17 Library introduction [lib.library]

1 This clause describes the contents of the *C++ Standard library*, how a well-formed C++ program makes use of the library, and how a conforming implementation may provide the entities in the library.

2 The C++ Standard library provides an extensible framework, and contains components for: language support, diagnostics, general utilities, strings, locales, containers, iterators, algorithms, numerics, and input/output. The language support components are required by certain parts of the C++ language, such as memory allocation (5.3.4, 5.3.5) and exception processing (_except.intro_).

3 The general utilities include components used by other library elements, such as a predefined storage allocator for dynamic storage management (3.7.3). The diagnostics components provide a consistent framework for reporting errors in a C++ program, including predefined exception classes.

4 The strings components provide support for manipulating text represented as sequences of type `char`, sequences of type `wchar_t`, or sequences of any other ''character-like'' type. The localization components extend Internationalization support for such text processing.

5 The containers, iterators, and algorithms provide a C++ program with access to a subset of the most widely used algorithms and data structures.

6 Numeric algorithms and the complex number components extend support for numeric processing. The `valarray` components provide support for *n*-at-a-time processing, potentially implemented as parallel operations on platforms that support such processing.

7 The `iostreams` components are the primary mechanism for C++ program input/output. They can easily be used with other elements of the library, particularly strings, locales, and iterators.

8 This library also makes available the facilities of the Standard C library, suitably adjusted to ensure static type safety.

9 The following subclauses describe the definitions (17.1), and method of description (17.2) for the library. Subclauses (17.3) and (18) through (27) specify the contents of the library, and library requirements and constraints on both well-formed C++ programs and conforming implementations.

17.1 Definitions [lib.definitions]

— **category:** A logical collection of library entities. Subclauses 18 through 27 each describe a single category of entities within the library.

— **comparison function:** An operator function (13.4) for any of the equality (5.10) or relational (5.9) operators.

— **component:** A group of library entities directly related as members, parameters, or return types. For example, the class `wstring` and the non-member functions that operate on wide-character strings can be considered the *wstring component*.

— **default behavior:** A description of *replacement function* and *handler function* semantics. Any specific behavior provided by the implementation, within the scope of the *required behavior*.

— **handler function:** A non- *reserved function* whose definition may be provided by a C++ program. A C++ program may designate a handler function at various points in its execution, by supplying a pointer to the function when calling any of the library functions that install handler functions (18).

— **modifier function:** A class member function (9.4), other than constructors, assignment, or destructor, that alters the state of an object of the class. The state of an object can be obtained by using one or more *observer functions*

— **observer function:** A class member function (9.4) that accesses the state of an object of the class, but does not alter that state. Observer functions are specified as `const` member functions (9.4.2).

— **replacement function:** A non- *reserved function* whose definition is provided by a C++ program. Only one definition for such a function is in effect for the duration of the program's execution, as the result of creating the program (2.1) and resolving the definitions of all translation units (3.5).

— **required behavior:** A description of *replacement function* and *handler function* semantics, applicable to both the behavior provided by the implementation and the behavior that shall be provided by any function definition in the program. If a function defined in a C++ program fails to meet the required behavior when it executes, the behavior is undefined.

— **reserved function:** A function, specified as part of the C++ Standard library, that must be defined by the implementation. If a C++ program provides a definition for any reserved function, the results are undefined. Subclause 1.3 defines additional terms used elsewhere in this International Standard.

17.2 Method of description (Informative) [lib.description]

1 This subclause describes the conventions used to describe the C++ Standard library. It describes the structures of the normative clauses 18 through 27 (17.2.1), conventions used to specify constraints on template arguments(_lib.template.constraints_), conventions to describe types defined by an implementation (_lib.implementation.types_), and other editorial conventions (_lib.res.and.conventions_).

17.2.1 Structure of each subclause [lib.structure]

1 Subclause 17.3.1 provides a summary of the C++ Standard library's contents. Other Library clauses provide detailed specifications for each of the components in the library, as shown in Table 10:

Table 10—Library Categories

Clause	Category
18	Language support
19	Diagnostics
20	General utilities
21	Strings
22	Localization
23	Containers
24	Iterators
25	Algorithms
26	Numerics
27	Input/output

2 Each Library clause contains the following elements, as applicable:[84)]

— Summary

— Requirements

— Detailed specifications

[84)] To save space, items that do not apply to a clause are omitted. For example, if a clause does not specify any requirements on template arguments, there will be no ''Requirements'' subclause.

— References to the Standard C library

17.2.1.1 Summary [lib.structure.summary]

1 The Summary provides a synopsis of the category, and introduces the first-level subclauses. Each subclause also provides a summary, listing the headers specified in the subclause and the library entities provided in each header.

2 Paragraphs labelled ‘‘Note(s):’’ or ‘‘Example(s):’’ are informative, other paragraphs are normative.

3 The summary and the detailed specifications are presented in the order:

— Macros

— Values

— Types

— Classes

— Functions

— Objects

17.2.1.2 Requirements [lib.structure.requirements]

1 The library can be extended by a C++ program. Each clause, as applicable, describes the requirements that such extensions must meet. Such extensions are generally one of the following:

— Template arguments

— Derived classes

— Containers, iterators, and/or algorithms that meet an interface convention

2 The string and iostreams components use an explicit representation of operations required of template arguments. They use a template class name *XXX*`_traits` to define these constraints.

3 Interface convention requirements are stated as generally as possible. Instead of stating ‘‘class X has to define a member function ’’`operator++()`, the interfce requires ‘‘for any object `x` of class `X`, `++x` is defined.’’ That is, whether the operator is a member or a global function is unspecified.

4 Requirements are stated in terms of well-defined expressions, which define valid terms of the types that satisfy the requirements. For every set of requirements there is a table that specifies an initial set of the valid expressions and their semantics. Any generic algorithm that uses the requirements is described in terms of the valid expressions for its formal type parameters.

5 In some cases the semantic requirements are presented as C++ code. Such code is intended as a specification of equivalence of a construct to another construct, not necessarily as the way the construct must be implemented.[85)]

17.2.1.3 Specifications [lib.structure.specifications]

1 The detailed specifications each contain the following elements:[86)]

— Name and brief description

— Synopsis (class definition or function prototype, as appropriate)

— Restrictions on template arguments, if any

— Decription of class invariants

85) Although in some cases the code given is unambiguously the optimum implementation.

86) The form of these specifications was designed to follow the conventions established by existing C++ library vendors.

— Description of function semantics

2 Descriptions of class member functions follow the order (as appropriate):[87)]

— Constructor(s) and destructor

— Copying & assignment functions

— Comparison functions

— Modifier functions

— Observer functions

— Operators and other non-member functions

3 Descriptions of function semantics contain the following elements (as appropriate):[88)]

— Requires, the preconditions for calling the function

— Effects, the actions performed by the function

— Postconditions, the observable results established by the function

— Returns, a description of the value(s) returned by the function

— Throws, any exceptions thrown by the function, and the conditions that would cause the exception

4 Complexity, the time and/or space complexity of the function

5 For non-reserved replacement and handler functions, this clause specifies two behaviors for the functions in question: their required and default behavior. The *default behavior* describes a function definition provided by the implementation. The *required behavior* describes the semantics of a function definition provided by either the implementation or a C++ program. Where no distinction is explicitly made in the description, the behavior described is the required behavior. *

6 If an operation is required to be linear time, it means no worse than linear time, and a constant time operation satisfies the requirement. *

17.2.1.4 C Library [lib.structure.see.also]

1 Paragraphs labelled ''SEE ALSO:'' contain cross-references to the relevant portions of this Standard and the ISO C standard, which is incorporated into this Standard by reference.

17.2.2 Other conventions [lib.conventions]

1 This subclause describes several editorial conventions used to describe the contents of the C++ Standard library. These conventions are for describing implementation-defined types (17.2.2.1), class and class template member functions (17.2.2.2), private members (17.2.2.3), and convenient names (17.2.2.4).

Box 76

ISSUE: this subclause needs more discussion by the Library WG.

87) To save space, items that do not apply to a class are omitted. For example, if a class does not specify any comparison functions, there will be no ''Comparison functions'' subclause.

88) To save space, items that do not apply to a function are omitted. For example, if a function does not specify any preconditions, there will be no ''Requires'' paragraph.

17.2.2.1 Type descriptions [lib.type.descriptions]

1 The Requirements subclauses describe names that are used to specify constraints on template arguments. These names are used, instead of the keyword `class`, to describe the types that may be supplied as arguments by a C++ program when instantiating template components from the library.

2 Certain types defined in this clause are used to describe implementation-defined types. They are based on other types, but with added constraints.

17.2.2.1.1 Enumerated types [lib.enumerated.types]

1 Several types defined in Clause 27 are *enumerated types*. Each enumerated type may be implemented as an enumeration or as a synonym for an enumeration.

2 The enumerated type *`enumerated`* can be written:

```
enum secret {
        V0, V1, V2, V3, .....};
typedef secret enumerated;
static const enumerated C0(V0);
static const enumerated C1(V1);
static const enumerated C2(V2);
static const enumerated C3(V3);
        .....
```

3 Here, the names *`C0`*, *`C1`*, etc. represent *enumerated elements* for this particular enumerated type. All such elements have distinct values.

17.2.2.1.2 Bitmask types [lib.bitmask.types]

1 Several types defined in Clause 27 are *bitmask types*. Each bitmask type can be implemented as an enumerated type that overloads certain operators.

2 The bitmask type *`bitmask`* can be written:

```
enum secret {
        V0 = 1 << 0, V1 = 1 << 1, V2 = 1 << 2, V3 = 1 << 3, .....};
typedef secret bitmask;
static const bitmask C0(V0);
static const bitmask C1(V1);
static const bitmask C2(V2);
static const bitmask C3(V3);
        .....
bitmask& operator&=(bitmask& X, bitmask Y)
        {X = (bitmask)(X & Y); return (X); }
bitmask& operator|=(bitmask& X, bitmask Y)
        {X = (bitmask)(X | Y); return (X); }
bitmask& operator^=(bitmask& X, bitmask Y)
        {X = (bitmask)(X ^ Y); return (X); }
bitmask operator&(bitmask X, bitmask Y)
        {return ((bitmask)(X & Y)); }
bitmask operator|(bitmask X, bitmask Y)
        {return ((bitmask)(X | Y)); }
bitmask operator^(bitmask X, bitmask Y)
        {return ((bitmask)(X ^ Y)); }
bitmask operator~(bitmask X)
        {return ((bitmask)~X); }
```

3 Here, the names *`C0`*, *`C1`*, etc. represent *bitmask elements* for this particular bitmask type. All such elements have distinct values such that, for any pair *`Ci`* and *`Cj`*, *`Ci & Ci`* is nonzero and *`Ci & Cj`* is zero.

4 The following terms apply to objects and values of bitmask types:

— To *set* a value `Y` in an object `X` is to evaluate the expression `X |= Y`.

— To *clear* a value `Y` in an object `X` is to evaluate the expression `X &= ~Y`.

— The value `Y` *is set* in the object `X` if the expression `X & Y` is nonzero.

17.2.2.1.3 Character sequences [lib.character.seq]

1 The Standard C library makes widespread use of characters and character sequences that follow a few uniform conventions:

— A *letter* is any of the 26 lowercase or 26 uppercase letters in the basic execution character set.[89)]

— The *decimal-point character* is the (single-byte) character used by functions that convert between a (single-byte) character sequence and a value of one of the floating-point types. It is used in the character sequence to denote the beginning of a fractional part. It is represented in this clause by a period, `'.'`, which is also its value in the `"C"` locale, but may change during program execution by a call to `setlocale(int, const char*)`, declared in `<clocale>`(22.3). *

— A *character sequence* is an array object (8.3.4) `A` that can be declared as `T A[N]`, where `T` is any of the types `char`, `unsigned char`, or `signed char` (3.8.1), optionally qualified by any combination of `const` or `volatile`. The initial elements of the array have defined contents up to and including an element determined by some predicate. A character sequence can be designated by a pointer value `S` that points to its first element.

17.2.2.1.3.1 Byte strings [lib.byte.strings]

1 A *null-terminated byte string*, or *NTBS*, is a character sequence whose highest-addressed element with defined content has the value zero (the *terminating null* character).[90)]

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

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

4 A *static NTBS* is an NTBS with static storage duration.[91)]

17.2.2.1.3.2 Multibyte strings [lib.multibyte.strings]

1 A *null-terminated multibyte string*, or *NTMBS*, is an NTBS that constitutes a sequence of valid multibyte characters, beginning and ending in the initial shift state.[92)]

2 A *static NTMBS* is an NTMBS with static storage duration.

17.2.2.1.3.3 Wide-character sequences [lib.wide.characters]

1 A *wide-character sequence* is an array object (8.3.4) `A` that can be declared as `T A[N]`, where `T` is type `wchar_t` (_basic.fundmental_), optionally qualified by any combination of `const` or `volatile`. The initial elements of the array have defined contents up to and including an element determined by some predicate. A character sequence can be designated by a pointer value `S` that designates its first element.

---

89) Note that this definition differs from the definition in ISO C subclause 7.1.1.

90) Many of the objects manipulated by function signatures declared in `<cstring>`(21.2) are character sequences or NTBSs. The size of some of these character sequences is limited by a length value, maintained separately from the character sequence.

91) A string literal, such as `abc`, is a static NTBS.

92) An NTBS that contains characters only from the basic execution character set is also an NTMBS. Each multibyte character then consists of a single byte.

2 A *null-terminated wide-character string*, or *NTWCS*, is a wide-character sequence whose highest-addressed element with defined content has the value zero.[93)]

3 The *length of an NTWCS* is the number of elements that precede the terminating null wide character. An *empty NTWCS* has a length of zero.

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

5 A *static NTWCS* is an NTWCS with static storage duration.[94)]

17.2.2.2 Functions within classes [lib.functions.within.classes]

1 For the sake of exposition, Clauses 18 through 27 do not describe copy constructors, assignment operators, or (non-virtual) destructors with the same apparent semantics as those that can be generated by default (12.1, 12.4, 12.8).

2 It is unspecified whether the implementation provides explicit definitions for such member function signatures, or for virtual destructors that can be generated by default.

3 For the sake of exposition, Clauses 18 through 27 repeat in derived class declarations any virtual member functions inherited from a base class. All such declarations are enclosed in a comment that ends with ***inherited***, as in:

```
//      virtual void do_raise();        inherited
```

4 If a virtual member function in the base class meets the semantic requirements of the derived class, it is unspecified whether the derived class provides an overriding definition for the function signature.

17.2.2.3 Private members [lib.objects.within.classes]

1 Objects of certain classes are sometimes required by the external specifications of their classes to store data, apparently in member objects. For the sake of exposition, this clause provides representative declarations, and semantic requirements, for private member objects of classes that meet the external specifications of the classes. The declarations for such member objects and the definitions of related member types in this clause are enclosed in a comment that ends with ***exposition only***, as in:

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

2 Any alternate implementation that provides equivalent external behavior is equally acceptable.

Box 77

Editorial Proposal: Eliminate the ‘‘exposition only’’ implementation specification. Instead, describe member function pre- and post-conditions in terms of observer functions. Retained state, and side effects to that state, will require careful wording.

17.2.2.4 Convenient names [lib.unreserved.names]

1 Certain types defined in C headers are sometimes needed to express declarations in other headers, where the required type names are neither defined nor reserved. In such cases, the implementation provides a synonym for the required type, using a name reserved to the implementation. Such cases are explicitly stated in this clause, and indicated by writing the required type name in *constant-width italic* characters.

---

93) Many of the objects manipulated by function signatures declared in <cwchar> are wide-character sequences or NTWCSs.

94) A wide string literal, such as L"abc", is a static NTWCS.

2 Certain names are sometimes convenient to supply for the sake of exposition, in the descriptions in this clause, even though the names are neither defined nor reserved. In such cases, the implementation either omits the name, where that is permitted, or provides a name reserved to the implementation. Such cases are also indicated in this clause by writing the convenient name in *constant-width italic* characters.

3 For example:

4 The class `filebuf`, defined in `<fstream>`, is described as containing the private member object:

```
FILE* file;
```

5 This notation indicates that the member *`file`* is a pointer to the type `FILE`, defined in `<cstdio>`, but the names `file` and `FILE` are neither defined nor reserved in `<fstream>`. An implementation need not implement class `filebuf` with an explicit member of type `FILE*`. If it does so, it can choose 1) to replace the name *`file`* with a name reserved to the implementation, and 2) to replace *`FILE`* with an incomplete type whose name is reserved, such as in:

```
struct _Filet* _Fname;
```

6 If the program needs to have type `FILE` defined, it must also include `<cstdio>`, which completes the definition of `_Filet`.

17.3 Library-wide requirements [lib.requirements]

1 This subclause specifies requirements that apply to the entire C++ Standard library. Clauses 18 through 27 specify the requirements of individual entities within the library.

2 The following subclauses describe the library's contents and organization (17.3.1), how well-formed C++ programs gain access to library entities (17.3.2), constraints on such programs (17.3.3), and constraints on conforming implementations (17.3.4).

17.3.1 Library contents and organization [lib.organization]

1 This subclause provides a summary of the entities defined in the C++ Standard library. Subclause 17.3.1.1 provides an alphabetical listing of entities by type, while subclause 17.3.1.2 provides an alphabetical listing of library headers.

17.3.1.1 Library contents [lib.contents]

1 The C++ Standard library provides definitions for the following types of entities:

— Macros

— Values

— Types

— Templates

— Classes

— Functions

— Objects

2 All library entities shall be defined within the namespace `std`.

3 The C++ Standard library provides 54 standard macros, as shown in Table 11.

4 The header names (enclosed in < and >) indicate that the macro may be defined in more than one header. All such definitions shall be equivalent (3.2).

Table 11—Standard Macros

assert	LC_ALL	NULL <cstring>	SIGILL	va_arg
BUFSIZ	LC_COLLATE	NULL <ctime>	SIGINT	va_end
CLOCKS_PER_SEC	LC_CTYPE	NULL <cwchar>	SIGSEGV	va_start
EDOM	LC_MONETARY	offsetof	SIGTERM	WCHAR_MAX
EOF	LC_NUMERIC	RAND_MAX	SIG_DFL	WCHAR_MIN
ERANGE	LC_TIME	SEEK_CUR	SIG_ERR	WEOF <cwchar>
EXIT_FAILURE	L_tmpnam	SEEK_END	SIG_IGN	WEOF <cwctype>
EXIT_SUCCESS	MB_CUR_MAX	SEEK_SET	stderr	_IOFBF
FILENAME_MAX	NDEBUG	setjmp	stdin	_IOLBF
FOPEN_MAX	NULL <cstddef>	SIGABRT	stdout	_IONBF
HUGE_VAL	NULL <cstdio>	SIGFPE	TMP_MAX	

5 The C++ Standard library provides 46 standard values, as shown in Table 12:

Table 12—Standard Values

CHAR_BIT	FLT_DIG	INT_MIN	MB_LEN_MAX
CHAR_MAX	FLT_EPSILON	LDBL_DIG	NPOS
CHAR_MIN	FLT_MANT_DIG	LDBL_EPSILON	SCHAR_MAX
DBL_DIG	FLT_MAX	LDBL_MANT_DIG	SCHAR_MIN
DBL_EPSILON	FLT_MAX_10_EXP	LDBL_MAX	SHRT_MAX
DBL_MANT_DIG	FLT_MAX_EXP	LDBL_MAX_10_EXP	SHRT_MIN
DBL_MAX	FLT_MIN	LDBL_MAX_EXP	UCHAR_MAX
DBL_MAX_10_EXP	FLT_MIN_10_EXP	LDBL_MIN	UINT_MAX
DBL_MAX_EXP	FLT_MIN_DIG	LDBL_MIN_10_EXP	ULONG_MAX
DBL_MIN	FLT_RADIX	LDBL_MIN_DIG	USHRT_MAX
DBL_MIN_10_EXP	FLT_ROUNDS	LONG_MAX	
DBL_MIN_DIG	INT_MAX	LONG_MIN	

6 The C++ Standard library provides 51 standard types, as shown in Table 13:

Table 13—Standard Types

clock_t	new_handler	streamoff	wint_t <stddef>
div_t	ofstream	streampos	wios
FILE	ostream	string	wios
filebuf	ostringstream	stringbuf	wistream
fpos_t	ptrdiff_t <stddef>	terminate_handler	wistringstream
fvoid_t	ptrdiff_t<cstddef>	unexpected_handler	wofstream
ifstream	sig_atomic_t	va_list	wostream
ios	size_t <cstddef>	wctrans_t	wostringstream
istream	size_t <cstdio>	wctype_t	wstreambuf
istringstream	size_t <cstring>	wfilebuf	wstreampos
jmp_buf	size_t <ctime>	wifstream	wstring
ldiv_t	size_t <stddef>	wint_t <cwchar>	wstringbuf
mbstate_t	streambuf	wint_t <cwctype>	

7 The C++ Standard library provides 68 standard template classes, as shown in Table 14:

Table 14—Standard Template classes

allocator	mask_array
auto_ptr	messages
back_insert_iterator	messages_byname
basic_convbuf	moneypunct
basic_filebuf	moneypunct_byname
basic_ifstream	money_get
basic_ios	money_put
basic_istream	multimap
basic_istringstream	multiset
basic_ofstream	numeric_limits
basic_ostream	numpunct
basic_ostringstream	num_get
basic_streambuf	num_put
basic_string	ostreambuf_iterator
basic_stringbuf	ostream_iterator
binary_negate	pointer_to_binary_function
binder1st	pointer_to_unary_function
binder2nd	priority_queue
bits	queue
codecvt	raw_storage_iterator
codecvt_byname	restrictor
collate	reverse_bidirectional_iterator
collate_byname	reverse_iterator
complex	set
ctype	slice_array
ctype_byname	stack
deque	time_get
front_insert_iterator	time_get_byname
gslice_array	time_put
indirect_array	time_put_byname
insert_iterator	unary_negate
istreambuf_iterator	valarray
istream_iterator	vector
list	
map	

8 The C++ Standard library provides 26 standard template structures, as shown in Table 15:

Table 15—Standard Template structs

bidirectional_iterator	ios_conv_traits	negate
binary_function	ios_traits	not_equal_to
conv_traits	less	pair
divides	less_equal	plus
equal_to	logical_and	random_access_iterator
forward_iterator	logical_not	string_char_traits
greater	logical_or	times
greater_equal	minus	unary_function
input_iterator	modulus	

9 The C++ Standard library provides 78 standard template operator functions, as shown in Table 16.

10 Types shown (enclosed in (and)) indicate that the given function is overloaded by that type. Numbers shown (enclosed in [and]) indicate how many overloaded functions are overloaded by that type.

Table 16—Standard Template operators

operator!= (basic_string) [5]	operator< (vector)
operator!= (complex) [3]	operator<< (complex)
operator!= (istreambuf_iterator)	operator<< (valarray) [3]
operator!= (ostreambuf_iterator)	operator<<=(valarray) [2]
operator!= (T)	operator<= (T)
operator!= (valarray) [3]	operator<= (valarray) [3]
operator% (valarray) [3]	operator== (basic_string) [5]
operator%= (valarray) [2]	operator== (complex) [3]
operator& (valarray) [3]	operator== (deque)
operator&& (valarray) [3]	operator== (istreambuf_iterator)
operator&= (valarray) [2]	operator== (istream_iterator)
operator* (complex) [3]	operator== (list)
operator* (valarray) [3]	operator== (map)
operator*= (complex)	operator== (multimap)
operator*= (valarray) [2]	operator== (multiset)
operator+ (basic_string) [5]	operator== (ostreambuf_iterator)
operator+ (complex) [4]	operator== (pair)
operator+ (reverse_iterator)	operator== (queue)
operator+ (valarray) [3]	operator== (restrictor)
operator+= (complex)	operator== (reverse_bidir_iter)
operator+= (valarray) [2]	operator== (reverse_iterator)
operator- (complex) [4]	operator== (set)
operator- (reverse_iterator)	operator== (stack)
operator- (valarray) [3]	operator== (valarray) [3]
operator-= (complex)	operator== (vector)
operator-= (valarray) [2]	operator> (T)
operator/ (complex) [3]	operator> (valarray) [3]
operator/ (valarray) [3]	operator>= (T)
operator/= (complex)	operator>= (valarray) [3]
operator/= (valarray) [2]	operator>> (basic_string)
operator< (deque)	operator>> (complex)
operator< (list)	operator>> (valarray) [3]
operator< (map)	operator>>=(valarray) [2]
operator< (multimap)	operator^ (valarray) [3]
operator< (multiset)	operator^= (valarray) [2]
operator< (pair)	operator| (valarray) [3]
operator< (restrictor)	operator|= (valarray) [2]
operator< (reverse_iterator)	operator|| (valarray) [3]
operator< (set)	
operator< (valarray) [3]	

11 The C++ Standard library provides 135 standard template functions, as shown in Table 17:

Table 17—Standard Template functions

abs (complex)	make_heap [2]
abs (valarray)	make_pair
accumulate [2]	max [2]
acos (valarray)	max_element [2]
adjacent_difference [2]	merge [2]
adjacent_find [2]	min [2]
advance	min_element [2]
allocate	mismatch [2]
arg (complex)	next_permutation [2]
asin (valarray)	norm (complex)
atan (valarray)	not1
atan2(valarray) [3]	not2
back_inserter	nth_element [2]
binary_search [2]	partial_sort [2]
bind1st	partial_sort_copy [2]
bind2nd	partial_sum [2]
conj (complex)	partition
construct	polar(complex)
copy	pop_heap [2]
copy_backward	pow (complex)
cos (complex)	pow (valarray) [3]
cos (valarray)	prev_permutation [2]
cosh (complex)	ptr_fun [2]
cosh (valarray)	push_heap [2]
count	random_shuffle [2]
count_if	real (complex)
deallocate	remove
destroy [2]	remove_copy
distance	remove_copy_if
distance_type (istreambuf_iterator)	remove_if
distance_type [5]	replace
equal [2]	replace_copy
equal_range [2]	replace_copy_if
exp (complex)	replace_if
exp (valarray)	reverse
fill	reverse_copy
fill_n	rotate
find	rotate_copy
find_end [4]	search [4]
find_first_of [2]	set_difference [2]
find_if	set_intersection [2]
for_each	set_symmetric_difference [2]
front_inserter	set_union [2]
generate	sin (complex)

generate_n	sin (valarray)
get_temporary_buffer	sinh (complex)
imag (complex)	sinh (valarray)
includes [2]	sort [2]
inner_product [2]	sort_heap [2]
inplace_merge [2]	sqrt (complex)
inserter	sqrt (valarray)
isalnum	stable_partition
isalpha	stable_sort [2]
iscntrl	swap
isdigit	swap_ranges
isgraph	tan (valarray)
islower	tanh (valarray)
isprint	tolower
ispunct	toupper
isspace	transform [2]
isupper	uninitialized_copy
isxdigit	uninitialized_fill_n
iterator_category [7]	unique [2]
lexicographical_compare [2]	unique_copy [2]
log (complex)	unititialized_fill
log (valarray)	upper_bound [2]
log10(valarray)	value_type [7]
lower_bound [2]	

12 The C++ Standard library provides 25 standard classes, as shown in Table 18.

13 Type names (enclosed in < and >) indicate that these are specific instances of templates.

Table 18—Standard Classes

bad_alloc	exception	out_of_range
bad_cast	gslice	overflow_error
bad_typeid	invalid_argument	range_error
complex<double>	ios_base	runtime_error
complex<float>	length_error	slice
complex<long double>	locale	type_info
ctype<char>	locale::facet	vector<bool,allocator>
ctype_byname<char>	locale::id	
domain_error	logic_error	

14 The C++ Standard library provides 21 standard structures, as shown in Table 19:

Table 19—Standard Structs

bidirectional_iterator_tag	money_base
codecvt_base	money_base::pattern
conv_traits<wchar_t>	output_iterator
ctype_base	output_iterator_tag
empty	random_access_iterator_tag
forward_iterator_tag	string_char_traits<char>
input_iterator_tag	string_char_traits<wchar_t>
ios4traits<wstreampos>	time_base
ios_traits<char>	tm <ctime>
ios_traits<wchar_t>	tm <cwchar>
lconv	

15 The C++ Standard library provides 18 standard operator functions, as shown in Table 20:

Table 20—Standard Operator functions

operator delete	operator<< (bits)
operator delete[]	operator<< (locale)
operator new	operator== (empty)
operator new (void*)	operator== (vector<bool,allocator>)
operator new[]	operator>> (bits)
operator new[] (void*)	operator>> (locale)
operator& (bits)	operator^ (bits)
operator< (empty)	operator| (bits)
operator< (vector<bool,allocator>)	
operator<< (basic_string)	

16 The C++ Standard library provides 243 standard functions, as shown in Table 21:

Table 21—Standard Functions

abort	getenv	oct	strxfrm
abs	getline	perror	swprintf
acos	gets	pow	swscanf
asctime	getwc	printf	system
asin	getwchar	putc	tan
atan	gmtime	puts	tanh
atan2	hex	putwc	terminate
atexit	internal	putwchar	time
atof	isalnum	qsort	tmpfile
atoi	isalpha	raise	tmpnam
atol	iscntrl	rand	tolower
bsearch	isdigit	realloc	toupper
btowc	isgraph	remove	towctrans
calloc	islower	rename	towlower
ceil	isprint	resetiosflags	towupper
clearerr	ispunct	rewind	unexpected
clock	isspace	right	ungetc
cos	isupper	scanf	ungetwc
cosh	iswalnum	scientific	uppercase
ctime	iswalpha	setbase	vfwprintf
dec	iswcntrl	setbuf	vprintf
difftime	iswctype	setfill	vprintf
div	iswdigit	setiosflags	vsprintf
endl	iswgraph	setlocale	vswprintf
ends	iswlower	setprecision	vwprintf
exit	iswprint	setvbuf	wcrtomb
exp	iswpunct	setw	wcscat
fabs	iswspace	set_new_handler	wcschr
fclose	iswupper	set_terminate	wcscmp
feof	iswxdigit	set_unexpected	wcscoll
ferror	isxdigit	showbase	wcscpy
fflush	iterator_category	showpoint	wcscspn
fgetc	labs	showpos	wcsftime
fgetpos	ldexp	signal	wcslen
fgets	ldiv	sin	wcsncat
fgetwc	left	sinh	wcsncmp
fgetws	localeconv	skipws	wcsncpy
fixed	localtime	sprintf	wcspbrk
floor	log	sqrt	wcsrchr
flush	log10	srand	wcsrtombs
fmod	longjmp	sscanf	wcsspn
fopen	malloc	strcat	wcsstr

fprintf	mblen	strchr	wcstod
fputc	mbrlen	strcmp	wcstok
fputs	mbrtowc	strcoll	wcstol
fputwc	mbsinit	strcpy	wcstombs
fputws	mbsrtowcs	strcspn	wcstoul
fread	mbstowcs	strerror	wcsxfrm
free	mbtowc	strftime	wctob
freopen	memchr	strlen	wctomb
frexp	memcmp	strncat	wctrans
fscanf	memcpy	strncmp	wctype
fseek	memmove	strncpy	wmemchr
fsetpos	memset	stroul	wmemcmp
ftell	mktime	strpbrk	wmemcpy
fwide	modf	strrchr	wmemmove
fwprintf	noshowbase	strspn	wmemset
fwrite	noshowpoint	strstr	wprintf
fwscanf	noshowpos	strtod	ws
getc	noskipws	strtok	wscanf
getchar	nouppercase	strtol	

17 The C++ Standard library provides 9 standard objects, as shown in Table 22:

Table 22—Standard Objects

cerr	cin	clog	cout	errno	werr	win	wlog	wout

17.3.1.2 Headers [lib.headers]

1 The elements of the C++ Standard library are declared or defined (as appropriate) in a [95] *header*.

2 The C++ Standard library provides 34 *C++ headers*, as shown in Table 23:

Table 23—C++ Library Headers

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

3 The facilities of the Standard C Library are provided in 18 additional headers, as shown in Table 24:

[95] A header is not necessarily a source file, nor are the sequences delimited by < and > in header names necessarily valid source file names (16.2).

Table 24—C++ Headers for C Library Facilities

<cassert>	<ciso646>	<csetjmp>	<cstdio>	<cwchar>
<cctype>	<climits>	<csignal>	<cstdlib>	<cwctype>
<cerrno>	<clocale>	<cstdarg>	<cstring>	
<cfloat>	<cmath>	<cstddef>	<ctime>	

Box 78

ISSUE: Header <ciso646> is not needed: see 2.8, Table 4.

4 Except as noted in Clauses 18 through 27, the contents of each header c *name* shall be the same as that of the corresponding header *name* .h, as specified in ISO C (Clause 4), or Amendment 1, (Clause 7), as appropriate. In this C++ Standard library, however, the declarations and definitions are within namespace scope (3.3.4) of the namespace std.

5 Subclause C.4 describes the effects of using the *name* .h (C header) form in a C++ program.

17.3.1.3 Freestanding implementations [lib.compliance]

1 Two kinds of implementations are defined: *hosted* and *freestanding* (1.7). For a hosted implementation, this International Standard definesdescribes the set of available headers.

2 A freestanding implementation is one in which execution may take place without the benefit of an operating system, and has an implementation-defined set of headers. This set shall include at least the following 8 headers:

Box 79

ISSUE: TBS. Requires a definition of a freestanding environment.

— the headers <new>, <exception>, and <typeinfo> that provide C++ language support (as described in 18)

— the C++ headers <cfloat>, <climits>, <cstdarg>, and <cstddef>

— a version of the C++ header <cstdlib> that declares at least the functions abort, atexit, and exit.

17.3.2 Using the library [lib.using]

1 This subclause describes how a C++ program gains access to the facilities of the C++ Standard library. Subclause 17.3.2.1 describes effects during translation phase 4, while subclause 17.3.2.2 describes effects during phase 8 (2.1).

17.3.2.1 Headers [lib.using.headers]

1 The entities in the C++ Standard library are defined in headers, whose contents are made available to a translation unit when it contains the appropriate #include preprocessing directive (16.2).

2 A translation unit may include library headers in any order (2). Each may be included more than once, with no effect different from being included exactly once, except that the effect of including either <cassert> or <assert.h> depends each time on the lexically current definition of NDEBUG.

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

17.3.2.2 Linkage [lib.using.linkage]

1 Entities in the C++ Standard library have external linkage (3.5). Unless otherwise specified, objects and functions have the default `"extern "C++""` linkage (7.5).

2 Objects and functions defined in the library and required by a C++ program are included in the program prior to program startup. *SEE ALSO:* replacement functions (17.3.3.4), run-time changes (17.3.3.5). *

17.3.3 Constraints on programs [lib.constraints]

1 This subclause describes restrictions on C++ programs that use the facilities of the C++ Standard library. The following subclauses specify constraints on the program's namespace (17.3.3.1), its use of headers (17.3.3.2), classes derived from standard library classes (17.3.3.3), definitions of replacement functions (17.3.3.4), and installation of handler functions during execution (17.3.3.5).

17.3.3.1 Reserved names [lib.reserved.names]

1 The C++ Standard library reserves the following kinds of names:

17.3.4.1 Headers [lib.res.on.headers]

1 Certain types and macros are defined in more than one header. For such an entity, a second or subsequent header that also defines it may be included after the header that provides its initial definition.

2 None of the C headers includes any of the other headers, except that each C header includes its corresponding C++ header, as described above. None of the C++ headers includes any of the C headers. However, any of the C++ headers can include any of the other C++ headers, and must include a C++ header that contains any needed definition.[99)]

17.3.4.2 Restrictions on macro definitions [lib.res.on.macro.definitions]

1 Only the names or global function signatures described in subclause 17.3.1.1 are reserved to the implementation.[100)]

2 All object-like macros defined by the C++ Standard library and described in this clause as expanding to integral constant expressions are also suitable for use in `#if` preprocessing directives, unless explicitly stated otherwise.

17.3.4.3 Global functions [lib.global.functions]

1 A call to a global function signature described in this clause behaves the same as if the implementation declares no additional global function signatures.[101)]

---

96) It is not permissible to remove a library macro definition by using the `#undef` directive.

97) The list of such reserved names includes `errno`, declared or defined in `<cerrno>`.

98) The list of such reserved function signatures with external linkage includes `setjmp(jmp_buf)`, declared or defined in `<csetjmp>`, and `va_end(va_list)`, declared or defined in `<cstdarg>`.

99) Including any one of the C++ headers can introduce all of the C++ headers into a translation unit, or just the one that is named in the `#include` preprocessing directive.

100) A global function cannot be declared by the implementation as taking additional default arguments. Also, the use of masking macros for function signatures declared in C headers is disallowed, notwithstanding the latitude granted in subclause 7.1.7 of the C Standard. The use of a masking macro can often be replaced by defining the function signature as *inline*.

101) A valid C++ program always calls the expected library global function. An implementation may also define additional global functions that would otherwise not be called by a valid C++ program.

17.3.4.4 Member functions [lib.member.functions]

1 An implementation can declare additional non-virtual member function signatures within a class:

— by adding arguments with default values to a member function signature described in this clause; Hence, taking the address of a member function has an unspecified type. The same latitude does *not* extend to the implementation of virtual or global functions, however.

— by replacing a member function signature with default values by two or more member function signatures with equivalent behavior;

— by adding a member function signature for a member function name described in this clause.

2 A call to a member function signature described in this clause behaves the same as if the implementation declares no additional member function signatures.[102)]

17.3.4.5 Reentrancy [lib.reentrancy]

1 Which of the functions in the C++ Standard Library are not *reentrant subroutines* is implementation-defined.

17.3.4.6 Protection within classes [lib.protection.within.classes]

1 It is unspecified whether a member described in this clause as private is private, protected, or public. It is unspecified whether a member described as protected is protected or public.

Box 82

Does this make *any* sense?

A member described as public is always public.

2 It is unspecified whether a function signature or class described in this clause is a friend of another class described in this clause.

17.3.4.7 Derived classes [lib.derivation]

1 Certain classes defined in this clause are derived from other classes in the C++ Standard library:

— It is unspecified whether a class described in this clause as a base class is itself derived from other base classes (with names reserved to the implementation). *

— It is unspecified whether a class described in this clause as derived from another class is derived from that class directly, or through other classes (with names reserved to the implementation) that are derived from the specified base class.

2 In any case:

— A base class described as virtual in this clause is always virtual;

— A base class described as non-virtual in this clause is never virtual;

— Unless explicitly stated otherwise, types with distinct names in this clause are distinct types.[103)]

[102)] A valid C++ program always calls the expected library member function, or one with equivalent behavior. An implementation may also define additional member functions that would otherwise not be called by a valid C++ program.

[103)] An implicit exception to this rule are types described as synonyms for basic integral types, such as `size_t` and `streamoff`.

17.3.4.8 Restrictions on exception handling [lib.res.on.exception.handling]

1 Any of the functions defined in the C++ Standard library can report a failure to allocate storage by throwing an exception of type `bad_alloc`, or a class derived from `bad_alloc`.

2 Otherwise, none of the functions defined in the C++ Standard library throw an exception that must be caught outside the function, unless explicitly stated otherwise.

Box 83

ISSUE: aren't these two statements conveyed by exception specifications?

18 Language support library [lib.language.support]

1 This clause describes the function signatures that are called implicitly, and the types of objects generated implicitly, during the execution of some C++ programs. It also describes the headers that declare these function signatures and define any related types.

2 The following subclauses describe common type definitions used throughout the library, characteristics of the predefined types, functions supporting start and termination of a C++ program, support for dynamic memory management, support for dynamic type identification, support for exception processing, and other runtime support, as summarized in Table 25:

Table 25—Language support library summary

Subclause	Header(s)
18.1 Types	`<stddef>` `<cstddef>`

18.2 Implementation properties	`<limits>`
	`<climits>`
	`<cfloat>`

18.3 Start and termination	`<cstdlib>`
18.4 Dynamic memory management	`<new>`
18.5 Type identification	`<typeinfo>`
18.6 Exception handling	`<exception>`

18.7 Other runtime support	`<cstdarg>`
	`<csetjmp>`
	`<ctime>`
	`<csignal>`
	`<cstdlib>`

18.1 Types [lib.support.types]

1 Common definitons.

2 Header `<stddef>` (Table 26):

Table 26—Header `<stddef>` synopsis

Type	Name(s)
Types:	
`capacity`	`ptrdiff_t <stddef>` `wint_t <stddef>`
`fvoid_t`	`size_t <stddef>`

3 Header <cstddef> (Table 27):

Table 27—Header <cstddef> synopsis

Type	Name(s)	
Macros:	NULL <cstddef>	offsetof
Types:	ptrdiff_t<cstddef>	size_t <cstddef>

4 The header <stddef> defines a constant and several types used widely throughout the C++ Standard library. Some are also redundantly defined in C headers.

SEE ALSO: subclauses 5.3.3 and 12.5.

18.1.1 Values [lib.stddef.values]

```
const size_t NPOS = (size_t)(-1);
```

1 which is the largest representable value of type size_t.

18.1.2 Types [lib.stddef.types]

```
typedef void fvoid_t();
```

1 The type fvoid_t is a function type used to simplify the writing of several declarations in this clause.

Box 84

ISSUE: This type is not used anywhere.

```
typedef T ptrdiff_t;
```

2 The type ptrdiff_t is a synonym for *T*, the implementation-defined signed integral type of the result of subtracting two pointers.

Box 85

ISSUE: This introduces a needless redundancy with <cstddef>.

```
typedef T size_t;
```

3 The type size_t is a synonym for *T*, the implementation-defined unsigned integral type of the result of the sizeofoperator.

Box 86

ISSUE: This introduces a needless redundancy with <ctime>, <cstddef>, <cstdio>, and <cstring>.

```
typedef T wint_t;
```

4 The type `wint_t` is a synonym for *T*, the implementation-defined integral type, unchanged by integral promotions, that can hold any value of type `wchar_t` as well as at least one value that does not correspond to the code for any member of the extended character set.[104)]

Box 87

ISSUE: This introduces a needless redundancy with <cwchar> and <cwctype>.

SEE ALSO: subclause 5.3.3, Sizeof, subclause 5.7, Additive operators, and ISO C subclause 7.1.6.

18.2 Implementation properties [lib.support.limits]

1 Limits.

18.2.1 Numeric limits [lib.limits]

Header <limits> synopsis

```
namespace std {
  template<class T> class numeric_limits;
  enum float_rounds_style;

  class numeric_limits<bool>;
  class numeric_limits<char>;
  class numeric_limits<short>;
  class numeric_limits<int>;
  class numeric_limits<long>;
  class numeric_limits<float>;
  class numeric_limits<double>;
  class numeric_limits<long double>;
}
```

SEE ALSO: subclause 3.8.1.

18.2.1.1 Template class numeric_limits [lib.numeric.limits]

```
namespace std {
  template<class T> class numeric_limits {
  public:
    static const bool is_specialized = false;
    static T min();
    static T max();

    static const int  digits     = implementation-defined
    static const int  digits10   = implementation-defined
    static const bool is_signed  = implementation-defined
    static const bool is_integer = implementation-defined
    static const bool is_exact   = implementation-defined
    static const int  radix      = implementation-defined
    static T epsilon();
    static T round_error();
```

[104)] The extra value is denoted by the macro `WEOF`, defined in <cwchar>. It is permissible for `WEOF` to be in the range of values representable by `wchar_t`.

```
    static const int  min_exponent   = implementation-defined
    static const int  min_exponent10 = implementation-defined
    static const int  max_exponent   = implementation-defined
    static const int  max_exponent10 = implementation-defined

    static const bool has_infinity   = implementation-defined
    static const bool has_quiet_NaN  = implementation-defined
    static const bool has_signaling_NaN = implementation-defined
    static const bool has_denorm     = implementation-defined
    static T infinity();
    static T quiet_NaN();
    static T signaling_NaN();
    static T denorm_min();

    static const bool is_iec559      = implementation-defined
    static const bool is_bounded     = implementation-defined
    static const bool is_modulo      = implementation-defined

    static const bool traps          = implementation-defined
    static const bool tinyness_before= implementation-defined
    static const float_round_style round_style = implementation-defined
  };
}
```

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

2 The members `radix`, `epsilon()`, and `round_error()` shall have meaningful values for all floating point type specializations.

3 For types with `has_denorm == false`, the member `denorm_min()` shall return the same value as the member `min()`.

4 The default `numeric_limits<T>` template shall have all members, but with meaningless (0 or `false`) values.

18.2.1.2 `numeric_limits` members [lib.numeric.limits.members]

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

1 Minimum finite value.[105)]

2 For floating types with denormalization, returns the minimum normalized value, `denorm_min()`.

3 Meaningful for all specializations in which or`is_bounded == true`, `is_bounded == false && is_signed == false`.

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

4 Maximum finite value.[106)]

5 Meaningful for all specializations in which `is_bounded == true`.

```
static const int  digits;
```

105) Equivalent to CHAR_MIN, SHRT_MIN, FLT_MIN, DBL_MIN, etc.
106) Equivalent to CHAR_MAX, SHRT_MAX, FLT_MAX, DBL_MAX, etc.

6 Number of `radix` digits which can be represented without change.

7 For built-in integer types, he number of non-sign bits in the representation.

8 For floating point types, the number of `radix` digits in the mantissa.[107)]

```
static const int  digits10;
```

9 Number of base 10 digits which can be represented without change.[108)]

10 Meaningful for all specializations in which `is_bounded == true`.

```
static const bool is_signed;
```

11 True if the type is signed.

12 Meaningful for all specializations.

```
static const bool is_integer;
```

13 True if the type is integer.

14 Meaningful for all specializations.

```
static const bool is_exact;
```

15 True if the type uses an exact representation. All integer types are exact, but not vice versa. For example, rational and fixed-exponent representations are exact but not integer.

16 Meaningful for all specializations.

```
static const int  radix;
```

17 For floating types, specifies the base or radix of the exponent representation (often 2).[109)]

18 For integer types, specifies the base of the representation.[110)]

19 Meaningful for all specializations.

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

20 Machine epsilon: the difference between 1 and the least value greater than 1 that is representable.[111)]

21 Meaningful only for floating point types.

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

22 Measure of the maximum rounding error.[112)]

```
static const int  min_exponent;
```

---

107) Equivalent to FLT_MANT_DIG, DBL_MANT_DIG, LDBL_MANT_DIG.
108) Equivalent to FLT_DIG, DBL_DIG, LDBL_DIG.
109) Equivalent to FLT_RADIX.
110) Distinguishes types with bases other than 2 (e.g. BCD).
111) Equivalent to FLT_EPSILON, DBL_EPSILON, LDBL_EPSILON.
112) This has a precise definition in the Language Independent Arithmetic (LIA-1) standard. Required by LIA-1.

Minimum negative integer such that `radix` raised to that power is in range.[113)]
Meaningful only for floating point types.

```
static const int  min_exponent10;
```

Minimum negative integer such that 10 raised to that power is in range.[114)]
Meaningful only for floating point types.

```
static const int  max_exponent;
```

Maximum positive integer such that `radix` raised to that power is in range.[115)]
Meaningful only for floating point types.

```
static const int  max_exponent10;
```

Maximum positive integer such that 10 raised to that power is in range.[116)]
Meaningful only for floating point types.

```
static const bool has_infinity;
```

True if the type has a representation for positive infinity.
Meaningful only for floating point types.
Shall be `true` for all specializations in which `is_iec559 == true`.

```
static const bool has_quiet_NaN;
```

True if the type has a representation for a quiet (non-signaling) ‘‘Not a Number.’’[117)]
Meaningful only for floating point types.
Shall be `true` for all specializations in which `is_iec559 == true`.

```
static const bool has_signaling_NaN;
```

True if the type has a representation for a signaling ‘‘Not a Number.’’[118)]
Meaningful only for floating point types.
Shall be `true` for all specializations in which `is_iec559 == true`.

```
static const bool has_denorm;
```

True if the type allows denormalized values (variable number of exponent bits).[119)]

---

113) Equivalent to FLT_MIN_EXP, DBL_MIN_EXP, LDBL_MIN_EXP.
114) Equivalent to FLT_MIN_10_EXP, DBL_MIN_10_EXP, LDBL_MIN_10_EXP.
115) Equivalent to FLT_MAX_EXP, DBL_MAX_EXP, LDBL_MAX_EXP.
116) Equivalent to FLT_MAX_10_EXP, DBL_MAX_10_EXP, LDBL_MAX_10_EXP.
117) Required by LIA-1.
118) Required by LIA-1.
119) Required by LIA-1.

41 Meaningful only for flotaing point types.

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

42 Representation of positive infinity, if available.[120)]

43 Meaningful only in specializations for which f5has_infinity == true. Required in specializations for which `is_iec559 == true`1.

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

44 Representation of a quiet ‘‘Not a Number,’’ if available.[121)]

45 Meaningful only in specializations for which `has_quiet_NaN == true`. Required in specializations for which `is_iec559 == true`.

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

46 Representation of a signaling ‘‘Not a Number,’’ if available.[122)]

47 Meaningful only in specializations for which `has_signaling_NaN == true`. Required in specializations for which `is_iec559 == true`.

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

48 Minimum denormalized value.[123)]

49 Meaningful for all floating point types.

50 In specializations for which `has_denorm == false`, returns the minimum normalized value.

```
static const bool is_iec559;
```

51 True if and only if the type adheres to IEC 559 standard.[124)]

52 Meaningful only for floating point types.

```
static const bool is_bounded;
```

53 True if the set of values representable by the type is finite.[125)] All built-in types are bounded, this member would be false for arbitrary precision types.

54 Meaningful for all specializations.

```
static const bool is_modulo;
```

55 True if the type is modulo.[126)] A type is modulo if it is possible to add two positive numbers and have a result which wraps around to a third number which is less.

120) Required by LIA-1.
121) Required by LIA-1.
122) Required by LIA-1.
123) Required by LIA-1.
124) International Electrotechnical Commission standard 559 is the same as IEEE 754.
125) Required by LIA-1.
126) Required by LIA-1.

56 Generally, this is `false` for floating types, `true` for unsigned integers, and `true` for signed integers on most machines.

57 Meaningful for all specializations.

```
static const bool traps;
```

58 `true` if trapping is implemented for the type.[127)]

59 Meaningful for all specializations.

```
static const bool tinyness_before;
```

60 `true` if tinyness is detected before rounding.[128)]

61 Meaningful only for floating point types.

```
static const float_round_style round_style;
```

62 The rounding style for the type.[129)]

63 Meaningful for all floating point types. Specializations for integer types shall return `round_toward_zero`.

18.2.1.3 Type `float_round_style` [lib.round.style]

```
namespace std {
  enum float_round_style {
    round_indeterminate       = -1,
    round_toward_zero         =  0,
    round_to_nearest          =  1,
    round_toward_infinity     =  2,
    round_toward_neg_infinity =  3
  };
}
```

18.2.1.4 `numeric_limits` specializations [lib.numeric.special]

1 Specializations shall be provided for all scalar numeric types, both floating point and integer, including `bool`. The member `is_specialized` shall be `true` for all such specializations of `numeric_limits`.

2 Non-scalar types such as `complex<T>` shall not have specializations.

3 All members shall be provided for all specializations. However, many values are only required to be meaningful under certain conditions (for example, `epsilon()` is only meaningful if `is_integer` is `false`). Any value which is not ‘‘meaningful’’ must be set to 0 or `false`.

4 For example,

127) Required by LIA-1.
128) Refer to IEC 559. Required by LIA-1.
129) Equivalent to FLT_ROUNDS. Required by LIA-1.

`numeric_limits` specializations

```
    static const int  min_exponent   = implementation-defined
    static const int  min_exponent10 = implementation-defined
    static const int  max_exponent   = implementation-defined
    static const int  max_exponent10 = implementation-defined

    static const bool has_infinity   = implementation-defined
    static const bool has_quiet_NaN  = implementation-defined
    static const bool has_signaling_NaN = implementation-defined
    static const bool has_denorm     = implementation-defined
    static T infinity();
    static T quiet_NaN();
    static T signaling_NaN();
    static T denorm_min();

    static const bool is_iec559      = implementation-defined
    static const bool is_bounded     = implementation-defined
    static const bool is_modulo      = implementation-defined

    static const bool traps          = implementation-defined
    static const bool tinyness_before= implementation-defined
    static const float_round_style round_style = implementation-defined
  };
}
```

```
namespace std {
  class numeric_limits<float> {
  public:
    static const bool is_specialized = true;

    inline static float min() { return 1.17549435E-38F; }
    inline static float max() { return 3.40282347E+38F; }

    static const int digits   = 24;
    static const int digits10 =  6;

    static const bool is_signed  = true;
    static const bool is_integer = false;
    static const bool is_exact   = false;

    static const int radix = 2;
    inline static float epsilon()     { return 1.19209290E-07F; }
    inline static float round_error() { return 0.5F; }
```

18.2.2 C Library [lib.c.limits]

1 Header <climits> (Table 28):

Table 28—Header `<climits>` synopsis

Type	Name(s)				
Values:					
CHAR_BIT	INT_MAX	LONG_MIN	SCHAR_MIN	UCHAR_MAX	USHRT_MAX
CHAR_MAX	INT_MIN	MB_LEN_MAX	SHRT_MAX	UINT_MAX	
CHAR_MIN	LONG_MAX	SCHAR_MAX	SHRT_MIN	ULONG_MAX	

2 The contents are the same as the Standard C library.

3 Header `<cfloat>` (Table 29):

Table 29—Header `<cfloat>` synopsis

Type	Name(s)		
Values:			
DBL_DIG	DBL_MIN_DIG	FLT_MIN_10_EXP	LDBL_MAX_10_EXP
DBL_EPSILON	FLT_DIG	FLT_MIN_DIG	LDBL_MAX_EXP
DBL_MANT_DIG	FLT_EPSILON	FLT_RADIX	LDBL_MIN
DBL_MAX	FLT_MANT_DIG	FLT_ROUNDS	LDBL_MIN_10_EXP
DBL_MAX_10_EXP	FLT_MAX	LDBL_DIG	LDBL_MIN_DIG
DBL_MAX_EXP	FLT_MAX_10_EXP	LDBL_EPSILON	
DBL_MIN	FLT_MAX_EXP	LDBL_MANT_DIG	
DBL_MIN_10_EXP	FLT_MIN	LDBL_MAX	

4 The contents are the same as the Standard C library.

SEE ALSO: ISO C subclause 7.1.5, 5.2.4.2.2, 5.2.4.2.1.

18.3 Start and termination [lib.support.start.term]

1 Header `<cstdlib>` (partial), Table 30:

Table 30—Header `<cstdlib>` synopsis

Type	Name(s)		
Macros:	EXIT_FAILURE		EXIT_SUCCESS
Functions:	abort	atexit	exit

2 The contents are the same as the Standard C library, with the following changes:

18.3.1 `atexit` [lib.atexit]

`atexit(void (*`*`f`*`)(void))`

1 The function `atexit`, has additional behavior in this International Standard:

— For the execution of a function registered with `atexit`, if control leaves the function because it provides no handler for a thrown exception, `terminate()` is called (18.6.1.3).

18.3.2 exit [lib.exit]

`exit(int status)`

1 The function `exit` has additional behavior in this International Standard:

— First, all functions *f* registered by calling `atexit(f)`, are called, in the reverse order of their registration.[130)]

— Next, all static objects are destroyed in the reverse order of their construction. (Automatic objects are not destroyed as a result of calling)[131)]`exit`.

— Next, all open C streams (as mediated by the function signatures declared in `<cstdio>`) with unwritten buffered data are flushed, all open C streams are closed, and all files created by calling `tmpfile()` are removed.[132)]

— Finally, control is returned to the host environment. If *status* is zero or `EXIT_SUCCESS`, an implementation-defined form of the status `successful termination` is returned. If *status* is `EXIT_FAILURE`, an implementation-defined form of the status `unsuccessful termination` is returned. Otherwise the status returned is implementation-defined.[133)]

2 The function `exit` never returns to its caller.

SEE ALSO: subclauses 3.6, 3.6.3, ISO C subclause 7.10.4.

18.4 Dynamic memory management [lib.support.dynamic]

1 The header `<new>` defines several functions that manage the allocation of dynamic storage in a program. It also defines components for reporting storage management errors.

Header `<new>` synopsis

```
#include <cstdlib>      // for size_t
#include <stdexcept>    // for runtime_error
namespace std {
  void* operator new(size_t size);
  void  operator delete(void* ptr);
  void* operator new[](size_t size);
  void  operator delete[](void* ptr);
  void* operator new  (size_t size, void* ptr);
  void* operator new[](size_t size, void* ptr);
  class bad_alloc;
  typedef void (*new_handler)();
  new_handler set_new_handler(new_handler new_p);
}
```

SEE ALSO: subclauses 1.5, 3.7.3, 5.3.4, 5.3.5, 12.5, subclause 20.4, Memory.

130) A function is called for every time it is registered. The function signature `atexit(void (*)())`, is declared in `<cstdlib>`.

131) Automatic objects are all destroyed in a program whose function `main` contains no automatic objects and executes the call to `exit`. Control can be transferred directly to such a `main` by throwing an exception that is caught in `main`.

132) Any C streams associated with `cin`, `cout`, etc (_lib.header.iostream_) are flushed and closed when static objects are destroyed in the previous phase. The function signature `tmpfile()` is declared in `<cstdio>`.

133) The macros `EXIT_FAILURE` and `EXIT_SUCCESS` are defined in `<cstdlib>`.

18.4.1 Storage allocation and deallocation [lib.new.delete]

18.4.1.1 Single-object forms [lib.new.delete.single]

18.4.1.1.1 `operator new` [lib.op.new]

```
void* operator new(size_t size);
```

1 The *allocation function* (3.7.3.1) called by a *new-expression* (5.3.4) to allocate *`size`* bytes of storage suitably aligned to represent any object of that size.

Notes: Replaceable: a C++ program may define a function with this function signature that displaces the default version defined by the C++ Standard library.

Required behavior: Return a pointer to dynamically allocated storage (3.7.3).

Default behavior: Executes a loop:

— Within the loop, the function first attempts to allocate the requested storage. Whether the attempt involves a call to the Standard C library function `malloc` is unspecified.

— Returns a pointer to the allocated storage if the attempt is successful. Otherwise, if the last argument to *`set_new_handler()`* was a null pointer, the result is implementation-defined.[134)]

— Otherwise, the function calls the current *`new_handler`* (_lib.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 *`new_handler`* function does not return.

18.4.1.1.2 `operator delete` [lib.op.delete]

```
void operator delete(void* ptr);
```

1 The *deallocation function* (3.7.3.2) called by a *delete-expression* to render the value of *`ptr`* invalid.

Notes: Replaceable: a C++ program may define a function with this function signature that displaces the default version defined by the C++ Standard library.

Required behavior: accept a value of *`ptr`* that is null or that was returned by an earlier call to `operator new(size_t)`.

Default behavior:

— For a null value of *`ptr`*, do nothing.

— Any other value of *`ptr`* shall be a value returned earlier by a call to the default [135)]`operator new(size_t)`. For such a non-null value of *`ptr`*, reclaims storage allocated by the earlier call to the default `operator new(size_t)`.

2 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 any of `calloc`, `malloc`, or `realloc`, declared in `<cstdlib>`.

[134)] A common extension when *`new_handler`* is a null pointer is for `operator new(size_t)` to return a null pointer, in accordance with many earlier implementations of C++.

[135)] The value must not have been invalidated by an intervening call to `operator delete(size_t)`, or it would be an invalid argument for a C++ Standard library function call.

18.4.1.2 Array forms [lib.new.delete.array]

18.4.1.3 `operator new[]` [lib.op.new.array]

```
void* operator new[](size_t size);
```

1 The *allocation function* (3.7.3.1) called by the array form of a *new-expression* (5.3.4) to allocate *size* bytes of storage suitably aligned to represent any array object of that size or smaller.[136)]

Notes: Replaceable: a C++ program can define a function with this function signature that displaces the default version defined by the C++ Standard library.

Required behavior: Same as for `operator new(size_t)`.

Default behavior: Returns `operator new(`*size*`)`.

18.4.1.4 `operator delete[]` [lib.op.delete.array]

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

1 The *deallocation function* (3.7.3.2) called by the array form of a *delete-expression* to render the value of *ptr* invalid.

Notes: Replaceable: a C++ program can define a function with this function signature that displaces the default version defined by the C++ Standard library.

Required behavior: accept a value of *ptr* that is null or that was returned by an earlier call to `operator new[](size_t)`.

Default behavior:

— For a null value of *ptr*, does nothing.

— Any other value of *ptr* shall be a value returned earlier by a call to the default [137)]`operator new[](size_t)`. For such a non-null value of *ptr*, reclaims storage allocated by the earlier call to the default `operator new[](size_t)`.

2 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 any of `calloc`, `malloc`, or `realloc`, declared in `<cstdlib>`.

18.4.1.5 Placement forms [lib.new.delete.placement]

1 These functions are reserved, a C++ program may not define functions that displace the versions in the Standard C++ library.

18.4.1.5.1 Placement `operator new` [lib.placement.op.new]

```
void* operator new(size_t size, void* ptr);
```

Returns: *ptr*.

---

136) It is not the direct responsibility of `operator new[](size_t)` or `operator delete[](void*)` to note the repetition count or element size of the array. Those operations are performed elsewhere in the array `new` and `delete`expressions. The array `new` expression, may, however, increase the *size* argument to `operator new[](size_t)` to obtain space to store supplemental information.

137) The value must not have been invalidated by an intervening call to `operator delete[](size_t)`, or it would be an invalid argument for a C++ Standard library function call.

18.4.1.5.2 Placement `operator new[]` [lib.placement.op.new.array]

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

Returns: *ptr*.

18.4.2 Storage allocation errors [lib.alloc.errors]

18.4.2.1 Class `bad_alloc` [lib.bad.alloc]

```
namespace std {
  class bad_alloc : public runtime_error {
  public:
    bad_alloc();
    virtual ~bad_alloc();
    virtual string what() const;
  private:
//  static string alloc_msg;     exposition only
  };
}
```

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

Box 88

For the sake of exposition, the maintained data is presented here as:

— `static string alloc_msg`, an object of type `string` whose value is intended to briefly describe an allocation failure, initialized to an unspecified value.

18.4.2.1.1 `bad_alloc` constructor [lib.cons.bad.alloc]

```
bad_alloc();
```

Effects: Constructs an object of class `alloc`, initializing the base class with `runtime_error()`.

18.4.2.1.2 `bad_alloc::what` [lib.bad.alloc::what]

```
virtual string what() const;
```

Returns: An implementation-defined value.[138)]

18.4.2.2 Type `new_handler` [lib.new.handler]

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

1 The type of a *handler function* to be called by `operator new()` or `operator new[]()` when they cannot satisfy a request for addtional storage.

Required behavior: A *new_handler* shall perform one of the following:

— make more storage available for allocation and then return;

— throw an exception of type `bad_alloc` or a class derived from `bad_alloc`;

— call either `abort()` or `exit()`;

[138)] A possible return value is `&alloc_msg`.

Default behavior: The implementation's default *new_handler* throws an exception of type bad_alloc.

18.4.2.3 set_new_handler [lib.set.new.handler]

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

Effects: Establishes the function designated by *new_p* as the current *new_handler*.

1 Returns the previous *new_handler*.

18.5 Type identification [lib.support.rtti]

1 The header <typeinfo> defines two types associated with type information generated by the implementation. It also defines two types for reporting dynamic type identification errors.

Header <typeinfo> synopsis

```
#include <stdexcept>    // for logic_error
namespace std {
  class type_info;
  class bad_cast;
  class bad_typeid;
}
```

SEE ALSO: subclauses 5.2.6, 5.2.7.

18.5.1 Type information [lib.rtti]

18.5.1.1 Class type_info [lib.type.info]

```
namespace std {
  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);
//  const char* name;    exposition only
//  T desc;              exposition only
  };
}
```

1 The class type_info describes type information generated 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, encoding rule, and collating sequence for types are all unspecified and may differ between programs.

Box 89

For the sake of exposition, the stored objects are presented here as:

— const char* *name*, points at a static NTMBS;

— *T desc*, an object of a type *T* that has distinct values for all the distinct types in the program, stores the value corresponding to *name*.

18.5.1.1.1 `type_info` comparisons [lib.type.info.compare]

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

Effects: Compares the value stored in *`desc`* with *`rhs.desc`*.
Returns: `true` if the two values represent the same type.

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

Returns: `!(*this == rhs)`.

18.5.1.1.2 `type_info::before` [lib.type.info::before]

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

Effects: Compares the value stored in *`desc`* with *`rhs.desc`*.
Returns: `true` if `*this` precedes *`rhs`* in the collation order.

18.5.1.1.3 `type_info::name` [lib.type.info::name]

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

Returns: *`name`*.

18.5.1.1.4 Copying and assignment [lib.cons.type.info]

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

Effects: Constructs an object of class `type_info` and initializes *`name`* to *`rhs.name`* and *`desc`* to *`rhs.desc`*.[139)]

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

Effects: Assigns *`rhs.name`* to *`name`* and *`rhs.desc`* to *`desc`*.
Returns: `*this`.

18.5.2 Type identification errors [lib.rtti.errors]

18.5.2.1 Class `bad_cast` [lib.bad.cast]

```
namespace std {
  class bad_cast : public logic_error {
  public:
    bad_cast(const string& what_arg);
    virtual ~bad_cast();
//  virtual string what() const;        inherited
  };
}
```

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

[139)] Since the copy constructor and assignment operator for `type_info` are private to the class, objects of this type cannot be copied, but objects of derived classes possibly can be.

18.5.2.1.1 `bad_cast` constructor [lib.cons.bad.cast]

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

Effects: Constructs an object of class `bad_cast`, initializing the base class with `logic_error(what_arg)`.

18.5.2.2 Class `bad_typeid` [lib.bad.typeid]

```
namespace std {
  class bad_typeid : public logic_error {
  public:
    bad_typeid();
    virtual ~bad_typeid();
  };
}
```

1 The class `bad_typeid` defines the type of objects thrown as exceptions by the implementation to report a null pointer *p* in an expression of the form `typeid (*p)`.

18.5.2.2.1 `bad_typeid` constructor [lib.cons.bad.type.id]

```
bad_typeid();
```

Effects: Constructs an object of class `bad_typeid`, initializing the base class `logic_error` with an unspecified constructor.

18.6 Exception handling [lib.support.exception]

1 The header `<exception>` defines several types and functions related to the handling of exceptions in a C++ program.

Header `<exception>` synopsis

```
namespace std {
  class XUNEXPECTED;
  typedef void (*terminate_handler)();
  terminate_handler set_terminate(terminate_handler f);
  void terminate();
  typedef void (*unexpected_handler)();
  unexpected_handler set_unexpected(unexpected_handler f);
  void unexpected();
}
```

SEE ALSO: subclause 15.5.

18.6.1 Abnormal termination [lib.exception.terminate]

18.6.1.1 Type `terminate_handler` [lib.terminate.handler]

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

1 The type of a *handler function* to be called by `terminate()` when terminating exception processing.
Required behavior: A *terminate_handler* shall terminate execution of the program without returning to the caller.
Default behavior: The implementation's default *terminate_handler* calls `abort()`.

18.6.1.2 `set_terminate` [lib.set.terminate]

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

Effects: Establishes the function designated by *f* as the current handler function for terminating exception processing.
Requires: *f* shall not be a null pointer.
Returns: The previous *terminate_handler*.

18.6.1.3 `terminate` [lib.terminate]

```
void terminate();
```

1 Called by the implementation when exception handling must be abandoned for any of several reasons (15.5.1).
Effects: Calls the current *terminate_handler* handler function (_lib.terminate.handler_).

18.6.2 Violating *exception-specifications* [lib.exception.unexpected]

18.6.2.1 Class `XUNEXPECTED` [lib.xunexpected]

Box 90
TO BE SPECIFIED

18.6.2.2 Type `unexpected_handler` [lib.unexpected.handler]

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

1 The type of a *handler function* to be called by `unexpected()` when a function attempts to throw an exception not listed in its *exception-specification*.
Required behavior: an *unexpected_handler* shall either throw an exception or terminate execution of the program without returning to the caller. An *unexpected_handler* may perform any of the following:

— throw an exception that satisfies the exception specification;

— throw an ***XUNEXPECTED*** exception;

— call `terminate()`;

— call either `abort()` or `exit()`;

Default behavior: The implementation's default *unexpected_handler* calls `terminate()`.

18.6.2.3 `set_unexpected` [lib.set.unexpected]

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

Effects: Establishes the function designated by *f* as the current *unexpected_handler*.
Requires: *f* shall not be a null pointer.
Returns: The previous *unexpected_handler*.

18.6.2.4 unexpected [lib.unexpected]

```
void unexpected();
```

1 Called by the implementation when a function with an *exception-specification* throws an exception that is not listed in the *exception-specification* (15.5.2).
Effects: Calls the current *unexpected_handler* handler function (_lib.unexpected.handler_).

18.7 Other runtime support [lib.support.runtime]

1 Headers <cstdarg> (variable arguments), <csetjmp> (nonlocal jumps), <ctime> (system clock clock(), time()), <csignal> (signal handling), and <cstdlib> (runtime environment getenv(), system()).

Table 30—Header <cstdarg> synopsis

Type	Name(s)		
Macros:	va_arg	va_end	va_start
Type:	va_list		

Table 30—Header <csetjmp> synopsis

Type	Name(s)
Macro:	setjmp
Type:	jmp_buf
Function:	longjmp

Table 30—Header <ctime> synopsis

Type	Name(s)
Macros:	CLOCKS_PER_SEC
Types:	clock_t
Functions:	clock

Table 30—Header <csignal> synopsis

Type	Name(s)			
Macros:	SIGABRT	SIGILL	SIGSEGV	SIG_DFL
SIG_IGN	SIGFPE	SIGINT	SIGTERM	SIG_ERR
Type:	sig_atomic_t			
Functions:	raise	signal		

Table 30—Header `<cstdlib>` synopsis

Type	Name(s)
Functions:	`getenv` `system`

2 The contents are the same as the Standard C library, with the following changes:

3 The function signature `longjmp((jmp_buf` *`jbuf`*`, int` *`val`*`))` has more restricted behavior in this International Standard. If any automatic objects would be destroyed by a thrown exception transferring control to another (destination) point in the program, then a call to `longjmp(`*`jbuf`*`,` *`val`*`)` at the throw point that transfers control to the same (destination) point has undefined behavior.

SEE ALSO: ISO C subclause 7.10.4, 7.8, 7.6, 7.12.

19 Diagnostics library [lib.diagnostics]

1 This clause describes components that C++ programs may use to detect and report error conditions.

2 The following subclauses describe components for reporting several kinds of exceptional conditions, documenting program assertions, and a global variable for error number codes, as summarized in Table 31:

Table 31—Diagnostics library summary

Subclause	Header(s)
19.1 Exception classes	`<stdexcept>`
19.2 Assertions	`<cassert>`
19.3 Error numbers	`<cerrno>`

19.1 Exception classes [lib.std.exceptions]

1 The Standard C++ library provides classes to be used to report errors in C++ programs. In the error model reflected in these classes, errors are divided into two broad categories: *logic* errors and *runtime* errors.

2 The distinguishing characteristic of logic errors is that they are due to errors in the internal logic of the program. In theory, they are preventable.

3 By contrast, runtime errors are due to events beyond the scope of the program. They cannot be easily predicted in advance. The header `<stdexcept>` defines several types of predefined exceptions for reporting errors in a C++ program. These exceptions are related via inheritance.

Header `<stdexcept>` synopsis

```
#include <string>
namespace std {
  class exception;
    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;
}
```

19.1.1 Class `exception` [lib.exception]

```
namespace std {
  class exception {
  public:
    exception(const string& what_arg);
    virtual ~exception();
    virtual string what() const;
  protected:
    exception();
  private:
//  const string* desc; exposition only
//  bool alloced;                exposition only
  };
}
```

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

Box 91

For the sake of exposition, the stored data is presented here as:

— `const string* what`, stores a null pointer or points to an object of type `string` whose value is intended to briefly describe the general nature of the exception thrown;

— `bool alloced`, stores a nonzero value if the string object `what` has been allocated by the object of class `exception`.

19.1.1.1 `exception` constructors [lib.exception.cons]

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

Effects: Constructs an object of class `exception` and initializes `desc` to `&string(what_arg)` and `alloced` to a nonzero value.

```
exception();
```

Effects: Constructs an object of class `exception` and initializes `desc` to an unspecified value and `alloced` to zero.[140)]

19.1.1.2 `exception` destructor [lib.exception.des]

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

Effects: Destroys an object of class `exception`. If `alloced` is nonzero, the function frees any object pointed to by `what`.

19.1.1.3 `exception::what` [lib.exception::what]

```
virtual string what() const;
```

Returns: `string(desc)` if `desc` is not a null pointer. Otherwise, the value returned is implementation defined.

140) This protected default constructor for `exception` can, and should, avoid allocating any additional storage.

19.1.2 Class `logic_error` [lib.logic.error]

```
namespace std {
  class logic_error : public exception {
  public:
    logic_error(const string& what_arg);
    virtual ~logic_error();
//  virtual string what() const;        inherited
  };
}
```

1 The class `logic_error` defines the type of objects thrown as exceptions by the implementation to report errors presumably detectable before the program executes, such as violations of logical preconditions or class invariants.

19.1.2.1 `logic_error` constructor [lib.logic.error.cons]

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

Effects: Constructs an object of class `logic_error`, initializing the base class with `exception(`*`what_arg`*`)`.

19.1.3 Class `domain_error` [lib.domain.error]

```
namespace std {
  class domain_error : public logic_error {
  public:
    domain_error(const string& what_arg);
    virtual ~domain_error();
//  virtual string what() const;        inherited
  };
}
```

1 The class `domain_error` defines the type of objects thrown as exceptions by the implementation to report domain errors.

19.1.3.1 `domain_error` constructor [lib.domain.error.cons]

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

Effects: Constructs an object of class `domain`, initializing the base class with `logic_error(`*`what_arg`*`)`.

19.1.4 Class `invalid_argument` [lib.invalid.argument]

```
namespace std {
  class invalid_argument : public logic_error {
  public:
    invalid_argument(const string& what_arg);
    virtual ~invalid_argument();
//  virtual string what() const;        inherited
  };
}
```

1 The class `invalid_argument` defines the base class for the types of all objects thrown as exceptions, by functions in the Standard C++ library, to report an invalid argument.

19.1.4.1 `invalid_argument` constructor [lib.invalid.argument.cons]

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

Effects: Constructs an object of class `invalid_argument`, initializing the base class with `logic_error(what_arg)`.

19.1.5 Class `length_error` [lib.length.error]

```
namespace std {
  class length_error : public logic_error {
  public:
    length_error(const string& what_arg);
    virtual ~length_error();
//  virtual string what() const;        inherited
  };
}
```

1 The class `length_error` defines the base class for the types of all objects thrown as exceptions, by functions in the Standard C++ library, to report an attempt to produce an object whose length equals or exceeds its maximum allowable size.

19.1.5.1 `length_error` constructor [lib.length.error.cons]

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

Effects: Constructs an object of class `length_error`, initializing the base class with `logic_error(what_arg)`.

19.1.6 Class `out_of_range` [lib.out.of.range]

```
namespace std {
  class out_of_range : public logic_error {
  public:
    out_of_range(const string& what_arg);
    virtual ~out_of_range();
//  virtual string what() const;        inherited
  };
}
```

1 The class `out_of_range` defines the base class for the types of all objects thrown as exceptions, by functions in the Standard C++ library, to report an out-of-range argument.

19.1.6.1 `out_of_range` constructor [lib.out.of.range.cons]

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

Effects: Constructs an object of class `out_of_range`, initializing the base class with `logic_error(what_arg)`.

19.1.7 Class `runtime_error` [lib.runtime.error]

```
namespace std {
  class runtime_error : public exception {
  public:
    runtime_error(const string& what_arg);
    virtual ~runtime_error();
//  virtual string what();      inherited
  protected:
    runtime_error();
  };
}
```

1 The class `runtime_error` defines the type of objects thrown as exceptions by the implementation to report errors presumably detectable only when the program executes.

19.1.7.1 `runtime_error` constructors [lib.runtime.error.cons]

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

Effects: Constructs an object of class `runtime`, initializing the base class with `exception(what_arg)`.

```
runtime_error();
```

Effects: Constructs an object of class `runtime`, initializing the base class with `exception()`.

19.1.8 Class `range_error` [lib.range.error]

```
namespae std {
  class range_error : public runtime_error {
  public:
    range_error(const string& what_arg);
    virtual ~range_error();
//  virtual string what() const;        inherited
  };
}
```

1 The class `range_error` defines the type of objects thrown as exceptions by the implementation to report range errors.

19.1.8.1 `range_error` constructor [lib.range.error.cons]

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

Effects: Constructs an object of class `range_error`, initializing the base class with `runtime_error(what_arg)`.

19.1.9 Class `overflow_error` [lib.overflow.error]

```
namespace std {
  class overflow_error : public runtime_error {
  public:
    overflow_error(const string& what_arg);
    virtual ~overflow_error();
//  virtual string what() const;        inherited
  };
}
```

1 The class `overflow_error` defines the base class for the types of all objects thrown as exceptions, by functions in the Standard C++ library, to report an arithmetic overflow error.

19.1.9.1 `overflow_error` constructor [lib.overflow.error.cons]

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

Effects: Constructs an object of class `overflow_error`, initializing the base class with `runtime_error(what_arg)`.

19.2 Assertions [lib.assertions]

1 Provides macros for documenting C++ program assertions, and for disabling the assertion checks.

2 Header `<cassert>` (Table 32):

Table 32—Header `<cassert>` synopsis

Type	Name(s)	
Macros:	`assert`	`NDEBUG`

3 The contents are the same as the Standard C library.

SEE ALSO: ISO C subclause 7.2.

