterator;
            typedef T4             local_iterator;
            typedef T5             const_local_iterator;
        };
}
```

```

// construct/destroy/copy
explicit unordered_multiset(size_type n = 3,
                             const hasher& hf = hasher(),
                             const key_equal& eql = key_equal(),
                             const allocator_type& a = allocator_type());

template <class InputIterator>
unordered_multiset(InputIterator f, InputIterator l,
                    size_type n = 3,
                    const hasher& hf = hasher(),
                    const key_equal& eql = key_equal(),
                    const allocator_type& a = allocator_type());
unordered_multiset(const unordered_multiset&);
~unordered_multiset();
unordered_multiset& operator=(const unordered_multiset&);
A get_allocator() const;

// size and capacity
bool empty() const;
size_type size() const;
size_type max_size() const;

// iterators
iterator begin();
const_iterator begin() const;
iterator end();
const_iterator end() const;

// modifiers
iterator insert(const value_type& obj);
iterator insert(const_iterator hint, const value_type& obj);
template <class InputIterator>
void insert(InputIterator first, InputIterator last);

void erase(const_iterator position);
size_type erase(const key_type& k);
void erase(const_iterator first, const_iterator last);
void clear();

void swap(unordered_multiset&);

// observers
hasher hash_function() const;
key_equal key_eq() const;

// lookup
iterator find(const key_type& k);
const_iterator find(const key_type& k) const;
size_type count(const key_type& k) const;
std::pair<iterator, iterator>
equal_range(const key_type& k);
std::pair<const_iterator, const_iterator>
equal_range(const key_type& k) const;

// bucket interface
size_type bucket_count() const;
size_type max_bucket_count() const;
size_type bucket_size(size_type n) const;
size_type bucket(const key_type& k) const;
local_iterator begin(size_type n);
const_local_iterator begin(size_type n) const;
local_iterator end(size_type n);
const_local_iterator end(size_type n) const;

// hash policy
float load_factor() const;

```

```

float max_load_factor() const;
void max_load_factor(float z);
void rehash(size_type n);
};

template <class Value, class Hash, class Pred, class Alloc>
void swap(const unordered_multiset<Value, Hash, Pred, Alloc>& x,
         const unordered_multiset<Value, Hash, Pred, Alloc>& y);
}

```

The template class describes an object that controls a varying-length sequence of elements of type `const Key`. The sequence is unordered. Each element serves as both a **sort key** and a **value**. If you have an optimal hash function, the number of operations performed during lookup, insertion, and removal of an arbitrary element does not depend on the number of elements in the sequence. Moreover, inserting an element invalidates no iterators, and removing an element invalidates only those iterators which point at the removed element.

The object allocates and frees storage for the sequence it controls through a stored allocator object (page 337) of class `A`. Such an allocator object must have the same external interface as an object of template class `allocator` (page 337). Note that the stored allocator object is *not* copied when the container object is assigned.

unordered_multiset::allocator_type

```
typedef A allocator_type;
```

The type is a synonym for the template parameter `A`.

unordered_multiset::begin

```

iterator      begin();
const_iterator begin() const;
local_iterator begin(size_type n);
const_local_iterator begin(size_type n) const;

```

The member function returns a bidirectional iterator that points at the first element of the sequence (or just beyond the end of an empty sequence).

unordered_multiset::bucket

```
size_type bucket(const key_type& k) const;
```

The member function returns the index of the bucket that contains the specified key.

unordered_multiset::bucket_count

```
size_type bucket_count() const;
```

The member function returns the number of buckets that the unordered multiset contains.

unordered_multiset::bucket_size

```
size_type bucket_size(size_type n) const;
```

The member function returns the number of elements in the `n`th bucket.

unordered_multiset::clear

```
void clear();
```

The member function calls `erase(begin(), end())`.

unordered_multiset::const_iterator

```
typedef T3 const_iterator;
```

The type describes an object that can serve as a constant bidirectional iterator for the controlled sequence. It is described here as a synonym for the implementation-defined type `T3`.

unordered_multiset::const_local_iterator

```
typedef T5 const_local_iterator;
```

The type describes an object that can serve as a constant bidirectional iterator for the controlled sequence. It is described here as a synonym for the implementation-defined type `T5`.

unordered_multiset::const_pointer

```
typedef typename A::const_pointer const_pointer;
```

The type describes an object that can serve as a constant pointer to an element of the controlled sequence.

unordered_multiset::const_reference

```
typedef typename A::const_reference const_reference;
```

The type describes an object that can serve as a constant reference to an element of the controlled sequence.

unordered_multiset::count

```
size_type count(const key_type& k) const;
```

The member function returns the number of elements in the multiset that have a key equivalent to `k`, based on the `key_eq` function.

unordered_multiset::difference_type

```
typedef T1 difference_type;
```

The signed integer type describes an object that can represent the difference between the addresses of any two elements in the controlled sequence. It is described here as a synonym for the implementation-defined type `T1`.

unordered_multiset::empty

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

The member function returns true for an empty controlled sequence.

unordered_multiset::end

```
iterator end();  
const_iterator end() const;  
local_iterator end(size_type n);  
const_local_iterator end(size_type n) const;
```

The member function returns a bidirectional iterator that points just beyond the end of the sequence.

unordered_multiset::equal_range

```
std::pair<iterator, iterator>  
    equal_range(const key_type& k);  
std::pair<const_iterator, const_iterator>  
    equal_range(const key_type& k) const;
```

The member function returns a range that contains all of the elements with the specified key. It returns `make_pair(end(), end())` if no such elements exist.

unordered_multiset::erase

```
void erase(const_iterator position);  
size_type erase(const key_type& k);  
void erase(const_iterator first, const_iterator last);
```

The first member function removes the element of the controlled sequence pointed to by it. The second member function removes the elements in the range `[first, last)`. Both return an iterator that designates the first element remaining beyond any elements removed, or `end()` if no such element exists.

The third member removes all the elements in the multiset. It returns the number of elements it removes.

The member functions never throw an exception.

unordered_multiset::find

```
iterator      find(const key_type& k);  
const_iterator find(const key_type& k) const;
```

The member function returns an iterator that designates the earliest element in the controlled sequence whose sort key has equivalent ordering (page 39) to key. If no such element exists, the function returns `end()`.

unordered_multiset::get_allocator