19.3 Error numbers [lib.errno]

1 Header `<cerrno>` (Table 33):

Table 33—Header `<cerrno>` synopsis

Type	Name(s)	
Macros:	`EDOM`	`ERANGE`
Object:	`errno`	

2 The contents are the same as the Standard C library.

SEE ALSO: ISO C subclause 7.1.4, 7.2, Amendment 1 subclause 4.3.

20 General utilities library [lib.utilities]

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

2 The following subclauses describe alocator requirements, utility components, function objects, dynamic memory management utilities, and date/time utilities, as summarized in Table 34:

Table 34—General utilities library summary

Subclause	Header(s)
20.1 Allocator requirements	
20.2 Utility components	`<utility>`
20.3 Function objects	`<functional>`
20.4 Memory	`<memory>`
20.5 Date and time	`<ctime>`

20.1 Allocator requirements [lib.allocator.requirements]

1 One of the common problems in portability is to be able to encapsulate the information about the memory model. This information includes the knowledge of pointer types, the type of their difference, the type of the size of objects in this memory model, as well as the memory allocation and deallocation primitives for it.

2 The library addresses this problem by providing a standard set of requirements for *allocators*, which are objects that encapsulate this information. All of the containers are parameterized in terms of allocators. That dramatically simplifies the task of dealing with multiple memory models.

3 In the following Table 35, we assume `X` is an allocator class for objects of type `T`, `a` is a value of `X`, `n` is of type `X::size_type`, `p` is of type `X::pointer` which was obtained from `X`.

4 All the operations on the allocators are expected to be amortized constant time.

Table 35—Allocator requirements

expression	return type	assertion/note pre/post-condition
`X::value_type`	`T`	
`X::pointer`	pointer to `T`	the result of `operator*` of values of `X::pointer` is of `value_type`.
`X::const_pointer`	pointer to `const T` type	the result of `operator*` of values of `X::const_pointer` is of `const value_type`; it is the same type of pointer as `X::pointer`, in particular, `sizeof(X::const_pointer) == sizeof(X::pointer)`.
`X::size_type`	unsigned integral type	the type that can represent the size of the largest object in the memory model.
`X::difference_type`	signed integral type	the type that can represent the difference between any two pointers in the memory model.
`X a;`		note: a destructor is assumed.
`a.allocate(n)`	`X::pointer`	memory is allocated for `n` objects of type `T` but objects are not constructed. `allocate` may raise an appropriate exception.
`a.deallocate(p)`	result is not used	all the objects in the area pointed by `p` should be destroyed prior to the call of the deallocate.
`a.init_page_size()`	`X::size_type`	the returned value is the optimal value for an initial buffer size of the given type. It is assumed that if `k` is returned by `init_page_size`, `t` is the construction time for `T`, and `u` is the time that it takes to do `allocate(k)`, then `k * t` is much greater than `u`.
`a.max_size()`	`X::size_type`	the size of the largest buffer that can be allocated

20.2 Utility components [lib.utility]

1 This subclause contains some basic template functions and classes that are used throughout the rest of the library.

Header <utility> synopsis

```
namespace std {
// subclause 20.2.1, operators:
  template<class T> bool operator!=(const T&, const T&);
  template<class T> bool operator> (const T&, const T&);
  template<class T> bool operator<=(const T&, const T&);
  template<class T> bool operator>=(const T&, const T&);

// subclause 20.2.2, tuples:
  struct empty;
  bool operator==(const empty&, const empty&);
  bool operator< (const empty&, const empty&);
  template <class T1, class T2> struct pair;
  template <class T1, class T2>
    bool operator==(const pair<T1,T2>&, const pair<T1,T2>&);
  template <class T1, class T2>
    bool operator< (const pair<T1,T2>&, const pair<T1,T2>&);
  template <class T1, class T2> pair<T1,T2> make_pair(const T1&, const T2&);

// subclause 20.2.3, restrictor:
  template <class T> class restrictor;
  template <class T>
    bool operator==(const restrictor<T>&, const restrictor<T>&);
  template <class T>
    bool operator< (const restrictor<T>&, const restrictor<T>&);
}
```

20.2.1 Operators [lib.operators]

1 To avoid redundant definitions of `operator!=` out of `operator==` and operators `>`, `<=`, and `>=` out of `operator<`, the library provides the following:

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

Returns: `!(x == y)`.

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

Returns: `y < x`.

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

Returns: `!(y < x)`.

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

Returns: `!(x < y)`.

20.2.2 Tuples [lib.tuples]

1 The library includes templates for heterogeneous n-tuples for n equal to 0 and 2. For non-empty tuples the library provides matching template functions to simplify their construction.[141)]

[141)] Users and library venders can provide additional *n*-tuples for *n* equal to 1 (singleton) and *n* greater than 2. For example, triples and quadruples may be defined.

2 For example, instead of saying,

```
return pair<int, double>(5, 3.1415926); // explicit types
```

3 one may say

```
return make_pair(5, 3.1415926); // types are deduced
```

20.2.2.1 Empty [lib.empty]

1 The class `empty` is used as a base class where only `==` and `<` are needed.

```
struct empty {};

bool operator==(const empty&, const empty&);
```

Returns: `true`.

```
bool operator< (const empty&, const empty&);
```

Returns: `false`.

20.2.2.2 Pair [lib.pair]

```
template <class T1, class T2>
struct pair {
  T1 first;
  T2 second;
  pair(const T1& x, const T2& y);
};
```

1 The constructor initializes `first` with *x* and `second` with *y*.

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

Returns: `x.first == y.first && x.second == y.second`.

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

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

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

Returns: `pair<T1, T2>(x, y)`.

20.2.3 Restrictor [lib.restrictor]

1 Restrictor is a template class that hides a value of any type and restricts the available operations to equality and less than (if they are provided for the type).

```
namespace std {
  template <class T>
  class restrictor {
  protected:
    T value;
  public:
    restrictor(const T& x) : value(x) {}
  };

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

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

Returns: `x.value == y.value`.

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

Returns: `x.value < y.value`.

20.3 Function objects [lib.function.objects]

Header `<functional>` synopsis

```
namespace std {
// subclause 20.3.1, base:
  template <class Arg, class Result>
    struct unary_function;
  template <class Arg1, class Arg2, class Result>
    struct binary_function;

// subclause 20.3.2, arithmetic operations:
  template <class T> struct plus;
  template <class T> struct minus;
  template <class T> struct times;
  template <class T> struct divides;
  template <class T> struct modulus;
  template <class T> struct negate;

// subclause 20.3.3, comparisons:
  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;

// subclause 20.3.4, logical operations:
  template <class T> struct logical_and;
  template <class T> struct logical_or;
  template <class T> struct logical_not;
```

```
// subclause 20.3.5, negators:
  template <class Predicate> struct unary_negate;
  template <class Predicate>
    unary_negate<Predicate>  not1(const Predicate&);
  template <class Predicate> struct binary_negate;
  template <class Predicate>
    binary_negate<Predicate> not2(const Predicate&);

// subclause 20.3.6, binders:
  template <class Operation>  struct binder1st;
  template <class Operation, class T>
    binder1st<Operation> bind1st(const Operation&, const T&);
  template <class Operation> class binder2nd;
  template <class Operation, class T>
    binder2nd<Operation> bind2nd(const Operation&, const T&);

// subclause 20.3.7, adaptors:
  template <class Arg, class Result> class pointer_to_unary_function;
  template <class Arg, class Result>
    pointer_to_unary_function<Arg,Result> ptr_fun(Result (*)(Arg));
  template <class Arg1, class Arg2, class Result>
    class pointer_to_binary_function;
  template <class Arg1, class Arg2, class Result>
    pointer_to_binary_function<Arg1,Arg2,Result> ptr_fun(Result (*)(Arg1,Arg2));
}
```

1 Function objects are objects with an `operator()` defined. They are important for the effective use of the library. In the places where one would expect to pass a pointer to a function to an algorithmic template, the interface is specified to accept an object with an `operator()` defined. This not only makes algorithmic templates work with pointers to functions, but also enables them to work with arbitrary function objects. Using function objects together with function templates increases the expressive power of the library as well as making the resulting code much more efficient. For example, if we want to have a by-element addition of two vectors `a` and `b` containing `double` and put the result into `a` we can do:

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

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

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

3 The corresponding functions will inline the addition and the negation.

4 To enable adaptors and other components to manipulate function objects that take one or two arguments it is required that they correspondingly provide typedefs `argument_type` and `result_type` for function objects that take one argument and `first_argument_type`, `second_argument_type`, and `result_type` for function objects that take two arguments.

20.3.1 Base [lib.base]

1 The following classes are provided to simplify the typedefs of the argument and result types:

```
template <class Arg, class Result>
struct unary_function : empty {
  typedef Arg    argument_type;
  typedef Result result_type;
};
```

```
template <class Arg1, class Arg2, class Result>
struct binary_function : empty {
  typedef Arg1   first_argument_type;
  typedef Arg2   second_argument_type;
  typedef Result result_type;
};
```

20.3.2 Arithmetic operations [lib.arithmetic.operations]

1 The library provides basic function object classes for all of the arithmetic operators in the language.

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

2 `operator()` returns `x + y`.

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

3 `operator()` returns `x - y`.

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

4 `operator()` returns `x * y`.

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

5 `operator()` returns `x / y`.

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

6 `operator()` returns `x % y`.

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

7 `operator()` returns `-x`.

20.3.3 Comparisons [lib.comparisons]

1 The library provides basic function object classes for all of the comparison operators in the language.

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

2 `operator()` returns `x == y`.

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

3 `operator()` returns `x != y`.

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

4 `operator()` returns `x > y`.

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

5 `operator()` returns `x < y`.

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

6 `operator()` returns `x >= y`.

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

7 `operator()` returns `x <= y`.

20.3.4 Logical operations [lib.logical.operations]

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

1 `operator()` returns `x && y`.

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

2 `operator()` returns `x || y`.

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

3 `operator()` returns `!x`.

20.3.5 Negators [lib.negators]

1 Negators `not1` and `not2` take a unary and a binary predicate, respectively, and return their complements.

```
template <class Predicate>
  class unary_negate : public unary_function<Predicate::argument_type,bool>,
                       restrictor<Predicate> {
public:
  unary_negate(const Predicate& x);
  bool operator()(const argument_type& x) const;
};
```

Returns: `!value(x)`.

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

Returns: `unary_negate<Predicate>(pred)`.

```
  template <class Predicate>
    class binary_negate
      : public binary_function<Predicate::first_argument_type,
                               Predicate::second_argument_type, bool>,
        restrictor<Predicate> {
    public:
        binary_negate(const Predicate& x);
        bool operator()(const first_argument_type&  x,
                        const second_argument_type& y) const;
    };
```

2 `operator()` returns `!value(x, y)`.

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

Returns: `binary_negate<Predicate>(pred)`.

20.3.6 Binders [lib.binders]

1 Binders `bind1st` and `bind2nd` take a function object `f` of two arguments and a value `x` and return a function object of one argument constructed out of `f` with the first or second argument correspondingly bound to `x`.

20.3.6.1 Template class `binder1st` [lib.binder.1st]

```
    template <class Operation>
    class binder1st
      : public unary_function<Operation::second_argument_type,
                              Operation::result_type> {
    protected:
        Operation     op;
        argument_type value;

    public:
        binder1st(const Operation& x, const Operation::first_argument_type& y);
        result_type operator()(const argument_type& x) const;
    };
```

1 The constructor initializes `op` with *x* and `value` with *y*.

2 `operator()` returns `op(value,x)`.

20.3.6.2 `bind1st` [lib.bind.1st]

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

Returns: `binder1st<Operation>(op, Operation::first_argument_type(x))`.

20.3.6.3 Template class `binder2nd` [lib.binder.2nd]

```
template <class Operation>
  class binder2nd
    : public unary_function<Operation::first_argument_type,
                            Operation::result_type> {
  protected:
      Operation op;
      argument_type value;

  public:
      binder2nd(const Operation& x, const Operation::second_argument_type& y);
      result_type operator()(const argument_type& x) const;
  };
```

1 The constructor initializes `op` with *x* and `value` with *y*.

2 `operator()` returns `op(x,value)`.

20.3.6.4 `bind2nd` [lib.bind.2nd]

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

Returns: `binder2nd<Operation>(op, Operation::second_argument_type(x))`.

1 For example,

```
find(v.begin(), v.end(), bind2nd(greater<int>(), 5));
```

finds the first integer in vector `v` greater than 5;

```
find(v.begin(), v.end(), bind1st(greater<int>(), 5));
```

finds the first integer in `v` not greater than 5.

20.3.7 Adaptors for pointers to functions [lib.function.pointer.adaptors]

1 To allow pointers to (unary and binary) functions to work with function adaptors the library provides:

```
template <class Arg, class Result>
class pointer_to_unary_function
  : public unary_function<Arg, Result>, restrictor<Result (*)(Arg)> {
public:
  pointer_to_unary_function(Result (*x)(Arg));
  Result operator()(const Arg& x) const;
};
```

2 `operator()` returns `value(x)`.

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

Returns: `pointer_to_unary_function<Arg, Result>(x)`.

```
template <class Arg1, class Arg2, class Result>
class pointer_to_binary_function
  : public binary_function<Arg1,Arg2,Result>, restrictor<Result (*)(Arg1, Arg2)> {
public:
  pointer_to_binary_function(Result (*x)(Arg1, Arg2));
  Result operator()(const Arg1& x, const Arg2& y) const;
};
```

3 `operator()` returns `value(x, y)`.

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

Returns: `pointer_to_binary_function<Arg1,Arg2,Result>(x)`.

4 For example,

```
replace_if(v.begin(), v.end(), not1(bind2nd(ptr_fun(strcmp), "C")), "C++");
```

replaces each `C` with `C++` in sequence `v`.[142)]

20.4 Memory [lib.memory]

Header `<memory>` synopsis

```
#include <cstddef>      // for size_t, ptrdiff_t
#include <iterator>     // for output_iterator
#include <utility>      // for restrictor
namespace std {
// subclause 20.4.1, the default allocator:
  class allocator;
  class allocator::types<void>;
  void* operator new(size_t N, allocator& a);

// subclause 20.4.2, raw storage iterator:
  template <class OutputIterator, class T>
    class raw_storage_iterator;

// subclause 20.4.3, memory handling primitives:
  template <class T> T*   allocate(ptrdiff_t n, T*);
  template <class T> void deallocate(T* buffer);
  template <class T1, class T2> void construct(T1* p, const T2& value);
  template <class T>            void destroy(T* pointer);
  template <class ForwardIterator>
    void destroy(ForwardIterator first, ForwardIterator last);
  template <class T>
    pair<T*,ptrdiff_t> get_temporary_buffer(ptrdiff_t n, T*);
```

142) Compilation systems that have multiple pointer to function types have to provide additional `ptr_fun` template functions.

```
// subclause 20.4.4, specialized algorithms:
  template <class InputIterator, class ForwardIterator>
    ForwardIterator
      uninitialized_copy(InputIterator first, InputIterator last,
                         ForwardIterator result);
  template <class ForwardIterator, class T>
    void uninitialized_fill(ForwardIterator first, ForwardIterator last,
                            const T& x);
  template <class ForwardIterator, class Size, class T>
    void uninitialized_fill_n(ForwardIterator first, Size n, const T& x);
// subclause 20.4.5, pointers:
  template<class X> class auto_ptr;
}
```

1 Header <cstdlib> (Table 36):

Table 36—Header <cstdlib> synopsis

Type	Name(s)	
Functions:	calloc	malloc
	free	realloc

2 Header <cstdlib> (Table 37):

Table 37—Header <cstring> synopsis

Type	Name(s)	
Macro:	NULL <cstring>	
Type:	size_t <cstring>	
Functions:	memchr	memcmp
memcpy	memmove	memset

3 The contents are the same as the Standard C library.

SEE ALSO: ISO C subclause 7.11.2.

20.4.1 The default allocator [lib.default.allocator]

```
namespace std {
  class allocator {
  public:
    typedef size_t    size_type;
    typedef ptrdiff_t difference_type;
    template <class T> class types {
      typedef T*        pointer;
      typedef const T*  const_pointer;
      typedef T&        reference;
      typedef const T&  const_reference;
      typedef T         value_type;
    };

    allocator();
   ~allocator();
```

```
  template<class T> types<T>::pointer
                      address(types<T>::reference x) const;
  template<class T> types<T>::const_pointer
                      address(types<T>::const_reference x) const;

  template<class T, class U>
    types<T>::pointer allocate(size_type, types<U>::const_pointer hint);
  template<class T> void deallocate(types<T>::pointer p);
  size_type max_size() const;
};
```

1 The members `allocate()` and `deallocate()` are parameterized to allow them to be specialized for particular types in user allocators.[143)]

2 It is assumed that any pointer types have a (possibly lossy) conversion to `void*`, yielding a pointer sufficient for use as the this value in a constructor or destructor, and conversions to `A::types<void>::pointer` (for appropriate `A`) as well, for use by `A::deallocate()`.

20.4.1.1 `allocator` members [lib.allocator.members]

```
template<class T> types<T>::pointer
ddress(types<T>::reference x) const;
```

Returns: `&x`.

```
template<class T> types<T>::const_pointer
ddress(types<T>::const_reference x) const;
```

Returns: `&x`.

```
template<class T, class U>
  types<T>::pointer
ocate(size_type n, types<U>::const_pointer hint);
```

Effects: Allocates storage, optionally using *`hint`* to

Box 92

TBS: To Be Specified

Returns: `new T`, if *`n`* `== 1`. Returns `new T[`*`n`*`]`, if *`n`* `> 1`.

Box 93

ISSUE: Is this right? How does `deallocate()` know which form of `delete` to use?

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

Requires: *`p`* shall be a pointer value obtained from `allocate()`.
Effects: Deallocates the storage referenced by *`p`*.

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

[143)] In addition to `allocator` the library vendors are expected to provide allocators for all supported memory models.

Returns:

Box 94
TBS

20.4.1.2 `allocator` placement `new` [lib.allocator.placement]

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

Returns: `a.allocate<char,void>(N,0)`.

20.4.1.3 Example `allocator` [lib.allocator.example]

1 For example, here is an allocator that allows objects in main memory, shared memory, or private heaps. Notably, with this allocator such objects stored under different disciplines have the same type; this is not necessarily the case for other allocators.

```
#include <memory>     // for allocator
class runtime_allocator : public std::allocator {
  class impl {
    impl();
    virtual ~impl();

    virtual void* allocate(size_t)  =0;
    virtual void  deallocate(void*) =0;
    friend class runtime_allocator
    // ... etc. (including a reference count)
  };

  impl* impl_;  // the actual storage manager;

protected:
  runtime_allocator(runtime_allocator::impl* i);
 ~runtime_allocator();

public:
  void* allocate(size_t n) { return impl_->allocate(n); }
  template<class T>
    void deallocate(T* p) { impl_->deallocate(p); }
};

inline void* operator new(size_t N, runtime_allocator& a)
  { return a.allocate(N); }

class shared_allocator : public runtime_allocator {

  class shared_impl : runtime_allocator::impl {
    shared_impl(void* region);
    virtual ~shared_impl();
    virtual void* allocate(size_t);
    virtual void  deallocate(void*);
  };

  shared_allocator(void* region)
    : runtime_allocator(new shared_impl(region)) {}
 ~shared_allocator() {}
};
```

```
class heap : public runtime_allocator {

  class heap_impl : runtime_allocator::impl {
    heap_impl();
    virtual ~heap_impl();
    virtual void* allocate(size_t);
    virtual void  deallocate(void*);
  };

  heap_allocator() : runtime_allocator(new heap_impl) {}
 ~heap_allocator() {}
};
```

20.4.2 Raw storage iterator [lib.storage.iterator]

1 `raw_storage_iterator` is provided to enable algorithms to store the results into uninitialized memory. The formal template parameter `OutputIterator` is required to have its `operator*` return an object for which `operator&` is defined and returns a pointer to `T`.

```
namespace std {
  template <class OutputIterator, class T>
  class raw_storage_iterator
    : public output_iterator, restrictor<OutputIterator> {
  public:
    raw_storage_iterator(OutputIterator x);
    raw_storage_iterator<OutputIterator,T>& operator*();

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

20.4.2.1 `raw_storage_iterator` members [lib.storage.members]

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

Effects: Initializes the iterator to point to the same value to which *x* points.

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

Returns: A reference to the value to which the iterator points.

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

Effects: Constructs a value from *element* at the location to which the iterator points.
Returns: A reference to the iterator.

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

Effects: Pre-increment: advances the iterator and returns a reference to the updated iterator.

```
raw_storage_iterator<OutputIterator,T> operator++(int);
```

Effects: Post-increment: advances the iterator and returns the old value of the iterator.

20.4.3 Memory handling primitives [lib.memory.primitives]

20.4.3.1 `allocate` [lib.allocate]

1 To obtain a typed pointer to an uninitialized memory buffer of a given size the following function is defined:

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

Requires: *n* shall be >= 0.
Effects: The size (in bytes) of the allocated buffer is no less than `n*sizeof(T)`.[144)]

20.4.3.2 `deallocate` [lib.deallocate]

1 Also, the following functions are provided:

```
template <class T> void deallocate(T* buffer);
```

Box 95
TBS

20.4.3.3 `construct` [lib.construct]

```
template <class T1, class T2> void construct(T1* p, const T2& value);
```

Effects: Initializes the location to which *p* points with `value`.

20.4.3.4 `destroy` [lib.destroy]

```
template <class T> void destroy(T* pointer);
```

Effects: Invokes the destructor for the value to which *pointer* points.

```
template <class ForwardIterator>
  void destroy(ForwardIterator first, ForwardIterator last);
```

Effects: Destroys all the values in the range `[first,last)`.

20.4.3.5 `get_temporary_buffer` [lib.get.temporary.buffer]

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

Effects: Finds the largest buffer not greater than `n*sizeof(T)`
Returns: A `pair` containing the buffer's address and capacity (in the units of `sizeof(T)`).[145)]

144) For every memory model there is a corresponding `allocate` template function defined with the first argument type being the distance type of the pointers in the memory model. For example, if a compilation system supports `huge` pointers with the distance type being `long long`, the following template function is provided:

```
template <class T> T huge* allocate(long long n, T*);
```

For every memory model there are corresponding `deallocate`, `construct` and `destroy` template functions defined with the first argument type being the pointer type of the memory model.

145) For every memory model that an implementation supports, there is a corresponding `get_temporary_buffer` template function defined which is overloaded on the corresponding signed integral type. For example, if a system supports `huge` pointers and their difference is of type `long long`, the following function has to be provided:

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

20.4.4 Specialized algorithms [lib.specialized.algorithms]

1 All the iterators that are used as formal template parameters in the following algorithms are required to have their `operator*` return an object for which `operator&` is defined and returns a pointer to `T`.

20.4.4.1 `uninitialized_copy` [lib.uninitialized.copy]

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

Effects: `while (first != last) construct(&*result++, *first++);`
Returns: *result*

20.4.4.2 `uninitialized_fill` [lib.uninitialized.fill]

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

Effects: `while (first != last) construct(&*first++, x);`

20.4.4.3 `uninitialized_fill` [lib.uninitialized.fill.n]

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

Effects: `while (n--) construct(&*first++, x);`

20.4.5 Pointers [lib.pointers]

20.4.5.1 Template class `auto_ptr` [lib.auto.ptr]

1 Template `auto_ptr` holds onto a pointer obtained via `new` and deletes it when it goes out of scope.

```
namespace std {
  template<class X> class auto_ptr {
    auto_ptr(auto_ptr&,void);
    void operator=(auto_ptr&);
  public:
    auto_ptr(X* p =0);
   ~auto_ptr();
    X& operator*() const;
    X* operator->() const;
    X* get() const;
    X* release();
    X* reset(X* =0);
  };
}
```

20.4.5.2 `auto_ptr` members [lib.auto.ptr.members]

20.4.5.2.1 `auto_ptr` constructor [lib.auto.ptr::ctor]

```
auto_ptr(X* p =0);
```

Requires: p points to an object of class X or a class derived from X for which delete p is defined and accessible, or else p is a null pointer.
Postcondition: a.get() == p

20.4.5.2.2 auto_ptr destructor [lib.auto.ptr::dtor]

```
~auto_ptr();
```

Effects: delete get()

20.4.5.2.3 operator* [lib.auto.ptr::op*]

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

Requires: get() != 0
Returns: *get()

20.4.5.2.4 operator-> [lib.auto.ptr::op->]

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

Returns: get()->m

20.4.5.2.5 release [lib.auto.ptr::release]

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

Postcondition: get() == 0

20.4.5.2.6 reset [lib.auto.ptr::reset]

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

Requires: p points to an object of class X or a class derived from X for which delete p is defined and accessible, or else p is a null pointer
Postcondition: get() == p

20.4.6 C library changes [lib.c.malloc]

1 The contents of <cstdlib> are the same as the Standard C library, with the following changes:

2 The functions calloc, malloc, and realloc do not attempt to allocate storage by calling operator new.

3 The function free does not attempt to deallocate storage by calling operator delete. *SEE ALSO:* ISO C subclause 7.11.2.

20.5 Date and time [lib.date.time]

1 Header <ctime> (Table 38):

Table 38—Header `<ctime>` synopsis

Type	Name(s)		
Macros:	`NULL` `<ctime>`		
Types:	`size_t` `<ctime>`		
Struct:	`tm` `<ctime>`		
Functions:			
`asctime`	`difftime`	`localtime`	`strftime`
`ctime`	`gmtime`	`mktime`	`time`

2 The contents are the same as the Standard C library. *SEE ALSO:* ISO C subclause 7.12, Amendment 1 subclause 4.6.4.

21 Strings library [lib.strings]

1 This clause describes components for manipulating sequences of ‘‘characters,’’ where characters may be of type `char`, `wchar_t`, or of a type defined in a C++ program.

2 The following subclauses describe string classes, and null-terminated sequence utilities, as summarized in Table 39:

Table 39—Strings library summary

Subclause	Header(s)
21.1 String classes	`<string>`

21.2 Null-terminated sequence utilities	`<cctype>`
	`<cwctype>`
	`<cstring>`
	`<cwchar>`
	`<cstdlib>`

21.1 String classes [lib.string.classes]

Header `<string>` synopsis

```
#include <memory>       // for allocator
namespace std {
// subclause 21.1.1, basic_string:
  template<class charT> struct string_char_traits;
  template<class charT, class Allocator = allocator,
           class traits = string_char_traits<charT> >
    class basic_string;

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

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

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

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

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

// subclause 21.1.2, string:
  struct string_char_traits<char>;
  typedef basic_string<char> string;
// subclause 21.1.4, wstring:
  struct string_char_traits<wchar_t>;
  typedef basic_string<wchar_t> wstring;
}
```

1 In this subclause, we call the basic character types ‘‘char-like’’ types, and also call the objects of char-like types ‘‘char-like’’ objects or simply ‘‘character’’s.

2 The header `<string>` defines a basic string class template and its traits that can handle all ‘‘char-like’’ template arguments with several function signatures for manipulating varying-length sequences of ‘‘char-like’’ objects.

3 The header `<string>` also defines two specific template classes `string` and `wstring` and their special traits.

21.1.1 Template class `basic_string` [lib.template.string]

21.1.1.1 Template class `string_char_traits` [lib.string.char.traits]

```
namespace std {
  template<class charT> struct string_char_traits {
    typedef charT char_type; // for users to acquire the basic character type

    // constraints
    static void assign(char_type& c1, const char_type& c2)
    static bool eq(const char_type& c1, const char_type& c2)
    static bool ne(const char_type& c1, const char_type& c2)
    static bool lt(const char_type& c1, const char_type& c2)
    static char_type eos(); // the null character
    static basic_istream<charT>&
      char_in(basic_istream<charT>& is, charT& a);
    static basic_ostream<charT>&
      char_out(basic_ostream<charT>& os, charT a);
    static bool is_del(charT a);
      // characteristic function for delimiters of charT

    // speed-up functions
    static int compare(const char_type* s1, const char_type* s2,
                       size_t n);
    static size_t length(const char_type* s);
    static char_type* copy(char_type* s1, const char_type* s2, size_t n);
  };
}
```

21.1.1.2 `string_char_traits` members [lib.string.char.traits.members]

1 Default definitions.

21.1.1.2.1 `assign` [lib.char.traits::assign]

```
static void assign(char_type& c1, const char_type& c2)
```

Effects: Assigns *c2* to *c1*.

21.1.1.2.2 `eq` [lib.char.traits::eq]

```
static bool eq(const char_type& c1, const char_type& c2)
```

Returns `c1 == c2`

21.1.1.2.3 `ne` [lib.char.traits::ne]

```
static bool ne(const char_type& c1, const char_type& c2)
```

Returns: `!(c1 == c2)`

21.1.1.2.4 `lt` [lib.char.traits::lt]

```
static bool lt(const char_type& c1, const char_type& c2)
```

Returns: `c1 < c2`

21.1.1.2.5 `eos` [lib.char.traits::eos]

```
static char_type eos();
```

Returns The null character, `char_type()`

21.1.1.2.6 `char_in` [lib.char.traits::char.in]

```
static basic_istream<charT>&
  char_in(basic_istream<charT>& is, charT& a);
```

Effects: Extracts a `charT` object.
Returns: `is >> a`

21.1.1.2.7 `char_out` [lib.char.traits::char.out]

```
static basic_ostream<charT>&
  char_out(basic_ostream<charT>& os, charT a);
```

Effects: Inserts a `charT` object.
Returns: `os << a`

21.1.1.2.8 `is_del` [lib.char.traits::is.del]

```
static bool is_del(charT a);
```

Effects: Characteristic function for delimiters of `charT`.
Returns: `isspace(a)`

21.1.1.2.9 compare [lib.char.traits::compare]

```
static int compare(const char_type* s1, const char_type* s2,
                   size_t n);
```

Effects:

```
  for (size_t i = 0; i < n; ++i, ++s1, ++s2)
    if (ne(*s1, *s2)) {
      return lt(*s1, *s2) ? -1 : 1;
    }
  return 0;
```

21.1.1.2.10 length [lib.char.traits::length]

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

Effects:

```
  size_t l = 0;
  while (ne(*s++, eos()) ++l;
    return l;
```

21.1.1.2.11 copy [lib.char.traits::copy]

```
static char_type* copy(char_type* s1, const char_type* s2, size_t n);
```

Effects:

```
  char_type* s = s1;
  for (size_t i = 0; i < n; ++i) assign(*++s1, *++s2);
  return s;
```

21.1.1.3 Template class basic_string [lib.basic.string]

```
namespace std {
  template<class charT, class Allocator = allocator,
           class traits = string_char_traits<charT> >
  class basic_string {
  public:
  // typedefs:
    typedef traits            traits_type;
    typedef traits::char_type value_type;
    typedef value_type*       iterator;
    typedef value_type* const const_iterator;
    typedef allocator::size_type       size_type;
    typedef allocator::difference_type difference_type;
// Added to make the typedefs work:
    typedef allocator::types<charT> alloc_type;

    typedef alloc_type::reference       reference;
    typedef alloc_type::const_reference const_reference;
    typedef alloc_type::pointer         pointer;
    typedef alloc_type::const_pointer   pointer;
```

```
  basic_string();
  basic_string(const basic_string& str, size_type pos = 0,
               size_type n = NPOS);
  basic_string(const_pointer s, size_type n);
  basic_string(const_pointer s);
  basic_string(size_type n, charT c);
 ~basic_string();
  basic_string& operator=(const basic_string& str);
  basic_string& operator=(const_pointer s);
  basic_string& operator=(charT c);

  basic_string& operator+=(const basic_string& rhs);
  basic_string& operator+=(const_pointer s);
  basic_string& operator+=(charT c);
  basic_string& append(const basic_string& str, size_type pos = 0,
                       size_type n = NPOS);
  basic_string& append(const_pointer s, size_type n);
  basic_string& append(const_pointer s);
  basic_string& append(size_type pos, size_type n, const charT c = charT());

  basic_string& assign(const basic_string& str, size_type pos = 0,
                       size_type n = NPOS);
  basic_string& assign(const_pointer s, size_type n);
  basic_string& assign(const_pointer s);
  basic_string& assign(size_type pos, size_type n, const charT c = charT());

  basic_string& insert(size_type pos1, const basic_string& str,
                       size_type pos2 = 0, size_type n = NPOS);
  basic_string& insert(size_type pos, const_pointer s, size_type n);
  basic_string& insert(size_type pos, const_pointer s);
  basic_string& insert(size_type pos, size_type n, const charT c = charT() );

  basic_string& remove(size_type pos = 0, size_type n = NPOS);

  basic_string& replace(size_type pos1, size_type n1, const basic_string& str,
                        size_type pos2 = 0, size_type n2 = NPOS);
  basic_string& replace(size_type pos, size_type n1, const_pointer s,
                        size_type n2);
  basic_string& replace(size_type pos, size_type n1, const_pointer s);
  basic_string& replace(size_type pos, size_type n, charT c = charT());

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

  const_pointer c_str() const;
  const_pointer data() const;
  size_type size() const;
  size_type max_size() const;
  void resize(size_type n, charT c);
  void resize(size_type n);
  size_type capacity() const;
  void reserve(size_type res_arg);

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

```
size_type find(const basic_string& str, size_type pos = 0) const;
size_type find(const_pointer s, size_type pos, size_type n) const;
size_type find(const_pointer s, size_type pos = 0) const;
size_type find(charT c, size_type pos = 0) const;
size_type rfind(const basic_string& str, size_type pos = NPOS) const;
size_type rfind(const_pointer s, size_type pos, size_type n) const;
size_type rfind(const_pointer s, size_type pos = NPOS) const;
size_type rfind(charT c, size_type pos = NPOS) const;

size_type find_first_of(const basic_string& str, size_type pos = 0) const;
size_type find_first_of(const_pointer s, size_type pos, size_type n) const;
size_type find_first_of(const_pointer s, size_type pos = 0) const;
size_type find_first_of(charT c, size_type pos = 0) const;
size_type find_last_of (const basic_string& str, size_type pos = NPOS) const;
size_type find_last_of (const_pointer s, size_type pos, size_type n) const;
size_type find_last_of (const_pointer s, size_type pos = NPOS) const;
size_type find_last_of (charT c, size_type pos = NPOS) const;

size_type find_first_not_of(const basic_string& str, size_type pos = 0) const;
size_type find_first_not_of(const_pointer s, size_type pos,
                            size_type n) const;
size_type find_first_not_of(const_pointer s, size_type pos = 0) const;
size_type find_first_not_of(charT c, size_type pos = 0) const;
size_type find_last_not_of (const basic_string& str, size_type pos = NPOS) const;
size_type find_last_not_of (const_pointer s, size_type pos,
                            size_type n) const;
size_type find_last_not_of (const_pointer s, size_type pos = NPOS) const;
size_type find_last_not_of (charT c, size_type pos = NPOS) const;
```

1 For a char-like type `charT`, the template class `basic_string` describes objects that can store a sequence consisting of a varying number of arbitrary char-like objects. The first element of the sequence is at position zero. Such a sequence is also called a ‘‘string’’ if the given char-like type is clear from context. In the rest of this clause, `charT` denotes a such given char-like type. Storage for the string is allocated and freed as necessary by the member functions of class `basic_string`.

2 In all cases, `size() <= capacity()`.

3 The functions described in this clause can report two kinds of errors, each associated with a distinct exception: *

— a *length* error is associated with exceptions of type `length_error`;

— an *out-of-range* error is associated with exceptions of type out_of_range .

21.1.1.4 `basic_string` member functions [lib.string.members]

21.1.1.4.1 `basic_string` constructors [lib.string.cons]

```
basic_string();
```

Effects: Constructs an object of class `basic_string`. The postconditions of this function are indicated in Table 40:

Table 40—basic_string() effects

Element	Value
data()	an unspecified value
size()	0
capacity()	an unspecified value

```
basic_string(const basic_string<charT,Allocator,traits>& str,
         size_type pos = 0, size_type n = NPOS);
```

Requires: `pos <= size()`
Throws: `out_of_range` if *pos* > *str* `.size()`.
Effects: Constructs an object of class `basic_string` and determines the effective length *rlen* of the initial string value as the smaller of *n* and *str* `.size() -` *pos*, as indicated in Table 41:

Table 41—basic_string(basic_string,size_type,size_type) effects

Element	Value
data()	points at the first element of an allocated copy of *rlen* elements of the string controlled by *str* beginning at position *pos*
size()	*rlen*
capacity()	a value at least as large as *len*

```
basic_string(const_pointer s, size_type n = NPOS);
```

Requires: *s* shall not be a null pointer.
Effects: Constructs an object of class `basic_string` and determines its initial string value from the array of *charT* of length *n* whose first element is designated by *s*, as indicated in Table 42:

Table 42—basic_string(const_pointer,size_type) effects

Element	Value
data()	points at the first element of an allocated copy of the array whose first element is pointed at by *s*
size()	*n*
capacity()	a value at least as large as *len*

```
basic_string(const_pointer s);
```

Requires: *s* shall not be a null pointer.
Effects: Constructs an object of class `basic_string` and determines its initial string value from the array of `charT` of length `traits::length(s)` whose first element is designated by *s*, as indicated in Table 43:

Table 43—`basic_string(const_pointer)` effects

Element	Value
data()	points at the first element of an allocated copy of the array whose first element is pointed at by *s*
size()	*traits::length(s)*
capacity()	a value at least as large as *len*

Notes: Uses `traits::length()`.

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

Requires: `n < NPOS`
Throws: `length_error` if *n* `== NPOS`.
Effects: Constructs an object of class `basic_string` and determines its initial string value by repeating the char-like object *c* for all *n* elements, as indicated in Table 44:

Table 44—`basic_string(charT,size_type)` effects

Element	Value
data()	points at the first element of an allocated array of *n* elements, each storing the initial value *c*
size()	*n*
capacity()	a value at least as large as *len*

21.1.1.4.2 `basic_string::operator=` [lib.string::op=]

```
basic_string<charT,Allocator,traits>&
  operator=(const basic_string<charT,Allocator,traits>& str);
```

Returns: `*this = basic_string<charT,Allocator,traits>(str)`.

```
basic_string<charT,Allocator,traits>&
  operator=(const_pointer s);
```

Returns: `*this = basic_string<charT,Allocator,traits>(s)`.
Notes: Uses `traits::length()`.

```
basic_string<charT,Allocator,traits>& operator=(charT c);
```

Returns: `*this = basic_string<charT,Allocator,traits>(c)`.

21.1.1.4.3 `basic_string::operator+=` [lib.string::op+=]

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

Returns: `append(rhs)`.

```
basic_string<charT,Allocator,traits>& operator+=(const_pointer s);
```

Returns: `*this += basic_string<charT,Allocator,traits>(s)`.
Notes: Uses `traits::length()`.

```
basic_string<charT,Allocator,traits>& operator+=(charT c);
```

Returns: `*this += basic_string<charT,Allocator,traits>(c)`.

21.1.1.4.4 `basic_string::append` [lib.string::append]

```
basic_string<charT,Allocator,traits>&
  append(const basic_string<charT,traits>& str,
     size_type pos = 0, size_type n = NPOS);
```

Requires: *pos* `<= size()`
Throws: `out_of_range` if *pos* `> str.size()`.
Effects: Determines the effective length *rlen* of the string to append as the smaller of *n* and *str* `.size() -` *pos*. The function then throws `length_error` if `size() >= NPOS -` *rlen*. Otherwise, the function replaces the string controlled by `*this` with a string of length `size() +` *rlen* whose first *len* elements are a copy of the original string controlled by `*this` and whose remaining elements are a copy of the initial elements of the string controlled by *str* beginning at position *pos*.
Returns: `*this`.

```
basic_string<charT,Allocator,traits>&
  append(const_pointer s, size_type n);
```

Returns: `append(basic_string<charT,Allocator,traits>(s,n))`.

```
basic_string<charT,Allocator,traits>& append(const_pointer s);
```

Returns: `append(basic_string<charT,Allocator,traits>(s))`.
Notes: Uses `traits::length()`.

```
basic_string<charT,Allocator,traits>&
  append(size_type pos, size_type n, const charT c = charT());
```

Returns: `append(basic_string<charT,Allocator,traits>(c,n),pos)`.

21.1.1.4.5 `basic_string::assign` [lib.string::assign]

```
basic_string<charT,Allocator,traits>&
  assign(const basic_string<charT,traits>& str,
     size_type pos = 0, size_type n = NPOS);
```

Requires: *pos* `<= size()`
Throws: `out_of_range` if *pos* `> str.size()`.
Effects: Determines the effective length *rlen* of the string to assign as the smaller of *n* and *str* `.size() -` *pos*.
The function then replaces the string controlled by `*this` with a string of length *rlen* whose elements are a copy of the string controlled by *str* beginning at position *pos*.
Returns: `*this`.

```
basic_string<charT,Allocator,traits>&
  assign(const_pointer s, size_type n);
```

Returns: assign(basic_string<charT,Allocator,traits>(s,n)).

```
basic_string<charT,Allocator,traits>& assign(const_pointer s);
```

Returns: assign(basic_string(s)).
Notes: Uses traits::length().

```
basic_string<charT,Allocator,traits>&
  assign(size_type pos, size_type n, const charT c = charT());
```

Returns: assign(basic_string<charT,Allocator,traits>(c,n),pos).

21.1.1.4.6 basic_string::insert [lib.string::insert]

```
basic_string<charT,Allocator,traits>&
  insert(size_type pos1,
     const basic_string<charT,Allocator,traits>& str,
     size_type pos2 = 0, size_type n = NPOS);
```

Requires *pos1* <= size()
Throws: out_of_range if *pos1* > size() or *pos2* > *str*.size().
Effects: Determines the effective length *rlen* of the string to insert as the smaller of *n* and *str*.len - *pos2*. Then throws length_error if size() >= NPOS - *rlen*.
Otherwise, the function replaces the string controlled by *this with a string of length size() + *rlen* whose first *pos1* elements are a copy of the initial elements of the original string controlled by *this, whose next *rlen* elements are a copy of the elements of the string controlled by *str* beginning at position *pos2*, and whose remaining elements are a copy of the remaining elements of the original string controlled by *this.
Returns: *this.

```
basic_string<charT,Allocator,traits>&
  insert(size_type pos, const_pointer s, size_type n);
```

Returns: insert(*pos*,basic_string<charT,Allocator,traits>(s,n)).

```
basic_string<charT,Allocator,traits>&
  insert(size_type pos, const_pointer s);
```

Returns: insert(*pos*,basic_string(s)).
Notes: Uses *traits::length()*.

```
basic_string<charT,Allocator,traits>&
  insert(size_type pos, size_type n, const charT c = charT());
```

Returns: insert(*pos*,basic_string<charT,Allocator,traits>(c,n)).

21.1.1.4.7 basic_string::remove [lib.string::remove]

```
basic_string<charT,Allocator,traits>&
  remove(size_type pos = 0, size_type n = NPOS);
```

Requires: *pos* <= size()
Throws: out_of_range if *pos* > size().
Effects: Determines the effective length *xlen* of the string to be removed as the smaller of *n* and size() - *pos*.
The function then replaces the string controlled by *this with a string of length size() - *xlen* whose first *pos* elements are a copy of the initial elements of the original string controlled by *this, and whose remaining elements are a copy of the elements of the original string controlled by *this

beginning at position *pos + xlen*.

Returns: *this.

21.1.1.4.8 basic_string::replace [lib.string::replace]

```
basic_string<charT,Allocator,traits>&
  replace(size_type pos1, size_type n1,
      const basic_string<charT,Allocator,traits>& str,
      size_type pos2 = 0, size_type n2 = NPOS);
```

Requires: *pos1* <= size() && *pos2* <= size().

Throws: out_of_range if *pos1* > size() or *pos2* > *str*.size().

Effects: Determines the effective length *xlen* of the string to be removed as the smaller of *n1* and size() - &*pos1*. It also determines the effective length *rlen* of the string to be inserted as the smaller of *n2* and *str*.size() - *pos2*. Then throws length_error if size() - *xlen* >= NPOS - *rlen*.

Otherwise, the function replaces the string controlled by *this with a string of length +size() - *xlen whose first pos1 elements are a copy of the initial elements of the original string controlled by* *this*, whose next rlen elements are a copy of the initial elements of the string controlled by str beginning at position pos2, and whose remaining elements are a copy of the elements of the original string controlled by* *this *beginning at position pos1 + xlen.*

Returns: *this.

```
  basic_string<charT,Allocator,traits>&
    replace(size_type pos, size_type n1, const_pointer s, size_type n2);
```

Returns: replace(*pos*,*n1*,basic_string<charT,Allocator,traits>(s,n2)).

```
  basic_string<charT,Allocator,traits>&
    replace(size_type pos, size_type n1, const_pointer s);
```

Returns: replace(*pos*,*n1*,basic_string<charT,Allocator,traits>(s)).

Notes: Uses traits::length().

```
  basic_string<charT,Allocator,traits>&
    replace(size_type pos, size_type n, charT c = charT());
```

Returns: replace(*pos*,*n*,basic_string<charT,Allocator,traits>(c,n)). *

21.1.1.4.9 basic_string::operator[] [lib.string::op.array]

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

Returns: If *pos* < size(), returns data()[*pos*]. Otherwise, if *pos* == size(), the const version returns traits::eos(). Otherwise, the behavior is undefined.

Notes: The reference returned by the non- const version is invalid after any subsequent call to c_str(), data(), or any non- const member function for the object.

21.1.1.4.10 basic_string::at [lib.string::at]

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

Requires: `pos < size()`
Throws: `out_of_range` if *`pos >= size()`*.
Returns: `operator[](`*`pos`*`)`.

Box 96

Previous definition of `put_at()`: Otherwise, if *`pos == len`*, the function replaces the string controlled by `*this` with a string of length *`len + 1`* whose first *`len`* elements are a copy of the original string and whose remaining element is initialized to *`c`*. Otherwise, the function assigns *`c`* to *`ptr[pos]`*.

21.1.1.4.11 `basic_string::c_str` [lib.string::c.str]

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

Returns: A pointer to the initial element of an array of length `size() + 1` whose first `size()` elements equal the corresponding elements of the string controlled by `*this` and whose last element is a null character specified by `traits::eos()`.
Requires: The program shall not alter any of the values stored in the array. Nor shall the program treat the returned value as a valid pointer value after any subsequent call to a non-const member function of the class `basic_string` that designates the same object as `this`.
Notes: Uses `traits::eos()`.

21.1.1.4.12 `basic_string::data` [lib.string::data]

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

Returns: `c_str()` if nonzero,`size()`is
Requires: The program shall not alter any of the values stored in the character array. Nor shall the program treat the returned value as a valid pointer value after any subsequent call to a non- `const` member function of `basic_string` that designates the same object as `this`.

21.1.1.4.13 `basic_string::size` [lib.string::size]

```
size_type size() const:
```

Returns: a count of the number of char-like objects currently in the string.
Notes: Uses `traits::length()`.

21.1.1.4.14 `basic_string::max_size` [lib.string::max.size]

```
size_type max_size() const:
```

Returns: The maximum size of the string.

21.1.1.4.15 `basic_string::resize` [lib.string::resize]

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

Requires: *`n`* `!= NPOS`
Throws: `length_error` if *`n`* `== NPOS`.
Effects: Alters the length of the string designated by `*this` as follows:

— If *`n`* `<= size()`, the function replaces the string designated by `*this` with a string of length *`n`* whose elements are a copy of the initial elements of the original string designated by `*this`.

— If *`n`* `> size()`, the function replaces the string designated by `*this` with a string of length *`n`* whose first `size()` elements are a copy of the original string designated by `*this`, and whose remaining elements are all initialized to *`c`*.

```
void resize(size_type n);
```

Returns: `resize(n, eos())`.
Notes: Uses `traits::eos()`.

21.1.1.4.16 basic_string::reserve [lib.string::reserve]

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

Returns: the size of the allocated storage in the string.

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

1 The member function `reserve()` is a directive that informs a `basic_string` of a planned change in size, so that it can manage the storage allocation accordingly.
Effects: After `reserve()`, `capacity()` is greater or equal to the argument of `reserve` if reallocation happens; and equal to the previous value of `capacity()` otherwise.
Reallocation happens at this point if and only if the current capacity is less than the argument of `reserve()`.
Complexity: It does not change the size of the sequence and takes at most linear time in the size of the sequence.
Notes: Reallocation invalidates all the references, pointers, and iterators referring to the elements in the sequence. It is guaranteed that no reallocation takes place during the insertions that happen after `reserve()` takes place till the time when the size of the string reaches the size specified by `reserve()`.

21.1.1.4.17 basic_string::empty [lib.string::empty]

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

Returns: `size() == 0`.

21.1.1.4.18 basic_string::copy [lib.string::copy]

```
size_type copy(pointer s, size_type n, size_type pos = 0);
```

Requires: *pos* `<= size()`
Throws: `out_of_range` if *pos* `> size()`.
Effects: Determines the effective length *rlen* of the string to copy as the smaller of *n* and `size() -` *pos*. *s* shall designate an array of at least *rlen* elements.
The function then replaces the string designated by *s* with a string of length *rlen* whose elements are a copy of the string controlled by `*this` beginning at position *pos*.[146)]
Returns: *rlen*.

21.1.1.4.19 basic_string::swap [lib.string::swap]

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

Effects: Swaps the contents of the two strings.
Postcondition: `*this` contains the characters that were in `s`, `s` contains the characters that were in `*this`.

146) The function does not append a null object to the string.

Complexity: Constant time.

21.1.1.4.20 basic_string::find [lib.string::find]

```
size_type find(const basic_string<charT,Allocator,traits>& str,
          size_type pos = 0) const;
```

Effects: Determines the lowest position *xpos*, if possible, such that both of the following conditions obtain:

— *pos* <= *xpos* and *xpos* + *str*.size() <= size();

— traits::eq(at(*xpos*+*I*),*str*.at(*I*)) for all elements *I* of the string controlled by *str*.
Returns: *xpos* if the function can determine such a value for *xpos*. Otherwise, returns NPOS.

```
size_type find(const_pointer s, size_type pos, size_type n) const;
```

Returns: find(basic_string<charT,Allocator,tratis>(*s*,*n*)*pos*).

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

Returns: find(basic_string<charT,Allocator,traits>(*s*),*pos*).
Notes: Uses traits::length().

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

Returns: find(basic_string<charT,Allocator,traits>(*c*),*pos*).

21.1.1.4.21 basic_string::rfind [lib.rfind]

```
size_type rfind(const basic_string<charT,Allocator,traits>& str,
               size_type pos = NPOS) const;
```

Effects: Determines the highest position *xpos*, if possible, such that both of the following conditions obtain:

— *xpos* <= *pos* and *xpos* + &*str* .size() <= size();

— traits::eq(at(*xpos*+*I*),*str*.at(*I*)) for all elements *I* of the string controlled by *str*.
Returns: *xpos* if the function can determine such a value for *xpos*. Otherwise, returns NPOS.

```
size_type rfind(const_pointer s, size_type pos, size_type n) const;
```

Returns: rfind(basic_string<charT,Allocator,traits>(*s*,*n*),*pos*).

```
size_type rfind(const_pointer s, size_type pos = NPOS) const;
```

Returns: rfind(basic_string<charT,Allocator,traits>(*s*),*pos*).
Notes: Uses traits::length().

```
size_type rfind(charT c, size_type pos = NPOS) const;
```

Returns: rfind(basic_string<charT,Allocator,traits>(*c*,*n*),*pos*).

21.1.1.4.22 basic_string::find_first_of [lib.string::find.first.of]

```
size_type
  find_first_of(const basic_string<charT,Allocator,traits>& str,
        size_type pos = 0) const;
```

Effects: Determines the lowest position *xpos*, if possible, such that both of the following conditions obtain:

— *pos* <= *xpos* and *xpos* < size();

— traits::eq(at(xpos), str.at(I)) for some element *I* of the string controlled by *str*.

Returns: *xpos* if the function can determine such a value for *xpos*. Otherwise, returns NPOS.

```
size_type
  find_first_of(const_pointer s, size_type pos, size_type n) const;
```

Returns: find_first_of(basic_string<charT,Allocator,tratis>(*s*,*n*),*pos*).

```
size_type find_first_of(const_pointer s, size_type pos = 0) const;
```

Returns: find_first_of(basic_string<charT,Allocator,tratis>(*s*),*pos*).
Notes: Uses traits::length().

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

Returns: find_first_of(basic_string<charT,Allocator,tratis>(*c*),*pos*).

21.1.1.4.23 basic_string::find_last_of [lib.string::find.last.of]

```
size_type
  find_last_of(const basic_string<charT,Allocator,traits>& str,
               size_type pos = NPOS) const;
```

Effects: Determines the highest position *xpos*, if possible, such that both of the following conditions obtain:

— *xpos* <= *pos* and *pos* < size();

— traits::eq(at(*xpos*),*str*.at(*I*)) for some element *I* of the string controlled by *str*.

Returns: *xpos* if the function can determine such a value for *xpos*. Otherwise, returns NPOS.

```
size_type find_last_of(const_pointer s, size_type pos, size_type n) const;
```

Returns: find_last_of(basic_string<charT,Allocator,tratis>(*s*,*n*),*pos*).

```
size_type find_last_of(const_pointer s, size_type pos = NPOS) const;
```

Returns: find_last_of(basic_string<charT,Allocator,tratis>(*s*),*pos*).
Notes: Uses traits::length().

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

Returns: find_last_of(basic_string<charT,Allocator,traits>(*c*),*pos*).

21.1.1.4.24 basic_string::find_first_not_of [lib.string::find.first.not.of]

```
size_type
  find_first_not_of(const basic_string<charT,Allocator,traits>& str,
                    size_type pos = 0) const;
```

Effects: Determines the lowest position *xpos*, if possible, such that both of the following conditions obtain:

— *pos* <= *xpos* and *xpos* < size();

— `traits::eq(at(xpos), str.at(I))` for no element *`I`* of the string controlled by *`str`*.
Returns: *`xpos`* if the function can determine such a value for *`xpos`*. Otherwise, returns `NPOS`.

```
size_type
  find_first_not_of(const_pointer s, size_type pos, size_type n) const;
```

Returns: `find_first_not_of(basic_string<charT,Allocator,traits>(`*`s`*`,`*`n`*`),`*`pos`*`)`.

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

Returns: `find_first_not_of(basic_string<charT,Allocator,traits>(`*`s`*`),`*`pos`*`)`.
Notes: Uses `traits::length()`.

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

Returns: `find_first_not_of(basic_string<charT,Allocator,tratis>(`*`c`*`),`*`pos`*`)`.

21.1.1.4.25 `basic_string::find_last_not_of` [lib.string::find.last.not.of]

```
size_type
  find_last_not_of(const basic_string<charT,Allocator,traits>& str,
           size_type pos = NPOS) const;
```

Effects: Determines the highest position *`xpos`*, if possible, such that both of the following conditions obtain:

— *`xpos`* `<=` *`pos`* and *`pos`* `< size()`;

— `traits::eq(at(xpos), str.at(I))` for no element *`I`* of the string controlled by *`str`*.
Returns: *`xpos`* if the function can determine such a value for *`xpos`*. Otherwise, returns `NPOS`.

```
size_type find_last_not_of(const_pointer s, size_type pos, size_type n) const;
```

Returns: `find_last_not_of(basic_string<charT,Allocator,traits>(`*`s`*`,`*`n`*`),`*`pos`*`)`.

```
size_type find_last_not_of(const_pointer s, size_type pos = NPOS) const;
```

Returns: `find_last_not_of(basic_string<charT,Allocator,traits>(`*`s`*`),`*`pos`*`)`.
Notes: Uses `traits::length()`.

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

Returns: `find_last_not_of(basic_string<charT,Allocator,traits>(`*`c`*`),`*`pos`*`)`.

21.1.1.4.26 `basic_string::substr` [lib.string::substr]

```
basic_string<charT,Allocator,traits>
  substr(size_type pos = 0, size_type n = NPOS) const;
```

Requires: *`pos`* `<= size()`
Throws: `out_of_range` if *`pos`* `> size()`.
Effects: Determines the effective length *`rlen`* of the string to copy as the smaller of *`n`* and `size() -` *`pos`*.
Returns: `basic_string<charT,Allocator,traits>(data()+`*`pos`*`,`*`rlen`*`)`.

21.1.1.4.27 basic_string::compare [lib.string::compare]

```
int compare(const basic_string<charT,Allocator,traits>& str,
        size_type pos = 0, size_type n = NPOS)
```

Requires: *pos* <= size()
Throws: out_of_range if *pos* > size().
Effects: Determines the effective length *rlen* of the strings to compare as the smallest of *n*, size() - *pos*, and *str* .size(). The function then compares the two strings by calling traits::compare(data()+*pos*,*str*.data(),*rlen*).
Returns: the nonzero result if the result of the comparison is nonzero. Otherwise, returns a value as indicated in Table 45:

Table 45—compare() results

Condition	Return Value
size() - *pos* < *str*.size()	a value < 0
size() - *pos* == *str*.size()	0
size() - *pos* > str.size()	a value > 0

1 Uses traits::compare().

```
int compare(const_pointer s, size_type pos, size_type n) const;
```

Returns: compare(basic_string<charT,Allocator,traits>(s, n), pos).
Notes: Uses traits::compare().

```
int compare(const_pointer s, size_type pos = 0) const;
```

Returns: compare(basic_string<charT,Allocator,traits>(s), pos).
Notes: Uses traits::length() and traits::compare().

21.1.1.5 basic_string non-member functions [lib.string.nonmembers]

21.1.1.5.1 operator+ [lib.string::op+]

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

Returns: basic_string<charT,Allocator,traits>(lhs).append(rhs).

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

Returns: *basic_string<charT,Allocator,traits>(lhs) + rhs*.
Notes: Uses traits::length().

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

Returns: basic_string<charT,Allocator,traits>(*lhs*)+*rhs*.

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

Returns: basic_string<charT,Allocator,tratis>(*lhs*)+ basic_string<charT,Allocator,tratis>(*rhs*).
Notes: Uses traits::length().

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

Returns: *lhs* + basic_string<charT,Allocator,traits>(*rhs*).

21.1.1.5.2 operator== [lib.string::operator==]

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

Returns: *lhs*.compare(*rhs*) == 0.

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

Returns: basic_string<charT,Allocator,tratis>(*lhs*) == *rhs*.

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

Returns: *lhs* == basic_string<charT,Allocator,traits>(*rhs*).
Notes: Uses traits::length().

21.1.1.5.3 operator!= [lib.string::op!=]

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


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


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

Returns: basic_string<charT,Allocator,traits>(*lhs*).compare(*rhs*) != 0.
Notes: Uses traits::length().

21.1.1.5.4 operator< [lib.string::op<]

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

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

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

Returns: basic_string<charT,Allocator,traits>(f6lhs).compare(*rhs*) < 0.

21.1.1.5.5 operator> [lib.string::op>]

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

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

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

Returns: basic_string<charT,Allocator,traits>(f6lhs).compare(*rhs*) > 0.

21.1.1.5.6 operator<= [lib.string::op<=]

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

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

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

Returns: basic_string<charT,Allocator,traits>(f6lhs).compare(*rhs*) <= 0.

21.1.1.5.7 operator>= [lib.string::op>=]

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

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

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

Returns: `basic_string<charT,Allocator,traits>(f6lhs).compare(`*rhs*`) >= 0`.

21.1.1.5.8 Inserters and extractors

1 **Notes:** Uses `traits::char_in` and `is_del()`.

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

Notes: Uses `traits::char_out()`.

21.1.2 Class string [lib.string]

```
namespace std {
  struct string_char_traits<char> {
    typedef char char_type;

    static void assign(char_type& c1, const char_type& c2);
    static bool eq(const char_type& c1, const char_type& c2);
    static bool ne(const char_type& c1, const char_type& c2);
    static bool lt(const char_type& c1, const char_type& c2);
    static char_type eos();

    static int compare(const char_type* s1, const char_type* s2,
                       size_type n);

    static size_type length(const char_type* s);
    static char_type* copy(char_type* s1, const char_type* s2, size_type n);
  };

  typedef basic_string<char> string;
}
```

21.1.3 string_char_traits<char> members [lib.string.traits.members]

21.1.3.1 assign [lib.string.traits::assign]

```
static void assign(char_type& c1, const char_type& c2);
```

Effects: c1 = c2

21.1.3.2 eq [lib.string.traits::eq]

```
static bool eq(const char_type& c1, const char_type& c2);
```

Returns: c1 == c2

21.1.3.3 ne [lib.string.traits::ne]

```
static bool ne(const char_type& c1, const char_type& c2);
```

Returns: c1 != c2

21.1.3.4 lt [lib.string.traits::lt]

```
static bool lt(const char_type& c1, const char_type& c2);
```

Returns: c1 < c2

21.1.3.5 eos [lib.string.traits::eos]

```
static char_type eos();
```

Returns: 0

21.1.3.6 compare [lib.string.traits::compare]

```
static int compare(const char_type* s1, const char_type* s2,
     size_type n);
```

Returns: memcmp(s1,s2,n)

21.1.3.7 length [lib.string.traits::length]

```
static size_type length(const char_type* s);
```

Returns: strlen(s)

21.1.3.8 copy [lib.string.traits::copy]

```
static char_type* copy(char_type* s1, const char_type* s2, size_type n);
```

Returns: memcpy(s1,s2,n)

21.1.4 Class wstring [lib.wstring]

```
namespace std {
  struct string_char_traits<wchar_t> {
    typedef wchar_t char_type;
    static void assign(char_type& c1, const char_type& c2);
    static bool eq(const char_type& c1, const char_type& c2);
    static bool ne(const char_type& c1, const char_type& c2);
    static bool lt(const char_type& c1, const char_type& c2);
    static char_type eos();

    static int compare(const char_type* s1, const char_type* s2, size_type n);
    static size_type length(const char_type* s);
```

```
    static char_type* copy(char_type* s1, const char_type* s2, size_type n);
  };

  typedef basic_string<wchar_t> wstring;
}
```

21.1.5 string_char_traits<wchar_t> members [lib.wstring.members]

21.1.5.1 assign [lib.wstring::assign]

```
static void assign(char_type& c1, const char_type& c2);
```

Effects: c1 = c2

21.1.5.2 eq [lib.wstring::eq]

```
static bool eq(const char_type& c1, const char_type& c2);
```

Returns: c1 == c2

21.1.5.3 ne [lib.wstring::ne]

```
static bool ne(const char_type& c1, const char_type& c2);
```

Returns: c1 != c2

21.1.5.4 lt [lib.wstring::lt]

```
static bool lt(const char_type& c1, const char_type& c2);
```

Returns: c1 < c2

21.1.5.5 eos [lib.wstring::eos]

```
static char_type eos();
```

Returns: 0

21.1.5.6 compare [lib.wstring::compare]

```
static int compare(const char_type* s1, const char_type* s2,
                   size_type n);
```

Returns: wmemcmp(s1,s2,n)

21.1.5.7 length [lib.wstring::length]

```
static size_type length(const char_type* s);
```

Returns: wcslen(s)

21.1.5.8 copy [lib.wstring::copy]

```
static char_type* copy(char_type* s1, const char_type* s2,
                       size_type n);
```

Returns: `wmemcpy(s1,s2,n)`

21.2 Null-terminated sequence utilities [lib.c.strings]

1 Headers <cctype>, <cwctype>, <cstring>, <cwchar>, <cstdlib>(multibyte conversions), and <ciso646>.

Table 45—Header <cctype> synopsis

Type	Name(s)			
Functions:				
isalnum	isdigit	isprint	isupper	tolower
isalpha	isgraph	ispunct	isxdigit	toupper
iscntrl	islower	isspace		

Table 45—Header <cwctype> synopsis

Type	Name(s)				
Macro:	WEOF <cwctype>				
Types:	wctrans_t	wctype_t	wint_t <cwctype>		
Functions:					
iswalnum	iswctype	iswlower	iswspace	towctrans	wctrans
iswalpha	iswdigit	iswprint	iswupper	towlower	wctype
iswcntrl	iswgraph	iswpunct	iswxdigit	towupper	

Table 45—Header <cstring> synopsis

Type	Name(s)			
Macro:	NULL <cstring>			
Type:	size_type <cstring>			
Functions:				
strcoll		strlen	strpbrk	strtok
strcat	strcpy	strncat	strrchr	strxfrm
strchr	strcspn	strncmp	strspn	
strcmp	strerror	strncpy	strstr	

Table 45—Header <cwchar> synopsis

Type	Name(s)				
Macros:	NULL <cwchar>	WCHAR_MAX	WCHAR_MIN	WEOF <cwchar>	
Types:	mbstate_t	wint_t <cwchar>			
Struct:	tm <cwchar>				
Functions:					
btowc	getwchar	ungetwc	wcscpy	wcsrtombs	wmemchr
fgetwc	mbrlen	vfwprintf	wcscspn	wcsspn	wmemcmp
fgetws	mbrtowc	vswprintf	wcsftime	wcsstr	wmemcpy
fputwc	mbsinit	vwprintf	wcslen	wcstod	wmemmove
fputws	mbsrtowcs	wcrtomb	wcsncat	wcstok	wmemset
fwide	putwc	wcscat	wcsncmp	wcstol	wprintf
fwprintf	putwchar	wcschr	wcsncpy	wcstoul	wscanf
fwscanf	swprintf	wcscmp	wcspbrk	wcsxfrm	
getwc	swscanf	wcscoll	wcsrchr	wctob	

Table 45—Header <cstdlib> synopsis

Type	Name(s)		
Macros:	MB_CUR_MAX		
Functions:			
atol	mblen	strtod	wctomb
atof	mbstowcs	strtol	wcstombs
atoi	mbtowc	stroul	

2 The contents are the same as the Standard C library, with the following modifications:

3 None of the headers shall define the type `wchar_t` (2.8).

4 The function signature `strchr(const char*, int)` is replaced by the two declarations:

```
const char* strchr(const char* s, int c);
      char* strchr(      char* s, int c);
```

5 both of which have the same behavior as the original declaration.

6 The function signature `strpbrk(const char*, const char*)` is replaced by the two declarations:

```
const char* strpbrk(const char* s1, const char* s2);
      char* strpbrk(      char* s1, const char* s2);
```

7 both of which have the same behavior as the original declaration.

8 The function signature `strrchr(const char*, int)` is replaced by the two declarations:

```
const char* strrchr(const char* s, int c);
      char* strrchr(      char* s, int c);
```

9 both of which have the same behavior as the original declaration.

10 The function signature `strstr(const char*, const char*)` is replaced by the two declarations:

```
const char* strstr(const char* s1, const char* s2);
      char* strstr(      char* s1, const char* s2);
```

11 both of which have the same behavior as the original declaration.

12 The function signature `memchr(const void*, int, size_type)` is replaced by the two declarations:

```
const void* memchr(const void* s, int c, size_type n);
      void* memchr(      void* s, int c, size_type n);
```

13 both of which have the same behavior as the original declaration. *SEE ALSO:* ISO C subclauses 7.3, 7.10.7, 7.10.8, and 7.11. Amendment 1 subclauses 4.4, 4.5, and 4.6.

22 Localization library [lib.localization]

1 This clause describes components that C++ programs may use to encapsulate (and therefore be largely independent of) cultural differences. The locale facility includes internationalization support for character classification and collation, numeric and currency punctuation, date and time formatting, and message retrieval.

2 The following subclauses describe components for locales, standard facets, and facilities from the ISO C library, as summarized in Table 46:

Table 46—Localization library summary

Subclause	Header(s)
22.1 Locales 22.2 Standard `locale` facets	`<locale>`
22.3 C library locales	`<clocale>`

22.1 Locales [lib.locales]

Header `<locale>` synopsis

```
#include <limits>
#include <string>
#include <iosfwd>
namespace std {
// subclause 22.1.1, locale:
  class locale;
  template <class charT>
    basic_ostream<charT>& operator<<(basic_ostream<charT>& s, const locale& loc);
  template <class charT>
    basic_istream<charT>& operator>>(basic_istream<charT>& s, locale& loc);

// subclause 22.1.2, convenience interfaces:
  template <class charT> bool isspace (charT c, const locale& loc) const;
  template <class charT> bool isprint (charT c, const locale& loc) const;
  template <class charT> bool iscntrl (charT c, const locale& loc) const;
  template <class charT> bool isupper (charT c, const locale& loc) const;
  template <class charT> bool islower (charT c, const locale& loc) const;
  template <class charT> bool isalpha (charT c, const locale& loc) const;
  template <class charT> bool isdigit (charT c, const locale& loc) const;
  template <class charT> bool ispunct (charT c, const locale& loc) const;
  template <class charT> bool isxdigit(charT c, const locale& loc) const;
  template <class charT> bool isalnum (charT c, const locale& loc) const;
  template <class charT> bool isgraph (charT c, const locale& loc) const;
  template <class charT> charT toupper(charT c, const locale& loc) const;
  template <class charT> charT tolower(charT c, const locale& loc) const;
```

```
// subclauses 22.2.1 and 22.2.2, ctype:
  class ctype_base;
  template <class charT> class ctype;
                         class ctype<char>;        // specialization
  template <class charT> class ctype_byname;
                         class ctype_byname<char>; // specialization

// subclauses 22.2.4 and 22.2.5, numeric:
  template <class charT, class InputIterator>  class num_get;
  template <class charT, class OutputIterator> class num_put;
  template <class charT> class numpunct;
  template <class charT> class numpunct_byname;

// subclause 22.2.6, collation:
  template <class charT> class collate;
  template <class charT> class collate_byname;

// subclause 22.2.7, codeset conversion:
  class codecvt_base;
  template <class fromT, class toT, class stateT> class codecvt;
  template <class fromT, class toT, class stateT> class codecvt_byname;

// subclause 22.2.8, date and time:
  class time_base;
  template <class charT, class InputIterator>  class time_get;
  template <class charT, class InputIterator>  class time_get_byname;
  template <class charT, class OutputIterator> class time_put;
  template <class charT, class OutputIterator> class time_put_byname;

// subclauses 22.2.9 and 22.2.10, money:
  class money_base;
  template <class charT, class InputIterator>  class money_get;
  template <class charT, class OutputIterator> class money_put;
  template <class charT> class moneypunct;
  template <class charT> class moneypunct_byname;

// subclause 22.2.11, message retrieval:
  template <class charT> class messages;
  template <class charT> class messages_byname;
}
```

1 The header `<locale>` defines classes and declares functions that encapsulate and manipulate the information peculiar to a locale.[147)]

22.1.1 Class `locale` [lib.locale]

```
namespace std {
  class locale {
  public:
    class facet;
    class id;
    typedef unsigned category;
    static const category
      collate  = 0x010, ctype    = 0x020,
      monetary = 0x040, numeric  = 0x080,
      time     = 0x100, messages = 0x200, all = 0x3f0;
```

147) In this subclause, the type name `struct tm` is an incomplete type that is defined in `<ctime>`.

```
    locale();
    locale(const locale& other);
    locale(const char* std_name);
    locale(const locale& other, const char* std_name, category);
    template <class Facet>
      locale(const locale& other, Facet* f);
    template <class Facet>
      locale(const locale& other, const locale& one);
    locale(const locale& other, const locale& one, category);
   ~locale();  // non-virtual
    const locale& operator=(const locale& other);

    template <class Facet> const Facet& use()  const;
    template <class Facet> bool         has()  const;
    const basic_string<char>&           name() const;

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

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

    static       locale  global(const locale&);
    static const locale& classic();
    static const locale  transparent()
  };
}
```

1 A `locale` constructed from a name string (such as ""), or from parts of two named locales, or read from a stream, has a name; all others do not. Named locales may be compared for equality; an unnamed locale is equal only to (copies of) itself. For an unnamed locale, `locale::name()` returns the string `*`.

2 A facet has a dual role: in one sense, it's just a class interface; at the same time, it's an index into a locale's set of facets. Access to the facets of a `locale` is via two member function templates, `locale::use<facet>()` and `locale::has<facet>()`.

3 For example, the standard iostream `operator<<` might be implemented as:

```
ostream& operator<<(ostream& s, double f)
{
  locale loc = s.rdloc();
  if (s.opfx()) {
    loc.use< num_put<char> >().put(s, s, loc, f);
    s.osfx();
  }
  return s;
}
```

In the call to `l.use<...>()`, the type argument chooses a facet, making available all members of the named type. If the named facet is not present in a locale (or, failing that, in the global locale), it throws the standard exception `bad_cast`. You can check if a locale implements a particular facet with the member `has<...>()`. User-defined facets may be bestowed on a locale, and used the same way as standard facets.

4 All locale semantics are accessed via `use<>()` and `has<>()`, with one exception: a member template function `operator()(basic_string<T>&, basic_string<T>&)` is provided so that a locale may be used as a predicate argument to the standard collections, to order strings. In addition, convenient global interfaces are provided for traditional ctype functions such as `isdigit()` and `isspace()`, so that given a locale object *`loc`* you can say `isspace(c,loc)`. These are provided to ease the conversion of existing extractors.

5 A locale which does not implement a facet delegates to the global locale.

6 The effect of imbuing on a stream, or installing as the global locale, the result of static member `locale::transparent()` (or any locale with similar behavior) is undefined.

22.1.1.1 `locale` types [lib.locale.types]

22.1.1.1.1 Type `locale::category` [lib.locale.category]

```
typedef unsigned category;
```

Notes: Uses values as defined in `<clocale>`, e.g. `LC_CTYPE`.

22.1.1.1.2 Class `locale::facet` [lib.locale.facet]

```
namespace std {
  class locale::facet {
    facet(const facet&);          // not defined
    void operator=(const facet&); // not defined
    void operator&() {}           // not usable

  protected:
    facet(size_t refs = 0);
    virtual ~facet();
  };
}
```

1 Base class for locale feature sets. Any class deriving from facet must declare a *static* member:

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

2 For some standard facets there is a ‘‘ `_byname`’’ class derived from it that implements POSIX locale semantics; they are specified by name in the standard to allow users to derive from them. If there is no ‘‘ `_byname`’’ version, the base class implements POSIX semantics itself, sometimes with the help of another facet.

22.1.1.1.3 Class `locale::id` [lib.locale.ctype.id]

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

  public:
    id();
  };
}
```

1 Identification of a locale facet interface, used as an index for lookup.

22.1.1.2 `locale` constructors [lib.locale.cons]

```
locale();
```

1 Default constructor: the current global locale.
Effects: `: imp_(global_ ? global_ : init()) { imp_->add_ref(); }`

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

Effects: `{ imp_ = other.imp_; imp_->add_ref(); }`

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

Effects:

```
if (imp_ != other.imp_) {
  imp_->rem_ref(); imp_ = other.imp_; imp_->add_ref();
}
```

Returns: `*this`

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

Effects: Constructs a locale using standard C locale names, e.g. "POSIX".

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

Effects: Makes a locale composed of ‘‘ `byname` ’’ facets, and assigns a name. Copies its first argument except for a specified component, which is constructed on the spot. If *`other`* has a name, the new locale does also.

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

2 To accrete or replace facet

Effects: Copies all facets but one from the first argument, installs the other argument as the remaining facet. The resulting locale has no name.

```
: imp_(new imp(*other.imp_, 1))
{
  f->add_ref();
  install(f, Facet::id, "*");
}
```

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

3 To replace a facet

Effects: Copies all facets but one from the first argument, and the remaining facet from the second argument. The resulting locale has a no name.

```
: imp_(new imp(*other.imp_, 1))
{
  facet* f = (Facet*) one.imp_->vec[Facet::id];  // check derivation
  install(f, Facet::id, merge_names(Facet::id, one.imp_->name_));
}
```

```
locale(const locale& other, const locale& one, categories);
```

Effects: Copies all facets from the first argument except those that implement the specified categories, which are copied from the second argument. The resulting locale has a name only if both locale arguments have names.

22.1.1.3 `locale` members [lib.locale.members]

22.1.1.3.1 `locale::use` [lib.locale.use]

```
template <class Facet> const Facet& use() const;
```

Effects:

```
int i = (const id&) Facet::id;    // verify is a locale::id.
facet* f = (Facet*) 0;            // verify derived from facet.
return static_cast<const Facet&>(
  (i < imp_->vec_.size() && (f = imp_->vec_[i])) ? *f : delegate(i));
```

Returns: A reference to a facet. If the facet requested is not present in *this, it returns the facet from the current global locale, if present there, or throws an exception.
Its result is guaranteed by locale's value semantics to last at least as long as the locale it came from.

22.1.1.3.2 `locale::has` [lib.locale.has]

```
template <class Facet> bool has() const;
```

Effects: Reports whether a locale implements a particular facet.

```
facet* null = (Facet*) 0;          // verify derived from facet.
size_t ix = (const id&) Facet::id; // verify is a locale::id.
```

Returns: `(ix < imp_->vec_.size()) && imp_->vec_[ix];`

1 If `(loc.has<facet>() || locale().has<facet>())` is false, `loc.use<facet>()` would throw an exception.

22.1.1.3.3 `locale::name` [lib.locale.name]

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

Effects:

```
{ return imp_->name_; }
```

22.1.1.4 `locale` operators [lib.locale.operators]

22.1.1.4.1 `locale::operator==` [lib.locale.op==]

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

Returns:

```
(imp_ == other.imp_) ||
(name() != "*" && name() == other.name())
```

22.1.1.4.2 `locale::operator!=` [lib.locale.op!=]

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

eturns: `!(*this == other)`

22.1.1.4.3 `locale::operator()` [lib.locale.op()]

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

1 This template function satisfies requirements for a comparator predicate template argument (25).

22.1.1.4.4 `operator<<` [lib.locale.op<<]

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

Returns: `s << loc.name() << endl`

22.1.1.4.5 `operator>>` [lib.locale.op>>]

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

Effects: Read a line, construct a locale, throw exception if cannot.

22.1.1.5 `locale` static members [lib.locale.statics]

22.1.1.5.1 `locale::global` [lib.locale.global]

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

1 Replaces `::setlocale(...)`.

22.1.1.5.2 `locale::classic` [lib.locale.classic]

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

1 The `locale`.

22.1.1.5.3 `locale::transparent` [lib.locale.transparent]

```
static const locale transparent();
```

1 Continuously updated global locale.
Returns: `locale(new imp(1, 0, true))`

22.1.2 Convenience interfaces [lib.locale.convenience]

22.1.2.1 Character classification [lib.classification]

22.1.2.1.1 `isspace` [lib.locale.isspace]

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

Returns: `loc.use<ctype<charT> >().is(ctype<charT>::space, c)`.

22.1.2.1.2 isprint [lib.locale.isprint]

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

Returns: loc.use<ctype<charT> >().is(ctype<charT>::print, c).

22.1.2.1.3 iscntrl [lib.locale.iscntrl]

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

Returns: loc.use<ctype<charT> >().is(ctype<charT>::cntrl, c).

22.1.2.1.4 isupper [lib.locale.isupper]

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

Returns: loc.use<ctype<charT> >().is(ctype<charT>::upper, c).

22.1.2.1.5 islower [lib.locale.islower]

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

Returns: loc.use<ctype<charT> >().is(ctype<charT>::lower, c).

22.1.2.1.6 isalpha [lib.locale.isalpha]

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

Returns: loc.use<ctype<charT> >().is(ctype<charT>::alpha, c).

22.1.2.1.7 isdigit [lib.locale.isdigit]

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

Returns: loc.use<ctype<charT> >().is(ctype<charT>::digit, c).

22.1.2.1.8 ispunct [lib.locale.ispunct]

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

Returns: loc.use<ctype<charT> >().is(ctype<charT>::punct, c).

22.1.2.1.9 isxdigit [lib.locale.isxdigit]

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

Returns: loc.use<ctype<charT> >().is(ctype<charT>::xdigit, c).

22.1.2.1.10 isalnum [lib.locale.isalnum]

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

Returns: loc.use<ctype<charT> >().is(ctype<charT>::alnum, c).

22.1.2.1.11 `isgraph` [lib.locale.isgraph]

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

Returns: `loc.use<ctype<charT> >().is(ctype<charT>::graph, c)`.

22.1.2.2 Character conversiona [lib.conversions]

22.1.2.2.1 `toupper` [lib.locale.toupper]

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

Returns: `loc.use<ctype<charT> >().toupper(c)`.

22.1.2.2.2 `tolower` [lib.locale.tolower]

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

Returns: `loc.use<ctype<charT> >().tolower(c)`.

22.2 Standard `locale` facets [lib.std.facets]

22.2.1 The `ctype` facet [lib.facet.ctype]

```
namespace std {
  struct ctype_base {
    enum ctype_mask {
      space=1<<0, print=1<<1, cntrl=1<<2, upper=1<<3, lower=1<<4,
      alpha=1<<5, digit=1<<6, punct=1<<7, xdigit=1<<8,
      alnum=(1<<5)|(1<<6), graph=(1<<7)|(1<<6)|(1<<5)
    };
  };
```

Box 97

ISSUE: should this be in the global namespace?

```
  ctype_base::ctype_mask
    operator|(ctype_base::ctype_mask a, ctype_base::ctype_mask b);
        { return (ctype_base::ctype_mask)(unsigned(a)|unsigned(b)); }
  template <class charT> class ctype;
  template <class charT> class ctype_byname
}
```

22.2.1.1 Template class `ctype` [lib.locale.ctype]

```
  template <class charT>
  class ctype : public locale::facet, public ctype_base {
  public:
    typedef charT char_type;
```

```
protected:
  virtual bool         do_is(ctype_mask mask, charT c) const;
  virtual const charT* do_is(const charT* low,
                             const charT* high, ctype_mask* vec) const;
  virtual const char*  do_scan_is(ctype_mask mask,
                                  const charT* low, const charT* high) const;
  virtual const char*  do_scan_not(ctype_mask mask,
                                   const charT* low, const charT* high) const;
  virtual charT        do_toupper(charT) const;
  virtual const charT* do_toupper(charT* low, const charT* high) const;
  virtual charT        do_tolower(charT) const;
  virtual const charT* do_tolower(charT* low, const charT* high) const;
  virtual charT        do_widen(char) const;
  virtual const char*  do_widen(const char* lo,
                                const char* hi, charT* dest) const;
  virtual char         do_narrow(charT, char dfault) const;
  virtual const charT* do_narrow(const charT* lo, const charT* hi,
                                 char dfault, char* dest) const;

public:
  bool         is(ctype_mask mask, charT c) const;
  const charT* is(const charT* low, const charT* high, ctype_mask* vec) const;
  const charT* scan_is(ctype_mask mask, const charT* low,
                       const charT* high) const;
  const charT* scan_not(ctype_mask mask, const charT* low,
                        const charT* high) const;
  charT        toupper(charT) const;
  const charT* toupper(charT* low, const charT* high) const;
  charT        tolower(charT c) const;
  const charT* tolower(charT* low, const charT* high) const;

  charT       widen(char c) const;
  const char* widen(const char* lo, const char* hi, charT* to) const;
  char         narrow(charT c, char dfault) const;
  const charT* narrow(const charT* lo, const charT*,
                      char dfault, char* to) const;

  static locale::id id;
  ctype(size_t refs = 0);
protected:
 ~ctype();
};
```

1 Class `ctype` encapsulates the C library `<cctype>` features. `istream` members are required to use `ctype<>` for character classing.

22.2.1.2 `ctype` members [lib.locale.ctype.members]

1 Most of these functions are

Box 98

To Be Specified

22.2.1.2.1 do_is [lib.locale.ctype::do.is]

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

22.2.1.2.2 do_scan_is [lib.locale.ctype::do.scan.is]

```
virtual const char*
  do_scan_is(ctype_mask mask,
         const charT* low, const charT* high) const;
```

22.2.1.2.3 do_scan_not [lib.locale.ctype::do.scan.not]

```
virtual const char*
  do_scan_not(ctype_mask mask,
          const charT* low, const charT* high) const;
```

22.2.1.2.4 do_toupper [lib.locale.ctype::do.toupper]

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

22.2.1.2.5 do_tolower [lib.locale.ctype::do.tolower]

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

22.2.1.2.6 do_widen [lib.locale.ctype::do.widen]

```
virtual charT        do_widen(char) const;
virtual const char*  do_widen(const char* lo,
                  const char* hi, charT* dest) const;
```

22.2.1.2.7 do_narrow [lib.locale.ctype::do.narrow]

```
virtual char         do_narrow(charT, char dfault) const;
virtual const charT* do_narrow(const charT* lo, const charT* hi,
                   char dfault, char* dest) const;
```

22.2.1.2.8 is [lib.locale.ctype::is]

```
bool         is(ctype_mask mask, charT c) const;
const charT* is(const charT* low, const charT* high, ctype_mask* vec) const;
```

Returns: do_is(*mask*,*c*) or do_is(*low*,*high*,*vec*)

22.2.1.2.9 scan_is [lib.locale.ctype::scan.is]

```
const charT* scan_is(ctype_mask mask, const charT* low,
            const charT* high) const;
```

Returns: do_scan_is(*mask*low,*high*)

22.2.1.2.10 scan_not [lib.locale.ctype::scan.not]

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

Returns: do_scan_not(*mask*,*low*,*high*)

22.2.1.2.11 toupper [lib.locale.ctype::toupper]

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

Returns: do_toupper(*c*) or do_toupper(*low*,*high*)

22.2.1.2.12 tolower [lib.locale.ctype::tolower]

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

Returns: do_tolower(*c*) or do_tolower(*low*,*high*)

22.2.1.2.13 widen [lib.locale.ctype::widen]

```
charT       widen(char c) const;
const char* widen(const char* lo, const char* hi, charT* to) const;
```

Returns: do_widen(*c*) or do_widen(*lo*,*hi*,*to*)

22.2.1.2.14 narrow [lib.locale.ctype::narrow]

```
char         narrow(charT c, char dfault) const;
const charT* narrow(const charT* lo, const charT*,
                    char dfault, char* to) const;
```

Returns: do_narrow(*c*,*dfault*) or do_narrow(*lo*,*hi*,*dfault*,*to*)

22.2.1.2.15 ctype constructor [lib.locale.ctype::cons]

```
ctype(size_t refs = 0);
```

Effects:

```
 : locale::facet(refs) {}
```

22.2.1.3 Template class ctype_byname [lib.locale.ctype.byname]

```
  template <class charT>
  class ctype_byname : public ctype<charT> {
    // this class is specialized by vendors for char and wchar_t.
  protected:
    virtual char        do_toupper(char) const;
    virtual const char* do_toupper(char* low, const char* high) const;
    virtual char        do_tolower(char) const;
    virtual const char* do_tolower(char* low, const char* high) const;
  public:
    ctype_byname(const char*, size_t refs = 0);
  protected:
   ~ctype_byname();
  };
}
```

22.2.2 `ctype` specializations [lib.facet.ctype.special]

```
namespace std {
  class ctype<char> : public locale::facet, public ctype_base {
  public:
    typedef char char_type;

  protected:
    const ctype_mask* const table_;
    static const ctype_mask
      classic_table_[numeric_limits<unsigned char>::max()+1];

    virtual char        do_toupper(char) const;
    virtual const char* do_toupper(char* low, const char* high) const;
    virtual char        do_tolower(char) const;
    virtual const char* do_tolower(char* low, const char* high) const;

  public:
    bool is(ctype_mask mask, char c) const;
    const char* is(const char* lo, const char* hi,
                   ctype_mask* vec) const;
    const char* scan_is(ctype_mask mask, const char* low,
                        const char* high) const;
    const char* scan_not(ctype_mask mask, const char* low,
                         const char* high) const;

    char        toupper(char c) const;
    const char* toupper(char* low, const char* high) const;
    char        tolower(char c) const;
    const char* tolower(char* low, const char* high) const;

    char  widen(char c) const;
    const char* widen(const char* lo, const char* hi,
                      char* to) const;
    char   narrow(char c, char /*dfault*/) const;
    const char* narrow(const char* lo, const char* hi,
                       char /*dfault*/, char* to) const;
```

```
    static locale::id id;
    ctype(const ctype_mask* tab = 0, bool del = false,
          size_t refs = 0);
  protected:
   ~ctype();
  };
}
```

1 A specialization ctype<char> is provided, so that the member functions on type char may be implemented inline. Definitions of these functions have been provided for exposition. Only the char, and not the unsigned char and signed char forms, has been provided. The specialization is specified in the standard (and not left as an implementation detail) because it affects the derivation interface for ctype<char>.

22.2.3 ctype<char> members [lib.facet.ctype.char.members]

22.2.3.1 is [lib.ctype.char::is]

```
bool is(ctype_mask mask, char c) const;
```

Returns: table_[(unsigned char)c] & mask

```
const char* is(const char* lo, const char* hi,
 ctype_mask* vec) const;
```

Effects: while (lo != hi) *vec++ = table_[(unsigned char)*lo++]; return lo;

22.2.3.2 scan_is [lib.ctype.char::scan.is]

```
const char* scan_is(ctype_mask mask, const char* low,
onst char* high) const;
```

Effects:

```
while (low != high && !(table_[(unsigned char) *low] & mask)) { ++low; }
return low;
```

22.2.3.3 scan_not [lib.ctype.char::scan.not]

```
const char* scan_not(ctype_mask mask, const char* low,
const char* high) const;
```

Effects:

```
while (low != high && (table_[(unsigned char)*low] & mask)) { ++low; }
return low;
```

22.2.3.4 toupper [lib.ctype.char::toupper]

```
char        toupper(char c) const;
const char* toupper(char* low, const char* high) const;
```

Returns: do_toupper(*c*) or do_toupper(*low*,*high*)

22.2.3.5 tolower [lib.ctype.char::tolower]

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

Returns: `do_tolower(c)` or `do_tolower(low,high)`

22.2.3.6 widen [lib.ctype.char::widen]

```
char  widen(char c) const;
const char* widen(const char* lo, const char* hi,
    char* to) const;
```

Returns: or `memcpy(to, lo, hi-lo); return hi`

22.2.3.7 narrow [lib.ctype.char::narrow]

```
char   narrow(char c, char /*dfault*/) const;
const char* narrow(const char* lo, const char* hi,
                   char /*dfault*/, char* to) const;
```

Returns: c or `memcpy(to, lo, hi-lo); return hi;`

22.2.3.8 ctype<char> [lib.ctype.char::ctor]

```
ctype(const ctype_mask* tab = 0, bool del = false,
      size_t refs = 0);
```

Effects:

```
      : locale::facet(refs), table_(tab ? tab : classic_table_),
        delete_it_(tab ? del : false) {}
```

22.2.3.9 ctype<char> destructor [lib.ctype.char::dtor]

```
~ctype();
```

Effects: `if (delete_it_) delete table_;`

22.2.4 The numeric facet [lib.facet.numeric]

22.2.4.1 Template class num_get [lib.locale.num.get]

```
namespace std {
  template <class charT, class InputIterator = istreambuf_iterator<charT> >
  class num_get : public locale::facet {
  public:
    typedef charT           char_type;
    typedef InputIterator   iter_type;
    typedef basic_ios<charT> ios;
```

```
  protected:
    virtual iter_type do_get(iter_type, ios&, const locale&,
                             bool& v) const;
    virtual iter_type do_get(iter_type, ios&, const locale&,
                             long& v) const;
    virtual iter_type do_get(iter_type, ios&, const locale&,
                             unsigned long& v) const;
//  virtual iter_type do_get(iter_type, ios&, const locale&,
//                           long long& v) const;
    virtual iter_type do_get(iter_type, ios&, const locale&,
                             double& v) const;
    virtual iter_type do_get(iter_type, ios&, const locale&,
                             long double& v) const;

  public:
    iter_type get(iter_type s, ios& f, const locale&, bool& v)          const;
    iter_type get(iter_type s, ios& f, const locale&, long& v)          const;
    iter_type get(iter_type s, ios& f, const locale&, unsigned long& v) const;
//  iter_type get(iter_type s, ios& f, const locale&, long long& v)   const;
    iter_type get(iter_type s, ios& f, const locale&, double& v)        const;
    iter_type get(iter_type s, ios& f, const locale&, long double& v)   const;
    static locale::id id;
    num_get(size_t refs = 0);
  protected:
   ~num_get();
  };
}
```

1 The classes num_get<> and num_put<> handle numeric formatting and parsing. Virtual functions are provided for several numeric types; implementations are allowed to delegate conversion of smaller types to extractors for larger ones, but are not required to. The functions take a locale argument because their base class implementation refers to numpunct features, which identify preferred numeric punctuation. Extractors and inserters for the standard iostreams are required to call num_get and num_put member functions. The ios& argument is used both for format control and to report errors.

2 Members of num_get take a locale argument because they may need to refer to the locale's numpunct facet.

22.2.4.2 Template class num_put [lib.locale.num.put]

```
namespace std {
  template <class charT, class OutputIterator = ostreambuf_iterator<charT> >
  class num_put : public locale::facet {
  public:
    typedef charT            char_type;
    typedef OutputIterator   iter_type;
    typedef basic_ios<charT> ios;

  protected:
    virtual iter_type do_put(iter_type, ios&, const locale&, bool v) const;
    virtual iter_type do_put(iter_type, ios&, const locale&, long v) const;
    virtual iter_type do_put(iter_type, ios&, const locale&, unsigned long) const;
//virtual iter_type do_put(iter_type, ios&, const locale&, long long v) const;
    virtual iter_type do_put(iter_type, ios&, const locale&, double v) const;
    virtual iter_type do_put(iter_type, ios&, const locale&, long double v) const;
```

```
  public:
    iter_type put(iter_type s, ios& f, const locale& loc,
                  bool v
5)          const;
    iter_type put(iter_type s, ios& f, const locale& loc,
                  long v) const;
    iter_type put(iter_type s, ios& f, const locale& loc,
                  unsigned long v) const;
//  iter_type put(iter_type s, ios& f, const locale& loc,
                  long long v) const;
    iter_type put(iter_type s, ios& f, const locale& loc,
                  double v) const;
    iter_type put(iter_type s, ios& f, const locale& loc,
                  long double v) const;
    static locale::id id;
    num_put(size_t refs = 0);
  protected:
   ~num_put();
  };
}
```

1 Members of num_put take a locale argument because they may need to refer to the locale's numpunct facet. The ios& argument is used for formatting reference only and error reporting.

22.2.5 The numeric punctuation facet [lib.facet.numpunct]

22.2.5.1 Template class numpunct [lib.locale.numpunct]

```
namespace std {
  template <class charT>
  class numpunct : public locale::facet {
  public:
    typedef charT              char_type;
    typedef basic_string<charT> string;

  protected:
    virtual string       do_decimal_point() const;
    virtual string       do_thousands_sep() const;
    virtual vector<char> do_grouping()      const;
    virtual string       do_truename()      const;  // for bool
    virtual string       do_falsename()     const;  // for bool

  public:
    string decimal_point()   const { return do_decimal_point(); }
    string thousands_sep()   const { return do_thousands_sep(); }
    vector<char>  grouping() const { return do_grouping(); }
    string truename()        const { return do_truename(); }
    string falsename()       const { return do_falsename(); }
    static locale::id id;
    numpunct(size_t refs = 0);
  protected:
   ~numpunct();
  };
}
```

1 numpunct<> specifies numeric punctuation. The base class provides classic numeric formats, while the _byname version supports general POSIX numeric formatting semantics.

22.2.5.2 Template class numpunct_byname [lib.locale.numpunct.byname]

```
namespace std {
  template <class charT>
  class numpunct_byname : public numpunct<charT> {
  // this class is specialized by vendors for char and wchar_t.
  protected:
    virtual string       do_decimal_point() const;
    virtual string       do_thousands_sep() const;
    virtual vector<char> do_grouping()      const;
    virtual string       do_truename()      const;  // for bool
    virtual string       do_falsename()     const;  // for bool
  public:
    numpunct_byname(const char*, size_t refs = 0);
  protected:
   ~numpunct_byname();
  };
}
```

1 numpunct is used by num_get and num_put facets.

22.2.6 The collation facet [lib.facet.collate]

22.2.6.1 Template class collate [lib.locale.collate]

```
namespace std {
  template <class charT>
  class collate : public locale::facet {
  public:
    typedef charT              char_type;
    typedef basic_string<charT> string;
  protected:
    virtual int    do_compare(const char* low1, const char* high1,
                              const char* low2, const char* high2) const;
    virtual string do_transform(const char* low, const char* high) const;
    virtual long   do_hash(     const char* low, const char* high) const;

  public:
    int compare(const char* low1, const char* high1,
                const char* low2, const char* high2) const;
      { return do_compare(low1, hight1, low2, high2); }
    string transform(const char* low, const char* high) const;
      { return do_transform(low, high); }
    long hash(const char* low, const char* high) const;
      { return do_hash(low, high); }
    static locale::id id;
    collate(size_t refs = 0);
  protected:
   ~collate();
  };
}
```

1 The class collate<charT> provides features for use in the collation of strings. A locale member function template, operator(), uses the collate facet to allow a locale to act directly as the predicate argument for algorithms. The base class uses lexicographic ordering.

2 locale.use<collate>() is used for string comparisons.

22.2.6.2 `collate` members [lib.locale.collate.members]

22.2.6.2.1 `compare` [lib.collate::compare]

```
int compare(const char* low1, const char* high1,
            const char* low2, const char* high2) const;
```

Returns: `do_compare(low1, hight1, low2, high2)`

22.2.6.2.2 `transform` [lib.collate::transform]

```
string transform(const char* low, const char* high) const;
```

Returns: `do_transform(low, high)`

22.2.6.2.3 `hash` [lib.collate::hash]

```
long hash(const char* low, const char* high) const;
```

Returns: `do_hash(low, high)`

22.2.6.3 Template class `collate_byname` [lib.locale.collate.byname]

```
namespace std {
  template <class charT>
  class collate_byname : public collate<charT> {
    // this class is specialized by vendors for char and wchar_t.
  protected:
    virtual int    do_compare(const char* low1, const char* high1,
                              const char* low2, const char* high2) const;
    virtual string do_transform(const char* low, const char* high) const;
    virtual long   do_hash(     const char* low, const char* high) const;
  public:
    collate_byname(const char*);
  protected:
   ~collate_byname();
  };
//  this template function satisfies requirements for a
//  comparator predicate template argument:
  template <class charT>
    bool locale::operator()(const basic_string<charT>& s1,
                            const basic_string<charT>& s2);
}
```

22.2.6.4 `collate_byname` members [lib.locale.collate.byname.members]

22.2.6.4.1 `operator()` [lib.collate::op.paren]

```
bool locale::operator()(const basic_string<charT>& s1,
                        const basic_string<charT>& s2);
```

Returns:

```
  use< collate<charT> >().compare(s1.begin(), s1.end(),
                                  s2.begin(), s2.end()) < 0;
```

22.2.7 The codeset conversion facet [lib.facet.codecvt]

22.2.7.1 Template class `codecvt` [lib.locale.codecvt]

```
namespace std {
  struct codecvt_base {
    enum result { ok, partial, error };
  };
  template <class fromT, class toT, class stateT>
  class codecvt : public locale::facet, public codecvt_base {
  public:
    typedef fromT  from_type;
    typedef toT    to_type;
    typedef stateT state_type;
  protected:
    virtual result do_convert(stateT& state,
      const fromT* from, const fromT* from_end, const fromT*& from_next,
            toT*   to,   toT*         to_limit,       toT*&   to_next) const;

  public:
     result convert(stateT& state,
       const fromT* from, const fromT* from_end, const fromT*& from_next,
                toT*   to,         toT* to_limit,         toT*& to_next) const;
    static locale::id id;
    codecvt(size_t refs = 0)
  protected:
   ~codecvt() {}
  };
}
```

1 The class `codecvt<fromT,toT,stateT>` is for use when converting from one codeset to another, such as from wide characters to multibyte characters, or between wide character sets such as Unicode and EUC. Instances of this facet are typically used in pairs.

2 Its only member function, `convert()`, returns an enumeration value which indicates whether it completed the conversion (`ok`), ran out of space in the destination (`partial`), or encountered a `from_type` character it could not convert (`error`).

3 In all cases it leaves the *`from_next`* and *`to_next`* pointers pointing one beyond the last character successfully converted.

4 The `stateT` argument selects the pair of codesets being mapped between. Base class members are pure virtual.

5 Implementations are obliged to provide specializations for `<char,wchar_t,mbstate_t>` and `<wchar_t,char,mbstate_t>`.

22.2.7.2 `codecvt` members [lib.locale.codecvt.members]

22.2.7.2.1 `convert` [lib.codecvt::convert]

```
result convert(stateT& state,
          const fromT* from, const fromT* from_end,
          const fromT*& from_next,
          toT*   to,         toT* to_limit,
          toT*& to_next) const;
```

Returns:

```
  do_convert(state, from, from_end, from_next,
             to, to_limit, to_next);
```

22.2.7.3 Template class codecvt_byname [lib.locale.codecvt.byname]

```
namespace std {
  template <class fromT, class toT, class stateT>
  class codecvt_byname : public codecvt<fromT, toT, stateT> {
  protected:
    virtual result do_convert(stateT& state,
      const fromT* from, const fromT* from_end, const fromT*& from_next,
            toT*   to,   toT*         to_limit,       toT*&   to_next) const;
  public:
    codecvt_byname(const char*, size_t refs = 0);
  protected:
   ~codecvt_byname();
  };
}
```

22.2.8 The date and time facet [lib.facet.date.time]

1 The classes time_get<charT> and time_put<charT> provide date and time formatting and parsing. The time formatting function put() takes an extra format argument to accommodate the POSIX strftime() extensions. The ios& argument is used for format control and to report errors.

22.2.8.1 Template class time_get [lib.locale.time.get]

```
namespace std {
  struct time_base {
    enum dateorder { no_order, dmy, mdy, ymd, ydm };
  };
  template <class charT, class InputIterator = istreambuf_iterator<charT> >
  class time_get : public locale::facet, public time_base {
  public:
    typedef charT            char_type;
    typedef InputIterator    iter_type;
    typedef basic_ios<charT> ios;

  protected:
    virtual dateorder do_date_order()  const;
    virtual iter_type do_get_time(iter_type s, ios&, const locale&, tm* t) const;
    virtual iter_type do_get_date(iter_type s, ios&, const locale&, tm* t) const;
    virtual iter_type do_get_weekday(iter_type s, ios&, const locale&, tm* t)
                    const;
    virtual iter_type do_get_monthname(iter_type s, ios&, const locale&, tm* t)
                    const;
    virtual iter_type do_get_year(iter_type s, ios&, const locale&, tm* t) const;
```

22.2.8.2 Template class time_get_byname [lib.locale.time.get.byname]

```
namespace std {
  template <class charT, class InputIterator = istreambuf_iterator<charT> >
  class time_get_byname : public time_get<charT, InputIterator> {
  protected:
    virtual dateorder do_date_order()  const;
    virtual iter_type do_get_time(iter_type s, ios&, const locale&, tm* t) const;
    virtual iter_type do_get_date(iter_type s, ios&, const locale&, tm* t) const;
    virtual iter_type do_get_weekday(iter_type s, ios&, const locale&, tm* t) const;
    virtual iter_type do_get_monthname(iter_type s, ios&, const locale&, tm* t) const;
    virtual iter_type do_get_year(iter_type s, ios&, const locale&, tm* t) const;
  public:
    time_get_byname(const char*, size_t refs = 0);
  protected:
   ~time_get_byname();
  };
}
```

22.2.8.3 Template class `time_put` [lib.locale.time.put]

```
namespace std {
  template <class charT, class OutputIterator = ostreambuf_iterator<charT> >
  class time_put : public locale::facet {
  public:
    typedef charT            char_type;
    typedef OutputIterator   iter_type;
    typedef basic_ios<charT> ios;

  protected:
    virtual iter_type do_put(iter_type s, ios&, const locale&, const tm* t,
                             char format, char modifier) const;

  public:
    // the following is implemented in terms of other member functions.
    iter_type put(iter_type s, ios& f, struct tm const* tmb,
                  const charT* pattern, const charT* pat_end) const;
    iter_type put(iter_type s, ios& f, const locale& loc, struct tm const*
Pt,
                  char format, char modifier = ' ') const;
    static locale::id id;
    time_put(size_t refs = 0);
  protected:
   ~time_put();
  };
}
```

22.2.8.4 Template class `time_put_byname` [lib.locale.time.put.byname]

```
namespace std {
  template <class charT, class OutputIterator = ostreambuf_iterator<charT> >
  class time_put_byname : public time_put<charT, OutputIterator>
  {
  protected:
    virtual iter_type do_put(iter_type s, ios&, const locale&, const tm* t,
                             char format, char modifier) const;
  public:
    time_put_byname(const char*, size_t refs = 0);
  protected:
   ~time_put_byname();
  };
}
```

22.2.9 The money facet [lib.facet.money]

1 These handle money formats. A template parameter indicates whether local or international monetary formats are to be used. money_get<> and money_put<> use moneypunct<> features if appropriate. moneypunct<> provides basic format information for money processing. The ios& argument is used for format control and to report errors.

22.2.9.1 Template class money_get [lib.locale.money.get]

```
namespace std {
  template <class charT, bool Intl = false,
            class InputIterator = istreambuf_iterator<charT> >
  class money_get : public locale::facet {
    typedef charT               char_type;
    typedef InputIterator       iter_type;
    typedef basic_string<charT> string;
    typedef basic_ios<charT>    ios;

  protected:
    virtual iter_type do_get(iter_type, ios&, const locale&, double& units) const;
    virtual iter_type do_get(iter_type, ios&, const locale&, string& digits) const;

  public:
    iter_type get(iter_type s, ios& f, const locale& loc, double& units
) const;
    iter_type get(iter_type s, ios& f, const locale& loc, string& digit
s) const;
    static const bool intl = Intl;
    static locale::id id;
    money_get(size_t refs = 0);
   protected:
   ~money_get();
  };
}
```

22.2.9.2 Template class money_put [lib.locale.money.put]

```
namespce std {
  template <class charT, bool Intl = false,
            class OutputIterator = ostreambuf_iterator<charT> >
  class money_put : public locale::facet {
  public:
    typedef charT               char_type;
    typedef OutputIterator      iter_type;
    typedef basic_string<charT> string;
    typedef basic_ios<charT>    ios;

  protected:
    virtual iter_type
      do_put(iter_type, ios&, const locale&, double units) const;
    virtual iter_type
      do_put(iter_type, ios&, const locale&, const string& digits) const;
```

```
  public:
    iter_type put(iter_type s, ios& f, const locale& loc, double units&) const;
    iter_type put(iter_type s, ios& f, const locale& loc, const string&
6dgts) const;
    static const bool intl = Intl;
    static locale::id id;
    money_put(size_t refs = 0);
  protected:
   ~money_put();
  };
}
```

22.2.10 The money punctuation facet [lib.facet.moneypunct]

22.2.10.1 Template class `moneypunct` [lib.locale.moneypunct]

```
namespace std {
  struct money_base {
    enum part { symbol='$', sign='=', space=' ', value='v', none=0 };
    struct pattern { char field[4]; };
  };
  template <class charT, bool International = false>
  class moneypunct : public locale::facet, public money_base {
  public:
    typedef charT char_type;
    static const bool intl = International;
    typedef basic_string<charT> string;

  protected:
    virtual charT        do_decimal_point() const;
    virtual charT        do_thousands_sep() const;
    virtual vector<char> do_grouping()      const;
    virtual string       do_curr_symbol()   const;
    virtual string       do_positive_sign() const;
    virtual string       do_negative_sign() const;
    virtual int          do_frac_digits()   const;
    virtual pattern      do_pos_format()    const;
    virtual pattern      do_neg_format()    const;

  public:
    charT        decimal_point() const { return do_decimal_point(); }
    charT        thousands_sep() const { return do_thousands_sep(); }
    vector<char> grouping()      const { return do_grouping(); }
    string       curr_symbol()   const { return do_curr_symbol(); };
    string       positive_sign() const { return do_positive_sign; }
    string       negative_sign() const { return do_negative_sign; }
    int          frac_digits()   const { return do_frac_digits(); }
    pattern      pos_format()    const { return do_pos_format(); }
    pattern      neg_format()    const { return do_neg_format(); }
    static locale::id id;
    moneypunct(size_t refs = 0);
  protected:
   ~moneypunct() {}
  };
}
```

1 This provides money punctuation, similar to numpunct above.

22.2.10.2 Template class `moneypunct_byname` [lib.locale.moneypunct.byname]

```
namespace std {
  template <class charT, bool Intl = false>
  class moneypunct_byname : public moneypunct<charT, Intl> {
  protected:
    virtual charT        do_decimal_point() const;
    virtual charT        do_thousands_sep() const;
    virtual vector<char> do_grouping()      const;
    virtual string       do_curr_symbol()   const;
    virtual string       do_positive_sign() const;
    virtual string       do_negative_sign() const;
    virtual int          do_frac_digits()   const;
    virtual pattern      do_pos_format()    const;
    virtual pattern      do_neg_format()    const;
  public:
    moneypunct_byname(const char*, size_t refs = 0);
  protected:
   ~moneypunct_byname();
  };
}
```

22.2.11 The message retrieval facet [lib.facet.messages]

1 Class messages<charT> implements POSIX message retrieval.[148)]

22.2.11.1 Template class `messages` [lib.locale.messages]

```
namespace std {
  template <class charT>
  class messages : public locale::facet {
  public:
    typedef charT char_type;
    typedef int   catalog;
    typedef basic_string<charT> string;

  protected:
    virtual catalog do_open(const string&, const locale&) const;
    virtual string  do_get(catalog, int set, int msgid,
                           const string& dfault) const;
    virtual void    do_close(catalog) const;

  public:
    catalog open(const string& fn, const locale& l) const
      { return do_open(fn, l); }
    string get(catalog c, int set, int msgid, const string& dfault) const;
    void close(catalog c) const;

    static locale::id id;
    messages(size_t refs = 0);
  protected:
   ~messages();
  };
}
```

148) It should be flexible enough to retrieve messages from X, MS Windows, or Macintosh resource files as well.

22.2.11.2 Tmessages members [lib.locale.messages.members]

22.2.11.3 Template class messages_byname [lib.locale.messages.byname]

```
namespace std {
  template <class charT>
  class messages_byname : public messages<charT> {
  protected:
    virtual catalog do_open(const string&, const locale&) const;
    virtual string  do_get(catalog, int set, int msgid,
                           const string& dfault) const;
    virtual void    do_close(catalog) const;
  public:
    messages_byname(const char*, size_t refs = 0);
  protected:
   ~messages_byname();
  };
}
```

22.2.12 User-defined facets [lib.facets.examples]

1 A user-defined facet may be added to a locale and used identically as the built-in facets. To create a new facet interface users simply derive, from `locale::facet`, a class containing a static member: `static locale::id id;`. (The locale member function templates verify its type and storage class.)

2 For those curious about the mechanics: this initialization/identification system depends only on the initialization to 0 of static objects, before static constructors are called. When an instance of a facet is installed in a locale, the locale checks whether an id has been assigned, and if not, assigns one. Before this occurs, any attempted use of its interface causes the `bad_cast` exception to be thrown.

3 Here is a program that just calls C functions:

4 In other words, C library localization is unaffected.

5 Traditional global localization is still easy:

```
#include <iostream>
#include <locale>
int main(int argc, char** argv)
{
  using namespace std;
  locale::global(locale(""));  // set the global locale
   cin.imbue(locale());        // imbue it on the std streams
  cout.imbue(locale());
  cerr.imbue(locale());
  return MyObject(argc, argv).doit();
}
```

6 Greater flexibility is possible:

```
#include <iostream>
#include <locale>
int main()
{
  using namespace std;
  cin.imbue(locale(""));  // the user's preferred locale
  cout.imbue(locale::classic());
  double f;
  while (cin >> f) cout << f << endl;
  return (cin.fail() != 0);
}
```

7 In a European locale, with input `3.456,78`, output is `3456.78`. This can be important even for simple programs, which may need to write a data file in a fixed format, regardless of a user's preference.

8 For example:

```
// file: Date.h
#include <locale>
   ...
class Date {
  ...
 public:
  Date(unsigned day, unsigned month, unsigned year);
  std::string asString(const std::locale& = std::locale());
};
istream& operator>>(istream& s, Date& d);
ostream& operator<<(ostream& s, Date d);
...
```

9 This example illustrates two architectural uses of class `locale`. The first is as a default argument in `Date::asString()`, where the default is the global (presumably user-preferred) locale. The second is in the operators `<<` and `>>`, where a locale ''hitchhikes'' on another object, in this case a stream, to the point where it is needed.

```
// file: Date.C
#include <Date>
#include <stringstream>
std::string Date::asString(const std::locale& l)
{
  using namespace std;
  stringstream s; s.imbue(l);
  s << *this; return s.data();
}

std::istream& operator>>(std::istream& s, Date& d)
{
  using namespace std;
  if (!s.ipfx(0)) return s;
  locale loc = s.rdloc();
  struct tm t;
  loc.use<time_get<char> >().get_date(s, s, loc, &t);
  if (s) d = Date(t.tm_day, t.tm_mon + 1, t.tm_year + 1900);
  s.isfx();
  return s;
}
```

10 A locale object may be extended with a new facet simply by constructing it with an instance of a class derived from `locale::facet`. The only member you must define is the static member `id`, which identifies your class interface as a new facet. For example, imagine we want to classify Japanese characters:

```
// file: jctype.h
#include <locale>
namespace My {
  using namespace std;
  class JCtype : public locale::facet {
  public:
    static locale::id id;  // required for use as a new locale facet
    bool is_kanji(wchar_t c);
    JCtype() {}
  protected:
   ~JCtype() {}
  };
}

// file: filt.C
#include <iostream>
#include <locale>
#include <jctype> // above
std::locale::id JCtype::id;  // the static JCtype member declared above.
int main()
{
  using namespace std;
  typedef ctype<wchar_t> ctype;
  locale loc(locale(""),       // the user's preferred locale ...
             new My::JCType);  // and a new feature ...
  wchar_t c = loc.use<ctype>().widen('!');
  if (loc.use<My::JCType>().is_kanji(c))
    cout << "no it isn't!" << endl;
  return 0;
}
```

11 The new facet is used exactly like the built-in facets.

12 Replacing an existing facet is even easier. Here we do not define a member `id` because we are reusing the `numpunct<charT>` facet interface:

```
// my_bool.C
#include <iostream>
#include <locale>
#include <string>
namespace My {
  using namespace std;
  typedef numpunct_byname<char> numpunct;
  class BoolNames : public numpunct {
    typedef basic_string<char> string;
  protected:
    string do_truename()  { return "Oui Oui!"; }
    string do_falsename() { return "Mais Non!"; }
   ~BoolNames() {}
  public:
    BoolNames(const char* name) : numpunct(name) {}
  };
}
```

```
int main(int argc, char** argv)
{
  using namespace std;
  // make the user's preferred locale, except for...
  locale loc(locale(""), new My::BoolNames(""));
  cout.imbue(loc);
  cout << "Any arguments today? " << (argc > 1) << endl;
  return 0;
}
```

22.3 C Library Locales [lib.c.locales]

1 Header <clocale> (Table 47):

Table 47—Header <clocale> synopsis

Type	Name(s)		
Macros:	LC_MONETARY	LC_NUMERIC	LC_TIME
Struct:	lconv		
Functions:	localeconv	setlocale	

2 The contents are the same as the Standard C library. *SEE ALSO:* ISO C subclause 7.10.4.

23 Containers library [lib.containers]

1 This clause describes components that C++ programs may use to organize collections of information.

2 The following subclauses describe container requirements, and components for sequences and associative containers, as summarized in Table 48:

Table 48—Containers library summary

Subclause	Header(s)
23.1 Requirements	
23.2 Sequences	<bits> <deque> <list> <queue> <stack> <vector>
23.3 Associative containers	<map> <set>

23.1 Container requirements [lib.container.requirements]

1 Containers are objects that store other objects. They control allocation and deallocation of these objects through constructors, destructors, insert and erase operations.

2 In the following Table 49, we assume X is a container class containing objects of type T, a and b are values of X, u is an identifier and r is a value of X&.

Table 49—Container requirements

expression	return type	assertion/note pre/post-condition	complexity
`X::value_type`	`T`		compile time
`X::iterator`	iterator type pointing to `T`	any iterator category except output iterator.	compile time
`X::const_ iterator`	iterator type pointing to `const T`	any iterator category except output iterator.	compile time
`X::difference_type`	signed integral type	is identical to the distance type of `X::iterator` and `X::const_iterator`	compile time
`X:: size_type`	unsigned integral type	size_type can represent any non-negative value of `difference_type`	compile time
`X u;`		post: `u.size() == 0`.	constant
`X()`		`X().size() == 0`.	constant
`X(a)`		`a == X(a)`.	linear
`X u(a);` `X u = a;`		post: `u == a`. Equivalent to: `X u; u = a;`	linear
`(&a)->` `~X()`	result is not used	post: `a.size() == 0`. note: the destructor is applied to every element of `a`, all the memory is returned.	linear
`a.begin()`	`iterator`; `const_iterator` for constant `a`		constant
`a.end()`	`iterator`; `const_iterator` for constant `a`		constant

`a == b`	convertible to `bool`	`==` is an equivalence relation.
		`a.size()==b.size()`
		`&& equal(a.begin(),`
		`a.end(), b.begin())`
			linear

`a != b`	convertible to `bool`	Equivalent to: `!(a == b)`	linear

expression	return type	operational semantics	assertion/note pre/post-condition	complexity
`r = a`	`X&`	`if (&r != &a) { (&r)->X::~X(); new (&r) X(a); return r; }`	post: `r == a`.	linear
`a.size()`	`size_type`	`a.end() - a.begin()`		constant
`a.max_ size()`	`size_type`		`size()` of the largest possible container.	constant
`a.empty()`	convertible to `bool`	`a.size() == 0`		constant
`a < b`	convertible to `bool`	`lexicographical_compare (a.begin(), a.end(), b.begin(), b.end())`	pre: `<` is defined for values of `T`. `<` is a total ordering relation.	linear
`a > b`	convertible to `bool`	`b < a`		linear
`a <= b`	convertible to `bool`	`!(a > b)`		linear
`a >= b`	convertible to `bool`	`!(a < b)`		linear

Notes: `equal` and `lexicographical_compare` are defined in Clause (25).

3 The member function `size()` returns the number of elements in the container. Its semantics is defined by the rules of constructors, inserts, and erases.

4 `begin()` returns an iterator referring to the first element in the container. `end()` returns an iterator which is the past-the-end value.

23.1.1 Sequences [lib.sequence.reqmts]

1 A sequence is a kind of container that organizes a finite set of objects, all of the same type, into a strictly linear arrangement. The library provides three basic kinds of sequence containers: `vector`, `list`, and `deque`. It also provides container adaptors that make it easy to construct abstract data types, such as `stack` s or `queue` s, out of the basic sequence kinds (or out of other kinds of sequences that the user might define).

2 In the following Table 50, `X` is a sequence class, `a` is value of `X`, `i` and `j` satisfy input iterator requirements, `[i, j)` is a valid range, `n` is a value of `X::size_type`, `p` is a valid iterator to `a`, `q`, `q1`, `q2` are valid dereferenceable iterators to `a`, `[q1, q2)` is a valid range, `t` is a value of `X::value_type`.

3 The complexities of the expressions are sequence dependent.

Table 50—Sequence requirements (in addition to container)

expression	return type	assertion/note pre/post-condition
`X(n, t)` `X a(n, t);`		post: `size() == n`. constructs a sequence with `n` copies of `t`.
`X(i, j)` `X a(i, j);`		post: `size() == distance` between `i` and `j`. constructs a sequence equal to the range `[i, j)`.
`a.insert(p, t)`	`iterator`	inserts a copy of `t` before `p`.
`a.insert(p, n, t)`	result is not used	inserts n copies of `t` before `p`.
`a.insert(p, i, j)`	result is not used	inserts copies of elements in `[i, j)` before `p`.
`a.erase(q)`	result is not used	erases the element pointed to by `q`.
`a.erase(q1, q2)`	result is not used	erases the elements in the range `[q1, q2)`.

4 `list`,`vector`, and `deque` offer the programmer different complexity trade-offs and should be used accordingly. `vector` is the type of sequence that should be used by default. `list` should be used when there are frequent insertions and deletions from the middle of the sequence. `deque` is the data structure of choice when most insertions and deletions take place at the beginning or at the end of the sequence.

5 `iterator` and `const_iterator` types for sequences have to be at least of the forward iterator category.

6 Table 51:

Table 51—Optional sequence operations

expression	return type	operational semantics	container
`a.front()`	`T&; const T&` for constant `a`	`*a.begin()`	`vector, list, deque`
`a.back()`	`T&; const T&` for constant `a`	`*a.end()`	`vector, list, deque`
`a.push_front(x)`	`void`	`a.insert(a.begin(),x)`	`list, deque`
`a.push_back(x)`	`void`	`a.insert(a.end(),x)`	`vector, list, deque`
`a.pop_front()`	`void`	`a.erase(a.begin())`	`list, deque`
`a.pop_back()`	`void`	`a.erase(--a.end())`	`vector, list, deque`
`a[n]`	`T&; const T&` for constant `a`	`*(a.begin() + n)`	`vector, deque`

7 All the operations in the above table are provided only for the containers for which they take constant time.

23.1.2 Associative containers [lib.associative.reqmts]

1 Associative containers provide an ability for fast retrieval of data based on keys. The library provides four basic kinds of associative containers: `set`, `multiset`, `map` and `multimap`.

2 All of them are parameterized on `Key` and an ordering relation `Compare` that induces a total ordering on elements of `Key`. In addition, `map` and `multimap` associate an arbitrary type `T` with the `Key`. The object of type `Compare` is called the *comparison object* of a container.

3 In this section when we talk about equality of keys we mean the equivalence relation imposed by the comparison and *not* the `operator==` on keys. That is, two keys `k1` and `k2` are considered to be equal if for the comparison object `comp`, `comp(k1, k2) == false && comp(k2, k1) == false`.

4 An associative container supports *unique keys* if it may contain at most one element for each key. Otherwise, it supports *equal keys*. `set` and `map` support unique keys. `multiset` and `multimap` support equal keys.

5 For `set` and `multiset` the value type is the same as the key type. For `map` and `multimap` it is equal to `pair<const Key, T>`.

6 `iterator` of an associative container is of the bidirectional iterator category.

7 In the following Table 52, `X` is an associative container class, `a` is a value of `X`, `a_uniq` is a value of `X` when `X` supports unique keys, and `a_eq` is a value of `X` when `X` supports multiple keys, `i` and `j` satisfy input iterator requirements and refer to elements of `value_type`, `[i, j)` is a valid range, `p` is a valid iterator to `a`, `q`, `q1`, `q2` are valid dereferenceable iterators to `a`, `[q1, q2)` is a valid range, `t` is a value of `X::value_type` and `k` is a value of `X::key_type`.

Table 52—Associative container requirements (in addition to container)

expression	return type	assertion/note pre/post-condition	complexity
`X::key_type`	`Key`		compile time
`X:: key_compare`	`Compare`	defaults to `less<key_type>`.	compile time
`X:: value_compare`	a binary predicate type	is the same as `key_compare` for `set` and `multiset`; is an ordering relation on pairs induced by the first component (i.e. `Key`) for `map` and `multimap`.	compile time
`X(c)` `X a(c);`		constructs an empty container; uses `c` as a comparison object.	constant
`X()` `X a;`		constructs an empty container; uses `Compare()` as a comparison object.	constant
`X(i,j,c);` `X a(i,j,c);`		constructs an empty container and inserts elements from the range `[i, j)` into it; uses `c` as a comparison object.	`NlogN` in general (`N` is the distance from `i` to `j`); linear if `[i, j)` is sorted with `value_comp()`
`X(i, j)` `X a(i, j);`		same as above, but uses `Compare()` as a comparison object.	same as above
`a.key_comp()`	`X:: key_compare`	returns the comparison object out of which `a` was constructed.	constant
`a.value_comp()`	`X:: value_compare`	returns an object of `value_compare` constructed out of the comparison object.	constant
`a_uniq.insert(t)`	`pair<iterator, bool>`	inserts `t` if and only if there is no element in the container with key equal to the key of `t`. The `bool` component of the returned pair indicates whether the insertion takes place and the `iterator` component of the pair points to the element with key equal to the key of `t`.	logarithmic

expression	return type	assertion/note pre/post-condition	complexity
`a_eq.insert(t)`	`iterator`	inserts `t` and returns the iterator pointing to the newly inserted element.	logarithmic
`a.insert(p, t)`	`iterator`	inserts `t` if and only if there is no element with key equal to the key of `t` in containers with unique keys; always inserts `t` in containers with equal keys. always returns the iterator pointing to the element with key equal to the key of `t`. iterator `p` is a hint pointing to where the insert should start to search.	logarithmic in general, but amortized constant if `t` is inserted right after `p`.
`a.insert(i, j)`	result is not used	inserts the elements from the range `[i, j)` into the container.	`N log(size()+N)` (`N` is the distance from `i` to `j`) in general; linear if `[i, j)` is sorted according to `value_comp()`
`a.erase(k)`	`size_type`	erases all the elements in the container with key equal to `k`. returns the number of erased elements.	`log(size()) + count(k)`
`a.erase(q)`	result is not used	erases the element pointed to by `q`.	amortized constant
`a.erase(q1, q2)`	result is not used	erases all the elements in the range `[q1, q2)`.	`log(size())+ N` where `N` is the distance from `q1` to `q2`.
`a.find(k)`	`iterator`; `const_iterator` for constant `a`	returns an iterator pointing to an element with the key equal to `k`, or `a.end()` if such an element is not found.	logarithmic
`a.count(k)`	`size_type`	returns the number of elements with key equal to `k`	`log(size())+ count(k)`
`a.lower_bound(k)`	`iterator`; `const_iterator` for constant `a`	returns an iterator pointing to the first element with key not less than `k`.	logarithmic
`a.upper_bound(k)`	`iterator`; `const_iterator` for constant `a`	returns an iterator pointing to the first element with key greater than `k`.	logarithmic
`a.equal_range(k)`	`pair< iterator,iterator>`; `pair< const_iterator, const_iterator>` for constant `a`	equivalent to `make_pair( a.lower_bound(k), a.upper_bound(k))`.	logarithmic

8 The fundamental property of iterators of associative containers is that they iterate through the containers in the non-descending order of keys where non-descending is defined by the comparison that was used to construct them. For any two dereferenceable iterators `i` and `j` such that distance from `i` to `j` is positive,

```
value_comp(*j, *i) == false
```

9 For associative containers with unique keys the stronger condition holds,

`value_comp(*i, *j) == true.`

23.2 Sequences [lib.sequences]

1 Headers <bits>, <deque>, <list>, <queue>, <stack>, and <vector>.

Header <bits> synopsis

```
#include <cstddef>      // for size_t
#include <string>
#include <stdexcept>    // for invalid_argument, out_of_range, overflow_error
#include <iosfwd>       // for istream, ostream
namespace std {
  template<size_t N> class bits;
  template <size_t N>
    istream& operator>>(istream& is, bits<N>& x);
  template <size_t N>
    ostream& operator<<(ostream& os, const bits<N>& x);
}
```

Header <deque> synopsis

```
#include <memory>       // for allocator
namespace std {
  template <class T, class Allocator = allocator> class deque;
  template <class T, class Allocator>
    bool operator==(const deque<T,Allocator>& x, const deque<T,Allocator>& y);
  template <class T, class Allocator>
    bool operator< (const deque<T,Allocator>& x, const deque<T,Allocator>& y);
}
```

Header <list> synopsis

```
#include <memory>       // for allocator
namespace std {
  template <class T, class Allocator = allocator> class list;
  template <class T, class Allocator>
    bool operator==(const list<T,Allocator>& x, const list<T,Allocator>& y);
  template <class T, class Allocator>
    bool operator< (const list<T,Allocator>& x, const list<T,Allocator>& y);
}
```

Header <queue> synopsis

```
#include <functional>   // for less
namespace std {
  template <class Container> class queue;
  template <class Container>
    bool operator==(const queue<Container>& x, const queue<Container>& y);
  template <class Container, class Compare = less<Container::value_type> >
    class priority_queue;
}
```

Header <stack> synopsis

```
namespace std {
  template <class Container> class stack;
  template <class Container>
    bool operator==(const stack<Container>& x, const stack<Container>& y);
}
```

Header `<vector>` synopsis

```
#include <memory>       // for allocator
namespace std {
  template <class T, class Allocator = allocator> class vector;
  template <class T, class Allocator>
    bool operator==(const vector<T,Allocator>& x, const vector<T,Allocator>& y);
  template <class T, class Allocator>
    bool operator< (const vector<T,Allocator>& x, const vector<T,Allocator>& y);

  class vector<bool,allocator>;
  bool operator==(const vector<bool,allocator>& x,
                  const vector<bool,allocator>& y);
  bool operator< (const vector<bool,allocator>& x,
                  const vector<bool,allocator>& y);
}
```

23.2.1 Template class `bits` [lib.template.bits]

1 The header `<bits>` defines a template class and several related functions for representing and manipulating fixed-size sequences of bits.

```
namespace std {
  template<size_t N> class bits {
  public:
    bits();
    bits(unsigned long val);
    bits(const string& str, size_t pos = 0, size_t n = NPOS);

    bits<N>& operator&=(const bits<N>& rhs);
    bits<N>& operator|=(const bits<N>& rhs);
    bits<N>& operator^=(const bits<N>& rhs);
    bits<N>& operator<<=(size_t pos);
    bits<N>& operator>>=(size_t pos);
    bits<N>& set();
    bits<N>& set(size_t pos, int val = 1);
    bits<N>& reset();
    bits<N>& reset(size_t pos);
    bits<N>  operator~() const;
    bits<N>& toggle();
    bits<N>& toggle(size_t pos);
```

```
    unsigned short to_ushort() const;
    unsigned long  to_ulong() const;
    string to_string() const;
    size_t count() const;
    size_t length() const;
    bool operator==(const bits<N>& rhs) const;
    bool operator!=(const bits<N>& rhs) const;
    bool test(size_t pos) const;
    bool any() const;
    bool none() const;
    bits<N> operator<<(size_t pos) const;
    bits<N> operator>>(size_t pos) const;
  private:
//  char array[N];      exposition only
  };
}
```

2 The template class `bits<N>` describes an object that can store a sequence consisting of a fixed number of bits, *N*.

3 Each bit represents either the value zero (reset) or one (set). To *toggle* a bit is to change the value zero to one, or the value one to zero. Each bit has a non-negative position *pos*. When converting between an object of class `bits<N>` and a value of some integral type, bit position *pos* corresponds to the *bit value* `1 << pos`. The integral value corresponding to two or more bits is the sum of their bit values.

Box 99

For the sake of exposition, the maintained data is presented here as:

— `char array[N]`, the sequence of bits, stored one bit per element.[149)]

4 The functions described in this subclause can report three kinds of errors, each associated with a distinct exception:

— an *invalid-argument* error is associated with exceptions of type `invalid_argument`;

— an *out-of-range* error is associated with exceptions of type `out_of_range`;

— an *overflow* error is associated with exceptions of type `overflow_error`.

23.2.1.1 `bits` constructors [lib.cons.bits]

```
bits();
```

Effects: Constructs an object of class `bits<N>`, initializing all bits to zero.

```
bits(unsigned long val);
```

Effects: Constructs an object of class `bits<N>`, initializing the first *M* bit positions to the corresponding bit values in *val*. *M* is the smaller of *N* and the value `CHAR_BIT * sizeof (unsigned long)`.[150)]
If *M* < *N*, remaining bit positions are initialized to zero.

```
bits(const string& str, size_t pos = 0, size_t n = NPOS);
```

149) An implementation is free to store the bit sequence more efficiently.
150) The macro `CHAR_BIT` is defined in `<climits>`(18.2).

Requires: str.size()pos<=
Throws: out_of_range if *pos > str.len*.
Effects: Determines the effective length *rlen* of the initializing string as the smaller of *n* and *str.len - pos*.
The function then throws invalid_argument if any of the *rlen* characters in *str* beginning at position *pos* is other than 0 or 1.
Otherwise, the function constructs an object of class bits<N>, initializing the first *M* bit positions to values determined from the corresponding characters in the string *str*. *M* is the smaller of *N* and *rlen*.

1 An element of the constructed string has value zero if the corresponding character in *str*, beginning at position *pos*, is 0. Otherwise, the element has the value one. Character position *pos + M - 1* corresponds to bit position zero. Subsequent decreasing character positions correspond to increasing bit positions.

2 If *M < N*, remaining bit positions are initialized to zero.

23.2.1.2 bits::operator&= [lib.bits::op&=.bt]

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

Effects: Clears each bit in *this for which the corresponding bit in *rhs* is clear, and leaves all other bits unchanged.
Returns: *this.

23.2.1.3 bits::operator|= [lib.bits::op|=.bt]

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

Effects: Sets each bit in *this for which the corresponding bit in *rhs* is set, and leaves all other bits unchanged.
Returns: *this.

23.2.1.4 bits::operator^= [lib.bits::op^=.bt]

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

Effects: Toggles each bit in *this for which the corresponding bit in *rhs* is set, and leaves all other bits unchanged.
Returns: *this.

23.2.1.5 bits::operator<<= [lib.bits::op.lsh=]

```
bits<N>& operator<<=(size_t pos);
```

Effects: Replaces each bit at position *I* in *this with a value determined as follows:

— If *I < pos*, the new value is zero;

— If *I >= pos*, the new value is the previous value of the bit at position *I - pos*.
Returns: *this.

23.2.1.6 bits::operator>>= [lib.bits::op.rsh=]

```
bits<N>& operator>>=(size_t pos);
```

Effects: Replaces each bit at position *I* in *this with a value determined as follows:

— If *pos >= N - I*, the new value is zero;

— If *pos < N - I*, the new value is the previous value of the bit at position *I + pos*.

Returns: `*this`.

23.2.1.7 bits::set [lib.bits::set]

```
bits<N>& set();
```

Effects: Sets all bits in `*this`.
Returns: `*this`.

```
bits<N>& set(size_t pos, int val = 1);
```

Requires: `pos is valid`
Throws: `out_of_range` if *pos* does not correspond to a valid bit position.
Effects: Stores a new value in the bit at position *pos* in `*this`. If *val* is nonzero, the stored value is one, otherwise it is zero.
Returns: `*this`.

23.2.1.8 bits::reset [lib.bits::reset]

```
bits<N>& reset();
```

Effects: Resets all bits in `*this`.
Returns: `*this`.

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

Requires: `pos is valid`
Throws: `out_of_range` if *pos* does not correspond to a valid bit position.
Effects: Resets the bit at position *pos* in `*this`.
Returns: `*this`.

23.2.1.9 bits::operator~ [lib.bits::op~]

```
bits<N> operator~() const;
```

Effects: Constructs an object *x* of class `bits<N>` and initializes it with `*this`.
Returns: *x*`.toggle()`.

23.2.1.10 bits::toggle [lib.bits::toggle]

```
bits<N>& toggle();
```

Effects: Toggles all bits in `*this`.
Returns: `*this`.

```
bits<N>& toggle(size_t pos);
```

Requires: `pos is valid`
Throws: `out_of_range` if *pos* does not correspond to a valid bit position.
Effects: Toggles the bit at position *pos* in `*this`.
Returns: `*this`.

23.2.1.11 bits::to_ushort [lib.bits::to.ushort]

```
unsigned short to_ushort() const;
```

Throws: `overflow_error` if the integral value *x* corresponding to the bits in `*this` cannot be represented as type `unsigned short`.

Returns: *x*.

23.2.1.12 bits::to_ulong [lib.bits::to.ulong]

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

Throws: `overflow_error` if the integral value *x* corresponding to the bits in `*this` cannot be represented as type `unsigned long`.
Returns: *x*.

23.2.1.13 bits::to_string [lib.bits::to.string]

```
string to_string() const;
```

Effects: Constructs an object of type `string` and initializes it to a string of length *N* characters. Each character is determined by the value of its corresponding bit position in `*this`. Character position *N* `- 1` corresponds to bit position zero. Subsequent decreasing character positions correspond to increasing bit positions. Bit value zero becomes the character `0`, bit value one becomes the character `1`.
Returns: The created object.

23.2.1.14 bits::count [lib.bits::count]

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

Returns: A count of the number of bits set in `*this`.

23.2.1.15 bits::length [lib.bits::length]

```
size_t length() const;
```

Returns: *N*.

23.2.1.16 bits::operator== [lib.bits::op==.bt]

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

Returns: A nonzero value if the value of each bit in `*this` equals the value of the corresponding bit in *rhs*.

23.2.1.17 bits::operator!= [lib.bits::op!=.bt]

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

Returns: A nonzero value if `!(*this == rhs)`.

23.2.1.18 bits::test [lib.bits::test]

```
bool test(size_t pos) const;
```

Requires: `pos is valid`
Throws: `out_of_range` if *pos* does not correspond to a valid bit position.
Returns: `true` if the bit at position *pos* in `*this` has the value one.

23.2.1.19 bits::any [lib.bits::any]

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

Returns: `true` if any bit in `*this` is one.

23.2.1.20 bits::none [lib.bits::none]

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

Returns: `true` if no bit in `*this` is one.

23.2.1.21 bits::operator<< [lib.bits::op.lsh]

```
bits<N> operator<<(size_t pos) const;
```

Returns: `bits<N>(*this) <<= pos`.

23.2.1.22 bits::operator>> [lib.bits::op.rsh]

```
bits<N> operator>>(size_t pos) const;
```

Returns: `bits<N>(*this) >>= pos`.

23.2.1.23 operator& [lib.bits::op&]

```
bits<N> operator&(const bits<N>& lhs, const bits<N>& rhs);
```

Returns: `bits<N>(lhs) &= pos`.

23.2.1.24 operator| [lib.bits::op|]

```
bits<N> operator|(const bits<N>& lhs, const bits<N>& rhs);
```

Returns: `bits<N>(lhs) |= pos`.

23.2.1.25 operator^ [lib.bits::opˆ]

```
bits<N> operator^(const bits<N>& lhs, const bits<N>& rhs);
```

Returns: `bits<N>(lhs) ^= pos`.

23.2.1.26 operator>> [lib.bits::ext]

```
template <size_t N>
  istream& operator>>(istream& is, bits<N>& x);
```

1 A formatted input function.

Effects: Extracts up to `N` (single-byte) characters from `is`. Stores these characters in a temporary object `str` of type `string`, then evaluates the expression `x = bits<N>(str)`. Characters are extracted and stored until any of the following occurs:

— `N` characters have been extracted and stored;

— end-of-file occurs on the input sequence;

— the next input character is neither `0` or `1` (in which case the input character is not extracted).

2 If no characters are stored in *str*, calls *is*.setstate(ios::failbit).
Returns: *is*.

23.2.1.27 operator<< [lib.bits::ins]

```
template <size_t N>
  ostream& operator<<(ostream& os, const bits<N>& x);
```

Returns: *os* << *x*.to_string().

23.2.2 Template class deque [lib.deque]

1 A deque is a kind of sequence that, like a vector (23.2.8), supports random access iterators. In addition, it supports constant time insert and erase operations at the beginning or the end; insert and erase in the middle take linear time. That is, a deque is especially optimized for pushing and popping elements at the beginning and end. As with vectors, storage management is handled automatically.

```
namespace std {
  template <class T, class Allocator = allocator>
  class deque {
  public:
  // types:
    typedef Allocator::types<T>::pointer       iterator;
    typedef Allocator::types<T>::const_pointer const_iterator;
    typedef Allocator::size_type       size_type;
    typedef Allocator::difference_type difference_type;
    typedef T value_type;

  // construct/copy/destroy:
    deque();
    deque(size_type n, const T& value = T());
    deque(const deque<T,Allocator>& x);
    template <class InputIterator>
      deque(InputIterator first, InputIterator last);
   ~deque();
    deque<T,Allocator>& operator=(const deque<T,Allocator>& x);
    template <class InputIterator>
      void assign(InputIterator first, InputIterator last);
    template <class Size, class T>
      void assign(Size n, const T& t = T());

  // observers:
    iterator       begin();
    const_iterator begin() const;
    iterator       end();
    const_iterator end() const;
    size_type size() const;
    size_type max_size() const;
    void      resize(size_type sz, T c = T());
    bool      empty() const;
    T&       operator[](size_type n);
    const T& operator[](size_type n) const;
    const T& at(size_type n) const;
    T&       at(size_type n);
    T&       front();
    const T& front() const;
    T&       back();
    const T& back() const;
```

```
  // modifiers:
    void push_front(const T& x);
    void push_back(const T& x);
    iterator insert(iterator position, const T& x = T());
    void     insert(iterator position, size_type n, const T& x = T());
    template <class InputIterator>
      void insert (iterator position, InputIterator first, InputIterator last);
    void pop_front();
    void pop_back();
    void erase(iterator position);
    void erase(iterator first, iterator last);
  };

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

23.2.2.1 deque members [lib.deque.members]

23.2.2.1.1 assign [lib.deque.assign]

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

Effects:

```
  erase(begin(), end());
  insert(begin(), first, last);
```

```
template <class Size, class T>
  void assign(Size n, const T& t = T());
```

Effects:

```
  erase(begin(), end());
  insert(begin(), n, t);
```

23.2.2.1.2 insert [lib.deque.insert]

```
iterator insert(iterator position, const T& x = T());
void     insert(iterator position, size_type n, const T& x = T());
template <class InputIterator>
  void insert(iterator position,
              InputIterator first, InputIterator last);
```

Effects: Invalidates all the iterators and references to the deque if the insertion pointer is not at either end. Insertion at either end does not affect iterators and references.

Complexity: In the worst case, inserting a single element into a deque takes time linear in the minimum of the distance from the insertion point to the beginning of the deque and the distance from the insertion point to the end of the deque. Inserting a single element either at the beginning or end of a deque always takes constant time and causes a single call to the copy constructor of T.

23.2.2.1.3 `erase` [lib.deque.erase]

```
void erase(iterator position);
void erase(iterator first, iterator last);
```

Effects: Invalidates all the iterators and references to the deque if the erasing point is not at either end. Erasing at either end does not affect iterators and references. The number of calls to the destructor is the same as the number of elements erased, but the number of the calls to the assignment operator is equal to the minimum of the number of elements before the erased elements and the number of element after the erased elements.

23.2.2.1.4 `resize` [lib.deque.resize]

```
void      resize(size_type sz, T c = T());
```

Effects:

```
if (sz > size())
  s.insert(s.end(), s.size()-sz, v);
else if (sz < size())
  s.erase(s.begin()+sz, s.end());
else
  ???
```

23.2.3 Template class `list` [lib.list]

1 A `list` is a kind of sequence that supports bidirectional iterators and allows constant time insert and erase operations anywhere within the sequence, with storage management handled automatically. Unlike vectors (23.2.8) and deques (23.2.2), fast random access to list elements is not supported, but many algorithms only need sequential access anyway.

```
namespace std {
  template <class T, class Allocator = allocator>
  class list {
  public:
  // types:
    typedef Allocator::types<T>::pointer       iterator;
    typedef Allocator::types<T>::const_pointer const_iterator;
    typedef Allocator::size_type       size_type;
    typedef Allocator::difference_type difference_type;
    typedef T value_type;

  // construct/copy/destroy:
    list();
    list(size_type n, const T& value = T());
    template <class InputIterator>
      list(InputIterator first, InputIterator last);
    list(const list<T,Allocator>& x);
   ~list();
    list<T,Allocator>& operator=(const list<T,Allocator>& x);
    template <class InputIterator>
      void assign(InputIterator first, InputIterator last);
    template <class Size, class T>
      void assign(Size n, const T& t = T());
```

```
  // observers:
    iterator       begin();
    const_iterator begin() const;
    iterator       end();
    const_iterator end() const;
    bool      empty() const;
    size_type size() const;
    size_type max_size() const;
    void      resize(size_type sz, T c = T());
    T&       front();
    const T& front() const;
    T&       back();
    const T& back() const;

  // modifiers:
    void push_front(const T& x);
    void pop_front();
    void push_back(const T& x);
    void pop_back();
    iterator insert(iterator position, const T& x = T());
    void     insert(iterator position, size_type n, const T& x = T());
    template <class InputIterator>
      void insert(iterator position, InputIterator first,
                  InputIterator last);
    void erase(iterator position);
    void erase(iterator position, iterator last);

  // special mutative operations on list:
    void splice(iterator position, list<T,Allocator>& x);
    void splice(iterator position, list<T,Allocator>& x, iterator i);
    void splice(iterator position, list<T,Allocator>& x, iterator first,
                iterator last);
    void remove(const T& value);
    template <class Predicate> void remove_if(Predicate pred);
    void   unique();
    template <class BinaryPredicate> void unique(BinaryPredicate binary_pred);
    void   merge(list<T,Allocator>& x);
    template <class Compare> void merge(list<T,Allocator>& x, Compare comp);
    void   sort();
    template <class Compare> void sort(Compare comp);
    void   reverse();
  };

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

23.2.3.1 `list` members [lib.list.members]

1 Since lists allow fast insertion and erasing from the middle of a list, certain operations are provided specifically for them.

2 `list` provides three splice operations that destructively move elements from one list to another.

23.2.3.1.1 assign [lib.list.assign]

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

Effects:

```
  erase(begin(), end());
  insert(begin(), first, last);
```

```
template <class Size, class T>
  void assign(Size n, const T& t = T());
```

Effects:

```
  erase(begin(), end());
  insert(begin(), n, t);
```

23.2.3.1.2 insert [lib.list.insert]

```
iterator insert(iterator position, const T& x = T());
void     insert(iterator position, size_type n, const T& x = T());
template <class InputIterator>
  void insert(iterator position, InputIterator first,
              InputIterator last);
```

Notes:

1 Does not affect the validity of iterators and references.

Complexity: Insertion of a single element into a list takes constant time and exactly one call to the copy constructor of T. Insertion of multiple elements into a list is linear in the number of elements inserted, and the number of calls to the copy constructor of T is exactly equal to the number of elements inserted.

23.2.3.1.3 erase [lib.list.erase]

```
void erase(iterator position);
void erase(iterator first, iterator last);
```

Effects: Invalidates only the iterators and references to the erased elements.

Complexity: Erasing a single element is a constant time operation with a single call to the destructor of T. Erasing a range in a list is linear time in the size of the range and the number of calls to the destructor of type T is exactly equal to the size of the range.

23.2.3.1.4 resize [lib.list.resize]

```
void resize(size_type sz, T c = T());
```

Effects:

```
  if (sz > size())
    s.insert(s.end(), s.size()-sz, v);
  else if (sz < size())
    s.erase(s.begin()+sz, s.end());
  else
    ???
```

23.2.3.1.5 `splice` [lib.list.splice]

```
void splice(iterator position, list<T,Allocator>& x);
```

Effects: Inserts the contents of `x` before `position` and `x` becomes empty.
Complexity: Constant time.

```
void splice(iterator position, list<T,Allocator>& x, iterator i);
```

Effects: Inserts an element pointed to by `i` from list `x` before position and removes the element from `x`.
Requires: `i` is a valid iterator of `x`.
Complexity: Constant time.

```
void splice(iterator position, list<T,Allocator>& x, iterator first,
        iterator last);
```

Effects: Inserts elements in the range `[first, last)` before `position` and removes the elements from `x`.
Requires: `[first, last)` is a valid range in `x`.
Complexity: Linear time.

23.2.3.1.6 `remove` [lib.list.remove]

```
void remove(const T& value);
template <class Predicate> void remove_if(Predicate pred);
```

Effects: Erases all the elements in the list referred by the list iterator `i` for which the following conditions hold: `*i == value, pred(*i) == true`.
Notes: Stable: the relative order of the elements that are not removed is the same as their relative order in the original list.
Complexity: Exactly `size()` applications of the corresponding predicate.

23.2.3.1.7 `unique` [lib.list.unique]

```
void   unique();
template <class BinaryPredicate>
  void unique(BinaryPredicate binary_pred);
```

Effects: Erases all but the first element from every consecutive group of equal elements in the list.
Complexity: Exactly `size() - 1` applications of the corresponding binary predicate.

23.2.3.1.8 `merge` [lib.list.merge]

```
void   merge(list<T,Allocator>& x);
template <class Compare> void merge(list<T,Allocator>& x, Compare comp);
```

Effects: Merges the argument list into the list (both are assumed to be sorted).
Notes: Stable: for equal elements in the two lists, the elements from the list always precede the elements from the argument list. `x` is empty after the merge.
Complexity: At most `size() + x.size() - 1` comparisons.

23.2.3.1.9 `reverse` [lib.list.reverse]

```
void   reverse();
```

Effects: Reverses the order of the elements in the list.

Complexity: Linear time.

23.2.3.1.10 sort [lib.list.sort]

```
void   sort();
template <class Compare> void sort(Compare comp);
```

Effects: Sorts the list according to the operator< or a compare function object.
Notes: Stable: the relative order of the equal elements is preserved.
Complexity: Approximately NlogN comparisons, where N == size().

23.2.4 Template class queue [lib.queue]

1 Any sequence supporting operations front(), back(), push_back() and pop_front() can be used to instantiate queue. In particular, list(23.2.3) and deque(23.2.2) can be used.

```
nmespace std {
  template <class Container>
  class queue {
  public:
    typedef Container::value_type value_type;
    typedef Container::size_type  size_type;
  protected:
    Container c;

  public:
    bool      empty() const            { return c.empty(); }
    size_type size()  const            { return c.size(); }
    value_type&       front()          { return c.front(); }
    const value_type& front() const    { return c.front(); }
    value_type&       back()           { return c.back(); }
    const value_type& back() const     { return c.back(); }
    void push(const value_type& x)     { c.push_back(x); }
    void pop()                         { c.pop_front(); }
  };

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

operator==
Returns: x.c == y.c.

23.2.5 Template class priority_queue [lib.priority.queue]

1 Any sequence with random access iterator and supporting operations front(), push_back() and pop_back() can be used to instantiate priority_queue. In particular, vector(23.2.8) and deque(23.2.2) can be used.

```
namespace std {
  template <class Container, class Compare = less<Container::value_type> >
  class priority_queue {
  public:
    typedef Container::value_type value_type;
    typedef Container::size_type  size_type;
  protected:
    Container c;
    Compare comp;
```

```
  public:
    priority_queue(const Compare& x = Compare());
    template <class InputIterator>
      priority_queue(InputIterator first, InputIterator last,
        const Compare& x = Compare());

    bool      empty() const       { return c.empty(); }
    size_type size()  const       { return c.size(); }
    const value_type& top() const { return c.front(); }
    void push(const value_type& x);

    void pop();
  };
  // no equality is provided
}
```

23.2.6 `priority_queue` members [lib.priority.queue.members]

23.2.6.1 `priority_queue` constructors [lib.priqueue.cons]

```
priority_queue(const Compare& x = Compare());
```

Effects:

```
: c(), comp(x) {}
```

```
template <class InputIterator>
  priority_queue(InputIterator first, InputIterator last,
    const Compare& x = Compare());
```

Effects:

```
      : c(first, last), comp(x) {
        make_heap(c.begin(), c.end(), comp);
    }
```

23.2.6.2 `push` [lib.priqueue.push]

```
void push(const value_type& x);
```

Effects:

```
      c.push_back(x);
      push_heap(c.begin(), c.end(), comp);
```

23.2.6.3 `pop` [lib.priqueue.pop]

```
void pop();
```

Effects:

```
      pop_heap(c.begin(), c.end(), comp);
      c.pop_back();
```

23.2.7 Template class `stack` [lib.stack]

1 Any sequence supporting operations `back()`, `push_back()` and `pop_back()` can be used to instantiate `stack`. In particular, `vector`(23.2.8), `list`(23.2.3) and `deque`(23.2.2) can be used.

2 For example, `stack<vector<int> >` is an integer stack made out of `vector`, and `stack<deque<char> >` is a character stack made out of `deque`.

```
namespace std {
  template <class Container>
  class stack {
  public:
    typedef Container::value_type value_type;
    typedef Container::size_type  size_type;
  protected:
    Container c;

  public:
    bool      empty() const            { return c.empty(); }
    size_type size()  const            { return c.size(); }
    value_type&       top()            { return c.back(); }
    const value_type& top() const      { return c.back(); }
    void push(const value_type& x)     { c.push_back(x); }
    void pop()                         { c.pop_back(); }
  };

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

`operator==`
Returns: `x.c == y.c`.

23.2.8 Template class `vector` [lib.vector]

1 A `vector` is a kind of sequence supports random access iterators. In addition, it supports (amortized) constant time insert and erase operations at the end; insert and erase in the middle take linear time. Storage management is handled automatically, though hints can be given to improve efficiency.

```
namespace std {
  template <class T, class Allocator = allocator>
  class vector {
  public:
  // types:
    typedef Allocator::types<T>::pointer       iterator;
    typedef Allocator::types<T>::const_pointer const_iterator;
    typedef Allocator::size_type       size_type;
    typedef Allocator::difference_type difference_type;
    typedef T value_type;

  // construct/copy/destroy:
    vector();
    vector(size_type n, const T& value = T());
    vector(const vector<T,Allocator>& x);
    template <class InputIterator>
      vector(InputIterator first, InputIterator last);
   ~vector();
    vector<T,Allocator>& operator=(const vector<T,Allocator>& x);
    template <class InputIterator>
      void assign(InputIterator first, InputIterator last);
    template <class Size, class T> void assign(Size n, const T& t = T());
    void reserve(size_type n);
```

```
  // observers:
    iterator       begin();
    const_iterator begin() const;
    iterator       end();
    const_iterator end() const;
    size_type size() const;
    size_type max_size() const;
    void      resize(size_type sz, T c = T());
    size_type capacity() const;
    bool      empty() const;
    T&       operator[](size_type n);
    const T& operator[](size_type n) const;
    const T& at(size_type n) const;
    T&       at(size_type n);
    T&       front();
    const T& front() const;
    T&       back();
    const T& back() const;

  // modifiers:
    void push_back(const T& x);
    void pop_back();
    iterator insert(iterator position, const T& x = T());
    void     insert(iterator position, size_type n, const T& x = T());
    template <class InputIterator>
        void insert(iterator position, InputIterator first, InputIterator last);
    void erase(iterator position);
    void erase(iterator first, iterator last);
  };

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

23.2.8.1 `vector` members [lib.vector.members]

23.2.8.1.1 `vector` constructor [lib.vector.cons]

```
vector();
vector(size_type n, const T& value = T());
vector(const vector<T,Allocator>& x);
template <class InputIterator>
  vector(InputIterator first, InputIterator last);
```

Complexity: The constructor `template <class InputIterator> vector(InputIterator first, InputIterator last)` makes only `N` calls to the copy constructor of `T` (where `N` is the distance between `first` and `last`) and no reallocations if iterators first and last are of forward, bidirectional, or random access categories. It does at most `2N` calls to the copy constructor of `T` and `logN` reallocations if they are just input iterators, since it is impossible to determine the distance between `first` and `last` and then do copying.

23.2.8.1.2 assign [lib.vector.assign]

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

Effects:

```
  erase(begin(), end());
  insert(begin(), first, last);
```

```
template <class Size, class T> void assign(Size n, const T& t = T());
```

Effects:

```
  erase(begin(), end());
  insert(begin(), n, t);
```

23.2.8.1.3 capacity [lib.vector.capacity]

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

Returns: The size of the allocated storage in the vector.

23.2.8.1.4 reserve [lib.vector.reserve]

```
void reserve(size_type n);
```

Effects: A directive that informs `vector` of a planned change in size, so that it can manage the storage allocation accordingly. It does not change the size of the sequence and takes at most linear time in the size of the sequence. Reallocation happens at this point if and only if the current capacity is less than the argument of `reserve`.
After `reserve`, `capacity` is greater or equal to the argument of `reserve` if reallocation happens; and equal to the previous value of `capacity` otherwise. Reallocation invalidates all the references, pointers, and iterators referring to the elements in the sequence. It is guaranteed that no reallocation takes place during the insertions that happen after `reserve` takes place till the time when the size of the vector reaches the size specified by `reserve`.

23.2.8.1.5 resize [lib.vector.resize]

```
void      resize(size_type sz, T c = T());
```

Effects:

```
  if (sz > size())
    s.insert(s.end(), s.size()-sz, v);
  else if (sz < size())
    s.erase(s.begin()+sz, s.end());
  else
    ???
```

23.2.8.1.6 insert [lib.vector.insert]

```
iterator insert(iterator position, const T& x = T());
void     insert(iterator position, size_type n, const T& x = T());
template <class InputIterator>
  void insert(iterator position, InputIterator first, InputIterator last);
```

Notes: Causes reallocation if the new size is greater than the old capacity. If no reallocation happens, all the iterators and references before the insertion point remain valid.

Complexity: Inserting a single element into a vector is linear in the distance from the insertion point to the end of the vector.

The amortized complexity over the lifetime of a vector of inserting a single element at its end is constant. Insertion of multiple elements into a vector with a single call of the insert member function is linear in the sum of the number of elements plus the distance to the end of the vector.[151)]

23.2.8.1.7 `erase` [lib.vector.erase]

```
void erase(iterator position);
void erase(iterator first, iterator last);
```

Effects: Invalidates all the iterators and references after the point of the erase. The destructor of `T` is called the number of times equal to the number of the elements erased, but the assignment operator of `T` is called the number of times equal to the number of elements in the vector after the erased elements.

23.2.9 Class `vector<bool>` [lib.vector.bool]

1 To optimize space allocation, a specialization for `bool` is provided:[152)]

```
namespace std {
  class vector<bool,allocator> {
  public:
  // bit reference:
    class reference {
    public:
       ~reference();
        operator bool() const;
        reference& operator=(const bool x);
        void flip();      // flips the bit
    };

  // types:
    typedef Allocator::types<T>::pointer       iterator;
    typedef Allocator::types<T>::const_pointer const_iterator;
    typedef Allocator::size_type       size_type;
    typedef Allocator::difference_type difference_type;
    typedef bool      value_type;

  // construct/copy/destroy:
    vector();
    vector(size_type n, const bool& value = bool());
    vector(const vector<bool,allocator>& x);
    template <class InputIterator>
      vector(InputIterator first, InputIterator last);
   ~vector();
    vector<bool,allocator>& operator=(const vector<bool,allocator>& x);
    void reserve(size_type n);
```

---

151) In other words, it is much faster to insert many elements into the middle of a vector at once than to do the insertion one at a time. The insert template member function preallocates enough storage for the insertion if the iterators `first` and `last` are of forward, bidirectional or random access category. Otherwise, it does insert elements one by one and should not be used for inserting into the middle of vectors.

152) Every implementation is expected to provide specializations of `vector<bool>` for all supported memory models.