```
A get_allocator() const;
```

The member function returns the stored allocator object (page 337).

unordered_multiset::hasher

```
typedef Hash hasher;
```

The type returns a value of type `std::size_t`.

unordered_multiset::hash_function

```
hasher hash_function() const;
```

The member function returns the hash function that was used to construct the multiset.

unordered_multiset::insert

```
iterator insert(const value_type& obj);  
iterator insert(const_iterator hint, const value_type& obj);  
template <class InputIterator>  
    void insert(InputIterator first, InputIterator last);
```

The first member function inserts the element `obj` in the controlled sequence, then returns the iterator that designates the inserted element. The second member function returns `insert(obj)`, using it as a starting place within the controlled

sequence to search for the insertion point. The third member function inserts the sequence of element values, for each it in the range [first, last), by calling insert(*it).

If an exception is thrown during the insertion of a single element, the container is left unaltered and the exception is rethrown. If an exception is thrown during the insertion of multiple elements, the container is left in a stable but unspecified state and the exception is rethrown.

unordered_multiset::iterator

```
typedef T2 iterator;
```

The type describes an object that can serve as a bidirectional iterator for the controlled sequence. It is described here as a synonym for the implementation-defined type T2.

unordered_multiset::key_eq

```
key_equal key_eq() const;
```

The member function returns the key equality function that was used to create the multiset.

unordered_multiset::key_equal

```
typedef Pred key_equal;
```

The type returns an equivalence relation.

unordered_multiset::key_type

```
typedef Value key_type;
```

The type describes the sort key object which constitutes each element of the controlled sequence.

unordered_multiset::load_factor

```
float load_factor() const;
```

The member function returns the average number of elements per bucket.

unordered_multiset::local_iterator

```
typedef T4 local_iterator;
```

The type describes an object that can serve as a constant bidirectional iterator for the controlled sequence. It is described here as a synonym for the implementation-defined type T4.

unordered_multiset::max_bucket_count

```
size_type max_bucket_count() const;
```

The member function returns the maximum number of buckets that the unordered multiset can contain.

unordered_multiset::max_load_factor

```
float max_load_factor() const;  
void max_load_factor(float z);
```

The first member function returns the maximum value that the container attempts to maintain for the load factor (the average number of elements per bucket). If the load factor increases beyond this value, the container creates more buckets.

The second member function sets the maximum load factor. If this value is less than the current load factor, the unordered multiset is rehashed.

unordered_multiset::max_size

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

The member function returns the length of the longest sequence that the object can control.

unordered_multiset::pointer

```
typedef typename A::pointer pointer;
```

The type describes an object that can serve as a pointer to an element of the controlled sequence.

unordered_multiset::reference

```
typedef typename A::reference reference;
```

The type describes an object that can serve as a reference to an element of the controlled sequence.

unordered_multiset::rehash

```
void rehash(size_type n);
```

The member function rehashes the unordered multiset, ensuring that it contains at least n buckets.

unordered_multiset::size

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

The member function returns the length of the controlled sequence.

unordered_multiset::size_type

```
typedef T0 size_type;
```

The unsigned integer type describes an object that can represent the length of any controlled sequence. It is described here as a synonym for the implementation-defined type T0.

unordered_multiset::swap

```
void swap(unordered_multiset& x);
```

The member function swaps the controlled sequences between *this and x. If `get_allocator() == x.get_allocator()`, it does so in constant time, it throws an exception only as a result of copying the stored function object of type Pred, and it invalidates no references, pointers, or iterators that designate elements in the two controlled sequences. Otherwise, it performs a number of element assignments and constructor calls proportional to the number of elements in the two controlled sequences.

unordered_multiset::unordered_multiset

```
explicit unordered_multiset(size_type n = 3,  
    const hasher& hf = hasher(),  
    const key_equal& eql = key_equal(),  
    const allocator_type& a = allocator_type();  
template <class InputIterator>  
    unordered_multiset(InputIterator f, InputIterator l,  
        size_type n = 3,  
        const hasher& hf = hasher(),  
        const key_equal& eql = key_equal(),  
        const allocator_type& a = allocator_type();  
unordered_multiset(const unordered_multiset&);  
~unordered_multiset();  
unordered_multiset& operator=(const unordered_multiset&);
```

All constructors store an allocator object (page 337) and initialize the controlled sequence. The allocator object is the argument `a`, if present. For the copy constructor, it is `x.get_allocator()`. Otherwise, it is `A()`.

The first three constructors specify an empty initial controlled sequence. The fourth constructor specifies a copy of the sequence controlled by `x`. The last three constructors specify the sequence of element values [`first`, `last`).

unordered_multiset::value_type

```
typedef Value value_type;
```

The type describes an element of the controlled sequence.

unordered_set

`allocator_type` (page 395) · `begin` (page 395) · `bucket` (page 395) · `bucket_count` (page 395) · `bucket_size` (page 396) · `clear` (page 396) · `const_iterator` (page 396) · `const_local_iterator` (page 396) · `const_pointer` (page 396) · `const_reference` (page 396) · `count` (page 396) · `difference_type` (page 396) · `empty` (page 396) · `end` (page 396) · `equal_range` (page 397) · `erase` (page 397) · `find` (page 397) · `get_allocator` (page 397) · `hash_function` (page 397) · `hasher` (page 397) · `insert` (page 397) · `iterator` (page 398) · `key_eq` (page 398) · `key_equal` (page 398) · `key_type` (page 398) · `load_factor` (page 398) · `local_iterator` (page 398) · `max_bucket_count` (page 398) · `max_load_factor` (page 399) · `max_size` (page 399) · `pointer` (page 399) · `reference` (page 399) · `rehash` (page 399) · `size` (page 399) · `size_type` (page 399) · `swap` (page 399) · `unordered_set` (page 400) · `value_type` (page 400)

```
namespace std {  
namespace tr1 {  
    template <class Value,  
        class Hash = hash<Value>,  
        class Pred = std::equal_to<Value>,  
        class A= std::allocator<Value> >  
class unordered_set  
{  
public:  
    // types  
    typedef Value key_type;  
    typedef Value value_type;  
    typedef Hash hasher;  
    typedef Pred key_equal;  
    typedef A allocator_type;  
    typedef typename A::pointer pointer;  
    typedef typename A::const_pointer const_pointer;  
    typedef typename A::reference reference;  
    typedef typename A::const_reference const_reference;  
    typedef T0 size_type;  
    typedef T1 difference_type;
```

```

typedef T2                iterator;
typedef T3                const_iterator;
typedef T4                local_iterator;
typedef T5                const_local_iterator;

// construct/destroy/copy
explicit unordered_set(size_type n = 3,
                      const hasher& hf = hasher(),
                      const key_equal& eql = key_equal(),
                      const allocator_type& a = allocator_type());
template <class InputIterator>
    unordered_set(InputIterator f, InputIterator l,
                  size_type n = 3,
                  const hasher& hf = hasher(),
                  const key_equal& eql = key_equal(),
                  const allocator_type& a = allocator_type());
unordered_set(const unordered_set&);
~unordered_set();
unordered_set& operator=(const unordered_set&);
A get_allocator() const;

// size and capacity
bool empty() const;
size_type size() const;
size_type max_size() const;

// iterators
iterator      begin();
const_iterator begin() const;
iterator      end();
const_iterator end() const;

// modifiers
std::pair<iterator, bool> insert(const value_type& obj);
iterator insert(const iterator hint, const value_type& obj);
template <class InputIterator>
    void insert(InputIterator first, InputIterator last);

void erase(const_iterator position);
size_type erase(const key_type& k);
void erase(const_iterator first, const_iterator last);
void clear();

void swap(unordered_set&);

// observers
hasher hash_function() const;
key_equal key_eq() const;

// lookup
iterator      find(const key_type& k);
const_iterator find(const key_type& k) const;
size_type count(const key_type& k) const;
std::pair<iterator, iterator>
    equal_range(const key_type& k);
std::pair<const_iterator, const_iterator>
    equal_range(const key_type& k) const;

// bucket interface
size_type bucket_count() const;
size_type max_bucket_count() const;
size_type bucket_size(size_type n) const;
size_type bucket(const key_type& k) const;
local_iterator begin(size_type n);
const_local_iterator begin(size_type n) const;

```

```

        local_iterator end(size_type n);
        const_local_iterator end(size_type n) const;

        // hash policy
        float load_factor() const;
        float max_load_factor() const;
        void max_load_factor(float z);
        void rehash(size_type n);
    };

    template <class Value, class Hash, class Pred, class Alloc>
        void swap(const unordered_set<Value, Hash, Pred, Alloc>& x,
                  const unordered_set<Value, Hash, Pred, Alloc>& y);

}

```

The template class describes an object that controls a varying-length sequence of elements of type `const Key`. The sequence is unordered. Each element serves as both a **sort key** and a **value**. If you have an optimal hash function, the number of operations performed during lookup, insertion, and removal of an arbitrary element does not depend on the number of elements in the sequence. Moreover, inserting an element invalidates no iterators, and removing an element invalidates only those iterators which point at the removed element.

The object allocates and frees storage for the sequence it controls through a stored allocator object (page 337) of class `A`. Such an allocator object must have the same external interface as an object of template class `allocator` (page 337). Note that the stored allocator object is *not* copied when the container object is assigned.

unordered_set::allocator_type

```
typedef A allocator_type;
```

The type is a synonym for the template parameter `A`.

unordered_set::begin

```

iterator      begin();
const_iterator begin() const;

local_iterator begin(size_type n);
const_local_iterator begin(size_type n) const;

```

The member function returns a bidirectional iterator that points at the first element of the sequence (or just beyond the end of an empty sequence).

unordered_set::bucket

```
size_type bucket(const key_type& k) const;
```

The member function returns the index of the bucket that contains the specified key.

unordered_set::bucket_count

```
size_type bucket_count() const;
```

The member function returns the number of buckets that the unordered set contains.

unordered_set::bucket_size

```
size_type bucket_size(size_type n) const;
```

The member function returns the number of elements in the nth bucket.

unordered_set::clear

```
void clear();
```

The member function calls `erase(begin(), end())`.

unordered_set::const_iterator

```
typedef T3 const_iterator;
```

The type describes an object that can serve as a constant bidirectional iterator for the controlled sequence. It is described here as a synonym for the implementation-defined type T3.

unordered_set::const_local_iterator

```
typedef T5 const_local_iterator;
```

The type describes an object that can serve as a constant bidirectional iterator for the controlled sequence. It is described here as a synonym for the implementation-defined type T5.

unordered_set::const_pointer

```
typedef typename A::const_pointer const_pointer;
```

The type describes an object that can serve as a constant pointer to an element of the controlled sequence.

unordered_set::const_reference

```
typedef typename A::const_reference const_reference;
```

The type describes an object that can serve as a constant reference to an element of the controlled sequence.

unordered_set::count

```
size_type count(const key_type& k) const;
```

The member function returns the number of elements in the set that have a key equivalent to k, based on the `key_eq` function.

unordered_set::difference_type

```
typedef T1 difference_type;
```

The signed integer type describes an object that can represent the difference between the addresses of any two elements in the controlled sequence. It is described here as a synonym for the implementation-defined type T1.

unordered_set::empty

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

The member function returns true for an empty controlled sequence.

unordered_set::end

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

```
local_iterator end(size_type n);
const_local_iterator end(size_type n) const;
```

The member function returns a bidirectional iterator that points just beyond the end of the sequence.

unordered_set::equal_range

```
std::pair<iterator, iterator>
    equal_range(const key_type& k);
std::pair<const_iterator, const_iterator>
    equal_range(const key_type& k) const;
```

The member function returns a range that contains all of the elements with the specified key. It returns `make_pair(end(), end())` if no such elements exist.

unordered_set::erase

```
void erase(const_iterator position);
size_type erase(const key_type& k);
void erase(const_iterator first, const_iterator last);
```

The first member function removes the element of the controlled sequence pointed to by it. The second member function removes the elements in the range `[first, last)`. Both return an iterator that designates the first element remaining beyond any elements removed, or `end()` if no such element exists.

The third member removes all the elements in the set. It returns the number of elements it removes.

The member functions never throw an exception.

unordered_set::find