```
  // observers:
    iterator       begin();
    const_iterator begin() const;
    iterator       end();
    const_iterator end() const;
    size_type size() const;
    size_type max_size() const;
    size_type capacity() const;
    bool empty() const;
    reference       operator[](size_type n);
    const reference operator[](size_type n) const;
    const reference at(size_type n) const;
    reference       at(size_type n);
    reference       front();
    const reference front() const;
    reference       back();
    const reference back() const;

  // modifiers:
    void push_back(const bool& x);
    void pop_back();
    iterator insert(iterator position, const bool& x = bool());
    void     insert (iterator position, size_type n, const bool& x = bool());
    template <class InputIterator>
        void insert (iterator position, InputIterator first, InputIterator last);
    void erase(iterator position);
    void erase(iterator first, iterator last);
  };

  bool operator==(const vector<bool,allocator>& x,
                  const vector<bool,allocator>& y);
  bool operator< (const vector<bool,allocator>& x,
                  const vector<bool,allocator>& y);
}
```

2 `reference` is a class that simulates the behavior of references of a single bit in `vector<bool>`.

23.3 Associative containers [lib.associative]

1 Headers `<map>` and `<set>`:

Header `<map>` synopsis

```
#include <memory>       // for allocator
#include <utility>      // for pair
#include <functional>   // for less
namespace std {
  template <class Key, class T, class Compare = less<Key>,
            class Allocator = allocator>
    class map;
  template <class Key, class T, class Compare, class Allocator>
    bool operator==(const map<Key,T,Compare,Allocator>& x,
                    const map<Key,T,Compare,Allocator>& y);
  template <class Key, class T, class Compare, class Allocator>
    bool operator< (const map<Key,T,Compare,Allocator>& x,
                    const map<Key,T,Compare,Allocator>& y);
```

```
  template <class Key, class T, class Compare = less<Key>,
            class Allocator = allocator>
    class multimap;
  template <class Key, class T, class Compare, class Allocator>
    bool operator==(const multimap<Key,T,Compare,Allocator>& x,
                    const multimap<Key,T,Compare,Allocator>& y);
  template <class Key, class T, class Compare, class Allocator>
    bool operator< (const multimap<Key,T,Compare,Allocator>& x,
                    const multimap<Key,T,Compare,Allocator>& y);
}
```

Header `<set>` synopsis

```
#include <memory>       // for allocator
#include <utility>      // for pair
#include <functional>   // for less
namespace std {
  template <class Key, class Compare = less<Key>, class Allocator = allocator>
    class set;
  template <class Key, class Compare, class Allocator>
    bool operator==(const set<Key,Compare,Allocator>& x,
                    const set<Key,Compare,Allocator>& y);
  template <class Key, class Compare, class Allocator>
    bool operator< (const set<Key,Compare,Allocator>& x,
                    const set<Key,Compare,Allocator>& y);

  template <class Key, class Compare = less<Key>, class Allocator = allocator>
    class multiset;
  template <class Key, class Compare, class Allocator>
    bool operator==(const multiset<Key,Compare,Allocator>& x,
                    const multiset<Key,Compare,Allocator>& y);
  template <class Key, class Compare, class Allocator>
    bool operator< (const multiset<Key,Compare,Allocator>& x,
                    const multiset<Key,Compare,Allocator>& y);
}
```

23.3.1 Template class `map` [lib.map]

1 A `map` is a kind of associative container that supports unique keys (contains at most one of each key value) and provides for fast retrieval of values of another type `T` based on the keys.

```
namespace std {
  template <class Key, class T, class Compare = less<Key>,
            class Allocator = allocator>
  class map {
  public:
  // types:
    typedef Key                key_type;
    typedef pair<const Key, T> value_type;
    typedef Compare            key_compare;
```

```
  class value_compare
    : public binary_function<value_type,value_type,bool> {
  friend class map;
  protected:
    Compare comp;
    value_compare(Compare c) : comp(c) {}
  public:
    bool operator()(const value_type& x, const value_type& y) {
      return comp(x.first, y.first);
    }
  };
```

ISSUE ***Are these typedefs correct?***

```
  typedef Allocator::types<T>::pointer       iterator;
  typedef Allocator::types<T>::const_pointer const_iterator;
  typedef Allocator::size_type       size_type;
  typedef Allocator::difference_type difference_type;

// construct/copy/destroy:
  map(const Compare& comp = Compare());
  template <class InputIterator>
    map(InputIterator first, InputIterator last,
        const Compare& comp = Compare());
  map(const map<Key,T,Compare,Allocator>& x);
 ~map();
  map<Key,T,Compare,Allocator>&
    operator=(const map<Key,T,Compare,Allocator>& x);

// observers:
  key_compare   key_comp() const;
  value_compare value_comp() const;
  iterator       begin();
  const_iterator begin() const;
  iterator       end();
  const_iterator end() const;
  bool      empty() const;
  size_type size() const;
  size_type max_size() const;
  T& operator[](const key_type& x);

// modifiers:
  pair<iterator, bool> insert(const value_type& x);
  iterator             insert(iterator position, const value_type& x);
  template <class InputIterator>
    void insert(InputIterator first, InputIterator last);
  void      erase(iterator position);
  size_type erase(const key_type& x);
  void      erase(iterator first, iterator last);

// map operations:
  iterator       find(const key_type& x);
  const_iterator find(const key_type& x) const;
  size_type      count(const key_type& x) const;
  iterator       lower_bound(const key_type& x);
  const_iterator lower_bound(const key_type& x) const;
  iterator       upper_bound(const key_type& x);
  const_iterator upper_bound(const key_type& x) const;
  pair<iterator,iterator>             equal_range(const key_type& x);
  pair<const_iterator,const_iterator> equal_range(const key_type& x) const;
};
```

```
  template <class Key, class T, class Compare, class Allocator>
    bool operator==(const map<Key,T,Compare,Allocator>& x,
                    const map<Key,T,Compare,Allocator>& y);
  template <class Key, class T, class Compare, class Allocator>
    bool operator< (const map<Key,T,Compare,Allocator>& x,
                    const map<Key,T,Compare,Allocator>& y);
}
```

23.3.1.1 map members [lib.map.members]

Box 100
To Be Specified

23.3.1.1.1 operator[] lib.map::subscript

```
T& operator[](const key_type& x);
```

Effects: (*((m.insert(make_pair(k, T()))).first)).second.

23.3.2 Template class multimap [lib.multimap]

1 A multimap is a kind of associative container that supports equal keys (possibly contains multiple copies of the same key value) and provides for fast retrieval of values of another type T based on the keys.

```
namespace std {
  template <class Key, class T, class Compare = less<Key>,
            class Allocator = allocator>
  class multimap {
  public:
  // types:
    typedef Key               key_type;
    typedef pair<const Key,T> value_type;
    typedef Compare           key_compare;

    class value_compare
      : public binary_function<value_type,value_type,bool> {
    friend class multimap;
    protected:
      Compare comp;
      value_compare(Compare c) : comp(c) {}
    public:
      bool operator()(const value_type& x, const value_type& y) {
        return comp(x.first, y.first);
      }
    };
```

ISSUE ***Are these typedefs correct?***

```
    typedef Allocator::types<T>::pointer       iterator;
    typedef Allocator::types<T>::const_pointer const_iterator;
    typedef Allocator::size_type       size_type;
    typedef Allocator::difference_type difference_type;
```

```
  // construct/copy/destroy:
    multimap(const Compare& comp = Compare());
    template <class InputIterator>
      multimap(InputIterator first, InputIterator last,
               const Compare& comp = Compare());
    multimap(const multimap<Key,T,Compare,Allocator>& x);
   ~multimap();
    multimap<Key,T,Compare,Allocator>&
      operator=(const multimap<Key,T,Compare,Allocator>& x);

  // observers:
    key_compare    key_comp() const;
    value_compare  value_comp() const;
    iterator       begin();
    const_iterator begin() const;
    iterator       end();
    const_iterator end() const;
    bool           empty() const;
    size_type      size() const;
    size_type      max_size() const;

  // modifiers:
    iterator insert(const value_type& x);
    iterator insert(iterator position, const value_type& x);
    template <class InputIterator>
      void insert(InputIterator first, InputIterator last);
    void      erase(iterator position);
    size_type erase(const key_type& x);
    void      erase(iterator first, iterator last);

  // multimap operations:
    iterator       find(const key_type& x);
    const_iterator find(const key_type& x) const;
    size_type      count(const key_type& x) const;
    iterator       lower_bound(const key_type& x);
    const_iterator lower_bound(const key_type& x) const;
    iterator       upper_bound(const key_type& x);
    const_iterator upper_bound(const key_type& x) const;
    pair<iterator,iterator>             equal_range(const key_type& x);
    pair<const_iterator,const_iterator> equal_range(const key_type& x) const;
  };

  template <class Key, class T, class Compare, class Allocator>
    bool operator==(const multimap<Key,T,Compare,Allocator>& x,
                    const multimap<Key,T,Compare,Allocator>& y);
  template <class Key, class T, class Compare, class Allocator>
    bool operator< (const multimap<Key,T,Compare,Allocator>& x,
                    const multimap<Key,T,Compare,Allocator>& y);
}
```

23.3.3 Template class `set` [lib.set]

1 A `set` is a kind of associative container that supports unique keys (contains at most one of each key value) and provides for fast retrieval of the keys themselves.

```
namespace std {
  template <class Key, class Compare = less<Key>, class Allocator = allocator>
  class set {
  public:
  // types:
    typedef Key     key_type;
    typedef Key     value_type;
    typedef Compare key_compare;
    typedef Compare value_compare;
    typedef Allocator::types<T>::pointer       iterator;
    typedef Allocator::types<T>::const_pointer const_iterator;
    typedef Allocator::size_type       size_type;
    typedef Allocator::difference_type difference_type;

  // construct/copy/destroy:
    set(const Compare& comp = Compare());
    template <class InputIterator>
      set(InputIterator first, InputIterator last,
          const Compare& comp = Compare());
    set(const set<Key,Compare,Allocator>& x);
   ~set();
    set<Key,Compare,Allocator>& operator=(const set<Key,Compare,Allocator>& x);

  // observers:
    key_compare   key_comp() const;
    value_compare value_comp() const;
    iterator      begin() const;
    iterator      end() const;
    bool          empty() const;
    size_type     size() const;
    size_type     max_size() const;

  // modifiers:
    pair<iterator,bool> insert(const value_type& x);
    iterator             insert(iterator position, const value_type& x);
    template <class InputIterator>
        void insert(InputIterator first, InputIterator last);
    void      erase(iterator position);
    size_type erase(const key_type& x);
    void      erase(iterator first, iterator last);

  // set operations:
    iterator  find(const key_type& x) const;
    size_type count(const key_type& x) const;
    iterator  lower_bound(const key_type& x) const;
    iterator  upper_bound(const key_type& x) const;
    pair<iterator,iterator> equal_range(const key_type& x) const;
  };

  template <class Key, class Compare, class Allocator>
    bool operator==(const set<Key,Compare,Allocator>& x,
                    const set<Key,Compare,Allocator>& y);
  template <class Key, class Compare, class Allocator>
    bool operator< (const set<Key,Compare,Allocator>& x,
                    const set<Key,Compare,Allocator>& y);
}
```

23.3.4 Template class `multiset` [lib.multiset]

1 A `multiset` is a kind of associative container that supports equal keys (possibly contains multiple copies of the same key value) and provides for fast retrieval of the keys themselves.

```
namespace std {
  template <class Key, class Compare = less<Key>, class Allocator = allocator>
  class multiset {
  public:
  // types:
    typedef Key     key_type;
    typedef Key     value_type;
    typedef Compare key_compare;
    typedef Compare value_compare;
    typedef Allocator::types<T>::pointer       iterator;
    typedef Allocator::types<T>::const_pointer const_iterator;
    typedef Allocator::size_type       size_type;
    typedef Allocator::difference_type difference_type;

  // construct/copy/destroy:
    multiset(const Compare& comp = Compare());
    template <class InputIterator>
      multiset(InputIterator first, InputIterator last,
               const Compare& comp = Compare());
    multiset(const multiset<Key,Compare,Allocator>& x);
   ~multiset();
    multiset<Key,Compare,Allocator>&
        operator=(const multiset<Key,Compare,Allocator>& x);

  // observers:
    key_compare   key_comp() const;
    value_compare value_comp() const;
    iterator      begin() const;
    iterator      end() const;
    bool          empty() const;
    size_type     size() const;
    size_type     max_size() const;

  // modifiers:
    iterator insert(const value_type& x);
    iterator insert(iterator position, const value_type& x);
    template <class InputIterator>
      void insert(InputIterator first, InputIterator last);
    void      erase(iterator position);
    size_type erase(const key_type& x);
    void      erase(iterator first, iterator last);

  // multiset operations:
    iterator  find(const key_type& x) const;
    size_type count(const key_type& x) const;
    iterator  lower_bound(const key_type& x) const;
    iterator  upper_bound(const key_type& x) const;
    pair<iterator,iterator> equal_range(const key_type& x) const;
  };
```

```
  template <class Key, class Compare, class Allocator>                          |
    bool operator==(const multiset<Key,Compare,Allocator>& x,                   |
                    const multiset<Key,Compare,Allocator>& y);                  |
  template <class Key, class Compare, class Allocator>                          |
    bool operator< (const multiset<Key,Compare,Allocator>& x,                   |
                    const multiset<Key,Compare,Allocator>& y);                  |
}                                                                               |
```

24 Iterators library [lib.iterators]

1 This clause describes components that C++ programs may use to perform iterations over containers (23), streams (_lib.default.iostreams_), and stream buffers (27.5).

2 The following subclauses describe iterator requirements, and components for iterator tags, predefined iterators, and stream iterators, as summarized in Table 53:

Table 53—Iterators library summary

Subclause	Header(s)
24.1 Requirements	
24.2 Iterator tags	
24.3 Predefined iterators	`<iterator>`
24.4 Stream iterators	

Header `<iterator>` synopsis

```
#include <cstddef>      // for ptrdiff_t
#include <utility>      // for empty
#include <iosfwd>       // for istream, ostream
#include <ios>          // for ios_traits
#include <streambuf>    // for streambuf
namespace std {
// subclause 24.2.2, primitives:
  struct input_iterator_tag;
  struct output_iterator_tag;
  struct forward_iterator_tag;
  struct bidirectional_iterator_tag;
  struct random_access_iterator_tag;

  template <class T, class Distance = ptrdiff_t>
    struct input_iterator;
  struct output_iterator;
  template <class T, class Distance = ptrdiff_t>
    struct forward_iterator;
  template <class T, class Distance = ptrdiff_t>
    struct bidirectional_iterator;
  template <class T, class Distance = ptrdiff_t>
    struct random_access_iterator;
```

```
  template <class T, class Distance>
    input_iterator_tag
      iterator_category(const input_iterator<T,Distance>&);
  output_iterator_tag iterator_category(const output_iterator&);
  template <class T, class Distance>
    forward_iterator_tag
      iterator_category(const forward_iterator<T,Distance>&);
  template <class T, class Distance>
    bidirectional_iterator_tag
      iterator_category(const bidirectional_iterator<T,Distance>&);
  template <class T, class Distance>
    random_access_iterator_tag
      iterator_category(const random_access_iterator<T,Distance>&);
  template <class T>
    random_access_iterator_tag iterator_category(const T*);

  template <class T, class Distance>
    T* value_type(const input_iterator<T,Distance>&);
  template <class T, class Distance>
    T* value_type(const forward_iterator<T,Distance>&);
  template <class T, class Distance>
    T* value_type(const bidirectional_iterator<T,Distance>&);
  template <class T, class Distance>
    T* value_type(const random_access_iterator<T,Distance>&);
  template <class T>
    T* value_type(const T*);

  template <class T, class Distance>
    Distance* distance_type(const input_iterator<T,Distance>&);
  template <class T, class Distance>
    Distance* distance_type(const forward_iterator<T,Distance>&);
  template <class T, class Distance>
    Distance* distance_type(const bidirectional_iterator<T,Distance>&);
  template <class T, class Distance>
    Distance* distance_type(const random_access_iterator<T,Distance>&);
  template <class T> ptrdiff_t* distance_type(const T*);
  template <class InputIterator, class Distance>
    void advance(InputIterator& i, Distance n);

  template <class InputIterator, class Distance>
    void distance(InputIterator first, InputIterator last, Distance& n);

// subclause 24.2.3, iterator operations:
  template <class InputIterator, class Distance>
    void advance(InputIterator& i, Distance n);

  template <class InputIterator, class Distance>
    void distance(InputIterator first, InputIterator last, Distance& n);
```

```
// subclause 24.3, predefined iterators:
  template <class BidirectionalIterator, class T, class Distance = ptrdiff_t>
  class reverse_bidirectional_iterator;
  template <class BidirectionalIterator, class T, class Distance>
    bool operator==(
      const reverse_bidirectional_iterator<BidirectionalIterator,T,Distance>& x,
      const reverse_bidirectional_iterator<BidirectionalIterator,T,Distance>& y);
  template <class RandomAccessIterator, class T, class Distance = ptrdiff_t>
  class reverse_iterator : public random_access_iterator<T,Distance>;
  template <class RandomAccessIterator, class T, class Distance>
    bool operator==(
      const reverse_iterator<RandomAccessIterator,T,Distance>& x,
      const reverse_iterator<RandomAccessIterator,T,Distance>& y);
  template <class RandomAccessIterator, class T, class Distance>
    bool operator<(
      const reverse_iterator<RandomAccessIterator,T,Distance>& x,
      const reverse_iterator<RandomAccessIterator,T,Distance>& y);
  template <class RandomAccessIterator, class T, class Distance>
    Distance operator-(
      const reverse_iterator<RandomAccessIterator,T,Distance>& x,
      const reverse_iterator<RandomAccessIterator,T,Distance>& y);
  template <class RandomAccessIterator, class T, class Distance>
    reverse_iterator<RandomAccessIterator,T,Distance> operator+(
      Distance n,
      const reverse_iterator<RandomAccessIterator,T,Distance>& x);
  template <class Container> class back_insert_iterator;
  template <class Container>
    back_insert_iterator<Container> back_inserter(Container& x);
  template <class Container> class front_insert_iterator;
  template <class Container>
    front_insert_iterator<Container> front_inserter(Container& x);
  template <class Container> class insert_iterator;
  template <class Container, class Iterator>
    insert_iterator<Container> inserter(Container& x, Iterator i);

// subclauses 24.4, stream iterators:
  template <class T, class Distance = ptrdiff_t>
    class istream_iterator;
  template <class T, class Distance>
    bool operator==(const istream_iterator<T,Distance>& x,
                    const istream_iterator<T,Distance>& y);
  template <class T> class ostream_iterator;
  template<class charT, class traits = ios_traits<charT> >
    class istreambuf_iterator;
  template <class charT, class traits = ios_traits<charT> >
    bool operator==(istreambuf_iterator<charT,traits>& a,
                    istreambuf_iterator<charT,traits>& b);
  template <class charT, class traits = ios_traits<charT> >
    bool operator!=(istreambuf_iterator<charT,traits>& a,
                    istreambuf_iterator<charT,traits>& b);
  template <class charT, class traits = ios_char_traits<charT> >
    class ostreambuf_iterator;
  output_iterator iterator_category (const ostreambuf_iterator&);
  template<class charT, class traits = ios_char_traits<charT> >
    bool operator==(ostreambuf_iterator<charT,traits>& a,
                    ostreambuf_iterator<charT,traits>& b);
  template<class charT, class traits = ios_char_traits<charT> >
    bool operator!=(ostreambuf_iterator<charT,traits>& a,
                    ostreambuf_iterator<charT,traits>& b);
}
```

24.1 Iterator requirements [lib.iterator.requirements]

1 Iterators are a generalization of pointers that allow a C++ program to work with different data structures (containers) in a uniform manner. To be able to construct template algorithms that work correctly and efficiently on different types of data structures, we need to formalize not just the interfaces but also the semantics and complexity assumptions of iterators. Iterators are objects that have `operator*` returning a value of some class or built-in type `T` called a *value type* of the iterator. For every iterator type `X` for which equality is defined, there is a corresponding signed integral type called the *distance type* of the iterator.

2 Since iterators are a generalization of pointers, their semantics is a generalization of the semantics of pointers in C++. This assures that every template function that takes iterators works with regular pointers. Depending on the operations defined on them, there are five categories of iterators: *input iterators*, *output iterators*, *forward iterators*, *bidirectional iterators* and *random access iterators*, as shown in Table 54.

Table 54—Relations among iterator categories

Random access	**→ Bidirectional**	**→ Forward**	**→ Input**
			→ Output

3 Forward iterators satisfy all the requirements of the input and output iterators and can be used whenever either kind is specified. Bidirectional iterators satisfy all the requirements of the forward iterators and can be used whenever a forward iterator is specified. Random access iterators satisfy all the requirements of bidirectional iterators and can be used whenever a bidirectional iterator is specified. There is an additional attribute that forward, bidirectional and random access iterators might have, that is, they can be *mutable* or *constant* depending on whether the result of the `operator*` behaves as a reference or as a reference to a constant. Constant iterators do not satisfy the requirements for output iterators.

4 Just as a regular pointer to an array guarantees that there is a pointer value pointing past the last element of the array, so for any iterator type there is an iterator value that points past the last element of a corresponding container. These values are called *past-the-end* values. Values of the iterator for which the `operator*` is defined are called *dereferenceable*. The library never assumes that past-the-end values are dereferenceable. Iterators might also have singular values that are not associated with any container. For example, after the declaration of an uninitialized pointer `x` (as with `int* x;`), `x` should alwa ys be assumed to have a singular value of a pointer. Results of most expressions are undefined for singular values. The only exception is an assignment of a non-singular value to an iterator that holds a singular value. In this case the singular value is overwritten the same way as any other value. Dereferenceable and past-the-end values are always non-singular.

5 An iterator `j` is called *reachable* from an iterator `i` if there is a finite sequence of application s of `operator++` to `i` that makes `i == j`. If `j` is reachable from `i`, they refer to the same container.

6 Most of the library's algorithmic templates that operate on data structures have interfaces that use ranges. A *range* is a pair of iterators that designate the beginning and end of the computation. A range `[i, i)` is an empty range; in general, a range `[i, j)` refers to the elements in the data structure starting with the one pointed to by `i` and up to but not including the one pointed to by `j`. Range `[i, j)` is valid if and only if `j` is reachable from `i`. The result of the application of the algorithms in the library to invalid ranges is undefined.

7 All the categories of iterators require only those functions that are realizable for a given category in constant time (amortized). Therefore, requirement tables for the iterators do not have a complexity column.

8 In the following sections, we assume: `a` and `b` are values of `X`, `n` is a value of the dist ance type `u`,`Distance`, `tmp`, and `m` are identifiers, `r` is a value of `X&`, `t` is a value of value type `T`.

24.1.1 Input iterators [lib.input.iterators]

1 A class or a built-in type X satisfies the requirements of an input iterator for the value type T if the fo llowing expressions are valid, as shown in Table 55:

Table 55—Input iterator requirements

expression	return type	operational semantics	assertion/note pre/post-condition
`X(a)`			`a == X(a)`. note: a destructor is assumed.
`X u(a);` `X u = a;`			post: `u == a`.
`a == b`	convertible to `bool`		`==` is an equivalence relation.
`a != b`	convertible to `bool`	`!(a == b)`	
`*a`	convertible to `T`		pre: `a` is dereferenceable. `a == b` implies `*a == *b`.
`++r`	`X&`		pre: `r` is dereferenceable. post: `r` is dereferenceable or `r` is past-the-end. `&r == &++r`.

`r++`	`X`	`{ X tmp = r;`
		`++r;`
		`return tmp; }`	

2 NOTE: For input iterators, `a == b` does not imply `++a == ++b`. (Equality does not guarantee the substitution property or referential transparency.) Algorithms on input iterators should never attempt to pass through the same iterator twice. They should be *single pass* algorithms. *Value type T is not required to be an lvalue type.* These algorithms can be used with istreams as the source of the input data through the `istream_iterator` class.

24.1.2 Output iterators [lib.output.iterators]

1 A class or a built-in type X satisfies the requirements of an output iterator if the following expressions are valid, as shown in Table 56:

Table 56—Output iterator requirements

expression	return type	operational semantics	assertion/note pre/post-condition
`X(a)`			`a = t` is equivalent to `X(a) = t`. note: a destructor is assumed.
`X u(a);` `X u = a;`			
`*a = t`	result is not used		pre: `a` is dereferenceable.
`++r`	`X&`		pre: `r` is dereferenceable. post: `r` is dereferenceable or `r` is past-the-end. `&a == &++a`.
`r++`	`X`	`{ X tmp = r; ++r; return tmp; }`	

2 NOTE: The only valid use of an `operator*` is on the left side of the assignment statement. *Assignment through the same value of the iterator happens only once.* Algorithms on output iterators should never attempt to pass through the same iterator twice. They should be *single pass* algorithms. Equality and inequality might not be defined. Algorithms that take output iterators can be used with ostreams as the destination for placing data through the `ostream_iterator` class as well as with insert iterators and insert pointers.

24.1.3 Forward iterators [lib.forward.iterators]

1 A class or a built-in type `X` satisfies the requirements of a forward iterator if the following expressions are valid, as shown in Table 57:

Table 57—Forward iterator requirements

expression	return type	operational semantics	assertion/note pre/post-condition
`X u;`			note: `u` might have a singular value. note: a destructor is assumed.
`X()`			note: `X()` might be singular.
`X(a)`			`a == X(a)`.
`X u(a);` `X u = a;`		`X u; u = a;`	post: `u == a`.
`a == b`	convertible to `bool`		`==` is an equivalence relation.
`a != b`	convertible to `bool`	`!(a == b)`	
`r = a`	X&		post: r == a.
`*a`	convertible to `T`		pre: `a` is dereferenceable. `a == b` implies `*a == *b`. If `X` is mutable, `*a = t` is valid.
`++r`	`X&`		pre: `r` is dereferenceable. post: `r` is dereferenceable or `r` is past-the-end. `r == s` and `r` is dereferenceable implies `++r == ++r`. `&a == &++a`.

`r++`	`X`	`{ X tmp = r;`
		`++r;`
		`return tmp; }`	

2 NOTE: The condition that `a == b` implies `++a == ++b` (which is not true for input and output iterators) and the removal of the restrictions on the number of the assignments through the iterator (which applies to output iterators) allows the use of multi-pass one-directional algorithms with forward iterators.

24.1.4 Bidirectional iterators [lib.bidirectional.iterators]

1 A class or a built-in type `X` satisfies the requirements of a bidirectional iterator if to the table that specifies forward iterators we add the following lines, as shown in Table 58:

Table 58—Bidirectional iterator requirements (in addition to forward iterator)

expression	return type	operational semantics	assertion/note pre/post-condition
`--r`	X&		pre: there exists `s` such that `r == ++s`. post: `s` is dereferenceable. `--(++r) == r`. `--r == --r` implies `r == s`. `&r == &--r`.

`r--`	X	`{ X tmp = r;`
		`--r;`
		`return tmp; }`	

2 NOTE: Bidirectional iterators allow algorithms to move iterators backward as well as forward.

24.1.5 Random access iterators [lib.random.access.iterators]

1 A class or a built-in type X satisfies the requirements of a random access iterator if to the table that specifies bidirectional iterators we add the following lines, as shown in Table 59:

Table 59—Random access iterator requirements (in addition to bidirectional iterator)

expression	return type	operational semantics	assertion/note pre/post-condition
`r += n`	`X&`	`{ Distance m = n; if (m >= 0) while (m--) ++r; else while (m++) --r; return r; }`	
`a + n` `n + a`	`X`	`{ X tmp = a; return tmp += n; }`	`a + n == n + a`.
`r -= n`	`X&`	`return r += -n;`	
`a - n`	`X`	`{ X tmp = a; return tmp -= n; }`	
`b - a`	`Distance`	`{ X tmp = a; Distance m = 0; while (tmp != b) ++tmp, ++m; return m; }`	pre: there exists a value `n` of `Distance` such that `a + n == b`. `b == a + (b - a)`.
`a[n]`	convertible to `T`	`*(a + n)`	
`a < b`	convertible to `bool`	`b - a > 0`	`<` is a total ordering relation
`a > b`	convertible to bool	`b < a`	`>` is a total ordering relation opposite to `<`.
`a >= b`	convertible to bool	`!(a < b)`	
`a <= b`	convertible to bool	`!(a > b)`	

24.2 Iterator tags [lib.iterator.tags]

1 To implement algorithms only in terms of iterators, it is often necessary to infer both of the value type and the distance type from the iterator. To enable this task it is required that for an iterator `i` of any category other than output iterator, the expression `value_type(i)` returns `(T*)(0)` and the expression `distance_type(i)` returns `(Distance*)(0)`. For output iterators, these expressions are not required.

24.2.1 Examples of using iterator tags [lib.examples]

1 For all the regular pointer types we can define `value_type` and `distance_type` with the help of:

```
template <class T>
inline T* value_type(const T*) { return (T*)(0); }
template <class T>
inline ptrdiff_t* distance_type(const T*) { return (ptrdiff_t*)(0); }
```

2 Then, if we want to implement a generic `reverse` function, we do the following:

```
template <class BidirectionalIterator>
inline void reverse(BidirectionalIterator first, BidirectionalIterator last) {
  __reverse(first, last, value_type(first), distance_type(first));
}
```

3 where `__reverse` is defined as:

```
template <class BidirectionalIterator, class T, class Distance>
void __reverse(BidirectionalIterator first, BidirectionalIterator last, T*,
               Distance*)
{
  Distance n;
  distance(first, last, n); // see Iterator operations section
  --n;
  while (n > 0) {
    T tmp = *first;
    *first++ = *--last;
    *last = tmp;
    n -= 2;
  }
}
```

4 If there is an additional pointer type `far` such that the difference of two far pointers is of the type `long`, we define:

```
template <class T>
inline T* value_type(const T far *) { return (T*)(0); }
template <class T>
inline long* distance_type(const T far *) { return (long*)(0); }
```

5 It is often desirable for a template function to find out what is the most specific category of its iterator argument, so that the function can select the most efficient algorithm at compile time. To facilitate this, the library introduces *category tag* classes which are used as compile time tags for algorithm selection. They are: `input_iterator_tag`, `output_iterator_tag`, `forward_iterator_tag`, `bidirectional_iterator_tag` and `random_access_iterator_tag`. Every iterator `i` must have an expression `iterator_category(i)` defined on it that returns the most specific category tag that describes its behavior. For example, we define that all the pointer types are in the random access iterator category by:

```
template <class T>
inline random_access_iterator_tag iterator_category(T*) {
  return random_access_iterator_tag();
}
```

6 For a user-defined iterator `BinaryTreeIterator`, it can be included into the bidirectional iterator category by saying:

```
template <class T>
inline bidirectional_iterator_tag iterator_category(
  const BinaryTreeIterator<T>&) {
  return bidirectional_iterator_tag();
}
```

7 If a template function `evolve` is well defined for bidirectional iterators, but can be implemented more efficiently for random access iterators, then the implementation is like:

```
template <class BidirectionalIterator>
inline void evolve(BidirectionalIterator first, BidirectionalIterator last) {
  evolve(first, last, iterator_category(first));
}

template <class BidirectionalIterator>
void evolve(BidirectionalIterator first, BidirectionalIterator last,
            bidirectional_iterator_tag) {
// ... more generic, but less efficient algorithm
}

template <class RandomAccessIterator>
void evolve(RandomAccessIterator first, RandomAccessIterator last,
  random_access_iterator_tag) {
// ... more efficient, but less generic algorithm
}
```

8 If a user wants to define a bidirectional iterator for some data structure containing `double` and such that it works on a large memory model of his computer, he can do it with:

```
class MyIterator : public bidirectional_iterator<double, long> {
// code implementing ++, etc.
};
```

9 Then there is no need to define `iterator_category`, `value_type`, and `distance_type` on `MyIterator`.

24.2.2 Library defined primitives [lib.library.primitives]

1 To simplify the task of defining the `iterator_category`, `value_type` and `distance_type` for user def inable iterators, the library provides the following predefined classes and functions:

24.2.2.1 Standard iterator tags [lib.std.iterator.tags]

```
namespace std {
  struct input_iterator_tag : empty {};
  struct output_iterator_tag : empty {};
  struct forward_iterator_tag : empty {};
  struct bidirectional_iterator_tag : empty {};
  struct random_access_iterator_tag : empty {};
}
```

24.2.2.2 Basic iterators [lib.basic.iterators]

```
namespace std {
  template <class T, class Distance = ptrdiff_t>
    struct input_iterator : empty{};
  struct output_iterator : empty{};
  template <class T, class Distance = ptrdiff_t>
    struct forward_iterator : empty{};
  template <class T, class Distance = ptrdiff_t>
    struct bidirectional_iterator : empty {};
  template <class T, class Distance = ptrdiff_t>
    struct random_access_iterator : empty {};
}
```

1 `output_iterator` is not a template because output iterators do not have either value type or distance type defined.

24.2.2.3 iterator_category [lib.iterator.category]

```
template <class T, class Distance>
  input_iterator_tag
    iterator_category(const input_iterator<T,Distance>&);
```

Returns: input_iterator_tag().

```
  output_iterator_tag iterator_category(const output_iterator&);
```

Returns: output_iterator_tag().

```
  template <class T, class Distance>
    forward_iterator_tag
      iterator_category(const forward_iterator<T,Distance>&);
```

Returns: forward_iterator_tag().

```
  template <class T, class Distance>
    bidirectional_iterator_tag
      iterator_category(const bidirectional_iterator<T,Distance>&);
```

Returns: bidirectional_iterator_tag().

```
  template <class T, class Distance>
    random_access_iterator_tag
      iterator_category(const random_access_iterator<T,Distance>&);
```

Returns: random_access_iterator_tag().

```
  template <class T>
    random_access_iterator_tag iterator_category(const T*);
```

Returns: random_access_iterator_tag().

24.2.2.4 value_type [lib.value.type]

```
template <class T, class Distance>
  T* value_type(const input_iterator<T,Distance>&);
```

Returns: (T*)(0).

```
  template <class T, class Distance>
    T* value_type(const forward_iterator<T,Distance>&);
```

Returns: (T*)(0).

```
  template <class T, class Distance>
    T* value_type(const bidirectional_iterator<T,Distance>&);
```

Returns: (T*)(0).

```
  template <class T, class Distance>
    T* value_type(const random_access_iterator<T,Distance>&);
```

Returns: (T*)(0).

```
  template <class T>
    T* value_type(const T*);
```

Returns: `(T*)(0)`.

24.2.2.5 `distance_type` [lib.distance.type]

```
template <class T, class Distance>
  Distance* distance_type(const input_iterator<T,Distance>&);
```

Returns: `(Distance*)(0)`.

```
  template <class T, class Distance>
    Distance* distance_type(const forward_iterator<T,Distance>&);
```

Returns: `(Distance*)(0)`.

```
  template <class T, class Distance>
    Distance* distance_type(const bidirectional_iterator<T,Distance>&);
```

Returns: `(Distance*)(0)`.

```
  template <class T, class Distance>
    Distance* distance_type(const random_access_iterator<T,Distance>&);
```

Returns: `(Distance*)(0)`.

```
  template <class T> ptrdiff_t* distance_type(const T*);
```

Returns: `(ptrdiff_t*)(0)`.

24.2.3 Iterator operations [lib.iterator.operations]

1 Since only random access iterators provide `+` and `-` operators, the library provides two template functions `advance` and `distance`. These functions use `+` and `-` for random access iterators (and are, therefore, constant time for them); for input, forward and bidirectional iterators they use `++` to provide linear time implementations.

```
template <class InputIterator, class Distance>
  void advance(InputIterator& i, Distance n);
```

Requires: `n` may be negative only for random access and bidirectional iterators.
Effects: Increments (or decrements for negative `n`) iterator reference `i` by `n`.

```
  template <class InputIterator, class Distance>
    void distance(InputIterator first, InputIterator last, Distance& n);
```

Effects: Increments `n` by the number of times it takes to get from `first` to [153] `last`.

24.3 Predefined iterators [lib.predef.iterators]

24.3.1 Reverse iterators [lib.reverse.iterators]

1 Bidirectional and random access iterators have corresponding reverse iterator adaptors that iterate through the data structure in the opposite direction. They have the same signatures as the corresponding iterators. The fundamental relation between a reverse iterator and its corresponding iterator `i` is established by the identity

[153] `distance` must be a three argument function storing the result into a reference instead of returning the result because the distance type cannot be deduced from built-in iterator types such as `int*`.

`&*(reverse_iterator(i)) == &*(i - 1)`.

2 This mapping is dictated by the fact that while there is always a pointer past the end of an array, there might not be a valid pointer before the beginning of an array.

3 The formal class parameter `T` of reverse iterators should be instantiated with the type that `Iterator::operator*` returns, which is usually a reference type. For example, to obtain a reverse iterator for `int*`, one should declare `reverse_iterator<int*, int&>`. To obtain a constant reverse iterator for `int*`, one should declare `reverse_iterator<const int*, const int&>`. The interface thus allows one to use reverse iterators with those iterator types for which `operator*` returns something other than a reference type.

24.3.1.1 Template class `reverse_bidirectional_iterator` [lib.reverse.bidir.iter]

```
namespace std {
  template <class BidirectionalIterator, class T, class Distance = ptrdiff_t>
  class reverse_bidirectional_iterator
    : public bidirectional_iterator<T,Distance> {
  protected:
    BidirectionalIterator current;
  public:
    reverse_bidirectional_iterator();
    reverse_bidirectional_iterator(BidirectionalIterator x);
    operator BidirectionalIterator();
    T operator*();
    reverse_bidirectional_iterator<BidirectionalIterator,T,Distance>&
      operator++();
    reverse_bidirectional_iterator<BidirectionalIterator,T,Distance>
      operator++(int);
    reverse_bidirectional_iterator<BidirectionalIterator,T,Distance>&
      operator--();
    reverse_bidirectional_iterator<BidirectionalIterator,T,Distance>
      operator--(int);
  };

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

24.3.1.2 `reverse_bidirectional_iterator` operations [lib.reverse.bidir.iter.ops]

24.3.1.2.1 `reverse_bidirectional_iterator` constructor [lib.reverse.bidir.iter.cons]

```
reverse_bidirectional_iterator(BidirectionalIterator x);
```

Effects: `: current(x) {}`

24.3.1.2.2 Conversion [lib.reverse.bidir.iter.conv]

```
operator BidirectionalIterator();
```

Returns: `current`

24.3.1.2.3 operator* [lib.reverse.bidir.iter.op.star]

```
T operator*();
```

Effects:

```
BidirectionalIterator tmp = current;
return *--tmp;
```

24.3.1.2.4 operator++ [lib.reverse.bidir.iter.op++]

```
reverse_bidirectional_iterator<BidirectionalIterator,T,Distance>&
  operator++();
```

Effects: `--current;`
Returns: `*this`

```
reverse_bidirectional_iterator<BidirectionalIterator,T,Distance>
  operator++(int);
```

Effects:

```
reverse_bidirectional_iterator<BidirectionalIterator,T,Distance>
  tmp = *this;
  --current;
  return tmp;
```

24.3.1.2.5 operator-- [lib.reverse.bidir.iter.op--]

```
reverse_bidirectional_iterator<BidirectionalIterator,T,Distance>&
  operator--();
```

Effects: `++current`
Returns:

```
reverse_bidirectional_iterator<BidirectionalIterator,T,Distance>
  operator--(int);
```

Effects:

```
reverse_bidirectional_iterator<BidirectionalIterator,T,Distance>
  tmp = *this;
++current;
return tmp;
```

24.3.1.2.6 operator== [lib.reverse.bidir.iter.op==]

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

Returns: `BidirectionalIterator(x) == BidirectionalIterator(y).`

24.3.1.3 Template class reverse_iterator [lib.reverse.iterator]

```
namespace std {
  template <class RandomAccessIterator, class T, class Distance = ptrdiff_t>
  class reverse_iterator : public random_access_iterator<T,Distance> {
  protected:
    RandomAccessIterator current;
  public:
    reverse_iterator();
    reverse_iterator(RandomAccessIterator x);

    operator RandomAccessIterator();
    T operator*();

    reverse_iterator<RandomAccessIterator,T,Distance>& operator++();
    reverse_iterator<RandomAccessIterator,T,Distance> operator++(int);
    reverse_iterator<RandomAccessIterator,T,Distance>& operator--();
    reverse_iterator<RandomAccessIterator,T,Distance> operator--(int);

    reverse_iterator<RandomAccessIterator,T,Distance>
      operator+(Distance n) const;
    reverse_iterator<RandomAccessIterator,T,Distance>&
      operator+=(Distance n) const;
    reverse_iterator<RandomAccessIterator,T,Distance>
      operator-(Distance n) const;
    reverse_iterator<RandomAccessIterator,T,Distance>
      operator-(Distance n) const;
    T operator[](Distance n);

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

    template <class RandomAccessIterator, class T, class Distance>
      bool operator<(
        const reverse_iterator<RandomAccessIterator,T,Distance>& x,
        const reverse_iterator<RandomAccessIterator,T,Distance>& y);

    template <class RandomAccessIterator, class T, class Distance>
      Distance operator-(
        const reverse_iterator<RandomAccessIterator,T,Distance>& x,
        const reverse_iterator<RandomAccessIterator,T,Distance>& y);

    template <class RandomAccessIterator, class T, class Distance>
      reverse_iterator<RandomAccessIterator,T,Distance> operator+(
        Distance n,
        const reverse_iterator<RandomAccessIterator,T,Distance>& x);
  };
}
```

1 There is no way a default for `T` can be expressed in terms of `BidirectionalIterator` because the value type cannot be deduced from built-in iterators such as `int*`. Otherwise, we would have written

```
  template <class BidirectionalIterator,
    class T = BidirectionalIterator::reference_type,
    class Distance = BidirectionalIterator::difference_type>
  class reverse_bidirectional_iterator: bidirectional_iterator<T,Distance> {
  /* ... */
  };
```

24.3.1.4 `reverse_iterator` operations [lib.reverse.iter.ops]

24.3.1.4.1 `reverse_iterator` constructor [lib.reverse.iter.cons]

```
reverse_iterator(RandomAccessIterator x);
```

Effects: `: current(x) {}`

24.3.1.4.2 Conversion [lib.reverse.iter.conv]

```
operator RandomAccessIterator();
```

Returns: `current`

24.3.1.4.3 `operator*` [lib.reverse.iter.op.star]

```
T operator*();
```

Effects:

```
RandomAccessIterator tmp = current;
return *--tmp;
```

24.3.1.4.4 `operator++` [lib.reverse.iter.op++]

```
reverse_iterator<RandomAccessIterator,T,Distance>&
  operator++();
```

Effects: `--current;`
Returns: `*this`

```
  reverse_iterator<RandomAccessIterator,T,Distance>
    operator++(int);
```

Effects:

```
reverse_iterator<RandomAccessIterator,T,Distance>
  tmp = *this;
  --current;
  return tmp;
```

24.3.1.4.5 `operator--` [lib.reverse.iter.op--]

```
reverse_iterator<RandomAccessIterator,T,Distance>&
  operator--();
```

Effects: `++current`
Returns:

```
  reverse_iterator<RandomAccessIterator,T,Distance>
    operator--(int);
```

Effects:

```
reverse_iterator<RandomAccessIterator,T,Distance>
  tmp = *this;
++current;
return tmp;
```

24.3.1.4.6 operator== [lib.reverse.iter.op==]

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

Returns: `x.current == y.current`

24.3.2 Insert iterators [lib.insert.iterators]

1 To make it possible to deal with insertion in the same way as writing into an array, a special kind of iterator adaptors, called *insert iterators*, are provided in the library. With regular iterator classes,

```
while (first != last) *result++ = *first++;
```

2 causes a range `[first, last)` to be copied into a range starting with result. The same code with `result` being an insert iterator will insert corresponding elements into the container. This device allows all of the copying algorithms in the library to work in the *insert mode* instead of the regular overwrite mode.

3 An insert iterator is constructed from a container and possibly one of its iterators pointing to where insertion takes place if it is neither at the beginning nor at the end of the container. Insert iterators satisfy the requirements of output iterators. `operator*` returns the insert iterator itself. The assignment `operator=(const T& x)` is defined on insert iterators to allow writing into them, it inserts x right before where the insert iterator is pointing. In other words, an insert iterator is like a cursor pointing into the container where the insertion takes place. `back_insert_iterator` inserts elements at the end of a container, `front_insert_iterator` inserts elements at the beginning of a container, and `insert_iterator` inserts elements where the iterator points to in a container. `back_inserter`, `front_inserter`, and `inserter` are three functions making the insert iterators out of a container.

24.3.2.1 Template class back_insert_iterator [lib.back.insert.iterator]

```
namespace std {
  template <class Container>
  class back_insert_iterator : public output_iterator {
  protected:
    Container& container;

  public:
    back_insert_iterator(Container& x);
    back_insert_iterator<Container>&
    operator=(const Container::value_type& value);

    back_insert_iterator<Container>& operator*();
    back_insert_iterator<Container>& operator++();
    back_insert_iterator<Container> operator++(int);
  };

  template <class Container>
    back_insert_iterator<Container> back_inserter(Container& x);
```

24.3.2.2 back_insert_iterator operations [lib.back.insert.iter.ops]

24.3.2.2.1 back_insert_iterator constructor [lib.back.insert.iter.cons]

```
back_insert_iterator(Container& x);
```

Effects: : container(x) {}

24.3.2.2.2 back_insert_iterator::operator= [lib.back.insert.iter.op=]

```
back_insert_iterator<Container>&
  operator=(const Container::value_type& value);
```

Effects: container.push_back(value);
Returns: *this.

24.3.2.2.3 back_insert_iterator::operator* [lib.back.insert.iter.op*]

```
back_insert_iterator<Container>& operator*();
```

Returns: *this.

24.3.2.2.4 back_insert_iterator::operator++ [lib.back.insert.iter.op++]

```
back_insert_iterator<Container>& operator++();
back_insert_iterator<Container>  operator++(int);
```

Returns: *this.

24.3.2.2.5 back_inserter [lib.back.inserter]

```
template <class Container>
  back_insert_iterator<Container> back_inserter(Container& x);
```

Returns: back_insert_iterator<Container>(x).

24.3.2.3 Template class front_insert_iterator [lib.front.insert.iterator]

```
namespace std {
  template <class Container>
  class front_insert_iterator : public output_iterator {
  protected:
    Container& container;

  public:
    front_insert_iterator(Container& x);
    front_insert_iterator<Container>&
      operator=(const Container::value_type& value);

    front_insert_iterator<Container>& operator*();
    front_insert_iterator<Container>& operator++();
    front_insert_iterator<Container> operator++(int);
  };

  template <class Container>
    front_insert_iterator<Container> front_inserter(Container& x);
}
```

Returns: front_insert_iterator<Container>(x).

24.3.2.4 `front_insert_iterator` operations [lib.front.insert.iter.ops]

24.3.2.4.1 `front_insert_iterator` constructor [lib.front.insert.iter.cons]

```
front_insert_iterator(Container& x);
```

Effects: `: container(x) {}`

24.3.2.4.2 `front_insert_iterator::operator=` [lib.front.insert.iter.op=]

```
front_insert_iterator<Container>&
  operator=(const Container::value_type& value);
```

Effects: `container.push_front(value);`
Returns: `*this`.

24.3.2.4.3 `front_insert_iterator::operator*` [lib.front.insert.iter.op*]

```
front_insert_iterator<Container>& operator*();
```

Returns: `*this`.

24.3.2.4.4 `front_insert_iterator::operator++` [lib.front.insert.iter.op++]

```
front_insert_iterator<Container>& operator++();
front_insert_iterator<Container>  operator++(int);
```

Returns: `*this`.

24.3.2.4.5 `front_inserter` [lib.front.inserter]

```
template <class Container>
  front_insert_iterator<Container> front_inserter(Container& x);
```

Returns: `front_insert_iterator<Container>(x)`.

24.3.2.5 Template class `insert_iterator` [lib.insert.iterator]

```
namespace std {
  template <class Container>
  class insert_iterator : public output_iterator {
  protected:
    Container& container;
    Container::iterator iter;

  public:
    insert_iterator(Container& x, Container::iterator i);
    insert_iterator<Container>& operator=(const Container::value_type& value);

    insert_iterator<Container>& operator*();
    insert_iterator<Container>& operator++();
    insert_iterator<Container> operator++(int);
  };

  template <class Container, class Iterator>
    insert_iterator<Container> inserter(Container& x, Iterator i);
}
```

24.3.2.6 `insert_iterator` operations [lib.insert.iter.ops]

24.3.2.6.1 `insert_iterator` constructor [lib.insert.iter.cons]

```
insert_iterator(Container& x);
```

Effects: `: container(x), iter(i) {}`

24.3.2.6.2 `insert_iterator::operator=` [lib.insert.iter.op=]

```
insert_iterator<Container>&
  operator=(const Container::value_type& value);
```

Effects:

```
  iter = container.insert(iter, value);
  ++iter;
```

Returns: `*this`.

24.3.2.6.3 `insert_iterator::operator*` [lib.insert.iter.op*]

```
insert_iterator<Container>& operator*();
```

Returns: `*this`.

24.3.2.6.4 `insert_iterator::operator++` [lib.insert.iter.op++]

```
insert_iterator<Container>& operator++();
insert_iterator<Container>  operator++(int);
```

Returns: `*this`.

24.3.2.6.5 `inserter` [lib.inserter]

```
template <class Container>
  insert_iterator<Container> inserter(Container& x);
```

Returns: `insert_iterator<Container>(x,Container::iterator(i))`.

24.4 Stream iterators [lib.stream.iterators]

1 To make it possible for algorithmic templates to work directly with input/output streams, appropriate iterator-like template classes are provided. For example,

```
partial_sum_copy(istream_iterator<double>(cin),  istream_iterator<double>(),
  ostream_iterator<double>(cout, "n"));
```

2 reads a file containing floating point numbers from cin, and prints the partial sums onto cout.

24.4.1 Template class `istream_iterator` [lib.istream.iterator]

1 `istream_iterator<T>` reads (using `operator>>`) successive elements from the input stream for which it was constructed. After it is constructed, and every time `++` is used, the iterator reads and stores a value of `T`. If the end of stream is reached (`operator void*()` on the stream returns `false`), the iterator becomes equal to the *end-of-stream* iterator value. The constructor with no arguments `istream_iterator()` always constructs an end of stream input iterator object, which is the only legitimate iterator to be used for the end condition. The result of `operator*` on an end of stream is not defined. For any other iterator value a `const T&` is returned. It is impossible to store things into istream iterators. The main peculiarity of the istream iterators is the fact that `++` operators are not equality

preserving, that is, `i == j` does not guarantee at all that `++i == ++j`. Every time `++` is used a new value is read.

2 The practical consequence of this fact is that istream iterators can be used only for one-pass algorithms, which actually makes perfect sense, since for multi-pass algorithms it is always more appropriate to use in-memory data structures. Two end-of-stream iterators are always equal. An end-of-stream iterator is not equal to a non-end-of-stream iterator. Two non-end-of-stream iterators are equal when they are constructed from the same stream.

```
namespace std {
  template <class T, class Distance = ptrdiff_t>
  class istream_iterator : input_iterator<T,Distance> {
  public:
    istream_iterator();
    istream_iterator(istream& s);
    istream_iterator(const istream_iterator<T,Distance>& x);
   ~istream_iterator();

    const T& operator*() const;
    istream_iterator<T,Distance>& operator++();
    istream_iterator<T,Distance>  operator++(int);
  };

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

24.4.2 Template class `ostream_iterator` [lib.ostream.iterator]

1 `ostream_iterator<T>` writes (using `operator<<`) successive elements onto the output stream from which it was constructed. If it was constructed with `char*` as a constructor argument, this string, called a *delimiter string*, is written to the stream after every `T` is written. It is not possible to get a value out of the output iterator. Its only use is as an output iterator in situations like

```
while (first != last) *result++ = *first++;
```

2 `ostream_iterator` is defined as:

```
namespace std {
  template <class T>
  class ostream_iterator : public output_iterator {
  public:
    ostream_iterator(ostream& s);
    ostream_iterator(const char* delimiter);
    ostream_iterator(ostream& s, const char* delimiter);
    ostream_iterator(const ostream_iterator<T>& x);
   ~ostream_iterator();
    ostream_iterator<T>& operator=(const T& value);

    ostream_iterator<T>& operator*();
    ostream_iterator<T>& operator++();
    ostream_iterator<T>  operator++(int);
  };
```

24.4.3 Template class `istreambuf_iterator` [lib.istreambuf.iterator]

```
namespace std {
  template<class charT, class traits = ios_traits<charT> >
  class istreambuf_iterator {
  public:
    typedef charT                          char_type;
    typedef traits                         traits_type;
    typedef traits::int_type               int_type;
    typedef basic_streambuf<charT,traits> streambuf;
    typedef basic_istream<charT,traits>   istream;

    class proxy;

    public:
      istreambuf_iterator();
      istreambuf_iterator(istream& s);
      istreambuf_iterator(streambuf* s);
      istreambuf_iterator(const proxy& p);
      charT operator*();
      istreambuf_iterator<charT,traits>& operator++();
      proxy operator++(int);
      bool equal(istreambuf_iterator& b);
    private:
      streambuf* sbuf_; exposition only
  };
}
```

1 The template class `istreambuf_iterator` reads successive *characters* from the streambuf for which it was constructed.

2 After it is constructed, and every time `operator++` is used, the iterator reads and stores a value of *character*. If the end of stream is reached (streambuf::sgetc() returns `traits::eof()`), the iterator becomes equal to the *end of stream* iterator value. The default constructor `istreambuf_iterator()` and the constructor `istreambuf_iterator(0)` always construct an end of stream iterator object, which is the only legitimate iterator to be used for the end condition.

3 The result of `operator*()` on an end of stream is undefined. For any other iterator value a `const char_type&` is returned. It is impossible to store things into input iterators.

4 Note that in the input iterators, `++` operators are not *equality preserving*, that is, `i == j` does not guarantee at all that `++i == ++j`. Every time `++` is used a new value is used.

5 The practical consequence of this fact is that an `istreambuf_iterator` object can be used only for *one-pass algorithms*, which actually makes perfect sense, since for multi-pass algorithms it is always more appropriate to use in-memory data structures. Two end of stream iterators are always equal. An end of stream iterator is not equal to a non-end of stream iterator. Two non-end of stream iterators are equal when they are constructed from the same stream.

24.4.3.1 Template class `istreambuf_iterator::proxy` [lib.istreambuf.iterator::proxy]

```
namespace std {
  template <class charT, class traits = ios_traits<charT> >
  class istream_iterator::proxy {
    charT keep_;
    streambuf* sbuf_;
    proxy(charT c, streambuf* sbuf);
    : keep_(c), sbuf_(sbuf) {}
  public:
    charT operator*() { return keep_; }
    friend class istreambuf_iterator;
  };
}
```

1 Class `istream_iterator<charT,traits>::proxy` provides a temporal placeholder as the return value of the post-increment operator (`operator++`). It keeps the character pointed to by the previous value of the iterator for some possible future access to get the character.

24.4.3.2 `istreambuf_iterator` constructors [lib.istreambuf.iterator.cons]

```
istreambuf_iterator();
```

Effects: Constructs the end-of-stream iterator.

```
istreambuf_iterator(basic_istream<charT,traits>& s);
```

Effects: Constructs the `istream_iterator` pointing to the `basic_streambuf` object `*(s.rdbuf())`.

```
istreambuf_iterator(const proxy& p);
```

Effects: Constructs the `istreambuf_iterator` pointing to the `basic_streambuf` object related to the `proxy` object *p*.

24.4.3.3 `istreambuf_iterator::operator*` [lib.istreambuf.iterator::op*]

```
charT operator*()
```

1 Extract one character pointed to by the `streambuf *`*`sbuf_`*.

24.4.3.4 `istreambuf_iterator::operator++` [lib.istreambuf.iterator::op++]

```
istreambuf_iterator<charT,traits>&
    istreambuf_iterator<charT,traits>::operator++();
```

Effects: Advances the iterator and returns the result

```
proxy istreambuf_iterator<charT,traits>::operator++(int);
```

Effects: Advances the iterator and returns the `proxy` object keeping the character pointed to by the previous iterator.

24.4.3.5 `istreambuf_iterator::equal` [lib.istreambuf.iterator::equal]

```
bool equal(istreambuf_iterator<charT,traits>& b);
```

Returns: `true` if the iterators are equal. Equality is defined as follows:

— If both *a* and *b* are end-of-stream iterators, *a == b*.

istreambuf_iterator::equal

— If either *a* or *b* is an end-of-stream iterator, if the other points end-of-file, *a == b*, otherwise *a != b*.

— If both *a* and *b* are not end-of-stream, the two `streambuf` pointed to by the both iterators are compared.

24.4.3.6 `iterator_category` [lib.iterator.category.i]

```
input_iterator iterator_category(const istreambuf_iterator& s);
```

Returns: the category of the iterator *s*.

24.4.3.7 `operator==` [lib.istreambuf.iterator::op==]

```
namespace std {
  template <class charT, class traits = ios_traits<charT> >
    bool operator==(istreambuf_iterator<charT,traits>& a,
                    istreambuf_iterator<charT,traits>& b);
}
```

Returns: `a.equal(b)`.

24.4.3.8 `operator!=` [lib.istreambuf.iterator::op!=]

```
namespace std {
  template <class charT, class traits = ios_traits<charT> >
    bool operator!=(istreambuf_iterator<charT,traits>& a,
                    istreambuf_iterator<charT,traits>& b);
}
```

Returns: `!a.equal(b)`.

24.4.4 Template class `ostreambuf_iterator` [lib.ostreambuf.iterator]

```
namespace std {
  template <class charT, class traits = ios_char_traits<charT> >
  class ostreambuf_iterator {
  public:
    typedef charT                          char_type;
    typedef traits                         traits_type;
    typedef basic_streambuf<charT,traits> streambuf;
    typedef basic_ostream<charT,traits>   ostream;

  public:
    ostreambuf_iterator();
    ostreambuf_iterator(ostream& s);
    ostreambuf_iterator(streambuf* s);
    ostreambuf_iterator& operator=(charT c);

    ostreambuf_iterator& operator*();
    ostreambuf_iterator& operator++();
    ostreambuf_iterator& operator++(int);

    bool equal(ostreambuf_iterator& b);

  private:
    streambuf* sbuf_;    exposition only

  };
```

```
  output_iterator iterator_category (const ostreambuf_iterator&);

  template<class charT, class traits = ios_char_traits<charT> >
    bool operator==(ostreambuf_iterator<charT,traits>& a,
                    ostreambuf_iterator<charT,traits>& b);
  template<class charT, class traits = ios_char_traits<charT> >
    bool operator!=(ostreambuf_iterator<charT,traits>& a,
                    ostreambuf_iterator<charT,traits>& b);
}
```

1 The template class `ostreambuf_iterator` writes successive *characters* onto the output stream from which it was constructed. It is not possible to get a value out of the output iterator.

2 Two output iterators are equal if they are constructed with the same output streambuf.

24.4.4.1 `ostreambuf_iterator` operations [lib.ostreambuf.iter.ops]

24.4.4.1.1 `ostreambuf_iterator` constructors [lib.ostreambuf.iter.cons]

```
ostreambuf_iterator();
```

Effects: `: sbuf_(0) {}`

```
ostreambuf_iterator(ostream& s);
```

Effects: `: sbuf_(s.rdbuf()) {}`

```
ostreambuf_iterator(streambuf* s);
```

Effects: `: sbuf_(s) {}`

24.4.4.1.2 `ostreambuf_iterator::operator=` [lib.ostreambuf.iter.op=]

```
ostreambuf_iterator<Container>&
  operator=(const Container::value_type& value);
```

Effects:

```
  sbuf_->sputc(traits::to_int_type(c));
```

Returns: `*this`.

24.4.4.1.3 `ostreambuf_iterator::operator*` [lib.ostreambuf.iter.op*]

```
ostreambuf_iterator<Container>& operator*();
```

Returns: `*this`.

24.4.4.1.4 `ostreambuf_iterator::operator++` [lib.ostreambuf.iter.op++]

```
ostreambuf_iterator<Container>& operator++();
ostreambuf_iterator<Container>  operator++(int);
```

Returns: `*this`.

24.4.4.1.5 ostreambuf_iterator::equal [lib.ostreambuf.iter.equal]

```
bool equal(ostreambuf_iterator& b);
```

Returns: `sbuf_ == b.sbuf`.

24.4.4.1.6 iterator_category [lib.ostreambuf.iterator.category]

```
output_iterator iterator_category (const ostreambuf_iterator&);
```

Returns: `output_iterator()`.

24.4.4.1.7 ostreambuf_iterator operator== [lib.ostreambuf.iterator.op==]

```
template<class charT, class traits = ios_char_traits<charT> >
  bool operator==(ostreambuf_iterator<charT,traits>& a,
          ostreambuf_iterator<charT,traits>& b);
```

Returns: *a*`.equal(`*b*`)`.

24.4.4.1.8 ostreambuf_iterator operator!= [lib.ostreambuf.iterator.op!=]

```
template<class charT, class traits = ios_char_traits<charT> >
  bool operator!=(ostreambuf_iterator<charT,traits>& a,
                  ostreambuf_iterator<charT,traits>& b);
```

Returns: `!a.equal(b)`.

25 Algorithms library [lib.algorithms]

1 This clause describes components that C++ programs may use to perform algorithmic operations on containers (23) and other sequences.

2 The following subclauses describe components for non-mutating sequence operation, mutating sequence operations, sorting and related operations, and algorithms from the ISO C library, as summarized in Table 60:

Table 60—Algorithms library summary

Subclause	Header(s)
25.1 Non-mutating sequence operations 25.2 Mutating sequence operations 25.3 Sorting and related operations	`<algorithm>`
25.4 C library algorithms	`<cstdlib>`

Header `<algorithm>` synopsis

```
namespace std {
// subclause 25.1, non-mutating sequence operations:
  template<class InputIterator, class Function>
    void for_each(InputIterator first, InputIterator last, Function f);
  template<class InputIterator, class T>
    InputIterator find(InputIterator first, InputIterator last,
                       const T& value);
  template<class InputIterator, class Predicate>
    InputIterator find_if(InputIterator first, InputIterator last,
                          Predicate pred);
  template<class ForwardIterator1, class ForwardIterator2>
    ForwardIterator1
      find_end(ForwardIterator1 first1, ForwardIterator1 last1,
               ForwardIterator2 first2, ForwardIterator2 last2);
  template<class ForwardIterator1, class ForwardIterator2,
           class BinaryPredicate>
    ForwardIterator1
      find_end(ForwardIterator1 first1, ForwardIterator1 last1,
               ForwardIterator2 first2, ForwardIterator2 last2,
```

```
template<class ForwardIterator1, class ForwardIterator2>
  ForwardIterator1
    find_first_of(ForwardIterator1 first1, ForwardIterator1 last1,
             ForwardIterator2 first2, ForwardIterator2 last2);
             BinaryPredicate pred);
template<class ForwardIterator1, class ForwardIterator2,
          class BinaryPredicate>
  ForwardIterator1
    find_first_of(ForwardIterator1 first1, ForwardIterator1 last1,
             ForwardIterator2 first2, ForwardIterator2 last2,
             BinaryPredicate pred);

template<class InputIterator>
  InputIterator adjacent_find(InputIterator first, InputIterator last);
template<class InputIterator, class BinaryPredicate>
  InputIterator adjacent_find(InputIterator first, InputIterator last,
                              BinaryPredicate pred);
template<class InputIterator, class T, class Size>
  void count(InputIterator first, InputIterator last, const T& value,
             Size& n);
template<class InputIterator, class Predicate, class Size>
  void count_if(InputIterator first, InputIterator last, Predicate pred,
                Size& n);
template<class InputIterator1, class InputIterator2>
  pair<InputIterator1, InputIterator2>
      mismatch(InputIterator1 first1, InputIterator1 last1,
               InputIterator2 first2);
template<class InputIterator1, class InputIterator2, class BinaryPredicate>
  pair<InputIterator1, InputIterator2>
      mismatch(InputIterator1 first1, InputIterator1 last1,
               InputIterator2 first2, BinaryPredicate pred);
template<class InputIterator1, class InputIterator2>
  bool equal(InputIterator1 first1, InputIterator1 last1,
             InputIterator2 first2);
template<class InputIterator1, class InputIterator2, class BinaryPredicate>
  bool equal(InputIterator1 first1, InputIterator1 last1,
             InputIterator2 first2, BinaryPredicate pred);

template<class ForwardIterator1, class ForwardIterator2>
  ForwardIterator1 search(ForwardIterator1 first1, ForwardIterator1 last1,
                          ForwardIterator2 first2, ForwardIterator2 last2);
template<class ForwardIterator1, class ForwardIterator2,
          class BinaryPredicate>
  ForwardIterator1 search(ForwardIterator1 first1, ForwardIterator1 last1,
                          ForwardIterator2 first2, ForwardIterator2 last2,
                          BinaryPredicate pred);
template<class ForwardIterator, class Size, class T>
  ForwardIterator search(ForwardIterator first, ForwardIterator last,
                          Size count, const T& value);
template<class ForwardIterator, class Size, class T, class BinaryPredicate>
  ForwardIterator1 search(ForwardIterator first, ForwardIterator last,
                          Size count, T value,
                          BinaryPredicate pred);
```

```
// subclause 25.2, mutating sequence operations:
// 25.2.1, copy:
  template<class InputIterator, class OutputIterator>
    OutputIterator copy(InputIterator first, InputIterator last,
                        OutputIterator result);
  template<class BidirectionalIterator1, class BidirectionalIterator2>
    BidirectionalIterator2
        copy_backward(BidirectionalIterator1 first,
                      BidirectionalIterator1 last,
                      BidirectionalIterator2 result);

// 25.2.2, swap:
  template<class T>
    void swap(T& a, T& b);
  template<class ForwardIterator1, class ForwardIterator2>
    ForwardIterator2 swap_ranges(ForwardIterator1 first1,
                                 ForwardIterator1 last1,
                                 ForwardIterator2 first2);

  template<class InputIterator, class OutputIterator, class UnaryOperation>
    OutputIterator transform(InputIterator first, InputIterator last,
                             OutputIterator result, UnaryOperation op);
  template<class InputIterator1, class InputIterator2, class OutputIterator,
           class BinaryOperation>
    OutputIterator transform(InputIterator1 first1, InputIterator1 last1,
                             InputIterator2 first2, OutputIterator result,
                             BinaryOperation binary_op);
  template<class ForwardIterator, class T>
    void replace(ForwardIterator first, ForwardIterator last,
                 const T& old_value, const T& new_value);
  template<class ForwardIterator, class Predicate, class T>
    void replace_if(ForwardIterator first, ForwardIterator last,
                    Predicate pred, const T& new_value);

  template<class InputIterator, class OutputIterator, class T>
    OutputIterator replace_copy(InputIterator first, InputIterator last,
                                OutputIterator result,
                                const T& old_value, const T& new_value);
  template<class Iterator, class OutputIterator, class Predicate, class T>
    OutputIterator replace_copy_if(Iterator first, Iterator last,
                                   OutputIterator result,
                                   Predicate pred, const T& new_value);
  template<class ForwardIterator, class T>
    void fill(ForwardIterator first, ForwardIterator last, const T& value);
  template<class OutputIterator, class Size, class T>
    void fill_n(OutputIterator first, Size n, const T& value);
  template<class ForwardIterator, class Generator>
    void generate(ForwardIterator first, ForwardIterator last, Generator gen);
  template<class OutputIterator, class Size, class Generator>
    void generate_n(OutputIterator first, Size n, Generator gen);
```

```
  template<class ForwardIterator, class T>
    ForwardIterator remove(ForwardIterator first, ForwardIterator last,
                           const T& value);
  template<class ForwardIterator, class Predicate>
    ForwardIterator remove_if(ForwardIterator first, ForwardIterator last,
                              Predicate pred);
  template<class InputIterator, class OutputIterator, class T>
    OutputIterator remove_copy(InputIterator first, InputIterator last,
                               OutputIterator result, const T& value);
  template<class InputIterator, class OutputIterator, class Predicate>
    OutputIterator remove_copy_if(InputIterator first, InputIterator last,
                                  OutputIterator result, Predicate pred);

  template<class ForwardIterator>
    ForwardIterator unique(ForwardIterator first, ForwardIterator last);
  template<class ForwardIterator, class BinaryPredicate>
    ForwardIterator unique(ForwardIterator first, ForwardIterator last,
                           BinaryPredicate pred);
  template<class InputIterator, class OutputIterator>
    OutputIterator unique_copy(InputIterator first, InputIterator last,
                               OutputIterator result);
  template<class InputIterator, class OutputIterator, class BinaryPredicate>
    OutputIterator unique_copy(InputIterator first, InputIterator last,
                               OutputIterator result, BinaryPredicate pred);
  template<class BidirectionalIterator>
    void reverse(BidirectionalIterator first, BidirectionalIterator last);
  template<class BidirectionalIterator, class OutputIterator>
    OutputIterator reverse_copy(BidirectionalIterator first,
                                BidirectionalIterator last,
                                OutputIterator result);
  template<class BidirectionalIterator>
    void rotate(BidirectionalIterator first, BidirectionalIterator middle,
                BidirectionalIterator last);
  template<class ForwardIterator, class OutputIterator>
    OutputIterator rotate_copy(ForwardIterator first, ForwardIterator middle,
                               ForwardIterator last, OutputIterator result);
  template<class RandomAccessIterator>
    void random_shuffle(RandomAccessIterator first, RandomAccessIterator last);
  template<class RandomAccessIterator, class RandomNumberGenerator>
    void random_shuffle(RandomAccessIterator first, RandomAccessIterator last,
                        RandomNumberGenerator& rand);

// 25.2.12, partitions:
  template<class BidirectionalIterator, class Predicate>
    BidirectionalIterator partition(BidirectionalIterator first,
                                    BidirectionalIterator last,
                                    Predicate pred);
  template<class BidirectionalIterator, class Predicate>
    ForwardIterator stable_partition(BidirectionalIterator first,
                                     BidirectionalIterator last,
                                     Predicate pred);
```

```
// subclause 25.3, sorting and related operations:
// 25.3.1, sorting:
  template<class RandomAccessIterator>
    void sort(RandomAccessIterator first, RandomAccessIterator last);
  template<class RandomAccessIterator, class Compare>
    void sort(RandomAccessIterator first, RandomAccessIterator last,
              Compare comp);
  template<class RandomAccessIterator>
    void stable_sort(RandomAccessIterator first, RandomAccessIterator last);
  template<class RandomAccessIterator, class Compare>
    void stable_sort(RandomAccessIterator first, RandomAccessIterator last,
                     Compare comp);

  template<class RandomAccessIterator>
    void partial_sort(RandomAccessIterator first, RandomAccessIterator middle,
                      RandomAccessIterator last);
  template<class RandomAccessIterator, class Compare>
    void partial_sort(RandomAccessIterator first, RandomAccessIterator middle,
                      RandomAccessIterator last, Compare comp);
  template<class InputIterator, class RandomAccessIterator>
    RandomAccessIterator
        partial_sort_copy(InputIterator first, InputIterator last,
                          RandomAccessIterator result_first,
                          RandomAccessIterator result_last);

  template<class InputIterator, class RandomAccessIterator, class Compare>
    RandomAccessIterator
        partial_sort_copy(InputIterator first, InputIterator last,
                          RandomAccessIterator result_first,
                          RandomAccessIterator result_last,
                          Compare comp);

  template<class RandomAccessIterator>
    void nth_element(RandomAccessIterator first, RandomAccessIterator nth,
                     RandomAccessIterator last);
  template<class RandomAccessIterator, class Compare>
    void nth_element(RandomAccessIterator first, RandomAccessIterator nth,
                     RandomAccessIterator last, Compare comp);

// 25.3.3, binary search:
  template<class ForwardIterator, class T>
    ForwardIterator lower_bound(ForwardIterator first, ForwardIterator last,
                                const T& value);
  template<class ForwardIterator, class T, class Compare>
    ForwardIterator lower_bound(ForwardIterator first, ForwardIterator last,
                                const T& value, Compare comp);
  template<class ForwardIterator, class T>
    ForwardIterator upper_bound(ForwardIterator first, ForwardIterator last,
                                const T& value);
  template<class ForwardIterator, class T, class Compare>
    ForwardIterator upper_bound(ForwardIterator first, ForwardIterator last,
                                const T& value, Compare comp);
  template<class ForwardIterator, class T>
    pair<ForwardIterator, ForwardIterator>
      equal_range(ForwardIterator first, ForwardIterator last, const T& value);
```

```
  template<class ForwardIterator, class T, class Compare>
    pair<ForwardIterator, ForwardIterator>
      equal_range(ForwardIterator first, ForwardIterator last, const T& value,
                  Compare comp);
  template<class ForwardIterator, class T>
    bool binary_search(ForwardIterator first, ForwardIterator last,
                       const T& value);
  template<class ForwardIterator, class T, class Compare>
    bool binary_search(ForwardIterator first, ForwardIterator last,
                       const T& value, Compare comp);

// 25.3.4, merge:
  template<class InputIterator1, class InputIterator2, class OutputIterator>
    OutputIterator merge(InputIterator1 first1, InputIterator1 last1,
                         InputIterator2 first2, InputIterator2 last2,
                         OutputIterator result);
  template<class InputIterator1, class InputIterator2, class OutputIterator,
            class Compare>
    OutputIterator merge(InputIterator1 first1, InputIterator1 last1,
                         InputIterator2 first2, InputIterator2 last2,
                         OutputIterator result, Compare comp);
  template<class BidirectionalIterator>
    void inplace_merge(BidirectionalIterator first,
                       BidirectionalIterator middle,
                       BidirectionalIterator last);
  template<class BidirectionalIterator, class Compare>
    void inplace_merge(BidirectionalIterator first,
                       BidirectionalIterator middle,
                       BidirectionalIterator last, Compare comp);

// 25.3.5, set operations:
  template<class InputIterator1, class InputIterator2>
    bool includes(InputIterator1 first1, InputIterator1 last1,
                  InputIterator2 first2, InputIterator2 last2);
  template<class InputIterator1, class InputIterator2, class Compare>
    bool includes(InputIterator1 first1, InputIterator1 last1,
                  InputIterator2 first2, InputIterator2 last2, Compare comp);
  template<class InputIterator1, class InputIterator2, class OutputIterator>
    OutputIterator set_union(InputIterator1 first1, InputIterator1 last1,
                             InputIterator2 first2, InputIterator2 last2,
                             OutputIterator result);
  template<class InputIterator1, class InputIterator2, class OutputIterator,
            class Compare>
    OutputIterator set_union(InputIterator1 first1, InputIterator1 last1,
                             InputIterator2 first2, InputIterator2 last2,
                             OutputIterator result, Compare comp);
```