```
iterator find(const key_type& k);
const_iterator find(const key_type& k) const;
```

The member function returns an iterator that designates the earliest element in the controlled sequence whose sort key has equivalent ordering (page 39) to key. If no such element exists, the function returns `end()`.

unordered_set::get_allocator

```
A get_allocator() const;
```

The member function returns the stored allocator object.

unordered_set::hasher

```
typedef Hash hasher;
```

The type returns a value of type `std::size_t`.

unordered_set::hash_function

```
hasher hash_function() const;
```

The member function returns the hash function that was used to construct the set.

unordered_set::insert

```
std::pair<iterator, bool> insert(const value_type& obj);
iterator insert(const_iterator hint, const value_type& obj);
template <class InputIterator>
    void insert(InputIterator first, InputIterator last);
```

The first member function determines whether an element *y* exists in the sequence whose key has equivalent ordering (page 39) to that of *obj*. If not, it creates such an element *y* and initializes it with *obj*. The function then determines the iterator *it* that designates *y*. If an insertion occurred, the function returns `pair(it, true)`. Otherwise, it returns `pair(it, false)`.

The second member function returns `insert(obj)`, using *it* as a starting place within the controlled sequence to search for the insertion point. The third member function inserts the sequence of element values, for each *it* in the range `[first, last)`, by calling `insert(*it)`.

If an exception is thrown during the insertion of a single element, the container is left unaltered and the exception is rethrown. If an exception is thrown during the insertion of multiple elements, the container is left in a stable but unspecified state and the exception is rethrown.

unordered_set::iterator

```
typedef T2 iterator;
```

The type describes an object that can serve as a bidirectional iterator for the controlled sequence. It is described here as a synonym for the implementation-defined type `T2`.

unordered_set::key_eq

```
key_equal key_eq() const;
```

The member function returns the key equality function that was used to create the map.

unordered_set::key_equal

```
typedef Pred key_equal;
```

The type returns an equivalence relation.

unordered_set::key_type

```
typedef Value key_type;
```

The type describes the sort key object which constitutes each element of the controlled sequence.

unordered_set::load_factor

```
float load_factor() const;
```

The member function returns the average number of elements per bucket.

unordered_set::local_iterator

```
typedef T4 local_iterator;
```

The type describes an object that can serve as a constant bidirectional iterator for the controlled sequence. It is described here as a synonym for the implementation-defined type `T4`.

unordered_set::max_bucket_count

```
size_type max_bucket_count() const;
```

The member function returns the maximum number of buckets that the unordered set can contain.

unordered_set::max_load_factor

```
float max_load_factor() const;  
void max_load_factor(float z);
```

The first member function returns the maximum value that the container attempts to maintain for the load factor (the average number of elements per bucket). If the load factor increases beyond this value, the container creates more buckets.

The second member function sets the maximum load factor. If this value is less than the current load factor, the unordered set is rehashed.

unordered_set::max_size

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

The member function returns the length of the longest sequence that the object can control.

unordered_set::pointer

```
typedef typename A::const_pointer pointer;
```

The type describes an object that can serve as a pointer to an element of the controlled sequence.

unordered_set::reference

```
typedef typename A::const_reference reference;
```

The type describes an object that can serve as a reference to an element of the controlled sequence.

unordered_set::rehash

```
void rehash(size_type n);
```

The member function rehashes the unordered set, ensuring that it contains at least *n* buckets.

unordered_set::size

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

The member function returns the length of the controlled sequence.

unordered_set::size_type

```
typedef T0 size_type;
```

The unsigned integer type describes an object that can represent the length of any controlled sequence. It is described here as a synonym for the implementation-defined type *T0*.

unordered_set::swap

```
void swap(unordered_set& x);
```

The member function swaps the controlled sequences between **this* and *x*. If *get_allocator() == x.get_allocator()*, it does so in constant time, it throws an exception only as a result of copying the stored function object of type *Pred*, and it invalidates no references, pointers, or iterators that designate elements in the two controlled sequences. Otherwise, it performs a number of element assignments and constructor calls proportional to the number of elements in the two controlled sequences.

unordered_set::unordered_set

```
explicit unordered_set(size_type n = 3,
    const hasher& hf = hasher(),
    const key_equal& eql = key_equal(),
    const allocator_type& a = allocator_type());
template <class InputIterator>
    unordered_set(InputIterator f, InputIterator l,
        size_type n = 3,
        const hasher& hf = hasher(),
        const key_equal& eql = key_equal(),
        const allocator_type& a = allocator_type());
unordered_set(const unordered_set&);
~unordered_set();
unordered_set& operator=(const unordered_set&);
```

All constructors store an allocator object (page 337) and initialize the controlled sequence. The allocator object is the argument `a`, if present. For the copy constructor, it is `x.get_allocator()`. Otherwise, it is `A()`.

The first three constructors specify an empty initial controlled sequence. The fourth constructor specifies a copy of the sequence controlled by `x`. The last three constructors specify the sequence of element values [`first`, `last`).

unordered_set::value_type

```
typedef Value value_type;
```

The type describes an element of the controlled sequence.

<utility>

```
namespace std {
    template<class T, class U>
        struct pair (page 402);

        // TEMPLATE FUNCTIONS
    template<class T, class U>
        pair<T, U> make_pair(T x, U y);
    template<class T, class U>
        bool operator==(const pair<T, U>& x,
            const pair<T, U>& y);
    template<class T, class U>
        bool operator!=(const pair<T, U>& x,
            const pair<T, U>& y);
    template<class T, class U>
        bool operator<(const pair<T, U>& x,
            const pair<T, U>& y);
    template<class T, class U>
        bool operator>(const pair<T, U>& x,
            const pair<T, U>& y);
    template<class T, class U>
        bool operator<=(const pair<T, U>& x,
            const pair<T, U>& y);
    template<class T, class U>
        bool operator>=(const pair<T, U>& x,
            const pair<T, U>& y);

    namespace rel_ops {
        template<class T>
```



```

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

```

Include the STL (page 1) standard header **<utility>** to define several templates of general use throughout the Standard Template Library.

Four template operators — **operator!=**, **operator<=**, **operator>**, and **operator>=** — define a **total ordering** on pairs of operands of the same type, given definitions of **operator==** and **operator<**.

If an implementation (page 3) supports namespaces, these template operators are defined in the **rel_ops** namespace, nested within the **std** namespace. If you wish to make use of these template operators, write the declaration:

```
using namespace std::rel_ops;
```

which promotes the template operators into the current namespace.

make_pair

```

template<class T, class U>
    pair<T, U> make_pair(T x, U y);

```

The template function returns **pair<T, U>(x, y)**.

operator!=

```

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

```

The template function returns **!(x == y)**.

operator==

```

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

```

The template function returns **x.first == y.first && x.second == y.second**.

operator<

```

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

```

The template function returns **x.first < y.first || !(y.first < x.first && x.second < y.second)**.

operator<=

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

The template function returns $!(y < x)$.

operator>

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

The template function returns $y < x$.

operator>=

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

The template function returns $!(x < y)$.

pair

```
template<class T, class U>
    struct pair {
        typedef T first_type;
        typedef U second_type
        T first;
        U second;
        pair();
        pair(const T& x, const U& y);
        template<class V, class W>
            pair(const pair<V, W>& pr);
    };

```

The template class stores a pair of objects, **first**, of type *T*, and **second**, of type *U*. The type definition **first_type**, is the same as the template parameter *T*, while **second_type**, is the same as the template parameter *U*.

The first (default) constructor initializes **first** to *T*() and **second** to *U*(). The second constructor initializes **first** to *x* and **second** to *y*. The third (template) constructor initializes **first** to *pr.first* and **second** to *pr.second*. *T* and *U* each need supply only a default constructor, single-argument constructor, and a destructor.

Portions derived from work copyright © 1994 by Hewlett-Packard Company. All rights reserved.

<vector>

```
namespace std {
template<class T, class A>
    class vector;
template<class A>
    class vector<bool>;

    // TEMPLATE FUNCTIONS
template<class T, class A>
    bool operator==(
        const vector<T, A>& lhs,
        const vector<T, A>& rhs);
template<class T, class A>
    bool operator!=(
        const vector<T, A>& lhs,
        const vector<T, A>& rhs);
template<class T, class A>
    bool operator<(
        const vector<T, A>& lhs,
        const vector<T, A>& rhs);
template<class T, class A>
    bool operator>(
        const vector<T, A>& lhs,
        const vector<T, A>& rhs);
template<class T, class A>
    bool operator<=(
        const vector<T, A>& lhs,
        const vector<T, A>& rhs);
template<class T, class A>
    bool operator>=(
        const vector<T, A>& lhs,
        const vector<T, A>& rhs);
template<class T, class A>
    void swap(
        vector<T, A>& lhs,
        vector<T, A>& rhs);
};
```

Include the STL (page 1) standard header **<vector>** to define the container (page 41) template class **vector** and several supporting templates.

operator!=

```
template<class T, class A>
    bool operator!=(
        const vector <T, A>& lhs,
        const vector <T, A>& rhs);
```

The template function returns `!(lhs == rhs)`.

operator==

```
template<class T, class A>
    bool operator==(
        const vector <T, A>& lhs,
        const vector <T, A>& rhs);
```

The template function overloads `operator==` to compare two objects of template class **vector** (page 404). The function returns `lhs.size (page 410)() == rhs.size() && equal (page 255)(lhs. begin (page 406)(), lhs. end (page 407)(), rhs.begin())`.

operator<

```
template<class T, class A>
bool operator<(
    const vector<T, A>& lhs,
    const vector<T, A>& rhs);
```

The template function overloads operator< to compare two objects of template class vector. The function returns `lexicographical_compare(lhs.begin(), lhs.end(), rhs.begin(), rhs.end())`.

operator<=

```
template<class T, class A>
bool operator<=(
    const vector<T, A>& lhs,
    const vector<T, A>& rhs);
```

The template function returns `!(rhs < lhs)`.

operator>

```
template<class T, class A>
bool operator>(
    const vector<T, A>& lhs,
    const vector<T, A>& rhs);
```

The template function returns `rhs < lhs`.

operator>=

```
template<class T, class A>
bool operator>=(
    const vector<T, A>& lhs,
    const vector<T, A>& rhs);
```

The template function returns `!(lhs < rhs)`.

swap

```
template<class T, class A>
void swap(
    vector<T, A>& lhs,
    vector<T, A>& rhs);
```

The template function executes `lhs.swap(rhs)`.

vector

`allocator_type` (page 406) · `assign` (page 406) · `at` (page 406) · `back` (page 406) · `begin` (page 406) · `capacity` (page 406) · `clear` (page 407) · `const_iterator` (page 407) · `const_pointer` (page 407) · `const_reference` (page 407) · `const_reverse_iterator` (page 407) · `difference_type` (page 407) · `empty` (page 407) · `end` (page 407) · `erase` (page 407) · `front` (page 408) · `get_allocator` (page 408) · `insert` (page 408) · `iterator` (page 409) · `max_size` (page 409) · `operator[]` (page 409) · `pointer` (page 409) · `pop_back` (page 409) · `push_back` (page 409) · `rbegin` (page 409) · `reference` (page 409) · `rend` (page 410) · `reserve` (page 410) · `resize` (page 410) · `reverse_iterator` (page 410) · `size` (page 410) · `size_type` (page 410) · `swap` (page 410) · `value_type` (page 411) · `vector` (page 411)