```
  template<class InputIterator1, class InputIterator2, class OutputIterator>
    OutputIterator set_intersection(InputIterator1 first1, InputIterator1 last1,
                                    InputIterator2 first2, InputIterator2 last2,
                                    OutputIterator result);
  template<class InputIterator1, class InputIterator2, class OutputIterator,
           class Compare>
    OutputIterator set_intersection(InputIterator1 first1, InputIterator1 last1,
                                    InputIterator2 first2, InputIterator2 last2,
                                    OutputIterator result, Compare comp);
  template<class InputIterator1, class InputIterator2, class OutputIterator>
    OutputIterator set_difference(InputIterator1 first1, InputIterator1 last1,
                                  InputIterator2 first2, InputIterator2 last2,
                                  OutputIterator result);
  template<class InputIterator1, class InputIterator2, class OutputIterator,
           class Compare>
    OutputIterator set_difference(InputIterator1 first1, InputIterator1 last1,
                                  InputIterator2 first2, InputIterator2 last2,
                                  OutputIterator result, Compare comp);
  template<class InputIterator1, class InputIterator2, class OutputIterator>
    OutputIterator
      set_symmetric_difference(InputIterator1 first1, InputIterator1 last1,
                               InputIterator2 first2, InputIterator2 last2,
                               OutputIterator result);
  template<class InputIterator1, class InputIterator2, class OutputIterator,
           class Compare>
    OutputIterator
      set_symmetric_difference(InputIterator1 first1, InputIterator1 last1,
                               InputIterator2 first2, InputIterator2 last2,
                               OutputIterator result, Compare comp);

// 25.3.6, heap operations:
  template<class RandomAccessIterator>
    void push_heap(RandomAccessIterator first, RandomAccessIterator last);
  template<class RandomAccessIterator, class Compare>
    void push_heap(RandomAccessIterator first, RandomAccessIterator last,
                   Compare comp);
  template<class RandomAccessIterator>
    void pop_heap(RandomAccessIterator first, RandomAccessIterator last);
  template<class RandomAccessIterator, class Compare>
    void pop_heap(RandomAccessIterator first, RandomAccessIterator last,
                  Compare comp);
  template<class RandomAccessIterator>
    void make_heap(RandomAccessIterator first, RandomAccessIterator last);
  template<class RandomAccessIterator, class Compare>
    void make_heap(RandomAccessIterator first, RandomAccessIterator last,
                   Compare comp);
  template<class RandomAccessIterator>
    void sort_heap(RandomAccessIterator first, RandomAccessIterator last);
  template<class RandomAccessIterator, class Compare>
    void sort_heap(RandomAccessIterator first, RandomAccessIterator last,
                   Compare comp);
```

```
// 25.3.7, minimum and maximum:
  template<class T>
    T min(const T& a, const T& b);
  template<class T, class Compare>
    T min(const T& a, const T& b, Compare comp);
  template<class T>
    T max(const T& a, const T& b);
  template<class T, class Compare>
    T max(const T& a, const T& b, Compare comp);
  template<class InputIterator>
    InputIterator max_element(InputIterator first, InputIterator last);
  template<class InputIterator, class Compare>
    InputIterator max_element(InputIterator first, InputIterator last,
                              Compare comp);
  template<class InputIterator>
    InputIterator min_element(InputIterator first, InputIterator last);
  template<class InputIterator, class Compare>
    InputIterator min_element(InputIterator first, InputIterator last,
                              Compare comp);

  template<class InputIterator1, class InputIterator2>
    bool lexicographical_compare(InputIterator1 first1, InputIterator1 last1,
                                 InputIterator2 first2, InputIterator2 last2);
  template<class InputIterator1, class InputIterator2, class Compare>
    bool lexicographical_compare(InputIterator1 first1, InputIterator1 last1,
                                 InputIterator2 first2, InputIterator2 last2,
                                 Compare comp);

// 25.3.9, permutations
  template<class BidirectionalIterator>
    bool next_permutation(BidirectionalIterator first,
                          BidirectionalIterator last);
  template<class BidirectionalIterator, class Compare>
    bool next_permutation(BidirectionalIterator first,
                          BidirectionalIterator last,
                          Compare comp);
  template<class BidirectionalIterator>
    bool prev_permutation(BidirectionalIterator first,
                          BidirectionalIterator last);
  template<class BidirectionalIterator, class Compare>
    bool prev_permutation(BidirectionalIterator first,
                          BidirectionalIterator last,
                          Compare comp);
}
```

3 All of the algorithms are separated from the particular implementations of data structures and are parameterized by iterator types. Because of this, they can work with program-defined data structures, as long as these data structures have iterator types satisfying the assumptions on the algorithms.

4 Both in-place and copying versions are provided for certain algorithms.[154)] When such a version is provided for *`algorithm`* it is called *`algorithm_copy`*. Algorithms that take predicates end with the suffix `_if` (which follows the suffix `_copy`).

5 The `Predicate` class is used whenever an algorithm expects a function object that when applied to the result of dereferencing the corresponding iterator returns a value testable as `true`. In other words, if an algorithm takes `Predicate pred` as its argument and `first` as its iterator argument, it should work

154) The decision whether to include a copying version was usually based on complexity considerations. When the cost of doing the operation dominates the cost of copy, the copying version is not included. For example, `sort_copy` is not included since the cost of sorting is much more significant, and users might as well do `copy` followed by `sort`.

correctly in the construct `if (pred(*first)){...}`. The function object `pred` is assumed not to apply any non-constant function through the dereferenced iterator.

6 The `BinaryPredicate` class is used whenever an algorithm expects a function object that when applied to the result of dereferencing two corresponding iterators or to dereferencing an iterator and type `T` when `T`is part of the signature returns a value testable as `true`. In other words, if an algorithm takes `BinaryPredicate binary_pred` as its argument and `first1` and `first2`as its iterator arguments, it should work correctly in the construct `if (`*`pred`*`(*first, *first2)){...}`. `BinaryPredicate` always takes the first iterator type as its first argument, that is, in those cases when `T` *`value`* is part of the signature, it should work correctly in the context of `if (`*`pred`*`(*first,` *`value`*`)){...}`. It is expected that `binary_pred` will not apply any non-constant function through the dereferenced iterators.

7 In the description of the algorithms operators `+` and `-`are used for some of the iterator categories for which they do not have to be defined. In these cases the semantics of `a+n` is the same is that of

```
{ X tmp = a;
  advance(tmp, n);
  return tmp;
}
```

and that of `a-b` is the same as of

```
{ Distance n;
  distance(a, b, n);
  return n;
}
```

Box 101

For the following algorithms: `reverse`, `rotate`, `partition`,`random_shuffle`, `stable_partition`, `sort`, `stable_sort` and `inplace_merge` the iterator requirement can be relaxed to `ForwardIterator`. These algorithms could then be dispatched upon the iterator category tags to use the most efficient implementation for each iterator category. We have not included the relaxation at this stage since it is not yet fully implemented.

25.1 Non-mutating sequence operations [lib.alg.nonmutating]

25.1.1 For each [lib.alg.foreach]

```
template<class InputIterator, class Function>
  void for_each(InputIterator first, InputIterator last, Function f);
```

Effects: Applies `f` to the result of dereferencing every iterator in the range [*`first`*, *`last`*).
Requires: `f` shall not apply any non-constant function through the dereferenced iterator.
Complexity: Applies `f` exactly `last - first` times.
Notes: If `f` returns a *`result`*, the result is ignored.

25.1.2 Find [lib.alg.find]

```
template<class InputIterator, class T>
  InputIterator find(InputIterator first, InputIterator last,
             const T& value);

template<class InputIterator, class Predicate>
  InputIterator find_if(InputIterator first, InputIterator last,
                        Predicate pred);
```

Returns: The first iterator `i` in the range [*first*, *last*) for which the following corresponding conditions hold: `*i == `*value*, *pred*`(*i) == true`. Returns `last` if no such iterator is found.
Complexity: At most `last - first` applications of the corresponding predicate.

25.1.3 Find End [lib.alg.find.end]

```
template<class ForwardIterator1, class ForwardIterator2>
  ForwardIterator1
    find_end(ForwardIterator1 first1, ForwardIterator1 last1,
         ForwardIterator2 first2, ForwardIterator2 last2);

template<class ForwardIterator1, class ForwardIterator2,
      class BinaryPredicate>
  ForwardIterator1
    find_end(ForwardIterator1 first1, ForwardIterator1 last1,
             ForwardIterator2 first2, ForwardIterator2 last2,
             BinaryPredicate pred);
```

Effects: Finds a subsequence of equal values in a sequence.
Returns: The last iterator `i` in the range [*first1 + (last2-first2)*, *last1*) such that for any non-negative integer `n < (last2-first2)`, the following corresponding conditions hold: `*(i-n) == `*`*(last2-n)`*, *pred*`(*(i-n),*(last2-n)) == true`. Returns `last1` if no such iterator is found.
Complexity: At most `last1 - first1` applications of the corresponding predicate.

25.1.4 Find First [lib.alg.find.first.of]

```
template<class ForwardIterator1, class ForwardIterator2>
  ForwardIterator1
    find_first_of(ForwardIterator1 first1, ForwardIterator1 last1,
             ForwardIterator2 first2, ForwardIterator2 last2);

template<class ForwardIterator1, class ForwardIterator2,
      class BinaryPredicate>
  ForwardIterator1
    find_first_of(ForwardIterator1 first1, ForwardIterator1 last1,
             ForwardIterator2 first2, ForwardIterator2 last2,
             BinaryPredicate pred);
```

Effects: Finds a subsequence of equal values in a sequence.
Returns: The first iterator `i` in the range [*first1*, *last1-(last2-first2)*) such that for any non-negative integer `n < (last2-first2)`, the following corresponding conditions hold: `*i == `*`*(first2+n)`*, *pred*`(i,first2+n) == true`. Returns `last1` if no such iterator is found.
Complexity: Exactly `find_first_of(first1,last1,first2+n)` applications of the corresponding predicate.

25.1.5 Adjacent find [lib.alg.adjacent.find]

```
template<class InputIterator>
  InputIterator adjacent_find(InputIterator first, InputIterator last);

template<class InputIterator, class BinaryPredicate>
  InputIterator adjacent_find(InputIterator first, InputIterator last,
                              BinaryPredicate pred);
```

Returns: The first iterator `i` such that both `i` and `i + 1` are in the range `[`*`first`*`, `*`last`*`)` for which the following corresponding conditions hold: `*i == *(i + 1)`, *`pred`*`(*i, *(i + 1)) == true`. Returns `last` if no such iterator is found.
Complexity: At most `max((last - first) - 1, 0)` applications of the corresponding predicate.

25.1.6 Count [lib.alg.count]

```
template<class InputIterator, class T, class Size>
  void count(InputIterator first, InputIterator last, const T& value,
             Size& n);

template<class InputIterator, class Predicate, class Size>
  void count_if(InputIterator first, InputIterator last, Predicate pred,
                Size& n);
```

Effects: Adds to `n` the number of iterators `i` in the range `[`*`first`*`, `*`last`*`)` for which the following corresponding conditions hold: `*i == `*`value`*, `pred(*i) == true`.
Complexity: Exactly `last - first` applications of the corresponding predicate.
Notes: `count` must store the result into a reference argument instead of returning the result because the size type cannot be deduced from built-in iterator types such as `int*`.

25.1.7 Mismatch [lib.mismatch]

```
template<class InputIterator1, class InputIterator2>
  pair<InputIterator1, InputIterator2>
      mismatch(InputIterator1 first1, InputIterator1 last1,
               InputIterator2 first2);

template<class InputIterator1, class InputIterator2,
         class BinaryPredicate>
  pair<InputIterator1, InputIterator2>
      mismatch(InputIterator1 first1, InputIterator1 last1,
               InputIterator2 first2, BinaryPredicate pred);
```

Returns: A pair of iterators `i` and `j` such that `j == first2 + (i - first1)` and `i` is the first iterator in the range `[`*`first1`*`, last1)` for which the following corresponding conditions hold:

`!(*i == *(first2 + (i - first1)))`, *`pred`*`(*i, *(first2 + (i - first1))) == false`

Returns the pair `last1` and `first2 + (last1 - first1)` if such an iterator `i` is not found.
Complexity: At most `last1 - first1` applications of the corresponding predicate.

25.1.8 Equal [lib.alg.equal]

```
template<class InputIterator1, class InputIterator2>
  bool equal(InputIterator1 first1, InputIterator1 last1,
             InputIterator2 first2);

template<class InputIterator1, class InputIterator2,
         class BinaryPredicate>
  bool equal(InputIterator1 first1, InputIterator1 last1,
             InputIterator2 first2, BinaryPredicate pred);
```

Returns: `true` if for every iterator `i` in the range `[`*`first1`*`, last1)` the following corresponding conditions hold: `*i == *(first2 + (i - first1))`, *`pred`*`(*i, *(first2 + (i - first1))) == true`. Otherwise, returns `false`.

Complexity: At most `last1 - first1` applications of the corresponding predicate.

25.1.9 Search [lib.alg.search]

```
template<class ForwardIterator1, class ForwardIterator2>
  ForwardIterator1
    search(ForwardIterator1 first1, ForwardIterator1 last1,
       ForwardIterator2 first2, ForwardIterator2 last2);

template<class ForwardIterator1, class ForwardIterator2,
      class BinaryPredicate>
  ForwardIterator1
    search(ForwardIterator1 first1, ForwardIterator1 last1,
       ForwardIterator2 first2, ForwardIterator2 last2,
       BinaryPredicate pred);
```

Effects: Finds a subsequence of equal values in a sequence.

Returns: The first iterator `i` in the range `[first1, last1 - (last2 - first2))` such that for any non-negative integer `n` less than `last2 - first2` the following corresponding conditions hold: `*(i + n) == *(first2 + n), pred(*(i + n), *(first2 + n)) == true`. Returns `last1` if no such iterator is found.[155)]

```
template<class ForwardIterator, class Size, class T>
  ForwardIterator
    search(ForwardIterator first, ForwardIterator last,
       Size count, const T& value);

template<class ForwardIterator, class Size, class T,
      class BinaryPredicate>
  ForwardIterator1
    search(ForwardIterator first, ForwardIterator last,
       Size count, T value,
           BinaryPredicate pred);
```

Effects: Finds a subsequence of equal values in a sequence.

Returns: The first iterator `i` in the range `[first, last - count)` such that for any non-negative integer `n` less than `count` the following corresponding conditions hold: `*(i + n) == value, pred(*(i + n),value) == true`. Returns `last` if no such iterator is found.

25.2 Mutating sequence operations [lib.alg.mutating.operations]

25.2.1 Copy [lib.alg.copy]

---

155) The Knuth-Morris-Pratt algorithm is not used here. While the KMP algorithm guarantees linear time, it tends to be slower in most practical cases than the naive algorithm with worst-case quadratic behavior. The worst case is extremely unlikely. We expect that most implementations will provide a specialization:

```
char* search(char* first1, char* last1,
             char* first2,  char* last2);
```

1 that will use a variation of the Boyer-Moore algorithm for fast string searching.

25.2.1.1 copy [lib.copy]

```
template<class InputIterator, class OutputIterator>
  OutputIterator copy(InputIterator first, InputIterator last,
                      OutputIterator result);
```

Effects: Copies elements. For each non-negative integer `n < (last - first)`, performs `*(result + n) = *(first + n)`.
Returns: `result + (last - first)`.
Requires: `result` shall not be in the range `[first, last)`.
Complexity: Exactly `last - first` assignments.

25.2.1.2 copy_backward [lib.copy.backward]

```
template<class BidirectionalIterator1, class BidirectionalIterator2>
  BidirectionalIterator2
    copy_backward(BidirectionalIterator1 first,
          BidirectionalIterator1 last,
          BidirectionalIterator2 result);
```

Effects: Copies elements in the `range [first, last)` into the range `[result - (last - first), result)` starting from `last - 1` and proceeding to [156] `first`. For each positive integer `n <= (last - first)`, Performs `*(result - n) = *(last - n)`.
Requires: `result` shall not be in the range `[first, last)`.
Returns: `result - (last - first)`.
Complexity: Exactly `last - first` assignments.

25.2.2 Swap [lib.alg.swap]

25.2.2.1 swap [lib.swap]

```
template<class T>
  void swap(T& a, T& b);
```

Effects: Exchanges values stored in two locations.

25.2.2.2 swap_ranges [lib.swap.ranges]

```
template<class ForwardIterator1, class ForwardIterator2>
  ForwardIterator2
    swap_ranges(ForwardIterator1 first1,
        ForwardIterator1 last1,
        ForwardIterator2 first2);
```

Effects: For each non-negative integer `n < (last1 - first1)` performs: `swap(*(first1 + n), *(first2 + n))`.
Requires: The two ranges `[first1, last1)` and `[first2, first2 + (last1 - first1))` shall not overlap.
Returns: `first2 + (last1 - first1)`.
Complexity: Exactly `last1 - first1` swaps.

---

[156] `copy_backward` (25.2.1.2) should be used instead of copy when `last` is in the range `[result - (last - first), result)`.

25.2.3 Transform [lib.alg.transform]

```
template<class InputIterator, class OutputIterator,
     class UnaryOperation>
  OutputIterator
    transform(InputIterator first, InputIterator last,
          OutputIterator result, UnaryOperation op);

template<class InputIterator1, class InputIterator2,
     class OutputIterator, class BinaryOperation>
  OutputIterator
    transform(InputIterator1 first1, InputIterator1 last1,
          InputIterator2 first2, OutputIterator result,
          BinaryOperation binary_op);
```

Effects: Assigns through every iterator `i` in the range `[result, result + (last1 - first1))` a new corresponding value equal to `op(*(first1 + (i - result))` or `binary_op(*(first1 + (i - result), *(first2 + (i - result)))`.
Requires: `op` and `binary_op` shall not have any side effects.
Returns: `result + (last1 - first1)`.
Complexity: Exactly `last1 - first1` applications of `op` or `binary_op`.
Notes: `result` may be equal to `first` in case of unary transform, or to `first1` or `first2` in case of binary transform.

25.2.4 Replace [lib.alg.replace]

25.2.4.1 `replace` [lib.replace]

```
template<class ForwardIterator, class T>
  void replace(ForwardIterator first, ForwardIterator last,
           const T& old_value, const T& new_value);

template<class ForwardIterator, class Predicate, class T>
  void replace_if(ForwardIterator first, ForwardIterator last,
          Predicate pred, const T& new_value);
```

Effects: Substitutes elements referred by the iterator `i` in the range `[first, last)` with `new_value`, when the following corresponding conditions hold: `*i == old_value`, `pred(*i) == true`.
Complexity: Exactly `last - first` applications of the corresponding predicate.

25.2.4.2 `replace_copy` [lib.replace.copy]

```
template<class InputIterator, class OutputIterator, class T>
  OutputIterator
    replace_copy(InputIterator first, InputIterator last,
         OutputIterator result,
         const T& old_value, const T& new_value);

template<class Iterator, class OutputIterator, class Predicate, class T>
  OutputIterator
    replace_copy_if(Iterator first, Iterator last,
            OutputIterator result,
            Predicate pred, const T& new_value);
```

Effects: Assigns to every iterator `i` in the range [*result*, `result + (last - first)`) either `new_value` or `*(first + (i - result))` depending on whether the following corresponding conditions hold:

```
*(first + (i - result)) == old_value,
pred(*(first + (i - result))) == true
```

Returns: `result + (last - first)`.
Complexity: Exactly `last - first` applications of the corresponding predicate.

25.2.5 Fill [lib.alg.fill]

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

template<class OutputIterator, class Size, class T>
  void fill_n(OutputIterator first, Size n, const T& value);
```

Effects: Assigns value through all the iterators in the range [*first*, *last*)or [*first*, first + n).
Complexity: Exactly `last - first` (or `n`) assignments.

25.2.6 Generate [lib.alg.generate]

```
template<class ForwardIterator, class Generator>
  void generate(ForwardIterator first, ForwardIterator last,
        Generator gen);

template<class OutputIterator, class Size, class Generator>
  void generate_n(OutputIterator first, Size n, Generator gen);
```

Effects: Invokes the function object gen and assigns the return value of *gen* though all the iterators in the range [*first*, *last*) or [*first*, first + n).
Requires: *gen* takes no arguments.
Complexity: Exactly `last - first` (or `n`) invocations of *gen* and assignments.

25.2.7 Remove [lib.alg.remove]

25.2.7.1 `remove` [lib.remove]

```
template<class ForwardIterator, class T>
  ForwardIterator remove(ForwardIterator first, ForwardIterator last,
              const T& value);

template<class ForwardIterator, class Predicate>
  ForwardIterator remove_if(ForwardIterator first, ForwardIterator last,
                Predicate pred);
```

Effects: Eliminates all the elements referred to by iterator `i` in the range [*first*, *last*) for which the following corresponding conditions hold: `*i == value`, `pred(*i) == true`.
Returns: The end of the resulting range.
Notes: Stable: the relative order of the elements that are not removed is the same as their relative order in the original range.
Complexity: Exactly `last - first` applications of the corresponding predicate.

25.2.7.2 remove_copy [lib.remove.copy]

```
template<class InputIterator, class OutputIterator, class T>
  OutputIterator
    remove_copy(InputIterator first, InputIterator last,
        OutputIterator result, const T& value);

template<class InputIterator, class OutputIterator, class Predicate>
  OutputIterator
    remove_copy_if(InputIterator first, InputIterator last,
                   OutputIterator result, Predicate pred);
```

Effects: Copies all the elements referred to by the iterator `i` in the range `[first, last)` for which the following corresponding conditions do not hold: `*i == value`, `pred(*i) == true`.
Returns: The end of the resulting range.
Complexity: Exactly `last - first` applications of the corresponding predicate.
Notes: Stable: the relative order of the elements in the resulting range is the same as their relative order in the original range.

25.2.8 Unique [lib.alg.unique]

25.2.8.1 unique [lib.unique]

```
template<class ForwardIterator>
  ForwardIterator unique(ForwardIterator first, ForwardIterator last);

template<class ForwardIterator, class BinaryPredicate>
  ForwardIterator unique(ForwardIterator first, ForwardIterator last,
                         BinaryPredicate pred);
```

Effects: Eliminates all but the first element from every consecutive group of equal elements referred to by the iterator `i` in the range `[first, last)` for which the following corresponding conditions hold: `*i == *(i - 1)` or `pred(*i, *(i - 1)) == true`
Returns: The end of the resulting range.
Complexity: Exactly `(last - first) - 1` applications of the corresponding predicate.

25.2.8.2 unique_copy [lib.unique.copy]

```
template<class InputIterator, class OutputIterator>
  OutputIterator
    unique_copy(InputIterator first, InputIterator last,
        OutputIterator result);

template<class InputIterator, class OutputIterator,
      class BinaryPredicate>
  OutputIterator
    unique_copy(InputIterator first, InputIterator last,
        OutputIterator result, BinaryPredicate pred);
```

Effects: Copies only the first element from every consecutive group of equal elements referred to by the iterator `i` in the range `[first, last)` for which the following corresponding conditions hold: `*i == *(i - 1)` or `pred(*i, *(i - 1)) == true`
Returns: The end of the resulting range.
Complexity: Exactly `last - first` applications of the corresponding predicate.

25.2.9 Reverse [lib.alg.reverse]

25.2.9.1 `reverse` [lib.reverse]

```
template<class BidirectionalIterator>
  void reverse(BidirectionalIterator first, BidirectionalIterator last);
```

Effects: For each non-negative integer `i <= (last - first)/2`, applies `swap` to all pairs of iterators `first + i, (last - i) - 1`.
Complexity: Exactly `(last - first)/2` swaps.

25.2.9.2 `reverse_copy` [lib.reverse.copy]

```
template<class BidirectionalIterator, class OutputIterator>
  OutputIterator
    reverse_copy(BidirectionalIterator first,
         BidirectionalIterator last, OutputIterator result);
```

Effects: Copies the range `[first, last)` to the range `[result, result + (last - first))` such that for any non-negative integer `i < (last - first)` the following assignment takes place:

```
*(result + (last - first) - i) = *(first + i)
```

Requires: The ranges `[first, last)` and `[result, result + (last - first))` shall not overlap.
Returns: `result + (last - first)`.
Complexity: Exactly `last - first` assignments.

25.2.10 Rotate [lib.alg.rotate]

25.2.10.1 `rotate` [lib.rotate]

```
template<class BidirectionalIterator>
  void rotate(BidirectionalIterator first, BidirectionalIterator middle,
              BidirectionalIterator last);
```

1 For each non-negative integer `rotatei < (last - first)`, places the element from the position `first + i` into position `first + (i + (middle - first)) % (last - first)`.
Complexity: At most `last - first` swaps.

25.2.10.2 `rotate_copy` [lib.rotate.copy]

```
template<class ForwardIterator, class OutputIterator>
  OutputIterator
    rotate_copy(ForwardIterator first, ForwardIterator middle,
         ForwardIterator last, OutputIterator result);
```

Effects: Copies the range `[first, last)` to the range `[result, result + (last - first))` such that for each non-negative integer `i < (last - first)` the following assignment takes place:

```
*(first + i) =  *(result + (i + (middle - first)) % (last - first))
```

Returns: `result + (last - first)`.
Requires The ranges `[first, last)` and `[result, result + (last - first))` shall not overlap.

Complexity: Exactly `last - first` assignments.

25.2.11 Random shuffle [lib.alg.random.shuffle]

```
template<class RandomAccessIterator>
  void random_shuffle(RandomAccessIterator first,
              RandomAccessIterator last);

template<class RandomAccessIterator, class RandomNumberGenerator>
  void random_shuffle(RandomAccessIterator first,
              RandomAccessIterator last,
              RandomNumberGenerator& rand);
```

Effects: Shuffles the elements in the range `[first, last)` with uniform distribution.
Complexity: Exactly `(last - first) - 1` swaps.
Notes: `random_shuffle` can take a particular random number generating function object `rand` such that `rand` returns a randomly chosen `double` in the interval `[0, 1)`.

25.2.12 Partitions [lib.alg.partitions]

25.2.12.1 `partition` [lib.partition]

```
template<class BidirectionalIterator, class Predicate>
  BidirectionalIterator
    partition(BidirectionalIterator first,
          BidirectionalIterator last, Predicate pred);
```

Effects: Places all the elements in the range `[first, last)` that satisfy `pred` before all the elements that do not satisfy it.
Returns: An iterator `i` such that for any iterator `j` in the range `[first, i), pred(*j) == true`, and for any iterator `k` in the range `[i, last), pred(*j) == false`.
Complexity: At most `(last - first)/2` swaps. Exactly `last - first` applications of the predicate is done.

25.2.12.2 `stable_partition` [lib.stable.partition]

```
template<class BidirectionalIterator, class Predicate>
  ForwardIterator
    stable_partition(BidirectionalIterator first,
             BidirectionalIterator last, Predicate pred);
```

Effects: Places all the elements in the range `[first, last)` that satisfy pred before all the elements that do not satisfy it.
Returns: An iterator `i` such that for any iterator `j` in the range `[first, i), pred(*j) == true`, and for any iterator `k` in the range `[i, last), pred(*j) == false`. The relative order of the elements in both groups is preserved.
Complexity: At most `(last - first) * log(last - first)` swaps, but only linear number of swaps if there is enough extra memory. Exactly `last - first` applications of the predicate.

25.3 Sorting and related operations [lib.alg.sorting]

1 All the operations in this section have two versions: one that takes a function object of type `Compare` and one that uses an `operator<`.

2 `Compare` is used as a function object which returns `true` if the first argument is less than the second, and `false` otherwise. `Compare comp` is used throughout for algorithms assuming an ordering relation. It is assumed that `comp` will not apply any non-constant function through the dereferenced iterator. For all

algorithms that take `Compare`, there is a version that uses `operator<` instead. That is, `comp(*i, *j) == true` defaults to `*i < *j == true`. For the algorithms to work correctly, `comp` has to induce a total ordering on the values.

3 A sequence is *sorted with respect to a comparator* `comp` if for any iterator `i` pointing to the sequence and any non-negative integer `n` such that `i + n` is a valid iterator pointing to an element of the sequence, `comp(*(i + n), *i) == false`.

4 In the descriptions of the functions that deal with ordering relationships we frequently use a notion of equality to describe concepts such as stability. The equality to which we refer is not necessarily an `operator==`, but an equality relation induced by the total ordering. That is, two element `a` and `b` are considered equal if and only if `!(a < b) && !(b < a)`.

25.3.1 Sorting [lib.alg.sort]

25.3.1.1 `sort` [lib.sort]

```
template<class RandomAccessIterator>
  void sort(RandomAccessIterator first, RandomAccessIterator last);

template<class RandomAccessIterator, class Compare>
  void sort(RandomAccessIterator first, RandomAccessIterator last,
            Compare comp);
```

Effects: Sorts the elements in the range `[first, last)`.
Complexity: Approximately `NlogN` (where `N == last - first`) comparisons on the average.[157)]

25.3.1.2 `stable_sort` [lib.stable.sort]

```
template<class RandomAccessIterator>
  void stable_sort(RandomAccessIterator first,
            RandomAccessIterator last);

template<class RandomAccessIterator, class Compare>
  void stable_sort(RandomAccessIterator first,
            RandomAccessIterator last,
            Compare comp);
```

Effects: Sorts the elements in the range `[first, last)`.
Complexity: It does at most `Nlog2N` (where `N == last - first`) comparisons; if enough extra memory is available, it is `NlogN`.
Notes: Stable: the relative order of the equal elements is preserved.

25.3.1.3 `partial_sort` [lib.partial.sort]

[157)] If the worst case behavior is important `stable_sort`(_lib.stable.sort_) or `partial_sort` (25.3.1.3) should be used.

```
template<class RandomAccessIterator>
  void partial_sort(RandomAccessIterator first,
            RandomAccessIterator middle,
            RandomAccessIterator last);

template<class RandomAccessIterator, class Compare>
  void partial_sort(RandomAccessIterator first,
            RandomAccessIterator middle,
            RandomAccessIterator last, Compare comp);
```

Effects: Places the first `middle - first` sorted elements from the range `[first, last)` into the range `[first, middle)`. The rest of the elements in the range `[middle, last)` are placed in an undefined order.
Complexity: It takes approximately `(last - first) * log(middle - first)` comparisons.

25.3.1.4 partial_sort_copy [lib.partial.sort.copy]

```
template<class InputIterator, class RandomAccessIterator>
  RandomAccessIterator
    partial_sort_copy(InputIterator first, InputIterator last,
              RandomAccessIterator result_first,
              RandomAccessIterator result_last);

template<class InputIterator, class RandomAccessIterator,
      class Compare>
  RandomAccessIterator
    partial_sort_copy(InputIterator first, InputIterator last,
              RandomAccessIterator result_first,
              RandomAccessIterator result_last,
                       Compare comp);
```

Effects: Places the first `min(last - first, result_last - result_first)` sorted elements into the range `[result_first, result_first + min(last - first, result_last - result_first))`.
Returns: The smaller of: `result_last` or `result_first + (last - first)`
Complexity: Approximately `(last - first) * log(min(last - first, result_last - result_first))` comparisons.

25.3.2 Nth element [lib.alg.nth.element]

```
template<class RandomAccessIterator>
  void nth_element(RandomAccessIterator first,
           RandomAccessIterator nth,
                  RandomAccessIterator last);

template<class RandomAccessIterator, class Compare>
  void nth_element(RandomAccessIterator first,
           RandomAccessIterator nth,
                  RandomAccessIterator last, Compare comp);
```

1 After `nth_element` the element in the position pointed to by `nth` is the element that would be in that position if the whole range were sorted. Also for any iterator `i` in the range `[first, nth)` and any iterator `j` in the range `[nth, last)` it holds that: `!(*i > *j)` or `comp(*i, *j) == false`.

Complexity: Linear on average.

25.3.3 Binary search [lib.alg.binary.search]

1 All of the algorithms in this section are versions of binary search. They work on non-random access iterators minimizing the number of comparisons, which will be logarithmic for all types of iterators. They are especially appropriate for random access iterators, since these algorithms do a logarithmic number of steps through the data structure. For non-random access iterators they execute a linear number of steps.

25.3.3.1 `lower_bound` [lib.lower.bound]

```
template<class ForwardIterator, class T>
  ForwardIterator
    lower_bound(ForwardIterator first, ForwardIterator last,
        const T& value);

template<class ForwardIterator, class T, class Compare>
  ForwardIterator
    lower_bound(ForwardIterator first, ForwardIterator last,
        const T& value, Compare comp);
```

Effects: Finds the first position into which value can be inserted without violating the ordering.
Returns: The furthermost iterator `i` in the range `[first, last)` such that for any iterator `j` in the range `[first, i)` the following corresponding conditions hold: `*j < value` or `comp(*j, value) == true`
Complexity: At most `log(last - first) + 1` comparisons.

25.3.3.2 `upper_bound` [lib.upper.bound]

```
template<class ForwardIterator, class T>
  ForwardIterator
    upper_bound(ForwardIterator first, ForwardIterator last,
        const T& value);

template<class ForwardIterator, class T, class Compare>
  ForwardIterator
    upper_bound(ForwardIterator first, ForwardIterator last,
        const T& value, Compare comp);
```

Effects: Finds the furthermost position into which value can be inserted without violating the ordering.
Returns: The furthermost iterator `i` in the range `[first, last)` such that for any iterator `j` in the range `[first, i)` the following corresponding conditions hold: `!(value < *j)` or `comp(value, *j) == false`
Complexity: At most `log(last - first) + 1` comparisons.

25.3.3.3 `equal_range` [lib.equal.range]

```
template<class ForwardIterator, class T>
  pair<ForwardIterator, ForwardIterator>
    equal_range(ForwardIterator first,
        ForwardIterator last, const T& value);

template<class ForwardIterator, class T, class Compare>
  pair<ForwardIterator, ForwardIterator>
    equal_range(ForwardIterator first,
        ForwardIterator last, const T& value,
               Compare comp);
```

Effects: Finds the largest subrange `[i, j)` such that the value can be inserted at any iterator `k` in it. `k` satisfies the corresponding conditions: `!(*k < value) && !(value < *k)` or `comp(*k, value) == false && comp(value, *k) == false`.
Complexity: At most `2 * log(last - first)` comparisons.

25.3.3.4 `binary_search` [lib.binary.search]

```
template<class ForwardIterator, class T>
  bool binary_search(ForwardIterator first, ForwardIterator last,
            const T& value);

template<class ForwardIterator, class T, class Compare>
  bool binary_search(ForwardIterator first, ForwardIterator last,
            const T& value, Compare comp);
```

Returns: `true` if there is an iterator `i` in the range `[first last)` that satisfies the corresponding conditions: `!(*i < value) && !(value < *i)` or `comp(*i, value) == false && comp(value, *i) == false`.
Complexity: At most `log(last - first) + 1` comparisons.

25.3.4 Merge [lib.alg.merge]

25.3.4.1 `merge` [lib.merge]

```
template<class InputIterator1, class InputIterator2,
      class OutputIterator>
  OutputIterator
    merge(InputIterator1 first1, InputIterator1 last1,
      InputIterator2 first2, InputIterator2 last2,
      OutputIterator result);

template<class InputIterator1, class InputIterator2,
      class OutputIterator, class Compare>
  OutputIterator
    merge(InputIterator1 first1, InputIterator1 last1,
      InputIterator2 first2, InputIterator2 last2,
      OutputIterator result, Compare comp);
```

Effects: Merges two sorted ranges `[first1, last1)` and `[first2, last2)` into the range `[result, result + (last1 - first1) + (last2 - first2))`.

1 The resulting range shall not overlap with either of the original ranges.
Returns: `result + (last1 - first1) + (last2 - first2)`.

Complexity: At most `(last1 - first1) + (last2 - first2) - 1` comparisons.
Notes: Stable: for equal elements in the two ranges, the elements from the first range always precede the elements from the second.

25.3.4.2 `inplace_merge` [lib.inplace.merge]

```
template<class BidirectionalIterator>
  void inplace_merge(BidirectionalIterator first,
             BidirectionalIterator middle,
             BidirectionalIterator last);

template<class BidirectionalIterator, class Compare>
  void inplace_merge(BidirectionalIterator first,
             BidirectionalIterator middle,
             BidirectionalIterator last, Compare comp);
```

Effects: Merges two sorted consecutive ranges `[first, middle)` and `[middle, last)`, putting the result of the merge into the range `[first, last)`.
Complexity: At most `last - first` comparisons. If no additional memory is available, the number of assignments can be equal to `NlogN` where `N` is equal to `last - first`.
Notes: Stable: for equal elements in the two ranges, the elements from the first range always precede the elements from the second.

25.3.5 Set operations on sorted structures [lib.alg.set.operations]

1 This section defines all the basic set operations on sorted structures. They even work with `multiset`s containing multiple copies of equal elements. The semantics of the set operations is generalized to multisets in a standard way by defining union to contain the maximum number of occurrences of every element, intersection to contain the minimum, and so on.

25.3.5.1 `includes` [lib.includes]

```
template<class InputIterator1, class InputIterator2>
  bool includes(InputIterator1 first1, InputIterator1 last1,
                InputIterator2 first2, InputIterator2 last2);

template<class InputIterator1, class InputIterator2, class Compare>
  bool includes(InputIterator1 first1, InputIterator1 last1,
        InputIterator2 first2, InputIterator2 last2,
        Compare comp);
```

Returns: `true` if every element in the range `[first2, last2)` is contained in the range `[first1, last1)`. Returns `false` otherwise.
Complexity: At most `((last1 - first1) + (last2 - first2)) * 2 - 1` comparisons.

25.3.5.2 `set_union` [lib.set.union]

```
template<class InputIterator1, class InputIterator2,
      class OutputIterator>
  OutputIterator
    set_union(InputIterator1 first1, InputIterator1 last1,
          InputIterator2 first2, InputIterator2 last2,
          OutputIterator result);

template<class InputIterator1, class InputIterator2,
      class OutputIterator, class Compare>
  OutputIterator
    set_union(InputIterator1 first1, InputIterator1 last1,
          InputIterator2 first2, InputIterator2 last2,
          OutputIterator result, Compare comp);
```

Effects: Constructs a sorted union of the elements from the two ranges.
Requires: The resulting range shall not overlap with either of the original ranges.
Returns: The end of the constructed range.
Complexity: At most `((last1 - first1) + (last2 - first2)) * 2 - 1` comparisons.
Notes: Stable: if an element is present in both ranges, the one from the first range is copied.

25.3.5.3 `set_intersection` [lib.set.intersection]

```
template<class InputIterator1, class InputIterator2,
      class OutputIterator>
  OutputIterator
    set_intersection(InputIterator1 first1, InputIterator1 last1,
            InputIterator2 first2, InputIterator2 last2,
            OutputIterator result);

template<class InputIterator1, class InputIterator2,
      class OutputIterator, class Compare>
  OutputIterator
    set_intersection(InputIterator1 first1, InputIterator1 last1,
            InputIterator2 first2, InputIterator2 last2,
            OutputIterator result, Compare comp);
```

Effects: Constructs a sorted intersection of the elements from the two ranges.
Requires: The resulting range shall not overlap with either of the original ranges.
Returns: The end of the constructed range.
Complexity: At most `((last1 - first1) + (last2 - first2)) * 2 - 1` comparisons.
Notes: Stable, that is, if an element is present in both ranges, the one from the first range is copied.

25.3.5.4 `set_difference` [lib.set.difference]

```
template<class InputIterator1, class InputIterator2,
     class OutputIterator>
  OutputIterator
    set_difference(InputIterator1 first1, InputIterator1 last1,
           InputIterator2 first2, InputIterator2 last2,
           OutputIterator result);

template<class InputIterator1, class InputIterator2,
     class OutputIterator, class Compare>
  OutputIterator
    set_difference(InputIterator1 first1, InputIterator1 last1,
           InputIterator2 first2, InputIterator2 last2,
           OutputIterator result, Compare comp);
```

Effects: Constructs a sorted difference of the elements from the two ranges.
Requires: The resulting range shall not overlap with either of the original ranges.
Returns: The end of the constructed range.
Complexity: At most `((last1 - first1) + (last2 - first2)) * 2 - 1` comparisons.

25.3.5.5 set_symmetric_difference [lib.set.symmetric.difference]

```
template<class InputIterator1, class InputIterator2,
     class OutputIterator>
  OutputIterator
    set_symmetric_difference(InputIterator1 first1, InputIterator1 last1,
                 InputIterator2 first2, InputIterator2 last2,
                 OutputIterator result);

template<class InputIterator1, class InputIterator2,
     class OutputIterator, class Compare>
  OutputIterator
    set_symmetric_difference(InputIterator1 first1, InputIterator1 last1,
                 InputIterator2 first2, InputIterator2 last2,
                 OutputIterator result, Compare comp);
```

Effects: Constructs a sorted symmetric difference of the elements from the two ranges.
Requires: The resulting range shall not overlap with either of the original ranges.
Returns: The end of the constructed range.
Complexity: At most `((last1 - first1) + (last2 - first2)) * 2 - 1` comparisons.

25.3.6 Heap operations [lib.alg.heap.operations]

1 A heap is a particular organization of elements in a range between two random access iterators `[a, b)`. Its two key properties are: (1) `*a` is the largest element in the range and (2) `*a` may be removed by `pop_heap`, or a new element added by `push_heap`, in `O(logN)` time. These properties make heaps useful as priority queues. `make_heap` converts a range into a heap and `sort_heap` turns a heap into a sorted sequence.

25.3.6.1 push_heap [lib.push.heap]

```
template<class RandomAccessIterator>
  void push_heap(RandomAccessIterator first, RandomAccessIterator last);

template<class RandomAccessIterator, class Compare>
  void push_heap(RandomAccessIterator first, RandomAccessIterator last,
                 Compare comp);
```

Requires: The range [*first*, last - 1) shall be a valid heap.
Effects: Places the value in the location last - 1 into the resulting heap [*first*, *last*).
Complexity: At most log(last - first) comparisons.

25.3.6.2 pop_heap [lib.pop.heap]

```
template<class RandomAccessIterator>
  void pop_heap(RandomAccessIterator first, RandomAccessIterator last);

template<class RandomAccessIterator, class Compare>
  void pop_heap(RandomAccessIterator first, RandomAccessIterator last,
                Compare comp);
```

Requires: The range [*first*, *last*) shall be a valid heap.
Effects: Swaps the value in the location first with the value in the location last - 1 and makes [*first*, last - 1) into a heap.
Complexity: At most 2 * log(last - first) comparisons.

25.3.6.3 make_heap [lib.make.heap]

```
template<class RandomAccessIterator>
  void make_heap(RandomAccessIterator first, RandomAccessIterator last);

template<class RandomAccessIterator, class Compare>
  void make_heap(RandomAccessIterator first, RandomAccessIterator last,
                 Compare comp);
```

Effects: Constructs a heap out of the range [*first*, *last*).
Complexity: At most 3*(last - first) comparisons.

25.3.6.4 sort_heap [lib.sort.heap]

```
template<class RandomAccessIterator>
  void sort_heap(RandomAccessIterator first, RandomAccessIterator last);

template<class RandomAccessIterator, class Compare>
  void sort_heap(RandomAccessIterator first, RandomAccessIterator last,
                 Compare comp);
```

Effects: Sorts elements in the heap [*first*, *last*).
Complexity: At most NlogN comparisons, where N == last - first.
Notes: Not stable.

25.3.7 Minimum and maximum [lib.alg.min.max]

25.3.7.1 min [lib.min]

```
template<class T>
  T min(const T& a, const T& b);

template<class T, class Compare>
  T min(const T& a, const T& b, Compare comp);
```

Returns: The smaller value.
Notes: Returns the first argument when their arguments are equal.

25.3.7.2 max [lib.max]

```
template<class T>
  T max(const T& a, const T& b);

template<class T, class Compare>
  T max(const T& a, const T& b, Compare comp);
```

Returns: The larger value.
Notes: Returns the first argument when their arguments are equal.

25.3.7.3 max_element [lib.max.element]

```
template<class InputIterator>
  InputIterator max_element(InputIterator first, InputIterator last);

template<class InputIterator, class Compare>
  InputIterator max_element(InputIterator first, InputIterator last,
                            Compare comp);
```

Returns: The first iterator `i` in the range `[first, last)` such that for any iterator `j` in the range `[first, last)` the following corresponding conditions hold: `!(*i < *j)` or `comp(*i, *j) == false`.

Complexity: Exactly `max((last - first) - 1, 0)` applications of the corresponding comparisons.

25.3.7.4 min_element [lib.min.element]

```
template<class InputIterator>
  InputIterator min_element(InputIterator first, InputIterator last);

template<class InputIterator, class Compare>
  InputIterator min_element(InputIterator first, InputIterator last,
                            Compare comp);
```

Returns: The first iterator `i` in the range `[first, last)` such that for any iterator `j` in the range `[first, last)` the following corresponding conditions hold: `!(*j < *i)` or `comp(*j, *i) == false`

Complexity: Exactly `max((last - first) - 1, 0)` applications of the corresponding comparisons.

25.3.8 Lexicographical comparison [lib.alg.lex.comparison]

```
template<class InputIterator1, class InputIterator2>
  bool lexicographical_compare(InputIterator1 first1,
                   InputIterator1 last1,
                   InputIterator2 first2,
                   InputIterator2 last2);

template<class InputIterator1, class InputIterator2, class Compare>
  bool lexicographical_compare(InputIterator1 first1,
                   InputIterator1 last1,
                   InputIterator2 first2,
                   InputIterator2 last2,
                   Compare comp);
```

Returns: `true` if the sequence of elements defined by the range `[first1, last1)` is lexicographically less than the sequence of elements defined by the range `[first2, last2)`. Returns `false` otherwise.

Complexity: At most `min((last1 - first1), (last2 - first2))` applications of the corresponding comparison.

25.3.9 Permutation generators [lib.alg.permutation.generators]

25.3.9.1 `next_permutation` [lib.next.permutation]

```
template<class BidirectionalIterator>
  bool next_permutation(BidirectionalIterator first,
                        BidirectionalIterator last);

template<class BidirectionalIterator, class Compare>
  bool next_permutation(BidirectionalIterator first,
                        BidirectionalIterator last,
                        Compare comp);
```

Effects: Takes a sequence defined by the range `[first, last)` and transforms it into the next permutation. The next permutation is found by assuming that the set of all permutations is lexicographically sorted with respect to `operator<` or `comp`. If such a permutation exists, it returns `true`. Otherwise, it transforms the sequence into the smallest permutation, that is, the ascendingly sorted one, and returns `false`.

Complexity: At most `(last - first)/2` swaps.

25.3.9.2 `prev_permutation` [lib.prev.permutation]

```
template<class BidirectionalIterator>
  bool prev_permutation(BidirectionalIterator first,
                        BidirectionalIterator last);

template<class BidirectionalIterator, class Compare>
  bool prev_permutation(BidirectionalIterator first,
                        BidirectionalIterator last,
                        Compare comp);
```

Effects: Takes a sequence defined by the range `[first, last)` and transforms it into the previous permutation. The previous permutation is found by assuming that the set of all permutations is lexicographically sorted with respect to `operator<` or `comp`.

Returns: `true` if such a permutation exists. Otherwise, it transforms the sequence into the largest permutation, that is, the descendingly sorted one, and returns `false`.
Complexity: At most `(last - first)/2` swaps.

25.4 C library algorithms [lib.alg.c.library]

1 Header `<cstdlib>` (partial, Table 61):

Table 61—Header `<cstdlib>` synopsis

Type	Name(s)
Functions:	`bsearch` `qsort`

2 The contents are the same as the Standard C library. *SEE ALSO:* ISO C subclause 7.10.5.

26 Numerics library [lib.numerics]

1 This clause describes components that C++ programs may use to perform seminumerical operations.

2 The following subclauses describe components for complex number types, numeric (*n* -at-a-time) arrays, generalized numeric algorithms, and facilities included from the ISO C library, as summarized in Table 62:

Table 62—Numerics library summary

Subclause	Header(s)
26.1 Complex numbers	`<complex>`
26.2 Numeric arrays	`<valarray>`
26.3 Generalized numeric operations	`<numeric>`
26.4 C library	`<cmath>` `<cstdlib>`

26.1 Complex numbers [lib.complex.numbers]

Header `<complex>` synopsis

```
namespace std {
  template<class T> class complex;
  class complex<float>;
  class complex<double>;
  class complex<long double>;

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

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

template<class T>
  complex<T> operator==(const complex<T>&, const complex<T>&);
template<class T>
  complex<T> operator==(const complex<T>&, const T&);
template<class T>
  complex<T> operator==(const T&, const complex<T>&);
template<class T>
  complex<T> operator!=(const complex<T>&, const complex<T>&);
template<class T>
  complex<T> operator!=(const complex<T>&, const T&);
template<class T>
  complex<T> operator!=(const T&, const complex<T>&);

template<class T>
  istream& operator>>(istream&, complex<T>&);
template<class T>
  ostream& operator<<(ostream&, const complex<T>&);

template<class T>
  T abs(const complex<T>&);
template<class T>
  T norm(const complex<T>&);
template<class T>
  T arg(const complex<T>&);
template<class T>
  T real(const complex<T>&);
template<class T>
  T imag(const complex<T>&);
```

```
  template<class T>
    complex<T> conj(const complex<T>&);
  template<class T>
    complex<T> cos(const complex<T>&);
  template<class T>
    complex<T> cosh(const complex<T>&);
  template<class T>
    complex<T> exp(const complex<T>&);
  template<class T>
    complex<T> log(const complex<T>&);
  template<class T>
    complex<T> sin(const complex<T>&);
  template<class T>
    complex<T> sinh(const complex<T>&);
  template<class T>
    complex<T> sqrt(const complex<T>&);

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

  template<class T>
    complex<T> polar(const T&, const T&);
}
```

1 The header <complex> defines a three types, and numerous functions for representing and manipulating complex numbers.

26.1.1 Complex numbers types [lib.complex.types]

26.1.1.1 Template class complex [lib.complex]

```
namespace std {
  template<class T>
  class complex {
  public:
    complex();
    complex(const T& re);
    complex(const T& re, const T& im);

    T real() const;
    T imag() const;
    template<class X> complex(const complex<X>&);
    template<class X> complex<T>& operator= (const complex<X>&);
    template<class X> complex<T>& operator+=(const complex<X>&);
    template<class X> complex<T>& operator-=(const complex<X>&);
    template<class X> complex<T>& operator*=(const complex<X>&);
    template<class X> complex<T>& operator/=(const complex<X>&);
  };

  class complex<float> {
  public:
    complex(float re = 0.0f, float im = 0.0f);
    explicit complex(const complex<double>&);
    explicit complex(const complex<long double>&);
```

```
  float real() const;
  float imag() const;
  template<class X> complex(const complex<X>&);
  template<class X> complex<float>& operator= (const complex<X>&);
  template<class X> complex<float>& operator+=(const complex<X>&);
  template<class X> complex<float>& operator-=(const complex<X>&);
  template<class X> complex<float>& operator*=(const complex<X>&);
  template<class X> complex<float>& operator/=(const complex<X>&);
};

class complex<double> {
public:
  complex(double re = 0.0, double im = 0.0);
  complex(const complex<float>&);
  explicit complex(const complex<long double>&);

  double real() const;
  double imag() const;
  template<class X> complex(const complex<X>&);
  template<class X> complex<double>& operator= (const complex<X>&);
  template<class X> complex<double>& operator+=(const complex<X>&);
  template<class X> complex<double>& operator-=(const complex<X>&);
  template<class X> complex<double>& operator*=(const complex<X>&);
  template<class X> complex<double>& operator/=(const complex<X>&);
};

class complex<long double> {
public:
  complex(long double re = 0.0L, long double im = 0.0L);
  complex(const complex<float>&);
  complex(const complex<double>&);

  long double real() const;
  long double imag() const;
  template<class X> complex(const complex<X>&);
  template<class X> complex<long double>& operator= (const complex<X>&);
  template<class X> complex<long double>& operator+=(const complex<X>&);
  template<class X> complex<long double>& operator-=(const complex<X>&);
  template<class X> complex<long double>& operator*=(const complex<X>&);
  template<class X> complex<long double>& operator/=(const complex<X>&);
};

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

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

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

template<class T>
  complex<T> operator/(const complex<T>&, const complex<T>&);
template<class T>
  complex<T> operator/(const complex<T>&, const T>&);
template<class T>
  complex<T> operator/(const T&, const complex<T>&);
template<class T>
  complex<T> operator+(const complex<T>&);
template<class T>
  complex<T> operator-(const complex<T>&);

template<class T>
  complex<T> operator==(const complex<T>&, const complex<T>&);
template<class T>
  complex<T> operator==(const complex<T>&, const T&);
template<class T>
  complex<T> operator==(const T&, const complex<T>&);

template<class T>
  complex<T> operator!=(const complex<T>&, const complex<T>&);
template<class T>
  complex<T> operator!=(const complex<T>&, const T&);
template<class T>
  complex<T> operator!=(const T&, const complex<T>&);

template<class T>
  istream& operator>>(istream&, complex<T>&);
template<class T>
  ostream& operator<<(ostream&, const complex<T>&);

template<class T>
  T abs(const complex<T>&);
template<class T>
  T norm(const complex<T>&);
template<class T>
  T arg(const complex<T>&);
template<class T>
  T real(const complex<T>&);
template<class T>
  T imag(const complex<T>&);
```

```
  template<class T>
    complex<T> conj(const complex<T>&);
  template<class T>
    complex<T> cos(const complex<T>&);
  template<class T>
    complex<T> cosh(const complex<T>&);
  template<class T>
    complex<T> exp(const complex<T>&);
  template<class T>
    complex<T> log(const complex<T>&);
  template<class T>
    complex<T> sin(const complex<T>&);
  template<class T>
    complex<T> sinh(const complex<T>&);
  template<class T>
    complex<T> sqrt(const complex<T>&);

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

  template<class T>
    complex<T> polar(const T&, const T&);
}
```

1 The class `complex` describes an object that can store the Cartesian components, `real()` and `imag()`, of a complex number.

26.1.1.1.1 `complex` constructor [lib.complex.cons]

```
template<class T>
  complex(const T& re = T(), const T& im = T());
```

Effects: Constructs an object of class `complex`.

1 Postcondition: `real() == re && imag() == im`.

26.1.1.1.2 `operator+=` [lib.complex.op+=]

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

Effects: Adds the complex value *rhs* to the complex value `*this` and stores the sum in `*this`.
Returns: `*this`.

26.1.1.1.3 `operator-=` [lib.complex.op-=]

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

Effects: Subtracts the complex value *rhs* from the complex value `*this` and stores the difference in `*this`.

Returns: `*this`.

26.1.1.1.4 `operator*=` [lib.complex.op*=]

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

Effects: Multiplies the complex value *rhs* by the complex value `*this` and stores the product in `*this`.
Returns: `*this`.

26.1.1.1.5 `operator/=` [lib.complex.op/=]

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

Effects: Divides the complex value *rhs* into the complex value `*this` and stores the quotient in `*this`.
Returns: `*this`.

26.1.1.2 `complex` operations [lib.complex.ops]

26.1.1.2.1 `operator+` [lib.op+.fc.fc]

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

Returns: `complex<T>(`*lhs*`)`.

```
  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);
```

Returns: `complex<T>(`*lhs*`) += `*rhs*.

26.1.1.2.2 `operator-` [lib.op-.fc.fc]

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

Returns: `complex<T>(-`*lhs*`.real(),-`*lhs*`.imag())`.

```
  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);
```

Returns: `complex<T>(`*lhs*`) -= `*rhs*.

26.1.1.2.3 `operator*` [lib.op*.fc.fc]

```
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 t7 rhs);
template<class T>
  complex<T> operator*(const T& lhs, const complex<T>& rhs);
```

Returns: complex<T>(*lhs*) *= *rhs*.

26.1.1.2.4 operator/ [lib.op/.fc.fc]

```
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);
```

Returns: complex<T>(*lhs*) /= *rhs*.

26.1.1.2.5 operator== [lib.op==.fc.fc]

```
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);
```

Returns: *lhs*.real() == *rhs*.real() && *lhs*.imag() == *rhs*.imag().[158)]

26.1.1.2.6 operator!= [lib.op!=.fc.fc]

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

Returns: *lhs*.real() != *rhs*.real() || *lhs*.imag() != *rhs*.imag().[159)]

26.1.1.2.7 operator>> [lib.ext.fc]

```
template<class T>
  istream& operator>>(istream& is, complex<T>& x);
```

Effects: Evaluates the expression:

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

[158)] The imaginary part is assumed to be T(), or 0.0, for the const T& arguments.
[159)] The imaginary part is assumed to be T(), or 0.0, for the const T& arguments.

1 where *ch* is an object of type `char` and *re* and *im* are objects of type `float`. If the result is nonzero, the function assigns `complex(`*re*`, `*im*`)` to *x*.
Returns: *is*.

26.1.1.2.8 operator<< [lib.ins.fc]

```
template<class T>
  ostream& operator<<(ostream& os, complex x);
```

Returns: *os* `<< '(' <<` *x*`.real() << ',' <<` *x*`.imag() << ')'`.

26.1.1.2.9 exp [lib.complex.exp]

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

Returns: the exponential of *x*.

26.1.1.3 imag [lib.complex.imag]

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

Returns: *x*`.imag()`.

26.1.1.4 log [lib.complex.log]

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

Returns: the logarithm of *x*.

26.1.1.5 norm [lib.complex.norm]

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

Returns: the squared magnitude of *x*.

26.1.1.6 polar [lib.complex.polar]

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

Returns: the `complex` value corresponding to a complex number whose magnitude is *rho* and whose phase angle is *theta*.

26.1.1.7 pow [lib.complex.pow]

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

Returns: *x* raised to the power *y*.

26.1.1.8 real [lib.complex.real]

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

Returns: *x*.real().

26.1.1.9 sin [lib.complex.sin]

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

Returns: the sine of *x*.

26.1.1.10 sinh [lib.complex.sinh]

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

Returns: the hyperbolic sine of *x*.

26.1.1.11 sqrt [lib.sqrt]

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

Returns: the square root of *x*.

26.2 Numeric arrays [lib.numarray]

Header <valarray> synopsis

```
namespace std {
  template<class T> class valarray;       // An array of type T
  class slice;                            // a BLAS-like slice out of an array
  template<class T> class slice_array;
  class gslice;                           // a generalized slice out of an array
  template<class T> class gslice_array;
  template<class T> class mask_array;     // a masked array
  template<class T> class indirect_array; // an indirected array
```

```
template<class T> const valarray<T> operator* (const valarray<T>&, const T&);   |
template<class T> const valarray<T> operator* (const T&, const valarray<T>&);   |
template<class T> const valarray<T> operator/ (const valarray<T>&, const T&);   |
template<class T> const valarray<T> operator/ (const T&, const valarray<T>&);   |
template<class T> const valarray<T> operator% (const valarray<T>&, const T&);   |
template<class T> const valarray<T> operator% (const T&, const valarray<T>&);   |
template<class T> const valarray<T> operator+ (const valarray<T>&, const T&);   |
template<class T> const valarray<T> operator+ (const T&, const valarray<T>&);   |
template<class T> const valarray<T> operator- (const valarray<T>&, const T&);   |
template<class T> const valarray<T> operator- (const T&, const valarray<T>&);   |
template<class T> const valarray<T> operator^ (const valarray<T>&, const T&);   |
template<class T> const valarray<T> operator^ (const T&, const valarray<T>&);   |
template<class T> const valarray<T> operator& (const valarray<T>&, const T&);   |
template<class T> const valarray<T> operator& (const T&, const valarray<T>&);   |
template<class T> const valarray<T> operator| (const valarray<T>&, const T&);   |
template<class T> const valarray<T> operator| (const T&, const valarray<T>&);   |
template<class T> const valarray<T> operator<<(const valarray<T>&, const T&);   |
template<class T> const valarray<T> operator<<(const T&, const valarray<T>&);   |
template<class T> const valarray<T> operator>>(const valarray<T>&, const T&);   |
template<class T> const valarray<T> operator>>(const T&, const valarray<T>&);   |
template<class T> const valarray<T> operator&&(const valarray<T>&, const T&);   |
template<class T> const valarray<T> operator&&(const T&, const valarray<T>&);   |
template<class T> const valarray<T> operator||(const valarray<T>&, const T&);   |
template<class T> const valarray<T> operator||(const T&, const valarray<T>&);   |

template<class T> const valarray<T> operator* (const valarray<T>&, const valarray<T>&);|
template<class T> const valarray<T> operator/ (const valarray<T>&, const valarray<T>&);|
template<class T> const valarray<T> operator% (const valarray<T>&, const valarray<T>&);|
template<class T> const valarray<T> operator+ (const valarray<T>&, const valarray<T>&);|
template<class T> const valarray<T> operator- (const valarray<T>&, const valarray<T>&);|
template<class T> const valarray<T> operator^ (const valarray<T>&, const valarray<T>&);|
template<class T> const valarray<T> operator| (const valarray<T>&, const valarray<T>&);|
template<class T> const valarray<T> operator& (const valarray<T>&, const valarray<T>&);|
template<class T> const valarray<T> operator<<(const valarray<T>&, const valarray<T>&);|
template<class T> const valarray<T> operator>>(const valarray<T>&, const valarray<T>&);|
template<class T> const valarray<T> operator&&(const valarray<T>&, const valarray<T>&);|
template<class T> const valarray<T> operator||(const valarray<T>&, const valarray<T>&);|

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

```
  template<class T> const valarray<T> abs  (const valarray<T>&);
  template<class T> const valarray<T> acos (const valarray<T>&);
  template<class T> const valarray<T> asin (const valarray<T>&);
  template<class T> const valarray<T> atan (const valarray<T>&);
  template<class T> const valarray<T> atan2(const valarray<T>&, const valarray<T>&);
  template<class T> const valarray<T> atan2(const valarray<T>&, const T&);
  template<class T> const valarray<T> atan2(const T&, const valarray<T>&);
  template<class T> const valarray<T> cos  (const valarray<T>&);
  template<class T> const valarray<T> cosh (const valarray<T>&);
  template<class T> const valarray<T> exp  (const valarray<T>&);
  template<class T> const valarray<T> log  (const valarray<T>&);
  template<class T> const valarray<T> log10(const valarray<T>&);
  template<class T> const valarray<T> pow  (const valarray<T>&, const valarray<T>&);
  template<class T> const valarray<T> pow  (const valarray<T>&, const T&);
  template<class T> const valarray<T> pow  (const T&, const valarray<T>&);
  template<class T> const valarray<T> sin  (const valarray<T>&);
  template<class T> const valarray<T> sinh (const valarray<T>&);
  template<class T> const valarray<T> sqrt (const valarray<T>&);
  template<class T> const valarray<T> tan  (const valarray<T>&);
  template<class T> const valarray<T> tanh (const valarray<T>&);
}
```

1 The header <valarray> defines five template classes (valarray, slice_array, gslice_array, mask_array, and indirect_array), two classes (slice and gslice), and a series of related function signatures for representing and manipulating arrays of values.[160)]

2 The valarray array classes are defined to be free of certain forms of aliasing, thus allowing operations on these classes to be optimized.

3 These library functions are permitted to throw an bad_alloc exception if there are not sufficient resources available to carry out the operation. Note that the exception is not mandated.

Box 102

The descriptions of valarray and the associated classes which follow lack any discussion of possible exceptions.

26.2.1 Template class valarray<*T*> [lib.template.valarray]

```
namespace std {
  template<class T> class valarray {
  public:
    valarray();
    valarray(enum Uninitialized, size_t);
    valarray(const T&, size_t);
    valarray(const T*, size_t);
    valarray(const valarray&);
    valarray(const slice_array<T>&);
    valarray(const gslice_array<T>&);
    valarray(const mask_array<T>&);
    valarray(const indirect_array<T>&);
   ~valarray();
```

[160)] If any of the names valarray, slice_array, gslice_array, mask_array, indirect_array, slice or gslice are introduced into a translation unit by any means other than inclusion of the <valarray> header file, the resulting behavior is undefined.

```
    valarray& operator=(const valarray&);
    valarray& operator=(const slice_array<T>&);
    valarray& operator=(const gslice_array<T>&);
    valarray& operator=(const mask_array<T>&);
    valarray& operator=(const indirect_array<T>&);

    size_t length() const;
    operator T*();
    operator const T*() const;

    const T           operator[](size_t);
    T&                operator[](size_t);
    const valarray    operator[](slice) const;
    slice_array<T>    operator[](slice);
    const valarray    operator[](const gslice&) const;
    gslice_array<T>   operator[](const gslice&);
    const valarray    operator[](const valarray<bool>&) const;
    mask_array<T>     operator[](const valarray<bool>&);
    const valarray    operator[](const valarray<int>&) const;
    indirect_array<T> operator[](const valarray<int>&);

    const valarray operator+() const;
    const valarray operator-() const;
    const valarray operator~() const;
    const valarray operator!() const;

    valarray<T>& operator*= (const T&);
    valarray<T>& operator/= (const T&);
    valarray<T>& operator%= (const T&);
    valarray<T>& operator+= (const T&);
    valarray<T>& operator-= (const T&);
    valarray<T>& operator^= (const T&);
    valarray<T>& operator&= (const T&);
    valarray<T>& operator|= (const T&);
    valarray<T>& operator<<=(const T&);
    valarray<T>& operator>>=(const T&);

    valarray<T>& operator*= (const valarray<T>& ab);
    valarray<T>& operator/= (const valarray<T>& ab);
    valarray<T>& operator%= (const valarray<T>& ab);
    valarray<T>& operator+= (const valarray<T>& ab);
    valarray<T>& operator-= (const valarray<T>& ab);
    valarray<T>& operator^= (const valarray<T>&);
    valarray<T>& operator|= (const valarray<T>&);
    valarray<T>& operator&= (const valarray<T>&);
    valarray<T>& operator<<=(const valarray<T>&);
    valarray<T>& operator>>=(const valarray<T>&);

    const T sum() const;
    void    fill(const T&);
    const T min() const;
    const T max() const;

    const valarray<T> shift(int) const;
    const valarray<T> apply(T func(T)) const;
    const valarray<T> apply(T func(const T&)) const;
    void free();
  };
}
```

1 The template class `valarray<T>` is a one-dimensional smart array, with elements numbered sequentially from zero. It is a representation of the mathematical concept of an ordered set of values. The illusion of higher dimensionality may be produced by the familiar idiom of computed indices, together with the powerful subsetting capabilities provided by the generalized subscript operators.[161)]

2 An implementation is permitted to qualify any of the functions declared in `<valarray>` as `inline`.

3 A specialization of `valarray` for a type *T* has well-defined behavior if and only if the type *T* satisfies the following requirements:[162)]

— *T* is not an abstract class (it has no pure virtual member functions);

— *T* is not a reference type;

— *T* is not cv-qualified;

— If *T* is a class, it has a public default constructor;

— If *T* is a class, it has a public copy constructor with the signature `T::T(const T&)`

— If *T* is a class, it has a public destructor;

— If *T* is a class, it has a public assignment operator whose signature is either

```
T& T::operator=(const T&)
```

or

```
T& T::operator=(T)
```

— If *T* is a class, its assignment operator, copy and default constructors, and destructor must correspond to each other in the following sense: Initialization of raw storage using the default constructor, followed by assignment, is semantically equivalent to initialization of raw storage using the copy constructor. Destruction of an object, followed by initialization of its raw storage using the copy constructor, is semantically equivalent to assignment to the original object. This rule states that there must not be any subtle differences in the semantics of initialization versus assignment. This gives an implementation considerable flexibility in how arrays are initialized. For example, an implementation is allowed to initialize a `valarray` by allocating storage using the `new` operator (which implies a call to the default constructor for each element) and then assigning each element its value. Or the implementation can allocate raw storage and use the copy constructor to initialize each element. If the distinction between initialization and assignment is important for a class, or if it fails to satisfy any of the other conditions listed above, the programmer should use `dynarray` instead of `valarray` for that class;

— If *T* is a class, it does not overload unary `operator&`.

4 In addition, many member and ***friend*** functions of `valarray<T>` can be successfully instantiated and will exhibit well-defined behavior if and only if *T* satisfies additional requirements specified for each such member or ***friend*** function.

5 For example, it is legitimate to instantiate `valarray<complex>`, but operator > will not be successfully instantiated for `valarray<complex>` operands, since `complex` does not have any ordering operators.

---

161) The intent is to specify an array template that has the minimum functionality necessary to address aliasing ambiguities and the proliferation of temporaries. Thus, the `valarray` template is neither a matrix class nor a field class. However, it is a very useful building block for designing such classes.

162) In other words, `valarray<T>` should only be instantiated for value types. These include built-in arithmetic types, pointers, the library class `complex`, and instantiations of `valarray` for value types.

26.2.1.1 `valarray` constructors [lib.valarray.cons]

```
valarray();
```

Effects: Constructs an object of class `valarray<T>`, [163] which has zero length until it is passed into a library function as a modifiable lvalue or through a non-constant `this` pointer. This default constructor is essential, since arrays of `valarray` are likely to prove useful. There must also be a way to change the size of an array after initialization; this is supplied by the semantics of the assignment operator.

```
valarray(enum Uninitialized, size_t);
```

1 The array created by this constructor has a length equal to the value of the second argument. The first argument is not used. The elements of the array are constructed using the default constructor for the instantiating type *T*. The extra argument is needed to prevent this constructor from being used by the compiler to silently convert integers to `valarray` objects.

```
valarray(const T&, size_t);
```

2 The array created by this constructor has a length equal to the second argument. The elements of the array are initialized with the value of the first argument.

```
valarray(const T*, size_t);
```

3 The array created by this constructor has a length equal to the second argument `n`. The values of the elements of the array are initialized with the first `n` values pointed to by the first argument. If the value of the second argument is greater than the number of values pointed to by the first argument, the behavior is undefined. This constructor is the preferred method for converting a C array to a `valarray` object.

```
valarray(const valarray&);
```

4 The array created by this constructor has the same length as the argument array. The elements are initialized with the values of the corresponding elements of the argument array. This copy constructor creates a distinct array rather than an alias. Implementations in which arrays share storage are permitted, but they must implement a copy-on-reference mechanism to ensure that arrays are conceptually distinct.

```
valarray(const slice_array<T>&);
valarray(const gslice_array<T>&);
valarray(const mask_array<T>&);
valarray(const indirect_array<T>&);
```

5 These conversion constructors convert one of the four reference templates to a `valarray`.

26.2.1.2 `valarray` destructor [lib.valarray.des]

```
~valarray();
```

[163] For convenience, such objects are referred to as ‘‘arrays’’ throughout the remainder of subclause 26.2.

26.2.1.3 `valarray` assignment [lib.valarray.op=]

```
valarray& operator=(const valarray&);
```

1 The assignment operator modifies the length of the `*this` array to be equal to that of the argument array. Each element of the `*this` array is then assigned the value of the corresponding element of the argument array. Assignment is the usual way to change the length of an array after initialization. Assignment results in a distinct array rather than an alias.

```
valarray& operator=(const slice_array<T>&);
valarray& operator=(const gslice_array<T>&);
valarray& operator=(const mask_array<T>&);
valarray& operator=(const indirect_array<T>&);
```

2 These operators allow the results of a generalized subscripting operation to be assigned directly to a `valarray`.

26.2.1.4 `valarray` length access [lib.valarray.length]

```
size_t length() const;
```

1 This function returns the number of elements in the `this` array.

26.2.1.5 `valarray` pointer conversion [lib.valarray.ptr]

```
operator T*();
operator const T*() const;
```

1 A non-constant array may be converted to a pointer to the instantiating type. A constant array may be converted to a pointer to the instantiating type, qualified by `const`.

2 It is guaranteed that

```
&a[0] == (T*)a
```

for any non-constant `valarray<T> a`. The pointer returned for a non-constant array (whether or not it points to a type qualified by `const`) is valid for the same duration as a reference returned by the `size_t` subscript operator. The pointer returned for a constant array is valid for the lifetime of the array.[164)]

26.2.1.6 `valarray` element access [lib.valarray.access]

```
const T operator[](size_t) const;
T& operator[](size_t);
```

1 When applied to a constant array, the subscript operator returns the value of the corresponding element of the array. When applied to a non-constant array, the subscript operator returns a reference to the corresponding element of the array.

2 Thus, the expression

```
(a[i] = q, a[i]) == q
```

evaluates as true for any non-constant `valarray<T> a`, any `T q`, and for any `size_t i` such that the

164) This form of access is essential for reusability and cross-language programming.

value of `i` is less than the length of `a`.

3 The expression

```
&a[i+j] == &a[i] + j
```

evaluates as true for all `size_t i` and `size_t j` such that `i+j` is less than the length of the non-constant array `a`.

4 Likewise, the expression

```
&a[i] != &b[j]
```

evaluates as true for any two non-constant arrays `a` and `b` and for any `size_t i` and `size_t j` such that `i` is less than the length of `a` and `j` is less than the length of `b`. This property indicates an absence of aliasing and may be used to advantage by optimizing compilers.[165)]

5 The reference returned by the subscript operator for a non-constant array is guaranteed to be valid until the array to whose data it refers is passed into any library function as a modifiable lvalue or through a non-const `this` pointer.

6 Computed assigns [such as `valarray& operator+=(const valarray&)`] do not by themselves invalidate references to array data. If the subscript operator is invoked with a `size_t` argument whose value is not less than the length of the array, the behavior is undefined.

26.2.1.7 `valarray` subset operations [lib.valarray.sub]

```
const valarray    operator[](slice) const;
slice_array<T>    operator[](slice);
const valarray    operator[](const gslice&) const;
gslice_array<T>   operator[](const gslice&);
const valarray    operator[](const valarray<bool>&) const;
mask_array<T>     operator[](const valarray<bool>&);
const valarray    operator[](const valarray<int>&) const;
indirect_array<T> operator[](const valarray<int>&);
```

1 Each of these operations returns a subset of the `this` array. The `const-` qualified versions return this subset as a new `valarray`. The non-`const` versions return a class template object which has reference semantics to the original array.

26.2.1.8 `valarray` unary operators [lib.valarray.unary]

```
const valarray operator+() const;
const valarray operator-() const;
const valarray operator~() const;
const valarray operator!() const;
```

1 Each of these operators may only be instantiated for a type *T* to which the indicated operator can be applied and for which the indicated operator returns a value which is of type &*T* or which may be unambiguously converted to type *T*.

2 Each of these operators returns an array whose length is equal to the length of the `this` array. Each element of the returned array is initialized with the result of applying the indicated operator to the corresponding element of the `this` array.

165) Compilers may take advantage of inlining, constant propagation, loop fusion, tracking of pointers obtained from `operator new`, and other techniques to generate efficient `valarray`s.

26.2.1.9 `valarray` binary operators with scalars [lib.valarray.binary.scal]

```
const valarray operator* (const valarray&, const T&);
const valarray operator/ (const valarray&, const T&);
const valarray operator% (const valarray&, const T&);
const valarray operator+ (const valarray&, const T&);
const valarray operator- (const valarray&, const T&);
const valarray operator^ (const valarray&, const T&);
const valarray operator& (const valarray&, const T&);
const valarray operator| (const valarray&, const T&);
const valarray operator<<(const valarray&, const T&);
const valarray operator>>(const valarray&, const T&);
const valarray operator&&(const valarray&, const T&);
const valarray operator||(const valarray&, const T&);

const valarray operator* (const T&, const valarray&);
const valarray operator/ (const T&, const valarray&);
const valarray operator% (const T&, const valarray&);
const valarray operator+ (const T&, const valarray&);
const valarray operator- (const T&, const valarray&);
const valarray operator^ (const T&, const valarray&);
const valarray operator& (const T&, const valarray&);
const valarray operator| (const T&, const valarray&);
const valarray operator<<(const T&, const valarray&);
const valarray operator>>(const T&, const valarray&);
const valarray operator&&(const T&, const valarray&);
const valarray operator||(const T&, const valarray&);
```

1 Each of these operators may only be instantiated for a type *T* to which the indicated operator can be applied and for which the indicated operator returns a value which is of type *T* or which can be unambiguously converted to type *T*.

2 Each of these operators returns an array whose length is equal to the length of the array argument. Each element of the returned array is initialized with the result of applying the indicated operator to the corresponding element of the array argument and the scalar argument.

26.2.1.10 `valarray` computed assigns with scalars [lib.valarray.cassign.scal]

```
valarray& operator*= (const T&);
valarray& operator/= (const T&);
valarray& operator%= (const T&);
valarray& operator+= (const T&);
valarray& operator-= (const T&);
valarray& operator^= (const T&);
valarray& operator&= (const T&);
valarray& operator|= (const T&);
valarray& operator<<=(const T&);
valarray& operator>>=(const T&);
```

1 Each of these operators may only be instantiated for a type *T* to which the indicated operator can be applied.

2 Each of these operators applies the indicated operation to each element of the `this` array and the scalar argument.

3 The `this` array is then returned by reference.

4 The appearance of an array on the left hand side of a computed assignment does *not* invalidate references or pointers to the elements of the array.

26.2.1.11 `valarray` binary operations with other arrays [lib.valarray.bin.array]

```
const valarray operator* (const valarray&, const valarray&);
const valarray operator/ (const valarray&, const valarray&);
const valarray operator% (const valarray&, const valarray&);
const valarray operator+ (const valarray&, const valarray&);
const valarray operator- (const valarray&, const valarray&);
const valarray operator^ (const valarray&, const valarray&);
const valarray operator& (const valarray&, const valarray&);
const valarray operator| (const valarray&, const valarray&);
const valarray operator<<(const valarray&, const valarray&);
const valarray operator>>(const valarray&, const valarray&);
const valarray operator&&(const valarray&, const valarray&);
const valarray operator||(const valarray&, const valarray&);
```

1 Each of these operators may only be instantiated for a type *T* to which the indicated operator can be applied and for which the indicated operator returns a value which is of type *T* or which can be unambiguously converted to type *T*.

2 Each of these operators returns an array whose length is equal to the lengths of the argument arrays. Each element of the returned array is initialized with the result of applying the indicated operator to the corresponding elements of the argument arrays.

3 If the argument arrays do not have the same length, the behavior is undefined.

26.2.1.12 `valarray` computed assignments with other arrays [lib.valarray.assign.array]

```
valarray& operator*= (const valarray&);
valarray& operator/= (const valarray&);
valarray& operator%= (const valarray&);
valarray& operator+= (const valarray&);
valarray& operator-= (const valarray&);
valarray& operator^= (const valarray&);
valarray& operator&= (const valarray&);
valarray& operator|= (const valarray&);
valarray& operator<<=(const valarray&);
valarray& operator>>=(const valarray&);
```

1 Each of these operators may only be instantiated for a type *T* to which the indicated operator can be applied. Each of these operators performs the indicated operation on each of its elements and the corresponding element of the argument array.

2 The `this` array is then returned by reference.

3 If the `this` array and the argument array do not have the same length, the behavior is undefined. The appearance of an array on the left hand side of a computed assignment does *not* invalidate references or pointers.

26.2.1.13 `valarray` comparison operators with scalars [lib.valarray.comp.scal]

```
const valarray<bool> operator==(const valarray&, const T&);
const valarray<bool> operator!=(const valarray&, const T&);
const valarray<bool> operator< (const valarray&, const T&);
const valarray<bool> operator> (const valarray&, const T&);
const valarray<bool> operator<=(const valarray&, const T&);
const valarray<bool> operator>=(const valarray&, const T&);
const valarray<bool> operator==(const T&, const valarray&);
const valarray<bool> operator!=(const T&, const valarray&);
const valarray<bool> operator< (const T&, const valarray&);
const valarray<bool> operator> (const T&, const valarray&);
const valarray<bool> operator<=(const T&, const valarray&);
const valarray<bool> operator>=(const T&, const valarray&);
```

1 Each of these operators may only be instantiated for a type *T* to which the indicated operator can be applied and for which the indicated operator returns a value which is of type *bool* or which can be unambiguously converted to type *bool*.

2 Each of these operators returns a *bool* array whose length is equal to the length of the array argument. Each element of the returned array is initialized with the result of applying the indicated operator to the corresponding element of the `this` array and the scalar argument.

26.2.1.14 `valarray` comparison operators with other arrays [lib.valarray.comp.array]

```
const valarray<bool> operator==(const valarray&, const valarray&);
const valarray<bool> operator!=(const valarray&, const valarray&);
const valarray<bool> operator< (const valarray&, const valarray&);
const valarray<bool> operator> (const valarray&, const valarray&);
const valarray<bool> operator<=(const valarray&, const valarray&);
const valarray<bool> operator>=(const valarray&, const valarray&);
```

1 Each of these operators may only be instantiated for a type *T* to which the indicated operator can be applied and for which the indicated operator returns a value which is of type *bool* or which can be unambiguously converted to type *bool*.

2 Each of these operators returns a *bool* array whose length is equal to the length of the array arguments. Each element of the returned array is initialized with the result of applying the indicated operator to the corresponding elements of the argument arrays.

3 If the two array arguments do not have the same length, the behavior is undefined.

26.2.1.15 `valarray` sum function [lib.valarray.sum]

```
const T sum() const;
```

This function may only be instantiated for a type *T* to which `operator+=` can be applied. This function returns the sum of all the elements of the array.

1 If the array has length 0, the behavior is undefined. If the array has length 1, `sum` returns the value of element 0. Otherwise, the returned value is calculated by applying `operator+=` to a copy of an element of the array and all other elements of the array in an unspecified order.

26.2.1.16 `valarray` fill function [lib.valarray.fill]

```
void fill(const T&);
```

This function assigns the value of the argument to all the elements of the `this` array. The length of the array is not changed, nor are any pointers or references to the elements of the array invalidated.

26.2.1.17 `valarray` transcendentals [lib.valarray.transcend]

```
const valarray abs  (const valarray&);
const valarray acos (const valarray&);
const valarray asin (const valarray&);
const valarray atan (const valarray&);
const valarray atan2(const valarray&, const valarray&);
const valarray atan2(const valarray&, const T&);
const valarray atan2(const T&, const valarray&);
const valarray cos  (const valarray&);
const valarray cosh (const valarray&);
const valarray exp  (const valarray&);
const valarray log  (const valarray&);
const valarray log10(const valarray&);
const valarray pow  (const valarray&, const valarray&);
const valarray pow  (const valarray&, const T&);
const valarray pow  (const T&, const valarray&);
const valarray sin  (const valarray&);
const valarray sinh (const valarray&);
const valarray sqrt (const valarray&);
const valarray tan  (const valarray&);
const valarray tanh (const valarray&);
```

1 Each of these functions may only be instantiated for a type *T* to which a unique function with the indicated name can be applied. This function must return a value which is of type *T* or which can be unambiguously converted to type *T*.

26.2.1.18 `valarray` min and max functions [lib.valarray.minmax]

```
const T min(const valarray&);
const T max(const valarray&);
```

1 These functions may only be instantiated for a type *T* to which `operator>` and `operator<` may be applied and for which `operator>` and `operator<` return a value which is of type *`bool`* or which can be unambiguously converted to type *`bool`*.

2 These functions return the minimum or maximum value found in the argument array.

3 The value returned for an array of length 0 is undefined. For an array of length 1, the value of element 0 is returned. For all other array lengths, the determination is made using `operator>` and `operator<`, in a manner analogous to the application of `operator+=` for the `sum` function.

26.2.1.19 `valarray` shift function [lib.valarray.shift]

```
const valarray shift(int) const;
```

1 This function returns an array whose length is identical to the `this` array, but whose element values are shifted the number of places indicated by the argument.

2 For example, if the argument has the value 2, the first two elements of the result will be constructed using the default constructor; the third element of the result will be assigned the value of the first element of the argument; etc.

Box 103

Should a cshift (circular shift) function also be defined? This is a common operation in Fortran.

26.2.1.20 `valarray` mapping functions [lib.valarray.map]

```
const valarray apply(T func(T)) const;
const valarray apply(T func(const T&)) const;
```

1 These functions return an array whose length is equal to the `this` array. Each element of the returned array is assigned the value returned by applying the argument function to the corresponding element of the `this` array.

26.2.1.21 `valarray` free function [lib.valarray.free]

```
void free();
```

1 This function sets the length of an array to zero.[166)]

26.2.2 Class `slice` [lib.class.slice]

```
namespace std {
  class slice {
  public:
    slice();
    slice(int, int, int);

    int start() const;
    int length() const;
    int stride() const;
  };
}
```

1 The `slice` class represents a BLAS-like slice from an array. Such a slice is specified by a starting index, a length, and a stride.[167)]

26.2.2.1 `slice` constructors [lib.cons.slice]

```
slice();
slice(int start, int length, int stride);
slice(const slice&);
```

1 The default constructor for `slice` creates a `slice` which specifies no elements. A default constructor is provided only to permit the declaration of arrays of slices. The constructor with arguments for a slice takes a start, length, and stride parameter.

2 For example,

166) An implementation may reclaim the storage used by the array when this function is called.
167) C++ programs may instantiate this class.