```
template<class T, class A = allocator<T> >
class vector {
public:
```

```

typedef A allocator_type;
typedef typename A::pointer pointer;
typedef typename A::const_pointer
    const_pointer;
typedef typename A::reference reference;
typedef typename A::const_reference
    const_reference;
typedef typename A::value_type value_type;
typedef T0 iterator;
typedef T1 const_iterator;
typedef T2 size_type;
typedef T3 difference_type;
typedef reverse_iterator<const_iterator>
    const_reverse_iterator;
typedef reverse_iterator<iterator>
    reverse_iterator;
vector();
explicit vector(const A& al);
explicit vector(size_type n);
vector(size_type n, const T& x);
vector(size_type n, const T& x,
    const A& al);
vector(const vector& x);
template<class InIt>
    vector(InIt first, InIt last);
template<class InIt>
    vector(InIt first, InIt last,
    const A& al);
void reserve(size_type n);
size_type capacity() const;
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;
void resize(size_type n);
void resize(size_type n, T x);
size_type size() const;
size_type max_size() const;
bool empty() const;
A get_allocator() const;
reference at(size_type pos);
const_reference at(size_type pos) const;
reference operator[](size_type pos);
const_reference operator[](size_type pos);
reference front();
const_reference front() const;
reference back();
const_reference back() const;
void push_back(const T& x);
void pop_back();
template<class InIt>
    void assign(InIt first, InIt last);
void assign(size_type n, const T& x);
iterator insert(iterator it, const T& x);
void insert(iterator it, size_type n, const T& x);
template<class InIt>
    void insert(iterator it, InIt first, InIt last);
iterator erase(iterator it);
iterator erase(iterator first, iterator last);
void clear();
void swap(vector& x);
};

```

The template class describes an object that controls a varying-length sequence of elements of type T. The sequence is stored as an array of T.

The object allocates and frees storage for the sequence it controls through a stored allocator object (page 337) of class A. Such an allocator object must have the same external interface as an object of template class allocator (page 337). Note that the stored allocator object is *not* copied when the container object is assigned.

Vector reallocation occurs when a member function must grow the controlled sequence beyond its current storage capacity (page 406). Other insertions and erasures may alter various storage addresses within the sequence. In all such cases, iterators or references that point at altered portions of the controlled sequence become **invalid**.

vector::allocator_type

```
typedef A allocator_type;
```

The type is a synonym for the template parameter A.

vector::assign

```
template<class InIt>
    void assign(InIt first, InIt last);
void assign(size_type n, const T& x);
```

If InIt is an integer type, the first member function behaves the same as assign((size_type)first, (T)last). Otherwise, the first member function replaces the sequence controlled by *this with the sequence [first, last), which must *not* overlap the initial controlled sequence. The second member function replaces the sequence controlled by *this with a repetition of n elements of value x.

vector::at

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

The member function returns a reference to the element of the controlled sequence at position pos. If that position is invalid, the function throws an object of class out_of_range.

vector::back

```
reference back();
const_reference back() const;
```

The member function returns a reference to the last element of the controlled sequence, which must be non-empty.

vector::begin

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

The member function returns a random-access iterator that points at the first element of the sequence (or just beyond the end of an empty sequence).

vector::capacity

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

The member function returns the storage currently allocated to hold the controlled sequence, a value at least as large as size().

vector::clear

```
void clear();
```

The member function calls `erase(begin(), end())`.

vector::const_iterator

```
typedef T1 const_iterator;
```

The type describes an object that can serve as a constant random-access iterator for the controlled sequence. It is described here as a synonym for the implementation-defined type `T1`.

vector::const_pointer

```
typedef typename A::const_pointer  
const_pointer;
```

The type describes an object that can serve as a constant pointer to an element of the controlled sequence.

vector::const_reference

```
typedef typename A::const_reference  
const_reference;
```

The type describes an object that can serve as a constant reference to an element of the controlled sequence.

vector::const_reverse_iterator

```
typedef reverse_iterator<const_iterator>  
const_reverse_iterator;
```

The type describes an object that can serve as a constant reverse iterator for the controlled sequence.

vector::difference_type

```
typedef T3 difference_type;
```

The signed integer type describes an object that can represent the difference between the addresses of any two elements in the controlled sequence. It is described here as a synonym for the implementation-defined type `T3`.

vector::empty

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

The member function returns true for an empty controlled sequence.

vector::end

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

The member function returns a random-access iterator that points just beyond the end of the sequence.

vector::erase

```
iterator erase(iterator it);  
iterator erase(iterator first, iterator last);
```

The first member function removes the element of the controlled sequence pointed to by `it`. The second member function removes the elements of the controlled sequence in the range `[first, last)`. Both return an iterator that designates the first element remaining beyond any elements removed, or `end()` if no such element exists.

Erasing `N` elements causes `N` destructor calls and an assignment for each of the elements between the insertion point and the end of the sequence. No reallocation (page 406) occurs, so iterators and references become invalid (page 406) only from the first element erased through the end of the sequence.

The member functions never throw an exception.

vector::front

```
reference front();  
const_reference front() const;
```

The member function returns a reference to the first element of the controlled sequence, which must be non-empty.

vector::get_allocator

```
A get_allocator() const;
```

The member function returns the stored allocator object (page 337).

vector::insert

```
iterator insert(iterator it, const T& x);  
void insert(iterator it, size_type n, const T& x);  
template<class InIt>  
    void insert(iterator it, InIt first, InIt last);
```

Each of the member functions inserts, before the element pointed to by `it` in the controlled sequence, a sequence specified by the remaining operands. The first member function inserts a single element with value `x` and returns an iterator that points to the newly inserted element. The second member function inserts a repetition of `n` elements of value `x`.

If `InIt` is an integer type, the last member function behaves the same as `insert(it, (size_type)first, (T)last)`. Otherwise, the last member function inserts the sequence `[first, last)`, which must *not* overlap the initial controlled sequence.

When inserting a single element, the number of element copies is linear in the number of elements between the insertion point and the end of the sequence. When inserting a single element at the end of the sequence, the amortized number of element copies is constant. When inserting `N` elements, the number of element copies is linear in `N` plus the number of elements between the insertion point and the end of the sequence — except when the template member is specialized for `InIt` an input iterator, which behaves like `N` single insertions.

If reallocation (page 406) occurs, the size of the controlled sequence at least doubles, and all iterators and references become invalid (page 406). If no reallocation occurs, iterators become invalid only from the point of insertion through the end of the sequence.

If an exception is thrown during the insertion of a single element, the container is left unaltered and the exception is rethrown. If an exception is thrown during the

insertion of multiple elements, and the exception is not thrown while copying an element, the container is left unaltered and the exception is rethrown.

vector::iterator

```
typedef T0 iterator;
```

The type describes an object that can serve as a random-access iterator for the controlled sequence. It is described here as a synonym for the implementation-defined type T0.

vector::max_size

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

The member function returns the length of the longest sequence that the object can control.

vector::operator[]

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

The member function returns a reference to the element of the controlled sequence at position pos. If that position is invalid, the behavior is undefined.

vector::pointer

```
typedef typename A::pointer pointer;
```

The type describes an object that can serve as a pointer to an element of the controlled sequence.

vector::pop_back

```
void pop_back();
```

The member function removes the last element of the controlled sequence, which must be non-empty.

The member function never throws an exception.

vector::push_back

```
void push_back(const T& x);
```

The member function inserts an element with value x at the end of the controlled sequence.

If an exception is thrown, the container is left unaltered and the exception is rethrown.

vector::rbegin

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

The member function returns a reverse iterator that points just beyond the end of the controlled sequence. Hence, it designates the beginning of the reverse sequence.

vector::reference

```
typedef typename A::reference reference;
```

The type describes an object that can serve as a reference to an element of the controlled sequence.

vector::rend

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

The member function returns a reverse iterator that points at the first element of the sequence (or just beyond the end of an empty sequence). Hence, it designates the end of the reverse sequence.

vector::reserve

```
void reserve(size_type n);
```

If *n* is greater than `max_size()`, the member function reports a **length error** by throwing an object of class `length_error`. Otherwise, it ensures that `capacity()` henceforth returns at least *n*.

vector::resize

```
void resize(size_type n);  
void resize(size_type n, T x);
```

The member functions both ensure that `size()` henceforth returns *n*. If it must make the controlled sequence longer, the first member function appends elements with value `T()`, while the second member function appends elements with value *x*. To make the controlled sequence shorter, both member functions call `erase(begin() + n, end())`.

vector::reverse_iterator

```
typedef reverse_iterator<iterator>  
    reverse_iterator;
```

The type describes an object that can serve as a reverse iterator for the controlled sequence.

vector::size

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

The member function returns the length of the controlled sequence.

vector::size_type

```
typedef T2 size_type;
```

The unsigned integer type describes an object that can represent the length of any controlled sequence. It is described here as a synonym for the implementation-defined type *T2*.

vector::swap

```
void swap(vector& x);
```

The member function swaps the controlled sequences between `*this` and *x*. If `get_allocator() == x.get_allocator()`, it does so in constant time, it throws no exceptions, and it invalidates no references, pointers, or iterators that designate elements in the two controlled sequences. Otherwise, it performs a number of element assignments and constructor calls proportional to the number of elements in the two controlled sequences.

vector::value_type

```
typedef typename A::value_type value_type;
```

The type is a synonym for the template parameter T.

vector::vector

```
vector();  
explicit vector(const A& a1);  
explicit vector(size_type n);  
vector(size_type n, const T& x);  
vector(size_type n, const T& x, const A& a1);  
vector(const vector& x);  
template<class InIt>  
    vector(InIt first, InIt last);  
template<class InIt>  
    vector(InIt first, InIt last, const A& a1);
```

All constructors store an allocator object (page 337) and initialize the controlled sequence. The allocator object is the argument a1, if present. For the copy constructor, it is x.get_allocator(). Otherwise, it is A().

The first two constructors specify an empty initial controlled sequence. The third constructor specifies a repetition of n elements of value T(). The fourth and fifth constructors specify a repetition of n elements of value x. The sixth constructor specifies a copy of the sequence controlled by x. If InIt is an integer type, the last two constructors specify a repetition of (size_type)first elements of value (T)last. Otherwise, the last two constructors specify the sequence [first, last).

All constructors copy N elements and perform no interim reallocation (page 406).

vector<bool, A>

```
template<class A>  
    class vector<bool, A> {  
public:  
    class reference;  
    typedef bool const_reference;  
    typedef T0 iterator;  
    typedef T1 const_iterator;  
    typedef T4 pointer;  
    typedef T5 const_pointer;  
    void flip();  
    static void swap(reference x, reference y);  
    // rest same as template class vector  
};
```

The class is a partial specialization of template class vector (page 404) for elements of type bool. It alters the definition of four member types (to optimize the packing and unpacking of elements) and adds two member functions. Its behavior is otherwise the same as for template class vector.

vector<bool, A>::const_iterator

```
typedef T1 const_iterator;
```

The type describes an object that can serve as a constant random-access iterator for the controlled sequence. It is described here as a synonym for the unspecified type T1.

vector<bool, A>::const_pointer

```
typedef T5 const_pointer;
```

The type describes an object that can serve as a pointer to a constant element of the controlled sequence. It is described here as a synonym for the unspecified type T5.

vector<bool, A>::const_reference

```
typedef bool const_reference;
```

The type describes an object that can serve as a constant reference to an element of the controlled sequence, in this case bool.

vector<bool, A>::flip

```
void flip();
```

The member function inverts the values of all the members of the controlled sequence.

vector<bool, A>::iterator

```
typedef T0 iterator;
```

The type describes an object that can serve as a random-access iterator for the controlled sequence. It is described here as a synonym for the unspecified type T0.

vector<bool, A>::pointer

```
typedef T4 pointer;
```

The type describes an object that can serve as a pointer to an element of the controlled sequence. It is described here as a synonym for the unspecified type T4.

vector<bool, A>::reference

```
class reference {
public:
    reference& operator=(const reference& x);
    reference& operator=(bool x);
    void flip();
    bool operator~() const;
    operator bool() const;
};
```

The type describes an object that can serve as a reference to an element of the controlled sequence. Specifically, for two objects x and y of class reference:

- **bool(x)** yields the value of the element designated by x
- **~x** yields the inverted value of the element designated by x
- **x.flip()** inverts the value stored in x
- **y = bool(x)** and **y = x** both assign the value of the element designated by x to the element designated by y

It is unspecified how member functions of class vector<bool> construct objects of class reference that designate elements of a controlled sequence. The default constructor for class reference generates an object that refers to no such element.

vector<bool, A>::swap