```
slice(3, 8, 2)
```

constructs a slice which selects elements 3, 5, 7, ... 17 from an array.

26.2.2.2 `slice` access functions [lib.slice.access]

```
int start() const;
int length() const;
int stride() const;
```

1 These functions return the start, length, or stride specified by a `slice` object.

26.2.3 Template class `slice_array` [lib.template.slice.array]

```
namespace std {
  template <class T> class slice_array {
  public:
    void operator=  (const valarray<T>&) const;
    void operator*= (const valarray<T>&) const;
    void operator/= (const valarray<T>&) const;
    void operator%= (const valarray<T>&) const;
    void operator+= (const valarray<T>&) const;
    void operator-= (const valarray<T>&) const;
    void operator^= (const valarray<T>&) const;
    void operator&= (const valarray<T>&) const;
    void operator|= (const valarray<T>&) const;
    void operator<<=(const valarray<T>&) const;
    void operator>>=(const valarray<T>&) const;

    void fill(const T&);
  private:
    slice_array();
    slice_array(const slice_array&);
    slice_array& operator=(const slice_array&);
    //    remainder implementation defined
  };
}
```

1 The `slice_array` template is a helper template used by the `slice` subscript operator

```
slice_array<T> valarray<T>::operator[](slice);
```

It has reference semantics to a subset of an array specified by a `slice` object.

2 For example, the expression

```
a[slice(1, 5, 3)] = b;
```

has the effect of assigning the elements of `b` to a slice of the elements in `a`. For the slice shown, the elements selected from `a` be 1, 4, ..., 13.

3 Note that programmers may not instantiate `slice_array`, since all its constructors are private. It is intended purely as a helper class and should be transparent to the user.

26.2.3.1 `slice_array` constructors [lib.cons.slice.arr]

```
slice_array();
slice_array(const slice_array&);
```

1 Note that the `slice_array` template has no public constructors. These constructors are declared to be private. These constructors need not be defined.

26.2.3.2 `slice_array` assignment [lib.slice.arr.assign]

```
void         operator=(const valarray<T>&) const;
slice_array& operator=(const slice_array&);
```

1 The second of these two assignment operators is declared private and need not be defined. The first has reference semantics, assigning the values of the argument array elements to selected elements of the `valarray<T>` object to which the `slice_array` object refers.

26.2.3.3 `slice_array` computed assignment [lib.slice.arr.comp.assign]

```
void operator*= (const valarray<T>&) const;
void operator/= (const valarray<T>&) const;
void operator%= (const valarray<T>&) const;
void operator+= (const valarray<T>&) const;
void operator-= (const valarray<T>&) const;
void operator^= (const valarray<T>&) const;
void operator&= (const valarray<T>&) const;
void operator|= (const valarray<T>&) const;
void operator<<=(const valarray<T>&) const;
void operator>>=(const valarray<T>&) const;
```

1 These computed assignments have reference semantics, applying the indicated operation to the elements of the argument array and selected elements of the `valarray<T>` object to which the `slice_array` object refers.

26.2.3.4 `slice_array` fill function [lib.slice.arr.fill]

```
void fill(const T&);
```

1 This function has reference semantics, assigning the value of its argument to the elements of the `valarray<T>` object to which the `slice_array` object refers.

26.2.4 The `gslice` class [lib.class.gslice]

```
namespace std {
  class gslice {
  public:
    gslice();
    gslice(int s, const valarray<int>& l, const valarray<int>& d);

    int           start() const;
    valarray<int> length() const;
    valarray<int> stride() const;
  };
}
```

1 This class represents a generalized slice out of an array. A `gslice` is defined by a starting offset (s), a set of lengths (l_j), and a set of strides (d_j). The number of lengths must equal the number of strides.

2 A `gslice` represents a mapping from a set of indices (i_j), equal in number to the number of strides, to a single index k. It is useful for building multidimensional array classes using the `valarray` template, which is one-dimensional. The set of one-dimensional index values specified by a `gslice` are

$$k = s + sum13j(i_j d_j)$$

where the multidimensional indices i_j range in value from 0 to $l_{ij}-1$.

3 For example, the `gslice` specification

```
start  = 3
length = {2, 4, 3}
stride = {19, 4, 1}
```

yields the sequence of one-dimensional indices

```
k = 3 + (0,1) x 19 = (0,1,2,3) x 4 + (0,1,2) x 1
```

which are ordered as shown in the following table:

```
(i0, i1, i2, k) =
        (0, 0, 0, 3),
        (0, 0, 1, 4),
        (0, 0, 2, 5),
        (0, 1, 0, 7),
        (0, 1, 1, 8),
        (0, 1, 2, 9),
        (0, 2, 0, 11),
        (0, 2, 1, 12),
        (0, 2, 2, 13),
        (0, 3, 0, 15),
        (0, 3, 1, 16),
        (0, 3, 2, 17),
        (1, 0, 0, 22),
        (1, 0, 1, 23),
        ...
        (1, 3, 2, 36)
```

That is, the highest-ordered index turns fastest.

4 It is possible to have degenerate generalized slices in which an address is repeated.

5 For example, if the stride parameters in the previous example are changed to {1, 1, 1}, the first few elements of the resulting sequence of indices will be

```
        (0, 0, 0, 3),
        (0, 0, 1, 4),
        (0, 0, 2, 5),
        (0, 1, 0, 4),
        (0, 1, 1, 5),
        (0, 1, 2, 6),
        ...
```

6 If a degenerate slice is used as the argument to the non-`const` version of `operator[](const gslice&)`, the resulting behavior is undefined.

26.2.4.1 gslice constructors [lib.gslice.cons]

```
gslice();
gslice(int start, const valarray<int>& lengths, const valarray<int>& strides);
gslice(const gslice&);
```

1 The default constructor creates a `gslice` which specifies no elements. The constructor with arguments builds a `gslice` based on a specification of start, lengths, and strides, as explained in the previous section.

26.2.4.2 gslice access functions [lib.gslice.access]

```
int           start() const;
valarray<int> length() const;
valarray<int> stride() const;
```

These access functions return the representation of the start, lengths, or strides specified for the `gslice`.

26.2.5 Template class gslice_array [lib.template.gslice.array]

```
namespace std {
  template <class T> class gslice_array {
  public:
    void operator=  (const valarray<T>&) const;
    void operator*= (const valarray<T>&) const;
    void operator/= (const valarray<T>&) const;
    void operator%= (const valarray<T>&) const;
    void operator+= (const valarray<T>&) const;
    void operator-= (const valarray<T>&) const;
    void operator^= (const valarray<T>&) const;
    void operator&= (const valarray<T>&) const;
    void operator|= (const valarray<T>&) const;
    void operator<<=(const valarray<T>&) const;
    void operator>>=(const valarray<T>&) const;

    void fill(const T&);
  private:
    gslice_array();
    gslice_array(const gslice_array&);
    gslice_array& operator=(const gslice_array&);
    //  remainder implementation defined
  };
}
```

1 This template is a helper template used by the `slice` subscript operator

```
gslice_array<T> valarray<T>::operator[](const gslice&);
```

It has reference semantics to a subset of an array specified by a `gslice` object.

2 Thus, the expression

```
a[gslice(1, length, stride)] = b
```

has the effect of assigning the elements of `b` to a generalized slice of the elements in `a`.

3 Note that programmers may not instantiate `gslice_array`, since all its constructors are private. It is intended purely as a helper class and should be transparent to the user.

26.2.5.1 gslice_array constructors [lib.gslice.array.cons]

```
gslice_array();
gslice_array(const gslice_array&);
```

1 The gslice_array template has no public constructors. It declares the above constructors to be private. These constructors need not be defined.

26.2.5.2 gslice_array assignment [lib.gslice.array.assign]

```
void operator=(const valarray<T>&) const;
gslice_array& operator=(const gslice_array&);
```

1 The second of these two assignment operators is declared private and need not be defined. The first has reference semantics, assigning the values of the argument array elements to selected elements of the valarray<T> object to which the gslice_array refers.

26.2.5.3 gslice_array computed assignment [lib.gslice.array.comp.assign]

```
void operator*= (const valarray<T>&) const;
void operator/= (const valarray<T>&) const;
void operator%= (const valarray<T>&) const;
void operator+= (const valarray<T>&) const;
void operator-= (const valarray<T>&) const;
void operator^= (const valarray<T>&) const;
void operator&= (const valarray<T>&) const;
void operator|= (const valarray<T>&) const;
void operator<<=(const valarray<T>&) const;
void operator>>=(const valarray<T>&) const;
```

1 These computed assignments have reference semantics, applying the indicated operation to the elements of the argument array and selected elements of the valarray<T> object to which the gslice_array object refers.

26.2.5.4 gslice_array fill function [lib.gslice.array.fill]

```
void fill(const T&);
```

1 This function has reference semantics, assigning the value of its argument to the elements of the valarray<T> object to which the gslice_array object refers.

26.2.6 Template class mask_array [lib.template.mask.array]

```
namespace std {
  template <class T> class mask_array {
  public:
    void operator=  (const valarray<T>&) const;
    void operator*= (const valarray<T>&) const;
    void operator/= (const valarray<T>&) const;
    void operator%= (const valarray<T>&) const;
    void operator+= (const valarray<T>&) const;
    void operator-= (const valarray<T>&) const;
    void operator^= (const valarray<T>&) const;
    void operator&= (const valarray<T>&) const;
    void operator|= (const valarray<T>&) const;
    void operator<<=(const valarray<T>&) const;
    void operator>>=(const valarray<T>&) const;

    void fill(const T&);
  private:
    mask_array();
    mask_array(const mask_array&);
    mask_array& operator=(const mask_array&);
    //   remainder implementation defined
  };
}
```

1 This template is a helper template used by the mask subscript operator

```
mask_array<T> valarray<T>::operator[](const valarray<bool>&);
```

It has reference semantics to a subset of an array specified by a boolean mask. Thus, the expression

```
a[mask] = b;
```

has the effect of assigning the elements of `b` to the masked elements in `a` (those for which the corresponding element in `mask` is `true`.

2 Note that C++ programs may not declare instances of `mask_array`, since all its constructors are private. It is intended purely as a helper class, and should be transparent to the user.

26.2.6.1 `mask_array` constructors [lib.mask.array.cons]

```
mask_array();
mask_array(const mask_array&);
```

1 The `mask_array` template has no public constructors. It declares the above constructors to be private. These constructors need not be defined.

26.2.6.2 `mask_array` assignment [lib.mask.array.assign]

```
void operator=(const valarray<T>&) const;
mask_array& operator=(const mask_array&);
```

1 The second of these two assignment operators is declared private and need not be defined. The first has reference semantics, assigning the values of the argument array elements to selected elements of the `valarray<T>` object to which it refers.

26.2.6.3 `mask_array` computed assignment [lib.mask.array.comp.assign]

```
void operator*= (const valarray<T>&) const;
void operator/= (const valarray<T>&) const;
void operator%= (const valarray<T>&) const;
void operator+= (const valarray<T>&) const;
void operator-= (const valarray<T>&) const;
void operator^= (const valarray<T>&) const;
void operator&= (const valarray<T>&) const;
void operator|= (const valarray<T>&) const;
void operator<<=(const valarray<T>&) const;
void operator>>=(const valarray<T>&) const;
```

1 These computed assignments have reference semantics, applying the indicated operation to the elements of the argument array and selected elements of the `valarray<T>` object to which the mask object refers.

26.2.6.4 `mask_array` fill function [lib.mask.array.fill]

```
void fill(const T&);
```

This function has reference semantics, assigning the value of its argument to the elements of the `valarray<T>` object to which the `mask_array` object refers.

26.2.7 Template class `indirect_array` [lib.template.indirect.array]

```
namespace std {
  template <class T> class indirect_array {
  public:
    void operator=  (const valarray<T>&) const;
    void operator*= (const valarray<T>&) const;
    void operator/= (const valarray<T>&) const;
    void operator%= (const valarray<T>&) const;
    void operator+= (const valarray<T>&) const;
    void operator-= (const valarray<T>&) const;
    void operator^= (const valarray<T>&) const;
    void operator&= (const valarray<T>&) const;
    void operator|= (const valarray<T>&) const;
    void operator<<=(const valarray<T>&) const;
    void operator>>=(const valarray<T>&) const;

    void fill(const T&);
  private:
    indirect_array();
    indirect_array(const indirect_array&);
    indirect_array& operator=(const indirect_array&);
    //  remainder implementation defined
  };
}
```

1 This template is a helper template used by the indirect subscript operator

```
indirect_array<T> valarray<T>::operator[](const valarray<int>&);
```

It has reference semantics to a subset of an array specified by an `indirect_array`. Thus the expression

```
a[indirect] = b;
```

has the effect of assigning the elements of `b` to the elements in `a` whose indices appear in `indirect`.

2 Note that C++ programs may not declare instances of `indirect_array`, since all its constructors are private. It is intended purely as a helper class, and should be transparent to the user.

26.2.7.1 `indirect_array` constructors [lib.indirect.array.cons]

```
indirect_array();
indirect_array(const indirect_array&);
```

The `indirect_array` template has no public constructors. The constructors listed above are private. These constructors need not be defined.

26.2.7.2 `indirect_array` assignment [lib.indirect.array.assign]

```
void operator=(const valarray<T>&) const;
indirect_array& operator=(const indirect_array&);
```

1 The second of these two assignment operators is declared private and need not be defined. The first has reference semantics, assigning the values of the argument array elements to selected elements of the `valarray<T>` object to which it refers.

2 If the `indirect_array` specifies an element in the `valarray<T>` object to which it refers more than once, the behavior is undefined.

3 For example,

```
int addr = {2, 3, 1, 4, 4};
valarray<int> indirect(addr, 5);
valarray<double> a(0., 10), b(1., 5);
array[indirect] = b;
```

results in undefined behavior since element 4 is specified twice in the indirection.

26.2.7.3 `indirect_array` computed assignment [lib.indirect.array.comp.assign]

```
void operator*= (const valarray<T>&) const;
void operator/= (const valarray<T>&) const;
void operator%= (const valarray<T>&) const;
void operator+= (const valarray<T>&) const;
void operator-= (const valarray<T>&) const;
void operator^= (const valarray<T>&) const;
void operator&= (const valarray<T>&) const;
void operator|= (const valarray<T>&) const;
void operator<<=(const valarray<T>&) const;
void operator>>=(const valarray<T>&) const;
```

1 These computed assignments have reference semantics, applying the indicated operation to the elements of the argument array and selected elements of the `valarray<T>` object to which the `indirect_array` object refers.

2 If the `indirect_array` specifies an element in the `valarray<T>` object to which it refers more than once, the behavior is undefined.

26.2.7.4 `indirect_array` fill function [lib.indirect.array.fill]

```
void fill(const T&);
```

1 This function has reference semantics, assigning the value of its argument to the elements of the `valarray<T>` object to which the `indirect_array` object refers.

26.3 Generalized numeric operations [lib.numeric.ops]

Header `<numeric>` synopsis

```
namspace std {
  template <class InputIterator, class T>
    T accumulate(InputIterator first, InputIterator last, T init);
  template <class InputIterator, class T, class BinaryOperation>
    T accumulate(InputIterator first, InputIterator last, T init,
                 BinaryOperation binary_op);
  template <class InputIterator1, class InputIterator2, class T>
    T inner_product(InputIterator1 first1, InputIterator1 last1,
                    InputIterator2 first2, T init);
  template <class InputIterator1, class InputIterator2, class T,
            class BinaryOperation1, class BinaryOperation2>
    T inner_product(InputIterator1 first1, InputIterator1 last1,
                    InputIterator2 first2, T init,
                    BinaryOperation1 binary_op1, BinaryOperation2 binary_op2);
  template <class InputIterator, class OutputIterator>
    OutputIterator partial_sum(InputIterator first, InputIterator last,
                               OutputIterator result);
  template <class InputIterator, class OutputIterator, class BinaryOperation>
    OutputIterator partial_sum(InputIterator first, InputIterator last,
                               OutputIterator result, BinaryOperation binary_op);
  template <class InputIterator, class OutputIterator>
    OutputIterator adjacent_difference(InputIterator first, InputIterator last,
                                       OutputIterator result);
  template <class InputIterator, class OutputIterator, class BinaryOperation>
    OutputIterator adjacent_difference(InputIterator first, InputIterator last,
                                       OutputIterator result, BinaryOperation binary_op);
}
```

26.3.1 Accumulate [lib.accumulate]

```
template <class InputIterator, class T>
  T accumulate(InputIterator first, InputIterator last, T init);
template <class InputIterator, class T, class BinaryOperation>
  T accumulate(InputIterator first, InputIterator last, T init,
 BinaryOperation binary_op);
```

Effects: Initializes the accumulator `acc` with the initial value `init` and then modifies it with `acc = acc + *i` or `acc = binary_op(acc, *i)` for every iterator `i` in the range `[first, last)` in order.[168] `binary_op` is assumed not to cause side effects.

[168] `accumulate` is similar to the APL reduction operator and Common Lisp reduce function, but it avoids the difficulty of defining the result of reduction on an empty sequence by always requiring an initial value.

26.3.2 Inner product [lib.inner.product]

```
template <class InputIterator1, class InputIterator2, class T>
  T inner_product(InputIterator1 first1, InputIterator1 last1,
                  InputIterator2 first2, T init);
template <class InputIterator1, class InputIterator2, class T,
          class BinaryOperation1, class BinaryOperation2>
  T inner_product(InputIterator1 first1, InputIterator1 last1,
                  InputIterator2 first2, T init,
                  BinaryOperation1 binary_op1, BinaryOperation2 binary_op2);
```

Effects: Computes its result by initializing the accumulator `acc` with the initial value `init` and then modifying it with

```
acc = acc + (*i1) * (*i2)
```

or

```
acc = binary_op1(acc, binary_op2(*i1, *i2))
```

for every iterator `i1` in the range `[first, last)` and iterator `i2` in the range `[first2, first2 + (last - first))` in order.
Requires: `binary_op1` and `binary_op2` shall not cause side effects.

26.3.3 Partial sum [lib.partial.sum]

```
template <class InputIterator, class OutputIterator>
  OutputIterator partial_sum(InputIterator first, InputIterator last,
           OutputIterator result);
template <class InputIterator, class OutputIterator, class BinaryOperation>
  OutputIterator partial_sum(InputIterator first, InputIterator last,
                             OutputIterator result, BinaryOperation binary_op);
```

Effects: Assigns to every iterator `i` in the range `[result, result + (last - first))` a value correspondingly equal to

```
((...(*first + *(first + 1)) + ...) + *(first + (i - result)))
```

or

```
binary_op(binary_op(..., binary_op(*first, *(first + 1)),...), *(first + (i - result)
```

Returns: `result + (last - first)`.
Complexity: Exactly `(last - first) - 1` applications of `binary_op`.
Requires: `binary_op` is expected not to have any side effects.
Notes: `result` may be equal to `first`.

26.3.4 Adjacent difference [lib.adjacent.difference]

```
template <class InputIterator, class OutputIterator>
  OutputIterator adjacent_difference(InputIterator first, InputIterator last,
                                     OutputIterator result);
template <class InputIterator, class OutputIterator, class BinaryOperation>
  OutputIterator adjacent_difference(InputIterator first, InputIterator last,
                                     OutputIterator result, BinaryOperation binary_op);
```

Effects: Assigns to every element referred to by iterator `i` in the range `[result + 1, result + (last - first))` a value correspondingly equal to

```
*(first + (i - result)) - *(first + (i - result) - 1)
```

or

```
binary_op(*(first + (i - result)), *(first + (i - result) - 1))
```

`result` gets the value of `*first`.
Requires: `binary_op` shall not have any side effects.
Notes: `result` may be equal to `first`.
Returns: `result + (last - first)`.
Complexity: Exactly `(last - first) - 1` applications of `binary_op`.

26.4 C Library [lib.c.math]

1 Headers `<cmath>` and `<cstdlib>` (`abs()`, `div()`, `rand()`, `srand()`).

Table 62—Header `<cmath>` synopsis

Type	Name(s)			
Macro:	`HUGE_VAL`			
Functions:				
`acos`	`ceil`	`fabs`	`ldexp`	`pow`
`asin`	`cos`	`floor`	`log`	`sin`
`atan`	`cosh`	`fmod`	`log10`	`sinh`
`atan2`	`exp`	`frexp`	`modf`	`sqrt`

Table 62—Header `<cstdlib>` synopsis

Type	Name(s)	
Macros:	`RAND_MAX`	
Types:	`div_t`	`ldiv_t`
Functions:		
`abs`	`labs`	`srand`
`div`	`ldiv`	`rand`

2 The contents are the same as the Standard C library.

SEE ALSO: ISO C subclauses 7.5, 7.10.2, 7.10.6.

27 Input/output library [lib.input.output]

1 This clause describes components that C++ programs may use to perform input/output operations.

2 The following subclauses describe requirements for stream parameters, and components for forward declarations of iostreams, predefined iostreams objects, base iostreams classes, stream buffering, stream formatting and manipulators, string streams, and file streams, as summarized in Table 63:

Table 63—Input/output library summary

Subclause	Header(s)
27.1 Requirements	
27.2 Forward declarations	`<iosfwd>`
27.3 Standard iostream objects	`<iostream>`
27.4 Iostreams base classes	`<ios>`
27.5 Stream buffers	`<streambuf>`

27.6 Formatting and manipulators	`<istream>`
	`<ostream>`
	`<iomanip>`

27.7 String streams	`<sstream>` `<cstdlib>`

27.8 File streams	`<cstream>`
	`<fstream>`
	`<cstdio>`
	`<cwchar>`

27.1 Iostreams requirements [lib.iostreams.requirements]

27.1.1 Definitions [lib.iostreams.definitions]

1 Additional definitions:

— **character** In this clause, the term ‘‘character’’ means the generalized element of text data. Each text data consist of a sequence of character. So the term does not only means the `char` type object, and the `wchar_t` type object, but also any sort of classes which provides the definitions specified in (21.1.1.1).

— **character container type** Character container type is a class or a type which represents a *character*. It is used for one of the template parameter of the template iostream classes.

— **template iostream classes** The template iostream classes are the template classes which takes two template parameters: `charT` and `traits`. The class, `charT`, represents the character container class and the class, traits, represents the traits structure which provides the definition of the functionality necessary to implement the template iostream classes.

— **narrow-oriented iostream classes** The narrow-oriented iostream classes are the versions of the template iostream classes specialized with the character container class, `char`. The traditional iostream classes are regarded as the narrow-oriented iostream classes (27.3.1).

— **wide-oriented iostream classes** The wide-oriented iostream classes are the versions of the template iostream classes specialized with the character container class, `wchar_t` (27.3.2).

— **repositional streams and arbitrary-positional streams** There are two types of stream: repositional and arbitrary-positional.

Box 104

There are also non-positional streams in which no seeking is possible.

2 With a *repositional stream*, we can seek to only the position where we previously encountered. On the other hand, with an *arbitrary-positional*stream, we can seek to any integral position within the length of the stream.

3 For a stream buffer which is corresponding to a repositional stream (but not an arbitrary-positional stream), all we can do is either to fetch the current position of the stream buffer or to specify the previous position which we have already fetched from the stream buffer.

4 Every arbitrary-positional stream is repositional.

5 If a repositional stream returns a ***POS_T*** object, and some arithmetic operations (`operator+=`, `operator-`, `operator+=`, `operator-=`) are applied, the behavior of the stream after restoring the position with the modified ***POS_T*** object is undefined. It means that a ***POS_T*** object whose parent stream is repositional shall not apply any arithmetic operations.

6 It is not ensured that in case the validity of a certain ***POS_T*** object is broken, the error shall be informed. Or there is no way to check the validity of a certain ***POS_T*** object.

— **Invalid *POS_T* value** The stream operations whose return type is ***POS_T*** may return ***POS_T***((***OFF_T***)(-1)) as signal to some error occurs. This return value is called the *invalid* ***POS_T*** *value*. The conversion ***POS_T***((***OFF_T***)(-1)) constructs the invalid ***POS_T*** value, which is available only for comparing to the return value of such member functions.

27.1.2 Type requirements [lib.iostreams.type.reqmts]

1 There are several types and functions needed for implementing the template iostream classes. Some of these types and functions depend on the definition of the character container type. The collection of these functions describes the behavior which the implementation of the template iostream class expects to the character container class.

27.1.2.1 Type *CHAR_T* [lib.iostreams.char.t]

1 Those who provide a character container type as the template parameter have to provide all of these functions as well as the container class itself. The collection of these functions can be regarded as the collection of the common definitions for the implementation of the character container class.

2 No special definition/declaration is provided here. The base class (or struct), `string_char_traits` provides the definitions common between the template string classes and the template iostream classes.

3 Convertible to type ***INT_T***.

27.1.2.2 Type *INT_T* [lib.iostreams.int.t]

1 Another *character container type* which can also hold an end-of-file value. It is used as the return type of some of the iostream class member functions. If ***CHAR_T*** is either `char` or `wchar_t`, ***INT_T*** shall be `int` or `wint_t`, respectively.

27.1.2.3 Type *OFF_T* [lib.iostreams.off.t]

1 A type that can represent offsets to positional information.[169] It is used to represent:

— a signed displacement, measured in characters, from a specified position within a sequence.

— an absolute position within a sequence.

2 The value ***OFF_T***(-1) can be used as an error indicator.

3 If an ***OFF_T*** object has a value other than the parent stream returns (for example, assigned an arbitrary integer), the value may not apply to any streams.

4 Convertible to type ***POS_T***.[170] But no validity of the resulting ***POS_T*** value is ensured, whether or not the ***OFF_T*** value is valid.

27.1.2.4 Type *POS_T* [lib.iostreams.pos.t]

1 An implementation-defined type for seek operations which describes an object that can store all the information necessary to reposition to the position.

2 The type ***POS_T*** describes an object that can store all the information necessary to restore an arbitrary sequence to a previous *stream position* and *conversion state.*[171]

3 A class or built-in type `P` satisfies the requirements of a position type, and a class or built-in type `O` satisfies the requirements of an offset type if the following expressions are valid, as shown in Table 64.

4 In the following table,

— `P` refers to type ***POS_T***,

— `p` and `q` refer to an values of type ***POS_T***,

— `O` refers to type ***OFF_T***,

— `o` refers to a value of type ***OFF_T***, and

— `i` refers to a value of type `int`.

169) It is usually a synonym for one if the signed basic integral types whose representation at least as many bits as type `long`.
170) An implementation may use the same type for both ***OFF_T*** and ***POS_T***.
171) The conversion state is used for sequences that translate between wide-character and generalized multibyte encoding, as described in Amendment 1 to the C Standard.

Table 64—Position type requirements

expression	return type	operational semantics	assertion/note pre/post-condition
`P(i)`			`p == P(i)` note: a destructor is assumed.
`P p(i);` `P p = i;`			post: `p == P(i)`.
`P(o)`	`POS_T`		converts from offset
`O(p)`	`OFF_T`		converts to offset
`p == q`	convertible to `bool`		== is an equivalence relation
`p != q`	convertible to `bool`	`!(p==q)`	
`q = p + o` `p += o`	`POS_T`	`+` offset	`q-o == p`
`q = p - o` `p -= o`	`POS_T`	`-` offset	`q+o == p`
`o = p - q`	`OFF_T`	distance	`q+o == p`

27.2 Forward declarations [lib.iostream.forward]

Header `<iosfwd>` synopsis

```
namespace std {
  template<class charT> class basic_ios;
  template<class charT> class basic_istream;
  template<class charT> class basic_ostream;

  typedef basic_ios<char>     ios;
  typedef basic_ios<wchar_t> wios;

  typedef basic_istream<char>     istream;
  typedef basic_istream<wchar_t> wistream;

  typedef basic_ostream<char>     ostream;
  typedef basic_ostream<wchar_t> wostream;
}
```

1 The template class `basic_ios<charT,traits>` serves as a base class for the classes `basic_istream<charT,traits>` and `basic_ostream<charT,traits>`.

2 The class `ios` is an instance of the template class `basic_ios`, specialized by the type `char`.

3 The class `wios` is a version of the template class `basic_ios` specialized by the type `wchar_t`.

27.3 Standard iostream objects [lib.iostream.objects]

Header `<iostream>` synopsis

```
#include <fstream>

namespace std {
  extern ifstream cin;
  extern ofstream cout;
  extern ofstream cerr;
  extern ofstream clog;

  extern wifstream win;
  extern wofstream wout;
  extern wofstream werr;
  extern wofstream wlog;
}
```

1 The header <iostream> declares objects that associate objects with the standard C streams provided for by the functions declared in <cstdio> (27.8.3).

2 Mixing operations on corresponding wide- and narrow-character streams follows the same semantics as mixing such operations on FILE s, as specified in Amendment 1 of the ISO C standard.

Box 105

ISSUE: These objects need to be constructed and associations established before dynamic initialization of file scope variables is begun.

The objects are constructed, and the associations are established, the first time an object of class basic_ios<charT,traits>::Init is constructed. The objects are *not* destroyed during program execution.[172)]

27.3.1 Narrow stream objects [lib.narrow.stream.objects]

27.3.1.1 Object cin [lib.cin]

```
ifstream cin;
```

1 The object cin controls input from an unbuffered stream buffer associated with the object stdin, declared in <cstdio>.

2 After the object cin is initialized, cin.tie() returns cout.

27.3.1.2 Object cout [lib.cout]

```
ofstream cout;
```

1 The object cout controls output to an unbuffered stream buffer associated with the object stdout, declared in <cstdio> (27.8.3).

27.3.1.3 Object cerr [lib.cerr]

```
ofstream cerr;
```

1 The object cerr controls output to an unbuffered stream buffer associated with the object stderr, declared in <cstdio> (27.8.3).

172) Constructors and destructors for static objects can access these objects to read input from stdin or write output to stdout or stderr.

2 After the object cerr is initialized, cerr.flags() & unitbuf is nonzero.

27.3.1.4 Object clog [lib.clog]

```
ofstream clog;
```

1 The object clog controls output to a stream buffer associated with the object stderr, declared in <cstdio> (27.8.3).

Box 106

ISSUE: The destination of clog ought to be implementation defined.

27.3.2 Wide stream objects [lib.wide.stream.objects]

1

27.3.2.1 Object win [lib.win]

```
wifstream win;
```

1 The object win controls input from an unbuffered stream buffer associated with the object stdin, declared in <cstdio>.

2 After the object win is initialized, win.tie() returns wout.

27.3.2.2 Object wout [lib.wout]

```
wofstream wout;
```

1 The object wout controls output to an unbuffered stream buffer associated with the object stdout, declared in <cstdio> (27.8.3).

27.3.2.3 Object werr [lib.werr]

```
wofstream werr;
```

1 The object werr controls output to an unbuffered stream buffer associated with the object stderr, declared in <cstdio> (27.8.3).

2 After the object werr is initialized, werr.flags() & unitbuf is nonzero.

27.3.2.4 Object wlog [lib.wlog]

```
wofstream wlog;
```

1 The object wlog controls output to a stream buffer associated with the object stderr, declared in <cstdio> (27.8.3).

Box 107

ISSUE: The destination of wlog ought to be implementation defined.

27.4 Iostreams base classes [lib.iostreams.base]

Header `<ios>` synopsis

```
namespace std {
  typedef OFF_T  streamoff;
  typedef OFF_T wstreamoff;
  typedef INT_T streamsize;

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

  class ios_base;
  template<class charT, class traits = ios_traits<charT> >
    class basic_ios;
  typedef basic_ios<char>     ios;
  typedef basic_ios<wchar_t> wios;

// 27.4.5, 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& uppercase  (ios_base& str);
  ios_base& nouppercase(ios_base& str);
// adjustfield:
  ios_base& internal   (ios_base& str);
  ios_base& left       (ios_base& str);
  ios_base& right      (ios_base& str);
// basefield:
  ios_base& dec        (ios_base& str);
  ios_base& hex        (ios_base& str);
  ios_base& oct        (ios_base& str);
// floatfield:
  ios_base& fixed      (ios_base& str);
  ios_base& scientific (ios_base& str);
}
```

27.4.1 Types [lib.stream.types]

27.4.1.1 Type `streamoff` [lib.streamoff]

```
typedef OFF_T streamoff;
```

1 The type `streamoff` is an implementation-defined type that satisfies the requirements of type ***OFF_T***(_lib.iostreams.off.t_).

27.4.1.2 Type `wstreamoff` [lib.wstreamoff]

```
typedef OFF_T wstreamoff;
```

1 The type `wstreamoff` is an implementation-defined type that satisfies the requirements of type ***OFF_T***(_lib.iostreams.off.t_).

27.4.1.3 Type `streampos` [lib.streampos]

```
typedef POS_T streampos;
```

1 The type `streampos` is an implementation-defined type that satisfies the requirements of type ***POS_T***(_lib.iostreams.pos.t_).

27.4.1.4 Type `wstreampos` [lib.wstreampos]

```
typedef POS_T wstreampos;
```

1 The type `streampos` is an implementation-defined type that satisfies the requirements of type ***POS_T***(_lib.iostreams.pos.t_).

27.4.1.5 Type `streamsize` [lib.streamsize]

```
typedef INT_T streamsize;
```

1 The type `streamsize` is a synonym for one of the signed basic integral types. It is used to represent the number of characters transferred in an I/O operation, or the size of I/O buffers.[173)]

27.4.2 Template struct `ios_traits` [lib.ios.traits]

```
namespace std {
  template <class charT> struct ios_traits<charT> {
    typedef charT char_type;
    typedef INT_T int_type;
    typedef POS_T pos_type;
    typedef OFF_T off_type;
    typedef INT_T state_type;

    static char_type  to_char_type(int_type);
    static int_type   to_int_type (char_type);
    static bool       eq_char_type(char_type, char_type);
    static bool       eq_int_type (int_type, int_type);
    static int_type   eof();
    static int_type   not_eof(char_type c);
    static bool       is_eof (int_type);
```

[173)] `streamsize` is used in most places where ISO C would use `size_t`. Most of the uses of `streamsize` could use `size_t`, except for the `strstreambuf` constructors, which require negative values. It should probably be the signed type corresponding to `size_t` (which is what Posix.2 calls `ssize_t`).

```
    static char_type  newline();
    static bool       is_whitespace(ctype<char_type> ctype, char_type c);
    static char_type  eos();
    static size_t     length(const char_type* s);
    static char_type* copy(char_type* dst, const char_type* src, size_t n) ;
  };
}
```

1 The template struct `ios_traits<charT>` is a traits class which maintains the definitions of the types and functions necessary to implement the template iostream classes. The template parameter `charT` represents the *character container type* and each specialized version provides the default definitions corresponding to the specialized character container type.

2 An implementation may provide the following two specializations of `ios_traits`:

```
struct ios_traits<char>
struct ios_traits<wchar_t>;
```

27.4.2.1 Member functions [lib.ios.traits.members]

27.4.2.1.1 `ios_traits::to_char_type` [lib.ios.char.traits::to.char.type]

```
char_type to_char_type(int_type c);
```

Effects: Converts a valid character value represented in the `int_type` to the corresponding `char_type` value. If *c* is the end-of-file value, the return value is unspecified.

27.4.2.1.2 `ios_traits::to_int_type` [lib.ios.char.traits::to.int.type]

```
int_type to_int_type(char_type c);
```

Effects: Converts a valid character value represented in the `char_type` to the corresponding `int_type` value.

27.4.2.1.3 `ios_traits::eq_char_type` [lib.ios.char.traits::eq.char.type]

```
bool eq_char_type(char_type c1, char_type c2);
```

Returns: `true` if *c1* and *c2* represent the same character.

27.4.2.1.4 `ios_traits::eq_int_type` [lib.ios.char.traits::eq.int.type]

```
bool eq_int_type(int_type c1, int_type c2);
```

Returns: `true` if *c1* and *c2* represent the same character.

27.4.2.1.5 `ios_traits::eof` [lib.ios.char.traits::eof]

```
int_type eof();
```

Returns: an `int_type` value which represents the end-of-file. It is returned by several functions to indicate end-of-file state (no more input from an input sequence or no more output permitted to an output sequence), or to indicate an invalid return value.

27.4.2.1.6 ios_traits::not_eof [lib.ios.char.traits::not.eof]

```
int_type not_eof(char_type c);
```

Box 108
Should the argument type be `int_type`?

Returns: a value other than the end-of-file, even if `c==eof()`.
Notes: It is used in `basic_streambuf<charT,traits>::overflow()`.
Returns: `int_type(c) if c!=eof()`.

27.4.2.1.7 ios_traits::is_eof [lib.ios.char.traits::is.eof]

```
bool is_eof(int_type c);
```

Returns: `true` if *c* represents the end-of-file.

27.4.2.1.8 ios_traits::newline [lib.ios.char.traits::newline]

```
char_type newline();
```

Returns: a character value which represent the newline character of the basic character set.
Notes: It appears as the default parameter of `basic_istream<charT,traits>::getline()`.

27.4.2.1.9 ios_traits::is_whitespace [lib.ios.char.traits::is.whitespace]

```
bool is_whitespace(char_type c, ctype<char_type> ctype);
```

Returns: `true` if *c* represents one of the white space characters. The default definition is as if it returns *ctype*`.isspace(c)`.

1 An implementation of the template iostream classes may use all of the above static member functions in addition to the following three functions provided from the base struct `string_char_traits<CHAR_T>`.

27.4.2.1.10 ios_traits::eos [lib.ios.char.traits::eos]

```
char_type eos();
```

Returns: The null character which is used for the terminator of null terminated character strings. The default constructor for the character container type provides the value.

27.4.2.1.11 ios_traits::length [lib.ios.char.traits::length]

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

Effects: Determines the length of a null terminated character string pointed to by *s*.

27.4.2.1.12 ios_traits::copy [lib.ios.char.traits::copy]

```
char_type* copy(char_type* dest, const char_type* src, size_t n);
```

Effects: Copies *n* characters from the object pointed to by *src* into the object pointed to by *dest*. If copying takes place between objects that overlap, the behavior is undefined.

27.4.3 Class ios_base [lib.ios.base]

```
namespace std {
  class ios_base {
    class failure;
    typedef T1 fmtflags;
    static const fmtflags boolalpha;
    static const fmtflags dec;
    static const fmtflags fixed;
    static const fmtflags hex;
    static const fmtflags internal;
    static const fmtflags left;
    static const fmtflags oct;
    static const fmtflags right;
    static const fmtflags scientific;
    static const fmtflags showbase;
    static const fmtflags showpoint;
    static const fmtflags showpos;
    static const fmtflags skipws;
    static const fmtflags unitbuf;
    static const fmtflags uppercase;
    static const fmtflags adjustfield;
    static const fmtflags basefield;
    static const fmtflags floatfield;

    typedef T2 iostate;
    static const iostate badbit;
    static const iostate eofbit;
    static const iostate failbit;
    static const iostate goodbit;

    typedef T3 openmode;
    static const openmode app;
    static const openmode ate;
    static const openmode binary;
    static const openmode in;
    static const openmode out;
    static const openmode trunc;

    typedef T4 seekdir;
    static const seekdir beg;
    static const seekdir cur;
    static const seekdir end;

    class Init;

    operator bool() const
    bool operator!() const
    ios_type& copyfmt(const ios_type& rhs);

    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;
    void exceptions(iostate except);
    fmtflags flags() const;
    fmtflags flags(fmtflags fmtfl);
    fmtflags setf(fmtflags fmtfl);
    fmtflags setf(fmtflags fmtfl, fmtflags mask);
    void unsetf(fmtflags mask);

    int_type fill() const;
    int_type fill(int_type ch);
    int precision() const;
    int precision(int prec);
    int width() const;
    int width(int wide);

    locale imbue(const locale& loc);
    locale getloc() const;
    static int xalloc();
    long&  iword(int index);
    void*& pword(int index);

protected:      // ?
    ios_base();
  private:
//  static int index;    exposition only
//  int* iarray;         exposition only
//  void** parray;       exposition only
  };
}
```

Box 109

ISSUE: `fill` can't work in the non-template base class. Specifying fill character. It is represented as `int_type`.

1 `ios_base` defines several member types:

— a class `failure` derived from `exception`;

— a class `Init`;

— three bitmask types, `fmtflags`, `iostate`, and `openmode`;

— an enumerated type, `seekdir`.

2 It maintains several kinds of data:

— state information that reflects the integrity of the stream buffer;

— control information that influences how to interpret (format) input sequences and how to generate (format) output sequences;

— additional information that is stored by the program for its private use.

Box 110

For the sake of exposition, the maintained data is presented here as:

— `static int` *`index`*, specifies the next available unique index for the integer or pointer arrays maintained for the private use of the program, initialized to an unspecified value;

— `int*` *`iarray`*, points to the first element of an arbitrary-length integer array maintained for the private use of the program;

— `void**` *`parray`*, points to the first element of an arbitrary-length pointer array maintained for the private use of the program.

27.4.3.1 Types [lib.ios.types]

27.4.3.1.1 Class `ios_base::failure` [lib.ios::failure]

```
namespace std {
  class ios_base::failure : public exception {
  public:
    failure(const string& what = message);
    virtual ~failure();
//  virtual string what() const;  inherited
 };
}
```

1 The class `failure` defines the base class for the types of all objects thrown as exceptions, by functions in the iostreams library, to report errors detected during stream buffer operations.

27.4.3.1.1.1 `ios_base::failure` constructor [lib.ios.base::failure.cons]

```
failure(const string& what = message);
```

Effects: Constructs an object of class `failure`, initializing the base class with `exception(`*`what`*`)`.

1 The string ***message*** is an implementation-defined default value.

27.4.3.1.2 Type `ios_base::fmtflags` [lib.ios::fmtflags]

```
typedef T1 fmtflags;
```

1 The type `fmtflags` is a bitmask type (17.2.2.1.2). Setting its elements has the effects indicated in Table 65:

Table 65—`fmtflags` effects

Element	Effect(s) if set
`boolalpha`	insert and extract `bool` type in alphabetic format
`dec`	converts integer input or generates integer output in decimal base
`fixed`	generate floating-point output in fixed-point notation;
`hex`	converts integer input or generates integer output in hexadecimal base;
`internal`	adds fill characters at a designated internal point in certain generated output;
`left`	adds fill characters on the left (initial positions) of certain generated output;
`oct`	converts integer input or generates integer output in octal base;
`right`	adds fill characters on the right (final positions) of certain generated output;
`scientific`	generates floating-point output in scientific notation;
`showbase`	generates a prefix indicating the numeric base of generated integer output;
`showpoint`	generates a decimal-point character unconditionally in generated floating-point output;
`showpos`	generates a + sign in non-negative generated numeric output;
`skipws`	skips leading white space before certain input operations;
`unitbuf`	flushes output after each output operation;
`uppercase`	replaces certain lowercase letters with their uppercase equivalents in generated output.

2 Type `fmtflags` also defines the constants indicated in Table 66:

Table 66—`fmtflags` constants

Constant	Allowable values
`adjustfield`	`left | right | internal`
`basefield`	`dec | oct | hex`
`floatfield`	`scientific | fixed`

27.4.3.1.3 Type `ios_base::iostate` [lib.ios::iostate]

```
typedef T2 iostate;
```

1 The type `iostate` is a bitmask type (17.2.2.1.2) that contains the elements indicated in Table 67:

Table 67—`iostate` effects

Element	Effect(s) if set
`badbit`	indicates a loss of integrity in an input or output sequence (such as an irrecoverable read error from a file);
`eofbit`	indicates that an input operation reached the end of an input sequence;
`failbit`	indicates that an input operation failed to read the expected characters, or that an output operation failed to generate the desired characters.

2 Type iostate also defines the constant:

— goodbit, the value zero.

27.4.3.1.4 Type ios_base::openmode [lib.ios::openmode]

```
typedef T3 openmode;
```

1 The type openmode is a bitmask type (17.2.2.1.2). It contains the elements indicated in Table 68:

Table 68—openmode effects

Element	Effect(s) if set
app	seek to end-of-file before each write to the file
ate	open a file and seek to end-of-file immediately after opening the file
binary	perform input and output in binary mode (as opposed to text mode)
in	open a file for input
out	open a file for output
trunc	truncate an existing file when opening it

27.4.3.1.5 Type ios_base::seekdir [lib.ios::seekdir]

```
typedef T4 seekdir;
```

1 The type seekdir is an enumerated type (17.2.2.1.1) that contains the elements indicated in Table 69:

Table 69—seekdir effects

Element	Meaning
beg	request a seek (positioning for subsequent input or output within a sequence) relative to the beginning of the stream
cur	request a seek relative to the current position within the sequence
end	request a seek relative to the current end of the sequence

27.4.3.1.6 Class ios_base::Init [lib.ios::Init]

```
namespace std {
  class ios_base::Init {
  public:
    Init();
   ~Init();
  private:
//  static int init_cnt; exposition only
  };
}
```

1 The class Init describes an object whose construction ensures the construction of the four objects declared in <iostream> that associate file stream buffers with the standard C streams provided for by the functions declared in <cstdio> (27.8.3).

Box 111

For the sake of exposition, the maintained data is presented here as:

— `static int` *`init_cnt`*, counts the number of constructor and destructor calls for class `Init`, initialized to zero.

27.4.3.1.6.1 `ios_base::Init` constructor [lib.ios.base::init.cons]

```
Init();
```

Effects: Constructs an object of class `Init`. If *`init_cnt`* is zero, the function stores the value one in *`init_cnt`*, then constructs and initializes the objects `cin` (27.3.1.1), `cout` (27.3.1.2), `cerr` (27.3.1.3), `clog` (27.3.1.4), `win` (27.3.2.1), `wout` (27.3.2.2), `werr` (27.3.2.3), and `wlog` (27.3.2.4). In any case, the function then adds one to the value stored in *`init_cnt`*.

27.4.3.1.6.2 `ios_base::Init` destructor [lib.ios.base::init.des]

```
~Init();
```

Effects: Destroys an object of class `Init`. The function subtracts one from the value stored in *`init_cnt`* and, if the resulting stored value is one, calls `cout.flush()`, `cerr.flush()`, and `clog.flush()`.

27.4.3.2 Member functions [lib.ios.members]

27.4.3.2.1 `ios_base::operator bool` [lib.ios::operator.bool]

```
operator bool() const
```

Returns: `fail() | bad()`.

27.4.3.2.2 `ios_base::operator!` [lib.ios::operator!]

```
bool operator!() const
```

Returns: if `fail() | bad()`.

27.4.3.2.3 `ios_base::copyfmt` [lib.ios::copyfmt]

```
ios_base<charT,traits>& copyfmt(const ios_base<charT,traits>& rhs);
```

Effects: Assigns to the member objects of `*this` the corresponding member objects of *`rhs`*, except that:

— *`sb`* and `rdstate()` are left unchanged;

— `exceptions()` is altered last by calling `exception(`*`rhs.except`*`)`.

— The contents of arrays pointed at by `pword` and `iword` are copied not the pointers themselves.[174)]

1 If any newly stored pointer values in *`*this`* point at objects stored outside the object *`rhs`*, and those objects are destroyed when *`rhs`* is destroyed, the newly stored pointer values are altered to point at newly constructed copies of the objects.

Returns: `*this`.

[174)] This suggests an infinite amount of copying, but the implementation can keep track of the maximum element of the arrays that is non-zero.

27.4.3.2.4 ios_base::rdstate [lib.ios::rdstate]

```
iostate rdstate() const;
```

Returns: The control state of the stream buffer.

27.4.3.2.5 ios_base::clear(iostate) [lib.ios::clear.basic.ios]

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

Postcondition: *state* == rdstate().
Effects: If *sb* is a null pointer, the function sets badbit in rdstate(). If rdstate() & exceptions() is zero, returns. Otherwise, the function throws an object *fail* of class *failure*, constructed with argument values that are implementation-defined.

27.4.3.2.6 ios_base::setstate(iostate) [lib.ios::setstate.basic.ios]

```
void setstate(iostate state);
```

Effects: Calls clear(rdstate() | *state*).

27.4.3.2.7 ios_base::good [lib.ios::good]

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

Returns: rdstate() == 0

27.4.3.2.8 ios_base::eof [lib.ios::eof]

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

Returns: true if eofbit is set in rdstate().

27.4.3.2.9 ios_base::fail [lib.ios::fail]

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

Returns: true if failbit or badbit is set in rdstate().[175)]

27.4.3.2.10 ios_base::bad [lib.ios::bad]

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

Returns: true if badbit is set in rdstate().

27.4.3.2.11 ios_base::exceptions [lib.ios::exceptions]

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

Returns: A mask that determines what elements set in rdstate() cause exceptions to be thrown.

```
void exceptions(iostate except);
```

Postcondition: *except* == exceptions().

[175)] Checking badbit also for fail() is historical practice.

Effects: Calls `clear(rdstate())`.

27.4.3.2.12 `ios_base::flags` [lib.ios::flags]

```
fmtflags flags() const;
```

Returns: The format control information for both input and output.

```
fmtflags flags(fmtflags fmtfl);
```

Postcondition: *`fmtfl`* `== flags()`.
Returns: The previous value of `flags()`.

27.4.3.2.13 `ios_base::setf(fmtflags)` [lib.ios::setf]

```
fmtflags setf(fmtflags fmtfl);
```

Effects: Sets *`fmtfl`* in `flags()`.
Returns: The previous value of `flags()`.

```
fmtflags setf(fmtflags fmtfl, fmtflags mask);
```

Effects: Clears *`mask`* in `flags()`, sets *`fmtfl`* & *`mask`* in `flags()`.
Returns: The previous value of `flags()`.

27.4.3.2.14 `ios_base::unsetf(fmtflags)` [lib.ios::unsetf]

```
void unsetf(fmtflags mask);
```

Effects: Clears *`mask`* in `flags()`.

27.4.3.2.15 `basic_ios::fill` [lib.ios::fill]

```
int_type fill() const;
```

Returns: The character to use to pad (fill) an output conversion to the specified field width.

```
int_type fill(int_type fillch);
```

Postcondition: *`fillch`* `== fill()`.
Returns: The previous value of `fill()`.

27.4.3.2.16 `ios_base::precision` [lib.ios::precision]

```
int precision() const;
```

Returns: The precision (number of digits after the decimal point) to generate on certain output conversions.

```
int precision(int prec);
```

Postcondition: *`prec`* `== precision()`.
Returns: The previous value of `precision()`.

27.4.3.2.17 ios_base::width [lib.ios::width]

```
int width() const;
```

Returns: The field width (number of characters) to generate on certain output conversions.

```
int width(int wide);
```

Postcondition: *wide* == width().
Returns: The previous value of width().

27.4.3.2.18 ios_base::imbue [lib.ios::imbue]

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

Postcondition: *loc* == getloc().
Notes: In case the member pointer *sb* of the basic_streambuf<charT,traits> has already initialized, the function also imbues the object pointed to by *sb*.
Returns: The previous value of getloc().

27.4.3.2.19 ios_base::getloc [lib.ios::getloc]

```
locale getloc() const;
```

Returns: The classic C locale if no locale has been imbued. Otherwise, returns the locale in which to perform locale-dependent input and output operations.

27.4.3.2.20 ios_base::xalloc [lib.ios::xalloc]

```
static int xalloc();
```

Returns: *index*++.

27.4.3.2.21 ios_base::iword [lib.ios::iword]

```
long& iword(int idx);
```

Effects: If *iarray* is a null pointer, allocates an array of int of unspecified size and stores a pointer to its first element in *iarray*. The function then extends the array pointed at by *iarray* as necessary to include the element *iarray*[*idx*]. Each newly allocated element of the array is initialized to zero.
Returns: *iarray*[*idx*].
Notes: After a subsequent call to iword(int) for the same object, the earlier return value may no longer be valid.[176)]

27.4.3.2.22 ios_base::pword [lib.ios::pword]

```
void* & pword(int idx);
```

Effects: If *parray* is a null pointer, allocates an array of pointers to *void* of unspecified size and stores a pointer to its first element in *parray*. The function then extends the array pointed at by *parray* as necessary to include the element *parray*[*idx*]. Each newly allocated element of the array is initialized to a null pointer.
Returns: *parray*[*idx*].

176) An implementation is free to implement both the integer array pointed at by *iarray* and the pointer array pointed at by *parray* as sparse data structures, possibly with a one-element cache for each.

Notes: After a subsequent call to `pword(int)` for the same object, the earlier return value may no longer be valid.

27.4.3.2.23 `ios_base` constructor [lib.ios.base.cons]

```
ios_base();
```

Effects: Constructs an object of class `ios_base`, assigning initial values to its member objects by calling `init(0)`.

27.4.3.2.24 `ios_base::init` [lib.ios.base::init]

```
void init(basic_streambuf<charT,traits>* sb);
```

Effects: The postconditions of this function are indicated in Table 70:

Table 70—`init` effects

Element	Value
`rdstate()`	`goodbit` if *`sb`* is not a null pointer, otherwise `badbit`.
`exceptions()`	`goodbit`
`flags()`	`skipws | dec`
`width()`	zero
`precision()`	6
`fill()`	the space character
`getloc()`	new `locale()`, which means the default value is the current global locale;[177]
`index`	***???***
`iarray`	a null pointer
`parray`	a null pointer

Box 112

Note: the default locale value shall be `global` but not transparent because the locality of the stream buffer will be unchanged between its lifetime.

27.4.4 Template class `basic_ios` [lib.ios]

```
namespace std {
  template<class charT, class traits = ios_traits<charT> >
  class basic_ios : public ios_base {
    typedef basic_ios<charT,traits> ios_type;
  public:
// Added for consistency:
    typedef INT_T            streamsize;
    typedef charT            char_type;
    typedef traits::int_type int_type;
    typedef traits::pos_type pos_type;
    typedef traits::off_type off_type;
    int_type  eof()     { return traits::eof(); }
    char_type newline() { return traits::newline(); }
```

[177] Usually, `classic()`.

```
    basic_ios(basic_streambuf<charT,traits>* sb);
    virtual ~basic_ios();
    basic_ostream<charT,traits>* tie() const;
    basic_ostream<charT,traits>* tie(basic_ostream<charT,traits>* tiestr);
    basic_streambuf<charT,traits>* rdbuf() const;
    basic_streambuf<charT,traits>* rdbuf(basic_streambuf<charT,traits>* sb);

  protected:
      basic_ios();
      void init(basic_streambuf<charT,traits>* sb);
  };
}
```

Box 113

Note: template parameter coupling between `basic_ios` and `basic_streambuf`: Both `basic_ios` and `basic_streambuf` corresponding to it should take the same template parameter `<charT,traits>`. We need not allow to make a couple of `basic_ios<wchar_t>` and `basic_streambuf<char>`.

27.4.4.1 `basic_ios` constructors [lib.basic.ios.sb.cons]

```
basic_ios(basic_streambuf<charT,traits>* sb);
```

Effects: Constructs an object of class `basic_ios`, assigning initial values to its member objects by calling `init(sb)`.

```
basic_ios();
```

Effects: Constructs an object of class `basic_ios`,

Box 114

TBS

```
void init(basic_streambuf<charT,traits>* sb);
```

Box 115

TBS

27.4.4.2 Member functions [lib.basic.ios.members]

27.4.4.2.1 `basic_ios::tie` [lib.ios::tie]

```
basic_ostream<charT,traits>* tie() const;
```

Returns: An output sequence that is *tied* to (synchronized with) an input sequence controlled by the stream buffer.

```
basic_ostream<charT,traits>*
  tie(basic_ostream<charT,traits>* tiestr);
```

Postcondition: *tiestr* == tie().
Returns: The previous value of tie().

27.4.4.2.2 basic_ios::rdbuf [lib.ios::rdbuf]

```
basic_streambuf<charT,traits>* rdbuf() const;
```

Returns: The streambuf associated with the stream.

```
basic_streambuf<charT,traits>*
  rdbuf(basic_streambuf<charT,traits>* sb);
```

Postcondition: *sb* == rdbuf().
Effects: Calls clear().

Box 116

Note: need to modify so as to describe the occurence of imbueing getloc()::codecvt into the argument stream buffer.

Returns: The previous value of rdbuf().

27.4.5 ios_base manipulators [lib.std.ios.manip]

27.4.5.1 fmtflags manipulators [lib.fmtflags.manip]

```
ios_base& boolalpha(ios_base& str);
```

Effects: Calls *str*.setf(ios_base::boolalpha).
Returns: *str*.[178)]

```
ios_base& noboolalpha(ios_base& str);
```

Effects: Calls *str*.unsetf(ios_base::boolalpha).
Returns: *str*.

```
ios_base& showbase(ios_base& str);
```

Effects: Calls *str*.setf(ios_base::showbase).
Returns: *str*.

```
ios_base& noshowbase(ios_base& str);
```

Effects: Calls *str*.unsetf(ios_base::showbase).
Returns: *str*.

```
ios_base& showpoint(ios_base& str);
```

Effects: Calls *str*.setf(ios_base::showpoint).
Returns: *str*.

```
ios_base& noshowpoint(ios_base& str);
```

Effects: Calls *str*.unsetf(ios_base::showpoint).
Returns: *str*.

```
ios_base& showpos(ios_base& str);
```

Effects: Calls *str*.setf(ios_base::showpos).
Returns: *str*.

```
ios_base& noshowpos(ios_base& str);
```

Effects: Calls *str*.unsetf(ios_base::showpos).
Returns: *str*.

```
ios_base& skipws(ios_base& str);
```

Effects: Calls *str*.setf(ios_base::skipws).
Returns: *str*.

```
ios_base& noskipws(ios_base& str);
```

Effects: Calls *str*.unsetf(ios_base::skipws).
Returns: *str*.

```
ios_base& uppercase(ios_base& str);
```

Effects: Calls *str*.setf(ios_base::uppercase).
Returns: *str*.

```
ios_base& nouppercase(ios_base& str);
```

Effects: Calls *str*.unsetf(ios_base::uppercase).
Returns: *str*.

27.4.5.2 `adjustfield` manipulators [lib.adjustfield.manip]

```
ios_base& internal(ios_base& str);
```

Effects: Calls .*str*.setf(ios_base::internal, ios_base::adjustfield)
Returns: *str*.

```
ios_base& left(ios_base& str);
```

Effects: Calls .*str*.setf(ios_base::left, ios_base::adjustfield)
Returns: *str*.

```
ios_base& right(ios_base& str);
```

Effects: Calls .*str*.setf(ios_base::right, ios_base::adjustfield)
Returns: *str*.

27.4.5.3 `basefield` manipulators [lib.basefield.manip]

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

Effects: Calls .*str*.setf(ios_base::dec, ios_base::basefield)
Returns: *str*.

```
ios_base& hex(ios_base& str);
```

178) The function signature `dec(ios_base&)` can be called by the function signature `basic_ostream& stream::operator<<(basic_ostream& (*)(basic_ostream&))` to permit expressions of the form `cout << dec` to change the format flags stored in `cout`.

Effects: Calls .*str*.`setf(ios_base::hex,` ios_base::basefield)
Returns: *str*.

```
ios_base& oct(ios_base& str);
```

Effects: Calls .*str*.`setf(ios_base::oct,` ios_base::basefield)
Returns: *str*.

27.4.5.4 `floatfield` manipulators [lib.floatfield.manip]

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

Effects: Calls .*str*.`setf(ios_base::fixed,` ios_base::floatfield)
Returns: *str*.

```
ios_base& scientific(ios_base& str);
```

Effects: Calls .*str*.`setf(ios_base::scientific,` ios_base::floatfield)
Returns: *str*.

27.5 Stream buffers [lib.stream.buffers]

Header `<streambuf>` synopsis

```
namespace std {
  template<class charT, class traits = ios_traits<charT> >
    class basic_streambuf;
  typedef basic_streambuf<char>     streambuf;
  typedef basic_streambuf<wchar_t> wstreambuf;
}
```

1 The header `<streambuf>` defines types that control input from and output to *character* sequences.

27.5.1 Stream buffer requirements [lib.streambuf.reqts]

27.5.2 Template class `basic_streambuf<charT,traits>` [lib.streambuf]

```
namespace std {
  template<class charT, class traits = ios_traits<charT> >
  class basic_streambuf {
  public:
    typedef charT            char_type;
    typedef traits::int_type int_type;
    typedef traits::pos_type pos_type;
    typedef traits::off_type off_type;

// In order to simplify descriptions and as a convenience for programmers:
    typedef basic_ios<char> ios;

    int_type  eof()     { return traits::eof(); }
    char_type newline() { return traits::newline(); }
```

Template class `basic_streambuf<charT,traits>`

```
 public:
    virtual ~basic_streambuf();
    pos_type pubseekoff(off_type off,
                        basic_ios<charT,traits>::seekdir way,
                        basic_ios<charT,traits>::openmode which
                            = basic_ios<charT,traits>::in
                            | basic_ios<charT,traits>::out);
    pos_type pubseekpos(pos_type sp,
                        basic_ios<charT,traits>::openmode which
                            = basic_ios<charT,traits>::in
                            | basic_ios<charT,traits>::out);
    basic_streambuf<char_type,traits>*
             pubsetbuf(char_type* s, streamsize n);

    int      in_avail();
    int      pubsync();
    int_type sbumpc();
    int_type sgetc();
    int      sgetn(char_type* s, streamsize n);
    int_type snextc();
    int_type sputbackc(char_type c);

    int      sungetc();
    int      sputc(int c);
    int_type 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 int_type overflow (int_type c = eof());
    virtual int_type pbackfail(int_type c = eof());

    virtual int      showmany();
    virtual int_type underflow();
    virtual int_type uflow();
    virtual streamsize xsgetn(char_type* s, streamsize n);
    virtual streamsize xsputn(const char_type* s, streamsize n);
    virtual pos_type seekoff(off_type off,
                             basic_ios<charT,traits>::seekdir way,
                             basic_ios<charT,traits>::openmode which = in | out);
    virtual pos_type seekpos(pos_type sp,
                             basic_ios<charT,traits>::openmode which = in | out);
    virtual basic_streambuf<char_type,traits>*
              setbuf(char_type* s, streamsize n);
    virtual int sync();
  };
}
```

1 The class template basic_streambuf<charT,traits> serves as an abstract base class for deriving various *stream buffers* whose objects each control two *character sequences*:

— a character *input sequence*;

— a character *output sequence*.

2 The class streambuf is an instantiation of the template class basic_streambuf specialized by the type char.

3 The class wstreambuf is an instantiation of the template class basic_streambuf specialized by the type wchar_t.

4 Stream buffers can impose various constraints on the sequences they control. Some constraints are:

— The controlled input sequence can be not readable.

— The controlled output sequence can be not writable.

— The controlled sequences can be associated with the contents of other representations for character sequences, such as external files.

— The controlled sequences can support operations *directly*to or from associated sequences.

— The controlled sequences can impose limitations on how the program can read characters from a sequence, write characters to a sequence, put characters back into an input sequence, or alter the stream position.

5 Each sequence is characterized by three pointers which, if non-null, all point into the same charT array object. The array object represents, at any moment, a (sub)sequence of characters from the sequence. Operations performed on a sequence alter the values stored in these pointers, perform reads and writes directly to or from associated sequences, and alter ''the stream position'' and conversion state as needed to maintain this subsequence relationship. The three pointers are:

— the *beginning pointer*, or lowest element address in the array (called *xbeg* here);

— the *next pointer*, or next element address that is a current candidate for reading or writing (called *xnext* here);

— the *end pointer*, or first element address beyond the end of the array (called *xend* here).

6 The following semantic constraints shall always apply for any set of three pointers for a sequence, using the pointer names given immediately above:

— If *xnext* is not a null pointer, then *xbeg* and *xend* shall also be non-null pointers into the same charT array, as described above.

— If *xnext* is not a null pointer and *xnext* < *xend* for an output sequence, then a *write position* is available. In this case, **xnext* shall be assignable as the next element to write (to put, or to store a character value, into the sequence).

— If *xnext* is not a null pointer and *xbeg* < *xnext* for an input sequence, then a *putback position* is available. In this case, *xnext*[-1] *shall have a defined value and is the next (preceding) element to store a character that is put back into the input sequence.*

— If *xnext* is not a null pointer and *xnext* < *xend* for an input sequence, then a *read position* is available. In this case, **xnext* shall have a defined value and is the next element to read (to get, or to obtain a character value, from the sequence).

27.5.2.1 basic_streambuf constructors [lib.basic.streambuf.cons]

```
basic_streambuf();
```

Effects: Constructs an object of class `basic_streambuf<charT,traits>` and initializes:[179)]

— all its pointer member objects to null pointers,

— the `getloc()` member object to the return value of `global()`.

Notes: Once the `getloc()` member is initialized its locale-dependent behavior does not change until the next imbueing of the locale.

27.5.2.2 Member functions [lib.streambuf.members]

27.5.2.2.1 basic_streambuf::pubseekoff [lib.streambuf::pubseekoff]

```
pos_type pubseekoff(off_type off,
                    basic_ios<charT,traits>::seekdir way,
                    basic_ios<charT,traits>::openmode which
    = basic_ios<charT,traits>::in | basic_ios<charT,traits>::out);
```

Returns: `seekoff(off,way,which)`.

27.5.2.2.2 basic_streambuf::pubseekpos [lib.streambuf::pubseekpos]

```
pos_type pubseekpos(pos_type sp,
      basic_ios<charT,traits>::openmode which
    = basic_ios<charT,traits>::in | basic_ios<charT,traits>::out);
```

Returns: `seekpos(sp,which)`.

27.5.2.2.3 basic_streambuf::pubsetbuf [lib.streambuf::pubsetbuf]

```
basic_streambuf<char_type,traits>*
setbuf(char_type* s, streamsize n);
```

Returns: `setbuf(s,n)`. *

27.5.2.2.4 basic_streambuf::in_avail [lib.streambuf::in.avail]

```
int in_avail();
```

Returns: If the input sequence read position is not available, returns `showmany()`. Otherwise, returns `egptr() - gptr()`.

27.5.2.2.5 basic_streambuf::pubsync [lib.streambuf::pubsync]

```
int pubsync();
```

Returns: `sync()`.

---

179) The default constructor is protected for class `basic_streambuf` to assure that only objects for classes derived from this class may be constructed.

27.5.2.2.6 basic_streambuf::sbumpc [lib.streambuf::sbumpc]

```
int_type sbumpc();
```

Returns: If the input sequence read position is not available, returns `uflow()`. Otherwise, returns `char_type(*gptr())` and increments the next pointer for the input sequence.

27.5.2.2.7 basic_streambuf::sgetc [lib.streambuf::sgetc]

```
int_type sgetc();
```

Returns: If the input sequence read position is not available, returns `underflow()`. Otherwise, returns `char_type(*gptr())`.

27.5.2.2.8 basic_streambuf::sgetn [lib.streambuf::sgetn]

```
int sgetn(char_type* s, streamsize n);
```

Returns: `xsgetn(s,n)`.

27.5.2.2.9 basic_streambuf::snextc [lib.streambuf::snextc]

```
int_type snextc();
```

Effects: Calls `sbumpc()` and, if that function returns `eof()`, returns `eof()`. Otherwise, returns `sgetc()`.
Notes: Uses `traits::eof()`.

27.5.2.2.10 basic_streambuf::sputbackc [lib.streambuf::sputbackc]

```
int_type sputbackc(char_type c);
```

Returns: If the input sequence putback position is not available, or if `c != gptr()[-1]`, returns `pbackfail(c)`. Otherwise, decrements the next pointer for the input sequence and returns `*gptr()`.

27.5.2.2.11 basic_streambuf::sungetc [lib.streambuf::sungetc]

```
int sungetc();
```

Returns: If the input sequence putback position is not available, returns `pbackfail()`. Otherwise, decrements the next pointer for the input sequence and returns `*gptr()`.

27.5.2.2.12 basic_streambuf::sputc [lib.streambuf::sputc]

```
int sputc(int c);
```

Returns If the output sequence write position is not available, returns `overflow(c)`. Otherwise, stores `c` at the next pointer for the output sequence, increments the pointer, and returns `*pptr()`.

27.5.2.2.13 basic_streambuf::sputn [lib.streambuf::sputn]

```
int_type sputn(const char_type* s, streamsize n);
```

Returns: `xsputn(s,n)`.

27.5.2.2.14 basic_streambuf::eback [lib.streambuf::eback]

```
char_type* eback() const;
```

Returns: The beginning pointer for the input sequence.

27.5.2.2.15 basic_streambuf::gptr [lib.streambuf::gptr]

```
char_type* gptr() const;
```

Returns: The next pointer for the input sequence.

27.5.2.2.16 basic_streambuf::egptr [lib.streambuf::egptr]

```
char_type* egptr() const;
```

Returns: The end pointer for the output sequence.

27.5.2.2.17 basic_streambuf::gbump [lib.streambuf::gbump]

```
void gbump(int n);
```

Effects: Advances the next pointer for the input sequence by *n*.

27.5.2.2.18 basic_streambuf::setg [lib.streambuf::setg]

```
void setg(char_type* gbeg,
   char_type* gnext,
   char_type* gend);
```

Postconditions: *gbeg* `==` `eback()`, *gnext* `==` `gptr()`, and *gend* `==` `egptr()`.

27.5.2.2.19 basic_streambuf::pbase [lib.streambuf::pbase]

```
char_type* pbase() const;
```

Returns: The beginning pointer for the output sequence.

27.5.2.2.20 basic_streambuf::pptr [lib.streambuf::pptr]

```
char_type* pptr() const;
```

Returns: The next pointer for the output sequence.

27.5.2.2.21 basic_streambuf::epptr [lib.streambuf::epptr]

```
char_type* epptr() const;
```

Returns: The end pointer for the output sequence.

27.5.2.2.22 basic_streambuf::pbump [lib.streambuf::pbump]

```
void pbump(int n);
```

Effects: Advances the next pointer for the output sequence by *n*.

27.5.2.2.23 basic_streambuf::setp [lib.streambuf::setp]

```
void setp(char_type* pbeg, char_type* pend);
```

Postconditions: *pbeg* == pbase(), *pbeg* == pptr(), and *pend* == epptr().

27.5.2.2.24 basic_streambuf::overflow [lib.streambuf::overflow]

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

Effects: Consumes some initial subsequence of the characters of the *pending sequence*. The pending sequence is defined as the concatenation of

a) if pbase() is NULL then the empty sequence otherwise, pptr() - pbase() characters beginning at pbase().

b) if *c* == eof() then the empty sequence otherwise, the sequence consisting of *c*.

Notes: The member functions sputc() and sputn() call this function in case that no room can be found in the put buffer enough to accomodate the argument character sequence.

1 Every overriding definition of this virtual function shall obey the following constraints:

1) The effect of consuming a character on the associated output sequence is specified[180)]

2) Let r be the number of characters in the pending sequence not consumed. If r is non-zero then pbase() and pptr() must be set so that: pptr() - pbase() == *r* and the *r* characters starting at pbase() are the associated output stream. In case *r* is zero (all characters of the pending sequence have been consumed) then either pbase() is set to NULL, or pbase() and pptr() are both set to the same non- NULL value.

3) The function may fail if either appending some character to the associated output stream fails or if it is unable to establish pbase() and pptr() according to the above rules.

Returns: eof() or throws an exception if the function fails.
Otherwise, returns some value other than eof() to indicate success.[181)]

Default behavior: Returns eof().

27.5.2.2.25 basic_streambuf::pbackfail [lib.streambuf::pbackfail]