```
void swap(reference x, reference y);
```

The static member function swaps the members of the controlled sequences designated by x and y.

Portions derived from work copyright © 1994 by Hewlett-Packard Company. All rights reserved.

Notices

Dinkumware, Ltd.

Genuine Software

Dinkumware, Ltd.
398 Main Street
Concord MA 01742
USA
+1-978-371-2773

Dinkum C++ Library developed by P.J. Plauger

Dinkum C++ Library Reference developed by P.J. Plauger

The Dinkum[®] C++ Library in machine-readable or printed form ("Dinkum Library") and the Dinkum C++ Library Reference in machine-readable or printed form ("Dinkum Reference"), hereafter in whole or in part the "Product," are all copyright © 1992-1999 by P.J. Plauger. ALL RIGHTS RESERVED. The Product is derived in part from books copyright © 1992-1999 by P.J. Plauger.

Dinkumware, Ltd. and P.J. Plauger ("Licensor") retain exclusive ownership of this Product. It is licensed to you ("Licensee") in accordance with the terms specifically stated in this Notice. If you have obtained this Product from a third party or under a special license from Dinkumware, Ltd., additional restrictions may also apply. You must otherwise treat the Product the same as other copyrighted material, such as a book or recording. You may also exercise certain rights peculiar to computer software under copyright law. In particular:

- You may use the Library portion of the Product (if present) to compile and link with C/C++ code to produce executable files.
- You may freely distribute such executable files for no additional license fee to Licensor.
- You may make one or more backup copies of the Product for archival purposes.
- You may permanently transfer ownership of the Product to another party only if the other party agrees to the terms stated in this Notice and you transfer or destroy all copies of the Product that are in your possession.
- You must preserve this Notice and all copyright notices with any copy you make of the Product.
- You may not loan, rent, or sublicense the Product.
- You may not copy or distribute, in any form, any part of this Product for any purpose not specifically permitted by this Notice.

This copy of the Product is licensed for use by a limited number of developers, which is specified as part of the packaging for this Product. A license for up to ten users, for example, limits to ten the number of developers reasonably able to use the Product at any instant of time. Thus, ten is the maximum number of **possible** concurrent users, not the number of **actual** concurrent users. A single-user license is for use by just one developer.

Anyone who accesses this software has a moral responsibility not to aid or abet illegal copying by others. Licensor recognizes that the machine-readable format of

the Product makes it particularly conducive to sharing within multi-user systems and across networks. Such use is permitted only so long as Licensee does not exceed the maximum number of possible concurrent users and takes reasonable precautions to protect the Product against unauthorized copying and against public access. In particular, please note that the ability to **access** this copy does not imply permission to **use** it or to **copy** it. Please note also that Licensor has expended considerable professional effort in the production of this Product, and continues to do so to keep it current.

Licensor warrants that the Product as shipped performs substantially in accordance with its documented purpose, and that the medium on which the Product is provided is free from defects in material and workmanship. To the extent permitted by law, any implied warranties on the Product are limited to 90 days.

Licensor's entire liability under this warranty shall be, at Licensor's option, either to refund the license fee paid by Licensee or to replace the medium on which the Product is provided. This is also Licensee's exclusive remedy. To qualify for this remedy, Licensee must demonstrate satisfactory proof of purchase to Licensor and return the Product in reasonably good condition to Licensor.

LICENSOR OTHERWISE MAKES NO REPRESENTATIONS OR WARRANTIES ABOUT THE SUITABILITY OF THIS PRODUCT, EITHER EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NON-INFRINGEMENT. LICENSOR SHALL NOT BE LIABLE FOR ANY DAMAGES SUFFERED BY LICENSEE AS A RESULT OF USING THIS PRODUCT, EVEN IF LICENSOR HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. TO THE EXTENT PERMITTED BY LAW, LICENSOR SHALL NOT BE LIABLE FOR CONSEQUENTIAL OR INCIDENTAL DAMAGES.

By using this Product, you agree to abide by the intellectual property laws and all other applicable laws of the USA, and the terms described above. You may be held legally responsible for any infringement that is caused or encouraged by your failure to abide by the terms of this Notice.

RESTRICTED RIGHTS: Use, duplication, or disclosure by the government is subject to the restrictions as set forth in subparagraph (c)(1)(ii) of the Rights in Technical Data and Computer Software Clause as DFARS 52.227-7013 and FAR 52.227-19. Unpublished rights are reserved under the Copyright Laws of the USA. Contractor/ Manufacturer is DINKUMWARE, LTD., 398 Main Street, Concord MA 01742.

The terms of this notice shall be governed by the laws of the Commonwealth of Massachusetts. THE RIGHTS AND OBLIGATIONS OF THE PARTIES SHALL NOT BE GOVERNED BY THE PROVISIONS OF THE U.N. CONVENTION FOR THE INTERNATIONAL SALE OF GOODS, 1980.

Dinkumware and Dinkum are registered trademarks of Dinkumware, Ltd.

End of Copyright and License Notice

References

- **ANSI Standard X3.159-1989** (New York NY: American National Standards Institute, 1989). The original C Standard, developed by the ANSI-authorized committee X3J11. The Rationale that accompanies the C Standard explains many of the decisions that went into it, if you can get your hands on a copy.
- **ISO/IEC Standard 9899:1990** (Geneva: International Standards Organization, 1990). The official C Standard around the world. Aside from formatting details and section numbering, the ISO C Standard is identical to the ANSI C Standard.
- **ISO/IEC Amendment 1 to Standard 9899:1990** (Geneva: International Standards Organization, 1995). The first (and only) amendment to the C Standard. It provides substantial support for manipulating large character sets.
- **ISO/IEC Standard 14882:1998** (Geneva: International Standards Organization, 1998). The official C++ Standard around the world. The ISO C++ Standard is identical to the ANSI C++ Standard.
- * P.J. Plauger, **The Standard C Library** (Englewood Cliffs NJ: Prentice Hall, 1992). Contains a complete implementation of the Standard C library, as well as text from the library portion of the C Standard and guidance in using the Standard C library.
- * P.J. Plauger, **The Draft Standard C++ Library** (Englewood Cliffs NJ: Prentice Hall, 1995). Contains a complete implementation of the draft Standard C++ library as of early 1994.
- P.J. Plauger, Alexander Stepanov, Meng Lee, and David R. Musser, **The Standard Template Library** (Englewood Cliffs NJ: Prentice Hall, 1999). Contains a complete implementation of the Standard Template Library as incorporated into the C++ Standard.

Bug Reports

The author welcomes reports of any errors or omissions. Please report any bugs or difficulties to:

P.J. Plauger
Dinkumware, Ltd.
398 Main Street
Concord MA 01742-2321
USA

+1-978-371-2773 (UTC -4 hours, -5 November through March)
+1-978-371-9014 (FAX)
service@dinkumware.com

Copyright © 1989-1999 by P.J. Plauger. All rights reserved.