```
virtual int_type pbackfail(int c = eof());
```

1 The public functions of basic_streambuf call this virtual only when gptr() is null, gptr()==eback(), or *gptr()!=c. Other calls shall also satisfy that constraint.

2 The *pending sequence* is defined as for underflow (in 27.5.2.2.27) with the modifications that

— If c==eof() then the input sequence is backed up one character before the pending sequence is determined.

— If c!=eof() then c is prepended. Whether the input sequence is backed up or modified in any other way is unspecified.

3 On return, the constraints of gptr(), eback(), and pptr() are the same as for underflow (in 27.5.2.2.27)

Returns: eof() to indicate failure. Failure may occur because the input sequence could not be backed up, or if for some other reason the pointers could not be set consistent with the constraints. pbackfail is called only when put back has really failed.

---

180) That is, for each class derived from an instance of basic_streambuf in this clause, a specification of how consume a character effects the associated output sequence is given. There is no requirement on a program-defined class.

181) Typically, overflow returns *c* to indicate success.

Returns some value other than `eof()` to indicate success.
Default behavior: Returns `eof()`.

27.5.2.2.26 `basic_streambuf::showmany`[182] [lib.streambuf::showmany]

```
virtual int showmany();
```

Returns: a count of the guaranteed number of characters that can be read from the input sequence before a call to `uflow()`or `underflow()` returns `eof()`. A positive return value of indicates that the next such call will not return `eof()`.[183]
Default behavior: Returns zero.

27.5.2.2.27 `basic_streambuf::underflow` [lib.streambuf::underflow]

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

1 The public members of `basic_streambuf` call this virtual only if `gptr()` is null or `gptr() >= egptr()`
Returns: the first *character* of the *pending sequence*, if possible, without moving the input sequence position past it. If the pending sequence is null then the function fails.

2 The pending sequence of characters is defined as the concatenation of:

a) If `gptr()` is non- `NULL`, then the `egptr() - gptr()` characters starting at `gptr()`, otherwise the empty sequence.

b) Some sequence (possibly empty) of characters read from the input sequence.

3 The *result character* is

a) If the pending sequence is non-empty, the first character of the sequence.

b) If the pending sequence empty then the next character that would be read from the input sequence.

4 The *backup sequence* is defined as the concatenation of:

a) If `eback()` is null then empty,

b) Otherwise the `gptr() - eback()` characters beginning at `eback()`.

5 The function sets up the `gptr()` and `egptr()` satisfying one of:

a) If the pending sequence is non-empty, `egptr()` is non-null and `egptr() - gptr()` characters starting at `gptr()` are the characters in the pending sequence

b) If the pending sequence is empty, either `gptr()` is null or `gptr()` and `egptr()` are set to the same non- `NULL` pointer.

6 If `eback()` and `gptr()` are non-null then the function is not constrained as to their contents, but the ‘‘usual backup condition’’ is that either:

a) If the backup sequence contains at least `gptr() - eback()` characters, then the `gptr() - eback()` characters starting at `eback()` agree with the last `gptr() - eback()` characters of the backup sequence.

b) Or the *n* characters starting at `gptr() - n` agree with the backup sequence (where *n* is the length of the backup sequence)

182) The morphemes of `showmany` are "es-how-manyS, not "show-many".
183) The next such call might fail by throwing an exception. The intention is that the next call will return ‘‘immediately.’’

Returns: `eof()` to indicate failure.
Default behavior: Returns `eof()`.

27.5.2.2.28 `basic_streambuf::uflow` [lib.streambuf::uflow]

```
virtual int_type uflow();
```

1 The constraints are the same as for `underflow` (27.5.2.2.27) except that the result character is transfered from the pending sequence to the backup sequence, and the pending sequence may not be empty before the transfer.
Default behavior: Calls `underflow(eof())`. If `underflow` returns `eof()`, returns `eof()`. Otherwise, does `gbump(-1)` and returns `*gptr()`.
Returns: `not_eof(c)`.

27.5.2.2.29 `basic_streambuf::xsgetn` [lib.streambuf::xsgetn]

```
virtual streamsize xsgetn(char_type* s, streamsize n);
```

Effects: Assigns up to *n* characters to successive elements of the array whose first element is designated by *s*. The characters assigned are read from the input sequence as if by repeated calls to `sbumpc()`. Assigning stops when either *n* characters have been assigned or a call to `sbumpc()` would return `eof()`.
Returns: The number of characters assigned.[184)]

27.5.2.2.30 `basic_streambuf::xsputn` [lib.streambuf::xsputn]

```
virtual streamsize xsputn(const char_type* s, streamsize n);
```

1 Writes up to *n* characters to the output sequence as if by repeated calls to `sputc(c)`. The characters written are obtained from successive elements of the array whose first element is designated by *s*. Writing stops when either *n* characters have been written or a call to `sputc(c)` would return `eof()`.
Returns: The number of characters written.

27.5.2.2.31 `basic_streambuf::seekoff` [lib.streambuf::seekoff]

```
virtual pos_type seekoff(off_type off,
ic_ios<charT,traits>::seekdir way,
ic_ios<charT,traits>::openmode which
 = basic_ios<charT,traits>::in | basic_ios<charT,traits>::out);
```

Effects: Alters the stream positions within one or more of the controlled sequences in a way that is defined separately for each class derived from `basic_streambuf` in this clause (27.7.1.2.5, 27.8.2.3.8).
Default behavior: Returns an object of class `pos_type` that stores an *invalid stream position* (27.1.1).

27.5.2.2.32 `basic_streambuf::seekpos` [lib.streambuf::seekpos]

```
virtual pos_type seekpos(pos_type sp,
ic_ios<charT,traits>::openmode which = in | out);
```

Effects: Alters the stream positions within one or more of the controlled sequences in a way that is defined separately for each class derived from `basic_streambuf` in this clause (27.7.1.2.6, 27.8.2.3.9).

184) Classes derived from `basic_streambuf` can provide more efficient ways to implement `xsgetn` and `xsputn` by overriding these definitions in the base class.

Default behavior: Returns an object of class `pos_type` that stores an *invalid stream position*.

27.5.2.2.33 `basic_streambuf::setbuf` [lib.streambuf::setbuf]

```
virtual basic_streambuf*
buf(char_type* s, streamsize n);
```

Effects: Performs an operation that is defined separately for each class derived from `basic_streambuf` in this clause (27.7.1.2.7, 27.8.2.3.10).
Default behavior: Returns `this`.

27.5.2.2.34 `basic_streambuf::sync` [lib.streambuf::sync]

```
virtual int sync();
```

Effects: Synchronizes the controlled sequences with the arrays. That is, if `pbase()` is non-null the characters between `pbase()` and `pptr()` are written to the controlled sequence, and if `gptr()` is non-null, the characters between `gptr()` and `egptr()` are restored to the input sequence. The pointers may then be reset as appropriate.
Returns: -1 on failure. What constitutes failure is determined by each derived class (27.8.2.3.11).
Default behavior: Returns zero.

27.6 Formatting and manipulators [lib.iostream.format]

Header `<istream>` synopsis

```
namespace std {
  template <class charT, class traits = ios_traits<charT> >
    class basic_istream;
  typedef basic_istream<char>     istream;
  typedef basic_istream<wchar_t> wistream;

  template<class charT, class traits>
    basic_istream<charT,traits>& ws(basic_istream<charT,traits>& is);
}
```

Header `<ostream>` synopsis

```
namespace std {
  template <class charT, class traits = ioc_traits<charT> >
    class basic_ostream;
  typedef basic_ostream<char>     ostream;
  typedef basic_ostream<wchar_t> wostream;

  template<class charT, class traits>
    basic_ostream<charT,traits>& endl(basic_ostream<charT,traits>& os);
  template<class charT, class traits>
    basic_ostream<charT,traits>& ends(basic_ostream<charT,traits>& os);
  template<class charT, class traits>
    basic_ostream<charT,traits>& flush(basic_ostream<charT,traits>& os);
}
```

Header `<iomanip>` synopsis

```
#include <istream>
#include <ostream>

namespace std {
    typedef ? smanip;

    smanip resetiosflags(ios_base::fmtflags mask);
    smanip setiosflags  (ios_base::fmtflags mask);
    smanip setbase(int base);
    smanip setfill(int c);
    smanip setprecision(int n);
    smanip setw(int n);
}
```

1 The header <iomanip> defines three template classes and several related functions that use these template classes to provide extractors and inserters that alter information maintained by class ios_base and its derived classes. It also defines several instantiations of these template classes and functions.

27.6.1 Input streams [lib.input.streams]

1 The header <istream> defines a type and a function signature that control input from a stream buffer.

27.6.1.1 Template class basic_istream [lib.istream]

```
namespace std {
  template <class charT, class traits = ios_traits<charT> >
  class basic_istream : virtual public basic_ios<charT,traits> {
  public:
    typedef charT            char_type;
    typedef traits::int_type int_type;
    typedef traits::pos_type pos_type;
    typedef traits::off_type off_type;
    int_type  eof()     { return traits::eof(); }
    char_type newline() { return traits::newline(); }

  private:
// for abbreviation:
    typedef basic_istream<char_type,traits> istream_type;
    typedef basic_ios<charT,traits> ios_type;

  public:
    basic_istream(basic_streambuf<charT,traits>* sb);
    virtual ~basic_istream();
    bool ipfx(bool noskipws = 0);
    void isfx();

    istream_type& operator>>(istream_type& (*pf)(istream_type&))
    istream_type& operator>>(ios_type& (*pf)(ios_type&))
    istream_type& operator>>(char_type* s);
    istream_type& operator>>(char_type& c);
```

```
    istream_type& operator>>(bool& n);
    istream_type& operator>>(short& n);
    istream_type& operator>>(unsigned short& n);
    istream_type& operator>>(int& n);
    istream_type& operator>>(unsigned int& n);
    istream_type& operator>>(long& n);
    istream_type& operator>>(unsigned long& n);
    istream_type& operator>>(float& f);
    istream_type& operator>>(double& f);
    istream_type& operator>>(long double& f);
    istream_type& operator>>(void*& p);
    istream_type& operator>>(basic_streambuf<char_type,traits>& sb);

    int_type get();
    istream_type& get(char_type* s, streamsize n,
                      char_type delim = newline());
    istream_type& get(char_type& c);
    istream_type& get(basic_streambuf<char_type,traits>& sb,
                      char_type delim = newline());

    istream_type& getline(char_type* s, streamsize n,
                          char_type delim = newline());

    istream_type& ignore(streamsize n = 1, int_type delim = eof());
    istream_type& read(char_type* s, streamsize n);

    int readsome(char_type* s, int n);
    int peek();
    istream_type& putback(char_type c);
    istream_type& unget();
    streamsize gcount() const;
    int sync();
  };
}
```

1 The class `basic_istream` defines a number of member function signatures that assist in reading and interpreting input from sequences controlled by a stream buffer.

2 Two groups of member function signatures share common properties: the *formatted input functions* (or *extractors*) and the *unformatted input functions.* Both groups of input functions are described as if they obtain (or *extract*) input *characters* by calling *sb*.sbumpc() *or sb.sgetc(). They may use other public members of* `istream` *except that they do not invoke any virtual members of sb except* `uflow()`.

3 If *sb*.sbumpc() *or sb.sgetc() returns* `eof()`, *then the input function, except as explicitly noted otherwise, completes its actions and does* `setstate(eofbit)` *before returning.*

4 If one of these called functions throws an exception, then unless explicitly noted otherwise the input function calls `setstate(badbit)` and if `badbit` is on in `sb.exception()` rethrows the exception without completing its actions.

27.6.1.2 `basic_istream` constructors [lib.basic.istream.cons]

```
basic_istream(basic_streambuf<charT,traits>* sb);
```

Effects: Constructs an object of class `basic_istream`, assigning initial values to the base class by calling `basic_ios::init(sb)`.

Postcondition: `gcount() == 0`

```
virtual ~basic_istream();
```

Effects: Destroys an object of class `basic_istream`.
Notes: Does not perform any operations of `rdbuf()`.

27.6.1.3 Member functions [lib.istream.members]

27.6.1.3.1 `basic_istream::ipfx` [lib.istream::ipfx]

```
bool ipfx(bool noskipws = 0);
```

Effects: If `good()` is `true`, prepares for formatted or unformatted input. First, if `tie()` is not a null pointer, the function calls `tie()->flush()` to synchronize the output sequence with any associated external C stream.[185] If *`noskipws`* is zero and `flags() & skipws` is nonzero, the function extracts and discards each character as long as the next available input character *`c`* is a whitespace character.

Notes: The function `basic_istream<charT,traits>::ipfx()` uses the function `bool traits::is_whitespace(charT, const locale*)` in the `traits` structure to determine whether the next input character is whitespace or not.

1 To decide if the character *`c`* is a whitespace character, the function performs as if it executes the following code fragment:

```
ctype<charT> ctype = getloc().use<ctype<charT> >();
if (traits::is_whitespace (c, ctype)!=0)
// c is a whitespace character.
```

Returns: If, after any preparation is completed, `good()` is `true`, returns `true`. Otherwise, it calls `setstate(failbit)` and returns `false`.[186]

2 A typical implementation of the `ipfx()` function may be as follows:

```
template <class charT, class traits = ios_traits<charT> >
int basic_istream<charT,traits>::ipfx() {
    ...
// skipping whitespace according to a constraint function,
// is_whitespace
    intT c;
    typedef ctype<charT> ctype_type;
    ctype_type& ctype = getloc().use<ctype_type>();
    while ((c = rdbuf()->snextc()) != eof()) {
      if (!traits::is_whitespace (c,ctype)==0) {
        rdbuf()->sputbackc (c);
        break;
      }
    }
    ...
}
```

3 In case we use `ios_traits<char>` or `ios_traits<wchar_t>`, the behavior of the constraint function `traits::is_whitespace()` is as if it invokes:

```
ctype<charT>& ctype = getloc().use<ctype<charT> >();
ctype.is(ctype<charT>::SPACE, c);
```

otherwise, the behavior of the function `traits::is_whitespace()` is unspecified.

---

185) The call `tie()->flush()` does not necessarily occur if the function can determine that no synchronization is necessary.
186) The functions `ipfx(int)` and `isfx()` can also perform additional implementation-dependent operations.

4 Those who want to use locale-independent whitespace predicate can specify their definition of `is_whitespace` in their new `ios_traits` as follows:

```
struct my_traits : public ios_traits<char> {
    typedef my_char_traits char_traits;
};

struct my_char_traits : public ios_traits<char> {
    static bool is_whitespace (char c, const ctype<charT>& ctype) {
    ....(my own implementation)...
    }
};
```

27.6.1.3.2 basic_istream::isfx [lib.istream::isfx]

```
void isfx();
```

Effects:

Box 117
TBS

27.6.1.3.3 basic_istream::sync [lib.istream::sync]

```
int sync();
```

Effects: If `rdbuf()` is a null pointer, returns `eof()`. Otherwise, calls `rdbuf()->pubsync()` and, if that function returns `eof()`, calls `setstate(badbit)` and returns `eof()`. Otherwise, returns zero.

27.6.1.4 Formatted input functions [lib.istream.formatted]

27.6.1.4.1 Common requirements [lib.istream.formatted.reqmts]

1 Each formatted input function begins execution by calling `ipfx()`. If that function returns `true`, the function endeavors to obtain the requested input. In any case, the formatted input function ends by calling `isfx()`, then returns `*this`

2 Some formatted input functions endeavor to obtain the requested input by parsing characters extracted from the input sequence, converting the result to a value of some scalar data type, and storing the converted value in an object of that scalar data type.

3 The numeric conversion behaviors of the following extractors are locale-dependent.

```
operator>>(short& val);
operator>>(unsigned short& val);
operator>>(int& val);
operator>>(unsigned int& val);
operator>>(long& val);
operator>>(unsigned long& val);
operator>>(float& val);
operator>>(double& val);
operator>>(long double& val);
```

As in the case of the inserters, these extractors depend on the locale's `num_get<>` (22.2.4.1) object to perform parsing the input stream data. The conversion occurs as if it performed the following code fragment:

```
HOLDTYPE tmp;
num_get<charT>& fmt = loc.use< num_get<charT> >();
fmt.get (iter, *this, loc, tmp);
if ((val = (TYPE)tmp) != tmp)
// set fail bit...
```

In the above fragment, `loc` stands for the private member of the `basic_ios` class, `TYPE` stands for the type of the argument of the extractor, and `HOLDTYPE` is as follows;

— for `short`, `int` and `long`, `HOLDTYPE` is `long`;

— for `unsigned short`, `unsigned int` and `unsigned long`, `HOLDTYPE` is `unsigned long`.

— for `float`, `double`, `HOLDTYPE` is `double`.

— for `long double`, `HOLDTYPE` is `long double`.

4 The first argument provides an object of the `istream_iterator` class which is an iterator pointed to an input stream. It bypasses istreams and uses streambufs directly. Class `locale` relies on this type as its interface to istream, since the flexibility it has been abstracted away from direct dependence on istream. *

5 In case the converting result is a value of either an integral type (`short`, `unsigned short`, `int`, `unsigned int`, `long`, `unsigned long`) or a float type (`float`, `double`, `long double`), performing to parse and convert the result depend on the imbued `locale` object. So the behavior of the above type extractors are locale-dependent. The imbued `locale` object uses an `istreambuf_iterator` to access the input character sequence.

6 The behavior of such functions is described in terms of the conversion specification for an equivalent call to the function `fscanf()`[187] operating with the global locale set to `getloc()`, with the following alterations:

— The formatted input function extracts characters from a stream buffer, rather than reading them from an input file.[188]

— If `flags() & skipws` is zero, the function does not skip any leading white space. In that case, if the next input character is white space, the scan fails.

— If the converted data value cannot be represented as a value of the specified scalar data type, a scan failure occurs.

Box 118

Can the current `num_put/num_get` facet handle `basefield` specification? Needs more discussion.

7 If the scan fails for any reason, the formatted input function calls `setstate(failbit)`.

8 For conversion to an integral type other than a character type, the function determines the integral conversion specifier as indicated in Table 71: *

187) The signature `fscanf(FILE*, const char*, ...)` is declared in <cstdio> (27.8.3)

188) The stream buffer can, of course, be associated with an input file, but it need not be.

Table 71—Integer conversions

State	`stdio` equivalent
`(flags() & basefield) == oct`	`%o`
`(flags() & basefield) == hex`	`%x`
`(flags() & uppercase) != 0`	`%X`
`(flags() & basefield) == 0`	`%i`
`Otherwise,`	
`signed` integral type	`%d`
`unsigned` integral type	`%u`

Box 119

Is this table clear with regards to %x vs. %X?

27.6.1.4.2 `basic_istream::operator>>` [lib.istream::extractors]

```
istream_type& operator>>(istream_type& (*pf)(istream_type&))
```

Returns: `pf(*this)`.[189)]

```
istream_type& operator>>(ios_type& (*pf)(ios_type&))
```

Effects: Calls `pf(*this)`, then returns `*this`.[190)]

```
istream_type& operator>>(char_type* s);
```

Effects: Extracts characters and stores them into successive locations of an array whose first element is designated by *s*.[191)] If `width()` is greater than zero, the maximum number of characters stored *n* is `width()`; otherwise it is `numeric_limits<int>::max()` (18.2.1).

1 Characters are extracted and stored until any of the following occurs:

— *n* `- 1` characters are stored;

— end-of-file occurs on the input sequence;

— `traits::is_whitespace(`*c,ctype*`)` is nonzero for the next available input character *c*. In the above code fragment, the argument *ctype* is acquired by `getloc().use<ctype<charT> >()`.

2 If the function stores no characters, it calls `setstate(failbit)`. In any case, it then stores a null character into the next successive location of the array and calls `width(0)`.
Returns: `*this`.

```
istream_type& operator>>(char_type& c);
```

Effects: Extracts a character, if one is available, and stores it in *c*. Otherwise, the function calls `setstate(failbit)`.
Returns: `*this`.

189) See, for example, the function signature `ws(basic_istream&)` (27.6.1.6).
190) See, for example, the function signature `dec(basic_ios<charT,traits>&)` (_lib.basefiled.manip_).
191) Note that this function is not overloaded on types `signed char` and `unsigned char`.

```
istream_type& operator>>(bool& n);
```

Effects: Converts a signed short integer, if one is available, and stores it in *x*.
Returns: `*this`.
Notes: Behaves as if:

```
if (flags() & ios::boolalpha) {
  getloc().extract(*this, n);
} else {
  int x;
  *this >> x;
  if (x == 0)
    n = false;
  else if (x == 1)
    n = true;
  else
    ; // indicate failure
}
return *this;
```

3 Locale extraction (`getloc().extract()`) of the string is something like:

```
istream i;
string bool_false = ...; // locale dependent
string bool_true  = ...;
string s;
i >> s;
if (s == bool_false)
  n = false;
else if (s == bool_true)
  n = true;
else
  ; // indicate failure
```

4 The strings for the default locale are `false` and `true`.

```
istream_type& operator>>(short& n);
```

Effects: Converts a signed short integer, if one is available, and stores it in *n*.
Returns: `*this`.

```
istream_type& operator>>(unsigned short& n);
```

Effects: Converts an unsigned short integer, if one is available, and stores it in *n*.
Returns: `*this`.

```
istream_type& operator>>(int& n);
```

Effects: Converts a signed integer, if one is available, and stores it in *n*.
Returns: `*this`.

```
istream_type& operator>>(unsigned int& n);
```

Effects: Converts an unsigned integer, if one is available, and stores it in *n*.
Returns: `*this`.

```
istream_type& operator>>(long& n);
```

Effects: Converts a signed long integer, if one is available, and stores it in *n*.

Returns: *this.

```
istream_type& operator>>(unsigned long& n);
```

Effects: Converts an unsigned long integer, if one is available, and stores it in *n*.
Returns: *this.

```
istream_type& operator>>(float& f);
```

Effects: Converts a float, if one is available, and stores it in *f*.
Returns: *this.

```
istream_type& operator>>(double& f);
```

Effects: Converts a double, if one is available, and stores it in *f*.
Returns: *this.

```
istream_type& operator>>(long double& f);
```

Effects: Converts a long double, if one is available, and stores it in *f*.
Returns: *this.

```
istream_type& operator>>(void*& p);
```

Effects: Converts a pointer to void, if one is available, and stores it in *p*.
Returns: *this.

```
istream_type& operator>>(basic_streambuf<charT,traits>& sb);
```

Effects: Extracts characters from *this and inserts them in the output sequence controlled by rdbuf(). Characters are extracted and inserted until any of the following occurs:

— end-of-file occurs on the input sequence;

— inserting in the output sequence fails (in which case the character to be inserted is not extracted);

— an exception occurs (in which case the exception is caught). setstate(badbit) is not called

5 If the function inserts no characters, it calls setstate(failbit). If failure was due to catching an exception thrown while extracting characters from rdbuf() and failbit is on in exceptions() (27.4.3.2.11), then the caught exception is rethrown.
Returns: *this.

27.6.1.5 Unformatted input functions [lib.istream.unformatted]

1 Each unformatted input function begins execution by calling ipfx(1). If that function returns nonzero, the function endeavors to extract the requested input. It also counts the number of characters extracted. In any case, the unformatted input function ends by storing the count in a member object and calling isfx(), then returning the value specified for the unformatted input function.

27.6.1.5.1 basic_istream::get [lib.istream::get]

```
int get();
```

Effects: Extracts a character *c*, if one is available. The function then returns (unsigned char)*c*. Otherwise, the function calls setstate(failbit) and then returns eof().

```
istream_type& get(char_type*  s, streamsize n,
          char_type delim = newline());
```

Effects: Extracts characters and stores them into successive locations of an array whose first element is designated by *s*.[192] Characters are extracted and stored until any of the following occurs:

— *n* - 1 characters are stored;

— end-of-file occurs on the input sequence (in which case the function calls `setstate(eofbit)`);

— *c* == *delim* for the next available input character *c* (in which case *c* is not extracted).

1 If the function stores no characters, it calls `setstate(failbit)`. In any case, it then stores a null character into the next successive location of the array.
Returns: `*this`.

```
istream_type& get(char_type& c);
```

Effects: Extracts a character, if one is available, and assigns it to *c*.[193] Otherwise, the function calls `setstate(failbit)`.
Returns: `*this`.

```
istream_type& get(basic_streambuf<char_type,traits>& sb,
                  char_type delim = newline());
```

Effects: Extracts characters and inserts them in the output sequence controlled by `rdbuf()`. Characters are extracted and inserted until any of the following occurs:

— end-of-file occurs on the input sequence;

— inserting in the output sequence fails (in which case the character to be inserted is not extracted);

— *c == delim* for the next available input character *c* (in which case *c* is not extracted);

— an exception occurs (in which case, the exception is caught but not rethrown).

2 If the function inserts no characters, it calls `setstate(failbit)`.
Returns: `*this`.

27.6.1.5.2 basic_istream::getline [lib.istream::getline]

```
istream_type& getline(char_type* s, streamsize n,
                      char_type delim = newline());
```

Effects: Extracts characters and stores them into successive locations of an array whose first element is designated by *s*.[194] Characters are extracted and stored until one of the following occurs:

1) end-of-file occurs on the input sequence (in which case the function calls `setstate(eofbit)`);

2) *c* == *delim* for the next available input character *c* (in which case the input character is extracted but not stored);[195]

3) *n* - 1 characters are stored (in which case the function calls `setstate(failbit)`).

1 These conditions are tested in the order shown.[196]

2 If the function extracts no characters, it calls `setstate(failbit)`.[197]

---

192) Note that this function is not overloaded on types `signed char` and `unsigned char`.
193) Note that this function is not overloaded on types `signed char` and `unsigned char`.
194) Note that this function is not overloaded on types `signed char` and `unsigned char`.
195) Since the final input character is ''extracted,'' it is counted in the `gcount()`, even though it is not stored.
196) This allows an input line which exactly fills the buffer, without setting `failbit`. This is different behavior than the historical AT&T implementation.
197) This implies an empty input line will not cause `failbit` to be set.

3 In any case, it then stores a null character into the next successive location of the array.
Returns: `*this`.

4 Example:

```
#include <iostream>
using namespace std;
const int line_buffer_size = 100;
int main()
{
  char buffer[line_buffer_size];
  int line_number = 0;
  while (cin.getline(buffer, line_buffer_size) || cin.gcount()) {
    int count = cin.gcount();
    if (cin.eof())
      cout << "Partial final line";   // cin.fail() is false
    else if (cin.fail()) {
      cout << "Partial long line";
      cin.clear(cin.rdstate() & ~ios::failbit);
    } else {
      count--;        // Don't include '\n' in count
      cout << "Line " << ++line_number;
    }
    cout << " (" << count << " chars): " << buffer << endl;
  }
}
```

27.6.1.5.3 `basic_istream::ignore` [lib.istream::ignore]

`istream_type& ignore(int` *n* `= 1, int_type` *delim* `= eof());`

Effects: Extracts characters and discards them. Characters are extracted until any of the following occurs:

— if *n* `!= numeric_limits<int>::max()` (18.2.1), *n* characters are extracted

— end-of-file occurs on the input sequence (in which case the function calls `setstate(eofbit)`);

— *c* `== delim` for the next available input character *c* (in which case *c* is extracted).

1 The last condition will never occur if *delim* `== eof()`.
Returns: `*this`.

27.6.1.5.4 `basic_istream::read` [lib.istream::read]

`istream_type& read(char_type*` *s*`, streamsize` *n*`);`

Effects: Extracts characters and stores them into successive locations of an array whose first element is designated by *s*.[198] Characters are extracted and stored until either of the following occurs:

— *n* characters are stored;

— end-of-file occurs on the input sequence (in which case the function calls `setstate(failbit)`).
Returns: `*this`.

[198] Note that this function is not overloaded on types `signed char` and `unsigned char`.

27.6.1.5.5 `basic_istream::readsome` [lib.istream::readsome]

```
int readsome(char_type* s, int n);
```

Effects: Extracts characters and stores them into successive locations of an array whose first element is designated by *s*. The function first determines *navail*, the value returned by calling `in_avail()`. If *navail* is `1`, the function calls `setstate(eofbit)` and returns zero.

1 Otherwise, the function determines the number of characters to extract *m* as the smaller of *n* and *navail*, and returns `read(`*s*`, `*m*`)`.

27.6.1.5.6 `basic_istream::peek` [lib.istream::peek]

```
int peek();
```

Returns: `eof()` if `good()` is false. Otherwise, returns `rdbuf()->sgetc()`.

27.6.1.5.7 `basic_istream::putback` [lib.istream::putback]

```
istream_type& putback(char_type c);
```

Effects: Calls `rdbuf->sputbackc(`*c*`)`. If that function returns `eof()`, calls `setstate(badbit)`.
Returns: `*this`.

27.6.1.5.8 `basic_istream::unget` [lib.istream::unget]

```
istream_type& unget();
```

Effects: Calls `rdbuf->sungetc()`. If that function returns `eof()`, calls `setstate(badbit)`.
Returns: `*this`.

27.6.1.5.9 `basic_istream::gcount` [lib.istream::gcount]

```
streamsize gcount() const;
```

Returns: The number of characters extracted by the last unformatted input member function called for the object.

27.6.1.6 Standard `basic_istream` manipulators [lib.basic.istream.manip]

```
namespace std {
  template<class charT, class traits>
    basic_istream<charT,traits>& ws(basic_istream<charT,traits>& is);
}
```

Effects: Saves a copy of *is*`.fmtflags`, then clears *is* `.skipws` in *is* `.flags()`. *Then calls is* `.ipfx()` *and is* `.isfx()`, *and restores is* `.flags()` *to its saved value.*[199] *

Returns: *is*.

[199] The effect of `cin >> ws` is to skip any white space in the input sequence controlled by `cin`.

27.6.2 Output streams [lib.output.streams]

1 The header <ostream> defines a type and several function signatures that control output to a stream buffer.

27.6.2.1 Template class basic_ostream [lib.ostream]

```
namespace std {
  template <class charT, class traits = ioc_traits<charT> >
  class basic_ostream : virtual public basic_ios<charT,traits> {
  public:
    typedef charT            char_type;
    typedef traits::int_type int_type;
    typedef traits::pos_type pos_type;
    typedef traits::off_type off_type;
    int_type  eof()     { return traits::eof(); }
    char_type newline() { return traits::newline(); }
  private:
    typedef basic_ostream<charT,traits> ostream_type;
  public:
    basic_ostream(basic_streambuf<char_type,traits>* sb);
    virtual ~basic_ostream();
    bool opfx();
    void osfx();

    ostream_type& operator<<(ostream_type& (*pf)(ostream_type&));
    ostream_type& operator<<(ios_type& (*pf)(ios_type&));
    ostream_type& operator<<(const char_type* s);
    ostream_type& operator<<(char_type c);

    ostream_type& operator<<(bool n);
    ostream_type& operator<<(short n);
    ostream_type& operator<<(unsigned short n);
    ostream_type& operator<<(int n);
    ostream_type& operator<<(unsigned int n);
    ostream_type& operator<<(long n);
    ostream_type& operator<<(unsigned long n);
    ostream_type& operator<<(float f);
    ostream_type& operator<<(double f);
    ostream_type& operator<<(long double f);
    ostream_type& operator<<(void* p);
    ostream_type& operator<<(basic_streambuf<char_type,traits>& sb);

    int put(char_type c);
    ostream_type& write(const char_type* s, streamsize n);

    ostream_type& flush();
  };
}
```

1 The class basic_ostream defines a number of member function signatures that assist in formatting and writing output to output sequences controlled by a stream buffer.

2 Two groups of member function signatures share common properties: the *formatted output functions* (or *inserters*) and the *unformatted output functions.* Both groups of output functions generate (or *insert*) output *characters* by actions equivalent to calling rdbuf().sputc(int). They may use other public members of basic_ostream except that they do not invoke any tuals members of rdbuf() except overflow. If the called function throws an exception, the output function calls setstate(badbit) and if badbit is on in exceptions() rethrows the exception.

27.6.2.2 basic_ostream constructors [lib.basic.ostream.sb.cons]

```
basic_ostream(basic_streambuf<charT,traits>* sb);
```

Effects: Constructs an object of class `basic_ostream`, assigning initial values to the base class by calling `basic_ios<charT,traits>::init(sb)`.

```
virtual ~basic_ostream();
```

Effects: Destroys an object of class `basic_ostream`.
Notes: Does not perform any operations on `rdbuf()`.

27.6.2.3 Member functions [lib.basic.ostream.members]

27.6.2.3.1 basic_ostream::opfx [lib.ostream::opfx]

```
bool opfx();
```

1 If `good()` is nonzero, prepares for formatted or unformatted output. If `tie()` is not a null pointer, calls `tie()->flush()`.[200)]

Box 120
Note: Need to append the `locale` dependency on appropriate extractors.

Returns: `good()`.[201)]

27.6.2.3.2 basic_ostream::osfx [lib.ostream::osfx]

```
void osfx();
```

1 If `flags() & unitbuf` is nonzero, calls `flush()`.

27.6.2.3.3 basic_ostream::flush [lib.ostream::flush]

```
basic_ostream& flush();
```

1 If `rdbuf()` is not a null pointer, calls `rdbuf()->pubsync()`. If that function returns `eof()`, calls `setstate(badbit)`.
Returns: `*this`.

27.6.2.4 Formatted output functions [lib.ostream.formatted]

27.6.2.4.1 Common requirements [lib.ostream.formatted.reqmts]

1 Each formatted output function begins execution by calling `opfx()`. If that function returns nonzero, the function endeavors to generate the requested output. In any case, the formatted output function ends by calling `osfx()`, then returning the value specified for the formatted output function.

2 The numeric conversion behaviors of the following inserters are locale-dependent:

---

200) The call `tie()->flush()` does not necessarily occur if the function can determine that no synchronization is necessary.
201) The function signatures `opfx()` and `osfx()` can also perform additional implementation-dependent operations.

```
operator<<(short val);
operator<<(unsigned short val);
operator<<(int val);
operator<<(unsigned int val);
operator<<(long val);
operator<<(unsigned long val);
operator<<(float val);
operator<<(double val);
operator<<(long double val);
```

3 The classes `num_get<>` and `num_put<>` handle locale-dependent numeric formatting and parsing. The above inserter functions refers the imbued `locale` value to utilize these numeric formatting functionality. The formatting conversion occurs as if it performed the following code fragment:

```
num_put<charT>& fmt = loc.use< num_put<charT> >();
fmt.put (ostreambuf_iterator(*this), *this, loc, val);
```

In the above fragment, `loc` stands for the private member of the `basic_ios` class which maintains the imbued `locale`object. The first argument provides an object of the `ostreambuf_iterator` class which is an iterator for ostream class. It bypasses ostreams and uses streambufs directly. Class `locale` relies on these types as its interface to iostreams, since for flexibility it has been abstracted away from direct dependence on ostream.

Box 120

Note: Need to append the `locale` dependency on appropriate extractors.

4 Some formatted output functions endeavor to generate the requested output by converting a value from some scalar or NTBS type to text form and inserting the converted text in the output sequence.

Box 121

Needs work: NTBS.

The behavior of such functions is described in terms of the conversion specification for an equivalent call to the function `fprintf`,[202] operating with the global locale set to `getloc()`, with the following alterations:

— The formatted output function inserts *characters* in a stream buffer, rather than writing them to an output file.[203]

— The formatted output function uses the fill character returned by `fill()` as the padding character (rather than the space character for left or right padding, or `0` for internal padding).

5 If the operation fails for any reason, the formatted output function calls `setstate(badbit)`.

6 For conversion from an integral type other than a character type, the function determines the integral conversion specifier as indicated in Table 72:

202) The signature `fprintf(FILE*, const char_type*, ...)` is declared in `<cstdio>` (27.8.3).

203) The stream buffer can, of course, be associated with an output file, but it need not be.

Table 72—Integer conversions

State	`stdio` equivalent
`(flags() & basefield) == oct`	`%o`
`(flags() & basefield) == hex`	`%x`
`(flags() & uppercase) != 0`	`%X`
`Otherwise,`	
`signed` integral type	`%d`
`unsigned` integral type	`%u`

Box 122

Is this table clear with regards to `%x` vs. `%X`?

7 For conversion from a floating-point type, the function determines the floating-point conversion specifier as indicated in Table 73:

Box 123

Can the current `num_put/num_get` facet handle `basefield` specification? Needs more discussion.

Table 73—Floating-point conversions

State	`stdio` equivalent
`(flags() & floatfield) == fixed`	`%f`
`(flags() & floatfield) == scientific`	`%e`
`(flags() & uppercase) != 0`	`%E`
`Otherwise,`	
	`%g`
`(flags() & uppercase) != 0`	`%G`

Box 124

Is this table clear with regards to `%e` vs. `%E`?

8 The conversion specifier has the following additional qualifiers prepended as indicated in Table 74:

Table 74—Floating-point conversions

Type(s)	State	`stdio` equivalent
an integral type other than a character type	`(flags() & showpos) != 0`	`+`
	`(flags() & showbase) != 0`	`#`
a floating-point type	`(flags() & showpos) != 0`	`+`
	`(flags() & showpoint) != 0`	`#`

— For any conversion, if `width()` is nonzero, then a field width is specified in the conversion specification. The value is `width()`.

— For conversion from a floating-point type, if `flags() & fixed` is nonzero or if `precision()` is

greater than zero, then a precision is specified in the conversion specification. The value is `precision()`.

9 Moreover, for any conversion, padding with the fill character returned by `fill()` behaves as follows:

— If `(flags() & adjustfield) == right`, no flag is prepended to the conversion specification, indicating right justification (any padding occurs before the converted text). A fill character occurs wherever `fprintf` generates a space character as padding.

— If `(flags() & adjustfield) == internal`, the flag `0` is prepended to the conversion specification, indicating internal justification (any padding occurs within the converted text). A fill character occurs wherever `fprintf` generates a `0` as padding.[204)]

10 Otherwise, the flag `-` is prepended to the conversion specification, indicating left justification (any padding occurs after the converted text). A fill character occurs wherever `fprintf()` would generate a space character as padding.

11 Unless explicitly stated otherwise for a particular inserter, each formatted output function calls `width(0)` after determining the field width.

27.6.2.4.2 `basic_ostream::operator<<` [lib.ostream.inserters]

```
ostream_type& operator<<(ostream_type& (*pf)(ostream_type&))
```

Returns: *pf*`(*this)`.[205)]

```
ostream_type& operator<<(ios_type& (*pf)(ios_type&))
```

Effects: Calls `(*(basic_ios<charT,traits>*)`*pf*`)(*this)`.
Returns: `*this`.[206)]

```
ostream_type& operator<<(const char_type* s);
```

Effects: Converts the NTBS *s* with the conversion specifier `s`.
Returns: `*this`.

```
ostream_type& operator<<(char_type c);
```

Effects: Converts the `char_type` *c* with the conversion specifier `c` and a field width of zero.[207)] The stored field width (`basic_ios<charT,traits>::width()`) is *not* set to zero.
Returns: `*this`.

```
ostream_type& operator<<(bool n);
```

1 Behaves as if:

```
{
  if (flags() & ios::boolalpha) {
    getloc().insert(*this, n);
  } else {
    *this << int(n);
  }
}
```

---

204) The conversion specification `#o` generates a leading `0` which is *not* a padding character.
205) See, for example, the function signature `endl(basic_ostream&)` (27.6.2.6) .
206) See, for example, the function signature `dec(ios_base&)` (27.4.5.3).
207) Note that this function is not overloaded on types `signed char` and `unsigned char`.

Returns: `*this`.

```
ostream_type& operator<<(short n);
```

Effects: Converts the signed short integer *n* with the integral conversion specifier preceded by `h`.
Returns: `*this`.

```
ostream_type& operator<<(unsigned short n);
```

Effects: Converts the unsigned short integer *n* with the integral conversion specifier preceded by `h`.
Returns: `*this`.

```
ostream_type& operator<<(int n);
```

Effects: Converts the signed integer *n* with the integral conversion specifier.
Returns: `*this`.

```
ostream_type& operator<<(unsigned int n);
```

Effects: Converts the unsigned integer *n* with the integral conversion specifier.
Returns: `*this`.

```
ostream_type& operator<<(long n);
```

Effects: Converts the signed long integer *n* with the integral conversion specifier preceded by `l`.
Returns: `*this`.

```
ostream_type& operator<<(unsigned long n);
```

Effects: Converts the unsigned long integer *n* with the integral conversion specifier preceded by `l`.
Returns: `*this`.

```
ostream_type& operator<<(float f);
```

Effects: Converts the `float` *f* with the floating-point conversion specifier.
Returns: `*this`.

```
ostream_type& operator<<(double f);
```

Effects: Converts the `double` *f* with the floating-point conversion specifier.
Returns: `*this`.

```
ostream_type& operator<<(long double f);
```

Effects: Converts the `long double` *f* with the floating-point conversion specifier preceded by `L`.
Returns: `*this`.

```
ostream_type& operator<<(void* p);
```

Effects: Converts the pointer to `void` *p* with the conversion specifier `p`.
Returns: `*this`.

```
ostream_type& operator<<(basic_streambuf<charT,traits>& sb);
```

2 Gets characters from `rdbuf()` and inserts them in `*this`. Characters are read from `rdbuf()` and inserted until any of the following occurs:

— end-of-file occurs on the input sequence;

— inserting in the output sequence fails (in which case the character to be inserted is not extracted); LI an exception occurs while getting a character from `rdbuf()` (in which case, the exception is rethrown).

3 If the function inserts no characters or if it stopped because an exception was thrown while extracting a character, it calls `setstate(failbit)`. If an exception was thrown while extracting a character and `failbit` is on in `exceptions()` the caught exception is rethrown.
Returns: `*this`.

27.6.2.5 Unformatted output functions [lib.ostream.unformatted]

1 Each unformatted output function begins execution by calling `opfx()`. If that function returns nonzero, the function endeavors to generate the requested output. In any case, the unformatted output function ends by calling `osfx()`, then returning the value specified for the unformatted output function.

27.6.2.5.1 `basic_ostream::put` [lib.ostream::put]

```
int put(char_type c);
```

Effects: Inserts the character *c*, if possible.[208] Then returns `(unsigned char)`*c*.

1 Otherwise, calls `setstate(badbit)` and returns `eof()`.

27.6.2.5.2 `basic_ostream::write` [lib.ostream::write]

```
basic_ostream& write(const char_type* s, streamsize n);
```

Effects: Obtains characters to insert from successive locations of an array whose first element is designated by *s*.[209] Characters are inserted until either of the following occurs:

— *n* characters are inserted;

— inserting in the output sequence fails (in which case the function calls `setstate(badbit)`).
Returns: `*this`.

27.6.2.6 Standard `basic_ostream` manipulators [lib.basic.ostream.manip]

27.6.2.6.1 `endl` [lib.endl]

```
namespace std {
  template<class charT, class traits>
    basic_ostream<charT,traits>& endl(basic_ostream<charT,traits>& os);
}
```

Effects: Calls *os*`.put(traits::newline())`, then *os*`.flush()`.
Returns: *os*.[210]

27.6.2.6.2 `ends` [lib.ends]

---

208) Note that this function is not overloaded on types `signed char` and `unsigned char`.
209) Note that this function is not overloaded on types `signed char` and `unsigned char`.
210) The effect of executing `cout << endl` is to insert a newline character in the output sequence controlled by `cout`, then synchronize it with any external file with which it might be associated.

```
namespce std {
  template<class charT, class traits>
    basic_ostream<charT,traits>& ends(basic_ostream<charT,traits>& os);
}
```

Effects: Calls *os*.put(traits::eos()).
Returns: *os*.[211)]

27.6.2.6.3 flush [lib.flush]

```
namespace std {
  template<class charT, class traits>
    basic_ostream<charT,traits>& flush(basic_ostream<charT,traits>& os);
}
```

Effects: Calls *os*.flush().
Returns: *os*.

27.6.3 Standard manipulators [lib.std.manip]

27.6.3.1 Type smanip [lib.smanip]

1 The type ***smanip*** is an implementation-defined function type (8.3.5) returned by the standard manipulators.

27.6.3.2 resetiosflags [lib.resetiosflags]

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

Returns: ***smanip***(*f*, *mask*)*, where f can be defined as:*[212)]

```
template<class charT, class traits>
  ios_base& f(ios_base& str,
              ios_base::fmtflags mask)
  { // reset specified flags
    str.setf(ios_base::fmtflags(0), mask);
    return str;
  }
```

27.6.3.3 setiosflags [lib.setiosflags]

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

Returns: ***smanip***(*f*,*mask*)*, where f can be defined as:*

```
  ios_base& f(ios_base& str,
              ios_base::fmtflags mask)
  { // set specified flags
    str.setf(mask);
    return str;
  }
```

[211)] The effect of executing *ostr* << ends is to insert a null character in the output sequence controlled by *ostr*. If *ostr* is an object of class basic_strstreambuf, the null character can terminate an NTBS constructed in an array object.

[212)] The expression cin >> resetiosflags(ios_base::skipws) clears ios_base::skipws in the format flags stored in the istream object cin (the same as cin >> noskipws), and the expression cout << resetiosflags(ios_base::showbase) clears ios_base::showbase in the format flags stored in the ostream object cout (the same as cout << noshowbase).

27.6.3.4 `setbase` [lib.setbase]

smanip `setbase(int base);`

Returns: ***smanip***(*f*, *base*)*, where f can be defined as:*

```
ios_base& f(ios_base& str, int base)
{ // set basefield
  str.setf(n ==  8 ? ios_base::oct :
                     n == 10 ? ios_base::dec :
                     n == 16 ? ios_base::hex :
                     ios_base::fmtflags(0),
                     ios_base::basefield);
  return str;
}
```

27.6.3.5 `setfill` [lib.setfill]

smanip `setfill(int c);`

Returns: ***smanip***(*f*, *c*)*, where f can be defined as:*

```
ios_base& f(ios_base& str, int c)
{ // set fill character
  str.fill(c);
  return str;
}
```

27.6.3.6 `setprecision` [lib.setprecision]

smanip `setprecision(int n);`

Returns: ***smanip***(*f*, *n*)*, where f can be defined as:*

```
ios_base& f(ios_base& str, int n)
{ // set precision
  str.precision(n);
  return str;
}
```

27.6.3.7 `setw` [lib.setw]

smanip `setw(int n);`

Returns: ***smanip***(*f*, *n*)*, where f can be defined as:*

```
ios_base& f(ios_base& str, int n)
{ // set width
  str.width(n);
  return str;
}
```

27.7 String-based streams [lib.string.streams]

Header `<sstream>` synopsis

```
#include <streambuf>
#include <istream>
#include <ostream>

namespace std {
  template <class charT, class traits = int_charT_traits<charT> >
    class basic_stringbuf;
  typedef basic_stringbuf<char>     stringbuf;
  typedef basic_stringbuf<wchar_t> wstringbuf;

  template <class charT, class traits = ios_traits<charT> >
    class basic_istringstream;
  typedef basic_istringstream<char>     istringstream;
  typedef basic_istringstream<wchar_t> wistringstream;

  template <class charT, class traits = ios_traits<charT> >
    class basic_ostringstream;
  typedef basic_ostringstream<char>     ostringstream;
  typedef basic_ostringstream<wchar_t> wostringstream;
}
```

Table 74—Header `<cstdlib>` synopsis

Type	Name(s)
Functions:	
	atoi strtod
	atol strtol

1 *SEE ALSO:* ISO C subclause 7.10.1.

2 The header `<sstream>` defines three types that associate stream buffers with objects of class `string`, as described in clause 21.1.2.

27.7.1 Template class `basic_stringbuf` [lib.stringbuf]

```
namespace std {
  template <class charT, class traits = int_charT_traits<charT> >
  class basic_stringbuf : public basic_streambuf<charT,traits> {
  public:
    typedef charT            char_type;
    typedef traits::int_type int_type;
    typedef traits::pos_type pos_type;
    typedef traits::off_type off_type;
    int_type  eof()     { return traits::eof(); }
    char_type newline() { return traits::newline(); }

  public:
    basic_stringbuf(ios_base::openmode which
                        = ios_base::in
                        | ios_base::out);
    basic_stringbuf(const basic_string<char_type>& str,
                    ios_base::openmode which
                        = ios_base::in
                        | ios_base::out);
    virtual ~basic_stringbuf();
    basic_string<char_type> str() const;
    void str(const basic_string<char_type>& s);
```

```
  protected:
//  virtual int_type   overflow (int_type c = eof());   inherited
//  virtual int_type   pbackfail(int_type c = eof());   inherited
//  virtual int        showmany();      inherited
// virtual int_type   underflow();   inherited
// virtual int_type   uflow();               inherited
// virtual streamsize xsgetn(char_type* s, streamsize n);  inherited
//  virtual streamsize xsputn(const char_type* s, streamsize n);        inherited

//  virtual pos_type   seekoff(off_type off,
//                             ios_base::seekdir way,
//                             ios_base::openmode which
//                               = ios_base::in
//                               | ios_base::out);      inherited
//  virtual pos_type   seekpos(pos_type sp,
//                             ios_base::openmode which
//                               = ios_base::in
//                               | ios_base::out);      inherited
//  virtual basic_streambuf<char_type,traits>*
//                     setbuf(char_type* s, streamsize n);      inherited
//  virtual int sync();         inherited
 private:
// ios_base::openmode mode;     exposition only
  };
}
```

Box 125

Note: same charT type string can be fed.

1 The class basic_stringbuf is derived from basic_streambuf to associate possibly the input sequence and possibly the output sequence with a sequence of arbitrary *characters*. The sequence can be initialized from, or made available as, an object of class basic_string.

Box 126

For the sake of exposition, the maintained data is presented here as:

— ios_base::openmode *mode*, has in set if the input sequence can be read, and out set if the output sequence can be written.

2 For the sake of exposition, the stored character sequence is described here as an array object.

Box 127

The descriptions of the virtuals in this class need to be brought into agreement with the new descriptions of the generic protocols. As part of that we have

```
//
//  string              data ;
```

exposition only

27.7.1.1 `basic_stringbuf` constructors [lib.basic.stringbuf.cons]

```
basic_stringbuf(ios_base::openmode which = ios_base::in
        ios_base::out);
```

Effects: Constructs an object of class `basic_stringbuf`, initializing the base class with `basic_streambuf()`, and initializing *mode* with *which*. The function allocates no array object.

```
basic_stringbuf(const basic_string<char_type>& str,
        ios_base::openmode which = ios_base::in
        ios_base::out);
```

Effects: Constructs an object of class `basic_stringbuf`, initializing the base class with `basic_streambuf()`, and initializing *mode* with *which*.

Notes: If *str*`.length()` is nonzero, the function allocates an array object *x* whose length *n* is *str*`.length()` and whose elements *x*`[`*I*`]` are initialized to *str*`[`*I*`]`. If *which* `& basic_ios::in` is nonzero, the function executes: `setg(`*x*`,`*x*`,`*x*`+`*n*`);`
If *which* `& basic_ios::out` is nonzero, the function executes: `setp(`*x*`,`*x*`+`*n*`);`

27.7.1.2 Member functions [lib.stringbuf.members]

27.7.1.2.1 `basic_stringbuf::str` [lib.stringbuf::str]

```
basic_string<char_type> str() const;
```

Returns: The return value of this function are indicated in Table 75:

Table 75—`str` return values

Condition	Return Value
(*mode* `& basic_ios::in) != 0` and `(gptr() != 0)`	`basic_string<char_type>(eback(),egptr() - eback())`
(*mode* `& basic_ios::out) != 0` and `(pptr() != 0`	`basic_string<char_type>(pbase(),pptr - pbase())`
Otherwise	`basic_string<char_type>()`

```
void str(const basic_string<char_type>& s);
```

Effects: If *s*.length() *is zero, executes:*

```
setg(0, 0, 0);
setp(0, 0);
```

and frees storage for any associated array object. Otherwise, the function allocates an array object *x* whose length *n* is *str*`.length()` and whose elements *x*`[`*I*`]` are initialized to *str*`[`*I*`]`. If *which* `& ios_base::in != 0`, the function executes: `setg(`*x*`,`*x*`,`*x*`+`*n*`);`

```
setg(0, 0, 0);
setp(0, 0);
```

If *which* `& ios_base::out != 0`, the function executes: `setp(`*x*`,`*x*`+`*n*`);`

27.7.1.2.2 basic_stringbuf::overflow [lib.stringbuf::overflow]

```
// virtual int_type overflow(int_type c = eof());
```
inherited

Effects: Appends the character designated by *c* to the output sequence, if possible, in one of two ways:

— If *c* != eof() and if either the output sequence has a write position available or the function makes a write position available (as described below), the function calls sputc(*c*). Signals success by returning *c*.

— If *c* == eof(), there is no character to append. Signals success by returning a value other than eof().

Notes: The function can alter the number of write positions available as a result of any call.
Returns: eof() to indicate failure.

1 The function can make a write position available only if (*mode* & ios_base::out) != 0. To make a write position available, the function reallocates (or initially allocates) an array object with a sufficient number of elements to hold the current array object (if any), plus one additional write position. If (*mode* & ios_base::in) != 0, the function alters the read end pointer egptr() to point just past the new write position (as does the write end pointer epptr()).

27.7.1.2.3 basic_stringbuf::pbackfail [lib.stringbuf::pbackfail]

```
// virtual int_type pbackfail(int_type c = eof());
```
inherited

Box 128
Check vs. 27.5.2.2.25 and 27.8.2.3.5

Effects: Puts back the character designated by *c* to the input sequence, if possible, in one of three ways:

— If *c* != eof(), if the input sequence has a putback position available, and if char_type(*c*) == char_type(gptr()[-1]), assigns gptr() - 1 to gptr(). Returns: *c*.

— If *c* != eof(), if the input sequence has a putback position available, and if *mode* & *ios_base::out* is nonzero, assigns *c* to *--gptr(). Returns: char_type(*c*).

— If *c* == eof() and if the input sequence has a putback position available, assigns gptr() - 1 to gptr(). Returns: char_type(*c*).

Returns: eof() to indicate failure.
Notes: If the function can succeed in more than one of these ways, it is unspecified which way is chosen. *

27.7.1.2.4 basic_stringbuf::underflow [lib.stringbuf::underflow]

```
// virtual int_type underflow();
```
inherited

Returns: If the input sequence has a read position available, returns char_type(*gptr()). Otherwise, returns eof(). *

27.7.1.2.5 basic_stringbuf::seekoff [lib.stringbuf::seekoff]

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

Box 129
Check vs. 27.8.2.3.9

Effects: Alters the stream position within one of the controlled sequences, if possible, as indicated in Table 76:

Table 76—`seekoff` positioning

Conditions	Result
(*which* & basic_ios::in) != 0	positions the input sequence
(*which* & basic_ios::out) != 0	positions the output sequence
Otherwise, (*which* & (basic_ios::in | basic_ios::out)) == (basic_ios::in | basic_ios::out)) and *way* == either basic_ios::beg or basic_ios::end	positions both the input and the output sequences
Otherwise,	the positioning operation fails.

1 For a sequence to be positioned, if its next pointer is a null pointer, the positioning operation fails. Otherwise, the function determines *newoff* as indicated in Table 77:

Table 77—`newoff` values

Condition	`newoff` Value
way == basic_ios::beg	0
way == basic_ios::cur	the next pointer minus the beginning pointer (*xnext* - *xbeg*).
way == basic_ios::end	the end pointer minus the beginning pointer (*xend* - *xbeg*)
If (*newoff* + *off*) < 0, or (*xend* - *xbeg*) < (*newoff* + *off*)	the positioning operation fails

2 Otherwise, the function assigns *xbeg* + *newoff* + *off* to the next pointer *xnext*.

Returns: pos_type(*newoff*), constructed from the resultant offset *newoff* (of type off_type), that stores the resultant stream position, if possible. If the positioning operation fails, or if the constructed object cannot represent the resultant stream position, the object stores an invalid stream position.

27.7.1.2.6 `basic_stringbuf::seekpos` [lib.stringbuf::seekpos]

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

Box 130
Check vs. 27.8.2.3.9

Effects: Alters the stream position within one of the controlled sequences, if possible, to correspond to the stream position stored in *sp* (as described below).

— If (*which* & basic_ios::in) != 0, positions the input sequence.

— If (*which* & basic_ios::out) != 0, positions the output sequence.

— If the function positions neither sequence, the positioning operation fails.

1 For a sequence to be positioned, if its next pointer is a null pointer, the positioning operation fails. Otherwise, the function determines *newoff* from *sp*.offset()*:*

— If *newoff* is an *invalid stream position*, has a negative value, or has a value greater than (*xend* - *xbeg*), the positioning operation fails.

— Otherwise, the function adds *newoff* to the beginning pointer *xbeg* and stores the result in the next pointer *xnext*.

Returns: pos_type(*newoff*), constructed from the resultant offset *newoff* (of type off_type), that stores the resultant stream position, if possible. If the positioning operation fails, or if the constructed object cannot represent the resultant stream position, the object stores an invalid stream position.

27.7.1.2.7 basic_stringbuf::setbuf [lib.stringbuf::setbuf]

```
//  virtual basic_streambuf<charT,traits>*
//      setbuf(char_type* s, streamsize n);     inherited
```

Effects: Performs an operation that is defined separately for each class derived from basic_stringbuf<charT,traits>.

Default behavior: The same as for basic_streambuf<charT,traits>::setbuf(char_type*,streamsize).

27.7.2 Template class basic_istringstream [lib.istringstream]

```
namespace std {
  template <class charT, class traits = ios_traits<charT> >
  class basic_istringstream : public basic_istream<charT,traits> {
  public:
    typedef charT           char_type;
    typedef traits::int_type int_type;
    typedef traits::pos_type pos_type;
    typedef traits::off_type off_type;
    int_type  eof()     { return traits::eof(); }
    char_type newline() { return traits::newline(); }
```

```
  public:
    basic_istringstream(ios_base::openmode which
                        = ios_base::in);
    basic_istringstream(const basic_string<charT>& str,
                        ios_base::openmode which
                        = ios_base::in);
    virtual ~basic_istringstream();
    basic_stringbuf<charT,traits>* rdbuf() const;
    basic_string<charT> str() const;
    void str(const basic_string<charT>& s);
  };
}
```

1 The class `basic_istringstream<charT,traits>` is a derivative of `basic_istream<charT,traits>` that assists in the reading of objects of class `basic_stringbuf<charT,traits>`. It supplies a `basic_stringbuf` object to control the associated array object.

27.7.2.1 `basic_istringstream` constructors [lib.basic.istringstream.m.cons]

```
basic_istringstream(ios_base::openmode which
   = ios_base::in);
```

Effects: Constructs an object of class `basic_istringstream<charT,traits>`, initializing the base class with `basic_istream<charT,traits>(&sb)`.

```
  basic_istringstream(const basic_string<charT>& str,
                      ios_base::openmode which = ios_base::in);
```

Effects: Constructs an object of class `basic_istringstream<charT,traits>`, initializing the base class with `basic_istream<charT,traits>(&sb)`.

27.7.2.2 Member functions [lib.istringstream.members]

27.7.2.2.1 `basic_istringstream::rdbuf` [lib.istringstream::rdbuf]

```
basic_stringbuf<charT,traits>* rdbuf() const;
```

Returns: `dynamic_cast<basic_stringbuf<charT,traits>*>(basic_istream<harT,traits>::rdbuf())`.

27.7.2.2.2 `basic_istringstream::str` [lib.istringstream::str]

```
basic_string<charT> str() const;
```

Returns: `rdbuf().str()`.

```
  void str(const basic_string<charT>& s);
```

Effects: Calls `rdbuf().str(s)`.

27.7.2.3 Class `basic_ostringstream` [lib.ostringstream]

Class `basic_ostringstream`

```
namespace std {
  template <class charT, class traits = ios_traits<charT> >
  class basic_ostringstream : public basic_ostream<charT,traits> {
  public:
    typedef charT            char_type;
    typedef traits::int_type int_type;
    typedef traits::pos_type pos_type;
    typedef traits::off_type off_type;
    int_type  eof()     { return traits::eof(); }
    char_type newline() { return traits::newline(); }

  public:
    basic_ostringstream(ios_base::openmode which
                        = ios_base::out);
    basic_ostringstream(const basic_string<charT>& str,
                        ios_base::openmode which
                        = ios_base::out);
    virtual ~basic_ostringstream();
    basic_stringbuf<charT,traits>* rdbuf() const;
    basic_string<charT> str() const;
    void str(const basic_string<charT>& s);
  };
}
```

1 The class `basic_ostringstream<charT,traits>` is a derivative of `basic_ostream<charT,traits>` that assists in the writing of objects of class `basic_stringbuf<charT,traits>`. It supplies a `basic_stringbuf` object to control the associated array object.

27.7.2.4 `basic_ostringstream` constructors [lib.basic.ostringstream.cons]

```
basic_ostringstream(ios_base::openmode which
   = ios_base::out);
```

Effects: Constructs an object of class `basic_ostringstream`, initializing the base class with `basic_ostream(&sb)`.

```
basic_ostringstream(const basic_string<charT>& str,
                    ios_base::openmode which = ios_base::out);
```

Effects: Constructs an object of class `basic_ostringstream<charT,traits>`, initializing the base class with `basic_ostream<charT,traits>(&sb)`.

27.7.2.5 Member functions [lib.ostringstream.members]

27.7.2.5.1 `basic_ostringstream::rdbuf` [lib.ostringstream::rdbuf]

```
basic_stringbuf<charT,traits>* rdbuf() const;
```

Returns: `dynamic_cast<basic_stringbuf<charT,tratis>*>(basic_ostream<charT,traits>::rdbuf())`.

27.7.2.5.2 `basic_ostringstream::str` [lib.ostringstream::str]

```
basic_string<charT> str() const;
```

Returns: `rdbuf().str()`.

```
void str(const basic_string<charT>& s);
```

Effects: Calls `rdbuf().str(s)`.

27.8 File-based streams [lib.file.streams]

1 Headers: `<cstream>` and `<fstream>` (Table 78):

Box 131

ISSUE: Disposition of `<cstream>` and `convbuf` (next 10.5 pages) TBD.

Table 78—Header `<cstream>` synopsis

Type	Name(s)	
Template class:	`basic_convbuf`	
Template structs:	`conv_traits`	`ios_conv_traits`
Structs:	`conv_traits<wchar_t>`	`ios4traits<wstreampos>`

27.8.1 Multi-byte conversions [lib.conv.fstreams]

1 The header `<cstream>` defines one type, `basic_convbuf` that associates its internal stream buffers holding a sequence of *characters* with the external source/sink stream representing a sequence of another type of characters.

— **Conversion** There is a bidirectional conversion between the external source/sink stream and *character* sequences held in the `basic_convbuf` class object.

— **uchar_type: underlaid character container type** The external source/sink stream can be regarded as a sequence of a *character container type*, which may be different to `charT`. The character container type on the external source/sink stream is called the *underlaid character container type*, or *uchar_type*.

— **Encoding rule** Performing the conversion from a *uchar_type* character sequences to the corresponding *character* sequence, the `basic_convbuf` may parse the uchar_type sequence to extract the corresponding character represented on the uchar_type sequence. The rule how a certain character is to be represented on a sequence of uchar_type characters is called the *encoding rule*. The `basic_convbuf` can be regarded as having a (virtual) conversion machine which obeys the encoding rule to parse uchar_type sequences.

— **Conversion state** The conversion machine has its internal state corresponding to a certain position of the external source/sink stream. If the `basic_convbuf` stops the conversion to resume later, it has to save its internal state, so a class or a type is prepared for saving the state onto it so that an user of the `basic_convbuf` class object can handle it. The class (or the type) is called the *conversion state*.

—

Box 132

Multibyte character I/O support The `basic_convbuf` can support multibyte character file I/O operations. Provided that the code conversion functions between multibyte characters and wide characters are prepared, we may specify the conversion state, and the conversion functions in the `ios4traits` so that the `basic_convbuf` can handle this multibyte characters.

27.8.1.1 Template class `conv_traits` [lib.conv.traits]

```
template <class charT> struct conv_traits {}

struct conv_traits<wchar_t> {
    typedef ios_traits<wchar_t>   char_traits;
    typedef ios_traits<wstreampos> pos_traits;
    typedef ios4traits<state_t>   conv_traits;
};
```

1 The template struct `conv_traits<charT>` is a *traits class*which maintains the definitions of the types and functions necessary to implement the template class `basic_convbuf`. The template parameter `charT` reprsents the character container type and each specialized version of `conv_traits` provides the default definitions corresponding to the specialized character container type.

2 The `conv_traits<charT>` declaration has three members, `ios_traits`, `ios_traits`, and `ios4traits`.

3 Although the former two of them are same as in the `ios_traits<charT>`, the last traits, `ios4traits<charT>` provides all the definitions about types and functions necessary to perform conversion.

27.8.1.2 Template class `ios4traits` [lib.ios.conv.traits]

```
template <class stateT> struct ios4traits {};
```

1 As is the other traits structs, there is no definition in the declaration of the template struct, `ios4traits`. All of the definitions related to the conversion and necessary to implement `basic_convbuf` class shall be provided in a template version, `ios4traits<STATE_T>` specialized for a conversion state `STATE_T`.

27.8.1.2.1 Struct `ios4traits<STATE_T>` [lib.ios.conv.traits.state.t]

```
struct ios4traits<STATE_T> {
    typedef CHAR_T  char_type;
    typedef STATE_T conv_state;
    typedef UPOS_T  conv_upos;
    typedef UCHAR_T uchar_type;

    typedef POS_T   pos_type;
    typedef OFF_T   off_type;

    typedef codecnv<uchar_type,char_type,conv_state> codecvt_in;
    typedef codecnv<char_type,uchar_type,conv_state> codecvt_out;
```

```
    result convin(codecvt_in*, conv_state&,
            const uchar_type*, const uchar_type*, const uchar_type*&,
            char_type*, char_typeT*, char_type*&);
    result convout(codecvt_out*, conv_state&,
            const char_type*, const char_type*, const char_type*&,
            uchar_type*, uchar_type*, uchar_type*&);

    pos_type   get_pos(conv_state&, conv_upos&);
    off_type   get_off(conv_state&, conv_upos&);

    conv_state get_posstate pos_type&);
    conv_state get_offstate(off_type&);
    conv_upos  get_posupos (pos_type&);
    conv_upos  get_offupos (off_type&);
};
```

1 A specialized version of the struct `ios4traits<STATE_T>`is necessary to use the `basic_convbuf`class. In order to construct a specialized version of the struct, the following set of types and functions shall be provided.

— `CHAR_T` : is the *character container type* which shall be same as the template parameter of the class, `conv_traits`.

— `STATE_T` : is the *conversion state type* which also the template parameter type of this traits. It is the parsing/tracing state which the function, convin and convout are taken.

— Every type of conversion states have a state, called *initial state*, which corresponds to the state beginning to parse a new sequence. The constructor, `STATE_T::STATE_T()` or `STATE_T::STATE_T(0)` constructs the initial state.

— `UCHAR_T` : is the *underlaid character container type* for the external source/sink stream handled by the `basic_convbuf`.

— `UPOS_T` : is the *repositional information type* for the external source/sink stream. It is used to implement the `seekpos`/`seekoff` member functions in the `basic_convbuf`.

— `POS_T` : is the *repositional information* which the `basic_convbuf` supports for the client of the object. It shall be the same as the template parameter type of the struct `ios_traits` in the same `conv_traits`.

— `OFF_T` : is an integral type which is used as the repositional information which the `basic_convbuf`supports.

27.8.1.2.1.1 `convin` [lib.ios.conv.traits.state.t::convin]

```
result
vin(codecvt_in* ccvt, conv_state& stat,
    const uchar_type* from, const uchar_type* from_end,
    const uchar_type*& from_next,
    char_type* to, char_typeT* to_limit, char_type*& to_next);
```

1 The conversion state designated by *`stat`* represents the parsing state on the underlaid source sequence. According to the conversion state designated by *`stat`*, it begins to parse the source character sequence whose begining and end position designated by *`from`* and *`from_end`* to convert them. It stores the result sequence (if any) into the successive location of an array pointed to by *`to`* and *`to_limit`*. The conversion stops when either of the following occurs:

1) parses all of the underlaid character sequences and stores the last *character* into the destination array.

2) fills all of the destination array and no more room to store.

3) encounters an invalid underlaid character in the source sequence.

2 In case (1), it returns ok. In case (2), it returns partial. In case (3), it returns error.

3 In all cases it leaves the *from_next* and *to_next* pointers pointing one beyond the last character successfully converted and leaves the conversion state onto the *stat* for the next time conversion. If case (3) occurs, the conversion state on the *stat* becomes undefined value. No more conversion with the value is available.

4 If the locale-dependent conversion functions are needed, the nconversion function which the codecvt facet object, *ccnv* provides is available as the following invokation:

```
ccnv.convert(stat, from, from_end, from_next, to, to_limit, to_next);
```

5 The default conversion function depends on the locale_codecvt facets. Users can customize the encoding scheme by specifying their own conversion functions in a new conv_traits class.

27.8.1.2.1.2 convout [lib.ios.conv.traits.state.t::convout]

```
result convout(codecvt_out* ccvt, conv_state&,
            const char_type from*, const char_type* from_end,
            const char_type*& from_next.,
            uchar_type*, uchar_type*, uchar_type*&);
```

1 The conversion state designated by *stat* represents the parsing state on the underlaid destination sequence. It begins to convert the source character sequence whose begining and end position designated by *from* and *from_end*. According to the conversion state designated by *stat*, it stores the result destination underlaid chracter sequence (if any) into the successive location of an array pointed to by *to* and *to_limit*. The conversion stops when either of the following occurs:

1) consumes all of the source character sequences and stores the last underlaid *character* into the destination array.

2) fills all of the destination array and no more room to store.

3) encounters an invalid character in the source sequence.

2 In case (1), it returns ok. In case (2), it returns partial. In case (3), it returns error.

3 In all cases it leaves the *from_next* and *to_next* pointers pointing one beyond the last character successfully converted and leaves the conversion state onto the *stat* for the next time conversion. If case (3) occurs, the conversion state on the *stat* becomes undefined value. No more conversion with the value is available.

4 If the locale-dependent conversion functions are needed, the conversion function which the codecvt facet object, *ccnv* provides is available as the following invocation;

```
ccnv.convert(stat, from, from_end, from_next, to, to_limit, to_next);
```

5 The default conversion function is depends on the locale_codecvt facets. Users can customize the encoding scheme by specifying their own conversion functions in a new conv_traits class.

27.8.1.2.1.3 get_pos and get_off [lib.ios.conv.traits.state.t::get.pos]

```
pos_type get_pos(conv_state& stat, conv_upos& upos);
off_type get_off(conv_state&, conv_upos&);
```

Effects: Constructs the pos_type object or off_type object from the conversion state, *stat* and the repositional information for the underlaid stream, *upos* and returns it. These are used to make the return value of the basic_convbuf::seekpos/seekoff protected member function.

27.8.1.2.1.4 get_posstate and get_offstate [lib.ios.conv.traits.state.t::get.pos.state]

```
conv_state get_posstate(pos_type& pos);
conv_state get_offstate(off_type& off);
```

1 Extract the conversion state held by the repositional information, *pos* or *off*. These are used to reset the conversion state maintained in the basic_convbuf class object when the basic_convbuf::seetpos/seekoff functions are performed.

27.8.1.2.1.5 ios4traits<STATE_T>::get_posupos and get_offupos [lib.ios.conv.traits.state.t::get.pos.upos]

```
conv_upos  get_posupos(pos_type&);
conv_upos  get_offupos(off_type&);
```

1 Extract the repositioning information for the underlaid stream from the repositioning information on the basic_convbuf stream. These are used in the basic_convbuf::seekpos/seekoff protected member functions to reposition the external source/sink stream.

27.8.1.2.2 Struct ios4traits<wstreampos> [lib.ios.conv.traits.wstreampos]

```
struct ios4traits<wstreampos> {
    typedef wchar_t char_type;
    typedef STATE_T conv_state;
    typedef UPOS_T  conv_upos;
    typedef char    uchar_type;
    typedef wstreampos pos_type;
    typedef wstreamoff off_type;

    typedef codecnv<char,wchar_t,STATE_T> codecnv_in;
    typedef codecnv<wchar_t,char,STATE_T> codecnv_out;

    result convin(codecvt_in* ccvt, conv_state& stat,
           const char *from, const char* from_end, const char*& from_next,
           wchar_t* to, wchar_t* to_limit, wchar_t*& to_next) {
            return ccvt->convert (stat, from, from_end, from_next,
                 to, to_limit, to_next);
        }
    result convout(codecvt_out* ccvt, conv_state& stat,
           const wchar_t* from, const wchar_t* from_end, const wchar_t*& from_next,
           char* to, char* to_limit, char* to_next) {
            return ccvt->convert (stat, from, from_end, from_next,
                 to, to_limit, to_next);
        }
    pos_type   get_pos(conv_state&, conv_upos&);
    off_type   get_off(conv_state&, conv_upos&);

    conv_state get_posstate(pos_type&);
    conv_state get_offstate(off_type&);
    conv_upos  get_posupos (pos_type&);
    conv_upos  get_offupos (off_type&);
};
```

Struct `ios4traits<wstreampos>`

1 The specialized version of the struct, `ios4traits<wstreampos>` supports the wide-oriented `basic_convbuf` class and the conversion between wide characters and multibyte characters as the underlaid character sequence. The types, `STATE_T` and `UPOS_T` are implementation-defined types. The behavior of the conversion functions depend on those of the `locale` object.

2 The encoding rule of the multibyte characters is implementation-defined.

27.8.1.3 Template class `basic_convbuf` [lib.basic.convbuf]

```
template <class charT, class traits = conv_traits<charT> >
class basic_convbuf : public basic_streambuf<charT,traits> {
public:
    typedef charT            char_type;
    typedef traits::int_type int_type;
    typedef traits::pos_type pos_type;
    typedef traits::off_type off_type;
    int_type  eof()     { return traits::eof(); }
    char_type newline() { return traits::newline(); }
private:
    typedef traits::conv_traits::conv_state state_type;
    typedef traits::conv_traits::conv_upos upos_type ;
public:
    basic_convbuf();
    virtual ~basic_convbuf();

protected:
//  virtual int_type overflow (int_type c = eof());     inherited
//  virtual int_type pbackfail(int_type c = eof());     inherited
//  virtual int_type underflow();                                    inherited
// virtual int_type uflow();                                  inherited
// virtual streamsize xsgetn(charT* s, streamsize n);          inherited
//  virtual streamsize xsputn(const charT* s, streamsize n);    inherited
//  virtual pos_type seekoff(off_type, ios_base::seekdir way,
//                           ios_base::openmode which
//                             = ios_base::in
//                             | ios_base::out);        inherited
//  virtual pos_type seekpos(pos_type  sp, ios_base::openmode which
//                             = ios_base::in
//                             | ios_base::out);        inherited
//  virtual basic_streambuf<charT,traits>*
//                  setbuf(charT* s, streamsize n);     inherited
//  virtual int sync();         inherited
private:
// state_type state;   exposition only
};
```

1 The template class `basic_convbuf<charT,traits>` is derived from `basic_streambuf<charT,traits>` to associate the *character* sequences and the underlaid character sequences. It performs the conversion between these two types of chracter sequences.

2 For the sake of exposition, the `basic_convbuf` maintains the conversion state to restart the conversion to read/write the character sequence from/to the underlaid stream.

27.8.1.3.1 `basic_convbuf` constructor [lib.basic.convbuf.cons]

```
basic_convbuf();
```

Effects: Constructs an object of template class `basic_convbuf<charT,traits>`, initializing the base class with `basic_streambuf<charT,traits>()`, and initializing;

— *state* with `state_type(0)` as to specify initial state.

27.8.1.3.2 `basic_convbuf` destructor [lib.basic.convbuf.des]

```
virtual ~basic_convbuf();
```

Effects: Destroys an object of class `basic_convbuf<charT,traits>`.

27.8.1.3.3 `basic_convbuf::overflow` [lib.basic.convbuf.overflow]

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

Effects: Behaves the same as `basic_streambuf<charT,traits>::overflow(c)`, except that the behavior of ''consuming a character'' is as follows:

— (1) Converting the characters to be consumed into the underlaid character sequence with the function `traits::conv_traits::convout()`.

— (2) The result underlaid character sequence is appended to the associated output stream.

27.8.1.3.4 `basic_convbuf::pbackfail` [lib.basic.convbuf::pbackfail]

```
virtual int_type pbackfail(int_type c);
```

Effects: Puts back the character designated by `c` to the input sequence, if possible, in one of three ways:

Box 133

Because the parsing on the underlaid character sequence generally can only go advance or most of parsing machined cannot go back, some of the `basic_convbuf` implementations cannot ''put back characters directly to the associated input sequence.'' So the behaviors related to putting back the associated input stream are removed.

— If *c* `!= eof()`, if either the input sequence has a putback position available or the function makes a putback position available, and if `(charT)`*c* `== (charT)gptr()[-1]`, assigns `gptr() - 1` to `gptr()`. Returns `(charT)`*c*.

— If *c* `!= eof()`, if either the input sequence has a putback position available or the function makes a putback position available, and if the function is permitted to assign to the putback position, assigns *c* to `*--gptr()`. Returns `(charT)`*c*.

— If *c* `== eof()` and if either the input sequence has a putback position available or the function makes a putback posotion available, assigns `gptr() - 1` to `gptr()`. Returns `(charT)`*c*.

Returns: `eof()` to signal the failure.

Notes:

1 The function does not put back a character directly to the input sequence. *

2 If the function can succeed in more than one of these ways, it is unspecified which way is chosen. The function can alter the number of putback positions available as a result of any call.

3 Default behavior: returns `traits::eof()`.

27.8.1.3.5 `basic_convbuf::underflow` [lib.basic.convbuf::underflow]

```
//  virtual int_type underflow();     inherited
```

Returns: the first *character* of the *pending sequence*, if possible, without moving the stream position past it. If the pending sequence is null then the function fails.

1 The *pending sequence* of characters is defined as the concatenation of:

a) If `gptr()` is non- `NULL`, then the `egptr() - gptr()` characters starting at `gptr()`, otherwise the empty sequence.

b) Some sequence (possibly empty) of characters read from the input stream.

— These underlaid character sequence shall be converted by the function `traits::conv_traits::convin()` before concatinating to construct the pending sequence. In case that there remains some underlaid characters which cannot be converted to any more *characters*, we treat it empty as the subsequence (b).

2 The *result character* is the first character of the *pending sequence*, if any. The *backup sequence* is defined as the concatenation of:

a) If `eback()` is non- `NULL` then empty, otherwise the `gptr() - eback()` characters beginning at `eback()`.

b) the result character.

3 The function sets up `gptr()` and `egptr()` satisfying:

a) In case the pending sequence has more than one character the `egptr() - gptr()` characters starting at `gptr()` are the characters in the pending sequence after the result character.

b) If the pending sequence has exactly one character, then `gptr()` and `egptr()` may be `NULL` or may both be set to the same non- `NULL` pointer.

4 If `eback()` and `gptr()` are non- `NULL` then the function is not constrained as to their contents, but the ‘‘unusual backup condition’’ is that either:

a) If the backup sequence contains at least `gptr() - eback()` characters, then the `gptr() - eback()` characters starting at `eback()` agree with the last `gptr() - eback()` characters of the backup sequence.

b) Or the n characters starting a `gptr() - n` agree with the backup sequence (where *n* is the length of the backup sequence)

Returns: `eof()` to indicate failure.

5 Default behavior: returns `traits::eof()`. *

27.8.1.3.6 basic_convbuf::seekoff [lib.convbuf::seekoff]

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

Effects: Behaves the same as `basic_streambuf<charT,traits>::seekoff(off,way,which)`, except that the behavior ‘‘alters the stream position’’ means that:

— (a) altering the underlaid character stream position which they get with the return value of the `traits::conv_traits::get_posupos(off_type&)`; and,

— (b) restoring the current conversion state by resetting the member, *state* with the return value of the `traits::conv_traits::get_offstate(off_type&)`.

27.8.1.3.7 basic_convbuf::seekpos [lib.convbuf::seekpos]

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

1 At first do sync() to clear up the get buffer, and alters the stream position, as follows:

— (a) altering the underlaid character stream position by which the function, traits::conv_traits::get_posupos(*pos*), returns. and,

— (b) restoring the member, *state* with the return value of the traits::conv_traits::get_posstate(*pos*).

Returns: pos_type, *newpos*, constructed from the resultant *upos* and *state* if both (a) and (b) are successfully terminated.

2 If either or both (a) or/and (b) fail(s), or if the constructed object cannot represent the resultant stream position, the object stores an invalid stream position.

3 Whichever value is specified in the *which* parameters, the basic_convbuf handles only one external source/sink stream.

4 If (a) succeeds, returns a newly constructed streampos object returned by traits::conv_traits::get_pos(*state, upos*) where *upos* is a UPOS_T type object which represent the current position of the underlaid stream.

5 Otherwise, the object stores an invalid stream position.

27.8.1.3.8 basic_convbuf::setbuf [lib.convbuf::setbuf]

```
//  virtual basic_streambuf<charT,traits>*
//      setbuf(char_type* s, streamsize n);     inherited
```

1 Makes the array of *n* (charT type) characters, whose first element is designated by *s*, available for use as a buffer area for the controlled sequences, if possible.

Box 134

The basic_convbuf does not fix the buffer management strategy. It remains alternatives for the derived class designers.

27.8.1.3.9 basic_convbuf::sync [lib.convbuf::sync]

```
//  virtual int sync();     inherited
```

1 Reflects the pending sequence to the external sink sequence and reset the get/put buffer pointers. It means that if there are some unread sequence on the get buffer and the external source sequense is not seekable, the unread sequence is perfectly lost.

2 The detailed behavior is as follows:

a) Consumes all of the pending sequence of characters, (as *pending sequence* and ‘‘consumes the sequence,’’ see the description in the basic_streambuf<charT,traits>::overflow() 27.5.2.2.24) In case that consuming means appending characters to the associated output stream, the *character* sequense shall be converted to the corresponding underlaid character sequence by the traits::conv_traits::convout().

b) Clears all of the following pointers: pbase(), pptr(), epptr(), eback(), gptr(), and

`egptr()`.

Returns: `-1` if (a) fails, otherwise 0.

27.8.1.4 Examples of trait specialization [lib.examples.traits]

```
class fstate_t { ... };       // Implementation-defined conversion state
                              // object which is for file I/O.
class wfstate_t { ... };

template <class charT> struct file_traits {};

struct file_traits<char> {
    typedef ios_traits<char> char_traits;
    typedef ios_traits<streampos> char_pos;
    typedef ios4traits<fstate_t> conv_pos;
};

struct file_traits<wchar_t>{
    typedef ios_traits<char> char_traits;
    typedef ios_traits<wstreampos> char_pos;
    typedef ios4traits<wfstate_t> conv_pos;
};

template <class stateT> struct ios_file_traits {};
//
// Specialized for the single-byte filebuf
//

struct ios4traits<fstate_t> {
    typedef char       char_type;       // char/wchar_t...
    typedef fstate_t   conv_state;      // key parameter(mainly depend on uchar_type)
    typedef streamoff  conv_upos;       // Physical file offset
    typedef char       uchar_type;      // Physical file I/O
    typedef streampos  pos_type;        // Enough to large
    typedef streamoff  off_type;        // Enough to large

    typedef codecnv<char,wchar_t,fstate_t> codecnv_in;
    typedef codecnv<wchar_t,char,fstate_t> codecnv_out;

    result convin(codecvt_in* ccvt, conv_state& stat,
                          const char *from, const char* from_end,
                          const char*& from_next,
                          char* to, char* to_limit, char*& to_next) {
        return ccvt->convert (stat, from, from_end, from_next,
                              to, to_limit, to_next);
    }
    result convout(codecvt_out* ccvt, conv_state& stat,
                           const char* from, const char* from_end,
                           const char*& from_next,
                           char* to, char* to_limit, char* to_next) {
        return ccvt->convert (stat, from, from_end, from_next,
                              to, to_limit, to_next);
    }
    pos_type get_pos(conv_state&, conv_upos&);
    off_type get_off(conv_state&, conv_upos&);
```

```
    conv_state get_posstate(pos_type&);
    conv_state get_offstate(off_type&);
    conv_upos  get_posupos (pos_type&);
    conv_upos  get_offupos (off_type&);
};

//
// Specialized for the wfilebuf
//

struct ios4traits<wfstate_t> {
    typedef wchar_t    char_type;       // char/wchar_t...
    typedef wfstate_t   conv_state;     // key parameter(mainly depend on uchar_type)
    typedef streamoff  conv_upos;       // Physical file offset
    typedef char       uchar_type;      // Physical file I/O
    typedef wstreampos pos_type;        // Enough to large
    typedef wstreamoff off_type;        // Enough to large

    typedef codecnv<char,wchar_t,fstate_t> codecnv_in;
    typedef codecnv<wchar_t,char,fstate_t> codecnv_out;

    result convin (codecvt_in* ccvt, conv_state& stat,
              const char *from, const char* from_end, const char*& from_next,
              wchar_t* to, wchar_t* to_limit, wchar_t*& to_next) {
        return ccvt->convert (stat, from, from_end, from_next,
                              to, to_limit, to_next);
    }
    result convout (codecvt_out* ccvt, conv_state& stat,
              const wchar_t* from, const wchar_t* from_end,
              const wchar_t*& from_next,
              char* to, char* to_limit, char* to_next) {
        return ccvt->convert (stat, from, from_end, from_next,
                              to, to_limit, to_next);
    }

    pos_type get_pos (conv_state&, conv_upos&);
        // wstreampos get_pos (wfstate_t&, streampos);
    off_type get_off (conv_state&, conv_upos&);
        // wstreamoff get_off (wfstate_t&, streampos);
    conv_state get_posstate (pos_type&);
        // wfstate_t get_posstate (wstreampos&);
    conv_state get_offstate (off_type&);
        // wfstate_t get_offstate (wstreamoff&);
    conv_upos  get_posupos (pos_type&);
        //
    conv_upos  get_offupos (off_type&);
};
```

27.8.2 File streams [lib.fstreams]

Header `<fstream>` synopsis

```
#include <cstream>      // for conv_traits, basic_convbuf
#include <streambuf>
#include <istream>
#include <ostream>

namespace std {
  template <class charT, class traits = conv_traits<charT> >
    class basic_filebuf;
  typedef basic_filebuf<char>    filebuf;
  typedef basic_filebuf<wchar_t> wfilebuf;

  template <class charT, class traits = file_traits<charT> >
    class basic_ifstream;
  typedef basic_ifstream<char>    ifstream;
  typedef basic_ifstream<wchar_t> wifstream;

  template <class charT, class traits = file_traits<charT> >
    class basic_ofstream;
  typedef basic_ofstream<char>    ofstream;
  typedef basic_ofstream<wchar_t> wofstream;
}
```

1 The header <fstream> defines six types that associate stream buffers with files and assist reading and writing files.

Box 135

basic_filebuf<charT,traits> should be specified so that it treats a file as a sequence of charT. Except for filebuf and wfilebuf that implies it treats the file as binary.

2 In this subclause, the type name *FILE* is a synonym for the type FILE.[213)]

— **File** A File provides an external source/sink stream whose *underlaid character type* is char (byte).

— **Multibyte characters and Files** Class basic_filebuf is derived from basic_convbuf to support multibyte character I/O. A File is a sequence of multibyte characters. In order to provide the contents as a wide character sequence, wfilebuf should convert between wide character sequences and multibyte character sequences.

— **basic_filebuf** Derived from basic_convbuf. Even the single byte character version because single byte code conversion may be necessary... A basic_filebuf provide... file I/O, char/wchar_t parity. fstate_t. ios4traits

— **Customize Mechanism** Change name (conv_traits-> file_traits), customize ios4traits<fstate_t>. Modify the definition of ios4traits.

— **Multibyte character and Files** A File provides byte sequences. So the streambuf (or its derived classes) treats a file as the external source/sink byte sequence. In a large character set environment, multibyte character sequences are held in files. In order to provide the contents of a file as wide character sequences, wide-oriented filebuf, namely wfilebuf should convert wide character sequences. Because of necessity of the conversion between the external source/sink streams and wide character sequences.

213) FILE is defined in <cstdio> (27.8.3).

27.8.2.1 Template class basic_filebuf [lib.filebuf]

```
namespace std {
  template <class charT, class traits = conv_traits<charT> >
  class basic_filebuf : public basic_convbuf<charT,traits> {
  public:
    typedef charT            char_type;
    typedef traits::int_type int_type;
    typedef traits::pos_type pos_type;
    typedef traits::off_type off_type;
    int_type  eof()     { return traits::eof(); }
    char_type newline() { return traits::newline(); }

  public:
    basic_filebuf();
    virtual ~basic_filebuf();
    bool is_open() const;
    basic_filebuf<charT,traits>*
         open(const char* s, ios_base::openmode mode);
//  basic_filebuf<charT,traits>* open(const char* s, ios::open_mode mode);
    basic_filebuf<charT,traits>* close();
  protected:
//  virtual int_type overflow (int_type c = eof());     inherited
//  virtual int_type pbackfail(int_type c = eof());     inherited

//  virtual int      showmany();                inherited
//  virtual int_type underflow();               inherited
// virtual int_type uflow();                    inherited
// virtual streamsize xsgetn(char_type* s streamsize n); inherited
//  virtual streamsize xsputn(const char_type* s, streamsize n);        inherited
//  virtual pos_type seekoff(off_type off,
//                           ios_base::seekdir way,
//                           ios_base::openmode which
//                             = ios_base::in
//                             | ios_base::out);       inherited
//  virtual pos_type seekpos(pos_type sp,
//                           ios_base::openmode which
//                             = ios_base::in
//                             | ios_base::out);       inherited
//  virtual basic_streambuf<charT,traits>*
//                    setbuf(char_type* s, streamsize n);       inherited
//  virtual int sync();                         inherited
 private:
// FILE* file;      exposition only
  };
}
```

1 The class basic_filebuf<charT,traits> is derived from basic_streambuf<charT,traits> to associate both the input sequence and the output sequence with an object of type FILE.

Box 136

For the sake of exposition, the maintained data is presented here as:

— *FILE* file*, points to the FILE associated with the object of class basic_filebuf<charT,traits>.

2 The restrictions on reading and writing a sequence controlled by an object of class basic_filebuf<charT,traits> are the same as for reading and writing its associated file. In particular:

— If the file is not open for reading or for update, the input sequence cannot be read.

— If the file is not open for writing or for update, the output sequence cannot be written.

— A joint file position is maintained for both the input sequence and the output sequence.

3 In order to support file I/O and multibyte/wide character conversion, the following arrangement is applied to the traits structures:

— Specify `char` as the underlaid character type, *uchar_type* defined in the `ios4traits<>`.

— Define `state_t`, the template parameter of the `ios4traits<>`, so that it can apply to the conversion function. In case the conversion function provided by the `codecvt` facet uses, we adopt a specialized template parameter `stateT` for the locale-oriented conversion function suitable for the multibyte/wide character conversion. Now we assume the name of the conversion state object is `fstate_t`.

— Define `streamoff` (or `streampos`) as the repositional information for the underlaid byte sequence.

— Define `streamoff`, `wstreampos` for the wide-oriented `streambuf`s (or `filebuf`s) so that some composite function in the `ios4traits()`, for example:

```
wstreampos get_pos(fstate_t fs, streampos s);
wstreamoff get_off(fstate_t fs, streampos s);
```

27.8.2.2 `basic_filebuf` constructors [lib.basic.filebuf.cons]

```
basic_filebuf();
```

Effects: Constructs an object of class `basic_filebuf<charT,traits>`, initializing the base class with `basic_streambuf<charT,traits>()`, and initializing *`file`* to a null pointer.

```
virtual ~basic_filebuf();
```

Effects: Destroys an object of class `basic_filebuf<charT,traits>`. Calls `close()`.

27.8.2.3 Member functions [lib.filebuf.members]

27.8.2.3.1 `basic_filebuf::is_open` [lib.filebuf::is.open]

```
bool is_open() const;
```

Returns: `true` if *`file`* is not a null pointer.

27.8.2.3.2 `basic_filebuf::open` [lib.filebuf::open]

```
basic_filebuf<charT,traits>* open(const char* s,
                  ios_base::openmode mode);
```

Effects: If *`file`* is not a null pointer, returns a null pointer. Otherwise, calls `basic_streambuf<charT,traits>::basic_streambuf()`. It then opens a file, if possible, whose name is the NTBS *`s`*, ''as if'' by calling `fopen(`*`s`*`,`*`modstr`*`)` and assigning the return value to *`file`*.

1 The NTBS *`modstr`* is determined from *`mode`* `& ~ios_base::ate` as indicated in Table 79:

Table 79—File open modes

ios_base Value(s)	stdio equivalent
in	"r"
out | trunc	"w"
out | app	"a"
in | out	"r+"
in | binary	"rb"
out | trunc | binary	"wb"
out | app | binary	"ab"
in | out	"r+
in | out | trunc	"w+"
in | out | app	"a+"
in | out | binary	"r+b"
in | out | trunc | binary	"w+b"
in | out | app | binary	"a+b"

2 If the resulting *file* is not a null pointer and *mode & ios_base::ate != 0*, calls fseek(*file*,0,SEEK_END).[214)]

3 If fseek returns a null pointer, calls close() and returns a null pointer. Otherwise, returns this.

```
basic_filebuf<charT,traits>* open(const char* s, ios::open_mode mode);
```

Returns: open(*s*, ios_base::openmode(*mode*)).

27.8.2.3.3 basic_filebuf::close [lib.filebuf::close]

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

Effects: If *file* is a null pointer, returns a null pointer. Otherwise, if the call fclose(*file*) returns zero, the function stores a null pointer in *file* and returns this.[215)] Otherwise, returns a null pointer.

27.8.2.3.4 basic_filebuf::overflow [lib.filebuf::overflow]

```
//  virtual int_type overflow(int_type c = eof());     inherited
```

Effects: Behaves the same as basic_streambuf<charT,traits>::overflow(*c*), except that the behavior of ''consuming a character'' is as follows:

— (1) Converting the characters to be consumed into the underlaid character sequence with the function traits::conv_traits::convout(). During the conversion, convout maintains the conversion state in the member rdstate(). At the end of execution of the conversion, the conversion state is saved on it.

— (2) The result underlaid character sequence is written to the file specified by *file*. At the consumption * of the pending characters, none of them are discarded. All of the characters are converted to be written to the file.

214) The macro SEEK_END is defined, and the function signatures fopen(const char_type*, const char_type*) and fseek(FILE*, long, int) are declared, in <cstdio> (27.8.3).
215) The function signature fclose(FILE*) is declared, in <cstdio> (27.8.3).

Returns: `eof()` to indicate failure. If *`file`* is a null pointer, the function always fails.

27.8.2.3.5 `basic_filebuf::pbackfail` [lib.filebuf::pbackfail]

```
//  virtual int_type pbackfail(int_type c = eof());
```
inherited

Box 137
Check vs. 27.5.2.2.25.

Effects: Puts back the character designated by *`c`* to the input sequence, if possible, in one of four ways:

Box 138
Because the parsing on the underlaid character sequence generally can only go advance or most of parsing machined cannot go back, some of the `basic_convbuf` implementations cannot ‘‘put back characters directly to the associated input sequence.’’ So the behaviors related to putting back the associated input stream are removed.

— If *`c`* `!= eof()` and if the function makes a putback position available and if `char_type(`*`c`*`) == char_type(gptr()[-1])`, assigns `gptr() - 1` to `gptr()`.

— If *`c`* `!= eof()` and if the function makes a putback position available and if the function is permitted to assign to the putback position, assigns *`c`* to `*--gptr()`. *

— If *`c`* `== eof()` and if either the input sequence has a putback position available or the function makes a putback position available, assigns `gptr() - 1` to `gptr()`.

Returns: `eof()` to indicate failure, otherwise *`c`*.

Notes: If *`file`* is a null pointer, the function always fails.
The function does not put back a character directly to the input sequence.
If the function can succeed in more than one of these ways, it is unspecified which way is chosen. The function can alter the number of putback positions available as a result of any call.

Default behavior: Returns `traits::eof()`.

Box 139
Shall we impose Library uses onto performing `sync()`... make gbuffer/pbuffer empty every time they try to write after read or vice versa, as in the current MSE?

No. On systems where this is necessary the `basic_filebuf` virtuals should keep track of the last operation performed on the `FILE` and do whatever is neccessary (equivalent of an `fseek`?) in order to reverse the direction. *

27.8.2.3.6 `basic_filebuf::showmany` [lib.filebuf::showmany]

```
//  virtual int showmany();
```
inherited

Effects: Behaves the same as `basic_streambuf::showmany()`.

Notes: An implementation might well provide an overriding definition for this function signature if it can determine that more characters can be read from the input sequence. *

27.8.2.3.7 basic_filebuf::underflow [lib.filebuf::underflow]

```
//  virtual int_type underflow();     inherited
```

Effects: Behaves the same as `basic_convbuf<charT,traits>::underflow()`, except that the underlaid input character sequence is the byte sequence in the file specified by *`file`*.

Box 140
Describing the behavior about maintaining the conversion state is needed.

27.8.2.3.8 basic_filebuf::seekoff [lib.filebuf::seekoff]

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

Box 141
Check vs. 27.7.1.2.5.

Effects: Alters the stream position within the controlled sequences, if possible, as described below.
Returns: a newly constructed `pos_type` object that stores the resultant stream position, if possible. If the positioning operation fails, or if the object cannot represent the resultant stream position, the object stores an invalid stream position.

1 If *`file`* is a null pointer, the positioning operation fails. Otherwise, the function determines one of three values for the argument *`whence`*, of type `int`, as indicated in Table 80:

Table 80—`seekoff` effects

`way` Value	`stdio` Equivalent
`basic_ios::beg`	`SEEK_SET`
`basic_ios::cur`	`SEEK_CUR`
`basic_ios::end`	`SEEK_END`

2 The function then calls `fseek(`*`file`*`,`*`off`*`,`*`whence`*`)` and, if that function returns nonzero, the positioning operation fails.[216)]

3 The function extracts the conversion state from *`off`* by means of `get_offstate()` to reset the `rdstate()` member.

27.8.2.3.9 basic_filebuf::seekpos [lib.filebuf::seekpos]

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

216) The macros `SEEK_SET`, `SEEK_CUR`, and `SEEK_END` are defined, and the function signature `fseek(FILE*, long, int)` is declared, in `<cstdio>` (27.8.3).

Box 142
[To Be Filled]

Check vs. 27.5.2.2.32 and 27.7.1.2.6.

27.8.2.3.10 basic_filebuf::setbuf [lib.filebuf::setbuf]

```
//  virtual basic_streambuf* setbuf(char_type* s, int n);     inherited
```

Box 143
[To Be Filled]

Check vs. 27.5.2.2.33 and 27.7.1.2.7.

27.8.2.3.11 basic_filebuf::sync [lib.filebuf::sync]

```
//  virtual int sync();     inherited
```

Box 144
[To Be Filled]

Check vs. 27.5.2.2.34.

27.8.2.4 Template class basic_ifstream [lib.ifstream]

```
namespace std {
  template <class charT, class traits = file_traits<charT> >
  class basic_ifstream : public basic_istream<charT,traits> {
  public:
    typedef charT            char_type;
    typedef traits::int_type int_type;
    typedef traits::pos_type pos_type;
    typedef traits::off_type off_type;
    int_type  eof()     { return traits::eof(); }
    char_type newline() { return traits::newline(); }

  public:
    basic_ifstream();
    basic_ifstream(const char* s, openmode mode = in);
    virtual ~basic_ifstream();
    basic_filebuf<charT,traits>* rdbuf() const;
    bool is_open();
    void open(const char* s, openmode mode = in);
//  void open(const char* s, open_mode mode = in);    optional
    void close();
  private:
//  basic_filebuf<charT,traits> fb;      exposition only
  };
}
```

1 The class `basic_ifstream<charT,traits>` is a derivative of `basic_istream<charT,traits>` that assists in the reading of named files. It supplies a `basic_filebuf<charT,traits>` object to control the associated sequence.

Box 145

For the sake of exposition, the maintained data is presented here as:

— `basic_filebuf<charT,traits> fb`, the `basic_filebuf` object.

27.8.2.5 `basic_ifstream` constructors [lib.basic.ifstream.cons]

```
basic_ifstream();
```

Effects: Constructs an object of class `basic_ifstream<charT,traits>`, initializing the base class with `basic_istream<charT,traits>(&fb)`.

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

Effects: Constructs an object of class `basic_ifstream`, initializing the base class with `basic_istream<charT,traits>(&fb)`, then calls `open(s, mode)`.

27.8.2.6 Member functions [lib.ifstream.members]

27.8.2.6.1 `basic_ifstream::rdbuf` [lib.ifstream::rdbuf]

```
basic_filebuf<charT,traits>* rdbuf() const;
```

Returns: `dynamic_cast<basic_filebuf<charT,traits>*>(basic_istream<charT,traits>::rdbuf())`.

27.8.2.6.2 `basic_ifstream::is_open` [lib.ifstream::is.open]

```
bool is_open();
```

Returns: `rdbuf().is_open()`.

27.8.2.6.3 `basic_ifstream::open` [lib.ifstream::open]

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

Effects: Calls `rdbuf().open(s,mode)`. If `is_open()` returns zero, calls `setstate(failbit)`.

27.8.2.6.4 `basic_ifstream::close` [lib.ifstream::close]

```
void close();
```

Effects: Calls `rdbuf().close()` and, if that function returns zero, calls `setstate(failbit)`.

27.8.2.7 Template class `basic_ofstream` [lib.ofstream]

Template class `basic_ofstream`

```
namespace std {
  template <class charT, class traits = file_traits<charT> >
  class basic_ofstream : public basic_ostream<charT,traits> {
  public:
    typedef charT            char_type;
    typedef traits::int_type int_type;
    typedef traits::pos_type pos_type;
    typedef traits::off_type off_type;
    int_type  eof()     { return traits::eof(); }
    char_type newline() { return traits::newline(); }

  public:
    basic_ofstream();
    basic_ofstream(const char* s, openmode mode = out);
    virtual ~basic_ofstream();
    basic_filebuf<charT,traits>* rdbuf() const;
    bool is_open();
    void open(const char* s, openmode mode = out | trunc);
//  void open(const char* s, open_mode mode = out | trunc);    optional
    void close();
  };
}
```

1 The class `basic_ofstream<charT,traits>` is a derivative of `basic_ostream<charT,traits>` that assists in the writing of named files. It supplies a `basic_filebuf<charT,traits>` object to control the associated sequence.

27.8.2.8 `basic_ofstream` constructors [lib.basic.ofstream.cons]

```
basic_ofstream();
```

Effects: Constructs an object of class `basic_ofstream<charT,traits>`, initializing the base class with `basic_ostream<charT,traits>(&fb)`.

```
basic_ofstream(const char* s, openmode mode = out);
```

Effects: Constructs an object of class `basic_ofstream<charT,traits>`, initializing the base class with `basic_ostream<charT,traits>(&fb)`, then calls `open(s, mode)`.

27.8.2.9 Member functions [lib.ofstream.members]

27.8.2.9.1 `basic_ofstream::rdbuf` [lib.ofstream::rdbuf]

```
basic_filebuf<charT,traits>* rdbuf() const;
```

Returns: `(basic_filebuf<charT,traits>*)&fb`.

27.8.2.9.2 `basic_ofstream::is_open` [lib.ofstream::is.open]

```
bool is_open();
```

Returns: `rdbuf().is_open()`.

27.8.2.9.3 basic_ofstream::open [lib.ofstream::open]

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

Effects: Calls `rdbuf().open(s,mode)`. If `is_open()` is then `false`, calls `setstate(failbit)`.

27.8.2.9.4 basic_ofstream::close [lib.ofstream::close]

```
void close();
```

Effects: Calls `rdbuf().close()` and, if that function returns zero, calls `setstate(failbit)`.

27.8.3 C Library files [lib.c.files]

1 Headers <cstdio>, and <cwchar>.

Table 80—Header <cstdio> synopsis

Type	Name(s)				
Macros:					
BUFSIZ	L_tmpnam	SEEK_SET	TMP_MAX		
EOF	NULL <cstdio>	stderr	_IOFBF		
FILENAME_MAX	SEEK_CUR	stdin	_IOLBF		
FOPEN_MAX	SEEK_END	stdout	_IONBF		
Types:	FILE	fpos_t	size_t <cstdio>		
Functions:					
clearerr	fgets	fscanf	gets	rewind	tmpfile
fclose	fopen	fseek	perror	scanf	tmpnam
feof	fprintf	fsetpos	printf	setbuf	ungetc
ferror	fputc	ftell	putc	setvbuf	vprintf
fflush	fputs	fwrite	puts	sprintf	vprintf
fgetc	fread	getc	remove	sscanf	vsprintf
fgetpos	freopen	getchar	rename	tmpfile	

Table 80—Header <cwchar> synopsis

Type	Name(s)				
Macros:	NULL <cwchar>	WCHAR_MAX	WCHAR_MIN	WEOF <cwchar>	
Types:	mbstate_t	wint_t <cwchar>			
Struct:	tm <cwchar>				
Functions:					
btowc	getwchar	ungetwc	wcscpy	wcsrtombs	wmemchr
fgetwc	mbrlen	vfwprintf	wcscspn	wcsspn	wmemcmp
fgetws	mbrtowc	vswprintf	wcsftime	wcsstr	wmemcpy
fputwc	mbsinit	vwprintf	wcslen	wcstod	wmemmove
fputws	mbsrtowcs	wcrtomb	wcsncat	wcstok	wmemset
fwide	putwc	wcscat	wcsncmp	wcstol	wprintf
fwprintf	putwchar	wcschr	wcsncpy	wcstoul	wscanf
fwscanf	swprintf	wcscmp	wcspbrk	wcsxfrm	
getwc	swscanf	wcscoll	wcsrchr	wctob	

2 The contents are the same as the Standard C library, except that none of the headers defines `wchar_t`.

SEE ALSO:

ISO C subclause 7.9, Amendment 1 subclause 4.6.2.

Annex A (informative)
Grammar summary [gram]

1 This summary of C++ syntax is intended to be an aid to comprehension. It is not an exact statement of the language. In particular, the grammar described here accepts a superset of valid C++ constructs. Disambiguation rules (6.8, 7.1, _class.ambig_) must be applied to distinguish expressions from declarations. Further, access control, ambiguity, and type rules must be used to weed out syntactically valid but meaningless constructs.

A.1 Keywords [gram.key]

1 New context-dependent keywords are introduced into a program by `typedef` (7.1.3), namespace (7.3.1), class (9), enumeration (7.2), and `template` (14) declarations.

typedef-name:
    identifier

namespace-name:
    original-namespace-name
    namespace-alias

original-namespace-name:
    identifier

namespace-alias:
    identifier

class-name:
    identifier
    template-class-id

enum-name:
    identifier

template-name:
    identifier

Note that a *typedef-name* naming a class is also a *class-name* (9.1).

A.2 Lexical conventions [gram.lex]

preprocessing-token:
    header-name
    identifier
    pp-number
    character-constant
    string-literal
    preprocessing-op-or-punc
    each non-white-space character that cannot be one of the above

token: *

identifier
keyword
literal
operator
punctuator

identifier:

nondigit
identifier nondigit
identifier digit

nondigit: one of

```
_ 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
```

digit: one of

```
0 1 2 3 4 5 6 7 8 9
```

preprocessing-op-or-punc: one of

```
{       }       [       ]       #       ##      =       (       )       , |
<:      :>      <%      %>      %:      %:%:    ;       :       ...       |
new     delete  new[]   delete[]        ?                                 |
+       -       *       /       %       ^       &       |       ~         |
!       =       <       >       +=      -=      *=      /=      %=        |
^=      &=      |=      <<      >>      >>=     <<=     ==      !=        |
<=      >=      &&      ||      ++      --      ,       ->*     ->        |
and     bitand  bitor   compl   new<%%> delete<%%>                         |
not     or      xor     and_eq  not_eq  or_eq   xor_eq                    |
```

literal:

integer-literal
character-literal
floating-literal
string-literal
boolean-literal

integer-literal:

decimal-literal $integer\text{-}suffix_{opt}$
octal-literal $integer\text{-}suffix_{opt}$
hexadecimal-literal $integer\text{-}suffix_{opt}$

decimal-literal:

nonzero-digit
decimal-literal digit

octal-literal:

`0`
octal-literal octal-digit

hexadecimal-literal:

`0x` *hexadecimal-digit*
`0X` *hexadecimal-digit*
hexadecimal-literal hexadecimal-digit

nonzero-digit: one of
```
1  2  3  4  5  6  7  8  9
```

octal-digit: one of
```
0  1  2  3  4  5  6  7
```

hexadecimal-digit: one of
```
0  1  2  3  4  5  6  7  8  9
a  b  c  d  e  f
A  B  C  D  E  F
```

integer-suffix:
unsigned-suffix long-suffix$_{opt}$
long-suffix unsigned-suffix$_{opt}$

unsigned-suffix: one of
```
u  U
```

long-suffix: one of
```
l  L
```

character-literal:
'*c-char-sequence*'
L'*c-char-sequence*'

c-char-sequence:
c-char
c-char-sequence c-char

c-char:
any member of the source character set except
the single-quote ', backslash \, or new-line character
escape-sequence

escape-sequence:
simple-escape-sequence
octal-escape-sequence
hexadecimal-escape-sequence

simple-escape-sequence: one of
```
\'  \"  \?  \\
\a  \b  \f  \n  \r  \t  \v
```

octal-escape-sequence:
\ *octal-digit*
octal-escape-sequence octal-digit

hexadecimal-escape-sequence:
\x *hexadecimal-digit*
hexadecimal-escape-sequence hexadecimal-digit

floating-constant:
fractional-constant exponent-part$_{opt}$ *floating-suffix*$_{opt}$
digit-sequence exponent-part floating-suffix$_{opt}$

fractional-constant:
digit-sequence$_{opt}$. *digit-sequence*
digit-sequence .

exponent-part:
e *sign*$_{opt}$ *digit-sequence*
E *sign*$_{opt}$ *digit-sequence*

sign: one of
`+` `-`

digit-sequence:
digit
digit-sequence digit

floating-suffix: one of
`f` `l` `F` `L`

string-literal:
"*s-char-sequence*$_{opt}$"
L"*s-char-sequence*$_{opt}$"

s-char-sequence:
s-char
s-char-sequence s-char

s-char:
any member of the source character set except
the double-quote ", backslash \, or new-line character
escape-sequence

boolean-literal:
`false`
`true`

A.3 Basic concepts [gram.basic]

translation unit:
declaration-seq$_{opt}$

A.4 Expressions [gram.expr]

primary-expression:
literal
`this`
`::` *identifier*
`::` *operator-function-id*
`::` *qualified-id*
`(` *expression* `)`
id-expression

id-expression:
unqualified-id
qualified-id

unqualified-id:
identifier
operator-function-id
conversion-function-id
`~` *class-name*

qualified-id:
nested-name-specifier unqualified-id

postfix-expression:
primary-expression
postfix-expression `[` *expression* `]`
postfix-expression `(` *expression-list*$_{opt}$ `)`
simple-type-specifier `(` *expression-list*$_{opt}$ `)`
postfix-expression `.` *id-expression*
postfix-expression `->` *id-expression*
postfix-expression `++`
postfix-expression `--`
`dynamic_cast` `<` *type-id* `>` `(` *expression* `)`
`static_cast` `<` *type-id* `>` `(` *expression* `)`
`reinterpret_cast` `<` *type-id* `>` `(` *expression* `)`
`const_cast` `<` *type-id* `>` `(` *expression* `)`
`typeid` `(` *expression* `)`
`typeid` `(` *type-id* `)`

expression-list:
assignment-expression
expression-list `,` *assignment-expression*

unary-expression:
postfix-expression
`++` *unary-expression*
`--` *unary-expression*
unary-operator cast-expression
`sizeof` *unary-expression*
`sizeof` `(` *type-id* `)`
new-expression
delete-expression

unary-operator: one of
`*` `&` `+` `-` `!` `~`

new-expression:
`::`$_{opt}$ `new` *new-placement*$_{opt}$ *new-type-id new-initializer*$_{opt}$
`::`$_{opt}$ `new` *new-placement*$_{opt}$ `(` *type-id* `)` *new-initializer*$_{opt}$

new-placement:
`(` *expression-list* `)`

new-type-id:
type-specifier-seq new-declarator$_{opt}$

new-declarator:
`*` *cv-qualifier-seq*$_{opt}$ *new-declarator*$_{opt}$
`::`$_{opt}$ *nested-name-specifier* `*` *cv-qualifier-seq*$_{opt}$ *new-declarator*$_{opt}$
direct-new-declarator

direct-new-declarator:
`[` *expression* `]`
direct-new-declarator `[` *constant-expression* `]`

new-initializer:
`(` *expression-list*$_{opt}$ `)`

delete-expression:
`::`$_{opt}$ `delete` *cast-expression*
`::`$_{opt}$ `delete [ ]` *cast-expression*

cast-expression:
unary-expression
(*type-id*) *cast-expression*

pm-expression:
cast-expression
pm-expression `.*` *cast-expression*
pm-expression `->*` *cast-expression*

multiplicative-expression:
pm-expression
multiplicative-expression `*` *pm-expression*
multiplicative-expression `/` *pm-expression*
multiplicative-expression `%` *pm-expression*

additive-expression:
multiplicative-expression
additive-expression `+` *multiplicative-expression*
additive-expression `-` *multiplicative-expression*

shift-expression:
additive-expression
shift-expression `<<` *additive-expression*
shift-expression `>>` *additive-expression*

relational-expression:
shift-expression
relational-expression `<` *shift-expression*
relational-expression `>` *shift-expression*
relational-expression `<=` *shift-expression*
relational-expression `>=` *shift-expression*

equality-expression:
relational-expression
equality-expression `==` *relational-expression*
equality-expression `!=` *relational-expression*

and-expression:
equality-expression
and-expression `&` *equality-expression*

exclusive-or-expression:
and-expression
exclusive-or-expression `^` *and-expression*

inclusive-or-expression:
exclusive-or-expression
inclusive-or-expression `|` *exclusive-or-expression*

logical-and-expression:
inclusive-or-expression
logical-and-expression `&&` *inclusive-or-expression*

logical-or-expression:
logical-and-expression
logical-or-expression `||` *logical-and-expression*

conditional-expression:
logical-or-expression
logical-or-expression `?` *expression* `:` *assignment-expression*

assignment-expression:
conditional-expression
unary-expression assignment-operator assignment-expression
throw-expression

assignment-operator: one of
`=` `*=` `/=` `%=` `+=` `-=` `>>=` `<<=` `&=` `^=` `|=`

expression:
assignment-expression
expression `,` *assignment-expression*

constant-expression:
conditional-expression

A.5 Statements [gram.stmt.stmt]

statement:
labeled-statement
expression-statement
compound-statement
selection-statement
iteration-statement
jump-statement
declaration-statement
try-block

labeled-statement:
identifier `:` *statement*
`case` *constant-expression* `:` *statement*
`default` `:` *statement*

expression-statement:
expression$_{opt}$ `;`

compound-statement:
`{` *statement-seq*$_{opt}$ `}`

statement-seq:
statement
statement-seq statement

selection-statement:
`if` `(` *condition* `)` *statement*
`if` `(` *condition* `)` *statement* `else` *statement*
`switch` `(` *condition* `)` *statement*

condition:
expression
type-specifier-seq declarator `=` *assignment-expression*

iteration-statement:
`while` `(` *condition* `)` *statement*
`do` *statement* `while` `(` *expression* `)` `;`
`for` `(` *for-init-statement condition*$_{opt}$ `;` *expression*$_{opt}$ `)` *statement*

for-init-statement:
expression-statement
declaration-statement

jump-statement:
`break ;`
`continue ;`
`return` *expression*$_{opt}$ `;`
`goto` *identifier* `;`

declaration-statement:
declaration

A.6 Declarations [gram.dcl.dcl]

declaration:
decl-specifier-seq$_{opt}$ *init-declarator-list*$_{opt}$ `;`
function-definition
template-declaration
asm-definition
linkage-specification
namespace-definition
namespace-alias-definition
using-declaration
using-directive

decl-specifier-seq$_{opt}$ *init-declarator-list*$_{opt}$ `;`

decl-specifier:
storage-class-specifier
type-specifier
function-specifier
`friend`
`typedef`

decl-specifier-seq:
decl-specifier-seq$_{opt}$ *decl-specifier*

storage-class-specifier:
`auto`
`register`
`static`
`extern`
`mutable`

function-specifier:
`inline`
`virtual`
`explicit`

typedef-name:
identifier

type-specifier:
simple-type-specifier
class-specifier
enum-specifier
elaborated-type-specifier
cv-qualifier

simple-type-specifier:
`::`$_{\texttt{opt}}$ *nested-name-specifier*$_{opt}$ *type-name*
`char`
`wchar_t`
`bool`
`short`
`int`
`long`
`signed`
`unsigned`
`float`
`double`
`void`

type-name:
class-name
enum-name
typedef-name

elaborated-type-specifier:
class-key `::`$_{\texttt{opt}}$ *nested-name-specifier*$_{opt}$ *identifier*
`enum` *::*$_{opt}$ *nested-name-specifier*$_{opt}$ *identifier*

class-key:
`class`
`struct`
`union`

enum-name:
identifier

enum-specifier:
`enum` *identifier*$_{opt}$ `{` *enumerator-list*$_{opt}$ `}`

enumerator-list:
enumerator-definition
enumerator-list `,` *enumerator-definition*

enumerator-definition:
enumerator
enumerator `=` *constant-expression*

enumerator:
identifier

original-namespace-name:
identifier

namespace-definition:
named-namespace-definition |
unnamed-namespace-definition |

named-namespace-definition: |
original-namespace-definition
extension-namespace-definition

original-namespace-definition: *
`namespace` *identifier* `{` *namespace-body* `}`

extension-namespace-definition:
`namespace` *original-namespace-name* `{` *namespace-body* `}`

unnamed-namespace-definition:
`namespace {` *namespace-body* `}`

namespace-body:
declaration-seq$_{opt}$

id-expression:
unqualified-id
qualified-id

nested-name-specifier:
class-or-namespace-name `::` *nested-name-specifier*$_{opt}$

class-or-namespace-name:
class-name
namespace-name

namespace-name:
original-namespace-name
namespace-alias

namespace-alias:
identifier

namespace-alias-definition:
`namespace` *identifier* `=` *qualified-namespace-specifier* `;`

qualified-namespace-specifier:
::$_{opt}$ *nested-name-specifier*$_{opt}$ *class-or-namespace-name*

using-declaration:
`using ::`$_{opt}$ *nested-name-specifier unqualified-id* `;`
`using ::` *unqualified-id* `;`

using-directive:
`using namespace ::`$_{opt}$ *nested-name-specifier*$_{opt}$ *namespace-name* `;`

asm-definition:
`asm (` *string-literal* `) ;`

linkage-specification:
`extern` *string-literal* `{` *declaration-seq*$_{opt}$ `}`
`extern` *string-literal declaration*

declaration-seq:
declaration
declaration-seq declaration

A.7 Declarators [gram.dcl.decl]

init-declarator-list:
init-declarator
init-declarator-list , *init-declarator*

init-declarator:
declarator initializer$_{opt}$

declarator:
direct-declarator
ptr-operator declarator

direct-declarator:
declarator-id
direct-declarator (*parameter-declaration-clause*) *cv-qualifier-seq*$_{opt}$ *exception-specification*$_{opt}$
direct-declarator [*constant-expression*$_{opt}$]
(*declarator*)

ptr-operator:
* *cv-qualifier-seq*$_{opt}$
&
::$_{opt}$ *nested-name-specifier* * *cv-qualifier-seq*$_{opt}$

cv-qualifier-seq:
cv-qualifier cv-qualifier-seq$_{opt}$

cv-qualifier:
`const`
`volatile`

declarator-id:
id-expression
nested-name-specifier$_{opt}$ *type-name*

type-id:
type-specifier-seq abstract-declarator$_{opt}$

type-specifier-seq:
type-specifier type-specifier-seq$_{opt}$

abstract-declarator:
ptr-operator abstract-declarator$_{opt}$
direct-abstract-declarator

direct-abstract-declarator:
direct-abstract-declarator$_{opt}$ (*parameter-declaration-clause*) *cv-qualifier-seq*$_{opt}$ *exception-specification*$_{opt}$
direct-abstract-declarator$_{opt}$ [*constant-expression*$_{opt}$]
(*abstract-declarator*)

parameter-declaration-clause:
parameter-declaration-list$_{opt}$ ...$_{opt}$
parameter-declaration-list , ...

parameter-declaration-list:
parameter-declaration
parameter-declaration-list `,` *parameter-declaration*

parameter-declaration:
decl-specifier-seq declarator
decl-specifier-seq declarator `=` *assignment-expression*
decl-specifier-seq abstract-declarator$_{opt}$
decl-specifier-seq abstract-declarator$_{opt}$ `=` *assignment-expression*

function-definition:
decl-specifier-seq$_{opt}$ *declarator ctor-initializer*$_{opt}$ *function-body*

function-body:
compound-statement

initializer:
`=` *initializer-clause*
`(` *expression-list* `)`

initializer-clause:
assignment-expression
`{` *initializer-list* `,`$_{opt}$ `}`
`{ }`

initializer-list:
initializer-clause
initializer-list `,` *initializer-clause*

A.8 Classes [gram.class]

class-name:
identifier
template-id

class-specifier:
class-head `{` *member-specification*$_{opt}$ `}`

class-head:
class-key identifier$_{opt}$ *base-clause*$_{opt}$
class-key nested-name-specifier identifier base-clause$_{opt}$

class-key:
`class`
`struct`
`union`

member-specification:
member-declaration member-specification$_{opt}$
access-specifier `:` *member-specification*$_{opt}$

member-declaration:
decl-specifier-seq$_{opt}$ *member-declarator-list*$_{opt}$ `;`
function-definition `;`$_{opt}$
qualified-id `;`
using-declaration

member-declarator-list:
member-declarator
member-declarator-list , *member-declarator*

member-declarator:
declarator pure-specifier$_{opt}$
declarator constant-initializer$_{opt}$
identifier$_{opt}$: *constant-expression*

pure-specifier:
= 0

constant-initializer:
= *constant-expression*

A.9 Derived classes [gram.class.derived]

base-clause:
: *base-specifier-list*

base-specifier-list:
base-specifier
base-specifier-list , *base-specifier*

base-specifier:
::$_{opt}$ *nested-name-specifier*$_{opt}$ *class-name*
virtual access-specifier$_{opt}$ ::$_{opt}$ *nested-name-specifier*$_{opt}$ *class-name*
access-specifier virtual$_{opt}$ ::$_{opt}$ *nested-name-specifier*$_{opt}$ *class-name*

access-specifier:
private
protected
public

A.10 Special member functions [gram.special]

class-name (*expression-list*$_{opt}$)

conversion-function-id:
operator *conversion-type-id*

conversion-type-id:
type-specifier-seq conversion-declarator$_{opt}$

conversion-declarator:
ptr-operator conversion-declarator$_{opt}$

ctor-initializer:
: *mem-initializer-list*

mem-initializer-list:
mem-initializer
mem-initializer , *mem-initializer-list*

mem-initializer:
::$_{opt}$ *nested-name-specifier*$_{opt}$ *class-name* (*expression-list*$_{opt}$)
identifier (*expression-list*$_{opt}$)

A.11 Overloading [gram.over]

operator-function-id:
`operator` *operator*

operator: one of

```
new  delete    new[]     delete[]
+    -    *    /    %    ^    &    |    ~
!    =    <    >    +=   -=   *=   /=   %=
^=   &=   |=   <<   >>   >>=  <<=  ==   !=
<=   >=   &&   ||   ++   --   ,    ->*  ->
()   []
```

A.12 Templates [gram.temp]

template-declaration:
`template <` *template-parameter-list* `>` *declaration*

template-parameter-list:
template-parameter
template-parameter-list `,` *template-parameter*

template-id:
template-name `<` *template-argument-list* `>`

template-name:
identifier

template-argument-list:
template-argument
template-argument-list `,` *template-argument*

template-argument:
assignment-expression
type-id
template-name

elaborated-type-specifier:
...
`typename` $::_{opt}$ *nested-name-specifier identifier full-template-argument-list*$_{opt}$ `;`

full-template-argument-list:
`<` *template-argument-list* `>`

explicit-instantiation:
`template` *inst* `;`

inst:
class-key template-id
type-specifier-seq template-id `(` *parameter-declaration-clause* `)`

specialization:
declaration

template-parameter:
type-parameter
parameter-declaration

type-parameter:
`class` *identifier*$_{opt}$
`class` *identifier*$_{opt}$ `=` *type-id*
`typename` *identifier*$_{opt}$
`typename` *identifier*$_{opt}$ `=` *type-id*
`template <` *template-parameter-list* `> class` *identifier*$_{opt}$
`template <` *template-parameter-list* `> class` *identifier*$_{opt}$ `=` *template-name*

A.13 Exception handling [gram.except]

try-block:
`try` *compound-statement handler-seq*

handler-seq:
handler handler-seq$_{opt}$

handler:
`catch (` *exception-declaration* `)` *compound-statement*

exception-declaration:
type-specifier-seq declarator
type-specifier-seq abstract-declarator
type-specifier-seq
`...`

throw-expression:
`throw` *assignment-expression*$_{\texttt{opt}}$

exception-specification:
`throw (` *type-id-list*$_{opt}$ `)`

type-id-list:
type-id
type-id-list `,` *type-id*

Annex B (informative)
Implementation quantities [limits]

1 Because computers are finite, C++ implementations are inevitably limited in the size of the programs they can successfully process. Every implementation shall

Box 146

This clause is non-normative, which means that this sentence must be restated in elsewhere as a normative requirement on implementations.

document those limitations where known. This documentation may cite fixed limits where they exist, say how to compute variable limits as a function of available resources, or say that fixed limits do not exist or are unknown.

2 The limits may constrain quantities that include those described below or others. The bracketed number following each quantity is recommended as the minimum for that quantity. However, these quantities are only guidelines and do not determine compliance.

— Nesting levels of compound statements, iteration control structures, and selection control structures [256].

— Nesting levels of conditional inclusion [256].

— Pointer, array, and function declarators (in any combination) modifying an arithmetic, structure, union, or incomplete type in a declaration [256].

— Nesting levels of parenthesized expressions within a full expression [256].

— Number of initial characters in an internal identifier or macro name [1 024].

— Number of initial characters in an external identifier [1 024].

— External identifiers in one translation unit [65 536].

— Identifiers with block scope declared in one block [1 024].

— Macro identifiers simultaneously defined in one transation unit [65 536].

— Parameters in one function definition [256].

— Arguments in one function call [256].

— Parameters in one macro definition [256].

— Arguments in one macro invocation [256].

— Characters in one logical source line [65 536].

— Characters in a character string literal or wide string literal (after concatenation) [65 536].

— Size of an object [262 144].

Box 147
This is trivial for some implementations to meet and very hard for others.

— Nesting levels for `#include` files [256].
— Case labels for a `switch` statement (excluding those for any nested `switch` statements) [16 384].
— Data members in a single class, structure, or union [16 384].
— Enumeration constants in a single enumeration [4 096].
— Levels of nested class, structure, or union definitions in a single *struct-declaration-list* [256].
— Functions registered by `atexit()` [32].
— Direct and indirect base classes [16 384].
— Direct base classes for a single class [1 024].
— Members declared in a single class [4 096].
— Final overriding virtual functions in a class, accessible or not [16 384].

Box 148
I'm not quite sure what this means, but it was passed in Munich in this form.

— Direct and indirect virtual bases of a class [1 024].
— Static members of a class [1 024].
— Friend declarations in a class [4 096].
— Access control declarations in a class [4 096].
— Member initializers in a constructor definition [6 144].
— Scope qualifications of one identifier [256].
— Nested external specifications [1 024].
— Template arguments in a template declaration [1 024].
— Recursively nested template instantiations [17].
— Handlers per `try` block [256].
— Throw specifications on a single function declaration [256].

Annex C (informative)
Compatibility [diff]

1 This Annex summarizes the evolution of C++ since the first edition of *The C++ Programming Language* and explains in detail the differences between C++ and C. Because the C language as described by this International Standard differs from the dialects of Classic C used up till now, we discuss the differences between C++ and ISO C as well as the differences between C++ and Classic C.

2 C++ is based on C (K&R78) and adopts most of the changes specified by the ISO C standard. Converting programs among C++, K&R C, and ISO C may be subject to vicissitudes of expression evaluation. All differences between C++ and ISO C can be diagnosed by a compiler. With the exceptions listed in this Annex, programs that are both C++ and ISO C have the same meaning in both languages.

C.1 Extensions [diff.c]

1 This subclause summarizes the major extensions to C provided by C++.

C.1.1 C++ features available in 1985 [diff.early]

1 This subclause summarizes the extensions to C provided by C++ in the 1985 version of its manual:

2 The types of function parameters can be specified (8.3.5) and will be checked (5.2.2). Type conversions will be performed (5.2.2). This is also in ISO C.

3 Single-precision floating point arithmetic may be used for `float` expressions; 3.8.1 and 4.8. This is also in ISO C.

4 Function names can be overloaded; 13.

5 Operators can be overloaded; 13.4.

6 Functions can be inline substituted; 7.1.2.

7 Data objects can be `const`; 7.1.5. This is also in ISO C.

8 Objects of reference type can be declared; 8.3.2 and 8.5.3.

9 A free store is provided by the `new` and `delete` operators; 5.3.4, 5.3.5.

10 Classes can provide data hiding (11), guaranteed initialization (12.1), user-defined conversions (12.3), and dynamic typing through use of virtual functions (10.3).

11 The name of a class or enumeration is a type name; 9.

12 A pointer to any non-`const` and non-`volatile` object type can be assigned to a `void*`; 4.10. This is also in ISO C.

13 A pointer to function can be assigned to a `void*`; 4.10.

14 A declaration within a block is a statement; 6.7.

15 Anonymous unions can be declared; 9.6.

C.1.2 C++ features added since 1985 [diff.c++]

1 This subclause summarizes the major extensions of C++ since the 1985 version of this manual:

2 A class can have more than one direct base class (multiple inheritance); 10.1.

3 Class members can be `protected`; 11 .

4 Pointers to class members can be declared and used; 8.3.3, 5.5.

5 Operators `new` and `delete` can be overloaded and declared for a class; 5.3.4, 5.3.5, 12.5. This allows the "assignment to `this`" technique for class specific storage management to be removed to the anachronism subclause; C.3.3.

6 Objects can be explicitly destroyed; 12.4.

7 Assignment and initialization are defined as memberwise assignment and initialization; 12.8.

8 The `overload` keyword was made redundant and moved to the anachronism subclause; C.3.

9 General expressions are allowed as initializers for static objects; 8.5.

10 Data objects can be `volatile`; 7.1.5. Also in ISO C.

11 Initializers are allowed for `static` class members; 9.5.

12 Member functions can be `static`; 9.5.

13 Member functions can be `const` and `volatile`; 9.4.2.

14 Linkage to non-C++ program fragments can be explicitly declared; 7.5.

15 Operators `->`, `->*`, and `,` can be overloaded; 13.4.

16 Classes can be abstract; 10.4.

17 Prefix and postfix application of `++` and `--` on a user-defined type can be distinguished.

18 Templates; 14.

19 Exception handling; 15.

20 The `bool` type (3.8.1).

C.2 C++ and ISO C [diff.iso]

1 The subclauses of this subclause list the differences between C++ and ISO C, by the chapters of this document.

C.2.1 Clause 2: lexical conventions [diff.lex]

Subclause 2.2

1 **Change:** C++ style comments (`//`) are added
A pair of slashes now introduce a one-line comment.
Rationale: This style of comments is a useful addition to the language.
Effect on original feature: Change to semantics of well-defined feature. A valid ISO C expression containing a division operator followed immediately by a C-style comment will now be treated as a C++ style comment. For example:

```
{
    int a = 4;
    int b = 8 //* divide by a*/ a;
    +a;
}
```

Difficulty of converting: Syntactic transformation. Just add white space after the division operator.
How widely used: The token sequence `//*` probably occurs very seldom.

Subclause 2.8

2 **Change:** New Keywords
New keywords are added to C++; see 2.8.
Rationale: These keywords were added in order to implement the new semantics of C++.
Effect on original feature: Change to semantics of well-defined feature. Any ISO C programs that used any of these keywords as identifiers are not valid C++ programs.
Difficulty of converting: Syntactic transformation. Converting one specific program is easy. Converting a large collection of related programs takes more work.
How widely used: Common.

Subclause 2.9.2

3 **Change:** Type of character literal is changed from `int` to `char`
Rationale: This is needed for improved overloaded function argument type matching. For example:

```
int function( int i );
int function( char c );

function( 'x' );
```

It is preferable that this call match the second version of function rather than the first.
Effect on original feature: Change to semantics of well-defined feature. ISO C programs which depend on

```
sizeof('x') == sizeof(int)
```

will not work the same as C++ programs.
Difficulty of converting: Simple.
How widely used: Programs which depend upon `sizeof('x')` are probably rare.

C.2.2 Clause 3: basic concepts [diff.basic]

Subclause 3.1

1 **Change:** C++ does not have "tentative definitions" as in C
E.g., at file scope,

```
int i;
int i;
```

is valid in C, invalid in C++. This makes it impossible to define mutually referential file-local static objects, if initializers are restricted to the syntactic forms of C. For example,

```
struct X { int i; struct X *next; };

static struct X a;
static struct X b = { 0, &a };
static struct X a = { 1, &b };
```

Rationale: This avoids having different initialization rules for built-in types and user-defined types.
Effect on original feature: Deletion of semantically well-defined feature.
Difficulty of converting: Semantic transformation. In C++, the initializer for one of a set of mutually-referential file-local static objects must invoke a function call to achieve the initialization.
How widely used: Seldom.

Subclause 3.3

2 **Change:** A `struct` is a scope in C++, not in C
Rationale: Class scope is crucial to C++, and a struct is a class.
Effect on original feature: Change to semantics of well-defined feature.
Difficulty of converting: Semantic transformation.
How widely used: C programs use `struct` extremely frequently, but the change is only noticeable when `struct`, enumeration, or enumerator names are referred to outside the `struct`. The latter is probably rare.

Subclause 3.5 [also 7.1.5]

3 **Change:** A name of file scope that is explicitly declared `const`, and not explicitly declared `extern`, has internal linkage, while in C it would have external linkage
Rationale: Because `const` objects can be used as compile-time values in C++, this feature urges programmers to provide explicit initializer values for each `const`. This feature allows the user to put `const` objects in header files that are included in many compilation units.
Effect on original feature: Change to semantics of well-defined feature.
Difficulty of converting: Semantic transformation
How widely used: Seldom

Subclause 3.6

4 **Change:** Main cannot be called recursively and cannot have its address taken
Rationale: The main function may require special actions.
Effect on original feature: Deletion of semantically well-defined feature
Difficulty of converting: Trivial: create an intermediary function such as `mymain(argc, argv)`.
How widely used: Seldom

Subclause 3.8

5 **Change:** C allows "compatible types" in several places, C++ does not
For example, otherwise-identical `struct` types with different tag names are "compatible" in C but are distinctly different types in C++.
Rationale: Stricter type checking is essential for C++.
Effect on original feature: Deletion of semantically well-defined feature.
Difficulty of converting: Semantic transformation The "typesafe linkage" mechanism will find many, but not all, of such problems. Those problems not found by typesafe linkage will continue to function properly, according to the "layout compatibility rules" of this International Standard.
How widely used: Common.

Subclause 4.10

6 **Change:** Converting `void*` to a pointer-to-object type requires casting

```
char a[10];
void *b=a;
void foo() {
char *c=b;
}
```

ISO C will accept this usage of pointer to void being assigned to a pointer to object type. C++ will not.
Rationale: C++ tries harder than C to enforce compile-time type safety.
Effect on original feature: Deletion of semantically well-defined feature.

Difficulty of converting: Could be automated. Violations will be diagnosed by the C++ translator. The fix is to add a cast. For example:

```
char *c = (char *) b;
```

How widely used: This is fairly widely used but it is good programming practice to add the cast when assigning pointer-to-void to pointer-to-object. Some ISO C translators will give a warning if the cast is not used.

Subclause 4.10

7 **Change:** Only pointers to non-const and non-volatile objects may be implicitly converted to `void*`
Rationale: This improves type safety.
Effect on original feature: Deletion of semantically well-defined feature.
Difficulty of converting: Could be automated. A C program containing such an implicit conversion from (e.g.) pointer-to-const-object to void* will receive a diagnostic message. The correction is to add an explicit cast.
How widely used: Seldom.

C.2.3 Clause 5: expressions [diff.expr]

Subclause 5.2.2

1 **Change:** Implicit declaration of functions is not allowed
Rationale: The type-safe nature of C++.
Effect on original feature: Deletion of semantically well-defined feature. Note: the original feature was labeled as "obsolescent" in ISO C.
Difficulty of converting: Syntactic transformation. Facilities for producing explicit function declarations are fairly widespread commercially.
How widely used: Common.

Subclause 5.3.3, 5.4

2 **Change:** Types must be declared in declarations, not in expressions
In C, a sizeof expression or cast expression may create a new type. For example,

```
p = (void*)(struct x {int i;} *)0;
```

declares a new type, struct x .
Rationale: This prohibition helps to clarify the location of declarations in the source code.
Effect on original feature: Deletion of a semantically well-defined feature.
Difficulty of converting: Syntactic transformation.
How widely used: Seldom.

C.2.4 Clause 6: statements [diff.stat]

Subclause 6.4.2, 6.6.4 (`switch` and `goto` statements)

1 **Change:** It is now invalid to jump past a declaration with explicit or implicit initializer (except across entire block not entered)
Rationale: Constructors used in initializers may allocate resources which need to be de-allocated upon leaving the block. Allowing jump past initializers would require complicated run-time determination of allocation. Furthermore, any use of the uninitialized object could be a disaster. With this simple compile-time rule, C++ assures that if an initialized variable is in scope, then it has assuredly been initialized.
Effect on original feature: Deletion of semantically well-defined feature.

Difficulty of converting: Semantic transformation.
How widely used: Seldom.

Subclause 6.6.3

2 **Change:** It is now invalid to return (explicitly or implicitly) from a function which is declared to return a value without actually returning a value
Rationale: The caller and callee may assume fairly elaborate return-value mechanisms for the return of class objects. If some flow paths execute a return without specifying any value, the compiler must embody many more complications. Besides, promising to return a value of a given type, and then not returning such a value, has always been recognized to be a questionable practice, tolerated only because very-old C had no distinction between void functions and int functions.
Effect on original feature: Deletion of semantically well-defined feature.
Difficulty of converting: Semantic transformation. Add an appropriate return value to the source code, e.g. zero.
How widely used: Seldom. For several years, many existing C compilers have produced warnings in this case.

C.2.5 Clause 7: declarations [diff.dcl]

Subclause 7.1.1

1 **Change:** In C++, the `static` or `extern` specifiers can only be applied to names of objects or functions
Using these specifiers with type declarations is illegal in C++. In C, these specifiers are ignored when used on type declarations. Example:

```
static struct S {      // valid C, invalid in C++
int i;
// ...
};
```

Rationale: Storage class specifiers don't have any meaning when associated with a type. In C++, class members can be defined with the `static` storage class specifier. Allowing storage class specifiers on type declarations could render the code confusing for users.
Effect on original feature: Deletion of semantically well-defined feature.
Difficulty of converting: Syntactic transformation.
How widely used: Seldom.

Subclause 7.1.3

2 **Change:** A C++ typedef name must be different from any class type name declared in the same scope (except if the typedef is a synonym of the class name with the same name). In C, a typedef name and a struct tag name declared in the same scope can have the same name (because they have different name spaces)
Example:

```
typedef struct name1 { /*...*/ } name1; // valid C and C++
struct name { /*...*/ };
typedef int name;                  // valid C, invalid C++
```

Rationale: For ease of use, C++ doesn't require that a type name be prefixed with the keywords `class`, `struct` or `union` when used in object declarations or type casts. Example:

```
class name { /*...*/ };
name i;                   // i has type 'class name'
```

Effect on original feature: Deletion of semantically well-defined feature.

Difficulty of converting: Semantic transformation. One of the 2 types has to be renamed.
How widely used: Seldom.

Subclause 7.1.5 [see also 3.5]

3 **Change:** const objects must be initialized in C++ but can be left uninitialized in C
Rationale: A const object cannot be assigned to so it must be initialized to hold a useful value.
Effect on original feature: Deletion of semantically well-defined feature.
Difficulty of converting: Semantic transformation.
How widely used: Seldom.

Subclause 7.2

4 **Change:** C++ objects of enumeration type can only be assigned values of the same enumeration type. In C, objects of enumeration type can be assigned values of any integral type
Example:

```
enum color { red, blue, green };
color c = 1;   // valid C, invalid C++
```

Rationale: The type-safe nature of C++.
Effect on original feature: Deletion of semantically well-defined feature.
Difficulty of converting: Syntactic transformation. (The type error produced by the assignment can be automatically corrected by applying an explicit cast.)
How widely used: Common.

Subclause 7.2

5 **Change:** In C++, the type of an enumerator is its enumeration. In C, the type of an enumerator is `int`.
Example:

```
enum e { A };
sizeof(A) == sizeof(int)  // in C
sizeof(A) == sizeof(e)    // in C++
/* and sizeof(int) is not necessary equal to sizeof(e) */
```

Rationale: In C++, an enumeration is a distinct type.
Effect on original feature: Change to semantics of well-defined feature.
Difficulty of converting: Semantic transformation.
How widely used: Seldom. The only time this affects existing C code is when the size of an enumerator is taken. Taking the size of an enumerator is not a common C coding practice.

C.2.6 Clause 8: declarators [diff.decl]

Subclause 8.3.5

1 **Change:** In C++, a function declared with an empty parameter list takes no arguments.
In C, an empty parameter list means that the number and type of the function arguments are unknown"
Example:

```
int f();  // means   int f(void)     in C++
                //          int f(unknown)  in C
```

Rationale: This is to avoid erroneous function calls (i.e. function calls with the wrong number or type of arguments).
Effect on original feature: Change to semantics of well-defined feature. This feature was marked as

"obsolescent" in C.
Difficulty of converting: Syntactic transformation. The function declarations using C incomplete declaration style must be completed to become full prototype declarations. A program may need to be updated further if different calls to the same (non-prototype) function have different numbers of arguments or if the type of corresponding arguments differed.
How widely used: Common.

Subclause 8.3.5 [see 5.3.3]

2 **Change:** In C++, types may not be defined in return or parameter types. In C, these type definitions are allowed
Example:

```
void f( struct S { int a; } arg ) {}     // valid C, invalid C++
enum E { A, B, C } f() {}                // valid C, invalid C++
```

Rationale: When comparing types in different compilation units, C++ relies on name equivalence when C relies on structural equivalence. Regarding parameter types: since the type defined in an parameter list would be in the scope of the function, the only legal calls in C++ would be from within the function itself.
Effect on original feature: Deletion of semantically well-defined feature.
Difficulty of converting: Semantic transformation. The type definitions must be moved to file scope, or in header files.
How widely used: Seldom. This style of type definitions is seen as poor coding style.

Subclause 8.4

3 **Change:** In C++, the syntax for function definition excludes the "old-style" C function. In C, "old-style" syntax is allowed, but deprecated as "obsolescent."
Rationale: Prototypes are essential to type safety.
Effect on original feature: Deletion of semantically well-defined feature.
Difficulty of converting: Syntactic transformation.
How widely used: Common in old programs, but already known to be obsolescent.

Subclause 8.5.2

4 **Change:** In C++, when initializing an array of character with a string, the number of characters in the string (including the terminating `'\0'`) must not exceed the number of elements in the array. In C, an array can be initialized with a string even if the array is not large enough to contain the string terminating `'\0'`
Example:

```
char array[4] = "abcd";    // valid C, invalid C++
```

Rationale: When these non-terminated arrays are manipulated by standard string routines, there is potential for major catastrophe.
Effect on original feature: Deletion of semantically well-defined feature.
Difficulty of converting: Semantic transformation. The arrays must be declared one element bigger to contain the string terminating `'\0'`.
How widely used: Seldom. This style of array initialization is seen as poor coding style.

C.2.7 Clause 9: classes [diff.class]

Subclause 9.1 [see also 7.1.3]

1 **Change:** In C++, a class declaration introduces the class name into the scope where it is declared and hides any object, function or other declaration of that name in an enclosing scope. In C, an inner scope declaration of a struct tag name never hides the name of an object or function in an outer scope
Example:

```
int x[99];
void f()
{
        struct x { int a; };
        sizeof(x);  /* size of the array in C    */
        /* size of the struct in C++ */
}
```

Rationale: This is one of the few incompatibilities between C and C++ that can be attributed to the new C++ name space definition where a name can be declared as a type and as a nontype in a single scope causing the nontype name to hide the type name and requiring that the keywords `class`, `struct`, `union` or `enum` be used to refer to the type name. This new name space definition provides important notational conveniences to C++ programmers and helps making the use of the user-defined types as similar as possible to the use of built-in types. The advantages of the new name space definition were judged to outweigh by far the incompatibility with C described above.
Effect on original feature: Change to semantics of well-defined feature.
Difficulty of converting: Semantic transformation. If the hidden name that needs to be accessed is at global scope, the `::` C++ operator can be used. If the hidden name is at block scope, either the type or the struct tag has to be renamed.
How widely used: Seldom.

Subclause 9.8

2 **Change:** In C++, the name of a nested class is local to its enclosing class. In C the name of the nested class belongs to the same scope as the name of the outermost enclosing class
Example:

```
struct X {
        struct Y { /* ... */ } y;
};
struct Y yy;     // valid C, invalid C++
```

Rationale: C++ classes have member functions which require that classes establish scopes. The C rule would leave classes as an incomplete scope mechanism which would prevent C++ programmers from maintaining locality within a class. A coherent set of scope rules for C++ based on the C rule would be very complicated and C++ programmers would be unable to predict reliably the meanings of nontrivial examples involving nested or local functions.
Effect on original feature: Change of semantics of well-defined feature.
Difficulty of converting: Semantic transformation. To make the struct type name visible in the scope of the enclosing struct, the struct tag could be declared in the scope of the enclosing struct, before the enclosing struct is defined. Example:

```
struct Y;  // struct Y and struct X are at the same scope
struct X {
        struct Y { /* ... */ } y;
};
```

All the definitions of C struct types enclosed in other struct definitions and accessed outside the scope of the enclosing struct could be exported to the scope of the enclosing struct. Note: this is a consequence of the difference in scope rules, which is documented at subclause 3.3 above.
How widely used: Seldom.

Subclause 9.10

3 **Change:** In C++, a typedef name may not be redefined in a class declaration after being used in the declaration
Example:

```
typedef int I;
struct S {
        I i;
        int I;       // valid C, invalid C++
};
```

Rationale: When classes become complicated, allowing such a redefinition after the type has been used can create confusion for C++ programmers as to what the meaning of 'I' really is.
Effect on original feature: Deletion of semantically well-defined feature.
Difficulty of converting: Semantic transformation. Either the type or the struct member has to be renamed.
How widely used: Seldom.

C.2.8 Clause 16: preprocessing directives [diff.cpp]

Subclause 16.8 (predefined names)

1 **Change:** Whether `__STDC__` is defined and if so, what its value is, are implementation-defined
Rationale: C++ is not identical to ISO C. Mandating that `__STDC__` be defined would require that translators make an incorrect claim. Each implementation must choose the behavior that will be most useful to its marketplace.
Effect on original feature: Change to semantics of well-defined feature.
Difficulty of converting: Semantic transformation.
How widely used: Programs and headers that reference `__STDC__` are quite common.

C.3 Anachronisms [diff.anac]

1 The extensions presented here may be provided by an implementation to ease the use of C programs as C++ programs or to provide continuity from earlier C++ implementations. Note that each of these features has undesirable aspects. An implementation providing them should also provide a way for the user to ensure that they do not occur in a source file. A C++ implementation is not obliged to provide these features.

2 The word `overload` may be used as a *decl-specifier* (7) in a function declaration or a function definition. When used as a *decl-specifier*, `overload` is a reserved word and cannot also be used as an identifier.

3 The definition of a static data member of a class for which initialization by default to all zeros applies (8.5, 9.5) may be omitted.

4 An old style (that is, pre-ISO C) C preprocessor may be used.

5 An `int` may be assigned to an object of enumeration type.

6 The number of elements in an array may be specified when deleting an array of a type for which there is no destructor; 5.3.5.

7 A single function `operator++()` may be used to overload both prefix and postfix `++` and a single function `operator--()` may be used to overload both prefix and postfix `--`; 13.4.6.

8

C.3.1 Old style function definitions [diff.fct.def]

1 The C function definition syntax

old-function-definition:
decl-specifiers$_{opt}$ *old-function-declarator* *declaration-seq*$_{opt}$ *function-body*

old-function-declarator:
declarator (*parameter-list*$_{opt}$)

parameter-list:
identifier
parameter-list , *identifier*

For example,

```
max(a,b) int b; { return (a<b) ? b : a; }
```

may be used. If a function defined like this has not been previously declared its parameter type will be taken to be `(...)`, that is, unchecked. If it has been declared its type must agree with that of the declaration.

2 Class member functions may not be defined with this syntax.

C.3.2 Old style base class initializer [diff.base.init]

1 In a *mem-initializer*(12.6.2), the *class-name* naming a base class may be left out provided there is exactly one immediate base class. For example,

```
class B {
    // ...
public:
    B (int);
};

class D : public B {
    // ...
    D(int i) : (i) { /* ... */ }
};
```

causes the `B` constructor to be called with the argument `i`.

C.3.3 Assignment to `this` [diff.this]

1 Memory management for objects of a specific class can be controlled by the user by suitable assignments to the `this` pointer. By assigning to the `this` pointer before any use of a member, a constructor can implement its own storage allocation. By assigning the null pointer to `this`, a destructor can avoid the standard deallocation operation for objects of its class. Assigning the null pointer to `this` in a destructor also suppressed the implicit calls of destructors for bases and members. For example,

```
class Z {
    int z[10];
    Z()  { this = my_allocator( sizeof(Z) ); }
    ~Z() { my_deallocator( this ); this = 0; }
};
```

2 On entry into a constructor, `this` is nonnull if allocation has already taken place (as it will have for `auto`, `static`, and member objects) and null otherwise.

3 Calls to constructors for a base class and for member objects will take place (only) after an assignment to `this`. If a base class's constructor assigns to `this`, the new value will also be used by the derived class's constructor (if any).

4 Note that if this anachronism exists either the type of the `this` pointer cannot be a `*const` or the enforcement of the rules for assignment to a constant pointer must be subverted for the `this` pointer.

C.3.4 Cast of bound pointer [diff.bound]

1 A pointer to member function for a particular object may be cast into a pointer to function, for example, `(int(*)())p->f`. The result is a pointer to the function that would have been called using that member function for that particular object. Any use of the resulting pointer is – as ever – undefined.

C.3.5 Nonnested classes [diff.class.nonnested]

1 Where a class is declared within another class and no other class of that name is declared in the program that class can be used as if it was declared outside its enclosing class (exactly as a C `struct`). For example,

```
struct S {
    struct T {
        int a;
    };
    int b;
};

struct T x;     // meaning 'S::T x;'
```

C.4 Standard C library [diff.library]

1 This subclause summarizes the explicit changes in definitions, declarations, or behavior within the Standard C library when it is part of the Standard C++ library. (Subclause _lib.introduction_ imposes some *implicit* changes in the behavior of the Standard C library.)

C.4.1 Modifications to headers [diff.mods.to.headers]

1 For compatibility with the Standard C library, the Standard C++ library provides the 18 *C headers*, as shown in Table 81:

Table 81—C Headers

`<assert.h>`	`<iso646.h>`	`<setjmp.h>`	`<stdio.h>`	`<wchar.h>`
`<ctype.h>`	`<limits.h>`	`<signal.h>`	`<stdlib.h>`	`<wctype.h>`
`<errno.h>`	`<locale.h>`	`<stdarg.h>`	`<string.h>`	
`<float.h>`	`<math.h>`	`<stddef.h>`	`<time.h>`	

2 Each C header, whose name has the form *`name`*`.h`, includes its corresponding C++ header `c`*`name`*, followed by an explicit using-declaration (7.3.3) for each name placed in the standard library namespace by the header (_lib.namespace_).[217)]

C.4.2 Modifications to definitions [diff.mods.to.definitions]

1

C.4.2.1 Type `wchar_t` [diff.wchar.t]

1 `wchar_t` is a keyword in this International Standard (_lex.key_). It does not appear as a type name defined in any of `<cstddef>`, `<stddef.h>`, `<cstdlib>`, `<stdlib.h>`, `<cwchar>`, or `<wchar.h>`.

---

217) For example, the header `<cstdlib>` provides its declarations and definitions within the namespace `std`. The header `<stdlib.h>`, makes these available in the global name space, much as in the C Standard.

C.4.2.2 Header <iso646.h> [diff.header.iso646.h]

1 The tokens and, and_eq, bitand, bitor, compl, not_eq, not, or, or_eq, xor, and xor_eq are keywords in this International Standard (2.8). They do not appear as macro names defined in <iso646.h>.

C.4.2.3 Macro NULL [diff.null]

1 The macro NULL, defined in any of <clocale>, <locale.h>, <cstddef>, <stddef.h>, <cstdio>, <stdio.h>, <cstdlib>, <stdlib.h>, <cstring>, <string.h>, <ctime>, <time.h>, <cwchar>, or <wchar.h>, is an implementation-defined C++ null-pointer constant in this International Standard.[218)]

C.4.3 Modifications to declarations [diff.mods.to.declarations]

1 Header <string.h>: The following functions have different declarations:

— strchr

— strpbrk

— strrchr

— strstr

— memchr

Subclause (21.2) describes the changes.

C.4.4 Modifications to behavior [diff.mods.to.behavior]

1 Header <stdlib.h>: The following functions have different behavior: atexit

— exit

Subclause (18.3) describes the changes.

2 Header <setjmp.h>: The following functions have different behavior: longjmp

Subclause (18.7) describes the changes.

3

C.4.4.1 Macro offsetof(*type, member-designator*) <stddef.h> [diff.offsetof]

1 The macro offsetof, defined in <stddef.h>, accepts a restricted set of *type* arguments in this International Standard. *type* shall be a POD structure or a POD union.

C.4.5 Names with external linkage [diff.extern.c.names]

— Each name declared with external linkage in a C header is reserved to the implementation for use as a name with extern "C" linkage.

— Each function signature declared with external linkage in a C header is reserved to the implementation for use as a function signature with both extern "C" and extern "C++" linkage.[219)]

---

218) Possible definitions include 0 and 0L, but not (void*)0.

219) The function signatures declared in <cwchar> and <cwctype> are always reserved, notwithstanding the restrictions imposed in subclause 4.5.1 of Amendment 1 to the C Standard for these headers.

1 It is unspecified whether a name declared with external linkage in a C header has either `extern "C"` or `extern "C++"` linkage.[220)]

220) The only reliable way to declare an object or function signature from the Standard C library is by including the header that declares it, notwithstanding the latitude granted in subclause 7.1.7 of the C Standard.

Annex D (informative)
Future directions [future.directions]